🛡️ Aura Auth Service

Este microservicio es el pilar de la seguridad y gestión de usuarios para la plataforma Aura. Construido con **Node.js**, **Express** y **Prisma**, proporciona un sistema de autenticación robusto, seguro y escalable basado en **JSON Web Tokens (JWT)**.

✨ Características Principales

- **Registro de Usuarios**: Creación de nuevas cuentas con contraseñas hasheadas.
- **Autenticación Segura**: Inicio de sesión mediante email y contraseña con generación de JWT.
- **Gestión de Perfiles**: Endpoints para que los usuarios vean y actualicen su información.
- **Control de Acceso Basado en Roles (RBAC)**: Middleware para proteger rutas según el rol del usuario (ej. `admin`, `user`).
- **Seguridad Reforzada**: Implementa **Helmet**, **CORS**, validación de entradas y sanitización para prevenir vulnerabilidades comunes.
- **Despliegue Automatizado**: Incluye un script para configurar el entorno y la base de datos en servidores Ubuntu.

🚀 Stack Tecnológico

- **Backend**: Node.js, Express.js
- **Base de Datos**: PostgreSQL
- **ORM**: Prisma
- **Autenticación**: JSON Web Tokens (JWT)
- **Seguridad**: Bcrypt.js (hashing), Helmet (cabeceras de seguridad), CORS
- **Validación**: express-validator, validator.js
- **Logging**: Morgan
- **Entorno**: Docker, Nginx (como parte del proyecto `aura_server`)

📁 Estructura del Proyecto

Archivo/Directorio	Descripción
`index.js`	Punto de entrada principal. Configura el servidor Express, los middlewares y las rutas.

Archivo/Directorio	Descripción
`deploy-auth-service.sh`	Script de despliegue automatizado para configurar el entorno en un servidor Ubuntu.
`package.json`	Define los metadatos del proyecto, scripts (`dev` , `start`) y dependencias.
`.env`	Archivo de configuración para variables de entorno (no versionado).
`prisma/schema.prisma`	Define el esquema de la base de datos y los modelos de datos para Prisma.
`src/routes/authRoutes.js`	Define todas las rutas de la API (`/register` , `/login` , etc.) y asocia los middlewares y controladores.
`src/controllers/authController.js`	Contiene la lógica de negocio para cada endpoint (interacción con la base de datos, etc.).
`src/middlewares/`	Directorio para los middlewares personalizados.
`src/middlewares/authMiddleware.js`	Middlewares para verificar tokens JWT y autorizar roles.
`src/middlewares/validationMiddleware.js`	Middlewares para validar y sanitizar los datos de entrada de las peticiones.

📝 Modelos de Datos

`roles`

Almacena los roles de usuario disponibles en el sistema.

Campo	Tipo	Descripción
`id_role`	`SERIAL` (PK)	Identificador único del rol.
`role_name`	`VARCHAR(50)` (UNIQUE, NOT NULL)	Nombre del rol (ej. `'admin'` , `'user'`).
`created_at`	`TIMESTAMP WITH TIME ZONE`	Marca de tiempo de creación del rol.

`users`

Contiene la información esencial de autenticación de los usuarios.

Campo	Tipo	Descripción
`user_id`	`UUID` (PK)	Identificador único universal del usuario.

Campo	Tipo	Descripción
`username`	`VARCHAR(100)` (UNIQUE)	Nombre de usuario único.
`email`	`VARCHAR(100)` (UNIQUE)	Correo electrónico único del usuario.
`password_hash`	`VARCHAR(255)`	Hash de la contraseña del usuario (generado con Bcrypt).
`id_role`	`INTEGER` (FK a `roles`)	ID del rol al que pertenece el usuario.
`created_at`	`TIMESTAMP WITH TIME ZONE`	Marca de tiempo de creación del usuario.

`user_profiles`

Almacena información adicional y extensible del perfil del usuario.

Campo	Tipo	Descripción
`user_id`	`UUID` (PK, FK a `users`)	Referencia al `user_id` de la tabla `users`.
`interests`	`TEXT[]`	Un array de strings con los intereses del usuario.
...	...	Futuros campos de perfil como `fullname`, `bio`, etc.

🛠️ Configuración y Ejecución

1. Configuración del Entorno

Crea un archivo `.env` en la raíz del proyecto (`auth-service/`) con las siguientes variables:

```
# URL de conexión a la base de datos PostgreSQL
DATABASE_URL="postgresql://aura_auth_user:aurapassword@localhost:5432/aur

# Clave secreta para firmar los tokens JWT. Debe ser larga y compleja.
JWT_SECRET="tu_clave_secreta_muy_larga_y_segura_aqui"

# Puerto en el que correrá el servicio
PORT=3001
```

2. Despliegue Automatizado (Recomendado)

El script `deploy-auth-service.sh` automatiza toda la configuración en un servidor Ubuntu.

Dar permisos de ejecución al script:

```
chmod +x deploy-auth-service.sh
```

Ejecutar el script:

```
./deploy-auth-service.sh
```

Este script se encargará de:

- Instalar **Node.js** y **PostgreSQL** si no están presentes.
- Crear el usuario y la base de datos `aura_auth_db`.
- Instalar las dependencias del proyecto con `npm install`.
- Aplicar las migraciones de Prisma para crear las tablas (`npx prisma migrate deploy`).
- Insertar los roles iniciales (`'admin'`, `'user'`).

3. Ejecución del Servicio

Una vez configurado el entorno, puedes iniciar el servidor:

- Para desarrollo (con recarga automática):

  ```
  npm run dev
  ```

- Para producción:

  ```
  npm start
  ```

El servicio estará escuchando en `http://localhost:3001`.

🌐 Endpoints de la API

La URL base para todos los endpoints es `http://98.95.86.245:3001/api/auth`.

Método	Endpoint	Descripción	Autenticación	Autorización	Cuerpo (Body) de Ejemplo	Respuesta Exitosa (2xx)

Método	Endpoint	Descripción	Autenticación	Autorización	Cuerpo (Body) de Ejemplo	Respuesta Exitosa (2xx)
`POST`	`/register`	Registra un nuevo usuario.	No	Cualquiera	`{"username": "nuevo_usuario", "email": "nuevo@email.com", "password": "Password123!"}`	`201 Created` con datos del usuario y token JWT.
`POST`	`/login`	Inicia sesión y obtiene un token JWT.	No	Cualquiera	`{"email": "nuevo@email.com", "password": "Password123!"}`	`200 OK` con el token JWT.
`GET`	`/profile`	Obtiene los datos del usuario autenticado.	JWT Requerido	Usuario	N/A	`200 OK` con los datos del perfil del usuario.
`GET`	`/users`	Obtiene una lista de todos los usuarios.	JWT Requerido	Admin	N/A	`200 OK` con un array de todos los usuarios.
`POST`	`/user/interests`	Guarda o actualiza los intereses de un usuario.	JWT Requerido	Usuario	`{"interests": ["tecnologia", "viajes", "musica"]}`	`200 OK` con un mensaje de éxito y la lista de intereses guardados.

👨‍💻 Tutorial de Uso

1. Probar la API con Postman

A continuación se detalla cómo probar cada endpoint usando la IP del servidor `98.95.86.245`.

Configuración Inicial:

- **Crear un Entorno en Postman**: Ve a `Environments > Create Environment`.
  - Nómbralo `"Aura API"`.

    - Añade una variable `baseURL` con el valor `http://98.95.86.245:3001/api/auth`.
    - Añade una variable `jwtToken` y déjala vacía por ahora.
- **Selecciona el Entorno**: Asegúrate de que `"Aura API"` esté seleccionado en la esquina superior derecha.

Paso 1: Registrar un Usuario

- **Método**: `POST`
- **URL**: `/register`
- **Body > raw > JSON**:

```
{
    "username": "testuser_postman",
    "email": "test.postman@example.com",
    "password": "SecurePassword123!"
}
```

- **Tests** (Pestaña para guardar el token automáticamente):

```
const response = pm.response.json();
if (response.token) {
    pm.environment.set("jwtToken", response.token);
    console.log("Token guardado en el entorno.");
}
```

- **Resultado**: Deberías recibir una respuesta `201 Created` con el token. El script en la pestaña "Tests" guardará este token en tu variable de entorno `jwtToken`.

Paso 2: Iniciar Sesión

- **Método**: `POST`
- **URL**: `/login`
- **Body > raw > JSON**:

```
{
    "email": "test.postman@example.com",
    "password": "SecurePassword123!"
}
```

- **Tests**: Usa el mismo script que en el registro para actualizar el token si es necesario.
- **Resultado**: Recibirás una respuesta `200 OK` con un nuevo token.

Paso 3: Obtener el Perfil (Ruta Protegida)

- **Método**: `GET`
- **URL**: `/profile`
- **Authorization**:
  - **Tipo**: `Bearer Token`
  - **Token**: (Postman reemplazará esto con el valor de tu variable de entorno).
- **Resultado**: Recibirás un `200 OK` con los datos del usuario que acabas de registrar.

Paso 4: Actualizar Intereses (Ruta Protegida)

- **Método**: `POST`
- **URL**: `/user/interests`
- **Authorization**:
  - **Tipo**: `Bearer Token`
  - **Token**:
- **Body > raw > JSON**:

```
{
    "interests": ["programacion", "inteligencia artificial", "viajes'
}
```

- **Resultado**: Recibirás un `200 OK` con un mensaje de éxito y la lista de intereses que enviaste.

2. Consumir la API desde un Cliente (Ejemplo con JavaScript fetch)

Este es un ejemplo básico de cómo un cliente frontend podría interactuar con la API.

```
// La URL base de tu API
const API_URL = 'http://98.95.86.245:3001/api/auth';

// Función para registrar un nuevo usuario
async function registerUser(username, email, password) {
    try {
        const response = await fetch(`${API_URL}/register`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ username, email, password }),
        });

        const data = await response.json();
```

```
        if (!response.ok) {
            throw new Error(data.message || 'Error en el registro');
        }

        console.log('Registro exitoso:', data);
        // Guardar el token en localStorage para futuras peticiones
        localStorage.setItem('authToken', data.token);
        return data;
    } catch (error) {
        console.error('Error en registerUser:', error);
    }
}

// Función para iniciar sesión
async function loginUser(email, password) {
    try {
        const response = await fetch(`${API_URL}/login`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ email, password }),
        });

        const data = await response.json();

        if (!response.ok) {
            throw new Error(data.message || 'Credenciales inválidas');
        }

        console.log('Login exitoso:', data);
        // Guardar el token en localStorage
        localStorage.setItem('authToken', data.token);
        return data;
    } catch (error) {
        console.error('Error en loginUser:', error);
    }
}

// Función para obtener el perfil del usuario (petición autenticada)
async function getProfile() {
    const token = localStorage.getItem('authToken');
    if (!token) {
        console.error('No hay token de autenticación.');
```

```
        return;
    }

    try {
        const response = await fetch(`${API_URL}/profile`, {
            method: 'GET',
            headers: {
                'Authorization': `Bearer ${token}`,
            },
        });

        const data = await response.json();

        if (!response.ok) {
            throw new Error(data.message || 'No se pudo obtener el perfil
        }

        console.log('Perfil del usuario:', data.user);
        return data.user;
    } catch (error) {
        console.error('Error en getProfile:', error);
    }
}

// --- Ejemplo de uso ---
// 1. Registrar un usuario
// registerUser('web_client_user', 'web@example.com', 'ClientPass123!');

// 2. O iniciar sesión para obtener un token
// loginUser('web@example.com', 'ClientPass123!').then(() => {
//     // 3. Una vez que tenemos el token, podemos obtener el perfil
//     getProfile();
// });
```

🔒 Principios de Seguridad Implementados

Este microservicio sigue las mejores prácticas de seguridad para proteger los datos de los usuarios.

1. Transacciones Seguras con Prisma

Prisma garantiza la **atomicidad** en las operaciones de escritura que involucran relaciones. En el endpoint de registro, la creación del usuario y la conexión (`connect`) con su rol se ejecutan dentro de

una única transacción. Esto asegura que si la conexión con el rol falla, la creación del usuario también se revierte, manteniendo la consistencia e integridad de los datos.

- **Librería**: `@prisma/client`
- **Implementación**: `src/controllers/authController.js`

```
const newUser = await prisma.user.create({
    data: {
        username,
        email,
        password_hash,
        role: {
            // Esta operación anidada se ejecuta en la misma transacción
            connect: { role_name: 'user' }
        }
    },
    // ...
});
```

2. Validación Rigurosa en el Servidor

Se implementan múltiples capas de validación para proteger los endpoints y la base de datos.

- **Autenticidad y Permisos**: Se verifica la validez de cada token JWT y se restringe el acceso a endpoints específicos (ej. solo `admin`) usando middlewares (`verifyToken`, `authorizeRole`).
- **Consistencia de Datos**: Antes de crear un usuario, se comprueba que el email y username no estén ya en uso para evitar duplicados.
- **Integridad del Token**: La firma del JWT se valida para asegurar que no ha sido manipulado.
- **Implementación**: `src/controllers/authController.js`

```
// Validación de Consistencia: Verificar si el usuario o email ya exister
const existingUser = await prisma.user.findUnique({ where: { email } });
if (existingUser) {
    return res.status(409).json({ message: 'User with this email already
}
const existingUsername = await prisma.user.findUnique({ where: { username
if (existingUsername) {
    return res.status(409).json({ message: 'Username is already taken.' }
}
```

3. Validación de Formato y Patrones

Se utiliza **express-validator** para asegurar que todos los datos de entrada cumplan con las reglas de negocio antes de ser procesados.

- **Tipos de Datos**: Se valida que campos como email y password tengan el formato y tipo correctos (`isEmail` , `isLength`).
- **Contraseñas Fuertes**: Se exige una combinación de mayúsculas, minúsculas, números y símbolos.
- **Nombres de Usuario**: Se valida un patrón (`/^[a-zA-Z0-9_]+$/`) para que solo contenga caracteres permitidos.
- **Implementación**: `src/middlewares/validationMiddleware.js` (Ejemplo de uso en rutas)

```
// En `src/routes/authRoutes.js`, se aplican las validaciones antes del c
const { registerValidation, loginValidation } = require('../middlewares/v

router.post('/register', registerValidation, authController.register);
router.post('/login', loginValidation, authController.login);
```

4. Sanitización de Entradas

Para prevenir ataques como **XSS (Cross-Site Scripting)**, todas las entradas son sanitizadas.

- **Escapado de Caracteres**: Se usa `escape()` para convertir caracteres HTML (`<` , `>` , `&` , etc.) en entidades, neutralizando scripts maliciosos.
- **Normalización**: Se normalizan los correos electrónicos (`normalizeEmail()`) para estandarizar su formato y evitar evasiones.
- **Implementación**: `src/middlewares/validationMiddleware.js`

```
// Ejemplo de regla de validación y sanitización en `validationMiddleware
const { body } = require('express-validator');
const registerValidation = [
    body('email').isEmail().normalizeEmail(),
    body('username').trim().escape(),
```

```
    // ... más validaciones
];
```

5. Uso de Librerías Seguras

La seguridad se delega en librerías auditadas y mantenidas por la comunidad.

- **ORM (Prisma)**: Previene ataques de inyección SQL al parametrizar todas las consultas a la base de datos de forma automática.
- **Validación (Express-validator)**: Proporciona un conjunto de herramientas robustas para validar y sanitizar datos de manera segura.
- **Implementación**: `src/controllers/authController.js`

```
// Prisma parametriza automáticamente el valor de 'email' para prevenir i
const user = await prisma.user.findUnique({
    where: { email }, // El valor de 'email' es manejado de forma segura
    include: { role: true }
});
```

6. Gestión Segura de Errores

Los errores se manejan de forma controlada para no exponer información sensible.

- **Mensajes Genéricos**: De cara al cliente, los errores (ej. "Invalid credentials") son intencionadamente ambiguos para no revelar si un usuario existe o no.
- **No Exposición de Stack Traces**: Los errores internos se registran en el servidor, pero nunca se envían los detalles completos al cliente.
- **Implementación**: `src/controllers/authController.js` y `index.js`

```
// Mensaje genérico en el login para no revelar información
const isMatch = await bcrypt.compare(password, user.password_hash);
if (!isMatch) {
    return res.status(401).json({ message: 'Invalid credentials.' });
}

// Middleware global en index.js para capturar errores no controlados
app.use((err, req, res, next) => {
    console.error(err.stack); // Loguea el error completo en el servidor
```

```
    res.status(500).json({ message: 'Something broke!' }); // Envía respu
});
```

7. Operación Robusta de Creación/Actualización de Perfil

El nuevo endpoint `/user/interests` permite a los usuarios guardar sus intereses. La lógica del controlador utiliza una operación **upsert** de Prisma, que es especialmente robusta y eficiente.

- **Atomicidad**: `upsert` combina una actualización (`update`) y una inserción (`create`) en una sola operación atómica a nivel de base de datos.
- **Idempotencia**: Si el perfil del usuario ya existe, se actualiza. Si no existe, se crea. Esto simplifica enormemente la lógica del cliente y del servidor, ya que no es necesario verificar primero si el perfil existe antes de decidir si crear o actualizar.
- **Implementación**: `src/controllers/authController.js`

```
// En `src/controllers/authController.js`
const userProfile = await prisma.userProfile.upsert({
    where: { user_id: userId }, // Condición para buscar el perfil
    update: {
        interests: interests, // Qué actualizar si se encuentra
    },
    create: {
        user_id: userId,      // Qué crear si no se encuentra
        interests: interests,
    },
});
```