Seminario Integrador
para
Gestión de agencias K-Pop.

K-Pop-World

Equipo #1:

1. Ronald Provance Valladares C-312 @Ron_1301
2. Agustín Alberto Carbajal Romero C-312 @lil_agus03
3. Daniel Amaranto Mares García C-312 @amarantomc
4. Dylan Ramsés Cabrera Morales C-312 @dylan2108
5. Sheila Roque Alemán C-312 @Luminara03

1. Introducción

En los últimos años, la industria del K-Pop ha alcanzado una relevancia global sin precedentes, extendiéndose más allá del ámbito musical y convirtiéndose en un fenómeno cultural y económico de gran impacto. El crecimiento acelerado de grupos, artistas solistas, giras internacionales y colaboraciones comerciales ha generado una creciente complejidad en los procesos de gestión interna de las agencias. Este aumento en la escala y en la demanda ha evidenciado la necesidad de soluciones tecnológicas capaces de organizar, monitorear y optimizar la administración del talento, permitiendo a las empresas manejar de manera eficiente agendas, contratos, actividades y relaciones profesionales en un entorno cada vez más competitivo.

1.1. Alcance del Producto

El sistema abarca la gestión integral de agencias de K-pop, incluyendo el registro de aprendices, artistas, grupos, álbumes, actividades, contratos e ingresos. La aplicación web permitirá a los usuarios realizar consultas, programar actividades, verificar conflictos de agenda, calcular ingresos y generar reportes exportables en PDF, optimizando la operatividad en la industria del entretenimiento coreano.

1.2. Descripción General del Producto

1.2.1. Perspectiva

El sistema se concibe como una solución integral que reemplaza la gestión manual o dispersa de agencias y managers. Funciona como una herramienta de trabajo unificada en la que los usuarios pueden:

- Consultar información actualizada en tiempo real.
- Agilizar la toma de decisiones en base a reportes y registros automáticos.
- Evitar conflictos en la agenda de artistas y grupos.
- Mantener un historial transparente de la popularidad de las canciones, contratos y actividades.

1.2.2. Características de los usuarios

- Managers: Gestionan artistas y programan actividades
- Directores: Aprueban la formación de grupos y validan información
- Artistas: Consultan agendas,solicitan la creación de grupos, reportan disponibilidad y visualizan ingresos.
- Aprendices: Consultan agendas e historial de evaluaciones,solicitan la creación de grupos,
- Administradores: Gestionan entidades del sistema y configuraciones

1.2.3. Restricciones generales

- La información debe ser verificada y validada por superiores
- Manejo de relaciones complejas (artistas en grupos multigencia).
- Garantizar confidencialidad e integridad de datos sensibles.

2. Requerimientos Específicos

2.1. Requerimientos Funcionales

El sistema debe permitir:

1. **Gestión de agencias**: registrar y consultar información de cada agencia incluyendo nombre, ubicación, fecha de fundación, artistas y grupos bajo contrato.
2. **Gestión de aprendices**: registrar información personal de los aprendices (nombre completo, edad, fecha de ingreso, nivel de entrenamiento, historial de evaluaciones y estado actual)
3. **Formación de grupos**: registrar el proceso de selección de aprendices para conformar grupos, convirtiéndolos en artistas y almacenar sus datos (nombre artístico y real, fecha de nacimiento, rol en el grupo, fecha de debut, estado actual).
4. **Gestión de grupos**: registrar y actualizar la información de los grupos (nombre, fecha de debut, concepto, cantidad de miembros, estado, agencia que los representa)
5. **Gestión de álbumes**: registrar datos de cada álbum (título, fecha de lanzamiento, número de canciones, productor principal, cantidad de copias vendidas, premios recibidos y posiciones en listas Billboard)
6. **Gestión de actividades**: programar y consultar actividades grupales (lanzamiento de álbumes, giras, conciertos, promociones) e individuales (dramas, campañas publicitarias, programas de variedades) en un calendario, evitando conflictos de horarios.
7. **Agenda de artistas**: permitir a cada artista consultar sus actividades, confirmar asistencia, reportar disponibilidad y enviar solicitudes de ajuste de horario al mánager.
8. **Gestión de contratos**: registrar con información del artista asociado, fechas de inicio y fin, condiciones principales, estado (activo, en renovación, finalizado, rescindido) y porcentaje de distribución de ingresos.
9. **Gestión de ingresos**: registrar los ingresos provenientes de actividades grupales e individuales especificando tipo, monto, fecha, actividad asociada y responsable de gestión.
10. **Historial profesional**: mantener un registro completo de la trayectoria de cada artista (agencias, grupos, debuts, colaboraciones y contratos)

11. **Control de formación de grupos por artistas**: permitir que un aprendiz o solista proponga la creación de un grupo, con aprobación final del director de la agencia.

12. **Exportación de información**: proveer la funcionalidad de exportar los reportes y resultados obtenidos a ficheros PDF, con posibilidad de ordenar y filtrar columnas.

13. **Reportes de éxito**: generar reportes (evolución de ventas de álbumes, premios obtenidos, posiciones en rankings musicales)

14. **Obtener listado de artistas activos por agencia**: Permitir a los mánagers obtener un listado de todos los artistas activos asociados a su agencia, mostrando el nombre artístico, grupo al que pertenecen y estado actual.

15. **Consultar calendario de actividades de un grupo**: Consultar el calendario completo de actividades programadas para un grupo en específico en un rango de fechas, con detalles de lugar, hora y tipo de actividad.

16. **Verificación de conflictos de agenda al programar**: Verificar qué artistas tienen conflictos de agenda al momento de programar una nueva actividad, cruzando las actividades grupales e individuales registradas en el sistema para un periodo de tiempo dado

17. **Identificación de artistas con contratos activos y debut**: Identificar a todos los artistas que han participado en al menos un debut y que actualmente se encuentren con contratos activos, mostrando también los datos del grupo y del contrato

18. **Cálculo detallado de ingresos por artista**: Calcular el total de ingresos generados por cada artista, considerando ingresos grupales e individuales durante un periodo específico, mostrando además sus principales éxitos como solista y trabajando en el último grupo que estuvo.

19. **Detección y listado de artistas con múltiples cambios de agencia/grupo**: Detectar a los artistas que han cambiado de agencia al menos dos veces y que han participado en más de un grupo, mostrando su historial cronológico completo de sus contratos, actividades y debuts.

20. **Obtención de historial de solistas ex-miembros de grupos exitosos**: Obtener el historial profesional completo de los artistas solistas que, en algún momento de su carrera, hayan sido miembros de un grupo disuelto y que a su vez haya tenido al menos un álbum que haya vendido más de un millón de copias y cuyas canciones hayan alcanzado el top 100 de las listas de Billboard internacionales y nacionales en el año siguiente a su lanzamiento.

2.2. Requerimientos no funcionales

2.2.1. Usabilidad

- La aplicación debe mostrar únicamente las funciones que cada tipo de usuario necesita para su rol en la industria del entretenimiento coreano:
    - Administrador: pantalla completa de gestión con acceso inmediato a todas las entidades

- Manager: vistas centradas en su agencia, con acceso directo a calendario de grupos, registros de artistas, validación de conflictos de agenda, registro de actividades y carga de información de ventas y premios.
- Artista: panel simplificado que muestra su calendario personal, actividades grupales, ingresos propios, historial profesional y notificaciones de cambios aceptados por el director.
- Aprendiz: vista del historial de evaluaciones,agenda de entrenamientos,perfil personal y notificaciones de cambios aceptados por el director.
- Director: vistas para visualizar información de los grupos asociados a la agencia a la cual dirige y validaciones de cambios

- La estructura visual debe permitir que:
  - Los managers puedan ver el calendario del grupo o detectar conflictos sin navegar por menús profundos
  - El administrador pueda saltar entre entidades relacionadas sin perder contexto, ya que gestiona datos complejos: grupos con miembros de distintas agencias, contratos vigentes, cambios de agencia, álbumes, ventas, etc
- El sistema debe representar de forma clara cada estado:
  - Estado del grupo: activo / en pausa / disuelto
  - Estado del artista: activo / en pausa / inactivo
  - Estado del aprendiz: en entrenamiento / en selección / transferido
  - Estado del contrato: activo / en renovación / finalizado / rescindido

  El usuario debe distinguir estos estados visualmente sin depender de texto extenso
- Coordinación de actividades grupales e individuales:
  - El calendario debe permitir ver superposiciones y conflictos visualmente
  - Los managers deben poder filtrar por tipo de actividad: grupal e individual.
  - Los artistas deben poder diferenciar sus actividades propias de las del grupo
- Los términos usados en la interfaz deben corresponder solo a la industria del K-pop: debut, actividad individual, actividad grupal, copias vendidas, Billboard, en pausa, disuelto, aprendiz, rol del grupo, etc. Esto evita ambigüedades y mantiene la coherencia con el contexto real del sistema.
- El sistema debe prevenir errores comunes:
  - No permitir programar una actividad si el artista ya tiene un conflicto de horario
  - Prevenir que un grupo sea registrado sin agencia
  - Evitar registrar un álbum sin productor principal o sin fecha de lanzamiento.
  - Impedir estados incoherentes: por ejemplo, artista inactivo con actividad programada.

2.2.2. Seguridad

- **Confidencialidad**: La información manejada por el sistema está protegida contra accesos no autorizados y su divulgación en reposo y en tránsito.
  - **Fortaleza**: Contraseñas almacenadas con bcrypt (12 iteraciones)
  - **Debilidad**: Archivo .env con credenciales en texto plano
  - **Riesgo**: JWT Secret con valor por defecto inseguro
  - **Fortaleza**: CORS configurado para desarrollo
- **Integridad**: La información estará protegida contra corrupción e inconsistencias, manteniendo fidelidad con la fuente de autoridad.
  - **Fortaleza**: Patrón UnitOfWork con transacciones
  - **Riesgo**: Validación limitada en DTOs
  - **Debilidad**: Falta sanitización centralizada
  - **Riesgo**: Ausencia de auditoría de cambios
- **Disponibilidad**: Los usuarios autorizados tendrán acceso garantizado a la información, sin que los mecanismos de seguridad obstaculicen el acceso.
  - **Debilidad**: No hay rate limiting implementado
  - **Riesgo**: Falta de health checks
  - **Debilidad**: No hay protección contra DDoS
  - **Riesgo**: Ausencia de monitorización

2.2.3. Diseño e Implementación

- Aplicación desacoplada, extensible y mantenible
- Sistema compatible con múltiples plataformas (Web accesible desde diferentes sistemas operativos)
- Todo el proyecto utilizará Git como sistema de control de versiones
- Github Flow como flujo de trabajo: Creación de ramas para cada tarea o funcionalidad. Ninguna modificación podrá entrar a main sin Pull Request y sin revisión de otro integrante del equipo. El historial de commits debe ser claro y descriptivo (nada de fix, arreglos o mensajes irrelevantes)
- La organización del trabajo, división de tareas y seguimiento del avance se realizará usando GitHub Projects
- React como framework para frontend: Interfaz para Administrador, Director, Manager, Artista y Aprendiz. Comunicación con el backend mediante API REST utilizando fetch
- Express.js como framework para backend: El backend funciona como API REST centralizada. Maneja autenticación JWT, lógica de negocio, verificación de roles, detección de conflictos de agenda, integración con Prisma y PostgreSQL.

- PostgreSQL para gestionar la BD: Todas las entidades del dominio deben persistir en Postgres. La estructura debe cumplir integridad referencial
- Prisma como ORM: El acceso a la base de datos debe realizarse exclusivamente a través de Prisma ORM. Prisma debe encargarse de migraciones, validación y consultas.

2.3. Requerimientos de entorno

- Hardware requerido
  - Computadora con capacidades mínimas de ejecutar un navegador moderno
  - Conexión estable a Internet para acceder al sistema y sincronizar calendarios
  - En caso de tareas más intensas (como generación de reportes): Procesador básico + 4 GB de RAM (mínimo) para navegación fluida
- Software requerido
  - Navegador web actualizado (Chrome, Firefox, Safari, Edge)
  - Lector de PDF para visualizar exportaciones
- Restricciones de entorno
  - La solución debe ser una aplicación web, lo que implica que debe ejecutarse correctamente en distintos navegadores y estar optimizada para diferentes sistemas operativos
  - La base de datos debe soportar consultas complejas de historial, contratos, ventas y calendario
  - Integridad referencial entre artistas, grupos, álbumes, agencias y actividades
  - El sistema debe ser capaz de manejar múltiples roles de usuario: artistas, aprendices, managers, administradores y directores.

3. Funcionalidades del Producto

A continuación se muestra un diagrama de casos de uso presentando las funcionalidades del producto para los distintos actores.

1. Funcionalidades del Aprendiz

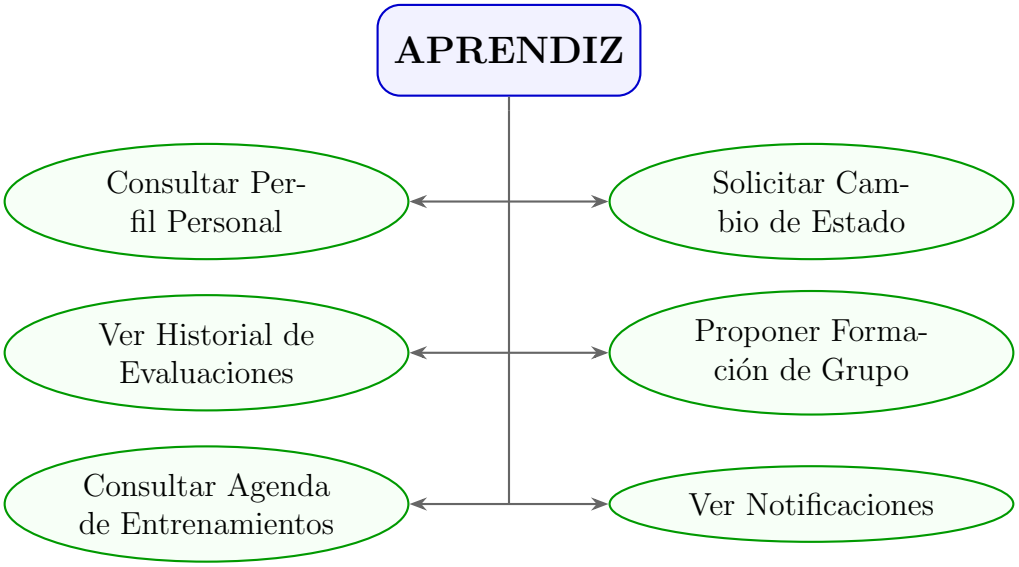
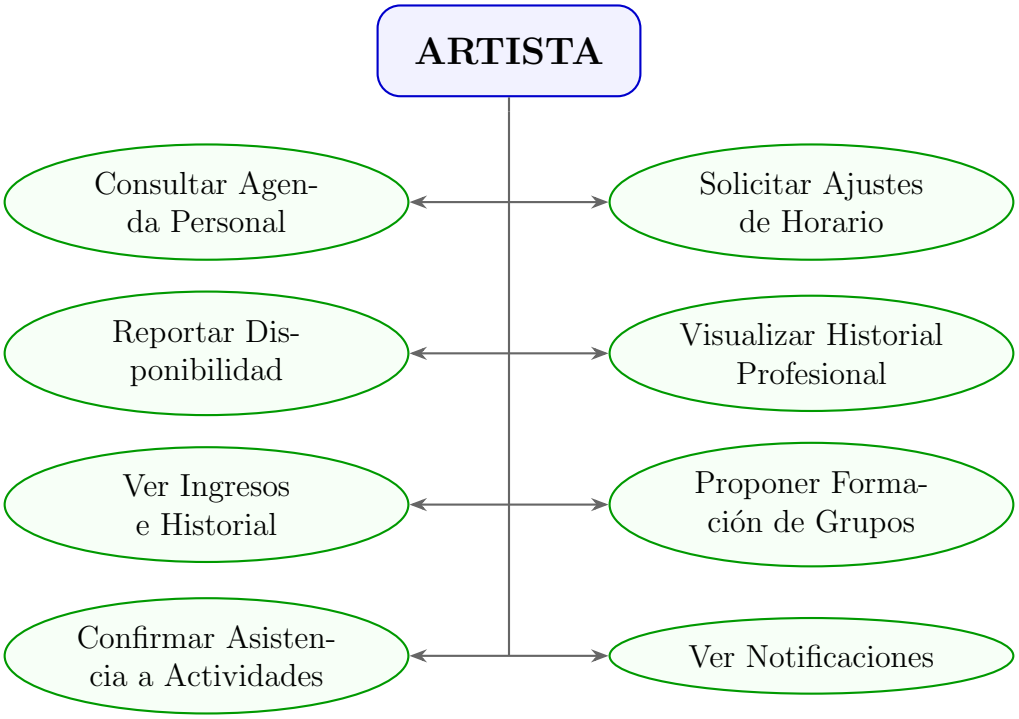


Figura 1: Diagrama de casos de uso - Aprendiz

2. Funcionalidades del Artista



3. Funcionalidades del Manager

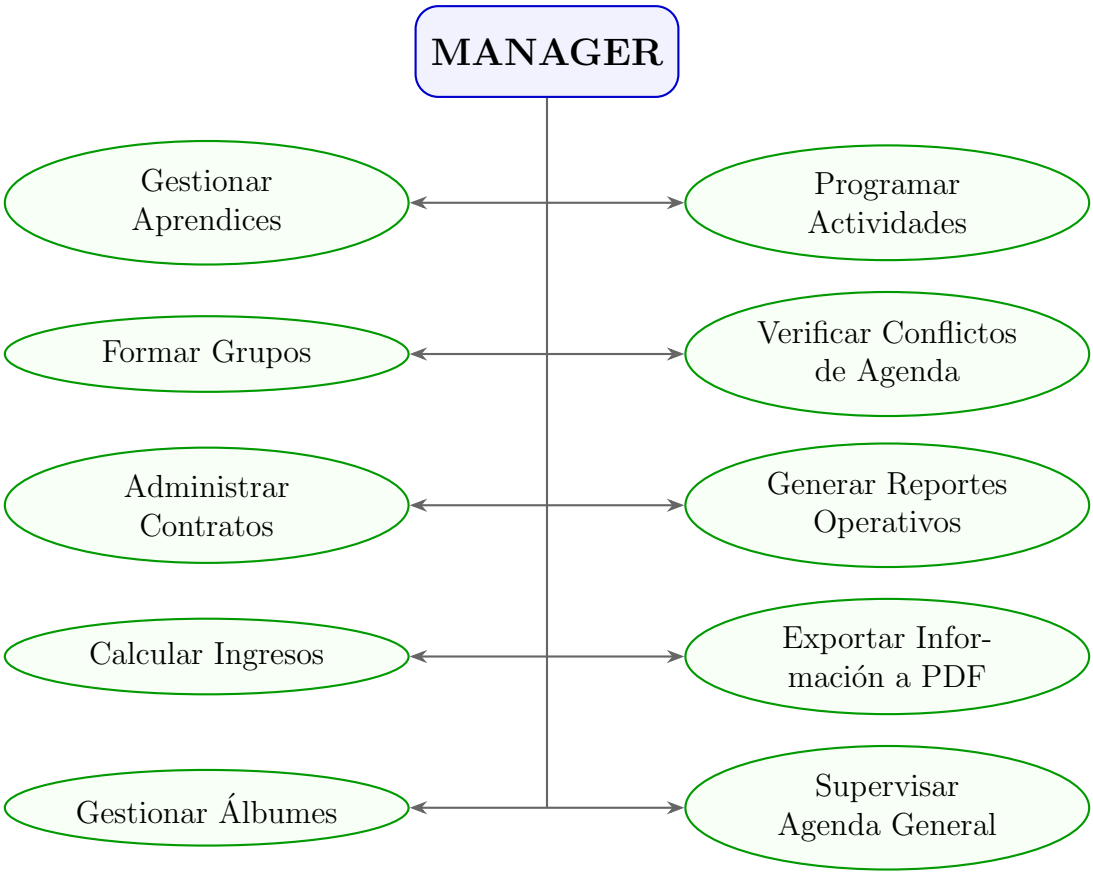
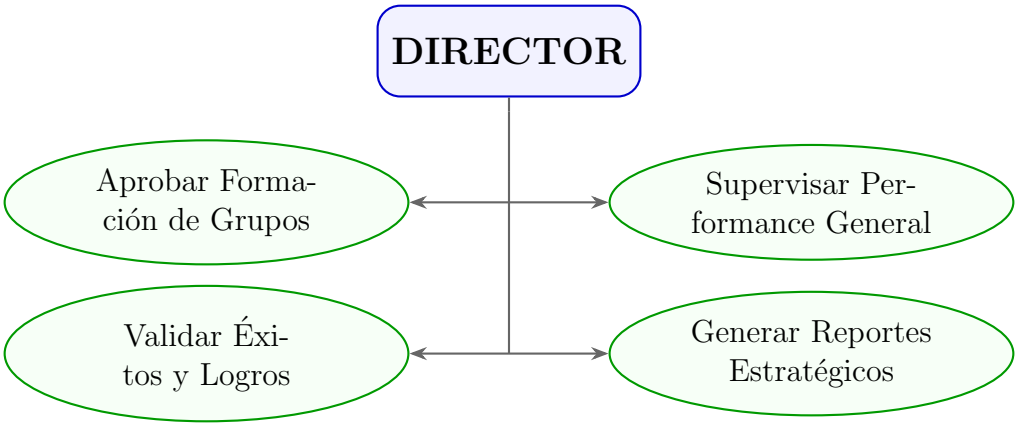


Figura 2: Diagrama de casos de uso - Artista y Manager

4. Funcionalidades del Director



5. Funcionalidades del Administrador

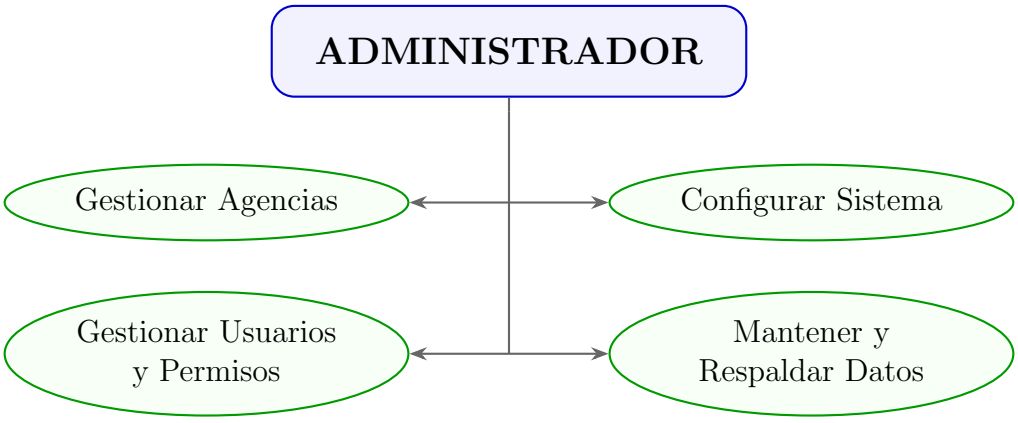


Figura 3: Diagrama de casos de uso - Director y Administrador

4. Enfoque Metodológico

4.1. ¿Qué metodología usamos?

Utilizamos una **metodología ágil adaptada** basada en Scrum, combinada con elementos del contexto académico.

4.2. ¿Cómo funciona nuestra metodología?

4.2.1. Principios Fundamentales

- **Desarrollo iterativo e incremental**: Dividimos el proyecto en hitos cortos
- **Entregas frecuentes**: Cada hito entrega software funcionando
- **Adaptación al cambio**: Flexibilidad para ajustar requisitos
- **Colaboración constante**: Comunicación diaria del equipo

4.2.2. Ciclo de Desarrollo

1. **Planificación**: Definir objetivos del hito
2. **Desarrollo**: Implementar funcionalidades priorizadas
3. **Revisión**: Demo del software funcionando
4. **Retrospectiva**: Mejoras para el siguiente ciclo

4.3. Características del Problema

4.3.1. Complejidad del Dominio

- **Sistema multi-rol**: 4 tipos de usuarios con necesidades diferentes
- **Procesos empresariales complejos**: Gestión de contratos, ingresos, agendas
- **Relaciones entrelazadas**: Artistas, grupos, agencias, actividades interconectadas
- **Requerimientos específicos**: Industria del K-pop con particularidades únicas

4.3.2. Retos Técnicos

- **Base de datos compleja**: Múltiples entidades y relaciones
- **Consultas avanzadas**: 7 consultas informacionales complejas
- **Interfaz multi-usuario**: Diferentes vistas según roles
- **Integración de módulos**: Coordinación entre componentes

4.4. Perfil del Cliente

4.4.1. Expectativas y Requisitos

- **Sistema integral**: Gestión completa del ciclo de vida de artistas
- **Usabilidad**: Interfaz intuitiva para usuarios no técnicos
- **Confiabilidad**: Manejo seguro de datos sensibles
- **Escalabilidad**: Capacidad de crecimiento futuro
- **Reportes**: Generación de informes estratégicos

4.5. Desafíos del Equipo de Desarrollo

4.5.1. Falta de Experiencia Técnica

- **Programación web**: Primer contacto con desarrollo web profesional
- **Tecnologías nuevas**: Express.js, Prisma, React
- **Frontend moderno**: Razor Pages, JavaScript, CSS avanzado
- **Base de datos compleja**: Diseño e implementación desde cero

4.5.2. Proceso de Aprendizaje

- **Curva de aprendizaje acelerada**: Aprender haciendo
- **Autodidactismo**: Investigación y estudio independiente
- **Colaboración**: Compartir conocimientos dentro del equipo
- **Práctica continua**: Aplicación inmediata de lo aprendido

4.5.3. Gestión de Proyecto en Equipo

Desafíos Iniciales

- **Proyecto de escala mediana**: Mayor que proyectos anteriores
- **Control de versiones**: Git con múltiples ramas de desarrollo
- **Coordinación**: 5 personas trabajando en paralelo
- **Integración**: Unir código de diferentes desarrolladores

Soluciones Implementadas

- **Flujo de trabajo Git**:
  - Rama principal (main) para releases estables
  - Rama develop para integración
  - Ramas feature para funcionalidades específicas
  - Pull requests y code reviews
- **División de tareas**: Asignación por módulos y roles
- **Comunicación diaria**: Reuniones de coordinación
- **Documentación compartida**: Manuales y guías internas

4.6. Aplicación en el Proyecto K-pop World

4.6.1. Hito 1: Implementado

- **Enfoque**: Desarrollar la infraestructura básica
- **Entregables**: Sistema de login, CRUD administrador, base de datos
- **Resultado**: Software básico funcionando en 4-5 semanas
- **Aprendizaje**: Primer contacto con Express.js y Prisma

4.6.2. Hito 2: En Proceso

- **Enfoque**: Implementar sistemas por roles
- **Entregables**: Módulos Artista, Manager, Director
- **Progreso**: Desarrollo simultáneo de múltiples funcionalidades
- **Desafío**: Coordinar trabajo paralelo en Git

4.6.3. Hito 3: Próximo

- **Enfoque**: Consultas complejas y testing avanzado
- **Entregables**: 7 consultas informacionales, manejo de errores
- **Reto**: Optimización de consultas SQL complejas

4.6.4. Hito Final: Planificado

- **Enfoque**: Pulido completo y seguridad
- **Entregables**: Interfaz amigable, medidas de seguridad, testing integral
- **Objetivo**: Producto profesional y listo para producción

4.7. Lecciones Aprendidas

4.7.1. Desarrollo Técnico

- **Aprendizaje progresivo**: De lo simple a lo complejo
- **Importancia de la planificación**: Diseño antes de codificar
- **Pruebas tempranas**: Detectar problemas pronto
- **Documentación**: Vital para el mantenimiento

4.7.2. Trabajo en Equipo

- **Comunicación**: Fundamental para la coordinación
- **División de trabajo**: Especialización por áreas
- **Control de versiones**: Esencial para proyectos grupales
- **Retroalimentación**: Mejora continua del proceso

5. Arquitectura

Clean Architecture es una metodología de diseño de software propuesta por Robert C. Martin (también conocido como Uncle Bob) que busca crear sistemas de software que sean fáciles de entender, desarrollar, probar y mantener. La idea central es separar las preocupaciones y responsabilidades del sistema en diferentes capas, asegurando que cada capa tenga un propósito claro y distinto. Clean Architecture organiza el sistema en varias capas concéntricas, cada una con responsabilidades distintas:

1. **Dominio**: Son los objetos del negocio y sus reglas, deben ser independientes de cualquier detalle técnico
2. **Aplicación**: Contienen la lógica de negocio específica de la aplicación y son independientes de las capas externas.
3. **Infraestructura**: Son adaptadores que convierten los datos de los casos de uso en una forma que pueda ser utilizada por la infraestructura o por la UI.
4. **Presentación**: Son los detalles técnicos y frameworks específicos que usa la aplicación, como bases de datos, interfaces de usuario y dispositivos externos.

5.1. Implementación de la arquitectura en nuestro proyecto

Se implementó Clean Architecture para el sistema K-Pop Groups Manager debido a las necesidades específicas del dominio de negocio. El sistema requiere gestionar entidades complejas como artistas, aprendices, grupos musicales, contratos, álbumes, etc, cada una con relaciones intrincadas y reglas de negocio específicas. Esta arquitectura permite mantener el núcleo del negocio completamente independiente de frameworks, bases de datos o interfaces de usuario, lo cual es fundamental para un sistema que necesita evolucionar constantemente según las dinámicas de la industria del entretenimiento.

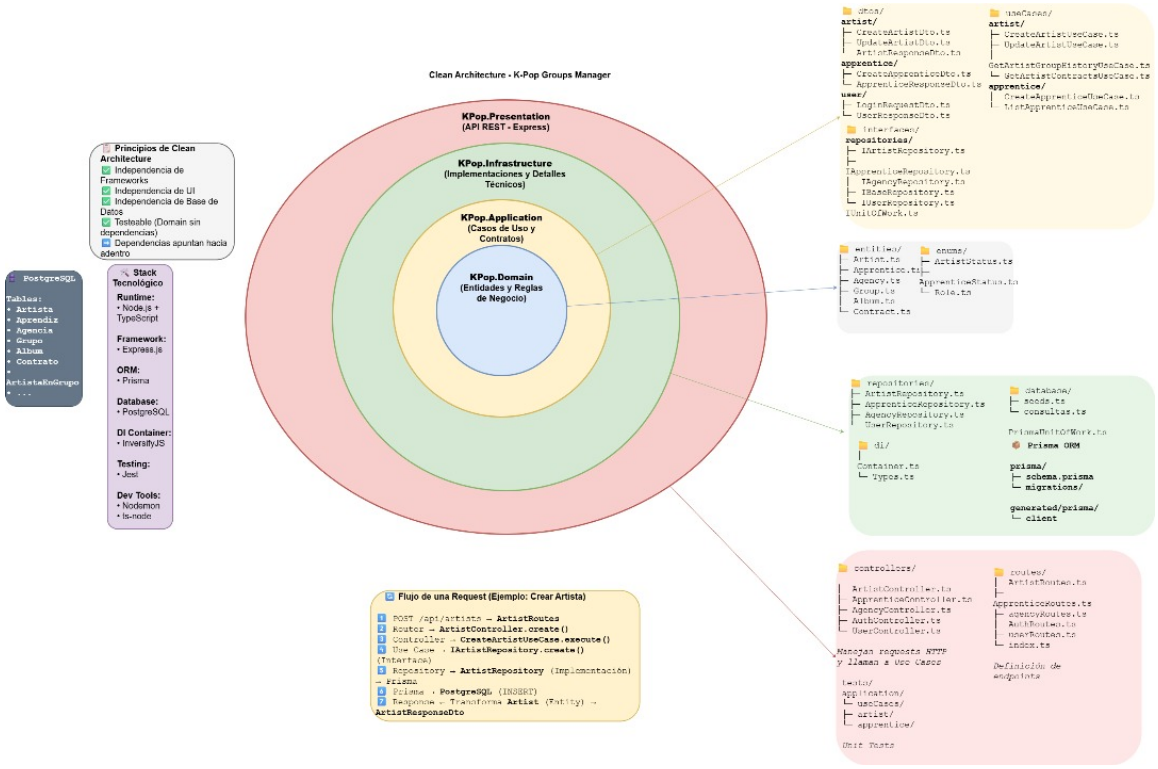


Figura 4: Diagrama de Clean Architecture implementado en el proyecto

La independencia de frameworks garantiza que las reglas fundamentales del negocio no estén acopladas a tecnologías específicas como Express.js o Prisma. Si en el futuro se necesita migrar a otro ORM o cambiar la interfaz de presentación, el dominio permanece intacto. La testabilidad es otro factor crítico, ya que la separación por capas con inversión de dependencias permite crear pruebas unitarias sin necesidad de una base de datos real, simplemente mockeando las interfaces de repositorios. Esto acelera el desarrollo y aumenta la confiabilidad del sistema.

6. Patrones de Diseño, visualización y de datos

6.1. Patrones Backend

6.1.1. Repository

El Repository Pattern se utiliza para abstraer completamente el acceso a datos del resto de la aplicación. Se definieron interfaces como IBaseRepository (CRUD Base), IArtistRepository, IApprenticeRepository y IAgencyRepository en la capa de Application que declaran contratos sin implementación. Las implementaciones concretas como ArtistRepository y ApprenticeRepository residen en la capa de Infrastructure y contienen toda la lógica específica de Prisma ORM. Si en el futuro se decide migrar de Prisma a TypeORM o Sequelize, solo se reimplementan los repositorios manteniendo las mismas interfaces y el resto del código permanece sin cambios.

Beneficios específicos del proyecto:

- Mantener integridad en modelos complejos (artistas en múltiples grupos, contratos históricos)

- Facilitar pruebas unitarias
- Desacoplar la lógica del motor de base de datos

6.1.2. Patrón Unit of Work

El Unit of Work Pattern gestiona transacciones de base de datos garantizando atomicidad en operaciones que involucran múltiples entidades. Se implementó PrismaUnitOfWork que envuelve el cliente de Prisma y expone métodos beginTransaction, commit y rollback. Los casos de uso que modifican datos reciben IUnitOfWork mediante inyección de dependencias y lo utilizan para coordinar transacciones.

CreateAlbumUseCase demuestra este patrón iniciando una transacción antes de ejecutar validaciones y operaciones de escritura. Si cualquier paso falla por validación de negocio o error de base de datos, se ejecuta rollback automático dejando la base de datos en estado consistente. Si todas las operaciones completan exitosamente, commit persiste los cambios. Este patrón elimina la posibilidad de estados inconsistentes donde por ejemplo se crea un usuario pero falla al crear su canción asociada.

6.1.3. Patrón Dependency Injection

Dependency Injection mediante InversifyJS desacopla completamente las clases de sus dependencias permitiendo inyectarlas desde el exterior. Se configuró un contenedor en Container.ts que bindea símbolos definidos en Types.ts a implementaciones concretas. Los controladores, casos de uso y repositorios declaran sus dependencias mediante decoradores @inject en constructores y el contenedor las resuelve automáticamente en tiempo de ejecución.

UserController declara dependencias de cuatro casos de uso mediante @inject (Types.CreateUserUseCase), @inject(Types.UpdateUserUseCase), etc. Cuando el contenedor instancia el controlador, automáticamente resuelve estas dependencias buscando los bindings configurados e inyectando las instancias apropiadas. Esto elimina el acoplamiento directo entre clases y permite cambiar implementaciones modificando solo la configuración del contenedor.

6.1.4. Patrón Data Transfer Object (DTO)

El DTO Pattern separa la representación de datos entre capas evitando que entidades de dominio se expongan directamente en APIs. Se crearon DTOs como CreateArtistDto, UpdateArtistDto y ArtistResponseDto organizados por entidad. Los DTOs de creación contienen solo campos necesarios para construir entidades mientras que los DTOs de respuesta incluyen campos relevantes para clientes HTTP transformando datos de dominio a formato serializable.

CreateArtistDto implementa un método estático Create que valida datos del request antes de construir el DTO. Verifica que campos requeridos como ArtistName, DebutDate y Status estén presentes y que Status sea válido según la enumeración ArtistStatus. Esta validación centralizada en el DTO evita código repetitivo en controladores y garantiza que solo DTOs válidos lleguen a casos de uso. Si la validación falla, se lanza excepción descriptiva que el controlador captura y convierte en respuesta HTTP apropiada.

6.2. Patrones Frontend

6.2.1. Patrón de Componentes (Component Pattern)

React organiza la interfaz en unidades pequeñas y reutilizables llamadas componentes. Cada componente combina HTML, lógica y estilos de una sección específica de la interfaz.

Esto sigue el principio de divide y vencerás: separar la UI en piezas coherentes y fácilmente manejables.

Aplicación en el proyecto: Como el sistema tiene múltiples vistas complejas—tablas de artistas, historial profesional, contratos, calendarios de actividades, reportes de ingresos—cada una se implementa como un componente aislado.

Por ejemplo:

```
<Agency />
<Apprentice />
<Profile />
<Datatable />
```

Por qué es útil en este proyecto:

- Evita duplicar código cuando la misma tabla o tarjeta aparece en varias páginas
- Facilita modificar una parte de la interfaz sin afectar las demás
- Promueve consistencia visual (todos los listados pueden compartir columnas y estilos)

6.2.2. Patrón Observador (Observer Pattern)

React implementa este patrón detrás de escena:cuando cambia un estado, los componentes suscritos se re-renderizan automáticamente.

Aplicación en el proyecto: Si la agenda de un artista cambia (por ejemplo, el manager aprueba un ajuste) React detecta el cambio, re-renderiza el calendario y muestra el nuevo estado sin refrescar la página.

Utilidad: Los usuarios del sistema (artistas y managers) necesitan ver los cambios sin recargar manualmente.

6.3. Patrones ORM

6.3.1. DAO (Data Access Object)

Este patrón encapsula todo el acceso a la base de datos para no trabajar con SQL directo. Prisma implementa este patrón por defecto.

En lugar de escribir SQL como:

```
SELECT * FROM user WHERE id = ...
```

Usas:

```
// Ejemplo tomado del backend (UserRepository.getUsers)
const prismaUsers = await prisma.user.findMany();

async getUsers(): Promise<User[]> {
 const prismaUsers = await this.db.user.findMany();
 return prismaUsers
}
```

Cada entidad importante del dominio (Artista, Grupo, Contrato, Álbum, Actividad, Ingreso) se accede mediante Prisma, que actúa como capa de acceso a datos.

6.3.2. Patrón Data Mapper

Transforma datos de la base de datos (tablas, filas, relaciones) en objetos de JavaScript fácilmente utilizables. Prisma hace:

- el JOIN,
- arma el objeto complejo,
- devuelve una estructura lista para usar en el backend

¿Por qué es clave en el sistema? El dominio tiene muchas relaciones: los álbumes tienen solistas, los grupos tienen artistas de distintas agencias, las actividades combinan agendas grupales e individuales. Prisma simplifica todo este mapeo.

7. Modelo de datos

El modelo de datos del sistema K-Pop Groups Manager está diseñado para manejar la complejidad de las relaciones en la industria del entretenimiento coreano.

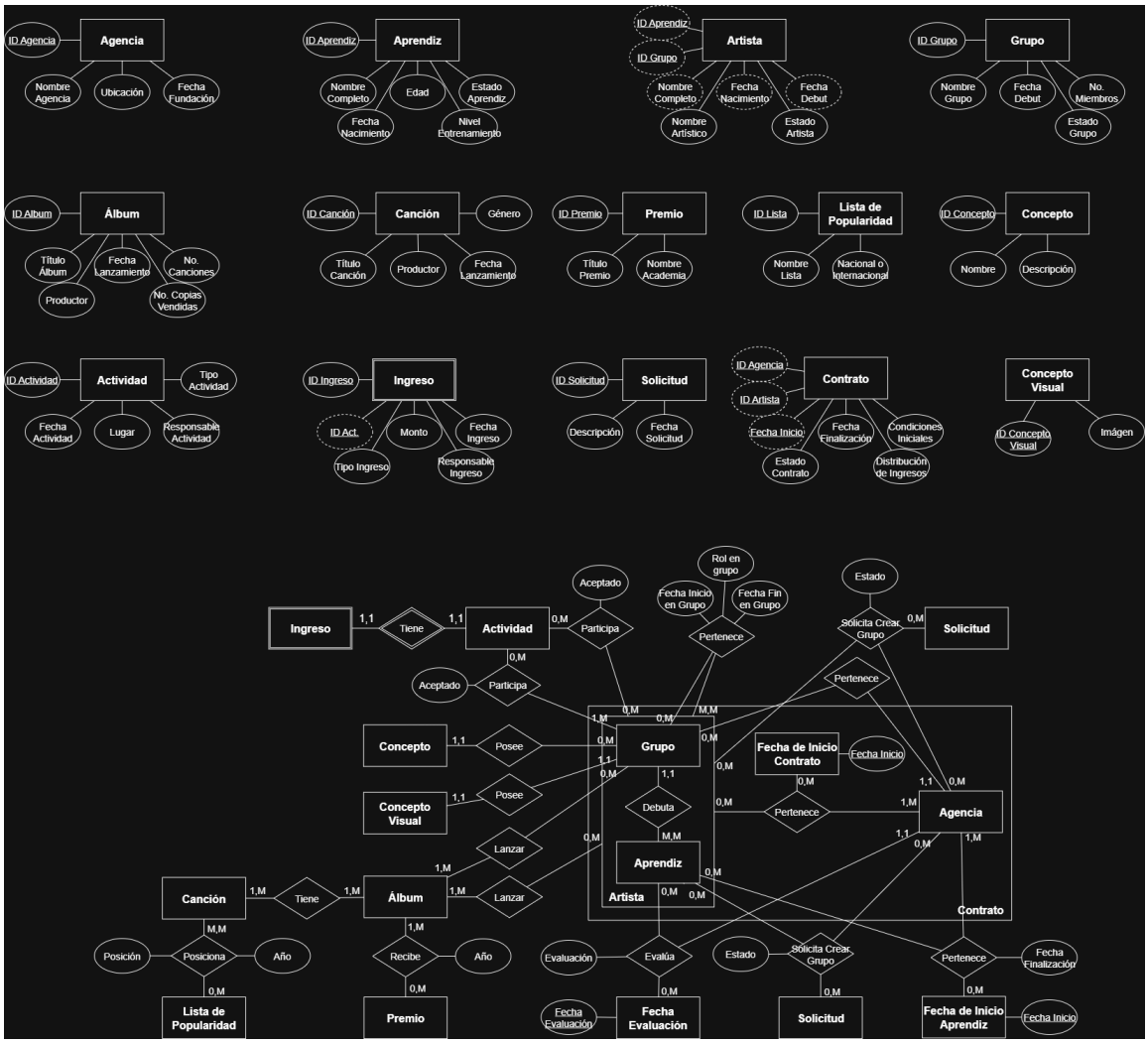


Figura 5: Modelo Entidad-Relación del sistema K-Pop Groups Manager

7.1. Entidades Principales

- **Agencia**: Representa las empresas de entretenimiento que gestionan artistas y grupos
- **Artista**: Personas individuales que pueden ser aprendices o artistas activos
- **Grupo**: Conjuntos musicales formados por múltiples artistas
- **Contrato**: Acuerdos legales entre artistas y agencias
- **Álbum**: Producciones musicales lanzadas por grupos o artistas individuales
- **Actividad**: Eventos y compromisos en la agenda de artistas y grupos
- **Ingreso**: Registros financieros generados por las actividades

7.2. Relaciones Clave

- Relación muchos a muchos entre Artista y Grupo con temporalidad (historial de grupos)
- Relación uno a muchos entre Agencia y Artista/Contrato
- Relación uno a muchos entre Grupo y Álbum
- Relaciones polimórficas entre Actividad y Artista/Grupo
- Herencia de estados para Artista (Aprendiz/Activo/Inactivo) y Grupo (Activo/En pausa/Disuelto)

7.3. Características del Modelo

- **Temporalidad**: Registro histórico de cambios de grupo, agencia y estado
- **Integridad referencial**: Restricciones para prevenir estados inconsistentes
- **Escalabilidad**: Diseño normalizado para soportar crecimiento futuro
- **Flexibilidad**: Capacidad de adaptarse a cambios en la industria

8. Conclusión

Nuestra metodología ágil adaptada ha demostrado ser efectiva para:

- Manejar la complejidad del sistema multi-rol
- Entregar valor demostrable en cada hito académico
- Mantener la calidad mediante testing progresivo
- Adaptarse a los requisitos específicos del contexto académico
- Gestionar las limitaciones de experiencia técnica inicial
- Coordinar eficientemente un equipo de 5 desarrolladores

Este enfoque nos ha permitido transformar nuestra falta de experiencia en una oportunidad de aprendizaje, entregando un sistema funcional mientras desarrollábamos habilidades profesionales en desarrollo web y trabajo en equipo.

La implementación de Clean Architecture junto con patrones de diseño específicos para frontend y backend ha resultado en un sistema robusto, mantenible y escalable que satisface los complejos requisitos de la industria del K-pop. El modelo de datos cuidadosamente diseñado permite gestionar las intrincadas relaciones temporales y estructurales propias de este dominio.