Decisiones basadas en la arquitectura

**Objetivo1: Creación de los micro-servicios**

**Decisión sobre los micro-servicios:**

Para dividir las entidades en dos microservicios coherentes, aplicaremos principio de **Dominio Limitado** (Bounded Context) de **Diseño Dirigido por el Dominio** (DDD). Esto implica identificar los límites naturales dentro del dominio de negocio y agrupar las entidades que comparten funcionalidad o datos.

En en nuestro caso, hay dos contextos principales:

1. **Gestión de Residuos** (Waste Management)
2. **Autorizaciones de Centros de Residuos** (Waste Center Authorizations)

Por lo que se decidió aplicar la siguiente división:

**Microservicio de Gestión de Residuos**:

* WasteManagerEntity: Contiene la información principal del gestor de residuos.
* WasteManagerAddressEntity: Relacionada con la dirección del gestor de residuos.

**Microservicio de Autorizaciones de Centros de Residuos**:

* WasteCenterAuthorizationEntity: Maneja las autorizaciones de los centros de residuos.

La entidad WasteManagerEntity ahora tiene una referencia a las autorizaciones a través de un identificador o clave foránea, en lugar de contener la lista completa de WasteCenterAuthorizationEntity. Esto ayuda a mantener los microservicios desacoplados y facilita la escalabilidad y el mantenimiento.

# // Microservicio de Gestión de Residuos

public class WasteManagerEntity {

private Long id;

private String nombre;

private String nif;

private WasteManagerAddressEntity wasteManagerAddressEntity;

// Referencia a las autorizaciones por ID

private List<Long> listOfWasteCenterAuthorizationIds;

private Boolean isEnabled = Boolean.TRUE;

private Long version = 0L;

private Date createdDate;

private Date lastModifiedDate;

}

public class WasteManagerAddressEntity {

private Long id;

private String direccion;

private Boolean isEnabled = Boolean.TRUE;

private Long version = 0L;

private Date createdDate;

private Date lastModifiedDate;

}

# // Microservicio de Autorizaciones de Centros de Residuos

public class WasteCenterAuthorizationEntity {

private Long id;

private String authorizationNumber;

}

Con esta división, cada microservicio tiene su propia base de datos y lógica de negocio, lo que permite que se desarrollen, desplieguen y escale de manera independiente. Además, la comunicación entre microservicios se puede realizar mediante llamadas a API REST.

**Decisión sobre la composición de las clases:**

Se extrae a una clase AuditableBase.java de tipo abstracta los campos comunes de fines meramente auditables y se extiende en las clases entidades de esta. De esta manera, el Principio de Responsabilidad Única **(Single Responsibility Principle)** queda mejor implementado.

public abstract class AuditableBase {  
  
 @CreationTimestamp  
 private Date createdDate;  
  
 @UpdateTimestamp  
 private Date lastModifiedDate;  
  
 @Version  
 private Long version;

}

**Decisión sobre la comunicación entre los microservicios:**

Se decidió por el uso de **Feign**, debido a las siguientes ventajas:

1. Es un cliente web declarativo que simplifica la escritura de clientes de servicios web.
2. Está más orientado al lado del servidor, especialmente en microservicios basados en Spring Cloud.
3. Proporciona soporte para anotaciones de Spring MVC y se integra bien con Eureka.

Para manejar respuestas negativas en el servicio Feign sin interrumpir el flujo de la aplicación, de decidió implementar un patrón de **fallback**. Este patrón permite definir una clase que se ejecutará como respaldo en caso de que la llamada al servicio falle.

Además se decidió por el uso de **Hystrix** para un control más avanzado de los mecanismos de tolerancia a fallos. **Hystrix** te permite definir métodos de **fallback**, timeouts, y circuit breakers para manejar fallos en las llamadas a servicios remotos.

**Arquitecturas implementadas y el por qué:**

Las principales ventajas de utilizar los patrones y arquitecturas CQRS, Clean Architecture y DDD son:

**CQRS (Command Query Responsibility Segregation):**

* **[Separación de responsabilidades:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [CQRS divide claramente las operaciones de lectura y escritura, lo que puede resultar en sistemas más fáciles de mantener y escalar](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).
* **[Optimización del rendimiento:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [Al separar las consultas de los comandos, se pueden optimizar independientemente, mejorando así el rendimiento general del sistema](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).
* **[Facilidad de prueba:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs)** [La segregación de responsabilidades facilita la creación de pruebas unitarias y de integración](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs).
* **[Escalabilidad:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs)** [Permite escalar las operaciones de lectura y escritura de manera independiente, lo que es beneficioso para sistemas con cargas desiguales entre estas operaciones](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs).

**Clean Architecture:**

* **[Independencia del framework:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [La arquitectura no está acoplada a ningún framework específico, lo que promueve la portabilidad del código](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).
* **[Testabilidad:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs)** [Facilita la escritura de pruebas unitarias al desacoplar la lógica de negocio de la infraestructura y la interfaz de usuario](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs).
* **[Independencia de la interfaz de usuario y la base de datos:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [Permite cambiar la UI o la base de datos sin afectar la lógica de negocio](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).
* **[Regla de dependencia:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [El código fuente solo puede tener dependencias hacia adentro, lo que significa que las capas internas no saben nada sobre las capas externas](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).

**DDD (Domain-Driven Design):**

* **[Enfoque en el dominio del negocio:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [DDD pone el énfasis en la lógica y las reglas del negocio, lo que ayuda a crear software que resuelve problemas del negocio de manera efectiva](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).
* **[Lenguaje ubicuo:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs)** [Fomenta la comunicación entre desarrolladores y expertos del dominio mediante un lenguaje común, reduciendo así los malentendidos](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs).
* **[Diseño modular:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs)** [Promueve un diseño modular del software, facilitando la mantenibilidad y evolución del sistema](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs).
* **[Integración con otras prácticas:](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank)** [DDD se integra bien con otras prácticas y patrones de diseño, como CQRS y Clean Architecture, para formar una metodología robusta](https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs" \t "/home/david/Documentos\\x/_blank).