Informe Técnico

Sistema de Gestión de Agencias de Agencias y Grupos K-POP

Autores

Aixa Bazán Rodríguez (C312)
Lianet Soto Aguirre (C312)
Amanda Medina Solis (C312)
Juan Miguel Maestre Rodríguez (C312)
Fabián Almeida Martínez (C311)

Índice

1 Introducción

Este informe técnico presenta el diseño y desarrollo de un sistema de gestión integral para agencias del entretenimiento, específicamente orientado a la industria K-pop. El sistema implementa una arquitectura moderna full-stack que combina un backend robusto basado en NestJS con un frontend desarrollado en React. A lo largo de este documento se detallan los aspectos fundamentales del sistema, incluyendo:

- El diccionario de datos completo con todas las entidades del sistema
- El diseño arquitectónico basado en Clean Architecture
- Los patrones de diseño implementados tanto en frontend como backend
- La organización modular del sistema
- Los flujos de trabajo principales

El objetivo principal de este sistema es proporcionar una plataforma escalable, mantenible y robusta para la gestión de artistas, grupos, contratos, actividades y todos los aspectos operativos de una agencia de entretenimiento.

2 Diccionario de Datos

En esta sección se presenta el modelo de datos completo del sistema, incluyendo todas las tablas principales y sus relaciones. El sistema utiliza PostgreSQL como base de datos y TypeORM para el mapeo objeto-relacional.

Para mejorar la legibilidad, las tablas se han organizado en grupos lógicos. Todas las tablas utilizan UUID como identificadores únicos y siguen convenciones de nomenclatura consistentes.

2.1 Tablas Principales

Tabla 1: Artist (Artista)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del artista (UUID)
stage_name	varchar	Not Null	Nombre artístico del artista
status	enum (ArtistStatus)	Not Null	Estado del artista (ACTIVO, INACTIVO)
birth_date	date	Not Null	Fecha de nacimiento
transition_date	date	Not Null	Fecha de debut

Tabla 2: Agency (Agencia)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único de la agencia (UUID)
date_fundation	date	Not Null	Fecha de fundación de la agencia
name	varchar	Not Null	Nombre de la agencia
place_id	varchar	Foreign Key (place.id), Not Null	Referencia al lugar de la agencia

Tabla 3: Group (Grupo)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del grupo (UUID)
name	varchar	Not Null	Nombre del grupo
num_members	integer	Not Null	Número de miembros
status	enum (GroupStatus)	Not Null, Default: EN_PAUSA	Estado del grupo
debutDate	date	Not Null	Fecha de debut
is_created	boolean	Not Null	Indica si el grupo está creado
concept	varchar	Not Null	Concepto artístico
agency_id	varchar	Foreign Key (agency.id), Not Null	Referencia a la agencia que lo representa

Tabla 4: Album (Álbum)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del álbum (UUID)
title	varchar	Not Null	Título del álbum
releaseDate	date	Not Null	Fecha de lanzamiento
mainProducer	varchar	Not Null	Productor principal
copiesSold	integer	Not Null	Número de copias vendidas
group_id	varchar	Foreign Key (group.id), Nullable	Referencia al grupo (opcional)
artist_id	varchar	Foreign Key (artist.id), Nullable	Referencia al artista (opcional)

Tabla 5: Song (Canción)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único de la canción (UUID)
name	varchar	Not Null	Nombre de la canción
entry_date	date	Not Null	Fecha de entrada
album_id	varchar	Foreign Key (album.id), Not Null	Referencia al álbum

Tabla 6: Contract (Contrato)

Columna	Tipo de Dato	Restricciones	Descripción
contract_id	varchar	Primary Key, Not Null	Identificador del contrato (UUID)
agencyID	varchar	Primary Key, Not Null	ID de la agencia
artistID	varchar	Primary Key, Not Null	ID del artista
startDate	date	Primary Key, Not Null	Fecha de inicio
end_date	date	Nullable	Fecha de fin
status	enum (ContractStatus)	Not Null, Default: ACTIVO	Estado del contrato
conditions	varchar	Not Null	Condiciones del contrato
distPercentage	decimal	Not Null	Porcentaje de distribución

Tabla 7: Activity (Actividad)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único de la actividad (UUID)
classification	enum (ActivityClassification)	Not Null	Clasificación de la actividad
type	enum (ActivityType)	Not Null	Tipo de actividad

Tabla 8: Apprentice (Aprendiz)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del aprendiz (UUID)

Columna	Tipo de Dato	Restricciones	Descripción
full_name	varchar	Not Null	Nombre completo
entry_date	date	Not Null	Fecha de ingreso
age	integer	Not Null	Edad
status	enum (ApprenticeStatus)	Not Null	Estado del aprendiz
trainingLevel	enum (ApprenticeTrainingLevel)	Not Null, Default: PRINCIPIANTE	Nivel de entrenamiento
agency_id	varchar	Foreign Key (agency.id), Not Null	Referencia a la agencia

Tabla 9: Award (Premio)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del premio (UUID)
name	varchar	Not Null	Nombre del premio
date	date	Not Null	Fecha del premio
album_id	varchar	Foreign Key (album.id), Nullable	Referencia al álbum premiado

Tabla 10: BillboardList (Lista Billboard)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único de la lista (UUID)
name	varchar	Not Null	Nombre de la lista
public_date	date	Not Null	Fecha de publicación
scope	enum (BillboardListScope)	Not Null	Alcance de la lista
end_list	integer	Not Null	Posición final

Tabla 11: Income (Ingreso)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del ingreso (UUID)

Columna	Tipo de Dato	Restricciones	Descripción
activity_id	varchar	Primary Key, Not Null	ID de la actividad
incomeType	enum (IncomeType)	Not Null, Default: EFECTIVO	Tipo de ingreso
mount	decimal(10,2)	Not Null	Monto del ingreso
date	date	Not Null	Fecha del ingreso
responsible	varchar	Not Null	Responsable del ingreso

Tabla 12: Place (Lugar)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del lugar (UUID)
place	varchar	Not Null	Nombre del lugar

Tabla 13: Responsible (Responsable)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Not Null	Identificador único del responsable (UUID)
name	varchar	Not Null	Nombre del responsable

Tabla 14: Users (Usuarios)

Columna	Tipo de Dato	Restricciones	Descripción
id	varchar	Primary Key, Auto-generated, Not Null	Identificador único del usuario
username	varchar	Unique, Not Null	Nombre de usuario
password	varchar	Not Null	Contraseña
role	enum (UserRole)	Not Null	Rol del usuario
isActive	boolean	Not Null	Indica si está activo
agency	varchar	Not Null	Agencia del usuario
artist	varchar	Not Null	Artista asociado

2.2 Tablas de Relación

En esta subsección se presentan las tablas que manejan relaciones many-to-many entre las entidades principales del sistema.

Tabla 15: Song_Billboard (Canción en Billboard)

Columna	Tipo de Dato	Restricciones	Descripción
song_id	varchar	Primary Key, Not Null	ID de la canción
billboard_list_id	varchar	Primary Key, Not Null	ID de la lista de billboard
place	integer	Not Null	Posición en la lista
entry_date	date	Not Null	Fecha de entrada

Tabla 16: Artist_Activity (Artista-Actividad)

Columna	Tipo de Dato	Restricciones	Descripción
artist_id	varchar	Primary Key, Foreign Key (artist.id)	ID del artista
activity_id	varchar	Primary Key, Foreign Key (activity.id)	ID de la actividad

Tabla 17: Group_Activity (Grupo-Actividad)

Columna	Tipo de Dato	Restricciones	Descripción
group_id	varchar	Primary Key, Foreign Key (group.id)	ID del grupo
activity_id	varchar	Primary Key, Foreign Key (activity.id)	ID de la actividad

Tabla 18: Activity_Date (Actividad-Fecha)

Columna	Tipo de Dato	Restricciones	Descripción
activity_id	varchar	Primary Key, Foreign Key (activity.id)	ID de la actividad
date	date	Primary Key	Fecha de la actividad

Tabla 19: Activity_Responsible (Actividad-Responsable)

Columna	Tipo de Dato	Restricciones	Descripción
activity_id	varchar	Primary Key, Foreign Key (activity.id)	ID de la actividad

Columna	Tipo de Dato	Restricciones	Descripción
responsible_id	varchar	Primary Key, Foreign Key (responsible.id)	ID del responsable

Tabla 20: Activity_Place (Actividad-Lugar)

Columna	Tipo de Dato	Restricciones	Descripción
activity_id	varchar	Primary Key, Foreign Key (activity.id)	ID de la actividad
place_id	varchar	Primary Key, Foreign Key (place.id)	ID del lugar

Tabla 21: Artist_Group_Membership (Artista-Grupo Membresía)

Columna	Tipo de Dato	Restricciones	Descripción
artist_id	varchar	Primary Key, Foreign Key (artist.id)	ID del artista
group_id	varchar	Primary Key, Foreign Key (group.id)	ID del grupo
start_date	date	Primary Key, Not Null	Fecha de inicio de la membresía en el grupo
rol	varchar	Not Null	Rol del artista dentro del grupo (ej: vocalista, bailarín, rapero)
artist_debut_date	date	Not Null	Fecha de debut del artista en el grupo
end_date	date	Nullable	Fecha de fin de la membresía (null si está activo en el grupo)

Tabla 22: Artist_Collaboration (Colaboración de Artista)

Columna	Tipo de Dato	Restricciones	Descripción
artist1_id	varchar	Primary Key, Foreign Key (artist.id)	ID del primer artista
artist2_id	varchar	Primary Key, Foreign Key (artist.id)	ID del segundo artista
date	date	Primary Key, Not Null	Fecha en la que ocurrió la colaboración entre los artistas

Tabla 23: Artist_Group_Collaboration (Colaboración Artista-Grupo)

Columna	Tipo de Dato	Restricciones	Descripción
artist_id	varchar	Primary Key, Foreign Key (artist.id)	ID del artista
group_id	varchar	Primary Key, Foreign Key (group.id)	ID del grupo
date	date	Primary Key, Not Null	Fecha en la que ocurrió la colaboración

Tabla 24: Artist_Agency_Membership (Artista-Agencia Membresía)

Columna	Tipo de Dato	Restricciones	Descripción
artist_id	varchar	Primary Key, Foreign Key (artist.id)	ID del artista
agency_id	varchar	Primary Key, Foreign Key (agency.id)	ID de la agencia
start_date	date	Primary Key, Not Null	Fecha de inicio de la membresía del artista en la agencia
end_date	date	Nullable	Fecha de fin de la membresía (null si la membresía está activa)

Tabla 25: Apprentice_Evaluation (Evaluación de Aprendiz)

Columna	Tipo de Dato	Restricciones	Descripción
apprentice_id	varchar	Primary Key, Foreign Key (apprentice.id)	ID del aprendiz
evaluation_id	varchar	Primary Key	ID de la evaluación

2.3 Consideraciones Técnicas

- **Identificadores Únicos:** Todas las tablas usan UUID generados automáticamente
- **Manejo de Fechas:** Formato ISO (YYYY-MM-DD)
- **Relaciones:** Implementadas con claves foráneas y tablas de relación
- **Consistencia:** Configuración de cascada según dependencias de negocio

3 Diseño de la Aplicación

Esta sección describe la arquitectura completa del sistema, incluyendo tanto el frontend como el backend, y cómo se integran para formar una aplicación full-stack cohesiva.

3.1 Arquitectura Full-Stack y Visión General

La aplicación es un sistema full-stack para la gestión integral de agencias del entretenimiento, implementado con una arquitectura moderna y separada que comprende un **backend API REST** y un **frontend de página única (SPA)**. El backend, construido con Node.js y NestJS, adopta una arquitectura en capas basada en los principios de *Clean Architecture* y *Domain-Driven Design (DDD)*. Por su parte, el frontend, desarrollado con React y TypeScript, sigue un patrón de diseño basado en componentes y estados gestionados por contextos, lo que permite una interfaz de usuario dinámica y fácil de mantener.

3.2 Diseño del Backend: Núcleo de Negocio y Servicios API

El backend constituye el núcleo del sistema, responsable de albergar toda la lógica de negocio, gestionar la persistencia de datos y exponer servicios a través de una API REST. Su estructura se describe en la sección 6

3.3 Diseño del Frontend: Interfaz de Usuario por Componentes

El frontend es una aplicación construida con React y TypeScript, diseñada para ofrecer una experiencia de usuario fluida e intuitiva para la gestión de agencias, artistas, grupos y actividades. Su arquitectura separa claramente la presentación visual, la gestión del estado y la comunicación con el backend.

La interfaz de usuario se construye a partir de **componentes React** reutilizables y modulares, organizados por roles funcionales (como `Admin`, `Artist` o `Manager`) y por elementos de interfaz genéricos (`ui`). Estos componentes, principalmente presentacionales, se encargan de renderizar la vista y capturar las interacciones del usuario. La lógica de estado y los efectos secundarios (como las llamadas a la API) se encapsulan en **contextos de React** específicos por dominio (por ejemplo, `AgencyContext`, `ArtistContext`). Estos contextos actúan como contenedores de estado global para su dominio, exponiendo funciones (acciones) que los componentes pueden invocar.

3.4 Integración, Comunicación y Flujo de Datos Full-Stack

La integración entre el frontend y el backend se realiza mediante una API RESTful documentada y consistente. El frontend actúa como un cliente que consume los endpoints proporcionados por los controladores del backend. La autenticación se gestiona mediante JSON Web Tokens (JWT).

3.5 Resumen Tecnológico y Conclusión del Diseño

- **Backend:** Node.js, TypeScript, NestJS, TypeORM, PostgreSQL, JWT.
- **Frontend:** React, TypeScript, Context API, React Router, Axios/Fetch.
- **Arquitectura Backend:** Clean Architecture con 4 capas (Presentación, Aplicación, Dominio, Infraestructura).
- **Arquitectura Frontend:** Patrón basado en Componentes, Contextos (State) y Servicios.

- **Comunicación:** API REST sobre HTTP/HTTPS, intercambio de datos en JSON, autenticación con JWT.

4 Esquema de Clases

Esta sección describe la organización de las clases en el sistema según los principios de Clean Architecture.

4.1 Organización por Capas

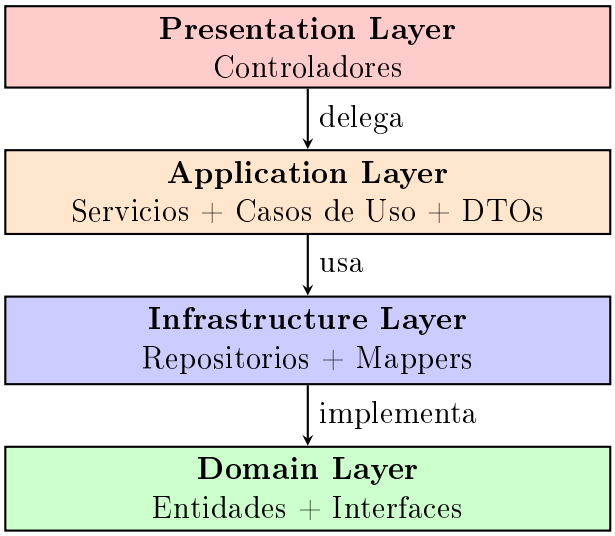

Figura 1: Clean Architecture - Dependencias entre capas

4.2 Componentes por Capa

4.2.1. Capa de Presentación

- `AgencyController`, `ArtistController`, `GroupController`
- Manejan peticiones HTTP y respuestas
- Validan datos de entrada

4.2.2. Capa de Aplicación

- `AgencyService`, `ArtistService`, `GroupService`
- `CreateAgencyUseCase`, `CreateArtistUseCase`
- Orquestan operaciones de negocio

4.2.3. Capa de Dominio

- Entidades: `Agency`, `Artist`, `Group`
- Interfaces: `IAgencyRepository`, `IArtistRepository`
- Contienen la lógica de negocio pura

4.2.4. Capa de Infraestructura

- `AgencyRepositoryImpl`, `ArtistRepositoryImpl`
- `AgencyEntity`, `ArtistEntity` (TypeORM)
- Implementan acceso a datos y servicios externos

5 Esquema MERX de la solución final

Se presenta el modelo que determina la estructura de la base de datos del proyecto en el archivo adjunto "Diagrama Gestión de Agencias.jpg".

6 Arquitectura del Sistema

Como hemos visto, se ha adoptado la Clean Architecture como modelo central del sistema, materializada en una estructura de cuatro capas bien diferenciadas. Esta elección garantiza un desacoplamiento total entre los componentes, separando claramente el dominio, la aplicación, la infraestructura y la presentación mediante contratos y abstracciones definidos con precisión. El flujo de dependencias sigue rigurosamente el Principio de Inversión de Dependencias (DIP), orientándose exclusivamente hacia el centro del sistema —la capa de dominio—. Esta organización permite el reemplazo modular de componentes tecnológicos externos sin perturbar el núcleo del negocio, estableciendo una base sólida para la evolución y mantenimiento del software.

La capa de presentación constituye el punto de contacto del sistema con los clientes externos. Esta capa se encarga de manejar todas las interfaces de entrada y salida, incluyendo la gestión de rutas HTTP y la coordinación de respuestas a través de controladores específicos (como `AgencyController`, `ArtistController` o `ContractController`), así como la validación, transformación de datos de entrada y el control de acceso. Su responsabilidad principal es recibir y validar las peticiones HTTP, formatear las respuestas, gestionar errores y aplicar los mecanismos de autenticación y autorización definidos.

La capa de aplicación se dedica a orquestar los casos de uso y a coordinar la lógica específica de la aplicación. En esta capa residen los diferentes casos de uso (como `CreateAgencyUseCase` o `ActivateGroupUseCase`) que implementan flujos de trabajo concretos, junto con servicios transversales (por ejemplo, `AuthService`) y los objetos de transferencia de datos (DTOs) que facilitan la validación y el formateo de la información. Sus funciones principales incluyen la coordinación de operaciones complejas, la validación de reglas de negocio transversales, la transformación de datos entre capas y la gestión de transacciones.

El núcleo del sistema reside en la capa de dominio, que encapsula toda la lógica y las reglas de negocio puras, manteniéndolas completamente independientes de cualquier framework o tecnología externa. Aquí se definen las entidades fundamentales con identidad propia (como `Agency`, `Artist`, `Group` y `Contract`), las enumeraciones que restringen los valores de dominio (como `ArtistStatus` o `GroupStatus`) y las interfaces de los repositorios (como `IAgencyRepository`) que establecen los contratos para el acceso a datos. Esta capa es responsable de la encapsulación de las reglas de negocio, la validación de invariantes, la definición del comportamiento del dominio y la especificación de los con-

tratos de persistencia, asegurando que el corazón de la aplicación permanezca estable y comprensible.

Finalmente, la capa de infraestructura proporciona todas las implementaciones concretas de las abstracciones definidas en las capas superiores, actuando como la capa más externa y dependiente. Incluye la configuración de módulos de NestJS (como `AgencyModule` y `DataBaseModule`), el mapeo objeto-relacional de las entidades de base de datos (como `ArtistEntity` y `AgencyEntity`), las implementaciones concretas de los repositorios (como `ArtistRepositoryImpl`) y los servicios de integración con sistemas externos, como la conexión a PostgreSQL mediante TypeORM o el servicio de autenticación JWT. Sus responsabilidades abarcan la implementación de la persistencia, la configuración de frameworks, la integración con servicios externos y el manejo de todos los detalles técnicos de la infraestructura.

Esta arquitectura en capas, organizada por capacidades de negocio y no por tipos técnicos horizontales, fomenta una clara separación de responsabilidades. El resultado es un sistema altamente mantenible, donde los cambios pueden localizarse en módulos específicos, y altamente testable, ya que cada capa puede verificarse de forma aislada mediante pruebas unitarias, mientras que las pruebas de integración y end-to-end validan los contratos y flujos completos. Además, la independencia del núcleo de dominio frente a los frameworks evita el *vendor lock-in* y permite una migración tecnológica progresiva, asegurando la escalabilidad y la longevidad del sistema de gestión de agencias.

7 Módulos de NestJS

Esta sección describe cómo se organiza el código del backend en módulos, siguiendo las mejores prácticas de NestJS para mantener un código limpio y mantenible.

Los módulos de NestJS actúan como contenedores de composición que agrupan componentes relacionados y gestionan sus dependencias.

7.1 Tipos de Módulos Implementados

1. **Módulos de Dominio** (AgencyModule, ArtistModule, GroupModule, etc.)
   - Agrupan controladores, servicios y repositorios de un dominio específico
   - Gestionan dependencias intra-dominio
   - Facilitan la separación de responsabilidades
2. **Módulo de Base de Datos** (DataBaseModule)
   - Configura la conexión TypeORM
   - Registra todas las entidades
   - Proporciona repositorios globales
3. **Módulo Raíz** (AppModule)
   - Importa todos los módulos de dominio
   - Configura middleware global
   - Define configuración global de la aplicación

7.2 Beneficios de la Modularización

La modularización ofrece importantes ventajas para el desarrollo y mantenimiento del sistema. En primer lugar, mejora la **mantenibilidad** al permitir que los cambios se localicen en módulos específicos, reduciendo el riesgo de afectar otras partes del sistema. Además, facilita la **reutilización** de código, ya que los módulos pueden importarse selectivamente en diferentes partes de la aplicación cuando se necesite su funcionalidad. Esto también mejora la **testabilidad**, pues cada módulo puede probarse de manera aislada, permitiendo identificar y corregir errores con mayor precisión. Finalmente, la arquitectura modular favorece la **escalabilidad** del sistema, ya que se pueden agregar nuevos módulos sin afectar los existentes, lo que permite expandir la funcionalidad de manera ordenada y controlada.

8 Patrones Implemetados

8.1 Patrones de Diseño

8.1.1. Patrón Singleton en NestJS

NestJS implementa nativamente el patrón Singleton como parte fundamental de su sistema de inyección de dependencias, garantizando que todos los proveedores marcados con el decorador `@Injectable()` sean instanciados una única vez durante el ciclo de vida de la aplicación. Esta implementación ofrece importantes beneficios como la gestión automática del ciclo de vida de las instancias, consistencia en el estado de los servicios a lo largo de todo el sistema, optimización en el uso de recursos mediante la reutilización de instancias, y una integración transparente con el resto del ecosistema del framework.

La implementación técnica se realiza mediante el contenedor de inyección de dependencias de NestJS, que actúa como registro centralizado gestionando las instancias únicas de cada proveedor. Cuando un servicio es solicitado por múltiples componentes, el contenedor siempre retorna la misma instancia, eliminando posibles inconsistencias y reduciendo el overhead de creación de objetos.

8.1.2. Patrón Decorator en NestJS y TypeORM

El patrón Decorator es ampliamente utilizado en NestJS y TypeORM para añadir funcionalidades de manera declarativa sin modificar la estructura base de las clases. Esta aproximación permite una configuración limpia y auto-documentada del código, separando adecuadamente las preocupaciones transversales de la lógica de negocio principal.

En NestJS, este patrón se manifiesta a través de decoradores como `@Controller()` para definir controladores de endpoints API, `@Injectable()` para marcar clases como proveedores inyectables, y los decoradores HTTP `@Get()`, `@Post()`, `@Put()`, `@Delete()` entre muchos otros que permiten configurar rutas y métodos de forma expresiva. Por su parte, TypeORM emplea decoradores como `@Entity()` para definir entidades de dominio, `@Column()` para mapear propiedades a columnas de base de datos, `@PrimaryGeneratedColumn()` para definir claves primarias, y decoradores de relaciones como `@OneToMany()` y `@ManyToOne()`, constituyendo solo algunos ejemplos de la extensa gama de decoradores disponibles en estas tecnologías.

Las principales ventajas de esta implementación incluyen la capacidad de escribir código altamente declarativo que se auto-documenta, la clara separación de concerns que

permite mantener la lógica de negocio libre de contaminación con aspectos transversales, y la flexibilidad para añadir o remover comportamientos mediante la simple adición o eliminación de decoradores.

Conclusión

La combinación estratégica de los patrones Singleton y Decorator en nuestra implementación proporciona una base técnica sólida que soporta la evolución continua del sistema. Esta aproximación asegura la consistencia en el estado de la aplicación mediante el Singleton, y mantiene un código limpio y declarativo mediante el uso extensivo de Decorators, garantizando así la calidad y mantenibilidad del código a largo plazo.

8.2 Patrones de Visualización

Patrón Seleccionado: MVVM (Model–View–ViewModel)

Para el frontend de nuestra aplicación web seleccionamos el patrón **MVVM (Model–View–ViewModel)**, debido a que se adapta de forma óptima a la arquitectura del framework utilizado (React) y permite lograr una separación clara de responsabilidades, mejorar la mantenibilidad del código y favorecer la escalabilidad del sistema.

8.2.1. Justificación

El patrón MVVM es ampliamente utilizado en aplicaciones interactivas debido a que:

- Separa la lógica de la interfaz del resto del sistema.
- Facilita el flujo de datos reactivo.
- Mantiene sincronizado el estado entre la vista y el modelo.
- Mejora la testabilidad y organización del código.

React, junto con Context API, permite implementar MVVM de manera natural y eficiente.

8.2.2. Aplicación del Patrón en el Proyecto

La implementación de MVVM en el proyecto se organiza en tres capas principales:

Vista (View)

Las vistas corresponden a los componentes React escritos en `.tsx`, tales como:

- `ApprenticesManagement.tsx`
- `ArtistManagement.tsx`
- `PlaceManagement.tsx`

Estas vistas manejan:

- El renderizado de la interfaz de usuario.

- Los formularios y eventos del usuario.
- Interacciones puramente visuales.

Las vistas **no contienen lógica de negocio** ni realizan llamadas a la API directamente.

ViewModel (Contextos de React)

El ViewModel está implementado mediante **React Context API**, por ejemplo:

- `ApprenticeContext.tsx`
- `ArtistContext.tsx`
- `PlaceContext.tsx`

Su responsabilidad es:

- Gestionar el estado de cada entidad (carga, error, lista de datos).
- Exponer funciones como:
  - `fetchAll()`
  - `create()`
  - `update()`
  - `delete()`
- Obtener datos desde los servicios.
- Preparar la información para que la vista la consuma.

De esta forma, el ViewModel es el intermediario entre la vista y el modelo.

Modelo (Model)

El modelo incluye:

- DTOs de creación y respuesta.
- Servicios de acceso a datos (servicios CRUD genéricos).
- Enumeraciones del dominio.
- Entidades definidas en el backend.

Los servicios se encargan únicamente de:

- Realizar operaciones CRUD contra el backend (NestJS).
- Retornar datos tipados.

La vista nunca se comunica directamente con el modelo, lo hace siempre por medio del ViewModel.

8.3 Patrones de Acceso a Datos

El sistema implementa una estrategia híbrida de patrones de acceso a datos que combina implementaciones propias con el aprovechamiento de funcionalidades nativas de TypeORM. Esta aproximación garantiza independencia tecnológica mientras se maximiza la productividad del desarrollo.

8.3.1. Data Mapper Pattern (Implementación Propia)

Propósito: Mantener una separación completa entre el modelo de dominio puro y el modelo de persistencia, asegurando que las entidades de negocio permanezcan libres de contaminación con concerns de infraestructura.

Implementación: Hemos implementado nuestro propio patrón Data Mapper mediante una interfaz genérica `IMapper` que define el contrato para transformación bidireccional entre entidades de dominio y entidades de persistencia. Cada entidad del sistema cuenta con su mapper específico que encapsula la lógica de conversión.

Uso: Este patrón se aplica sistemáticamente en todos los repositorios para convertir entre la capa de dominio y la capa de persistencia, garantizando que las entidades de dominio permanezcan inmutables y libres de decoradores de TypeORM.

8.3.2. Repository Pattern (Implementación Híbrida)

Propósito: Proporcionar una abstracción cohesiva del acceso a datos que oculte los detalles de implementación específicos de la capa de persistencia, permitiendo al dominio operar sobre contratos en lugar de implementaciones concretas.

Implementación Híbrida: Combinamos nuestro propio contrato de repositorio `Repository` con la implementación concreta utilizando `Repository` de TypeORM. Esta aproximación nos permite aprovechar la potencia de TypeORM mientras mantenemos la independencia del ORM.

Uso: Todos los repositorios específicos (como `ResponsibleRepository`) extienden `BaseRepository (Implementación concreta de la clase abstracta Repository)` y se inyectan en los servicios, manteniendo el principio de inversión de dependencias.

8.3.3. Unit of Work Pattern (TypeORM)

Propósito: Gestionar transacciones complejas y mantener la consistencia de datos a través de operaciones atómicas que involucran múltiples entidades.

Implementación por TypeORM: Utilizamos el `EntityManager` de TypeORM que implementa naturalmente el patrón Unit of Work, realizando seguimiento automático de cambios y gestionando la persistencia atómica.

Aplicación: Este patrón se utiliza en operaciones complejas que requieren consistencia transaccional, como la creación de un grupo musical con todos sus artistas y contratos asociados.

8.3.4. Query Builder Pattern (TypeORM)

Propósito: Construir consultas complejas de manera programática y type-safe, manteniendo la expresividad y el control sobre el SQL generado.

Implementación por TypeORM: Utilizamos `createQueryBuilder` de TypeORM para consultas que requieren joins complejos, condiciones dinámicas o agrupaciones avanzadas.

Aplicación: Este patrón se utiliza para consultas de reporting, búsquedas avanzadas y cualquier operación de lectura que supere las capacidades de los métodos básicos del repositorio.

Conclusión de la Estrategia de Acceso a Datos

La combinación estratégica de estos patrones nos permite mantener una arquitectura limpia y desacoplada mientras aprovechamos las capacidades robustas de TypeORM. Los patrones implementados directamente por nosotros (Data Mapper, Repository con abstracción propia) garantizan la independencia del dominio, mientras que los patrones proporcionados por TypeORM (Unit of Work, Query Builder) nos ofrecen potencia y productividad sin sacrificar el control sobre las operaciones complejas de persistencia.

9 Conclusiones Técnicas

El desarrollo del sistema ha permitido implementar un modelo de datos completo y normalizado para la gestión integral de agencias de entretenimiento, logrando la automatización de procesos clave como la gestión de artistas, contratos, actividades y métricas de rendimiento. El sistema demuestra capacidad para manejar las complejidades del dominio específico, incluyendo relaciones many-to-many, estados transaccionales y flujos de trabajo empresariales.

9.1 Logros Principales

- Arquitectura sólida basada en Clean Architecture
- Separación clara entre frontend y backend
- Modelo de datos completo y normalizado
- Implementación de patrones de diseño probados
- Código mantenible, testeable y escalable