Guía de Actividades Práctico-Experimentales Nro. 003

1. Datos Generales

Asignatura	Desarrollo Basado en Plataformas
Ciclo	5 A
Unidad	1
Resultado de aprendizaje de la unidad	Discute cómo los estándares Web influyen en el desarrollo de software, bajo los principios de solidaridad, transparencia, responsabilidad y honestidad.
Práctica Nro.	003
Nombre del Docente	Edison Leonardo Coronel Romero
Integrantes	Eberson Guayllas Wilmer Pardo Santiago Tamayo
Fecha	viernes 17 de octubre
Horario	07h30 – 10h30
Lugar	Aula 232
Tiempo planificado en el Sílabo	3 horas

2. Título:

Implementación del flujo de autenticación y autorización en el backend, aplicando mecanismos de seguridad (JWT u OAuth2), validaciones, CORS y principios OWASP Top 10, e incorporación de estos componentes al modelo C4.

3. Objetivo:

Configurar e implementar un mecanismo de autenticación y autorización seguro en el backend del proyecto.

Aplicar políticas CORS, validaciones y manejo de errores según buenas prácticas OWASP.

Documentar el proceso mediante pruebas Postman/Swagger y actualizar los diagramas C4 (Container y Component) reflejando los puntos de seguridad.

4. Materiales y herramientas:

- Computador con acceso a Internet.
- Framework backend (Django REST Framework / Express.js / Spring Boot).
- IDE (VS Code / IntelliJ IDEA).
- Git, GitKraken con flujo GitFlow.

- Postman o Swagger para pruebas.”.
- Equipos personales
- IDE: Visual Studio Code / IntelliJ IDEA.
- Git, GitKraken (flujo GitFlow), Postman o Swagger.
- Repositorio remoto en GitHub.

5. Procedimiento / Metodología

Para implementar un mecanismo de autenticacion y autorizacion en el backend, primeramente se realizó una revisión teórica de los mecanismos de seguridad (JWT y OAuth2), con el fin de seleccionar el método más adecuado para una API REST desarrollada en flask; JWT es un estándar que define un formato compacto y seguro para representar información entre dos partes como un objeto JSON firmado digitalmente, mientras que, OAuth2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a recursos protegidos en nombre de un usuario. Luego se procedió a configurar el entorno de trabajo, instalando las dependencias necesarias para el manejo de tokens y control de accesos

El desarrollo se centró en la creación de las rutas /auth/login y /auth/register, encargadas de la validación de tokens JWT, los cuales incluyen información como el identificador del usuario, su rol y el tiempo de expiración. Asimismo, se implementaron validaciones de entrada y políticas CORS que limitan los orígenes permitidos para las solicitudes, fortaleciendo la seguridad del sistema. Finalmente, se realizaron pruebas de funcionalidad con Postman, verificando el acceso a rutas protegidas y documentando las respuestas HTTP obtenidas.

6. Resultados:

- **Backend con flujo JWT u OAuth2 funcional.**

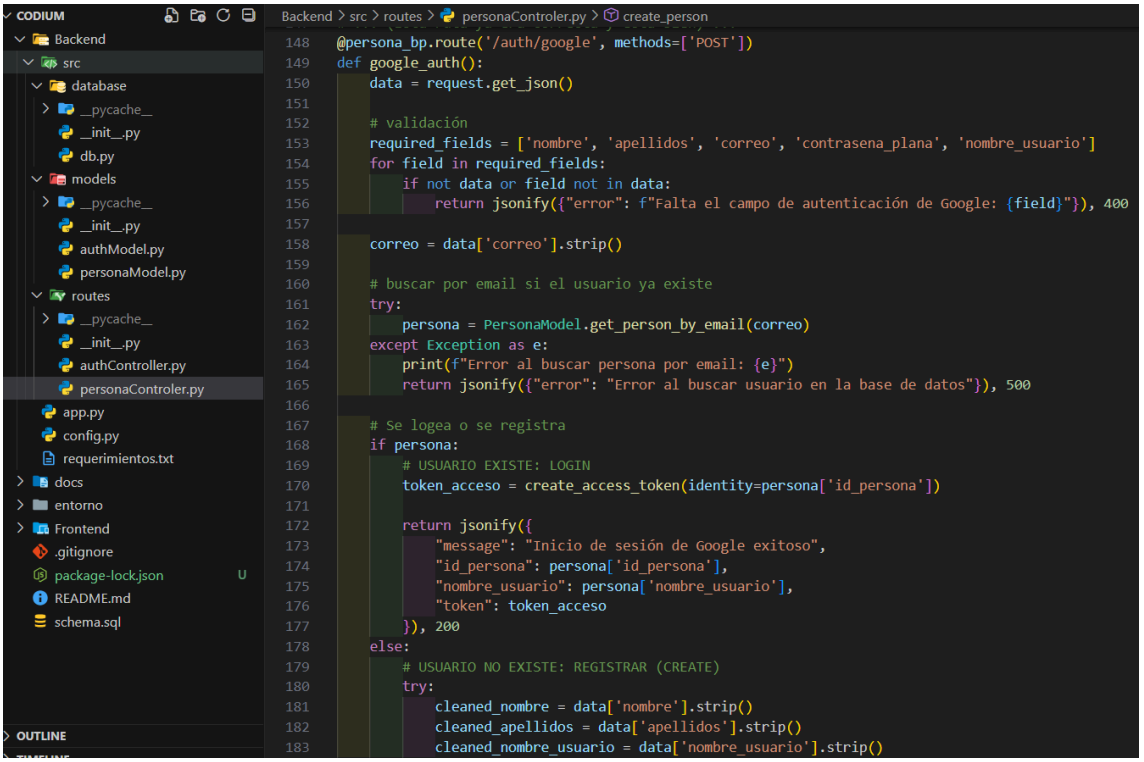
  Instalar dependencias necesarias para seguridad

```
(entorno) PS C:\Users\andre\OneDrive\Desktop\codium\Backend> pip show flask-jwt-extended
>>
Name: Flask-JWT-Extended
Version: 4.7.1
Summary: Extended JWT integration with Flask
Home-page: https://github.com/vimalloc/flask-jwt-extended
Author: Lily Acadia Gilbert
Author-email: lily.gilbert@hey.com
License: MIT
Location: C:\Users\andre\OneDrive\Desktop\codium\entorno\Lib\site-packages
Requires: Flask, PyJWT, Werkzeug
Required-by:
```

Crear rutas /auth/login y /auth/register.

Generar token JWT con exp, iat y roles de usuario.

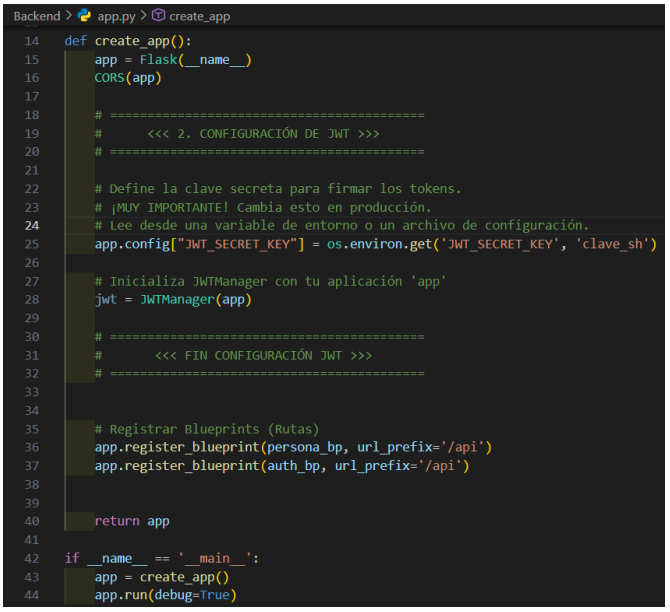
Crear middleware de verificación de token y roles (RBAC).




```python
@jwt_required()
def profile():
    print("llego a profile")
    # Si el token es inválido o no se provee (o es un Refresh Token),
    # Flask-JWT-Extended devolverá un error 401 o 422 automáticamente.

    # <<< 3. OBTENER EL ID DEL USUARIO DESDE EL TOKEN >>>
    # Esto es el 'identity' que pasamos en create_access_token()
    current_user_id = get_jwt_identity()

    try:
        # <<< 4. BUSCAR AL USUARIO EN LA BD CON ESE ID >>>
        persona = PersonaModel.get_persona_by_id(current_user_id)

        if not persona:
            return jsonify({"error": "Usuario no encontrado"}), 404

        # Opcional: No devolver información sensible
        if 'contrasena' in persona:
            del persona['contrasena']
        if 'token_refresco' in persona:
            del persona['token_refresco']

        # <<< 5. DEVOLVER LOS DATOS DEL PERFIL >>>
        return jsonify(persona), 200

    except Exception as e:
        print(f"Error en /profile: {e}")
        return jsonify({"error": "Error interno al obtener el perfil"}), 500
```

Se usa el decorador `@jwt_required()` en la ruta `/profile` (middleware de verificación).

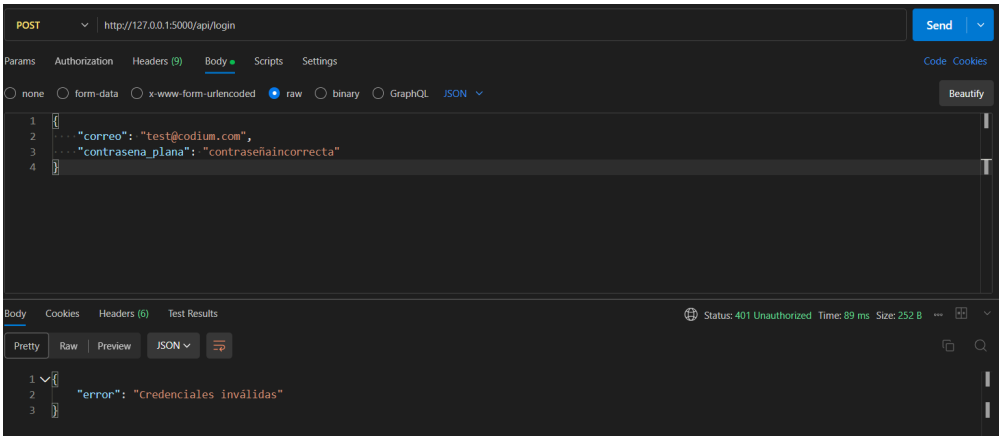
- **Políticas CORS y validaciones implementadas.**
  Definir orígenes permitidos y métodos HTTP.

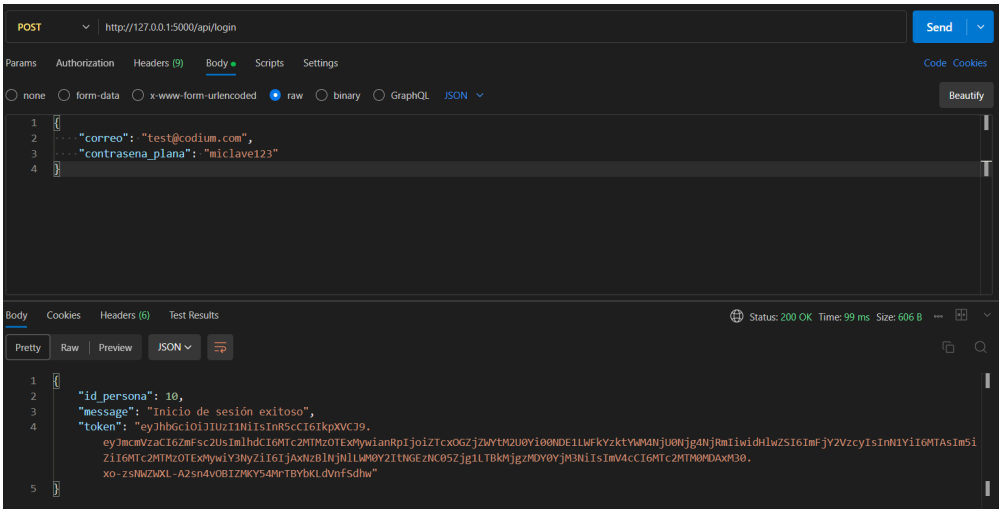


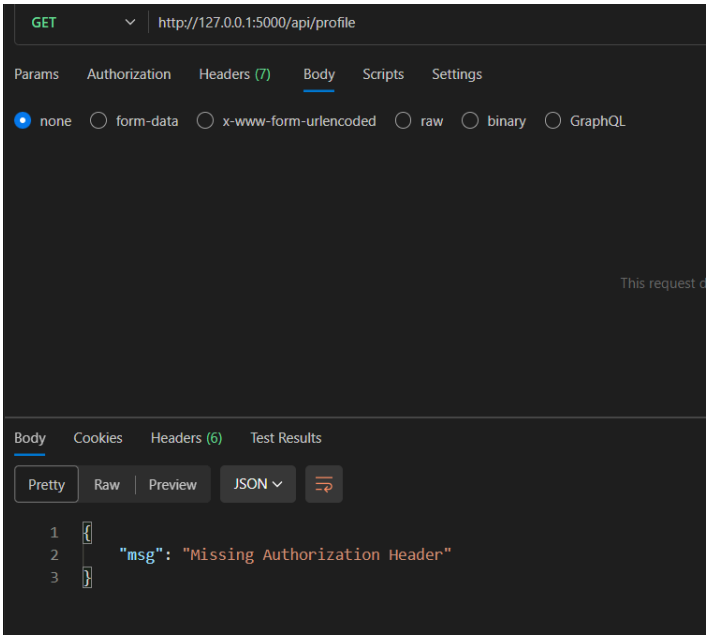
Agregar validación de entrada (request body y params).
Implementar manejo de errores uniforme (códigos HTTP y mensajes JSON).

```python
@persona_bp.route('/personas', methods=['POST'])
def create_person():
    data = request.get_json()
    # ... (Validaciones - sin cambios) ...
    if not data:
        return jsonify({"error": "No se proporcionaron datos"}), 400
    required_fields = ['nombre', 'apellidos', 'correo', 'contrasena_plana', 'nombre_usuario']
    for field in required_fields:
        if field not in data or data[field] is None or (isinstance(data[field], str) and data[field].strip() == ''):
            return jsonify({"error": f"Falta el campo requerido: {field}"}), 400
    correo = data['correo']
    if not isinstance(correo, str):
        return jsonify({"error": "Campo 'correo' debe ser texto (string)"}), 400
    correo = correo.strip()
    if '@' not in correo or '.' not in correo:
        return jsonify({"error": "Correo inválido: debe contener '@' y '.'"}), 400
    if not re.match(r'^[^@]+@[^@]+\.[^@]+$', correo):
        return jsonify({"error": "Correo inválido: formato incorrecto"}), 400
    nombre = data['nombre']
    apellidos = data['apellidos']
    if not isinstance(nombre, str):
        return jsonify({"error": "Campo 'nombre' debe ser texto (string)"}), 400
    if not isinstance(apellidos, str):
        return jsonify({"error": "Campo 'apellidos' debe ser texto (string)"}), 400
    nombre = nombre.strip()
    apellidos = apellidos.strip()
    nombre_pattern = r'^[A-Za-zÁÉÍÓÚáéíóúÑñ\s]+$'
    if not re.match(nombre_pattern, nombre):
        return jsonify({"error": "El nombre solo debe contener letras y espacios"}), 400
    if not re.match(nombre_pattern, apellidos):
        return jsonify({"error": "Los apellidos solo deben contener letras y espacios"}), 400
    # ... (Fin Validaciones) ...
```

- **Colección Postman/Swagger con pruebas de autenticación.**
  Probar login y rutas protegidas con Postman/Swagger.







- **Diagramas C4 actualizados mostrando componentes de seguridad.**
  Diagrama de contenedores

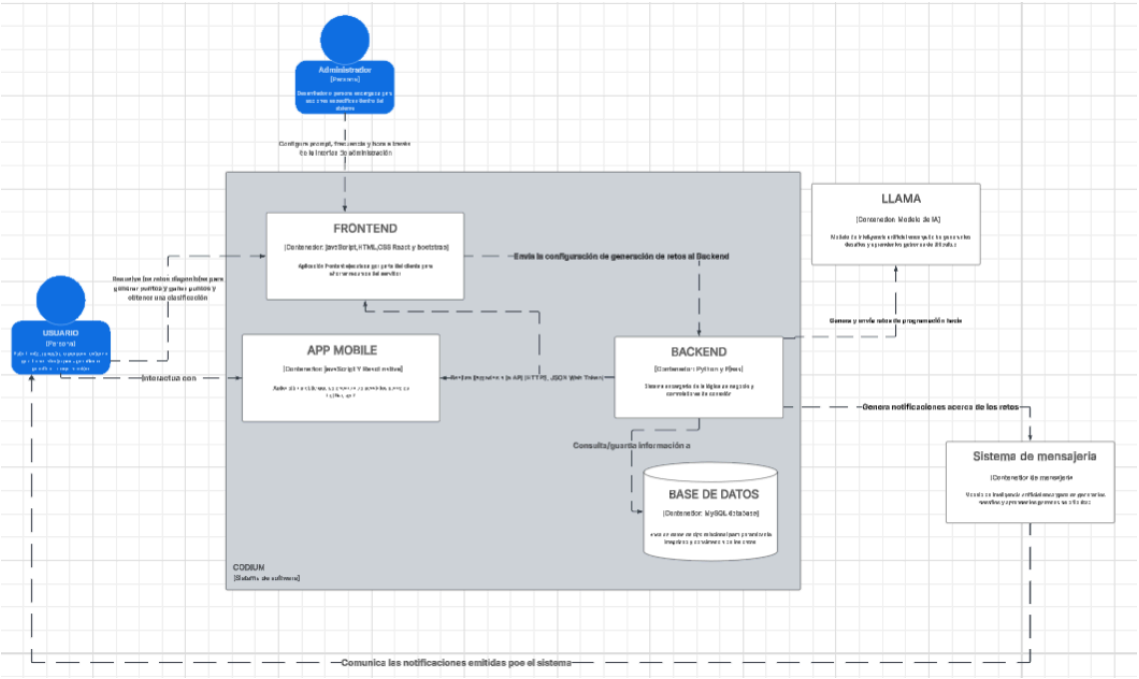
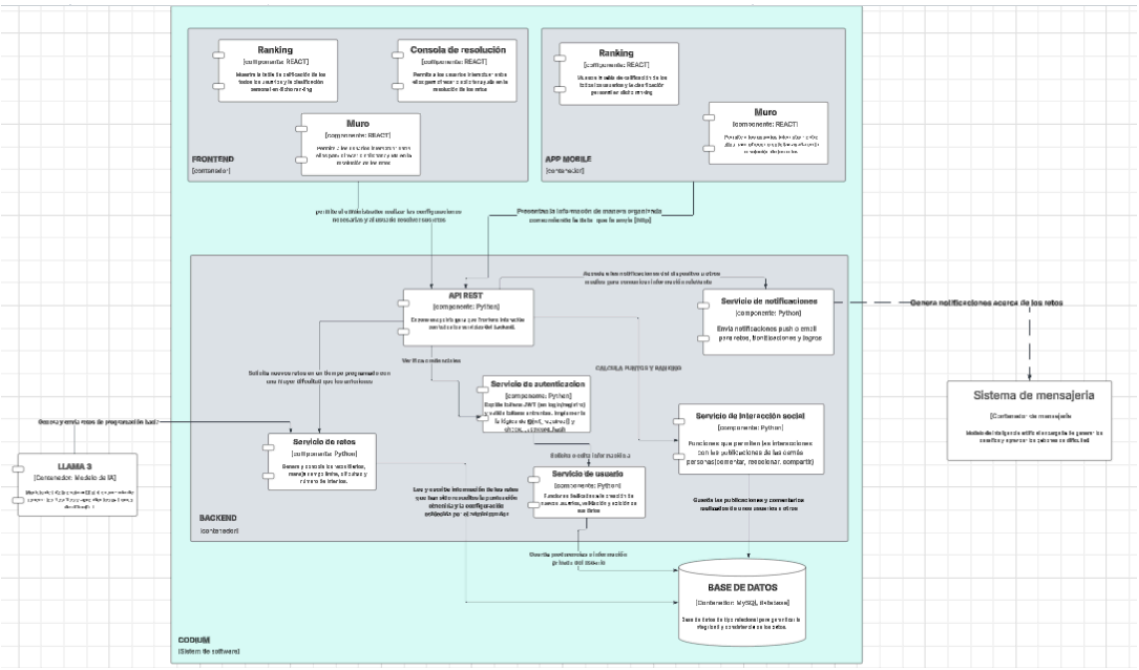


Diagrama de componentes



7. Preguntas de Control:

1. **¿Cuál es la diferencia fundamental entre autenticación y autorización dentro de un sistema backend?**

La autenticación es un proceso donde un sistema verifica la identidad de un usuario, es decir, realiza una comprobación de quien está accediendo al sistema, generalmente el usuario debe ingresar sus credenciales.

Autorización, se refiere al proceso siguiente a la autenticación, este determina las acciones o recursos a los que puede acceder un usuario autenticado.

2. **¿Qué ventajas ofrece JWT frente a sesiones tradicionales de servidor y qué vulnerabilidades puede tener?**

JWT ofrece algunas ventajas como son la escalabilidad, los sistemas de sesiones tradicionales almacenando la información de cada usuario en un servidor, JWT no requiere este almacenamiento en servidor ya que la información necesaria viaja en el token; del mismo modo, al no requerir consultas al servidor las solicitudes autenticadas son más rápidas; también presenta facilidad de uso en APIs REST, especialmente en arquitecturas donde el cliente y el servidor están desacoplados, ya que el token puede enviar en cada petición HTTP sin mantener una sesión activa en el servidor.

Los riesgos que se pueden tener son: el robo del token, cuando un atacante obtiene un token, puede acceder al sistema hasta que el token expire; además, como el servidor no almacena los tokens, no puede invalidarlos fácilmente antes de su expiración.

3. **Explique cómo CORS protege (o restringe) la comunicación entre cliente y servidor.**

CORS es un mecanismo de seguridad que sirve para controlar el acceso a recursos desde distintos orígenes. Los navegadores suelen bloquear las solicitudes cross-origin para prevenir ataques como CSRF (un ataque que engaña al navegador de un usuario autenticado para que envíe solicitudes no deseadas a otro sitio web donde el usuario tiene sesión activa).

Funciona como un filtro de seguridad que el servidor debe configurar, especificando explícitamente los orígenes permitidos.

4. **Mencione tres vulnerabilidades del OWASP Top 10 que podrían afectar su API y cómo las mitigaría.**

- Fallas Criptográficas: Si las contraseñas se almacenan en un texto plano, una filtración de datos podría revelar todas las contraseñas de los usuarios. Además, el manejo de token_refresco en texto plano en la BD puede ser un riesgo.

  Medida Planificada: Uso de hashing seguro. La vulnerabilidad de almacenar contraseñas en texto plano se resolvió implementando el hashing seguro de contraseñas mediante la librería werkzeug.security de Flask.

- Inyección: Si las consultas SQL no usaran la sustitución de variables segura del conector de MySQL (es decir, concatenando directamente las entradas del usuario a la query), un atacante podría enviar comandos SQL maliciosos a través de las rutas API. Un atacante podría obtener información sensible de la tabla ‘persona’.

  Medida planificada: Uso exclusivo de consultas parametrizadas. Se ha implementado en personaModel todas las queries que utilizan el formado ‘%s’ y la tupla de datos separada en el método cursor.execute(query, (data,)). Esto garantiza que los datos del usuario se envíen por separado del comando SQL, mitigando la inyección SQL.

- Diseño Inseguro: El uso directo de credenciales de base de datos (DB_CONFIG) dentro del archivo de código fuente es una práctica de diseño insegura. Si el código fuente se filtrara, las credenciales de acceso total a la BD quedarían expuestas. Además, que faltan patrones de diseños seguros.

Medida Planificada: Incorporar seguridad desde el diseño (roles). Seguir el principio de menor privilegio para el usuario DB app_user. Se limitaron los permisos del usuario que se conecta desde el backend, este se encuentra al final del archivo schema.sql

5. **¿En qué parte del modelo C4 se deben representar las capas o componentes de seguridad y por qué?**

Los componentes de seguridad se deben representar en la capa 3, Diagrama de componentes, esto es debido a que estos elementos forman parte de la implementación interna de un contenedor, y en el nivel 3 se especifica como el backend implementa la autenticacion y autorizacion, y como dichos elementos interactúan con los controladores y modelos para proteger el acceso a los datos.

6. **¿Qué buenas prácticas debe seguir al almacenar contraseñas y manejar tokens en su proyecto?**

Para el almacenamiento de contraseñas se debe usar siempre hashing y no, guardarlas en un texto plano. Para el manejo de tokens, se usa tokens JWT de corta duración que son guardados en las cookies del navegador, los tokens de refresco para almacenarlos de forma segura debería ser en la base de datos, como en la tabla 'Persona'.

8. Conclusiones:

- La implementación de tokens JWT (JSON Web Tokens) permite darle al usuario ciertos controles. El servidor no tiene que memorizar cada usuario que entra, ya que simplemente revisa que el token sea auténtico cada vez que el usuario quiere acceder a una sección protegida. Esto es muy práctico porque el sistema no se sobrecarga y puede manejar a muchísimos usuarios al mismo tiempo sin problemas.
- El código del frontend se organizó de la siguiente manera para manejar la seguridad: una parte (`AuthContext`) actúa como el llavero donde se guarda la llave digital (el token) del usuario. Otra parte (`ProtectedRoute`) es como un guardia que revisa que tengas esa llave antes de dejar entrar a las páginas protegidas. Finalmente, (`api.js`) es el encargado de mostrar esa llave automáticamente cada vez que se le pide algo al servidor.

9. Recomendaciones:

- No exponer claves ni tokens en el código ni repositorio público.
- Probar rutas protegidas antes de fusionar a develop.
- Documentar las decisiones de seguridad en README o en un anexo /docs/security_notes.md.

10. Bibliografía

- OWASP Foundation. (2023). OWASP Top 10 – Web Application Security Risks.
- Auth0. JWT Handbook. https://auth0.com/learn/json-web-tokens
- Spring Security / Django Auth / Express JWT docs.
- PlantUML / Mermaid Model C4 Reference.