Enunciado - API REST con Node.js, Express y Prisma

Descripción General

Desarrollar una **API REST** basada en **Node.js** con **Express**, utilizando el **ORM Prisma** para la gestión de base de datos. La aplicación debe trabajar con una arquitectura por capas siguiendo la estructura de carpetas especificada.

Tecnologías Requeridas

- **Node.js** (versión 18 o superior)
- **Express.js** - Framework web para Node.js
- **Prisma** - ORM para gestión de base de datos
- **PostgreSQL** - Sistema de gestión de bases de datos relacional
- **express-validator** - Librería obligatoria para validaciones de datos
- **JavaScript** (ES Modules) - **NO se debe usar TypeScript**

Arquitectura y Estructura

La aplicación debe seguir la arquitectura por capas con la siguiente estructura de carpetas:

```
src/
  ├── app.js
  ├── servidor/
  │   └── server.js
  ├── rutas/
  │   ├── test.rutas.js
  │   └── ...
  ├── controladores/
  │   ├── test.controladores.js
  │   └── ...
  ├── servicios/
  │   ├── test.servicios.js
  │   └── ...
  ├── config/
  │   └── prisma.config.js
  └── validators/
      ├── test.validator.js
      └── ...
```

Nota: Los archivos `test.*.js` son ejemplos de referencia que muestran cómo debe estructurarse el código para cada capa. Deben implementarse archivos similares para todas las entidades (`roles` , `categories` , `warehouses` , `users` , `areas` , `products`).

Repositorio de Referencia Principal - Ejemplo de Entidad Test

Se proporciona un repositorio de referencia que contiene la implementación completa de la entidad `test` desarrollada en el taller de introducción. Este repositorio es el **principal y obligatorio** para desarrollar todas las demás entidades del proyecto.

Repositorio de referencia principal: https://github.com/Zedmous/iujo-clase-foc-backend-2025-2

Características del repositorio principal:

- **JavaScript puro** (sin TypeScript)
- **Sin Swagger**
- Implementación completa de la entidad `test` con todas sus capas (rutas, controladores, servicios, validadores)
- Estructura de carpetas de referencia
- Ejemplos de código siguiendo las mejores prácticas establecidas

Uso del repositorio principal:

- Los estudiantes deben usar este repositorio como **guía principal y obligatoria** para implementar las demás entidades
- La estructura y patrones de código de `test` deben **replicarse** para `roles`, `categories`, `warehouses`, `users`, `areas`, y `products`
- Este repositorio demuestra cómo implementar correctamente la arquitectura por capas y la estructura de respuestas obligatoria
- **Este es el repositorio base que deben seguir** ya que está desarrollado con JavaScript puro y sin Swagger, igual que el proyecto que deben desarrollar

---

Modelo de Datos

La base de datos debe contener las siguientes tablas organizadas por niveles de dependencia:

Nivel 1 - Tablas sin dependencias (sin claves foráneas)

Tabla: `roles`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	Campo único
`status`	boolean	Campo por defecto `true`

Tabla: `categories`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	Campo único
`status`	boolean	Campo por defecto `true`

Tabla: `warehouses`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	Campo único
`status`	boolean	Campo por defecto `true`

Nivel 2 - Tablas que dependen de Nivel 1

Tabla: `users`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	-
`email`	string	Campo único
`password`	string	-
`role_id`	number	Clave foránea a `roles.id`
`status`	boolean	Campo por defecto `true`

Tabla: `areas`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	Campo único
`warehouse_id`	number	Clave foránea a `warehouses.id`
`status`	boolean	Campo por defecto `true`

Nivel 3 - Tablas que dependen de Nivel 2

Tabla: `products`

Campo	Tipo	Restricciones
`id`	number	Autoincrementable, clave primaria
`name`	string	-
`price`	number	-
`quantity`	number	-
`category_id`	number	Clave foránea a `categories.id`
`area_id`	number	Clave foránea a `areas.id`
`status`	boolean	Campo por defecto `true`
Constraint único compuesto	-	`(name, area_id)` - El nombre debe ser único por área

---

Explicación de Niveles

Los niveles indican el grado de dependencia entre tablas:

- **Nivel 1**: Tablas independientes que no tienen claves foráneas. Pueden crearse sin depender de otras tablas.
- **Nivel 2**: Tablas que tienen al menos una clave foránea hacia una tabla de Nivel 1. Requieren que existan registros en las tablas de Nivel 1 antes de poder crear registros.
- **Nivel 3**: Tablas que tienen claves foráneas hacia tablas de Nivel 2. Requieren que existan registros en las tablas de Nivel 2 (y por ende, también en Nivel 1) antes de poder crear registros.

Ejemplo de dependencias:

- `users` (Nivel 2) depende de `roles` (Nivel 1)
- `areas` (Nivel 2) depende de `warehouses` (Nivel 1)
- `products` (Nivel 3) depende de `categories` (Nivel 1) y `areas` (Nivel 2)
- `products` también depende indirectamente de `warehouses` a través de `areas`

Requisitos Funcionales

Endpoints del Backend (Obligatorios)

Para cada entidad (`roles` , `categories` , `warehouses` , `users` , `areas` , `products`) se deben implementar **todos** los siguientes endpoints CRUD en el backend:

- `GET /api/<entidad>` → Listar todos los registros
- `GET /api/<entidad>/:id` → Obtener un registro por su ID
- `POST /api/<entidad>` → Crear un nuevo registro
- `PUT /api/<entidad>/:id` → Actualizar un registro existente
- `DELETE /api/<entidad>/:id` → Eliminar un registro (soft delete)

Endpoints del Frontend (Obligatorios)

En el frontend, solo se requiere consumir el endpoint de **listar** para cada entidad:

- `GET /api/<entidad>` → Listar todos los registros (para todas las entidades)

Nota: El frontend solo necesita consumir el endpoint de listar. Los demás endpoints (POST, PUT, DELETE) deben estar implementados en el backend pero no es necesario consumirlos desde el frontend.

Consideraciones

- El DELETE debe ser un **soft delete** (marcar `status = false` en lugar de eliminar físicamente)
- Las validaciones de datos deben implementarse usando **express-validator** en la capa de validadores
- Se debe validar la existencia de claves foráneas antes de crear/actualizar registros
- **Backend**: Implementar todos los endpoints CRUD
- **Frontend**: Solo consumir el endpoint de listar (GET) para cada entidad

---

Regla Obligatoria: Estructura de Respuestas

TODOS los endpoints DEBEN seguir esta estructura de respuesta JSON de forma obligatoria y sin excepción.

Importante: Esta estructura es obligatoria porque el frontend que se proporcionará para consumir la API ya está configurado para esperar esta estructura de respuesta. El repositorio del frontend de referencia es: https://github.com/Zedmous/iujo-frontend-foc-2025-2

Los estudiantes deberán adaptar ese frontend para consumir su API, por lo que **deben seguir esta estructura exacta** para garantizar la compatibilidad.

Estructura en Servicios

Los servicios deben retornar un objeto con la siguiente estructura:

```
return {
  message: "Mensaje descriptivo",
  status: 200,  // Código HTTP
  data: {
    // Datos de la respuesta
  }
};
```

Ejemplo en servicio:

```
// En servicios (ej: users.servicios.js)
return {
  message: `Record found`,
  status: 200,
  data: {
    user,
  },
};
```

Estructura en Controladores

Los controladores deben recibir la respuesta del servicio. El `status` se usa **únicamente** para establecer el código HTTP con `res.status()`, pero **NO se incluye en el JSON de respuesta**. Solo se retorna `{ message, data }`:

```
// En controladores (ej: users.controladores.js)
const { message, status, data } = await UserServices.getById(id);
res.status(status).json({ message, data });  // status solo para el código HTTP
```

Ejemplo completo en controlador:

```
getOne = async (req, res) => {
  const { id } = req.params;
  const { message, status, data } = await UserServices.getById(id);
  return res.status(status).json({ message, data });  // JSON solo contiene message y data
};
```

Respuesta JSON final al cliente:

```
{
  "message": "Record found",
  "data": {
    "user": { ... }
  }
}
```

Nota: El código HTTP se establece con `res.status(status)`, pero el JSON de respuesta **NO incluye el campo `status`**.

⚠️ **ADVERTENCIA**: Cualquier desviación de esta estructura hará que el frontend no funcione correctamente. Esta regla es **obligatoria y sin excepción**.

Códigos de Estado HTTP

- `200` - Operación exitosa (GET, PUT)
- `201` - Recurso creado exitosamente (POST)
- `204` - Recurso eliminado exitosamente (DELETE)
- `400` - Error de validación o solicitud incorrecta
- `404` - Recurso no encontrado
- `500` - Error interno del servidor

Nota importante: Esta estructura es **obligatoria** y debe aplicarse a **todos los endpoints** sin excepción.

Requisitos Técnicos

Base de Datos

- **Sistema**: PostgreSQL
- **ORM**: Prisma
- **Migraciones**: Deben crearse migraciones de Prisma para todas las tablas
- **Schema**: **IMPORTANTE**: Todas las tablas deben definirse en el archivo `prisma/schema.prisma`. Este es el archivo donde se especifican todos los modelos de la base de datos usando la sintaxis de Prisma.

Validaciones

- Implementar validaciones usando **express-validator**
- Crear archivos de validadores en la carpeta `src/validators/`
- Validar:
  - Campos requeridos
  - Formatos de datos (email, números, etc.)
  - Unicidad de campos únicos
  - Existencia de claves foráneas
  - Reglas de negocio específicas

Manejo de Errores

- Implementar manejo centralizado de errores
- Retornar códigos HTTP apropiados siguiendo la estructura obligatoria de respuestas
- Mensajes de error descriptivos y consistentes

Entregables

Repositorios

Cada equipo debe entregar **2 repositorios en GitHub**:

1. **Repositorio Backend**: Conteniendo toda la API REST desarrollada
2. **Repositorio Frontend**: Conteniendo el frontend clonado y adaptado para consumir la API

Requisitos de los Repositorios

Repositorio Backend

1. **Código fuente completo** con la estructura por capas
2. **Schema de Prisma** (`prisma/schema.prisma`) con todos los modelos
3. **Migraciones de Prisma** aplicadas y funcionando
4. **Endpoints CRUD** implementados para todas las entidades
5. **Validadores** implementados con express-validator
6. **README.md** con:
   - Instrucciones de instalación
   - Configuración paso a paso
   - Cómo levantar el proyecto
   - Variables de entorno necesarias
   - Cómo ejecutar migraciones
   - Documentación de cómo funciona el proyecto
7. **Archivo** `.env.example` con las variables de entorno necesarias
8. **Diagrama de arquitectura** del proyecto en formato imagen (usando Draw.io)
   - Debe mostrar la estructura por capas
   - Debe mostrar el flujo de datos entre capas
   - Debe incluir la relación con la base de datos
   - Guardar como imagen (PNG, JPG o SVG) en el repositorio

Repositorio Frontend

1. **Código fuente completo** del frontend adaptado
2. **Consumo de endpoints**: Implementar el consumo del endpoint de listar (GET) para todas las entidades:
   - `GET /api/roles`
   - `GET /api/categories`
   - `GET /api/warehouses`
   - `GET /api/users`
   - `GET /api/areas`
   - `GET /api/products`
3. **README.md** con:
   - Instrucciones de instalación
   - Configuración paso a paso
   - Cómo levantar el proyecto
   - Variables de entorno necesarias
   - Documentación de cómo funciona el proyecto
4. **Archivo** `.env.example` con las variables de entorno necesarias

Nota: El frontend solo debe consumir el endpoint de listar. Los demás endpoints (POST, PUT, DELETE) deben estar implementados en el backend pero no es necesario consumirlos desde el frontend.

Requisitos de Colaboración en GitHub

- **Mínimo 2 usuarios de GitHub** por equipo deben tener commits en ambos repositorios
- Los commits deben estar distribuidos entre los miembros del equipo
- Se evaluará la participación activa de todos los miembros del equipo en el desarrollo

---

Criterios de Evaluación

- **Estructura del proyecto**: Organización clara por capas, siguiendo la estructura de carpetas especificada
- **Funcionalidad**: CRUD completo para todas las entidades
- **Validaciones**: Implementación correcta de validaciones con express-validator
- **Relaciones**: Manejo correcto de claves foráneas y dependencias entre tablas
- **Calidad del código**: Uso adecuado de controladores, servicios y validadores
- **Manejo de errores**: Respuestas HTTP apropiadas y mensajes claros
- **Documentación**: README completo y claro

Notas Importantes

- El proyecto debe usar **módulos ES** (`"type": "module"` en `package.json`)
- **NO se debe usar TypeScript**, solo JavaScript puro
- La base de datos debe ser **PostgreSQL**
- Se debe usar **Prisma** como ORM
- Las validaciones deben estar en la capa de **validators**, no en servicios
- El soft delete debe implementarse marcando `status = false` y `deleted_at = new Date()`
- Seguir la estructura de carpetas y arquitectura por capas especificada

---

Repositorios de Referencia - Ejemplos de Conexión Frontend-Backend

Se proporcionan dos repositorios que sirven como **ejemplos de referencia** de cómo conectar el frontend con el backend. Ambos proyectos están adaptados y funcionando como ejemplo de uso de conexión.

Frontend de Referencia (Obligatorio)

Repositorio: https://github.com/Zedmous/iujo-frontend-foc-2025-2

⚠️ **OBLIGATORIO**: Este frontend es **obligatorio** y debe ser clonado y adaptado por los estudiantes.

Este repositorio contiene:

- Interfaz desarrollada en Vite + TypeScript
- Ejemplo funcional de consumo de API usando Axios (solo la entidad `test` como ejemplo)
- Configuración de variables de entorno para la URL del backend
- Estructura lista para adaptar y consumir la API desarrollada

Tarea obligatoria: Los estudiantes deben clonar este frontend y adaptarlo para consumir todas las entidades establecidas en el enunciado (`roles` , `categories` , `warehouses` , `users` , `areas` , `products`). El frontend solo incluye la entidad `test` como ejemplo de referencia.

Backend de Referencia Avanzado (Opcional - Solo para Consulta)

Repositorio: https://github.com/Zedmous/iujo-backend-foc-2025-2

⚠️ **IMPORTANTE**: Este repositorio es **solo para consulta y referencia avanzada**, NO debe replicarse directamente porque tiene diferencias importantes:

Diferencias con el proyecto a desarrollar:

- Desarrollado con **TypeScript** (el proyecto debe ser JavaScript puro)
- Incluye **Swagger** para documentación (el proyecto NO debe incluir Swagger)
- Tiene configuración de **Docker** como base (opcional, no requerido)

Contenido útil de este repositorio:

- Ejemplos de **paginación** en endpoints
- Implementaciones avanzadas de validaciones
- Ejemplos de estructura de respuestas
- Ejemplos de conexión frontend-backend

Uso del repositorio avanzado:

- **Solo consulta**: Revisar ejemplos de paginación y otras funcionalidades avanzadas
- **NO replicar**: No copiar la estructura completa porque usa TypeScript y Swagger
- **Referencia limitada**: Usar únicamente como referencia para entender conceptos avanzados como paginación, pero implementarlos en JavaScript puro

Uso de los Repositorios de Referencia

Resumen:

1. **Repositorio principal (obligatorio)**: iujo-clase-foc-backend-2025-2 - JavaScript puro, sin Swagger - **Este es el que deben replicar**
2. **Repositorio avanzado (consulta)**: iujo-backend-foc-2025-2 - TypeScript, con Swagger - **Solo para ver ejemplos avanzados como paginación**
3. **Frontend (obligatorio)**: iujo-frontend-foc-2025-2 - **Debe ser clonado y adaptado** para consumir todas las entidades

Guía de uso:

- **Replicar**: El repositorio de clase (JavaScript puro, sin Swagger) para desarrollar el backend
- **Consultar**: El repositorio avanzado solo para ver ejemplos de paginación y funcionalidades avanzadas
- **Clonar y adaptar (obligatorio)**: El frontend debe ser clonado y adaptado para consumir todas las entidades establecidas (`roles` , `categories` , `warehouses` , `users` , `areas` , `products`)

Tarea Obligatoria del Frontend

Los estudiantes **DEBEN**:

1. **Clonar** el repositorio del frontend: https://github.com/Zedmous/iujo-frontend-foc-2025-2
2. **Adaptar** el frontend para consumir **solo el endpoint de listar (GET)** para todas las entidades establecidas:
   - `roles` - `GET /api/roles`
   - `categories` - `GET /api/categories`
   - `warehouses` - `GET /api/warehouses`
   - `users` - `GET /api/users`
   - `areas` - `GET /api/areas`
   - `products` - `GET /api/products`
3. **Configurar** la URL base de la API en las variables de entorno del frontend
4. **Implementar** las vistas y servicios necesarios para consumir únicamente el endpoint de listar de cada entidad
5. **Verificar** que todas las vistas de listado funcionen correctamente con los endpoints implementados en su API

Importante:

- En el **frontend** solo se requiere consumir el endpoint de **listar (GET)** para cada entidad

- En el **backend** sí se deben implementar **todos los endpoints CRUD** (GET, POST, PUT, DELETE) para todas las entidades
- El frontend actual solo tiene implementado el consumo de la entidad `test` como ejemplo. Los estudiantes deben extenderlo para incluir todas las entidades establecidas, pero solo para el endpoint de listar

Importante:

- El frontend ya está configurado para esperar la estructura de respuestas especificada en la sección "Regla Obligatoria: Estructura de Respuestas"
- El backend de ejemplo muestra cómo implementar correctamente esa estructura
- Por esta razón, **es crítico** que todos los endpoints sigan exactamente esa estructura sin excepciones, de lo contrario el frontend no funcionará correctamente
- Ambos repositorios de referencia están adaptados y funcionando como ejemplo de conexión, úsenlos como guía

---

Orden de Implementación Sugerido

1. Configurar Prisma y crear el schema con todas las tablas
2. Crear y aplicar migraciones
3. Implementar servicios y controladores para tablas de **Nivel 1** (`roles` , `categories` , `warehouses`)
4. Implementar servicios y controladores para tablas de **Nivel 2** (`users` , `areas`)
5. Implementar servicios y controladores para tablas de **Nivel 3** (`products`)
6. Implementar validadores con express-validator para todas las entidades
7. Probar todos los endpoints y validaciones
8. Clonar y adaptar el frontend de referencia para consumir la API desarrollada
9. Verificar que todas las funcionalidades del frontend trabajen correctamente con la API

Fecha de entrega: Antes del domingo 30/11/2025
Autor: Zedmous