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

   Elegimos ese ORM porque nos ofrecía una integración sencilla con PostgreSQL y permitía trabajar con estructuras más limpias en el código. Además, consideramos que para un API bastante simple JS es una buena opción en cuanto a lenguaje de programación. Como beneficios, encontramos que simplificó la escritura de queries y relaciones, en vez de realizar queries complejos pudimos hacer queries directamente desde JS. Como dificultades, extrañamos profundamente el control que nos da el SQL puro. Al momento de hacer debugs tuvimos bastante estorbo debido a la capa de abstracción y además tuvimos que leer bastante documentación para entender cómo se manejaban las migraciones.

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

   Para la lógica master-detail en el mismo formulario, usamos un solo endpoint que recibía toda la estructura del objeto maestro con sus detalles. Así se podían validar y guardar todos los datos dentro de una misma transacción, asegurando consistencia y simplificando el flujo en frontend y backend. Más específicamente, utilizando Elysia para el API podemos validar campos declarando algunos como opcionales y validándolos si existen.

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

   En la base de datos validamos cosas como claves únicas, restricciones de integridad y tipos de datos. En el código nos enfocamos en validar estructura del JSON, campos requeridos y reglas de negocio que la base de datos no podía manejar directamente, como condiciones específicas entre campos. Otra vez mencionamos a Elysia, dónde nos permitió hacer verificaciones de tipos de datos extensas para validar los payloads.

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

   Usar tipos de datos personalizados nos ayudó a estandarizar valores repetitivos y a darle más sentido semántico a ciertos campos. Así reducimos errores por valores incorrectos y el código se volvió más claro en cuanto al tipo de información que se esperaba.

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

   Usar una VIEW como base del índice nos dio la ventaja de encapsular lógica compleja en un solo lugar y reutilizarla fácilmente sin repetir joins o cálculos. Además, nos permitió mantener el rendimiento sin sacrificar legibilidad.

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

   Si no tuviéramos restricciones, podríamos tener asignaciones duplicadas, datos huérfanos o relaciones inválidas. Escenarios como insertar detalles sin un maestro o asignar roles repetidos serían posibles y romperían la lógica del sistema. Adicionalmente, nuestras restricciones también verifican reglas de negocio cómo lo pueden ser goles o asistencias negativas o un jugador encontrándose en dos equipos a la vez.

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

Aprendimos que separar la lógica de aplicación de la de persistencia ayuda a mantener el código más limpio, escalable y fácil de testear. Dejar la validación estructural en la base de datos y la de negocio en el código fue clave para evitar bugs difíciles de rastrear.

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

Este diseño escalaría bien en bases grandes si se optimizan índices, se evita la sobrecarga de joins y se separan vistas/materializadas para reportes. El uso de transacciones controladas y claves bien definidas también ayuda a mantener el rendimiento.

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

Sí, este diseño puede adaptarse bien a microservicios si se separan claramente los contextos de cada entidad y se encapsula la lógica. Las tablas intermedias y vistas pueden exponerse como servicios internos y seguir funcionando correctamente.

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

La vista se puede reutilizar fácilmente en reportes si se expone a través de una API, ya que encapsula toda la lógica necesaria y se mantiene actualizada automáticamente. También puede ser usada por herramientas de BI o dashboards sin reescribir lógica.

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

Decidimos estructurar el modelo de datos pensando primero en las relaciones reales entre entidades y luego en la facilidad de consulta. Usamos claves foráneas, tablas intermedias y normalización donde tenía sentido para mantener consistencia y evitar redundancia.

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

Documentamos el modelo con diagramas ER, comentarios en el código SQL y en los controladores, además de un README donde explicamos los casos de uso principales. Así otros desarrolladores pueden entender cómo se relacionan las tablas y qué espera cada campo.

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

Para evitar duplicación en la tabla intermedia, aplicamos una restricción de clave única combinada entre las columnas de relación y fecha. Esto nos asegura de no insertar erróneamente la misma información duplicada. Además, en nuestro caso, un jugador puede irse y volver al mismo equipo, por lo que tuvimos que implementar una validación dentro del backend que hiciera esta revisión por nosotros.