Informe Detallado de la API del Desarrollo Impulso Capital

Este es el resumen detallado del **backend** que hemos desarrollado para la plataforma de Impulso Capital, donde se manejan usuarios, roles, permisos, creación y manejo de tablas dinámicas tanto para los módulos de inscripción como los módulos de proveedores asegurando que el sistema sea siempre modular, escalable y seguro.

---

1. Configuración Inicial

Estructura del Proyecto

El proyecto backend se ha organizado dentro de la carpeta `back`. Utilizamos **Node.js**, **Express** como framework para el servidor, **Sequelize** como ORM y **PostgreSQL** como la base de datos relacional. A continuación, detallamos los principales archivos y directorios que estructuran el backend:

- **`server.js`**: Punto de entrada del servidor, donde se configuran las rutas y se inicializa la conexión con la base de datos.
- **`src/models/`**: Contiene los modelos Sequelize para las entidades `User`, `Role`, `Permission`, y las asociaciones entre ellas.
- **`src/routes/`**: Define las rutas de la API para usuarios, roles y permisos.
- **`src/controllers/`**: Lógica de control para gestionar las operaciones CRUD de usuarios, roles, y permisos.
- **`src/middlewares/`**: Autenticación JWT y autorización basada en roles y permisos.

---

2. Base de Datos y Modelos

Modelos Definidos

Modelo de Asociaciones (`association.js`)

Este archivo define las asociaciones entre los modelos de `Role` (Rol) y `Permission` (Permiso), estableciendo una relación de muchos a muchos (many-to-many). Este tipo de relación es clave para asignar múltiples permisos a un rol y para que varios roles puedan tener acceso a un mismo permiso.

---

1. Asociación de Muchos a Muchos entre Roles y Permisos

- **Relación:** `Role.belongsToMany(Permission, { through: 'role_permissions', foreignKey: 'role_id', as: 'permissions' })`
- **Descripción:** Define una relación muchos a muchos entre el modelo `Role` (Rol) y el modelo `Permission` (Permiso). Un rol puede tener múltiples permisos, y un permiso puede estar asociado a varios roles. La relación se maneja a través de una tabla intermedia llamada `role_permissions`.
- **Parámetros:**
  - `through`: Define la tabla intermedia que almacena las relaciones (`role_permissions`).
  - `foreignKey`: Define la clave foránea de la relación, en este caso `role_id`.
  - `as`: Define un alias para acceder a los permisos desde el rol, en este caso `'permissions'`.

---

2. Asociación de Muchos a Muchos entre Permisos y Roles

- **Relación:** `Permission.belongsToMany(Role, { through: 'role_permissions', foreignKey: 'permission_id', as: 'roles' })`
- **Descripción:** Similar a la anterior, esta asociación define que un permiso puede estar asociado a varios roles. También usa la tabla intermedia `role_permissions` y define las claves foráneas correspondientes.
- **Parámetros:**
  - `through`: Define la tabla intermedia que almacena las relaciones (`role_permissions`).
  - `foreignKey`: Define la clave foránea de la relación, en este caso `permission_id`.
  - `as`: Define un alias para acceder a los roles desde el permiso, en este caso `'roles'`.

Modelo de Archivo (`File.js`)

Este archivo define el modelo `File`, que representa los archivos subidos en la plataforma. Cada archivo está asociado a un registro de una tabla dinámica en la plataforma (por ejemplo, una tabla de inscripciones o proveedores). Este modelo incluye información clave sobre el archivo, como su nombre, la tabla y el registro al que está vinculado.

---

1. Definición del Modelo `File`

- **Modelo:** `File`
- **Descripción:** El modelo `File` representa un archivo almacenado en el sistema. Los archivos se asocian a registros específicos dentro de tablas dinámicas, como `inscription_` o `provider_`, y este modelo guarda la información necesaria para gestionar los archivos.
- **Campos del Modelo:**
    - `id` (entero, clave primaria, autoincremental): Identificador único del archivo.
    - `table_name` (cadena de texto): El nombre de la tabla a la que está asociado el archivo (por ejemplo, `inscription_123` o `provider_456`).
    - `record_id` (entero): El ID del registro dentro de la tabla asociada.
    - `name` (cadena de texto): El nombre del archivo, incluyendo su extensión (por ejemplo, `documento.pdf`).
    - `file_path` (cadena de texto): Ruta relativa al archivo dentro del sistema de archivos. Esta ruta se usa para acceder al archivo en el servidor.
    - `created_at` (fecha y hora): Fecha y hora en que el archivo fue creado/subido.
    - `updated_at` (fecha y hora): Fecha y hora en que el archivo fue actualizado por última vez.

---

2. Asociaciones

Actualmente, el modelo `File` no establece asociaciones explícitas con otros modelos en el archivo. Sin embargo, cada archivo está asociado indirectamente con un registro en una tabla dinámica (por ejemplo, `inscription_` o `provider_`), y la relación se gestiona mediante los campos `table_name` y `record_id`.

Modelo de Estructura de Inscripción (`InscriptionStructure.js`)

Este archivo define el modelo `InscriptionStructure`, que representa la estructura de las tablas dinámicas relacionadas con las inscripciones en la plataforma. El modelo almacena los nombres y tipos de los campos de cada tabla de inscripción.

---

1. Definición del Modelo `InscriptionStructure`

- **Modelo:** `InscriptionStructure`
- **Descripción:** Este modelo se utiliza para almacenar la estructura de las tablas dinámicas de inscripción, incluyendo el nombre de la tabla, el nombre de los campos y el tipo de dato correspondiente a cada campo.
- **Campos del Modelo:**

    - `table_name` (cadena de texto, no nulo): El nombre de la tabla de inscripción a la que pertenece la estructura.
    - `field_name` (cadena de texto, no nulo): El nombre del campo dentro de la tabla de inscripción.
    - `field_type` (cadena de texto, no nulo): El tipo de dato asociado al campo (por ejemplo, `VARCHAR`, `INTEGER`, `DATE`, etc.).

---

2. Opciones del Modelo

- **tableName:** `'inscription_structure'` – Define el nombre de la tabla en la base de datos. Este modelo utiliza una tabla llamada `inscription_structure`.
- **freezeTableName:** `true` – Desactiva la pluralización automática que realiza Sequelize, asegurando que el nombre de la tabla en la base de datos no se pluralice (por ejemplo, evita que se convierta en `inscription_structures`).

Modelo de Permiso (`Permission.js`)

Este archivo define el modelo `Permission`, que representa los permisos del sistema. Cada permiso está asociado a un nombre único y es clave para el sistema de autorización, ya que permite definir qué acciones puede realizar un usuario o rol en el sistema.

---

1. Definición del Modelo `Permission`

- **Modelo:** `Permission`
- **Descripción:** El modelo `Permission` se utiliza para almacenar los permisos disponibles en la plataforma. Cada permiso tiene un nombre único que identifica las acciones que pueden ser realizadas por un rol o usuario.
- **Campos del Modelo:**
    - `permission_name` (cadena de texto, no nulo, único): El nombre del permiso. Este campo es único, lo que significa que no puede haber dos permisos con el mismo nombre en la base de datos. Ejemplos de nombres de permisos podrían ser `manage_users`, `view_reports`, `edit_content`, etc.

---

2. Opciones del Modelo

- **tableName:** `'permissions'` – Define el nombre de la tabla en la base de datos. Este modelo utiliza una tabla llamada `permissions`.

- **timestamps:** `false` – Desactiva la creación automática de los campos `createdAt` y `updatedAt` por parte de Sequelize, ya que no se necesitan en este caso.

Modelo de Rol (`Role.js`)

Este archivo define el modelo `Role`, que representa los roles en el sistema. Los roles son fundamentales para la administración de permisos y el control de acceso, ya que definen qué permisos tiene un usuario dentro de la plataforma.

---

1. Definición del Modelo `Role`

- **Modelo:** `Role`
- **Descripción:** El modelo `Role` se utiliza para representar los diferentes roles dentro del sistema. Cada rol tiene un nombre único y una descripción opcional, lo que permite identificar y clasificar las responsabilidades y permisos de los usuarios asignados a estos roles.
- **Campos del Modelo:**
  - `role_name` (cadena de texto, no nulo, único): El nombre del rol. Este campo es único, lo que significa que no puede haber dos roles con el mismo nombre en la base de datos. Ejemplos de nombres de roles incluyen `admin`, `editor`, `viewer`, etc.
  - `description` (texto, opcional): Una descripción opcional del rol, que puede ser utilizada para dar más contexto sobre las responsabilidades y permisos asociados al rol.

---

2. Opciones del Modelo

- **tableName:** `'roles'` – Define el nombre de la tabla en la base de datos. Este modelo utiliza una tabla llamada `roles`.
- **timestamps:** `false` – Desactiva la creación automática de los campos `createdAt` y `updatedAt` por parte de Sequelize, ya que no se necesitan en este caso.

Modelo de Relación entre Roles y Permisos (`RolePermission.js`)

Este archivo define el modelo `RolePermission`, que representa la relación de muchos a muchos entre los roles y los permisos en la plataforma. Este modelo se utiliza para gestionar qué permisos están asignados a cada rol, permitiendo así un control de acceso detallado en el sistema.

---

1. Definición del Modelo `RolePermission`

- **Modelo:** `RolePermission`
- **Descripción:** El modelo `RolePermission` es una tabla intermedia que gestiona la relación muchos a muchos entre los roles y los permisos. Cada entrada en esta tabla asocia un rol con uno o más permisos y viceversa.
- **Campos del Modelo:**
  - `role_id` (entero, clave foránea): El identificador único del rol, que hace referencia al modelo `Role`.
  - `permission_id` (entero, clave foránea): El identificador único del permiso, que hace referencia al modelo `Permission`.

---

2. Relaciones Muchos a Muchos

Este modelo define las asociaciones muchos a muchos entre los roles y los permisos, permitiendo que un rol tenga múltiples permisos y que un permiso pueda ser asignado a varios roles:

- **Relación:** `Role.belongsToMany(Permission, { through: RolePermission, foreignKey: 'role_id' })`
  - Esta relación establece que un rol puede tener varios permisos a través de la tabla intermedia `RolePermission`.
- **Relación:** `Permission.belongsToMany(Role, { through: RolePermission, foreignKey: 'permission_id' })`
  - Esta relación establece que un permiso puede estar asignado a varios roles a través de la tabla intermedia `RolePermission`.

---

3. Opciones del Modelo

- **tableName:** `'role_permissions'` – Define el nombre de la tabla en la base de datos que almacena las relaciones entre roles y permisos.
- **timestamps:** `false` – Desactiva la creación automática de los campos `createdAt` y `updatedAt` por parte de Sequelize, ya que no se necesitan en este caso.

Modelo de Metadatos de Tablas (`TablesMetadata.js`)

Este archivo define el modelo `TablesMetadata`, que se utiliza para almacenar información adicional sobre las tablas dinámicas creadas en la plataforma. El modelo permite marcar una tabla como principal (`is_primary`) y almacena el nombre único de la tabla.

---

1. Definición del Modelo `TablesMetadata`

- **Modelo:** `TablesMetadata`
- **Descripción:** El modelo `TablesMetadata` se utiliza para almacenar metadatos relacionados con las tablas dinámicas creadas en la plataforma. Este modelo proporciona una forma de registrar el nombre de cada tabla y de identificar si una tabla es la principal o no.
- **Campos del Modelo:**
  - `table_name` (cadena de texto, no nulo, único): El nombre de la tabla dinámica en la plataforma. Este campo es único, lo que asegura que no haya duplicados de nombres de tabla en la base de datos.
  - `is_primary` (booleano, valor por defecto `false`): Este campo indica si la tabla es principal o no. Por defecto, el valor es `false`.

---

2. Opciones del Modelo

- **timestamps:** `false` – Desactiva la creación automática de los campos `createdAt` y `updatedAt` por parte de Sequelize, ya que no se necesitan en este caso.

Modelo de Usuario (`User.js`)

Este archivo define el modelo `User`, que representa a los usuarios en la plataforma. Cada usuario tiene un nombre de usuario, un correo electrónico, una contraseña encriptada, un rol asociado y un estado que indica si el usuario está activo o inactivo. Además, el modelo registra la fecha y hora del último inicio de sesión.

---

1. Definición del Modelo `User`

- **Modelo:** `User`
- **Descripción:** El modelo `User` gestiona la información de los usuarios en la plataforma. Cada usuario tiene un rol asignado, lo que le otorga ciertos permisos y accesos según las configuraciones del sistema de roles y permisos.
- **Campos del Modelo:**
  - `username` (cadena de texto, no nulo, único): Nombre de usuario. Debe ser único en la plataforma.
  - `email` (cadena de texto, no nulo, único): Correo electrónico del usuario. También debe ser único.
  - `password` (cadena de texto, no nulo): Contraseña del usuario, que se almacena de manera encriptada.

- `role_id` (entero, clave foránea): El identificador del rol asignado al usuario. La clave foránea hace referencia al modelo `Role`.
- `status` (entero, no nulo, valor por defecto `1`): Indica si el usuario está activo (`1`) o inactivo (`0`).
- `last_login` (fecha y hora, opcional): Fecha y hora del último inicio de sesión del usuario.
- `created_at` (fecha y hora): Fecha en que se creó el registro del usuario.
- `updated_at` (fecha y hora): Fecha de la última actualización del registro del usuario.

---

2. Relación entre Usuario y Rol

- **Relación:** `User.belongsTo(Role, { foreignKey: 'role_id' })`
  - Esta relación define que cada usuario está asociado a un solo rol. Un usuario puede tener un único rol, pero un rol puede estar asignado a múltiples usuarios.
  - **Foreign Key:** `role_id` – Clave foránea que conecta el modelo `User` con el modelo `Role`.

---

3. Opciones del Modelo

- **tableName:** `'users'` – Define el nombre de la tabla en la base de datos que almacena los usuarios.
- **timestamps:** `true` – Activa la creación automática de los campos `createdAt` y `updatedAt`.
  - `createdAt`: Renombrado a `created_at`.
  - `updatedAt`: Renombrado a `updated_at`.

---

3. Autenticación y Autorización

Autenticación con JWT

- Utilizamos **JWT (JSON Web Tokens)** para autenticar a los usuarios.
- Durante el inicio de sesión, se genera un token JWT que contiene el `id` del usuario, su `email`, y su `role_id`.
- El token tiene una duración de 1 hora y debe ser enviado en el encabezado `Authorization` en todas las solicitudes protegidas.

Autorización por Roles y Permisos

- Implementamos un middleware `authenticateRole` para verificar si el usuario tiene el rol adecuado para acceder a determinadas rutas.
- También creamos el middleware `authorizePermission` para verificar si el usuario tiene el **permiso** necesario para acceder a una ruta específica.

Ejemplo de Rutas Protegidas

- **Ruta para obtener todos los usuarios**: Protegida con el permiso `view_users`.
- **Ruta para crear un nuevo usuario**: Protegida con el permiso `manage_users`.

---

4. Funcionalidades CRUD Implementadas (Archivos de Rutas)

Rutas de Inscripción (`inscriptionRoutes.js`)

Este archivo define las rutas para gestionar las operaciones relacionadas con las tablas dinámicas de inscripción y proveedores, como la creación de tablas, la gestión de registros, la carga de archivos CSV, y la descarga de datos. Las rutas están protegidas por autenticación JWT y permisos específicos.

---

1. Crear una Nueva Tabla

- **Ruta:** `POST /create-table`
- **Descripción:** Crea una nueva tabla dinámica para inscripciones o proveedores.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.createTable`

---

2. Listar Todas las Tablas

- **Ruta:** `GET /tables`
- **Descripción:** Devuelve una lista de todas las tablas creadas en el sistema.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.listTables`

---

3. Eliminar una Tabla

- **Ruta:** `DELETE /tables/:table_name`
- **Descripción:** Elimina una tabla específica siempre y cuando esté vacía.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.deleteTable`

---

4. Editar una Tabla (Agregar o Eliminar Columnas)

- **Ruta:** `PUT /tables/:table_name`
- **Descripción:** Permite agregar o eliminar columnas de una tabla existente.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.editTable`

---

5. Agregar un Registro a una Tabla

- **Ruta:** `POST /tables/:table_name/record`
- **Descripción:** Agrega un nuevo registro a una tabla dinámica.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.addRecord`

---

6. Obtener los Campos de una Tabla

- **Ruta:** `GET /tables/:table_name/fields`
- **Descripción:** Devuelve la estructura de campos de una tabla específica.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**

    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.getTableFields`

---

7. Descargar Plantilla CSV

- **Ruta:** `GET /tables/:table_name/csv-template`
- **Descripción:** Descarga una plantilla en formato CSV con los campos de la tabla para facilitar la carga de datos.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.downloadCsvTemplate`

---

8. Cargar un Archivo CSV

- **Ruta:** `POST /tables/:table_name/upload-csv`
- **Descripción:** Permite cargar un archivo CSV para agregar datos en masa a una tabla específica.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
    - `multer`: Middleware para gestionar la carga de archivos.
- **Controlador asociado:** `inscriptionController.uploadCsv`

---

9. Descargar Datos de una Tabla en CSV

- **Ruta:** `GET /tables/:table_name/download-csv`
- **Descripción:** Descarga los datos de una tabla en formato CSV.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.downloadCsvData`

---

10. Obtener Registros de una Tabla

- **Ruta:** `GET /tables/:table_name/records`
- **Descripción:** Devuelve los registros almacenados en una tabla específica.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.getTableRecords`

---

11. Obtener un Registro Específico

- **Ruta:** `GET /tables/:table_name/record/:record_id`
- **Descripción:** Devuelve los detalles de un registro específico por su ID.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.getTableRecordById`

---

12. Actualizar un Registro Específico

- **Ruta:** `PUT /tables/:table_name/record/:record_id`
- **Descripción:** Actualiza los detalles de un registro específico por su ID.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.updateTableRecord`

---

13. Actualizar el Estado de Principal de una Tabla

- **Ruta:** `PUT /tables/:table_name/principal`
- **Descripción:** Actualiza el estado de "tabla principal" para una tabla específica.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**

    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.updatePrincipalStatus`

---

14. Actualización Masiva de Registros

- **Ruta:** `PUT /tables/:table_name/bulk-update`
- **Descripción:** Permite realizar actualizaciones masivas en los registros de una tabla.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.bulkUpdateRecords`

---

15. Obtener Opciones de un Campo Específico

- **Ruta:** `GET /tables/:table_name/field-options/:field_name`
- **Descripción:** Devuelve las opciones de un campo específico, generalmente en el caso de claves foráneas.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.getFieldOptions`

---

16. Subir un Archivo Asociado a un Registro

- **Ruta:** `POST /tables/:table_name/record/:record_id/upload`
- **Descripción:** Sube un archivo asociado a un registro específico en la tabla.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT.
    - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
    - `multer`: Middleware para gestionar la carga de archivos.
- **Controlador asociado:** `inscriptionController.uploadFile`

---

17. Obtener Archivos Asociados a un Registro

- **Ruta:** `GET /tables/:table_name/record/:record_id/files`
- **Descripción:** Devuelve los archivos asociados a un registro específico.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.getFiles`

---

18. Eliminar un Archivo Asociado a un Registro

- **Ruta:** `DELETE /tables/:table_name/record/:record_id/file/:file_id`
- **Descripción:** Elimina un archivo específico asociado a un registro.
- **Permisos requeridos:** `manage_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('manage_tables')`: Verifica que el usuario tenga el permiso de `manage_tables`.
- **Controlador asociado:** `inscriptionController.deleteFile`

---

19. Descargar un ZIP con Archivos Asociados a un Registro

- **Ruta:** `GET /tables/:table_name/record/:record_id/download-zip`
- **Descripción:** Descarga todos los archivos asociados a un registro específico en formato ZIP.
- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.downloadZip`

---

20. Descargar Múltiples Archivos en ZIP

- **Ruta:** `POST /download-multiple-zip`
- **Descripción:** Descarga archivos seleccionados de varias tablas y registros en un único archivo ZIP.

- **Permisos requeridos:** `view_tables`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT.
  - `authorizePermission('view_tables')`: Verifica que el usuario tenga el permiso de `view_tables`.
- **Controlador asociado:** `inscriptionController.downloadMultipleZip`

Rutas de Permisos (`permissionRoutes.js`)

Este archivo define las rutas para gestionar las operaciones relacionadas con los permisos del sistema, como la creación, actualización, obtención y eliminación de permisos. Las rutas están protegidas por autenticación JWT.

---

1. Crear un Nuevo Permiso

- **Ruta:** `POST /`
- **Descripción:** Crea un nuevo permiso en el sistema.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `permissionController.createPermission`

---

2. Obtener Todos los Permisos

- **Ruta:** `GET /`
- **Descripción:** Obtiene una lista de todos los permisos existentes en el sistema.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `permissionController.getPermissions`

---

3. Actualizar un Permiso

- **Ruta:** `PUT /:id`
- **Descripción:** Actualiza un permiso existente según su ID.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.

- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `permissionController.updatePermission`

---

4. Eliminar un Permiso

- **Ruta:** `DELETE /:id`
- **Descripción:** Elimina un permiso existente según su ID.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `permissionController.deletePermission`

Rutas de Asignación de Permisos a Roles (`rolePermissionRoutes.js`)

Este archivo define la ruta para gestionar la asignación de permisos a los roles en la plataforma. La ruta está protegida por autenticación JWT.

---

1. Asignar Permisos a un Rol

- **Ruta:** `POST /assign`
- **Descripción:** Asigna una lista de permisos a un rol específico en el sistema.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `rolePermissionController.assignPermissionsToRole`

Rutas de Roles (`roleRoutes.js`)

Este archivo define las rutas para gestionar los roles en la plataforma, como la creación, obtención, actualización y eliminación de roles. Las rutas están protegidas por autenticación JWT.

---

1. Crear un Nuevo Rol

- **Ruta:** `POST /`
- **Descripción:** Crea un nuevo rol en el sistema.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `roleController.createRole`

---

2. Obtener Todos los Roles

- **Ruta:** `GET /`
- **Descripción:** Obtiene una lista de todos los roles en el sistema.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `roleController.getRoles`

---

3. Actualizar un Rol

- **Ruta:** `PUT /:id`
- **Descripción:** Actualiza los detalles de un rol específico según su ID.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `roleController.updateRole`

---

4. Eliminar un Rol

- **Ruta:** `DELETE /:id`
- **Descripción:** Elimina un rol específico del sistema según su ID.
- **Permisos requeridos:** Ninguno específico, pero requiere autenticación.
- **Middlewares:**
    - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `roleController.deleteRole`

Rutas de Usuarios (`userRoutes.js`)

Este archivo define las rutas para gestionar las operaciones relacionadas con los usuarios, como la creación, inicio de sesión, recuperación de contraseñas, actualización, eliminación y cambio de estado de usuarios. Algunas rutas están protegidas por autenticación JWT y requieren permisos específicos.

---

1. Crear un Nuevo Usuario

- **Ruta:** `POST /`
- **Descripción:** Crea un nuevo usuario en el sistema.
- **Permisos requeridos:** `manage_users`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
  - `authorizePermission('manage_users')`: Verifica que el usuario tenga el permiso de `manage_users`.
- **Controlador asociado:** `userController.createUser`

---

2. Inicio de Sesión

- **Ruta:** `POST /login`
- **Descripción:** Permite a los usuarios iniciar sesión en el sistema. No requiere autenticación previa.
- **Permisos requeridos:** Ninguno.
- **Middlewares:**
  - Ninguno.
- **Controlador asociado:** `userController.login`

---

3. Obtener Todos los Usuarios

- **Ruta:** `GET /`
- **Descripción:** Devuelve una lista de todos los usuarios registrados en el sistema.
- **Permisos requeridos:** `view_users`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
  - `authorizePermission('view_users')`: Verifica que el usuario tenga el permiso de `view_users`.
- **Controlador asociado:** `userController.getUsers`

---

4. Obtener un Usuario por ID

- **Ruta:** `GET /:id`
- **Descripción:** Devuelve los detalles de un usuario específico según su ID.
- **Permisos requeridos:** `view_users`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
  - `authorizePermission('view_users')`: Verifica que el usuario tenga el permiso de `view_users`.
- **Controlador asociado:** `userController.getUserById`

---

5. Actualizar un Usuario

- **Ruta:** `PUT /:id`
- **Descripción:** Actualiza los detalles de un usuario específico según su ID.
- **Permisos requeridos:** `manage_users`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
  - `authorizePermission('manage_users')`: Verifica que el usuario tenga el permiso de `manage_users`.
- **Controlador asociado:** `userController.updateUser`

---

6. Eliminar un Usuario

- **Ruta:** `DELETE /:id`
- **Descripción:** Elimina un usuario específico del sistema según su ID.
- **Permisos requeridos:** `manage_users`
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
  - `authorizePermission('manage_users')`: Verifica que el usuario tenga el permiso de `manage_users`.
- **Controlador asociado:** `userController.deleteUser`

---

7. Solicitar Recuperación de Contraseña

- **Ruta:** `POST /forgot-password`

- **Descripción:** Envía un correo con un token de recuperación de contraseña. Esta ruta es pública y no requiere autenticación.
- **Permisos requeridos:** Ninguno.
- **Middlewares:**
  - Ninguno.
- **Controlador asociado:** `userController.forgotPassword`

---

8. Restablecer Contraseña

- **Ruta:** `POST /reset-password/:token`
- **Descripción:** Permite al usuario restablecer su contraseña usando un token recibido por correo. No requiere autenticación previa.
- **Permisos requeridos:** Ninguno.
- **Middlewares:**
  - Ninguno.
- **Controlador asociado:** `userController.resetPassword`

---

9. Cambiar el Estado de un Usuario

- **Ruta:** `PUT /:id/toggle-status`
- **Descripción:** Cambia el estado de un usuario (activo/inactivo).
- **Permisos requeridos:** Ninguno, pero requiere autenticación.
- **Middlewares:**
  - `authenticateJWT`: Verifica el token JWT para asegurar que el usuario esté autenticado.
- **Controlador asociado:** `userController.toggleUserStatus`

---

5. Seguridad

- **Encriptación de contraseñas**: Utilizamos **bcryptjs** para encriptar las contraseñas antes de almacenarlas en la base de datos.
- **Autenticación segura**: JWT asegura que solo los usuarios autenticados puedan acceder a rutas protegidas.
- **Protección de rutas**: Las rutas críticas están protegidas por roles y permisos, lo que garantiza que solo los usuarios con los privilegios adecuados puedan acceder a ciertas funcionalidades.

---

7. Tablas en la Base de Datos

Las tablas principales de la base de datos son:

- `users`: Almacena los datos de los usuarios del sistema.
- `roles`: Almacena los roles definidos en el sistema.
- `permissions`: Almacena los permisos del sistema.
- `role_permissions`: Tabla intermedia que almacena la relación muchos a muchos entre roles y permisos.

---

8. Controladores

Controlador de Inscripciones (`inscriptionController.js`)

Este archivo contiene varios controladores para manejar tablas dinámicas relacionadas con inscripciones y proveedores en la plataforma. Los principales puntos incluyen la creación y manipulación de tablas, gestión de registros, manejo de archivos CSV, y soporte para claves foráneas.

---

1. Crear Tabla Dinámica con Soporte para Claves Foráneas

- **Ruta:** `POST /api/inscription/create-table`
- **Descripción:** Crea una nueva tabla dinámica con los campos especificados, con soporte para claves foráneas.
- **Entradas:**
  - `table_name`: Nombre de la tabla (debe comenzar con `inscription_` o `provider_`).
  - `fields`: Array con los detalles de cada campo (nombre, tipo, nulos permitidos, etc.).
- **Respuesta exitosa:**
  - Estado 201 con un mensaje de éxito.
- **Errores comunes:**
  - Falta de datos requeridos o tipos de datos inválidos.

---

2. Listar Tablas Dinámicas

- **Ruta:** `GET /api/inscription/list-tables`
- **Descripción:** Lista las tablas dinámicas creadas en la plataforma que comienzan con `inscription_` o `provider_`.
- **Entradas opcionales:**
  - `tableType`: Tipo de tabla a listar (`provider` o `inscription`).

    - `isPrimary`: Filtro opcional para mostrar solo las tablas principales.
- **Respuesta exitosa:**
    - Estado 200 con una lista de tablas.

---

3. Eliminar una Tabla Vacía

- **Ruta:** `DELETE /api/inscription/delete-table/:table_name`
- **Descripción:** Elimina una tabla, pero solo si está vacía.
- **Entradas:**
    - `table_name`: Nombre de la tabla a eliminar.
- **Errores comunes:**
    - La tabla no está vacía.

---

4. Editar una Tabla Dinámica

- **Ruta:** `PUT /api/inscription/edit-table/:table_name`
- **Descripción:** Agrega o elimina columnas de una tabla existente. No se permite editar campos ya existentes.
- **Entradas:**
    - `fieldsToAdd`: Campos a agregar.
    - `fieldsToDelete`: Campos a eliminar.
- **Errores comunes:**
    - Intentar editar campos existentes o eliminar columnas con datos.

---

5. Agregar un Registro a una Tabla Dinámica

- **Ruta:** `POST /api/inscription/add-record/:table_name`
- **Descripción:** Inserta un nuevo registro en una tabla dinámica, con validación para claves foráneas.
- **Entradas:**
    - `table_name`: Nombre de la tabla.
    - `recordData`: Datos del registro.
- **Errores comunes:**
    - Claves foráneas que no existen en la tabla relacionada.

---

6. Obtener los Campos de una Tabla Específica

- **Ruta:** `GET /api/inscription/get-fields/:table_name`
- **Descripción:** Obtiene la lista de columnas de una tabla dinámica, incluyendo claves foráneas.

- **Entradas:**
    - `table_name`: Nombre de la tabla.

---

7. Descargar Plantilla CSV

- **Ruta:** `GET /api/inscription/download-csv-template/:table_name`
- **Descripción:** Genera y descarga una plantilla CSV con columnas basadas en los campos de la tabla.
- **Entradas:**
    - `table_name`: Nombre de la tabla.

---

8. Subir y Procesar un Archivo CSV

- **Ruta:** `POST /api/inscription/upload-csv/:table_name`
- **Descripción:** Sube un archivo CSV, lo procesa e inserta sus datos en la tabla dinámica especificada.
- **Entradas:**
    - `table_name`: Nombre de la tabla.
    - Archivo CSV en la solicitud.
- **Errores comunes:**
    - El archivo no tiene el formato correcto o faltan columnas obligatorias.

---

9. Descargar Datos en CSV

- **Ruta:** `GET /api/inscription/download-csv/:table_name`
- **Descripción:** Descarga todos los registros de una tabla en formato CSV.
- **Entradas:**
    - `table_name`: Nombre de la tabla.

---

10. Obtener Registros de una Tabla Dinámica con Relaciones

- **Ruta:** `GET /api/inscription/get-records/:table_name`
- **Descripción:** Obtiene todos los registros de una tabla dinámica, incluyendo los valores de las claves foráneas relacionadas.
- **Entradas:**
    - `table_name`: Nombre de la tabla.

---

11. Obtener un Registro por ID

- **Ruta:** `GET /api/inscription/get-record/:table_name/:record_id`
- **Descripción:** Devuelve un registro específico por su ID, incluyendo relaciones con otras tablas.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `record_id`: ID del registro.

---

12. Actualizar un Registro

- **Ruta:** `PUT /api/inscription/update-record/:table_name/:record_id`
- **Descripción:** Actualiza un registro específico por su ID.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `record_id`: ID del registro.
  - Datos a actualizar en el cuerpo de la solicitud.

---

13. Actualizar Estado de Principal de una Tabla

- **Ruta:** `PUT /api/inscription/update-principal-status/:table_name`
- **Descripción:** Actualiza el estado de "principal" de una tabla dinámica en la metadata.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `is_primary`: Valor booleano que indica si la tabla es principal.

---

14. Actualizar Múltiples Registros

- **Ruta:** `PUT /api/inscription/bulk-update-records/:table_name`
- **Descripción:** Actualiza múltiples registros de una tabla dinámica.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `recordIds`: Array con los IDs de los registros a actualizar.
  - `updates`: Datos a actualizar.

---

15. Obtener Opciones para un Campo Específico

- **Ruta:** `GET /api/inscription/get-field-options/:table_name/:field_name`
- **Descripción:** Devuelve opciones para un campo de clave foránea en una tabla.
- **Entradas:**

- `table_name`: Nombre de la tabla.
- `field_name`: Nombre del campo.

---

16. Subir un Archivo

- **Ruta:** `POST /api/inscription/upload-file/:table_name/:record_id`
- **Descripción:** Sube un archivo relacionado a un registro específico en una tabla dinámica.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `record_id`: ID del registro.
  - Archivo a subir.

---

17. Obtener Archivos Relacionados a un Registro

- **Ruta:** `GET /api/inscription/get-files/:table_name/:record_id`
- **Descripción:** Obtiene todos los archivos asociados a un registro específico en una tabla dinámica.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `record_id`: ID del registro.

---

18. Descargar Archivos como ZIP

- **Ruta:** `GET /api/inscription/download-zip/:table_name/:record_id`
- **Descripción:** Descarga todos los archivos relacionados con un registro específico en formato ZIP.
- **Entradas:**
  - `table_name`: Nombre de la tabla.
  - `record_id`: ID del registro.

---

19. Eliminar un Archivo

- **Ruta:** `DELETE /api/inscription/delete-file/:file_id/:record_id`
- **Descripción:** Elimina un archivo de un registro específico en una tabla dinámica.
- **Entradas:**
  - `file_id`: ID del archivo.
  - `record_id`: ID del registro asociado.

---

20. Listar Tablas de Proveedores

- **Ruta:** `GET /api/inscription/list-provider-tables`
- **Descripción:** Lista todas las tablas dinámicas que empiezan con `provider_` junto con su información de metadata.

---

21. Descargar Múltiples Archivos como ZIP

- **Ruta:** `POST /api/inscription/download-multiple-zip`
- **Descripción:** Permite descargar múltiples archivos desde diferentes tablas dinámicas en un solo archivo ZIP.
- **Entradas:**
  - `tables`: Array con los nombres de las tablas.
  - `recordIds`: Array con los IDs de los registros.

Controlador de Permisos (`permissionController.js`)

Este archivo contiene los controladores para gestionar los permisos en la plataforma. Los permisos son una parte esencial del sistema de autorización, permitiendo que ciertos usuarios accedan o modifiquen recursos específicos.

---

1. Crear un Nuevo Permiso

- **Ruta:** `POST /api/permissions`
- **Descripción:** Crea un nuevo permiso en el sistema.
- **Entradas:**
  - `permission_name`: Nombre del permiso.
- **Respuesta exitosa:**
  - Estado 201 con un mensaje de éxito y el nuevo permiso creado.
- **Errores comunes:**
  - Error de conexión a la base de datos o problemas al crear el permiso.

---

2. Obtener Todos los Permisos

- **Ruta:** `GET /api/permissions`
- **Descripción:** Devuelve una lista de todos los permisos disponibles en el sistema.
- **Entradas:**
  - Ninguna.
- **Respuesta exitosa:**
  - Estado 200 con una lista de todos los permisos.

- **Errores comunes:**
    - Error al acceder a la base de datos.

---

3. Actualizar un Permiso

- **Ruta:** `PUT /api/permissions/:id`
- **Descripción:** Actualiza los detalles de un permiso específico por su ID.
- **Entradas:**
    - `id`: ID del permiso a actualizar.
    - `permission_name`: Nuevo nombre del permiso (opcional).
- **Validaciones:**
    - Se verifica que el permiso exista antes de realizar la actualización.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito y el permiso actualizado.
- **Errores comunes:**
    - El permiso no fue encontrado o error al guardar los cambios.

---

4. Eliminar un Permiso

- **Ruta:** `DELETE /api/permissions/:id`
- **Descripción:** Elimina un permiso específico por su ID.
- **Entradas:**
    - `id`: ID del permiso a eliminar.
- **Validaciones:**
    - Se verifica que el permiso exista antes de eliminarlo.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito si el permiso fue eliminado correctamente.
- **Errores comunes:**
    - El permiso no fue encontrado o error al eliminarlo.

---

5. Crear Permisos para las Tablas (Interno)

- **Función interna:** `createTablePermissions`
- **Descripción:** Crea permisos predeterminados para la gestión de tablas en el sistema. Esta función es invocada dentro del controlador para asegurar que los permisos necesarios (`view_tables` y `manage_tables`) estén presentes en la base de datos.
- **Uso:**
    - Esta función se ejecuta automáticamente y no está expuesta como una ruta pública.
- **Errores comunes:**

- Error al crear los permisos debido a problemas de conexión o de base de datos.

Conclusión y Estado Actual del Controlador permissionController.js

Este controlador maneja todas las operaciones CRUD relacionadas con los permisos en la plataforma. La creación de permisos específicos para la gestión de tablas garantiza que los usuarios tengan los accesos necesarios para ver y administrar las tablas dinámicas en el sistema.

---

1. Crear un Nuevo Rol

- **Ruta:** `POST /api/roles`
- **Descripción:** Crea un nuevo rol en el sistema, especificando su nombre y descripción.
- **Entradas:**
  - `role_name`: Nombre del rol (obligatorio).
  - `description`: Descripción del rol (opcional).
- **Respuesta exitosa:**
  - Estado 201 con un mensaje de éxito y el nuevo rol creado.
- **Errores comunes:**
  - Error de conexión a la base de datos o problemas al crear el rol.

---

2. Obtener Todos los Roles

- **Ruta:** `GET /api/roles`
- **Descripción:** Devuelve una lista de todos los roles disponibles en el sistema.
- **Entradas:**
  - Ninguna.
- **Respuesta exitosa:**
  - Estado 200 con una lista de todos los roles.
- **Errores comunes:**
  - Error al acceder a la base de datos.

---

3. Actualizar un Rol

- **Ruta:** `PUT /api/roles/:id`
- **Descripción:** Actualiza los detalles de un rol específico por su ID.
- **Entradas:**
  - `id`: ID del rol a actualizar.

    - `role_name`: Nuevo nombre del rol (opcional).
    - `description`: Nueva descripción del rol (opcional).
- **Validaciones:**
    - Se verifica que el rol exista antes de realizar la actualización.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito y el rol actualizado.
- **Errores comunes:**
    - El rol no fue encontrado o error al guardar los cambios.

---

4. Eliminar un Rol

- **Ruta:** `DELETE /api/roles/:id`
- **Descripción:** Elimina un rol específico por su ID.
- **Entradas:**
    - `id`: ID del rol a eliminar.
- **Validaciones:**
    - Se verifica que el rol exista antes de eliminarlo.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito si el rol fue eliminado correctamente.
- **Errores comunes:**
    - El rol no fue encontrado o error al eliminarlo.

Controlador de Asignación de Permisos a Roles (`rolePermissionController.js`)

Este archivo contiene el controlador encargado de asignar múltiples permisos a un rol en la plataforma. La asignación de permisos a roles permite gestionar las autorizaciones y niveles de acceso de los usuarios en función del rol que tengan asignado.

---

1. Asignar Permisos a un Rol

- **Ruta:** `POST /api/role-permissions/assign`
- **Descripción:** Asigna una lista de permisos a un rol específico.
- **Entradas:**
    - `roleId`: ID del rol al cual se le asignarán los permisos.
    - `permissionIds`: Lista de IDs de permisos que se asignarán al rol.
- **Validaciones:**
    - Se valida que el `roleId` y la lista de `permissionIds` sean proporcionados y válidos.
    - Se verifica que el rol exista antes de proceder con la asignación.
    - Se verifica que los permisos proporcionados existan en el sistema.

- **Proceso:**
  - Busca el rol por su `ID`.
  - Busca los permisos por sus `IDs`.
  - Asigna los permisos al rol usando la relación muchos a muchos entre roles y permisos.
- **Respuesta exitosa:**
  - Estado 200 con un mensaje de éxito y el rol actualizado con los nuevos permisos asignados.
- **Errores comunes:**
  - Rol no encontrado: Si no se encuentra el rol especificado por el `roleId`.
  - Permisos no encontrados: Si no se encuentran permisos con los `IDs` proporcionados.
  - Error de validación: Si no se proporcionan el `roleId` o una lista de `permissionIds` válida.

Controlador de Usuarios (`userController.js`)

Este archivo contiene los controladores para gestionar las operaciones relacionadas con los usuarios en la plataforma, incluyendo la creación de usuarios, la autenticación mediante login, la recuperación de contraseñas, y el manejo de roles asociados a los usuarios.

---

1. Crear un Nuevo Usuario

- **Ruta:** `POST /api/users`
- **Descripción:** Crea un nuevo usuario en el sistema, con su rol asociado.
- **Entradas:**
  - `username`: Nombre de usuario.
  - `email`: Correo electrónico del usuario.
  - `password`: Contraseña (se encripta antes de almacenarse).
  - `role_id`: ID del rol asignado al usuario.
- **Respuesta exitosa:**
  - Estado 201 con un mensaje de éxito y el usuario creado.
- **Errores comunes:**
  - Error de conexión a la base de datos o problemas al crear el usuario.

---

2. Obtener Todos los Usuarios

- **Ruta:** `GET /api/users`
- **Descripción:** Devuelve una lista de todos los usuarios en el sistema, junto con sus roles.
- **Entradas:**

    - Ninguna.
- **Respuesta exitosa:**
    - Estado 200 con una lista de todos los usuarios.
- **Errores comunes:**
    - Error al acceder a la base de datos.

---

3. Obtener un Usuario por ID

- **Ruta:** `GET /api/users/:id`
- **Descripción:** Devuelve los detalles de un usuario específico por su ID, incluyendo el rol asignado.
- **Entradas:**
    - `id`: ID del usuario.
- **Respuesta exitosa:**
    - Estado 200 con los datos del usuario.
- **Errores comunes:**
    - Usuario no encontrado o error al acceder a la base de datos.

---

4. Actualizar un Usuario

- **Ruta:** `PUT /api/users/:id`
- **Descripción:** Actualiza los detalles de un usuario específico, incluyendo su rol y otros datos.
- **Entradas:**
    - `id`: ID del usuario.
    - `username`: Nombre de usuario (opcional).
    - `email`: Correo electrónico (opcional).
    - `password`: Contraseña (opcional, se encripta si se proporciona).
    - `role_id`: ID del rol (opcional).
    - `status`: Estado del usuario (activo/inactivo) (opcional).
- **Validaciones:**
    - Se verifica que el usuario exista antes de actualizar.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito y el usuario actualizado.
- **Errores comunes:**
    - Usuario no encontrado o error al guardar los cambios.

---

5. Eliminar un Usuario

- **Ruta:** `DELETE /api/users/:id`
- **Descripción:** Elimina un usuario específico por su ID.
- **Entradas:**

    - `id`: ID del usuario a eliminar.
- **Validaciones:**
    - Se verifica que el usuario exista antes de eliminarlo.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje de éxito si el usuario fue eliminado correctamente.
- **Errores comunes:**
    - Usuario no encontrado o error al eliminarlo.

---

6. Inicio de Sesión (Login)

- **Ruta:** `POST /api/users/login`
- **Descripción:** Autentica a un usuario mediante su correo electrónico y contraseña. Devuelve un token JWT si el login es exitoso.
- **Entradas:**
    - `email`: Correo electrónico del usuario.
    - `password`: Contraseña del usuario.
- **Proceso:**
    - Verifica si el usuario existe en la base de datos.
    - Compara la contraseña ingresada con la contraseña encriptada en la base de datos.
    - Si la contraseña es válida, genera un token JWT y actualiza el campo `last_login` con la fecha actual.
- **Respuesta exitosa:**
    - Estado 200 con el token JWT y los datos del usuario (incluyendo el último login).
- **Errores comunes:**
    - Usuario no encontrado o contraseña incorrecta.

---

7. Solicitar Recuperación de Contraseña

- **Ruta:** `POST /api/users/forgot-password`
- **Descripción:** Envía un correo electrónico con un enlace para restablecer la contraseña.
- **Entradas:**
    - `email`: Correo electrónico del usuario.
- **Proceso:**
    - Verifica si el usuario existe en la base de datos.
    - Genera un token JWT válido por 15 minutos y lo envía por correo.
- **Respuesta exitosa:**
    - Estado 200 con un mensaje indicando que el correo fue enviado.
- **Errores comunes:**
    - Usuario no encontrado o error al enviar el correo.

---

8. Restablecer Contraseña

- **Ruta:** `POST /api/users/reset-password/:token`
- **Descripción:** Restablece la contraseña de un usuario usando el token proporcionado.
- **Entradas:**
  - `token`: Token de recuperación de contraseña.
  - `newPassword`: Nueva contraseña para el usuario.
- **Proceso:**
  - Verifica el token JWT para obtener el ID del usuario.
  - Encripta la nueva contraseña y la guarda en la base de datos.
- **Respuesta exitosa:**
  - Estado 200 con un mensaje de éxito indicando que la contraseña fue restablecida.
- **Errores comunes:**
  - Token inválido o expirado, o usuario no encontrado.

---

9. Cambiar el Estado de un Usuario (Activo/Inactivo)

- **Ruta:** `PUT /api/users/toggle-status/:id`
- **Descripción:** Cambia el estado de un usuario (de activo a inactivo o viceversa).
- **Entradas:**
  - `id`: ID del usuario.
- **Proceso:**
  - Verifica si el usuario existe y alterna su estado entre activo (`1`) e inactivo (`0`).
- **Respuesta exitosa:**
  - Estado 200 con el nuevo estado del usuario.
- **Errores comunes:**
  - Usuario no encontrado o error al cambiar el estado.

Middleware de Autenticación y Autorización (`authMiddleware.js`)

Este archivo contiene los middlewares encargados de la autenticación y autorización en la plataforma, usando JWT (JSON Web Token) para validar las solicitudes y asegurarse de que los usuarios tengan los roles y permisos adecuados para acceder a recursos específicos.

---

1. Autenticación del Token JWT

- **Middleware:** `authenticateJWT`

- **Descripción:** Verifica si la solicitud incluye un token JWT válido en el encabezado `Authorization`. Si el token es válido, la información del usuario se adjunta al objeto `req` y se permite el acceso a las rutas protegidas. Si no es válido, se bloquea el acceso.
- **Proceso:**
  - Verifica que el encabezado `Authorization` esté presente y tenga el formato adecuado (`Bearer <token>`).
  - Si el token es válido, se decodifica y la información del usuario se almacena en `req.user`.
  - Si el token es inválido o no está presente, devuelve un estado 401 o 403 con un mensaje de error.
- **Uso:**
  - Este middleware se debe aplicar en todas las rutas que requieren autenticación mediante JWT.
- **Errores comunes:**
  - Token no proporcionado, mal formateado o inválido.

---

2. Autorización por Rol

- **Middleware:** `authenticateRole`
- **Descripción:** Verifica que el usuario autenticado tenga el rol necesario para acceder a una ruta protegida. Si el rol del usuario no coincide con el rol requerido, se deniega el acceso.
- **Entradas:**
  - `requiredRole`: El rol necesario para acceder a la ruta (ej. `admin`).
- **Proceso:**
  - Compara el rol del usuario (`req.user.role`) con el rol requerido.
  - Si no coincide, devuelve un estado 403 con un mensaje de "Acceso denegado".
  - Si coincide, permite el acceso a la ruta.
- **Uso:**
  - Se aplica en rutas que requieren que el usuario tenga un rol específico.
- **Errores comunes:**
  - El usuario no tiene el rol adecuado para acceder a la ruta.

---

3. Autorización por Permiso

- **Middleware:** `authorizePermission`
- **Descripción:** Verifica si el usuario tiene el permiso necesario para acceder a una ruta específica. El permiso se verifica en función del rol del usuario y los permisos asociados a dicho rol.
- **Entradas:**

    - `requiredPermission`: El permiso requerido para acceder a la ruta (ej. `manage_users`).
- **Proceso:**
    - Busca el rol del usuario en la base de datos e incluye los permisos asociados.
    - Verifica si el rol del usuario tiene el permiso requerido.
    - Si no tiene el permiso, devuelve un estado 403 con un mensaje de "Permiso denegado".
    - Si tiene el permiso, permite el acceso a la ruta.
- **Uso:**
    - Este middleware se aplica en rutas que requieren permisos específicos, como `manage_users`, `view_reports`, etc.
- **Errores comunes:**
    - El usuario no tiene el permiso adecuado para acceder a la ruta.

---

9. Utils

Configuración de Multer para Carga de Archivos (`multerConfig.js`)

Este archivo define la configuración para gestionar la carga de archivos en la plataforma utilizando **Multer**, un middleware de Node.js que permite el manejo de archivos subidos en las solicitudes HTTP.

---

1. Dependencia Utilizada

- **Multer:** Este middleware es utilizado para gestionar la carga de archivos. Permite especificar dónde y cómo almacenar los archivos subidos.

---

2. Configuración del Almacenamiento (`storage`)

La configuración de almacenamiento define cómo y dónde se almacenarán temporalmente los archivos antes de ser procesados por los controladores.

- **destination:** Define la carpeta donde se almacenarán temporalmente los archivos. En este caso, los archivos se guardan en la carpeta `uploads/temp`. Luego, el archivo puede ser movido o procesado por los controladores correspondientes.
    - **Función:** `destination: function (req, file, cb)`
        - El callback `cb(null, 'uploads/temp')` establece que el archivo será almacenado en la carpeta `uploads/temp`.
- **filename:** Define el nombre del archivo temporal para asegurarse de que sea único.

    - **Función:** `filename: function (req, file, cb)`
        - El archivo se guarda temporalmente con un nombre único que combina la marca de tiempo actual (`Date.now()`) y el nombre original del archivo (`file.originalname`), asegurando que no haya colisiones de nombres.

---

3. Configuración de Multer

- **upload:** Se crea una instancia de `multer` utilizando la configuración de almacenamiento definida. Esta instancia puede ser utilizada como middleware en las rutas para gestionar la carga de archivos.

---

4. Exportación del Módulo

- **Módulo exportado:** `upload`
    - Este middleware configurado está listo para ser utilizado en cualquier controlador donde se requiera la carga de archivos, permitiendo la integración rápida y eficiente de esta funcionalidad en las rutas del sistema.

Configuración de Conexión a la Base de Datos con Sequelize (`sequelize.js`)

Este archivo configura y establece la conexión con una base de datos PostgreSQL utilizando **Sequelize**, un ORM (Object Relational Mapper) que facilita la interacción con bases de datos SQL en Node.js.

---

1. Dependencias Utilizadas

- **Sequelize:** Es el ORM utilizado para interactuar con la base de datos PostgreSQL.
- **dotenv:** Carga variables de entorno desde un archivo `.env` para mantener la información sensible, como las credenciales de la base de datos, fuera del código fuente.

---

2. Configuración de la Conexión

La conexión a la base de datos se establece utilizando las variables de entorno definidas en el archivo `.env`.

- **Sequelize:**
  - **new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASSWORD, { host: process.env.DB_HOST, dialect: 'postgres' })**
    - Se crea una nueva instancia de Sequelize utilizando las variables de entorno:
      - `DB_NAME`: Nombre de la base de datos.
      - `DB_USER`: Usuario de la base de datos.
      - `DB_PASSWORD`: Contraseña del usuario.
      - `DB_HOST`: Dirección del servidor de la base de datos.
    - **Dialecto:** Se especifica que el dialecto utilizado es `postgres`, lo que indica que se trabajará con PostgreSQL.

---

3. Verificación de la Conexión

Después de la configuración, se verifica la conexión con la base de datos utilizando el método `authenticate()` de Sequelize:

- **sequelize.authenticate():**
  - **then()**: Si la conexión es exitosa, se muestra el mensaje "Conexión exitosa a PostgreSQL mediante Sequelize".
  - **catch()**: Si la conexión falla, se captura el error y se muestra el mensaje "Error conectando a la base de datos", seguido de los detalles del error.

---

4. Exportación del Módulo

- **Módulo exportado:** `sequelize`
  - Este módulo puede ser utilizado en otros archivos de la aplicación para realizar operaciones con la base de datos, como consultas, creación de modelos y ejecución de transacciones.

Archivo `server.js`

Este archivo es el punto de entrada para el servidor backend de la aplicación, que utiliza **Express** como framework web. Se configura la API para gestionar rutas relacionadas con usuarios, roles, permisos, inscripción, y otras funcionalidades. Además, se conecta con la base de datos PostgreSQL a través de **Sequelize**, un ORM (Object-Relational Mapping).

Dependencias Importadas

1. **express:** Framework minimalista para crear servidores web en Node.js.

2. **cors:** Middleware que permite habilitar CORS (Cross-Origin Resource Sharing) para permitir solicitudes desde diferentes dominios.
3. **sequelize:** Se utiliza para interactuar con la base de datos PostgreSQL. El archivo `./src/utils/sequelize` configura la conexión con la base de datos.
4. **userRoutes, roleRoutes, permissionRoutes, rolePermissionRoutes, inscriptionRoutes:** Rutas específicas que se encargan de gestionar usuarios, roles, permisos, asignación de permisos a roles, y la lógica de inscripción.
5. **path:** Módulo de Node.js que proporciona utilidades para trabajar con rutas y archivos en el sistema de archivos.
6. **dotenv:** Carga variables de entorno desde un archivo `.env` para mantener seguras las configuraciones sensibles como credenciales de base de datos.
7. **associations:** Configura las relaciones entre los modelos de Sequelize, como las relaciones muchos a muchos entre roles y permisos.

Configuración del Servidor

- **app.use(cors()):**
  - Activa el middleware CORS para que el servidor pueda aceptar solicitudes desde dominios externos.
- **app.use(express.json()):**
  - Habilita el middleware para interpretar los cuerpos de las solicitudes en formato JSON.
- **app.use('/uploads', express.static(path.join(__dirname, 'uploads'))):**
  - Define una ruta estática para servir archivos subidos. Los archivos en la carpeta `uploads` estarán disponibles a través de `/uploads`.

Rutas Definidas

Las rutas están organizadas para separar la lógica de usuarios, roles, permisos, asignación de permisos y tablas dinámicas (inscripción).

- **app.use('/api/users', userRoutes):**
  - Gestiona las operaciones relacionadas con usuarios (crear, actualizar, eliminar, etc.).
- **app.use('/api/roles', roleRoutes):**
  - Gestiona las operaciones relacionadas con los roles.
- **app.use('/api/permissions', permissionRoutes):**
  - Gestiona las operaciones relacionadas con los permisos.
- **app.use('/api/role-permissions', rolePermissionRoutes):**
  - Gestiona la asignación de permisos a los roles.
- **app.use('/api/inscriptions', inscriptionRoutes):**
  - Gestiona la lógica relacionada con la creación de tablas dinámicas y la manipulación de registros en la parte de inscripción.
- **app.get('/'):**

- Ruta de prueba para confirmar que la API está en funcionamiento. Responde con el mensaje: **"API Impulso Capital funcionando"**.

Iniciar el Servidor

- **PORT = process.env.PORT || 5000:**
  - El puerto del servidor se toma de la variable de entorno `PORT` o se establece a `5000` si no está definida.
- **app.listen(PORT, async () => {...}):**
  - Inicia el servidor en el puerto definido. También realiza la sincronización de la base de datos utilizando **sequelize.sync({ alter: true })**, que ajusta las tablas según los modelos definidos en Sequelize.
  - **try-catch block:**
    - En el bloque `try`, el servidor intenta sincronizar las tablas de la base de datos. Si tiene éxito, se muestran mensajes de confirmación en la consola:
      - **"Base de datos sincronizada y tablas ajustadas"**
      - **"Servidor corriendo en el puerto ..."**
    - En caso de error, se captura y muestra en la consola el mensaje **"Error sincronizando la base de datos:"** seguido del detalle del error.

Conclusión Final sobre la API y el Estado del Desarrollo de la Plataforma Impulso Capital

La API desarrollada hasta el momento para **Impulso Capital** está completamente funcional y bien estructurada, permitiendo la gestión eficiente de usuarios, roles, permisos, y procesos de inscripción a través de una serie de endpoints seguros y escalables.

La autenticación y autorización mediante **JWT** aseguran que solo los usuarios con los permisos adecuados puedan acceder y realizar acciones críticas, como la creación de nuevos usuarios o la gestión de tablas dinámicas. Esto proporciona un control total sobre los accesos y facilita la administración del sistema en términos de seguridad.

Uno de los elementos clave de esta API es la capacidad de crear y gestionar tablas dinámicas, lo que aporta gran flexibilidad en el manejo de datos relacionados con los procesos de inscripción y proveedores. Además, la funcionalidad para cargar y descargar archivos en formatos como **CSV** está implementada, lo que facilita el manejo masivo de datos y su integración con sistemas externos.

La API está respaldada por una conexión a una base de datos **PostgreSQL** utilizando **Sequelize**, lo que garantiza una gestión de datos eficiente y segura. Las relaciones entre los distintos modelos están claramente definidas, y el sistema ya sincroniza automáticamente los cambios necesarios en la base de datos.