

Ángel de Jesús Mérida Jiménez -23661

André Emilio Pivaral López - 23574

Reflexión:

Responde las siguientes preguntas:

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

R//Escogimos Sequelize porque es el estándar de facto en Node.js y ya veníamos usándolo en el curso. Sus beneficios principales fueron:

- Abstracción completa de SQL, lo que nos permitió concentrarnos en la lógica de negocio.
- Migraciones versionadas, ideales para el trabajo en equipo.
- Soporte nativo para ENUM y para relacioneshasMany / belongsToMany, que encajaban con nuestro modelo.
- Dificultades: la curva de aprendizaje de las asociaciones N-M y la poca claridad de sus mensajes de error cuando violas restricciones de la BD; por eso añadimos validaciones explícitas en el código.

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

R//Se crea el curso (Curso.create).

Tomamos el curso.id y mapeamos el arreglo de estudiantes para construir el array de inscripciones (Inscripcion.bulkCreate).

Usamos una transacción Sequelize (transaction: t) de modo que si falla cualquiera de los inserts se hace rollback y no queda el curso huérfano.

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

En la base:

- NOT NULL en nombres y claves foráneas.
- UNIQUE (CursoId, EstudianteId) en Inscripcions.
- CHECK (char_length(nombre) > 0) para evitar cadenas vacías.
- En el código (Sequelize):
- allowNull: false, validate: { len: [1, 100] } en modelos.
- Comprobación de que el nivel y el estado pertenezcan al ENUM antes de guardar.
- Validación custom que rechaza inscribir al mismo estudiante dos veces en el mismo request.

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

- Los tipos ENUM (enum_cursos_nivel, enum_estudiantes_estado) nos dieron:

- Seguridad de datos: sólo valores permitidos llegan a la tabla.
- Simplicidad en la interfaz: el formulario puede autogenerar <select> a partir del propio ENUM.
- Lectura más clara en consultas y reportes, ya que el texto del nivel/estado viene directo, sin hacer join a tablas auxiliares.

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

- Separa la lógica de combinación de tablas de la capa aplicación.
- Podemos cambiar la estructura interna sin tocar el front-end.
- Permite otorgar permisos de SOLO LECTURA a la vista y no a las tablas.
- Facilita indexar (materializar) si la vista crece y necesitamos rendimiento.

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

- Duplicar inscripciones del mismo estudiante al mismo curso.
- Quedar con inscripciones huérfanas si se elimina un curso sin cascada.
- Insertar un estado inexistente y que luego el front-end no sepa interpretarlo.
- Almacenar nombres vacíos que vuelven confuso un reporte.

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

R// Que los reglamentos críticos (integridad referencial, unicidad) deben residir en la BD; el código sólo refuerza experiencia de usuario. Así, cualquier otro servicio o script que consuma la BD hereda las mismas garantías.

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

- Índices en FK y en (CursoId, EstudianteId) mantienen el tiempo de búsqueda O(log n).
- La VIEW podría volverse costosa; se puede materializar o paginar la consulta.
- Sharding no sería necesario al principio; bastaría con replicas de lectura y pooling de conexiones desde Sequelize.

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

R// Parcialmente. El bounded context “Gestión Académica” podría ser un microservicio; sin embargo, compartir la misma BD con otros servicios violaría la independencia. Para microservicios verdaderos exportaríamos la VIEW como endpoint REST/GraphQL y cada servicio mantendría su propio esquema o su propia instancia.

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

- Reportes administrativos pueden consultar la vista para exportar CSV sin tocar el código.
- En APIs, creariamos un modelo Sequelize mapeado a la VIEW y expondremos /api/reportes/cursos que hace findAll sobre ella.
- Para BI (Power BI / Tableau) se expone la vista con permiso SELECT y ya integra el modelo de forma directa.

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

- Relación N-M entre Cursos y Estudiantes porque un estudiante puede tomar varios cursos y viceversa.
- Usamos nombres en singular camel-case para tablas (estilo Sequelize) pero con PK explícitas (id) para compatibilidad con otras herramientas.
- Los ENUM se definieron dentro de PostgreSQL, no como strings libres, para asegurar consistencia.

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

- ERD.pdf exportado desde pgModeler con claves y cardinalidades.
- Comentarios SQL (COMMENT ON TABLE/ COLUMN ...) que describen cada entidad y atributo.
- README con diagrama ASCII simplificado y link al ERD.
- JSDoc en modelos Sequelize explicando atributos, asociaciones y ejemplos de uso.

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

- Restricción UNIQUE (CursoId, EstudianteId) en la tabla Inscripcions.
- En la capa servicio, usamos findOrCreate antes de insertar y capturamos la excepción SequelizeUniqueConstraintError para mostrar un mensaje amigable.
- La transacción master-detail asegura que si un estudiante se repite en el formulario el rollback impide que queden medias inscripciones.