

PONTIFICIA UNIVERSIDAD JAVERIANA

ARQUITECTURA DE SOFTWARE

Laboratorio 2 – Patrón Hexagonal

Jorge Torrado
Juan Andrés Ramírez
Sofía Céspedes

Ingeniería de Sistemas
24/05/2025
Bogotá D.C

Marco Conceptual.....2

| | |
|----------------------------|----|
| Diseño | 7 |
| Procedimiento | 13 |
| Conclusiones | 51 |
| Lecciones Aprendidas | 52 |
| Referencias..... | 53 |

Marco Conceptual

1. Stack de Patrones:

El stack respecto a los patrones para este proyecto consiste en las siguientes (para cada una se explicarán brevemente sus características):

Arquitectura Hexagonal:

La arquitectura hexagonal, también conocida como Ports and Adapters, busca desacoplar el núcleo de la aplicación (la lógica de negocio) de sus mecanismos externos como interfaces de usuario, bases de datos o servicios externos. Esto lo logra, como su nombre indica, mediante puertos y adaptadores, los cuales son la única forma de acceder a los otros componentes y comunicarlos entre sí.

Sus características principales son:

- El núcleo de la aplicación se comunica con el mundo exterior únicamente a través de "puertos" (interfaces) y los "adaptadores" (implementaciones).
- Facilita la inversión de dependencias, permitiendo que componentes externos como bases de datos o APIs sean fácilmente intercambiables, pues no depende de implementaciones sino de abstracciones (las interfaces de los puertos en este caso).
- Favorece pruebas unitarias puras sin necesidad de entornos completos gracias a la gran modularidad que presenta.

Pros y contras en la Arquitectura Hexagonal:

| Pros | Contras |
|---|---|
| Bajo acoplamiento y alta cohesión: Al estar separado en varias capas y solo permitir comunicación entre capas a partir de los puertos, se presenta un gran desacoplamiento que facilita la modularidad y alta cohesión entre los componentes, facilitando así la modificación y mantenimiento del sistema a futuro. | Mayor cantidad de clases y configuración: esto dificulta la implementación inicial por toda la cantidad de pasos extra y código que se debe hacer en los primeros pasos del proyecto, pero a futuro facilita la mantenibilidad |
| Facilita el testing, mantenimiento y evolución del sistema: Gracias a la modularidad que aporta el hexagonal (explicada anteriormente), el sistema se vuelve mucho más manejable a futuro en cuanto a mantenimiento evolución, testing y reutilización. | Curva de aprendizaje más alta comparada con otras arquitecturas como MVC: Al tener más componentes y conceptos nuevos como puertos y adapters, se requiere más tiempo para aprender y adaptarse al hexagonal a comparación de patrones más simples como MVC (que solo tenía únicamente manejan 3 capas) |
| Permite múltiples interfaces (REST, CLI, SOAP, etc.) sin afectar el núcleo: Gracias a que | Mayor complejidad inicial en proyectos pequeños: La implementación de esta |

| | |
|---|---|
| el núcleo está muy desacoplado del resto de la app, se pueden usar distintas interfaces/protocolos de comunicación sin alterar la lógica de negocio | arquitectura puede resultar innecesariamente compleja y sobreestructurada para aplicaciones simples o con pocos requisitos, cayendo en el error (overengineering) |
|---|---|

Patrón Repository:

El patrón Repository tiene como finalidad funcionar como una abstracción del acceso a datos, encapsulando así la lógica necesaria para acceder a la fuente de datos (María para SQL y Mongo para noSQL en este caso). El punto fuerte de este patrón es que permite desacoplar la lógica de negocio de los detalles del almacenamiento, facilitando así cambios en uno sin alterar el otro (cambiar la tecnología de persistencia o credenciales de la BD, sin alterar la lógica de negocio y viceversa: cambiar lógica de negocio sin que afecte la lógica de acceso a datos).

Sus características principales son:

- Oculta la lógica de acceso a la base de datos y presenta una interfaz limpia y coherente para trabajar con objetos de dominio.
- La capa de dominio no necesita saber cómo se almacenan o recuperan los datos (SQL, NoSQL, archivos, etc.).
- Al depender de interfaces, es fácil sustituir la implementación real por una versión mock o in-memory en pruebas unitarias.
- Centraliza el acceso a datos en un único lugar, lo que facilita aplicar cambios (como cambiar de base de datos o de framework) sin afectar otras capas como la de negocio.
- Es muy compatible con SOLID, en especial con la “Dependency Inversion”, pues no dependes de una implementación, sino únicamente de la interfaz del repositorio
- Viene con varios métodos por default como los findBy y el save, además de permitirte crear los tuyos propios dependiendo de las necesidades del proyecto.

Pros y contras en el Patrón Repository:

| Pros | Contras |
|------|---------|
|------|---------|

| | |
|---|---|
| Aísla la lógica de acceso a datos de la lógica del dominio. | Introduce capas adicionales, lo que puede resultar innecesario para proyectos pequeños. (overengineering) |
| Facilita cambiar de motor de base de datos (Como es en este caso el cambio entre MongoDB y MariaDB según se requiera) sin afectar el core de la app (su lógica de negocio). | |
| Mejora el testing al permitir inyectar repositorios falsos (mocks). | |

Patrón Service:

El patrón Service organiza la lógica de negocio en clases que agrupan operaciones relacionadas con entidades específicas, sirviendo como puente entre controladores y repositorios.

Sus características principales son:

- Contiene lógica transaccional y validaciones complejas.
- Facilita la organización modular del código.
- Promueve la reutilización de lógica de negocio.

Pros y contras en el Patrón Service:

| Pros | Contras |
|---|---|
| Permite centralizar la lógica de negocio en una capa específica, separándola de las capas de presentación (como los controladores) y de acceso a datos (repositorios). Esto mejora la cohesión y facilita el mantenimiento. | Al igual que con Repository, para aplicaciones pequeñas o prototipado, agregar esta capa adicional puede resultar siendo más complicado que útil (overengineering) |
| Facilita la reutilización de código en varias partes de la aplicación, esto ya que los servicios pueden ser llamados por diferentes controladores, lo que evita duplicación de código y aumenta la modularidad. | Introduce una capa extra de abstracción que puede hacer que el diseño sea más complejo, especialmente cuando hay muchos servicios que gestionan distintas funcionalidades, lo cual puede resultar en más clases y más código para gestionar, dificultando así su lectura. |
| Los servicios encapsulan la lógica de negocio y se pueden probar de manera independiente (unitariamente), sin depender de la infraestructura externa como la base de datos o la interfaz de usuario. Esto mejora la testabilidad. | Los servicios deben ser manejados con buen criterio y siendo cuidadoso con el dev, si no se terminaran duplicando código o creando servicios demasiado grandes que favorezcan el acoplamiento en vez de la cohesión |
| Los servicios pueden ser fácilmente modificados o extendidos sin afectar el resto | |

| | |
|--|--|
| de la aplicación. Si es necesario modificar la lógica de negocio o añadir nuevas funcionalidades, el servicio centralizado facilita esta evolución de forma aislada. | |
| Es de fácil integración con otros patrones como el Repository | |

2. Stack Tecnológico:

El stack tecnológico propuesto para este proyecto consiste en las siguientes herramientas (para cada una se explicarán brevemente sus características):

JDK 11:

- Presenta long term support (no se corre el riesgo de que lo manden a end of life en un futuro cercano)
- Es muy compatible con Spring boot versiones 2.x en adelante
- No tiene tantas funcionalidades como versiones más nuevas y estables como lo sería por ejemplo

JDK

17

Spring Boot:

- Tiene una gran variedad de herramientas con las que se puede integrar con facilidad, como por ejemplo Swagger 3 en este caso (y también Mongo y María)
- Gracias a su auto configuración se pueden realizar implementaciones más rápido de lo que normalmente tomaría en otros frameworks
- Al “ocultar” varias cosas al usuario con su auto configuración y transparencia, a veces puede ser difícil rastrear errores si no se tiene un buen dominio del framework

MariaDB:

- Su esquema SQL facilita el almacenamiento de datos con fuertes relaciones entre sí (datos estructurados)
- Muy buena para garantizar disponibilidad mediante replicación.
- Puede ser algo lenta para consultas sobre estructuras de datos muy jerárquicas o con muchas relaciones si no se tiene buen manejo de índices.

MongoDB:

- Al ser noSQL es una buena alternativa para almacenamiento de datos no estructurados.
- Muy rápido para lectura y escritura.
- Permite escalabilidad horizontal (replicación).
- Al igual que las otras noSQL, esta no permite garantizar integridad estructural (pues no hay relaciones).

- Si no se maneja con buen criterio pueden aparecer datos duplicados o fantasmas.

REST y CLI:

- Es muy fácil de entender e implementar
- Es ampliamente usando, en especial para aplicaciones web
- Posee gran interoperabilidad (cualquier cliente que pueda entender http puede consumir las API's RESTful)
- En sistemas complejos puede ser un poco lento, requiriendo alternativas más rápidas como GraphQL
- No tiene estándares tan claros respecto a seguridad, lo cual puede poner en riesgo el sistema entero si no se aborda este tema de manera apropiada.

Swagger 3 (OpenAPI):

- Permite generar documentación de manera automática de la API
- Presenta una interfaz muy user-friendly facilitando a los devs el hecho de familiarizarse e interactuar con las API's
- Al permitir tener una referencia clara de la API y los endpoints desde un inicio, facilita la integración y trabajo en paralelo (de los devs de back y front)
- Depende fuertemente del formato de OpenAPI, por lo que limita mucho la personalización

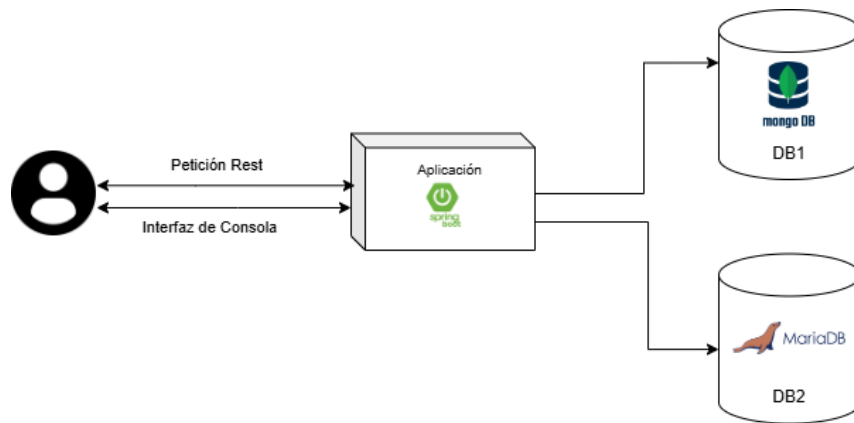
Docker y Docker compose:

- son herramientas fundamentales para la contenerización y orquestación de servicios dentro del entorno de desarrollo y producción.
- Su inclusión en este stack tecnológico permite simplificar la configuración, despliegue y mantenimiento del sistema, haciendolo así más portable y fácil de usar.
- La principal diferencia entre docker y docker compos es que el primero permite crear imagenes y contenedores mientras que Docker Compose se encarga de definir y gestionar múltiples contenedores a la vez que conforman una aplicación completa, mediante un único archivo de configuración (docker-compose.yml).

Diseño

Diagrama de Alto nivel:

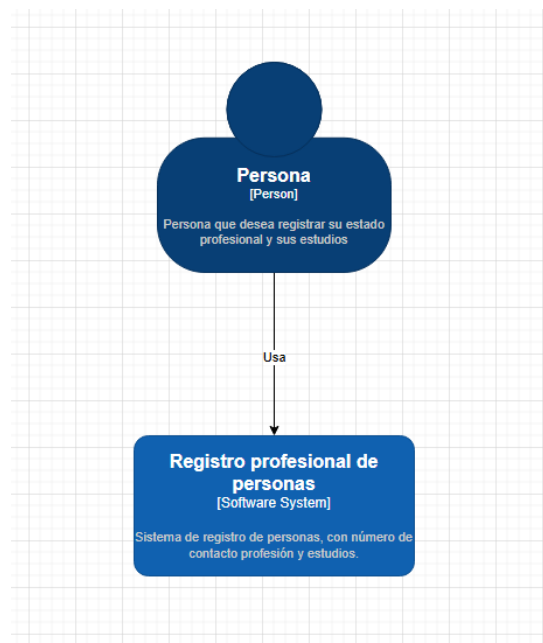
https://drive.google.com/file/d/1HOP88ow_HOc79MQK9Ye4BLOnQuFMxdsa/view?usp=sharing



En este diagrama se puede apreciar la vista de alto nivel del sistema, con una aplicación hexagonal en springboot interactuando con sus respectivas bases de datos (mongo y maría) a petición del usuario, quien se comunica con la app mediante peticiones REST o interfaz de consola

Diagrama de contexto:

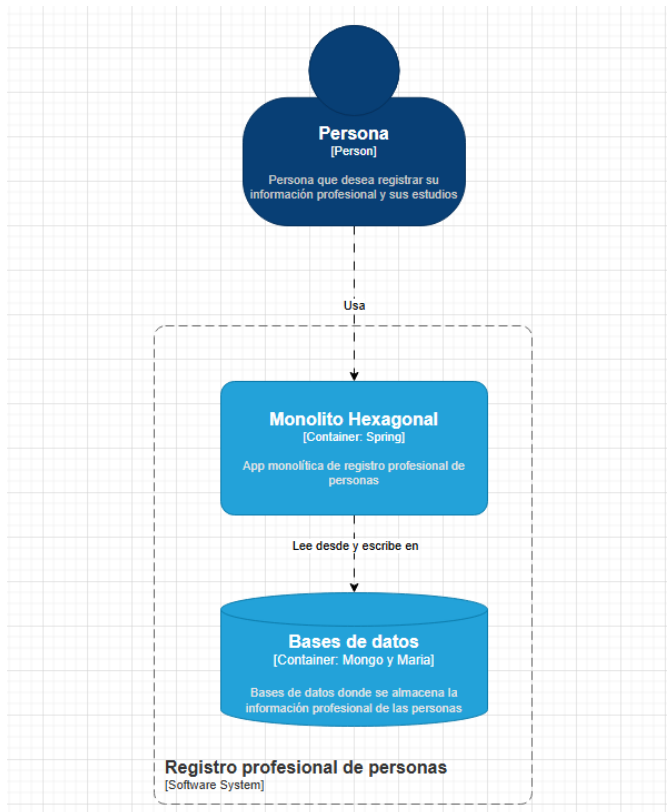
<https://drive.google.com/file/d/1jHdU-UKzw5IEd61-FXqeGuSKDo9QMIN/view?usp=sharing>



En el diagrama anterior se puede evidenciar la interacción del usuario (persona) con el sistema de registro profesional, en el cual puede registrar, consultar, actualizar y eliminar información relevante como lo son: sus datos personales, datos profesionales y datos académicos

Diagrama de contenedor:

https://drive.google.com/file/d/13SLcO3UBXLYOcuO12kJgXdDbjIX_Un-u/view?usp=sharing

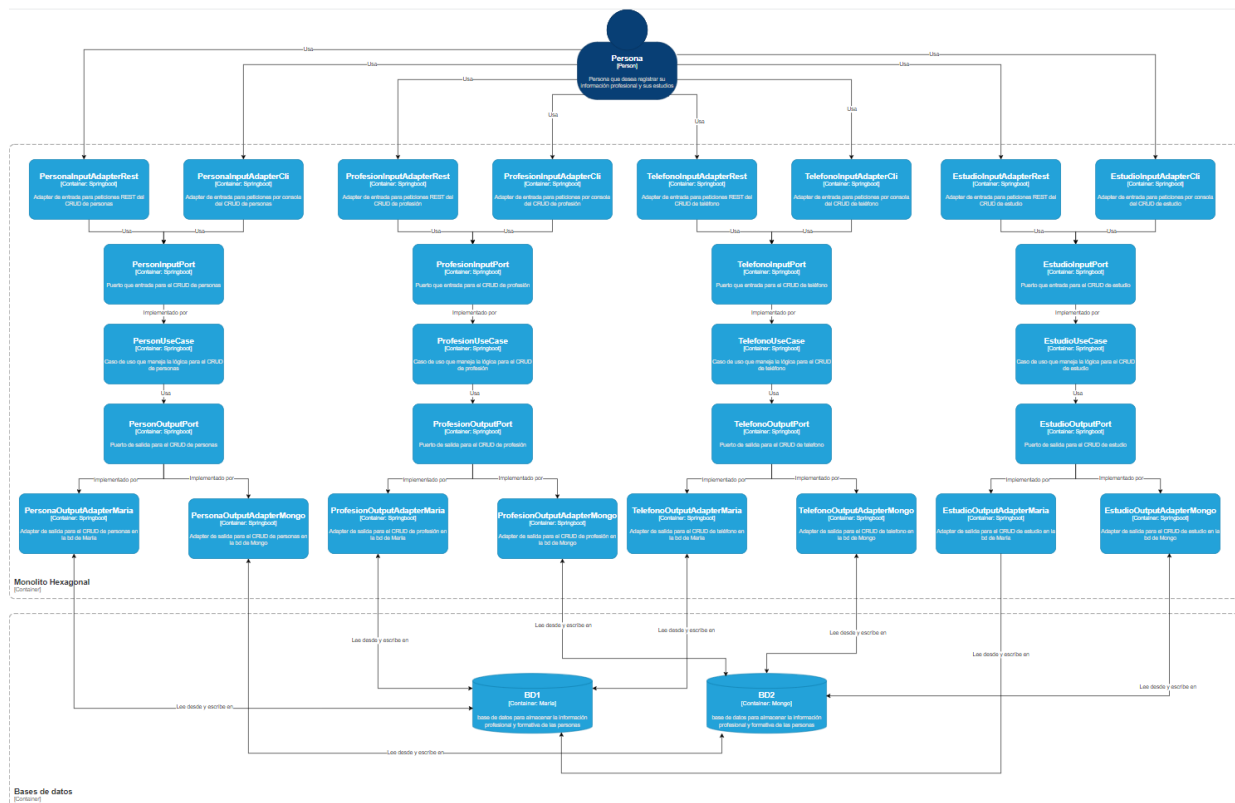


En este diagrama de contexto, se puede apreciar cómo se encuentra conformado el sistema:

- **Aplicación monolito hexagonal:** Contiene todo lo relacionado con la lógica de negocio, además de las interfaces requeridas para la comunicación del usuario con la aplicación y de la aplicación con las bds.
- **Bases de datos:** Contiene las bases de datos en las cuales el sistema registra la información, específicamente en el contexto de este laboratorio, son 2: Una base de datos con Mongo y otra con María

Diagrama de componentes

<https://drive.google.com/file/d/1BvsDQY2RQRKA-TfjGVyidEnPMeJ1AXp4/view?usp=sharing>



En este diagrama se evidencian a detalle los componentes del sistema, los cuales son:

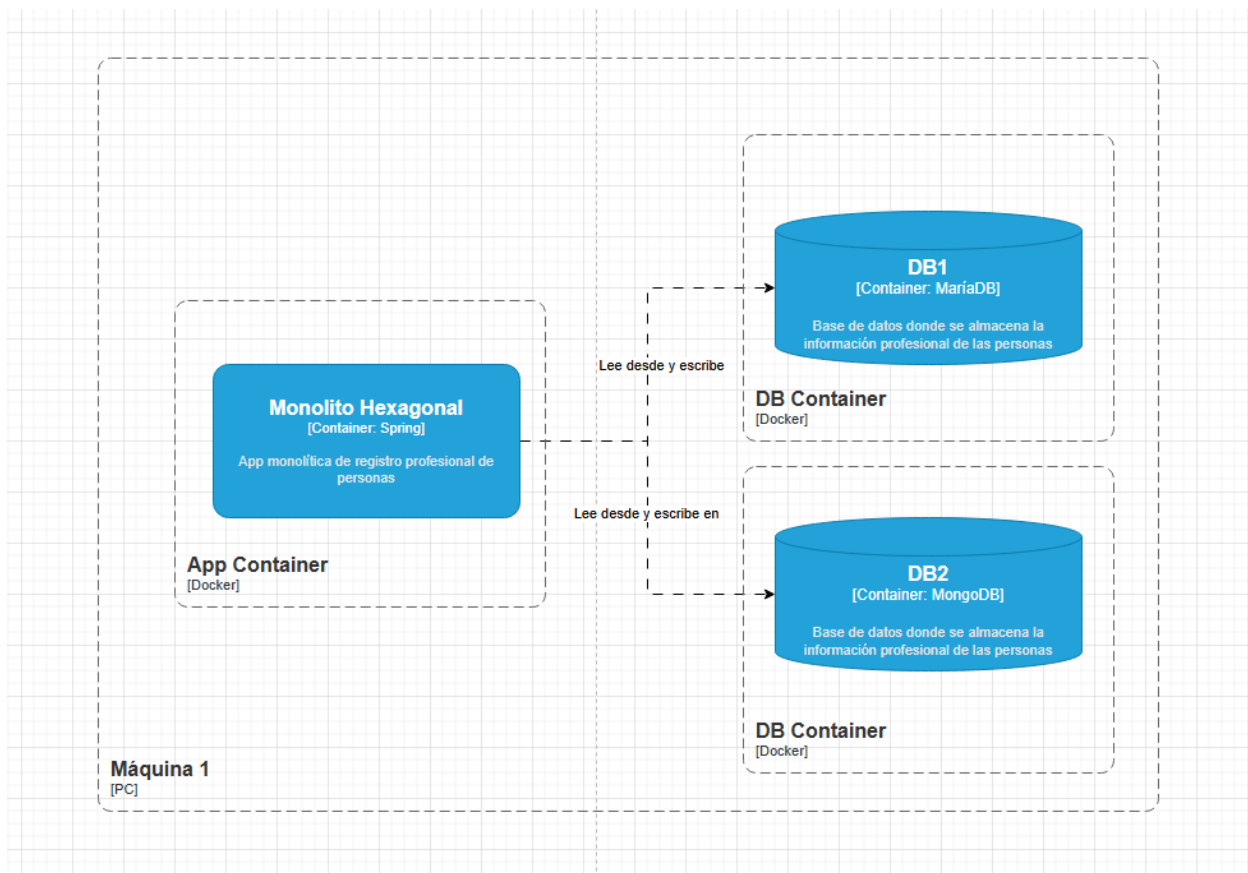
- **Adaptadores de entrada:** Permiten la interacción del usuario con el sistema, implementando los puertos de entrada para traducir las solicitudes externas en comandos comprensibles por la aplicación. En este caso en concreto, se clasifican en:
 - **Adaptador CLI:** Permite la interacción a través de la línea de comandos, útil para entornos de desarrollo o administración técnica.
 - **Adaptador REST:** Exposición de endpoints HTTP para que clientes web, móviles u otros sistemas consuman la lógica del sistema mediante una API.
- **Puertos de entrada:** Interfaces que definen los métodos disponibles para los adaptadores de entrada. Establecen un contrato claro entre la capa externa y los casos de uso, asegurando que la lógica de aplicación sea invocada correctamente.
- **Casos de Uso:** Contienen la lógica de aplicación. En este sistema, su responsabilidad principal es realizar las operaciones CRUD (crear, leer, actualizar, eliminar) sobre las distintas entidades del dominio.

- Adaptadores de salida: Permiten la interacción con los sistemas de persistencia, implementando los puertos de salida. Se dividen según el tipo de base de datos:
 - **Adaptador MariaDB:** Utilizado para operaciones sobre datos estructurados, como las entidades persona y profesión.
 - **Adaptador MongoDB:** Utilizado para entidades que requieren almacenamiento más flexible o no estructurado, como teléfono y estudios.
- Puertos de salida: Interfaces que definen los métodos requeridos para acceder y manipular datos en los sistemas de persistencia. Son utilizados por los casos de uso y deben ser implementados por los adaptadores de salida correspondientes.
- Base de datos Mongo: Base de datos no relacional en la que se almacena la información de las 4 entidades de este laboratorio, usando colecciones y documentos.
- Base de datos María: Base de datos relacional en la que se almacena la información de las 4 entidades de este laboratorio mediante el uso de tablas y relaciones.

Cabe agregar que, cada entidad (persona, profesión, estudios y teléfono) cuenta con su propio conjunto de adaptadores, puertos y casos de uso, organizados como se explicó anteriormente.

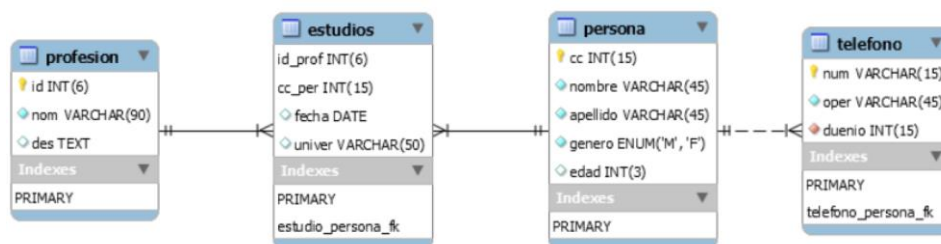
Diagrama de despliegue:

https://drive.google.com/file/d/1FffFD5MalmLnBX6wraDM_YuEx2ROC7Ss/view?usp=sharing



En el anterior diagrama se muestra cómo será desplegado el sistema: se utilizará un contenedor Docker para la aplicación y un contenedor adicional para cada una de las dos bases de datos (MariaDB y MongoDB).

Modelo de datos:



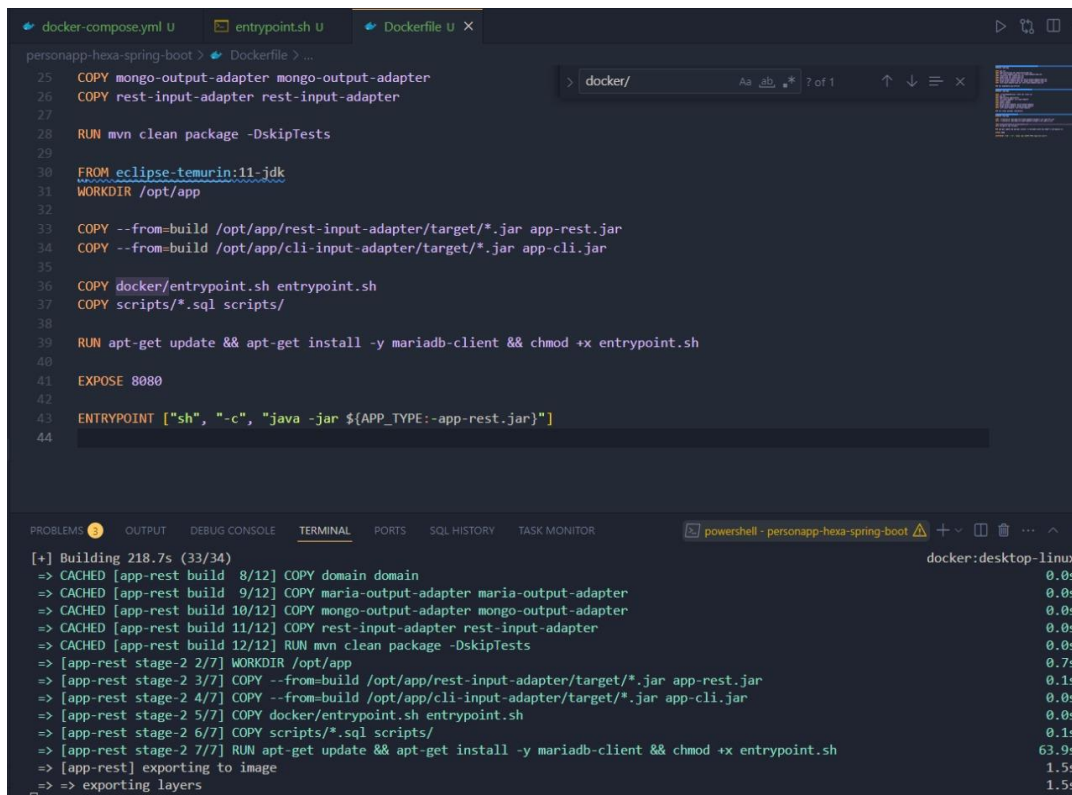
- **Profesion:** contiene la información sobre las distintas profesiones que una persona puede tener. Los atributos clave de esta entidad incluyen el identificador único id (de tipo entero), el nombre de la profesión nom (de tipo VARCHAR con una longitud máxima de 90 caracteres), y una descripción detallada de la profesión des (de tipo

TEXT). Esta entidad tiene una relación de uno a muchos con la entidad Estudios, lo que implica que una profesión puede estar asociada con múltiples estudios realizados por diversas personas.

- **Estudios:** Contiene los detalles de los estudios realizados por las personas. Esta tabla incluye atributos como `id_prof`, que es una clave foránea que enlaza con la tabla de Profesión para identificar la profesión asociada al estudio, y `cc_per`, otra clave foránea que vincula a la persona que realizó el estudio. Además, incluye la fecha en la que se llevaron a cabo los estudios (`fecha`) y el nombre de la universidad (`univer`). En cuanto a las relaciones, Estudios tiene una relación de muchos a uno con Persona y Profesión, lo que significa que cada estudio pertenece a una única persona y está relacionado con una única profesión.
- **Persona:** Contiene la información personal de los individuos. Entre los atributos de esta entidad se encuentra la cédula de ciudadanía (`cc`), que actúa como clave primaria y sirve como identificador único. También incluye el nombre y apellido de la persona (ambos de tipo `VARCHAR` con una longitud máxima de 45 caracteres), el género (`genero`, de tipo `ENUM` con valores 'M' para masculino y 'F' para femenino), y la edad de la persona. Esta entidad tiene una relación de uno a muchos tanto con la entidad Estudios como con la entidad Teléfono, permitiendo que una persona pueda estar asociada a múltiples estudios y poseer varios números de teléfono.
- **Teléfono:** Contiene los números de teléfono y la información asociada a cada uno. Los principales atributos de esta tabla incluyen el número de teléfono `num` (de tipo `VARCHAR` con una longitud máxima de 15 caracteres), el operador de telefonía `oper`, y una clave foránea `duenio` que enlaza a la persona propietaria del teléfono mediante su cédula de ciudadanía. Esta entidad tiene una relación de muchos a uno con la entidad Persona, lo que significa que cada teléfono está asociado a una única persona.

Procedimiento

1. **Dockerizar el proyecto:** Crear una imagen en Docker del proyecto, empaquetando los módulos REST y CLI y preparando todo para su despliegue en contenedores, como se mostrará a continuación.



The screenshot shows an IDE with a Dockerfile editor and a terminal window. The Dockerfile contains the following instructions:

```
25 COPY mongo-output-adapter mongo-output-adapter
26 COPY rest-input-adapter rest-input-adapter
27
28 RUN mvn clean package -DskipTests
29
30 FROM eclipse-temurin:11-jdk
31 WORKDIR /opt/app
32
33 COPY --from-build /opt/app/rest-input-adapter/target/*.jar app-rest.jar
34 COPY --from-build /opt/app/cli-input-adapter/target/*.jar app-cli.jar
35
36 COPY docker/entrypoint.sh entrypoint.sh
37 COPY scripts/*.sql scripts/
38
39 RUN apt-get update && apt-get install -y mariadb-client && chmod +x entrypoint.sh
40
41 EXPOSE 8080
42
43 ENTRYPOINT ["sh", "-c", "java -jar ${APP_TYPE:-app-rest.jar}"]
44
```

The terminal window shows the build output for the Docker image:

```
[+] Building 218.7s (33/34)
=> CACHED [app-rest build 8/12] COPY domain domain 0.0s
=> CACHED [app-rest build 9/12] COPY maria-output-adapter maria-output-adapter 0.0s
=> CACHED [app-rest build 10/12] COPY mongo-output-adapter mongo-output-adapter 0.0s
=> CACHED [app-rest build 11/12] COPY rest-input-adapter rest-input-adapter 0.0s
=> CACHED [app-rest build 12/12] RUN mvn clean package -DskipTests 0.0s
=> [app-rest stage-2 2/7] WORKDIR /opt/app 0.7s
=> [app-rest stage-2 3/7] COPY --from-build /opt/app/rest-input-adapter/target/*.jar app-rest.jar 0.1s
=> [app-rest stage-2 4/7] COPY --from-build /opt/app/cli-input-adapter/target/*.jar app-cli.jar 0.0s
=> [app-rest stage-2 5/7] COPY docker/entrypoint.sh entrypoint.sh 0.0s
=> [app-rest stage-2 6/7] COPY scripts/*.sql scripts/ 0.1s
=> [app-rest stage-2 7/7] RUN apt-get update && apt-get install -y mariadb-client && chmod +x entrypoint.sh 63.9s
=> [app-rest] exporting to image 1.5s
=> exporting layers 1.5s
```

Imagen 1. Dockerizar el servicio web

Procedimiento:

Etapas del Docker file:

1. Construcción de la app:

```
COPY mongo-output-adapter mongo-output-adapter
COPY rest-input-adapter rest-input-adapter
RUN mvn clean package -DskipTests
```

- Se copian los módulos relevantes del proyecto.
- Se ejecuta el empaquetado de la aplicación con Maven, omitiendo los tests

2. Configuración de imagen final

```
FROM eclipse-temurin:11-jdk
WORKDIR /opt/app
```

- Se utiliza como base la imagen oficial de Java 11 (temurin).
- Se define el directorio de trabajo en el contenedor.

3. Copiado de JARs y scripts

```
COPY --from=build ... app-rest.jar  
COPY --from=build ... app-cli.jar  
COPY docker/entrypoint.sh entrypoint.sh  
COPY scripts/*.sql scripts/
```

- Se copian los ejecutables .jar resultantes de los módulos REST y CLI.
- Se agregan el script de entrada (entrypoint.sh) y scripts SQL.

4. Instalación de herramientas

```
RUN apt-get update && apt-get install -y mariadb-client && chmod +x e  
ntrypoint.sh
```

- Se instala el cliente de MariaDB dentro del contenedor para ejecutar scripts SQL si es necesario.
- Se da permiso de ejecución al script entrypoint.sh.

5. Exposición y ejecución

```
EXPOSE 8080
```

```
ENTRYPOINT ["sh", "-c", "java -jar ${APP_TYPE:-app-rest.jar}"]
```

- El contenedor expone el puerto 8080 (por defecto en Spring Boot).
- Usa ENTRYPOINT para ejecutar el .jar, permitiendo cambiar entre REST y CLI dinámicamente mediante la variable APP_TYPE.

6. Terminal: Resultado del Build

- Como se observa en la imagen, en la terminal El build del contenedor se ejecutó correctamente además de mostrar los tiempos en que cada etapa ejecutada y finalmente la imagen final se exportó correctamente.

2. Configurar correctamente los pom.xml para ejecución en Docker

Es necesario configurar los pom.xml .jar para que se puedan ejecutar dentro del contenedor Docker, esto porque Docker necesita saber qué clase principal ejecutar, ya que si no se indica explícitamente la clase principal en un proyecto con múltiples módulos (como este), el .jar generado puede no ser ejecutable o lanzar errores. Por otro lado, se generan dos artefactos separados (app-rest.jar y app-cli.jar): Esto permite que el contenedor pueda ejecutar cualquiera de los dos, según la variable de entorno APP_TYPE, por eso cada módulo debe tener su propia clase principal bien configurada. Finalmente Se excluye Lombok porque no es necesario en producción, dado que lombok solo se usa para generar código en tiempo de compilación (getters/setters, etc.). Si se empaqueta, puede causar errores innecesarios. A continuación, se mostrará el procedimiento detalladamente.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${global.springframework.boot.version}</version>
      <configuration>
        <mainClass>co.edu.javeriana.as.personapp.PersonAppCli</mainClass>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Imagen 2. Cambiar configuración en los archivos pom.xml

Procedimiento:

1. Detalles técnicos:

Se maneja el siguiente Plugin: Plugin spring-boot-maven-plugin. Este plugin permite empaquetar la aplicación Spring Boot como un .jar ejecutable

<plugin>


```
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-maven-plugin</artifactId>  
<version>${global.springframework.boot.version}</version>
```

2. Se declara mainClass: Se especifica la clase principal del módulo CLI:

```
<mainClass>co.edu.javeriana.as.personapp.PersonAppCli</mainClass>
```

Esto es fundamental para que Docker sepa qué clase ejecutar si APP_TYPE=app-cli.jar.

3. Se excluye lombok Finalmente se excluye lombok para evitar errores durante el empaquetado:

```
<excludes>  
  <exclude>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
  </exclude>  
</excludes>
```

Una vez realizado este proceso Queda correctamente configurado el pom.xml del adaptador CLI (y de forma similar se haría con el REST) para que Maven genere un .jar ejecutable independiente, funcional dentro del contenedor Docker.

3. crear model, mapper y menú por entidad

Dado que la aplicación maneja una arquitectura hexagonal, el adaptador CLI (interfaz de línea de comandos) necesita los siguientes componentes:

- Model: Representa cómo se muestra o solicita la información al usuario en consola. Puede diferir del modelo de dominio.
- Mapper: Traduce entre el modelo CLI (ModelCli) y la entidad del dominio. Separa la lógica de presentación de la lógica de negocio.
- Menú: Maneja la interacción con el usuario en consola: opciones disponibles, llamadas al adaptador, entrada de datos

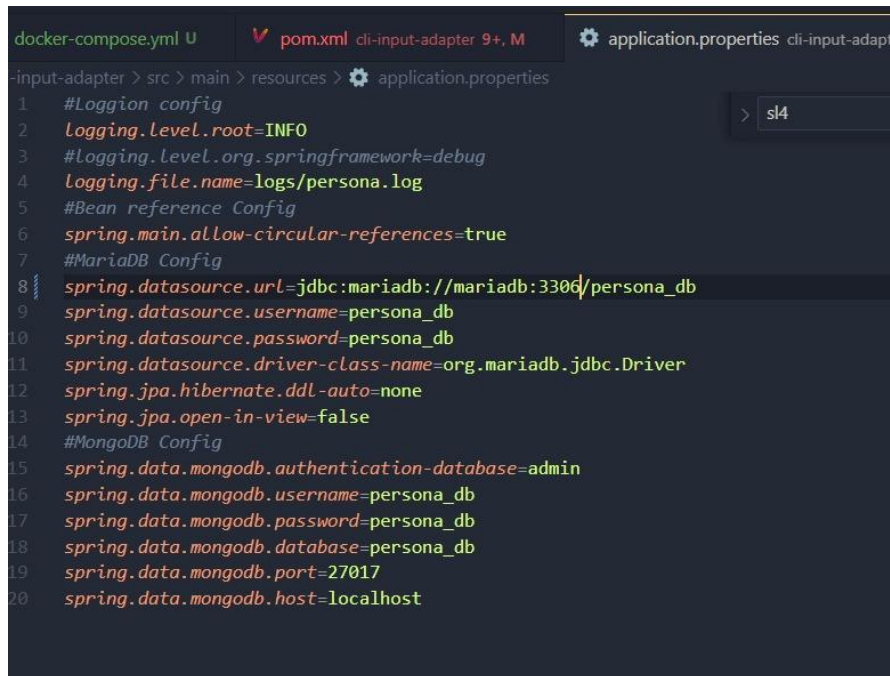


4. Configurar application.properties para entorno Docker

En este paso Se modificaron los archivos application.properties para evitar usar localhost como host para las bases de datos y se usaron los nombres de servicio definidos en docker-compose.yml, es decir:

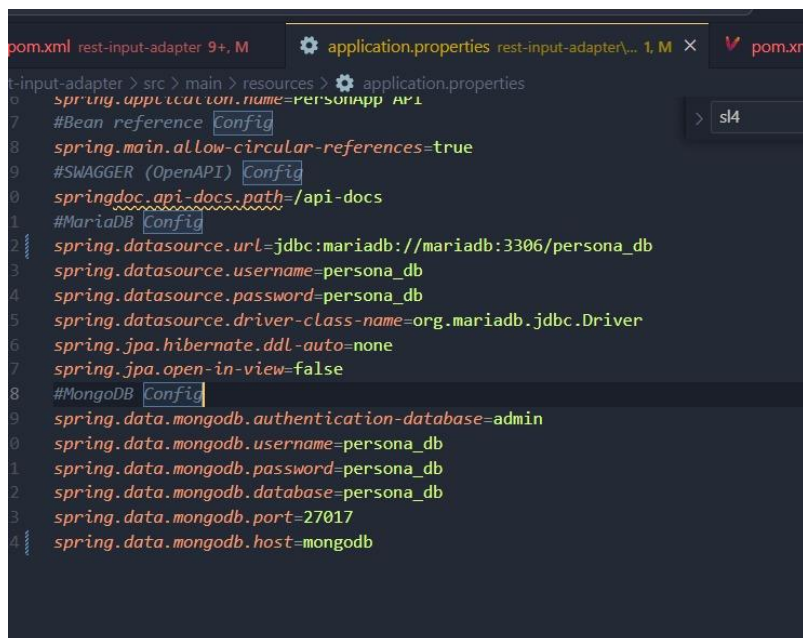
- mariadb para MariaDB
- mongodb para MongoDB

Este proceso se realiza por que, dado que los contenedores se comunican entre sí en un entorno Docker, no pueden usar localhost, ya que se refiere al contenedor actual. En su lugar, deben usar el nombre del servicio definido en Docker Compose (como mariadb o mongodb). Este cambio permite que las aplicaciones encuentren las bases de datos dentro de la red de Docker.



```
docker-compose.yml U pom.xml cli-input-adapter 9+, M application.properties cli-input-adapt
t-input-adapter > src > main > resources > application.properties
1 #Logging config
2 logging.level.root=INFO
3 #logging.level.org.springframework=debug
4 logging.file.name=logs/persona.log
5 #Bean reference Config
6 spring.main.allow-circular-references=true
7 #MariaDB Config
8 spring.datasource.url=jdbc:mariadb://mariadb:3306/persona_db
9 spring.datasource.username=persona_db
10 spring.datasource.password=persona_db
11 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
12 spring.jpa.hibernate.ddl-auto=none
13 spring.jpa.open-in-view=false
14 #MongoDB Config
15 spring.data.mongodb.authentication-database=admin
16 spring.data.mongodb.username=persona_db
17 spring.data.mongodb.password=persona_db
18 spring.data.mongodb.database=persona_db
19 spring.data.mongodb.port=27017
20 spring.data.mongodb.host=localhost
```

Imagen 3. pt1 Configurar application.properties para entorno Docker



```
pom.xml rest-input-adapter 9+, M application.properties rest-input-adapter\... 1, M x pom.xml
t-input-adapter > src > main > resources > application.properties
0 spring.application.name=personapp API
1 #Bean reference Config
2 spring.main.allow-circular-references=true
3 #SWAGGER (OpenAPI) Config
4 springdoc.api-docs.path=/api-docs
5 #MariaDB Config
6 spring.datasource.url=jdbc:mariadb://mariadb:3306/persona_db
7 spring.datasource.username=persona_db
8 spring.datasource.password=persona_db
9 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
10 spring.jpa.hibernate.ddl-auto=none
11 spring.jpa.open-in-view=false
12 #MongoDB Config
13 spring.data.mongodb.authentication-database=admin
14 spring.data.mongodb.username=persona_db
15 spring.data.mongodb.password=persona_db
16 spring.data.mongodb.database=persona_db
17 spring.data.mongodb.port=27017
18 spring.data.mongodb.host=mongodb
```

Imagen 4. pt2. Configurar application.properties para entorno Docker

Procedimiento:

1. Identificar nombres de servicio en docker-compose.yml:

```
services:
  mariadb:
    image: mariadb:latest
  mongodb:
    image: mongo:latest
```

2. Modificar mariadb y mongodb:

```
spring.datasource.url=jdbc:mariadb://mariadb:3306/persona_db
spring.datasource.username=persona_db
spring.datasource.password=persona_db
```

```
spring.data.mongodb.host=mongodb
spring.data.mongodb.port=27017
spring.data.mongodb.database=persona_db
spring.data.mongodb.username=persona_db
spring.data.mongodb.password=persona_db
```

3. Guardar los cambios y reconstruir los contenedores si ya estaban creados

Una vez realizado este proceso se garantiza que la aplicación pueda conectarse correctamente a las bases de datos cuando se ejecuta dentro de Docker, evitando errores de conexión



Imagen 5. Configurar application.properties para entorno Docker

Procedimiento:

1. Crear el PhoneModelCli.java

Ubicación: terminal/model/

Ejemplo:

```
public class PhoneModelCli {  
  
    private String number;  
  
    private String type;  
  
    // Getters, setters, toString() }
```

2. Crear el PhoneMapperCli.java

Ubicación: terminal/mapper/

Ejemplo:

```
public class PhoneMapperCli {  
  
    public static Phone toDomain(PhoneModelCli cli) {  
  
        return new Phone(cli.getNumber(), cli.getType());  
    }  
    public static PhoneModelCli toCli(Phone phone) {  
  
        PhoneModelCli cli = new PhoneModelCli();  
        cli.setNumber(phone.getNumber());  
        cli.setType(phone.getType());  
        return cli;  
    }  
}
```

3. Crear el PhoneMenu.java

Ubicación: terminal/menú/

Este archivo tiene como objetivo presentar un menú de opciones al usuario para realizar acciones sobre teléfonos (crear, consultar, eliminar, etc.) y gestionar las entradas que el usuario digita por consola.

4. Crear InputPort, OutputPort y UseCase por entidad

Es necesario realizar este proceso dado que forma parte de la arquitectura hexagonal (o Ports and Adapters), donde se promueve la separación de responsabilidades y el aislamiento del dominio respecto a las tecnologías externas.

Cada entidad del dominio (por ejemplo: Persona, Profession, Phone, Study) debe tener su propio conjunto de interfaces y lógica.

Componentes:

- InputPort: Define las operaciones del lado del usuario (controlador, CLI, API)
- OutputPort: Define las operaciones necesarias para acceder a fuentes externas, como bases de datos
- UseCase: Contiene la lógica de negocio central que interactúa con los puertos.

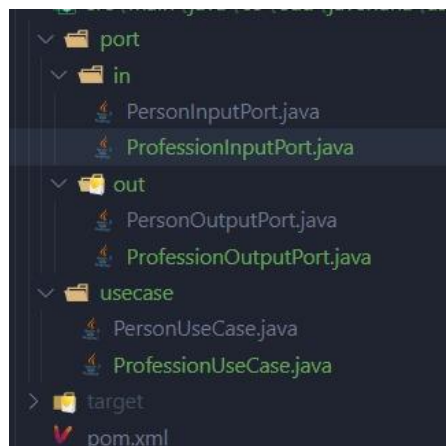


Imagen 6. Crear InputPort, OutputPort y UseCase por entidad

Procedimiento:

1. Crear el InputPort

- Ubicación: core/port/in/
- Nombre: <Entidad>InputPort.java

```
public interface PhoneInputPort {  
  
    void createPhone(Phone phone);  
  
    List<Phone> getAllPhones(); }  

```

2. Crear el OutputPort

- Ubicación: core/port/out/
- Nombre: <Entidad>OutputPort.java

```
public interface PhoneOutputPort {  
  
    void savePhone(Phone phone);  
  
    List<Phone> findAllPhones(); }  

```

3. Crear el UseCase

- Ubicación: core/usecase/
- Nombre: <Entidad>UseCase.java

```
public class PhoneUseCase implements PhoneInputPort {  
  
    private final PhoneOutputPort phoneOutputPort;  
  
    public PhoneUseCase(PhoneOutputPort phoneOutputPort) {  
  
        this.phoneOutputPort = phoneOutputPort;  
  
    }  
  
    @Override  
  
    public void createPhone(Phone phone) {  
  
        phoneOutputPort.savePhone(phone);  
  
    }  
  
    @Override
```

```
public List<Phone> getAllPhones() {  
    return phoneOutputPort.findAllPhones(); } }
```

4. Registrar en los adaptadores (CLI y REST)

Más adelante, estos UseCase deben ser inyectados en los controladores CLI o REST (dependiendo del adaptador), conectando con las implementaciones reales del OutputPort (por ejemplo, un repositorio JPA).

5. Generar los adapters y repository para MariaDB

Este paso consiste en construir los 4 elementos esenciales de un adaptador de salida (OutputAdapter) hacia una base de datos relacional. Aquí se deben crear todos los componentes necesarios para que la entidad Study (Estudios) pueda ser persistida y consultada desde una base de datos MariaDB, respetando el diseño de arquitectura hexagonal. Este paso es importante porque permite que la entidad Study tenga persistencia real dentro de la aplicación. Es decir:

- Se puede guardar información en la base de datos.
- Se puede consultar registros desde la lógica de negocio (use cases). todo sin acoplarse directamente a MariaDB (gracias a los puertos y adaptadores)

A continuación, se muestra el procedimiento detalladamente:

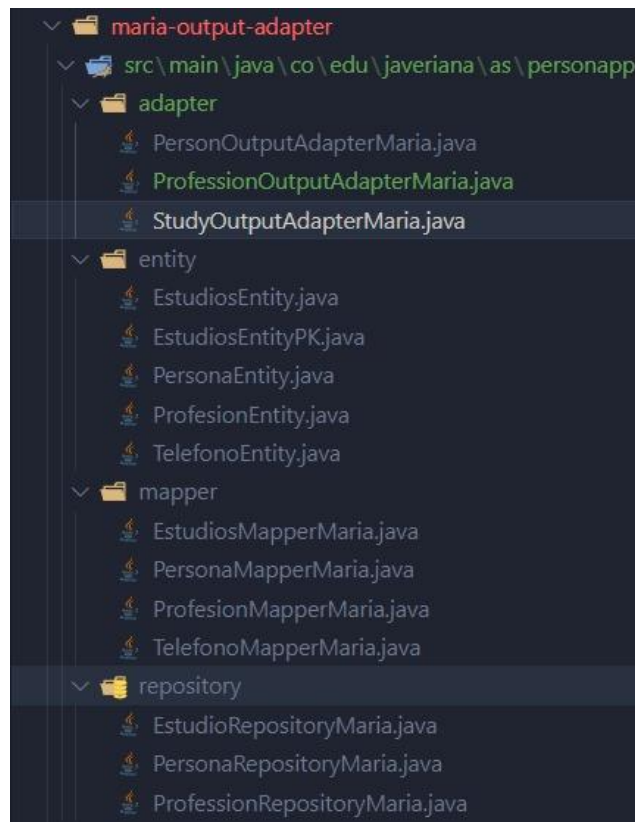


Imagen 7. Generar los adapters y repository para MariaDB

Procedimiento:

Directorio:

maria-output-adapter

Subcarpetas:

- adapter/
- entity/
- mapper/
- Repository/

1. Crear el archivo StudyOutputAdapterMaria.java

Ubicación: adapter/

Esta es la clase que implementa la interfaz del puerto de salida (StudyOutputPort) y se encarga de conectar el dominio con la base de datos.

- Recibe objetos del dominio (Study).
- Usa el mapper para convertirlos en entidades JPA (EstudiosEntity).

- Llama al repositorio JPA para guardarlos o consultarlos.
- Esta clase es el “puente” entre la lógica de negocio y MariaDB.

2. Tener EstudiosEntity.java y EstudiosEntityPK.java

Ubicación: entity/

Estas son las representaciones JPA de la tabla estudios en la base de datos.

- Modelan la estructura de la tabla (columnas, tipos de datos, claves).
- EstudiosEntityPK modela la clave primaria compuesta, si aplica.
- Estos archivos permiten que Spring Data JPA sepa cómo mapear objetos a registros en la base de datos.

3. Crear el archivo EstudioRepositoryMaria.jav

Ubicación: repository/

Esta es una interfaz que extiende JpaRepository.

- Expone métodos CRUD (guardar, buscar, eliminar...).
- Spring se encarga de la implementación automática.
- Este repositorio es el que realmente se comunica con MariaDB.

4. Crear el archivo EstudioRepositoryMaria.java

Ubicación: repository/

Es una interfaz que extiende JpaRepository.

- Expone métodos CRUD (guardar, buscar, eliminar...).
- Spring se encarga de la implementación automática.
- Este repositorio es el que realmente habla con MariaDB.

6. Testear funcionamiento de los adaptadores hacia MariaDB y MongoDB

Se realizaron unas pruebas en las que el equipo interactúa con un menú de consola (CLI) para manejar teléfonos (Phone). Selecciona la opción 1 para ver todos los teléfonos. Este es el resultado que se obtuvo:

- El sistema muestra una traza (INFO) donde indica que está usando el Input Adapter para la entidad Phone.

- Se muestra el objeto recuperado (PhoneModelCli) con los campos correctos (number, operator, ownerCc).
- Luego, el menú ofrece escoger:
 - 1 para MariaDB
 - 2 para MongoDB
 - 0 para regresar

```

5 para buscar teléfono
0 para regresar
Ingrese una opción: 1
2025-05-24 18:19:37.742 INFO 7 --- [          main] c.e.j.a.p.t.a.PhoneInputAda
rial PhoneEntity in Input Adapter
PhoneModelCli(number=1234, operator=movistar, ownerCc=123456789)
-----
1 para ver todos los teléfonos
2 para crear teléfono
3 para editar teléfono
4 para eliminar teléfono
5 para buscar teléfono
0 para regresar
Ingrese una opción: 0
-----
1 para MariaDB
2 para MongoDB
0 para regresar
Ingrese una opción: █

```

Imagen 8. Testear funcionamiento de los adaptadores hacia MariaDB y MongoDB

Esto confirma que la aplicación permite al usuario decidir en tiempo de ejecución a qué base de datos persistente acceder: MariaDB o MongoDB, gracias a que ambas están implementadas como adaptadores separados.

7. Crear adaptadores de salida (OutputAdapter) y repositorios (Repository) para MongoDB

(Ubicación del módulo)

src/main/java/co/edu/javeriana/as/personapp/mongo/

Este paso consiste en crear todos los componentes necesarios para permitir la persistencia de datos en MongoDB usando una arquitectura hexagonal. Se crearán:

- Documentos para cada entidad.
- Mappers para transformar entre documentos y modelos del dominio.
- Repositorios Spring Data para MongoDB.
- Adaptadores de salida que implementan los puertos de salida.

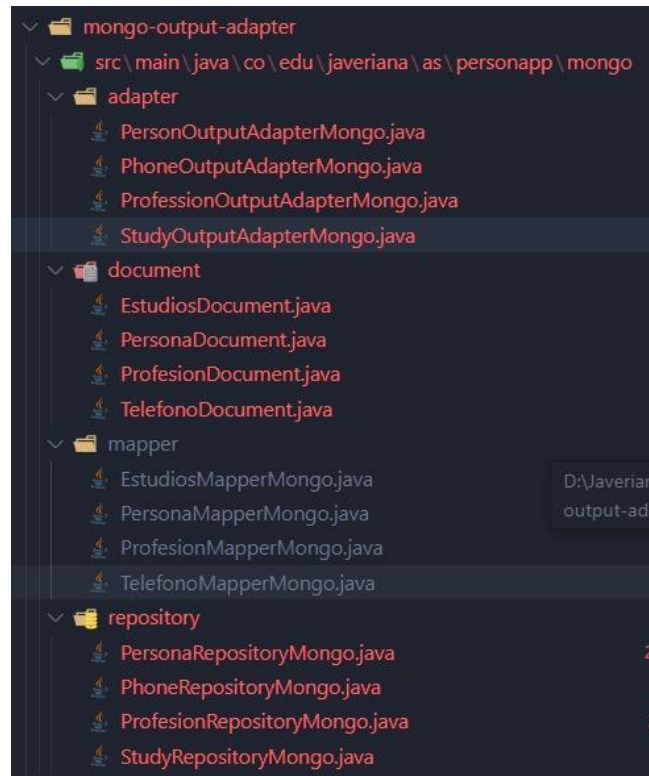


Imagen 9. Crear adaptadores de salida (OutputAdapter) y repositorios (Repository)

Procedimiento:

1. Crear documentos

Ubicación: document/

Los documentos representan las entidades como se almacenan en MongoDB. Cada clase debe estar anotada con `@Document` y contener los campos anotados con `@Id`, `@Field`, etc.

Clases creadas:

- PersonaDocument.java
- TelefonoDocument.java
- ProfesionDocument.java
- EstudiosDocument.java

Ejemplo:

```
Document(collection = "personas")  
  
public class PersonaDocument {  
  
    @Id  
  
    private String cc;  
  
    private String nombre  
  
    private String apellido;  
  
  
    // ... otros campos}
```

2. Crear mappers

Ubicación: mapper/

Se encargan de convertir entre las entidades del dominio y los documentos de Mongo.

Clases creadas:

- PersonaMapperMongo.java
- TelefonoMapperMongo.java
- ProfesionMapperMongo.java
- EstudiosMapperMongo.java
- Responsabilidades:
- toDocument() y toDomain() para cada entidad.

3. Crear repositorios

Ubicación: repository/

Definen las interfaces que extienden de MongoRepository, lo que permite a Spring Data generar la implementación automáticamente.

Clases creadas:

- PersonaRepositoryMongo.java
- PhoneRepositoryMongo.java
- ProfessionRepositoryMongo.java
- StudyRepositoryMongo.java

Ejemplo:

```
public interface PersonaRepositoryMongo extends MongoRepository<PersonaDocument, String>
```

4. Crear adaptadores de salida (OutputAdapters)

Ubicación: adapter/

Implementan los puertos de salida definidos en el dominio (OutputPort) y utilizan los mappers y repositorios para interactuar con MongoDB.

Clases creadas:

- PersonOutputAdapterMongo.java
- PhoneOutputAdapterMongo.java
- ProfessionOutputAdapterMongo.java
- StudyOutputAdapterMongo.java

Ejemplo de responsabilidades del adaptador:

- **save() guarda usando el repositorio y transforma con el mapper.**
- **find() busca y convierte de documento a modelo del dominio.**

8. Evitar recursividad entre entidades (toDomainBasic)

Ubicaciones:

- mongo-output-adapter > mapper > PersonaMapperMongo.java
- maria-output-adapter > mapper > PersonaMapperMaria.java

Con este paso se busca evitar errores de recursividad infinita y problemas de serialización al mapear entidades que tienen relaciones bidireccionales (por ejemplo, Persona ↔ Teléfono).

Se crea un método llamado `fromAdapterToDomainBasic()` que convierte un `PersonaDocument` (MongoDB) o un `PersonaEntity` (MariaDB) en un objeto `Person` sin cargar las relaciones (teléfonos, estudios, etc.). Este método se usa cuando se necesita referenciar una persona desde otra entidad (como `Phone`) sin provocar una carga cíclica de datos.

```
70     }
71     public Person fromAdapterToDomainBasic(PersonaDocument personaDocument) {
72         Person person = new Person();
73         person.setIdentification(personaDocument.getId());
74         person.setFirstName(personaDocument.getNombre());
75         person.setLastName(personaDocument.getApellido());
76         person.setGender(validateGender(personaDocument.getGenero()));
77         person.setAge(validateAge(personaDocument.getEdad()));
78         return person;
79     }
80 }
```

Imagen 10. Evitar recursividad entre persona y teléfono

Como se observa en la imagen, al realizar este paso El sistema puede mapear los datos entre entidades sin caer en ciclos infinitos y Se mantiene la separación de responsabilidades entre los mappers y las entidades de dominio. Por lo que este patrón se encarga de evitar relaciones cíclicas con mappers “básicos”. Esta es una buena práctica cuando trabajamos con modelos ricos en relaciones como los de tipo `OneToMany` / `ManyToOne`.

9. Evitar recursividad en el mapeo de estudios

Método a modificar: `fromDomainToAdapter(Study study)`

Ubicación: `Estudio sMapperMaria.java`

Este paso es bastante similar que el anterior, se busca evitar que la entidad `Study` al ser convertida a `EstudiosEntity` genere una cadena infinita de llamadas recursivas debido a referencias completas a `Person` y `Profession`.

Actualmente el código se veía de la siguiente manera:

```
estudio.setPersona(personaMapperMaria.fromDomainToAdapter(study.getPerson()));

estudio.setProfesion(profesionMapperMaria.fromDomainToAdapter(study.getProfession()
));
```

Estas llamadas están convirtiendo la persona y la profesión con toda su información, lo cual puede provocar recursividad, especialmente si `Person` tiene de vuelta una lista de `Study`. Por esta razón el código se corrigió de la siguiente manera:

```
estudio.setPersona(personaMapperMaria.fromDomainToAdapterBasic(study.getPerson()))  
;  
  
estudio.setProfesion(profesionMapperMaria.fromDomainToAdapterBasic(study.getProfesion()));
```

A screenshot of a code editor showing a Java method named 'fromDomainToAdapter' that takes a 'Study' object as input and returns an 'EstudiosEntity'. The code includes a null check for the input, initializes an 'EstudiosEntityPK' object, and then sets various attributes of the 'EstudiosEntity' by calling other mapping methods like 'fromDomainToAdapterBasic' and validation methods like 'validateFecha' and 'validateUniver'.

```
public EstudiosEntity fromDomainToAdapter(Study study) {  
    if (study == null) {  
        return null;  
    }  
    EstudiosEntityPK estudioPK = new EstudiosEntityPK();  
    estudioPK.setCcPer(study.getPerson().getIdentification());  
    estudioPK.setIdProf(study.getProfesion().getIdentification());  
    EstudiosEntity estudio = new EstudiosEntity();  
    estudio.setEstudiosPK(estudioPK);  
    estudio.setFecha(validateFecha(study.getGraduationDate()));  
    estudio.setUniver(validateUniver(study.getUniversityName()));  
    estudio.setPersona(personaMapperMaria.fromDomainToAdapterBasic(study.getPerson()));  
    estudio.setProfesion(profesionMapperMaria.fromDomainToAdapterBasic(study.getProfesion()));  
    return estudio;  
}
```

Imagen 11. Evitar recursividad en el mapeo de estudios

Con los cambios agregados asegura que al mapear el Study, no se incluya recursivamente de nuevo a la lista de estudios dentro de la persona o profesión. Dando como resultado Conversión segura sin loops recursivos, integridad del modelo sin errores de desbordamiento de pila y mejor control de qué atributos se serializan o no.

10. Implementación del método fromAdapterToDomainBasic para Study (MongoDB)

Clase modificada: EstudiosMapperMongo.java

En este paso se realizará la implementación correcta del método fromAdapterToDomainBasic para el mapper de MongoDB, específicamente para evitar recursividad entre Study, Person y Profession.


```

    }
    public Study fromAdapterToDomainBasic(EstudiosDocument estudiosDocument) {
        Study study = new Study();
        study.setPerson(personaMapperMongo.fromAdapterToDomainBasic(estudiosDocument.getPrimaryPersona()));
        study.setProfession(profesionMapperMongo.fromAdapterToDomainBasic(estudiosDocument.getPrimaryProfesion()));
        study.setGraduationDate(validateGraduationDate(estudiosDocument.getFecha()));
        study.setUniversityName(validateUniversityName(estudiosDocument.getUniver()));
        return study;
    }
}

```

Imagen 12. Implementación del método fromAdapterToDomainBasic para Study (MongoDB)

Mediante este método `getPrimaryPersona()` y `getPrimaryProfesion()` retornan solo las referencias básicas (ID) sin colecciones asociadas como estudios o teléfonos, además se usan métodos `fromAdapterToDomainBasic()` para evitar que se conviertan las estructuras completas de `Person` y `Profession` y finalmente la fecha y universidad se validan antes de asignarlas al modelo de dominio.

11. Prueba del endpoint DELETE de teléfono con Swagger

Para este paso se realizaron múltiples pruebas con el fin de Verificar el correcto funcionamiento del endpoint DELETE para eliminar un teléfono en la base de datos MongoDB a través de Swagger UI. Para realizar este proceso inicialmente se configuro correctamente Swagger utilizando aplicaciones como `@DeleteMapping` y bien definida la ruta `/api/v1/phone/{database}/{number}`

Endopint usado:

DELETE `/api/v1/phone/{database}/{number}`

Parámetros:

- Database: MONGO,
 - Tipo: Path
 - Indica la base de datos a usar
- Number: 12369
 - Tipo: Path
 - Número de teléfono a eliminar

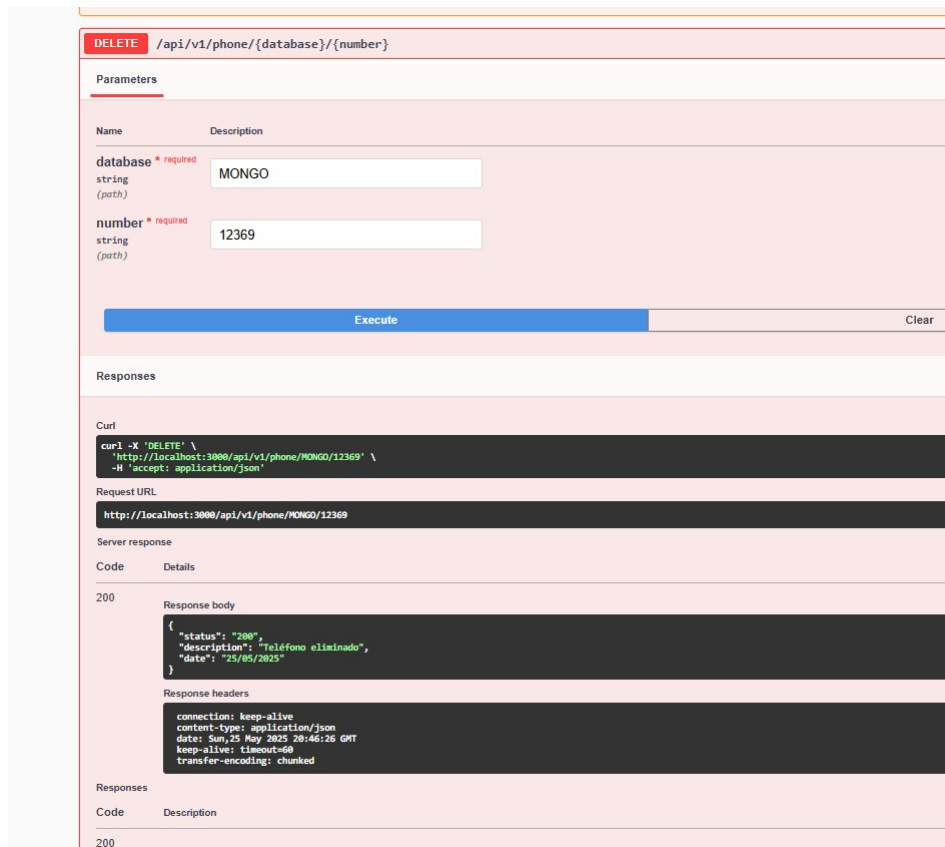


Imagen 13. Prueba del endpoint DELETE de teléfono con Swagger

Procedimiento:

- Desde Swagger, se ingresan los parámetros requeridos.
- Se presiona el botón "Execute" para enviar la petición DELETE.
- El servidor procesa la solicitud y responde.

Resultado de la prueba: Como se muestra en la imagen el resultado fue el esperado:

{

"status": "200",

"description": "Teléfono eliminado",

"date": "25/05/2025"

```
}
```

Esto indica que el teléfono con número 12369 fue eliminado de MongoDB.

12. Añadir soporte REST a una entidad en PersonApp

Como se dio a conocer anteriormente ya está la funcionando la funcionalidad con CLI . Ahora el paso a seguir es trasladar esa lógica al REST. Para ello, es necesario replicar ciertos componentes siguiendo el patrón que ya se ha implementado en otras entidades

rest-input-adapter

```
├─ adapter
|   └─ [Entity]InputAdapterRest.java
├─ controller
|   └─ [Entity]ControllerV1.java
├─ mapper
|   └─ [Entity]MapperRest.java
├─ model
|   └─ request
|       └─ [Entity]Request.java
|       └─ response
|           └─ [Entity]Response.java
```

A continuación, se observa la estructura a detalle:

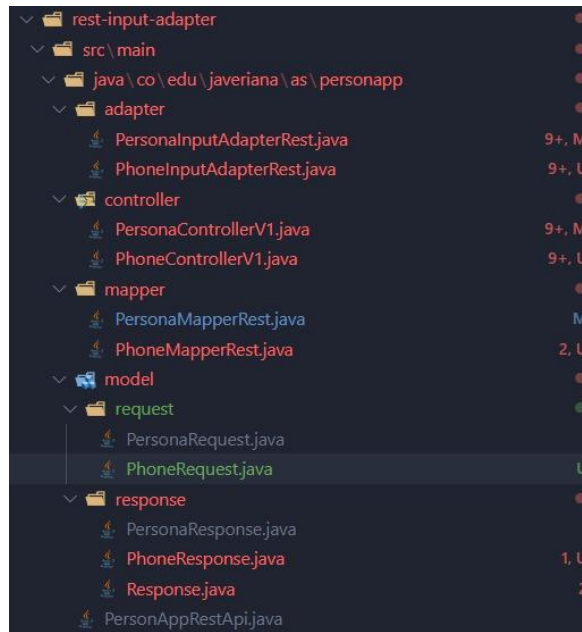


Imagen 14. Añadir soporte REST a una entidad en PersonApp

Procedimiento:

1. Crear clase Request y Response

Ubicadas en:

- model/request/[Entity]Request.java
- model/response/[Entity]Response.java

Estas clases son DTOs que representan la entrada y salida de datos para el consumidor REST. Por ejemplo:

```
public class PhoneRequest {  
    private String number;  
    private String cityCode;  
    private String countryCode;  
}
```

```
public class PhoneResponse {  
    private String number;  
    private String fullCode;}
```

2. Crear el MapperRest para convertir entre domain y DTO

Ubicado en:

mapper/[Entity]MapperRest.java

```

public class PhoneMapperRest {

    public Phone toDomain(PhoneRequest request) {

        return new Phone(request.getNumber(), request.getCityCode(),
request.getCountryCode());

    }

    public PhoneResponse toResponse(Phone phone) {

        PhoneResponse response = new PhoneResponse();

        response.setNumber(phone.getNumber());

        response.setFullCode(phone.getCityCode() + "-" + phone.getCountryCode());

        return response;

    }

}

```

3. Crear el InputAdapterRest

Ubicado
 adapter/[Entity]InputAdapterRest.java

en:

Este se conecta con el caso de uso (puerto primario) desde REST:

@RestController

```

public class PhoneInputAdapterRest {

    private final PhoneUseCase phoneUseCase;

    public PhoneInputAdapterRest(PhoneUseCase phoneUseCase) {

        this.phoneUseCase = phoneUseCase;}

}

```

```

    public Phone createPhone(PhoneRequest request) {

        return phoneUseCase.createPhone(...);

    }

}

```

Una vez completados estos pasos es posible crear, leer, actualizar o eliminar esa entidad usando Swagger, por lo que a continuación se haran pruebas sobre la base de datos MONGODB.

13. Endpoint para consultar todos los registros de estudios

Este endpoint está diseñado para consultar todos los registros de estudios (entidad Study) desde la base de datos especificada, en este caso MongoDB.

Se probará lo siguiente:

| | |
|--------------------|--------------------------|
| Método: | GET |
| Ruta del endpoint: | /api/v1/study/{database} |
| Parámetro usado: | |

- database = MONGO

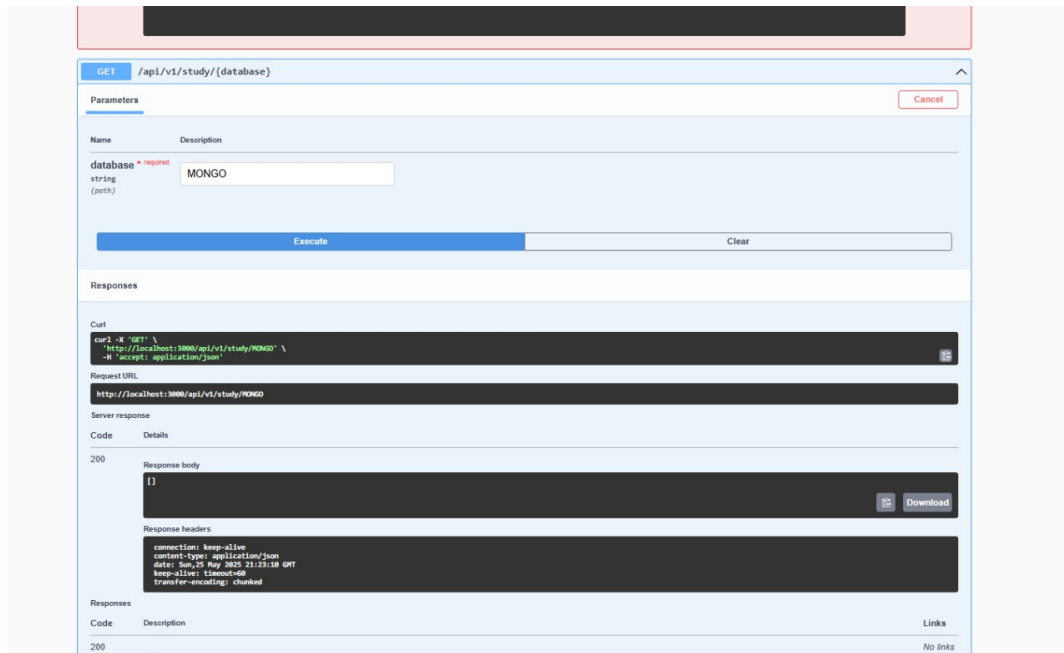


Imagen 15. Endpoint para consultar todos los registros de estudios

Procedimiento:

1. URL real que se llamó: `http://localhost:3000/api/v1/study/MONGO`
2. Método HTTP: GET
3. Headers: Accept: application/json

Respuesta del servidor:

- Código HTTP: 200 OK
Esto significa que la solicitud fue procesada correctamente
- Response body: []
Es un arreglo vacío, lo que indica que no se encontraron registros de estudios en MongoDB en ese momento.
- Headers de respuesta:
content-type: application/json
transfer-encoding: chunked

connection: keep-alive

Analizando los resultados de las pruebas se puede concluir que fue exitosa y que el endpoint está funcionando bien dado que no hay errores de lógica ni del servidor.

14. consultar datos de la base de datos MariaDB

Nuevamente realizamos una prueba la cual es un get, sin embargo, se probó en la base de datos MariaDB como se muestra a continuación:

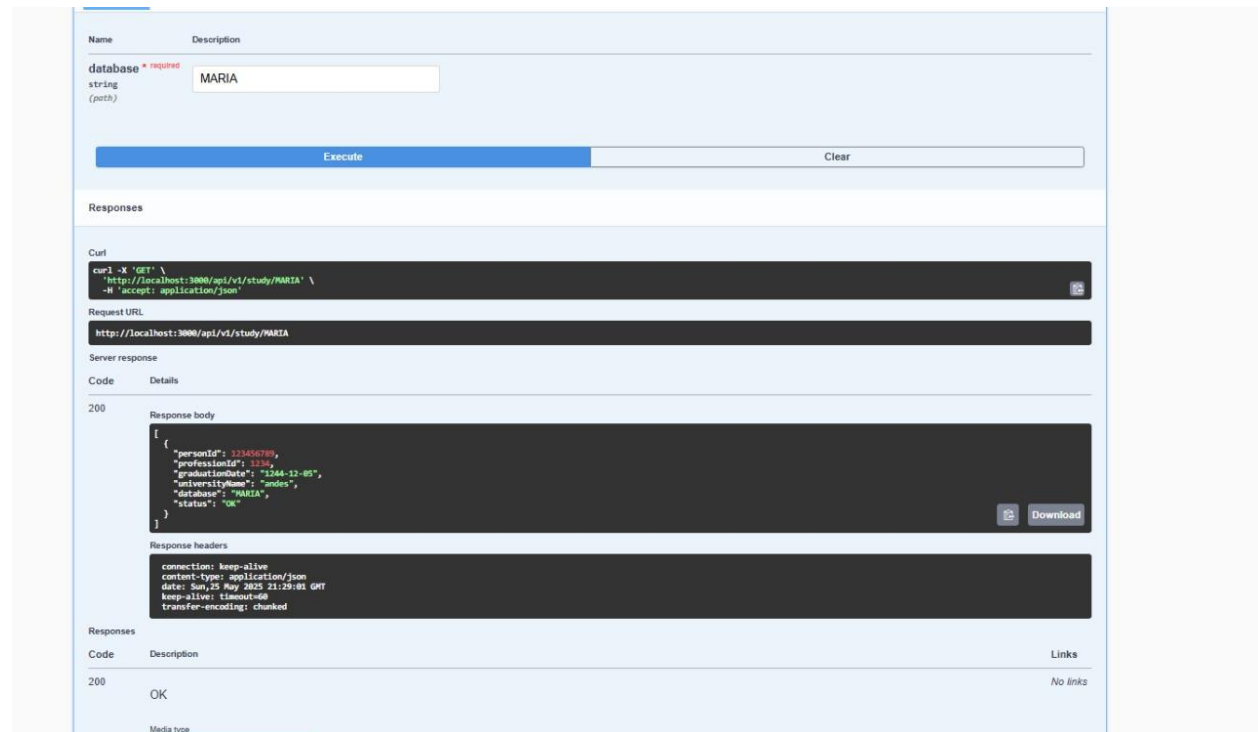


Imagen 16. consultar datos de la base de datos MariaDB

Endpoint:

GET

/api/v1/study/{database}

Con el valor: database = MARIA

Resultado

Código

HTTP:

200

OK

Todo funcionó correctamente.

Cuerpo de la respuesta (Response body)

```
[
  {
    "personId": 123456789,
```



```
"professionId": 1234,  
"graduationDate": "1244-12-05",  
"universityName": "andes",  
"database": "MARIA",  
"status": "OK"}]
```

Eso significa que ya hay registros en MariaDB para la entidad Study, y el endpoint los devolvió correctamente.

Cada objeto en el arreglo representa un **registro de estudio** de una persona, con estos datos:

- personId: ID de la persona
- professionId: ID de la profesión
- graduationDate: Fecha de graduación
- universityName: Universidad
- database: Indica que vino de MARIA (MariaDB)
- status: Mensaje de estado (opcionalmente útil para debug)

De esta prueba se confirma que el endpoint REST `/api/v1/study/{database}` funciona correctamente tanto para Mongo como MariaDB.

14. Pruebas desde la CLI

A continuación, se realizarán unas pruebas desde la CLI, para esto es necesario editar un estudio para una persona y su profesión en MariaDB como se muestra a continuación:

```

0 para salir
Ingrese una opción: 4
-----
1 para MariaDB
2 para MongoDB
0 para regresar
Ingrese una opción: 1
-----
1 para ver todos los estudios
2 para crear estudio
3 para eliminar estudio
4 para buscar estudio
5 para editar estudio
0 para regresar
Ingrese una opción: 5
ID Persona (del estudio a editar): 123456789
ID Profesion (del estudio a editar): 1234
Nueva fecha de graduación (YYYY-MM-DD): 1345-10-12
Nuevo nombre de universidad: jave
Saving study in MariaDB: Study(person=Person(identification=123456789, firstName=JUAN, lastName=RAMIREZ,
gender=FEMALE, age=21), profession=Profession(identification=1234, name=portero, description=porterear),
graduationDate=1345-10-12, universityName=jave)
Estudio actualizado: StudyModelCli(personId=123456789, professionId=1234, graduationDate=1345-10-12, univ
ersityName=jave)
-----

```

Imagen 17. Pruebas desde la CLI

Procedimiento:

Se edita un estudio ya existente en MariaDB con los siguientes datos:

- ID Persona: 123456789
- ID Profesión: 1234
- Nueva fecha de graduación: 1345-10-12
- Nueva universidad: jave

El sistema muestra:

- **Estudio actualizado: StudyModelCli(personId=123456789, professionId=1234, graduationDate=1345-10-12, universityName=jave)**

El backend CLI y el backend REST están usando los mismos modelos y repositorios, por lo que los cambios son visibles en ambos.

15. Prueba ejecutada desde Swagger UI sobre el endpoint GET

Para este paso se va a Probar correctamente la conexión con la base de datos MariaDB y recuperar información del historial académico (tabla estudios) asociada a personas, a través del endpoint correspondiente. Para esto se realizará una prueba ejecutada nuevamente desde Swagger UI sobre el endpoint GET /api/v1/study/{database} con el valor MARIA como parámetro de ruta. Como se muestra a continuación:

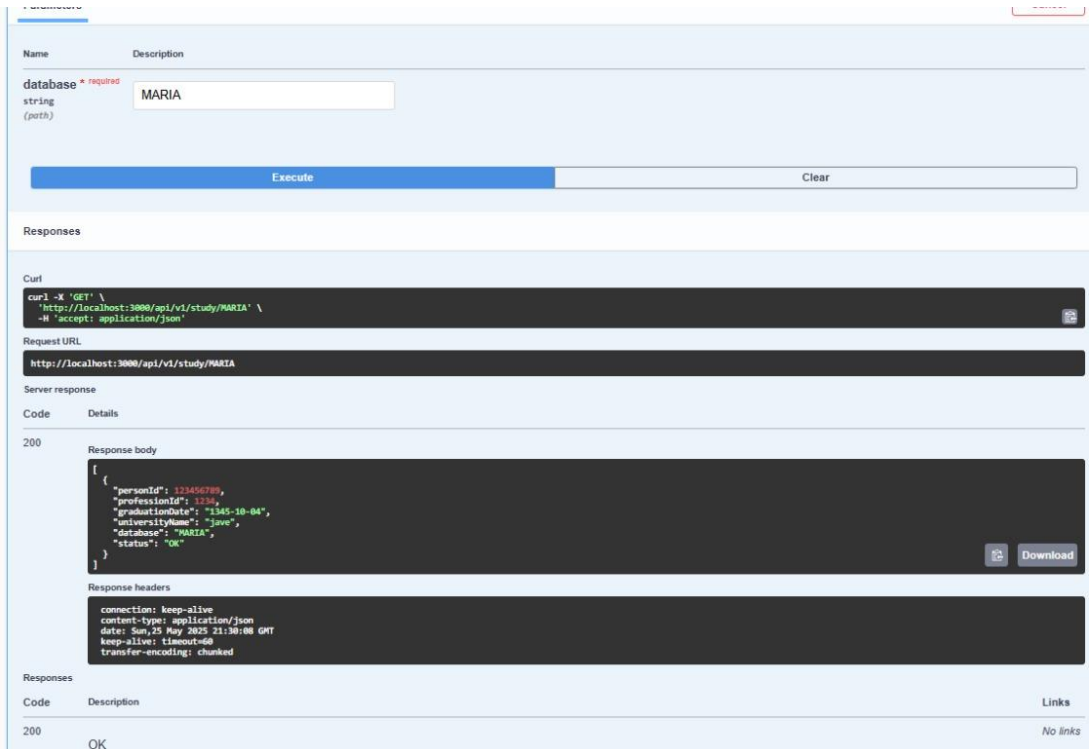


Imagen 18. • Prueba ejecutada desde Swagger UI

Procedimiento:

1. Ingresar nuevamente a Swagger UI
2. Localizar la sección correspondiente al recurso: study-controller.
3. Seleccionar el método GET /api/v1/study/{database}.
4. Ingresar el valor MARIA en el campo database (este parámetro indica que la fuente de datos será MariaDB).
5. Presionar Execute.
6. Observar la respuesta del servidor, que debe incluir:
 - Código HTTP: 200 OK
 - Cuerpo de respuesta (Response body) con información en JSON de los estudios registrados.

Resultado:

{

```
"personId": 123456789,  
"professionId": 1234,  
"graduationDate": "1345-10-04",  
"universityNAME": "Jave",  
"database": "MARIA",  
"status": "OK"  
}
```

La respuesta debe mostrar datos reales o simulados de la base de datos MariaDB, como se observa es posible confirmar que el servicio fue correctamente conectado al repositorio correspondiente a MariaDB.

16. prueba desde la interfaz CLI

La siguiente prueba que se realizara busca verificar la funcionalidad del adaptador CLI para interactuar con el servicio de estudios, permitiendo visualizar la información almacenada (en este caso, en MongoDB). A continuación, se mostrará la ejecución de la operación desde el CLI del sistema.

```
Conexión [conexionId={id=123456789, serverId=1234}] to mongo27017  
-----  
1 para ver todos los estudios  
2 para crear estudio  
3 para eliminar estudio  
4 para buscar estudio  
5 para editar estudio  
0 para regresar  
Ingrese una opción: 1  
2025-05-25 21:30:59.263 INFO 7 --- [main] c.e.j.a.p.t.a.StudyInputAdapterCli : Into his  
torial Study in Input Adapter  
StudyModelCli(personId=123456789, professionId=1234, graduationDate=1244-12-12, universityName=andes)  
-----
```

Imagen 19. prueba desde la interfaz CLI

El menú presenta varias opciones CRUD sobre los estudios, y se ha seleccionado la opción 1 para listar todos los registros. Se imprime en consola un log del InputAdapterCli donde se muestra un objeto StudyModelCli con los siguientes datos:

- personId = 123456789
- professionId = 1234
- graduationDate = 1244-12-12
- universityName = andes

Procedimiento:

1. Ejecutar el proyecto desde consola o terminal.
2. Seleccionar la opción 1 del menú mostrado para **ver todos los estudios**.
3. El sistema establece conexión con MongoDB y realiza la consulta.
4. Se imprime en consola el log del adaptador `StudyInputAdapterCli` con los datos obtenidos en forma de objeto `StudyModelCli`:

StudyModelCli(personId=123456789, professionId=1234, graduationDate=1244-12-12, universityName=andes)

Resultado:

Como se muestra anteriormente el resultado fue el esperado, dado que el sistema listo correctamente los registros desde la base MongoDB, confirmando así la funcionalidad de lectura (GET) desde CLI, además de que el log interno muestra el flujo correcto de ejecución desde el adaptador hasta el modelo.

17. Inserción de un estudio a través del servicio HTTP REST (MongoDB)

A continuación, se realizará una prueba prueba desde el servicio HTTP RESTful (usando Swagger nuevamente, esta prueba busca verificar la correcta operación del servicio RESTful para registrar un estudio en la base de datos MongoDB.



Imagen 19. prueba desde la interfaz CLI

Endpoint:

solicitud HTTP POST al endpoint: <http://localhost:3000/api/v1/study/MONGO>

Cuerpo de la solicitud (JSON):

```
{ "personId": 123456789,  
  "professionId": 1234,  
  "graduationDate": "1244-12-12",  
  "universityName": "andes", "database": "MONGO"}
```

Procedimiento:

1. Realizar una solicitud POST al endpoint: **http://localhost:3000/api/v1/study/MONGO**
2. Enviar como cuerpo un JSON con la información del estudio:

- personId: 123456789
- professionId: 1234
- graduationDate: "1244-12-12"
- universityName: "andes"
- database: "MONGO"

Respuesta del servidor:

- Código: 200 OK
- Cuerpo:

```
{  
  "personId": 123456789,  
  "professionId": 1234,  
  "graduationDate": "1244-12-12",  
  "universityName": "andes",  
  "database": "MONGO",  
  "status": "OK"
```

}

Esto indica que la creación del registro fue exitosa.

18. Edición de un estudio en CLI para MongoDB

A continuación, se realizará una prueba que busca validar que se puede actualizar un estudio almacenado en MongoDB desde la línea de comandos.

```
-----
1 para MariaDB
2 para MongoDB
0 para regresar
Ingrese una opción: 2
-----
1 para ver todos los estudios
2 para crear estudio
3 para eliminar estudio
4 para buscar estudio
5 para editar estudio
0 para regresar
Ingrese una opción: 5
ID Persona (del estudio a editar): 123456789
ID Profesion (del estudio a editar): 1234
Nueva fecha de graduación (YYYY-MM-DD): 1256-12-12
Nuevo nombre de universidad: andinosFC
2025-05-25 21:33:04.570 INFO 7 --- [main] org.mongodb.driver.connection : Opened c
onnection [connectionId{localValue:3, serverValue:11}] to mongo:27017
Estudio actualizado: StudyModelCli(personId=123456789, professionId=1234, graduationDate=1256-12-12, univ
ersityName=andinosFC)
```

Imagen 20. Edición de un estudio en CLI para MongoDB

Procedimiento:

1. Seleccionar el origen de datos: opción 2 (MongoDB).
2. Seleccionar la opción 5 para editar estudio.
3. Ingresar los siguientes datos:
 - ID Persona: 123456789
 - ID Profesión: 1234
 - Nueva fecha de graduación: 1256-12-12
 - Nuevo nombre de universidad: andinosFC
4. Confirmar que se muestra el mensaje:

Estudio **actualizado:**
professionId=1234,
universityName=andinosFC)

StudyModelCli(personId=123456789,
graduationDate=1256-12-12,

5. Verificar que la conexión a MongoDB (mongo:27017) fue exitosa.

Resultado:

El resultado fue exitoso dado que el estudio correspondiente es localizado por personId y professionId, los campos graduationDate y universityName son actualizados y se muestra la confirmación con los nuevos valores en consola.

19. Validación de actualización desde API REST

A continuación, se realizará una consulta GET al endpoint REST /api/v1/study/{database} usando Swagger UI, apuntando a la base de datos MONGO. El objetivo de esta prueba es Confirmar que los cambios realizados en el estudio desde la CLI (MongoDB) se reflejan al consultar la API.

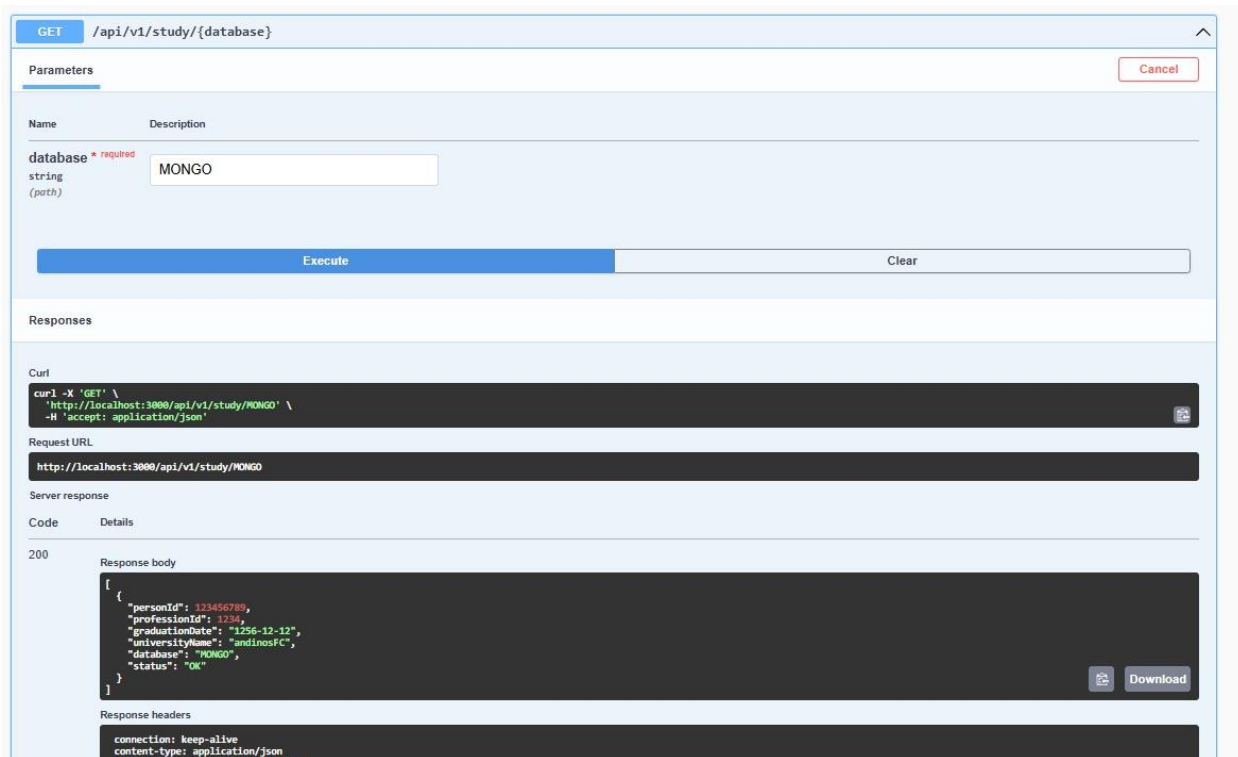


Imagen 21. Validación de actualización desde API REST

Parámetros:

- Endpoint: GET /api/v1/study/{database}
- Parámetro: database = MONGO
- Request URL: http://localhost:3000/api/v1/study/MONGO
- Código de respuesta: 200 OK

Procedimiento:

1. Acceder a la documentación Swagger en <http://localhost:3000>.
2. Usar el endpoint GET /api/v1/study/{database}.
3. En el campo database, ingresar: MONGO.
4. Ejecutar la solicitud.
5. Verificar que el response body contenga los datos actualizados:

```
[
  {
    "personId": 123456789,
    "professionId": 1234,
    "graduationDate": "1256-12-12",
    "universityName": "andinosFC",
    "database": "MONGO",
    "status": "OK"
  }
]
```

Resultado: Se recibió una respuesta HTTP 200 con los valores correctamente actualizados.

Nota: Con este paso finaliza exitosamente toda la etapa de las pruebas.

21. Ejecución en entorno Docker (Windows)

Este paso describe el uso de docker-compose en entorno Windows, incluyendo las instrucciones necesarias para ejecutar los servicios y resolver un problema común relacionado con el CLI y los archivos de inicialización.

```
volumes:
  - mongo_data:/data/db
  - ./scripts/persona_ddl_mongo.js:/docker-entrypoint-initdb.d/ppersona_ddl_mongo.js
  - ./scripts/persona_dml_mongo.js:/docker-entrypoint-initdb.d/ppersona_dml_mongo.js
ports:
  - 27017:27017

Run Service
Mariadb:
image: mariadb:10.3.10
container_name: mariadb
ports:
  - "3307:3306"
environment:
  MYSQL_ROOT_PASSWORD: persona_db
  MYSQL_DATABASE: persona_db
  MYSQL_USER: persona_db
  MYSQL_PASSWORD: persona_db
volumes:
  - maria_data:/var/lib/mysql
  - ./scripts/persona_ddl_maria.sql:/docker-entrypoint-initdb.d/ppersona_ddl_maria.sql
  - ./scripts/persona_dml_maria.sql:/docker-entrypoint-initdb.d/ppersona_dml_maria.sql
restart: always
```

Imagen 22. Ejecución en entorno Docker (Windows)

Procedimiento:

Opción 1. Desde la raíz del proyecto, se deben ejecutar los siguientes comandos:

docker-compose up --build

Esto levanta todos los servicios definidos en el archivo docker-compose.yml.

Nota: El servicio CLI falla en esta primera terminal ya que requiere una consola interactiva propia.

Opción 2. Ejecución manual del CLI (en una terminal separada), mediante el siguiente comando:

docker-compose run --rm -it cli-service

Este comando se utiliza para correr el servicio cli-service en modo interactivo.

- El flag `--rm` elimina el contenedor al salir.
- El flag `-it` permite la interacción mediante consola.

Consideraciones especiales para Docker en Windows:

A veces, la carga inicial de datos falla, haciendo que el servicio REST se caiga la primera vez, la causa de esto es que Docker en Windows presenta inconsistencias al ejecutar los scripts de inicialización (*.js, *.sql) en la carpeta **docker-entrypoint-initdb.d**.

Solución provisional:

Cambiar el nombre de los archivos de inicialización, por ejemplo:

- `persona_ddl_mongo.js` → `ppersona_ddl_mongo.js`
- `persona_dml_maria.sql` → `ppersona_dml_maria.sql`

Esto puede forzar a Docker a no reutilizar una cache previa o a forzar una re-ejecución. Es una solución no óptima pero funcional dentro de las limitaciones de Docker Desktop para Windows.

Archivos Involucrados:

- MongoDB:
volumes:./scripts/ppersona_ddl_mongo.js:/docker-entrypoint-initdb.d/ppersona_ddl_mongo.js
- ./scripts/ppersona_dml_mongo.js:/docker-entrypoint-initdb.d/ppersona_dml_mongo.js

Conclusiones

- La arquitectura hexagonal mejora la escalabilidad y mantenibilidad: Separar el núcleo del sistema de las tecnologías externas facilita su evolución, permitiendo cambiar bases de datos o interfaces sin afectar la lógica de negocio.
- Repository y Service son claves para una buena organización del código: Estos patrones evitan el acoplamiento entre las capas y mejoran la calidad y testabilidad del sistema.

- El uso de dos motores de base de datos (MongoDB y MariaDB) es enriquecedor: Permite comprender y aprovechar las ventajas de cada enfoque (NoSQL vs SQL), adaptando cada uno a casos específicos dentro del sistema.
- Spring Boot acelera el desarrollo, pero puede ser exigente: Aunque provee muchas facilidades (autoconfiguración, integración con Swagger, CLI), exige un conocimiento sólido de conceptos de arquitectura y diseño para evitar errores de novato.
- Durante el desarrollo del laboratorio se implementó con éxito una arquitectura hexagonal que permite interactuar con las bases de datos MariaDB y MongoDB mediante dos interfaces: una API REST documentada con Swagger y una interfaz CLI. Ambas interfaces funcionan sobre la misma lógica de negocio desacoplada, garantizando consistencia en las operaciones realizadas, como crear, consultar y editar estudios. Las pruebas realizadas demostraron que tanto la CLI como la API responden correctamente según la base de datos seleccionada, gracias a una correcta separación de capas, uso de mapeadores y adaptadores específicos. Esta implementación evidencia un sistema modular, extensible y bien estructurado, que cumple con los principios de mantenibilidad, reutilización y facilidad para futuras integraciones.

Lecciones Aprendidas

- El overengineering puede ralentizar mucho el desarrollo de un proyecto pequeño o prototipado, pues aplicar patrones complejos como Hexagonal en estos casos puede añadir una complejidad innecesaria.
- La implementación de varios adaptadores de entrada para distintos accesos a la app (por REST y por consola), nos permitió entender cómo, mediante el buen uso de patrones como Hexagonal, un sistema puede adaptarse a distintos modos de interacción con el usuario manteniendo la lógica de negocio completamente aislada. Esta flexibilidad refuerza la capacidad del sistema para escalar y facilitar futuras adaptaciones o migraciones de tecnología sin necesidad de alterar su núcleo funcional.
- La documentación de la API es clave para el trabajo colaborativo, pues gracias a herramientas como Swagger se puede generar documentación automática y fácil de entender lo cual mejora la colaboración entre equipos de frontend y backend, al proporcionarle a ambos una misma visión acerca de los endpoints disponibles.

- Aplicar la arquitectura hexagonal desde el inicio obliga a pensar en la lógica de negocio como el núcleo del sistema, independiente de cualquier tecnología, lo cual facilita su evolución, pero requiere una planificación más rigurosa y tiempo adicional en fases tempranas del proyecto, para que en el futuro uno no tenga que sufrirla con malos diseños y que, por el contrario, sea más fácil meterle mano al código en etapas tardías.
- El control de versiones y la modularidad son esenciales, pues gracias al uso de GitHub y la segmentación del proyecto en capas y carpetas organizadas fue fácil mantener un flujo de trabajo limpio y controlado.
- El uso del patrón Repositorio es muy útil cuando se requiere desacoplar la lógica de acceso a datos del resto del sistema, lo que no solo mejora la testabilidad del código, sino que también facilita el cambio de tecnología de persistencia sin afectar el dominio ni la lógica de negocio.
- Al tener que usar distintas bases de datos (como en este caso fueron Mongo y María) aprendimos a gestionar adecuadamente las dependencias y a implementar inyección de dependencias para adaptar al sistema a distintos contextos sin cambiar la lógica de negocio.

Referencias

- <https://www.youtube.com/watch?v=yJy5Ha4v2jQ>
- <https://martinfowler.com/bliki/PresentationDomainDataLayering.html>
- <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>
- <https://spring.io/guides/tutorials/rest>
- https://swagger.io/docs/specification/v3_0/about/
- <https://www.mongodb.com/resources/products/compatibilities/spring-boot>
- <https://mariadb.com/docs/server/connect/programming-languages/java-r2dbc/spring/>
- <https://swagger.io/tools/swagger-ui/>

- <https://docs.spring.io/spring-data/jpa/reference/repositories/custom-implementations.html>
- <https://www.mongodb.com/developer/code-examples/java/rest-apis-java-spring-boot/>