CUESTIONARIO TEORICO

Nombre:

Juan Carlos Berdugo Gómez

FABRICA DE SOFTWARE

Dirigido:

JEFE E INSTRUCTORES

CENTRO DE GESTION DE MERCADOS LOGISTICA Y TECNOLOGIAS DE LA INFORMACION

SECCION A:

1. Explique los 4 layers principales de Clean Architecture y su responsabilidad

Clean Architecture se basa en la separación de responsabilidades en capas concéntricas. Las 4 capas principales son:

- **Entities (Entidades):**
    - Contienen las reglas de negocio más generales y duraderas.
    - Son independientes de frameworks, bases de datos o interfaces de usuario.
    - Ejemplo: Una clase User con reglas como isAccountActive().
- **Use Cases (Casos de uso):**
    - Coordinan las acciones del sistema de acuerdo a reglas de aplicación.
    - Orquestan la interacción entre entidades y controladores.
    - Ejemplo: RegisterUser o ProcessOrder.
- **Interface Adapters (Adaptadores de Interfaz):**
    - Transforman datos entre el formato interno (entidades) y el externo (HTTP, UI, DB).
    - Contienen controladores, presentadores y gateways.
    - Ejemplo: UserController, OrderRepositoryImpl.
- **Frameworks and Drivers (Frameworks y controladores externos):**
    - Aquí residen tecnologías externas como frameworks web, bases de datos o UI.
    - Esta capa puede cambiar sin afectar el dominio.
    - Ejemplo: Express (Node.js), Spring Boot, PostgreSQL.

2. ¿Qué es la Regla de Dependencia en Clean Architecture?

La Regla de Dependencia establece que el código de las capas externas puede depender de las internas, pero nunca al revés. Es decir, las capas de dominio y aplicación no deben depender de detalles de infraestructura o presentación. Esto garantiza independencia, testabilidad y facilidad de mantenimiento.

3. Diferencias entre Entities y Value Objects con ejemplos del proyecto

- **Entities (Entidades):**
  Son objetos con identidad propia y ciclo de vida definido. Suelen tener un identificador único (preferiblemente UUID). Ejemplo:
    - User en userservice/app/domain/entities/user.py representa un usuario con atributos como id, email, nombre, etc.
- **Value Objects (Objetos de Valor):**
  Son objetos que representan un valor y no tienen identidad propia. Son inmutables y se consideran iguales si sus atributos son iguales. Ejemplo:
    - DocumentNumber (si está implementado) para validar y encapsular la lógica de números de documento colombiano.

4. Explique el patrón Repository y su implementación en el proyecto

El patrón Repository abstrae el acceso a los datos, permitiendo que la lógica de negocio no dependa de detalles de almacenamiento.

- **Interface:**
  Definida en userservice/app/domain/repositories/user_repository.py, describe los métodos que debe implementar cualquier repositorio de usuarios (por ejemplo, find_by_email, save, etc.).
- **Implementación:**
  En userservice/app/infrastructure/repositories/user_repository_impl.py, se implementan esos métodos usando una base de datos real (por ejemplo, SQLAlchemy).

Esto permite cambiar la fuente de datos sin afectar la lógica de negocio.

5. ¿Qué son los Use Cases y cuál es su función?

Los Use Cases representan las acciones del sistema desde la perspectiva del negocio. Cada caso de uso encapsula la lógica específica para cumplir una tarea.

Responsabilidades:

- Coordinar entidades, servicios y repositorios.
- Aplicar reglas de negocio específicas de la aplicación.
- No dependen de controladores, frameworks ni UI.

SECCION B:

6. Principios fundamentales de la arquitectura de microservicios

- **Descomposición por dominios:** Cada microservicio representa un dominio de negocio específico y autónomo.
- **Despliegue independiente:** Cada servicio puede ser desplegado, escalado y actualizado de forma aislada.
- **Comunicación ligera:** Los servicios se comunican mediante APIs ligeras, generalmente HTTP/REST o mensajería.
- **Descentralización:** Cada microservicio gestiona su propia base de datos y lógica de negocio.
- **Resiliencia:** Los servicios están diseñados para tolerar fallos y recuperarse de ellos.
- **Automatización:** Uso de CI/CD, pruebas y despliegue automatizado para mantener la calidad y velocidad de entrega.

7. Ventajas y desventajas de microservicios vs monolitos

MICROSERVICIOS	MONOLITOS
VENTAJAS	VENTAJAS
escalabilidad independiente por componente.	Simplicidad en el desarrollo y despliegue inicial.
Mejor mantenibilidad y separación de responsabilidades.	Menor complejidad de comunicación interna.
Despliegue y desarrollo desacoplado por equipos.	Más fácil de depurar en etapas tempranas.
DESVENTAJAS	DESVENTAJAS
Mayor complejidad operativa y de infraestructura.	Dificultad para escalar partes específicas.
Despliegue y desarrollo desacoplado por equipos.	Riesgo de acoplamiento excesivo.
Mayor resiliencia ante fallos parciales.	Despliegues más riesgosos y lentos.

8. ¿Cómo se comunican los microservicios en este proyecto?

En este proyecto, los microservicios se comunican principalmente a través de APIs HTTP REST, utilizando endpoints bien definidos y gestionados por el API Gateway. Cada servicio expone su propia API y el gateway enruta las solicitudes externas e internas según corresponda. La comunicación es síncrona y se realiza mediante peticiones HTTP, siguiendo las mejores prácticas de desacoplamiento y versionado de APIs.

9. ¿Qué es el API Gateway y cuál es su función?

El API Gateway es un componente central que actúa como punto de entrada único para todas las solicitudes externas hacia los microservicios. Sus funciones principales son:

- Enrutamiento de solicitudes a los microservicios correctos.
- Autenticación y autorización centralizada.
- Agregación de respuestas de varios servicios.
- Manejo de políticas de seguridad, rate limiting y logging.
- Simplificación del consumo de APIs para los clientes.

10. ¿Qué son los health checks y por qué son importantes?

Los health checks son mecanismos o endpoints que permiten verificar si una aplicación o servicio está funcionando correctamente. Son importantes porque ayudan a detectar fallos de manera temprana, permiten a sistemas de orquestación (como Docker Compose o Kubernetes) reiniciar servicios caídos automáticamente y facilitan el monitoreo de la salud general del sistema, mejorando la confiabilidad y disponibilidad en producción.

REFERENCIAS:

Transparity. (2024, 29 agosto). Clean Architecture Layers | Guide & Benefits | Transparity. *Transparity*. https://www.transparity.com/app-innovation/clean-architecture-layers-what-they-are-and-the-benefits/

DrunknCode. (2025, 23 enero). Clean Architecture: Simplified and In-Depth Guide - DrunknCode - Medium. *Medium*. https://medium.com/@DrunknCode/clean-architecture-simplified-and-in-depth-guide-026333c54454

Mhodroid. (2025, 3 junio). *Clean Architecture en Android y diferentes formas de aplicarlo*. https://platzi.com. https://platzi.com/blog/clean-architecture-en-android/

Ipek, I. (2024, 27 noviembre). UseCase Red Flags and Best Practices in Clean Architecture. *Medium*. https://engineering.teknasyon.com/usecase-red-flags-and-best-practices-in-clean-architecture-76e2f6d921eb

Azevedo, D. (2024, 6 octubre). *3 - Clean Architecture: understanding use cases*. DEV Community. https://dev.to/dazevedo/3-clean-architecture-understanding-use-cases-2a55

What Is an Adaptive Interface? (s. f.). Monitask. https://www.monitask.com/en/business-glossary/adaptive-interface

What are Adaptive User Interfaces? (2017, 18 diciembre). GPII DeveloperSpace. https://ds.gpii.net/content/what-are-adaptive-user-interfaces

Arkano. (2024, 17 julio). El Framework I.D.E.A: un driver clave para tener impacto. *Arkano - Unique solutions for Unique clients*. https://arkanosoft.com/es/el-framework-i-d-e-a-un-driver-clave-para-tener-impacto/

Blog, W. D., & Team, M. D. (2018, 15 agosto). *Introducing Driver Module Framework*. Windows Developer Blog. https://blogs.windows.com/windowsdeveloper/2018/08/15/introducing-driver-module-framework/

Parra, D. (2025, 5 junio). *✳ La Regla de la Dependencia, el anti código spaghetti - 🎮 The power ups 💪*. 🎮 The Power Ups 💪. https://thepowerups-learning.com/regla-de-la-dependencia/

Frye, M. (2023, 29 junio). *6 Fundamental Principles of Microservice Design*. Salesforce. https://www.salesforce.com/blog/microservice-design-principles/

Atlassian. (s. f.). *Comparación entre la arquitectura monolítica y la arquitectura de microservicios*. https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith