

Laboratorio: TypeORM

Este laboratorio tiene como propósito, migrar del ORM actual (Sequelize) a uno con mejor experiencia de desarrollo (o DX), con herramientas modernas y fáciles de usar como migraciones y que aproveche todas las capacidades de Postgres actual.

Durante la investigación, se encontraron dos ORM que cumplen con las funcionalidades básicas esperadas: Prisma y TypeORM. Este laboratorio está hecho con base en TypeORM.

PROBLEMA

Sequelize presenta limitaciones al ser una herramienta antigua, entre ellas:

- Tooling menos flexible y poco amigable con ESM moderno
- Integración limitada y poco estricta con TypeScript
- Manejo de migraciones y seeders menos robusto
- Mayor fricción al escalar o mantener proyectos con arquitecturas modernas

Estas limitaciones motivaron la evaluación y migración hacia una alternativa más actual que permita mayor control, mantenibilidad y claridad en el desarrollo.

RÚBRICAS

Para este laboratorio, se decidió realizar un servicio web pequeño de autenticación basado en usuarios y un recurso que cada usuario pueda acceder al iniciar sesión: Logros.

Con este pequeño servicio, se pusieron a prueba las siguientes rúbricas **en comparación con Sequelize**:

- Complejidad de Instalación y configuración.
- Creación de modelos (o entidades).
- Asociación de modelos (o entidades).
- Consultas a la entidad.
- Dependency Injection.
- Type-Safety.
- Migraciones.
- Testing.

Instalación y Configuración

La diferencia en cuanto a complejidad de instalación y configuración (sin tomar en cuenta la configuración inicial de TypeScript) es relativamente similar: Cada uno tiene su propia instancia inicial (DataSource para TypeORM y Sequelize para Sequelize), ambas tienen los mismos campos básicos para conectarse a la base de datos y campos extra para logging de consultas y sincronización en modo desarrollo y ambas se conectan de forma similar a la base de datos.

La diferencia entre la instancia DataSource y Sequelize es que DataSource tiene soporte a migraciones y eliminación manual de la base de datos. Así como también, agregar la dependencia “reflect-metadata” al principio en el [app.ts](#).

Creación de Entidades

La diferencia en comparación con Sequelize depende de su uso. Si en sequelize, solo creas modelos como objetos, puede parecer confuso el cambio, debido a que TypeORM depende de clases. Caso contrario si en Sequelize creas modelos como clases, el cambio puede notarse pero no confundir a primera vista. En ambos casos, la API para entidades es legible y fácil de usar.

Las únicas diferencias son que las propiedades (columnas) de las clases (entidades) se definen con decoradores y que, propiedades como “timestamps” o “paranoid” en Sequelize deben integrarse manualmente.

Asociación de Entidades

A diferencia de Sequelize, donde las relaciones (o asociaciones) se definen mediante métodos directos (“hasOne”, “hasMany”, “belongsTo”, “belongsToMany”, ...), en TypeORM, se definen dentro de las entidades mediante decoradores y entidades ya existentes.

Para casos complejos como asociaciones mucho a muchos (N:M), la documentación de ambos ORMs ofrecen ejemplos de uso detallados y funcionales.

Consultas a una Entidad

Mientras que, en Sequelize, para realizar una petición a una entidad (llámese “users”, por ejemplo) se usa el modelo directamente, en TypeORM se instancia el método “getRepository” junto con la entidad a consultar y luego el método de consulta (llámese “find”, “findAndCount”, “create”, etc...).

Esto, a nivel visual es un poco más verboso que en Sequelize, puesto que debes usar el “DataSource” más la entidad. Sin embargo, una de las implementaciones que ayudaron a eliminar dicha verbosidad dentro del laboratorio fue Dependency Injection (o DI).

De resto, ambas APIs se usan de forma similar y logran el mismo resultado, con excepción de Eager Loading, donde en Sequelize se usa el modelo (o modelos) que se van a cargar durante la consulta, en TypeORM, se pasa la propiedad “eager: true” en el modelo o “relations: [‘relation-name’]” en la consulta.

Dependency Injection

DI es un patrón de diseño principalmente hecho para desacoplar funcionalidad y mejor testabilidad.

Sin embargo, Sequelize no se beneficia completamente de dicho patrón ya que el ORM no está pensado para usarse de tal forma. Parte de la lógica de infraestructura del mismo vive en su configuración y conexión y al final se termina creando solo un wrapper propio: hace posible el desacoplamiento y el mocking dentro de pruebas, pero no es limpio ni natural.

Por otro lado, TypeORM se beneficia completa y directamente de este patrón, ya que el “DataSource” fue pensado como una dependencia explícita, los repositorios son dependencias claras y el uso de decoradores ayuda a potenciar este patrón, dejándole la responsabilidad al mismo y al provider de injectar las dependencias correctamente.

Type-Safety

Sequelize ya de plano, no fue pensado para seguridad de tipado, la integración de TypeScript a partir de la versión 6 se siente como una beta o un parche y es mucho más incómodo de implementar teniendo en cuenta que otros ORMs se sienten mucho más naturales al momento de implementar tipado.

TypeORM es uno de ellos, basado en clases y asociaciones “at compile time” mediante “property association” y manejando correctamente la carga de entidades hijas, los tipos de cada uno se manejan de forma segura, natural y sin hacks.

Migraciones

Sequelize tiene una librería independiente para manejo de migraciones llamada “sequelize-cli”, la cual ya posee comandos de migración fáciles de familiarizarse. No obstante, a día de hoy siguen siendo CommonJS-only, quiere decir que no funcionan si hay sintaxis ESM dentro, lo cual rompe con la sintaxis actual de la organización.

Otro problema de esta librería es la configuración, es larga, tediosa, poco segura y todo se debe configurar de forma manual incluyendo rutas de archivos.

TypeORM, por otro lado, tiene el sistema de migraciones totalmente integrado, con soporte nativo a ESM y la configuración forma parte de la instancia “DataSource” inicial, cubriendo los problemas principales de “sequelize-cli”.

Aunque, uno de los problemas de dicho CLI es que es JS-first: los comandos de la CLI por defecto apuntan a archivos JS transpilados, para poder ejecutar directamente migraciones de archivos TS, se debe instalar “ts-node” y ejecutar el comando “npx typeorm-ts-node-esm ...”.

Otro problema es lo inconsistente que puede ser la documentación al momento de configurar los comandos para migraciones: Al momento de integrar el comando “migration:run” la documentación da el siguiente comando de ejecución: “npx typeorm-ts-node-esm migration:run -- -d path-to-datasource-config”, el cual falla porque el doble guión (“--”) no es necesario, pero no se hace alusión dentro de la documentación.

Quitando pequeños pormenores, los comandos son igual de fáciles de usar y de familiarizarse.

Testing

En este apartado, tomamos en cuenta vitest como test runner, y debido a que vitest tiene la capacidad de mockear un módulo entero, la comparación aquí será cuál toma menos en mockear evitando errores de tipado.

En Sequelize, el mockeo es directo y sencillo: Espías la respuesta del modelo o simplemente reemplazas el método del modelo que se esté usando por una función mock de vitest.

En TypeORM, el mockeo es relativamente más largo que el de Sequelize, debido a que se debe mockear el “DataSource” con base en la entidad que se usa en el momento al principio de los test. De ahí en adelante, esa instancia es la que se usa para realizar el mockeo al iniciar cada test.

En general, cada acercamiento es relativamente sencillo de implementar, uno relativamente más largo que el otro.

Salto de Dificultad

Teniendo en cuenta que el backend está en JS y cuya calidad de tipado en JSDocs sigue siendo algo que se deba mejorar, el salto hacia TypeORM + TypeScript es relativamente grande pero abordable.

Si cada quien es autónomo de implementar su propio patrón de desarrollo, entonces cada quien es libre de usar TypeORM al estilo de Sequelize (usando el “DataSource” en cada consulta) o mediante DI (solo se inyectan los repositorios mediante providers y decoradores), en caso de implementar la segunda, el laboratorio cuenta con decoradores para inyección de dependencias y manejo de servicios, así como un contenedor global básico para manejo de instancias de repositorios y servicios que pueden ser integrados al creador de repositorios o como ejemplos de implementación en la wiki de la organización.