Decisiones de la arquitectura y su decisión

ARQUITECTURA DE LA APLICACIÓN "OPEN MARKET"

La primera decisión es la de utilizar una arquitectura cliente-servidor en lugar de una ariquitectura monolítica.

Desventajas de una arquitectura monolítica

En el contexto de las plataformas de ventas de productos, la arquitectura monolítica tiene unas desventajes notables como el fuerte acoplamiento de todos los componentes del sistema, esto tiene varias consecuencias en el transcurso de la ejecución del sistema. Uno de los mayores riesgos es el fallo del sistema, ya que al estar fuertemente acomplado, si ocurre un solo fallo , puede afectar a todo el sistema. Uno de estos fallos puede implicar que se tenga que reiniciar todo el sistema y afectaría también el tiempo de recuperación. Solo por mencionar otros problemas de la arquitectura monolítica tenemos:

- Acoplamiento y dependencias: Cualquier cambio en una parte del sistema puede tener impacto en otras áreas, lo que dificulta la evolución y el mantenimiento del sistema.
- Escalabilidad limitada: Al ejecutarse la alquitectura monolítica en una sola instancia, limita su capacidad de escalar, ya que puede empezar incluso a tener límites físico y económicos en el servidor.
- Dificultad para adoptar nuevas tecnologías: La adopción de nuevas tecnologías o frameworks puede ser complicada debido a las dependencias y restricciones existentes en el sistema.

Además, por el modo de trabajo que se necesitaba (de manera colaborativa), diferentes equipos pueden necesitar trabajar en diferentes partes del código simultáneamente. Sin embargo, dado que todo el código está en un solo repositorio y se compila y despliega como una sola unidad, esto puede dificultar el desarrollo colaborativo y aumentar las posibilidades de conflictos en el código.

Ventajas de la arquitectura cliente-servidor

A nivel general en una arquitectura cliente-servidor al separar la lógica del servidor del cliente, es posible escalar de forma independiente cada componente según sea necesario. Esto permite agregar más servidores para manejar la carga y distribuir el procesamiento de manera eficiente, facilita el mantenimiento y la evolución del sistema.

Uno de los mayores principios SOLID que se cunplen al utlizar la arquitectura cliente-servidor es el principio de Responsabilidad Única (Single Responsibility Principle). Al separar las responsabilidades entre el cliente y el servidor, los cambios y actualizaciones en el servidor pueden realizarse sin afectar directamente al cliente. Esto brinda flexibilidad y facilita la introducción de mejoras y correcciones sin interrumpir la operación del sistema en su conjunto. En resumen, estas son algunas ventajas que proporciana esta arquitectura enfocada en la aplicación "Open Market":

- Escalabilidad: Permite escalar tanto el cliente como el servidor de forma independiente. Esto es especialmente útil en una aplicación de compra y ventas, donde la carga puede variar significativamente en diferentes momentos. Se puede escalar el servidor para manejar el aumento de tráfico y agregar más recursos según sea necesario.
- Separación de responsabilidades: El cliente se encarga de la interfaz de usuario y la interacción con el usuario, mientras que el servidor se encarga de la lógica de relacional para la base de datos, el procesamiento de pagos y la gestión de inventario.
- Acceso a datos centralizado: Con una arquitectura cliente-servidor, la base de datos y los datos del sistema se almacenan en el servidor, lo que proporciona un acceso centralizado y consistente a los datos para todos los clientes. Esto garantiza la integridad de los datos y evita problemas de sincronización o inconsistencias.
- Seguridad: Se puede implementar medidas de seguridad en el servidor para proteger los datos confidenciales y asegurar las transacciones. Esto incluye autenticación y control de acceso, lo que proporciona un entorno más

seguro para las transacciones de compra y ventas.
- Integración con sistemas externos: En una aplicación de compra y ventas, la lógica de integración pueden gestionarse en el servidor, manteniendo al cliente más liviano y enfocado en la experiencia del usuario.

DECISIONES DE DISEÑO ARQUITECTÓNICO

Patrones de diseño arquitectónico

Arquitectura en capas (Layered Architecture)

En el contexto de desarrollar una aplicación de compra y venta de productos, con el fin de resolver la organización y la claridad de las responsabilidaes, nos decidimos por usar el patrón de diseño por capas y descartamos la opción de hacer una implementación directa, para lograr una estructura organizada, modular y facilitar la escalabilidad, aceptando que esto conlleva una sobrecarga de comunicación porque al tener capas separadas, se requiere una comunicación constante entre ellas para transmitir datos y cumplir con las solicitudes del sistema.

Arquitectura basada en microservicios (Microservices Architecture)

En el contexto de proporcionar un conjunto específico de funcionalidades a diferentes clases que lo requieran, con el fin de resolver que diferentes clientes puedan interactuar con los mismos microservicios solicitando y consumiendo sus servicios, nos decidimos por usar una arquitectura basada en microservicios, estos microservicios pueden ser vistos como componentes de servidor individuales dentro de una arquitectura cliente-servidor más amplia, y descartamos la opción de hacer una implementación de métodos directos en las clases, para lograr la modificabilidad del sistema cuando se requiera agregar funcionalidades a distintas clases que usen las mismas solicitudes, lo que permite una mayor flexibilidad y escalabilidad en el diseño y desarrollo del sistema, aceptando que esto agrega desafíos en términos de gestión, comunicación entre servicios y complejidad de infraestructura que deben abordarse de manera adecuada para obtener los resultados deseados.

Instanciación de un patrón de diseño arquitectónico

Command

En el contexto de parametrizar los clientes con diferentes solicitudes, con el fin de desacoplar el objeto que solicita una acción del objeto que realiza la acción, nos decidimos por la implementación de un patrón de diseño Command para agregar y remover un producto y agregar una compra y descartamos la opción de hacer los llamados directamente desde el cliente, para lograr las una mayor flexibilidad y extensibilidad en la aplicación, ya que los clientes pueden ser configurados con diferentes comandos y pueden realizar operaciones de deshacer y rehacer en un futuro, aceptando que puede aumentar la complejidad de la arquitectura y la cantidad de clases y objetos involucrados a pesar de realizar solicitudes sencillas porque en un futuro se pueden realizar acciones que permitirían aumentar las funcionalidades de la aplicación, tal vez, violando un principio general como los es YAGNI(You Ain't Gonna Need It), pero que se encuentra dentro de las capacidades y que además nos sirve como objeto de estudio.

Observer

En el contexto de actualizar la información que reciven los clientes, con el fin de permitir una comunicación desacoplada entre los sujetos, nos decidimos por la implementación de un patrón de diseño Observer y descartamos la actualización de las vistas cada vez que se cambie de estado, para lograr mantener informados a los objetos sobre los cambios en el estado de otro objeto, por ejemplo, al agregar un producto y querer editarlo en otro panel ya tener el producto en la vista del panerl de edición, aceptando que puede si no se implementa correctamente, puede haber riesgos de inconsistencia en la notificación de cambios, lo que podría llevar a resultados incorrectos o impredecibles, porque da flexibilidad y desacoplamiento, además de también ser un buen objeto de estudio.
