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

Elegimos SQLAlchemy porque es el ORM más maduro y flexible para Python, con buen soporte para PostgreSQL y una integración sencilla con FastAPI. Los beneficios incluyen la abstracción de SQL, migraciones fáciles y validación de modelos. La principal dificultad fue la curva de aprendizaje inicial y la gestión de relaciones complejas.

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

Utilizamos modelos Pydantic anidados para enviar datos y detalles en una sola petición. En el backend, procesamos el objeto principal y sus detalles en una transacción para asegurar la integridad.

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

En la base de datos usamos restricciones de clave foránea, unicidad y tipos de datos. En el código validamos formatos, rangos y reglas de negocio usando Pydantic y lógica adicional en los endpoints.

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

Los tipos personalizados, como enums, mejoran la legibilidad y evitan valores inválidos, asegurando que solo se usen opciones válidas tanto en la base como en el código.

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

Una VIEW permite reutilizar lógica compleja, centralizar reglas de negocio y simplificar consultas, facilitando el mantenimiento y la consistencia de los datos mostrados.

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

Sin restricciones podríamos tener registros huérfanos, duplicados o inconsistentes, como reservas para usuarios o colas inexistentes, lo que afectaría la integridad de la información.

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

Aprendimos que separar ambas capas facilita el mantenimiento, las pruebas y la escalabilidad, permitiendo cambiar la base de datos o la lógica de negocio sin afectar todo el sistema.

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

El diseño escalaría bien si se agregan índices adecuados, particionamiento y optimización de consultas. Sin embargo, habría que revisar la eficiencia de las transacciones y el uso de vistas para grandes volúmenes.

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

Sí, porque la separación de modelos, lógica y endpoints permite dividir el sistema en servicios independientes, aunque habría que adaptar la comunicación y la gestión de transacciones distribuidas.

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

La vista puede ser consultada directamente desde otros endpoints, scripts de reporte o herramientas de BI, asegurando que todos los consumidores vean datos consistentes.

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

Optamos por normalizar las tablas, usar claves foráneas y enums para mantener la integridad y claridad, y facilitar futuras extensiones del modelo.

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

Incluimos comentarios en el código, diagramas ER, además de ejemplos de uso en los endpoints.

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

Usamos restricciones de unicidad y validaciones en el backend para evitar duplicados, y controlamos las inserciones mediante transacciones atómicas.