JUAN SEBASTIAN BARONA – 2266123

CODELAB 3

1. ¿Cuál es el propósito principal de Clean Architecture en el desarrollo de software?

El propósito principal de Clean Architecture es organizar el código de forma que sea fácil de entender, mantener y escalar. Se busca separar claramente las responsabilidades para que los cambios en una parte del sistema (como la base de datos o el framework web) no afecten la lógica de negocio, que es lo más importante.

2. ¿Qué beneficios aporta Clean Architecture a un microservicio en Spring Boot?

En un microservicio hecho con Spring Boot, Clean Architecture ayuda a que el servicio sea:

- -Más mantenible, porque cada cosa está en su lugar.
- -Fácil de probar, ya que la lógica no depende directamente de JPA, RabbitMQ u otros frameworks.
- -Escalable, porque al estar modularizado, es más sencillo agregar nuevas funcionalidades sin romper lo anterior.
- -Además, facilita el trabajo en equipo, ya que es más fácil de entender por otros desarrolladores.

3. ¿Cuáles son las principales capas de Clean Architecture y qué responsabilidad tiene cada una?

Las capas son:

- Entidad (domain model): Aquí van las clases con las reglas del negocio.
- Aplicación (use cases): Contiene la lógica de los casos de uso.
- Interfaces (adapters): Son los controladores, DTOs, mappers, etc., que conectan el mundo externo con los casos de uso.
- Infraestructura (frameworks/drivers): Aquí van cosas específicas del framework, como JPA, Feign, RabbitMQ, etc.

4. ¿Por qué se recomienda desacoplar la lógica de negocio de la infraestructura en un microservicio?

Porque la lógica de negocio es lo más importante del sistema, y si la mezclamos con detalles técnicos (como JPA o Spring), se vuelve más difícil de probar, de mantener y de cambiar. Desacoplarla permite reutilizarla, probarla de forma aislada y evitar que cambie cada vez que cambiamos la tecnología que usamos.

5. ¿Cuál es el rol de la capa de aplicación y qué tipo de lógica debería contener?

La capa de aplicación se encarga de orquestar los casos de uso, es decir, qué pasa cuando alguien quiere crear, actualizar o eliminar algo. Aquí no va la lógica de negocio profunda, sino el flujo general: validar datos, consultar el repositorio, llamar a servicios externos, etc.

6. ¿Qué diferencia hay entre un UseCase y un Service en Clean Architecture?

Un UseCase es una clase que representa una acción del negocio, como "CrearProductoUseCase" o EliminarProductoUseCase".

Un Service puede ser más genérico. En Spring se tiende a meter muchas cosas en los servicios, pero en Clean Architecture los UseCases son más específicos y no deberían tener lógica técnica, solo lógica del dominio o de negocio.

7. ¿Por qué se recomienda definir Repositories como interfaces en la capa de dominio en lugar de usar directamente JpaRepository?

Porque al usar interfaces en el dominio:

- Estamos desacoplando la lógica de negocio de la tecnología.
- Podemos cambiar de base de datos o de forma de persistencia sin tocar el dominio.
- Es más fácil hacer tests unitarios, porque podemos usar mocks en lugar de una base de datos real.

8. ¿Cómo se implementa un UseCase en un microservicio con Spring Boot y qué ventajas tiene?

Un UseCase se puede implementar como una clase en el paquete application que recibe los datos, hace validaciones, llama al repositorio (a través de una interfaz) y retorna la respuesta. Se anota como @Service para que Spring lo maneje, pero no debería contener código de JPA ni dependencias del framework.

La ventaja es que ese UseCase puede usarse desde un controlador REST, desde un listener de RabbitMQ o incluso desde otro microservicio, sin tener que modificarlo.

9. ¿Qué problemas podrían surgir si no aplicamos Clean Architecture en un proyecto de microservicios?

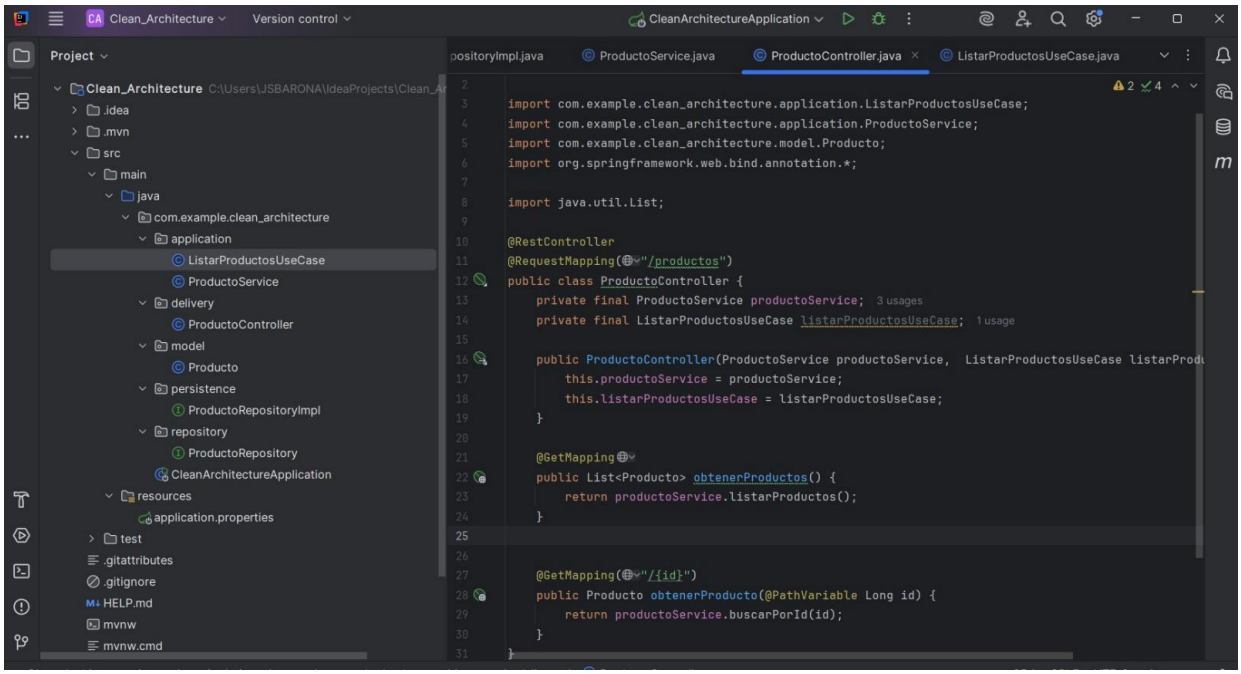
Sin Clean Architecture:

- Terminamos con código desorganizado y acoplado, difícil de mantener.
- Los tests son complicados de escribir porque todo depende de todo.
- Al cambiar algo en la infraestructura (como la base de datos), "rompemos todo".
- Cuesta más trabajo entender el sistema y es difícil crecer o dividir responsabilidades entre el equipo.

10. ¿Cómo Clean Architecture facilita la escalabilidad y mantenibilidad en un entorno basado en microservicios?

Facilita la escalabilidad porque cada microservicio tiene una estructura modular, donde se pueden agregar nuevas funciones o adaptarse a otros contextos sin reescribir todo. Y facilita la mantenibilidad porque:

- El código está bien dividido en capas.
- Podemos cambiar detalles técnicos sin afectar la lógica.
- Se puede trabajar por partes y hacer pruebas unitarias sin depender del resto del sistema.


Clean_Architecture
Version control
CleanArchitectureApplication
Project
Clean_Architecture C:\Users\JSBARONA\IdeaProjects\Clean_Ar
.idea
.mvn
src
main
java
com.example.clean_architecture
application
ListarProductosUseCase
ProductoService
delivery
ProductoController
model
Producto
persistence
ProductoRepositoryImpl
repository
ProductoRepository
CleanArchitectureApplication
resources
application.properties
test
.gitattributes
.gitignore
HELP.md
mvnw
mvnw.cmd
positoryImpl.java
ProductoService.java
ProductoController.java
ListarProductosUseCase.java
2 4
import com.example.clean_architecture.application.ListarProductosUseCase;
import com.example.clean_architecture.application.ProductoService;
import com.example.clean_architecture.model.Producto;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/productos")
public class ProductoController {
private final ProductoService productoService; 3 usages
private final ListarProductosUseCase listarProductosUseCase; 1 usage
public ProductoController(ProductoService productoService, ListarProductosUseCase listarProdu
this.productoService = productoService;
this.listarProductosUseCase = listarProductosUseCase;
}
@GetMapping
public List<Producto> obtenerProductos() {
return productoService.listarProductos();
}
@GetMapping("/{id}")
public Producto obtenerProducto(@PathVariable Long id) {
return productoService.buscarPorId(id);
}
}