Informe Técnico BD grupo4

December 2025

1. Generación automática de éxamenes

2. Integrantes:

- Kelen Alfaro García C311
- Darío Francisco Alfonso Urrutia C311
- Adriana Boué García C311
- Carlos Alejandro Mazorra Matos C311
- Darián Santamarina Hernández C311

3. Diccionario de datos:

Cuadro 1: Tabla `User`

Campo	**Tipo**	**Descripción**
id_us	INT (PK)	Identificador único del usuario.
name	STRING	Nombre completo del usuario.
password	STRING	Contraseña hash para autenticación.
account	STRING (ÚNICO)	Nombre de usuario o cuenta única para login.
age	INT	Edad del usuario.
course	STRING	Curso del usuario.
role	ENUM(Role)	Rol del usuario: STUDENT, TEACHER o ADMIN.
isActive	BOOLEAN	Indica si la cuenta está activa.

Cuadro 2: Tabla `Student`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único del estudiante.
user	INT (FK)	Referencia a `User.id_us`.

Cuadro 3: Tabla `Teacher`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único del profesor.
user	INT (FK)	Referencia a `User.id_us`.
isHeadTeacher	BOOLEAN	Indica si es jefe de departamento.
specialty	STRING	Especialidad o asignatura principal del profesor.

Cuadro 4: Tabla `Head_Teacher`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único del jefe de departamento.
teacher	INT (FK)	Referencia al profesor correspondiente (`Teacher.id`).

Cuadro 5: Tabla `Subject`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único de la asignatura.
name	STRING	Nombre de la asignatura.
head_teacher_id	INT (FK)	Referencia al jefe de departamento (`Head_Teacher.id`).
program	STRING	Programa académico de la asignatura.

Cuadro 6: Tabla `Exam`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único del examen.
name	STRING	Nombre del examen.
status	STRING	Estado del examen (pendiente, aprobado, rechazado, asignado.).
difficulty	STRING	Nivel de dificultad del examen.
subject_id	INT (FK)	Referencia a la asignatura (`Subject.id`).
teacher_id	INT (FK)	Referencia al profesor creador del examen (`Teacher.id`).
parameters_id	INT (FK)	Referencia a los parámetros del examen (`Parameters.id`).
head_teacher_id	INT (FK)	Referencia al jefe de departamento que supervisa el examen (`Head_Teacher.id`).

Cuadro 7: Tabla `Parameters`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único de los parámetros.
proportion	STRING	Proporción de tipos de preguntas en el examen.
amount_quest	STRING	Cantidad total de preguntas en el examen.
quest_topics	STRING	Distribución de preguntas por tema.

Cuadro 8: Tabla `Question`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único de la pregunta.
question_text	STRING	Texto de la pregunta.
difficulty	STRING	Nivel de dificultad.
answer	STRING	Respuesta correcta.
type	STRING	Tipo de pregunta (opción múltiple, verdadero o falso, argumentación).
score	FLOAT	Valor de la pregunta en puntos.
subject_id	INT (FK)	Referencia a la asignatura (`Subject.id`).
sub_topic_id	INT (FK)	Referencia al subtema (`Sub_Topic.id`).
topic_id	INT (FK)	Referencia al tema principal (`Topic.id`).
teacher_id	INT (FK)	Referencia al profesor creador (`Teacher.id`).

Cuadro 9: Tabla `Topic`

Campo	**Tipo**	**Descripción**
id	INT (PK)	Identificador único del tema.
name	STRING	Nombre del tema.

Cuadro 10: Tabla `Sub_Topic`

Campo	**Tipo**	**Descripción**
id	INT	Identificador único del subtema (combinado con topic_id como PK).
name	STRING	Nombre del subtema.
topic_id	INT (FK)	Referencia al tema principal (`Topic.id`).
PK	(id, topic_id)	Clave primaria compuesta.

Cuadro 11: Tabla `Exam_Student`

Campo	**Tipo**	**Descripción**
score	FLOAT	Puntaje obtenido por el estudiante en el examen.
exam_id	INT (FK)	Referencia al examen (`Exam.id`).
student_id	INT (FK)	Referencia al estudiante (`Student.id`).
teacher_id	INT (FK)	Referencia al profesor que asignó el examen (`Teacher.id`).
PK	(exam_id, student_id)	Clave primaria compuesta.

Cuadro 12: Tabla `Approved_Exam`

Campo	**Tipo**	**Descripción**
exam_id	INT (FK)	Referencia al examen aprobado (`Exam.id`).
head_teacher_id	INT (FK)	Referencia al jefe de departamento que aprueba el examen (`Head_Teacher.id`).
guidelines	STRING	Observaciones o pautas para la aprobación del examen.
date	DATETIME	Fecha de revisión del examen.
seen	BOOLEAN	Indica, en caso de ser rechazado por el jefe de asignatura, si el profesor que creó el examen ya hizo los cambios indicados .
PK	(date, exam_id, head_teacher_id)	Clave primaria compuesta.

Cuadro 13: Tabla `Exam_Question`

Campo	**Tipo**	**Descripción**
exam_id	INT (FK)	Referencia al examen (`Exam.id`).
question_id	INT (FK)	Referencia a la pregunta (`Question.id`).
PK	(exam_id, question_id)	Clave primaria compuesta.

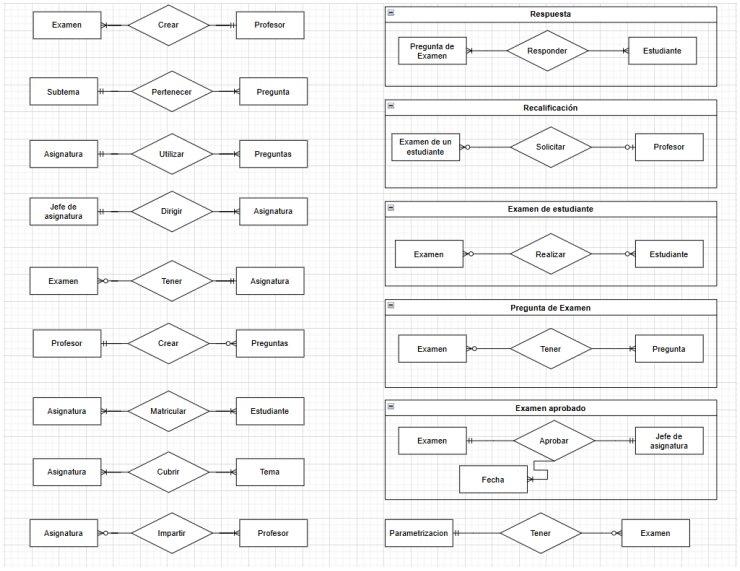
Cuadro 14: Tabla `Answer`

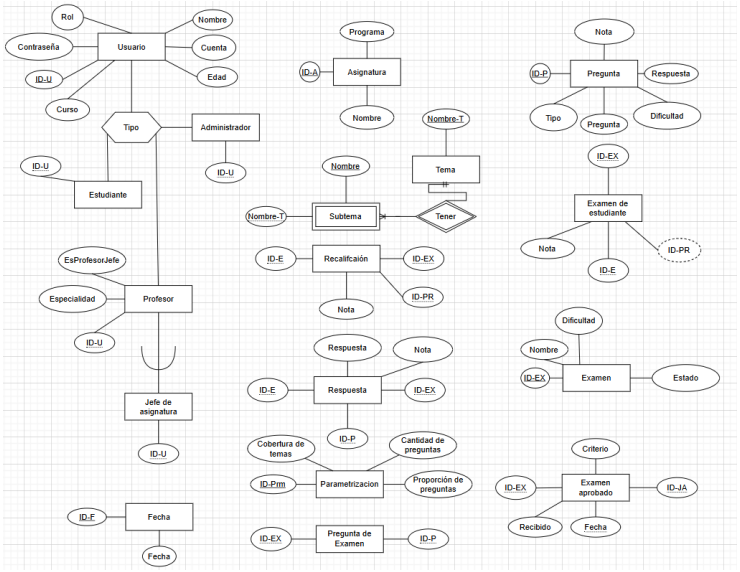
Campo	**Tipo**	**Descripción**
exam_id	INT (FK)	Referencia al examen.
question_id	INT (FK)	Referencia a la pregunta.
student_id	INT (FK)	Referencia al estudiante que respondió.
answer_text	STRING	Texto de la respuesta proporcionada por el estudiante.
score	FLOAT	Puntaje obtenido por la respuesta.
PK	(exam_id, question_id, student_id)	Clave primaria compuesta.

Cuadro 15: Tabla `Reevaluation`

Campo	**Tipo**	**Descripción**
exam_id	INT (FK)	Referencia al examen que se solicita recalificación.
student_id	INT (FK)	Referencia al estudiante que solicita recalificación.
teacher_id	INT (FK)	Referencia al profesor que evalúa la recalificación.
score	FLOAT?	Puntaje revisado en la recalificación (opcional).
PK	(exam_id, student_id, teacher_id)	Clave primaria compuesta.

4. Esquema MERX:





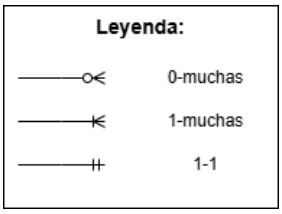


Figura 1: Leyenda

5. Diseño de la aplicación:

En nuestro proyecto, la arquitectura de módulos sigue el patrón modular de NestJS, donde cada módulo encapsula su lógica, servicios, controladores y DTOs. Esto permite mantener el código organizado, modular, y escalable. A continuación, se describe la estructura de los módulos:

5.1. Estructura General de un Módulo

```
mi-modulo/
|
|- dto/
|  |- create-mi-modulo.dto.ts
|  |- update-mi-modulo.dto.ts
|
|- mi-modulo.controller.ts
|- mi-modulo.service.ts
|- mi-modulo.module.ts
\- mi-modulo.controller.spec.ts
\- mi-modulo.service.spec.ts
```

Descripción de los Archivos

- **dto/** Contiene los Data Transfer Objects (DTO), que definen la estructura de los datos que entran y salen de la aplicación.
  - `create-mi-modulo.dto.ts`: Define los campos y validaciones necesarias para crear una nueva entidad del módulo.
  - `update-mi-modulo.dto.ts`: Define los campos opcionales o editables para actualizar una entidad existente.
- **mi-modulo.controller.ts** El controlador expone los endpoints del módulo. Su función es recibir las solicitudes HTTP, delegarlas al servicio correspondiente y devolver la respuesta al cliente. En este archivo se manejan rutas como `GET`, `POST`, `PUT` y `DELETE`.

- **mi-modulo.service.ts** El servicio contiene la lógica de negocio del módulo. Es responsable de interactuar con la base de datos (a través de PrismaService), aplicar reglas de negocio y procesar los datos antes de enviarlos al controlador.
- **mi-modulo.module.ts** Este archivo define el módulo y registra sus proveedores y controladores. Es el punto de integración del módulo dentro de la aplicación NestJS. Por ejemplo:

```
@Module({
  controllers: [MiModuloController],
  providers: [MiModuloService],
})
export class MiModuloModule {}
```

- **mi-modulo.spec.ts** Archivo de pruebas unitarias (test) que verifica el correcto funcionamiento del módulo. Se suelen usar frameworks como Jest para validar servicios y controladores, asegurando que la lógica de negocio y las rutas respondan correctamente a distintos escenarios.

5.2. Resumen de la Organización

Cada módulo del proyecto sigue el patrón:

- DTOs separados por operación (`create` y `update`) para un control claro de validación.
- Controlador para exponer rutas HTTP.
- Servicio para encapsular la lógica de negocio.
- Archivo de pruebas `.spec.ts` para garantizar la calidad y confiabilidad del código.
- Archivo `module.ts` para registrar e integrar el módulo dentro de la aplicación.

Esta organización modular asegura mantenibilidad, escalabilidad y un fácil entendimiento del sistema para nuevos desarrolladores.

6. Solución propuesta:

6.1. Arquitectura empleada:

La estructura arquitectónica del proyecto se organiza bajo un enfoque de dos componentes principales: Frontend y Backend. El Backend sigue una arquitectura monolítica modular, donde cada módulo aplica un patrón de N-Capas. Esta

organización garantiza mantenibilidad, escalabilidad y una clara separación de responsabilidades.

La aplicación se compone de:

- Frontend (Interfaz de Usuario)
- Backend (Aplicación Servidora con arquitectura modular y N-Capas)

El Frontend maneja la interacción directa con el usuario, mientras que el Backend gestiona la lógica de negocio, los datos y expone servicios a través de una API.

Frontend

El Frontend constituye la capa visual del sistema, permitiendo la interacción del usuario con la aplicación. Envía solicitudes al Backend y muestra las respuestas recibidas.

Funciones principales

- Renderizar interfaces y componentes visuales.
- Gestionar la navegación del usuario.
- Enviar peticiones HTTP al Backend.
- Presentar datos provenientes del servidor.
- Manejar estados y validaciones a nivel de cliente.

Backend

El Backend sigue un diseño monolítico modular, donde cada módulo representa una funcionalidad del dominio (usuarios, autenticación, publicaciones, etc.).

Características principales

- Independencia lógica entre módulos.
- Organización clara de funcionalidades.
- Reutilización de componentes comunes.
- Centralización en un único despliegue monolítico.
- Cada módulo aplica internamente el patrón N-Capas.

Arquitectura interna de cada módulo

Cada módulo se estructura en tres capas para asegurar separación de responsabilidades y escalabilidad:

Capa de Presentación (Controller)

Esta capa recibe solicitudes HTTP y actúa como punto de entrada.
Responsabilidades:

- Manejo de rutas y endpoints.
- Recepción y validación inicial de datos mediante DTOs.
- Delegación de operaciones a la capa de aplicación.
- Envío de respuestas al cliente.

Limitaciones:

- No contiene lógica de negocio.
- No realiza consultas a la base de datos.
- Su responsabilidad es comunicacional.

Capa de Aplicación (Service)

Contiene la lógica de negocio y coordina la interacción entre presentación y datos.
Responsabilidades:

- Implementación de reglas de negocio.
- Coordinación de casos de uso.
- Validaciones del dominio.
- Interacción con la capa de acceso a datos.

Limitaciones:

- No gestiona rutas del sistema.
- No ejecuta consultas directas a la base de datos.

Capa de Acceso a Datos (Prisma Service)

Encargada de la interacción con la base de datos mediante Prisma.
Responsabilidades:

- Operaciones CRUD.
- Abstracción de consultas a la base de datos.
- Provisión de métodos estructurados para persistencia.
- Comunicación directa con el motor de base de datos.

Limitaciones:

- No contiene lógica de negocio.
- No gestiona solicitudes HTTP.

Puede implementarse de forma centralizada (`PrismaService`) o mediante repositorios especializados por módulo.

Relación entre capas

El flujo de interacción se mantiene descendente y controlado:

$$\text{Frontend} \rightarrow \text{Controller} \rightarrow \text{Service} \rightarrow \text{PrismaService} \rightarrow \text{Base de Datos}$$

$$\text{Base de Datos} \rightarrow \text{PrismaService} \rightarrow \text{Service} \rightarrow \text{Controller} \rightarrow \text{Frontend}$$

Este flujo garantiza responsabilidades claras y evita accesos indebidos entre capas.

La arquitectura adoptada proporciona un diseño ordenado, escalable y mantenible. La combinación de un Frontend independiente, un Backend modular y el patrón N-Capas asegura separación de responsabilidades, facilita el trabajo en equipo y permite la extensibilidad futura del sistema.

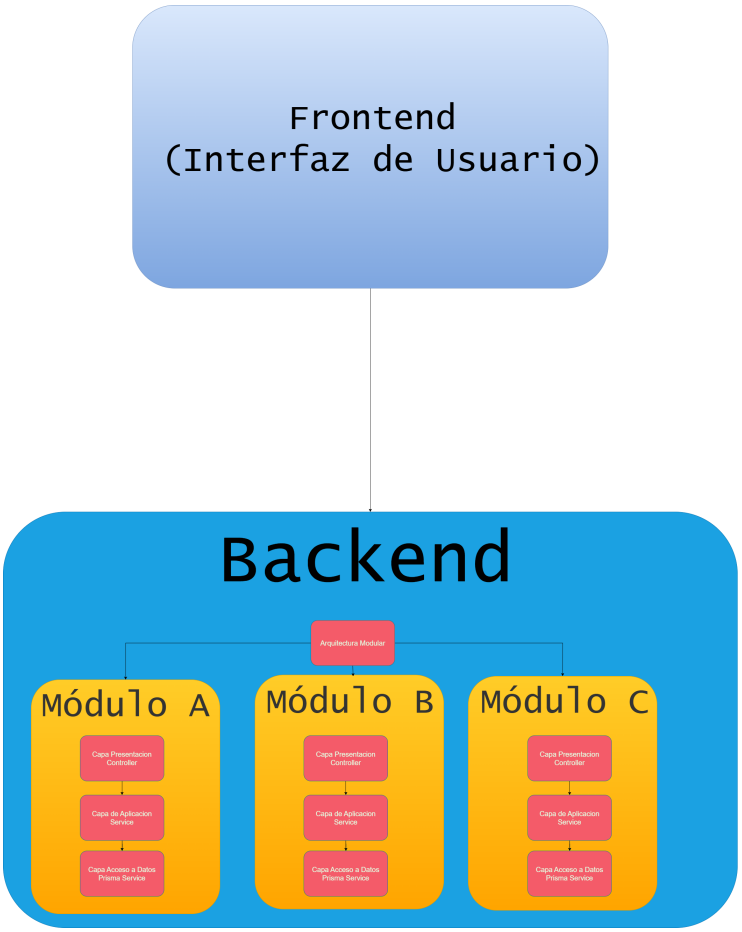


Figura 2: Diseño de la arquitectura empleada

6.2. Descripción de los módulos:

Por cada modelo definido en el esquema de Prisma, se generó un módulo independiente. Cada módulo contiene los siguientes componentes:

- **DTO** (Data Transfer Object): Define la estructura de los datos que se transfieren entre la capa de presentación y la capa de negocio, garantizando consistencia y validación.
- **Service**: Implementa la lógica de negocio específica del modelo y gestiona la interacción con la base de datos mediante Prisma.
- **Controller**: Expone los endpoints HTTP del módulo, actuando como punto de entrada para las solicitudes de los clientes.

- **Module**: Agrupa y configura los componentes del módulo, declarando proveedores, controladores y dependencias.

Esta estrategia asegura que cada entidad del esquema tenga su propio módulo autónomo, facilitando la mantenibilidad y permitiendo que cada módulo pueda evolucionar de manera independiente sin afectar otros componentes.

Además Todos los reportes del sistema se encuentran agrupados dentro de un módulo de reportes, siguiendo la misma estructura modular (DTO, Service, Controller, Module). Este módulo centraliza la generación de estadísticas y resultados de los reportes, garantizando consistencia y reutilización de la lógica.

Se utiliza un Prisma Service centralizado que administra la conexión con la base de datos y proporciona un cliente Prisma reutilizable en todos los módulos. Esto optimiza la comunicación con la base de datos y permite una gestión centralizada de las transacciones. En la arquitectura de NestJS, existen componentes centrales que permiten inicializar y gestionar la aplicación, además de servir como punto de integración de los módulos generados. En este proyecto se destacan los siguientes:

AppModule

El AppModule es el módulo raíz de la aplicación y actúa como el punto central de configuración de NestJS. Sus funciones principales son:

- Importar todos los módulos funcionales generados a partir del esquema de Prisma, así como el módulo de reportes.
- Configurar la inyección de dependencias entre los distintos módulos y servicios.
- Servir como estructura organizativa que permite que NestJS registre todos los controladores y servicios disponibles en la aplicación.

AppService

El AppService es un servicio general de la aplicación que puede contener lógica de negocio transversal o funciones compartidas que no pertenecen a ningún módulo específico. Sus funciones principales son:

- Proveer métodos reutilizables que puedan ser invocados por los controladores.
- Servir como ejemplo inicial de la estructura de servicios de NestJS para la aplicación.

AppController

El AppController es el controlador principal de la aplicación y sirve como punto de entrada de solicitudes HTTP generales o de prueba. Sus funciones incluyen:

- Exponer endpoints básicos que permiten verificar que la aplicación está funcionando correctamente.
- Delegar la ejecución de la lógica de negocio al AppService u otros servicios inyectados.

main.ts

El archivo main.ts es el punto de arranque de la aplicación NestJS y cumple las siguientes funciones:

- Inicializa el `NestFactory`, que crea la instancia principal de la aplicación.
- Carga el **AppModule**, registrando todos los módulos, controladores y servicios.
- Configura aspectos globales de la aplicación, como middleware, pipes de validación, filtros de excepciones y cualquier otra funcionalidad transversal.
- Lanza el servidor HTTP y pone la aplicación a disposición de los clientes en el puerto configurado.

En resumen, main.ts es el archivo que inicia la aplicación, mientras que AppModule, AppService y AppController conforman la base estructural sobre la cual se integran todos los módulos específicos generados a partir del esquema.

6.3. Gestión de Base de Datos y Patrones de Diseño

La aplicación utiliza **PostgreSQL** como sistema de gestión de bases de datos relacional y **Prisma** como ORM para Node.js y TypeScript. Prisma permite interactuar con la base de datos de manera eficiente, segura y consistente, proporcionando una capa de abstracción que separa la lógica de negocio de la lógica de persistencia.

Data Mapper

Prisma implementa internamente el patrón de diseño **Data Mapper**. Este patrón tiene como objetivo principal separar la lógica de negocio de la persistencia de datos. Para ello, Prisma Client actúa como un componente intermediario que transforma objetos de dominio en consultas SQL y convierte los resultados de la base de datos en objetos TypeScript.

Un ejemplo de su funcionamiento es el siguiente:

```
async create(data: CreateStudentDto) {
    return this.prisma.student.create({ data });
}
```

En este fragmento, el objeto `CreateStudentDto` se serializa, se convierte en una instrucción SQL que se ejecuta en PostgreSQL y se devuelve como objeto TypeScript. La lógica del dominio no necesita conocer los detalles de SQL ni de la base de datos, demostrando una implementación limpia del patrón Data Mapper.

Factory Method

Prisma utiliza el patrón **Factory Method** durante la generación del Prisma Client. Al ejecutar `npx prisma generate`, Prisma crea automáticamente clases y métodos para interactuar con cada modelo, como `PrismaClient`, `prisma.user`, `prisma.student`, etc.

Estos métodos actúan como fábricas que producen objetos configurados para cada modelo, ocultando la complejidad de instanciación al desarrollador y estandarizando la creación de objetos de acceso a datos.

Singleton

El patrón **Singleton** se utiliza para garantizar que exista una única instancia compartida de PrismaClient en toda la aplicación. En nuestro proyecto, esto se implementa mediante el `PrismaService` de NestJS, que extiende PrismaClient, se registra como *provider* global y se inyecta en todos los servicios que requieren acceso a la base de datos.

De esta manera:

- Se evita la apertura de múltiples conexiones.
- Se mejora el rendimiento.
- Se asegura la consistencia de los datos en toda la aplicación.

Gracias a la combinación de **Prisma como ORM**, el patrón **Data Mapper**, el **Factory Method** y el **Singleton**, la aplicación logra:

- Separación clara entre lógica de negocio y persistencia de datos.
- Creación estandarizada y segura de objetos de acceso a datos.
- Eficiencia y consistencia en el manejo de la base de datos PostgreSQL.

6.4. Patrones de visualización en el frontend

En el frontend del proyecto se aplican diversos patrones de diseño y arquitectura para la gestión de la interfaz y la interacción con el usuario:

- **SPA (Single Page Application):** La aplicación se carga una sola vez y el contenido se actualiza dinámicamente mediante peticiones asíncronas a la API, sin recargar completamente la página. Esto permite separación clara entre frontend y backend, navegación fluida y renderizado del lado del cliente. Por ejemplo, en React:

```
"use client";
export default function Dashboard() {
  const [subjects, setSubjects] = useState([]);
  useEffect(() => {
    fetch("/api/subjects").then(r => r.json()).then(setSubjects);
  }, []);
  return <ul>{subjects.map(s => <li key={s.id}>{s.name}</li>)}</ul>;
}
```

- **MVVM/MVI (View-Model-ViewModel):** React utiliza hooks personalizados que actúan como ViewModel, gestionando estado, lógica y comunicación con APIs, mientras que los componentes representan la Vista. El Model se corresponde con los datos obtenidos del backend. Por ejemplo:

```
// ViewModel
function useSubjects() {
  const [subjects, setSubjects] = useState([]);
  const load = async () => { setSubjects(await fetch("/api/subjects").then(r => r.json(
  return { subjects, load };
}
// View
export default function SubjectsPage() {
  const { subjects, load } = useSubjects();
  useEffect(() => { load(); }, []);
  return <ul>{subjects.map(s => <li key={s.id}>{s.name}</li>)}</ul>;
}
```

- **Arquitectura basada en componentes:** Se divide la interfaz en componentes independientes y reutilizables, encapsulando lógica, estilo y funcionalidad. Esto promueve modularidad y reutilización:

```
import { Button } from "@/components/ui/button";
export function SaveButton({ onClick }) {
  return <Button onClick={onClick}>Guardar</Button>;
}
```

- **Server Components:** En Next.js, los componentes que no incluyen "use client" se ejecutan únicamente en el servidor, generando HTML listo para enviar al cliente y permitiendo acceder de forma segura a APIs:

```
export default async function SubjectsServerPage() {
  const subjects = await fetch(process.env.API_URL + "/subjects", { cache: "no-store" }
  return <ul>{subjects.map(s => <li key={s.id}>{s.name}</li>)}</ul>;
}
```

- **Hybrid Rendering:** Se combinan estrategias de renderizado como SSR (Server-Side Rendering), SSG (Static Site Generation) y CSR (Client-Side Rendering) según el tipo de página. Esto permite optimizar seguridad, rendimiento, interactividad y experiencia de usuario:

```
// SSR
export const dynamic = "force-dynamic";
export default async function Dashboard() {
  const user = await fetchUserFromServer();
  return <h1>Bienvenido, {user.name}</h1>;
}

// CSR
"use client";
export default function Counter() {
  const [n, setN] = useState(0);
  return <button onClick={()=>setN(n+1)}>{n}</button>;
}
```

6.5. Seguridad y Autenticación

La autenticación en la aplicación se implementa mediante el servicio `AuthService`, que se encarga de validar las credenciales de los usuarios y gestionar los tokens de sesión.

Validación de Credenciales

El método `login` recibe un *account* y *password* como entrada y realiza los siguientes pasos:

1. Consulta la base de datos PostgreSQL a través de `PrismaService` para obtener la información del usuario, incluyendo roles y relaciones con los modelos `teachers` y `students`.
2. Verifica que el usuario exista y que su cuenta esté activa. En caso contrario, se lanza una excepción `UnauthorizedException`.
3. Compara la contraseña proporcionada con la almacenada en la base de datos utilizando `bcrypt`, asegurando que las contraseñas se manejen de manera segura mediante hashing.
4. Determina atributos adicionales del usuario, como si un docente es *head-Teacher*, para incluirlos en el token de autenticación.

Generación de Token JWT

Una vez validadas las credenciales, se genera un **token JWT** mediante `JwtService`, incluyendo información relevante del usuario:

- Identificador del usuario (`id_us`).
- Cuenta del usuario (`account`).
- Rol del usuario (`role`).
- Atributos adicionales según el rol, como `headTeacher`.

El token JWT permite la autenticación de las solicitudes posteriores sin necesidad de revalidar las credenciales, manteniendo la seguridad de la sesión de manera eficiente y escalable.

Cierre de Sesión

El método `logout` proporciona una respuesta estándar para cerrar la sesión, indicando que la sesión ha sido finalizada y sugiriendo la limpieza de cookies del cliente, reforzando así la seguridad de la aplicación.

Resumen

En conjunto, el servicio de autenticación garantiza la verificación segura de usuarios, controla el acceso basado en roles, maneja sesiones seguras con JWT y protege la integridad de la información del usuario, cumpliendo con buenas prácticas de seguridad en aplicaciones web.

7. Instrucciones para la Instalación, Ejecución y Mantenimiento del Proyecto

Esta sección describe los pasos necesarios para poner en funcionamiento el proyecto, así como las tareas de mantenimiento recomendadas para garantizar su correcto funcionamiento a lo largo del tiempo.

7.1. Requisitos Previos

Antes de instalar y ejecutar el sistema, es necesario contar con los siguientes componentes:

- **Node.js** versión 18 o superior.
- **npm** o **yarn** como gestor de paquetes.
- **PostgreSQL** como motor de base de datos.
- Archivo `.env` configurado con las variables de entorno necesarias:
  - `DATABASE_URL`
  - `PORT`
  - `JWT_SECRET`

7.2. Instalación del Proyecto

1. Clonar el repositorio:

   ```
   git clone <url-del-repositorio>
   cd <nombre-del-proyecto>
   ```

2. Instalar dependencias:

   ```
   npm install
   ```

3. Generar el cliente de Prisma:

   ```
   npx prisma generate
   ```

7.3. Configuración de la Base de Datos

El proyecto utiliza Prisma como ORM, por lo que se requiere ejecutar las migraciones antes del primer inicio:

```
npx prisma migrate deploy
```

En caso de desarrollo local, pueden aplicarse las migraciones directamente:

```
npx prisma migrate dev
```

7.4. Ejecución de Datos Semilla

El sistema cuenta con varios archivos de semillas independientes. Para ejecutarlos en secuencia se dispone del comando:

```
npm run db:seed
```

Este comando ejecuta múltiples scripts de seed ubicados dentro de la carpeta `prisma/seed/`, permitiendo poblar la base de datos con datos iniciales como usuarios, roles, asignaturas, temas, etc.

7.5. Ejecución del Proyecto

- Modo desarrollo:

  ```
  npm run start:dev
  ```

- Modo producción:

  ```
  npm run build
  npm run start:prod
  ```

7.6. Mantenimiento del Sistema

Para garantizar la estabilidad del sistema, se recomiendan las siguientes prácticas:

- **Actualizar dependencias de manera periódica** mediante:

  ```
  npm update
  ```

- **Supervisar cambios en el esquema de Prisma**. Toda modificación en los modelos debe acompañarse de una nueva migración:

  ```
  npx prisma migrate dev --name <descripcion>
  ```

- **Regenerar el cliente de Prisma** cuando cambie el esquema:

  ```
  npx prisma generate
  ```

- **Respaldar la base de datos** regularmente, especialmente antes de aplicar migraciones en producción.
- **Mantener sincronización entre seeds y el esquema**. Si se añaden datos nuevos o relaciones, actualizar los archivos de semillas correspondientes.
- **Revisión continua de logs**. En entornos productivos se recomienda un sistema como PM2 o Docker para monitoreo y reinicio automático.

Documentación de la API: Todos los endpoints del backend están documentados mediante OpenAPI/Swagger. La documentación se genera automáticamente a partir de los controladores y DTOs definidos en NestJS, y se encuentra disponible en el directorio `docs/` del proyecto. Esto permite a los desarrolladores conocer la estructura de la API, los parámetros requeridos, los tipos de respuesta y los códigos de estado HTTP asociados, facilitando la integración y pruebas de manera consistente.

Esta guía cubre todo el ciclo básico de vida del sistema, permitiendo que cualquier desarrollador pueda instalar, ejecutar y mantener el proyecto de forma adecuada.