

1. Introducción

En el **apunte 10** sobre acceso a base de datos mencionamos una clase **AccesoDB** que abstraía las operaciones CRUD detrás de una interfaz de alto nivel. Esa clase nos permitió dejar de escribir SQL repetido en cada parte de la aplicación, centralizar el manejo de la conexión y separar la lógica de persistencia de la lógica de negocio. Fue un avance importante, pero todavía quedaba un trabajo manual considerable: cada vez que agregábamos una entidad nueva al sistema (un jugador, una partida, un ítem), teníamos que escribir sus métodos específicos, mapear manualmente entre filas y objetos, y mantener esa correspondencia a mano cuando el esquema cambiaba.

En este apunte vamos a estudiar una herramienta que automatiza precisamente esa tarea: el **mapeo objeto-relacional** (ORM, por *Object-Relational Mapping*). Un ORM es un tipo particular de framework que se ocupa de traducir automáticamente entre el modelo de objetos del programa y el modelo relacional de la base de datos. Tablas y clases, filas y objetos, columnas y atributos: el ORM mantiene esa correspondencia y genera el SQL por nosotros.

Este apunte retoma varios conceptos que ya vimos. Por un lado, la idea de framework como aplicación semi-construida que extiende el desarrollador (**apunte 11**). Por otro, los pilares de inversión de control, inyección de dependencias y separación de responsabilidades. Y por último, el problema del acceso manual a datos que resolvimos parcialmente con **AccesoDB**.

2. El problema del mapeo objeto-relacional

Antes de definir qué es un ORM conviene entender por qué existe. La razón es que los dos modelos que queremos conectar —el orientado a objetos y el relacional— son fundamentalmente distintos. Esta diferencia tiene un nombre en la literatura: se la conoce como **desajuste de impedancia** (*impedance mismatch*)¹.

Veamos las diferencias más importantes:

¹Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. El término se usa en ingeniería eléctrica/electrónica para describir el desajuste entre dos componentes que no comparten la misma impedancia; Fowler lo retoma como analogía para describir la fricción entre el modelo de objetos y el modelo relacional.

- **Las tablas son planas, los objetos no.** Una tabla es una estructura bidimensional de filas y columnas. Un objeto puede tener atributos que a su vez son objetos, listas de objetos, o jerarquías de herencia.
- **Las tablas no tienen herencia.** En el modelo relacional, no existe la relación “es un”. Si queremos representar una jerarquía como **Vehiculo** → **Auto**, tenemos que hacerlo con trucos (una tabla por clase, una tabla con un campo discriminador, etc.). En cambio, en el modelo de objetos la herencia es un mecanismo de primera clase.
- **Las tablas tienen claves foráneas, los objetos tienen referencias.** Una relación “un jugador tiene muchas partidas” se representa con una columna **jugador_id** en la tabla **partidas**. En el modelo de objetos, esa misma relación se representa con una lista de objetos **Partida** dentro del objeto **Jugador**.
- **La identidad es distinta.** En el modelo relacional, la identidad de una fila está dada por su clave primaria. En el modelo de objetos, la identidad está dada por la referencia en memoria (el objeto es único aunque sus atributos sean idénticos a los de otro).

Estas diferencias hacen que la traducción entre ambos modelos no sea trivial. Sin una herramienta que la automatice, el desarrollador tiene que escribir manualmente el código de conversión en ambas direcciones, lo cual es repetitivo, propenso a errores y difícil de mantener cuando el esquema cambia.

3. Qué es un ORM

Un **ORM** es una herramienta que automatiza la traducción entre el modelo de objetos y el modelo relacional. El desarrollador define sus clases y las relaciones entre ellas; el ORM se encarga de:

- **Generar el SQL** necesario para crear las tablas correspondientes.
- **Traducir objetos a filas** cuando se persiste un objeto (INSERT, UPDATE).

- **Traducir filas a objetos** cuando se consulta la base de datos (SELECT).
- **Gestionar las relaciones** entre entidades: si un objeto tiene una lista de objetos relacionados, el ORM sabe cómo cargarlos y cómo mantener la consistencia.
- **Gestionar transacciones y sesiones** de manera controlada.

En términos de los pilares de frameworks que vimos en el apunte anterior, un ORM aplica el mismo principio de **inversión de control**: el desarrollador declara *qué* quiere (por ejemplo, “todos los jugadores con nivel mayor a 3”), y el ORM decide *cómo* traducirlo a SQL y cómo mapear los resultados. También aplica **inversión de dependencias**: el código de la aplicación depende de abstracciones (las clases del dominio) y no de SQL concreto.

Un ORM no reemplaza a la base de datos: sigue siendo MySQL, PostgreSQL o SQLite por debajo. Lo que reemplaza es la **forma de interactuar con ella**: en lugar de escribir SQL y mapear filas a objetos manualmente, se trabaja directamente con objetos.

4. Enfoques de diseño: Active Record y Data Mapper

Existen dos grandes estilos de ORM, que se diferencian en dónde vive la responsabilidad de la persistencia. Ambos fueron descritos por Fowler².

4.1. Active Record

En el patrón **Active Record**, cada objeto sabe cómo persistirse a sí mismo. La clase del dominio tiene métodos como `save()`, `delete()` o `find()` que ejecutan las operaciones correspondientes sobre la base de datos.

```
1 # Ejemplo conceptual
2 jugador = Jugador(nombre="Arturo", nivel=5)
3 jugador.save()           # el objeto se persiste a si mismo
4 Jugador.find(nombre="Arturo") # la clase sabe como buscar
```

Es el estilo que usan frameworks como Ruby on Rails y, en Python, Peewee y el ORM de Django. Es simple de entender y de usar, pero mezcla dos responsabilidades en la

²Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. Los patrones *Active Record* y *Data Mapper* aparecen descritos en detalle en dicho libro.

misma clase: la lógica de dominio y la lógica de persistencia. Un objeto **Jugador** que sabe cómo guardarse en MySQL es difícil de testear sin una base de datos real, y complica la aplicación del principio de responsabilidad única (SRP).

4.2. Data Mapper

En el patrón **Data Mapper**, la clase del dominio es totalmente ignorante de la persistencia. No tiene métodos `save()`, ni `find()`, ni nada por el estilo. La traducción entre objetos y filas la realiza una capa externa, típicamente una **sesión** (*Session*) o un **repositorio**.

```
1 # Ejemplo conceptual
2 jugador = Jugador(nombre="Arturo", nivel=5)
3 sesion.add(jugador)          # la sesion se encarga de persistirlo
4 sesion.commit()
```

Es el estilo que usa SQLAlchemy en su API moderna. La ventaja es que las clases del dominio quedan limpias, sin dependencias hacia la base de datos, lo cual favorece el SRP y facilita el testing: se pueden probar las clases del dominio sin necesidad de una base de datos. La desventaja es que requiere entender un concepto adicional (la sesión) y su ciclo de vida.

Para quien viene de los apuntes de principios de diseño, la diferencia se puede resumir así: **Active Record** tiende a violar SRP al mezclar dominio y persistencia; **Data Mapper** respeta SRP al mantener cada responsabilidad en su propia capa.

5. Panorama de ORMs en Python

En Python existen varios ORMs, cada uno con sus particularidades. Los más utilizados son:

ORM	Enfoque	Observaciones
SQLAlchemy	Data Mapper	El estándar de la industria. Potente y flexible. Se integra con Flask a través de Flask-SQLAlchemy.

ORM	Enfoque	Observaciones
Peewee	Active Record	Ligero, simple de aprender. Ideal para proyectos chicos.
Django ORM	Active Record	Viene incluido en Django. Muy cómodo si se usa Django, difícil de usar fuera de él.
Tortoise ORM	Active Record	Diseñado para aplicaciones asíncronas. Inspirado en Django ORM.

En este curso trabajaremos con **SQLAlchemy**, que es el ORM más usado en el ecosistema Python profesional y el que mejor se alinea con lo que venimos viendo: implementa Data Mapper, respeta SRP, y se integra naturalmente con Flask.

6. SQLAlchemy en el ecosistema Flask

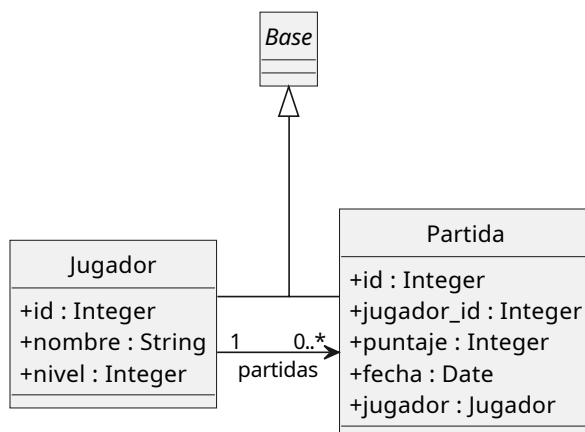
Flask no trae un ORM integrado. Esto no es un defecto sino una consecuencia de su filosofía: Flask provee lo esencial (enrutamiento, manejo de peticiones, plantillas) y deja que el desarrollador elija el resto de las herramientas. Esto lo hace más flexible que frameworks como Django, que traen un ORM propio y predefinido.

Para integrar SQLAlchemy con Flask se usa la extensión **Flask-SQLAlchemy**, que provee un objeto `db` con la configuración del ORM ya inicializada. Este objeto se comporta como una dependencia inyectada por el framework: las vistas lo importan y lo usan sin tener que configurarlo cada vez.

Desde el punto de vista de los pilares de frameworks que vimos antes, esto es un ejemplo concreto de **inversión de control** y **convención sobre configuración**: el desarrollador declara sus modelos y Flask-SQLAlchemy se encarga de conectarlos con la base de datos, gestionar las sesiones y proveer la infraestructura común.

7. Ejemplo funcional con SQLAlchemy 2.0

Veamos un ejemplo completo con dos entidades relacionadas: **Jugador** y **Partida**. Un jugador tiene muchas partidas, y cada partida pertenece a un único jugador.



El código completo se ve así:

```
1 from datetime import date
2 from sqlalchemy import create_engine, select, ForeignKey, String
3 from sqlalchemy.orm import (DeclarativeBase, Mapped,
4                             mapped_column, relationship, Session)
5 class Base(DeclarativeBase):
6     pass
7 class Jugador(Base):
8     __tablename__ = "jugadores"
9
10    id: Mapped[int] = mapped_column(primary_key=True)
11    nombre: Mapped[str]
12    nivel: Mapped[int]
13
14    partidas: Mapped[list["Partida"]] = relationship(
15        back_populates="jugador")
16
17    def __repr__(self) -> str:
18        return f"Jugador({self.nombre!r}, nivel={self.nivel})"
19 class Partida(Base):
20     __tablename__ = "partidas"
21
22    id: Mapped[int] = mapped_column(primary_key=True)
23    jugador_id: Mapped[int] = mapped_column(
24        ForeignKey("jugadores.id"))
25    puntaje: Mapped[int]
26    fecha: Mapped[date]
27
28    jugador: Mapped["Jugador"] = relationship(
29        back_populates="partidas")
```

```
30
31 # --- Conexion ---
32 engine = create_engine(
33     "mysql+mysqlconnector://root:@localhost/juego")
34
35 # --- Crear las tablas (si no existen) ---
36 Base.metadata.create_all(engine)
37
38 # --- Insertar un jugador con dos partidas ---
39 with Session(engine) as session:
40     arturo = Jugador(nombre="Arturo", nivel=5)
41     arturo.partidas.append(Partida(puntaje=9500,
42                                   fecha=date(2026, 3, 15)))
43     arturo.partidas.append(Partida(puntaje=8200,
44                                   fecha=date(2026, 3, 16)))
45     session.add(arturo)
46     session.commit()
47
48 # --- Consultar: el mejor puntaje de cada jugador ---
49 with Session(engine) as session:
50     jugadores = session.scalars(select(Jugador)).all()
51     for jugador in jugadores:
52         mejor = max(p.puntaje for p in jugador.partidas)
53         print(f"{jugador.nombre} (nivel {jugador.nivel}): {mejor}")
```

Analicemos las partes más importantes:

Clase Base

La clase **Base** parece vacía, pero cumple un rol central en el mapeo. Todas las entidades del proyecto heredan de ella, y eso hace que compartan una misma **metadata**: el registro interno donde SQLAlchemy lleva la cuenta de qué tablas existen, cómo se llaman y cómo están definidas.

Esa metadata compartida es lo que permite que **Base.metadata.create_all(engine)** recorra todas las clases y cree las tablas correspondientes de una sola vez. También es el lugar natural para agregar comportamiento común a todos los modelos, como un método **to_dict()**.

Técnicamente se podría heredar directamente de **DeclarativeBase** en cada clase, y el código funcionaría. Pero es una mala práctica: cada clase tendría su propia metadata separada, se perdería el punto central de extensión, y no habría forma clara de distinguir qué entidades pertenecen a un mismo proyecto. Definir una **Base** propia es la convención de SQLAlchemy y una decisión de diseño, no un requisito del lenguaje.

Declaración de los modelos

La clase **Base** hereda de **DeclarativeBase** y sirve como punto de partida para todas las entidades. Cada modelo hereda de **Base** y define su tabla con `__tablename__`. Los atributos se declaran con **Mapped[TIPO]**, que indica tanto el nombre de la columna como su tipo, y **mapped_column()**, que agrega configuración específica como la clave primaria.

Descriptores y declaración de atributos

En POO clásica, los atributos se declaran en el `__init__` con `self.__atributo = valor`. En SQLAlchemy, en cambio, los atributos mapeados se declaran a **nivel de clase**:

```
1 class Jugador(Base):
2     __tablename__ = "jugadores"
3
4     id: Mapped[int] = mapped_column(primary_key=True)
5     nombre: Mapped[str]
6     nivel: Mapped[int]
```

Esta diferencia responde a una necesidad concreta: SQLAlchemy debe **inspeccionar la clase en el momento en que se define**, no cuando se instancia, para construir el mapeo entre la clase y la tabla. Si los atributos solo existieran dentro de un `__init__`, el ORM nunca podría verlos al analizar la clase, porque `__init__` se ejecuta recién al crear cada objeto.

Lo que se asigna en el cuerpo de la clase **no es un valor**: es un **descriptor** que SQLAlchemy instala para mediar entre el objeto y la base de datos. Cuando el código cliente hace `jugador.nombre`, no está leyendo un atributo común, sino pasando por ese descriptor, que decide si el valor está en memoria, si hay que ir a buscarlo a la base, o si hay que marcarlo como modificado.

SQLAlchemy, además, **genera automáticamente un `__init__`** que acepta como parámetros los atributos mapeados. Por eso `Jugador(nombre="Arturo", nivel=5)` funciona sin que hayamos escrito un `__init__` explícito. Si se necesita lógica de inicialización adicional (calcular un valor, validar algo), se puede definir un `__init__` propio, llamando a `super().__init__()` cuando corresponda.

Relaciones

El atributo `partidas` en `Jugador` y el atributo `jugador` en `Partida` representan la relación uno-a-muchos. El parámetro `back_populates` le indica al ORM que ambas pro-

piezas son dos caras de la misma relación, de modo que mantener una actualice la otra. Navegar por `jugador.partidas` es equivalente a un `JOIN` en SQL, pero el ORM lo resuelve automáticamente.

Engine y sesión

El **engine** representa la conexión con la base de datos y se crea una sola vez. La **Session** es un objeto de corta vida que agrupa un conjunto de operaciones y las confirma atómicamente con `commit()`. El uso del bloque `with Session(engine) as session` garantiza que la sesión se cierre al finalizar, incluso si ocurre una excepción.

Consultas

Las consultas se expresan con la función `select()`, que construye una expresión en lugar de un string. El método `session.scalars(stmt)` ejecuta la consulta y devuelve directamente los objetos mapeados, sin que el desarrollador tenga que convertir filas manualmente.

La salida del programa sería:

Arturo (nivel 5): 9500

Nota: encapsulamiento y atributos públicos

En el ejemplo anterior los atributos del modelo aparecen como públicos (`jugador.nombre`, `jugador.nivel`). Esta es la práctica habitual con ORMs, y conviene aclarar que **no constituye una violación del encapsulamiento**.

El atributo `nombre` no es un atributo común: es un **descriptor** que SQLAlchemy instala en la clase y que media entre el objeto y la sesión. A través de él se controla cuándo se lee, cuándo se escribe y cuándo se marca el objeto como modificado. Dicho de otro modo, el descriptor cumple la función que en POO clásica cumple el guion bajo: señalar que hay una capa de indirección entre el cliente y el valor interno.

Cuando se necesita lógica adicional –validar un valor, transformarlo, calcular un atributo derivado–, SQLAlchemy permite declarar el atributo mapeado con guion bajo y exponer una propiedad pública sobre él:

```
1 class Jugador(Base):
2     __tablename__ = "jugadores"
3
4     id: Mapped[int] = mapped_column(primary_key=True)
5     _nombre: Mapped[str] = mapped_column("nombre", String
6         (50))
7     _nivel: Mapped[int] = mapped_column("nivel")
8
9     @property
10    def nombre(self) -> str:
11        return self._nombre
```

No es necesario hacerlo desde el comienzo. La recomendación es **empezar con atributos públicos** y convertir a propiedades solo cuando la lógica lo justifique, evitando así el ruido de decenas de *getters* y *setters* sin valor agregado.

8. Ventajas y trade-offs

Como toda herramienta, un ORM no es una solución universal. Conviene conocer sus ventajas y sus limitaciones.

8.1. Ventajas

- **Elimina el código repetitivo** de traducción entre objetos y filas.
- **Centraliza el acceso a datos** en las clases del dominio, evitando que el SQL se disperse por la aplicación.
- **Gestiona las relaciones** entre entidades de manera transparente.
- **Permite cambiar de motor de base de datos** con cambios mínimos, ya que el ORM abstrae las diferencias entre dialectos SQL.
- **Favorece el testing**: se pueden usar bases de datos en memoria (como SQLite) durante las pruebas.

8.2. Trade-offs

- **Curva de aprendizaje.** La API de un ORM como SQLAlchemy no es trivial. Requiere entender conceptos nuevos (sesión, identidad, lazy loading, eager loading) que no aparecen en SQL directo.
- **Consultas generadas ineficientemente.** Un ORM puede generar consultas N+1 o JOINs innecesarios si no se usa con cuidado. En aplicaciones de alto rendimiento, a veces conviene escribir SQL directo.
- **No reemplaza el conocimiento de SQL.** Saber qué consulta está ejecutando el ORM por debajo sigue siendo necesario para diagnosticar problemas y optimizar.
- **Complejidad del mapeo.** En esquemas con herencia o relaciones muchos-a-muchos complejas, la configuración del ORM puede volverse engorrosa.

La regla práctica es: **usar ORM por defecto, y bajar a SQL directo cuando sea necesario.** La mayoría de las operaciones de una aplicación se benefician del ORM; las consultas críticas pueden escribirse a mano y ejecutarse dentro del mismo ORM.

Con esto cerramos el estudio de los frameworks ORM. Los conceptos vistos aquí – mapeo objeto-relacional, los patrones Active Record y Data Mapper, el rol de la sesión y las relaciones entre entidades – son la base sobre la que se construyen las aplicaciones que integran persistencia con otros componentes. En próximos apuntes veremos cómo esas piezas se combinan en el desarrollo de aplicaciones completas.

9. Referencias

Para más información puede consultar los siguientes enlaces:

- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Documentación oficial de SQLAlchemy: <https://docs.sqlalchemy.org/>
- Documentación de Flask-SQLAlchemy:
<https://flask-sqlalchemy.palletsprojects.com/>
- Refactoring Guru: <https://refactoring.guru/es/>