Guía de Actividades Práctico-Experimentales Nro. 002

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
Fecha	viernes 17 de octubre
Horario	07h30 – 10h30
Lugar	Aula 422
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 reactivos (Si aplica):

- 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.”.

5. Equipos y herramientas

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

6. Procedimiento / Metodología

Inicio

- Presentación de los objetivos y repaso de los conceptos JWT/OAuth2.
- Conformación de equipos y revisión de la rama de desarrollo (feature/security).

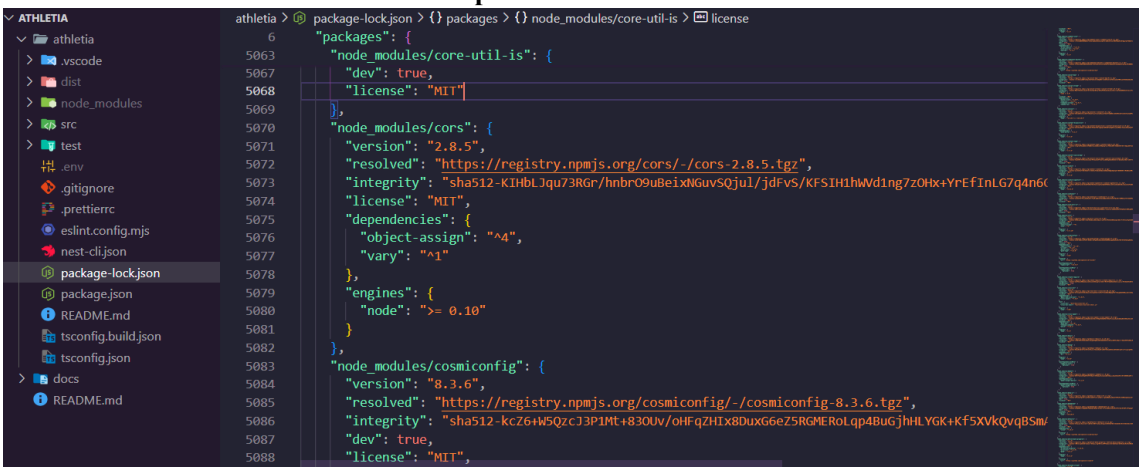
Desarrollo

1. **Configuración de entorno**

    - Instalar dependencias necesarias para seguridad.

Dependencias JWT

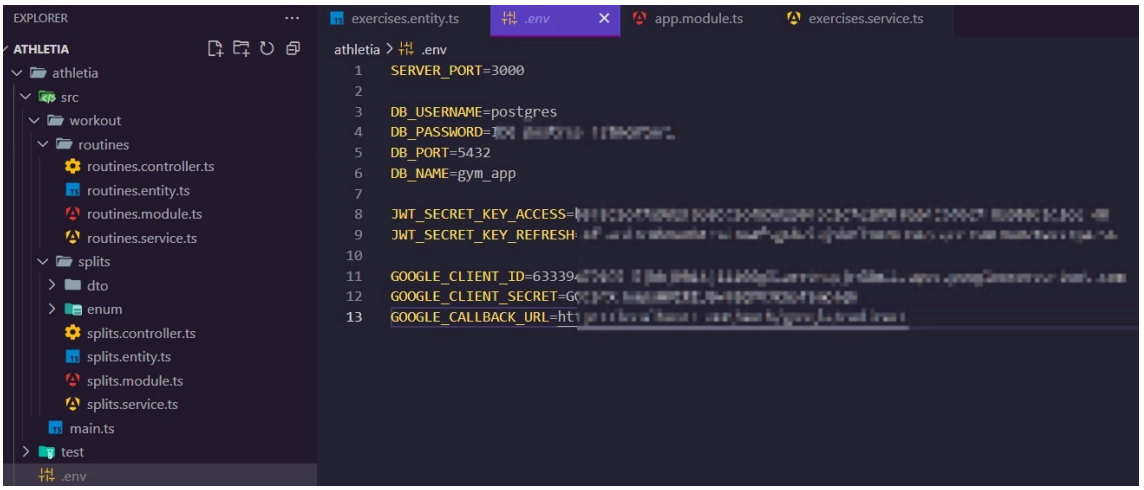
Dependencias CORS



Dependencias Argon2 (Se utilizó en lugar de Bcrypt)

- Crear archivo .env con claves secretas (no versionadas).

Archivo .env



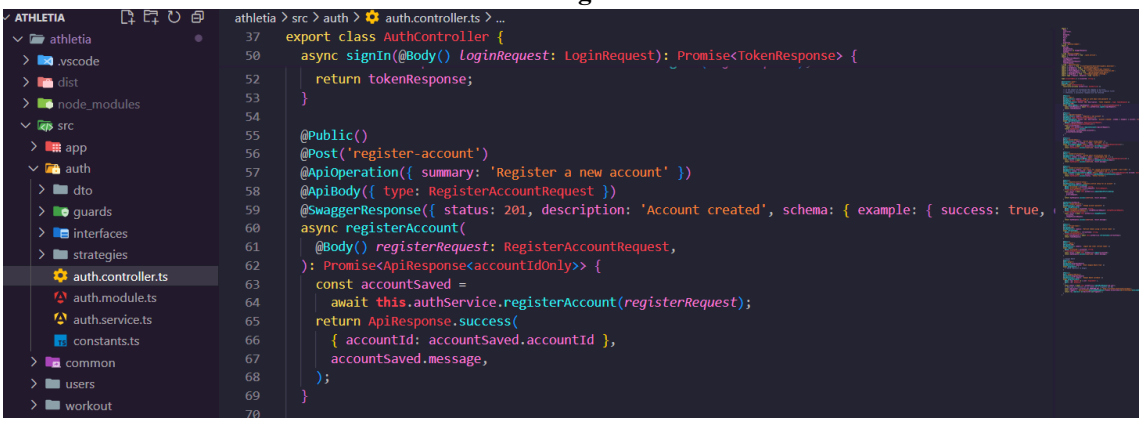
Archivo .env creado en el entorno de desarrollo con las respectivas claves secretas a utilizar en el proyecto.

2. **Implementación del flujo JWT / OAuth2**

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

Ruta /auth/login

Ruta /auth/register-account

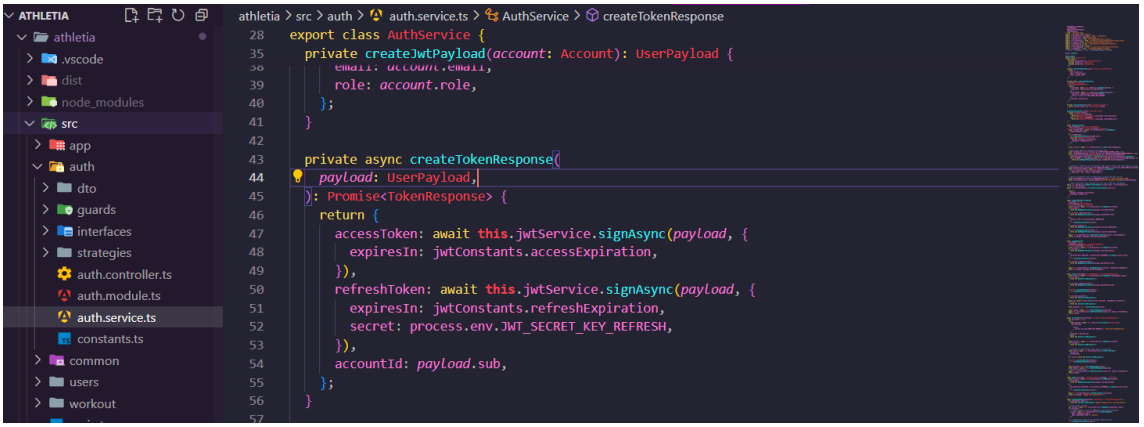


athletia > src > auth > auth.controller.ts > ...

```
export class AuthController {
  async signIn(@Body() loginRequest: LoginRequest): Promise<TokenResponse> {
    return tokenResponse;
  }

  @Public()
  @Post('register-account')
  @ApiOperation({ summary: 'Register a new account' })
  @ApiBody({ type: RegisterAccountRequest })
  @SwaggerResponse({ status: 201, description: 'Account created', schema: { example: { success: true,
  async registerAccount(
    @Body() registerRequest: RegisterAccountRequest,
  ): Promise<ApiResponse<accountIdOnly>> {
    const accountSaved =
      await this.authService.registerAccount(registerRequest);
    return ApiResponse.success(
      { accountId: accountSaved.accountId },
      accountSaved.message,
    );
  }
```

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

Generación Token JWT dentro de auth.module



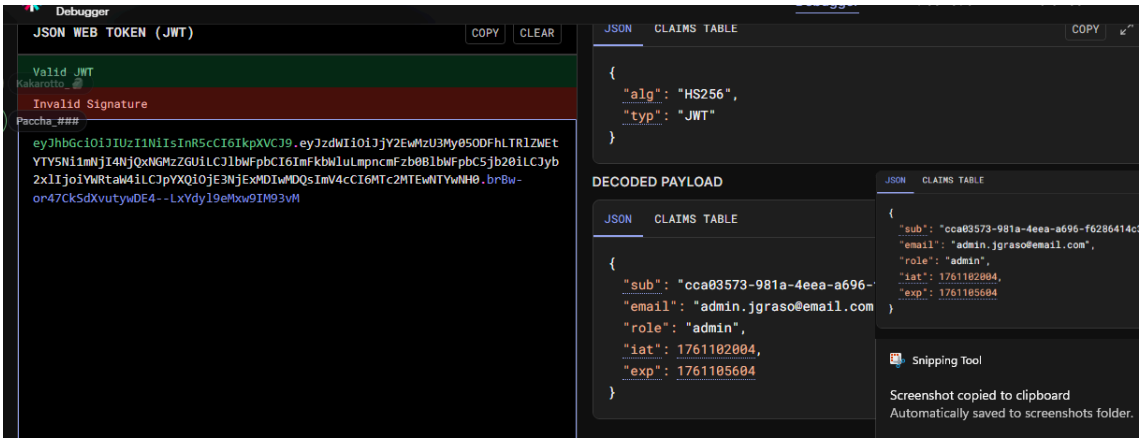
athletia > src > auth > auth.service.ts > AuthService > createTokenResponse

```
export class AuthService {
  private createJwtPayload(account: Account): UserPayload {
      email: account.email,
      role: account.role,
    };
  }

  private async createTokenResponse(
    payload: UserPayload,
  ): Promise<TokenResponse> {
    return {
      accessToken: await this.jwtService.signAsync(payload, {
        expiresIn: jwtConstants.accessExpiration,
      }),
      refreshToken: await this.jwtService.signAsync(payload, {
        expiresIn: jwtConstants.refreshExpiration,
        secret: process.env.JWT_SECRET_KEY_REFRESH,
      }),
      accountId: payload.sub,
    };
  }
```

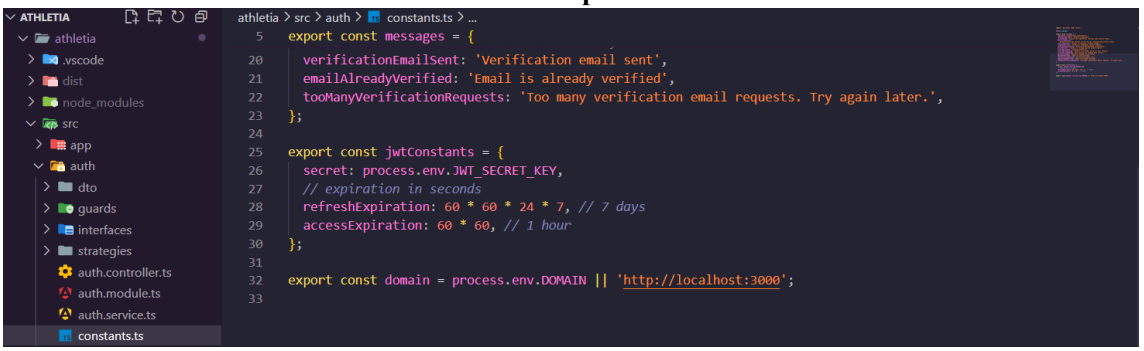
athletia > src > auth > auth.module.ts > ...

```
import { PassportModule } from '@nestjs/passport';
import { GoogleStrategy } from './strategies/google.strategy';
import { MailService } from 'src/common/mail/mail.service';

@Module({
  imports: [
    PassportModule.register({ session: false }),
    JwtModule.register({
      global: true,
      secret: process.env.JWT_SECRET_KEY_ACCESS || 'defaultSecretKey',
    }),
    AccountsModule,
  ],
  providers: [
    AuthService,
    GoogleStrategy,
    MailService,
  ],
  controllers: [AuthController],
  exports: [AuthService],
})
export class AuthModule {}
```

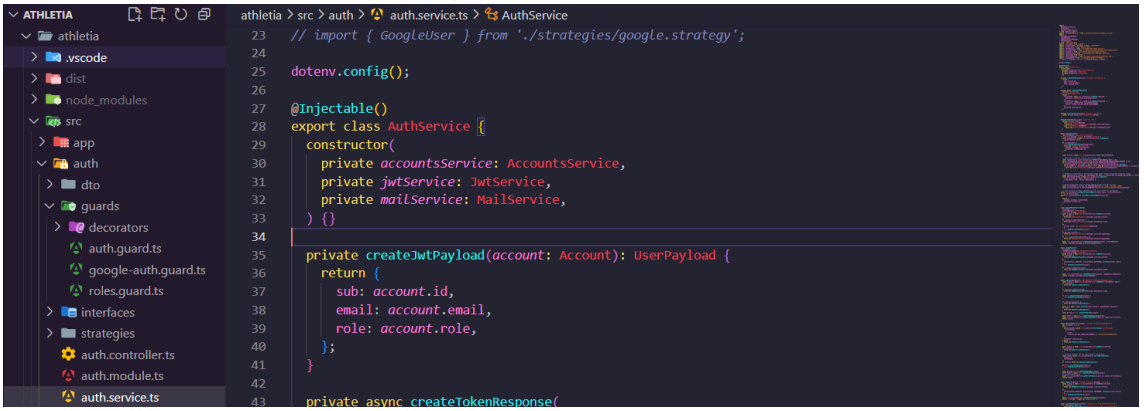
Definicion del iat del token JWT



Definición exp del JWT

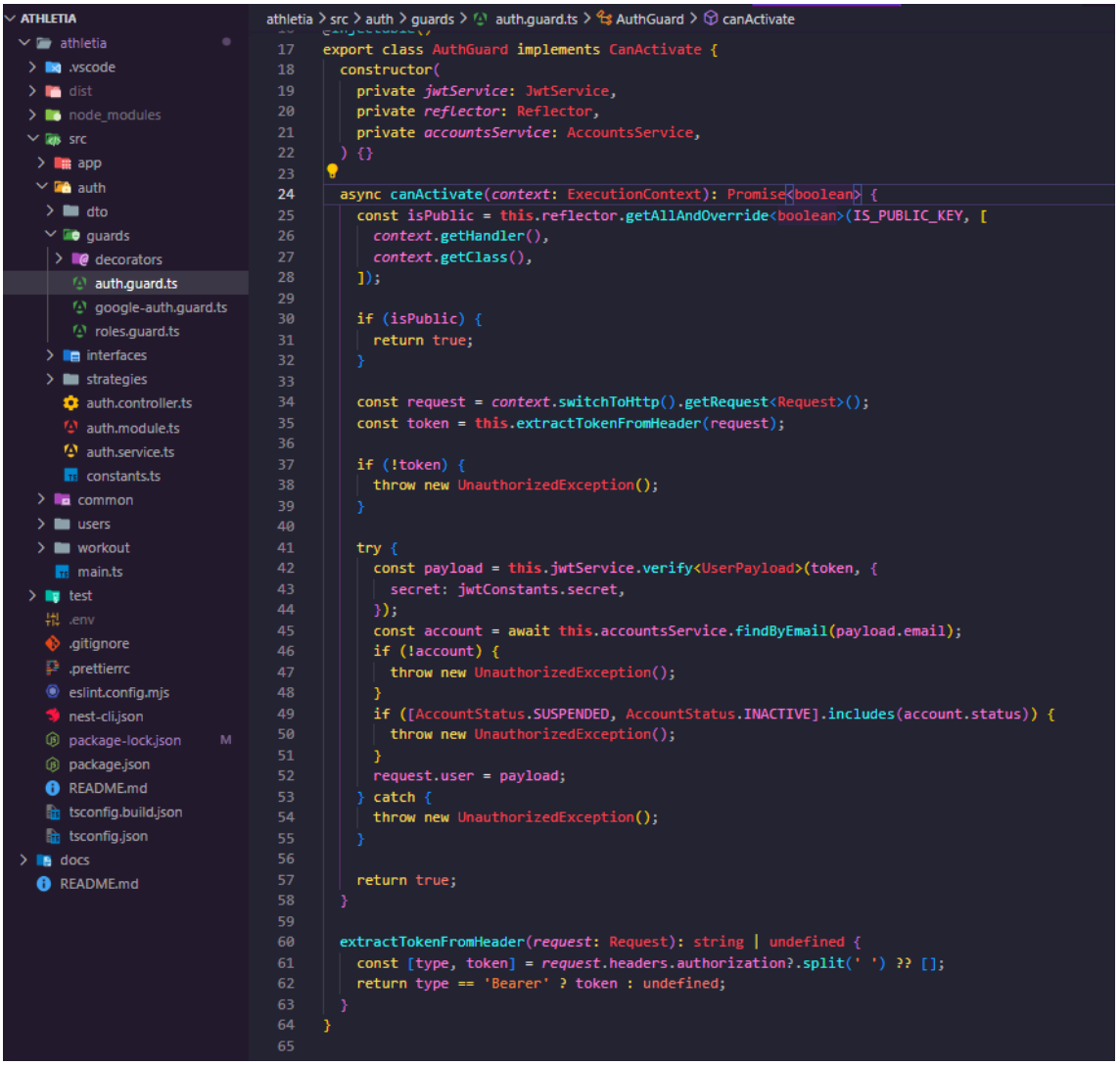


JWTPayload con Roles de Usuario



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

Middleware de Verificación de Token

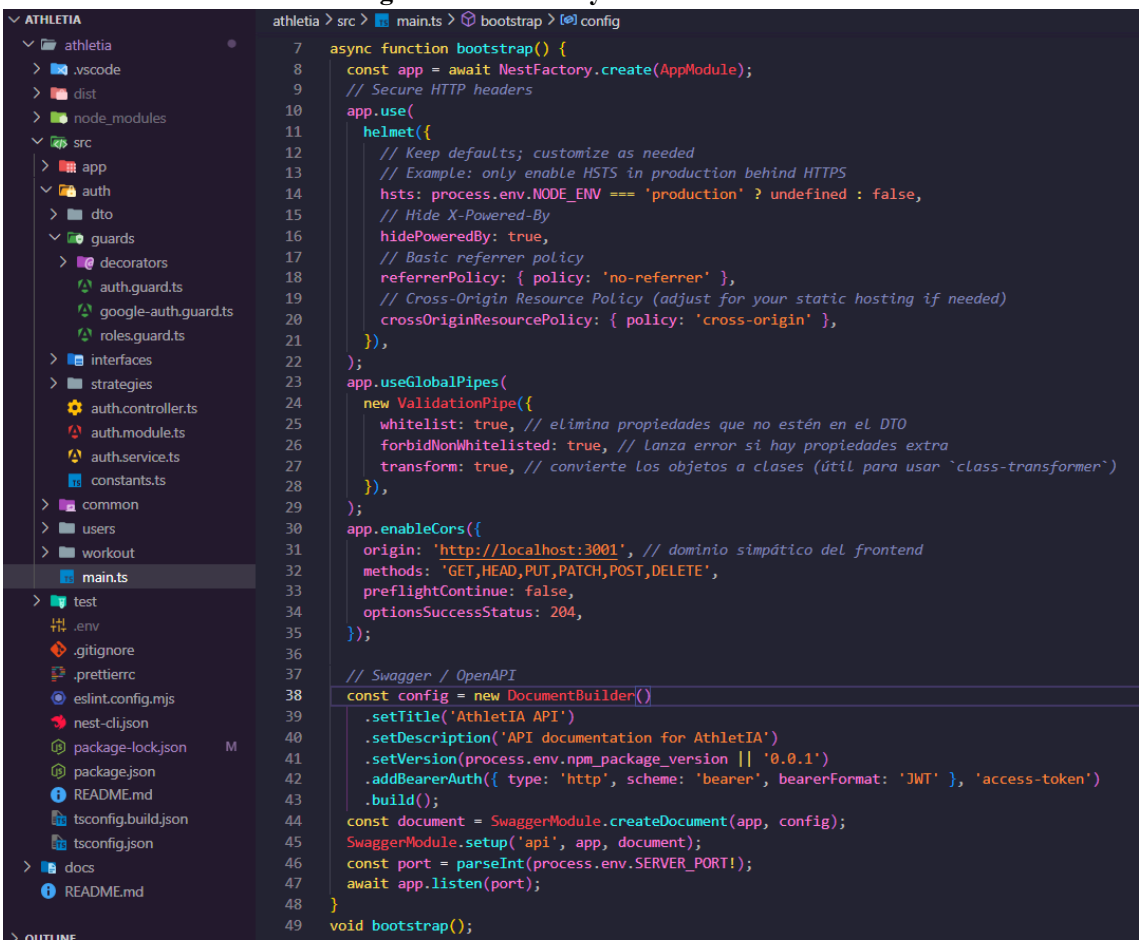


Roles (RBAC)

3. **Configuración CORS y validaciones**

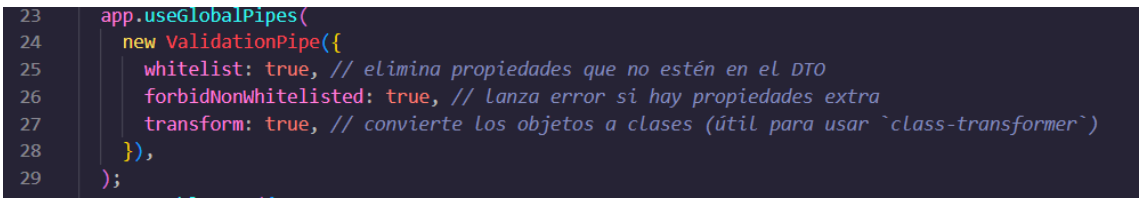
   - Definir orígenes permitidos y métodos HTTP.

Orígenes Permitidos y Métodos HTTP

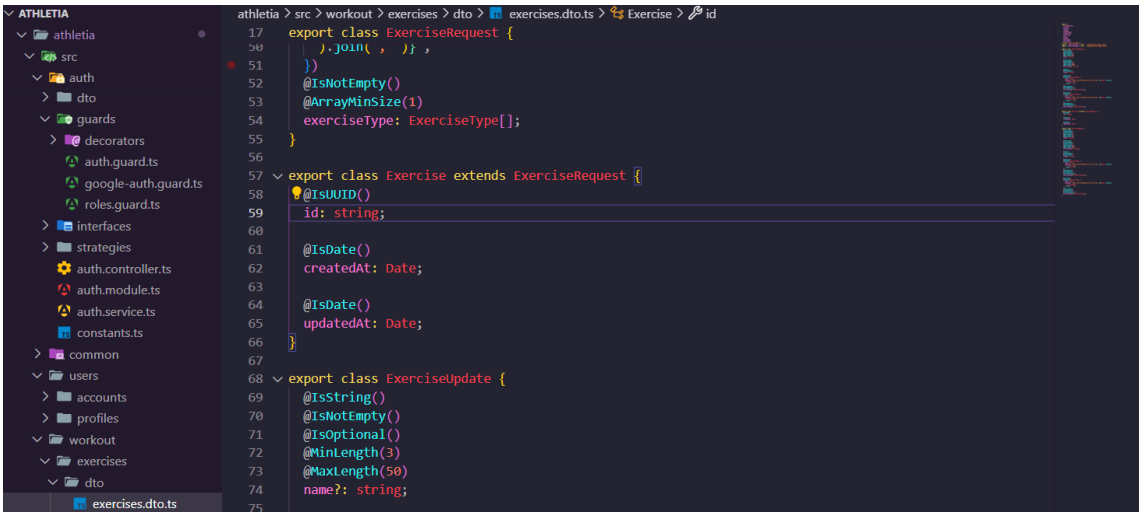


   - Agregar validación de entrada (request body y params).

Validación de Entrada

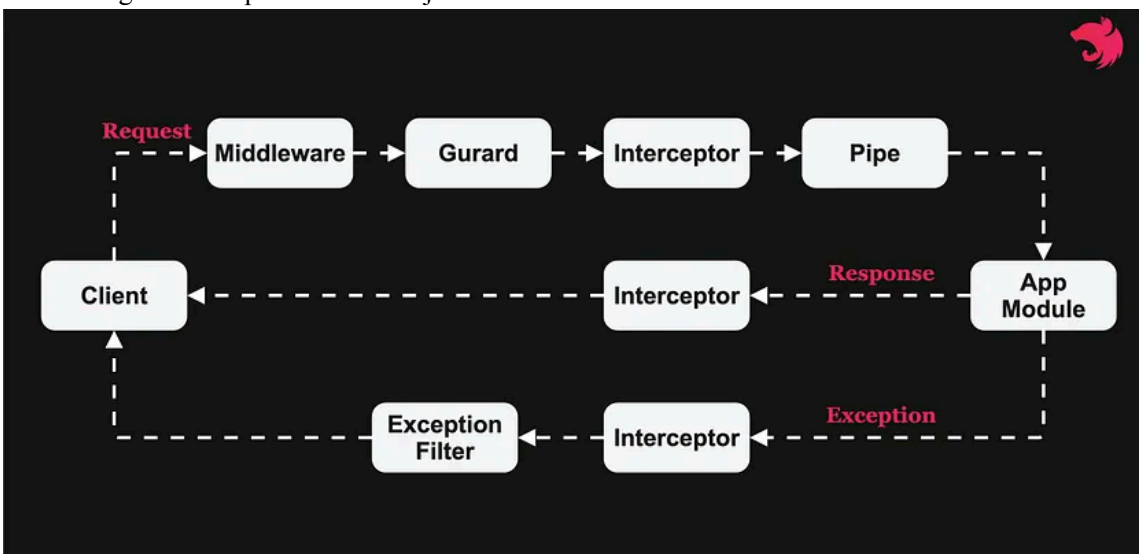


EJEMPLO DTO: exercises.dto



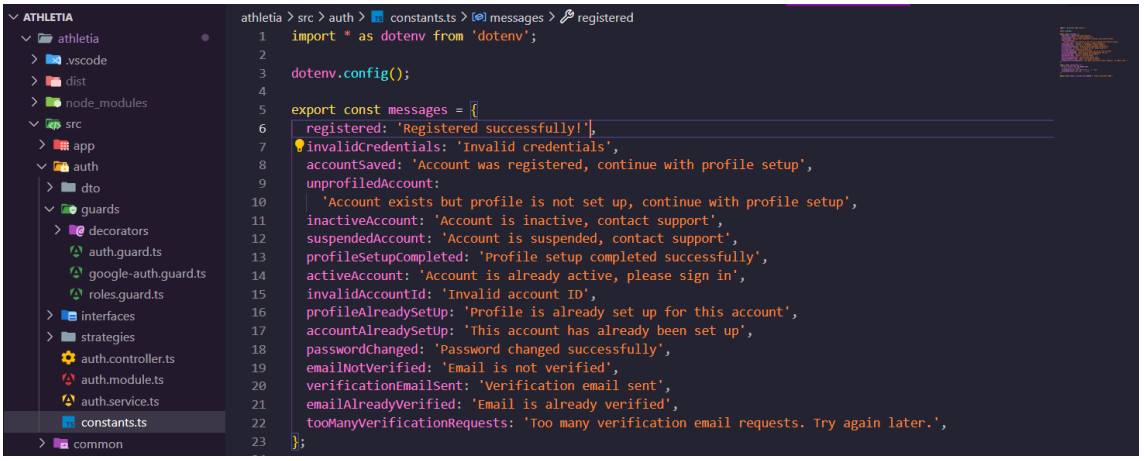
- Implementar manejo de errores uniforme (códigos HTTP y mensajes JSON).

La tecnología NestJS permite el manejo de errores de forma automática.



Ciclo de Vida de una petición HTTP en NestJS [4]

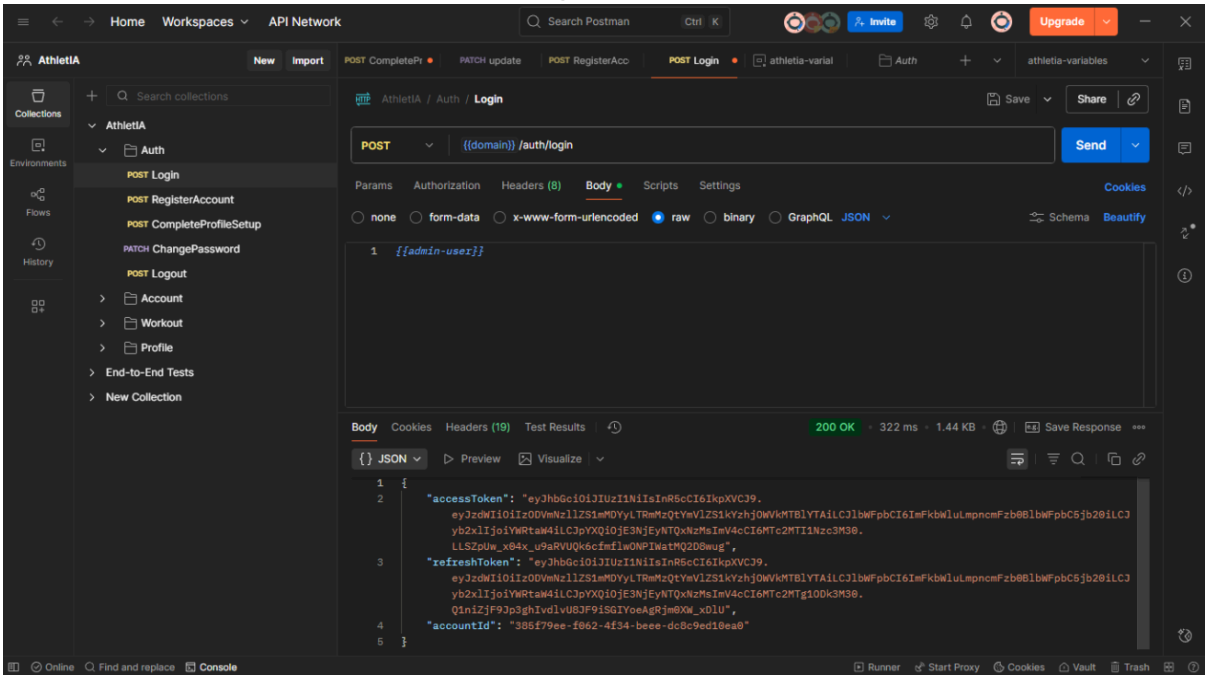
MENSAJES JSON



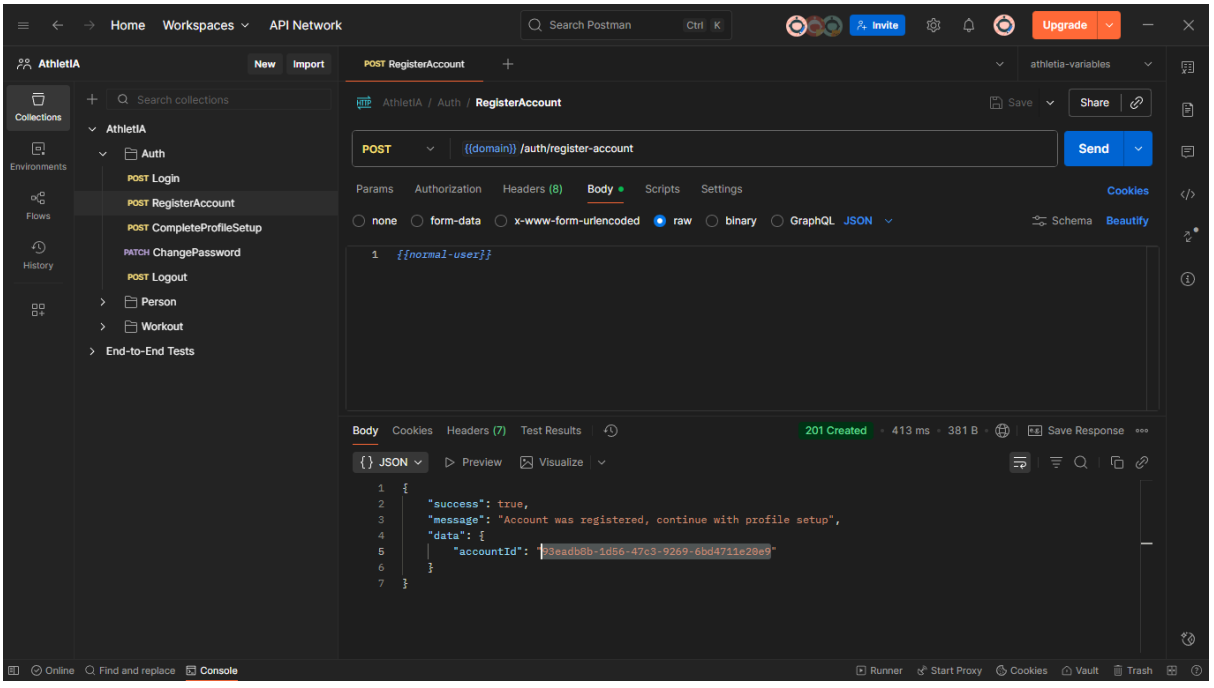
4. **Pruebas y verificación**

   - Probar login y rutas protegidas con Postman/Swagger.

Prueba de Login en Postman

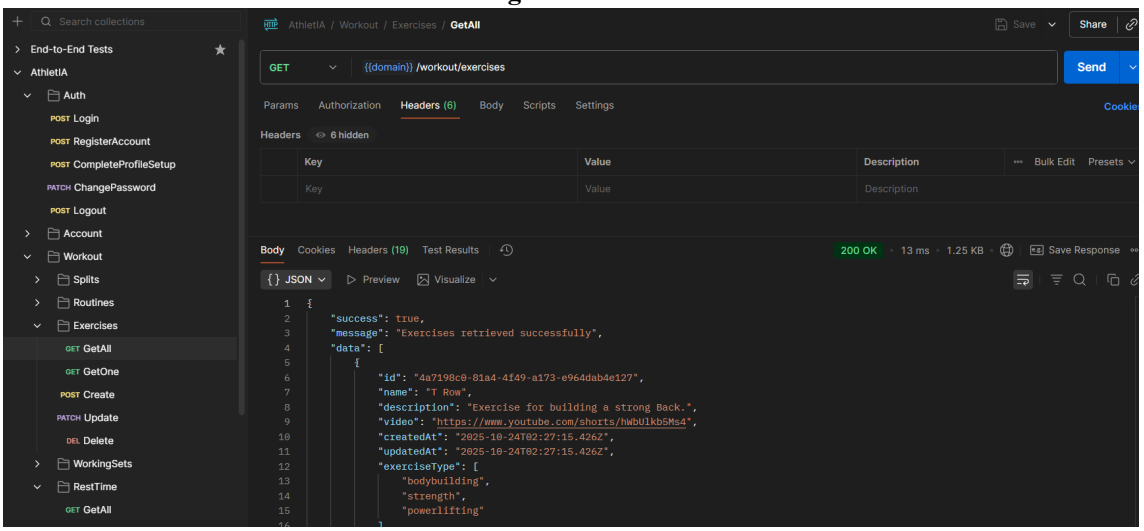


Prueba de Register en Postman

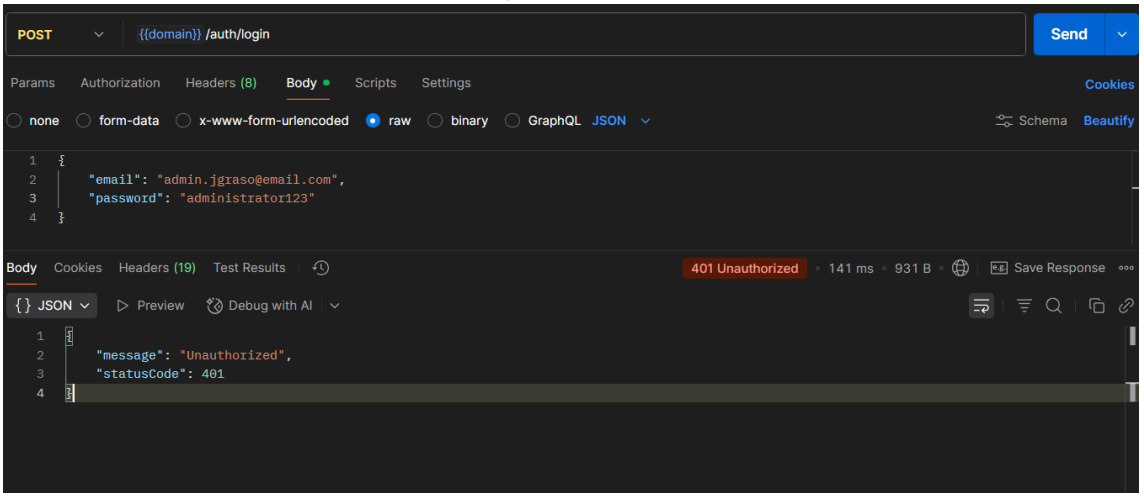


- Documentar resultados (capturas de respuesta 200, 401, 403).

Código de éxito 200



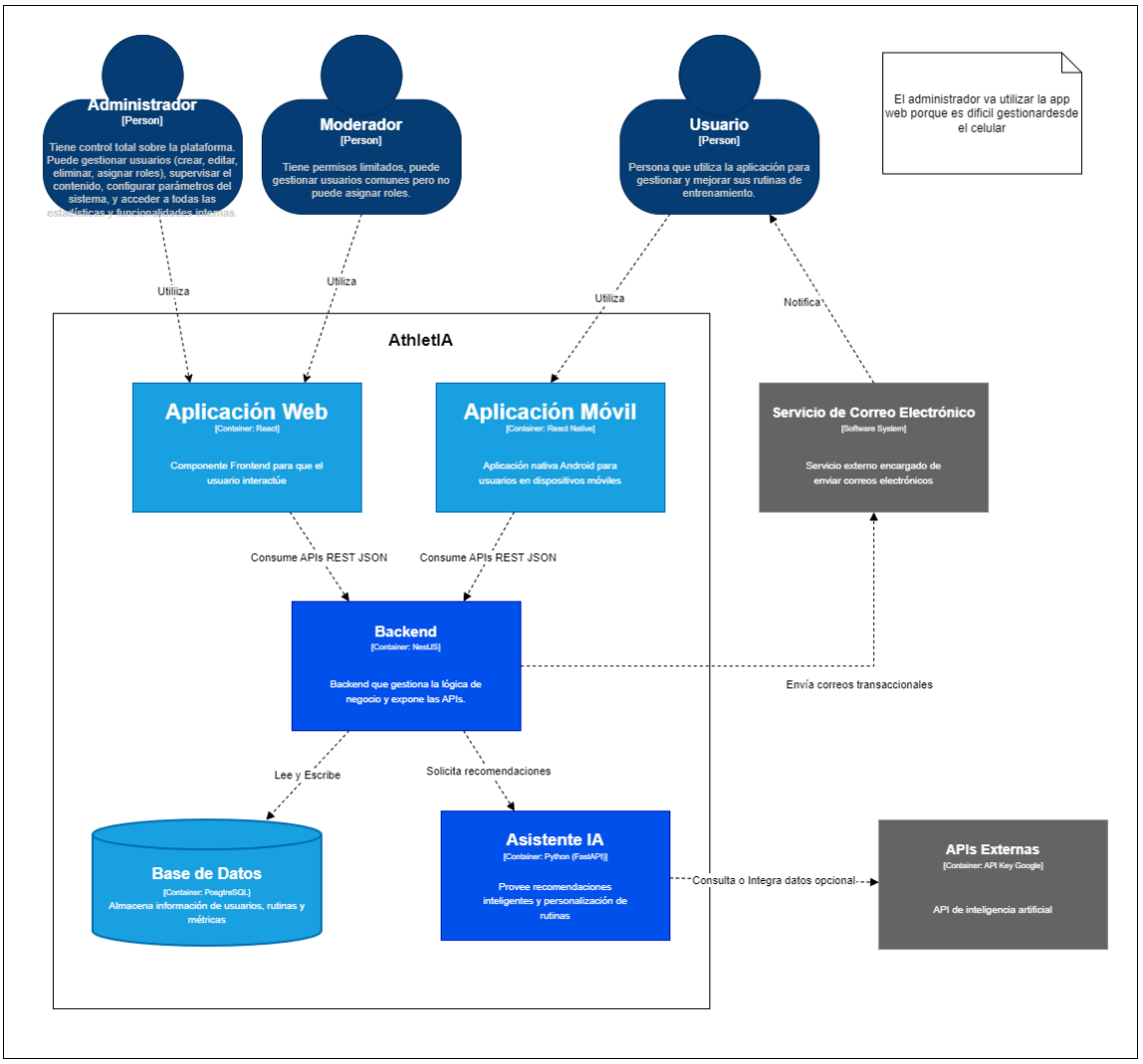
Código de Error 401



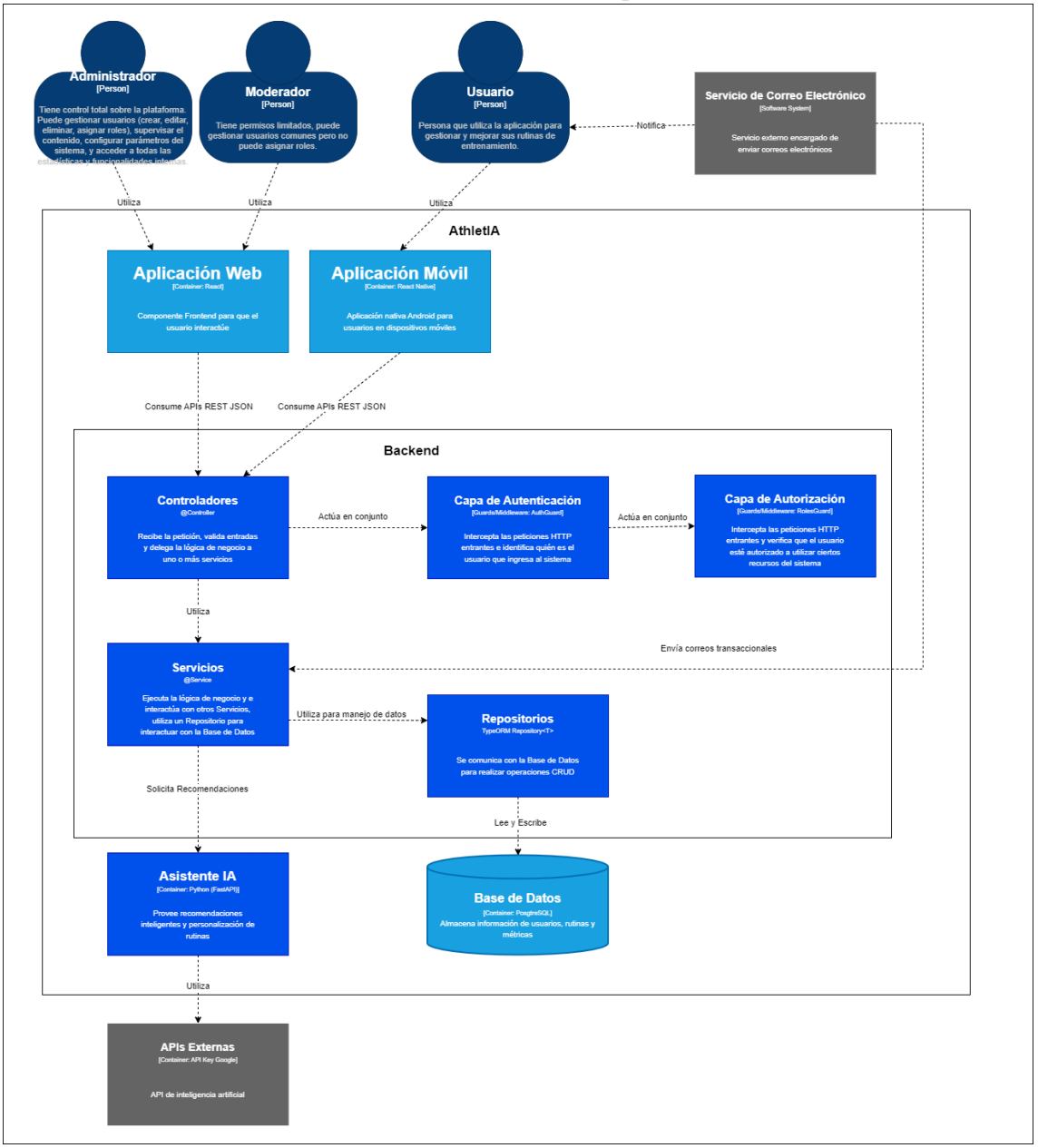
5. **Modelo C4 con seguridad**

   - Actualizar diagramas Container y Component para incluir los servicios de autenticación y módulos de seguridad.

Modelo C4 - Nivel 2 - Contenedor

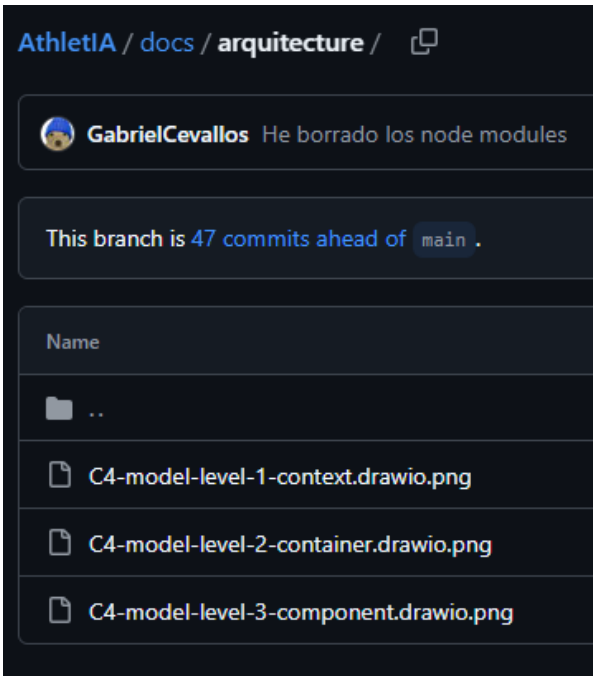


Modelo C4 - Nivel 3 - Component

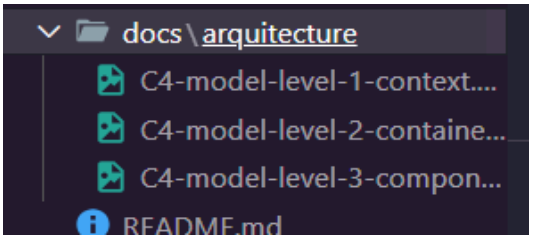


- Guardar los diagramas actualizados en /docs/architecture/.

Diagramas Actualizados en Repositorio GitHub



Diagramas Actualizados en el Proyecto



Cierre

- Socialización de resultados entre equipos.
- Retroalimentación docente sobre configuración y buenas prácticas OWASP.

7. Resultados esperados:

- Backend con flujo JWT u OAuth2 funcional.
- Políticas CORS y validaciones implementadas.
- Colección Postman/Swagger con pruebas de autenticación.
- Diagramas C4 actualizados mostrando componentes de seguridad.
- Evidencias (capturas de pantalla + README actualizado).

8. Preguntas de Control:

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

   La autenticación verifica las identidades de los usuarios, mientras que la autorización controla sus derechos de acceso dentro de una aplicación [1].

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

   Ventajas:

   - I**nteroperabilidad:** Un JWT puede transmitirse fácilmente entre varias aplicaciones, incluso si utilizan tecnologías diferentes [2].
   - **Gestión simplificada:** A diferencia de las sesiones tradicionales, no es necesario almacenar un estado en el lado del servidor [2].
   - **Autenticación eficiente:** Simplemente verificando la forma del token se autentica al usuario, sin necesidad de consultar una base de datos o un sistema de sesiones [2].
   - **Formato compacto y legible:** El formato (JSON codificado en base64url) es una forma compacta y legible de transportar datos, útil en arquitecturas distribuidas o APIs [2].

   Vulnerabilidades:

   - **Firma del JWT no verificada**: Ocurre cuando el servidor no comprueba la firma del token y simplemente decodifica el contenido para leerlo [2].
   - **Explotación de algoritmo "none":** Un atacante modifica la cabecera del token para establecer el algoritmo como "none". Si el servidor está mal configurado acepta este algoritmo, valida el token sin verificar ninguna firma [2].
   - **Secretos débiles y ataques de fuerza bruta:** Si se utiliza un algoritmo simétrico, la clave secreta compartida puede ser adivinada mediante ataques de fuerza bruta si es demasiado simple o predecible [2].

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

   CORS introduce el acceso medido, lo que ayuda a reducir el riesgo de amenazas e infracciones de seguridad. Estas son las formas en que lo hace.

   - **Evita el acceso no autorizado:** CORS especifica qué dominios pueden acceder a sus recursos, lo que reduce el riesgo de exposición de datos [3].
   - **Mitiga la violacion de datos:** CORS controla el acceso a través de encabezados definidos. Eso garantiza que los intrusos no puedan acceder a los datos [3].
   - **Directivas predefinidas:** Las directivas de CORS solo permiten solicitudes específicas. Al hacerlo, protege el sistema de ataques y violaciones de seguridad [3].

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

   1. Inyección (A03)

   - **¿Qué pasa?:** Consultas SQL/NoSQL o comandos con datos no válidos permiten ejecucion de codigo/inyección y robo/modificación de datos
   - **Mitigación**:
     - Validar y sanitizar entradas con class-validator + ValidationPipe
     - Evitar eval/exec y construir queries dinámicos solo con listas blancas.
     - Revisar logs para detectar patterns y añadir pruebas que intenten payloads de inyección.

2. Broken Access Control (A01)

- **¿Qué pasa?:** Usuarios acceden a recursos ajenos (IDs, endpoints, admin) por falta de verificación de roles/propietario.
- **Mitigación**:
  - Implementar guards y middleware RBAC (RolesGuard) y verificar propiedad del recurso (owner checks) en services.
  - No confiar solo en datos del cliente.
  - Principio de menor privilegio en endpoints y datos retornados.
  - Probar escenarios: escalado de privilegios, acceso directo por id, endpoints admin.

3. Cryptographic Failures (A02)

- **¿Qué pasa?:** Secretos débiles, tokens no rotados, contraseñas almacenadas en texto, errores con trazas que exponen información.
- **Mitigación**:
  - Hashing de contraseñas y evitar logging de contraseñas ni tokens.
  - Usar JTW secret fuerte en .env, expiración corta para access tokens y refresh tokens con almacenamiento seguro.
  - No publicar .env en repo, revisar .gitignore, evitar mensajes de error con stack trace en producción.
  - Usar rotación de claves y revocación de refresh tokens (lista negra) si es necesario.

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

   En nuestro caso lo colocamos en la capa 3 (nivel de componentes) porque al expandir el módulo del backend se muestra la existencia de la capa de autenticación y autorización.

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

   - Hashing de contraseñas con Argon2.
   - No almacenar ni hacer logging de contraseñas en texto plano.
   - Validar fuerza de contraseña y limitar intentos de login
   - Tener un reset seguro, es decir un token de un solo uso con respiración corta y guardar el token hashed.
   - Access tokens cortos, refresh tokens más largos pero rotables y guardados después del hashing en BD.
   - HTTPS, CORS restringidos.
   - No subir .env a repositorio

9. Conclusiones:

Se implementó de manera exitosa el flujo de autenticación mediante JWT y OAuth2, lo que incluye la instalación de las dependencias y la configuración necesaria para poder ejecutar estas funcionalidades, dentro de lo cual se incluye también el uso de variables de entorno para la gestión de claves secretas e información que no debe ser de conocimiento público.

Se aplicaron algunas prácticas del desarrollo seguro como el manejo de restricciones para orígenes desconocidos, la distinción entre los entornos de desarrollo y producción, utilización de middlewares para verificar la autenticidad de los emisores de las peticiones HTTP así como sus permisos dentro de nuestro sistema y una intensiva sesión de pruebas en la herramienta Postman que nos permite asegurar un funcionamiento correcto, seguro y limpio del módulo de autenticación y autorización.

10. Recomendaciones:

- Utilizar el archivo .gitignore para evitar exponer las claves secretas al repositorio remoto.
- Antes de utilizar cualquier biblioteca o herramienta, verificar que no esté obsoleta y que no presente fallos de seguridad o vulnerabilidades graves conocidas.
- Utilizar diagramas en lugar de texto para comprender de mejor manera el flujo de los procesos de autenticación y autorización.
- Realizar pruebas con la herramienta Postman para los endpoints de autenticación.
- Automatizar procesos repetitivos en el entorno de desarrollo, por ejemplo la preparación del IDE.
- Documentar de manera clara el esquema de las peticiones para evitar un uso incorrecto de la API.
- Utilizar IA generativa de manera ética y responsable para agilizar los procesos de construcción y pruebas.
- Configurar el entorno de ejecución antes de probar el sistema (en la fase de desarrollo).

11. Evaluación

Criterio	2 – Logro Alto	1 – Logro Medio	0 – Bajo / Sin Evidencia
1. Implementación de autenticación y autorización (JWT/OAuth2)	Flujo completo, tokens válidos y rutas protegidas operativas.	Flujo parcial o errores de validación de token.	No implementa autenticación funcional.
2. Configuración de CORS y validaciones	Configuración correcta, verificada en pruebas.	Parcial o con advertencias en consola.	Sin configuración verificable.
3. Aplicación de principios OWASP Top 10	Checklist completo y medidas de mitigación documentadas.	Checklist parcial o sin evidencia de mitigación.	No evidencia revisión OWASP.
4. Actualización del modelo C4 (Container y Component)	Diagramas actualizados y coherentes con las modificaciones de seguridad.	Diagramas incompletos o sin claridad en los componentes de seguridad.	No presenta actualización del C4.
5. Documentación y entrega de evidencias	PDF y README completos con capturas y referencias a la implementación.	Entrega parcial o poco clara.	No entrega evidencias o sin documentación.

12. 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.

[1] R. Raj, "Demystifying authentication and authorization in backend systems", *Medium*, 10-may-2024. [En línea]. Disponible en: https://medium.com/@rohitraj1912000/demystifying-authentication-and-authorization-in-backend-systems-52489c3fae8c. [Consultado: 23-oct-2025].

[2] "JWT (JSON Web Token): Vulnerabilities, common attacks and security best practices", *VAADATA - Ethical Hacking Services*, 30-abr-2025. [En línea]. Disponible en: https://www.vaadata.com/blog/jwt-json-web-token-vulnerabilities-common-attacks-and-security-best-practices/. [Consultado: 23-oct-2025].

[3] "Secure API configurations: Key CORS headers for safe resource sharing", *Contentstack*. [En línea]. Disponible en: https://www.contentstack.com/blog/tech-talk/secure-api-configurations-key-cors-headers-for-safe-resource-sharing. [Consultado: 23-oct-2025].

[4] Dan. "NestJS Request Lifecycle: A Complete Guide to the Architecture". Medium. Accedido el 23 de octubre de 2025. [En línea]. Disponible: https://medium.com/@daiki01240/nestjs-request-lifecycle-a-complete-guide-to-the-architecture-5ada9666867a

13. Elaboración y Aprobación

Elaborado por	Edison L Coronel Romero **Docente**	
Aprobado por	Edison L Coronel Romero **Director de Carrera**	