Clean Architecture en Microservicio
2259519 - Juan Pablo Escobar

Preguntas:

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

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

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

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

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

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

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

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

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

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

Solución:

a. El propósito principal de Clean Architecture es crear software que sea independiente de frameworks, bases de datos y interfaces externas. Busca separar las preocupaciones del negocio de los detalles técnicos, organizando el código en capas concéntricas donde las dependencias apuntan hacia adentro, hacia el dominio central. Esto permite que el código pueda ser evaluado de manera más sencilla, sea mantenible y flexible ante cambios en la infraestructura o tecnologías externas.

b. Clean Architecture aporta una mayor capacidad de análisis al permitir pruebas unitarias sin dependencias externas, facilita el mantenimiento al tener responsabilidades bien definidas, mejora la flexibilidad para cambiar tecnologías sin afectar la lógica de negocio, reduce el acoplamiento entre componentes, y hace el código más legible y comprensible. Además, facilita la evolución independiente del microservicio y su integración con otros servicios, haciendo su conexión con el resto de la aplicación un trabajo más sencillo.

c. **Las principales capas son:** La capa de Dominio que contiene entidades de negocio, reglas de negocio fundamentales, interfaces de repositorios y Value Objects.
La capa de Aplicación que incluye casos de uso, orquestación de la lógica de negocio, DTOs de entrada y salida e interfaces de servicios externos.
Una capa de Infraestructura con implementación de repositorios, configuración de base de datos, clientes HTTP y mappers entre entidades y DTOs.
Y la capa de Presentación que maneja controladores REST, manejo de excepciones, validaciones de entrada y serialización/deserialización.

d. Debido a que este es crucial ya que permite el testing unitario sin dependencias externas, facilita cambios tecnológicos sin afectar reglas de negocio, mejora la portabilidad del código, y hace que el sistema sea más resiliente a cambios en la infraestructura. En resumidas cuentas, nos permite cambiar bases de datos, frameworks o servicios externos sin tocar la lógica central del negocio.

e. La capa de aplicación es la que orquesta los casos de uso, siendo la que contiene casos de uso específicos del negocio, coordinación entre diferentes entidades de dominio, manejo de transacciones, validaciones de flujo, y transformación de datos entre capas. Esta capa no debe contener las reglas de negocio puras ni detalles de infraestructura y actúa como coordinadora entre el dominio y las capas externas.

f. Un “UseCase” representa una acción específica del usuario con entrada y salida claramente definidas, es decir, es atómico (que posee una sola responsabilidad). Por su parte, un “Service” agrupa funcionalidades relacionadas, que pueden contener múltiples operaciones. Se resuelve en que los “UseCases” son más granulares y expresivos sobre la intención del negocio, mientras que los “Services” son más amplios en su alcance.

g. Es recomendable definir repositories como interfaces en dominio ya que este aplica inversión de dependencias donde el dominio no depende de Spring Data, además de mejorar la capacidad de evaluación al permitir crear implementaciones mock fácilmente, proporcionando flexibilidad para cambiar de JPA a otra tecnología sin llegar a afectar el dominio, y mantiene la abstracción donde el dominio define qué necesita y no el cómo se implementa.

h. “UseCase” está implementado como una clase que encapsula una operación específica del negocio, utilizando inyección de dependencias de Spring para recibir los repositorios y servicios necesarios. Algunas de las ventajas que incluyen son un fácil testing con mocks, código expresivo y claro, reutilizable desde diferentes controladores, y separación clara de responsabilidades.

i. Cuando no se emplea Clean Architecture, surgen problemas como código acoplado (Básicamente, los cambios hechos en la infraestructura afectan el negocio), además de proveer testing complejo que necesita base de datos para probar lógica.
   Tiene un mantenimiento costoso gracias a código mezclado y difícil de entender, añadiendo dificultad para escalar los equipos de desarrollo, y generando una resistencia a los cambios tecnológicos. También se complica la evolución independiente de cada uno de los microservicios y la reutilización de sus componentes.

j. En el caso de microservicios, Clean Architecture facilita la escalabilidad permitiendo que los equipos independientes trabajen en diferentes capas, generando un deployment independiente de componentes con una optimización específica por capa y scaling horizontal más sencillo. Para el caso de la mantenibilidad proporciona código organizado y predecible permitiendo testing automatizado más efectivo, además de un refactoring seguro con tests de regresión, documentación implícita a través de la estructura, y onboarding rápido de nuevos desarrolladores.