

13 Patrones de arquitectura en DDD

Arquitectura por capas

La **arquitectura por capas** es un patrón de diseño estructural que organiza el software en distintas capas, cada una con una responsabilidad específica. Este enfoque facilita el desarrollo, el mantenimiento y la escalabilidad de las aplicaciones al separar las diferentes preocupaciones del sistema, como la presentación de la información, la lógica de negocio y el acceso a datos.

1.1 Definición de arquitectura por capas

En la **arquitectura por capas**, el sistema se organiza en **niveles** o **capas** que agrupan funcionalidades relacionadas. Cada capa es responsable de una parte del sistema y se comunica únicamente con la capa inmediatamente superior o inferior. Este enfoque ayuda a reducir el acoplamiento entre diferentes partes de la aplicación, lo que facilita el mantenimiento y la evolución del software.

1.2 Propósito de la arquitectura por capas

El propósito de la arquitectura por capas es promover la **separación de responsabilidades**, lo que permite que cada capa se enfoque en una función específica del sistema. Esto no solo mejora la **modularidad** y la **mantenibilidad** del software, sino que también facilita la **reutilización** de componentes y la realización de pruebas.

- **Separación de responsabilidades:** Cada capa tiene una única responsabilidad, lo que ayuda a aislar los cambios en una parte del sistema sin afectar a las demás. Por ejemplo, cambiar la interfaz de usuario no debería afectar a la lógica de negocio o al acceso a datos.

1.3 Capas comunes en una arquitectura por capas

- **Capa de presentación:**
 - Gestiona la interacción con los usuarios o sistemas externos.
 - Convierte las solicitudes del usuario en comandos que serán procesados por la capa de lógica empresarial.
 - **Ejemplo:** En un sistema web, la capa de presentación incluye los controladores que reciben las solicitudes HTTP y presentan respuestas al cliente.
- **Capa de lógica empresarial (dominio):**
 - Contiene las **reglas de negocio** y define cómo deben operar los diferentes componentes del sistema.
 - Interactúa con la capa de acceso a datos para obtener o modificar información, pero la lógica empresarial no depende de la forma en que se almacenan los datos.

- **Ejemplo:** Servicios o componentes que gestionan transacciones de negocios, como el procesamiento de pedidos o el cálculo de impuestos.
- **Capa de acceso a datos:**
 - Responsable de interactuar con las bases de datos o cualquier otra forma de almacenamiento.
 - Aísla los detalles de persistencia de los demás componentes del sistema, proporcionando una interfaz que permite acceder a los datos sin exponer la infraestructura subyacente.
 - **Ejemplo:** Repositorios que utilizan **Spring Data JPA** para acceder a una base de datos relacional.

1.4 Ventajas de la arquitectura por capas

- **Desacoplamiento:** Las capas permiten una clara separación de responsabilidades, lo que mejora la modularidad y facilita el trabajo en equipo. Los desarrolladores pueden trabajar en capas individuales sin afectar otras partes del sistema.
- **Facilidad para realizar pruebas:** Cada capa puede probarse de forma independiente. Por ejemplo, la capa de lógica de negocio puede probarse mediante **pruebas unitarias** sin preocuparse por la capa de presentación o el acceso a datos.
- **Reutilización:** Los componentes dentro de una capa pueden ser reutilizados en diferentes partes del sistema. Por ejemplo, un servicio de cálculo de impuestos en la capa de lógica empresarial puede ser utilizado tanto en la interfaz de usuario como en una API.
- **Mantenimiento:** Al estar las responsabilidades claramente definidas en cada capa, es más fácil realizar cambios o mejoras en una capa específica sin afectar a las demás, lo que mejora la mantenibilidad del sistema.

1.5 Desventajas de la arquitectura por capas

- **Aumento del acoplamiento entre capas:** Aunque cada capa está desacoplada de las demás, el flujo de comunicación entre las capas puede generar dependencias fuertes. Si se cambia la forma en que una capa interactúa con otra, es posible que se requieran cambios en varias partes del sistema.
- **Latencia en aplicaciones grandes:** En sistemas grandes y complejos, la comunicación constante entre las capas puede introducir latencia, ya que los datos deben pasar por múltiples capas antes de llegar a su destino final.
- **Rígida estructuración:** La arquitectura por capas puede ser rígida si no se maneja adecuadamente. Por ejemplo, si el sistema requiere una interacción directa entre la presentación y la capa de acceso a datos (bypass de la capa de lógica de negocio), esta arquitectura no es la más adecuada.

Capa de presentación

La **capa de presentación** es la capa responsable de interactuar directamente con los **usuarios** o **sistemas externos**. Es la capa más externa en una arquitectura por capas y su principal responsabilidad es gestionar las solicitudes entrantes, ya sea de interfaces de usuario, APIs u otros sistemas, y transformarlas en comandos que serán procesados por la capa de lógica empresarial.

En una aplicación basada en **Spring Boot**, la capa de presentación generalmente se implementa mediante **controladores** (por ejemplo, controladores REST en APIs) que reciben las solicitudes HTTP, procesan los datos, y luego delegan la lógica de negocio a la capa subyacente.

2.1 Propósito de la capa de presentación

El principal propósito de la **capa de presentación** es actuar como un intermediario entre el usuario o sistema externo y la lógica del negocio. Gestiona la **entrada y salida de datos** de manera que sean comprensibles para los usuarios, y luego los traduce en comandos o consultas que pueden ser manejados por la capa de lógica empresarial.

- **Gestión de la interfaz de usuario:** En aplicaciones con interfaz de usuario, la capa de presentación puede incluir componentes como vistas, controladores y modelos que gestionan cómo se presenta la información a los usuarios.
- **Exponer APIs:** En aplicaciones sin una interfaz de usuario (como servicios web), la capa de presentación se encarga de exponer **APIs REST** u otras interfaces que permiten que los sistemas externos interactúen con la aplicación.

2.2 Responsabilidades de la capa de presentación

La capa de presentación tiene varias responsabilidades clave:

- **Recepción de solicitudes:** La capa de presentación recibe las solicitudes del usuario o del sistema externo, ya sea a través de una interfaz de usuario o una API.
- **Validación de datos de entrada:** Antes de delegar la lógica a la capa de negocio, la capa de presentación valida los datos de entrada para asegurar que sean correctos y estén en el formato adecuado.
- **Transformación de datos:** Convierte las solicitudes externas en comandos o consultas que pueden ser procesados por la capa de lógica empresarial.
- **Manejo de respuestas:** Después de que la lógica empresarial ha procesado la solicitud, la capa de presentación transforma los resultados en un formato adecuado para devolver al usuario o sistema externo (como un JSON en una API REST).

2.3 Ejemplos en Spring Boot

En Spring Boot, la capa de presentación generalmente se implementa utilizando **controladores** que manejan las solicitudes HTTP y responden a ellas. Los controladores REST son un ejemplo típico de un componente de la capa de presentación.

- **Controladores REST:** Un controlador REST en Spring Boot gestiona las solicitudes entrantes (GET, POST, PUT, DELETE), valida los datos de entrada, y delega la lógica de negocio a los servicios correspondientes en la capa de lógica empresarial.

Ejemplo de un controlador REST en Spring:

```
@RestController
@RequestMapping("/pedidos")
```

```

public class PedidoController {

    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido pedido =
gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
        return ResponseEntity.ok(pedido);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
        Pedido pedido = gestorDePedidos.obtenerPedido(id);
        return ResponseEntity.ok(pedido);
    }
}

```

En este ejemplo, el controlador recibe una solicitud para crear o consultar un pedido y delega la operación a la capa de lógica empresarial (**GestorDePedidos**). El controlador también se encarga de manejar la respuesta y devolverla en el formato adecuado (por ejemplo, JSON).

2.4 Validación de datos en la capa de presentación

La **validación** es una de las responsabilidades más importantes de la capa de presentación. Antes de pasar los datos a la lógica empresarial, la capa de presentación debe validar que la entrada sea correcta. Spring Boot proporciona herramientas como **@Valid** y **@RequestBody** para gestionar esta validación de manera efectiva.

Ejemplo de validación de entrada en Spring Boot:

```

@PostMapping
public ResponseEntity<Pedido> crearPedido(@Valid @RequestBody
PedidoRequest request) {
    Pedido pedido = gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
}

```

```
return ResponseEntity.ok(pedido);  
}
```

En este ejemplo, se utiliza la anotación **@Valid** para validar automáticamente los datos del cuerpo de la solicitud antes de delegarlos a la capa de lógica empresarial.

2.5 Transformación de datos de entrada y salida

La capa de presentación también es responsable de transformar los **datos de entrada** (por ejemplo, JSON en una API) en objetos que puedan ser utilizados por la lógica empresarial y, posteriormente, transformar los **resultados** de la lógica de negocio en un formato que pueda ser presentado al usuario o sistema externo.

- **Entrada:** La capa de presentación recibe solicitudes en formatos como JSON, XML, o formularios web, y los convierte en objetos de dominio o comandos.
- **Salida:** Los resultados de la capa de lógica empresarial son transformados en el formato requerido por el cliente o sistema que hizo la solicitud (generalmente JSON en una API REST).

2.6 Manejo de errores en la capa de presentación

Otro aspecto importante de la capa de presentación es el manejo de **errores**. La capa de presentación es responsable de devolver mensajes de error significativos y adecuados cuando ocurre una excepción o cuando los datos proporcionados por el usuario no son válidos.

Manejo de excepciones en Spring Boot:

```
@RestControllerAdvice  
public class PedidoControllerAdvice {  
  
    @ExceptionHandler(PedidoNoEncontradoException.class)  
    public ResponseEntity<String>  
    handlePedidoNoEncontrado(PedidoNoEncontradoException ex) {  
        return  
        ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());  
    }  
}
```

En este ejemplo, se define un manejador de excepciones para capturar errores como un **Pedido no encontrado** y devolver una respuesta adecuada al cliente.

2.7 Desafíos en la capa de presentación

Aunque la capa de presentación tiene varias ventajas, también presenta desafíos específicos que deben ser gestionados adecuadamente:

- **Control de flujo excesivo:** Si la capa de presentación gestiona demasiada lógica, puede llegar a convertirse en una capa compleja y sobrecargada.
- **Manejo de datos de entrada:** Transformar grandes volúmenes de datos o realizar múltiples validaciones puede agregar latencia a la respuesta del sistema si no se maneja correctamente.

Capa de lógica empresarial

La **capa de lógica empresarial**, también conocida como la **capa del dominio**, es el corazón de cualquier aplicación en la **arquitectura por capas**. En esta capa se definen las reglas del negocio, los procesos y las políticas que determinan cómo el sistema debe funcionar. La lógica empresarial es independiente de los detalles técnicos, como la persistencia de datos o la interacción con APIs externas, lo que la convierte en la capa más crítica para garantizar que el sistema cumpla con los requisitos del negocio.

3.1 Propósito de la capa de lógica empresarial

El principal propósito de la **capa de lógica empresarial** es encapsular toda la lógica del negocio, asegurando que las operaciones realizadas en el sistema se alineen con las reglas y procesos del negocio. Esta capa debe estar completamente desacoplada de la infraestructura técnica, como bases de datos, interfaces de usuario o servicios de mensajería.

- **Definir las reglas del negocio:** La capa de lógica empresarial contiene los **Domain Objects**, que representan las entidades y conceptos centrales del negocio. Estos objetos encapsulan tanto los datos como las operaciones que reflejan las reglas del negocio.
- **Orquestar procesos de negocio:** Los **servicios de aplicación** o **casos de uso** en esta capa coordinan las acciones que deben realizarse sobre los Domain Objects, asegurando que se ejecuten de manera correcta y coherente.

3.2 Domain Objects y casos de uso en la capa de lógica empresarial

Dentro de la capa de lógica empresarial, los **Domain Objects** (entidades y objetos de valor) y los **casos de uso** son los principales componentes. Los Domain Objects representan los datos y las reglas del negocio, mientras que los casos de uso orquestan las interacciones entre los objetos y los servicios externos.

- **Domain Objects:** Son las representaciones del negocio dentro del sistema. Estos objetos no solo contienen datos, sino también comportamientos, es decir, las reglas que determinan cómo pueden modificarse sus atributos o interactuar con otros objetos.
- **Casos de uso:** También conocidos como **servicios de aplicación**, los casos de uso representan procesos completos que involucran la coordinación de múltiples Domain Objects. Cada caso de uso es responsable de ejecutar un flujo de negocio

específico, como el proceso de crear un pedido o realizar una transferencia bancaria.

Ejemplo de un servicio de aplicación que maneja un caso de uso:

```
@Service
public class GestorDePedidosImpl implements GestorDePedidos {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public GestorDePedidosImpl(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }

    @Override
    public Pedido obtenerPedido(Long id) {
        return repositorioDePedidos.buscarPorId(id)
            .orElseThrow(() -> new IllegalArgumentException("Pedido
no encontrado"));
    }
}
```

En este ejemplo, el servicio `GestorDePedidosImpl` es un caso de uso que coordina la creación de un pedido y la interacción con el repositorio.

3.3 Independencia de la infraestructura técnica

Una de las características más importantes de la capa de lógica empresarial es que debe estar **desacoplada de la infraestructura técnica**. Esto significa que la lógica del negocio no debe depender de la forma en que los datos se almacenan, ni de cómo se presentan o de cómo interactúa el sistema con APIs externas.

- **Independencia de la base de datos:** Aunque la capa de lógica empresarial puede interactuar con la base de datos, esta interacción debe realizarse a través de puertos y adaptadores (repositorios o servicios), sin que la lógica empresarial tenga conocimiento de cómo se almacenan los datos.

- **Independencia de la presentación:** La lógica empresarial no debería depender de cómo se muestran los datos o cómo se gestionan las solicitudes del usuario. Por ejemplo, la lógica de negocio debe funcionar independientemente de si se trata de una aplicación web, una API REST o una aplicación móvil.

3.4 Ejemplo en Spring Boot

En un proyecto de **Spring Boot**, la capa de lógica empresarial suele implementarse mediante **servicios** que contienen la lógica del negocio. Estos servicios interactúan con los **Domain Objects** y utilizan **repositorios** (adaptadores de salida) para almacenar o recuperar datos.

Ejemplo de servicio de lógica empresarial en Spring:

```
@Service
public class PedidoService {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoService(RepositorioDePedidos repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    public Pedido completarPedido(Long id) {
        Pedido pedido = repositorioDePedidos.buscarPorId(id)
            .orElseThrow(() -> new PedidoNoEncontradoException(id));

        pedido.completarPedido();
        return repositorioDePedidos.guardar(pedido);
    }
}
```

En este ejemplo, **PedidoService** es un servicio que contiene la lógica para completar un pedido. No depende de cómo se gestiona la persistencia de los pedidos; simplemente llama al repositorio para guardar el estado actualizado del pedido.

3.5 Beneficios de la separación de la lógica empresarial

- **Reutilización de lógica:** Al mantener la lógica de negocio independiente de la infraestructura técnica, es posible reutilizar la lógica en diferentes interfaces (por ejemplo, APIs REST, aplicaciones móviles, interfaces gráficas).
- **Facilidad de mantenimiento:** Si las reglas del negocio cambian, la modificación se realiza en un solo lugar: la capa de lógica empresarial, sin necesidad de tocar las otras capas (presentación o acceso a datos).

- **Testabilidad:** La lógica empresarial es más fácil de probar porque no tiene dependencias técnicas que compliquen las pruebas. Las **pruebas unitarias** pueden centrarse en validar que las reglas de negocio se aplican correctamente.

3.6 Desafíos en la capa de lógica empresarial

Aunque la separación de la lógica empresarial presenta varias ventajas, también plantea algunos desafíos:

- **Complejidad en la coordinación:** En sistemas grandes, coordinar correctamente las interacciones entre múltiples Domain Objects y servicios puede volverse complejo.
- **Manejo de transacciones:** La capa de lógica empresarial debe asegurarse de que las transacciones entre múltiples objetos del dominio se gestionen de manera segura y consistente, lo que puede ser complicado en sistemas distribuidos.

Capa de acceso de datos

La **capa de acceso de datos** es responsable de gestionar la interacción entre la aplicación y los sistemas de almacenamiento, como bases de datos, sistemas de mensajería, o cualquier otra fuente de datos externa. Esta capa proporciona una **abstracción** sobre los detalles de cómo se almacenan o recuperan los datos, permitiendo que la lógica empresarial interactúe con los datos sin preocuparse por los detalles técnicos del almacenamiento.

4.1 Propósito de la capa de acceso de datos

El principal propósito de la **capa de acceso de datos** es desacoplar la lógica de negocio de los detalles de persistencia. De esta manera, el sistema puede cambiar de tecnología de almacenamiento o realizar optimizaciones sin afectar el núcleo de la aplicación.

- **Encapsulación del acceso a los datos:** Esta capa oculta los detalles de la tecnología subyacente, como el uso de una base de datos SQL, NoSQL, o un servicio de almacenamiento en la nube.
- **Abstracción:** Proporciona una interfaz o conjunto de métodos que permiten realizar operaciones CRUD (Crear, Leer, Actualizar, Borrar) sobre los datos sin exponer cómo se implementan estas operaciones.

4.2 Responsabilidad de la capa de acceso de datos

La **capa de acceso de datos** tiene la responsabilidad de gestionar todas las operaciones relacionadas con la persistencia de la información. Esto incluye desde la conexión a la base de datos hasta la ejecución de consultas y la transformación de los resultados en objetos del dominio.

- **Operaciones CRUD:** Las operaciones básicas que gestiona esta capa son **crear**, **leer**, **actualizar** y **eliminar** entidades en la base de datos.

- **Interacción con servicios externos:** En algunos casos, la capa de acceso de datos también gestiona la comunicación con servicios externos que proveen datos, como APIs de terceros o sistemas de mensajería.

4.3 Ejemplos de uso en Spring Boot

En **Spring Boot**, la capa de acceso de datos se implementa comúnmente utilizando **repositorios**. Estos repositorios definen métodos que permiten realizar operaciones sobre las entidades, interactuando con la base de datos a través de frameworks como **Spring Data JPA**, **MongoDB**, o cualquier otra tecnología de persistencia.

Ejemplo de un repositorio usando Spring Data JPA:

```
public interface RepositorioDePedidos extends JpaRepository<Pedido, Long> {  
    // Métodos adicionales si es necesario  
    Optional<Pedido> findByClienteId(Long clienteId);  
}
```

En este ejemplo, **RepositorioDePedidos** extiende de **JpaRepository**, lo que le proporciona acceso a las operaciones CRUD estándar, como **save()**, **findById()**, y **deleteById()**. También se puede agregar un método personalizado, como **findByClienteId**, que busca pedidos de un cliente específico.

4.4 Abstracción de la persistencia de datos

La **capa de acceso de datos** debe proporcionar una abstracción clara entre la lógica de negocio y los detalles técnicos de la persistencia. En una arquitectura por capas, la capa de lógica empresarial interactúa con los **repositorios** sin necesidad de saber cómo están implementadas las operaciones de persistencia.

- **Independencia de la base de datos:** Los repositorios y otros componentes de la capa de acceso de datos pueden utilizar cualquier tecnología de almacenamiento sin que esto afecte la lógica de negocio. Esto permite que se cambie la tecnología subyacente (por ejemplo, de una base de datos SQL a una NoSQL) sin tener que modificar el código del dominio.
- **Consultas y optimización:** Aunque la lógica empresarial no debe conocer los detalles de la base de datos, la capa de acceso de datos puede optimizar las consultas y hacer uso de características específicas de la tecnología de almacenamiento para mejorar el rendimiento.

4.5 Ejemplo de interacción entre la lógica empresarial y la capa de acceso de datos

La interacción entre la capa de lógica empresarial y la capa de acceso de datos ocurre mediante **repositorios** o **servicios** que proporcionan métodos para acceder y modificar los datos.

Ejemplo de un servicio que utiliza un repositorio:

```
@Service
public class PedidoService {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoService(RepositorioDePedidos repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    public Pedido obtenerPedido(Long id) {
        return repositorioDePedidos.buscarPorId(id)
            .orElseThrow(() -> new PedidoNoEncontradoException(id));
    }

    public Pedido guardarPedido(Pedido pedido) {
        return repositorioDePedidos.save(pedido);
    }
}
```

En este ejemplo, el servicio `PedidoService` utiliza el repositorio `RepositorioDePedidos` para realizar operaciones de persistencia sin preocuparse por los detalles de cómo se implementa la persistencia. El servicio simplemente llama a los métodos `save()` o `buscarPorId()` para interactuar con los datos.

4.6 Relación con los Domain Objects

La **capa de acceso de datos** maneja la **persistencia de los Domain Objects**, que son los objetos que representan las entidades y objetos de valor en el sistema. Aunque los **repositorios** manejan los detalles de cómo se almacenan estos objetos, la lógica empresarial sigue utilizando los Domain Objects en su estado puro, sin saber nada sobre la tecnología de persistencia.

- **Transformación de datos:** Los repositorios se encargan de transformar los datos de la base de datos en Domain Objects y viceversa, permitiendo que la lógica de negocio trabaje solo con los conceptos del dominio sin preocuparse por las bases de datos.

4.7 Beneficios de la capa de acceso de datos

- **Abstracción y encapsulamiento:** La capa de acceso de datos encapsula todos los detalles relacionados con la persistencia, lo que permite que la lógica empresarial se mantenga limpia y enfocada en las reglas de negocio.

- **Facilidad para cambiar de tecnología:** Al utilizar repositorios que abstraen el acceso a los datos, se facilita la migración de una tecnología de almacenamiento a otra (por ejemplo, cambiar de SQL a NoSQL) sin que la lógica empresarial se vea afectada.
- **Separación de responsabilidades:** Esta capa asegura que las operaciones relacionadas con la persistencia no se mezclen con la lógica de negocio, lo que facilita la mantenibilidad y la escalabilidad del sistema.

4.8 Desafíos en la capa de acceso de datos

Aunque la capa de acceso de datos es fundamental para mantener una buena arquitectura, también presenta algunos desafíos:

- **Optimización de consultas:** En sistemas grandes, puede ser difícil optimizar las consultas y evitar problemas de rendimiento (por ejemplo, consultas N+1). Es necesario gestionar cuidadosamente el acceso a los datos para evitar problemas de eficiencia.
- **Gestión de transacciones:** En algunos casos, la capa de acceso de datos debe gestionar transacciones complejas, asegurando que los cambios se realicen de manera consistente y atómica en toda la aplicación.

Comunicación entre capas

En una **arquitectura por capas**, la comunicación entre las capas es clave para el correcto funcionamiento del sistema. Cada capa tiene responsabilidades bien definidas y debe interactuar con las capas adyacentes de manera controlada para garantizar que se mantenga el **desacoplamiento** y la **modularidad**. La **comunicación entre capas** generalmente se realiza de arriba hacia abajo, con la capa de presentación delegando tareas a la capa de lógica empresarial, y esta última interactuando con la capa de acceso a datos.

5.1 Patrón de comunicación entre capas

El patrón común de la **arquitectura por capas** implica que cada capa solo se comunique con la capa inmediatamente inferior o superior. Esto asegura que no haya una dependencia directa entre capas que no deberían estar relacionadas, como la capa de presentación y la capa de acceso a datos.

- **Capa de presentación a capa de lógica empresarial:** La capa de presentación recibe solicitudes (de usuarios o sistemas externos) y las delega a la capa de lógica empresarial para que se procesen.
- **Capa de lógica empresarial a capa de acceso a datos:** La lógica empresarial interactúa con la capa de acceso a datos para recuperar o almacenar información en bases de datos u otros sistemas de almacenamiento.

Este enfoque garantiza que cada capa se mantenga **independiente** y **desacoplada**, facilitando el mantenimiento, la escalabilidad y la evolución del sistema.

5.2 Independencia de las capas

La independencia de las capas es uno de los principios fundamentales de la **arquitectura por capas**. Ninguna capa debe tener conocimiento de cómo funciona una capa que no está directamente relacionada con ella.

- **Capa de presentación:** No debería conocer detalles sobre cómo se accede o almacena la información. Simplemente delega tareas a la capa de lógica empresarial, que se encarga de ejecutar las operaciones necesarias.
- **Capa de lógica empresarial:** Gestiona las reglas de negocio y utiliza interfaces definidas (como **repositorios**) para interactuar con la capa de acceso a datos, sin necesidad de conocer los detalles de cómo se implementan esas operaciones.

Este **desacoplamiento** es crucial para mantener la modularidad del sistema. Si las capas estuvieran fuertemente acopladas, cualquier cambio en una capa podría tener un impacto directo en las demás.

5.3 Ejemplo de comunicación entre capas en Spring Boot

Veamos un ejemplo en **Spring Boot** de cómo las capas interactúan entre sí. Imaginemos una aplicación de pedidos donde la capa de presentación recibe una solicitud para crear un pedido, la capa de lógica empresarial valida los datos y coordina la creación del pedido, y la capa de acceso a datos almacena el pedido en la base de datos.

Capa de presentación (controlador):

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    private final PedidoService pedidoService;

    @Autowired
    public PedidoController(PedidoService pedidoService) {
        this.pedidoService = pedidoService;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido nuevoPedido =
        pedidoService.crearPedido(request.getCliente(), request.getProductos());
        return ResponseEntity.ok(nuevoPedido);
    }
}
```

- **Explicación:** La capa de presentación (controlador) recibe la solicitud de crear un pedido, valida los datos de entrada, y delega la operación a la capa de lógica empresarial (**PedidoService**).

Capa de lógica empresarial (servicio):

```
@Service
public class PedidoService {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoService(RepositorioDePedidos repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
    {
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }
}
```

- **Explicación:** El servicio de la capa de lógica empresarial recibe la solicitud de crear un pedido, valida las reglas de negocio (como la existencia de productos) y delega la operación de persistencia a la capa de acceso de datos.

Capa de acceso a datos (repositorio):

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

```
}  
}
```

- **Explicación:** El repositorio de la capa de acceso de datos se encarga de persistir el pedido en la base de datos usando **Spring Data JPA**.

5.4 Beneficios del patrón de comunicación entre capas

- **Desacoplamiento:** Cada capa tiene una responsabilidad clara y se comunica únicamente con las capas que le son directamente relevantes. Esto asegura que las capas superiores no necesiten conocer los detalles de cómo funcionan las capas inferiores.
- **Modularidad:** Este patrón permite realizar cambios en una capa sin afectar a las demás. Por ejemplo, podrías cambiar la base de datos (capa de acceso de datos) sin tener que modificar la capa de presentación o la capa de lógica empresarial.
- **Mantenibilidad:** El sistema es más fácil de mantener y evolucionar porque las responsabilidades están claramente definidas, lo que facilita realizar mejoras o correcciones sin generar efectos colaterales.

5.5 Buenas prácticas en la comunicación entre capas

- **Uso de interfaces:** Utilizar **interfaces** para comunicar las capas (especialmente entre la lógica empresarial y la capa de acceso a datos) es una buena práctica, ya que permite desacoplar la implementación y facilita el cambio de tecnología.
- **Validación en la capa adecuada:** Cada capa debe validar los datos según su responsabilidad. Por ejemplo, la validación de datos de entrada ocurre en la capa de presentación, mientras que las reglas de negocio se validan en la capa de lógica empresarial.
- **Manejo de excepciones:** Las excepciones deben manejarse en la capa donde ocurren y propagarse de manera controlada a las capas superiores. Esto asegura que los errores se gestionen correctamente y que las capas superiores reciban información significativa.

5.6 Desafíos en la comunicación entre capas

Aunque el patrón de comunicación entre capas tiene varias ventajas, también puede presentar desafíos, como:

- **Sobrecarga de la comunicación:** En aplicaciones complejas con muchas capas, la comunicación entre capas puede agregar latencia. Esto es especialmente problemático en sistemas distribuidos donde cada capa puede estar separada por una red.
- **Complejidad adicional:** Mantener interfaces claras y bien definidas entre las capas puede agregar complejidad al sistema, especialmente si las capas no están bien estructuradas.

Puertos de entrada y salida en la arquitectura hexagonal

En la **arquitectura hexagonal**, también conocida como el patrón de **puertos y adaptadores**, los **puertos de entrada y salida** son interfaces que definen cómo el dominio interactúa con el mundo exterior. Este enfoque desacopla la lógica de negocio de las infraestructuras y las tecnologías externas, permitiendo que el sistema sea más flexible, mantenible y testeable. Los puertos actúan como **puntos de conexión** entre el núcleo del dominio y los adaptadores que gestionan las interacciones con sistemas externos, como bases de datos, APIs o interfaces de usuario.

6.1 Definición de puertos de entrada y salida

- **Puertos de entrada:** Son interfaces que permiten que actores externos (como usuarios o sistemas) interactúen con la lógica de negocio. Estos puertos representan las acciones o comandos que se pueden ejecutar sobre el dominio.
- **Puertos de salida:** Son interfaces que definen cómo la lógica de negocio interactúa con sistemas externos, como bases de datos, sistemas de mensajería, servicios externos o cualquier infraestructura necesaria. Los puertos de salida desacoplan el dominio de las tecnologías externas.

6.2 Propósito de los puertos en la arquitectura hexagonal

El propósito de los **puertos de entrada y salida** es proporcionar una capa de abstracción que separe la lógica del dominio de las dependencias externas. Esto permite que la lógica de negocio permanezca limpia y no dependa de detalles tecnológicos, lo que facilita el mantenimiento y la evolución del sistema.

- **Abstracción de las tecnologías externas:** Los puertos permiten que la lógica del negocio no dependa de tecnologías específicas (como bases de datos o frameworks de mensajería). Los detalles de implementación están gestionados por los **adaptadores**, mientras que la lógica empresarial solo interactúa con interfaces (puertos).
- **Flexibilidad:** Si se necesita cambiar una tecnología subyacente (por ejemplo, cambiar de una base de datos relacional a una NoSQL), solo se requiere modificar el adaptador correspondiente, sin afectar la lógica de negocio.

6.3 Puertos de entrada: Comunicación con el sistema

Los **puertos de entrada** son las interfaces que definen cómo interactuar con el núcleo del sistema. Representan las operaciones de negocio que pueden ser ejecutadas, como crear un pedido, consultar el estado de una transacción o procesar un pago.

Ejemplo de puerto de entrada en una aplicación de pedidos:

```
public interface GestorDePedidos {  
    Pedido crearPedido(Cliente cliente, List<Producto> productos);  
    Pedido obtenerPedido(Long id);  
}
```


- Este **puerto de entrada** define dos operaciones que los actores externos pueden realizar: crear un pedido y obtener un pedido existente.

6.4 Puertos de salida: Interacción con infraestructuras externas

Los **puertos de salida** definen cómo la lógica empresarial interactúa con infraestructuras técnicas o servicios externos. Por ejemplo, si la lógica de negocio necesita guardar datos en una base de datos o consumir una API de terceros, lo hace a través de un puerto de salida.

Ejemplo de puerto de salida para persistir pedidos:

```
public interface RepositorioDePedidos {  
    Pedido guardar(Pedido pedido);  
    Optional<Pedido> buscarPorId(Long id);  
}
```

- Este puerto de salida define cómo el sistema guarda y recupera pedidos. La lógica empresarial no sabe ni necesita saber cómo se implementa el almacenamiento, lo que permite una gran flexibilidad.

6.5 Relación entre puertos y adaptadores

Los **adaptadores** son las implementaciones concretas de los puertos de entrada y salida. Mientras que los puertos definen las operaciones necesarias, los adaptadores se encargan de implementar estas operaciones utilizando tecnologías específicas, como frameworks de persistencia (Spring Data) o controladores REST.

- **Adaptadores de entrada:** Implementan los puertos de entrada y permiten que los actores externos (usuarios o sistemas) interactúen con la aplicación. Un ejemplo común es un controlador REST que expone un puerto de entrada a través de una API HTTP.
- **Adaptadores de salida:** Implementan los puertos de salida y se encargan de interactuar con sistemas externos, como bases de datos o APIs. Un ejemplo sería un repositorio que utiliza Spring Data JPA para interactuar con una base de datos relacional.

6.6 Ejemplo de puertos y adaptadores en Spring Boot

Veamos cómo los **puertos de entrada y salida** se implementan en un proyecto **Spring Boot** utilizando un ejemplo de gestión de pedidos.

Puerto de entrada (GestorDePedidos):

```
public interface GestorDePedidos {  
    Pedido crearPedido(Cliente cliente, List<Producto> productos);  
    Pedido obtenerPedido(Long id);  
}
```

Puerto de salida (RepositorioDePedidos**):**

```
public interface RepositorioDePedidos {  
    Pedido guardar(Pedido pedido);  
    Optional<Pedido> buscarPorId(Long id);  
}
```

Adaptador de entrada (controlador REST):

```
@RestController  
@RequestMapping("/pedidos")  
public class PedidoController {  
  
    private final GestorDePedidos gestorDePedidos;  
  
    @Autowired  
    public PedidoController(GestorDePedidos gestorDePedidos) {  
        this.gestorDePedidos = gestorDePedidos;  
    }  
  
    @PostMapping  
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest  
request) {  
        Pedido nuevoPedido =  
gestorDePedidos.crearPedido(request.getCliente(),  
request.getProductos());  
        return ResponseEntity.ok(nuevoPedido);  
    }  
  
    @GetMapping("/{id}")  
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {  
        Pedido pedido = gestorDePedidos.obtenerPedido(id);  
        return ResponseEntity.ok(pedido);  
    }  
}
```

Adaptador de salida (repositorio JPA):

```
@Repository
```

```

public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}

```

En este ejemplo, el **puerto de entrada** define las operaciones disponibles (crear y obtener un pedido), mientras que el **adaptador de entrada** (controlador REST) implementa la interfaz para permitir que los actores externos interactúen con la aplicación. El **puerto de salida** define cómo el sistema interactúa con el almacenamiento, y el **adaptador de salida** utiliza Spring Data JPA para persistir los pedidos en la base de datos.

6.7 Ventajas del uso de puertos en la arquitectura hexagonal

- **Desacoplamiento:** Los puertos proporcionan una capa de abstracción que separa la lógica del negocio de las tecnologías externas, lo que permite cambiar o actualizar la infraestructura sin afectar la lógica empresarial.
- **Testabilidad:** Al utilizar puertos, la lógica de negocio se puede probar fácilmente mediante **mocks** o **stubs**, ya que no está directamente acoplada a la infraestructura técnica.
- **Modularidad y flexibilidad:** El sistema se vuelve más modular, lo que facilita la introducción de nuevas tecnologías o adaptadores sin necesidad de modificar el núcleo del sistema.

6.8 Desafíos en la implementación de puertos y adaptadores

Aunque los **puertos y adaptadores** proporcionan un gran nivel de flexibilidad, también pueden introducir complejidad adicional en el sistema:

- **Sobrecarga de implementación:** En proyectos pequeños, la separación entre puertos y adaptadores puede parecer excesiva y generar más código. Sin embargo, en sistemas grandes, la modularidad que ofrecen es fundamental.
- **Sincronización de interfaces:** A medida que el sistema evoluciona, los puertos y adaptadores deben mantenerse alineados para asegurar que las interfaces sigan siendo consistentes.

¿Cómo se relacionan los puertos con los Domain Object y los casos de uso?

En la **arquitectura hexagonal**, los **puertos**, los **Domain Objects**, y los **casos de uso** forman un sistema cohesivo que permite una clara separación de responsabilidades. Los puertos actúan como interfaces que conectan la lógica del negocio con el mundo exterior, los Domain Objects encapsulan los datos y las reglas del dominio, y los casos de uso orquestan la interacción entre los puertos y los Domain Objects para cumplir con los requisitos del negocio. Esta interacción asegura que el núcleo del dominio se mantenga independiente de los detalles técnicos.

7.1 Relación entre puertos y casos de uso

Los **puertos** son los puntos de conexión que permiten que los **casos de uso** interactúen con actores externos (a través de los puertos de entrada) o infraestructuras técnicas (a través de los puertos de salida). Los **casos de uso**, que son responsables de implementar la lógica de negocio, utilizan estos puertos para interactuar con sistemas externos, como bases de datos o APIs.

Puertos de entrada y casos de uso: Los puertos de entrada son interfaces que permiten que actores externos (usuarios o sistemas) inicien un caso de uso. Por ejemplo, un puerto de entrada podría exponer un método para crear un pedido, que sería implementado por un **caso de uso** encargado de gestionar toda la lógica de negocio para esta operación.

Ejemplo:

```
public interface GestorDePedidos {  
    Pedido crearPedido(Cliente cliente, List<Producto> productos);  
}
```

- El **caso de uso** `GestorDePedidosImpl` implementa este puerto de entrada y orquesta la lógica de negocio necesaria para crear un pedido.

Puertos de salida y casos de uso: Los puertos de salida permiten que los casos de uso interactúen con infraestructuras externas, como persistencia de datos o servicios de mensajería. Por ejemplo, el caso de uso `GestorDePedidosImpl` puede utilizar un **puerto de salida** para guardar un pedido en la base de datos.

Ejemplo:

```
public interface RepositorioDePedidos {  
    Pedido guardar(Pedido pedido);  
}
```

- El caso de uso utilizará este puerto para interactuar con la capa de persistencia sin preocuparse por los detalles de implementación de la base de datos.

7.2 Relación entre puertos y Domain Objects

Los **Domain Objects** son los objetos principales que representan las entidades y reglas del negocio en el sistema. Aunque los puertos permiten la interacción con sistemas externos, los **Domain Objects** encapsulan las reglas y comportamientos que definen cómo el sistema funciona desde la perspectiva del negocio.

Puertos de entrada y Domain Objects: A través de los puertos de entrada, los casos de uso invocan operaciones sobre los **Domain Objects**. Por ejemplo, en el caso de uso de `crearPedido()`, un puerto de entrada podría recibir una solicitud para crear un nuevo pedido. El caso de uso, a través de este puerto, interactúa con los **Domain Objects** para crear una nueva instancia de `Pedido` y verificar que se cumplan las reglas del negocio (como asegurarse de que los productos tengan stock disponible).

Ejemplo de interacción con un Domain Object:

```
public class PedidoService {  
  
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)  
    {  
        Pedido pedido = new Pedido(cliente, productos);  
        pedido.validarStock(); // Valida reglas de negocio  
        return pedido;  
    }  
}
```

Puertos de salida y Domain Objects: Los puertos de salida permiten que los **Domain Objects** se almacenen o interactúen con sistemas externos sin que la lógica empresarial dependa de esos sistemas. Los Domain Objects no deben conocer los detalles de cómo se guardan en una base de datos o cómo se integran con otras infraestructuras. Esto es gestionado por los puertos de salida y los adaptadores correspondientes.

Ejemplo:

```
public class PedidoService {  
    private final RepositorioDePedidos repositorioDePedidos;  
  
    public Pedido guardarPedido(Pedido pedido) {  
        return repositorioDePedidos.guardar(pedido); // Persistencia a  
        través de un puerto de salida  
    }  
}
```

7.3 Roles de los casos de uso en la interacción con los puertos

Los **casos de uso** orquestan la interacción entre los puertos y los **Domain Objects**. Su función principal es coordinar las acciones necesarias para completar una operación de negocio, utilizando los puertos para comunicarse con el exterior y los Domain Objects para ejecutar la lógica de negocio interna.

- **Coordinación de la lógica de negocio:** Los casos de uso actúan como controladores de la lógica de negocio, asegurándose de que las operaciones que involucren múltiples Domain Objects se realicen de manera coherente. También gestionan las interacciones con infraestructuras externas a través de los puertos de salida.
- **Independencia de los detalles técnicos:** Aunque los casos de uso utilizan puertos de salida para interactuar con sistemas externos, no necesitan conocer los detalles de cómo funcionan esos sistemas. Esto asegura que los casos de uso se mantengan independientes de las tecnologías específicas y se centren únicamente en las reglas del negocio.

Ejemplo de caso de uso en acción:

```
@Service
public class GestorDePedidosImpl implements GestorDePedidos {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public GestorDePedidosImpl(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        pedido.validarStock(); // Validación de reglas de negocio
        return repositorioDePedidos.guardar(pedido); // Interacción con
la capa de persistencia a través de un puerto de salida
    }
}
```

En este ejemplo, el **caso de uso** `GestorDePedidosImpl` utiliza un **puerto de salida** para almacenar un pedido en la base de datos, manteniendo la lógica empresarial separada de los detalles de persistencia.

7.4 Beneficios de la separación entre puertos, Domain Objects y casos de uso

- **Desacoplamiento de la infraestructura:** Los puertos permiten que la lógica empresarial, representada por los Domain Objects y los casos de uso, no dependa de los detalles de implementación de la infraestructura técnica. Esto facilita el mantenimiento y la evolución del sistema.
- **Modularidad y reutilización:** Al tener los casos de uso separados de los detalles técnicos, es más fácil reutilizar la lógica empresarial en diferentes interfaces o adaptar la lógica a nuevas tecnologías sin cambiar el núcleo del sistema.
- **Flexibilidad y testabilidad:** La separación entre puertos, Domain Objects y casos de uso permite que cada parte del sistema se pruebe de manera independiente, utilizando **mocks** o **stubs** para simular la infraestructura o las interacciones externas.

7.5 Desafíos en la implementación

- **Gestión de la complejidad:** Aunque la separación entre puertos, Domain Objects y casos de uso mejora la modularidad, también puede introducir complejidad adicional en términos de gestión y sincronización de interfaces, especialmente en sistemas grandes.
- **Sobrecarga inicial:** En sistemas pequeños, implementar puertos y casos de uso puede parecer innecesario y generar sobrecarga, pero es una inversión a largo plazo que facilita la escalabilidad y mantenibilidad.

Creando un puerto de entrada y salida en un proyecto Spring bajo un modelo de arquitectura hexagonal

En un proyecto basado en la **arquitectura hexagonal**, la creación de **puertos de entrada y salida** permite desacoplar la lógica de negocio de las tecnologías y sistemas externos. Los **puertos de entrada** permiten que los actores externos (usuarios, sistemas, APIs) interactúen con el sistema, mientras que los **puertos de salida** permiten que la lógica empresarial interactúe con bases de datos, servicios externos o sistemas de mensajería. En un proyecto **Spring Boot**, estos puertos se definen como interfaces y se implementan a través de **adaptadores** que manejan la interacción con la infraestructura.

8.1 Creación de un puerto de entrada

Un **puerto de entrada** es una interfaz que define las acciones que se pueden realizar sobre la lógica de negocio. Representa los casos de uso que están disponibles para los actores externos. Por ejemplo, en una aplicación de pedidos, un puerto de entrada puede definir las operaciones que los usuarios pueden realizar, como crear un pedido o consultar el estado de uno existente.

Paso 1: Definir el puerto de entrada (interfaz): En un proyecto Spring, el puerto de entrada se define como una **interfaz** que expone los casos de uso. Esta interfaz será implementada por un servicio que orqueste la lógica de negocio.

Ejemplo de puerto de entrada:

```
public interface GestorDePedidos {
```

```
Pedido crearPedido(Cliente cliente, List<Producto> productos);  
Pedido obtenerPedido(Long id);  
}
```

Paso 2: Implementar el puerto de entrada (caso de uso): La implementación de este puerto se realiza en un **servicio** que contiene la lógica empresarial para los casos de uso correspondientes. Este servicio orquestará las interacciones con los **Domain Objects** y utilizará los **puertos de salida** para acceder a la persistencia o a servicios externos.

Ejemplo de implementación del puerto de entrada:

```
@Service  
public class GestorDePedidosImpl implements GestorDePedidos {  
  
    private final RepositorioDePedidos repositorioDePedidos;  
  
    @Autowired  
    public GestorDePedidosImpl(RepositorioDePedidos  
repositorioDePedidos) {  
        this.repositorioDePedidos = repositorioDePedidos;  
    }  
  
    @Override  
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)  
    {  
        Pedido pedido = new Pedido(cliente, productos);  
        return repositorioDePedidos.guardar(pedido); // Utiliza un  
puerto de salida para persistir el pedido  
    }  
  
    @Override  
    public Pedido obtenerPedido(Long id) {  
        return repositorioDePedidos.buscarPorId(id)  
            .orElseThrow(() -> new PedidoNoEncontradoException(id));  
    }  
}
```

En este ejemplo, el servicio `GestorDePedidosImpl` implementa el puerto de entrada y delega la persistencia de los pedidos al **puerto de salida** `RepositorioDePedidos`.

8.2 Creación de un puerto de salida

Un **puerto de salida** es una interfaz que define cómo interactuar con sistemas externos desde la lógica empresarial. Estos puertos permiten que la lógica de negocio acceda a bases de datos, APIs de terceros, servicios de mensajería, entre otros, sin estar acoplada a tecnologías específicas.

Paso 1: Definir el puerto de salida (interfaz): El puerto de salida es una **interfaz** que expone las operaciones que la lógica empresarial necesita realizar sobre sistemas externos. Por ejemplo, el puerto de salida para la persistencia de un pedido se podría definir así:

Ejemplo de puerto de salida:

```
public interface RepositorioDePedidos {  
    Pedido guardar(Pedido pedido);  
    Optional<Pedido> buscarPorId(Long id);  
}
```

Paso 2: Implementar el puerto de salida (adaptador): Los adaptadores son las implementaciones concretas de los puertos de salida. En el caso de un puerto de salida para la persistencia, el adaptador utilizará Spring Data JPA o cualquier otra tecnología de persistencia para implementar las operaciones definidas en el puerto.

Ejemplo de implementación del puerto de salida con Spring Data JPA:

```
@Repository  
public class RepositorioDePedidosImpl implements RepositorioDePedidos {  
  
    @Autowired  
    private JpaPedidoRepository jpaPedidoRepository;  
  
    @Override  
    public Pedido guardar(Pedido pedido) {  
        return jpaPedidoRepository.save(pedido);  
    }  
  
    @Override  
    public Optional<Pedido> buscarPorId(Long id) {  
        return jpaPedidoRepository.findById(id);  
    }  
}
```

- En este ejemplo, `RepositorioDePedidosImpl` es el adaptador que implementa el puerto de salida, utilizando **Spring Data JPA** para realizar las operaciones de persistencia sobre la entidad `Pedido`.

8.3 Integración con los adaptadores de entrada

En un proyecto Spring Boot, los **adaptadores de entrada** generalmente son **controladores REST** que manejan las solicitudes de los usuarios. Estos controladores implementan los puertos de entrada para recibir las solicitudes, validar los datos y delegar la ejecución de la lógica empresarial al caso de uso correspondiente.

Paso 1: Implementar el adaptador de entrada (controlador REST): El adaptador de entrada implementa el puerto de entrada mediante un controlador REST que expone las operaciones a través de una API HTTP.

Ejemplo de controlador REST que actúa como adaptador de entrada:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido nuevoPedido =
            gestorDePedidos.crearPedido(request.getCliente(),
            request.getProductos());
        return ResponseEntity.ok(nuevoPedido);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
        Pedido pedido = gestorDePedidos.obtenerPedido(id);
        return ResponseEntity.ok(pedido);
    }
}
```

- En este ejemplo, `PedidoController` es el adaptador de entrada que recibe las solicitudes HTTP y utiliza el **puerto de entrada** `GestorDePedidos` para procesarlas.

8.4 Ventajas de los puertos y adaptadores en la arquitectura hexagonal

- **Desacoplamiento:** Los puertos y adaptadores permiten mantener una separación clara entre la lógica empresarial y los detalles técnicos, como la persistencia o la comunicación con sistemas externos.

- **Flexibilidad:** Si se necesita cambiar la tecnología de persistencia o de comunicación externa, solo es necesario modificar el adaptador correspondiente, sin afectar la lógica de negocio.
- **Testabilidad:** Al utilizar puertos, es posible **mockear** fácilmente los adaptadores en las pruebas unitarias, permitiendo que la lógica de negocio se pruebe de manera aislada sin depender de bases de datos u otros sistemas externos.

8.5 Desafíos en la implementación de puertos y adaptadores

- **Sobrecarga en proyectos pequeños:** En proyectos pequeños, la separación entre puertos y adaptadores puede parecer innecesaria o agregar complejidad. Sin embargo, en sistemas grandes o a largo plazo, esta separación es crucial para mantener la flexibilidad y escalabilidad.
- **Mantenimiento de interfaces:** A medida que el sistema crece, es importante mantener actualizados los puertos y adaptadores, asegurando que las interfaces estén bien definidas y sincronizadas.

Tipos de adaptadores en la arquitectura hexagonal

En la **arquitectura hexagonal**, los **adaptadores** son las implementaciones concretas de los **puertos de entrada y salida** que permiten que el sistema interactúe con el mundo exterior, ya sea con actores externos (usuarios, otros sistemas) o con infraestructuras técnicas (bases de datos, APIs, sistemas de mensajería). Los adaptadores actúan como una capa de traducción entre el núcleo del dominio y los detalles técnicos, manteniendo el **desacoplamiento** entre la lógica empresarial y las tecnologías específicas.

Existen principalmente dos tipos de adaptadores en la arquitectura hexagonal:

- **Adaptadores de entrada:** Reciben y procesan las solicitudes externas, ya sea de usuarios o sistemas externos, y las transforman en comandos o consultas que puedan ser manejados por los **casos de uso** o **servicios** dentro del núcleo del dominio.
- **Adaptadores de salida:** Interactúan con infraestructuras técnicas externas (como bases de datos, servicios externos o sistemas de mensajería) para realizar operaciones como la persistencia de datos, la comunicación con APIs, o la integración con servicios de mensajería.

9.1 Adaptadores de entrada

Los **adaptadores de entrada** permiten que los usuarios o sistemas externos interactúen con el sistema. Estos adaptadores son responsables de traducir las solicitudes de los usuarios (por ejemplo, una solicitud HTTP en una API REST) en comandos que la lógica de negocio pueda procesar.

Ejemplo común de adaptadores de entrada:

- **Controladores REST:** En aplicaciones basadas en Spring Boot, los controladores REST son adaptadores de entrada que exponen las operaciones de negocio a través de una API HTTP. Estos controladores reciben solicitudes (generalmente en

formato JSON o XML), validan los datos y delegan las operaciones a los **casos de uso** definidos en el núcleo del sistema.

Ejemplo de un controlador REST en Spring Boot:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido nuevoPedido =
            gestorDePedidos.crearPedido(request.getCliente(),
            request.getProductos());
        return ResponseEntity.ok(nuevoPedido);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
        Pedido pedido = gestorDePedidos.obtenerPedido(id);
        return ResponseEntity.ok(pedido);
    }
}
```

En este ejemplo, el **adaptador de entrada** `PedidoController` recibe solicitudes HTTP y utiliza el **puerto de entrada** `GestorDePedidos` para procesar la lógica de negocio.

Otros tipos de adaptadores de entrada:

- **Interfaces de usuario gráficas (GUIs):** Las interfaces de usuario en aplicaciones web o de escritorio actúan como adaptadores de entrada al permitir que los usuarios interactúen con la lógica de negocio a través de una interfaz gráfica.
- **APIs SOAP o gRPC:** Estas interfaces permiten que sistemas externos interactúen con la aplicación utilizando diferentes protocolos de comunicación, como SOAP o gRPC.

9.2 Adaptadores de salida

Los **adaptadores de salida** son los encargados de manejar la interacción con infraestructuras técnicas externas. Estos adaptadores se encargan de traducir las operaciones del sistema en operaciones que puedan ser ejecutadas por una base de datos, un servicio externo o un sistema de mensajería. La lógica de negocio solo interactúa con **puertos de salida**, mientras que los adaptadores gestionan los detalles de la implementación.

Ejemplo común de adaptadores de salida:

- **Repositorios para la persistencia:** En Spring Boot, un adaptador de salida común es un **repositorio** que utiliza tecnologías como **Spring Data JPA** o **MongoDB** para interactuar con una base de datos. Estos repositorios implementan los puertos de salida definidos por la lógica de negocio.

Ejemplo de un repositorio como adaptador de salida:

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

En este caso, el **adaptador de salida** `RepositorioDePedidosImpl` implementa el puerto de salida `RepositorioDePedidos` utilizando Spring Data JPA para persistir los datos en una base de datos relacional.

Otros tipos de adaptadores de salida:

- **Integración con servicios externos:** Los adaptadores de salida pueden interactuar con **APIs de terceros** para obtener información o realizar operaciones externas, como la integración con una pasarela de pagos o una API de envío.
- **Sistemas de mensajería:** Los adaptadores de salida pueden interactuar con **colas de mensajes** o **brokers de mensajería** (como RabbitMQ o Kafka) para enviar y recibir mensajes de otros sistemas, desacoplando la lógica empresarial de los detalles de la comunicación asíncrona.

9.3 Relación entre puertos y adaptadores

La relación entre los **puertos** y los **adaptadores** es esencial para mantener el desacoplamiento de la lógica de negocio de las tecnologías externas:

- **Puertos de entrada:** Definen las interfaces que los adaptadores de entrada implementan. Estos puertos representan las operaciones del sistema que los usuarios o sistemas externos pueden realizar.
- **Puertos de salida:** Definen las interfaces que los adaptadores de salida implementan. Estos puertos proporcionan las operaciones que la lógica empresarial necesita para interactuar con sistemas externos, como la persistencia o la comunicación con APIs.

Los **adaptadores** implementan los **puertos**, proporcionando una capa de abstracción que permite que la lógica empresarial permanezca independiente de los detalles técnicos. Si se necesita cambiar la tecnología de una base de datos o la forma en que se comunica con una API externa, solo se modifica el adaptador, manteniendo el puerto (la interfaz) inalterada.

9.4 Beneficios de los adaptadores en la arquitectura hexagonal

- **Desacoplamiento total:** Los adaptadores permiten que la lógica de negocio se mantenga completamente desacoplada de los detalles técnicos, ya que interactúa solo con interfaces (puertos) que son implementadas por los adaptadores.
- **Flexibilidad:** Si se cambia una tecnología subyacente (por ejemplo, cambiar de una base de datos relacional a una NoSQL), solo es necesario modificar el adaptador, manteniendo la lógica empresarial intacta.
- **Testabilidad:** Al estar los adaptadores desacoplados de la lógica de negocio, es más fácil realizar pruebas unitarias utilizando mocks o stubs para simular los adaptadores y los sistemas externos.

9.5 Desafíos de los adaptadores en la arquitectura hexagonal

- **Sobrecarga en proyectos pequeños:** En proyectos pequeños, puede parecer innecesario crear múltiples adaptadores para cada tecnología. Sin embargo, en proyectos grandes y a largo plazo, esta separación permite una gran flexibilidad y escalabilidad.
- **Mantenimiento de interfaces:** A medida que el sistema crece, es importante mantener las interfaces (puertos) actualizadas y consistentes con los adaptadores que las implementan.

Introducción al Web Adaptor y Persistent Adaptor

En la **arquitectura hexagonal**, los adaptadores juegan un papel fundamental en conectar el núcleo de la aplicación con el mundo exterior. Dos de los adaptadores más comunes son el **Web Adaptor** y el **Persistent Adaptor**, que permiten que el sistema interactúe con interfaces web (como APIs REST) y con bases de datos, respectivamente. Estos

adaptadores implementan los **puertos de entrada y salida**, permitiendo que la lógica empresarial se mantenga desacoplada de los detalles técnicos de la infraestructura.

10.1 ¿Qué es un Web Adaptor?

El **Web Adaptor** es un tipo de **adaptador de entrada** que maneja la interacción entre la aplicación y las solicitudes entrantes a través de **interfaces web**, como **APIs REST**, **SOAP**, o incluso interfaces gráficas web. El Web Adaptor actúa como intermediario entre las solicitudes HTTP realizadas por los usuarios o sistemas externos y la lógica de negocio de la aplicación.

- **Responsabilidad:** Convertir las solicitudes HTTP (generalmente en formato JSON o XML) en comandos o consultas que la lógica empresarial pueda entender. Luego, el Web Adaptor delega la ejecución de estos comandos a los **casos de uso** o servicios correspondientes.
- **Tecnología común:** En **Spring Boot**, el Web Adaptor suele implementarse mediante **controladores REST** que manejan las rutas HTTP y delegan las operaciones a la lógica de negocio a través de los **puertos de entrada**.

Ejemplo de Web Adaptor (controlador REST):

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido nuevoPedido =
            gestorDePedidos.crearPedido(request.getCliente(),
            request.getProductos());
        return ResponseEntity.ok(nuevoPedido);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
        Pedido pedido = gestorDePedidos.obtenerPedido(id);
        return ResponseEntity.ok(pedido);
    }
}
```

En este ejemplo, el **PedidoController** es un **Web Adaptor** que recibe las solicitudes HTTP entrantes, las valida y luego delega la lógica de negocio al servicio correspondiente (**GestorDePedidos**). Este adaptador se encarga de gestionar la **comunicación HTTP** y convertirla en acciones que la aplicación pueda procesar.

10.2 ¿Qué es un Persistent Adaptor?

El **Persistent Adaptor** es un tipo de **adaptador de salida** que maneja la interacción entre la lógica empresarial y los **sistemas de almacenamiento**, como **bases de datos relacionales** o **NoSQL**. Su responsabilidad es permitir que los **Domain Objects** sean almacenados o recuperados de los sistemas de persistencia sin que la lógica empresarial esté acoplada a una tecnología específica.

- **Responsabilidad:** Implementar las operaciones necesarias para almacenar, actualizar, recuperar y eliminar datos en la base de datos, ocultando los detalles de implementación de la lógica empresarial. El Persistent Adaptor interactúa con la base de datos a través de herramientas como **JPA**, **JDBC**, **MongoDB**, u otras tecnologías de persistencia.
- **Tecnología común:** En Spring Boot, los **repositorios** de Spring Data son un ejemplo común de **Persistent Adaptors**, ya que permiten interactuar con bases de datos mediante una interfaz bien definida.

Ejemplo de Persistent Adaptor (repositorio JPA):

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

En este caso, el **RepositorioDePedidosImpl** es un **Persistent Adaptor** que utiliza **Spring Data JPA** para interactuar con una base de datos relacional. Implementa el puerto de salida

RepositorioDePedidos, lo que permite que la lógica de negocio interactúe con la base de datos sin necesidad de conocer los detalles de cómo se realiza la persistencia.

10.3 Relación entre Web Adaptor y Persistent Adaptor

El **Web Adaptor** y el **Persistent Adaptor** suelen trabajar juntos en muchas aplicaciones:

1. El **Web Adaptor** recibe una solicitud HTTP, como una solicitud para crear un pedido.
2. Esta solicitud es validada y luego delegada a un **caso de uso** o servicio en la lógica empresarial.
3. El servicio interactúa con los **Domain Objects** para aplicar las reglas de negocio, y cuando es necesario almacenar o recuperar datos, utiliza un **Persistent Adaptor**.
4. El **Persistent Adaptor** se encarga de guardar o recuperar los datos en una base de datos u otro sistema de almacenamiento.
5. Finalmente, el **Web Adaptor** envía la respuesta al cliente con el resultado de la operación.

10.4 Ejemplo de interacción entre Web Adaptor y Persistent Adaptor

Supongamos que tenemos una API REST para gestionar pedidos. Cuando un usuario hace una solicitud para crear un pedido, el flujo sería el siguiente:

1. **Solicitud HTTP (Web Adaptor):** El usuario envía una solicitud POST a la API **/pedidos** con los detalles del pedido.
2. **Validación y delegación (Web Adaptor):** El Web Adaptor valida la solicitud y llama al servicio correspondiente (**GestorDePedidos**) para que maneje la lógica de negocio.
3. **Persistencia (Persistent Adaptor):** El servicio **GestorDePedidos** crea un nuevo **Domain Object Pedido** y luego utiliza el **Persistent Adaptor** (el repositorio de pedidos) para guardar el pedido en la base de datos.
4. **Respuesta HTTP (Web Adaptor):** El Web Adaptor recibe el resultado de la operación (el nuevo pedido creado) y lo devuelve al usuario en una respuesta HTTP.

Flujo en código:

Web Adaptor (PedidoController):

```
@PostMapping
public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
    Pedido nuevoPedido =
gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
    return ResponseEntity.ok(nuevoPedido);
}
```

Lógica empresarial (GestorDePedidosImpl):

```
@Service
public class GestorDePedidosImpl implements GestorDePedidos {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public GestorDePedidosImpl(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }
}
```

Persistent Adaptor (RepositorioDePedidosImpl):

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }
}
```

10.5 Beneficios del Web Adaptor y Persistent Adaptor

- **Desacoplamiento:** Estos adaptadores permiten que la lógica empresarial se mantenga completamente independiente de las tecnologías web y de persistencia. La lógica de negocio solo interactúa con interfaces (puertos), y los detalles técnicos son gestionados por los adaptadores.

- **Flexibilidad tecnológica:** Cambiar la tecnología de persistencia (por ejemplo, de una base de datos relacional a NoSQL) o la interfaz web (por ejemplo, de REST a gRPC) es sencillo, ya que solo es necesario modificar los adaptadores correspondientes sin afectar la lógica de negocio.
- **Testabilidad:** Al estar desacoplados, tanto el Web Adaptor como el Persistent Adaptor pueden ser simulados o testeados de forma aislada, lo que facilita la creación de pruebas unitarias y de integración.

10.6 Desafíos en el uso de Web Adaptor y Persistent Adaptor

- **Mantenimiento de interfaces:** A medida que la aplicación crece, es necesario mantener actualizados los puertos y adaptadores para asegurar que las interfaces entre la lógica de negocio y los adaptadores estén bien definidas y sincronizadas.
- **Sobrecarga en proyectos pequeños:** En proyectos pequeños, la introducción de adaptadores puede parecer innecesaria, pero en sistemas grandes y escalables, esta separación resulta esencial.

Web Adaptor y Persistent Adaptor en un proyecto Spring bajo un modelo de arquitectura hexagonal

En un proyecto **Spring Boot** que sigue el patrón de **arquitectura hexagonal**, el **Web Adaptor** y el **Persistent Adaptor** desempeñan roles esenciales para conectar la lógica de negocio con el mundo exterior, manteniendo el sistema **desacoplado** de los detalles técnicos. Estos adaptadores implementan los **puertos de entrada y salida**, permitiendo que la lógica empresarial interactúe con actores externos (a través del Web Adaptor) y con infraestructuras de almacenamiento (a través del Persistent Adaptor).

11.1 Web Adaptor en un proyecto Spring Boot

El **Web Adaptor** en un proyecto Spring Boot se implementa típicamente mediante **controladores REST** que exponen los **puertos de entrada** y permiten que los usuarios o sistemas externos interactúen con la lógica de negocio. Estos controladores actúan como puntos de entrada para las solicitudes HTTP y manejan la comunicación entre los clientes y el sistema.

- **Responsabilidad del Web Adaptor:** Recibe las solicitudes HTTP, valida los datos de entrada y delega la lógica empresarial a los casos de uso o servicios correspondientes.
- **Tecnología utilizada:** En **Spring Boot**, el Web Adaptor se implementa mediante la anotación `@RestController` y el uso de **Spring MVC** o **Spring WebFlux** para gestionar las rutas HTTP.

Ejemplo de Web Adaptor en Spring Boot:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {
```

```

private final GestorDePedidos gestorDePedidos;

@Autowired
public PedidoController(GestorDePedidos gestorDePedidos) {
    this.gestorDePedidos = gestorDePedidos;
}

@PostMapping
public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
    Pedido nuevoPedido =
gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
    return ResponseEntity.ok(nuevoPedido);
}

@GetMapping("/{id}")
public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
    Pedido pedido = gestorDePedidos.obtenerPedido(id);
    return ResponseEntity.ok(pedido);
}
}

```

En este ejemplo, `PedidoController` es el **Web Adaptor** que implementa el puerto de entrada `GestorDePedidos`. Recibe solicitudes para crear y obtener pedidos, valida los datos de entrada, y luego delega las operaciones a la capa de lógica empresarial.

11.2 Persistent Adaptor en un proyecto Spring Boot

El **Persistent Adaptor** se utiliza para gestionar la interacción con las **bases de datos** o cualquier otro sistema de almacenamiento. En un proyecto Spring Boot, los **repositorios** de Spring Data son una solución común para implementar el **puerto de salida**, permitiendo que la lógica empresarial realice operaciones CRUD sin preocuparse por los detalles de la persistencia.

- **Responsabilidad del Persistent Adaptor:** Implementar las interfaces definidas por los puertos de salida y proporcionar una implementación específica para interactuar con una base de datos u otro sistema de almacenamiento.
- **Tecnología utilizada:** **Spring Data JPA**, **Spring Data MongoDB**, **JDBC**, u otras tecnologías de persistencia se utilizan para implementar estos adaptadores.

Ejemplo de Persistent Adaptor en Spring Boot:

```

@Repository

```

```

public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }

    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}

```

En este ejemplo, el **Persistent Adaptor** `RepositorioDePedidosImpl` implementa el puerto de salida `RepositorioDePedidos` utilizando **Spring Data JPA** para interactuar con una base de datos relacional.

11.3 Flujo completo: Interacción entre Web Adaptor y Persistent Adaptor

Veamos cómo interactúan el **Web Adaptor** y el **Persistent Adaptor** en un proyecto **Spring Boot** siguiendo el modelo de arquitectura hexagonal.

1. **Solicitud del cliente (Web Adaptor):**
 - Un usuario o sistema externo realiza una solicitud HTTP (por ejemplo, una solicitud POST para crear un nuevo pedido).
2. **Procesamiento de la solicitud (Web Adaptor):**
 - El **controlador REST** (Web Adaptor) recibe la solicitud, valida los datos de entrada y delega la ejecución de la lógica empresarial a un **caso de uso** o servicio correspondiente (`GestorDePedidos`).
3. **Ejecución de la lógica empresarial (caso de uso):**
 - El **caso de uso** interactúa con los **Domain Objects** (por ejemplo, `Pedido`) y aplica las reglas de negocio necesarias.
4. **Persistencia de datos (Persistent Adaptor):**
 - Si la lógica empresarial necesita guardar o recuperar información de una base de datos, utiliza un **puerto de salida** implementado por un **Persistent Adaptor**. Este adaptador interactúa con la base de datos utilizando tecnologías como **Spring Data JPA** o **JDBC**.
5. **Respuesta al cliente (Web Adaptor):**
 - Después de completar la operación, el **Web Adaptor** envía una respuesta HTTP al cliente con el resultado de la operación (por ejemplo, el pedido creado o un mensaje de error si algo salió mal).

Flujo en código:

Web Adaptor (PedidoController):

```
@PostMapping
public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest
request) {
    Pedido nuevoPedido =
gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
    return ResponseEntity.ok(nuevoPedido);
}
```

Lógica empresarial (GestorDePedidosImpl):

```
@Service
public class GestorDePedidosImpl implements GestorDePedidos {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public GestorDePedidosImpl(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }
}
```

Persistent Adaptor (RepositorioDePedidosImpl):

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
```

```
public Pedido guardar(Pedido pedido) {  
    return jpaPedidoRepository.save(pedido);  
}  
}
```

11.4 Beneficios del uso de Web Adaptor y Persistent Adaptor en un proyecto Spring Boot

- **Desacoplamiento:** El uso de **Web Adaptor** y **Persistent Adaptor** asegura que la lógica de negocio esté completamente desacoplada de la tecnología de presentación (HTTP, REST) y de la tecnología de persistencia (bases de datos). Esto permite que el sistema sea más flexible y fácil de mantener.
- **Testabilidad:** La lógica empresarial puede probarse de forma aislada, utilizando mocks para simular los adaptadores. Esto permite realizar pruebas unitarias más eficientes y fáciles de implementar.
- **Reutilización de componentes:** Los adaptadores permiten que los puertos sean reutilizados con diferentes tecnologías. Por ejemplo, un sistema puede tener tanto un Web Adaptor basado en REST como otro basado en WebSockets, utilizando los mismos puertos de entrada.

11.5 Desafíos en la implementación

- **Complejidad adicional en sistemas pequeños:** Para aplicaciones pequeñas, la introducción de adaptadores puede parecer una sobrecarga innecesaria. Sin embargo, a medida que la aplicación crece en complejidad, esta separación entre los puertos y los adaptadores facilita el mantenimiento y la escalabilidad del sistema.
- **Mantenimiento de interfaces:** A medida que el sistema evoluciona, es necesario mantener las interfaces de los puertos actualizadas y sincronizadas con las implementaciones de los adaptadores, lo que puede agregar una carga adicional de mantenimiento.

Segregación de responsabilidades entre comandos y consultas

La **segregación de responsabilidades entre comandos y consultas**, también conocida como **CQRS (Command Query Responsibility Segregation)**, es un patrón arquitectónico que separa las operaciones de lectura y escritura en diferentes modelos. En lugar de utilizar un único modelo de datos para realizar tanto las operaciones de **lectura** (consultas) como las de **escritura** (comandos), CQRS propone dividir estas responsabilidades, permitiendo que las consultas y los comandos se manejen de manera independiente.

Este enfoque es útil cuando las operaciones de lectura y escritura tienen diferentes requisitos en términos de rendimiento, complejidad o escalabilidad.

12.1 Definición de CQRS

CQRS separa las responsabilidades entre:

- **Comandos (Commands):** Son las operaciones que **modifican el estado** del sistema. Los comandos deben ser operaciones que cambien algo en el estado de la aplicación, como crear, actualizar o eliminar un recurso.
- **Consultas (Queries):** Son las operaciones que **recuperan información** sin modificar el estado del sistema. Las consultas deben ser operaciones que simplemente leen datos y devuelven información sin ningún efecto secundario.

Al aplicar **CQRS**, las operaciones de comandos y consultas se gestionan de manera independiente, lo que permite optimizar cada una según sus necesidades específicas.

12.2 Ventajas de la segregación entre comandos y consultas

1. **Optimización de rendimiento:** Al separar las lecturas y escrituras, es posible optimizar cada modelo de forma independiente. Por ejemplo, el modelo de consultas puede ser diseñado para ser más rápido y eficiente, mientras que el modelo de comandos puede enfocarse en garantizar la consistencia y robustez de los datos.
2. **Escalabilidad:** En sistemas con muchas más operaciones de lectura que de escritura (o viceversa), CQRS permite escalar las lecturas y las escrituras de manera independiente, optimizando el uso de los recursos.
3. **Simplicidad en la lógica de negocio:** Separar las responsabilidades entre comandos y consultas puede simplificar la lógica de negocio, ya que cada operación tiene un propósito claro y específico. Las operaciones de lectura no tienen que lidiar con la complejidad de las actualizaciones, y viceversa.
4. **Flexibilidad para el uso de diferentes tecnologías:** CQRS permite que el sistema use diferentes tecnologías o bases de datos para manejar las lecturas y escrituras. Por ejemplo, las lecturas pueden servirse desde una base de datos NoSQL optimizada para consultas rápidas, mientras que las escrituras se gestionan en una base de datos relacional que garantiza la integridad de los datos.

12.3 Implementación de CQRS en un proyecto Spring Boot

En un proyecto **Spring Boot**, CQRS se implementa separando los **controladores**, **servicios** y **repositorios** para las operaciones de lectura y escritura. Cada operación tiene su propio conjunto de clases y modelos para asegurar que no haya solapamiento entre las responsabilidades.

12.3.1 Comandos (Commands)

Los **comandos** son operaciones que cambian el estado del sistema, como crear, actualizar o eliminar una entidad. En Spring Boot, los comandos suelen manejarse mediante servicios que implementan la lógica de negocio para actualizar el estado de los **Domain Objects**.

Ejemplo de comando (crear un pedido):

Servicio de comandos:

```
@Service
public class PedidoCommandService {
```



```

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoCommandService(RepositorioDePedidos
repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
{
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }
}

```

Controlador para manejar comandos:

```

@RestController
@RequestMapping("/pedidos")
public class PedidoCommandController {

    private final PedidoCommandService pedidoCommandService;

    @Autowired
    public PedidoCommandController(PedidoCommandService
pedidoCommandService) {
        this.pedidoCommandService = pedidoCommandService;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest
request) {
        Pedido nuevoPedido =
pedidoCommandService.crearPedido(request.getClient(),
request.getProductos());
        return ResponseEntity.ok(nuevoPedido);
    }
}

```

En este caso, el **PedidoCommandService** maneja la lógica de negocio relacionada con la creación de un pedido, mientras que el **PedidoCommandController** se encarga de recibir las solicitudes HTTP que modifican el estado del sistema (en este caso, la creación de un pedido).

12.3.2 Consultas (Queries)

Las **consultas** son operaciones que solo leen datos del sistema sin modificar su estado. En Spring Boot, las consultas pueden manejarse mediante un servicio especializado en recuperar datos y un controlador que exponga esas consultas.

Ejemplo de consulta (obtener un pedido):

Servicio de consultas:

```
@Service
public class PedidoQueryService {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoQueryService(RepositorioDePedidos repositorioDePedidos)
    {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    public Pedido obtenerPedido(Long id) {
        return repositorioDePedidos.buscarPorId(id)
            .orElseThrow(() -> new PedidoNoEncontradoException(id));
    }
}
```

Controlador para manejar consultas:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoQueryController {

    private final PedidoQueryService pedidoQueryService;

    @Autowired
    public PedidoQueryController(PedidoQueryService pedidoQueryService)
    {
        this.pedidoQueryService = pedidoQueryService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<Pedido> obtenerPedido(@PathVariable Long id) {
        Pedido pedido = pedidoQueryService.obtenerPedido(id);
    }
}
```

```
        return ResponseEntity.ok(pedido);
    }
}
```

Aquí, el **PedidoQueryService** se encarga de leer datos de la base de datos sin modificar el estado de los objetos. El **PedidoQueryController** maneja las solicitudes HTTP para realizar consultas (en este caso, obtener los detalles de un pedido).

12.4 Desafíos en la implementación de CQRS

Aunque **CQRS** ofrece muchos beneficios, también presenta ciertos desafíos:

- **Complejidad adicional:** Al separar los comandos y las consultas, la complejidad del sistema puede aumentar, ya que se deben gestionar dos conjuntos de servicios y controladores por separado.
- **Consistencia eventual:** En algunas implementaciones de CQRS, especialmente cuando se usan bases de datos separadas para comandos y consultas, puede haber una **consistencia eventual** entre las lecturas y las escrituras, lo que significa que los datos no siempre estarán sincronizados en tiempo real.
- **Mantenimiento:** Mantener dos modelos separados para lectura y escritura requiere un esfuerzo adicional en cuanto a diseño y mantenimiento del sistema.

12.5 Cuándo aplicar CQRS

El patrón **CQRS** no es necesario en todas las aplicaciones. Es más útil en sistemas donde:

- Hay una alta **carga de consultas** en comparación con las operaciones de escritura.
- Las consultas y comandos tienen diferentes **requisitos de rendimiento o escalabilidad**.
- El sistema maneja **transacciones complejas** que requieren un enfoque más granular en la segregación de responsabilidades.
- Se requiere **escalabilidad independiente** para las operaciones de lectura y escritura.

Scope

El concepto de **scope** en el desarrollo de software, particularmente en la **arquitectura hexagonal**, se refiere al **alcance** o la **visibilidad** de las clases, objetos, y componentes dentro de una aplicación. En el contexto de la arquitectura hexagonal y proyectos **Spring Boot**, el **scope** define los límites en los que las dependencias y los servicios están disponibles, lo que impacta directamente en cómo los objetos se crean, gestionan y destruyen dentro del ciclo de vida de la aplicación.

Spring Framework proporciona varios tipos de **scope** para gestionar el ciclo de vida de los beans (componentes), y comprender su correcto uso es fundamental para diseñar aplicaciones modulares, escalables y eficientes.

13.1 Tipos de scope en Spring

Spring Framework soporta varios tipos de **scope** para los beans, que determinan cómo y cuándo se crean y gestionan los objetos. Estos son los más comunes:

1. Singleton:

- **Descripción:** Es el scope por defecto en Spring. Un solo bean de este tipo es creado y compartido a lo largo de toda la aplicación. Todas las solicitudes del mismo bean obtendrán la misma instancia.
- **Uso recomendado:** Es útil cuando se necesita que el bean sea único y compartido por todos los componentes de la aplicación. Ejemplos incluyen servicios de lógica empresarial o controladores.

Ejemplo:

```
@Service
public class PedidoService {
    // Este bean será singleton, por lo tanto compartido en toda la
    // aplicación.
}
```

2. Prototype:

- **Descripción:** Cada vez que se solicita un bean con este scope, se crea una nueva instancia. A diferencia del scope singleton, cada cliente obtiene su propia instancia.
- **Uso recomendado:** Útil para componentes que necesitan ser independientes y no compartir estado, o cuando se desea un comportamiento sin estado.

Ejemplo:

```
@Scope("prototype")
@Component
public class CarritoCompra {
    // Cada solicitud crea una nueva instancia de CarritoCompra.
}
```

3. Request:

- **Descripción:** Los beans con este scope se crean una vez por cada solicitud HTTP y se destruyen al final de la solicitud. Este scope es útil en aplicaciones web.
- **Uso recomendado:** Ideal para componentes que manejan información temporal por solicitud, como datos de sesión o de formularios.

Ejemplo:

```
@Scope("request")
@Component
public class PedidoActual {
    // Este bean es creado y destruido con cada solicitud HTTP.
}
```

4. Session:

- **Descripción:** Los beans con scope de sesión se crean una vez por cada sesión HTTP y persisten durante toda la vida de la sesión.
- **Uso recomendado:** Este scope es útil para mantener datos específicos de un usuario durante toda la sesión, como información de autenticación o carrito de compras en una aplicación web.

Ejemplo:

```
@Scope("session")
@Component
public class SesionUsuario {
    // Bean que persiste durante toda la sesión del usuario.
}
```

5. Application:

- **Descripción:** Similar al scope singleton, pero está vinculado a todo el ciclo de vida de la aplicación, siendo compartido por todas las solicitudes de todos los usuarios. Útil para recursos compartidos globalmente en la aplicación.
- **Uso recomendado:** Aplicable cuando se necesita mantener un recurso común a nivel de la aplicación, como configuraciones globales o estadísticas compartidas.

Ejemplo:

```
@Scope("application")
@Component
public class ConfiguracionGlobal {
    // Este bean estará disponible en toda la aplicación.
}
```

13.2 Scope en la arquitectura hexagonal

En el contexto de la **arquitectura hexagonal**, el **scope** juega un papel importante al determinar la visibilidad y el ciclo de vida de los adaptadores, puertos y servicios. Es fundamental asegurarse de que el scope de cada componente esté bien definido según su función dentro de la arquitectura.

- **Scope en adaptadores:** Los **adaptadores** (tanto de entrada como de salida) suelen tener un **scope singleton**, ya que generalmente no mantienen estado y se utilizan a lo largo de toda la aplicación. Un adaptador como un controlador REST o un repositorio no necesita ser recreado para cada solicitud, ya que su función es servir como intermediario.
- **Scope en puertos:** Los **puertos**, que definen las interfaces de interacción con la lógica de negocio o infraestructura, también deben tener un **scope singleton** en la mayoría de los casos, ya que simplemente actúan como contratos que las implementaciones (adaptadores) cumplen.
- **Scope en servicios de lógica empresarial:** Los servicios que implementan la lógica de negocio suelen ser **singleton** si no mantienen estado o **prototype** si necesitan una instancia separada para cada solicitud o transacción.

13.3 Ejemplo de scopes en un proyecto Spring Boot basado en arquitectura hexagonal

Supongamos una aplicación de gestión de pedidos con los siguientes componentes:

- **Web Adaptor (PedidoController):** Maneja las solicitudes HTTP.
- **GestorDePedidos:** Puerto de entrada para gestionar pedidos.
- **PedidoService:** Servicio de lógica de negocio que implementa el puerto de entrada.
- **RepositorioDePedidos:** Puerto de salida para persistencia.
- **RepositorioDePedidosImpl:** Implementación del puerto de salida que utiliza JPA para interactuar con la base de datos.

PedidoController (Web Adaptor):

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    private final GestorDePedidos gestorDePedidos;

    @Autowired
    public PedidoController(GestorDePedidos gestorDePedidos) {
        this.gestorDePedidos = gestorDePedidos;
    }

    @PostMapping
    public ResponseEntity<Pedido> crearPedido(@RequestBody PedidoRequest request) {
        Pedido nuevoPedido =
```

```

gestorDePedidos.crearPedido(request.getClient(),
request.getProductos());
    return ResponseEntity.ok(nuevoPedido);
}
}

```

- **Scope:** `@RestController` en Spring es un **singleton**, lo que significa que hay una única instancia de `PedidoController` para manejar todas las solicitudes HTTP relacionadas con pedidos.

PedidoService (Servicio de lógica de negocio):

```

@Service
public class PedidoService implements GestorDePedidos {

    private final RepositorioDePedidos repositorioDePedidos;

    @Autowired
    public PedidoService(RepositorioDePedidos repositorioDePedidos) {
        this.repositorioDePedidos = repositorioDePedidos;
    }

    @Override
    public Pedido crearPedido(Cliente cliente, List<Producto> productos)
    {
        Pedido pedido = new Pedido(cliente, productos);
        return repositorioDePedidos.guardar(pedido);
    }
}

```

- **Scope:** `@Service` también tiene el scope **singleton** por defecto, lo que significa que `PedidoService` será una única instancia en la aplicación. Dado que este servicio no tiene estado, el scope singleton es adecuado.

RepositorioDePedidos (Puerto de salida):

```

@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {

    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override

```

```

public Pedido guardar(Pedido pedido) {
    return jpaPedidoRepository.save(pedido);
}

@Override
public Optional<Pedido> buscarPorId(Long id) {
    return jpaPedidoRepository.findById(id);
}
}

```

- **Scope:** `@Repository` también es un **singleton**, lo que significa que el repositorio de pedidos será una única instancia que maneja todas las operaciones de persistencia.

13.4 Consideraciones al definir scopes

- **Stateless vs Stateful:** Si un componente es **stateless** (sin estado), puede ser **singleton**. Si el componente debe mantener estado entre solicitudes o necesita manejar transacciones individuales, se debe considerar el **scope prototype** o el **scope de request**.
- **Ciclo de vida del bean:** Dependiendo de cómo se utilicen los componentes en el ciclo de vida de la aplicación (por ejemplo, por solicitud, por sesión o por transacción), es importante ajustar el scope para evitar problemas de concurrencia o compartir accidentalmente estado entre usuarios.

13.5 Beneficios del uso correcto de scope

- **Rendimiento:** Un uso adecuado del **scope singleton** mejora el rendimiento, ya que minimiza la creación de nuevos objetos innecesarios y reutiliza instancias existentes.
- **Mantenibilidad:** Definir correctamente el scope asegura que los componentes se utilicen de manera coherente y eficiente, lo que simplifica la comprensión y el mantenimiento del código.
- **Escalabilidad:** En sistemas con alta concurrencia, elegir el **scope** adecuado es crucial para evitar problemas de rendimiento o sobrecarga del sistema.