Laboratorio 2

Arquitectura de Software

Autores:
Juan Felipe González Quintero
Oscar Alejandro Rodríguez Gómez
Andrés Felipe Ruge Passito

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE SISTEMAS
BOGOTÁ, D.C.

2 de junio de 2025

Contenido

1. Introducción

Para este laboratorio se plantea la implementación de una aplicación con estilo arquitectónico hexagonal (monolítico) haciendo uso de los patrones Service y Repository. El lenguaje seleccionado para la codificación será springboot (java), y se realiza la integración con dos tipos de bases de datos, una relacional (MariaDB) y la otra no relacional (MongoDB). El objetivo de este laboratorio es ampliar nuestro conocimiento en base a un estilo arquitectural distinto y además utilizando bases de datos de diferentes orígenes, lo que nos proporciona herramientas para trabajar en distinto frameworks.

La aplicación se compone de cuatro entidades (profesión, estudios, persona y teléfono), de las cuáles se debe realizar el CRUD y luego probar los endpoints correspondientes. A continuación, se explican a detalle los componentes del stack tecnológico que integra la aplicación.

1.1. Arquitectura Hexagonal

Una arquitectura hexagonal es un enfoque de diseño arquitectónico y desarrollo de software que busca desacoplar el dominio de la aplicación de sus elementos externos como bases de datos, interfaces de usuario o servicios de terceros. En esta arquitectura, el dominio contiene la lógica de negocio principal y se comunica con el exterior a través de puertos (interfaces) y adaptadores, lo que permite una alta modularidad y facilita el cambio o sustitución de tecnologías sin afectar el funcionamiento interno.

Este estilo arquitectónico es ideal para aplicaciones que requieren mantenibilidad, pruebas automatizadas y evolución constante, ya que permite aislar el comportamiento del negocio de las preocupaciones técnicas. Sin embargo, su implementación puede resultar compleja en proyectos pequeños o con recursos limitados, debido al esfuerzo inicial que implica definir y mantener las capas de abstracción adecuadas.

1.2. Patrones de diseño Repository y Service

Repository y Service son patrones que ayudan a organizar la lógica de una aplicación separando responsabilidades de manera clara. El patrón Repository actúa como una capa de abstracción entre el dominio y la fuente de datos, permitiendo acceder, almacenar o eliminar objetos del negocio sin que el resto de la aplicación se preocupe por los detalles de cómo se realiza ese acceso (por ejemplo, a una base

de datos o una API). Por su parte, el patrón Service encapsula la lógica de negocio, coordinando acciones que involucran una o varias operaciones del dominio y utilizando repositorios cuando es necesario. Esta separación mejora la mantenibilidad, facilita las pruebas y promueve un código más limpio y reutilizable.

1.3. JDK 11

JDK 11 (Java Development Kit 11) es una de las versiones de soporte a largo plazo (LTS) del kit de desarrollo de Java, lanzada por Oracle en 2018, que proporciona un entorno completo para compilar, ejecutar y depurar aplicaciones Java. Esta versión introdujo mejoras en el rendimiento, seguridad y mantenimiento del lenguaje, y eliminó algunas características obsoletas presentes en versiones anteriores, como Java EE y JavaFX del JDK principal.

JDK 11 incluye herramientas esenciales como el compilador `javac`, la máquina virtual de Java (JVM), y bibliotecas estándar actualizadas. Es ampliamente adoptado en entornos empresariales debido a su estabilidad y soporte extendido, convirtiéndolo en una base confiable para el desarrollo de aplicaciones modernas.

1.4. Spring Boot

Spring Boot es un framework de desarrollo para aplicaciones Java que simplifica la creación de proyectos basados en Spring. Su objetivo principal es permitir el desarrollo rápido y eficiente de aplicaciones listas para producción, ofreciendo una estructura predeterminada, un servidor embebido (como Tomcat), y una amplia integración con herramientas y tecnologías modernas.

Spring Boot permite crear aplicaciones web, RESTful APIs, microservicios y más, con facilidad, gracias a configuraciones automáticas inteligentes y componentes preconfigurados que reducen el esfuerzo de desarrollo y mejoran la productividad.

1.5. MongoDB

MongoDB es una base de datos NoSQL orientada a documentos que almacena la información en estructuras similares a JSON llamadas BSON, lo que permite manejar datos semi-estructurados de forma flexible y dinámica. Está diseñada para escalar horizontalmente y soportar grandes volúmenes de datos distribuidos en múltiples servidores, facilitando la alta disponibilidad y el rendimiento.

MongoDB es ideal para aplicaciones modernas que requieren rapidez en el desarrollo, flexibilidad en el esquema y capacidad para manejar datos variados sin necesidad de un esquema rígido como en las bases de datos relacionales tradicionales.

1.6. MariaDB

MariaDB es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, derivado de MySQL, que utiliza el lenguaje SQL para el manejo y consulta de datos estructurados. Está diseñado para ser totalmente compatible con MySQL, pero con mejoras en rendimiento, seguridad y nuevas funcionalidades desarrolladas de forma independiente.

MariaDB es ampliamente utilizado en aplicaciones empresariales y sitios web gracias a su estabilidad, escalabilidad y soporte para replicación, transacciones y almacenamiento en múltiples motores. Su

naturaleza abierta y el respaldo de una comunidad activa lo convierten en una opción robusta y confiable para gestionar datos críticos en diversos entornos.

1.7. REST

REST (Representational State Transfer) es un estilo de arquitectura para el diseño de servicios web que se basa en principios como la utilización de métodos HTTP estándar (GET, POST, PUT, DELETE) y el acceso a recursos identificados por URLs. En una API RESTful, cada recurso (como usuarios, productos o pedidos) puede ser representado y manipulado mediante peticiones HTTP, lo que permite una comunicación clara y uniforme entre sistemas.

REST es ampliamente adoptado por su simplicidad, escalabilidad y compatibilidad con múltiples plataformas y lenguajes, lo que lo hace ideal para construir APIs ligeras y eficientes en aplicaciones web y móviles.

1.8. CLI

CLI (Command Line Interface) es una interfaz de usuario que permite interactuar con un sistema operativo o una aplicación mediante la introducción de comandos en una consola o terminal. A diferencia de las interfaces gráficas, la CLI ofrece un control más directo y preciso sobre las operaciones, lo que la hace especialmente útil para usuarios avanzados, desarrolladores y administradores de sistemas. Mediante la CLI, se pueden realizar tareas como navegar por el sistema de archivos, ejecutar programas, automatizar procesos y configurar sistemas de manera rápida y eficiente.

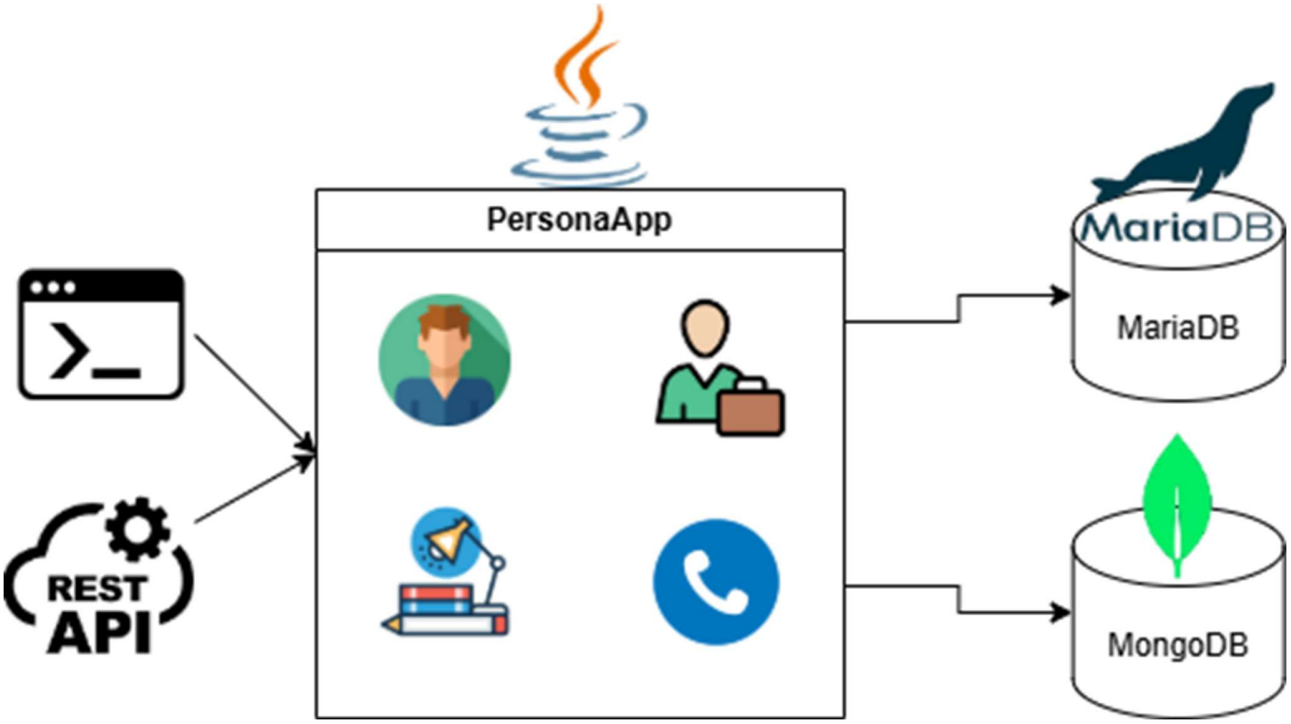
Su bajo consumo de recursos y capacidad para integrarse con scripts la convierten en una herramienta poderosa en entornos técnicos.

1.9. Swagger 3

Swagger es una herramienta de código abierto utilizada para documentar, diseñar y probar APIs RESTful de forma interactiva. Su núcleo es el uso de un formato estándar que describe los endpoints, parámetros, respuestas, y otros detalles de una API en un archivo legible tanto por humanos como por máquinas (generalmente en formato JSON o YAML). Swagger genera automáticamente una interfaz web que permite explorar y probar las operaciones de la API sin necesidad de escribir código adicional o usar herramientas externas como Postman.

2. Modelado de la aplicación

2.1. Diagrama de Alto nivel



Este diagrama de alto nivel representa la arquitectura general del sistema PersonaApp, una aplicación desarrollada en Java que permite gestionar información de personas, profesiones, estudios y teléfonos.

Existen dos adaptadores de entrada: una interfaz de línea de comandos (CLI) y una API REST. Ambos interactúan con el núcleo de la aplicación, que sigue principios de arquitectura limpia. Dependiendo del tipo de entidad, la aplicación se comunica con dos bases de datos distintas:

- MariaDB para almacenar personas, teléfonos, profesiones y estudios.
- MongoDB para gestionar personas, teléfonos, profesiones y estudios.

Esto permite desacoplar la lógica de negocio de los mecanismos de entrada/salida y facilita la escalabilidad y mantenibilidad del sistema.

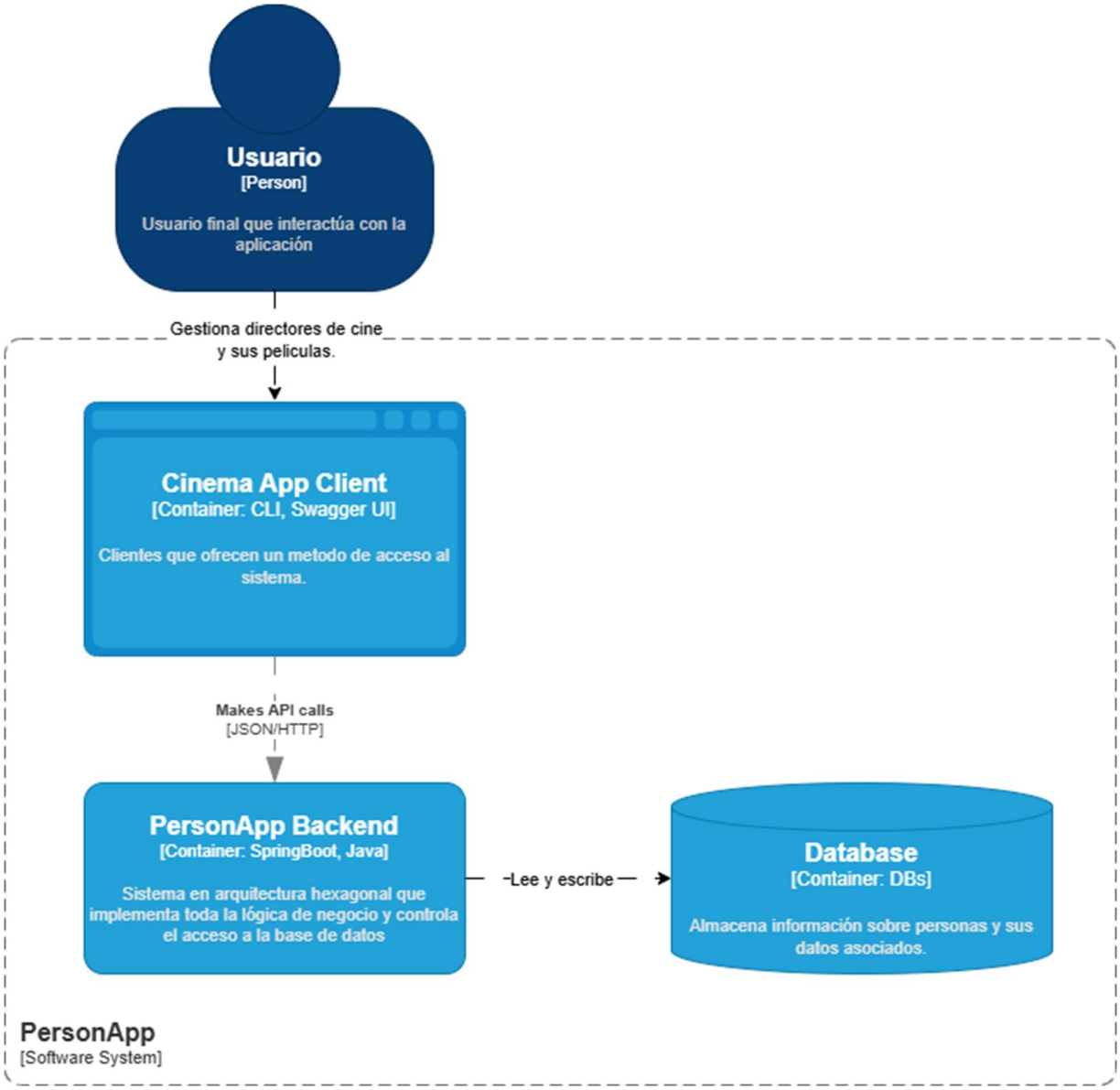
2.2. Diagrama de Contexto

Este diagrama de contexto muestra la relación entre el usuario y el sistema PersonApp desde una perspectiva externa y simplificada.

- El usuario representa a una persona interesada en administrar datos personales, como nombres, teléfonos, profesiones y estudios.
- PersonApp es el sistema software que ofrece las funcionalidades necesarias para gestionar esta información de forma centralizada.

La interacción principal es que el usuario gestiona personas con sus datos asociados a través de PersonApp. Este diagrama permite entender rápidamente quién usa el sistema y para qué propósito, sin entrar aún en detalles técnicos o de implementación.

2.3. Diagrama de Contenedores



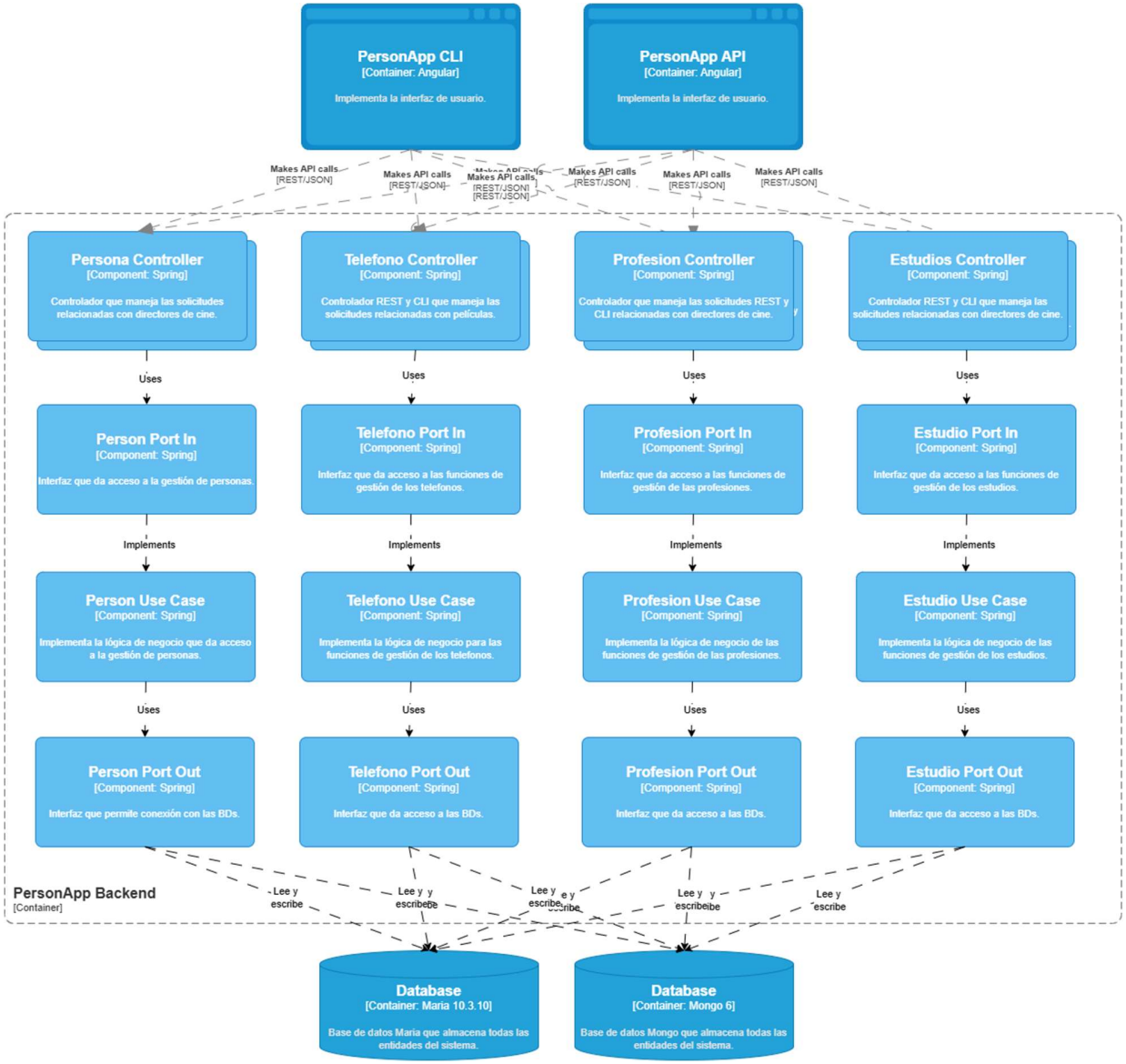
Este diagrama de contenedores describe los principales componentes que conforman el sistema PersonApp, así como sus interacciones.

- El usuario es el actor principal que interactúa con el sistema para gestionar información (en este ejemplo, de directores de cine y sus películas).
- El Cinema App Client representa los adaptadores de entrada (por ejemplo, una interfaz de línea de comandos o Swagger UI). Estos clientes permiten al usuario enviar solicitudes al sistema mediante llamadas HTTP/JSON.
- El PersonApp Backend, desarrollado en Spring Boot y Java, es el núcleo del sistema. Implementa una arquitectura hexagonal, encapsulando la lógica de negocio y controlando el acceso a las bases de datos mediante puertos y adaptadores.

- La base de datos (o conjunto de bases de datos) actúa como contenedor de persistencia, almacenando toda la información relacionada con personas y sus datos asociados (como profesiones, estudios y teléfonos).

En conjunto, el diagrama muestra cómo se separan las responsabilidades entre los distintos contenedores para lograr una arquitectura modular, escalable y mantenible.

2.4. Diagrama de Componentes



Este diagrama de componentes muestra de forma estructurada la arquitectura interna de PersonApp, organizada bajo el enfoque de arquitectura hexagonal, también conocida como puertos y adaptadores. La aplicación está dividida en capas claramente separadas, lo que facilita su comprensión, mantenimiento y escalabilidad.

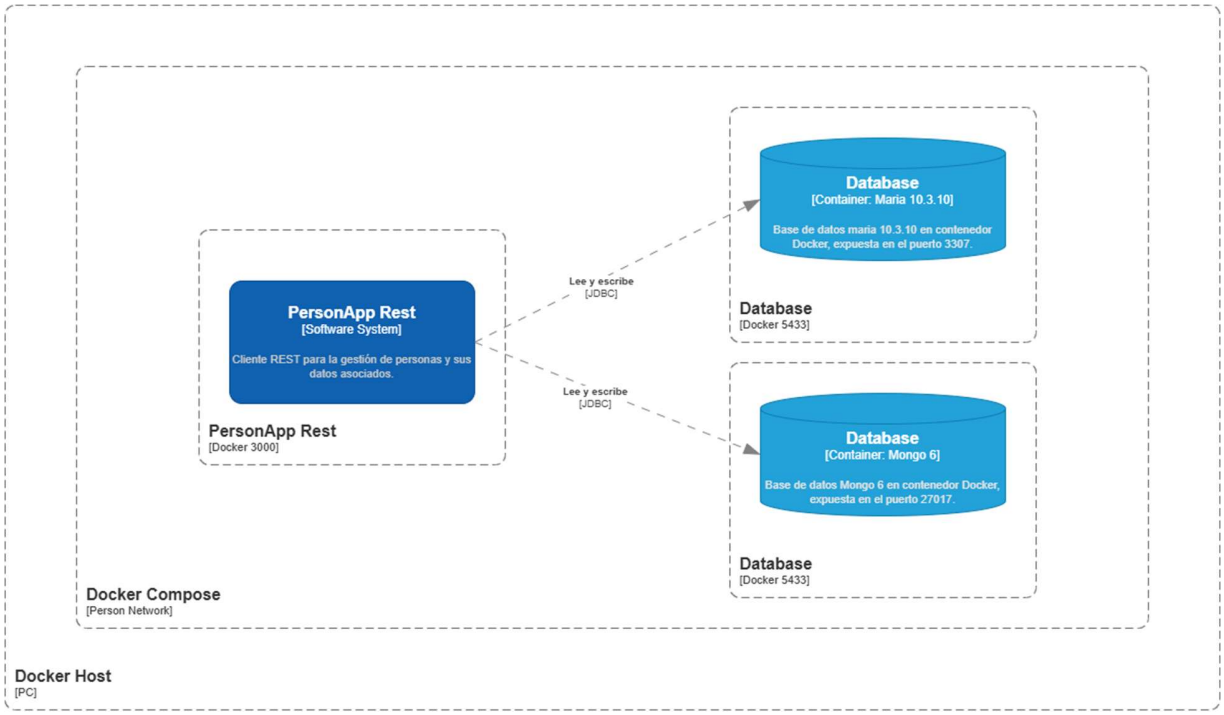
En la parte superior del diagrama se encuentran los adaptadores de entrada, que representan las interfaces de usuario. Existen dos tipos: PersonApp CLI, una interfaz de línea de comandos, y PersonApp API, que expone una API REST documentada mediante Swagger. Ambas permiten a los usuarios interactuar con el sistema enviando solicitudes para gestionar datos como personas, teléfonos, profesiones y estudios.

Las solicitudes del usuario son recibidas por los controladores (Controllers), uno por cada tipo de entidad. Estos controladores están desarrollados con Spring y se encargan de manejar las peticiones entrantes, ya sea desde el CLI o desde la API REST. Su rol es traducir estas solicitudes a llamadas a las interfaces definidas en los puertos de entrada (Port In), que son contratos o interfaces que describen las operaciones del sistema.

Cada puerto de entrada es implementado por un caso de uso (Use Case), que contiene la lógica de negocio específica para cada entidad. Esta capa centraliza las reglas del sistema, asegurando que el comportamiento del dominio se mantenga coherente sin importar desde qué tipo de cliente se realice la solicitud. Para cumplir su función, los casos de uso hacen uso de los puertos de salida (Port Out), que son interfaces que abstraen la interacción con los mecanismos de persistencia. Esta capa permite cambiar o extender las fuentes de datos sin alterar la lógica del negocio.

Finalmente, los adaptadores de salida se conectan a las bases de datos correspondientes. Se usan dos bases de datos diferentes: MariaDB, que almacena la información relacionada con personas y profesiones; y MongoDB, que guarda los datos de estudios y teléfonos. Esto permite aprovechar las ventajas de diferentes tecnologías de persistencia según el tipo de información.

2.5. Diagrama de Despliegue



Este diagrama de despliegue ilustra cómo se implementa la aplicación PersonApp Rest en un entorno basado en Docker, especificando los contenedores, puertos y conexiones involucradas. En el centro del

despliegue se encuentra el servicio PersonApp Rest, que corresponde al sistema principal encargado de la gestión de personas y sus datos asociados (como teléfonos, estudios y profesiones). Este servicio está desplegado dentro de un contenedor Docker que expone el puerto 3000, y se comunica con dos bases de datos a través del protocolo JDBC.

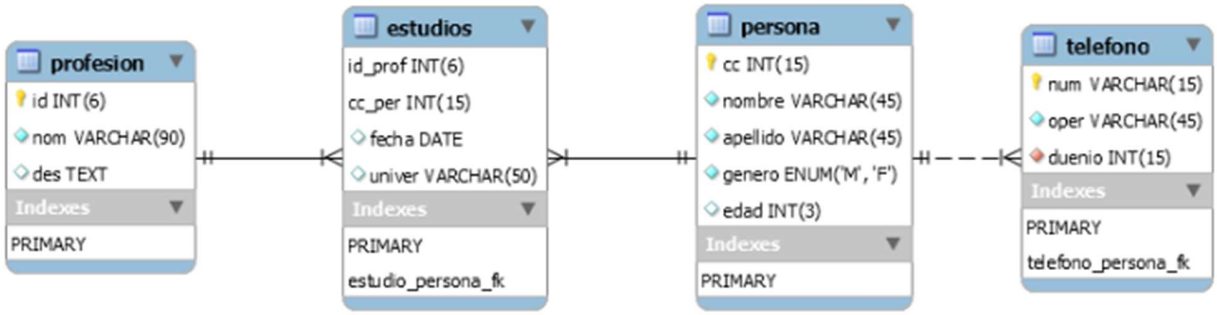
Las bases de datos están contenidas también en contenedores independientes dentro del mismo entorno Docker. La primera es una base de datos MariaDB (versión 10.3.10), la cual se encuentra expuesta en el puerto interno 3307 y conectada a través del puerto 5433 en Docker. Esta base de datos almacena parte de la información estructurada de la aplicación.

La segunda base de datos es MongoDB (versión 6), desplegada en otro contenedor y expuesta en el puerto 27017. Al igual que MariaDB, este contenedor está disponible internamente en Docker a través del puerto 5433, permitiendo el acceso desde el cliente REST y el cliente CLI.

3. Modelado de la aplicación

3.1. Configuración MongoDB y MariaDB

Para preparar el entorno de desarrollo, lo primero que se realizó fue la creación de la base de datos persona_db en MariaDB, así como la configuración de un esquema equivalente en MongoDB, ambos construidos a partir del siguiente modelo de datos y de los siguientes scripts:



El modelo de datos de este proyecto se caracteriza por su implementación en dos sistemas de bases de datos: MariaDB y MongoDB. En MariaDB, se define una estructura relacional con cuatro tablas principales: persona, profesion, telefono y estudios. Se establecen relaciones clave como una relación uno a muchos entre persona y telefono, permitiendo que una persona tenga múltiples teléfonos, y una relación muchos a muchos entre persona y profesion, resuelta mediante la tabla intermedia estudios, que también almacena información adicional como la universidad y la fecha del estudio. Estas relaciones están soportadas por claves foráneas que garantizan la integridad referencial y la consistencia de los datos.

Por otro lado, en MongoDB se replica la colección persona, modelando cada documento como una representación directa de la entidad persona del modelo relacional, lo que permite una consulta ágil y flexible en entornos donde se requiere escalabilidad y alto rendimiento. Ambos entornos fueron configurados con sus respectivos usuarios, roles y privilegios, y se cargaron los datos iniciales mediante instrucciones DML en MariaDB e inserciones insertMany en MongoDB, asegurando coherencia y sincronización de la información entre ambos sistemas.

3.2. Implementación

3.2.1. DockerFile y Docker-Compose

Para la construcción de los archivos de configuración de los Docker-compose y Dockerfile se ejecutaron los siguientes pasos:

Docker-compose.yml

Este archivo define un entorno de desarrollo compuesto por 3 servicios, los cuáles son la base de datos previamente configurada MariaDB, la base de datos MongoDB, y la aplicación hecha en SpringBoot, a la cuál llamamos *laboratorio2-rest.* A continuación, se detallan las secciones en las cuáles se divide el archivo y que garantizan el correcto funcionamiento de este:

```
mariadb:
  image: mariadb:10.3.10
  container_name: mariadb-personapp
  ports:
    - "3307:3306"
  environment:
    MYSQL_ROOT_PASSWORD: pass123
    MYSQL_DATABASE: persona_db
    MYSQL_USER: persona_db
    MYSQL_PASSWORD: pass123
  volumes:
    - ./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
```

En esta primera sección del archivo (después de poner la línea *services:* para definir cuáles serán los servicios que componen el entorno) se define el servicio de mariadb el cuál utiliza específicamente la versión 10.3.10 para la creación de la imagen. Se expone esta imagen en el puerto interno 3306 en el host como el 3307 y se configuran las variables de entorno necesarias como las credenciales de acceso. Por último, en el apartado *volumes:* se definen los scripts a ejecutar para la definición de la base de datos (DDL) y la carga de datos (DML). El contenedor se reinicia en caso de que llegase a caerse con la última línea de la sección.

```
mongodb:
  image: mongo:6
  container_name: mongodb-personapp
  ports:
    - "27017:27017"
  volumes:
    - ./scripts/persona_ddl_mongo.js:/docker-entrypoint-initdb.d/persona_ddl_mongo.js
    - ./scripts/persona_dml_mongo.js:/docker-entrypoint-initdb.d/persona_dml_mongo.js
  restart: unless-stopped
```

En este segundo apartado se define el servicio de la base de datos MongoDB, el cuál utilizará para la creación de la imagen la versión 6. Igual que con MariaDB se expone al contenedor en un puerto

específico, para el caso 27017. Luego se definen en la sección apartado *volumes:* los scripts a ejecutar para la definición de la colección (DDL) y la carga de datos (DML). Por último, en la línea final se agrega un reinicio a menos que el usuario lo detenga manualmente.

```
laboratorio2-rest:
  image: lab2-rest-image
  container_name: lab2-rest
  build: .
  ports:
    - "3000:3000"
  environment:
    - SPRING_DATASOURCE_URL=jdbc:mariadb://mariadb:3306/persona_db
    - SPRING_DATASOURCE_USERNAME=root
    - SPRING_DATASOURCE_PASSWORD=pass123
    - SPRING_DATA_MONGODB_URI=mongodb://mongodb:27017/persona_db
  depends_on:
    - mariadb
    - mongodb
  # Añadir un delay antes de iniciar la aplicación
  command: sh -c "sleep 30 && java -jar /app/laboratorio2-rest.jar"
```

En este último apartado se define el servicio de la aplicación de springboot, la cuál de no existir se construye con la línea "*build: . ".* Luego, se expone la aplicación en el puerto 3000 y después se definen las variables de entorno, en las que se incluye la conexión a las bases de datos mediante sus url. Después, en la sección *depends on* se asegura de que el contenedor se levante justo después de haber iniciado los de las bases de datos mariadb y mongodb, y para garantizar que esto sea así, también se añade un comando que hace que se espere 30 segundos antes de ejecutar el JAR, dando tiempo a los contenedores de las bases de datos para levantarse.

Dockerfile

Este archivo crea una imagen de una aplicación **Java (Spring Boot)** siguiendo el patrón multietapa, para compilar y luego ejecutar solo lo que es necesario. A continuación, se segmenta el archivo con fines explicativos.

```
FROM maven:3.9.9-amazoncorretto-11 AS build
```

Se inicia la construcción de la imagen con base en Maven incluyendo el JDK Amazo Corretto 11.

```
WORKDIR /app
COPY pom.xml .
COPY common/pom.xml ./common/
COPY domain/pom.xml ./domain/
COPY application/pom.xml ./application/
COPY maria-output-adapter/pom.xml ./maria-output-adapter/
COPY mongo-output-adapter/pom.xml ./mongo-output-adapter/
COPY rest-input-adapter/pom.xml ./rest-input-adapter/
COPY cli-input-adapter/pom.xml ./cli-input-adapter/
```

Luego, se establece /app como directorio de trabajo dentro del contenedor para la ejecución del resto de comandos. Todas las rutas siguientes serán relativas a esas. Se copian los pom.xml de cada módulo. Esto permite a Maven resolver dependencias lo antes posible, sin necesidad de copiar aún el código fuente, optimizando el uso de la caché para evitar recompilar todo al evidenciar cambios en el código fuente.

```
COPY common/src ./common/src
COPY domain/src ./domain/src
COPY application/src ./application/src
COPY maria-output-adapter/src ./maria-output-adapter/src
COPY mongo-output-adapter/src ./mongo-output-adapter/src
COPY rest-input-adapter/src ./rest-input-adapter/src
COPY cli-input-adapter/src ./cli-input-adapter/src
```

En esta parte se copian todos los códigos fuente de todos los módulos del proyecto Maven, esto siguiendo el estilo arquitectónico hexagonal haciendo uso de los puertos de entrada y salida.

```
RUN mvn install
```

Se ejecuta la construcción del proyecto: compila el código y genera los artefactos (.jar).

```
FROM amazoncorretto:11.0.27-alpine3.21 AS deploy

WORKDIR /app
COPY --from=build /app/rest-input-adapter/target/rest-input-adapter-0.0.1-SNAPSHOT.jar /app/laboratorio2-rest.jar

CMD ["java", "-jar", "/app/laboratorio2-rest.jar"]
```

En esta parte del archivo se reduce considerablemente el tamaño de la imagen y luego se establece el directorio de trabajo donde se colocará él .jar para ejecutarlo, para luego copiar desde la primera etapa (build) el .jar generado por el módulo rest-input-adapter y lo guarda con un nombre más genérico.

Por último, se define el comando que se ejecutará al iniciar el contenedor utilizando la forma array para mejor manejo de espacios/argumentos.

3.2.2. Port

Siguiendo la implementación del port de persona (tanto para input como para output) sumistrado por el profesor para el proyecto, se realizaron los respectivos port de las clases: Phone, Profession y Study, a continuación, se describirá que se ve en cada una de estas clases.

3.2.2.1. Input

- **Phone:** Este puerto se encargará de manejar todo lo relacionado con los teléfonos dentro de la aplicación. Es la interfaz de entrada para que los casos de uso puedan interactuar con la lógica de negocio que maneja los teléfonos.
  Gracias a este puerto, podemos crear nuevos teléfonos, editar los que ya existen, eliminarlos, buscarlos por número o ver la lista completa. Incluso nos permite saber cuántos teléfonos hay registrados.
  - **setPersintence(PhoneOutputPort phoneOutputPortPersintence)**: Establece el puerto de salida que se usará para las operaciones de persistencia.
  - **create(Phone phone):** Crea un nuevo registro de tipo Phone y lo retorna.
  - **edit(String number, Phone phone):** Edita la información de un número telefónico existente identificado por number.
  - **drop(String number):** Elimina un teléfono del sistema según su número.
  - **findAll()**: Retorna una lista con todos los teléfonos registrados.
  - **findOne(String number)**: Busca y retorna un teléfono específico usando su número.
  - **count()**: Devuelve el total de teléfonos registrados en el sistema.
  - **List<Phone> findByOwner(Integer ownerIdentification):** Retorna una lista de teléfonos asociados a una persona específica, identificada por su número de identificación. Lanza excepción si no existe esa persona.

- **Profession:** Este puerto es el que define las operaciones que se pueden realizar sobre las profesiones dentro del sistema. A través de él, se puede crear una nueva profesión, editar una ya existente, eliminarla, o buscarla por su identificador. Además, permite consultar todos los registros de profesiones y saber cuántas hay actualmente en el sistema.
  - **setPersintence(ProfessionOutputPort ProfessionOutputPortPersintence):** Establece el puerto de salida que se usará para las operaciones de persistencia.
  - **create(Profession profession):** Crea un nuevo registro de tipo Profession y lo retorna.
  - **edit(Integer identification, Profession profession):** Edita la información de una profesión identificada por su ID.
  - **drop(Integer identification):** Elimina una profesión del sistema según su ID.
  - **findAll():** Retorna una lista con todas las profesiones registradas.
  - **findOne(Integer identification):** Busca y retorna una profesión específica usando su ID.
  - **count():** Devuelve el total de profesiones registradas en el sistema.
  - **getStudies(Integer identification):** Retorna la lista de estudios asociados a una profesión según su ID.

- **Study:** En este puerto se definen todas las acciones que se pueden realizar sobre los estudios registrados en el sistema. Es el punto de entrada para gestionar esta entidad desde la lógica de negocio. A través de él, se puede crear un nuevo estudio, editar uno existente, eliminar un estudio específico según la persona y la profesión involucradas, o consultar estudios individuales o en conjunto. Por último, también permite obtener el total de estudios registrados.
  - **setPersintence(StudyOutputPort phoneOutputPortPersintence):** Establece el puerto de salida que se usará para las operaciones de persistencia relacionadas con estudios.

- **create(Study study):** Crea un nuevo registro de estudio y lo retorna.
- **edit(Study study):** Edita un estudio existente. Si no se encuentra, lanza una excepción.
- **drop(Integer personidentification, Integer professionIdentification):** Elimina un estudio específico asociando una persona con una profesión.
- **findAll():** Retorna una lista con todos los estudios registrados en el sistema.
- **findOne(String number):** Busca y retorna un estudio específico por su identificador.
- **count():** Devuelve la cantidad total de estudios registrados.
- **List<Study> findByPerson(Integer personId):** Devuelve todos los estudios realizados por una persona según su identificación. Si la persona no existe, se lanza una excepción.
- **List<Study> findByProfession(Integer professionId):** Devuelve todos los estudios que están relacionados con una profesión específica, identificada por su ID. Si no se encuentra esa profesión, lanza una excepción.

3.2.2.2. Output

- **Phone:** Este puerto de salida define las operaciones necesarias para interactuar con las bases de datos en lo que respecta a los teléfonos. Por lo que, usando este puerto, la aplicación puede guardar un teléfono, eliminarlo, buscar todos los teléfonos existentes o consultar uno específico por su número.
  - **Phone save(Phone phone):** Guarda un teléfono en la capa de persistencia, puede ser una creación o una actualización y retorna el objeto almacenado.
  - **Boolean delete(String number):** Elimina un teléfono del sistema a partir de su número. Devuelve true si se eliminó correctamente, false en caso contrario.
  - **List<Phone> find():** Retorna una lista con todos los teléfonos registrados en el sistema.
  - **Phone findById(String number):** Busca un teléfono específico a partir de su número y lo devuelve. Equivalente al método findByNumberPhone, pero con un nombre más claro.
  - **List<Phone> findByOwnerId(Integer ownerIdentification):** Devuelve todos los teléfonos que están asociados a una persona según su número de identificación.

- **Profession:** Este puerto de salida permite a la aplicación interactuar con la capa donde se almacenan los datos de las profesiones, en este caso la base de datos.
  Su principal función es servir como intermediario entre la lógica de negocio y el sistema de persistencia, manejando tareas como guardar profesiones, eliminarlas, consultarlas todas o buscar una en particular.
  - **Profession save(Profession profession):** Guarda o actualiza una profesión en el sistema de persistencia y retorna el objeto almacenado.
  - **Boolean delete(String number):** Elimina una profesión usando un identificador (aunque el nombre del parámetro es number, en este contexto parece referirse al código de la profesión). Retorna true si se eliminó correctamente.
  - **List<Profession> find():** Devuelve una lista con todas las profesiones registradas.
  - **Profession findById(Integer identification):** Busca y retorna una profesión según su identificador único.

- **Study:** Este puerto se encarga de todo lo relacionado con guardar, eliminar y consultar los estudios que una persona ha hecho. Es como el puente entre la lógica de la aplicación y la base de datos.
  Con este puerto, la aplicación puede guardar, eliminar, consultar o buscar estudios específicos sin preocuparse por cómo se almacenan los datos realmente.

    - **Study save(Study study):** Guarda un nuevo estudio o actualiza uno existente en el sistema de persistencia. Retorna el objeto guardado.
    - **Boolean delete(Integer personId, Integer professionId):** Elimina un estudio específico según el ID de la persona y el ID de la profesión asociados. Devuelve true si la eliminación fue exitosa.
    - **List<Study> find():** Devuelve una lista con todos los estudios almacenados en el sistema.
    - **Study findById(Integer personId, Integer professionId):** Busca y devuelve un estudio en particular, identificado por el par persona-profesión.
    - **List<Study> findByPersonId(Integer personId):** Retorna todos los estudios asociados a una persona en específico, utilizando su ID.
      List<Study> findByProfessionId(Integer professionId): Retorna todos los estudios vinculados a una profesión específica, identificada por su ID.

3.2.3. UseCase

Los UseCase en la arquitectura hexagonal representan la lógica de negocio principal de la aplicación. Sirven como capa intermedia entre: los controladores/adaptadores de entrada (REST, CLI), y **los** adaptadores de salida (repositorios, bases de datos, etc.).

Su objetivo es orquestar operaciones sobre las entidades del dominio, aplicar reglas de negocio y delegar persistencia a través de puertos (interfaces).

A continuación, se explica en detalle como se definieron todos los UseCase para las diferentes clases de las cuáles se compone el programa.

3.2.3.1. PersonUseCase

Implementa PersonInputPort. Orquesta todas las operaciones sobre la entidad *Person* y usa el puerto PersonOutputPort para interactuar con la base de datos.

Métodos:

- **create(Person person):** crea una nueva persona.
- **edit(Integer id, Person person):** edita una persona si existe.
- **drop(Integer id):** elimina una persona si existe.
- **findAll():** lista todas las personas.
- **findOne(Integer id):** busca una persona por ID.
- **count():** cantidad total de personas.
- **getPhones(Integer id):** devuelve teléfonos asociados a una persona.

- **getStudies(Integer id):** devuelve estudios asociados a una persona.

3.2.3.2. PhoneUseCase

Implementa PhoneInputPort. Orquesta todas las operaciones sobre la entidad *Phone*. Este UseCase usa el puerto PhoneOutputPort para interactuar con la base de datos y también utiliza PersonOutputPort para verificar que el dueño del teléfono realmente exista.

Métodos:

- **create(Phone phone):** crea un teléfono.
- **edit(String number, Phone phone):** edita un teléfono si existe.
- **drop(String number):** elimina un teléfono si existe.
- **findAll():** lista todos los teléfonos.
- **findOne(String number):** busca un teléfono por número.
- **count():** cantidad total de teléfonos.
- **findByOwner(Integer ownerId):** obtiene teléfonos de una persona específica.

3.2.3.3. ProfessionUseCase

Implementa ProfessionInputPort. Orquesta todas las operaciones sobre la entidad *Profession*. Este UseCase usa el puerto ProfessionOutputPort para interactuar con la base de datos.

Métodos:

- **create(Profession profession):** crea una nueva profesión.
- **edit(Integer id, Profession profession):** edita una profesión si existe.
- **drop(Integer id):** elimina una profesión si existe.
- **findAll():** lista todas las profesiones.
- **findOne(Integer id):** busca una profesión por su identificador.
- **count():** cuenta el número de profesiones.
- **getStudies(Integer id):** devuelve los estudios relacionados con una profesión.

3.2.3.4. StudyUseCase

Implementa StudyInputPort. Orquesta todas las operaciones sobre la entidad *Study*. Este UseCase usa el puerto StudyOutputPort para interactuar con la base de datos y también utiliza PersonOutputPort para verificar que una persona a la cuál se le asignan esos estudios exista. Por último, este UseCase también implementa ProfessionOutputPort para validar la profesión con la cual se asocia.

Métodos:

- **create(Study study):** crea un nuevo estudio.
- **edit(Integer personId, Integer professionId, Study study):** edita si existe combinación persona-profesión.
- **drop(Integer personId, Integer professionId):** elimina un estudio si existe.
- **findAll():** lista todos los estudios.

- **findOne(Integer personId, Integer professionId):** busca un estudio específico.
- **count():** cantidad de estudios.
- **findByPerson(Integer personId):** busca estudios de una persona.
- **findByProfession(Integer professionId):** busca estudios de una profesión.

3.2.4. Cli-input-adapter

El paquete cli-input-adapter forma parte de los Adapters de Entrada en una arquitectura hexagonal, su propósito principal es propósito principal es permitir que los usuarios interactúen con la aplicación desde la línea de comandos (consola), actuando como puente entre el usuario y los **casos de uso (UseCases)** definidos en la lógica de negocio. A continuación, se desglosan los roles que tiene cada parte del cli-input-adapter (adaptadores, mappers y menús):

3.**2.4.1. Adaptadores**

Traducen las acciones del usuario en llamadas a los casos de uso (UseCase) de la lógica de negocio, y se encargan de inyectar el puerto de salida según el motor de base de datos seleccionado (MariaDB o MongoDB).

3.2.4.2. Mappers

Permiten la conversión de objetos del dominio hacia modelos que pueden ser mostrados por consol**a**, y viceversa. Separan la lógica de dominio del formato de presentación.

3.2.4.3. Menús

Organiza la interacción con el usuario final. Funciona como el "router" de la CLI: presenta las opciones, lee la entrada, y delega a los adaptadores correspondientes.

3.2.4.4 PersonAppCli

Es la clase principal del CLI. Esta contiene el main() y ejecuta menuPrincipal.inicio(). Es el punto de arranque de la aplicación en consola.

En general, todos estos componentes permiten usar la aplicación desde consola como si fuera una interfaz gráfica o una API REST, pero en línea de comandos.

3.2.5. Rest-input-adapter

La carpeta rest-input-adapter representa el adaptador de entrada REST en una arquitectura hexagonal. Su objetivo principal es exponer la lógica de la aplicación a través de una API HTTP usando Spring Boot, para que otros sistemas (como el frontend, aplicaciones móviles o clientes externos) puedan interactuar con ella.

Desde aquí se traducen las solicitudes REST en llamadas al caso de uso que corresponda, se manejan las entradas y salidas de datos, se decide con qué base de datos trabajar (Mongo o Maria) y se convierten objetos de dominio y DTOs.

A continuación, se desglosan los componentes que integran al adaptador de entrada REST:

3.2.5.1. Adaptadores

Son el puente entre el controlador REST y la lógica de negocio. Este componente recibe la base de datos elegida (MONGO o MARIA) y configura el puerto de salida correspondiente, llama a los métodos del caso de uso dependiendo de la clase que implemente y usa el mapper para transformar los objetos de dominio en DTOs (PersonaResponse) o viceversa.

3.2.5.2. Controladores

Este componente está anotado con @RestController y es el punto de entrada HTTP. Desde aquí se mapean las rutas HTTP a métodos Java, se llama al adapter para ejecutar las operaciones y se devuelve los resultados al cliente como JSON.

3.2.5.3 Mappers

El mapper convierte entre: Objetos de dominio y DTOs (Request y Response). Esto es necesario porque los objetos de dominio no deben exponerse directamente por la API, asegurando que el formato de los datos sea adecuado para la comunicación externa (API).

3.2.5.4. Requests

Estos son DTOs (Data Transfer Object) que representan los datos que el cliente envía al servidor.

3.2.5.5. Responses

Igual que las peticiones o requests que son parte del modelo, son DTOs, pero en este caso representan la respuesta que el servidor envía al cliente después de procesar la solicitud.

3.2.6. Common

La carpeta common contiene componentes reutilizables y transversales a toda la aplicación, como:

- **annotations:** decoradores personalizados (por ejemplo, para marcar servicios o comportamientos especiales).
- **exceptions:** clases para manejar errores comunes personalizados.
- **setup:** configuración general o inicialización de componentes (como bases de datos o frameworks).

Esta carpeta ayuda a no duplicar código y centraliza comportamientos comunes.

3.2.7. Domain

La carpeta domain define el modelo de dominio de la aplicación, es decir, las entidades centrales del negocio con sus atributos y relaciones. Estas clases conforman la estructura central de datos del sistema: representan los conceptos clave que se procesan y transfieren entre las distintas capas de la arquitectura, como la lógica de negocio, la persistencia y la interfaz.

El objetivo de esta carpeta es mantener una representación clara y dividida de los datos esenciales del sistema, sirviendo como base para implementar las funcionalidades de la aplicación. En esta carpeta encontraremos las siguientes clases:

- **Gender:** Define los valores posibles para el género de una persona: MALE, FEMALE y OTHER.
- **Person:** Representa a una persona con sus atributos: identificación, nombre, apellido, género y edad. Además, una persona contiene listas de teléfonos y estudios asociados. Aca tambien se incluye una validación básica de edad.
- **Phone:** Representa una línea telefónica con su número, la compañía que lo provee y la persona a la que pertenece.
- **Profession:** Define una profesión con su identificador, nombre, descripción y los estudios que la vinculan con personas.
- **Study:** Representa la relación entre una persona y una profesión. Y incluye cosas como la universidad donde estudió y la fecha de graduación.

3.2.8. Maria-output-adapter

El paquete maria-output-adapter hace parte de los adaptadores de salida en la arquitectura hexagonal de nuestra aplicación y su función principal es manejar la comunicación entre la lógica de negocio y la base de datos relacional MariaDB. Desde este paquete, se implementan los puertos de salida definidos en la capa de aplicación, permitiendo realizar operaciones como guardar, consultar, actualizar o eliminar información directamente en la base de datos.

Está organizado en cuatro subcomponentes principales: Adapters, Entities, Mappers y Repositories, cada uno con una responsabilidad clara dentro del proceso de persistencia.

3.2.8.1. Adapters

Los adaptadores implementan los puertos de salida definidos para cada entidad (Person, Phone, Profession y Study) y su responsabilidad es traducir las operaciones solicitadas desde los casos de uso a acciones concretas sobre la base de datos, utilizando los repositorios Spring y los mappers. Además, gestionan la conversión de entidades del dominio a entidades de base de datos y viceversa, delegando esta tarea a los mappers correspondientes.

3.2.8.2. Entities

Las entidades en esta carpeta representan las tablas de MariaDB. Están anotadas con @Entity y definen el mapeo entre los objetos Java y las estructuras relacionales. Cada clase (PersonEntity, PhoneEntity, ProfessionEntity, StudyEntity) contiene los atributos necesarios y las relaciones (como @OneToMany, @ManyToOne, etc.) para reflejar la estructura de la base de datos.

3.2.8.3. Mappers

Los mappers se encargan de transformar datos entre el modelo de dominio y las entidades de base de datos. Su uso permite mantener desacopladas ambas capas, facilitando la lectura, escritura y

mantenimiento del código. Por ejemplo, un PersonMapper puede convertir un Person (del dominio) en un PersonEntity (de base de datos), y viceversa.

3.2.8.4 Repositories

En los repositorios se encuentran las interfaces que extienden de JpaRepository o similares, permitiendo realizar operaciones CRUD directamente sobre la base de datos MariaDB. Cada repositorio está asociado a una entidad y proporciona métodos personalizados o derivados del nombre para realizar búsquedas y operaciones específicas (por ejemplo, findById, findByOwner, etc.).

3.2.9. Mongo-output-adapter

El paquete mongo-output-adapter hace parte de los adaptadores de salida en la arquitectura hexagonal de la aplicación. Su función principal es permitir la interacción entre la lógica de negocio y la base de datos MongoDB, que en este caso funciona como base de datos NoSQL para almacenar documentos relacionados con personas, teléfonos, profesiones y estudios.

Este paquete está compuesto por los siguientes subcomponentes: Adapters, Documents, Mappers y Repositories, los cuales colaboran para implementar los puertos de salida (OutputPorts) definidos en la capa de aplicación.

3.2.9.1 Adapters

Los adaptadores implementan las interfaces de salida para cada entidad del dominio (Person, Phone, Profession, Study). Son responsables de recibir las solicitudes desde los casos de uso, preparar los datos usando los mappers y delegar la operación en el repositorio Mongo correspondiente.
Desde aquí se traduce la lógica del dominio a operaciones sobre documentos, manteniendo la arquitectura desacoplada del motor de base de datos.

3.2.9.2 Documents

Esta carpeta contiene las clases que representan los documentos que se almacenan en MongoDB. A diferencia de las entidades en bases de datos relacionales, estos documentos están diseñados para reflejar la estructura flexible y jerárquica de Mongo. Cada clase se anota con @Document y contiene los atributos que serán guardados como campos en los documentos BSON.

3.2.9.3 Mappers

Los mappers de Mongo se encargan de convertir objetos entre el modelo de dominio y los documentos Mongo. Esta transformación es necesaria para mantener separadas las estructuras internas del negocio y las estructuras usadas para persistencia NoSQL. Cada mapper convierte en ambas direcciones (Hacia el dominio y hacia el documento), asegurando la compatibilidad entre las capas sin acoplamiento directo.

3.2.9.4 Repositories

Aquí se definen los repositorios Mongo para cada documento, extendiendo de interfaces como MongoRepository o CrudRepository. Estos repositorios permiten ejecutar operaciones CRUD directamente sobre las colecciones en MongoDB. Además, pueden incluir consultas personalizadas utilizando convenciones de nombres o anotaciones propias de Spring Data MongoDB.

4. Conclusiones

- **La arquitectura hexagonal facilita la escalabilidad y el mantenimiento del sistema.** Al separar la lógica del negocio del código específico de infraestructura, esta arquitectura permite desarrollar, probar y evolucionar cada componente de forma aislada. Gracias a esta división clara, es posible incorporar nuevas interfaces (como APIs REST, CLIs o incluso interfaces gráficas) sin necesidad de modificar el núcleo del sistema, lo que hace que sea especialmente útil en proyectos que requieren crecimiento progresivo o múltiples canales de entrada/salida.
- **La integración de MariaDB y MongoDB permite aprovechar lo mejor de cada tipo de base de datos según su propósito.** El uso de MariaDB para almacenar entidades estructuradas y MongoDB para gestionar información de historial demuestra cómo una arquitectura flexible puede adaptarse a distintos tipos de persistencia. Esta integración permite optimizar el rendimiento y la organización de los datos, utilizando una base relacional para consistencia estructural y una NoSQL para almacenar información más dinámica o de lectura intensiva.
- **El uso de puertos y adaptadores en la arquitectura hexagonal simplifica la reutilización de lógica y el desacoplamiento de tecnologías.** Los puertos actúan como contratos independientes del modo en que se accede o se almacena la información, mientras que los adaptadores concretan estos contratos para cada tecnología o interfaz. Esta estrategia hace que la lógica del sistema no dependa ni de la base de datos, ni del tipo de entrada del usuario, facilitando el reemplazo o adición de tecnologías sin necesidad de reescribir la lógica de negocio.
- **La contenerización con Docker permite replicar entornos de ejecución de forma confiable y portátil**. Al empaquetar el sistema, sus dependencias y sus servicios asociados (como las bases de datos) en contenedores, Docker asegura que el sistema se ejecute de la misma forma en cualquier entorno, reduciendo errores por diferencias de configuración. Esto es especialmente útil en procesos de despliegue, pruebas en distintos entornos y colaboración entre desarrolladores.
- **La reutilización de los casos de uso entre CLI y API REST demuestra la eficiencia del diseño modular**. El hecho de que ambos canales de comunicación utilicen la misma lógica central confirma que el sistema está correctamente desacoplado y que las responsabilidades están bien definidas. Esto no solo reduce la duplicación de código, sino que también facilita la validación funcional y la evolución futura del sistema con nuevas interfaces.

5. Lecciones aprendidas

- A lo largo del desarrollo del laboratorio, pudimos comprobar cómo la arquitectura hexagonal realmente facilita la organización del proyecto. Separar la lógica de negocio de los detalles técnicos nos dio más libertad para trabajar por partes, sin que un cambio afectara todo lo demás.
- Trabajar con dos tipos de bases de datos (MariaDB y MongoDB) nos permitió ver en la práctica cómo una buena estructura nos permite integrar tecnologías distintas sin complicar la lógica central. Fue interesante notar cómo el mismo sistema puede adaptarse a motores relacionales o NoSQL sin mayores modificaciones.
- El uso de patrones como Service, Repository y los distintos mappers nos ayudó mucho a mantener el código claro y entendible. Aunque al principio puede parecer más trabajo, a medida

que el proyecto crecía se hizo evidente que seguir estas buenas prácticas nos ahorró muchos dolores de cabeza.

- Desarrollar tanto una interfaz de línea de comandos (CLI) como una API REST fue una experiencia muy útil. Pudimos reutilizar completamente los mismos casos de uso, lo cual confirmó que tener una arquitectura limpia y modular realmente funciona y se nota.
- Más allá del código, este laboratorio nos permitió familiarizarnos con herramientas clave como Docker, Swagger y Spring Boot. No solo aprendimos a usarlas, sino también a entender su rol dentro de un proyecto bien estructurado, lo que sin duda será útil en futuros desarrollos.