



**Implementación de back-end con Clean Architecture para sistema de matrículas de  
Ingeni@**

**Andrés Grisales González**

Trabajo de grado presentado como requisito parcial para optar al título de:  
**Ingeniero de Sistemas**

Asesores

Sandra Patricia Zabala Orrego, Especialista (Esp) en Gerencia de proyectos  
Clara Lucía Monsalve Ríos, Magíster (MSc) en Educación

Universidad de Antioquia  
Ingeniería de Sistemas  
Pregrado  
Medellín, Colombia  
2023

<b>Cita</b>	Grisales González [1]	
<b>Referencia</b>	[1]	A. Grisales González, “Implementación de back-end con Clean Architecture para sistema de matrículas de Ingeni@”, Trabajo de grado profesional, Ingeniería de Sistemas, Universidad de Antioquia, Medellín, 2023.
Estilo	IEEE	
(2020)		



CENDOI

**Repositorio Institucional:** <https://bibliotecadigital.udea.edu.co>

Universidad de Antioquia - [www.udea.edu.co](http://www.udea.edu.co)

**Rector:** John Jairo Arboleda Céspedes.

**Decano/Director:** Julio César Saldarriaga Molina.

**Jefe departamento:** José Luis Botía Valderrama.

El contenido de esta obra corresponde al derecho de expresión de los autores y no compromete el pensamiento institucional de la Universidad de Antioquia ni desata su responsabilidad frente a terceros. Los autores asumen la responsabilidad por los derechos de autor y conexos

## TABLA DE CONTENIDO

<b>RESUMEN</b> .....	<b>2</b>
<b>ABSTRACT</b> .....	<b>2</b>
<b>I. INTRODUCCIÓN</b> .....	<b>3</b>
<b>II. OBJETIVOS</b> .....	<b>4</b>
<b>A. Objetivo general</b> .....	<b>4</b>
<b>B. Objetivos específicos</b> .....	<b>4</b>
<b>III. MARCO TEÓRICO</b> .....	<b>4</b>
<b>A. Modelo de dominio</b> .....	<b>4</b>
<b>B. Tecnologías</b> .....	<b>6</b>
<b>C. Arquitectura y organización</b> .....	<b>7</b>
a. Clean Architecture.....	7
b. CQS.....	8
c. Package by layer / feature / component.....	9
<b>IV. METODOLOGÍA</b> .....	<b>10</b>
<b>A. Desarrollo y forma de trabajo</b> .....	<b>10</b>
a. Inception.....	10
b. Sprints.....	10
c. Dailys.....	10
d. Planning.....	11
e. Review.....	11
<b>B. Elicitación de requisitos</b> .....	<b>11</b>
a. Identificación del problema.....	11
b. Definición modelo de dominio. ....	11
c. Los requisitos funcionales.....	11
d. Los requisitos no funcionales. ....	11
e. Alcance y limitaciones.....	11
<b>C. Selección de tecnologías</b> .....	<b>12</b>
a. Búsqueda de lenguajes de programación, frameworks, tipos y gestores de bases de datos. ....	12
b. Clasificación.....	12
c. Filtro y selección. ....	12
<b>D. Selección de patrones de arquitectura, diseño y convenciones</b> .....	<b>12</b>
a. Búsqueda de arquitecturas, patrones de diseño, convenciones de acuerdo al contexto del problema y las tecnologías previamente seleccionadas.....	12
b. Filtro y selección. ....	12
<b>V. RESULTADOS</b> .....	<b>12</b>

<b>A. Implementación .....</b>	<b>12</b>
a. Nombramiento de recursos (endpoints / servicios) .....	13
b. Los tipos de modelos (Body) recibe la API para los POST/PUT/PATCH .....	13
c. Tipos de modelo que responde la API cuando se consultan recursos GET .....	14
d. Otros tipos de modelos a nivel de API .....	15
e. Organización de paquetes por componente y aplicación de Clean Architecture en el sistema .....	16
f. Restricciones y flujos en la comunicación .....	21
<b>B. Métricas e indicadores del proyecto.....</b>	<b>22</b>
<b>VI. CONCLUSIONES .....</b>	<b>24</b>
<b>REFERENCIAS.....</b>	<b>25</b>

## TABLA DE FIGURAS

FIG 1. MODELO DE DOMINIO Y DE DATA COMO EJEMPLO .....	5
FIG 2. ARQUITECTURA LIMPIA.....	7
FIG 3. CQRS.....	8
FIG 4. DIFERENTES MANERAS DE ORGANIZACIÓN DE CÓDIGO. ....	9
FIG 5. ORGANIZACIÓN POR COMPONENTES.....	10
FIG 6. SWAGGER UI - GRUPO DE CONTROLADORES PARA LA GESTIÓN DE COHORTES.....	13
FIG 7. SWAGGER UI - DESCRIPCIÓN DE CONTRATO PARA EL REGISTRO DE UNA COHORTE.....	14
FIG 8. SWAGGER UI - DESCRIPCIÓN DE CONTRATO PARA LA CONSULTA DE UNA COHORTE. ....	15
FIG 9. SWAGGER UI - MODELO DE RESPUESTA DE CONSULTA DE VARIAS COHORTES CON PAGINACIÓN.....	16
FIG 10. ORGANIZACIÓN DEL PROYECTO. ESTRUCTURACIÓN EXTERNA DE PAQUETES POR COMPONENTE. ....	17
FIG 11. ORGANIZACIÓN A PRIMER NIVEL DE CADA COMPONENTE.....	18
FIG 12. ORGANIZACIÓN COMPLETA DE PAQUETES, CLASES E INTERFACES EN UN COMPONENTE.....	19
FIG 13. EJEMPLO DE ORGANIZACIÓN CON MÚLTIPLES IMPLEMENTACIONES DE GATEWAYS Y REPOSITORIES.....	20
FIG 14. MÉTRICAS SONAR.....	22

## SIGLAS, ACRÓNIMOS Y ABREVIATURAS

<b>CQS</b>	Command Query Separation
<b>CQRS</b>	Command Query Responsibility Segregation
<b>API</b>	Application Programming Interface
<b>DTO</b>	Data Transfer Object
<b>URI</b>	Uniform Resource Identifier
<b>REST</b>	Representational State Transfer
<b>SQALE</b>	Software Quality Assessment based on Lifecycle Expectations
<b>DRAI</b>	Departamento de Recursos de Apoyo e Informática

---

## RESUMEN

En el presente trabajo se presenta el proceso de construcción e implementación de un sistema de matrículas utilizando Clean Architecture y Command Query Separation (CQS) haciéndose uso de tecnologías del ecosistema Java. Se da una introducción al modelo de dominio para entender el contexto y problema, se hace énfasis en la descripción del proceso de selección de tecnologías, prácticas, convenciones y metodologías que fueron utilizadas durante el desarrollo del sistema. Posteriormente se muestra de manera muy detallada cómo se aplicó Clean Architecture y CQS a nivel de organización en el sistema en conjunto con el resto de las convenciones de la industria que resultaban convenientes integrar en el sistema. Finalmente se presenta información que es de interés acerca de los informes de métricas del sistema en lo que respecta a calidad y aseguramiento de código realizado con herramientas de análisis de código estático como Sonar.

***Palabras clave*** — **Arquitectura limpia, CQS, diseño back-end, arquitectura de software, java**

## ABSTRACT

This document presents the process of construction and implementation of an enrollment school system using Clean Architecture and CQS with Java ecosystem technologies. An introduction to the domain model is given to understand the problem, emphasizing the description of the selection process of technologies, practices, conventions and methodologies that were used during the development of the system. Subsequently, it is shown in detail how Clean Architecture and CQS were applied at the organizational level in the system together with the rest of the industry conventions that were convenient to integrate in the system. Finally, information is presented regarding the quality and code assurance reports performed with static code analysis tools such as Sonar.

***Keywords*** — **Clean architecture, CQS, back-end design, software architecture, java**

---

## I. INTRODUCCIÓN

Ingeni@ es la unidad del Departamento de Recursos de Apoyo e Informática –DRAI– de la Facultad de Ingeniería de la Universidad de Antioquia responsable de expandir el conocimiento de la institución a todas las estructuras productivas y de desarrollo social del país mediante soluciones TIC. Desde Ingeni@ se tenía la necesidad de un sistema de matrícula multi-proyecto que permitiera dar soporte al ciclo de vida de matrícula de estudiantes en los cursos ofertados en cohortes de proyectos que se gestionan allí. Se tuvo así la oportunidad de realizar las prácticas académicas en la unidad con este proyecto. Adicionalmente, se esperaba que el proyecto, desde su gestión hasta la arquitectura, estructuración e implementación se convirtiera en una guía para los nuevos proyectos venideros que se inicien en el DRAI. Luego, con el objetivo de construir un sistema que tuviera como base las mejores prácticas y altos estándares de la industria, se hizo uso de una arquitectura limpia en la implementación que permitió lograr que se garantizará un bajo nivel de acoplamiento, alta cohesión, puntos de extensión y demás ventajas que tienen los sistemas que son implementados utilizando este tipo de prácticas.



## II. OBJETIVOS

### A. Objetivo general

Implementar el Back-End del sistema de matrículas utilizando Clean Architecture y CQS para garantizar a nivel de código que el sistema sea simple, mantenible, testeable y extensible gracias a puntos de extensión.

### B. Objetivos específicos

- Establecer la manera en que se implementará Clean Architecture y CQS en el proyecto: Organización de paquetes y clases, reglas de dependencia, restricciones, convenciones de nombramiento y en general dónde debe ir cada cosa.
- Identificar estándares y propuestas en la industria del software que sean beneficiosas en la implementación de acuerdo a las tecnologías utilizadas.
- Asegurar la calidad del código y arquitectura de la solución.

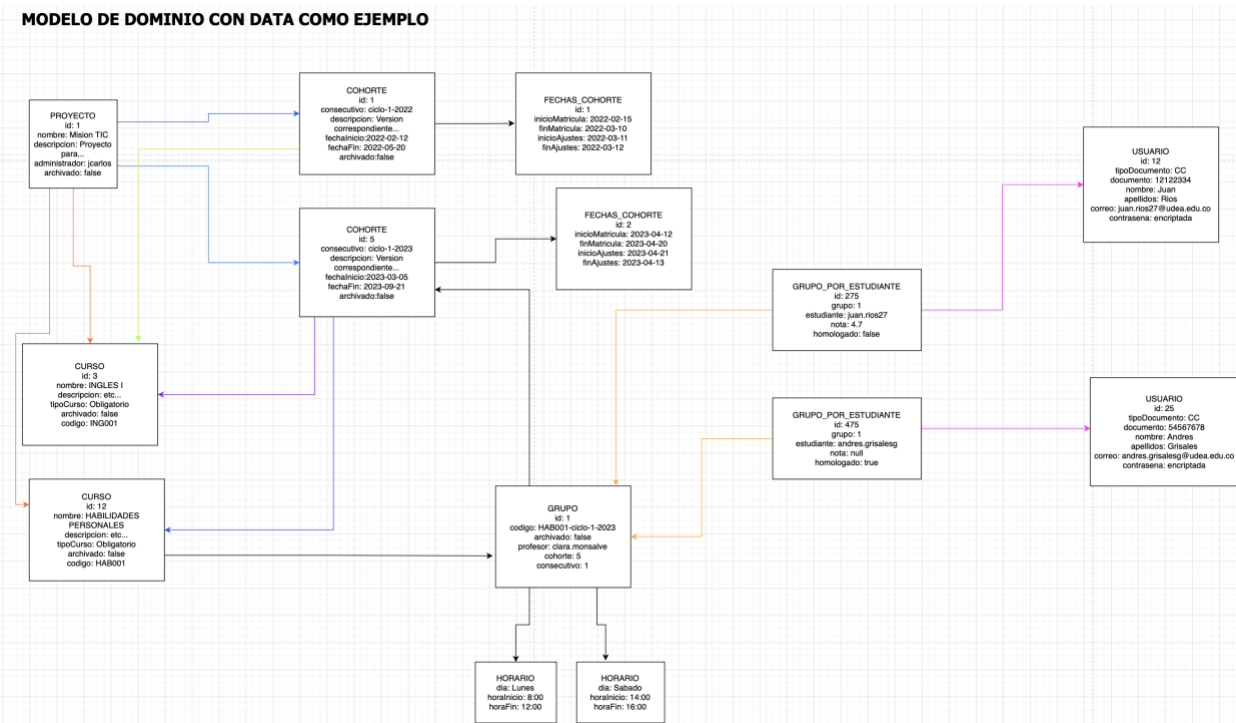
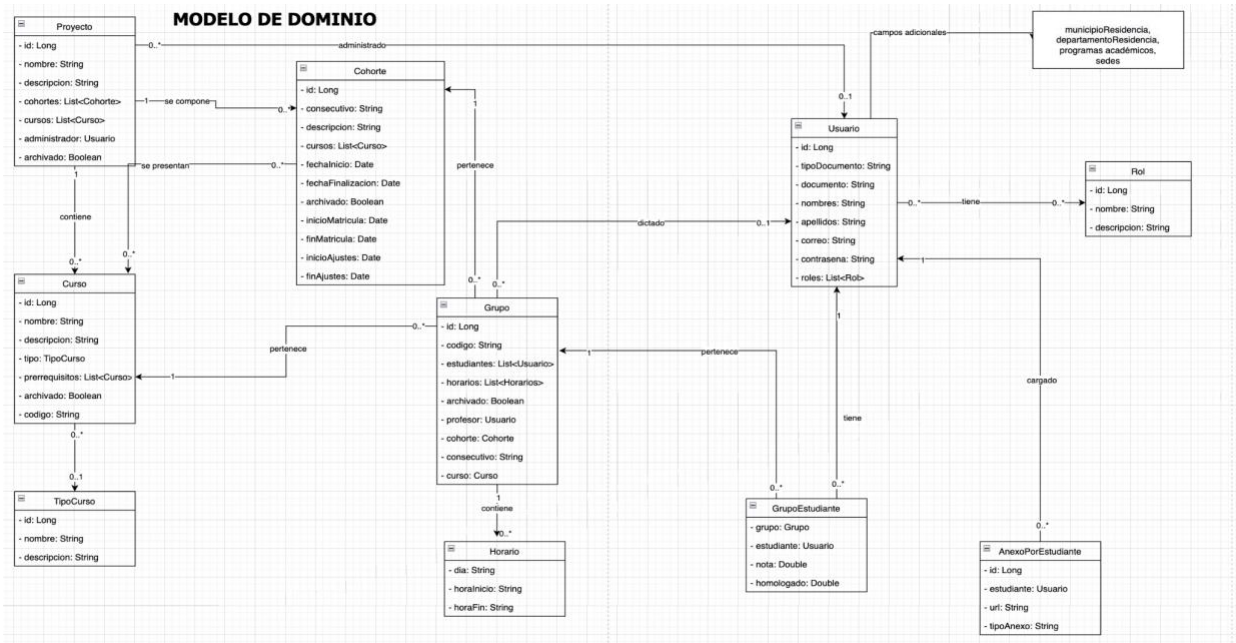
## III. MARCO TEÓRICO

### A. Modelo de dominio

Un modelo de dominio es utilizado con el objetivo de visualizar y expresar el entendimiento obtenido durante un proceso de análisis en el que se introduce un vocabulario y conceptos claves del dominio del problema por parte del cliente y así identificar relaciones, tipos de relación, restricciones y atributos.

Por lo tanto, previa implementación del sistema de matrículas se realizó una elicitación de requisitos y en consecuencia se desarrolló un modelo de dominio que describe las distintas entidades, atributos y relaciones que rigen el dominio del problema.

Fig 1. Modelo de dominio y de data como ejemplo



Desde una vista general, en la Figura 1 que muestra el modelo de dominio se describe que:

- Un proyecto se compone de cero o más cohortes, y una cohorte sólo puede pertenecer a un proyecto [Relación de composición de uno a muchos].
- Un proyecto se compone de cero o más cursos, y un curso solo puede pertenecer a un proyecto [Relación de composición de uno a muchos].

- En una cohorte se pueden ofrecer cero o más cursos, y un curso puede ofrecerse en más de una cohorte [Relación de asociación de muchos a muchos].
- En una cohorte se abren cero o más grupos, y un grupo está asociado a una cohorte y a un curso en específico [Relaciones de composición de uno a muchos desde cohorte y curso hacia grupo].
- El modelo GrupoEstudiante es el resultado de la inscripción de un estudiante (usuario) en un grupo: Un estudiante puede estar inscrito en cero o más grupos, y un grupo puede tener cero o más estudiantes [Relación de asociación de muchos a muchos entre grupo y estudiante (usuario), donde la tabla pivote se mapea explícitamente como un modelo debido a que contiene información de negocio como lo es la nota aprobatoria del estudiante en ese grupo, si canceló, homologó, etc.].
- Uno de los casos de uso principales es el siguiente: Un administrador de proyecto (usuario) precarga un listado de estudiantes para una cohorte, luego durante la fecha de matrículas y ajustes en la cohorte los estudiantes podrán matricularse en los grupos de los cursos que le son ofertados desde la carga.
- La matrícula que realiza un estudiante en los grupos es validada por el motor de reglas: Parametrizaciones de cupos máximos en grupo, si se permite cruce de horarios, prerequisites de cursos, matrícula mínima o máxima de por tipos de curso, si debe matricular oferta completa, entre otros.

La implementación de la solución hace uso de Clean Architecture y CQS, sin embargo, también está presente un conjunto amplio de tecnologías que fueron seleccionadas para la construcción de la aplicación, por esa razón a continuación se describen los conceptos/tecnologías más relevantes dentro del proyecto.

## B. Tecnologías

- **Java - Spring / Spring Boot:** El lenguaje de programación utilizado para la construcción del Back-End es Java en su versión 17. Sobre Java, para la construcción se hace uso de Spring Framework 6 (Spring Boot 3) y varios de sus sub-proyectos / módulos para la construcción de aplicaciones Back-End. Estos módulos son **Spring Starter Web** para definición de los servicios REST, **Spring Data JPA** para gestionar las entidades con la base de datos y acelerar el desarrollo, **Spring Security** para la autorización y autenticación de usuarios (se hace uso de autenticación básica e integración con JWT, acceso a recursos basado en roles), **Spring Starter Test** para las pruebas con Junit 5 / Jupiter, Mockito, WebMvc Test, etc., **Spring Batch** para procesamiento por lotes con resiliencia, por ejemplo, para el envío masivo de correos.
- **PostgreSQL:** Base de datos relacional para el almacenamiento de los datos relacionados y estructurados, como también para el almacenamiento de información en formato json (documentos).
- **Docker y Kubernetes:** Docker es una tecnología para la contenerización de aplicaciones, en este caso del Back-End (artefacto –Jar– con la aplicación), el contenedor posteriormente se despliega en un cluster de Kubernetes donde se puede escalar el Back-End de manera horizontal y hacer balanceo de carga.

- **Flyway:** Sistema robusto de migración de base de datos.
- **Swagger - Open API:** Es el conjunto de herramientas utilizadas para la simplificación de la documentación de los servicios expuestos por el Back-End.
- **React - TypeScript / JS:** React es la librería utilizada para la construcción del frontal (Front-End) de la aplicación con el lenguaje de programación Javascript / TypeScript.
- **GitLab - GitLab CI/CD:** Se hace uso de GitLab (hosteado en los servidores de la Facultad de Ingeniería) como repositorio de código, gestión de historias de usuario en los tableros de tareas y como herramienta para integración y despliegue continuo de los artefactos.

## C. Arquitectura y organización

### a. Clean Architecture

Clean Architecture fue popularizado por Robert C. Martin, conocido como “Uncle Bob” quien propone estructurar el código en capas contiguas, es decir, que solo tienen comunicación con las capas que están inmediatamente a sus lados. Basados en esta idea se encontrarán artículos que hablan sobre Clean Architecture, Onion Architecture, Hexagonal Architecture, en todas ellas podemos ver diferentes enfoques o no dependiendo la manera en que lo analicemos, pero en sí comparten la misma idea de que cada nivel/capa debe realizar sus propias tareas y se comunica únicamente con sus niveles inmediatamente contiguos.

Fig 2. Arquitectura limpia

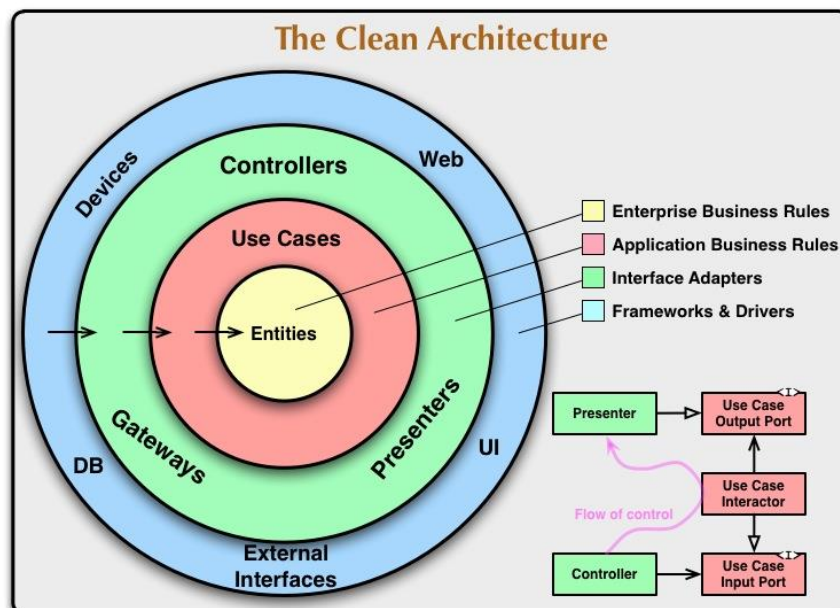


Imagen tomada de <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

En la Figura 2 podemos observar cómo cada uno de los niveles son representados, podría considerarse dentro de la estructura de nuestra aplicación como carpetas independientes, diferentes módulos o librerías que se incluyen como dependencias del proyecto principal. Esta manera de estructurar la aplicación permite principalmente que el sistema cuente con puntos de extensión (el objetivo), lo que permite una alta maniobrabilidad para la evolución de la aplicación durante todo su ciclo de vida y actualizaciones de tecnologías.

## b. CQS

Command Query Separation (CQS) es un concepto bastante relacionado con CQRS (Command Query Responsibility Segregation), CQRS es un patrón de software que busca tratar de manera diferente la recuperación / consulta de datos de la mutación o cambios en el estado de un sistema, debido principalmente a diferentes requerimientos de escalabilidad por el tipo de tráfico que se espera, esto tiene como resultado que por ejemplo en un sistema que aplica este patrón, si el sistema se llama Sistema-X los endpoints de consulta de información (GET) estén en un microservicio que llamaremos Sistema-X-Queries, mientras que los endpoints que modifican el estado del sistema (Creaciones, Actualizaciones, Eliminaciones - POST/PUT/PATCH/DELETE) estarán en otro microservicio que llamaremos, por ejemplo, Sistema-X-Commands, de esa manera estos dos microservicios se pueden escalar de manera **independiente** según sean los requerimientos por tráfico en el sistema. Para el sistema de matrículas se identifica que este patrón arquitectónico (CQRS) no es necesario para su implementación, sin embargo, en proyectos venideros en la unidad Ingeni@ se tendrá en consideración el patrón y se hará un análisis de pertinencia.

Fig 3. CQRS.

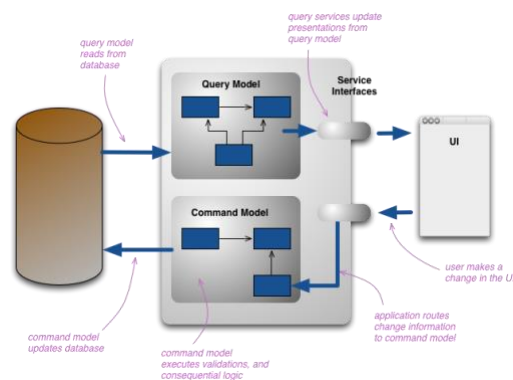


Imagen tomada de <https://martinfowler.com/bliki/CQRS.html>

Por otra parte, tenemos a CQS el cual lo podemos pensar a un nivel más micro, es decir de clases y componentes, mientras que CQRS lo vemos a nivel macro de contextos delimitados de dominio (microservicios). CQS nos invita a separar los métodos y modelos que son de simple consulta (Query) de los que alteran el estado del sistema (Commands), ambos modelos (tipo Command y Query) se construyen a partir de modelos tipo Request (modelo que puede recibirse

en un endpoints o que puede consumirse en forma de Evento de un artefacto de mensajería como Kafka, etc.). Los modelos de tipo Query están estrechamente relacionados a un tipo de modelo de respuesta que llamaremos Response, cuando hacemos una consulta (Query) el sistema devuelve modelos de tipo Response y no directamente modelos que definen entidades, de esta manera no expone el modelo de dominio (entidad pura) del sistema ni acoplamos a los clientes.

### c. Package by layer / feature / component

Históricamente la segregación de código por capas (layer) ha sido la más ampliamente utilizada para la organización a un alto nivel del código (clases, interfaces, etc.), sin embargo, existen otras propuestas que dependiendo el tamaño de la aplicación pueden ser una mejor opción para la organización de la aplicación, estas son el empaquetamiento por feature (característica) y el empaquetamiento por component (componente).

*Fig 4. Diferentes maneras de organización de código.*

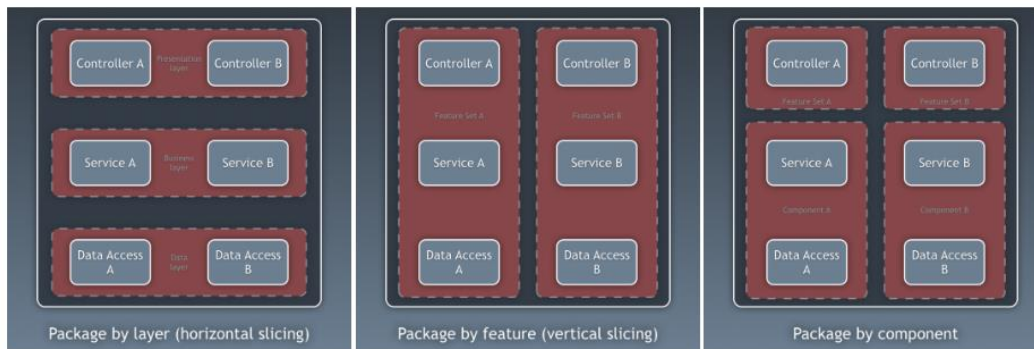


Imagen tomada de <https://herbertgraca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/>

El empaquetamiento por componente es bastante conveniente si aplicamos Clean Architecture (o una de sus variantes) y CQS, pues ofrece una manera de organización muy adecuada y conveniente al momento de seguir las prácticas, lineamientos y restricciones que nos proponen en los dos patrones de arquitectura. Haciendo zoom a esta manera de empaquetar la aplicación veremos que la organización sería de la siguiente manera.

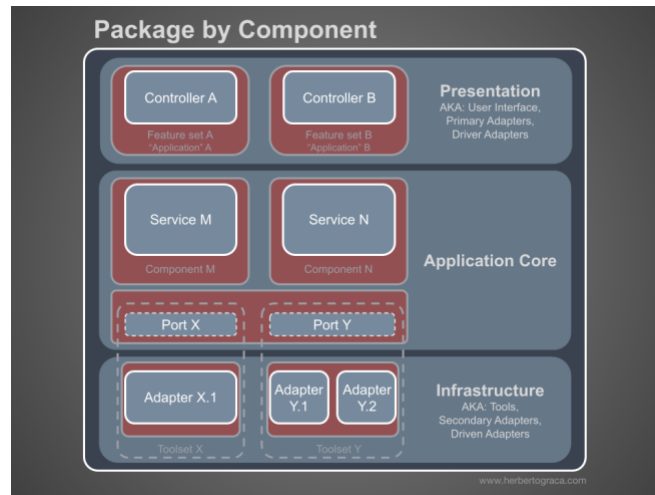
*Fig 5. Organización por componentes.*

Imagen tomada de <https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/>

## IV. METODOLOGÍA

### A. Desarrollo y forma de trabajo

Scrum fue la metodología ágil seleccionada para la gestión del proyecto, se trabajó de la manera en que se describe a continuación.

#### a. Inception.

Para el proyecto esta fase tuvo una duración de 1 mes y se realizaron las 3 actividades previamente descritas en la metodología (1) Elicitación de requisitos, 2) selección de tecnologías, 3) selección de patrones de arquitectura, diseño y convenciones). Esta fase puede variar considerablemente según el proyecto y su alcance, cliente, si los requisitos están ya o no definidos, complejidad del dominio.

#### b. Sprints.

Se manejaron ciclos/iteraciones durante el desarrollo que tuvieron una duración de 2 semanas (10 días de trabajo).

#### c. Dailys.

Estos se llevaron a cabo todos los días laborales y tuvieron una duración promedio de 15 minutos, cuando era necesario que se extendiera se hacía si los miembros del equipo no tenían otros compromisos, en caso tal de que no se tuviera disponibilidad en el momento, se acordaba una reunión para la revisión del pendiente lo más pronto posible.

d. Planning.

Se llevaron a cabo los lunes de la primera semana de cada ciclo/iteración, aquí se definía con el cliente cuales eran las prioridades para el Sprint que iniciaba y se aclaraban o modificaban las historias en caso de ser necesario.

e. Review.

Se llevaron a cabo los viernes de la segunda semana de cada ciclo/iteración de los Sprint, allí se le presentaba al cliente los avances durante el sprint.

## B. Elicitación de requisitos

En conjunto con el cliente quien es el experto en dominio del negocio se describieron los requerimientos funcionales y no funcionales que debe satisfacer el sistema.

a. Identificación del problema.

Se introdujo el problema, contexto y particularidades, allí se empezó a detallar y definir propiamente el contexto de dominio.

b. Definición modelo de dominio.

La definición de la estructura de datos empezó a tomar forma, a partir de los conceptos de dominio y tipos de relaciones (De uso, asociación, composición, agregación y herencia) que se identificaron con la información proveída por el cliente.

c. Los requisitos funcionales.

Aquí se identificaron los casos de uso por tipo de usuario, reglas de negocio, validación de información entrante, procesos batch y capacidades del motor de reglas del sistema de matrículas.

d. Los requisitos no funcionales.

Se establecieron los requisitos no funcionales con el acompañamiento del equipo. Desde este se dieron las recomendaciones iniciales para el desarrollo: Tipos de arquitectura, organización, tecnologías, mecanismos de persistencia de la información, diseño de APIs, convenciones y estilos de nombramiento de variables, constantes, métodos, clases, interfaces y paquetes.

e. Alcance y limitaciones.

Una vez los requisitos fueron definidos, se dejó claro el alcance del proyecto en la primera etapa, se indicó de manera detallada el sistema que no hará y las limitaciones de este.



### C. Selección de tecnologías

- a. Búsqueda de lenguajes de programación, frameworks, tipos y gestores de bases de datos.

Se identificaron las tecnologías disponibles –stacks–, de código abierto (como requisito), disponibles para la implementación.

- b. Clasificación.

Se realizó un compendio de las características de los stacks disponibles: Ventajas y desventajas.

- c. Filtro y selección.

Se hizo un filtro y selección de las tecnologías a utilizar posterior al análisis del compendio de la clasificación, donde se tomaron las decisiones de acuerdo con las curvas de aprendizaje, conocimiento previo y principalmente peso de la tecnología en la solución (que tan apta y conveniente es para este contexto –proyecto– en específico).

Selección: Java en su versión 17, Spring (Data JPA/HIBERNATE, Spring Boot 3, Security, WebMVC para servicios REST), Maven, PostgreSQL.

### D. Selección de patrones de arquitectura, diseño y convenciones

- a. Búsqueda de arquitecturas, patrones de diseño, convenciones de acuerdo al contexto del problema y las tecnologías previamente seleccionadas.

Se realizó una búsqueda exhaustiva de las arquitecturas y prácticas dentro del stack de tecnologías seleccionadas, como también estándares de la industria a un nivel más transversal, errores frecuentes y recomendaciones de uso en la implementación, siempre acorde con las tecnologías que se usaron, por ejemplo: Spring, Rest-MVC, Transacciones de Spring, JPA/HIBERNATE, Batch, Docker, Mail testing, etc.

- b. Filtro y selección.

De acuerdo con las fuentes de consulta y conocimiento previo en el equipo se realizó un filtro y selección de los recursos y fuentes a seguir como guía, ya sea parcial o total, durante la implementación.

## V. RESULTADOS

### A. Implementación

A continuación, se muestran los conceptos claves de la implementación, organización, diferentes prácticas y estándares de la industria en acción, en búsqueda de construir un sistema con arquitectura limpia: puntos de extensión, bajo acoplamiento y alta cohesión.

a. Nombramiento de recursos (endpoints / servicios)

Se implementaron 58 servicios en el Back-End, la Figura 6 muestra la documentación generada con Swagger-UI para el controlador de grupos donde actualmente se tienen 6 servicios creados, allí podemos ver el método HTTP del servicio, URL-Recurso y una breve descripción.

Fig 6. Swagger UI - Grupo de controladores para la gestión de Cohortes.

Gestión de cohortes <small>Diferentes endpoints para la gestión de cohortes.</small>			
GET	/api/v1/cohortes/{id}	Buscar por id. hasAnyAuthority('ADMIN', 'CONSULTOR', 'ESTUDIANTE')	✓ 🔒
PUT	/api/v1/cohortes/{id}	Actualizar una cohorte. hasAuthority('ADMIN')	✓ 🔒
GET	/api/v1/cohortes	Buscar cohortes por proyecto. hasAnyAuthority('ADMIN', 'SUPER_ADMIN', 'CONSULTOR')	✓ 🔒
POST	/api/v1/cohortes	Registrar una nueva cohorte para un proyecto. hasAuthority('ADMIN')	✓ 🔒
GET	/api/v1/cohortes/{id}/registro-estudiantes	Buscar estudiantes registrados en cohorte. hasAnyAuthority('ADMIN')	✓ 🔒
POST	/api/v1/cohortes/{id}/registro-estudiantes	Registrar o actualizar estudiantes en una cohorte. hasAuthority('ADMIN')	✓ 🔒
POST	/api/v1/cohortes/{cohorteId}/cursos/{cursoId}	Asociar curso a una cohorte. hasAuthority('ADMIN')	✓ 🔒
DELETE	/api/v1/cohortes/{cohorteId}/cursos/{cursoId}	Remover curso a una cohorte. hasAuthority('ADMIN')	✓ 🔒
GET	/api/v1/cohortes/{id}/registro-estudiantes/{documento}	Buscar estudiante registrado en cohorte por cohorteld y documento. hasAnyAuthority('ADMIN')	✓ 🔒

Algunas de las buenas prácticas recomendadas (convenciones) que podemos ver están aplicadas en la Figura 6, son:

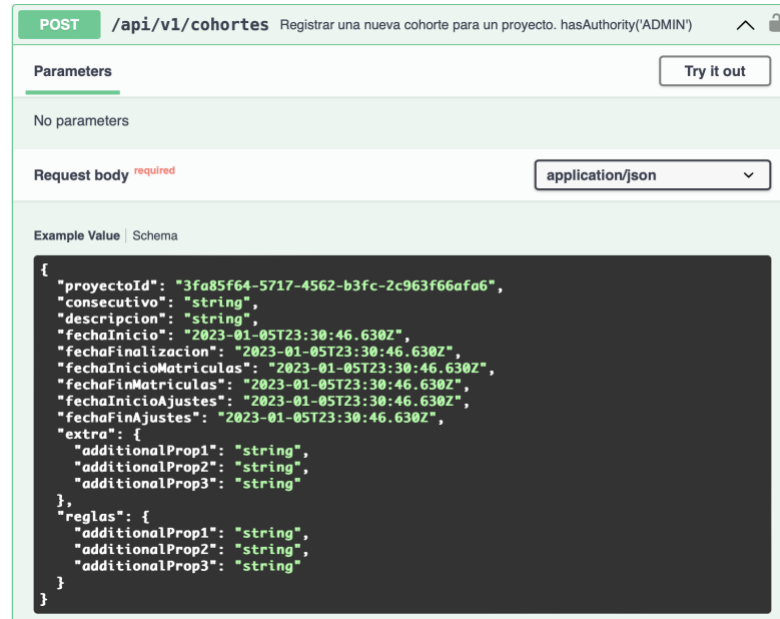
1. Uso de sustantivos para representar recursos.
2. El recurso puede ser una colección o un singleton: /api/v1/cohortes representa una colección de cohortes, mientras que /api/v1/cohortes/{id} representa a una cohorte en específico.
3. Uso de la barra inclinada / para indicar relaciones de jerarquía.
4. No utiliza la barra inclinada / al final de los recursos.
5. Nombramiento de toda la URI en minúscula (lowercase).
6. Uso de guiones para mejorar la legibilidad.
7. No se hace uso de URIs para indicar funciones CRUD, para esto se utilizan los métodos Http.
8. Versionamiento de la API.

b. Los tipos de modelos (Body) recibe la API para los POST/PUT/PATCH

Los modelos que recibe la API siempre son de tipo Request, no son más que un DTO (clase en Java) que representa un modelo que llega al controlador. El nombre Request hace que sea más diciente que simplemente llamarlo DTO, en este contexto. En la imagen vemos la definición del

endpoint que permite registrar una nueva cohorte a un proyecto. El modelo que recibe es del tipo `CohorteSaveRequest`, el cual contiene los atributos que Swagger muestra allí en la Figura 7.

Fig 7. Swagger UI - Descripción de contrato para el registro de una cohorte.



En la Figura 7 se evidencia que el modelo `CohorteSaveRequest`:

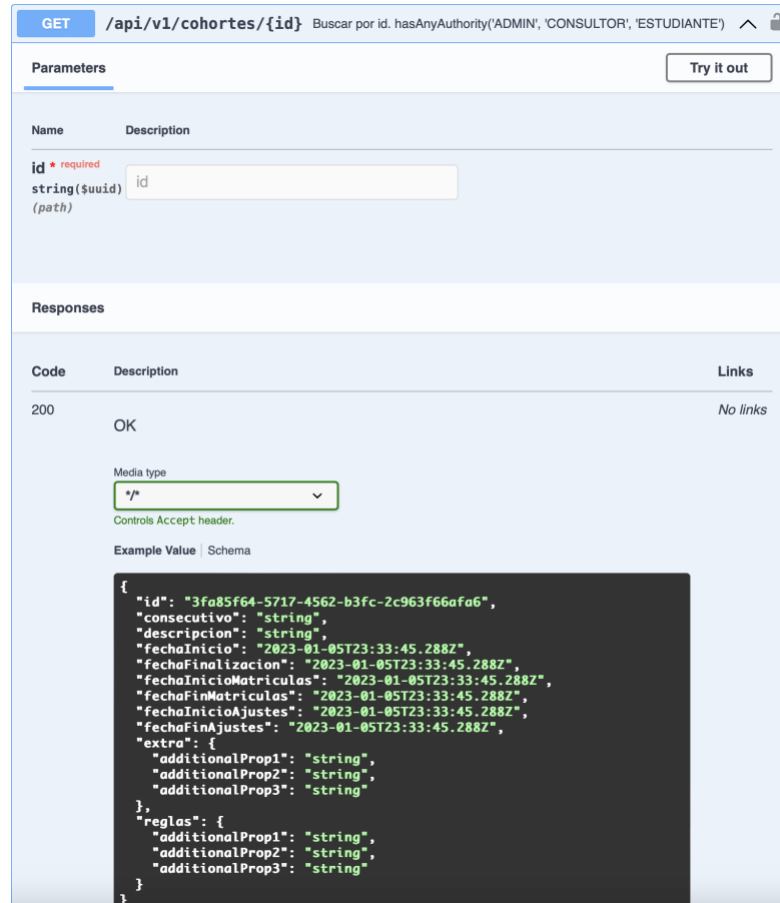
1. Tiene atributos propios de la entidad interna que representa una Cohorte (por ejemplo, el consecutivo, descripción, fecha inicio, etc.)
2. No tiene el id de la cohorte puesto que es creado internamente por el sistema, tampoco información de auditoría como la fecha de creación y actualización, pues esto no es responsabilidad del cliente.
3. Tiene un atributo llamado proyecto id que representa el identificador del proyecto al cual se va a asociar a esa cohorte. El id del proyecto es la única información que requerimos internamente para asociar la cohorte a un proyecto, nótese que no se está recibiendo el modelo completo de un proyecto, si se recibiera el modelo completo de proyecto agregaría complejidad y ruido en el contrato, es información que no es necesaria.
4. El método del controlador que mapea el endpoint recibe el objeto del tipo `ProyectoSaveRequest`, luego lo convierte en un objeto del tipo `ProyectoSaveCommand` y lo envía al servicio / caso de uso. De esta manera no acoplamos el comando al modelo del cliente en una versión específica de la API que puede cambiar en versiones posteriores.

c. Tipos de modelo que responde la API cuando se consultan recursos GET

Los modelos que responde la API son de tipo `Response`, nuevamente es una clase Java – DTO– que representa la información a retornar como respuesta a un GET, o incluso a un PUT /

PATCH si es el caso. Por ejemplo, al consultar una cohorte en específico se devuelve un objeto de clase `CohorteSaveResponse`.

Fig 8. Swagger UI - Descripción de contrato para la consulta de una cohorte.



The screenshot shows the Swagger UI for the endpoint `GET /api/v1/cohortes/{id}`. The parameters section shows a required parameter `id` of type `string($uuid)` located at the path. The responses section shows a 200 OK response with no links. Below the response, there is a media type dropdown set to `*/*` and an example value for the response body:

```
{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "consecutivo": "string",
  "descripcion": "string",
  "fechaInicio": "2023-01-05T23:33:45.288Z",
  "fechaFinalizacion": "2023-01-05T23:33:45.288Z",
  "fechaInicioMatriculas": "2023-01-05T23:33:45.288Z",
  "fechaFinMatriculas": "2023-01-05T23:33:45.288Z",
  "fechaInicioAjustes": "2023-01-05T23:33:45.288Z",
  "fechaFinAjustes": "2023-01-05T23:33:45.288Z",
  "extra": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  },
  "reglas": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  }
}
```

En el modelo de respuesta `CohorteSaveResponse`, vemos que:

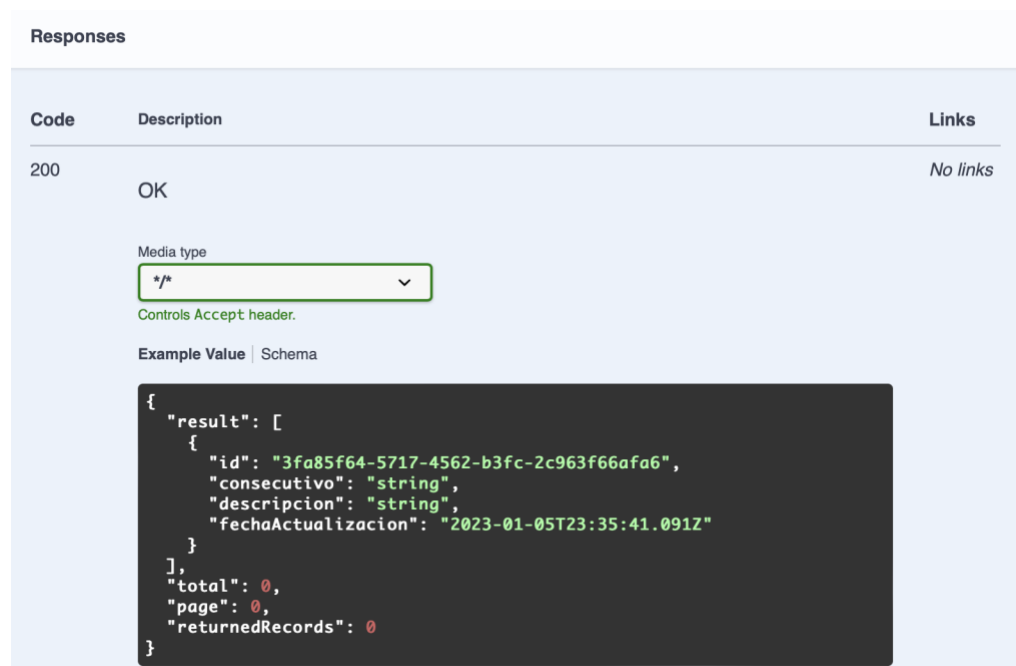
1. A diferencia del modelo `CohorteSaveRequest` este modelo tiene el `id` de la cohorte y no tiene el `id` del proyecto (en el `Request` se necesitaba para asociar la cohorte al proyecto).
2. Retorna información de la fecha de creación y actualización del recurso.

d. Otros tipos de modelos a nivel de API

Si el modelo de actualización completa (PUT) coincide con el de creación, se puede utilizar el mismo modelo de tipo `Request` de la creación (en el ejemplo: `CohorteSaveRequest`), ya que el `id` de la cohorte a actualizar llegará como variable en la URI. Sin embargo, `CohorteSaveRequest` contenía un atributo llamado `proyecto id`, en caso de que para el negocio no tenga sentido que se pueda actualizar el proyecto de una cohorte, como es el caso, entonces se debe crear un modelo específico que se llame, por ejemplo, `CohorteUpdateRequest` el cual se diferenciaría únicamente del `CohorteSaveRequest` en que el `update` no recibirá el atributo `proyecto id`.

Otro tipo de modelo que puede ser común a todas las aplicaciones es cuando no se retorna un único recurso sino una lista de recursos. Por ejemplo, las cohortes que hagan match con un criterio de búsqueda, si para las listas devolvemos menos información (por rendimiento-tráfico de la API, por ejemplo), en lugar de utilizar el modelo `CohorteSaveResponse`, utilizaremos uno llamado `CohorteSaveListResponse` que tiene menos información y que estaría embebido (al igual que el `CohorteSaveResponse` en caso de devolver más de un recurso) en uno genérico del tipo `ResponsePagination` como se observa en la Figura 9 que por buenas prácticas indicaría el número de página consultada, total de páginas disponibles y elementos retornados. Por ejemplo, el modelo de respuesta para la consulta de cohortes vía parámetros en Swagger nos indica que este es el response que debe esperar el cliente, como se observa en la Figura 9.

Fig 9. Swagger UI - Modelo de respuesta de consulta de varias cohortes con paginación.



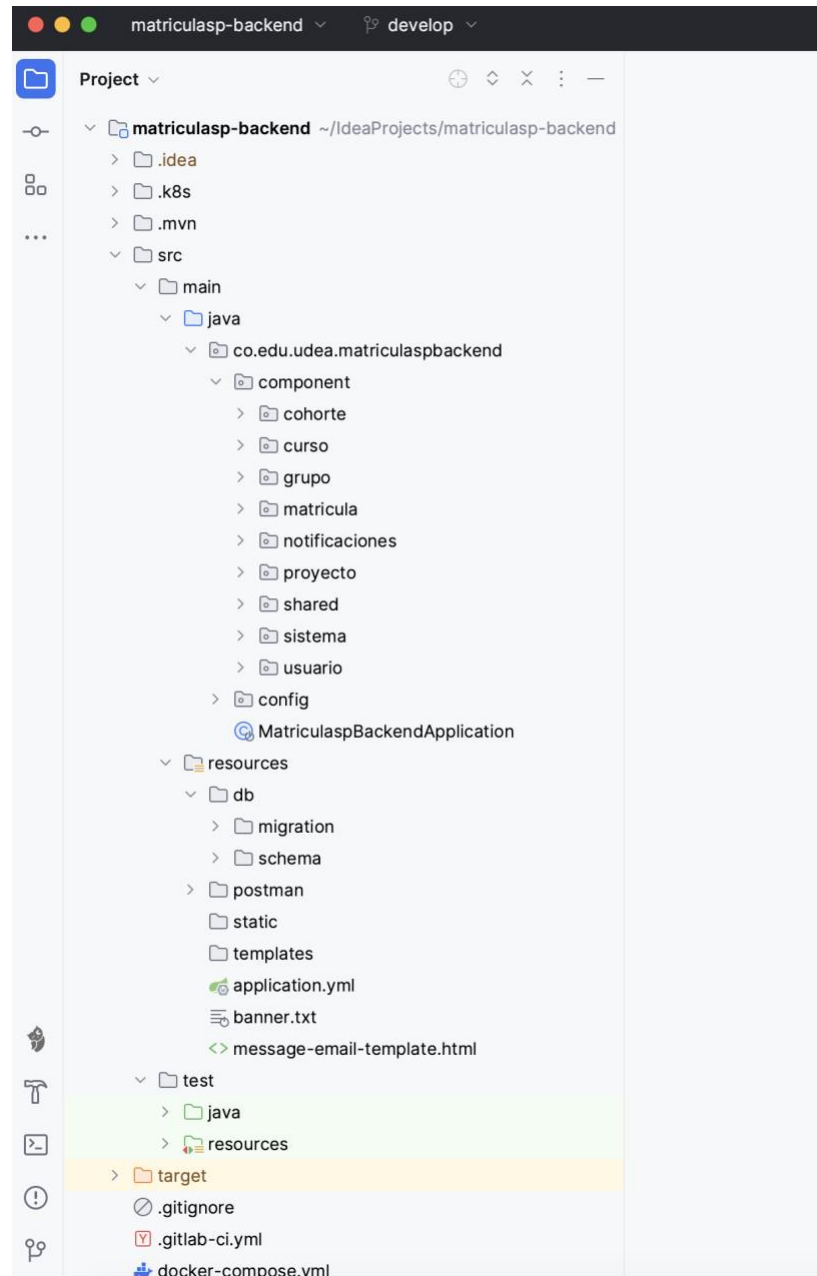
The screenshot displays the Swagger UI interface for a 200 OK response. The response is described as 'OK' with 'No links'. A dropdown menu for 'Media type' is set to '\*/\*'. Below this, there is a section for 'Example Value' and 'Schema'. The example value is a JSON object representing a paginated list of cohort records.

```
{
  "result": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "consecutivo": "string",
      "descripcion": "string",
      "fechaActualizacion": "2023-01-05T23:35:41.091Z"
    }
  ],
  "total": 0,
  "page": 0,
  "returnedRecords": 0
}
```

e. Organización de paquetes por componente y aplicación de Clean Architecture en el sistema

Al seguir una organización por componentes la estructura en el sistema se ve de la siguiente manera al más alto nivel como se ilustra en la Figura 10.

Fig 10. Organización del proyecto. Estructuración externa de paquetes por componente.



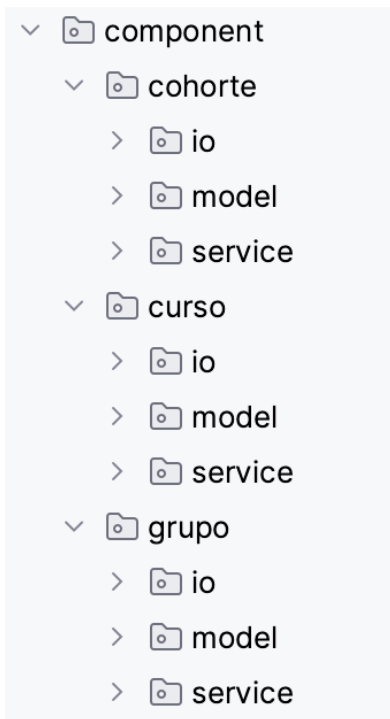
En la Figura 10 se observa que:

1. Es un proyecto Maven como cualquier proyecto Java que utiliza este gestor de proyectos, dockerizado, con configuración para la CI con GitLab y CD en K8s, etc.
2. Vemos un paquete llamado component al nivel de config (este último –config– contiene configuración transversal de la aplicación, por ejemplo, la seguridad, Swagger, Asincronismo, configuración de procesos batch, configuraciones de mailing, sistemas de mensajería, etc.).

3. El paquete component internamente tiene esas features que son regidas por los conceptos de dominio para las operaciones y gestión de proyectos, usuarios, cursos, cohortes, grupos, matrículas, además se tiene uno llamado shared para lógica transversal a los componentes.

A excepción del paquete shared, cada componente se estructura internamente de la manera que se muestra en la Figura 11:

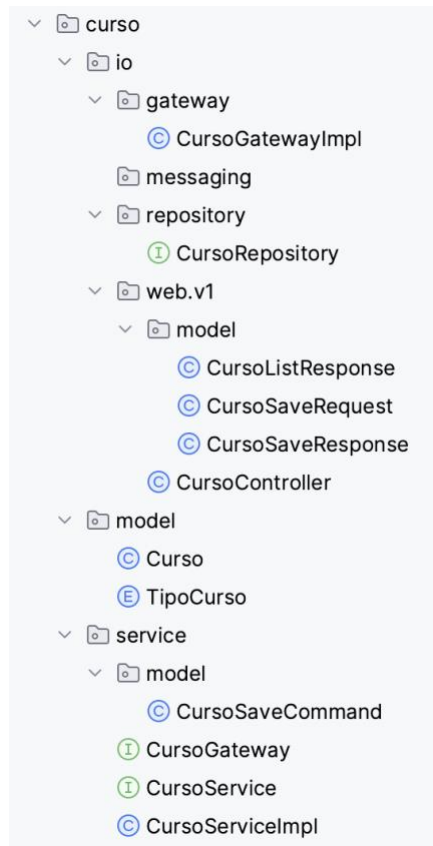
*Fig 11. Organización a primer nivel de cada componente.*



En la Figura 11 vemos los siguientes paquetes en los diferentes componentes:

1. Paquete **io**: Contiene la infraestructura de la aplicación, en términos de Clean Architecture, allí residen: Controladores y en general detalles de implementación de acceso a base de datos, código de integración con sistemas de mensajería, etc., que no hacen parte de la lógica principal / casos de uso de la aplicación.
2. Paquete **model**: Contiene los modelos / entidades principales de la aplicación, son esas entidades que son manejadas por JPA para el mapeo contra la base de datos.
3. Paquete **service**: Contiene la lógica de la aplicación, como también las abstracciones del sistema (interfaces).

En general, si abrimos cualquier jerarquía de los componentes, veremos que la aplicación se estructura de la siguiente manera, en este caso vemos el ejemplo con el componente curso:

*Fig 12. Organización completa de paquetes, clases e interfaces en un componente.*

En la Figura 12 se observa que:

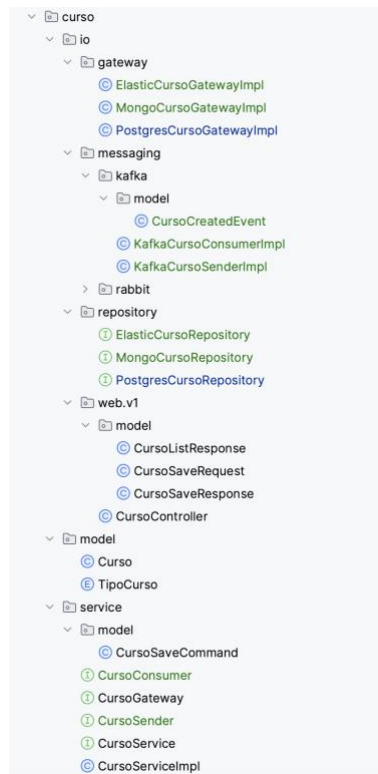
1. El componente (paquete) curso contiene los paquetes de io, model y service (los tres al mismo nivel).
2. El paquete io (la infraestructura) contiene el paquete gateway el cual a su vez tiene internamente una clase (implementación que usa el repository).
3. El paquete **io** contiene el paquete repository el cual a su vez contiene una interface la cual es un detalle de implementación, pues a pesar de ser una interface está acoplada a Spring ya que al abrirla veremos la anotación @Repository que extiende de JpaRepository para gestionar la entidad Curso, es decir, la interface no es una abstracción - puerto.
4. El paquete **io** contiene el paquete web, este a su vez contiene el paquete v1, y este a su vez contiene uno llamado model y el controlador de curso. El paquete model dentro web.v1 contiene los modelos de tipo Request y Response para esa versión del controlador (v1) en lo que respecta a curso. Así garantizamos que se pueda extender, puesto que, si se requiere una v2, creamos la carpeta al nivel de la v1, y esa carpeta v2 de igual manera tendrá un paquete de modelos (contratos) correspondientes a la v2 como su controlador (internamente la clase CursoController mapea en la raíz al endpoint /api/v1/cursos).



5. El paquete `io` contiene el paquete `messaging`, dentro de este paquete estarían las implementaciones (clases) con lógica de integración para la producción y eventos con Kafka, RabbitMQ, SQS, etc. Las interfaces / abstracciones residirían en el paquete `service`.
6. El paquete `model` (que está al mismo nivel de `io` y `service`) en este caso contiene la definición de la entidad `Curso` y `TipoCurso` el cual es un `ValueObject` del cual se compone la entidad `Curso` internamente.
7. El paquete `service` internamente tiene el paquete `model` con los modelos (DTO's) correspondientes a comandos.
8. El paquete `service` internamente contiene todas las abstracciones del sistema (interfaces / puertos) y únicamente contiene implementaciones (clases / adaptadores) de la lógica de negocio - casos de uso. Es allí donde tenemos restricciones como que las implementaciones que estén en `service` únicamente deben conocer abstracciones y no detalles de implementación, los detalles de implementación viven en la infraestructura (`io`) y por inversión de control (inyección de dependencias) se inyectan esas implementaciones en las clases con lógica / casos de uso, de esta manera estamos revirtiendo las dependencias y dejando puntos de extensión, puesto que en el futuro si se requiere una implementación diferente de cualquier cosa que resida en la infraestructura, simplemente se crea y se inyecta en reemplazo de una anterior.

Actualmente no es el caso del proyecto, pero para ejemplificar y ver lo que podemos lograr veamos los cambios realizados en la paquetería a modo de ejemplo.

Fig 13. Ejemplo de organización con múltiples implementaciones de gateways y repositories.



En la figura 13 se observa que:

1. La interface `CursoRepository` y la clase `CursoGatewayImpl` fueron renombradas a `PostgresCursoRepository` y `PostgresCursoGatewayImpl`.
  2. Tenemos 2 nuevas interfaces llamadas `ElasticCursoRepository` y `MongoCursoRepository`.
  3. Tenemos 2 nuevas clases (implementaciones) llamadas `ElasticCursoGatewayImpl` y `MongoCursoGatewayImpl`.
  4. Las clases impl tienen lógica específica del tipo de base de datos y se acoplan al tipo de repositorio gestionado por Spring para ese proveedor.
  5. Sin embargo, las 3 clases impl implementan la interface `CursoGateway`, luego `CursoServiceImpl` conoce únicamente a `CursoGateway`, por lo que para el servicio / lógica de negocio / casos de uso, es transparente el sistema o mecanismo de persistencia y puede intercambiar en cualquier momento cambiar (inyectar) la implementación que se desee, o incluso usar las 3 a la vez.
  6. La mensajería sigue la misma idea descrita anteriormente.
- f. Restricciones y flujos en la comunicación

Restricciones en la comunicación

1. La comunicación entre componentes únicamente se permite entre Servicios definidos internamente. Por ejemplo, la implementación de `CohorteServiceImpl` puede tener por uso a la interface `ProyectoService` (del componente Proyecto) e inyectar una implementación. De ninguna manera desde `CohorteServiceImpl` se puede conocer el Gateway de Proyecto o cualquiera fuera de su componente.
2. Entre repositories y gateways no hay relación de uso.
3. Entre los controladores no hay relación de uso.
4. Los controladores solo se comunican con los servicios de su propio componente a través de la interface / puerto del servicio.
5. Los controladores no deben tener comunicación directa con gateways, repositories, producers o consumers, incluso si se encuentran en su mismo componente.
6. Hay excepciones para las utilidades transversales en la aplicación como las utilidades definidas en el componente shared, pero en general debemos buscar dar la mayor garantía en cuanto a puntos de extensión se refiere utilizando abstracciones / interfaces / puertos.

Flujo más común en aplicaciones que utilicen la estructura propuesta

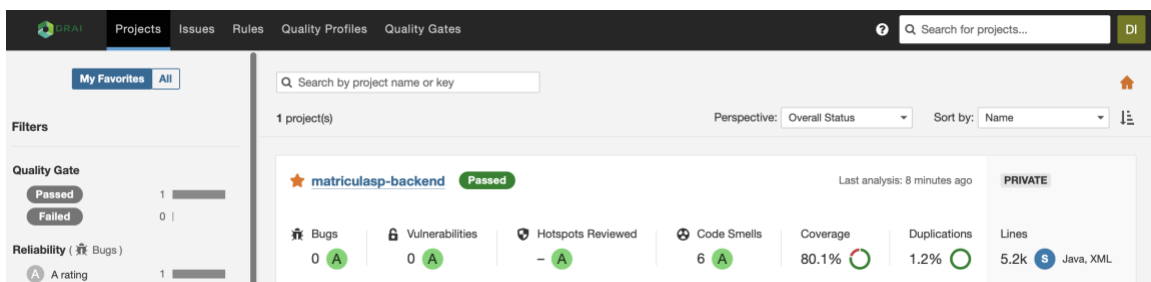
El flujo más común que nos encontraremos en sistemas con esta arquitectura es el siguiente:

1. Un endpoint definido en un controlador es invocado por el cliente (Frontal - Postman - Android, etc).
2. El controlador recibe la petición (en caso de traer body en un modelo de tipo Request). El controlador hace uso de un servicio a través de la abstracción / interface del servicio / caso de uso.

3. El controlador, en caso de haber recibido un modelo de tipo Request lo convierte en un modelo de tipo Comando e invoca al servicio (Caso de Uso) con el comando que espera como parámetro en el método.
4. El servicio interpreta el comando, por ejemplo, hace una consulta y modifica un valor, realiza validaciones de negocio, aplica lógica de negocio, convierte el comando en una entidad para posteriormente persistir si es el caso, disparar un evento haciendo uso de un Producer (por la interface), etc. El servicio únicamente conoce de abstracciones (puertos - interfaces) para la consulta, persistencia, emisión o consumo de eventos a través del uso de gateways, publishers, consumers, producers, demás servicios de los otros componentes, etc.
5. Si el servicio responde modelos de entidad como retorno al controlador, este último tiene la responsabilidad de convertir la entidad en un modelo de tipo Response si es que retornará información al cliente, puesto que no debe acoplar de ninguna manera al cliente a un esquema de dominio que puede cambiar (como las entidades), los cuales puede provocar que se rompan los clientes.
6. Una técnica útil para los mapeos es el patrón static factory method. Allí básicamente las clases de tipo Request, Command y Response definen un método estático que implementando el patrón builder con la ayuda de Lombok, se encargan de convertir objetos de tipo Request y Command (en caso de la clase Request), Command a Entidad (en caso del Command) y Entidad a Response (en caso del Response). Debe ser de esta manera para no acoplar el sistema y mantener las reglas dependencia hacia el centro como nos propone Clean Architecture.

## B. Métricas e indicadores del proyecto

Fig 14. Métricas SONAR



Desde el Sonar del DRAI, a fecha de enero de 2023 la revisión muestra que:

- El número de líneas físicas de código (LoC) que contienen al menos un carácter y que no son un espacio en blanco, tabulación o comentario es de 5200 líneas de código aproximadamente con una cobertura por pruebas unitarias del 80.1%.
- No se identifican issues de tipo bug (Rating A).
- No se identifican issues de tipo vulnerability (Rating A).
- El Hotspots security reviewed obtiene una calificación en Rating de A.
- Se identifican 6 casos de code smells de tipo minor (Rating A).
- El porcentaje de código duplicado es del 1.2%.

- La deuda técnica es de 1 hora 29 minutos.
- Respecto al SQALE la mantenibilidad, fiabilidad, seguridad y revisión de seguridad obtienes todas un Rating un A.

---

## VI. CONCLUSIONES

Es muy importante tener una correcta definición de los requerimientos funcionales y no funcionales, ya que marcan el punto de partida para actividades como el análisis, diseño e implementación.

Cuando se siguen buenas prácticas en el uso de las diferentes tecnologías que se utilizan en un proyecto, se logra minimizar la posibilidad al máximo de que se tengan problemas técnicos de alta criticidad como un bajo rendimiento o alto acoplamiento que impidan evolucionar la aplicación de manera simple.

Una vez finalizada la fase de selección se debe profundizar, según sea necesario, en cada tecnología y patrón seleccionado de cara a tener una visión más amplia de las herramientas disponibles para hacer un correcto uso de ellas en las integraciones y desarrollo como tal.

La calidad del código es muy importante, por esta razón es necesario hacer disposición de herramientas como Sonar que entregan información del proyecto en ese aspecto de cara a realizar correcciones de manera oportuna y así minimizar los riesgos durante y después del desarrollo cuando los sistemas se encuentren en ambientes productivos.

La práctica permitió poner en acción las capacidades investigativas y de conocimientos específicos adquiridos durante toda la etapa formativa del pregrado, especialmente del curso Análisis y diseño de sistemas II el cual fue uno de los más motivantes para darle un enfoque a la formación profesional en esta rama del desarrollo de software y ecosistema de tecnologías.

---

## REFERENCIAS

1. Evans, E., & Evans, E. J. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.
2. Millett, S., & Tune, N. (2015). *Patterns, principles, and practices of domain-driven design*. John Wiley & Sons.
3. Bloch, J. (2008). *Effective java*. Addison-Wesley Professional.
4. Teorey, T. J. (1999). *Database modeling and design*. Morgan Kaufmann.
5. Arango, E. C., & Loaiza, O. L. (2021, April). SCRUM Framework Extended with Clean Architecture Practices for Software Maintainability. In *Computer Science On-line Conference* (pp. 667-681). Springer, Cham.
6. Brown, P. S., Dimitrova, V., Hart, G., Cohn, A. G., & Moura, P. (2021). Refactoring the Whitby Intelligent Tutoring System for Clean Architecture. *Theory and Practice of Logic Programming*, 21(6), 818-834.
7. Aguilar, P., & Figueira, L. (2020). Clean Architecture is not only about business logic. In *I Workshop de Tecnologia da Fatec Ribeirão Preto*.
8. Mihalcea, V. (2016). High-Performance Java Persistence.
9. Kuchana, P. (2004). *Software architecture design patterns in Java*. Auerbach Publications.
10. Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.
11. Steinegger, R. H., Giessler, P., Hippchen, B., & Abeck, S. (2017, April). Overview of a domain-driven design approach to build microservice-based applications. In *The Thrid Int. Conf. on Advances and Trends in Software Engineering*.
12. Le, D. M., Dang, D. H., & Nguyen, V. H. (2018). On domain driven design using annotation-based domain specific language. *Computer Languages, Systems & Structures*, 54, 199-235.
13. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., & Percannella, G. (2016, June). REST APIs: a large-scale analysis of compliance with principles and best practices. In *International conference on web engineering* (pp. 21-39). Springer, Cham.