



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

The screenshot shows the VS Code editor with the 'package-lock.json' file open. The left sidebar shows the project structure with 'package-lock.json' selected. The main editor displays the JSON content of the file. The following dependencies are highlighted in the image:

- `node_modules/@types/json-schema`
- `node_modules/@types/jsonwebtoken`
- `node_modules/@types/methods`

Dependencias CORS

The screenshot shows the VS Code editor with the 'package-lock.json' file open. The left sidebar shows the project structure with 'package-lock.json' selected. The main editor displays the JSON content of the file. The following dependencies are highlighted in the image:

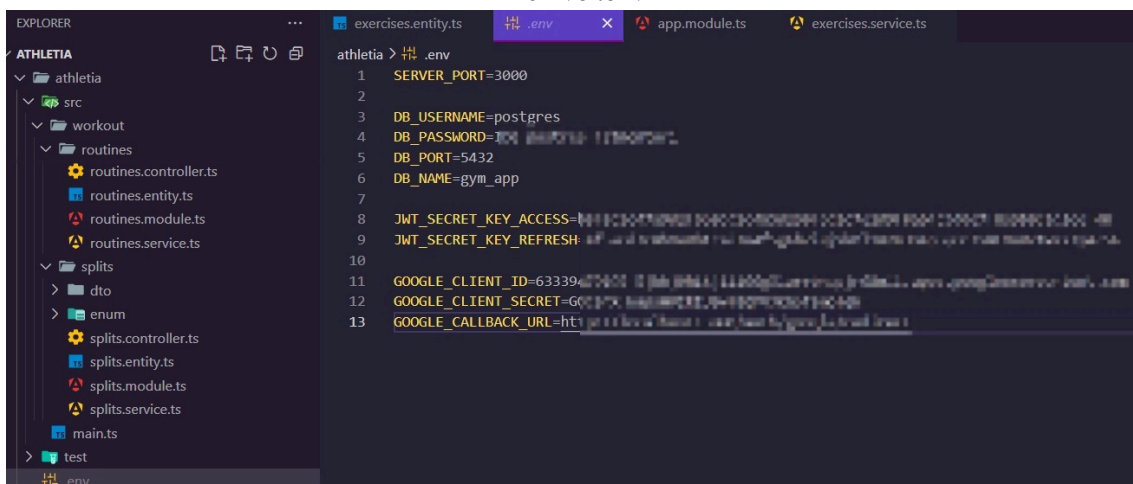
- `node_modules/core-util-is`
- `node_modules/cors`
- `node_modules/cosmiconfig`

Dependencias Argon2 (Se utilizó en lugar de Bcrypt)



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

Archivo .env

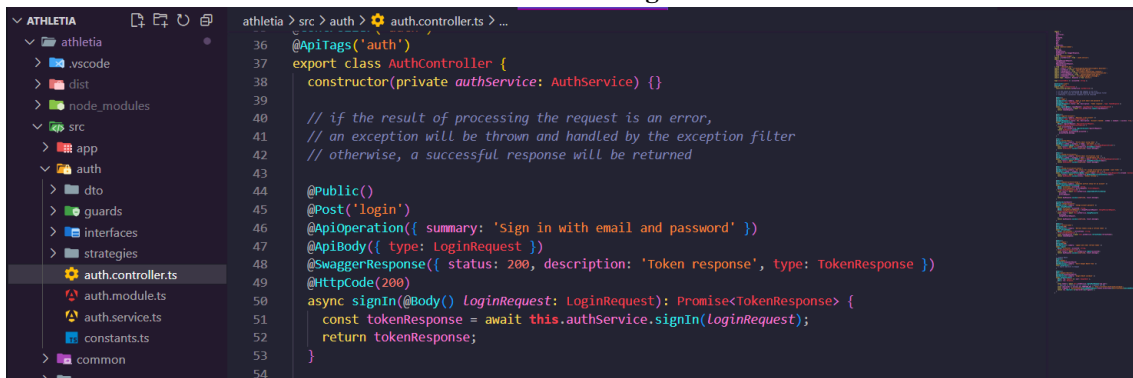


Archivo .env creado en el entorno de desarrollo con las respectivas claves secretas a utilizar en el proyecto.

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 > ...
37 export class AuthController {
50   async signIn(@Body() loginRequest: LoginRequest): Promise<TokenResponse> {
52     return tokenResponse;
53   }
54
55   @Public()
56   @Post('register-account')
57   @ApiOperation({ summary: 'Register a new account' })
58   @ApiBody({ type: RegisterAccountRequest })
59   @SwaggerResponse({ status: 201, description: 'Account created', schema: { example: { success: true,
60   async registerAccount(
61     @Body() registerRequest: RegisterAccountRequest,
62   ): Promise<ApiResponse<accountIdOnly>> {
63     const accountSaved =
64       await this.authService.registerAccount(registerRequest);
65     return ApiResponse.success(
66       { accountId: accountSaved.accountId },
67       accountSaved.message,
68     );
69   }
70 }
```

- 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
28 export class AuthService {
35   private createJwtPayload(account: Account): UserPayload {
36     email: account.email,
39     role: account.role,
40   };
41 }
42
43 private async createTokenResponse(
44   payload: UserPayload,
45 ): Promise<TokenResponse> {
46   return {
47     accessToken: await this.jwtService.signAsync(payload, {
48       expiresIn: jwtConstants.accessExpiration,
49     }),
50     refreshToken: await this.jwtService.signAsync(payload, {
51       expiresIn: jwtConstants.refreshExpiration,
52       secret: process.env.JWT_SECRET_KEY_REFRESH,
53     }),
54     accountId: payload.sub,
55   };
56 }
57 }
```

```
ATHLETIA  src > auth > auth.module.ts > ...
5 import { AccountsModule } from 'src/users/accounts/accounts.module';
6 import { PassportModule } from '@nestjs/passport';
7 import { GoogleStrategy } from './strategies/google.strategy';
8 import { MailService } from 'src/common/mail/mail.service';
9
10 @Module({
11   imports: [
12     PassportModule.register({ session: false }),
13     JwtModule.register({
14       global: true,
15       secret: process.env.JWT_SECRET_KEY_ACCESS || 'defaultSecretKey',
16     }),
17     AccountsModule,
18   ],
19   providers: [
20     AuthService,
21     GoogleStrategy,
22     MailService,
23   ],
24   controllers: [AuthController],
25   exports: [AuthService],
26 })
27 export class AuthModule {}
```

Definición del iat del token JWT

The screenshot shows a web browser's developer console with the following content:

- Debugger** (top left)
- JSON WEB TOKEN (JWT)** (tab)
- Valid JWT** (green bar)
- Invalid Signature** (red bar)
- Paccha_###** (text)
- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ3ZGVhZDl0IjY2ZWZlU3Myb050DFhLTRL1ZMETYVYSNI1mWjI4NjQxNGMzZGUiLCJlbWVpbC1mFkbWluLmNpcmlFZm80bWVpbC5jb201LCJyb2x1IjoieWRTak41LCJpYXQiOiJlZmVjExMDIwMDQ5ImV4cCI6MTc2MTEwNTYwNH0.br8W-or47CkSdXvutywDE4--LxYdy19eBxw9IM93vM** (JWT token)
- JSON CLAIMS TABLE** (tab)
- JSON** (tab)
- CLAIMS TABLE** (tab)
- DECODED PAYLOAD** (tab)
- JSON** (tab)
- CLAIMS TABLE** (tab)
- sub**: "cca03573-981a-4eea-a696-f6286414c"
- email**: "admin.jgraso@email.com"
- role**: "admin"
- iat**: 1761102804
- exp**: 1761105604
- sub**: "cca03573-981a-4eea-a696-f6286414c"
- email**: "admin.jgraso@email.com"
- role**: "admin"
- iat**: 1761102804
- exp**: 1761105604

Definición exp del JWT

```
athletia > src > auth > constants.ts > ...
5  export const messages = {
20    verificationEmailSent: 'Verification email sent',
21    emailAlreadyVerified: 'Email is already verified',
22    tooManyVerificationRequests: 'Too many verification email requests. Try again later.',
23  };
24
25  export const jwtConstants = {
26    secret: process.env.JWT_SECRET_KEY,
27    // expiration in seconds
28    refreshExpiration: 60 * 60 * 24 * 7, // 7 days
29    accessExpiration: 60 * 60, // 1 hour
30  };
31
32  export const domain = process.env.DOMAIN || 'http://localhost:3000';
33
```

JWTPayload con Roles de Usuario

The screenshot shows the VS Code interface. On the left, the file explorer displays the project structure. The 'athletia' folder is expanded, showing subfolders like 'src', 'dto', 'guards', 'decorators', 'interfaces', and 'strategies'. The 'src' folder is also expanded, showing files like 'auth.guard.ts', 'google-auth.guard.ts', 'roles.guard.ts', 'auth.controller.ts', 'auth.module.ts', and 'auth.service.ts'. On the right, the source code of 'auth.service.ts' is displayed. The code shows the 'AuthService' class with a constructor and a 'createJwtPayload' method. The constructor takes 'accountsService', 'jwtService', and 'mailService' as dependencies. The 'createJwtPayload' method takes an 'Account' object and returns a 'UserPayload' object with 'id', 'email', and 'role' properties.

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

Middleware de Verificación de Token

```

ATHLETIA > src > auth > guards > auth.guard.ts > AuthGuard > canActivate
17 export class AuthGuard implements CanActivate {
18   constructor(
19     private jwtService: JwtService,
20     private reflector: Reflector,
21     private accountsService: AccountsService,
22   ) {}
23
24   async canActivate(context: ExecutionContext): Promise<boolean> {
25     const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
26       context.getHandler(),
27       context.getClass(),
28     ]);
29
30     if (isPublic) {
31       return true;
32     }
33
34     const request = context.switchToHttp().getRequest<Request>();
35     const token = this.extractTokenFromHeader(request);
36
37     if (!token) {
38       throw new UnauthorizedException();
39     }
40
41     try {
42       const payload = this.jwtService.verify<UserPayload>(token, {
43         secret: jwtConstants.secret,
44       });
45       const account = await this.accountsService.findByEmail(payload.email);
46       if (!account) {
47         throw new UnauthorizedException();
48       }
49       if ([AccountStatus.SUSPENDED, AccountStatus.INACTIVE].includes(account.status)) {
50         throw new UnauthorizedException();
51       }
52       request.user = payload;
53     } catch {
54       throw new UnauthorizedException();
55     }
56
57     return true;
58   }
59
60   extractTokenFromHeader(request: Request): string | undefined {
61     const [type, token] = request.headers.authorization?.split(' ') ?? [];
62     return type == 'Bearer' ? token : undefined;
63   }
64 }
65

```

Roles (RBAC)

```

ATHLETIA > src > auth > guards > roles.guard.ts > ...
1 import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
2 import { Reflector } from '@nestjs/core';
3 import { Request } from 'express';
4 import { ROLES_KEY } from 'src/auth/guards/decorators/roles.decorator';
5
6 @Injectable()
7 export class RolesGuard implements CanActivate {
8   constructor(private reflector: Reflector) {}
9
10  canActivate(context: ExecutionContext): boolean {
11    const roles = this.reflector.get<string[]>(ROLES_KEY, context.getHandler());
12    if (!roles) {
13      return true;
14    }
15    const request = context.switchToHttp().getRequest<Request>();
16    const user = request.user;
17    return user && roles.includes(user.role);
18  }
19 }

```

3. Configuración CORS y validaciones

- Definir orígenes permitidos y métodos HTTP.

Orígenes Permitidos y Métodos HTTP

```
athletia > src > main.ts > bootstrap > config
7  async function bootstrap() {
8    const app = await NestFactory.create(AppModule);
9    // Secure HTTP headers
10   app.use(
11     helmet({
12       // Keep defaults; customize as needed
13       // Example: only enable HSTS in production behind HTTPS
14       hsts: process.env.NODE_ENV === 'production' ? undefined : false,
15       // Hide X-Powered-By
16       hidePoweredBy: true,
17       // Basic referrer policy
18       referrerPolicy: { policy: 'no-referrer' },
19       // Cross-Origin Resource Policy (adjust for your static hosting if needed)
20       crossOriginResourcePolicy: { policy: 'cross-origin' },
21     }),
22   );
23   app.useGlobalPipes(
24     new ValidationPipe({
25       whitelist: true, // elimina propiedades que no estén en el DTO
26       forbidNonWhitelisted: true, // lanza error si hay propiedades extra
27       transform: true, // convierte los objetos a clases (útil para usar `class-transformer`)
28     }),
29   );
30   app.enableCors({
31     origin: 'http://localhost:3001', // dominio simpático del frontend
32     methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
33     preflightContinue: false,
34     optionsSuccessStatus: 204,
35   });
36
37   // Swagger / OpenAPI
38   const config = new DocumentBuilder()
39     .setTitle('AthletIA API')
40     .setDescription('API documentation for AthletIA')
41     .setVersion(process.env.npm_package_version || '0.0.1')
42     .addBearerAuth({ type: 'http', scheme: 'bearer', bearerFormat: 'JWT' }, 'access-token')
43     .build();
44   const document = SwaggerModule.createDocument(app, config);
45   SwaggerModule.setup('api', app, document);
46   const port = parseInt(process.env.SERVER_PORT!);
47   await app.listen(port);
48 }
49 void bootstrap();
```

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

Validación de Entrada

```
23   app.useGlobalPipes(
24     new ValidationPipe({
25       whitelist: true, // elimina propiedades que no estén en el DTO
26       forbidNonWhitelisted: true, // lanza error si hay propiedades extra
27       transform: true, // convierte los objetos a clases (útil para usar `class-transformer`)
28     }),
29   );
```

EJEMPLO DTO: exercises.dto

```

ATHLETIA
├── athletia
│   ├── src
│   │   ├── dto
│   │   │   └── exercises.dto.ts
│   │   ├── guards
│   │   │   ├── decorators
│   │   │   │   ├── auth.guard.ts
│   │   │   │   ├── google-auth.guard.ts
│   │   │   │   └── roles.guard.ts
│   │   │   ├── interfaces
│   │   │   ├── strategies
│   │   │   ├── auth.controller.ts
│   │   │   ├── auth.module.ts
│   │   │   ├── auth.service.ts
│   │   │   └── constants.ts
│   │   ├── common
│   │   ├── users
│   │   │   ├── accounts
│   │   │   ├── profiles
│   │   │   └── workouts
│   │   │       ├── exercises
│   │   │       │   └── dto
│   │   │       └── exercises.dto.ts
│   │   └── workouts
│   │       ├── exercises
│   │       │   └── dto
│   │       └── exercises.dto.ts
│   └── workouts
│       ├── exercises
│       │   └── dto
│       └── exercises.dto.ts
└── workouts
    ├── exercises
    │   └── dto
    └── exercises.dto.ts

```

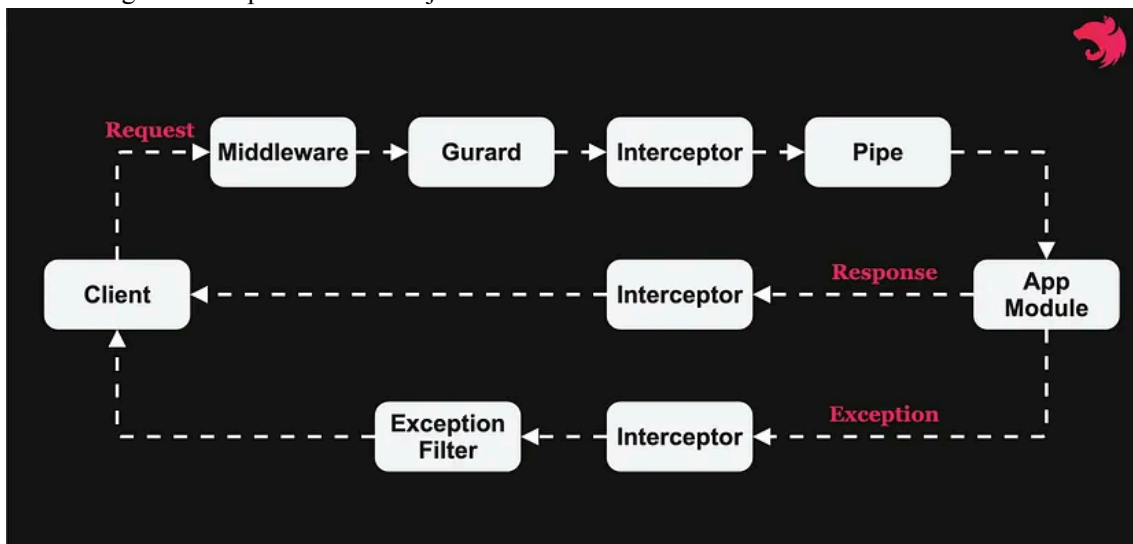
```

17 export class ExerciseRequest {
18   @IsNotEmpty()
19   @ArrayMinSize(1)
20   exerciseType: ExerciseType[];
21 }
22
23 export class Exercise extends ExerciseRequest {
24   @IsUUID()
25   id: string;
26
27   @IsDate()
28   createdAt: Date;
29
30   @IsDate()
31   updatedAt: Date;
32 }
33
34 export class ExerciseUpdate {
35   @IsString()
36   @IsNotEmpty()
37   @IsOptional()
38   @MinLength(3)
39   @MaxLength(50)
40   name?: string;
41 }

```

- 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

```

ATHLETIA
├── athletia
│   ├── .vscode
│   ├── dist
│   ├── node_modules
│   ├── src
│   │   ├── app
│   │   ├── auth
│   │   │   ├── dto
│   │   │   ├── guards
│   │   │   │   ├── decorators
│   │   │   │   │   ├── auth.guard.ts
│   │   │   │   │   ├── google-auth.guard.ts
│   │   │   │   │   └── roles.guard.ts
│   │   │   │   ├── interfaces
│   │   │   │   ├── strategies
│   │   │   │   ├── auth.controller.ts
│   │   │   │   ├── auth.module.ts
│   │   │   │   ├── auth.service.ts
│   │   │   │   └── constants.ts
│   │   │   ├── common
│   │   ├── users
│   │   │   ├── accounts
│   │   │   ├── profiles
│   │   │   └── workouts
│   │   │       ├── exercises
│   │   │       │   └── dto
│   │   │       └── exercises.dto.ts
│   │   └── workouts
│   │       ├── exercises
│   │       │   └── dto
│   │       └── exercises.dto.ts
│   └── workouts
│       ├── exercises
│       │   └── dto
│       └── exercises.dto.ts
└── workouts
    ├── exercises
    │   └── dto
    └── exercises.dto.ts

```

```

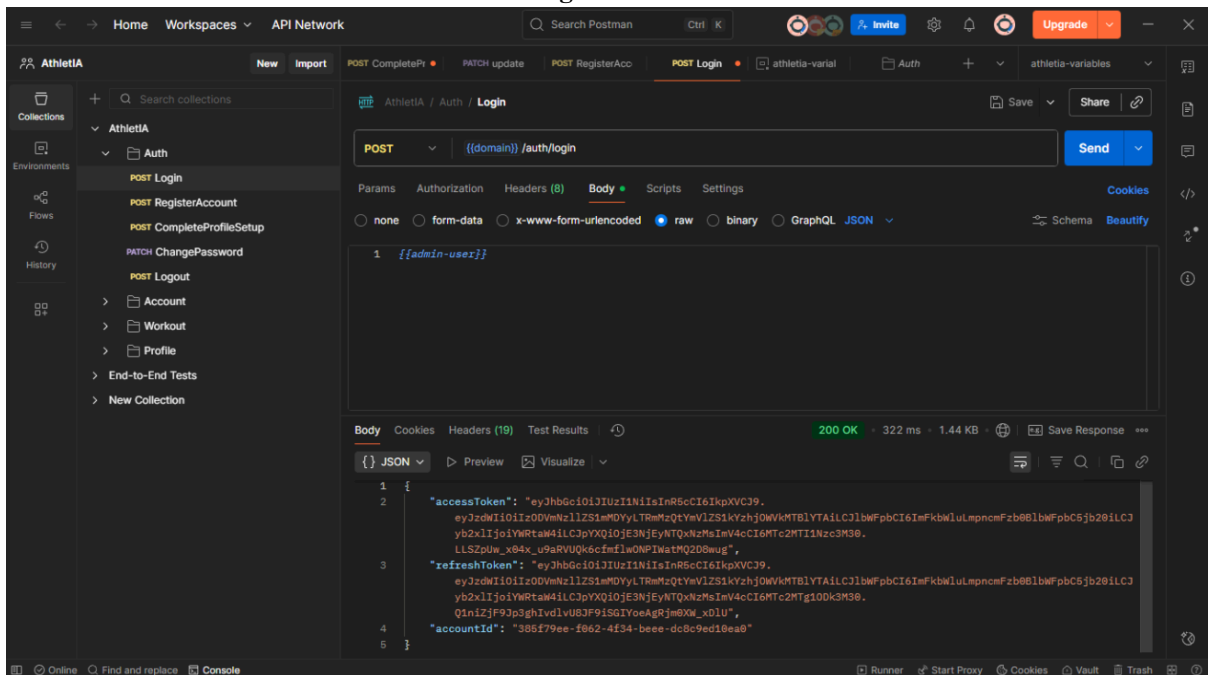
1 import * as dotenv from 'dotenv';
2
3 dotenv.config();
4
5 export const messages = {
6   registered: 'Registered successfully!',
7   invalidCredentials: 'Invalid credentials',
8   accountsSaved: 'Account was registered, continue with profile setup',
9   unprofiledAccount:
10     'Account exists but profile is not set up, continue with profile setup',
11   inactiveAccount: 'Account is inactive, contact support',
12   suspendedAccount: 'Account is suspended, contact support',
13   profileSetupCompleted: 'Profile setup completed successfully',
14   activeAccount: 'Account is already active, please sign in',
15   invalidAccountId: 'Invalid account ID',
16   profileAlreadySetup: 'Profile is already set up for this account',
17   accountAlreadySetup: 'This account has already been set up',
18   passwordChanged: 'Password changed successfully',
19   emailNotVerified: 'Email is not verified',
20   verificationEmailSent: 'Verification email sent',
21   emailAlreadyVerified: 'Email is already verified',
22   tooManyVerificationRequests: 'Too many verification email requests. Try again later.',
23 };

```

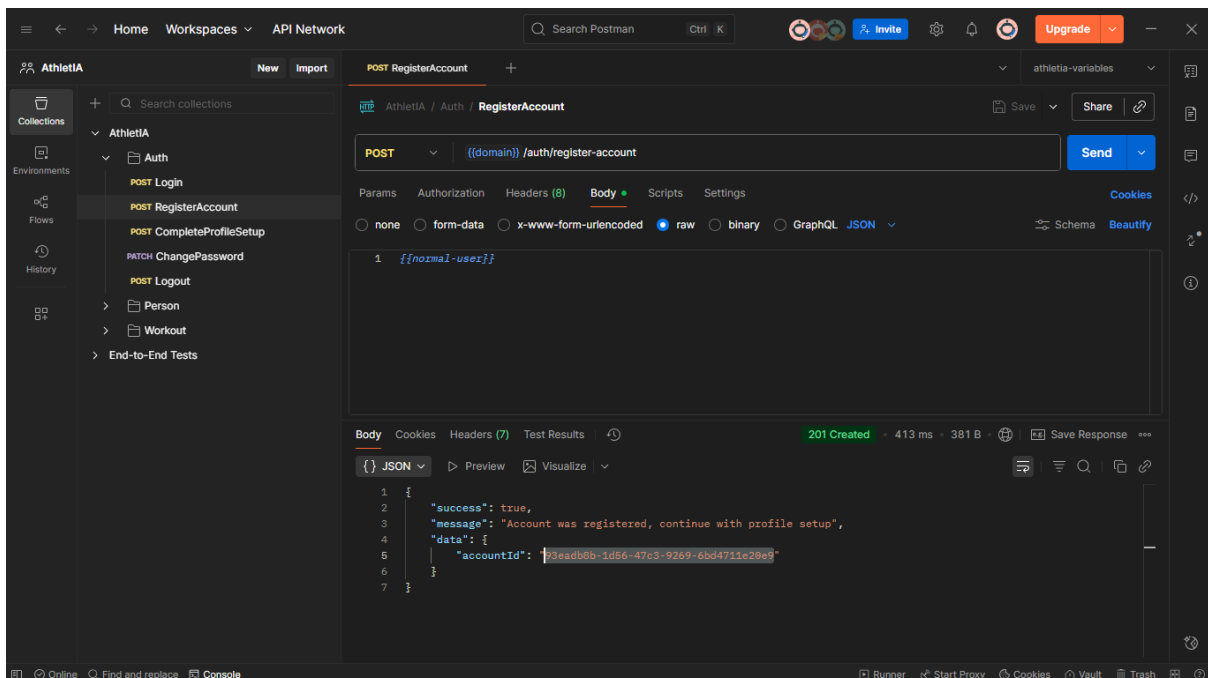

4. Pruebas y verificación

- 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

The screenshot shows a REST client interface with a sidebar on the left containing a collection of endpoints. The main panel displays a GET request to `{{domain}}/workout/exercises`. The response is a 200 OK status with a response time of 13 ms and a body size of 1.25 KB. The response body is shown in JSON format, indicating a successful retrieval of exercises.

```
{
  "success": true,
  "message": "Exercises retrieved successfully",
  "data": [
    {
      "id": "4a7198c0-81a4-4f49-a173-e964dabde127",
      "name": "I Row",
      "description": "Exercise for building a strong Back.",
      "video": "https://www.youtube.com/shorts/hmbUlkbsMsd",
      "createdAt": "2025-10-24T02:27:15.426Z",
      "updatedAt": "2025-10-24T02:27:15.426Z",
      "exerciseType": [
        "bodybuilding",
        "strength",
        "powerlifting"
      ]
    }
  ]
}
```

Código de Error 401

The screenshot shows a REST client interface with a sidebar on the left. The main panel displays a POST request to `{{domain}}/auth/login`. The request body is in raw format, containing email and password fields. The response is a 401 Unauthorized status with a response time of 141 ms and a body size of 931 B. The response body is shown in JSON format, indicating an unauthorized access attempt.

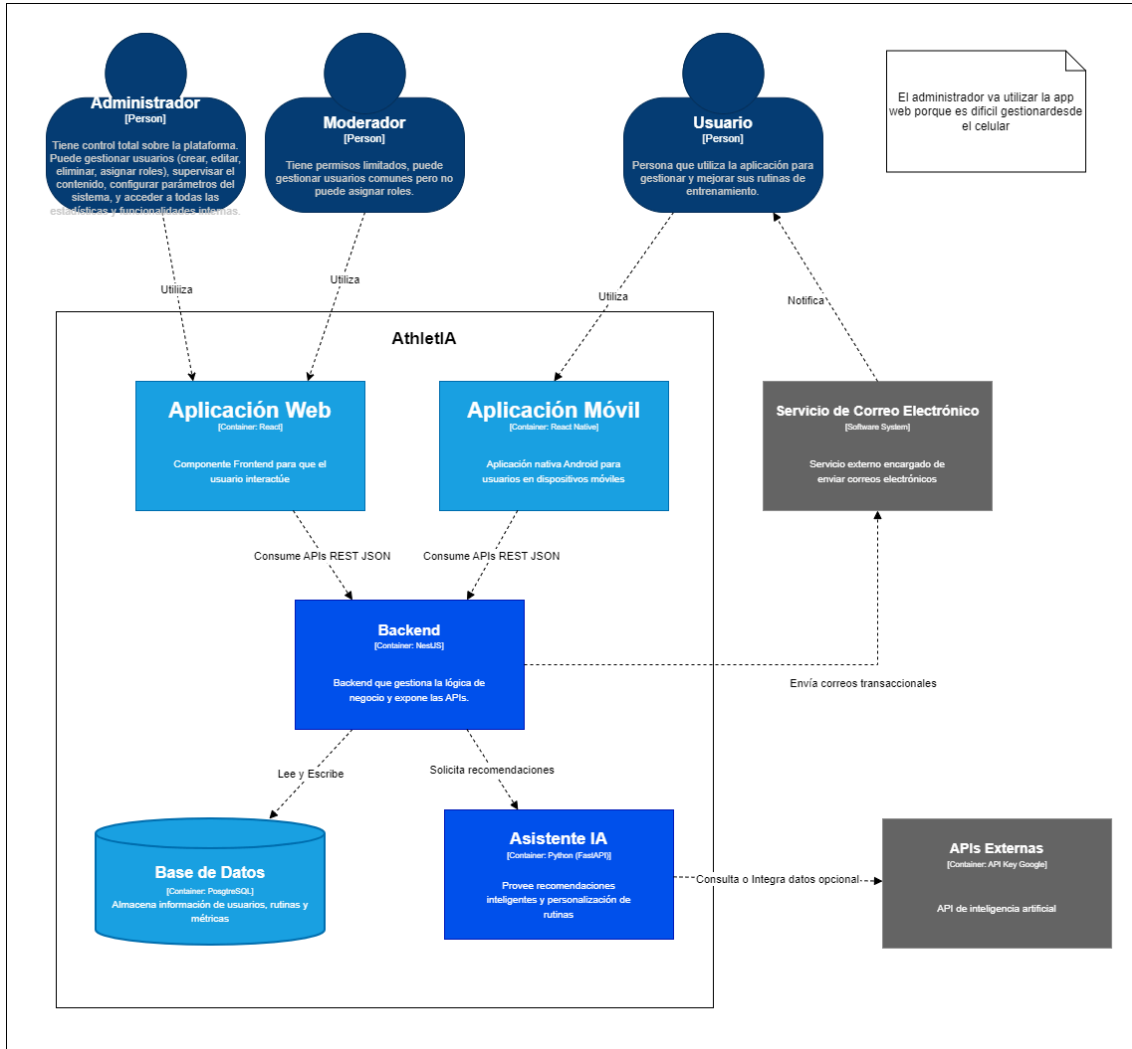
```
{
  "email": "admin.jgraso@email.com",
  "password": "administrator123"
}
```

```
{
  "message": "Unauthorized",
  "statusCode": 401
}
```

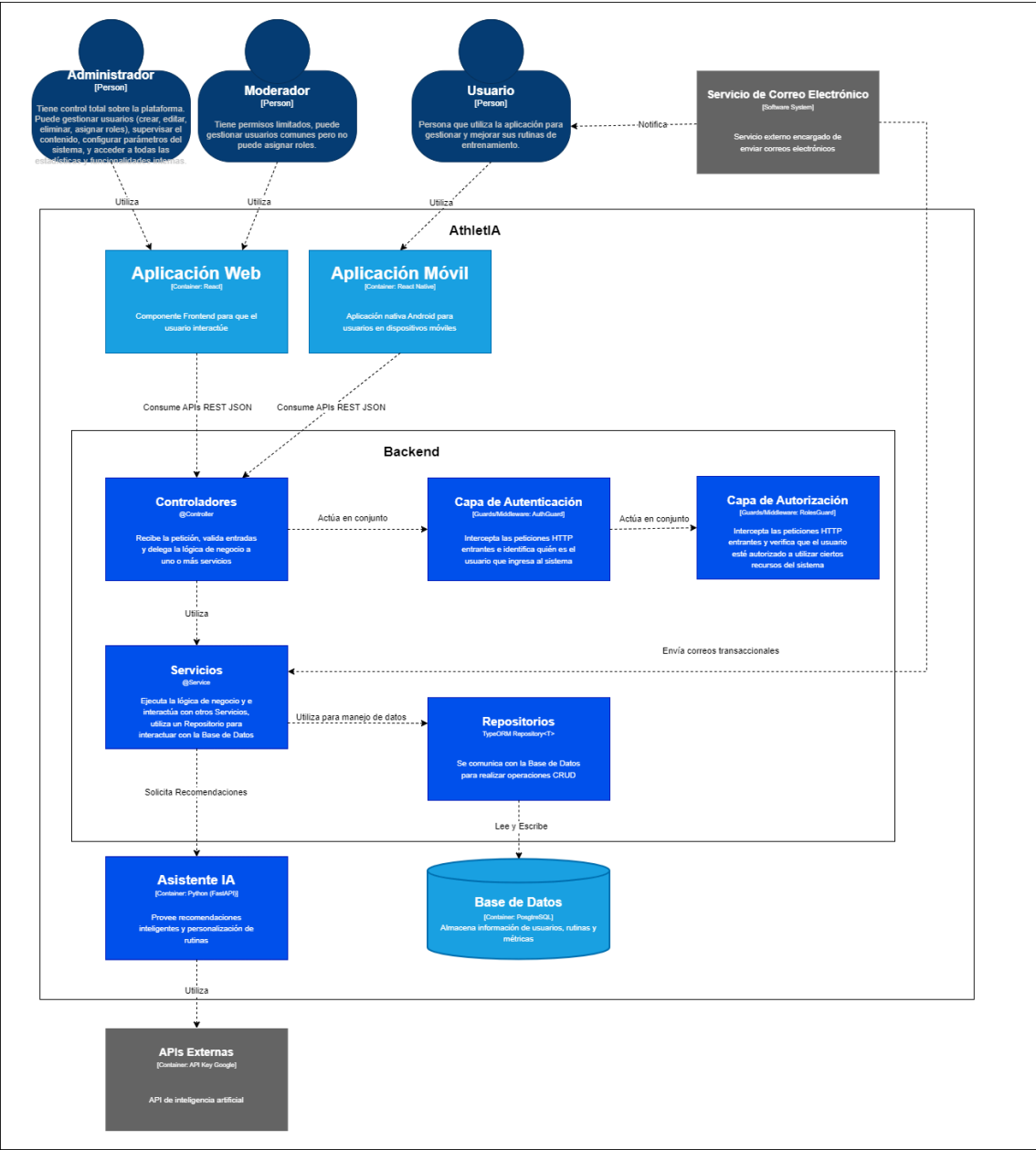
5. Modelo C4 con seguridad

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

Modelo C4 - Nivel 2 - Contenedor

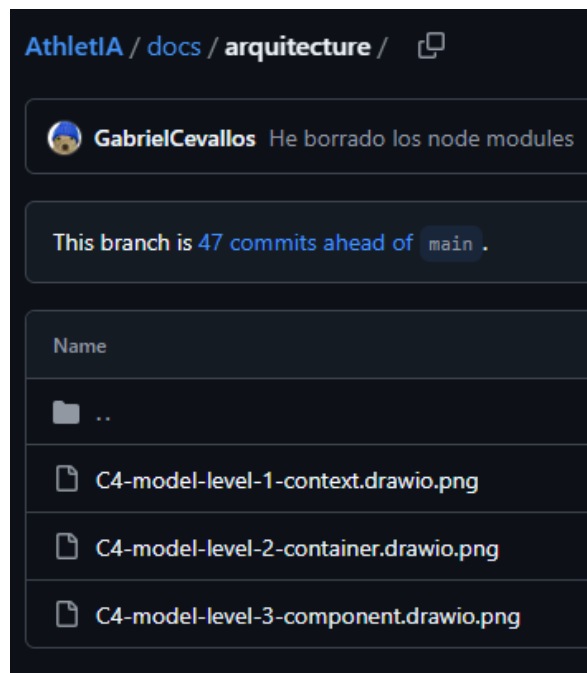


Modelo C4 - Nivel 3 - Component

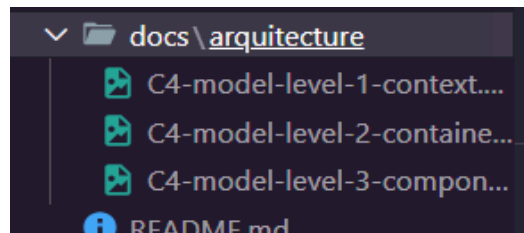


- Guardar los diagramas actualizados en </docs/architecture/>.

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:

- **Interoperabilidad:** 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 violación 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 ejecución de código/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 JWT 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	