PONTIFICIA UNIVERSIDAD JAVERIANA

FACULTAD DE INGENIERÍA

DEPARTAMENTO INGENIERÍA DE SISTEMAS

Documento Investigación Arquitectura Hexagonal

Arquitectura de Software

Grupo 1

2024-2

Introducción

Este documento tiene como objetivo proporcionar un análisis de varias herramientas como son Django, PReact, Cassandra y GraphQL. La idea es explorar no solo su definición y características, sino también su historia, evolución y los casos de uso en los que son más efectivas.

Se verán aspectos clave como las ventajas y desventajas de las herramientas en cuestión y se presentarán ejemplos de éxito en la industria para ilustrar su aplicación práctica. Además, se explorarán relaciones entre las tecnologías designadas, analizando qué tan común es su uso dentro del stack de desarrollo en la actualidad. Por último, todo será implementado en un ejemplo práctico donde se utilizarán las herramientas trabajando en conjunto y se utilizará una arquitectura hexagonal como base para la construcción del backend del ejemplo.

El análisis será complementado por diversas matrices que evalúan la alineación de estas herramientas con principios de diseño y calidad. Se incluirán matrices de comparación entre los principios SOLID, atributos de calidad, tácticas de diseño, y el impacto de estas herramientas en el mercado laboral. También se presentará una revisión de los patrones de arquitectura que guían el uso de las tecnologías dentro de equipos de desarrollo.

Herramientas

1. Django:

Definición

> Django es un framework de desarrollo web de alto nivel escrito en Python, diseñado para facilitar el desarrollo rápido de aplicaciones web, siguiendo el principio de "No repetir código" (DRY) y el enfoque Model-View-Template (MVT).

Características

- ORM Integrado: Facilita la interacción con bases de datos.
- Admin Panel automático: Un panel de administración listo para usar.
- Escalabilidad: Soporta aplicaciones pequeñas y grandes.
- Enrutamiento basado en URL: Define URL amigables y flexibles.

Historia y evolución

Django fue creado en 2003 por desarrolladores de The World Company, un grupo de medios, para gestionar sus sitios de noticias. En 2005 fue liberado como software de código abierto. Desde entonces, ha evolucionado continuamente, introduciendo soporte para versiones más modernas de Python y adaptándose a las tendencias de desarrollo web, como el soporte para REST y GraphQL.

Ventajas y desventajas

Ventajas:

- Desarrollo rápido: Herramientas preconfiguradas para agilizar el proceso.
- Comunidad amplia: Soporte y abundantes recursos.
- Escalabilidad: Capaz de manejar proyectos complejos.
- Seguridad: Características integradas para proteger aplicaciones.

Desventajas:

- Curva de aprendizaje: Su sistema de ORM y estructura puede ser complejo al principio.
- Rigidez: No es tan flexible como microframeworks como Flask.
- Rendimiento: Puede ser menos eficiente que frameworks más ligeros para aplicaciones simples.

Casos de uso

- Desarrollo rápido de aplicaciones web: Django es ideal cuando se necesita construir rápidamente una aplicación robusta con manejo integrado de bases de datos y un panel administrativo.
- Sistemas de autenticación: Implementación rápida de sistemas de usuarios y permisos.
- Aplicaciones escalables: Se usa en proyectos que requieren crecer en funcionalidades sin comprometer el rendimiento.

Casos de aplicación

- **Instagram:** Usó Django en sus primeras fases debido a la capacidad de escalar rápidamente.
- **Mozilla:** Emplea Django en varios proyectos, como MDN (Mozilla Developer Network).

2. Preact

Definición

Preact es una biblioteca de JavaScript ligera que ofrece una alternativa más rápida y compacta a React. Se enfoca en ser pequeña (alrededor de 3KB gzip), con el objetivo de proporcionar la misma funcionalidad esencial que React pero con un tamaño significativamente menor.

Características

- Tamaño reducido: Su pequeño tamaño permite que las aplicaciones carguen más rápido y ocupen menos ancho de banda.
- Compatibilidad con React: Preact es compatible con la mayoría de los ecosistemas y bibliotecas de React, facilitando la transición entre ambas bibliotecas.
- DOM virtual eficiente: Al igual que React, Preact utiliza un DOM virtual para mejorar la eficiencia de las actualizaciones en la interfaz de usuario.
- Componentes modulares: Permite construir interfaces de usuario utilizando componentes reutilizables.
- Simple integración: Se integra fácilmente con otras bibliotecas y frameworks de JavaScript.

Historia y evolución

Preact fue lanzado en 2015 por Jason Miller como una versión más ligera de React. En sus inicios, el objetivo principal era crear una biblioteca rápida y con menor tamaño para proyectos donde React podría considerarse excesivo. Con el tiempo, Preact ha evolucionado para mantener su promesa de velocidad y eficiencia, agregando compatibilidad con nuevas características de React.

Ventajas y desventajas

Ventajas:

- Rendimiento: La velocidad y eficiencia de Preact lo hacen ideal para aplicaciones de alto rendimiento.
- Tamaño pequeño: Ideal para aplicaciones móviles o sitios web donde el peso del bundle es crucial.
- Compatibilidad: Puede aprovechar muchas bibliotecas del ecosistema React.

Desventajas:

- Funcionalidades avanzadas de React: Algunas de las características más recientes de React, como los hooks o el contexto, pueden no estar completamente disponibles o tener pequeñas diferencias en Preact.
- Comunidad más pequeña: Aunque está creciendo, la comunidad de Preact es más pequeña que la de React, lo que puede significar menos recursos y soporte.

Casos de Uso

- Aplicaciones web ligeras: Preact es perfecto para sitios web o aplicaciones que requieren tiempos de carga rápidos y optimización en el uso de recursos.
- Proyectos donde se busca minimizar el tamaño del bundle: En plataformas donde la reducción de tiempo de carga es crucial, como en dispositivos móviles o en regiones con conexiones lentas.
- Transición de aplicaciones pequeñas de React a una versión más ligera.

Casos de Aplicación

- Pinterest: Pinterest utilizó Preact para mejorar el rendimiento de su versión móvil, logrando tiempos de carga más rápidos sin sacrificar la funcionalidad.
- Uber: Uber utilizó Preact en su sistema de gestión de conductores para proporcionar una experiencia fluida con tiempos de respuesta más cortos.
- Smashing Magazine: Utilizó Preact para reducir el tiempo de carga y mejorar el rendimiento en dispositivos móviles.

3. Cassandra

Definición

Apache Cassandra es una base de datos distribuida NoSQL, diseñada para manejar grandes cantidades de datos estructurados a través de múltiples servidores, sin un solo punto de falla. Ofrece una alta disponibilidad y escalabilidad horizontal.

Características

- Distribución y replicación: Los datos se replican automáticamente entre nodos en diferentes centros de datos.
- Alta escalabilidad: Permite agregar más nodos sin comprometer el rendimiento.
- Sin un solo punto de falla: Los nodos son iguales, lo que elimina riesgos de fallos críticos.
- CQL (Cassandra Query Language): Un lenguaje similar a SQL para interactuar con la base de datos.
- Alto rendimiento en lecturas y escrituras: Optimizado para aplicaciones con grandes volúmenes de transacciones de datos.

Historia y evolución

Cassandra fue desarrollada en Facebook en 2008 para el sistema de mensajería de la red social, y luego fue liberada como un proyecto de código abierto de Apache. Desde entonces, ha sido adoptada por muchas empresas tecnológicas para gestionar grandes volúmenes de datos en tiempo real, y ha evolucionado para mejorar su rendimiento, facilidad de uso y capacidad de escalado.

Ventajas y desventajas

Ventajas:

- Alta disponibilidad y tolerancia a fallos: Resiste caídas de nodos sin perder acceso a los datos.
- Escalabilidad horizontal: Permite agregar nodos sin interrupción del servicio.
- Rendimiento de escritura: Ideal para aplicaciones que requieren un gran volumen de escritura.
- Distribución geográfica: Maneja eficientemente la replicación de datos entre diferentes centros de datos.

Desventajas:

- Consistencia eventual: Puede no ser ideal para aplicaciones que requieran una consistencia estricta de los datos.
- Complejidad operativa: Configuración y mantenimiento pueden ser más complicados que bases de datos tradicionales.
- Limitaciones en consultas complejas: Carece de soporte nativo para operaciones como uniones (joins) y subconsultas.

Casos de Uso

- Aplicaciones con altos volúmenes de escritura: Ideal para sistemas de logging o monitoreo, donde se necesita almacenar y procesar rápidamente grandes cantidades de datos.
- Sistemas distribuidos globalmente: Cassandra es óptima para aplicaciones que requieren replicación de datos en múltiples ubicaciones geográficas.
- Aplicaciones de IoT: Debido a su capacidad para gestionar ingestas masivas de datos y consultas rápidas, es ideal para dispositivos IoT.

Casos de aplicación

- Netflix: Utiliza Cassandra para almacenar y gestionar grandes volúmenes de datos sobre usuarios, preferencias y visualizaciones en tiempo real.
- Instagram: Almacena datos de usuario y publicaciones de manera distribuida con Cassandra para garantizar alta disponibilidad y escalabilidad.
- Uber: Usa Cassandra para su sistema de monitoreo de eventos y datos de telemetría, debido a su alta capacidad de escritura y recuperación rápida de datos.

4. GraphQL

Definición

GraphQL es un lenguaje de consulta para APIs, que permite a los clientes solicitar únicamente los datos que necesitan. Fue desarrollado por Facebook en 2012 y lanzado públicamente en 2015. A diferencia de REST, donde se hacen múltiples llamadas a diferentes endpoints, con GraphQL es posible obtener todos los datos requeridos en una sola consulta.

Características

- Consultas específicas: Los clientes pueden definir exactamente qué datos necesitan, reduciendo la sobrecarga de datos innecesarios.

- Esquema de tipos fuertemente tipado: GraphQL utiliza un esquema fuertemente tipado que describe los tipos de datos disponibles y sus relaciones.
- Consultas, mutaciones y suscripciones: Soporta tres operaciones principales: consultas para obtener datos, mutaciones para modificarlos y suscripciones para escuchar cambios en tiempo real.
- Flexibilidad en los endpoints: En lugar de múltiples endpoints como en REST, GraphQL utiliza un único endpoint para todas las operaciones.
- API autodocumentada: Gracias a su esquema tipado, los desarrolladores pueden ver de manera clara qué datos y operaciones están disponibles sin necesidad de documentación externa.

Historia y evolución

GraphQL fue creado por Facebook para mejorar la eficiencia de las consultas de datos en sus aplicaciones móviles. La necesidad surgió porque REST no era lo suficientemente eficiente para sus grandes y complejas necesidades de datos. En 2015, se hizo open source, y desde entonces ha sido adoptado por grandes compañías y mantenido por la GraphQL Foundation. Ha evolucionado para convertirse en una alternativa poderosa a las APIs REST tradicionales.

Ventajas y desventajas

Ventajas:

- Menos sobrecarga de datos: Los clientes obtienen solo los datos que necesitan, lo que reduce el tamaño de las respuestas y optimiza el rendimiento.
- Un único endpoint: Simplifica las interacciones cliente-servidor con un único punto de acceso para todas las operaciones.
- Autodocumentación y facilidad de uso: Los esquemas hacen que las APIs sean fáciles de entender y documentar.
- Compatibilidad con el tiempo real: Con las suscripciones, es ideal para aplicaciones que necesitan actualizaciones en tiempo real.

Desventajas:

- Curva de aprendizaje: Para desarrolladores acostumbrados a REST, aprender GraphQL puede implicar un cambio de paradigma.

- Complejidad en el servidor: Aunque es flexible para el cliente, puede aumentar la complejidad en el backend, ya que el servidor necesita manejar consultas dinámicas.
- Problemas de optimización: Consultas complejas pueden generar problemas de rendimiento si no se gestionan adecuadamente.

Casos de Uso

- Aplicaciones con datos complejos y variados: Ideal para aplicaciones con relaciones de datos complejas o donde se necesita optimización en la obtención de datos.
- Aplicaciones móviles o de baja conectividad: GraphQL minimiza las consultas y reduce la sobrecarga, ideal para dispositivos móviles o en entornos de red limitados.
- Sistemas con múltiples fuentes de datos: Permite orquestar diferentes fuentes de datos (bases de datos, servicios de terceros, etc.) a través de un solo endpoint.

Casos de Aplicación

- GitHub: GitHub implementó GraphQL para su API v4, lo que permitió a los usuarios realizar consultas más precisas y reducir la carga de datos.
- Shopify: Shopify utiliza GraphQL para ofrecer a sus clientes una API más flexible y eficiente, mejorando el rendimiento y la personalización de su plataforma.
- Twitter: Twitter ha utilizado GraphQL para optimizar la entrega de datos en su plataforma móvil, mejorando la velocidad y la experiencia del usuario.

5. Arquitectura Hexagonal

La Arquitectura Hexagonal proporciona una forma efectiva de construir aplicaciones donde la lógica de negocio se mantiene independiente de los detalles tecnológicos. Con herramientas como Django, Preact, Cassandra y GraphQL, se puede implementar una solución flexible, escalable y desacoplada, donde cada tecnología cumple su rol sin generar dependencias fuertes con el core de la aplicación.

Definición

La arquitectura hexagonal, también conocida como Arquitectura de Puertos y Adaptadores, es un estilo de arquitectura de software diseñado para separar el núcleo lógico de una aplicación (la lógica de negocio) de los sistemas externos con los que interactúa (bases de datos, interfaces de usuario, APIs, etc.). Se estructura en torno a un núcleo central (la aplicación) y diversos "adaptadores" que permiten interactuar con servicios externos, encapsulando las dependencias externas y permitiendo que el sistema sea más flexible, mantenible y adaptable a cambios.

Características

1. Desacoplamiento: Aisla la lógica de negocio de las implementaciones tecnológicas, facilitando cambios en una sin afectar a la otra.
2. Independencia de Infraestructura: El núcleo no depende de la base de datos, frameworks o UI, lo que permite cambiar estas tecnologías sin modificar la lógica principal.
3. Facilidad para Testing: Al desacoplar las dependencias externas, es más sencillo realizar pruebas unitarias y de integración en la lógica de negocio.
4. Adaptadores: Los adaptadores se encargan de la interacción entre el núcleo y los sistemas externos (entradas o salidas). Ejemplos incluyen bases de datos, APIs, interfaces de usuario, etc.
5. Puertos: Define interfaces (puertos) que permiten la entrada y salida de datos entre el núcleo y los adaptadores. Los puertos pueden ser interfaces de entrada (comandos) o salida (consultas).

Historia y Evolución

La arquitectura hexagonal fue propuesta por Alistair Cockburn en 2005 como una respuesta a la complejidad derivada de la alta dependencia que muchas aplicaciones tenían de los frameworks y tecnologías externas. La idea era hacer el software más tolerante a cambios, permitiendo que la lógica central no estuviera acoplada a detalles externos como bases de datos o interfaces de usuario.

Con el tiempo, este enfoque ha evolucionado y se ha integrado con otros patrones de diseño y arquitecturas como DDD (Domain-Driven Design) y CQRS (Command Query Responsibility Segregation), haciéndola popular en sistemas que requieren alta escalabilidad, flexibilidad, y pruebas automatizadas robustas.

Ventajas y Desventajas

Ventajas:

- Mantenibilidad: Al separar el núcleo de las dependencias externas, es más fácil modificar o reemplazar componentes sin afectar la lógica de negocio.

- Escalabilidad: Facilita la adición de nuevas entradas o salidas (nuevos adaptadores) sin alterar el núcleo.
- Flexibilidad tecnológica: Puedes cambiar o actualizar tecnologías (bases de datos, frameworks) sin reescribir la lógica del negocio.
- Facilidad para realizar pruebas: Al poder aislar la lógica del negocio, es más sencillo realizar pruebas unitarias sin depender de servicios externos.
- Modularidad: Fomenta la creación de un sistema modular, lo que facilita su evolución.

Desventajas:

- Complejidad inicial: Implementar esta arquitectura requiere un esfuerzo extra, sobre todo en proyectos pequeños, donde puede parecer una sobrecarga innecesaria.
- Curva de aprendizaje: Para equipos no familiarizados con el patrón, puede haber una curva de aprendizaje considerable.
- Mayor número de capas: Introduce más capas y componentes en el sistema, lo que puede aumentar la complejidad si no se gestiona adecuadamente.

Casos de Uso

La arquitectura hexagonal es útil en escenarios donde:

- Se requiere un alto nivel de independencia tecnológica: Por ejemplo, aplicaciones que necesitan ser portadas a diferentes plataformas o trabajar con múltiples bases de datos o sistemas externos.
- Proyectos a largo plazo: Ideal para sistemas que van a evolucionar durante varios años y que, por lo tanto, requerirán cambios frecuentes en la infraestructura tecnológica.
- Desarrollo orientado a microservicios: Permite desacoplar las dependencias externas, facilitando la escalabilidad y flexibilidad que los microservicios requieren.
- Pruebas exhaustivas: Proyectos donde es importante aislar la lógica de negocio para realizar pruebas unitarias independientes de la infraestructura.

Casos de Aplicación

1. Spotify: Utiliza una arquitectura hexagonal para desacoplar su lógica de negocio de sistemas externos como bases de datos y servicios de streaming. Esto les permite evolucionar su aplicación sin interrupciones en sus servicios.
2. Alipay: La plataforma de pagos Alipay, utilizada por millones de usuarios, implementa una arquitectura hexagonal para asegurar que los cambios en sus

interfaces de usuario o servicios externos no afecten el procesamiento central de pagos.

3. Mercado Libre: Ha adoptado la arquitectura hexagonal para mantener la lógica de negocio independiente de los servicios de pagos, logística y bases de datos, permitiendo la rápida adopción de nuevas tecnologías en distintas regiones.
4. Twitch: Utiliza una arquitectura hexagonal para gestionar su infraestructura de transmisión de video, lo que les permite integrar nuevas soluciones de CDN o proveedores sin afectar la experiencia de los usuarios.

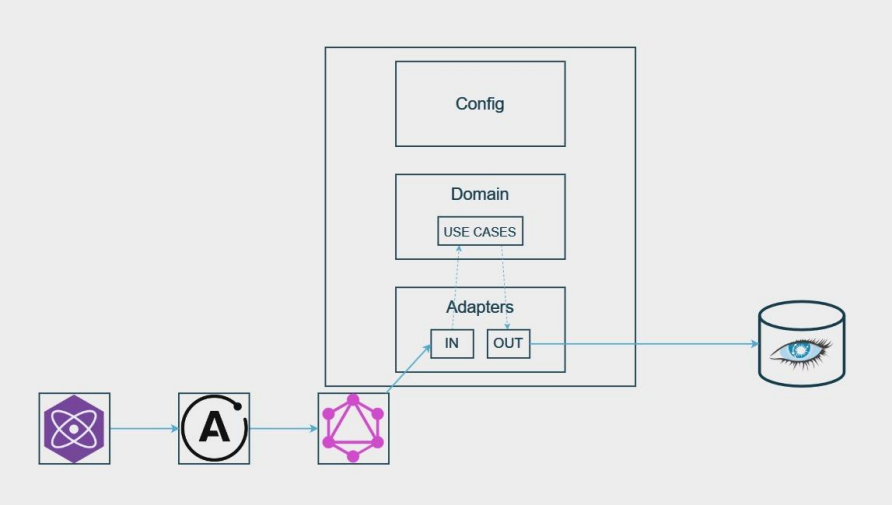

Imagen 1. Arquitectura de Alto Nivel Proyecto

6. Arquitectura Limpia

Definición

La Arquitectura Clean (Arquitectura Limpia) es un enfoque de diseño de software propuesto por Robert C. Martin, también conocido como Uncle Bob, que busca crear sistemas modulares, mantenibles y flexibles. Su principal objetivo es separar las responsabilidades del sistema en capas, de modo que el dominio de la aplicación sea independiente de los detalles de implementación (frameworks, bases de datos, interfaces, etc.).

Características

- Independencia de frameworks: La lógica de negocio no depende de un framework específico, lo que permite que la arquitectura sea adaptable y flexible ante cambios tecnológicos.

- Independencia de la UI (Interfaz de Usuario): Los cambios en la interfaz de usuario no afectan la lógica de negocio.
- Capas concéntricas: La arquitectura se estructura en capas, donde cada una depende solo de la capa interna más cercana. Las capas más internas representan el dominio y las reglas de negocio, mientras que las más externas son detalles de infraestructura.
- Inversión de Dependencias: Las dependencias apuntan hacia el dominio central de la aplicación, siguiendo el principio de inversión de dependencias (DIP), lo que permite que los detalles externos dependan del núcleo de negocio, no al revés.

Historia y Evolución

La Arquitectura Clean fue propuesta por Robert C. Martin en su libro "*Clean Architecture: A Craftsman's Guide to Software Structure and Design*" (2017). Esta propuesta fue una evolución de principios ya existentes en el mundo de la ingeniería de software, como la Arquitectura Hexagonal (Ports and Adapters) de Alistair Cockburn y el enfoque Onion Architecture de Jeffrey Palermo. Todos estos enfoques tienen como base la separación de responsabilidades y la independencia entre el dominio de negocio y la infraestructura.

Martin introdujo la arquitectura Clean para resolver problemas comunes en el desarrollo de software a gran escala, como la rigidez ante cambios, la falta de testabilidad y la dificultad en la evolución de sistemas complejos. A lo largo de los años, ha ganado popularidad en entornos empresariales que buscan longevidad y mantenibilidad de sus productos de software.

Ventajas y Desventajas

Ventajas:

- Testabilidad: La separación de responsabilidades facilita la creación de pruebas unitarias y funcionales, ya que la lógica de negocio está desacoplada de los detalles externos.
- Mantenibilidad: Dado que los cambios en las capas externas no afectan las internas, la arquitectura facilita la evolución y modificación del sistema sin introducir problemas en el núcleo de la aplicación.
- Escalabilidad: La modularidad de las capas permite añadir nuevas funcionalidades o adaptar nuevas tecnologías sin afectar el núcleo del sistema.
- Flexibilidad: Cambiar frameworks, bases de datos o detalles de infraestructura es más sencillo porque estos detalles no están fuertemente acoplados a la lógica de negocio.

Desventajas:

- Curva de aprendizaje: Implementar correctamente la Arquitectura Clean puede requerir una curva de aprendizaje considerable para equipos que no estén acostumbrados a la separación de capas o a la inversión de dependencias.
- Complejidad inicial: En aplicaciones pequeñas o simples, este enfoque puede parecer sobredimensionado debido a la cantidad de capas y abstracciones involucradas.
- Sobrecarga: Para aplicaciones que no tienen requerimientos de escalabilidad o mantenibilidad a largo plazo, la implementación de esta arquitectura puede agregar complejidad innecesaria.

Casos de Uso

- Aplicaciones de larga duración: Ideal para proyectos que van a crecer y evolucionar con el tiempo, ya que proporciona una estructura modular y flexible.
- Sistemas empresariales complejos: Proyectos donde el dominio de negocio es extenso y requiere de una separación clara entre la lógica de negocio y los detalles de infraestructura.
- Proyectos multi-tecnología: Aplicaciones que necesitan soportar múltiples interfaces de usuario (web, móvil, escritorio) o múltiples bases de datos pueden beneficiarse de la flexibilidad que ofrece la Clean Architecture.
- Sistemas altamente testables: Es una excelente opción para aquellos proyectos que requieren de pruebas unitarias y funcionales extensas, debido a su clara separación de responsabilidades.

Casos de Aplicación

- Android: Google ha promovido el uso de Clean Architecture en el desarrollo de aplicaciones Android para mejorar la mantenibilidad y la separación de las capas de presentación y lógica de negocio.
- Fitbit: La empresa ha utilizado la Arquitectura Clean para separar su lógica de negocio de los detalles de la infraestructura, como la sincronización de datos y la integración con dispositivos móviles.
- Proyectos SaaS: Empresas que desarrollan productos de software como servicio han adoptado la Arquitectura Clean para asegurar que sus productos puedan crecer y escalar sin comprometer la calidad o introducir deuda técnica.

7. Onion Architecture

Definición

La Arquitectura Onion es un enfoque de diseño de software que, como la Arquitectura Hexagonal y la Arquitectura Clean, se basa en el principio de separar las responsabilidades de la aplicación en capas. Cada capa representa un conjunto de reglas o funciones que dependen solo de la capa que le es inmediatamente interna. El núcleo más profundo es el dominio, que contiene las reglas y entidades de negocio más fundamentales. Las capas externas incluyen las interfaces de usuario, servicios y bases de datos, de manera que los detalles de la infraestructura están desacoplados del núcleo.

Características

- Estructura concéntrica: Las capas se organizan de forma concéntrica, con el núcleo de dominio en el centro. Cada capa externa interactúa únicamente con la capa inmediatamente interna, evitando dependencias directas de las capas externas hacia las internas.
- Dependencias hacia el núcleo: Las dependencias van desde las capas externas hacia el núcleo del dominio, siguiendo el principio de inversión de dependencias.
- Desacoplamiento de la infraestructura: Las capas de infraestructura, como bases de datos, servicios externos y frameworks, se sitúan en las capas más externas. Esto asegura que el dominio del negocio no dependa de tecnologías específicas.
- Separación clara del dominio y los detalles: El dominio de negocio es independiente de las interfaces de usuario, bases de datos y otros detalles, permitiendo que los cambios en estas capas externas no afecten el núcleo del sistema.

Historia y Evolución

La Arquitectura Onion fue introducida por Jeffrey Palermo en 2008 como una respuesta a las limitaciones observadas en la Arquitectura en Capas tradicional, donde los detalles de la infraestructura tendían a permear hacia la lógica de negocio. Palermo propuso que la lógica central del negocio debería estar completamente aislada de los detalles técnicos para facilitar la mantenibilidad, escalabilidad y testabilidad del software.

La evolución de la Arquitectura Onion se alinea con otros patrones arquitectónicos que promueven la separación de responsabilidades, como la Arquitectura Hexagonal y la Clean Architecture, todos ellos con el objetivo de desacoplar la lógica de negocio de los aspectos externos de la aplicación.

Ventajas y Desventajas

Ventajas:

- Modularidad: Permite una clara separación entre el núcleo de negocio y las capas externas, lo que facilita el mantenimiento y la extensión del software.
- Testabilidad: Al estar el dominio desacoplado de los detalles de infraestructura, es más fácil realizar pruebas unitarias y de integración.
- Escalabilidad: Al desacoplar el núcleo del sistema de las tecnologías externas, la arquitectura puede adaptarse a nuevas tecnologías sin necesidad de refactorizar el dominio de negocio.
- Reutilización: Las reglas de negocio pueden reutilizarse sin cambios en diferentes aplicaciones o interfaces, ya que no están acopladas a ningún framework o tecnología específica.

Desventajas:

- Curva de aprendizaje: Para equipos que no están acostumbrados a este tipo de arquitecturas, puede ser difícil entender y aplicar correctamente los principios.
- Sobrecarga inicial: En proyectos pequeños o de corta duración, la Arquitectura Onion puede parecer demasiado compleja o innecesaria.
- Abstracción excesiva: En ciertos casos, si no se implementa de manera cuidadosa, puede generar una excesiva cantidad de capas y abstracciones que complican el desarrollo.

Casos de Uso

- Sistemas empresariales de larga duración: Ideal para sistemas que requieren una base sólida que pueda evolucionar y adaptarse a lo largo del tiempo sin comprometer la lógica de negocio.
- Aplicaciones con múltiples interfaces: En casos donde una aplicación necesita tener varias interfaces (por ejemplo, móvil, web, APIs), la Arquitectura Onion permite mantener la lógica de negocio separada y reutilizable.
- Sistemas complejos con lógica de negocio centralizada: Proyectos donde el núcleo de negocio es crítico y debe permanecer estable, mientras que las tecnologías externas pueden cambiar, como ERP o CRM personalizados.

- Proyectos donde la infraestructura cambia frecuentemente: Ideal para entornos donde las tecnologías externas, como bases de datos o servicios web, pueden cambiar con el tiempo sin necesidad de modificar la lógica de negocio.

Casos de Aplicación

- eBay: La plataforma ha usado patrones de arquitecturas como Onion y Hexagonal para gestionar su lógica de negocio de forma independiente de las tecnologías externas, permitiendo la adaptación a nuevas interfaces y la incorporación de servicios sin comprometer la estabilidad de su núcleo de negocio.
- Banca y Finanzas: En este sector, donde la lógica de negocio es altamente crítica, la Arquitectura Onion permite que los sistemas centrales permanezcan estables mientras las tecnologías de interfaz (aplicaciones móviles, portales web) y la infraestructura (bases de datos) evolucionan.
- Sistemas de pagos: En plataformas que gestionan pagos y transacciones, es crucial mantener un núcleo de negocio sólido, mientras que las interfaces de usuario y las integraciones con terceros, como pasarelas de pago, pueden cambiar o ampliarse sin afectar la lógica central.

Aspecto	Hexagonal	Onion	Clean
Estructura	Interfaces internas y externas (puertos y adaptadores) que rodean el núcleo del dominio.	Capas concéntricas que rodean el dominio de negocio.	Capas jerárquicas con enfoque en separar políticas de implementación.
Enfoque principal	Separación clara entre la lógica de negocio y las interacciones con el mundo externo mediante puertos y adaptadores.	Lógica de negocio completamente en el centro, sin permitir que dependencias externas lo permeen.	Divide entre entidades de negocio, casos de uso, controladores y frameworks, con énfasis en separar políticas de implementación.
Interacción con el exterior	Los adaptadores manejan la comunicación con el exterior, aislando la lógica de negocio mediante puertos.	Las capas externas interactúan solo con la capa inmediatamente interna (dependencias en dirección hacia el núcleo).	Los controladores y frameworks externos son tratados como detalles periféricos, sin acceso a la lógica de negocio.
Términos específicos	Puertos y Adaptadores	Capas Concéntricas	Capas de Entidades, Casos de Uso, Controladores

Facilidad de comprender para principiantes	Moderada, debido a los conceptos de puertos y adaptadores.	Relativamente simple por la estructura de capas concéntricas.	Moderada a avanzada por la cantidad de abstracciones.
Organización del código	Código dividido en puertos y adaptadores que interactúan con el dominio.	Código estructurado en capas, desde el dominio en el centro hacia detalles técnicos en las capas externas.	Código organizado en cuatro capas (entidades, casos de uso, controladores y frameworks), con un fuerte énfasis en el desacoplamiento.

8. Conclusión

Las tres arquitecturas comparten la misma filosofía de aislar la lógica de negocio del mundo externo, pero cada una tiene su propio enfoque y ventajas dependiendo del caso de uso.

Hexagonal se destaca cuando se necesita flexibilidad y facilidad de integración con múltiples interfaces externas, ofreciendo un patrón sólido para múltiples puntos de interacción. Por otro lado, Onion es ideal para proyectos donde la lógica de negocio es el aspecto más crítico y debe permanecer inmutable, siendo adecuada para sistemas de larga duración con lógica centralizada. Por último, Clean es más abstracta y detallada, enfocándose en mantener la independencia del framework y la infraestructura, siendo ideal para aplicaciones con lógica de negocio extremadamente compleja.