Reporte Técnico de Actividades Práctico-Experimentales Nro. 001

1. Datos de Identificación del Estudiante y la Práctica

Nombre del estudiante(s)	Gabriel Ricardo Cevallos Medina Ivan Alexander Fernandez Cañar David Alexander Paccha Gallegos
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.	001
Título de la Práctica	Implementar un servicio REST con Node.js
Nombre del Docente	Edison Leonardo Coronel Romero
Fecha	Viernes 3 de octubre
Horario	07h30 – 10h30
Lugar	Laboratorio Computación aplicada Laboratorio Desarrollo de Software Laboratorio de redes y Sistemas Operativos Laboratorio Virtual EVA Aula
Tiempo planificado en el Sílabo	3 horas

2. Objetivo(s) de la Práctica

- Diseñar y desplegar un microservicio funcional que se comunique con el backend principal del proyecto.
- Implementar comunicación REST entre servicios utilizando un API Gateway o endpoint compartido.
- Documentar la arquitectura de microservicios en el modelo C4 y registrar evidencias del despliegue.

3. Materiales, Reactivos, Equipos y Herramientas

- Computador con acceso a Internet.
- Docker / Docker Compose.
- Repositorio del proyecto (GitHub/GitLab).

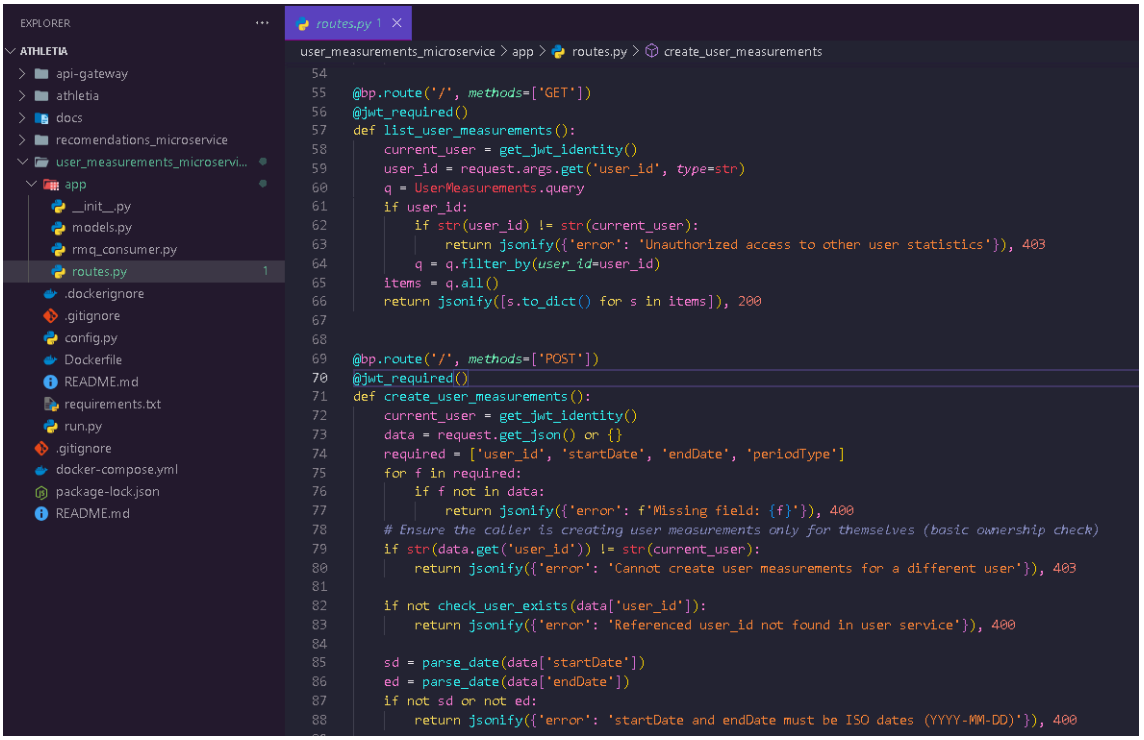
- Postman / Swagger.
- NestJS.
- Laboratorio de Desarrollo de Software o equipo personal.
- IDE: Visual Studio Code.
- Git.
- Terminal de comandos y contenedores Docker.
- Herramientas de modelado C4 (Draw.io).

4. Procedimiento / Metodología Ejecutada

4.1 Diseño del microservicio

- **Seleccionar una funcionalidad del proyecto que pueda desacoplarse:** Medidas Corporales del Usuario (UserMeasurements)
- **Definir su API interna y endpoints principales:**

API Interna



```python
@bp.route('/<stat_id>', methods=['GET'])
@jwt_required()
def get_user_measurements(stat_id: str):
    current_user = get_jwt_identity()
    stat = UserMeasurements.query.get_or_404(stat_id)
    if str(stat.user_id) != str(current_user):
        return jsonify({'error': 'Unauthorized access to this statistics resource'}), 403
    return jsonify(stat.to_dict(include_metrics=True)), 200


@bp.route('/<stat_id>', methods=['PUT'])
@jwt_required()
def update_user_measurements(stat_id: str):
    current_user = get_jwt_identity()
    stat = UserMeasurements.query.get_or_404(stat_id)
    if str(stat.user_id) != str(current_user):
        return jsonify({'error': 'Unauthorized to update this statistics resource'}), 403
    data = request.get_json() or {}
```

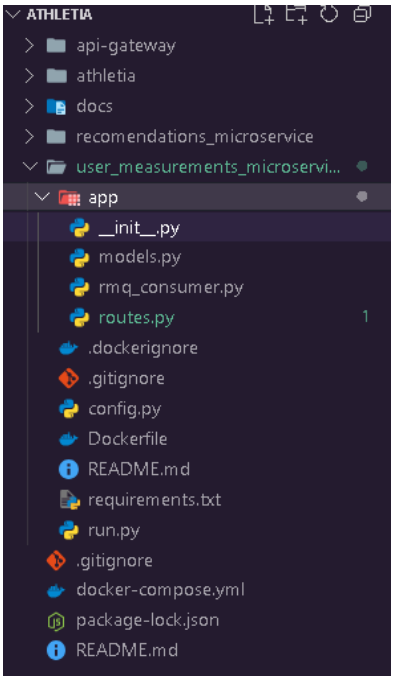
```python
@bp.route('/<stat_id>', methods=['DELETE'])
@jwt_required()
def delete_user_measurements(stat_id: str):
    current_user = get_jwt_identity()
    stat = UserMeasurements.query.get_or_404(stat_id)
    if str(stat.user_id) != str(current_user):
        return jsonify({'error': 'Unauthorized to delete this statistics resource'}), 403
    try:
        db.session.delete(stat)
        db.session.commit()
    except Exception as e:
        db.session.rollback()
        return jsonify({'error': 'Database error', 'details': str(e)}), 500
    return jsonify({'deleted': stat_id}), 200

```

4.2 Configuración del entorno

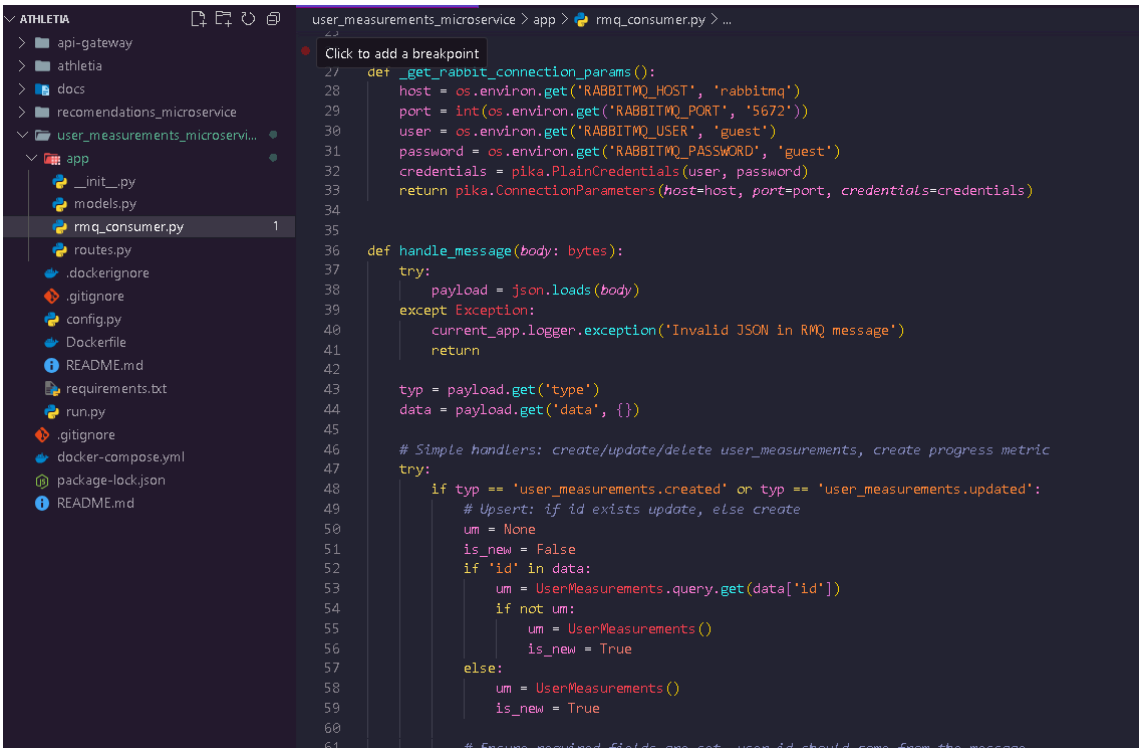
- **Crear una nueva carpeta <nombre>_microservice y configurarlo como módulo independiente:**

user_measurements_microservice



- **Implementar la estructura base con servicios y rutas.**

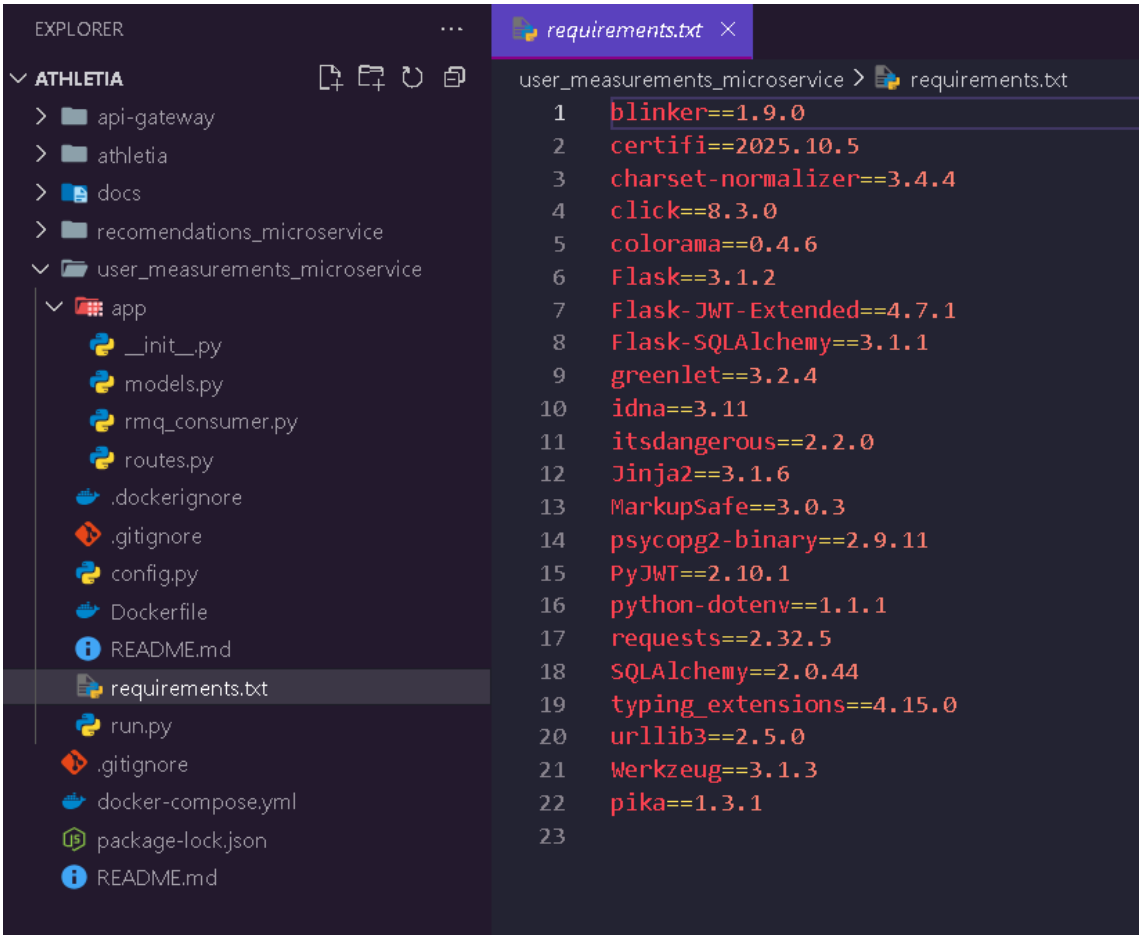
Servicio RabbitMQ



- **Configurar variables de entorno, dependencias y documentación básica (README).**

Variables de Entorno

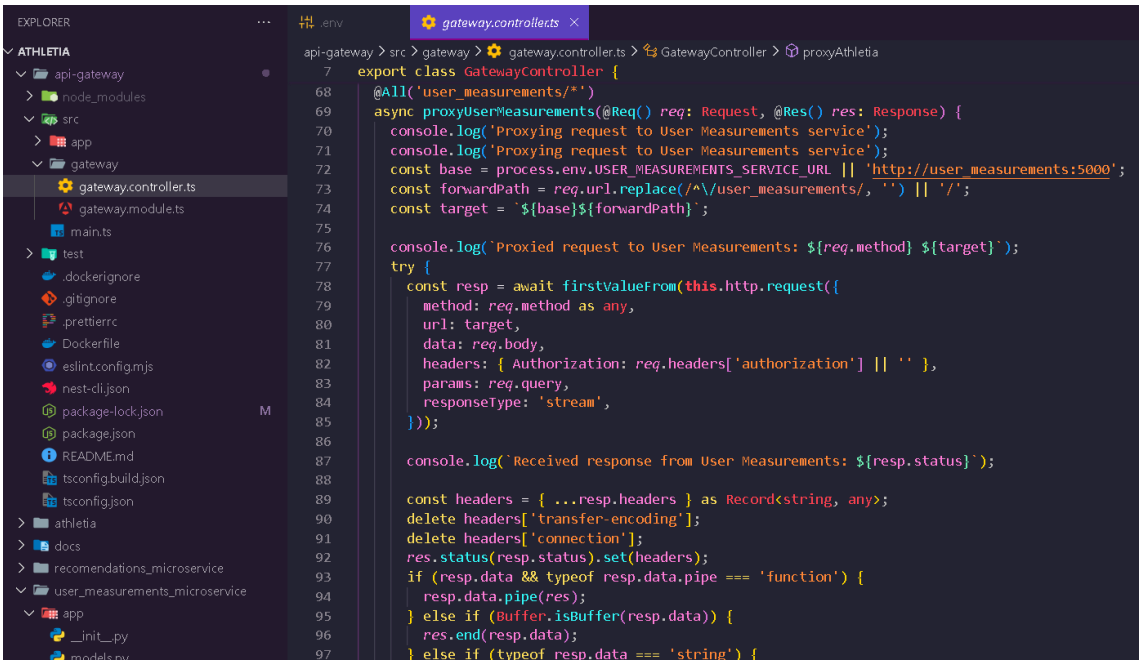
Dependencias



4.3 Comunicación entre servicios

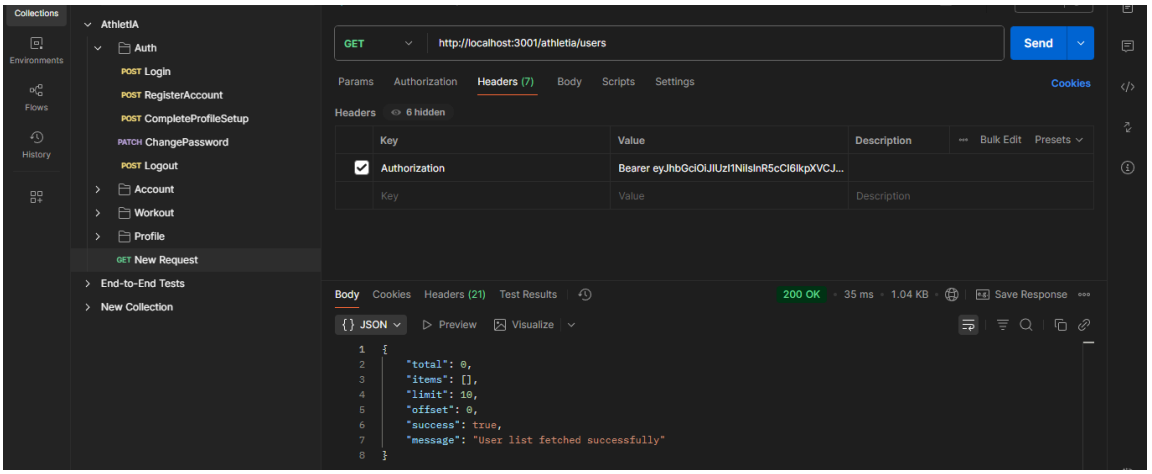
- **Configurar API Gateway o endpoint del backend principal para enlazar el nuevo servicio.**

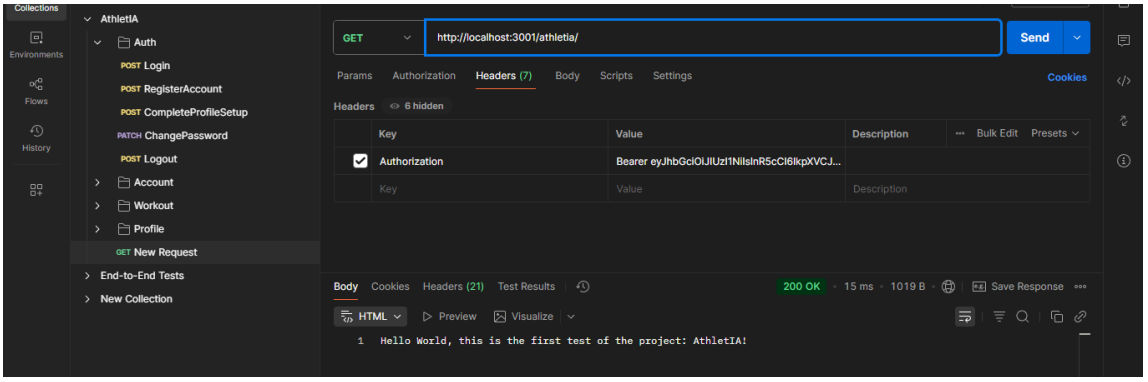
API Gateway

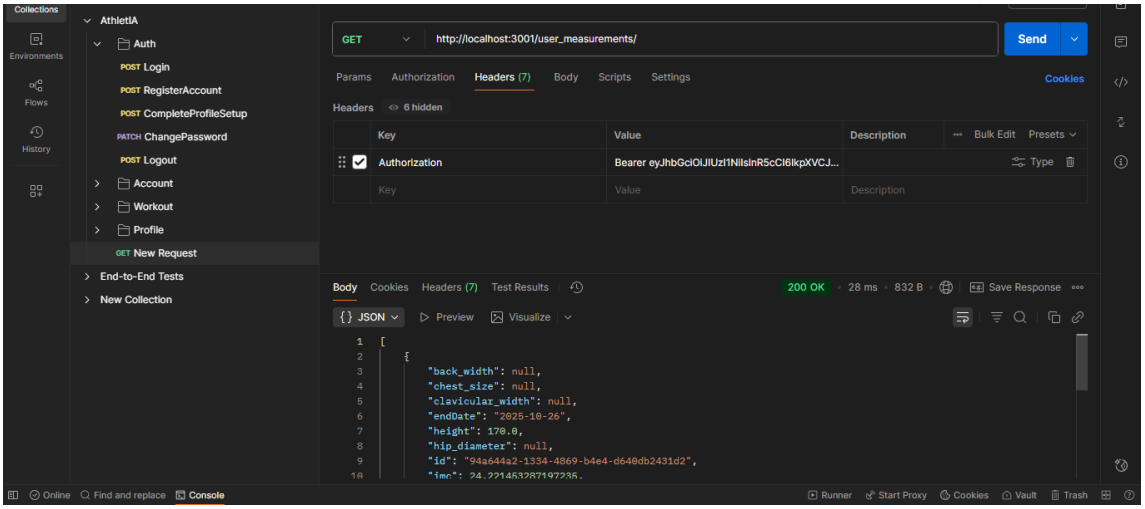


- **Probar el intercambio de datos mediante peticiones HTTP.**

Peticiones HTTP mediante Postman

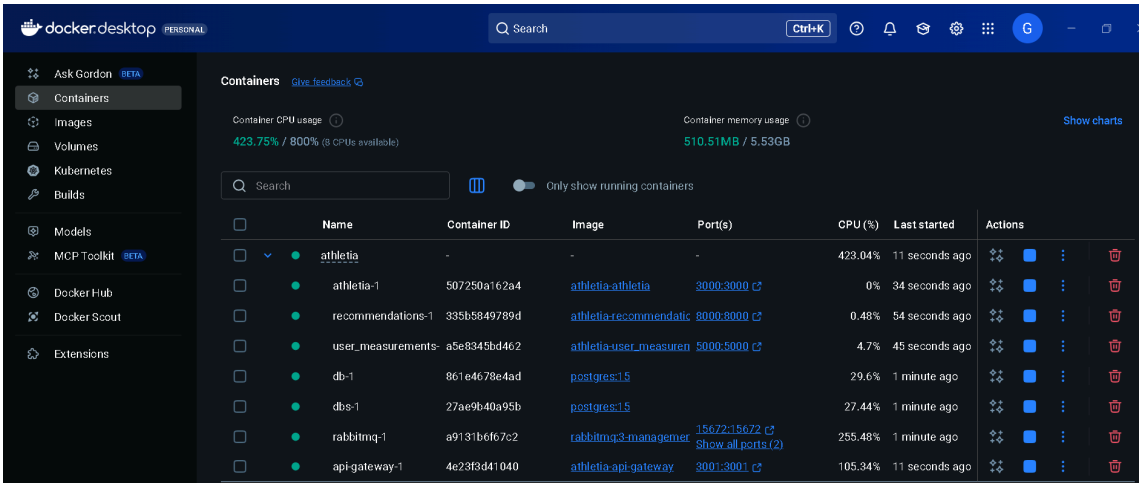






4.4 Despliegue y validación

- **Ejecutar el microservicio en contenedor Docker o entorno local.**

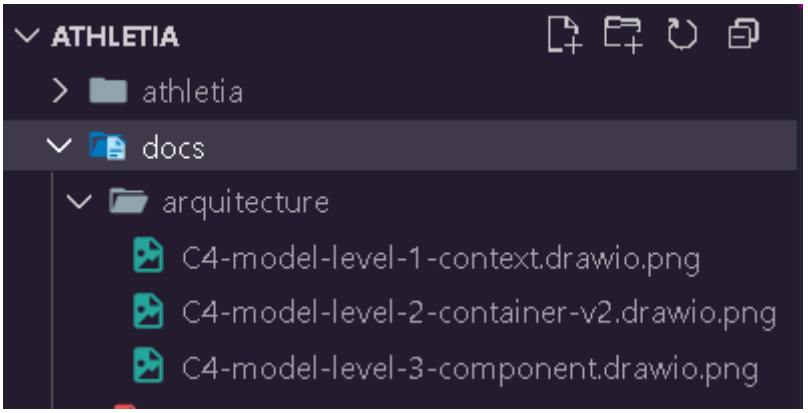


- **Validar la conexión y registrar logs de comunicación.**

Core (AthletIA) conectado con servicio RabbitMQ

```
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RouterExplorer] Mapped {/workout/exercises, POST} route +1ms
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RouterExplorer] Mapped {/workout/exercises, GET} route +0ms
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RouterExplorer] Mapped {/workout/exercises/:id, GET} route +2ms
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RouterExplorer] Mapped {/workout/exercises/:id, PATCH} route +1ms
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RouterExplorer] Mapped {/workout/exercises/:id, DELETE} route +1ms
athletia-1  | [Nest] 1  - 10/27/2025, 12:32:53 AM     LOG [RabbitmqService] Connecting to RabbitMQ rabbitmq:5672
```

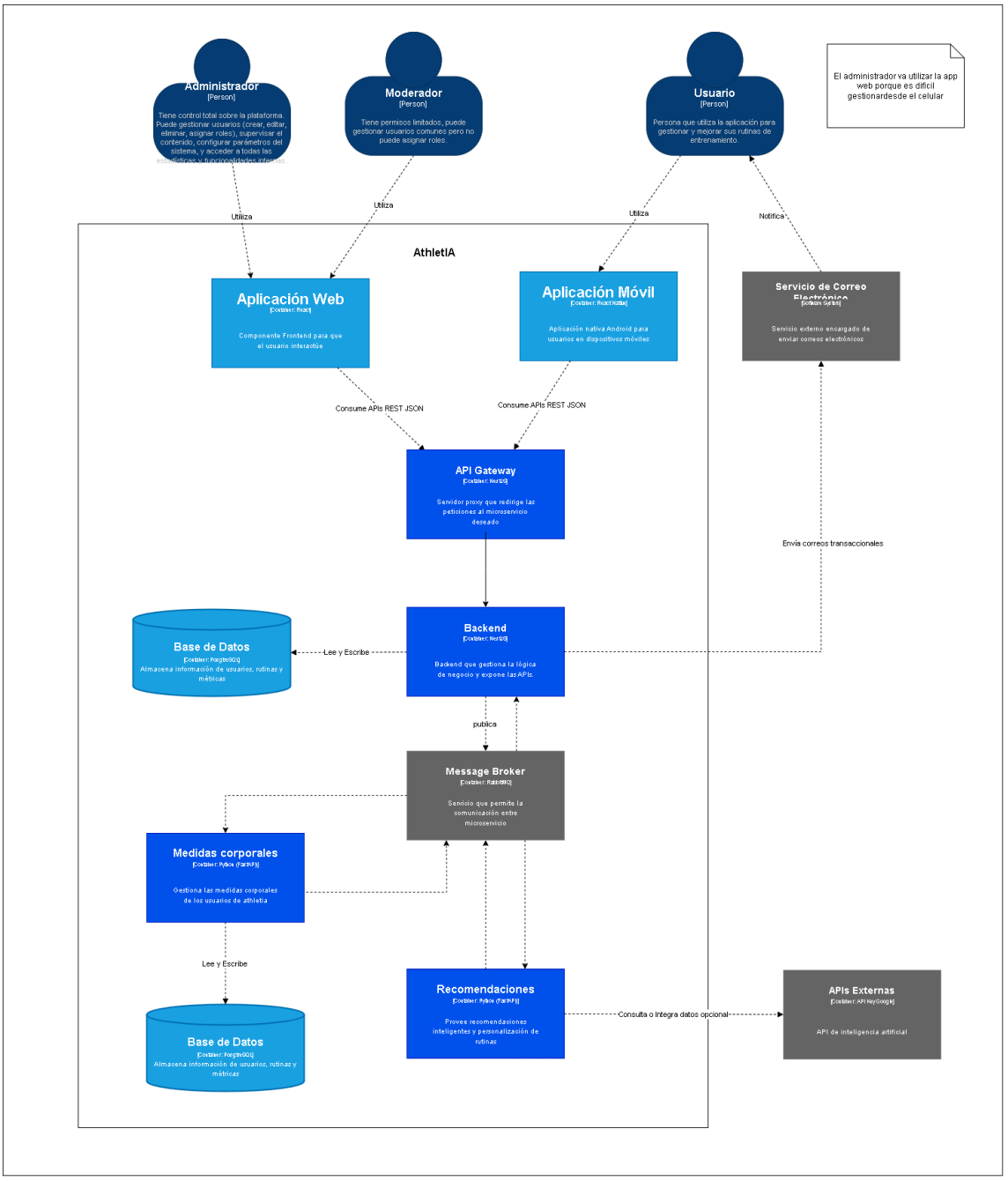
- **Documentar resultados en /docs/architecture/container-diagram-v2.png.**



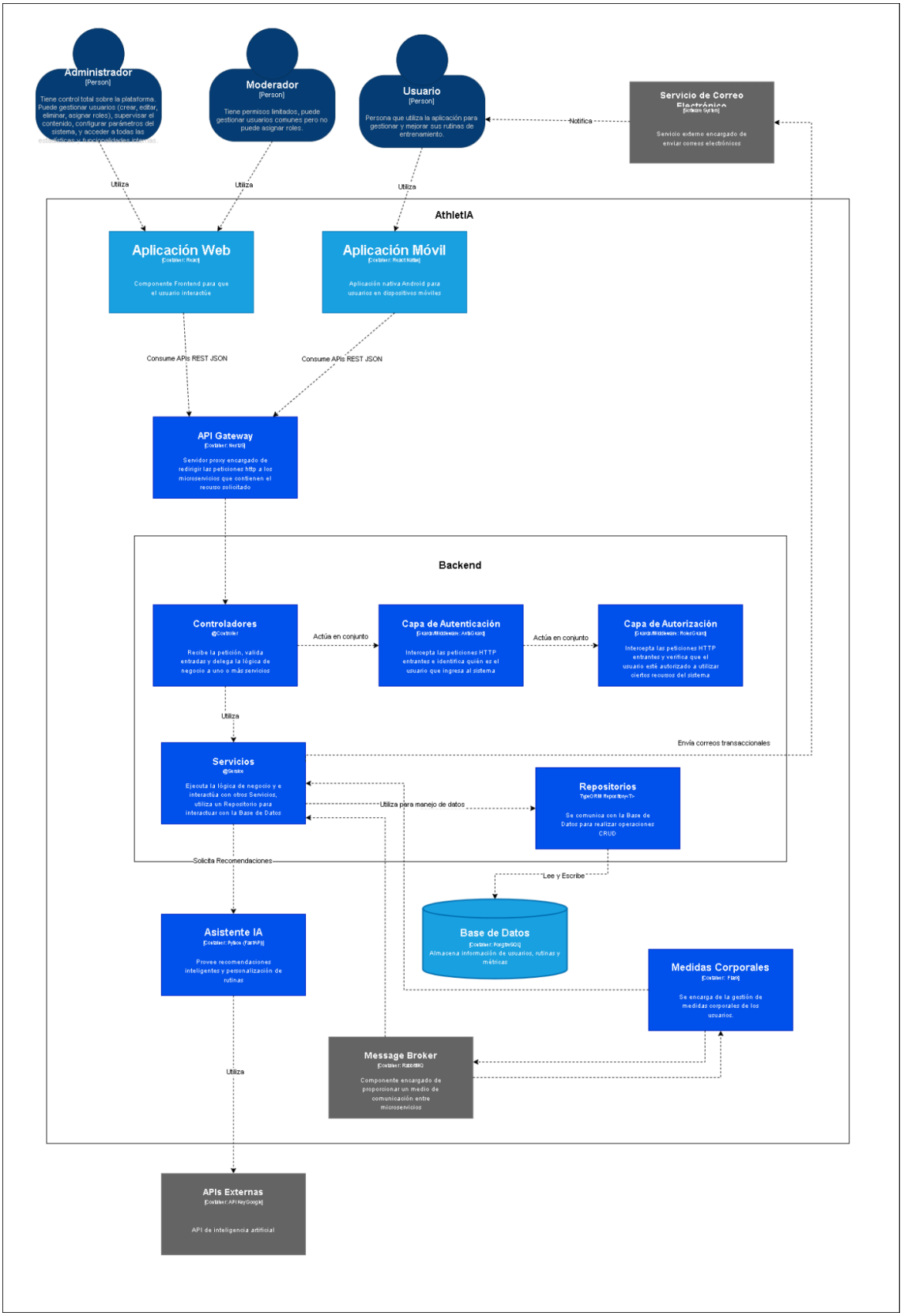
4.5 Registro en modelo C4

- **Actualizar los diagramas Container y Component incorporando el microservicio y el gateway.**
- **Registrar capturas o exportar los diagramas actualizados.**

MODELO C4 - NIVEL 2 - CONTAINER



MODELO C4 - NIVEL 3 - COMPONENT



5. Resultados

- Microservicio desplegado y comunicándose correctamente con el backend principal.
- API Gateway o punto de integración funcional.
- Documentación técnica y diagramas C4 actualizados.
- Logs y capturas de comunicación.

6. Preguntas de Control

- **¿Qué diferencia principal existe entre una arquitectura monolítica y una basada en microservicios?**

  Una arquitectura monolítica es un sistema todo en uno, es decir que está unido en un solo bloque de código, mientras que la arquitectura basada en microservicios se divide en servicios independientes, es decir que si uno deja de funcionar no le afecta a los demás y pueden seguir funcionando correctamente.

- **¿Qué beneficios aporta el uso de un API Gateway en la integración de servicios?**

  El uso de un API Gateway ayuda a mejorar significativamente la seguridad, escalabilidad y la observabilidad, al tiempo que se desacopla los clientes de los servicios del backend, permitiendo que los desarrolladores se concentren en la lógica del negocio.

- **¿Qué riesgos se presentan al no manejar correctamente la comunicación entre microservicios?**

  Esto puede ocasionar riesgos como fallas en cascada, latencia de red, inconsistencia de datos, complejidad en la depuración y monitoreo, y vulnerabilidades de seguridad.

- **¿Cómo se refleja la modularidad en los niveles Container y Component del modelo C4**

  Esto se refleja al descomponer el sistema en niveles jerárquicos. El nivel de Contenedor muestra las divisiones físicas o lógicas de alto nivel, mientras que el nivel de Componentes detalla la estructura interna de cada contenedor.

- **¿Qué consideraciones de seguridad se deben mantener al exponer endpoints entre servicios?**

  Es muy importante implementar medidas de seguridad como la autenticación y autorización robustas, el cifrado de datos, el uso de APIs seguras y la implementación de una arquitectura de confianza cero. Además es crucial realizar monitoreos continuos y registrar los accesos, mantener los sistemas actualizados y parcheados, también tener un inventario de APIs para identificar los riesgos.

7. Conclusiones

- En esta práctica se diseñó y se implementó un microservicio para medidas corporales de los usuarios, integrado con el backend principal de la aplicación mediante API Gateway y RabbitMQ.
- Se validó la comunicación REST con herramientas como Postman y Docker, y se actualizó el documento C4 con diagramas de contenedores.

8. Recomendaciones

- Incorporar pruebas automatizadas para detectar errores tempranos.
- Usar monitoreo avanzado en producción para logs en tiempo real.
- Aplicar seguridad adicional para endpoint.
- extender el modelo C4 a niveles más detallados y probar despliegues en la nube para escenarios reales.