Codelab:

Clean Architecture en microservicios Spring Boot

Desarrollo de Software III

Miguel Angel Ceballos Yate -2259619

Docente:

Salazar, ÁLVARO

Sede Tuluá

Junio de 2025

1. ¿Cuál es el propósito principal de Clean Architecture en el desarrollo de software?
El propósito principal de **Clean Architecture** es crear **sistemas de software mantenibles, escalables y fáciles de entender**, mediante una estructura que **separa claramente las responsabilidades** y **reduce el acoplamiento** entre las distintas partes del sistema.

En concreto, Clean Architecture busca:

1. **Independencia** de frameworks, bases de datos o interfaces externas (UI, APIs).
2. **Centralizar la lógica de negocio** en capas internas que no dependan de detalles técnicos.
3. **Facilitar el cambio**: se puede modificar la interfaz, la base de datos o detalles técnicos sin afectar la lógica principal.
4. **Mejorar la prueba del sistema**: al estar desacoplado, se puede probar cada parte de forma aislada.

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

- **Separación de responsabilidades**
  Divide el código en capas (entidades, casos de uso, controladores, adaptadores) que cumplen funciones específicas, lo que evita la lógica mezclada.
- **Independencia del framework**
  Aunque uses Spring Boot, la lógica central (dominio y casos de uso) no depende de él. Esto permite probar, migrar o modificar componentes sin afectar la base del sistema.
- **Alta mantenibilidad**
  Al tener una estructura clara, es más fácil entender, modificar o extender el microservicio sin romper otras partes.
- **Facilidad de pruebas**
  Como la lógica de negocio está desacoplada del acceso a datos o la red, se pueden hacer **pruebas unitarias sin necesidad de mockear Spring** o la base de datos.
- **Escalabilidad del desarrollo**
  Varios equipos pueden trabajar en capas distintas (como lógica de negocio, API REST o infraestructura) sin interferirse.
- **Mejor gestión de dependencias externas**
  Las dependencias (como bases de datos, APIs externas o librerías) se aíslan en los adaptadores, lo que reduce el riesgo de acoplamiento fuerte.
- **Reutilización y consistencia**

Casos de uso bien definidos pueden ser reutilizados por distintas interfaces (por ejemplo, una API REST y una cola de mensajes).

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

- **Dominio (Enterprise Business Rules):** Contiene las entidades y reglas de negocio puras, sin dependencia de frameworks.

- **Aplicación (Application Business Rules):** Define los casos de uso (UseCases) y orquesta la lógica del dominio.

- **Infraestructura (Adapters & Frameworks):** Implementa servicios externos (bases de datos, colas, APIs) y configura tecnologías como JPA o Spring.

- **Presentación (Delivery Mechanisms):** Controladores REST y validaciones, gestiona la entrada/salida del sistema.

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

- **Facilidad para probar**
  Puedes probar la lógica sin necesidad de levantar bases de datos, servidores ni Spring. Solo necesitas probar clases puras.
- **Mantenibilidad y evolución**
  Puedes cambiar la infraestructura (por ejemplo, pasar de MongoDB a PostgreSQL, o de REST a gRPC) **sin tocar la lógica del dominio**.
- **Reutilización**
  La lógica desacoplada puede ser usada en otras interfaces (CLI, batch, eventos) sin duplicar código.
- **Menor acoplamiento a tecnologías específicas**
  Evita que tu sistema dependa fuertemente de un framework o tecnología que puede quedar obsoleto.
- **Claridad arquitectónica**
  Se entiende mejor **qué hace el sistema** (dominio), **cómo lo usa la aplicación** (casos de uso) y **con qué lo conecta** (infraestructura).

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

La capa de aplicación orquesta los flujos del sistema mediante casos de uso. Contiene la lógica de aplicación, como validaciones, cálculos y reglas que coordinan las entidades del dominio, sin preocuparse por cómo se almacenan o muestran los datos.

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

UseCase

- Pertenece a la **capa de Aplicación**.

- Define **qué debe hacer el sistema** desde el punto de vista del negocio (ej. “crear pedido”).

- Orquesta entidades y servicios, pero no contiene reglas complejas.

Service

- Puede estar en la **capa de Dominio** (reglas de negocio complejas) o **Infraestructura** (acceso a bases de datos, correos, etc.).

- Define **cómo se hace** una operación (ej. calcular total, enviar email).

- Es usado por los UseCases para cumplir su tarea.

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

Definir repositorios como interfaces en el dominio protege la lógica de negocio del acoplamiento con tecnologías específicas, mejora la mantenibilidad y permite aplicar buenas prácticas de diseño.

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

Un UseCase se implementa como una clase de servicio en la capa de aplicación que orquesta el acceso a los repositorios del dominio. Ventajas:

- Aisla la lógica de negocio.
- Facilita cambios en reglas sin afectar controladores o infraestructura.
- Favorece pruebas unitarias.
- Mejora la organización del código por responsabilidades.

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

- Acoplamiento entre capas (por ejemplo, lógica de negocio mezclada con controladores o repositorios).
- Dificultad para realizar pruebas.
- Problemas al cambiar la base de datos o la tecnología.
- Menor mantenibilidad y escalabilidad.
- Riesgo de errores al introducir nuevas funcionalidades.

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

Clean Architecture permite que cada microservicio tenga una estructura modular, desacoplada y bien organizada. Esto hace que cada parte del sistema pueda escalar, testearse, desplegarse y mantenerse de forma independiente, lo cual es clave en arquitecturas distribuidas.