Universidad del Valle de Guatemala
Base de datos 1 – Sección 10
Fernando Hernández – 23645
Fernando Rueda – 23748

Análisis

1. **¿Por qué eligieron ese ORM y qué beneficios o dificultades encontraron?**
   El uso de SQLAlchemy fue elegido por su integración perfecta con FastAPI y su capacidad para manejar esquemas complejos mediante modelos declarativos. La configuración del SessionLocal y el engine permitió una gestión eficiente de conexiones a PostgreSQL, mientras que el soporte nativo para tipos ENUM (como SexoEnum y EstadoAnimalEnum) aseguró consistencia en los datos.

2. **¿Cómo implementaron la lógica master-detail dentro del mismo formulario?**
   En main.py, el endpoint "animales" implementó un CRUD completo con lógica master-detail para animales y su dieta. La operación de creación (POST) maneja una transacción atómica, primero inserta el animal y luego sus alimentos asociados, evitando inconsistencias. Para actualizaciones (PUT), se eliminó y recreó toda la dieta, garantizando que los cambios se reflejen completamente.

3. **¿Qué validaciones implementaron en la base de datos y cuáles en el código?**
   Las validaciones en schemas.py con Pydantic (como SexoEnum y rangos de fechas) complementaron las restricciones de la BD. Por ejemplo, el esquema AnimalBase valida el formato de los datos antes de que lleguen a SQLAlchemy, mientras que las claves foráneas en los modelos (models.py) aseguran integridad referencial.

4. **¿Qué beneficios encontraron al usar tipos de datos personalizados?**
   Los ENUMs en models.py (como TipoAlimentoEnum) no solo estandarizaron valores posibles, sino que también mejoraron la legibilidad del código. Al replicar los tipos personalizados de PostgreSQL (tipo_alimento), se creó un puente natural entre la aplicación y la BD. Esto fue clave para evitar errores como alimentos con tipos inválidos, ya que tanto Pydantic como SQLAlchemy aplicaron las mismas reglas.

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

   La vista vista_animales_dieta se consumió en main.py mediante una consulta SQL directa, lo que simplificó el frontend en app.py. Streamlit pudo mostrar los datos combinados sin lógica adicional, demostrando cómo las vistas abstraen complejidad. Además, al usar esta vista como fuente única para el reporte, se evitó duplicar la lógica de JOINs en múltiples endpoints.

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

   El código de eliminación en main.py evidenció los riesgos de no tener restricciones, pues al borrar un animal, fue necesario eliminar manualmente sus registros relacionados en 4 tablas. Sin esto, habrían quedado datos huérfanos. La transacción explícita con try/except y rollback() aseguró que, ante fallos, la BD no quedara en estado inconsistente.

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

   Aprendimos que la arquitectura dividió claramente las responsabilidades, los modelos (models.py) definieron la estructura de la BD, los esquemas Pydantic (schemas.py) validaron los datos de entrada/salida, y los endpoints (main.py) se encargaron de la lógica. Esta separación permitió, por ejemplo, cambiar el tipo decimal a float en los esquemas sin afectar los modelos, adaptándose a las necesidades del frontend.

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

   Añadiendo índices a campos como el nombre del animal para búsquedas más rápidas, o partiendo en trozos la tabla de registros médicos por fechas. Si el sistema crece mucho, habría que pensar en cachés para no saturar la base con consultas repetidas.

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

   El diseño es adecuado, con tablas bien definidas que podrían corresponder a servicios distintos (animales, empleados, visitantes).

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

    En app.py, la vista se consumió directamente desde Streamlit mediante una llamada API. La misma vista podría usarse en otros contextos, como generar reportes CSV o alimentar un dashboard en tiempo real, sin modificar la lógica del backend.

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

    Al diseñar las tablas, buscamos el punto justo, ni tan desorganizado que haya datos repetidos por todos lados, ni tan fragmentado que hacer una consulta simple requiera unir 10 tablas. Un buen ejemplo es la dieta, en vez de poner los alimentos directamente en la tabla de animales, creamos una tabla intermedia. Así podemos preguntar tanto "qué come el león" como "qué animales comen carne", manteniendo todo ordenado.

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

    Se incluyeron comentarios claros y se generó diagrama ER para clarificar relaciones.

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

    El constraint UNIQUE en Dieta evitó duplicados, pero el frontend en app.py también incluyó validaciones como verificar IDs de alimento existentes.