1. ¿Por qué eligieron ese ORM y qué beneficios o dificultades encontraron?

Elección: SQLAlchemy por ser el ORM más completo y maduro para Python con soporte nativo para PostgreSQL.
Beneficios:

- Sintaxis Pythonica para operaciones complejas
- Soporte para tipos personalizados y vistas
- Generación automática de DDL
- Mapeo declarativo intuitivo
  Dificultades:
- Curva de aprendizaje para relaciones avanzadas
- Manejo de transacciones explícitas
- Depuración de consultas generadas automáticamente

2. ¿Cómo implementaron la lógica master-detail dentro del mismo formulario?

- Formulario dinámico: Usamos JavaScript para agregar/eliminar bloques de detalles (títulos y participaciones)
- Procesamiento backend: Iteramos sobre campos indexados (titulo_id_1, fecha_obtencion_1, etc.)
- Transacción única: Todas las operaciones (luchador + relaciones) se ejecutan en una sola transacción ACID

3. ¿Qué validaciones implementaron en la base de datos y cuáles en el código?

Nivel	Validaciones
Base de datos	- CHECK para peso (0-300 kg) - CHECK para altura (100-230 cm) - UNIQUE en combinaciones críticas - NOT NULL en campos obligatorios - FOREIGN KEY para integridad referencial

Nivel	Validaciones
Código	- Formato de fechas - Rangos numéricos - Campos requeridos - Relaciones existentes - Coherencia temporal (fecha_perdida > fecha_obtencion)

4. ¿Qué beneficios encontraron al usar tipos de datos personalizados?

- Consistencia: Garantizan que todos los pesos/alturas cumplan reglas de negocio
- Reutilización: Definidos una vez y usados en múltiples modelos
- Documentación viva: Hacen explícitos los requisitos de dominio
- Seguridad: Previenen datos inválidos antes de llegar a la base

5. ¿Qué ventajas ofrece usar una VIEW como base del índice en vez de una consulta directa?

- Abstracción: Oculta complejidad de joins y agregaciones
- Mantenibilidad: Cambios en estructura no afectan código de aplicación
- Rendimiento: Optimizada como objeto persistente en PostgreSQL
- Consistencia: Misma lógica para aplicación y reportes
- Seguridad: Control granular de acceso a datos sensibles

6. ¿Qué escenarios podrían romper la lógica actual si no existieran las restricciones?

1. Luchador con peso 500 kg (CHECK constraint)
2. Dos títulos idénticos para mismo luchador en misma fecha (UNIQUE)
3. Participación en lucha inexistente (FOREIGN KEY)
4. Fecha de pérdida anterior a obtención (CHECK temporal)
5. Apodos duplicados para mismo luchador (UNIQUE parcial)

7. ¿Qué aprendieron sobre la separación entre lógica de aplicación y lógica de persistencia?

- Ventajas:

    - Cambios en DB no afectan lógica de negocio
    - Testing más sencillo (mocks de persistencia)
    - Portabilidad entre motores de DB
- Lecciones clave:
    - ORM como abstracción, no como varita mágica
    - Validaciones deben existir en ambos niveles
    - Transacciones como unidad atómica de negocio

8. ¿Cómo escalaría este diseño en una base de datos de gran tamaño?

- Estrategias:
    1. Sharding: Por federación (ej: luchadores activos/históricos)
    2. Indexación: Índices en campos de búsqueda frecuente (nombre, fecha)
    3. Particionamiento: Tablas de eventos por año
    4. Caching: Redis para consultas de vista
    5. Réplicas de lectura: Para reportes
    6. Compresión: TOAST para grandes textos (historiales)

9. ¿Consideran que este diseño es adecuado para una arquitectura con microservicios?

Sí, con adaptaciones:

- Ventajas actuales:
    - Modelo autónomo con límites definidos
    - API REST clara (CRUD + relaciones)
- Mejoras para microservicios:
    1. Dividir en servicios: Luchadores, Eventos, Títulos
    2. Usar eventos de dominio para consistencia eventual
    3. API Gateway para unificar endpoints
    4. Circuit Breaker para dependencias entre servicios

10. ¿Cómo reutilizarían la vista en otros contextos como reportes o APIs?

1. APIs REST
    - Proporciona datos preprocesados para respuestas JSON eficientes.
    - Evita joins repetitivos en consultas frecuentes.
2. Reportes y Exportaciones
    - Base estructurada para generación de CSV, Excel o informes PDF.
    - Simplifica agregaciones y cálculos estadísticos.
3. Seguridad y Consistencia
    - Centraliza reglas de acceso a datos sensibles.
    - Garantiza que todos los sistemas consuman la misma lógica de negocio.
4. Microservicios
    - Funciona como contrato de datos desacoplado para otros servicios.
    - Facilita actualizaciones sin impactar consumidores.

11. ¿Qué decisiones tomaron para estructurar su modelo de datos y por qué?

Decisión	Justificación
Tablas intermedias explícitas	Permitir atributos adicionales (fechas, resultados)
Tipos ENUM para categorías fijas	Validar dominios conocidos (tipos de lucha)
Vista materializada	Optimizar consulta frecuente de resumen
Claves naturales vs. artificiales	Rendimiento en relaciones (IDs autoincrementales)
Modelo híbrido (ORM + SQL crudo)	Balance entre abstracción y control

12. ¿Cómo documentaron su modelo para facilitar su comprensión por otros desarrolladores?

1. Diagrama ER: Con cardinalidades y atributos clave
2. Docstrings en modelos: Descripción de cada entidad y relación
3. Schema versionado: schema.sql generado automáticamente
4. Diccionario de datos: En README.md con descripción de campos
5. Ejemplos operacionales: Archivo data.sql con casos realistas

13. ¿Cómo evitaron la duplicación de registros o errores de asignación en la tabla intermedia?

- Restricciones de base de datos:

UNIQUE(luchador_id, titulo_id, fecha_obtencion)

UNIQUE(luchador_id, lucha_id)

- Validación en aplicación:

```
if LuchadorTitulo.query.filter_by(
  luchador_id=id_luchador,
  titulo_id=id_titulo,
  fecha_obtencion=fecha
).first():
  raise ValidationError("Título ya registrado en esa fecha")
```

- Patrones de interfaz:
  - Combos desplegables con opciones válidas
  - Bloqueo de selecciones duplicadas en UI
  - Confirmación de transacción atómica