

Pontificia Universidad Javeriana

Laboratorio II - Personapp

Arquitectura de Software



Pontificia Universidad
JAVERIANA
Colombia

Integrantes:

Javier Alejandro Moyano Cipamocha

Juan Francisco Ramírez Escobar

Esteban Salazar Arbeláez

Presentado a:

Andrés Armando Sánchez

4/11/2024

Bogotá, Colombia

Tabla de Contenido

1. Marco Conceptual	1
1.1 Estilo Arquitectónico Hexagonal	1
1.2 Patrón Modular.....	2
1.3 SpringBoot	2
1.3.1 Ventajas de Utilizar Spring Boot frente a Spring Framework:	3
1.4 JDK11.....	3
1.5 Mongo DB.....	4
1.6 María DB.....	5
1.7 REST	5
1.8 CLI	6
1.9 Swagger.....	6
1.10 Docker	8
1.10.1 Docker Compose.....	8
2. Diseño	9
2.1 Diagrama de Alto Nivel	9
2.2 Vistas Estándar C4	11
2.2.1 Vista de Contexto.....	11
2.2.2 Vista de Contenedores	12
2.2.3 Vista de Componentes	13
2.2.4 Vista de Datos	15
3. Procedimiento	17
3.1 Fork del repositorio	17
3.2 Base de datos.....	18

3.3	Implementación del Proyecto en Spring Boot.....	21
3.3.1	Documentación Desarrollo de Puertos y Adaptadores	21
4.2	Pruebas de Ejecución	84
4.3	TAG.....	84
5.	Conclusiones y Lecciones Aprendidas	85
6.	Referencias.....	87

1. Marco Conceptual

Este laboratorio tiene como objetivo desarrollar un sistema monolítico web empleando el patrón modular para dividir el sistema en módulos, siguiendo el estilo arquitectónico Hexagonal. Durante el laboratorio, se trabajó con tecnologías como SpringBoot, JDK 11 y las bases de datos MongoDB y MariaDB, que permitieron estructurar el sistema utilizando puertos y adaptadores.

Así mismo, se implementaron dos adaptadores de entrada para acceder a los datos de las entidades en el módulo de dominio (el módulo aislado del sistema). Finalmente, se utilizó Swagger 3 para documentar y probar el adaptador de entrada de la API REST.

1.1 Estilo Arquitectónico Hexagonal

La arquitectura hexagonal, también conocida como arquitectura de puertos y adaptadores, es un enfoque de diseño que busca desacoplar la lógica central de una aplicación de sus mecanismos de interacción externos, como bases de datos, interfaces de usuario o servicios externos. Este modelo divide la aplicación en una parte interna (que contiene la lógica de negocio y el modelo de dominio) y una parte externa (encargada de manejar las interacciones con el mundo exterior). Los puertos actúan como puntos de entrada y salida de la lógica de negocio, mientras que los adaptadores implementan estos puertos para conectarlos con sistemas externos específicos.

El principal beneficio de la arquitectura hexagonal es su flexibilidad y facilidad para realizar pruebas. Gracias a su estructura modular, permite reemplazar componentes externos sin afectar la lógica interna, lo que facilita la realización de pruebas unitarias y la integración de nuevos servicios o tecnologías. Además, al encapsular las interacciones con sistemas externos mediante adaptadores, se garantiza que los cambios en la tecnología externa no afecten el núcleo de la aplicación. Esta arquitectura promueve una separación clara de responsabilidades, lo que resulta en un sistema más mantenable y escalable a largo plazo.

1.2 Patrón Modular

En ingeniería de software, el patrón módulo es un patrón de diseño utilizado para implementar el concepto de módulos de software, definido por la programación modular, en un lenguaje de programación con soporte directo incompleto para el concepto.

Este patrón se puede implementar de varias maneras dependiendo del lenguaje de programación anfitrión, como el patrón de diseño singleton, miembros estáticos orientados a objetos en una clase y funciones globales de procedimiento. En Python, el patrón está integrado en el lenguaje, y cada archivo .py es automáticamente un módulo. Lo mismo ocurre en Ada, donde el paquete puede considerarse un módulo (similar a una clase estática).

Aplicado sobre el contexto del proyecto (Marco de trabajo en Java), el patrón Módulo puede considerarse un patrón de creación y un patrón estructural. Gestiona la creación y organización de otros elementos, y los agrupa como lo hace el patrón estructural.

Un objeto que aplica este patrón puede proporcionar el equivalente de un espacio de nombres, proporcionando el proceso de inicialización y finalización de una clase estática o de una clase con miembros estáticos con una sintaxis y una semántica más limpias y concisas.

1.3 SpringBoot

Java Spring Framework (Spring Framework) es una popular estructura empresarial de código abierto para crear aplicaciones independientes de nivel de producción que se ejecutan en la máquina virtual Java (JVM).

Java Spring Boot (Spring Boot) es una herramienta que hace que el desarrollo de aplicaciones web y microservicios con Spring Framework sea más rápido y fácil a través de tres funcionalidades principales:

1. Configuración automática: La configuración automática significa que las aplicaciones se inicializan con dependencias preestablecidas que no tienen que configurarse manualmente. Como Java Spring Boot viene con funciones de configuración automática integradas, configura automáticamente tanto el Spring Framework subyacente como los paquetes de

terceros según su configuración (y según las mejores prácticas, lo que ayuda a evitar errores).

2. Un enfoque obstinado de la configuración: Spring Boot utiliza un enfoque obstinado para agregar y configurar dependencias de inicialización, según las necesidades de su proyecto. Siguiendo su propio criterio, Spring Boot elige qué paquetes instalar y qué valores predeterminados usar, en lugar de pedirle que tome todas esas decisiones usted mismo y configure todo manualmente.
3. La capacidad de crear aplicaciones independientes

Estas características funcionan juntas para brindarle una herramienta que le permite configurar una aplicación basada en Spring con una configuración y preparación mínimas.

1.3.1 Ventajas de Utilizar Spring Boot frente a Spring Framework:

Las mayores ventajas de usar Spring Boot frente a Spring Framework son la facilidad de uso y el desarrollo más rápido. En teoría, esto se debe a la mayor flexibilidad que se obtiene al trabajar directamente con Spring Framework.

Pero, en la práctica, a menos que necesite o desee implementar una configuración única, vale la pena hacer esta concesión a cambio de poder utilizar Spring Boot. Aún puede usar el muy popular sistema de anotación de Spring Framework que le permite injectar fácilmente dependencias adicionales (no cubiertas por Spring Starters) en su aplicación. Y todavía tendrá acceso a todas las características de Spring Framework, incluido el manejo sencillo de eventos, la validación, el enlace de datos, la conversión de tipos y las funciones de prueba y seguridad integradas. En pocas palabras, si el alcance de su proyecto está cubierto incluso por un solo Spring Starter, Spring Boot puede agilizar significativamente el desarrollo.

1.4 JDK11

JDK 11, lanzado por Oracle en septiembre de 2018, es una versión de soporte a largo plazo (LTS) de Java que incorpora importantes mejoras y nuevas características en el lenguaje y la plataforma. Esta versión introdujo funcionalidades como el uso de var en expresiones lambda locales, mejoras

en la API HTTP para facilitar la comunicación con servicios web, y la capacidad de ejecutar archivos Java sin compilación previa, simplificando el proceso de desarrollo y pruebas rápidas. Además, JDK 11 incluye varias mejoras de rendimiento y seguridad, lo que lo convierte en una opción robusta y eficiente para el desarrollo de aplicaciones modernas.

Otra novedad destacada en JDK 11 fue la eliminación de módulos y características que ya no se consideraban esenciales, como el soporte para JavaFX y las herramientas javaws y appletviewer. Esto representa un esfuerzo por mantener la plataforma ligera y enfocada en lo que es crucial para los desarrolladores. Con estas mejoras y el soporte extendido de Oracle, JDK 11 sigue siendo una opción popular y confiable en el ecosistema de desarrollo de Java, especialmente para empresas que buscan estabilidad y soporte a largo plazo.

1.5 Mongo DB



MongoDB es una base de datos NoSQL orientada a documentos, diseñada para gestionar grandes volúmenes de datos no estructurados o semiestructurados de manera flexible y escalable. A diferencia de las bases de datos relacionales tradicionales, que almacenan datos en tablas y filas, MongoDB utiliza un modelo de documentos basado en JSON (BSON, en su formato interno), permitiendo una mayor flexibilidad en la estructura de los datos, lo que es ideal para aplicaciones modernas con requisitos cambiantes y datos complejos.

Este enfoque orientado a documentos permite a MongoDB escalar horizontalmente, facilitando el manejo de grandes cantidades de datos distribuidos en múltiples servidores o clústeres, una capacidad especialmente valiosa para aplicaciones web, móviles y big data. Además, MongoDB ofrece características como consultas avanzadas, índices secundarios, y soporte para replicación y alta disponibilidad, lo que la convierte en una solución potente y versátil para desarrolladores y empresas que necesitan agilidad y rendimiento en la gestión de sus datos.

1.6 María DB



MariaDB es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, desarrollado como una alternativa a MySQL con el objetivo de ofrecer un alto rendimiento, seguridad y escalabilidad. MariaDB retiene gran parte de la compatibilidad con MySQL, facilitando la migración y el uso de aplicaciones que anteriormente trabajaban con este sistema, pero incorpora mejoras significativas en términos de optimización y nuevas funcionalidades.

Este sistema está diseñado para soportar aplicaciones empresariales con grandes volúmenes de datos y permite una arquitectura escalable, ya sea en entornos locales o en la nube. Entre sus características avanzadas se incluyen el soporte para consultas distribuidas, replicación, alta disponibilidad y potentes motores de almacenamiento. Gracias a su comunidad activa y enfoque en la innovación continua, MariaDB es ampliamente adoptada en sectores que requieren fiabilidad y rendimiento, desde aplicaciones de análisis de datos hasta sistemas críticos de negocios.

1.7 REST

Una API REST (Representational State Transfer) es una interfaz de programación que permite la comunicación entre sistemas utilizando el protocolo HTTP, diseñada bajo principios que facilitan la creación de servicios web ligeros, escalables y eficientes. REST define un conjunto de reglas para estructurar las interacciones entre el cliente y el servidor, donde los recursos (datos o funcionalidades) se representan a través de URL y se accede a ellos mediante operaciones estándar de HTTP como GET, POST, PUT y DELETE.

Las APIs REST son independientes del lenguaje y permiten que diferentes aplicaciones intercambien datos en formatos simples, como JSON o XML, optimizando así el rendimiento y la interoperabilidad entre sistemas heterogéneos. Gracias a su arquitectura orientada a recursos y su

flexibilidad, las APIs REST se han convertido en una tecnología fundamental para el desarrollo de aplicaciones modernas, permitiendo integrar servicios en la nube, aplicaciones móviles y sistemas distribuidos de manera robusta y escalable.

1.8 CLI

La interfaz de línea de comandos (CLI) es una herramienta que permite a los usuarios interactuar directamente con un sistema operativo o software a través de comandos de texto, en lugar de utilizar una interfaz gráfica. A través de la CLI, los usuarios pueden ejecutar comandos específicos que controlan y manipulan las funciones y servicios del sistema. Esta interfaz es utilizada comúnmente por administradores de sistemas y desarrolladores para realizar tareas como la gestión de archivos, la administración de procesos y el control de configuraciones.

La CLI ofrece una alta eficiencia y precisión, permitiendo a los usuarios realizar tareas complejas con rapidez y consumir menos recursos en comparación con una interfaz gráfica. Además, la CLI es altamente flexible, ofreciendo una amplia gama de comandos y opciones, y se puede automatizar mediante scripts, lo cual es ideal para operaciones repetitivas o avanzadas.

1.9 Swagger



Swagger es un conjunto de herramientas de código abierto diseñadas en torno a la Especificación OpenAPI, que facilita el diseño, creación, documentación y consumo de APIs REST. Una de sus principales herramientas es **Swagger Editor**, un editor basado en el navegador que permite escribir y editar definiciones OpenAPI de manera interactiva. Proporciona una interfaz que permite a los desarrolladores visualizar los cambios en tiempo real, lo que optimiza el proceso de diseño de las APIs al identificar errores o inconsistencias rápidamente.

Por otro lado, **Swagger UI** genera documentación interactiva a partir de las definiciones OpenAPI, permitiendo a los usuarios explorar y probar las APIs directamente desde la interfaz. Esto no solo

facilita la comprensión de los endpoints, sino que también permite una integración más sencilla para los consumidores de la API al no requerir herramientas adicionales para probar su funcionalidad.

Swagger Codegen es otra herramienta destacada que automatiza la generación de servidores y bibliotecas cliente basados en las definiciones OpenAPI. Esto acelera el proceso de desarrollo al proporcionar automáticamente el código necesario en distintos lenguajes de programación, facilitando la implementación y el consumo de las APIs en diversos entornos tecnológicos.

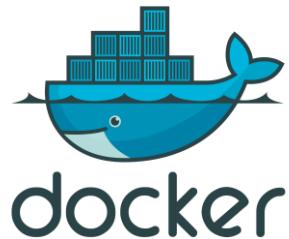
Además, **Swagger Editor Next (beta)** es una versión mejorada del Swagger Editor, que, además de permitir trabajar con definiciones OpenAPI, soporta también AsyncAPI, expandiendo su funcionalidad a APIs basadas en eventos. Esta herramienta mejora la experiencia de edición al proporcionar nuevas características y una interfaz más intuitiva.

Para aquellos que trabajan con el ecosistema Java, **Swagger Core** proporciona un conjunto de bibliotecas orientadas a la creación, consumo y gestión de definiciones OpenAPI. Con estas bibliotecas, los desarrolladores pueden integrar y gestionar APIs dentro de sus aplicaciones Java de manera más eficiente y ágil.

Por su parte, **Swagger Parser** es una biblioteca independiente diseñada específicamente para analizar y procesar definiciones OpenAPI. Esta herramienta permite la validación y manipulación de especificaciones, lo que facilita a los desarrolladores trabajar con definiciones estructuradas y coherentes en sus proyectos.

Finalmente, **Swagger APIDom** ofrece una estructura unificadora para describir APIs utilizando varios lenguajes de descripción y formatos de serialización. Esto es especialmente útil en proyectos que requieren flexibilidad en la forma en que se describen las APIs, brindando un enfoque más estándar y cohesivo para la documentación y el desarrollo de APIs.

1.10 Docker



Docker que es una plataforma de contenedorización que permite empaquetar una aplicación y todas sus dependencias en un contenedor estandarizado, lo que facilita su despliegue y ejecución en cualquier entorno. Al usar Docker en este proyecto, se aseguró que el sistema fuera portable y replicable, eliminando los problemas de configuración y compatibilidad entre entornos de desarrollo, prueba y producción. Los contenedores Docker encapsulan todos los componentes del sistema, desde la aplicación web implementada en ASP.NET 8 hasta la base de datos con MS SQL Server, lo que permite gestionar cada parte del sistema de forma independiente.

1.10.1 Docker Compose



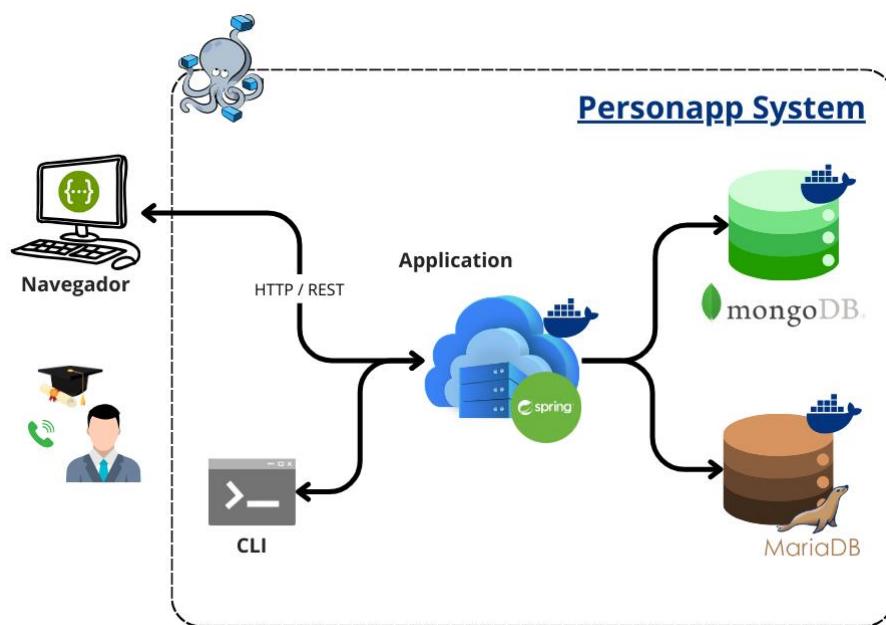
Docker Compose es una herramienta para definir y ejecutar aplicaciones multicontenedor. Es la clave para lograr una experiencia de desarrollo e implementación optimizada y eficiente.

Compose simplifica el control de toda la pila de aplicaciones, lo que facilita la administración de servicios, redes y volúmenes en un único archivo de configuración YAML comprensible. Luego, con un solo comando, crea e inicia todos los servicios desde su archivo de configuración.

2. Diseño

De acuerdo con el paso a seguir indicado para el laboratorio; el sistema se diseñó e implementó con base al marco de trabajo del . Para dicho fin, se plantea el diseño del sistema a través del estándar C4.

2.1 Diagrama de Alto Nivel



Este diagrama de alto nivel (High-Level Design - HLD) representa la arquitectura general de la aplicación **Personapp System** y cómo sus componentes interactúan entre sí y con el entorno externo. Aquí está una explicación de cada elemento del diagramma:

1. Usuarios y Canales de Interacción

- Navegador:** Representa a los usuarios que interactúan con la aplicación mediante una interfaz web. Estos usuarios se conectan a la aplicación a través de **HTTP/REST**.
- CLI (Command Line Interface):** Representa otro tipo de usuario o proceso que interactúa con la aplicación desde la línea de comandos. Esta interacción también ocurre dentro del mismo entorno de aplicación, aunque no utiliza HTTP o REST.

2. Application

Es el núcleo de la lógica de negocio y se implementa en **SpringBoot**. Actúa como el centro de procesamiento, manejando las solicitudes provenientes del navegador (a través de la API REST) y del CLI. La aplicación transforma estas solicitudes en operaciones de negocio, consulta las bases de datos y retorna los resultados.

3. Protocolos de Comunicación

- **HTTP/REST:** Es el protocolo que facilita la comunicación entre el navegador y la aplicación. Los usuarios pueden enviar solicitudes RESTful que la aplicación procesa y responde, generalmente en formato JSON.

4. Bases de Datos

- **MongoDB:** Representa una base de datos NoSQL, ideal para almacenar datos semi-estructurados o no estructurados en formato de documentos JSON. La aplicación interactúa con MongoDB para almacenar o recuperar datos no relacionales.
- **MariaDB:** Representa una base de datos relacional SQL utilizada para almacenar datos estructurados. La aplicación utiliza esta base de datos para operaciones que requieren estructura y relaciones entre datos.

5. Interacciones entre Componentes

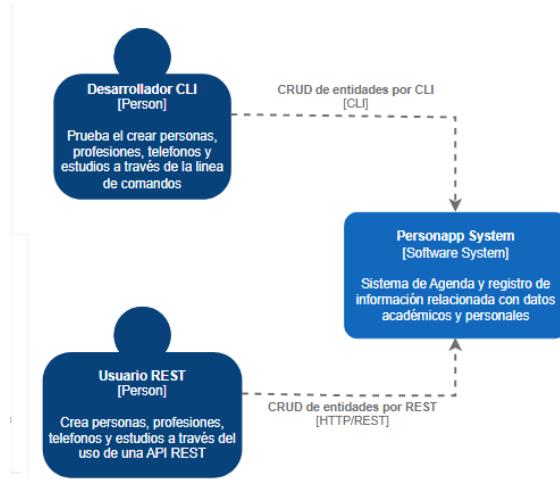
La aplicación se conecta con **MongoDB** y **MariaDB** para realizar operaciones de lectura y escritura de datos, cada una utilizando adaptadores o controladores específicos para gestionar la comunicación con las bases de datos correspondientes.

La aplicación puede procesar las solicitudes de dos fuentes distintas:

- **Navegador:** Envía solicitudes HTTP/REST, típicamente desde una interfaz de usuario web.
- **CLI:** Envía comandos de línea de comandos que la aplicación procesa directamente.

2.2 Vistas Estándar C4

2.2.1 Vista de Contexto

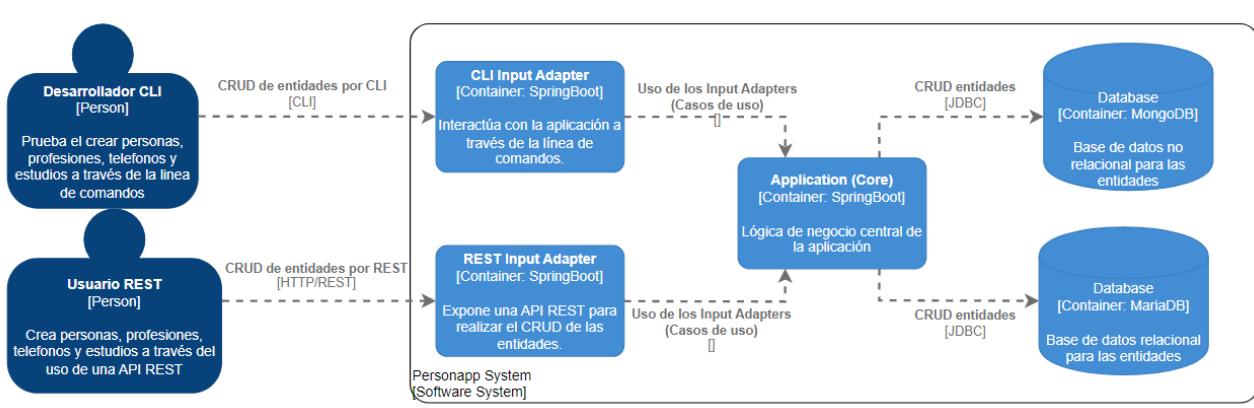


El diagrama de contexto muestra cómo el **Personapp System** permite gestionar información académica y personal mediante operaciones CRUD (Crear, Leer, Actualizar, Eliminar). Dos tipos de usuarios interactúan con el sistema:

- **Desarrollador CLI**: Este usuario accede al sistema a través de la línea de comandos (CLI) para crear y administrar entidades como personas, profesiones, teléfonos y estudios. El CLI permite realizar pruebas sobre la lógica antes de realizar la implementación sobre el controlador REST.
- **Usuario REST**: Accede mediante la API REST para realizar las mismas operaciones CRUD, utilizando solicitudes HTTP desde un navegador o cliente de API.

Ambos usuarios interactúan con el sistema de manera distinta, pero comparten el objetivo de gestionar datos clave en Personapp.

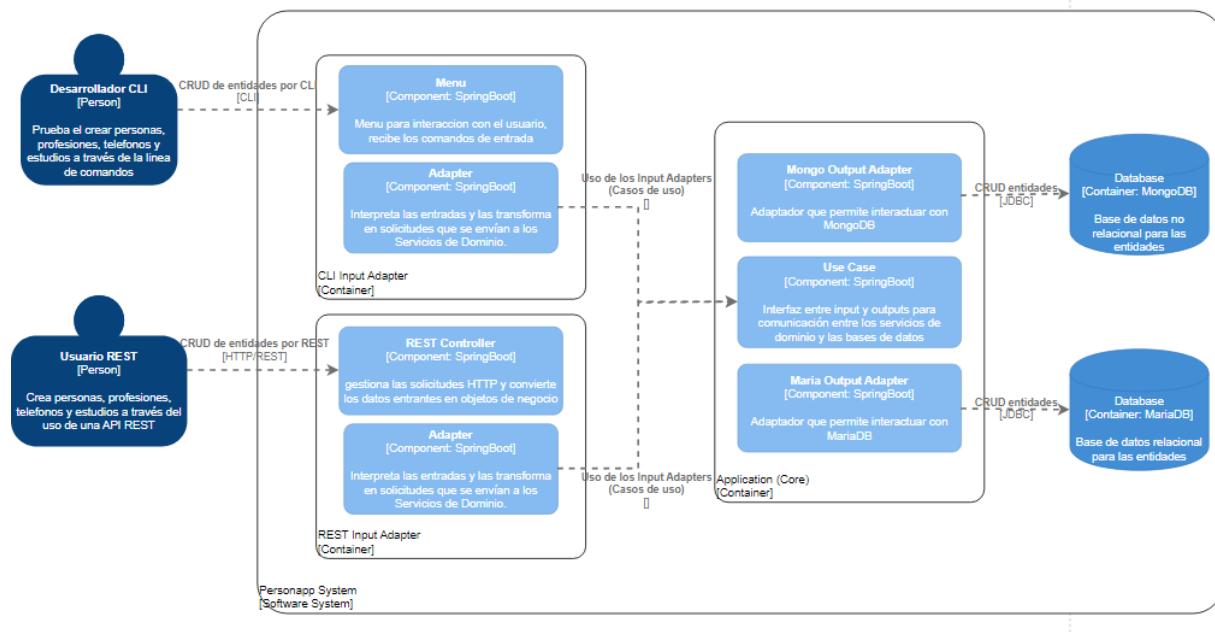
2.2.2 Vista de Contenedores



En el nivel de contenedores, el diagrama muestra cómo el **Personapp System** está compuesto por varios contenedores que interactúan para gestionar datos académicos y personales. El sistema tiene los siguientes contenedores:

- **CLI Input Adapter**: Este contenedor, desarrollado con SpringBoot, permite al **Desarrollador CLI** interactuar con el sistema mediante la línea de comandos para realizar operaciones CRUD sobre las entidades.
- **REST Input Adapter**: También basado en SpringBoot, expone una API REST que permite al **Usuario REST** realizar operaciones CRUD sobre las entidades a través de solicitudes HTTP.
- **Application (Core)**: Es el núcleo de la lógica de negocio de la aplicación. Este contenedor recibe solicitudes de ambos adaptadores (CLI y REST), procesándolas según las reglas de negocio y enviando las operaciones a las bases de datos correspondientes.
- **Database [MongoDB]**: Base de datos NoSQL que almacena datos en formato de documentos, adecuada para información no estructurada. Application (Core) interactúa con MongoDB mediante JDBC para realizar operaciones CRUD.
- **Database [MariaDB]**: Base de datos relacional que almacena información estructurada de las entidades. Application (Core) usa JDBC para realizar operaciones CRUD en esta base de datos.

2.2.3 Vista de Componentes



CLI Input Adapter

- **Menu:** Componente que permite la interacción con el usuario de línea de comandos. Recibe los comandos de entrada y muestra las respuestas.
- **Adapter:** Componente que interpreta las entradas del usuario desde el CLI, las transforma en solicitudes y las envía a los Servicios de Dominio en Application (Core).

REST Input Adapter

- **REST Controller:** Componente que gestiona las solicitudes HTTP provenientes del Usuario REST. Convierte los datos entrantes en objetos de negocio y delega las solicitudes al siguiente componente.
- **Adapter:** Interpreta las solicitudes recibidas por el REST Controller y las transforma en solicitudes que serán enviadas a los Servicios de Dominio en Application (Core).

Application (Core)

- **Mongo Output Adapter:** Componente que permite interactuar con la base de datos MongoDB, utilizando JDBC para realizar operaciones CRUD en una base de datos NoSQL.

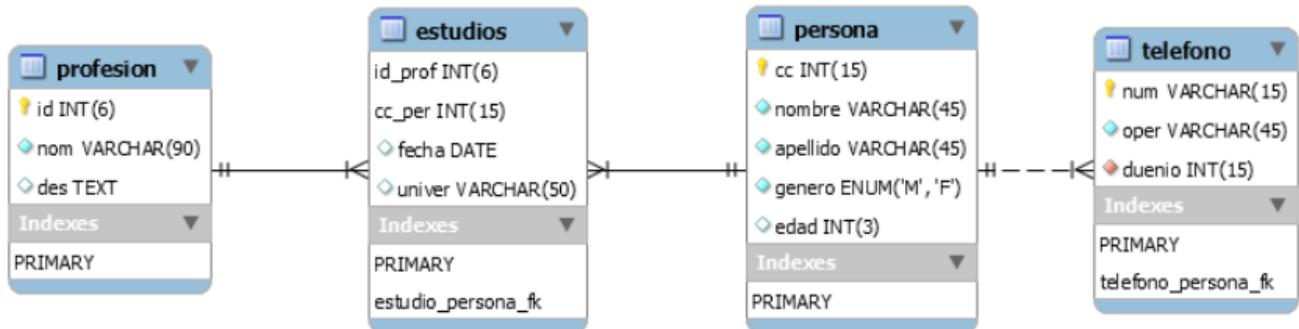
- **Use Case:** Componente que actúa como intermediario entre la lógica de negocio y los adaptadores de bases de datos, permitiendo la comunicación entre los servicios de dominio y las bases de datos.
- **Maria Output Adapter:** Componente encargado de interactuar con la base de datos relacional MariaDB mediante JDBC, realizando operaciones CRUD en datos estructurados.

Bases de Datos

- **Database [MongoDB]:** Base de datos NoSQL que almacena datos en formato de documentos.
- **Database [MariaDB]:** Base de datos relacional que almacena datos estructurados.

Este nivel de componentes detalla cómo los adaptadores (CLI y REST) interactúan con el núcleo de la aplicación y cómo los adaptadores de salida (Mongo Output Adapter y Maria Output Adapter) manejan la persistencia en sus respectivas bases de datos.

2.2.4 Vista de Datos



Entidad Profesión

La entidad **Profesión** está representada en la tabla `profesion`, la cual 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.

Entidad Estudios

La entidad **Estudios** se gestiona a través de la tabla `estudios`, que almacena 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.

Entidad Persona

La entidad **Persona** está representada por la tabla `persona`, que almacena 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.

Entidad Teléfono

La entidad **Teléfono** se maneja en la tabla `telefono`, que 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.

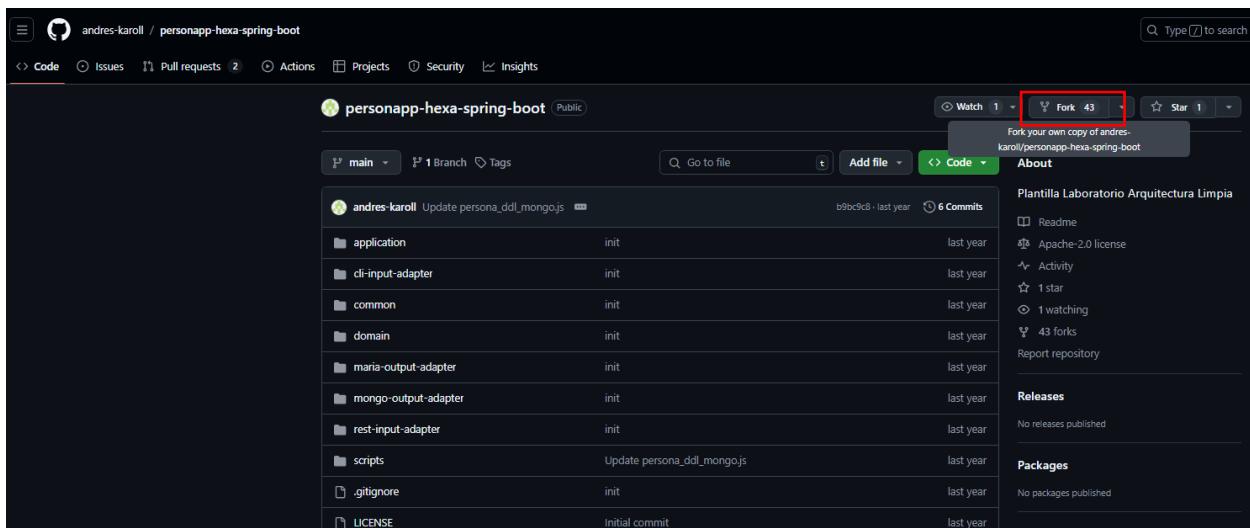
Manejo de datos en MongoDB

En MongoDB, las entidades del modelo relacional se gestionan como documentos en lugar de tablas, aprovechando la flexibilidad del almacenamiento de datos en formato JSON. La entidad **Profesión** puede ser representada como un documento independiente, donde cada profesión contiene su identificador único, nombre y descripción. Para la entidad **Estudios**, en lugar de manejar claves foráneas, los datos pueden embebirse en el documento de la persona correspondiente o referenciarse mediante identificadores. **Persona** y de la **Profesión** puede almacenarse como un documento principal que incluye los datos personales y, opcionalmente, listas de estudios y teléfonos como subdocumentos, evitando la necesidad de relaciones explícitas.

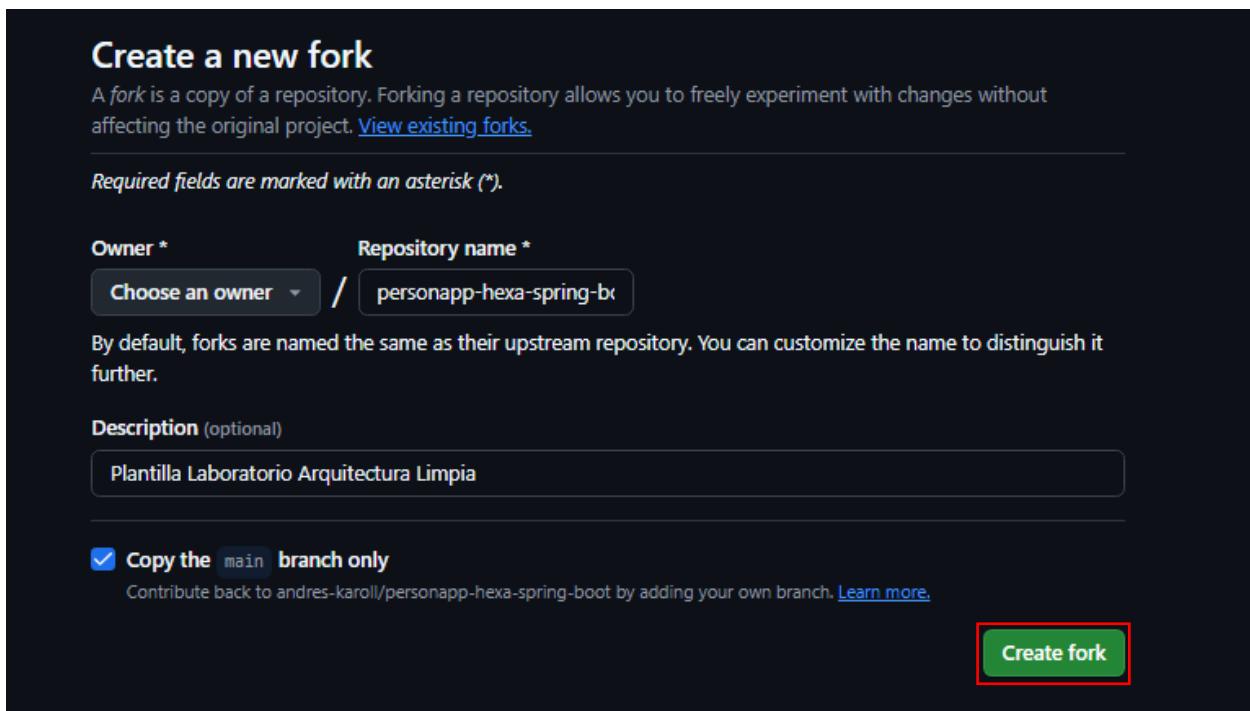
3. Procedimiento

3.1 Fork del repositorio

Como primer paso se debe hacer fork del repositorio en la url: <https://github.com/andres-karoll/personapp-hexa-spring-boot>



The screenshot shows the GitHub repository page for 'personapp-hexa-spring-boot'. At the top right, there is a 'Fork' button with a dropdown menu showing '43' forks. A red box highlights this button. Below the header, there is a list of files and their commit history. On the right side, there is a sidebar with repository statistics: 'Plantilla Laboratorio Arquitectura Limpia', 'Readme', 'Apache-2.0 license', 'Activity', '1 star', '1 watching', '43 forks', 'Report repository', 'Releases' (No releases published), and 'Packages' (No packages published).



The screenshot shows the 'Create a new fork' form. It includes fields for 'Owner *' (a dropdown menu with 'Choose an owner') and 'Repository name *' (a text input field containing 'personapp-hexa-spring-boot'). Below these, a note states: 'By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.' There is also a 'Description (optional)' field with the placeholder 'Plantilla Laboratorio Arquitectura Limpia'. At the bottom, there is a checked checkbox for 'Copy the main branch only' and a note: 'Contribute back to andres-karoll/personapp-hexa-spring-boot by adding your own branch. [Learn more](#)'. A large green 'Create fork' button is at the bottom right, with a red box highlighting it.

Por último, se clona el repositorio de manera local para empezar el desarrollo.

3.2 Base de datos

Para la configuración de las bases de datos se debe crear un Docker compose con los siguientes servicios:

```
mongo:
  image: mongo
  restart: always
  environment:
    MONGO_INITDB_ROOT_USERNAME: root
    MONGO_INITDB_ROOT_PASSWORD: example
    MONGO_INITDB_DATABASE: "persona_db"
  volumes:
    - mongo_data:/data/db
  ports:
    - 27017:27017
```

```
mariadb:
  image: mariadb:10.3.10
  container_name: mariadb
  ports:
    - "3307:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
    MYSQL_DATABASE: persona_db
    MYSQL_USER: user
    MYSQL_PASSWORD: password
  volumes:
    - maria_data:/var/lib/mysql
    - ./scripts/persona_ddl_maria.sql:/docker-entrypoint-initdb.d/01_persona_ddl_maria.sql.sql
    - ./scripts/persona_dml_maria.sql:/docker-entrypoint-initdb.d/02_persona_dml_maria.sql
  restart: always
```

Es necesario aclarar que sobre la raíz del proyecto debe existir una carpeta llamada scripts con los siguientes scripts para inicializar las bases de datos:

persona_ddl_maria.sql

```
FLUSH PRIVILEGES;
-- 
DROP USER IF EXISTS 'persona_db'@'%';
DROP SCHEMA IF EXISTS `persona_db`;
-- 
CREATE USER IF NOT EXISTS 'persona_db'@'%' IDENTIFIED BY 'persona_db';
CREATE SCHEMA IF NOT EXISTS `persona_db`;
-- 
GRANT EXECUTE, TRIGGER, INSERT, UPDATE, DELETE, SELECT ON `persona_db`.* TO 'persona_db'@'%';
FLUSH PRIVILEGES;
-- 
USE `persona_db`;
-- 
CREATE TABLE IF NOT EXISTS `persona_db`.`persona` (
`cc` INT(15) NOT NULL,
`nombre` VARCHAR(45) NOT NULL,
`apellido` VARCHAR(45) NOT NULL,
`genero` ENUM('M', 'F') NOT NULL,
`edad` INT(3) NULL DEFAULT NULL,
CONSTRAINT `persona_pk` PRIMARY KEY (`cc`)
);
-- 
CREATE TABLE IF NOT EXISTS `persona_db`.`profesion` (
`id` INT(6) NOT NULL,
`nom` VARCHAR(90) NOT NULL,
`des` TEXT NULL DEFAULT NULL,
CONSTRAINT `profesion_pk` PRIMARY KEY (`id`)
);
-- 
CREATE TABLE IF NOT EXISTS `persona_db`.`telefono` (
`num` VARCHAR(15) NOT NULL,
`oper` VARCHAR(45) NOT NULL,
`duenio` INT(15) NOT NULL,
CONSTRAINT `telefono_pk` PRIMARY KEY (`num`),
CONSTRAINT `telefono_persona_fk` FOREIGN KEY (`duenio`) REFERENCES `persona_db`.`persona` (`cc`)
);
-- 
CREATE TABLE IF NOT EXISTS `persona_db`.`estudios` (
`id_prof` INT(6) NOT NULL,
`cc_per` INT(15) NOT NULL,
`fecha` DATE NULL DEFAULT NULL,
`univer` VARCHAR(50) NULL DEFAULT NULL,
CONSTRAINT `estudios_pk` PRIMARY KEY (`id_prof`, `cc_per`),
CONSTRAINT `estudio_persona_fk` FOREIGN KEY (`cc_per`) REFERENCES `persona_db`.`persona` (`cc`),
CONSTRAINT `estudio_profesion_fk` FOREIGN KEY (`id_prof`) REFERENCES `persona_db`.`profesion` (`id`)
);
-- 
COMMIT;
FLUSH PRIVILEGES;
```

persona_dml_maria.sql

```
INSERT INTO
    `persona_db`.`persona`(`cc`, `nombre`, `apellido`, `genero`, `edad`)
VALUES
    (123456789, 'Pepe', 'Perez', 'M', 30),
    (987654321, 'Pepito', 'Perez', 'M', null),
    (321654987, 'Pepa', 'Juarez', 'F', 30),
    (147258369, 'Pepita', 'Juarez', 'F', 10),
    (963852741, 'Fede', 'Perez', 'M', 18);
```

persona_ddl_mongo.sql

```
conn = new Mongo();
db = conn.getDB("admin");
db.createUser({
    user: "persona_db",
    pwd: "persona_db",
    roles: [
        { role: "read", db: "persona_db" },
        { role: "readWrite", db: "persona_db" },
        { role: "dbAdmin", db: "persona_db" }
    ],
    mechanisms: ["SCRAM-SHA-1", "SCRAM-SHA-256"]
});
```

persona_dml_mongo.sql

```
conn = new Mongo();
db = conn.getDB("personaDb");
db.persona.insertMany([
    {
        "_id": NumberInt(123456789),  Unresolved function or method NumberInt().
        "nombre": "Pepe",
        "apellido": "Perez",
        "genero": "M",
        "edad": NumberInt(30),  Unresolved function or method NumberInt().
        "_class": "co.edu.javeriana.as.personapp.mongo.document.PersonaDocument"
    },
    {
        "_id": NumberInt(987654321),  Unresolved function or method NumberInt().
        "nombre": "Pepito",
        "apellido": "Perez",
        "genero": "M",
        "edad": null,
        "_class": "co.edu.javeriana.as.personapp.mongo.document.PersonaDocument"
    },
    {
        "_id": NumberInt(321654987),  Unresolved function or method NumberInt().
        "nombre": "Pepa",
        "apellido": "Juarez",
        "genero": "F",
        "edad": 30,
        "_class": "co.edu.javeriana.as.personapp.mongo.document.PersonaDocument"
    },
    {
        "_id": NumberInt(147258369),  Unresolved function or method NumberInt().
        "nombre": "Pepita",
        "apellido": "Juarez",
        "genero": "F",
        "edad": 10,
        "_class": "co.edu.javeriana.as.personapp.mongo.document.PersonaDocument"
    },
    {
        "_id": NumberInt(963852741),  Unresolved function or method NumberInt().
        "nombre": "Fede",
        "apellido": "Perez",
        "genero": "M",
        "edad": 18,
        "_class": "co.edu.javeriana.as.personapp.mongo.document.PersonaDocument"
    }
]);
```

3.3 Implementación del Proyecto en Spring Boot

Con el objetivo de implementar una arquitectura hexagonal centrada en el dominio y utilizando un patrón modular, se propone el siguiente POM, en el cual el orden de los componentes es fundamental para alcanzar dicho objetivo. En esta estructura, las entidades se mantienen lo más aisladas posible, mientras que posteriormente se desarrollan los puertos y adaptadores correspondientes.

```
<modules>
    <module>common</module>
    <module>domain</module>
    <module>application</module>
    <module>maria-output-adapter</module>
    <module>mongo-output-adapter</module>
    <module>rest-input-adapter</module>
    <module>cli-input-adapter</module>
</modules>
```

En este contexto, y dado que el módulo del dominio correspondiente a las entidades ha sido implementado en su totalidad siguiendo el enfoque del ejercicio, a continuación, se documentará el proceso relacionado con el módulo "Application".

3.3.1 Documentación Desarrollo de Puertos y Adaptadores

La capa de "application" debe interactuar únicamente con la capa de "domain" y acceder a la infraestructura a través de la inyección de dependencias. En la entidad implementada parcialmente para la guía (Persona), se puede observar el proceso mencionado anteriormente en la clase correspondiente a los casos de uso, donde se utiliza la anotación @UseCase.

Durante el tiempo de ejecución, se selecciona la base de datos en la que se persistirán los datos, ya que los adaptadores cuentan con un *qualifier* específico tal y cómo se evidencia a continuación.

```
@Slf4j
@UseCase
public class PersonUseCase implements PersonInputPort {

    12 usages
    private PersonOutputPort personPersistence;

    public PersonUseCase(@Qualifier("personOutputAdapterMaria") PersonOutputPort personPersistence) {
        this.personPersistence=personPersistence;
    }
}
```

En este contexto, se documentará el proceso de desarrollo de los puertos de entrada y salida, así como las implementaciones correspondientes para la base de datos MariaDB y MongoDB. Este proceso concluirá con la descripción de la implementación de los casos de uso para cada una de las entidades restantes.

Solución Problemas Previos con Mappers:

Para el desarrollo del proyecto, el profesor proporcionó una guía inicial que sirve como base para continuar con el avance del mismo. No obstante, es fundamental señalar que se identificaron múltiples errores en los mapeadores (Mappers) en la mayoría de los módulos del proyecto. Estos problemas afectan directamente la funcionalidad y ejecución correcta de las distintas partes del sistema.

A continuación, se presenta el código fuente de cada uno de los Mappers, incluyendo las correcciones necesarias para asegurar el adecuado funcionamiento del proyecto. Estas modificaciones se realizaron con el fin de resolver el problema de StackOverflow que se manifestaba en los mapeadores de los adaptadores de salida para las bases de datos MongoDB y MariaDB. Para cada entidad, se aplicaron ajustes específicos en ambos módulos, garantizando que el código sea legible, mantenable y escalable sin introducir nuevas complicaciones.

Mappers Módulo maria-output-adapter

Los mapeadores (Mappers) en el módulo maria_output_adapter se encargan de transformar las entidades de dominio a entidades adaptadas para su almacenamiento en bases de datos específicas, y viceversa. Este proceso de conversión asegura la correcta persistencia y recuperación de datos, manteniendo la coherencia entre el modelo de dominio y el modelo de almacenamiento.

Funcionalidades Generales:

1. **Método de Mapeo de Dominio a Adaptador:** Este método recibe una entidad de dominio (por ejemplo, Persona, Telefono, Profesion o Estudios) y la transforma en una entidad adaptada para la base de datos específica. La transformación incluye asignar valores de los atributos de dominio a los atributos de la entidad de almacenamiento, aplicando validaciones según el tipo de dato (por ejemplo, validación de género o edad en el caso de Persona).
2. **Método de Mapeo de Adaptador a Dominio:** Este método realiza la operación inversa, convirtiendo una entidad de base de datos en una entidad de dominio. Permite que los datos recuperados de la base de datos puedan ser interpretados y utilizados por la lógica de negocio en su forma original.
3. **Validaciones:** Los mapeadores incluyen métodos de validación específicos para los atributos clave de cada entidad, como puede ser la validación de género, edad, o listas de relaciones (como listas de Telefono o Estudios). Estas validaciones aseguran que los datos que se procesan cumplen con los requisitos del dominio, mejorando la calidad de los datos.
4. **Transformación de Colecciones:** Cuando una entidad contiene colecciones de objetos relacionados (por ejemplo, una Persona puede tener múltiples Telefono o Estudios), el mapeador se encarga de iterar y transformar cada elemento de la colección para mantener la integridad de la relación entre entidades.
5. **Inyección de Dependencias:** Los mapeadores utilizan inyección de dependencias para otros mapeadores de entidades relacionadas, permitiendo una transformación completa y modular sin replicar código innecesario. Por ejemplo, el PersonaMapperMaria puede

depender de TelefonoMapperMaria y EstudiosMapperMaria para transformar los respectivos atributos.

4. Adaptabilidad:

Esta estructura modular permite que los mapeadores sean fácilmente adaptables para diferentes entidades en el sistema, como Telefono, Profesion y Estudios, simplemente replicando el esquema de métodos de transformación (de dominio a adaptador y viceversa), las validaciones necesarias, y las colecciones asociadas. Esta consistencia facilita el mantenimiento y la escalabilidad del código.

Mapper Entidad Persona:

```
@Mapper
public class PersonaMapperMaria {

    @Autowired
    private EstudiosMapperMaria estudiosMapperMaria;

    @Autowired
    private TelefonoMapperMaria telefonoMapperMaria;

    2 usages
    public PersonaEntity fromDomainToAdapter(Person person) {
        if (person == null) return null;

        PersonaEntity personaEntity = new PersonaEntity();
        personaEntity.setCc(person.getIdentification());
        personaEntity.setNombre(person.getFirstName());
        personaEntity.setApellido(person.getLastName());
        personaEntity.setGenero(validateGenero(person.getGender()));
        personaEntity.setEdad(validateEdad(person.getAge()));
        personaEntity.setEstudios(validateEstudios(person.getStudies()));
        personaEntity.setTelefonos(mapPhonesWithoutOwnerValidation(person.getPhoneNumbers()));
        return personaEntity;
    }

    1 usage
    private Character validateGenero(@NotNull Gender gender) {
        return gender == Gender.FEMALE ? 'F' : gender == Gender.MALE ? 'M' : ' ';
    }

    1 usage
    private Integer validateEdad(Integer age) { return age != null && age >= 0 ? age : null; }

    1 usage
    private List<EstudiosEntity> validateEstudios(List<Study> studies) {
        return studies != null && !studies.isEmpty()
            ? studies.stream().map(estudiosMapperMaria::fromDomainToAdapter).collect(Collectors.toList())
            : new ArrayList<>();
    }
}
```

```

private List<TelefonoEntity> mapPhonesWithoutOwnerValidation(List<Phone> phoneNumbers) {
    return phoneNumbers != null && !phoneNumbers.isEmpty()
        ? phoneNumbers.stream() Stream<Phone>
            .map(telefonoMapperMaria::fromDomainToAdapterWithoutOwner) Stream<TelefonoEntity>
            .collect(Collectors.toList())
            : new ArrayList<>();
}

4 usages
public Person fromAdapterToDomain(PersonaEntity personaEntity) {
    if (personaEntity == null) return null;

    return Person.builder()
        .identification(personaEntity.getCc())
        .firstName(personaEntity.getNombre())
        .lastName(personaEntity.getApellido())
        .gender(validateGender(personaEntity.getGenero()))
        .age(validateAge(personaEntity.getEdad()))
        .studies(validateStudies(personaEntity.getEstudios()))
        .phoneNumbers(mapPhonesWithoutOwner(personaEntity.getTelefonos()))
        .build();
}

```

```

1 usage
public Person fromAdapterToDomainBasic(PersonaEntity personaEntity) {
    if (personaEntity == null) return null;

    return Person.builder()
        .identification(personaEntity.getCc())
        .firstName(personaEntity.getNombre())
        .lastName(personaEntity.getApellido())
        .gender(Gender.OTHER)
        .build();
}

1 usage
private @NotNull Gender validateGender(Character genero) {
    return genero == 'F' ? Gender.FEMALE : genero == 'M' ? Gender.MALE : Gender.OTHER;
}

1 usage
private Integer validateAge(Integer edad) { return edad != null && edad >= 0 ? edad : null; }

1 usage
private List<Study> validateStudies(List<EstudiosEntity> estudiosEntity) {
    return estudiosEntity != null && !estudiosEntity.isEmpty()
        ? estudiosEntity.stream() Stream<EstudiosEntity>
            .map(estudiosMapperMaria::fromAdapterToDomain) Stream<Study>
            .collect(Collectors.toList())
            : new ArrayList<>();
}

1 usage
private List<Phone> mapPhonesWithoutOwner(List<TelefonoEntity> telefonoEntities) {
    return telefonoEntities != null && !telefonoEntities.isEmpty()
        ? telefonoEntities.stream() Stream<TelefonoEntity>
            .map(telefonoMapperMaria::fromAdapterToDomainWithoutOwner) Stream<Phone>
            .collect(Collectors.toList())
            : new ArrayList<>();
}

```

Mappers Entidad Estudio:

```
@Mapper
public class EstudiosMapperMaria {

    @Autowired
    private PersonaMapperMaria personaMapperMaria;

    @Autowired
    private ProfesionMapperMaria profesionMapperMaria;

    3 usages
    public EstudiosEntity fromDomainToAdapter(Study study) {
        if (study == null) {
            return null;
        }
        EstudiosEntityPK estudioPK = new EstudiosEntityPK();
        estudioPK.setCcPer(study.getPerson().getIdentification());
        estudioPK.setIdProf(study.getProfession().getIdentification());
        EstudiosEntity estudio = new EstudiosEntity();
        estudio.setEstudiosPK(estudioPK);
        estudio.setFecha(validateFecha(study.getGraduationDate()));
        estudio.setUniver(validateUniver(study.getUniversityName()));
        return estudio;
    }

    1 usage
    private Date validateFecha(LocalDate graduationDate) {
        return graduationDate != null
            ? Date.from(graduationDate.atStartOfDay().atZone(ZoneId.systemDefault()).toInstant())
            : null;
    }
}
```

```
1 usage
private String validateUniver(String universityName) { return universityName != null ? universityName : ""; }

5 usages
public Study fromAdapterToDomain(EstudiosEntity estudiosEntity) {
    if (estudiosEntity == null || estudiosEntity.getPersona() == null || estudiosEntity.getProfesion() == null) {
        return null;
    }

    return Study.builder()
        .person(personaMapperMaria.fromAdapterToDomainBasic(estudiosEntity.getPersona()))
        .profession(profesionMapperMaria.fromAdapterToDomainBasic(estudiosEntity.getProfesion()))
        .graduationDate(validateGraduationDate(estudiosEntity.getFecha()))
        .universityName(validateUniversityName(estudiosEntity.getUniver()))
        .build();
}

1 usage
private LocalDate validateGraduationDate(Date fecha) {
    if (fecha == null) {
        return null;
    }

    if (fecha instanceof java.sql.Date) {
        return ((java.sql.Date) fecha).toLocalDate();
    }
    return fecha.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
}

1 usage
private String validateUniversityName(String univer) { return univer != null ? univer : ""; }
}
```

Mappers Entidad Profesión:

```
@Mapper
public class ProfesionMapperMaria {

    @Autowired
    private EstudiosMapperMaria estudiosMapperMaria;

    1 usage
    public ProfesionEntity fromDomainToAdapter(Profession profession) {
        if (profession == null) return null;

        ProfesionEntity profesionEntity = new ProfesionEntity();
        profesionEntity.setId(profession.getIdentification());
        profesionEntity.setNom(profession.getName());
        profesionEntity.setDes(validateDescription(profession.getDescription()));
        profesionEntity.setEstudios(validateEstudios(profession.getStudies())); // Solo en mapeo completo
        return profesionEntity;
    }

    3 usages
    private String validateDescription(String description) { return description != null ? description : ""; }

    1 usage
    private List<EstudiosEntity> validateEstudios(List<Study> studies) {
        return studies != null && !studies.isEmpty()
            ? studies.stream()
                .map(estudiosMapperMaria::fromDomainToAdapter) // Mapeo básico de estudios
                .collect(Collectors.toList())
                : new ArrayList<>();
    }

    3 usages
    public Profession fromAdapterToDomain(ProfesionEntity profesionEntity) {
        if (profesionEntity == null) return null;

        return Profession.builder()
            .identification(profesionEntity.getId())
            .name(profesionEntity.getNom())
            .description(validateDescription(profesionEntity.getDes()))
            .studies(validateStudies(profesionEntity.getEstudios())) // Solo en mapeo completo
            .build();
    }
}
```

```
public Profession fromAdapterToDomainBasic(ProfesionEntity profesionEntity) {
    if (profesionEntity == null) return null;

    return Profession.builder()
        .identification(profesionEntity.getId())
        .name(profesionEntity.getNom())
        .description(validateDescription(profesionEntity.getDes()))
        .build(); // Sin estudios para evitar ciclos
}

1 usage
private List<Study> validateStudies(List<EstudiosEntity> estudiosEntity) {
    return estudiosEntity != null && !estudiosEntity.isEmpty()
        ? estudiosEntity.stream()
            .map(estudiosMapperMaria::fromAdapterToDomain) // Mapeo básico de estudios
            .collect(Collectors.toList())
            : new ArrayList<>();
}
```

Mappers Entidad Teléfono:

```
@Mapper
public class TelefonoMapperMaria {
    @Autowired
    private PersonaMapperMaria personaMapperMaria;
    1 usage
    public TelefonoEntity fromDomainToAdapter(Phone phone) {
        TelefonoEntity telefonoEntity = new TelefonoEntity();
        telefonoEntity.setNum(phone.getNumber());
        telefonoEntity.setOper(phone.getCompany());
        telefonoEntity.setDuenio(mapOwnerWithoutPhones(phone.getOwner()));
        return telefonoEntity;
    }

    1 usage
    public TelefonoEntity fromDomainToAdapterWithoutOwner(Phone phone) {
        TelefonoEntity telefonoEntity = new TelefonoEntity();
        telefonoEntity.setNum(phone.getNumber());
        telefonoEntity.setOper(phone.getCompany());
        return telefonoEntity;
    }
    1 usage
    private PersonaEntity mapOwnerWithoutPhones(Person owner) {
        return owner != null ? personaMapperMaria.fromDomainToAdapter(owner) : new PersonaEntity();
    }
    3 usages
    public Phone fromAdapterToDomain(TelefonoEntity telefonoEntity) {
        Phone phone = new Phone();
        phone.setNumber(telefonoEntity.getNum());
        phone.setCompany(telefonoEntity.getOper());
        Person owner = personaMapperMaria.fromAdapterToDomain(telefonoEntity.getDuenio());
        phone.setOwner(owner);
        return phone;
    }
    1 usage
    public Phone fromAdapterToDomainWithoutOwner(TelefonoEntity telefonoEntity) {
        Phone phone = new Phone();
        phone.setNumber(telefonoEntity.getNum());
        phone.setCompany(telefonoEntity.getOper());
        return phone;
    }
}
```

Mappers Módulo maria-output-adapter

Los mapeadores (Mappers) en el módulo mongo_output_adapter se encargan de transformar las entidades de dominio a entidades adaptadas para su almacenamiento en MongoDB, y viceversa. Este proceso de conversión garantiza la correcta persistencia y recuperación de datos en una estructura de documentos, manteniendo la consistencia entre el modelo de dominio y el modelo de almacenamiento en MongoDB.

Funcionalidades Generales:

1. **Método de Mapeo de Dominio a Adaptador:** Este método recibe una entidad de dominio (como Persona, Telefono, Profesion o Estudios) y la convierte en un documento adaptado para ser almacenado en MongoDB. La transformación implica asignar valores desde los atributos de la entidad de dominio a los atributos del documento MongoDB, aplicando las validaciones necesarias según el tipo de dato (por ejemplo, validación de género o edad en el caso de Persona).
2. **Método de Mapeo de Adaptador a Dominio:** Este método realiza el proceso inverso, transformando un documento de MongoDB en una entidad de dominio. Así, los datos recuperados de la base de datos pueden ser interpretados y utilizados por la lógica de negocio en su forma de dominio original.
3. **Validaciones:** Los mapeadores incluyen métodos de validación específicos para los atributos clave de cada entidad, como la validación de género, edad, o listas de relaciones (por ejemplo, listas de Telefono o Estudios). Estas validaciones aseguran que los datos procesados cumplen con los requisitos del dominio, mejorando la integridad de la información almacenada.
4. **Transformación de Colecciones:** Cuando una entidad contiene colecciones de objetos relacionados (por ejemplo, una Persona puede tener múltiples Telefono o Estudios), el mapeador se encarga de iterar y transformar cada elemento de la colección para mantener la consistencia de la relación en MongoDB, respetando la estructura de documentos y embebidos característica de esta base de datos.
5. **Inyección de Dependencias:** Los mapeadores utilizan inyección de dependencias para otros mapeadores de entidades relacionadas, lo que permite una transformación completa y modular sin duplicación de código. Por ejemplo, el PersonaMapperMongo puede depender de TelefonoMapperMongo y EstudiosMapperMongo para realizar las transformaciones correspondientes de los atributos de Persona relacionados con Telefono y Estudios.

Adaptabilidad:

Esta estructura modular permite que los mapeadores sean fácilmente adaptables para diferentes entidades en el sistema, como Telefono, Profesion y Estudios, replicando el mismo esquema de métodos de transformación (de dominio a adaptador y viceversa), las validaciones específicas, y las transformaciones de colecciones asociadas. La consistencia en el diseño facilita el mantenimiento y la escalabilidad del código en MongoDB, permitiendo un desarrollo más eficiente y estructurado.

Mapper Entidad Persona:

```
@Mapper
public class PersonaMapperMongo {

    @Autowired
    @Lazy
    private EstudiosMapperMongo estudiosMapperMongo;

    3 usages
    public PersonaDocument fromDomainToAdapter(Person person) {
        return PersonaDocument.builder()
            .id(person.getIdentification())
            .nombre(person.getFirstName())
            .apellido(person.getLastName())
            .genero(validateGenero(person.getGender()))
            .edad(person.getAge() != null ? person.getAge() : 0)
            .estudios(validateEstudios(person.getStudies()))
            .telefonos(new ArrayList<>())
            .build();
    }

    4 usages
    public Person fromAdapterToDomain(PersonaDocument personaDocument) {
        return Person.builder()
            .identification(personaDocument.getId() != null ? personaDocument.getId() : 0) // Maneja nulo con un valor por defecto
            .firstName(personaDocument.getNombre() != null ? personaDocument.getNombre() : "Desconocido")
            .lastName(personaDocument.getApellido() != null ? personaDocument.getApellido() : "Desconocido")
            .gender(validateGenero(personaDocument.getGenero()))
            .age(personaDocument.getEdad() != null ? personaDocument.getEdad() : 0)
            .studies(validateStudies(personaDocument.getEstudios()))
            .phoneNumbers(new ArrayList<>())
            .build();
    }

    1 usage
    private String validateGenero(Gender gender) {
        return gender == Gender.FEMALE ? "F" : gender == Gender.MALE ? "M" : "";
    }
}
```

```

private Gender validateGender(String genero) {
    return "F".equals(genero) ? Gender.FEMALE : "M".equals(genero) ? Gender.MALE : Gender.OTHER;
}

1 usage
private List<EstudiosDocument> validateEstudios(List<Study> studies) {
    return studies != null ? studies.stream() Stream<Study>
        .map(study -> {
            EstudiosDocument doc = estudiosMapperMongo.fromDomainToAdapter(study);
            doc.setPrimaryPersona(null);
            return doc;
        }) Stream<EstudiosDocument>
        .collect(Collectors.toList()) : new ArrayList<>();
}

1 usage
private List<Study> validateStudies(List<EstudiosDocument> estudiosDocuments) {
    return estudiosDocuments != null ? estudiosDocuments.stream() Stream<EstudiosDocument>
        .map(estudiosMapperMongo::fromAdapterToDomainBasic) Stream<Study>
        .collect(Collectors.toList()) : new ArrayList<>();
}

2 usages
public Person fromAdapterToDomainBasic(PersonaDocument personaDocument) {
    return Person.builder()
        .identification(personaDocument.getId() != null ? personaDocument.getId() : 0)
        .firstName(personaDocument.getNombre() != null ? personaDocument.getNombre() : "Desconocido")
        .lastName(personaDocument.getApellido() != null ? personaDocument.getApellido() : "Desconocido")
        .gender(validateGender(personaDocument.getGenero()))
        .age(personaDocument.getEdad() != null ? personaDocument.getEdad() : 0)
        .build(); // No cargar 'studies' ni 'phoneNumbers' para evitar referencias ciclicas
}

```

Mapper Entidad Teléfono:

```

@Mapper
public class TelefonoMapperMongo {

    @Autowired
    @Lazy
    private PersonaMapperMongo personaMapperMongo;

    1 usage
    public TelefonoDocument fromDomainToAdapter(Phone phone) {
        PersonaDocument ownerDocument = personaMapperMongo.fromDomainToAdapter(phone.getOwner());
        return TelefonoDocument.builder()
            .id(phone.getNumber())
            .oper(phone.getCompany())
            .primaryDuenio(ownerDocument) // Asegura que el dueño no sea null
            .build();
    }

    3 usages
    public Phone fromAdapterToDomain(TelefonoDocument telefonoDocument) {
        return Phone.builder()
            .number(telefonoDocument.getId())
            .company(telefonoDocument.getOper())
            .owner(telefonoDocument.getPrimaryDuenio() != null ?
                personaMapperMongo.fromAdapterToDomainBasic(telefonoDocument.getPrimaryDuenio()) : null)
            .build();
    }
}

```

Mapper Entidad Profesión:

```
@Mapper
public class ProfesionMapperMongo {

    @Autowired
    private EstudiosMapperMongo estudiosMapperMongo;

    2 usages
    public ProfesionDocument fromDomainToAdapter(Profession profession) {
        ProfesionDocument profesionDocument = new ProfesionDocument();
        profesionDocument.setId(profession.getIdentification());
        profesionDocument.setNom(profession.getName());
        profesionDocument.setDes(validateDes(profession.getDescription()));
        profesionDocument.setEstudios(validateEstudios(profession.getStudies()));
        return profesionDocument;
    }

    1 usage  4 related problems
    private String validateDes(String description) {
        return description != null ? description : "";
    }

    1 usage
    private List<EstudiosDocument> validateEstudios(List<Study> studies) {
        return studies != null && !studies.isEmpty() ? studies.stream()
            .map(study -> estudiosMapperMongo.fromDomainToAdapter(study)).collect(Collectors.toList())
            : new ArrayList<EstudiosDocument>();
    }

    2 usages
    public Profession fromAdapterToDomain(ProfesionDocument profesionDocument) {
        Profession profession = new Profession();
        profession.setIdentification(profesionDocument.getId());
        profession.setName(profesionDocument.getNom());
        profession.setDescription(validateDescription(profesionDocument.getDes()));
        profession.setStudies(validateStudies(profesionDocument.getEstudios()));
        return profession;
    }
}
```

```
private String validateDescription(String des) {
    return des != null ? des : "";
}

3 usages
public Profession fromAdapterToDomainBasic(ProfesionDocument profesionDocument) {
    return Profession.builder()
        .identification(profesionDocument.getId() != null ? profesionDocument.getId() : 0)
        .name(profesionDocument.getNom() != null ? profesionDocument.getNom() : "Desconocido")
        .description(profesionDocument.getDes() != null ? profesionDocument.getDes() : "")
        .build(); // No cargar 'studies' para evitar referencias cíclicas
}

1 usage
private List<Study> validateStudies(List<EstudiosDocument> estudiosDocument) {
    return estudiosDocument != null && !estudiosDocument.isEmpty() ? estudiosDocument.stream()
        .map(estudio -> estudiosMapperMongo.fromAdapterToDomainBasic(estudio)).collect(Collectors.toList())
        : new ArrayList<Study>();
}
```

Mappers Entidad Estudio:

```

@Mapper
public class EstudiosMapperMongo {
    @Autowired
    @Lazy
    private PersonaMapperMongo personaMapperMongo;

    @Autowired
    private ProfesionMapperMongo profesionMapperMongo;
    3 usages
    public EstudiosDocument fromDomainToAdapter(Study study) {
        EstudiosDocument estudio = new EstudiosDocument();
        estudio.setId(validateId(study.getPerson().getIdentification(), study.getProfession().getIdentification()));
        estudio.setPrimaryPersona(validatePrimaryPersona(study.getPerson()));
        estudio.setPrimaryProfesion(validatePrimaryProfesion(study.getProfession()));
        estudio.setFecha(validateFecha(study.getGraduationDate()));
        estudio.setUniver(validateUniver(study.getUniversityName()));
        return estudio;
    }
    1 usage
    private String validateId(@NotNull Integer identificationPerson, @NotNull Integer identificationProfession) {
        return identificationPerson + "-" + identificationProfession;
    }
    1 usage
    private PersonaDocument validatePrimaryPersona(@NotNull Person person) {
        PersonaDocument personaDoc = personaMapperMongo.fromDomainToAdapter(person);
        personaDoc.setEstudios(null);
        return personaDoc;
    }
    1 usage
    private ProfesionDocument validatePrimaryProfesion(@NotNull Profession profession) {
        return profession != null ? profesionMapperMongo.fromDomainToAdapter(profession) : new ProfesionDocument();
    }
    1 usage
    private LocalDate validateFecha(LocalDate graduationDate) {
        return graduationDate != null ? graduationDate : null;
    }

    1 usage
    private String validateUniver(String universityName) {
        return universityName != null ? universityName : "";
    }
}

```

```

2 usages
public Study fromAdapterToDomain(EstudiosDocument estudiosDocument) {
    Study study = new Study();
    study.setPerson(personaMapperMongo.fromAdapterToDomain(estudiosDocument.getPrimaryPersona()));
    study.setProfesion(profesionMapperMongo.fromAdapterToDomain(estudiosDocument.getPrimaryProfesion()));
    study.setGraduationDate(validateGraduationDate(estudiosDocument.getFecha()));
    study.setUniversityName(validateUniversityName(estudiosDocument.getUniver()));
    return study;
}
3 usages
public Study fromAdapterToDomainBasic(EstudiosDocument estudiosDocument) {
    Study.StudyBuilder studyBuilder = Study.builder();

    studyBuilder.person(estudiosDocument.getPrimaryPersona() != null
        ? personaMapperMongo.fromAdapterToDomainBasic(estudiosDocument.getPrimaryPersona())
        : Person.builder().identification(0).firstName("Desconocido").build());
    studyBuilder.profession(estudiosDocument.getPrimaryProfesion() != null
        ? profesionMapperMongo.fromAdapterToDomainBasic(estudiosDocument.getPrimaryProfesion())
        : Profession.builder().identification(0).name("Desconocido").build());
    studyBuilder.graduationDate(validateGraduationDate(estudiosDocument.getFecha()));
    studyBuilder.universityName(validateUniversityName(estudiosDocument.getUniver()));
    return studyBuilder.build();
}
2 usages
private LocalDate validateGraduationDate(LocalDate fecha) {
    return fecha != null ? fecha : null;
}
2 usages
private String validateUniversityName(String univer) {
    return univer != null ? univer : "";
}
}

```

Documentación Desarrollo del Estilo Arquitectónico Hexagonal - Entidad Profesion

Creación del Puerto de Salida (ProfessionOutputPort): Módulo Application:

```
@Port
public interface ProfessionOutputPort {
    2 implementations
    public Profession save(Profession profession);
    2 implementations
    public Boolean delete(Integer identification);
    2 implementations
    public List<Profession> find();
    2 implementations
    public Profession findById(Integer identification);
}
```

La interfaz ProfessionOutputPort define los métodos esenciales para la gestión de profesiones en el sistema, alineándose con la arquitectura hexagonal. Su propósito es facilitar la interacción con la persistencia de datos, garantizando una separación de responsabilidades.

Creación del Puerto de Entrada (ProfessionInputPort): Módulo Application:

```
@Port
public interface ProfessionInputPort {
    no usages 1 implementation
    public void setPersistence(ProfessionOutputPort professionPersistance);
    1 implementation
    public Profession create(Profession profession);
    no usages 1 implementation
    public Profession edit(Integer identification, Profession profession) throws NoExistException;
    no usages 1 implementation
    public Boolean drop(Integer identification) throws NoExistException;
    1 implementation
    public List<Profession> findAll();
    1 usage 1 implementation
    public Profession findOne(Integer identification) throws NoExistException;
    no usages 1 implementation
    public Integer count();
}
```

La interfaz ProfessionInputPort tiene como objetivo definir los métodos necesarios para la gestión de profesiones en el sistema, siguiendo los principios de la arquitectura hexagonal. Anotada con @Port, esta interfaz permite la inyección de dependencias, asegurando que la lógica de negocio no dependa de las implementaciones concretas de la infraestructura.

Nota Importante:

Antes de implementar el UseCase para la entidad, se llevarán a cabo las implementaciones de los adaptadores de salida para los dos sistemas de bases de datos (MariaDB y MongoDB). De este modo, el *UseCase* al que se conectan los adaptadores de entrada podrá realizar las operaciones a través de la inyección de dependencias sobre la persistencia de datos.

Implementación Adaptador de Salida Maria DB para la Entidad Profesión

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio JPA y el desarrollo del adaptador de salida para la entidad Profesión.

```
package co.edu.javeriana.as.personapp.mariadb.repository;

import co.edu.javeriana.as.personapp.mariadb.entity.ProfesionEntity;
import org.springframework.data.jpa.repository.JpaRepository;

no usages
public interface ProfesionRepositoryMaria extends JpaRepository<ProfesionEntity, Integer> { }
```

La interfaz ProfesionRepositoryMaria es un repositorio JPA que gestiona la persistencia de la entidad ProfesionEntity en una base de datos MariaDB. Al extender JpaRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar profesiones. Parametrizada con ProfesionEntity e Integer, permite encapsular la lógica relacionada con una profesión y su identificador único. Esta interfaz proporciona una capa de abstracción que facilita el enfoque en la lógica de negocio

```

@Slf4j
@Adapter("profesionOutPutAdapterMaria")
@Transactional
public class ProfesionOutputAdapterMaria implements ProfessionOutputPort {
    @Autowired
    private ProfesionRepositoryMaria profesionRepositoryMaria;
    @Autowired
    private ProfesionMapperMaria profesionMapperMaria;

    @Override
    public Profession save(Profession profession) {
        log.debug("Into save on Adapter MariaDB");
        ProfesionEntity persistedProfesion = profesionRepositoryMaria.save(profesionMapperMaria.fromDomainToAdapter(profession));
        return profesionMapperMaria.fromAdapterToDomain(persistedProfesion);
    }

    @Override
    public Boolean delete(Integer identification) {
        log.debug("Into delete on Adapter MariaDB");
        profesionRepositoryMaria.deleteById(identification);
        return profesionRepositoryMaria.findById(identification).isEmpty();
    }

    @Override
    public Profession findById(Integer identification) {
        log.debug("Into findById on Adapter MariaDB");
        if (profesionRepositoryMaria.findById(identification).isEmpty()) {
            return null;
        } else {
            return profesionMapperMaria.fromAdapterToDomain(profesionRepositoryMaria.findById(identification).get());
        }
    }

    @Override
    public List<Profession> find() {
        log.debug("Into find on Adapter MariaDB");
        return profesionRepositoryMaria.findAll().stream().map(profesionMapperMaria::fromAdapterToDomain).collect(Collectors.toList());
    }
}

```

El adaptador de salida ProfesionOutputAdapterMaria es una implementación clave dentro de la arquitectura hexagonal, ya que establece la conexión entre la lógica de negocio y la base de datos MariaDB. Al implementar la interfaz ProfessionOutputPort, este adaptador permite que el sistema interactúe con la infraestructura de manera desacoplada, lo que es fundamental para mantener la flexibilidad y la mantenibilidad de la aplicación. Este enfoque garantiza que los cambios en la infraestructura, como la migración a otra base de datos, no afecten la lógica de negocio.

Implementación Adaptador de Salida Mongo DB para la Entidad Profesión

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio Mongo y el desarrollo del adaptador de salida para la entidad Profesión.

```

package co.edu.javeriana.as.personapp.mongo.repository;

import co.edu.javeriana.as.personapp.mongo.document.ProfesionDocument;
import org.springframework.data.mongodb.repository.MongoRepository;

no usages
public interface ProfesionRepositoryMongo extends MongoRepository<ProfesionDocument, Integer> {
}

```

La interfaz ProfesionRepositoryMongo es un repositorio que gestiona la persistencia de la entidad ProfesionDocument en una base de datos MongoDB. Al extender MongoRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar profesiones en un entorno NoSQL. Parametrizada con ProfesionDocument e Integer, esta interfaz encapsula la lógica relacionada con una profesión y su identificador único. Su diseño permite una interacción eficiente con la base de datos, proporcionando una capa de abstracción que facilita la separación entre la lógica de negocio y la infraestructura de datos, lo que resulta en una arquitectura más limpia y mantenible.

```

@Slf4j
@Adapter("profesionOutputAdapterMongo")
public class ProfesionOutputAdapterMongo implements ProfesionOutputPort {
    @Autowired
    private ProfesionRepositoryMongo profesionRepositoryMongo;

    @Autowired
    private ProfesionMapperMongo profesionMapperMongo;

    @Override
    public Profession save(Profession profession) {
        log.debug("Into save on Adapter MongoDB");
        try {
            ProfesionDocument persistedProfession = profesionRepositoryMongo.save(profesionMapperMongo.fromDomainToAdapter(profession));
            return profesionMapperMongo.fromAdapterToDomainBasic(persistedProfession);
        } catch (MongoWriteException e) {
            log.warn(e.getMessage());
            return profession;
        }
    }

    @Override
    public Boolean delete(Integer identification) {
        log.debug("Into delete on Adapter MongoDB");
        profesionRepositoryMongo.deleteById(identification);
        return profesionRepositoryMongo.findById(identification).isEmpty();
    }

    @Override
    public Profession findById(Integer identification) {
        log.debug("Into findById on Adapter MongoDB");
        if (profesionRepositoryMongo.findById(identification).isEmpty()) {
            return null;
        } else {
            return profesionMapperMongo.fromAdapterToDomainBasic(profesionRepositoryMongo.findById(identification).get());
        }
    }

    @Override
    public List<Profession> find() {
        log.debug("Into find on Adapter MongoDB");
        return profesionRepositoryMongo.findAll().stream().map(profesionMapperMongo::fromAdapterToDomain)
            .collect(Collectors.toList());
    }
}

```

El adaptador de salida ProfessionOutputAdapterMongo es esencial en la arquitectura hexagonal, ya que conecta la lógica de negocio con la base de datos MongoDB mediante la implementación de la interfaz ProfessionOutputPort, lo que permite una interacción desacoplada y flexible con la infraestructura.

Implementación Use Case Entidad Profesión: Módulo Application

```
@Slf4j
@UseCase
public class ProfessionUseCase implements ProfessionInputPort {
    1 usages
    private ProfessionOutputPort profesionPersistence;
    public ProfessionUseCase(@Qualifier("professionOutputAdapterMaria") ProfessionOutputPort profesionPersistence) {
        this.profesionPersistence = profesionPersistence;
    }
    no usages
    @Override
    public void setPersistence (ProfessionOutputPort professionPersistance) {
        this.profesionPersistence = professionPersistance;
    }
    @Override
    public Profession create(Profession profession) {
        log.debug("Into create on Application Domain");
        return profesionPersistence.save(profession);
    }
    no usages
    @Override
    public Profession edit(Integer identification, Profession profession) throws NoExistException {
        Profession oldProfession = profesionPersistence.findById(identification);
        if (oldProfession != null)
            return profesionPersistence.save(profession);
        throw new NoExistException("The profession with id " + identification + " does not exist into db, cannot be edited");
    }
    no usages
    @Override
    public Boolean drop(Integer identification) throws NoExistException {
        Profession oldProfession = profesionPersistence.findById(identification);
        if (oldProfession != null)
            return profesionPersistence.delete(identification);
        throw new NoExistException("The profession with id " + identification + " does not exist into db, cannot be dropped");
    }
}
```

```
@Override
public List<Profession> findAll() {
    log.info("Output: " + profesionPersistence.getClass());
    return profesionPersistence.find();
}
1 usage
@Override
public Profession findOne(Integer identification) throws NoExistException {
    Profession profession = profesionPersistence.findById(identification);
    if (profession == null)
        throw new NoExistException("The profession with id " + identification + " does not exist into db");
    return profession;
}
no usages
@Override
public Integer count() { return profesionPersistence.find().size(); }
}
```

El ProfessionUseCase es una implementación clave dentro de la arquitectura hexagonal, ya que actúa como el puerto de entrada que permite la interacción con la lógica de negocio relacionada con las profesiones. Al implementar la interfaz ProfessionInputPort, este caso de uso permite que la aplicación se comunique de manera estructurada con los adaptadores de salida, como ProfesionOutputAdapterMaria.

Documentación Desarrollo del Estilo Arquitectónico Hexagonal - Entidad Teléfono

Creación del Puerto de Salida (PhoneOutputPort): Módulo Application:

```
@Port
public interface PhoneOutputPort {
    2 implementations
    public Phone save(Phone phone);
    2 implementations
    public Boolean delete(String identification);
    2 implementations
    public Phone findById(String identification);
    2 implementations
    public List<Phone> find();
}
```

La interfaz PhoneOutputPort define los métodos esenciales para la gestión de teléfonos en el sistema, alineándose con la arquitectura hexagonal. Su propósito es facilitar la interacción con la persistencia de datos, garantizando una separación de responsabilidades.

Creación del Puerto de Entrada (ProfessionInputPort): Módulo Application

```
package co.edu.javeriana.as.personapp.application.port.in;

import ...
4 usages 1 implementation
@Port
public interface PhoneInputPort {
    no usages 1 implementation
    public void setPersistence (PhoneOutputPort phonePersistence, PersonOutputPort personPersistence);
    1 implementation
    public Phone create(Phone phone, int ccPerson) throws NoExistException;
    no usages 1 implementation
    public Phone edit(String identification, Phone phone, int ccPerson) throws NoExistException;
    ⚡ no usages 1 implementation
    public Boolean drop(String identification) throws NoExistException;
    no usages 1 implementation
    public Phone findOne(String identification) throws NoExistException;
    1 implementation
    public List<Phone> findAll();
    no usages 1 implementation
    public Integer count();
}
```

La interfaz PhoneInputPort define los métodos necesarios para la gestión de teléfonos en el sistema, alineándose con los principios de la arquitectura hexagonal. Anotada con @Port, permite la inyección de dependencias, asegurando que la lógica de negocio esté desacoplada de la infraestructura.

Nota Importante:

Antes de implementar el UseCase para la entidad Teléfono, se llevarán a cabo las implementaciones de los adaptadores de salida para los dos sistemas de bases de datos (MariaDB y MongoDB). De este modo, el UseCase al que se conectan los adaptadores de entrada podrá realizar las operaciones correspondientes sobre las entidades del dominio.

Implementación Adaptador de Salida Maria DB para la Entidad Teléfono

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio para MongoDB y el desarrollo del adaptador de salida para la entidad Teléfono.

```

package co.edu.javeriana.as.personapp.mariadb.repository;

import ...

2 usages
public interface TelefonoRepositoryMaria extends JpaRepository<TelefonoEntity, String> {
}

```

La interfaz TelefonoRepositoryMaria es un repositorio JPA que gestiona la persistencia de la entidad TelefonoEntity en una base de datos MariaDB. Al extender JpaRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar teléfonos. Parametrizada con TelefonoEntity e Integer, permite encapsular la lógica relacionada con un teléfono y su identificador único. Esta interfaz proporciona una capa de abstracción que facilita el enfoque en la lógica de negocio.

```

@Slf4j
@Adapter("phoneOutputAdapterMaria")
@Transactional
public class PhoneOutputAdapterMaria implements PhoneOutputPort {
    @Autowired
    private TelefonoRepositoryMaria telefonoRepositoryMaria;
    @Autowired
    private TelefonoMapperMaria telefonoMapperMaria;
    @Override
    public Phone save(Phone phone) {
        log.debug("Into save on Adapter MariaDB");
        return telefonoMapperMaria.fromAdapterToDomain(telefonoRepositoryMaria.save(telefonoMapperMaria.fromDomainToAdapter(phone)));
    }
    @Override
    public Boolean delete(String identification) {
        log.debug("Into delete on Adapter MariaDB");
        telefonoRepositoryMaria.deleteById(identification);
        return telefonoRepositoryMaria.findById(identification).isEmpty();
    }
    @Override
    public Phone findById(String identification) {
        log.debug("Into findById on Adapter MariaDB");
        if (telefonoRepositoryMaria.findById(identification).isEmpty()) {
            return null;
        } else {
            return telefonoMapperMaria.fromAdapterToDomain(telefonoRepositoryMaria.findById(identification).get());
        }
    }
    @Override
    public List<Phone> find() {
        log.debug("Into find on Adapter MariaDB");
        return telefonoRepositoryMaria.findAll().stream().map(telefonoMapperMaria::fromAdapterToDomain).collect(Collectors.toList());
    }
}

```

El adaptador de salida TelefonoOutputAdapterMaria es una implementación clave dentro de la arquitectura hexagonal, ya que establece la conexión entre la lógica de negocio y la base de datos MariaDB. Al implementar la interfaz PhoneOutputPort, este adaptador permite que el sistema interactúe con la infraestructura de manera desacoplada, lo que es fundamental para mantener la flexibilidad y la mantenibilidad de la aplicación. Este enfoque garantiza que los cambios en la infraestructura, como la migración a otra base de datos, no afecten la lógica de negocio.

Implementación Adaptador de Salida Mongo DB para la Entidad Profesión

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio Mongo y el desarrollo del adaptador de salida para la entidad Teléfono.

```
package co.edu.javeriana.as.personapp.mongo.repository;

import ...

2 usages
public interface TelefonoRepositoryMongo extends MongoRepository<TelefonoDocument, String> {
```

La interfaz TelefonoRepositoryMongo es un repositorio que gestiona la persistencia de la entidad TelefonoDocument en una base de datos MongoDB. Al extender MongoRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar teléfonos. Parametrizada con TelefonoDocument e Integer, esta interfaz encapsula la lógica relacionada con un teléfono y su identificador único. Proporciona una capa de abstracción que facilita la interacción con la base de datos y permite que la lógica de negocio opere sin depender de los detalles de implementación de la infraestructura, favoreciendo un diseño más limpio y mantenible.

```

@Slf4j
@Adapter("phoneOutputAdapterMaria")
@Transactional
public class PhoneOutputAdapterMaria implements PhoneOutputPort {
    @Autowired
    private TelefonoRepositoryMaria telefonoRepositoryMaria;
    @Autowired
    private TelefonoMapperMaria telefonoMapperMaria;
    @Override
    public Phone save(Phone phone) {
        log.debug("Into save on Adapter MariaDB");
        return telefonoMapperMaria.fromAdapterToDomain(telefonoRepositoryMaria.save(telefonoMapperMaria.fromDomainToAdapter(phone)));
    }
    @Override
    public Boolean delete(String identification) {
        log.debug("Into delete on Adapter MariaDB");
        telefonoRepositoryMaria.deleteById(identification);
        return telefonoRepositoryMaria.findById(identification).isEmpty();
    }
    @Override
    public Phone findById(String identification) {
        log.debug("Into findById on Adapter MariaDB");
        if (telefonoRepositoryMaria.findById(identification).isEmpty()) {
            return null;
        } else {
            return telefonoMapperMaria.fromAdapterToDomain(telefonoRepositoryMaria.findById(identification).get());
        }
    }
    @Override
    public List<Phone> find() {
        log.debug("Into find on Adapter MariaDB");
        return telefonoRepositoryMaria.findAll().stream().map(telefonoMapperMaria::fromAdapterToDomain).collect(Collectors.toList());
    }
}

```

El adaptador de salida TelefonoOutputAdapterMongo es esencial en la arquitectura hexagonal, ya que conecta la lógica de negocio con la base de datos MongoDB mediante la implementación de la interfaz PhoneOutputPort, lo que permite una interacción desacoplada y flexible con la infraestructura.

Implementación Use Case Entidad Teléfono: Módulo Application

```

@Slf4j
@UseCase
public class PhoneUseCase implements PhoneInputPort {
    11 usages
    private PhoneOutputPort phonePersistence;
    3 usages
    private PersonOutputPort personPersistence;
    public PhoneUseCase(@Qualifier("phoneOutputAdapterMaria") PhoneOutputPort phonePersistence,
                        @Qualifier("personOutputAdapterMaria") PersonOutputPort personPersistence) {
        this.phonePersistence = phonePersistence;
        this.personPersistence = personPersistence;
    }
    no usages
    @Override
    public void setPersistence(PhoneOutputPort phonePersistence, PersonOutputPort personPersistence) {
        this.phonePersistence = phonePersistence;
        this.personPersistence = personPersistence;
    }
    @Override
    public Phone create(Phone phone, int ccPerson) throws NoExistException {
        log.debug("Into create on Application Domain");
        Person person = personPersistence.findById(ccPerson);
        if (person == null) {
            log.error("The person with id " + ccPerson + " does not exist into db, cannot be created");
            throw new NoExistException("The person with id " + ccPerson + " does not exist into db, cannot be created");
        }
        phone.setOwner(person);
        return phonePersistence.save(phone);
    }
    no usages
    @Override
    public Phone edit(String identification, Phone phone, int ccPerson) throws NoExistException {
        Phone oldPhone = phonePersistence.findById(identification);
        if (oldPhone != null)
            return phonePersistence.save(phone);
        throw new NoExistException(
            "The phone with id " + identification + " does not exist into db, cannot be edited");
    }
    no usages
}

```

```

@Override
public Boolean drop(String identification) throws NoExistException {
    Phone oldPhone = phonePersistence.findById(identification);
    if (oldPhone != null)
        return phonePersistence.delete(identification);
    throw new NoExistException(
        "The phone with id " + identification + " does not exist into db, cannot be dropped");
}

no usages
@Override
public Phone findOne(String identification) throws NoExistException {
    Phone oldPhone = phonePersistence.findById(identification);
    if (oldPhone != null)
        return oldPhone;
    throw new NoExistException(
        "The phone with id " + identification + " does not exist into db, cannot be found");
}

no usages
@Override
public Integer count() { return phonePersistence.find().size(); }

@Override
public List<Phone> findAll() {
    log.info("Output: " + phonePersistence.getClass());
    return phonePersistence.find();
}

```

El PhoneUseCase es una implementación clave dentro de la arquitectura hexagonal, ya que actúa como el puerto de entrada que permite la interacción con la lógica de negocio relacionada con los teléfonos. Al implementar la interfaz PhoneInputPort, este caso de uso permite que la aplicación se comunique de manera estructurada con los adaptadores de salida, como TelefonoOutputAdapterMaria.

Documentación Desarrollo del Estilo Arquitectónico Hexagonal - Entidad *Estudios*

Creación del Puerto de Salida (StudyOutputPort); Módulo Application:

```

@Port
public interface StudyOutputPort {
    2 implementations
    public Study save(Study study);
    2 implementations
    public Boolean delete(Integer identification, Integer idPerson);
    2 implementations
    public Study findById(Integer identification, Integer idPerson);
    2 implementations
    public List<Study> find();
}

```

La interfaz StudyOutputPort define los métodos esenciales para la gestión de estudios en el sistema, alineándose con la arquitectura hexagonal. Su propósito es facilitar la interacción con la persistencia de datos, garantizando una separación de responsabilidades.

Creación del Puerto de Entrada (StudyInputPort): Módulo Application:

```
@Port
public interface StudyInputPort {
    no usages 1 implementation
    public void setPersistence(StudyOutputPort studyPersistence);
    1 implementation
    public Study create(Study study, int ccPerson, int idProfession) throws NoExistException;
    no usages 1 implementation
    public Study edit(Integer identification, Integer user_identificacion, Study study) throws NoExistException;
    no usages 1 implementation
    public Boolean drop(Integer identification, Integer user_identificacion) throws NoExistException;
    no usages 1 implementation
    public Study findOne(Integer identification, Integer user_identificacion) throws NoExistException;
    1 implementation
    public List<Study> findAll();
    no usages 1 implementation
    public Integer count();
}
```

La interfaz StudyInputPort define los métodos necesarios para la gestión de estudios en el sistema, alineándose con los principios de la arquitectura hexagonal. Anotada con @Port, permite la inyección de dependencias, asegurando que la lógica de negocio esté desacoplada de la infraestructura.

Nota Importante:

Antes de implementar el UseCase para la entidad Estudio, se llevarán a cabo las implementaciones de los adaptadores de salida para los dos sistemas de bases de datos (MariaDB y MongoDB). De este modo, el UseCase al que se conectan los adaptadores de entrada podrá realizar las operaciones correspondientes sobre las entidades del dominio.

Implementación Adaptador de Salida MariaDB para la Entidad Estudio

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio para MongoDB y el desarrollo del adaptador de salida para la entidad Estudio.

```

package co.edu.javeriana.as.personapp.mariadb.repository;

import co.edu.javeriana.as.personapp.mariadb.entity.EstudiosEntity;
import co.edu.javeriana.as.personapp.mariadb.entity.EstudiosEntityPK;
import org.springframework.data.jpa.repository.JpaRepository;

no usages
public interface EstudioRepositoryMaria extends JpaRepository<EstudiosEntity, EstudiosEntityPK> {
}

```

La interfaz EstudioRepositoryMaria es un repositorio JPA que gestiona la persistencia de la entidad EstudiosEntity en una base de datos MariaDB. Al extender JpaRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar estudios. Parametrizada con EstudiosEntity y EstudiosEntityPK, permite encapsular la lógica relacionada con un estudio y su identificador compuesto. Esta interfaz proporciona una capa de abstracción que facilita el enfoque en la lógica de negocio.

```

@Slf4j
@Adapter("studyOutputAdapterMaria")
@Transactional
public class StudyOutputAdapterMaria implements StudyOutputPort {
    @Autowired
    private EstudioRepositoryMaria estudioRepositoryMaria;
    @Autowired
    private EstudiosMapperMaria estudiosMapperMaria;
    @Override
    public Study save(Study study) {
        log.debug("Into save on Adapter MariaDB");
        EstudiosEntity persistedEstudio = estudioRepositoryMaria.save(estudiosMapperMaria.fromDomainToAdapter(study));
        return estudiosMapperMaria.fromAdapterToDomain(persistedEstudio);
    }
    @Override
    public Boolean delete(Integer identification, Integer idPerson) {
        log.debug("Into delete on Adapter MariaDB");
        EstudiosEntityPK estudiosEntityPK = new EstudiosEntityPK();
        estudiosEntityPK.setIdProf(identification);
        estudiosEntityPK.setCcPerIdPerson();
        estudioRepositoryMaria.deleteById(estudiosEntityPK);
        return estudioRepositoryMaria.findById(estudiosEntityPK).isEmpty();
    }
    @Override
    public Study findById(Integer identification, Integer idPerson) {
        log.debug("Into findById on Adapter MariaDB");
        EstudiosEntityPK estudiosEntityPK = new EstudiosEntityPK();
        estudiosEntityPK.setIdProf(identification);
        estudiosEntityPK.setCcPerIdPerson();
        if (estudioRepositoryMaria.findById(estudiosEntityPK).isEmpty()) {
            return null;
        } else {
            return estudiosMapperMaria.fromAdapterToDomain(estudioRepositoryMaria.findById(estudiosEntityPK).get());
        }
    }
    @Override
    public List<Study> find() {
        log.debug("Into find on Adapter MariaDB");
        return estudioRepositoryMaria.findAll().stream().map(estudiosMapperMaria::fromAdapterToDomain).collect(Collectors.toList());
    }
}

```

El adaptador de salida EstudioOutputAdapterMaria es una implementación clave dentro de la arquitectura hexagonal, ya que establece la conexión entre la lógica de negocio y la base de datos MariaDB. Al implementar la interfaz StudyOutputPort, este adaptador permite que el sistema interactúe con la infraestructura de manera desacoplada, lo que es fundamental para mantener la flexibilidad y la mantenibilidad de la aplicación. Este enfoque garantiza que los cambios en la infraestructura, como la migración a otra base de datos, no afecten la lógica de negocio.

Implementación Adaptador de Salida Mongo DB para la Entidad Profesión

Considerando que las entidades y su mapeo con el módulo de dominio ya están implementados dentro de la arquitectura hexagonal, es necesario llevar a cabo dos acciones específicas debido a la naturaleza de la base de datos: la creación del repositorio Mongo y el desarrollo del adaptador de salida para la entidad Estudio.

```
package co.edu.javeriana.as.personapp.mongo.repository;

import co.edu.javeriana.as.personapp.mongo.document.EstudiosDocument;
import org.springframework.data.mongodb.repository.MongoRepository;

no usages
public interface EstudioRepositoryMongo extends MongoRepository<EstudiosDocument, String> { }
```

La interfaz EstudioRepositoryMongo es un repositorio que gestiona la persistencia de la entidad EstudiosEntity en una base de datos MongoDB. Al extender la interfaz MongoRepository, hereda métodos que simplifican operaciones como guardar, eliminar y buscar estudios. Parametrizada con EstudiosEntity y EstudiosEntityPK, permite encapsular la lógica relacionada con un estudio y su identificador compuesto. Esta interfaz proporciona una capa de abstracción que facilita el enfoque en la lógica de negocio, permitiendo realizar operaciones de manera más eficiente y sencilla en el contexto de una base de datos no relacional.

```

@Slf4j
@Adapter("studyOutputAdapterMongo")
public class StudyOutputAdapterMongo implements StudyOutputPort {
    @Autowired
    private EstudioRepositoryMongo estudioRepositoryMongo;
    @Autowired
    private EstudiosMapperMongo estudiosMapperMongo;

    @Autowired
    private PersonaRepositoryMongo personaRepositoryMongo;
    @Autowired
    private PersonaMapperMongo personaMapperMongo;

    @Autowired
    private ProfesionRepositoryMongo profesionRepositoryMongo;
    @Autowired
    private ProfesionMapperMongo profesionMapperMongo;

    @Override
    public Study save(Study study) {
        log.debug("Into save on Adapter MongoDB");
        try{
            EstudiosDocument persistedEstudio = estudioRepositoryMongo.save(estudiosMapperMongo.fromDomainToAdapter(study));
            return estudiosMapperMongo.fromAdapterToDomain(persistedEstudio);
        } catch (MongoWriteException e) {
            log.warn(e.getMessage());
            return study;
        }
    }
}

```

```

@Override
public Boolean delete (Integer identification, Integer idPerson) {
    log.debug("Into delete on Adapter MongoDB");
    Optional<PersonaDocument> persona = personaRepositoryMongo.findById(idPerson);
    Optional<ProfesionDocument> profesion = profesionRepositoryMongo.findById(identification);

    if (persona.isPresent() && profesion.isPresent()) {
        EstudiosDocument estudio = estudioRepositoryMongo.findByPrimaryPersonaAndPrimaryProfesion(persona.get(), profesion.get().orElse(null));
        if (estudio != null) {
            estudioRepositoryMongo.delete(estudio);
            return true;
        }
    }
    return false;
}

@Override
public Study findById(Integer identification, Integer idPerson) {
    log.debug("Into findById on Adapter MongoDB");
    Optional<PersonaDocument> persona = personaRepositoryMongo.findById(idPerson);
    Optional<ProfesionDocument> profesion = profesionRepositoryMongo.findById(identification);
    if (persona.isPresent() && profesion.isPresent()) {
        EstudiosDocument estudio = estudioRepositoryMongo.findByPrimaryPersonaAndPrimaryProfesion(persona.get(), profesion.get().orElse(null));
        if (estudio != null) {
            return estudiosMapperMongo.fromAdapterToDomain(estudio);
        }
    }
    return null;
}

@Override
public List<Study> find() {
    log.debug("Into find on Adapter MongoDB");
    return estudioRepositoryMongo.findAll().stream().map(estudiosMapperMongo::fromAdapterToDomainBasic).collect(Collectors.toList());
}
}

```

El adaptador de salida StudyOutputAdapterMongo es esencial en la arquitectura hexagonal, ya que conecta la lógica de negocio con la base de datos MongoDB mediante la implementación de la

interfaz StudyOutputPort, lo que permite una interacción desacoplada y flexible con la infraestructura. Anotado con @Adapter y utilizando @Autowired para inyectar dependencias como EstudioRepositoryMongo y EstudiosMapperMongo.

Implementación Use Case Entidad Estudio: Módulo Application

```

@SLF4J
@UseCase
public class StudyUseCase implements StudyInputPort {
    11 usages
    private StudyOutputPort studyPersistence;
    2 usages
    private PersonOutputPort personPersistence;
    2 usages
    private ProfessionOutputPort professionPersistence;

    public StudyUseCase(@Qualifier("studyOutputAdapterMaria") StudyOutputPort studyPersistence,
                        @Qualifier("personOutputAdapterMaria") PersonOutputPort personPersistence,
                        @Qualifier("professionOutputAdapterMaria") ProfessionOutputPort professionPersistence) {
        this.studyPersistence = studyPersistence;
        this.personPersistence = personPersistence;
        this.professionPersistence = professionPersistence;
    }

    no usages
    @Override
    public void setPersistence(StudyOutputPort studyPersistence) { this.studyPersistence = studyPersistence; }
    @Override
    public Study create(Study study, int ccPerson, int idProfession) throws NoExistException {
        log.debug("Info create on Application Domain");
        Person person = personPersistence.findById(ccPerson);
        Profession profession = professionPersistence.findById(idProfession);
        if (person == null) {
            log.error("The person with id " + ccPerson + " does not exist into db, cannot be created");
            throw new NoExistException("The person with id " + ccPerson + " does not exist into db, cannot be created");
        }
        if (profession == null) {
            log.error("The profession with id " + idProfession + " does not exist into db, cannot be created");
            throw new NoExistException("The profession with id " + idProfession + " does not exist into db, cannot be created");
        }
        study.setPerson(person);
        study.setProfession(profession);
        return studyPersistence.save(study);
    }
}

```

```

@Override
public Study edit(Integer identification, Integer user_identificacion, Study study) {
    Study oldStudy = studyPersistence.findById(identification, user_identificacion);
    if (oldStudy != null)
        return studyPersistence.save(study);
    return null;
}

no usages
@Override
public Boolean drop(Integer identification, Integer user_identificacion) {
    Study oldStudy = studyPersistence.findById(identification, user_identificacion);
    if (oldStudy != null)
        return studyPersistence.delete(identification, user_identificacion);
    return false;
}

no usages
@Override
public Study findOne(Integer identification, Integer user_identificacion) {
    Study oldStudy = studyPersistence.findById(identification, user_identificacion);
    if (oldStudy != null)
        return oldStudy;
    return null;
}

no usages
@Override
public Integer count() { return studyPersistence.find().size(); }

@Override
public List<Study> findAll() {
    log.info("Output: " + studyPersistence.getClass());
    return studyPersistence.find();
}
}

```

El StudyUseCase es una implementación clave dentro de la arquitectura hexagonal, ya que actúa como el puerto de entrada que permite la interacción con la lógica de negocio relacionada con los estudios. Al implementar la interfaz StudyInputPort, este caso de uso facilita la comunicación estructurada de la aplicación con los adaptadores de salida, como StudyOutputAdapterMaria. La clase utiliza la anotación @UseCase, lo que indica su rol dentro del sistema.

Documentación Adaptadores de Entrada

Cli-input-adapter

El cli-input-adapter actúa como un puerto de entrada esencial dentro de la arquitectura hexagonal, permitiendo la interacción entre el usuario y el sistema de forma estructurada y modular. Su implementación facilita la ejecución de funcionalidades del sistema desde la línea de comandos, asegurando una separación clara entre la lógica de negocio y la capa de presentación. Este adaptador se conecta con los puertos de entrada definidos en el dominio, de manera que las solicitudes del usuario puedan ser procesadas a través de casos de uso específicos, manteniendo así la independencia y flexibilidad de los componentes internos.

***Desarrollo* Modelo de Entidades CLI:**

El primer paso en el desarrollo consiste en implementar todas las entidades que conforman el módulo, siguiendo una estructura modular que permita su fácil mantenimiento y evolución que sea coherente con el modelo del dominio.

Entidad Persona:

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class PersonaModelCli {  
    private Integer cc;  
    private String nombre;  
    private String apellido;  
    private String genero;  
    private Integer edad;  
}
```

Entidad Profesión:

```
package co.edu.javeriana.as.personapp.terminal.model;  
import ...  
  
14 usages  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class ProfesionModelCli {  
    private Integer id;  
    private String name;  
    private String description;  
}
```

Entidad Teléfono:

```
11 usages
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class TelefonoModelCli {
    private String num;
    private String oper;
    private PersonaModelCli dueño;

    @Override
    public String toString() {
        return "TelefonoModelCli [num=" + num + ", oper=" + oper + ", dueño=" +
            getDueño().getNombre() + " " + getDueño().getApellido() + "]";
    }
}
```

Entidad Estudio:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class EstudioModelCli {
    private Person person;
    private Profession profession;
    private LocalDate graduationDate;
    private String universityName;

    @Override
    public String toString() {
        return "EstudioModelCli [person=" + person.getFirstName() + " " + person.getLastName() + ", " +
            "profession=" + profession.getName() + ", graduationDate=" + graduationDate +
            ", universityName=" + universityName + "]";
    }
}
```

Desarrollo Mappers CLI:

Cada entidad necesita un mapper que gestione la transformación precisa entre las entidades del dominio y las propias del módulo, asegurando que los datos fluyan de manera coherente y efectiva a través de la arquitectura. Estos mapeadores permiten que las entidades del dominio, que representan la lógica de negocio central, se adapten fácilmente a las estructuras específicas de cada módulo, sin comprometer la integridad del modelo ni introducir dependencias innecesarias.

Al contar con un mapper dedicado por entidad, se establece una capa de interoperabilidad que facilita la traducción de datos entre el dominio y las interfaces externas, preservando la independencia de la lógica de negocio frente a detalles de implementación.

A continuación se adjunta la evidencia de la implementación de los mappers para las entidades: Persona, profesión, teléfono y estudio.

Mapper PersonaCli:

```
@Mapper
public class PersonaMapperCli {

    10 usages
    public PersonaModelCli fromDomainToBasicModelCli(Person person) {
        return person != null ? PersonaModelCli.builder()
            .cc(person.getIdentification())
            .nombre(person.getFirstName())
            .apellido(person.getLastName())
            .build() : null;
    }

    7 usages
    public Person fromBasicModelCliToDomain(PersonaModelCli personaModelCli) {
        return personaModelCli != null ? Person.builder()
            .identification(personaModelCli.getCc())
            .firstName(personaModelCli.getNombre())
            .lastName(personaModelCli.getApellido())
            .gender(personaModelCli.getGenero() != null ? parseGender(personaModelCli.getGenero()) : Gender.OTHER)
            .build() : null;
    }

    1 usage
    private Gender parseGender(String genero) {
        switch (genero.toUpperCase()) {
            case "M":
                return Gender.MALE;
            case "F":
                return Gender.FEMALE;
            default:
                return Gender.OTHER;
        }
    }
}
```

Mapper ProfesiónCli:

```
@Mapper
public class ProfesionMapperCli {
    6 usages
    public ProfesionModelCli fromDomainToBasicModelCli(Profession profession) {
        return profession != null ? ProfesionModelCli.builder()
            .id(profession.getIdentification())
            .name(profession.getName())
            .description(profession.getDescription())
            .build() : null;
    }
    6 usages
    public Profession fromBasicModelCliToDomain(ProfesionModelCli profesionModelCli) {
        return profesionModelCli != null ? Profession.builder()
            .identification(profesionModelCli.getId())
            .name(profesionModelCli.getName())
            .description(profesionModelCli.getDescription())
            .build() : null;
    }
}
```

Mapper EstudioCli:

```
@Mapper
public class EstudioMapperCli {
    @Autowired
    private PersonaMapperCli personaMapperCli;

    @Autowired
    private ProfesionMapperCli profesionMapperCli;
    1 usage
    public EstudioModelCli fromDomainToCli(Study study) {
        return EstudioModelCli.builder()
            .person(personaMapperCli.fromBasicModelCliToDomain(personaMapperCli.fromDomainToBasicModelCli(study.getPerson())))
            .profession(profesionMapperCli.fromBasicModelCliToDomain(profesionMapperCli.fromDomainToBasicModelCli(study.getProfession())))
            .graduationDate(study.getGraduationDate())
            .universityName(study.getUniversityName())
            .build();
    }

    2 usages
    public Study fromCliToDomain(EstudioModelCli estudioModelCli) {
        // Se asignan valores solo si están presentes en el modelo CLI
        Person person = estudioModelCli.getPerson() != null
            ? personaMapperCli.fromBasicModelCliToDomain(personaMapperCli.fromDomainToBasicModelCli(estudioModelCli.getPerson()))
            : null;

        Profession profession = estudioModelCli.getProfession() != null
            ? profesionMapperCli.fromBasicModelCliToDomain(profesionMapperCli.fromDomainToBasicModelCli(estudioModelCli.getProfession()))
            : null;

        return Study.builder()
            .person(person)
            .profession(profession)
            .graduationDate(estudioModelCli.getGraduationDate())
            .universityName(estudioModelCli.getUniversityName())
            .build();
    }
}
```

Mapper TelefonoCli:

```
@Mapper
public class TelefonoMapperCli {
    @Autowired
    private PersonaMapperCli personaMapperCli;
    3 usages
    public TelefonoModelCli fromDomainToCli(Phone phone) {
        return TelefonoModelCli.builder()
            .num(phone.getNumber())
            .oper(phone.getCompany())
            .duenio(personaMapperCli.fromDomainToBasicModelCli(phone.getOwner())) // Mapeo básico
            .build();
    }
    2 usages
    public Phone fromCliToDomain(TelefonoModelCli telefonoModelCli) {
        Person owner = telefonoModelCli.getDuenio() != null
            ? personaMapperCli.fromBasicModelCliToDomain(telefonoModelCli.getDuenio())
            : new Person();
        return Phone.builder()
            .number(telefonoModelCli.getNum())
            .company(telefonoModelCli.getOper())
            .owner(owner)
            .build();
    }
}
```

Desarrollo Adapters CLI:

Los *input adapters* en esta arquitectura hexagonal implementan los puertos de entrada definidos en el dominio para recibir y procesar las solicitudes externas, permitiendo la interacción controlada entre el usuario y el núcleo de la aplicación. En el caso de los *input adapters* CLI, su rol es traducir las acciones del usuario en comandos que el sistema pueda interpretar y ejecutar, todo sin comprometer la lógica de negocio subyacente. Cada *input adapter* está estructurado para gestionar las entidades relevantes de su módulo como Person, Profession, Phone o Study y se conecta con los puertos de entrada necesarios para invocar casos de uso específicos.

Cada adaptador configura los puertos de entrada y salida de forma dinámica, permitiendo elegir entre distintas implementaciones de persistencia (por ejemplo, MariaDB o MongoDB) mediante el método setOutputPortInjection.

Este mecanismo asegura que el adaptador se mantenga independiente de los detalles de infraestructura, pudiendo conectarse a diferentes bases de datos sin necesidad de modificar la lógica del adaptador. Además, los *input adapters* utilizan mappers para convertir las entidades del dominio a sus correspondientes modelos de CLI, y viceversa.

InputAdapter Persona:

```
@Adapter
public class PersonaInputAdapterCli {
    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;
    @Autowired
    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    private PersonaMapperCli personaMapperCli;
    8 usages
    PersonInputPort personInputPort;
    2 usages
    public void setPersonOutputPortInjection(String dbOption) throws InvalidOptionException {
        if (dbOption.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            personInputPort = new PersonUseCase(personOutputPortMaria);
        } else if (dbOption.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
            personInputPort = new PersonUseCase(personOutputPortMongo);
        } else {
            throw new InvalidOptionException("Invalid database option: " + dbOption);
        }
    }
    1 usage
    public void historial() {
        log.info("Into historial PersonaEntity in Input Adapter");
        personInputPort.findAll().stream() Stream<Person>
            .map(personMapperCli::fromDomainToBasicModelCli) Stream<PersonaModelCli>
            .forEach(System.out::println);
    }

    public void create (int cc, String nombre, String apellido, String genero, Integer edad){
        PersonaModelCli persona = PersonaModelCli.builder()
            .cc(cc)
            .nombre(nombre)
            .apellido(apellido)
            .genero(genero)
            .edad(edad)
            .build();
        personInputPort.create(personaMapperCli.fromBasicModelCliToDomain(persona));
    }
}
```

```
    public void edit (int cc, String nombre, String apellido, String genero, Integer edad) throws NoExistException {
        PersonaModelCli persona = PersonaModelCli.builder()
            .cc(cc)
            .nombre(nombre)
            .apellido(apellido)
            .genero(genero)
            .edad(edad)
            .build();
        personInputPort.edit(cc, personaMapperCli.fromBasicModelCliToDomain(persona));
    }
    1 usage
    public void drop (int cc) throws NoExistException {
        personInputPort.drop(cc);
    }
    1 usage
    public void findOne (int cc) throws NoExistException {
        PersonaModelCli persona = personaMapperCli.fromDomainToBasicModelCli(personInputPort.findOne(cc));
        if (persona != null) {
            System.out.println(persona.toString());
        }
    }
    1 usage
    public void count () { System.out.println(personInputPort.count()); }
}
```

InputAdapter Teléfono:

```
@Slf4j
@Adapter
public class TelefonoInputAdapterCli {
    @Autowired
    @Qualifier("phoneOutputAdapterMaria")
    private PhoneOutputPort phoneOutputPortMaria;

    @Autowired
    @Qualifier("phoneOutputAdapterMongo")
    private PhoneOutputPort phoneOutputPortMongo;

    @Autowired
    private TelefonoMapperCli telefonoMapperCli;
    9 usages
    private PhoneInputPort phoneInputPort;

    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;

    @Autowired
    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    private PersonaMapperCli personaMapperCli;
    4 usages
    private PersonInputPort personInputPort;
    2 usages
    public void setPhoneOutputPortInjection(String dbOption) {
        if (dbOption.equalsIgnoreCase("MARIA")) {
            this.phoneInputPort = new PhoneUseCase(phoneOutputPortMaria, personOutputPortMaria);
            this.personInputPort = new PersonUseCase(personOutputPortMaria);
        } else if (dbOption.equalsIgnoreCase("MONGO")) {
            this.phoneInputPort = new PhoneUseCase(phoneOutputPortMongo, personOutputPortMongo);
            this.personInputPort = new PersonUseCase(personOutputPortMongo);
        } else {
            throw new IllegalArgumentException("Invalid database option: " + dbOption);
        }
    }
}
```

```
public void create(String number, String company, int ownerId) throws NoExistException {
    PersonaModelCli ownerModel = personaMapperCli.fromDomainToBasicModelCli(personInputPort.findOne(ownerId));
    if (ownerModel == null) {
        throw new NoExistException("The owner with id " + ownerId + " does not exist in the database, cannot create phone.");
    }
    TelefonoModelcli telefonoModel = TelefonoModelCli.builder().num(number).oper(company).duenio(ownerModel).build();
    phoneInputPort.create(telefonoMapperCli.fromCliToDomain(telefonoModel), ownerId);
}

1 usage
public void drop(String number) throws NoExistException {
    TelefonoModelcli telefonoModel = telefonoMapperCli.fromDomainToCli(phoneInputPort.findOne(number));
    if (telefonoModel == null) {
        throw new NoExistException("The phone with number " + number + " does not exist in the database, cannot be deleted.");
    }
    phoneInputPort.drop(number);
    System.out.println("Teléfono eliminado con éxito.");
}

1 usage
public void edit(String number, String company, int ownerId) throws NoExistException {
    PersonaModelCli ownerModel = personaMapperCli.fromDomainToBasicModelCli(personInputPort.findOne(ownerId));
    if (ownerModel == null) {
        throw new NoExistException("The owner with id " + ownerId + " does not exist in the database, cannot edit phone.");
    }
    TelefonoModelcli telefonoModel = TelefonoModelCli.builder().num(number).oper(company).duenio(ownerModel).build();
    phoneInputPort.edit(number, telefonoMapperCli.fromCliToDomain(telefonoModel), ownerId);
    System.out.println("Teléfono editado con éxito.");
}
```

```

public void findOne(String number) throws NoExistException {
    TelefonoModelCli telefonoModel = telefonoMapperCli.fromDomainToCli(phoneInputPort.findOne(number));
    if (telefonoModel == null) {
        throw new NoExistException("The phone with number " + number + " does not exist in the database, cannot be found.");
    }
    System.out.println(telefonoModel);
}
1 usage
public void count() { System.out.println(phoneInputPort.count()); }
}

```

InputAdapter Profesión:

```

@Slf4j
@Adapter
public class ProfesionInputAdapterCli {
    @Autowired
    @Qualifier("profesionOutPutAdapterMaria")
    private ProfessionOutputPort professionOutputPortMaria;

    @Autowired
    @Qualifier("profesionOutPutAdapterMongo")
    private ProfessionOutputPort professionOutputPortMongo;

    @Autowired
    private ProfesionMapperCli profesionMapperCli;
    9 usages
    private ProfessionInputPort professionInputPort;

    2 usages
    public void setProfessionOutputPortInjection (String dbOptions) throws InvalidOptionException {
        if (dbOptions.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            professionInputPort = new ProfessionUseCase(professionOutputPortMaria);
        } else if (dbOptions.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
            professionInputPort = new ProfessionUseCase(professionOutputPortMongo);
        } else {
            throw new InvalidOptionException("Invalid database option: " + dbOptions);
        }
    }
    1 usage
    public void historial() {
        log.info("Into historial ProfesionEntity in Input Adapter");
        professionInputPort.findAll().stream() Stream<Profession>
            .map(profesionMapperCli::fromDomainToBasicModelCli) Stream<ProfesionModelCli>
            .forEach(System.out::println);
    }
    public void create (int id, String name, String description) {
        log.info("Into create ProfesionEntity in Input Adapter");
        ProfesionModelCli profesionModelCli = new ProfesionModelCli();
        profesionModelCli.setId(id);
        profesionModelCli.setDescription(description);
        profesionModelCli.setName(name);
        professionInputPort.create(profesionMapperCli.fromBasicModelCliToDomain(profesionModelCli));
    }
}

```

```

public void drop (int id) throws NoExistException {
    Optional.ofNullable(professionInputPort.findOne(id))
        .orElseThrow(() -> new NoExistException("The profession with id " + id + " does not exist into db, cannot be deleted"));
    professionInputPort.drop(id);
    System.out.println("Profesión eliminada con éxito.");
}
1 usage
public void edit (int id, String name, String description) throws NoExistException {
    log.info("Into edit ProfesionEntity in Input Adapter");
    ProfesionModelCli profesionModelCli = new ProfesionModelCli();
    profesionModelCli.setId(id);
    profesionModelCli.setDescription(description);
    profesionModelCli.setName(name);
    professionInputPort.edit(id, profesionMapperCli.fromBasicModelCliToDomain(profesionModelCli));
    System.out.println("Profesión editada con éxito.");
}
1 usage
public void findOne (int id) throws NoExistException {
    log.info("Into findOne ProfesionEntity in Input Adapter");
    ProfesionModelCli profesionModelCli = profesionMapperCli.fromDomainToBasicModelCli(professionInputPort.findOne(id));
    if (profesionModelCli == null) {
        throw new NoExistException("The profession with id " + id + " does not exist into db, cannot be found");
    }
    System.out.println(profesionModelCli.toString());
}
1 usage
public void count () { System.out.println(professionInputPort.count()); }
}

```

InputAdapter Estudio:

```
@Slf4j
@Adapter
public class EstudioInputAdapterCli {
    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;

    @Autowired
    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    private PersonaMapperCli personaMapperCli;
    4 usages
    private PersonInputPort personInputPort;

    @Autowired
    @Qualifier("professionOutputAdapterMaria")
    private ProfessionOutputPort professionOutputPortMaria;

    @Autowired
    @Qualifier("professionOutputAdapterMongo")
    private ProfessionOutputPort professionOutputPortMongo;

    @Autowired
    private ProfesionMapperCli profesionMapperCli;
    4 usages
    private ProfessionInputPort professionInputPort;

    @Autowired
    @Qualifier("studyOutputAdapterMaria")
    private StudyOutputPort studyOutputPortMaria;

    @Autowired
    @Qualifier("studyOutputAdapterMongo")
    private StudyOutputPort studyOutputPortMongo;

    @Autowired
    private EstudioMapperCli estudioMapperCli;
    10 usages
    private StudyInputPort studyInputPort;
```

```
public void setStudyOutputPortInjection (String dbOption){
    if (dbOption.equalsIgnoreCase (anotherString: "MARIA")) {
        this.studyInputPort = new StudyUseCase(studyOutputPortMaria, personOutputPortMaria, professionOutputPortMaria);
        this.personInputPort = new PersonUseCase(personOutputPortMaria);
        this.professionInputPort = new ProfessionUseCase(professionOutputPortMaria);
    } else if (dbOption.equalsIgnoreCase (anotherString: "MONGO")) {
        this.studyInputPort = new StudyUseCase(studyOutputPortMongo, personOutputPortMongo, professionOutputPortMongo);
        this.personInputPort = new PersonUseCase(personOutputPortMongo);
        this.professionInputPort = new ProfessionUseCase(professionOutputPortMongo);
    } else {
        throw new IllegalArgumentException("Invalid database option: " + dbOption);
    }
}

1 usage
public void historial(){
    log.info("Into historial Estudio Entity in Input Adapter");
    studyInputPort.findAll().stream() Stream<Study>
        .map(estudioMapperCli::fromDomainToCli) Stream<EstudioModelCli>
        .forEach(System.out::println);
}

public void create (int ccPerson, int idProf, String college, LocalDate date) throws NoExistException{
    PersonaModelCli personModel = personaMapperCli.fromDomainToBasicModelCli(personInputPort.findOne(ccPerson));
    ProfesionModelCli professionModel = profesionMapperCli.fromDomainToBasicModelCli(professionInputPort.findOne(idProf));

    if (personModel == null)
        throw new NoExistException("The person with id " + ccPerson + " does not exist in the database, cannot create study.");

    if (professionModel == null)
        throw new NoExistException("The profession with id " + idProf + " does not exist in the database, cannot create study.");

    // Create cli with the person and profession
    EstudioModelCli estudioModelCli = EstudioModelCli.builder()
        .person(personaMapperCli.fromBasicModelCliToDomain(personModel))
        .profession(profesionMapperCli.fromBasicModelCliToDomain(professionModel))
        .graduationDate(date)
        .universityName(college)
        .build();
    studyInputPort.create(estudioMapperCli.fromCliToDomain(estudioModelCli), ccPerson, idProf);
}
```

```

public void drop (int id, int ccPerson) throws NoExistException{
    Optional.ofNullable(studyInputPort.findOne(id, ccPerson))
        .orElseThrow(() -> new NoExistException("The study with id " + id + " does not exist in the database, cannot drop study."));
    studyInputPort.drop(id, ccPerson);
    System.out.println("Estudio eliminado con éxito.");
}

1 usage
public void findOne (int id, int ccPerson) throws NoExistException{
    Optional.ofNullable(studyInputPort.findOne(id, ccPerson))
        .orElseThrow(() -> new NoExistException("The study with id " + id + " does not exist in the database, cannot find study."));
    System.out.println(studyInputPort.findOne(id, ccPerson));
}

1 usage
public void count (){
    System.out.println(studyInputPort.count());
}

1 usage
public void edit(int ccPerson, int idProf, String college, LocalDate date) throws NoExistException {
    PersonaModelCli personModel = personaMapperCli.fromDomainToBasicModelCli(personInputPort.findOne(ccPerson));
    ProfesionModelCli professionModel = profesionMapperCli.fromDomainToBasicModelCli(professionInputPort.findOne(idProf));

    if (personModel == null)
        throw new NoExistException("The person with id " + ccPerson + " does not exist in the database, cannot edit study.");

    if (professionModel == null)
        throw new NoExistException("The profession with id " + idProf + " does not exist in the database, cannot edit study.");

    EstudioModelCli estudioModelCli = EstudioModelCli.builder().person(personaMapperCli.fromBasicModelCliToDomain(personModel))
        .profession(profesionMapperCli.fromBasicModelCliToDomain(professionModel)).graduationDate(date).universityName(college)
        .build();

    Study updatedStudy = studyInputPort.edit(idProf, ccPerson, estudioMapperCli.fromCliToDomain(estudioModelCli));
    if (updatedStudy == null) {
        throw new NoExistException("Failed to update the study. No existing study found with person ID " + ccPerson + " and profession ID " + idProf);
    }
    System.out.println("Estudio actualizado con éxito.");
}
}

```

Desarrollo Menu CLI:

La estructura del sistema utiliza un menú principal como punto de acceso inicial, desde el cual el usuario puede navegar a submenús específicos de cada módulo: Persona, Profesión, Teléfono y Estudio. Este diseño facilita la selección de un módulo particular para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) y otras funciones complementarias de manera organizada y modular, manteniendo cada submenú autónomo y cohesivo en su funcionalidad.

Menú Principal

El MenuPrincipal proporciona una interfaz de selección inicial para que el usuario elija el módulo sobre el cual desea operar. Desde aquí, se puede acceder a cada submenú de forma independiente y realizar las acciones disponibles en cada módulo. El menú principal muestra opciones para cada uno de los módulos y permite seleccionar entre las opciones de persistencia disponibles (MariaDB o MongoDB) antes de iniciar el flujo específico del módulo seleccionado.

Submenús de Módulo (Persona, Profesión, Teléfono, Estudio)

Cada submenú (como EstudioMenu) opera de manera independiente y contiene opciones específicas para realizar operaciones CRUD y otras acciones en su respectivo módulo:

1. **Opciones de Persistencia:** Cada submenú permite al usuario seleccionar la opción de persistencia, configurando la base de datos (MariaDB o MongoDB) que se utilizará para ejecutar las acciones. Una vez seleccionada, el sistema direcciona las operaciones a la base de datos especificada a través del adaptador correspondiente.
2. **Opciones de Operación:**
 - **Ver Todo:** Muestra todos los registros del módulo seleccionado.
 - **Crear:** Permite la creación de un nuevo registro en el módulo, solicitando al usuario los datos específicos requeridos.
 - **Buscar:** Realiza una búsqueda de un registro específico mediante un identificador clave, como un ID o un número de teléfono.
 - **Editar:** Actualiza los datos de un registro existente, permitiendo al usuario ingresar nuevos valores.
 - **Eliminar:** Borra un registro específico del módulo.
 - **Contar:** Muestra el total de registros en el módulo seleccionado.

Este enfoque modular garantiza que cada operación se maneje dentro de su propio flujo y mantiene la independencia entre los diferentes submenús, siguiendo los principios de la arquitectura hexagonal. Los submenús se desligan del menú principal, permitiendo al usuario concentrarse en un flujo específico y simplificando la interacción mediante una estructura organizada y segmentada de acuerdo con cada funcionalidad.

Nota: La implementación del menú específica se encuentra en la ruta del repositorio: “cli-input-adapter/src/main/java/co/edu/javeriana/as/personapp/terminal/menú”

Desarrollo Adaptador de Entrada para la API REST

El rest-input-adapter actúa como un puerto de entrada clave dentro de la arquitectura hexagonal, permitiendo que las solicitudes externas interactúen con el sistema a través de una API REST estructurada y modular. Su implementación facilita la comunicación entre el cliente y el sistema, asegurando una separación clara entre la lógica de negocio y la capa de presentación. Este adaptador se conecta con los puertos de entrada definidos en el dominio, lo que permite que las solicitudes HTTP se mapeen a casos de uso específicos, procesando datos de entrada y devolviendo respuestas consistentes y bien estructuradas. Al adoptar un enfoque modular, el rest-input-adapter asegura que los componentes internos del sistema mantengan su independencia y flexibilidad, permitiendo que la lógica de negocio permanezca desacoplada de los detalles de transporte y presentación de datos.

Desarrollo Entidades de Modelo

Las entidades modelo dentro de la estructura del proyecto representan las clases fundamentales para el manejo de datos entre el cliente y la lógica de negocio, manteniendo un enfoque claro y coherente con la arquitectura hexagonal y el patrón modular. Estas entidades modelo se dividen en dos tipos principales: Request y Response, que corresponden a las estructuras de datos de entrada y salida respectivamente, facilitando la comunicación controlada entre el adaptador REST y el núcleo de la aplicación. Al definir entidades separadas para cada flujo (entrada y salida), se logra una mayor claridad y control sobre los datos que se transmiten a través de la API, evitando exponer directamente las entidades de dominio internas.

Las entidades Request como EstudioRequest, PersonaRequest, ProfesionRequest, y TelefonoRequest encapsulan la información que el usuario necesita enviar para realizar diversas operaciones, como la creación o actualización de registros. Estas entidades se enfocan en captar únicamente los datos esenciales para cada caso de uso específico, evitando sobrecargar la lógica de negocio con detalles innecesarios y promoviendo una validación centralizada de los datos. Cada Request es mapeada a una entidad de dominio correspondiente mediante mapeadores dedicados, asegurando que los datos pasen por un proceso de validación y transformación adecuado antes de ingresar a la capa de negocio.

Por su parte, las entidades Response como EstudioResponse, PersonaResponse, ProfesionResponse, y TelefonoResponse están diseñadas para estructurar las respuestas que el servidor envía de regreso al cliente. Estas clases encapsulan no solo los datos resultantes de la operación solicitada, sino también metadatos como el estado de la operación y la base de datos donde se procesó la solicitud, mejorando la trazabilidad.

Entidad Request Persona

```
package co.edu.javeriana.as.personapp.model.request;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class PersonaRequest {
    private String dni;
    private String firstName;
    private String lastName;
    private String age;
    private String sex;
    private String database;
}
```

Entidad Response Persona

```
package co.edu.javeriana.as.personapp.model.response;

import co.edu.javeriana.as.personapp.model.request.PersonaRequest;

public class PersonaResponse extends PersonaRequest{
    private String status;

    public PersonaResponse(String dni, String firstName, String lastName, String age, String sex, String database, String status) {
        super(dni, firstName, lastName, age, sex, database);
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Entidad Request Profesión

```
package co.edu.javeriana.as.personapp.model.request;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class ProfesionRequest {
    private String id;
    private String name;
    private String description;
    private String database;
}
```

Entidad Response Profesión

```
package co.edu.javeriana.as.personapp.model.response;

import co.edu.javeriana.as.personapp.model.request.ProfesionRequest;

public class ProfesionResponse extends ProfesionRequest {
    private String status;

    public ProfesionResponse(String id, String name, String description, String database, String status) {
        super(id, name, description, database);
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Entidad Request Teléfono

```
package co.edu.javeriana.as.personapp.model.request;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class TelefonoRequest {
    private String num;
    private String oper;
    private String ownerId; // Solo el ID del dueño
    private String database;
}
```

Entidad Response Teléfono

```
package co.edu.javeriana.as.personapp.model.response;

import co.edu.javeriana.as.personapp.model.request.TelefonoRequest;

public class TelefonoResponse extends TelefonoRequest {
    private String status;

    public TelefonoResponse(String num, String oper, String ownerId, String database, String status) {
        super(num, oper, ownerId, database);
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Entidad Request Estudio

```
package co.edu.javeriana.as.personapp.model.request;

import lombok.AllArgsConstructorConstructor;
import lombok.Data;
import lombok.NoArgsConstructorConstructor;

@Data
@NoArgsConstructorConstructor
@AllArgsConstructorConstructor
public class EstudioRequest {
    private String personId;
    private String professionId;
    private String graduationDate;
    private String universityName;
}
```

Entidad Response Estudio

```
package co.edu.javeriana.as.personapp.model.response;

import co.edu.javeriana.as.personapp.model.request.EstudioRequest;

public class EstudioResponse extends EstudioRequest {
    private String status;

    public EstudioResponse(String personId, String professionId, String graduationDate, String universityName, String database, String st
        super(personId, professionId, graduationDate, universityName);
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Desarrollo Mappers

Los mappers en este proyecto cumplen una función crucial dentro de la arquitectura hexagonal, facilitando la transformación de datos entre diferentes capas y asegurando que cada entidad mantenga la integridad de su representación en cada contexto. Los mappers convierten los datos desde y hacia los objetos de dominio hacia los objetos de transporte (request y response), permitiendo que las entidades de negocio, que operan en el núcleo del sistema, permanezcan independientes de las particularidades de la API REST o de cualquier otra capa de infraestructura. Al mapear los datos de entrada y salida, los mappers aseguran que la lógica de negocio no esté directamente expuesta a las estructuras de datos externas, garantizando así la modularidad y escalabilidad del sistema.

Cada mapper se especializa en una entidad, como Persona, Profesión, Teléfono y Estudio, asegurando que cada una de estas entidades se convierta de forma precisa entre los modelos de dominio y los modelos específicos de la API REST (request y response).

PersonaMapperRest

```
package co.edu.javeriana.as.personapp.mapper;

import co.edu.javeriana.as.personapp.common.annotations.Mapper;
import co.edu.javeriana.as.personapp.domain.Gender;
import co.edu.javeriana.as.personapp.domain.Person;
import co.edu.javeriana.as.personapp.model.request.PersonaRequest;
import co.edu.javeriana.as.personapp.model.response.PersonaResponse;

@Mapper
public class PersonaMapperRest {

    public PersonaResponse fromDomainToAdapterRestMaria(Person person) {
        return fromDomainToAdapterRest(person, "MariaDB");
    }

    public PersonaResponse fromDomainToAdapterRestMongo(Person person) {
        return fromDomainToAdapterRest(person, "MongoDB");
    }

    public PersonaResponse fromDomainToAdapterRest(Person person, String database) {
        return new PersonaResponse(
            person.getIdentification(),
            person.getFirstName(),
            person.getLastName(),
            person.getAge(),
            person.getGender().toString(),
            database,
            "OK");
    }

    public Person fromAdapterToDomain(PersonaRequest request) {
        return Person.builder()
            .identification(Integer.parseInt(request.getDni()))
            .firstName(request.getFirstName())
            .lastName(request.getLastName())
            .gender(Gender.valueOf(request.getSex()))
            .age(Integer.parseInt(request.getAge())).build();
    }
}
```

ProfesionMapperRest

```
package co.edu.javeriana.as.personapp.mapper;

import co.edu.javeriana.as.personapp.common.annotations.Mapper;
import co.edu.javeriana.as.personapp.domain.Profession;
import co.edu.javeriana.as.personapp.model.request.ProfesionRequest;
import co.edu.javeriana.as.personapp.model.response.ProfesionResponse;

@Mapper
public class ProfesionMapperRest {

    public ProfesionResponse fromDomainToAdapterRestMaria(Profession profession) {
        return fromDomainToAdapterRest(profession, "MariaDB");
    }

    public ProfesionResponse fromDomainToAdapterRestMongo(Profession profession) {
        return fromDomainToAdapterRest(profession, "MongoDB");
    }

    public ProfesionResponse fromDomainToAdapterRest(Profession profession, String database) {
        return new ProfesionResponse(
            String.valueOf(profession.getIdentification()),
            profession.getName(),
            profession.getDescription(),
            database,
            "OK"
        );
    }

    public Profession fromAdapterToDomain(ProfesionRequest request) {
        return Profession.builder()
            .identification(Integer.parseInt(request.getId()))
            .name(request.getName())
            .description(request.getDescription())
            .build();
    }
}
```

Telefono MapperRest

```
package co.edu.javeriana.as.personapp.mapper;

import co.edu.javeriana.as.personapp.common.annotations.Mapper;
import co.edu.javeriana.as.personapp.domain.Person;
import co.edu.javeriana.as.personapp.domain.Phone;
import co.edu.javeriana.as.personapp.model.request.TelefonoRequest;
import co.edu.javeriana.as.personapp.model.response.TelefonoResponse;

@Mapper
public class TelefonoMapperRest {

    public TelefonoResponse fromDomainToAdapterRestMaria(Phone phone) {
        return fromDomainToAdapterRest(phone, "MariaDB");
    }

    public TelefonoResponse fromDomainToAdapterRestMongo(Phone phone) {
        return fromDomainToAdapterRest(phone, "MongoDB");
    }

    public TelefonoResponse fromDomainToAdapterRest(Phone phone, String database) {
        return new TelefonoResponse(
            phone.getNumber(),
            phone.getCompany(),
            phone.getOwner() != null ? String.valueOf(phone.getOwner().getIdentification()) : null,
            database,
            "OK"
        );
    }

    public Phone fromAdapterToDomain(TelefonoRequest request, Person owner) {
        return Phone.builder()
            .number(request.getNum())
            .company(request.getOper())
            .owner(owner)
            .build();
    }
}
```

EstudioMapperRest

```
package co.edu.javeriana.as.personapp.mapper;

import co.edu.javeriana.as.personapp.common.annotations.Mapper;
import co.edu.javeriana.as.personapp.domain.Person;
import co.edu.javeriana.as.personapp.domain.Profession;
import co.edu.javeriana.as.personapp.domain.Study;
import co.edu.javeriana.as.personapp.model.request.EstudioRequest;
import co.edu.javeriana.as.personapp.model.response.EstudioResponse;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;

@Mapper
public class EstudioMapperRest {

    private static final DateTimeFormatter DATE_FORMATTER = DateTimeFormatter.ofPattern("dd-MM-yyyy");

    // Mapea desde el dominio a la respuesta para MariaDB
    public EstudioResponse fromDomainToAdapterRestMaria(Study study) {
        return fromDomainToAdapterRest(study, "MariaDB");
    }

    // Mapea desde el dominio a la respuesta para MongoDB
    public EstudioResponse fromDomainToAdapterRestMongo(Study study) {
        return fromDomainToAdapterRest(study, "MongoDB");
    }
}
```

```
// Método genérico para mapear Study a EstudioResponse
public EstudioResponse fromDomainToAdapterRest(Study study, String database) {
    if (study == null || study.getPerson() == null || study.getProfession() == null) {
        return new EstudioResponse(null, null, null, null, database, "Error: Incomplete study data");
    }

    return new EstudioResponse(
        String.valueOf(study.getPerson().getIdentification()),
        String.valueOf(study.getProfession().getIdentification()),
        study.getUniversityName(),
        study.getGraduationDate() != null ? study.getGraduationDate().format(DATE_FORMATTER) : null,
        database,
        "OK"
    );
}

// Convierte el request en un objeto de dominio study
public Study fromAdapterToDomain(EstudioRequest request, Person person, Profession profession) {
    LocalDate graduationDate;
    try {
        graduationDate = LocalDate.parse(request.getGraduationDate(), DATE_FORMATTER);
    } catch (DateTimeParseException e) {
        throw new IllegalArgumentException("Invalid date format for graduationDate. Expected format: dd-MM-yyyy");
    }

    return Study.builder()
        .person(person)
        .profession(profession)
        .universityName(request.getUniversityName())
        .graduationDate(graduationDate)
        .build();
}
```

Desarrollo Adapters

Los adapters de entrada rest cumplen un rol esencial en la arquitectura hexagonal, actuando como puntos de interacción entre las solicitudes externas y la lógica de negocio del sistema. En este caso, cada adapter de entrada, como el de Estudio, Persona, Profesión y Teléfono, está diseñado para recibir y procesar solicitudes HTTP provenientes de clientes REST y ejecutar los casos de uso

correspondientes mediante los puertos de entrada. Al recibir una solicitud, el adapter valida los datos, establece la conexión con los puertos de entrada correspondientes (que representan la lógica de negocio) y procesa la respuesta adecuada.

Cada adapter también es responsable de traducir las entidades externas (requests y responses) en entidades de dominio, utilizando mappers para transformar los datos entre las diferentes representaciones de la información. Este enfoque garantiza que el sistema mantenga una separación clara entre las estructuras de datos externas y la lógica de negocio interna, permitiendo que el núcleo de la aplicación permanezca aislado de la infraestructura de comunicación, y asegurando la modularidad y la fácil escalabilidad del proyecto.

En el contexto de los adapters de entrada REST, estos componentes no solo actúan como intermediarios, sino que también manejan errores y excepciones comunes, devolviendo respuestas significativas a los clientes, ya sea en el caso de una operación exitosa o cuando ocurre algún problema (como un error de validación o una entidad no encontrada).

PersonaInputAdapterRest

```
@Slf4j
@Adapter
public class PersonaInputAdapterRest {

    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;

    @Autowired
    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    private PersonaMapperRest personaMapperRest;

    PersonInputPort personInputPort;

    private String setPersonOutputPortInjection(String dbOption) throws InvalidOptionException {
        if (dbOption.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            personInputPort = new PersonUseCase(personOutputPortMaria);
            return DatabaseOption.MARIA.toString();
        } else if (dbOption.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
            personInputPort = new PersonUseCase(personOutputPortMongo);
            return DatabaseOption.MONGO.toString();
        } else {
            throw new InvalidOptionException("Invalid database option: " + dbOption);
        }
    }

    public List<PersonaResponse> historial(String database) {
        log.info("Into historial PersonaEntity in Input Adapter");
        try {
            if (setPersonOutputPortInjection(database).equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
                return personInputPort.findAll().stream().map(personaMapperRest::fromDomainToAdapterRestMaria)
                    .collect(Collectors.toList());
            } else {
                return personInputPort.findAll().stream().map(personaMapperRest::fromDomainToAdapterRestMongo)
                    .collect(Collectors.toList());
            }
        } catch (InvalidOptionException e) {
            log.warn(e.getMessage());
            return new ArrayList<PersonaResponse>();
        }
    }
}
```

```

public PersonaResponse crearPersona(PersonaRequest request) {
    try {
        setPersonOutputPortInjection(request.getDatabase());
        Person person = personInputPort.create(personaMapperRest.fromAdapterToDomain(request));
        return personaMapperRest.fromDomainToAdapterRestMaria(person);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
        //return new PersonaResponse("", "", "", "", "", "", "");
    }
    return null;
}

public ResponseEntity<?> obtenerPersona(String database, int cc) throws NoExistException {
    try {
        setPersonOutputPortInjection(database);
        Person person = personInputPort.findOne(cc);
        return ResponseEntity.ok(personaMapperRest.fromDomainToAdapterRestMaria(person));
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

public ResponseEntity<?> actualizarPersona(String database, int cc, PersonaRequest request) throws NoExistException {
    try {
        setPersonOutputPortInjection(database);
        Person person = personInputPort.edit(cc, personaMapperRest.fromAdapterToDomain(request));
        return ResponseEntity.ok(personaMapperRest.fromDomainToAdapterRestMaria(person));
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    } catch (NoExistException e) {
        return ResponseEntity.status(status:404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Persona con cc " + cc + " no existe en la base de datos ", LocalDateTime.now()));
    } catch (Exception e) {
        return ResponseEntity.status(status:500).body(new Response(String.valueOf(HttpStatus.INTERNAL_SERVER_ERROR),
            description:"Error interno en el servidor", LocalDateTime.now()));
    }
    return ResponseEntity.notFound().build();
}

```

```

public ResponseEntity<?> eliminarPersona(String database, int cc) throws InvalidOptionException, NoExistException {
    setPersonOutputPortInjection(database);
    Optional<Person> person = Optional.ofNullable(personInputPort.findOne(cc));
    if (person.isEmpty()) {
        throw new NoExistException("The person with id " + cc + " does not exist into db, cannot be deleted");
    }
    personInputPort.drop(cc);
    return ResponseEntity.ok(new Response(String.valueOf(HttpStatus.OK),
        "Person with id " + cc + " deleted successfully",
        LocalDateTime.now()));
}

public ResponseEntity<?> contarPersonas(String database) {
    try {
        setPersonOutputPortInjection(database);
        return ResponseEntity.ok(personInputPort.count());
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

```

ProfesionInputAdapterRest

```
@Slf4j
@Adapter
public class ProfesionInputAdapterRest {

    @Autowired
    @Qualifier("profesionOutPutAdapterMaria")
    private ProfessionOutputPort professionOutputPortMaria;

    @Autowired
    @Qualifier("profesionOutPutAdapterMongo")
    private ProfessionOutputPort professionOutputPortMongo;

    @Autowired
    private ProfesionMapperRest professionMapperRest;

    private ProfessionInputPort professionInputPort;

    private String setProfessionOutputPortInjection(String dbOption) throws InvalidOptionException {
        if (dbOption.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            professionInputPort = new ProfessionUseCase(professionOutputPortMaria);
            return DatabaseOption.MARIA.toString();
        } else if (dbOption.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
            professionInputPort = new ProfessionUseCase(professionOutputPortMongo);
            return DatabaseOption.MONGO.toString();
        } else {
            throw new InvalidOptionException("Invalid database option: " + dbOption);
        }
    }

    public List<ProfesionResponse> historial(String database) {
        log.info("Into historial ProfessionEntity in Input Adapter");
        try {
            if (setProfessionOutputPortInjection(database).equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
                return professionInputPort.findAll().stream().map(professionMapperRest::fromDomainToAdapterRestMaria)
                    .collect(Collectors.toList());
            } else {
                return professionInputPort.findAll().stream().map(professionMapperRest::fromDomainToAdapterRestMongo)
                    .collect(Collectors.toList());
            }
        } catch (InvalidOptionException e) {
            log.warn(e.getMessage());
            return new ArrayList<>();
        }
    }
}
```

```

public ProfesionResponse crearProfesion(ProfesionRequest request, String database) {
    try {
        setProfessionOutputPortInjection(database);
        Profession profession = professionInputPort.create(professionMapperRest.fromAdapterToDomain(request));
        return professionMapperRest.fromDomainToAdapterRestMaria(profession);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return null;
}

public ResponseEntity<?> obtenerProfesion(String database, int id) throws NoExistException {
    try {
        setProfessionOutputPortInjection(database);
        Profession profession = professionInputPort.findOne(id);
        return ResponseEntity.ok(professionMapperRest.fromDomainToAdapterRestMaria(profession));
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

public ResponseEntity<?> actualizarProfesion(String database, int id, ProfesionRequest request) throws NoExistException {
    try {
        setProfessionOutputPortInjection(database);
        Profession profession = professionInputPort.edit(id, professionMapperRest.fromAdapterToDomain(request));
        return ResponseEntity.ok(professionMapperRest.fromDomainToAdapterRestMaria(profession));
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    } catch (NoExistException e) {
        return ResponseEntity.status(status:404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Profession with id " + id + " does not exist in the database", LocalDateTime.now()));
    } catch (Exception e) {
        return ResponseEntity.status(status:500).body(new Response(String.valueOf(HttpStatus.INTERNAL_SERVER_ERROR),
            description:"Internal server error", LocalDateTime.now()));
    }
    return ResponseEntity.notFound().build();
}

```

```

public ResponseEntity<?> eliminarProfesion(String database, int id) throws InvalidOptionException, NoExistException {
    setProfessionOutputPortInjection(database);
    Optional<Profesion> profession = Optional.ofNullable(professionInputPort.findOne(id));
    if (profession.isEmpty()) {
        throw new NoExistException("The profession with id " + id + " does not exist in the database, cannot be deleted");
    }
    professionInputPort.drop(id);
    return ResponseEntity.ok(new Response(String.valueOf(HttpStatus.OK),
        "Profession with id " + id + " deleted successfully",
        LocalDateTime.now()));
}

public ResponseEntity<?> contarProfesiones(String database) {
    try {
        setProfessionOutputPortInjection(database);
        return ResponseEntity.ok(professionInputPort.count());
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

```

TelefonoInputAdapterRest

```
@Slf4j
@Adapter
public class TelefonoInputAdapterRest {

    @Autowired
    @Qualifier("phoneOutputAdapterMaria")
    private PhoneOutputPort phoneOutputPortMaria;

    @Autowired
    @Qualifier("phoneOutputAdapterMongo")
    private PhoneOutputPort phoneOutputPortMongo;

    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;

    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    private TelefonoMapperRest telefonoMapperRest;

    private PhoneInputPort phoneInputPort;

    private PersonInputPort personInputPort;

    private String setPhoneOutputPortInjection(String dbOption) throws InvalidOptionException {
        if (dbOption.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            phoneInputPort = new PhoneUseCase(phoneOutputPortMaria, personOutputPortMaria);
            personInputPort = new PersonUseCase(personOutputPortMaria);
            return DatabaseOption.MARIA.toString();
        } else if (dbOption.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
            phoneInputPort = new PhoneUseCase(phoneOutputPortMongo, personOutputPortMongo);
            personInputPort = new PersonUseCase(personOutputPortMongo);
            return DatabaseOption.MONGO.toString();
        } else {
            throw new InvalidOptionException("Invalid database option: " + dbOption);
        }
    }

    public List<TelefonoResponse> historial(String database) {
        log.info("Into historial PhoneEntity in Input Adapter");
        try {
            if (setPhoneOutputPortInjection(database).equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
                return phoneInputPort.findAll().stream().map(telefonoMapperRest::fromDomainToAdapterRestMaria)
                    .collect(Collectors.toList());
            } else {
                return phoneInputPort.findAll().stream().map(telefonoMapperRest::fromDomainToAdapterRestMongo)
                    .collect(Collectors.toList());
            }
        } catch (InvalidOptionException e) {
            log.warn(e.getMessage());
            return new ArrayList<>();
        }
    }

    public ResponseEntity<String> crearTelefono(TelefonoRequest request, String database) throws NoExistException {
        try {
            setPhoneOutputPortInjection(database);
            Optional<Person> personOptional = Optional.ofNullable(personInputPort.findOne(Integer.valueOf(request.getOwnerId())));
            if (personOptional.isEmpty()) {
                throw new NoExistException("The person with id " + request.getOwnerId() + " does not exist in the database, cannot be updated");
            }
            Person person = personOptional.get();
            Phone phone = phoneInputPort.create(telefonoMapperRest.fromAdapterToDomain(request, person), Integer.parseInt(request.getOwnerId()));
            return ResponseEntity.ok(database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
                telefonoMapperRest.fromDomainToAdapterRestMaria(phone) :
                telefonoMapperRest.fromDomainToAdapterRestMongo(phone));
        } catch (InvalidOptionException e) {
            log.warn(e.getMessage());
        }
        catch (NoExistException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new Response(HttpStatus.NOT_FOUND.toString(), description:"ID of owner not found", LocalDateTime.now()));
        }
        return null;
    }
}
```

```

public ResponseEntity<?> obtenerTelefono(String database, String number) throws NoExistException {
    try {
        setPhoneOutputPortInjection(database);
        Phone phone = phoneInputPort.findOne(number);
        TelefonoResponse response = database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
            telefonoMapperRest.fromDomainToAdapterRestMaria(phone) :
            telefonoMapperRest.fromDomainToAdapterRestMongo(phone);
        return ResponseEntity.ok(response);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

public ResponseEntity<?> actualizarTelefono(String database, String number, TelefonoRequest request) throws NoExistException {
    try {
        setPhoneOutputPortInjection(database);
        Optional<Phone> phoneOptional = Optional.ofNullable(phoneInputPort.findOne(number));
        if (phoneOptional.isEmpty()) {
            throw new NoExistException("The phone with number " + number + " does not exist in the database, cannot be updated");
        }

        Optional<Person> personOptional = Optional.ofNullable(personInputPort.findOne(Integer.valueOf(request.getOwnerId())));
        if (personOptional.isEmpty()) {
            throw new NoExistException("The person with id " + request.getOwnerId() + " does not exist in the database, cannot be updated");
        }

        Person person = personOptional.get();
        Phone updatedPhone = phoneInputPort.edit(number, telefonoMapperRest.fromAdapterToDomain(request, person), Integer.parseInt(request.getOwnerId()));

        TelefonoResponse response = database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
            telefonoMapperRest.fromDomainToAdapterRestMaria(updatedPhone) :
            telefonoMapperRest.fromDomainToAdapterRestMongo(updatedPhone);

        return ResponseEntity.ok(response);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    } catch (NoExistException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new Response(HttpStatus.NOT_FOUND.toString(), description:"ID of owner not found", LocalDateTime.now()));
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new Response(HttpStatus.INTERNAL_SERVER_ERROR.toString(), description:"Internal server error", LocalDateTime.now()));
    }
    return ResponseEntity.notFound().build();
}
}

public ResponseEntity<?> eliminarTelefono(String database, String number) throws NoExistException, InvalidOptionException {
    setPhoneOutputPortInjection(database);
    Optional<Phone> phone = Optional.ofNullable(phoneInputPort.findOne(number));
    if (phone.isEmpty()) {
        throw new NoExistException("The phone with number " + number + " does not exist in the database, cannot be deleted");
    }
    phoneInputPort.drop(number);
    return ResponseEntity.ok(new Response(HttpStatus.OK.toString(),
        "Phone with number " + number + " deleted successfully", LocalDateTime.now()));
}

public ResponseEntity<?> contarTelefonos(String database) {
    try {
        setPhoneOutputPortInjection(database);
        return ResponseEntity.ok(phoneInputPort.count());
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}
}

```

EstudioInputAdapterRest

```
@Slf4j
@Adapter
public class EstudioInputAdapterRest {

    @Autowired
    @Qualifier("studyOutputAdapterMaria")
    private StudyOutputPort studyOutputPortMaria;

    @Autowired
    @Qualifier("studyOutputAdapterMongo")
    private StudyOutputPort studyOutputPortMongo;

    @Autowired
    @Qualifier("personOutputAdapterMaria")
    private PersonOutputPort personOutputPortMaria;

    @Autowired
    @Qualifier("personOutputAdapterMongo")
    private PersonOutputPort personOutputPortMongo;

    @Autowired
    @Qualifier("profesionOutPutAdapterMaria")
    private ProfessionOutputPort professionOutputPortMaria;

    @Autowired
    @Qualifier("profesionOutPutAdapterMongo")
    private ProfessionOutputPort professionOutputPortMongo;

    @Autowired
    private EstudioMapperRest estudioMapperRest;

    private StudyInputPort studyInputPort;
    private PersonInputPort personInputPort;
    private ProfessionInputPort professionInputPort;
```

```
private String setStudyOutputPortInjection(String dbOption) throws InvalidOptionException {
    if (dbOption.equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
        studyInputPort = new StudyUseCase(studyOutputPortMaria, personOutputPortMaria, professionOutputPortMaria);
        personInputPort = new PersonUseCase(personOutputPortMaria);
        professionInputPort = new ProfessionUseCase(professionOutputPortMaria);
        return DatabaseOption.MARIA.toString();
    } else if (dbOption.equalsIgnoreCase(DatabaseOption.MONGO.toString())) {
        studyInputPort = new StudyUseCase(studyOutputPortMongo, personOutputPortMongo, professionOutputPortMongo);
        personInputPort = new PersonUseCase(personOutputPortMongo);
        professionInputPort = new ProfessionUseCase(professionOutputPortMongo);
        return DatabaseOption.MONGO.toString();
    } else {
        throw new InvalidOptionException("Invalid database option: " + dbOption);
    }
}

public List<EstudioResponse> historial(String database) {
    log.info("Into historial StudyEntity in Input Adapter");
    try {
        if (setStudyOutputPortInjection(database).equalsIgnoreCase(DatabaseOption.MARIA.toString())) {
            return studyInputPort.findAll().stream()
                .map(estudioMapperRest::fromDomainToAdapterRestMaria)
                .collect(Collectors.toList());
        } else {
            return studyInputPort.findAll().stream()
                .map(estudioMapperRest::fromDomainToAdapterRestMongo)
                .collect(Collectors.toList());
        }
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
        return new ArrayList<>();
    }
}
```

```

public ResponseEntity<?> crearEstudio(EstudioRequest request, String database) {
    try {
        if (request.getPersonId() == null || request.getProfessionId() == null ||
            request.getGraduationDate() == null || request.getUniversityName() == null) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                .body(new Response(HttpStatus.BAD_REQUEST.toString(),
                    "Study data is incomplete or null", LocalDateTime.now()));
        }
        setStudyOutputPortInjection(database);

        Person person = personInputPort.findOne(Integer.parseInt(request.getPersonId()));
        if (person == null) {
            throw new NoExistException("The person with id " + request.getPersonId() + " does not exist in the database, cannot create study.");
        }

        Profession profession = professionInputPort.findOne(Integer.parseInt(request.getProfessionId()));
        if (profession == null) {
            throw new NoExistException("The profession with id " + request.getProfessionId() + " does not exist in the database, cannot create study.");
        }

        Study existingStudy = studyInputPort.findOne(Integer.parseInt(request.getPersonId()), Integer.parseInt(request.getProfessionId()));
        if (existingStudy != null) {
            return ResponseEntity.status(HttpStatus.CONFLICT)
                .body(new Response(HttpStatus.CONFLICT.toString(),
                    "Person already has a study with the same profession", LocalDateTime.now()));
        }

        Study study = Study.builder()
            .person(person)
            .profession(profession)
            .graduationDate(LocalDate.parse(request.getGraduationDate()))
            .universityName(request.getUniversityName())
            .build();
        studyInputPort.create(study, Integer.parseInt(request.getPersonId()), Integer.parseInt(request.getProfessionId()));

        try {
            response = database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
                estudioMapperRest.fromDomainToAdapterRestMaria(study) :
                estudioMapperRest.fromDomainToAdapterRestMongo(study);
        } catch (NullPointerException ex) {
            log.warn("NullPointerException during response mapping: ", ex);
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(new Response(HttpStatus.INTERNAL_SERVER_ERROR.toString(), "Error mapping study response", LocalDateTime.now()));
        }
        return ResponseEntity.ok(response);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body(new Response(HttpStatus.BAD_REQUEST.toString(), "Invalid database option", LocalDateTime.now()));
    } catch (NoExistException e) {
        log.warn(e.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new Response(HttpStatus.NOT_FOUND.toString(), e.getMessage(), LocalDateTime.now()));
    } catch (DateTimeParseException e) {
        log.warn("Invalid date format: " + e.getParsedString());
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body(new Response(HttpStatus.BAD_REQUEST.toString(), "Invalid date format", LocalDateTime.now()));
    } catch (Exception e) {
        log.error("Unexpected error while creating study: ", e);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new Response(HttpStatus.INTERNAL_SERVER_ERROR.toString(), "Internal server error", LocalDateTime.now()));
    }
}

```

```

public ResponseEntity<?> obtenerEstudio(String database, int ccPerson, int idProf) throws NoExistException {
    try {
        setStudyOutputPortInjection(database);
        Study study = studyInputPort.findOne(idProf, ccPerson);
        EstudioResponse response = database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
            estudioMapperRest.fromDomainToAdapterRestMaria(study) :
            estudioMapperRest.fromDomainToAdapterRestMongo(study);
        return ResponseEntity.ok(response);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

public ResponseEntity<?> actualizarEstudio(String database, int ccPerson, int idProf, EstudioRequest request) throws NoExistException {
    try {
        setStudyOutputPortInjection(database);

        Optional<Person> personOptional = Optional.ofNullable(personInputPort.findOne(Integer.valueOf(request.getPersonId())));
        Optional<Profession> professionOptional = Optional.ofNullable(professionInputPort.findOne(Integer.valueOf(request.getProfessionId())));

        if (personOptional.isEmpty() || professionOptional.isEmpty()) {
            throw new NoExistException("Person or Profession not found, cannot update study.");
        }

        // Build the updated study
        Study updatedStudy = Study.builder()
            .person(personOptional.get())
            .profession(professionOptional.get())
            .graduationDate(LocalDate.parse(request.getGraduationDate()))
            .universityName(request.getUniversityName())
            .build();
        studyInputPort.edit(idProf, ccPerson, updatedStudy);

        EstudioResponse response = database.equalsIgnoreCase(DatabaseOption.MARIA.toString()) ?
            estudioMapperRest.fromDomainToAdapterRestMaria(updatedStudy) :
            estudioMapperRest.fromDomainToAdapterRestMongo(updatedStudy);

        return ResponseEntity.ok(response);
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new Response(HttpStatus.INTERNAL_SERVER_ERROR.toString(), "Internal server error", LocalDateTime.now()));
    }
}

public ResponseEntity<?> eliminarEstudio(String database, int ccPerson, int idProf) throws NoExistException, InvalidOptionException {
    setStudyOutputPortInjection(database);
    Optional<Study> study = Optional.ofNullable(studyInputPort.findOne(idProf, ccPerson));
    if (study.isEmpty()) {
        throw new NoExistException("The study with id " + idProf + " and person ID " + ccPerson + " does not exist, cannot be deleted");
    }
    studyInputPort.drop(idProf, ccPerson);
    return ResponseEntity.ok(new Response(HttpStatus.OK.toString(),
        "Study deleted successfully", LocalDateTime.now()));
}

public ResponseEntity<?> contarEstudios(String database) {
    try {
        setStudyOutputPortInjection(database);
        return ResponseEntity.ok(studyInputPort.count());
    } catch (InvalidOptionException e) {
        log.warn(e.getMessage());
    }
    return ResponseEntity.notFound().build();
}

```

Desarrollo Controllers

Los *controllers* en el contexto de la arquitectura hexagonal actúan como la capa de presentación y sirven como puntos de entrada para manejar las solicitudes HTTP que llegan al sistema. Cada controlador, como EstudioControllerV1, PersonaControllerV1, ProfessionControllerV1 y TelefonoControllerV1, está diseñado para recibir solicitudes específicas, delegar la lógica de negocio a los *adapters* de entrada REST, y responder al cliente con el resultado de la operación. Estos *controllers* siguen un estándar RESTful, lo cual facilita la interacción mediante métodos HTTP como GET, POST, PUT y DELETE.

Además de servir como intermediarios entre el cliente y los *adapters* de entrada REST, los *controllers* también son responsables de gestionar las respuestas y los códigos de estado apropiados, como 200 OK para solicitudes exitosas, 404 Not Found cuando no se encuentran los datos solicitados, 400 Bad Request para errores de validación, y 500 Internal Server Error en caso de fallos internos. Esto asegura que el cliente obtenga feedback claro y consistente sobre el estado de su solicitud.

PersonaControllerV1

```
@Slf4j
@RestController
@RequestMapping("/api/v1/persona")
public class PersonaControllerV1 {

    @Autowired
    private PersonaInputAdapterRest personaInputAdapterRest;

    @ResponseBody
    @GetMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<PersonaResponse> personas(@PathVariable String database) {
        log.info("Into personas REST API");
        return personaInputAdapterRest.historial(database.toUpperCase());
    }

    @ResponseBody
    @PostMapping(path = "", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
    public PersonaResponse crearPersona(@RequestBody PersonaRequest request) {
        log.info("esta en el metodo crearTarea en el controller del api");
        return personaInputAdapterRest.crearPersona(request);
    }

    @ResponseBody
    @GetMapping(path = "{database}/{cc}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> persona(@PathVariable String database, @PathVariable int cc) {
        log.info("Into persona REST API");
        try {
            return ResponseEntity.ok(personaInputAdapterRest.obtenerPersona(database.toUpperCase(), cc));
        }
        catch (NoExistException e) {
            return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
                "Persona con cc " + cc + " no existe en la base de datos ", LocalDateTime.now()));
        }
    }
}
```

```
@ResponseBody
@PutMapping(path = "/{database}/{cc}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> actualizarPersona(@PathVariable String database, @PathVariable int cc, @RequestBody PersonaRequest request) {
    log.info("Into actualizarPersona REST API");
    try {
        return ResponseEntity.ok(personaInputAdapterRest.actualizarPersona(database.toUpperCase(), cc, request));
    }
    catch (NoExistException e) {
        return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Persona con cc " + cc + " no existe en la base de datos ", LocalDateTime.now()));
    }
}

@ResponseBody
@DeleteMapping(path = "/{database}/{cc}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> eliminarPersona(@PathVariable String database, @PathVariable int cc) {
    log.info("Into eliminarPersona REST API");
    try {
        return personaInputAdapterRest.eliminarPersona(database.toUpperCase(), cc);
    }
    catch (NoExistException e) {
        return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Persona con cc " + cc + " no existe en la base de datos ", LocalDateTime.now()));
    }
    catch (InvalidOptionException e) {
        return ResponseEntity.status(500).body(new Response(String.valueOf(HttpStatus.INTERNAL_SERVER_ERROR),
            "Bad Request", LocalDateTime.now()));
    }
}

@ResponseBody
@GetMapping(path = "{database}/count", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> contarPersonas(@PathVariable String database) {
    log.info("Into contarPersonas REST API");
    return ResponseEntity.ok(personaInputAdapterRest.contarPersonas(database.toUpperCase()));
}
```

ProfesionControllerV1

```
@Slf4j
@RestController
@RequestMapping("/api/v1/profession")
public class ProfesionControllerV1 {

    @Autowired
    private ProfesionInputAdapterRest profesionInputAdapterRest;

    @ResponseBody
    @GetMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<ProfesionResponse> profesions(@PathVariable String database) {
        log.info("Into profesions REST API");
        return profesionInputAdapterRest.historial(database.toUpperCase());
    }

    @ResponseBody
    @PostMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
    public ProfesionResponse crearProfesion(@RequestBody ProfesionRequest request, @PathVariable String database) {
        log.info("Into crearProfesion REST API");
        return profesionInputAdapterRest.crearProfesion(request, database.toUpperCase());
    }

    @ResponseBody
    @GetMapping(path = "/{database}/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> profesion(@PathVariable String database, @PathVariable int id) {
        log.info("Into profesion REST API");
        try {
            return ResponseEntity.ok(profesionInputAdapterRest.obtenerProfesion(database.toUpperCase(), id));
        } catch (NotExistException e) {
            return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
                "Profession with id " + id + " does not exist in the database", LocalDateTime.now()));
        }
    }
}
```

```
@ResponseBody
@PutMapping(path = "/{database}/{id}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> actualizarProfesion(@PathVariable String database, @PathVariable int id, @RequestBody ProfesionRequest request) {
    log.info("Into actualizarProfesion REST API");
    try {
        return ResponseEntity.ok(profesionInputAdapterRest.actualizarProfesion(database.toUpperCase(), id, request));
    } catch (NotExistException e) {
        return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Profession with id " + id + " does not exist in the database", LocalDateTime.now()));
    }
}

@ResponseBody
@DeleteMapping(path = "/{database}/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> eliminarProfesion(@PathVariable String database, @PathVariable int id) {
    log.info("Into eliminarProfesion REST API");
    try {
        return profesionInputAdapterRest.eliminarProfesion(database.toUpperCase(), id);
    } catch (NotExistException e) {
        return ResponseEntity.status(404).body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
            "Profession with id " + id + " does not exist in the database", LocalDateTime.now()));
    } catch (InvalidOperationException e) {
        return ResponseEntity.status(500).body(new Response(String.valueOf(HttpStatus.INTERNAL_SERVER_ERROR),
            "Bad Request", LocalDateTime.now()));
    }
}

@ResponseBody
@GetMapping(path = "/{database}/count", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> contarProfesiones(@PathVariable String database) {
    log.info("Into contarProfesiones REST API");
    return ResponseEntity.ok(profesionInputAdapterRest.contarProfesiones(database.toUpperCase()));
}
```

TelefonoControllerV1

```
@Slf4j
@RestController
@RequestMapping("/api/v1/phone")
public class TelefonoControllerV1 {

    @Autowired
    private TelefonoInputAdapterRest telefonoInputAdapterRest;

    @ResponseBody
    @GetMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TelefonoResponse> obtenerTelefonos(@PathVariable String database) {
        log.info("Into obtenerTelefonos REST API");
        return telefonoInputAdapterRest.historial(database.toUpperCase());
    }

    @ResponseBody
    @PostMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> crearTelefono(@RequestBody TelefonoRequest request, @PathVariable String database) {
        log.info("Into crearTelefono REST API");
        try {
            return telefonoInputAdapterRest.crearTelefono(request, database.toUpperCase());
        } catch (NoExistException e) {
            log.warn(e.getMessage());
            return null;
        }
    }

    @ResponseBody
    @GetMapping(path = "/{database}/{number}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> obtenerTelefono(@PathVariable String database, @PathVariable String number) {
        log.info("Into obtenerTelefono REST API");
        try {
            return telefonoInputAdapterRest.obtenerTelefono(database.toUpperCase(), number);
        } catch (NoExistException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
                    "Phone with number " + number + " does not exist in the database", LocalDateTime.now()));
        }
    }

    @ResponseBody
    @PutMapping(path = "{database}/{number}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> actualizarTelefono(@PathVariable String database, @PathVariable String number, @RequestBody TelefonoRequest request) {
        log.info("Into actualizarTelefono REST API");
        try {
            return telefonoInputAdapterRest.actualizarTelefono(database.toUpperCase(), number, request);
        } catch (NoExistException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
                    "Phone with number " + number + " does not exist in the database", LocalDateTime.now()));
        }
    }

    @ResponseBody
    @DeleteMapping(path = "{database}/{number}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> eliminarTelefono(@PathVariable String database, @PathVariable String number) {
        log.info("Into eliminarTelefono REST API");
        try {
            return telefonoInputAdapterRest.eliminarTelefono(database.toUpperCase(), number);
        } catch (NoExistException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new Response(String.valueOf(HttpStatus.NOT_FOUND),
                    "Phone with number " + number + " does not exist in the database", LocalDateTime.now()));
        } catch (InvalidOperationException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                .body(new Response(String.valueOf(HttpStatus.BAD_REQUEST),
                    "Invalid database option", LocalDateTime.now()));
        }
    }

    @ResponseBody
    @GetMapping(path = "{database}/count", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> contarTelefonos(@PathVariable String database) {
        log.info("Into contarTelefonos REST API");
        return telefonoInputAdapterRest.contarTelefonos(database.toUpperCase());
    }
}
```

EstudioControllerV1

```
@Slf4j
@RestController
@RequestMapping("/api/v1/estudios")
public class EstudioControllerV1 {

    @Autowired
    private EstudioInputAdapterRest estudioInputAdapterRest;

    @ResponseBody
    @GetMapping(path = "/{database}", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<EstudioResponse> obtenerEstudios(@PathVariable String database) {
        log.info("Fetching all studies from database: {}", database);
        return estudioInputAdapterRest.historial(database.toUpperCase());
    }

    @ResponseBody
    @PostMapping(path = "", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> crearEstudio(@RequestBody EstudioRequest request, @RequestParam String database) {
        log.info("Creating a new study in database: {}", database);
        return estudioInputAdapterRest.crearEstudio(request, database.toUpperCase());
    }

    @ResponseBody
    @GetMapping(path = "/{database}/{ccPerson}/{idProf}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> obtenerEstudio(@PathVariable String database, @PathVariable int ccPerson, @PathVariable int idProf) {
        log.info("Fetching study with person ID: {} and profession ID: {} from database: {}", ccPerson, idProf, database);
        try {
            return estudioInputAdapterRest.obtenerEstudio(database.toUpperCase(), ccPerson, idProf);
        } catch (NoExistException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new Response(HttpStatus.NOT_FOUND.toString(), e.getMessage(), LocalDateTime.now()));
        }
    }
}
```

```
@ResponseBody
@PutMapping(path = "{database}/{ccPerson}/{idProf}", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> actualizarEstudio(@PathVariable String database, @PathVariable int ccPerson, @PathVariable int idProf, @RequestBody EstudioRequest request) {
    log.info("Updating study with person ID: {} and profession ID: {} in database: {}", ccPerson, idProf, database);
    try {
        return estudioInputAdapterRest.actualizarEstudio(database.toUpperCase(), ccPerson, idProf, request);
    } catch (NoExistException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new Response(HttpStatus.NOT_FOUND.toString(), e.getMessage(), LocalDateTime.now()));
    }
}

@ResponseBody
@DeleteMapping(path = "{database}/{ccPerson}/{idProf}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> eliminarEstudio(@PathVariable String database, @PathVariable int ccPerson, @PathVariable int idProf) {
    log.info("Deleting study with person ID: {} and profession ID: {} from database: {}", ccPerson, idProf, database);
    try {
        return estudioInputAdapterRest.eliminarEstudio(database.toUpperCase(), ccPerson, idProf);
    } catch (NoExistException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new Response(HttpStatus.NOT_FOUND.toString(), e.getMessage(), LocalDateTime.now()));
    } catch (InvalidOperationException e) {
        throw new RuntimeException(e);
    }
}

@ResponseBody
@GetMapping(path = "{database}/count", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> contarEstudios(@PathVariable String database) {
    log.info("Counting all studies in database: {}", database);
    return estudioInputAdapterRest.contarEstudios(database.toUpperCase());
}
```

4.2 Pruebas de Ejecución

Para demostrar la correcta ejecución del proyecto, se elaboró un video en el que se presentan pruebas de funcionamiento a través de ambos adaptadores de entrada implementados: el adaptador API Rest, documentado y probado con Swagger, y el adaptador de consola CLI. En este video se ilustra cómo cada uno de estos adaptadores permite la interacción con las funcionalidades del sistema. La demostración abarca tanto la consulta y manipulación de datos como el flujo completo de las principales operaciones.

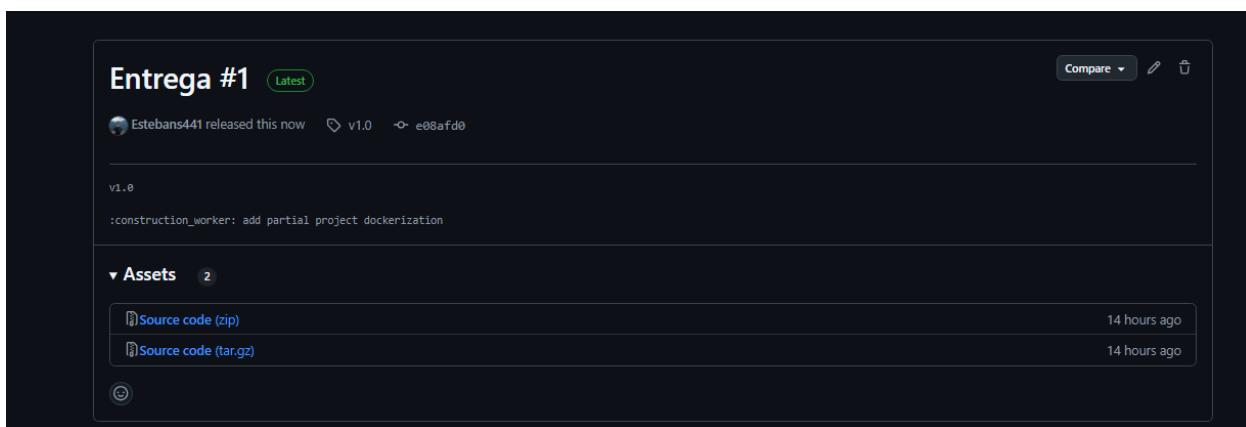
Url del Video:

https://www.canva.com/design/DAGVcuiHIYE/SWndk1rtvpzY5VgO5iyQXA/watch?utm_content=DAGVcuiHIYE&utm_campaign=designshare&utm_medium=link&utm_source=editor

4.3 TAG

Url para la revisión del estado final del repositorio al momento de la entrega:

<https://github.com/Estebans441/personapp-hexa-spring-boot/releases/tag/v1.0>



5. Conclusiones y Lecciones Aprendidas

1. La práctica de construir el sistema utilizando una arquitectura hexagonal y modular no solo fue un ejercicio enriquecedor, sino también un paso crucial en la comprensión de cómo convertir un monolito en una arquitectura de microservicios en el futuro. La estructura modular facilitó la separación de responsabilidades, lo cual es fundamental para una transición eficiente y menos costosa a microservicios sin tener que rediseñar completamente el sistema.
2. La implementación de dos adaptadores de entrada, CLI y API REST, permitió ver en la práctica cómo 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.
3. La incorporación de dos bases de datos distintas (MongoDB y MariaDB) representó un reto que subrayó la importancia de un diseño claro en los puertos y adaptadores. Aprendimos a gestionar adecuadamente las dependencias y a implementar inyección de dependencias que permita una mayor adaptabilidad y un uso optimizado de recursos, aislando de manera eficiente.
4. Las pruebas de los métodos de inyección, configuración de adaptadores y los mapeos entre entidades de dominio e interfaces de entrada resultaron ser un ejercicio clave para garantizar la estabilidad y consistencia del sistema. Estos pasos subrayaron la importancia de contar con pruebas exhaustivas y un mapeo riguroso que facilite el mantenimiento y reduzca el riesgo de errores en un sistema en expansión.
5. La separación de capas y la definición de interfaces bien documentadas brindaron al sistema una estructura que puede crecer y adaptarse sin dificultades adicionales. La modularidad facilita no solo el mantenimiento sino también la incorporación de nuevas funcionalidades sin alterar los componentes preexistentes, lo cual es esencial para sistemas que buscan ser escalables y mantener su integridad a lo largo del tiempo.
6. Esta implementación de un monolito modular basado en la arquitectura hexagonal proporciona una comprensión profunda y práctica de cómo se podría migrar gradualmente a microservicios. Al

contar con un núcleo bien separado y con adaptadores versátiles, el sistema podría fragmentarse en servicios independientes con menor esfuerzo, evitando reestructuraciones importantes. Este ejercicio es altamente valioso ya que brinda una visión detallada de cómo convertir una arquitectura monolítica en microservicios en el futuro.

6. Referencias

Bass, L., Clements, P. and Kazman, R. (2022) 13.4 Patterns for Usability', in Software Architecture in Practice. 4th edn. Pearson Education, pp. 222–223.

Novoseltseva, E. (2024, 19 febrero). ¿Qué es arquitectura Hexagonal o arquitectura de puertos y adaptadores ? Apiumhub. <https://apiumhub.com/es/tech-blog-barcelona/arquitectura-hexagonal/>

Iluwatar. (s. f.). *Module. Patrones de Diseño Java.* <https://java-design-patterns.com/es/patterns/module/#explicacion>

Ibm. (2023, 17 julio). ¿Qué es Java Spring Boot? | IBM. ¿Qué es Java Spring Boot? <https://www.ibm.com/mx-es/topics/java-spring-boot>

Keppi, E. (2024, 29 marzo). *Lanzamiento de Java 11: nuevas características y capacidades.* JavaRush. <https://javarush.com/es/groups/posts/es.1961.lanzamiento-de-java-11-nuevas-caracteristicas-y-capacidades>

Dowsett, C. (2023, 3 febrero). *What is MongoDB?* Built In. <https://builtin.com/data-science/mongodb>

Rouse, M. (2020, 3 noviembre). *Interfaz de línea de comandos o CLI.* ComputerWeekly.es. <https://www.computerweekly.com/es/definicion/Interfaz-de-linea-de-comandos-o-CLI>

Ibm. (2024, 3 octubre). ¿Qué es una API REST? IBM. <https://www.ibm.com/mx-es/topics/rest-apis>

MariaDB: La solución de base de datos de código abierto que necesitas - hack(io) - honest & result-driven school. (s. f.). <https://www.hackio.com/blog/mariadb-solucion-de-base-de-datos>

Home (2024) *Docker Documentation.* Available at: <https://docs.docker.com/> (Accessed: 18 October 2024).