3. Diseño de componentes.

El diseño eficiente de componentes sigue principios como la **minimización de dependencias cíclicas**, el cumplimiento del **principio "open/closed"**, y la maximización de la **reusabilidad y configurabilidad**. Además, el uso de **patrones de diseño**, librerías, interfaces y protocolos de comunicación optimiza la integración de los componentes dentro del sistema.

Los componentes también deben cumplir con ciertas especificaciones, como la definición de **servicios esenciales (transacciones, seguridad, persistencia y acceso remoto)**, la formalización de **interfaces**, y la estructuración de **unidades de despliegue** que faciliten su implementación y actualización.

3.1. Principios de diseño de componentes.

La construcción de **componentes** en el contexto de **diseño de elementos software con tecnologías basadas en componentes** requiere tomar en cuenta un conjunto de principios que garantizan la **solidez** y la **mantenibilidad** del sistema. Al diseñar cada módulo, conviene prestar atención a cómo se gestionan las **dependencias**, la forma de extender o modificar la funcionalidad sin alterar partes que ya funcionan, la capacidad de reutilizar código y la posibilidad de configurar y adaptar el comportamiento del componente a entornos diversos. A continuación, se exponen los principios de diseño que resultan más **significativos** en esta tarea.

3.1.1. Dependencias no cíclicas.

El principio de **dependencias no cíclicas** sugiere que, en un sistema bien estructurado, los componentes no deberían crear bucles de referencias donde uno dependa del otro y a la vez ese otro necesite algo del primero. Esta práctica elimina situaciones confusas que pueden dificultar la actualización o la integración de nuevas funcionalidades, además de complicar el despliegue en entornos distribuidos. Por ejemplo, si un módulo de facturación depende de un módulo de clientes, pero este a su vez depende del de facturación, cualquier cambio en uno podría generar efectos en cadena y provocar errores difíciles de rastrear. Al mantener dependencias unidireccionales, la arquitectura se hace más transparente y reduce el acoplamiento, lo que favorece la escalabilidad del proyecto.

3.1.2. Principio “open/closed”.

El principio “open/closed” sostiene que un componente debería estar **abierto a la extensión** pero **cerrado a la modificación**. Esto significa que, cuando se añaden nuevas características o se ajusta la lógica existente, no debería ser necesario alterar el código original que funciona correctamente. Para lograrlo, se acostumbra a establecer **interfaces** o puntos de extensión bien definidos, de modo que otros desarrolladores puedan ampliar la funcionalidad sin introducir errores o afectar la estabilidad de lo que ya existe. Un ejemplo podría ser un módulo de pago que ofrezca un punto de entrada para nuevos métodos de cobro, de manera que si se integra un proveedor adicional, el código base se mantenga intacto y la nueva lógica se añada mediante la implementación de la interfaz correspondiente.

3.1.3. Reusabilidad.

La **reusabilidad** abarca la idea de que cada componente debería diseñarse con la intención de ser empleado en distintos proyectos o en diferentes contextos dentro de un mismo sistema. Cuando las clases y métodos se crean con un propósito muy específico y no contemplan casos de uso más

amplios, la posibilidad de reaprovecharlos disminuye considerablemente. Para favorecer la reusabilidad, es **adecuado** pensar en componentes más genéricos, modularizar la lógica y apoyarse en convenciones que permitan la configuración o personalización de aspectos puntuales sin duplicar funcionalidad. Esto ahorra tiempo de desarrollo y homogeneiza la forma en que se construyen y mantienen los proyectos en la organización.

3.1.4. Configurabilidad.

La **configurabilidad** determina el grado de **flexibilidad** que un componente ofrece para ajustarse a distintos entornos o requisitos sin tener que modificar su código. Imaginemos un servicio de notificaciones que, según la configuración, puede enviar correos electrónicos o mensajes SMS. Al permitir parámetros externos, archivos de propiedades o directivas de entorno, el mismo componente puede operar de maneras diversas sin transformarse en un bloque rígido. Esto adquiere más relevancia en la era de la computación en la nube y los contenedores, donde cada instancia puede requerir ajustes específicos de memoria, rutas de acceso o incluso credenciales de seguridad. Un diseño configurado de esta forma hace posible reutilizar el mismo artefacto en escenarios diferentes, simplificando la administración y la optimización de recursos.

3.1.5. Abstracción.

La **abstracción** se refiere a ofrecer una visión simplificada de la complejidad, de modo que las capas superiores no necesiten preocuparse por los detalles internos. Al diseñar componentes, se busca exponer **interfaces claras** que escondan la lógica subyacente, permitiendo que otras partes del sistema interactúen con el componente sin tener que conocer su implementación exacta. Este enfoque hace que la sustitución de un componente por otro sea más sencilla, siempre y cuando se preserve el mismo contrato. Además, refuerza la idea de separar la lógica de negocio de la forma en que se realiza dicha lógica. Un buen ejemplo se ve en una capa de persistencia que maneja la comunicación con la base de datos, mientras que el resto del sistema se limita a enviar peticiones y recibir resultados, sin importar cómo se conecta o se ejecutan las consultas internamente.

3.1.6. Dependencias.

La gestión de **dependencias** engloba cómo y en qué forma se asocian los componentes entre sí y con librerías externas. Este ámbito incluye las versiones utilizadas y la forma en que se inyectan o se resuelven dichas dependencias. Herramientas de construcción y empaquetado como Maven, Gradle o NuGet han simplificado el manejo de dependencias en lenguajes como Java y C#. Sin embargo, es **necesario** no abusar de librerías externas o de terceros que puedan introducir vulnerabilidades o desalineaciones con la estrategia del proyecto. Además, un principio esencial consiste en definir los **contratos mínimos** que se precisan para que un componente funcione. Esto impide caer en dependencias excesivas que pueden ralentizar el desarrollo o causar incompatibilidades difíciles de solucionar.

Actividad 8

El principio “open/closed” es fundamental para la extensibilidad del software. Busca un ejemplo de su aplicación en un lenguaje de programación como Java, C# o Python, y explica cómo el código se diseñó para permitir nuevas funcionalidades sin modificar la estructura existente.

Las dependencias pueden generar acoplamientos innecesarios en los sistemas. Investiga qué herramientas existen para gestionar dependencias en un entorno de desarrollo moderno (por ejemplo, Maven, Gradle o NuGet) y describe cómo ayudan a mejorar la escalabilidad y mantenibilidad de los proyectos.

3.2. Técnicas de reusabilidad.

Las **técnicas de reusabilidad** sirven para crear soluciones en las que los componentes pueden aprovecharse en múltiples proyectos o escenarios sin tener que reconstruir la misma lógica una y otra vez. Este principio resulta **vital** en el **diseño de elementos software con tecnologías basadas en componentes**, ya que un sistema compuesto por módulos reutilizables tiende a ser más sencillo de mantener y ampliar. A continuación, se exponen diferentes métodos que ayudan a lograr este objetivo, abarcando desde los fundamentos de la arquitectura hasta la elección de lenguajes de programación y esquemas de comunicación.

3.2.1. Patrones.

Los **patrones de diseño** se han convertido en un punto de referencia para los desarrolladores que enfrentan retos comunes en el desarrollo de software. En la década de los noventa, obras como *Design Patterns: Elements of Reusable Object-Oriented Software* popularizaron soluciones como *Factory Method*, *Observer* o *Singleton*, que permiten abordar problemas recurrentes a través de estructuras de clases y objetos ya experimentadas en numerosos proyectos. Cuando estos patrones se trasladan a un entorno de componentes, se traduce en una mayor uniformidad en la forma de diseñar, pues cada módulo puede implementar o colaborar con otros siguiendo lineamientos claros y reconocidos. Además, los patrones fomentan un lenguaje compartido entre los equipos, lo que simplifica la comunicación y acelera la resolución de incidencias.

3.2.2. Librerías.

Las **librerías** representan uno de los mecanismos más **efectivos** para promover la reusabilidad. Permiten agrupar funcionalidad genérica que distintos componentes pueden invocar sin volver a programar la misma lógica. En lenguajes como Java, se empaquetan como JAR; en .NET, se generan DLL; y en Python, se distribuyen como paquetes instalables. Estas colecciones pueden abarcar desde utilidades matemáticas o criptográficas hasta conectores con servicios externos, como plataformas de pago o almacenamiento en la nube. Al adoptar librerías versionadas, se pueden realizar mejoras o correcciones de fallos de manera centralizada, asegurando que todos los

componentes que dependan de ellas puedan aprovechar esos cambios con una simple actualización, siempre que se respete la compatibilidad de interfaces.

3.2.3. Interfaces.

Las **interfaces** establecen el contrato que define cómo un componente se comunica con otro. Al especificar métodos y tipos de datos sin descender a detalles de implementación, se logra separar la manera en que funciona un módulo internamente de la forma en que otros elementos lo utilizan. Esto es muy **valioso** para la reusabilidad, porque si una interfaz permanece estable, es posible sustituir o mejorar el componente sin que los consumidores se vean afectados. En un entorno de microservicios, por ejemplo, las interfaces se pueden expresar mediante descripciones en OpenAPI/Swagger, lo que facilita generar clientes y servidores en diferentes lenguajes y favorece la integración de módulos que incluso pueden correr en plataformas distintas.

3.2.4. Protocolos y esquemas de mensajes.

En arquitecturas distribuidas, los **protocolos y esquemas de mensajes** desempeñan un papel determinante para la colaboración entre componentes. Protocolos como HTTP, gRPC o AMQP se usan ampliamente para comunicarse a través de la red, aprovechando estándares abiertos y librerías que simplifican la implementación. Asimismo, los **esquemas** (por ejemplo, JSON, Protobuf o Avro) describen la estructura de los datos intercambiados, de modo que cada parte del sistema sepa exactamente qué esperar. Al definirse estos esquemas de forma independiente del código, se otorga más **flexibilidad** a los desarrolladores, que pueden actualizar o extender campos sin romper las integraciones existentes, siempre que respeten los elementos ya establecidos. Esta estrategia encaja muy bien con la evolución constante que se da en los sistemas actuales.

3.2.5. Uso de lenguajes de programación.

La reusabilidad también depende del **lenguaje de programación** que se elija, dado que cada uno ofrece ciertas características que pueden facilitar o entorpecer el diseño de componentes genéricos. Por ejemplo, lenguajes orientados a objetos como Java o C# brindan mecanismos nativos (interfaces, clases abstractas, genéricos) que promueven la creación de bloques independientes y altamente configurables. En tanto, lenguajes como Python o JavaScript ofrecen un enfoque dinámico que puede resultar más flexible, pero demanda mayor disciplina para asegurar la coherencia de los contratos a lo largo del tiempo. La adopción de un lenguaje u otro debe analizarse en función de las necesidades del proyecto, la experiencia del equipo y los requerimientos de rendimiento o compatibilidad con otras tecnologías.

3.2.6. Estructuras y jerarquías de estructuras.

El modo en que se organizan los directorios, módulos y paquetes dentro de un proyecto es **decisivo** para el reuso, ya que, si el código está enredado y disperso, la extracción de un componente para emplearlo en otro proyecto se complica. Un ejemplo sería dividir el sistema en capas (presentación, negocio, acceso a datos) y, dentro de cada capa, crear submódulos que atiendan diferentes dominios funcionales. En muchas organizaciones, se siguen convenciones estándar (por ejemplo, la convención de paquetes en Java o la estructura modular de Node.js) para que los desarrolladores reconozcan rápidamente dónde se hallan los componentes y cuál es su responsabilidad. Esta jerarquía también facilita la aplicación de herramientas de automatización, que se benefician de estructuras de proyecto consistentes para compilar, probar y desplegar.

3.2.7. Arquitecturas de sistemas.

La **arquitectura de sistemas** tiene un **impacto** directo en la reusabilidad de los componentes. Diseños monolíticos con alto acoplamiento tienden a dificultar el reaprovechamiento de funcionalidad, pues todo se encuentra integrado en un solo bloque. Sin embargo, si se adopta una visión modular o de microservicios, cada pieza puede transformarse en un producto autónomo que cumpla con una función específica dentro de la organización o incluso en distintos proyectos. Además, arquitecturas donde se promueve la independencia de cada componente (por ejemplo, usando contenedores o servicios de mensajería) hacen que sea más fácil incorporar otros lenguajes de programación y nuevos equipos de trabajo. Esto incrementa la **colaboración** y permite aprovechar librerías o soluciones existentes sin tener que rehacer la infraestructura de comunicación.

3.3. Modelo de componente.

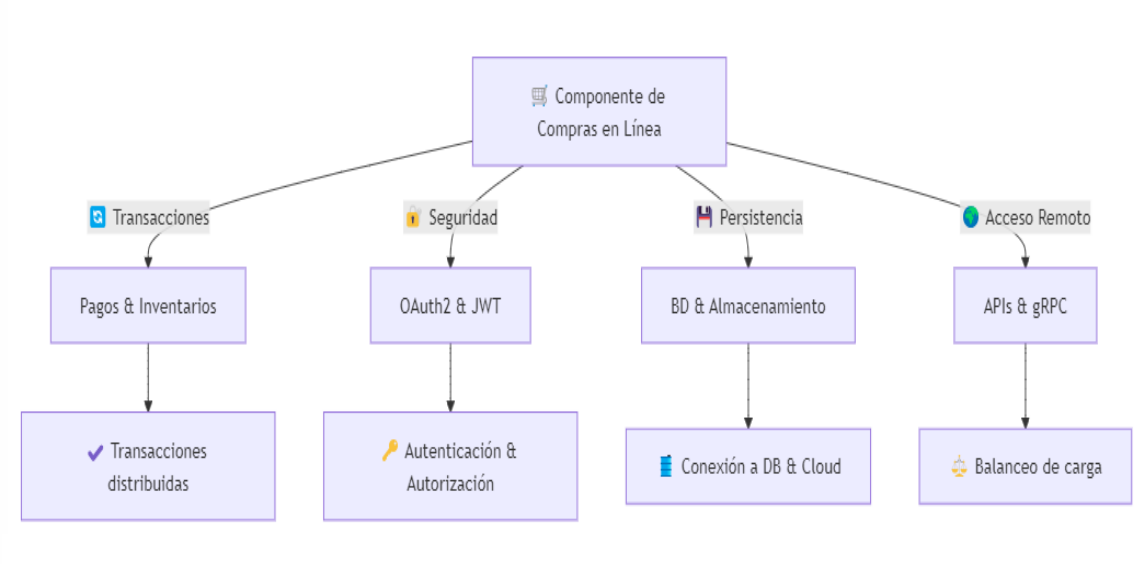
El **modelo de componente** define de manera detallada la forma en que cada parte del sistema se diseñará, se comunicará con las demás y será desplegada. En el **diseño de elementos software con tecnologías basadas en componentes**, este modelo marca las pautas para garantizar que el desarrollo se lleve a cabo de forma coherente, aprovechando las ventajas que ofrecen la modularidad y la independencia de cada bloque. Cada componente debe especificar claramente sus servicios, su interfaz, la forma en que se implementa y cómo se empaqueta para su ejecución. A continuación, se explican los distintos aspectos de un modelo de componente y su relevancia en proyectos donde la escalabilidad y la mantención continua son factores muy **importantes**.

3.3.1. Especificación de servicios: transacciones, seguridad, persistencia y acceso remote.

En el modelo de componente, resulta **esencial** describir los servicios que provee, incluyendo aquellos referentes a transacciones, seguridad, persistencia y acceso remoto. Por ejemplo, un componente que maneje compras en línea podría ofrecer transacciones distribuidas que abarquen varios pasos, integrando pagos e inventarios de manera confiable. En términos de **seguridad**, deben definirse controles de autenticación y autorización, quizá empleando estándares como OAuth2 o JWT en un entorno de microservicios. Para la **persistencia**, un componente puede trabajar con diversas bases de datos o incluso servicios de almacenamiento en la nube, por lo que es fundamental describir cómo se gestionan las conexiones o cómo se mantiene la integridad de los datos.

El **acceso remoto** también adquiere mucha importancia, ya que en una arquitectura moderna los componentes suelen desplegarse en distintos contenedores, servidores o regiones geográficas. Si el servicio se expone mediante APIs REST, gRPC o colas de mensajería, la especificación debe indicar claramente los endpoints y los protocolos admitidos. Así se garantiza que otros componentes o clientes externos sepan con precisión cómo consumir el servicio.

El siguiente diagrama representa los servicios esenciales que puede proveer un componente en un modelo de arquitectura basado en componentes:

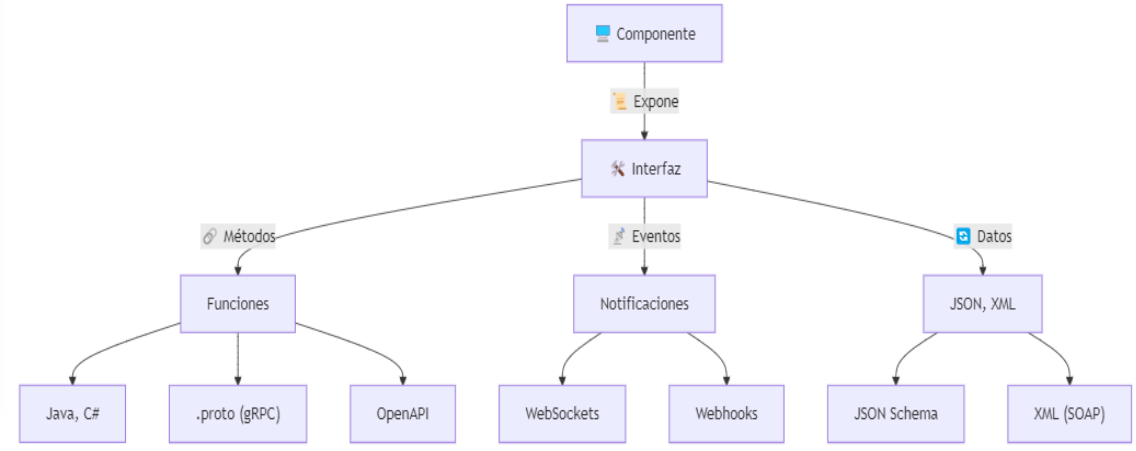


1. Nodo central: Componente de Compras en Línea
    - Representa un módulo clave dentro del sistema.
2. Servicios principales (niveles compactos):
    - Transacciones: Maneja pagos e inventarios, asegurando transacciones distribuidas.
    - Seguridad: Implementa OAuth2, JWT y control de acceso.
    - Persistencia: Se conecta a bases de datos y almacenamiento en la nube.
    - Acceso remoto: Expone APIs y gRPC con balanceo de carga.

3.3.2. Especificación de Interface.

La **especificación de la interfaz** define qué métodos, endpoints o eventos ofrece el componente a los consumidores. Al adoptar principios de diseño como la independencia y el encapsulado, la interfaz se convierte en el **contrato** con el que otras partes del sistema pueden interactuar sin conocer la lógica interna. Dentro de entornos de desarrollo como Java, C# o Python, esta descripción puede plasmarse en forma de clases de interfaz, documentación API o archivos de definición (por ejemplo, archivos .proto en gRPC). Si el sistema está basado en servicios web, la interfaz puede describirse con OpenAPI o Swagger, asegurando que todos los equipos entiendan con claridad los parámetros, tipos de datos y formatos de respuesta.

El siguiente diagrama representa la especificación de la interfaz en un sistema basado en componentes, asegurando la comunicación entre módulos de manera independiente y encapsulada:



1. Nodo principal: Componente

- Representa un módulo que proporciona servicios o funcionalidades.
- Expone una interfaz que define cómo otros sistemas pueden interactuar con él.

2. Interfaz (Contrato del componente):

- Define los métodos, eventos y formatos de datos permitidos.
- Garantiza independencia y encapsulación, ocultando la lógica interna.

3. Detalles de la interfaz:

- Métodos:
  - Implementados como clases de interfaz en lenguajes como Java, C#.
  - Definidos en .proto para gRPC o documentados con OpenAPI/Swagger en REST.
- Eventos:
  - Usa mecanismos como WebSockets, MQTT para comunicación en tiempo real.
  - Permite suscripciones con Webhooks en sistemas event-driven.
- Formatos de datos:
  - Datos en JSON o XML, dependiendo del estándar (REST, SOAP).
  - ProtoBuf en entornos gRPC para mayor eficiencia.

Mantener la interfaz estable es **muy necesario** para la evolución del sistema. Cuando se necesita agregar funcionalidades, la práctica habitual es incorporar nuevos métodos o versiones, sin alterar drásticamente los existentes, para no romper la compatibilidad con los consumidores que confían en la versión anterior de la interfaz.

3.3.3. Especificación de la implementación.

La **implementación** del componente comprende la lógica real que ejecuta los servicios, a menudo basada en clases, patrones de diseño y uso de librerías o frameworks. Aquí se definen los algoritmos, la estructura de datos y las interacciones con sistemas externos, siempre siguiendo los lineamientos establecidos en la interfaz. En muchos casos, cada implementación puede dividirse en submódulos o capas internas, de manera que la parte que maneja la persistencia se separe de la lógica de negocio principal. Esto se traduce en un **mantenimiento** más sencillo, ya que se puede actualizar una librería o mejorar un método sin influir en el resto del componente.

Asimismo, es **interesante** señalar que un mismo contrato (interfaz) puede contar con diferentes implementaciones para escenarios distintos. Por ejemplo, un componente que manipule datos podría tener una versión en memoria para pruebas rápidas y otra con persistencia real en una base de datos SQL o NoSQL. Mientras se respete la misma interfaz, el resto de los sistemas pueden emplear el componente sin enterarse de los cambios internos.

Ejemplo

Supongamos que estamos desarrollando un componente de autenticación para una aplicación web. La interfaz del componente define los métodos iniciarSesion(usuario, contraseña) y validarToken(token), pero la implementación puede variar según las necesidades del entorno.

1. Implementación con Base de Datos SQL:

   - Se usa Spring Boot (Java) con Spring Security para manejar autenticación. Spring Security proporciona filtros de autenticación, validación de usuarios y manejo de

sesiones sin necesidad de programar cada aspecto manualmente. Cuando agregamos Spring Security a una aplicación de Spring Boot, automáticamente protege todos los endpoints de la API, exigiendo autenticación.

- Se almacena la información en PostgreSQL con la tabla usuarios(id, nombre, contraseña_hash).


```
       Columna  Tipo de Dato  \
            id        SERIAL
        nombre  VARCHAR(100)
        correo  VARCHAR(150)
contraseña_hash          TEXT
           rol   VARCHAR(50)

                                  Restricciones
                                    PRIMARY KEY
                                       NOT NULL
                                 UNIQUE NOT NULL
                                       NOT NULL
CHECK ('admin', 'usuario', 'moderador') DEFAUL...
```

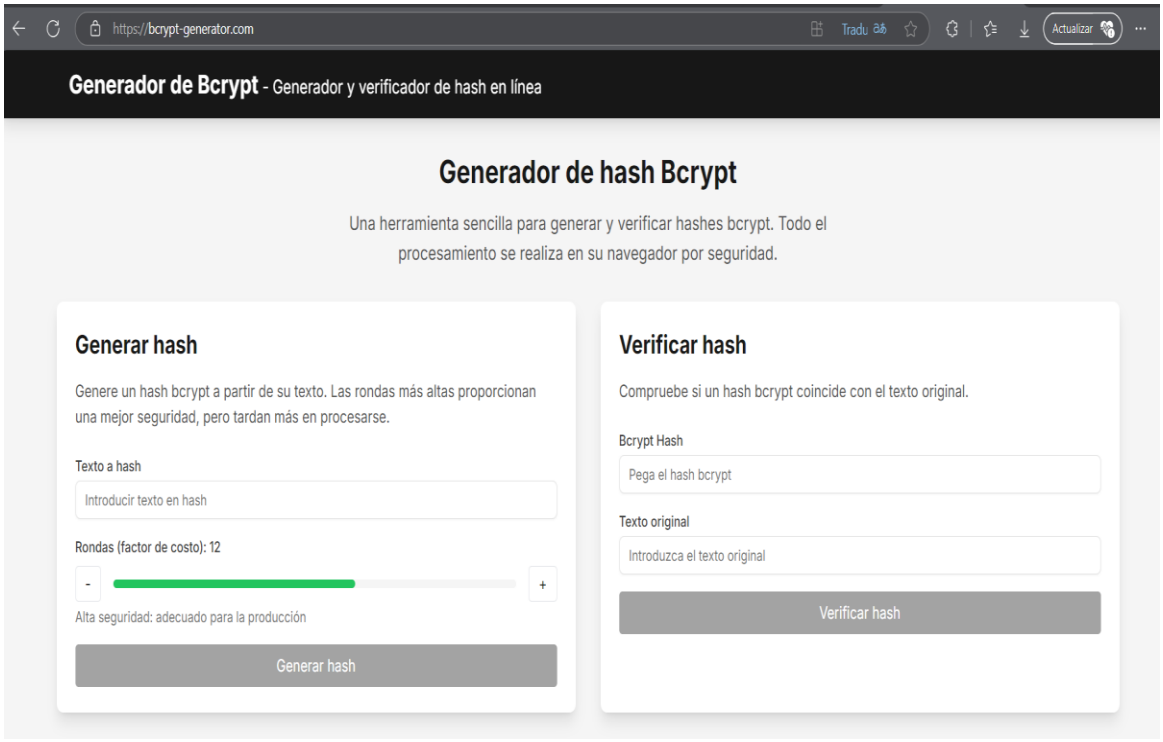
Estructura de la Tabla 'usuarios' en PostgreSQL

Columna	Tipo de Dato	Restricciones
id	SERIAL	PRIMARY KEY
nombre	VARCHAR(100)	NOT NULL
correo	VARCHAR(150)	UNIQUE NOT NULL
contraseña_hash	TEXT	NOT NULL
rol	VARCHAR(50)	CHECK ('admin', 'usuario', 'moderador') DEFAULT 'usuario'
creado_en	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP

Usuarios en la Tabla 'usuarios' en PostgreSQL

ID	Nombre	Correo	Rol	Creado En
1	Amarïe	amarie@elfos.com	admin	2024-02-11 10:00:00
2	Arwen	arwen@elfos.com	usuario	2024-02-11 10:05:00
3	Caranthir	caranthir@elfos.com	moderador	2024-02-11 10:10:00
4	Celeborn	celeborn@elfos.com	usuario	2024-02-11 10:15:00
5	Celebrindal	celebrindal@elfos.com	admin	2024-02-11 10:20:00
6	Curufin	curufin@elfos.com	usuario	2024-02-11 10:25:00
7	Elentari	elentari@elfos.com	moderador	2024-02-11 10:30:00
8	Elrond	elrond@elfos.com	admin	2024-02-11 10:35:00
9	Ëowyn	eowyn@elfos.com	usuario	2024-02-11 10:40:00
10	Fëanor	feanor@elfos.com	usuario	2024-02-11 10:45:00
11	Fingolfin	fingolfin@elfos.com	admin	2024-02-11 10:50:00
12	Galadriel	galadriel@elfos.com	moderador	2024-02-11 10:55:00

- Se usa BCrypt para almacenar las contraseñas de forma segura.



- Se generan tokens JWT para la autenticación de usuarios.

2. Implementación en Memoria para Pruebas:
   - Para evitar depender de una base de datos en pruebas unitarias, se implementa una versión en memoria.
   - Se usa un HashMap<String, String> para almacenar usuarios temporalmente sin persistencia. Este servicio almacenará usuarios temporalmente en un HashMap<String, String>, donde la clave es el email y el valor es la contraseña cifrada con BCrypt:

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

@Service
public class AuthServiceMemoria {
    private final Map<String, String> usuarios = new HashMap<>();
    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    public AuthServiceMemoria() {
        // Usuarios de prueba en memoria
        registrarUsuario("usuario1@pruebas.com", "contraseña123");
        registrarUsuario("usuario2@pruebas.com", "claveSegura");
    }

    public void registrarUsuario(String email, String password) {
        usuarios.put(email, passwordEncoder.encode(password));
    }

    public boolean autenticarUsuario(String email, String password) {
        return Optional.ofNullable(usuarios.get(email))
                .map(hash -> passwordEncoder.matches(password, hash))
                .orElse(false);
    }
}
```

- Este controlador expone los endpoints /auth/login y /auth/register, permitiendo a los usuarios autenticarse y registrarse en la memoria temporal.

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/auth")
public class AuthControllerMemoria {
    private final AuthServiceMemoria authService;
```

```
    public AuthControllerMemoria(AuthServiceMemoria authService) {
        this.authService = authService;
    }

    @PostMapping("/register")
    public String registrar(@RequestParam String email, @RequestParam String password) {
        authService.registrarUsuario(email, password);
        return "Usuario registrado en memoria: " + email;
    }

    @PostMapping("/login")
    public String login(@RequestParam String email, @RequestParam String password) {
        if (authService.autenticarUsuario(email, password)) {
            return "Inicio de sesión exitoso para " + email;
        }
        return "Credenciales incorrectas";
    }
}
```

1. Al iniciar la aplicación, se crean dos usuarios de prueba en memoria.
2. Si un usuario quiere registrarse, se usa /auth/register?email=usuario@correo.com&password=clave.
3. Para iniciar sesión, se hace un POST a /auth/login?email=usuario@correo.com&password=clave.
4. No hay persistencia, ya que los datos se almacenan en un HashMap<String, String>, útil para pruebas rápidas sin depender de PostgreSQL.

- Esta variante es útil para tests automatizados sin necesidad de configurar PostgreSQL.

3. Interoperabilidad:

- Ambas implementaciones respetan la misma interfaz (Autenticador), por lo que los consumidores no necesitan saber qué versión están usando.
- Un entorno de desarrollo puede ejecutar la versión en memoria, mientras que producción usa la implementación con base de datos.

3.3.4. Especificación de las unidades de despliegue (modulos).

La **especificación de las unidades de despliegue** describe cómo se empaquetan y distribuyen los componentes para su ejecución. Cada módulo puede representarse como un archivo JAR, WAR o EAR en el ecosistema Java, una DLL en .NET o un contenedor Docker en un escenario de microservicios. Aquí se definen detalles como las dependencias necesarias, los requisitos de configuración (variables de entorno, archivos de propiedades, etc.) y los metadatos para la orquestación de los servicios.

En una arquitectura distribuida, cobra especial relevancia la descripción de cómo se replican los componentes, cómo se monitorizan y qué escalado se aplica. Se pueden incluir también políticas de versionado y migración, para que, al lanzar una nueva versión del módulo, los consumidores sepan si se mantiene la compatibilidad hacia atrás o deben realizar ajustes. Esta aproximación estructurada permite a los equipos de desarrollo y operación planificar la entrega continua (CI/CD) y la puesta en producción con pocas sorpresas, contribuyendo a la **estabilidad** general del sistema.

Ejemplo

Consideremos cómo se distribuye y despliega el componente de autenticación en diferentes entornos.

1. Despliegue en Java con JAR:
    - Se empaqueta el componente como un archivo JAR (autenticacion.jar).
    - Sus dependencias incluyen Spring Boot, PostgreSQL JDBC Driver y JWT Library.
    - Se ejecuta en un servidor con java -jar autenticacion.jar.
2. Despliegue en Microservicios con Docker:
    - Se define un Dockerfile con la imagen de openjdk:17.
    - Se establece la configuración por variables de entorno (DB_HOST, SECRET_KEY).
    - Se usa Kubernetes para escalar automáticamente cuando hay muchas solicitudes.
3. Monitorización y Versionado:
    - Se integran logs con Prometheus y alertas con Grafana.
    - Las versiones siguen Semantic Versioning (1.2.3) para evitar incompatibilidades en producción.
    - Se utiliza un pipeline de CI/CD en GitHub Actions para automatizar despliegues seguros.

3.4. Modelos de integración de componentes.

En un entorno de **diseño de elementos software con tecnologías basadas en componentes**, se requiere establecer métodos claros y confiables para que los distintos módulos del sistema se reconozcan, colaboren y compartan información entre sí. Los **modelos de integración de componentes** abarcan desde la referencia y el uso de identidades únicas, hasta la elección de mecanismos de comunicación como objetos distribuidos, servicios web o la utilización de middleware especializado. Este proceso de integración no solo afecta a la manera en que los componentes se localizan y llaman unos a otros, sino también a la forma en que se escalan, se actualizan y se mantienen en el tiempo. A continuación, se describen algunos elementos clave de estos modelos, dando una perspectiva de cómo se organizan las interacciones entre componentes en sistemas modernos.

3.4.1. Referencias e identidad de objetos, componentes e interfaces.

En muchos lenguajes de programación, las **referencias** permiten acceder a objetos en memoria, pero en un sistema de componentes distribuidos cada módulo puede hallarse en un proceso, un contenedor o incluso un servidor distinto. Por lo tanto, se necesita un modo de identificar de forma inequívoca cada objeto o servicio. Esta **identidad** puede expresarse mediante identificadores únicos universales (UUID), nombres lógicos o rutas de red. Por ejemplo, un componente denominado "CarritoDeCompra" en una arquitectura de microservicios tal vez se identifique en el entorno de Kubernetes con un nombre concreto, o a través de una URL si se trata de un servicio web.

Asimismo, en sistemas de objetos distribuidos tradicionales (como CORBA o RMI en Java), la identidad se encapsula en "stubs" o proxies que representan a los objetos remotos. En las interfaces, esta identidad se complementa con los métodos o endpoints que pueden invocarse. Cada vez que un cliente requiere una operación, debe contar con una referencia válida que le indique dónde y cómo puede llamar al componente. Este aspecto se vuelve **trascendental** cuando hay múltiples réplicas de un mismo servicio, ya que la identidad y la referencia deben apuntar a una instancia viva y funcional, incluso si la infraestructura realiza cambios en segundo plano para balancear la carga o reemplazar contenedores.

Ejemplo

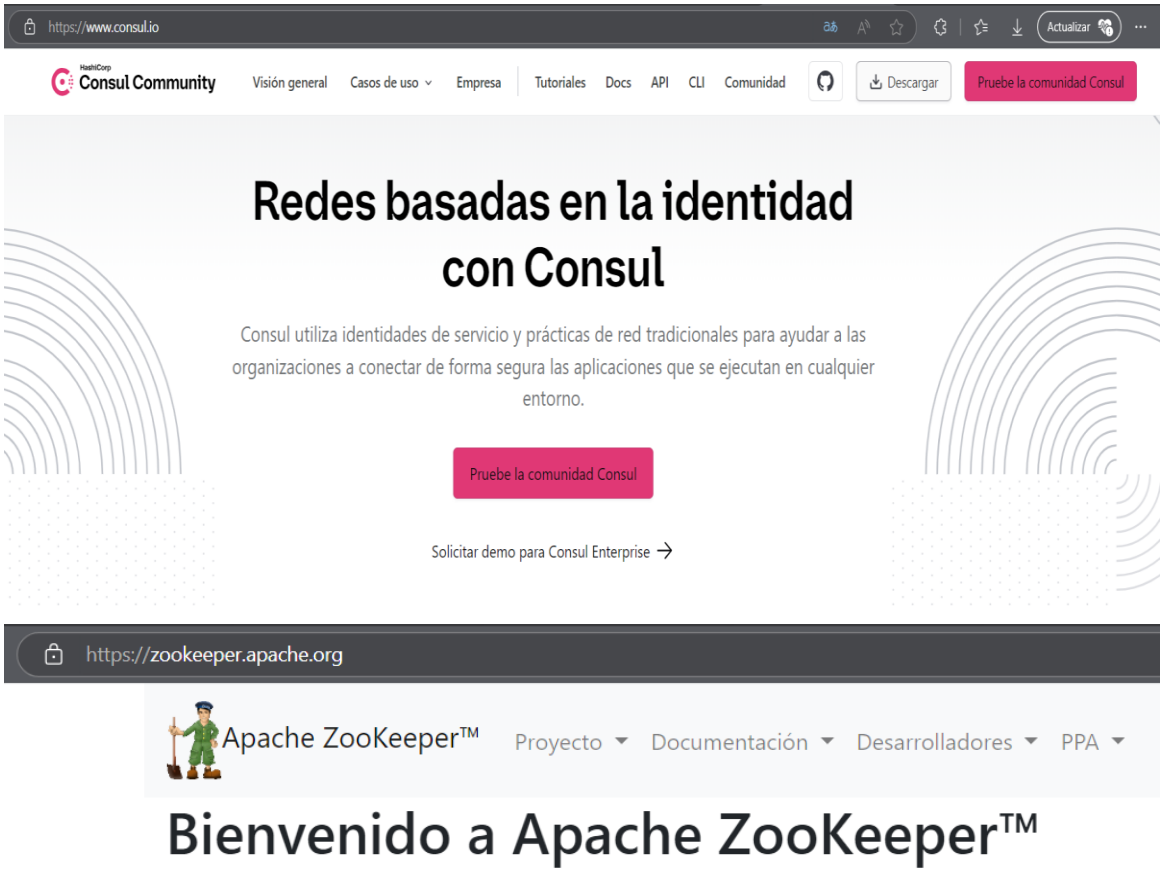
Imagina una aplicación de comercio electrónico basada en microservicios. En este sistema, el componente CarritoDeCompra debe almacenar los productos que un usuario añade antes de realizar la compra. Como el sistema está distribuido en contenedores dentro de Kubernetes, cada instancia del carrito podría tener una identidad única generada por un UUID. De este modo, cuando un usuario inicia sesión, el sistema puede recuperar su carrito mediante un identificador como cart-1234-5678-9101. Si el sistema utilizara solo referencias en memoria sin un identificador persistente, cada vez que un contenedor del microservicio se reiniciara, la información del carrito se perdería.

Supongamos que una empresa de logística tiene un sistema distribuido donde varios servidores gestionan pedidos en diferentes almacenes. Utilizando Java RMI, cada servidor almacena objetos de pedidos en diferentes ubicaciones geográficas. Para que un cliente pueda consultar el estado de su pedido, necesita una referencia a un objeto remoto. En este caso, cuando un usuario busca su pedido #98765, el sistema obtiene un stub (proxy) que representa el objeto remoto del pedido y permite interactuar con él como si fuera un objeto local. Sin este mecanismo de referencia, cada solicitud debería rastrear manualmente en qué servidor se encuentra el pedido, complicando la escalabilidad.

En un sistema de banca en línea, múltiples servidores manejan solicitudes de clientes que quieren consultar el saldo de su cuenta. Si la API de CuentasBancarias tiene varias réplicas activas, cada solicitud HTTP debe dirigirse a una instancia funcional. Para lograrlo, los balances de carga modernos utilizan identificadores lógicos como https://api.banco.com/cuentas/{id} en lugar de direcciones IP fijas. Así, cuando un cliente consulta su cuenta 12345678, la infraestructura selecciona dinámicamente un servidor disponible que maneje la solicitud. Si un servidor deja de estar disponible, la identidad del servicio se mantiene sin que los clientes necesiten conocer detalles internos.

3.4.2. Servicios de localización

Para que los componentes se encuentren y se comuniquen entre sí, suelen implementarse **servicios de localización**, también denominados servicios de descubrimiento o *service discovery*. Herramientas como **Consul** o **Zookeeper** facilitan que cada componente registre su ubicación y sepa la de los demás. De este modo, si un microservicio necesita llamar a un servicio de facturación, no recurre a una dirección fija, sino que consulta al servicio de localización para saber qué instancias están disponibles y en qué hosts o puertos se encuentran. Esto hace que la arquitectura sea más **dinámica**, pues los componentes pueden arrancar y apagarse sin afectar a otros, siempre que se actualice la información en el servicio de localización.



En sistemas más antiguos, este rol se cumplía mediante directorios globales, como servidores LDAP o modelos de nombres en CORBA, donde cada objeto remoto se anunciaba para que los clientes supieran cómo acceder a él. Aunque dichas soluciones no han desaparecido, en la actualidad los despliegues en contenedores y la adopción masiva de la nube han impulsado la popularidad de soluciones más orientadas a la virtualización y la escalabilidad automática, lo que coloca a los servicios de localización en un lugar predominante dentro de las arquitecturas distribuidas.

3.4.3. Modelos de intercambio: objetos distribuidos, capa intermedia (Middleware) e interacción e integración mediante servicios web.

Como hemos visto en apartados anteriores, existen distintos **modelos de intercambio** que dan forma a la interacción entre componentes, reflejándose en la manera en que se envían peticiones, se reciben respuestas y se comparten eventos:

1. **Objetos distribuidos**: Se basan en la idea de que los objetos pueden estar en distintos procesos o máquinas, pero los desarrolladores pueden llamarlos como si estuvieran en la

misma aplicación. Ejemplos históricos incluyen CORBA o DCOM, y en el ecosistema Java, RMI. Este esquema supone que la llamada a un método remoto es similar a la de uno local, aunque en realidad se realice mediante red y requiera serializar parámetros y resultados. Suelen proporcionar un alto nivel de transparencia, pero pueden imponer complejidades asociadas a la latencia de red, a la gestión de proxies y a la necesidad de un runtime compatible en ambas partes.

2. **Capa intermedia (Middleware)**: En lugar de llamar directamente a objetos remotos, se introduce un **middleware** que ofrece servicios transversales, como balanceo de carga, seguridad unificada o gestión de mensajería. De esta forma, cada componente se comunica con la capa intermedia, y no directamente con otros componentes. Es el middleware quien se encarga de enrutar las peticiones, ofrecer transacciones distribuidas o coordinar los mensajes. Esta aproximación, típica de los servidores de aplicaciones JEE o de buses de servicios empresariales (ESB), se integra frecuentemente en contextos corporativos para unificar la comunicación entre sistemas legados y aplicaciones nuevas. En la actualidad, muchos proyectos adoptan colas de mensajería como RabbitMQ, ActiveMQ o Kafka para lograr un acoplamiento más bajo y una interacción asíncrona.
3. **Interacción e integración mediante servicios web**: Con la explosión de la web y las arquitecturas de microservicios, los **servicios web** (REST, SOAP, gRPC, GraphQL, etc.) se han convertido en un método **muy extendido** para la integración de componentes. Cada servicio se expone como un endpoint, y los clientes se comunican usando protocolos HTTP o HTTP/2, intercambiando datos en formatos como JSON o Protobuf. Esta metodología aporta un acoplamiento moderado, ya que cada componente puede evolucionar de manera independiente mientras mantenga la compatibilidad con los esquemas y endpoints definidos. Al combinarse con servicios de localización y herramientas de despliegue automático, se obtiene una arquitectura flexible y propicia para la evolución continua.

La elección entre objetos distribuidos, middleware o servicios web depende de factores como la complejidad del sistema, la disponibilidad de entornos runtime, la necesidad de transacciones y la adopción de patrones como sincronicidad o asincronía. Lo más frecuente hoy en día es emplear un modelo basado en APIs web, ya que permite integrar distintos lenguajes y plataformas con un coste relativamente bajo de desarrollo y mantenimiento. Aun así, sectores que manejan transacciones muy estrictas o requieren baja latencia pueden optar por enfoques más específicos.

A continuación, se presenta una tabla que muestra ejemplos **reales y concretos**, describiendo distintas situaciones y la **elección** más adecuada entre **objetos distribuidos**, **middleware** o **servicios web**, junto con el **motivo** principal de la selección. Cada escenario parte de necesidades específicas, como la complejidad del sistema, la forma de desplegar o la exigencia de transacciones:

Escenario	*Complejidad o requisitos*	*Enfoque más adecuado*	*Razón de la elección*
Pequeña plataforma de comercio electrónico que ofrece un catálogo y un módulo de pago	Escenario web, demanda moderada, equipos con experiencia en REST, variedad de lenguajes	Servicios web (APIs REST)	Integración sencilla entre frontend y backend, aprovechamiento de librerías disponibles para JSON y HTTP, flexibilidad al gestionar datos en diferentes lenguajes
Empresa grande con sistemas heredados (ERP y CRM) y nuevas aplicaciones	Infraestructura heterogénea, datos en varios formatos, equipos que trabajan	Middleware, por ejemplo un bus	Proporciona funcionalidades de enrutado y transformación de mensajes, integra

que deben intercambiar información	en plataformas distintas	empresarial de servicios	sistemas antiguos y modernos, maneja seguridad y transacciones sin exponer complejidad al resto
Plataforma financiera con necesidad de transacciones distribuidas en múltiples pasos	Datos muy delicados, coherencia total ante fallos, cumplimiento de estándares estrictos	Middleware transaccional o enfoque de objetos distribuidos en casos puntuales	Garantiza consistencia en cada operación, ofrece rollback automático y control centralizado de las transacciones, fundamental para entornos bancarios
Sistema multijugador en línea con juegos en tiempo real y mínima latencia	Latencia muy baja, sincronización inmediata, posible implementación con sockets o protocolos específicos	Objetos distribuidos muy especializados o uso de sockets directos	Llamadas casi instantáneas para coordinar movimientos, se busca reducir el retardo de la red, se suelen usar protocolos como WebSockets o UDP
Microservicios que gestionan eventos de una plataforma de dispositivos conectados	Muchos productores y consumidores de datos, necesidad de comunicación asincrónica, escalado rápido	Servicios web con mensajería asíncrona (Kafka, RabbitMQ, etc)	Acoplamiento flexible mediante suscripciones y colas, cada microservicio publica y consume eventos sin bloquear a los demás
Red de logística con sucursales distribuidas y conexiones de red poco confiables	Caídas frecuentes en la comunicación, requerimiento de alta disponibilidad, picos de tráfico variables	Middleware de mensajería asíncrona	Garantiza reintentos y entrega fiable, cada sucursal puede funcionar de forma aislada y luego sincronizar pedidos y envíos en cuanto se restablezca la conexión
Empresa que comercializa un motor de informes a varios clientes con entornos de desarrollo diferentes	Necesidad de exponer la misma funcionalidad a clientes con lenguajes variados, mantenimiento prolongado	Servicios web (REST o gRPC)	Facilita la publicación de un contrato uniforme, soporta la integración con clientes en lenguajes distintos, la documentación con OpenAPI o gRPC agiliza el soporte
Aplicación interna de colaboración documental con prioridad en la comunicación dentro de la misma red local	Todos los equipos están en la misma LAN, sin apertura a Internet, frecuencia alta de llamadas entre módulos	Objetos distribuidos (RMI, .NET Remoting) o middleware local	Simplifica la llamada a métodos remotos, se aprovecha la baja latencia de la red interna, no se requiere exponer interfaces REST de manera pública

Actividad 9

A continuación, se presentan una serie de afirmaciones relacionadas con el modelo de componentes. Cada afirmación tiene una palabra clave desordenada. Tu tarea es ordenar las letras para descubrir la respuesta correcta.

En un sistema basado en componentes, cada parte del software debe definir qué servicios ofrece y cómo se comunica con los demás módulos. Para ello, se establecen contratos conocidos como:

treainref → _____

Uno de los elementos fundamentales del modelo de componentes es la forma en que cada unidad se distribuye e instala en los distintos entornos. A este proceso se le llama:

oepylsdeg → _____

Cuando un componente ofrece funcionalidades como transacciones, seguridad y persistencia de datos, se dice que define su:

icvesesorpiadón → _____

Para que un componente pueda evolucionar sin afectar a los demás, debe mantener estable su:

zcntriao → _____

El acceso remoto de los componentes se logra mediante protocolos como HTTP, gRPC o colas de mensajería. Estas formas de comunicación se conocen como:

dlmoeos ed riciaagnte → _____

4. Comparación entre métodos de intercambio en las principales infraestructuras de componentes: OMG: CORBA, OMA, Java: JavaBeans, EJBs y Microsoft: COM, OLE/ActiveX, .NET.

Las **infraestructuras de componentes** han evolucionado a lo largo del tiempo, ofreciendo diferentes estrategias para el **intercambio de información** y la integración de módulos en aplicaciones distribuidas. En el **diseño de elementos software con tecnologías basadas en componentes**, es **importante** conocer las opciones históricas y las más actuales, ya que, según el entorno y los requisitos, pueden resultar más o menos adecuadas. A continuación, se expone una comparación entre algunos métodos de intercambio propuestos por OMG (CORBA y OMA), las soluciones de Java (JavaBeans y EJBs) y las ofrecidas por Microsoft (COM, OLE/ActiveX y .NET), destacando sus usos y evolución en los proyectos modernos.

En primer lugar, el **Object Management Group (OMG)** impulsó **CORBA** (Common Object Request Broker Architecture) como una manera de lograr que objetos escritos en distintos lenguajes y ejecutados en plataformas diferentes pudiesen comunicarse a través de una capa conocida como ORB (Object Request Broker). CORBA ofrecía un modelo de objetos distribuidos, un lenguaje de definición de interfaces (IDL) y la promesa de independencia respecto a sistemas operativos o lenguajes de programación. No obstante, con el tiempo, varios desarrolladores han optado por alternativas más ligeras o más sencillas de configurar, especialmente con la irrupción de las arquitecturas de microservicios. Pese a ello, aún se encuentran implementaciones de CORBA en sectores donde la confiabilidad y la interoperabilidad entre sistemas heterogéneos siguen siendo prioritarias. La **OMA** (Object Management Architecture) complementó esta visión al definir un conjunto de servicios y facilidades que podían extender CORBA, dotando al ecosistema de un enfoque más amplio para la gestión de objetos en entornos distribuidos.

En el mundo de **Java**, se han presentado distintas aproximaciones para la creación y el intercambio de componentes. **JavaBeans** surgió como un modelo para encapsular propiedades, eventos y métodos en objetos reutilizables que pudieran trabajar fácilmente en IDEs y entornos visuales de desarrollo. Se usaban con frecuencia para construir elementos de interfaz o lógicas sencillas que debían moverse de un lugar a otro dentro de una misma aplicación. Por otra parte, **Enterprise JavaBeans (EJBs)** elevaron la idea de un componente a un entorno más corporativo, centrándose en servicios de negocio y transacciones distribuidas. En los servidores de aplicaciones (por ejemplo, Jakarta EE), los EJBs se gestionan de manera que el programador se beneficia de funciones como transacciones, seguridad y concurrencia, sin tener que lidiar con la complejidad de implementarlas. Con la llegada de Spring y arquitecturas de microservicios, muchos desarrolladores prefieren soluciones más ligeras, aunque los EJBs todavía se usan en sistemas de gran escala que requieren la robustez del modelo clásico.

En el lado de **Microsoft**, la evolución ha seguido un trayecto notable. **COM** (Component Object Model) sirvió como base para crear componentes binarios que podían integrarse en distintas aplicaciones de Windows, incluyendo soporte para lenguajes variados. De COM derivaron tecnologías como **OLE (Object Linking and Embedding)** y **ActiveX**, usadas en su momento para insertar documentos, reproducir contenidos o interactuar con controles dentro de navegadores. Sin embargo, con la expansión de la web y la aparición de nuevas necesidades de seguridad y

escalabilidad, muchos de estos enfoques quedaron limitados al ámbito Windows o al software legacy. Con la llegada de la plataforma **.NET**, Microsoft facilitó el diseño de componentes y servicios más flexibles, apoyándose en lenguajes como C# o VB.NET y en un entorno de ejecución unificado (CLR). Hoy en día, la plataforma .NET se ha modernizado (incluyendo .NET 5, .NET 6 y .NET 7), ampliando su alcance a entornos Linux y macOS, y ofreciendo un modelo más abierto para el desarrollo de soluciones distribuidas y la integración de microservicios.

En el presente, la tendencia se inclina hacia arquitecturas más abiertas y basadas en protocolos o estándares web, como REST o gRPC, en parte porque facilitan la interoperabilidad con múltiples lenguajes y se ajustan mejor al mundo de los contenedores y la nube. Sin embargo, los modelos clásicos de **componentes** (CORBA, EJBs, COM, etc.) persisten en numerosos sistemas que aún valoran la estabilidad y la robustez de esas tecnologías, sobre todo en organizaciones donde la migración a alternativas más modernas conlleva riesgos o costos elevados. Esta coexistencia de enfoques demuestra que cada solución responde a un conjunto de problemas y prioridades diferentes.

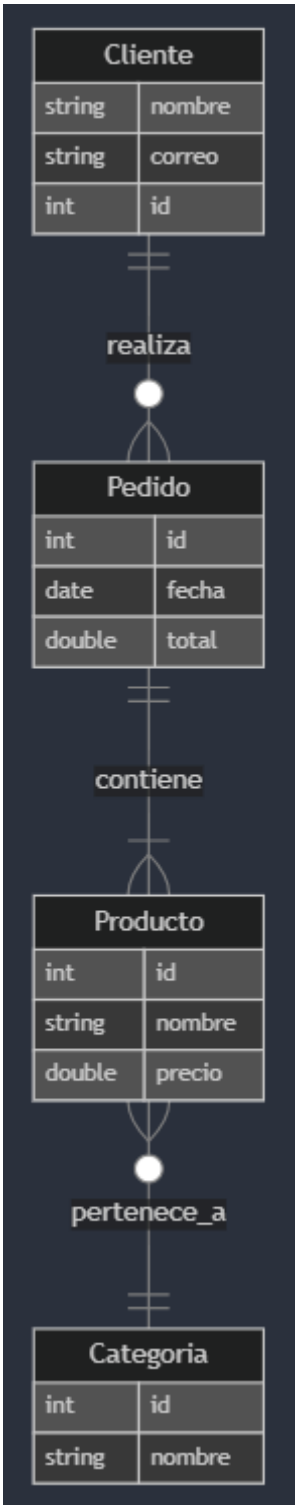
Así, cuando se aborda el **diseño de elementos software con tecnologías basadas en componentes**, conviene estudiar con detalle los requerimientos de interoperabilidad, la infraestructura disponible, los patrones de uso y las expectativas a largo plazo. CORBA puede ser idóneo si se necesita integrar sistemas muy dispares con énfasis en la neutralidad de lenguaje. Los EJBs funcionan bien en entornos donde la capacidad transaccional y la gestión de la seguridad a nivel empresarial están muy presentes. Mientras que .NET, con su ecosistema y la adopción de estándares más recientes, se ajusta perfectamente a proyectos que deseen abarcar plataformas distintas y aprovechar herramientas modernas de despliegue.

4.1. Diagramación y documentación de componentes.

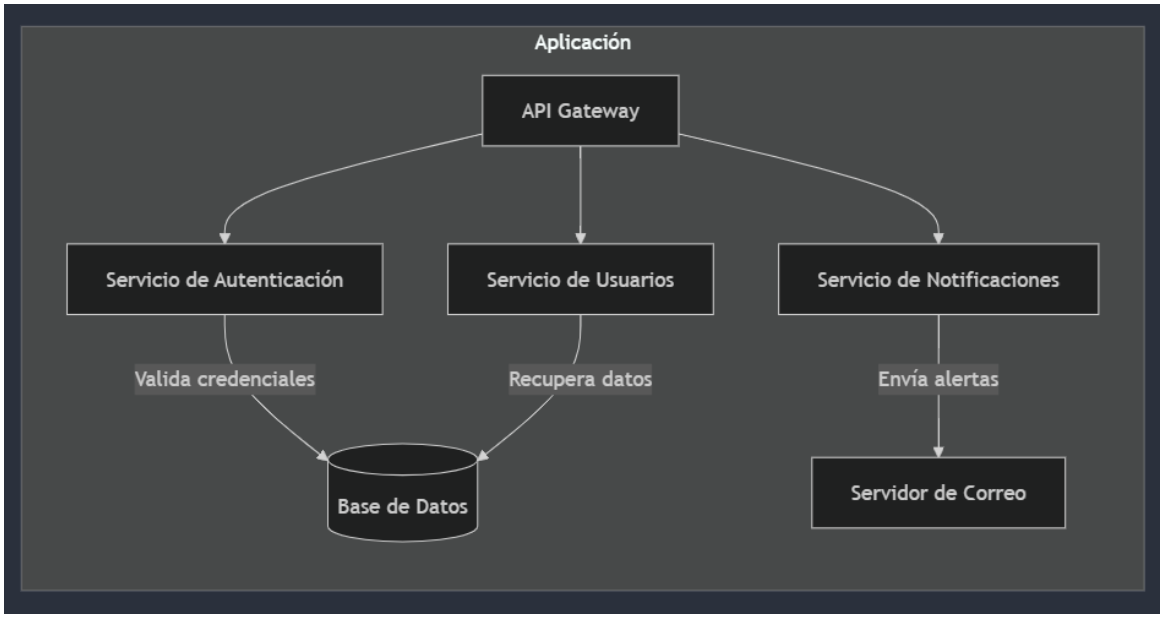
La **diagramación y documentación de componentes** permite ilustrar de manera gráfica y ordenada la forma en que cada parte del sistema se conecta con las demás, facilitando la comprensión y el mantenimiento a lo largo de su ciclo de vida. En el **diseño de elementos software con tecnologías basadas en componentes**, se recomiendan distintas representaciones que describan tanto la estructura informativa como el comportamiento dinámico de las aplicaciones. A través de estos recursos, los equipos de desarrollo y los interesados en el proyecto adquieren una visión clara de los módulos existentes, sus dependencias, cómo se despliegan y de qué manera interactúan para cubrir los requerimientos.

4.1.1. Modelo de información: diagramas conceptuales, diagramas de arquitectura de componentes y diagramas de despliegue.

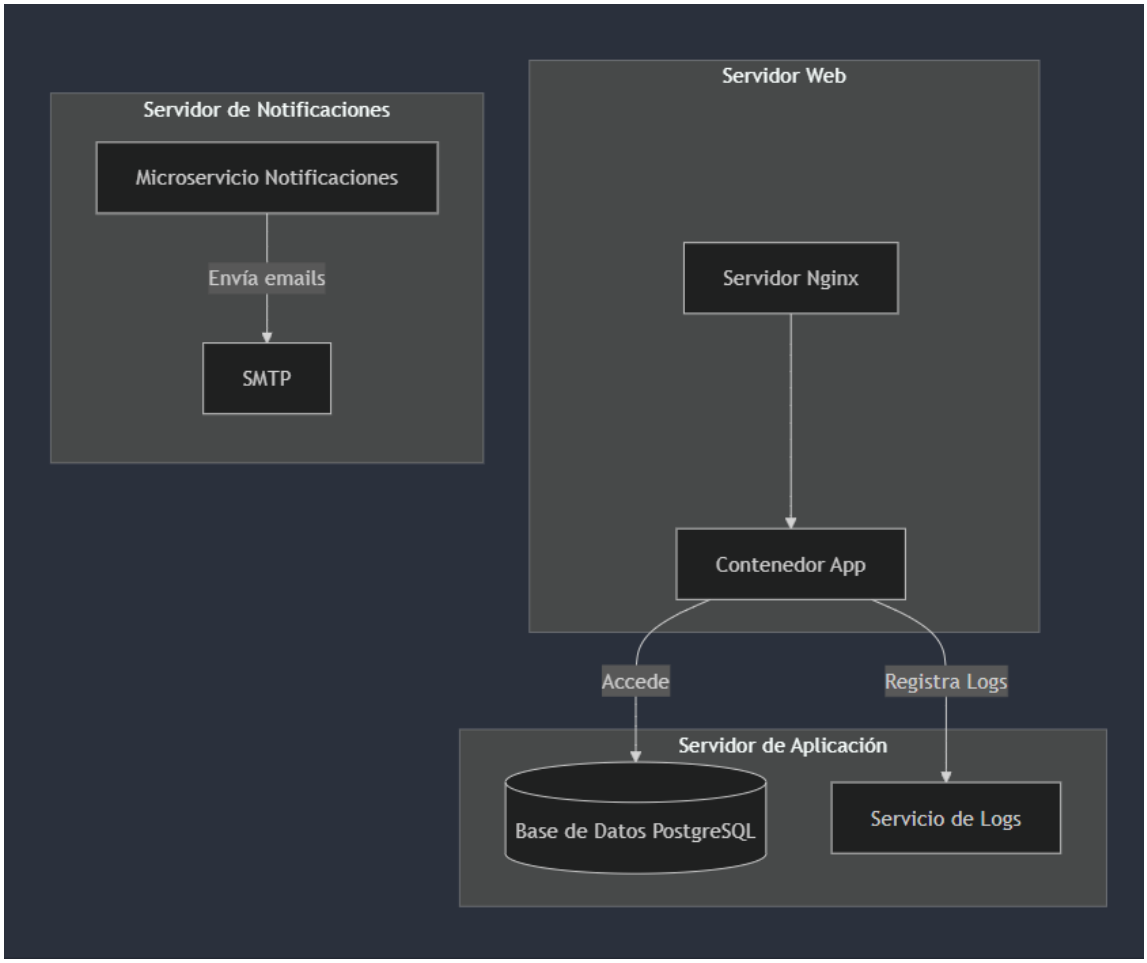
El **modelo de información** abarca, de forma **importante**, las perspectivas estáticas del sistema. Dentro de este modelo, los **diagramas conceptuales** presentan las entidades principales y las relaciones que comparten, resultando útiles para entender los elementos claves del dominio y su vínculo con las reglas de negocio. El diagrama conceptual que se expone a continuación representa entidades principales y sus relaciones en un sistema de comercio electrónico. Muestra cómo un Cliente realiza uno o varios Pedidos, cada pedido puede contener múltiples Productos, y cada producto pertenece a una Categoría:



Por otro lado, los **diagramas de arquitectura de componentes** detallan cómo se agrupan y se conectan los módulos, especificando qué interfaces exponen y de qué otros módulos dependen. Esta representación es muy valiosa cuando se trabaja en equipos grandes o se planifica la evolución de la solución, ya que permite identificar rápidamente si un componente tiene demasiadas dependencias o si puede reutilizarse en distintos entornos. El siguiente diagrama representa los módulos de una aplicación de gestión de usuarios con autenticación y notificaciones. Se visualiza cómo los servicios de autenticación, usuarios y notificaciones interactúan con la base de datos y el servidor de correo, mientras que una API Gateway centraliza el acceso:



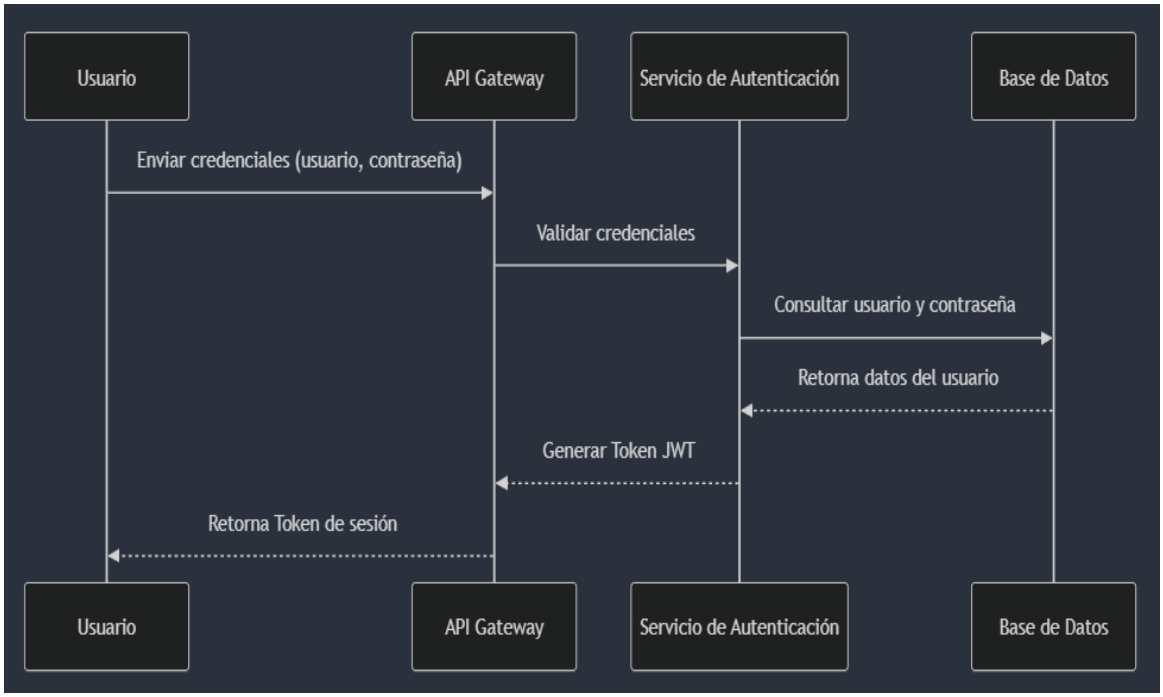
Además, los **diagramas de despliegue** muestran cómo los componentes se distribuyen en la infraestructura, indicando los servidores, contenedores o procesos donde se ejecutan. Esto ayuda a planificar la escalabilidad y la resiliencia, al mostrar de manera visual el flujo de comunicación entre nodos y la ubicación de cada módulo en diferentes niveles de la plataforma. A continuación, se representa la distribución de componentes en una infraestructura con contenedores Docker. Muestra cómo la aplicación se despliega en tres servidores, separando la funcionalidad en módulos independientes:



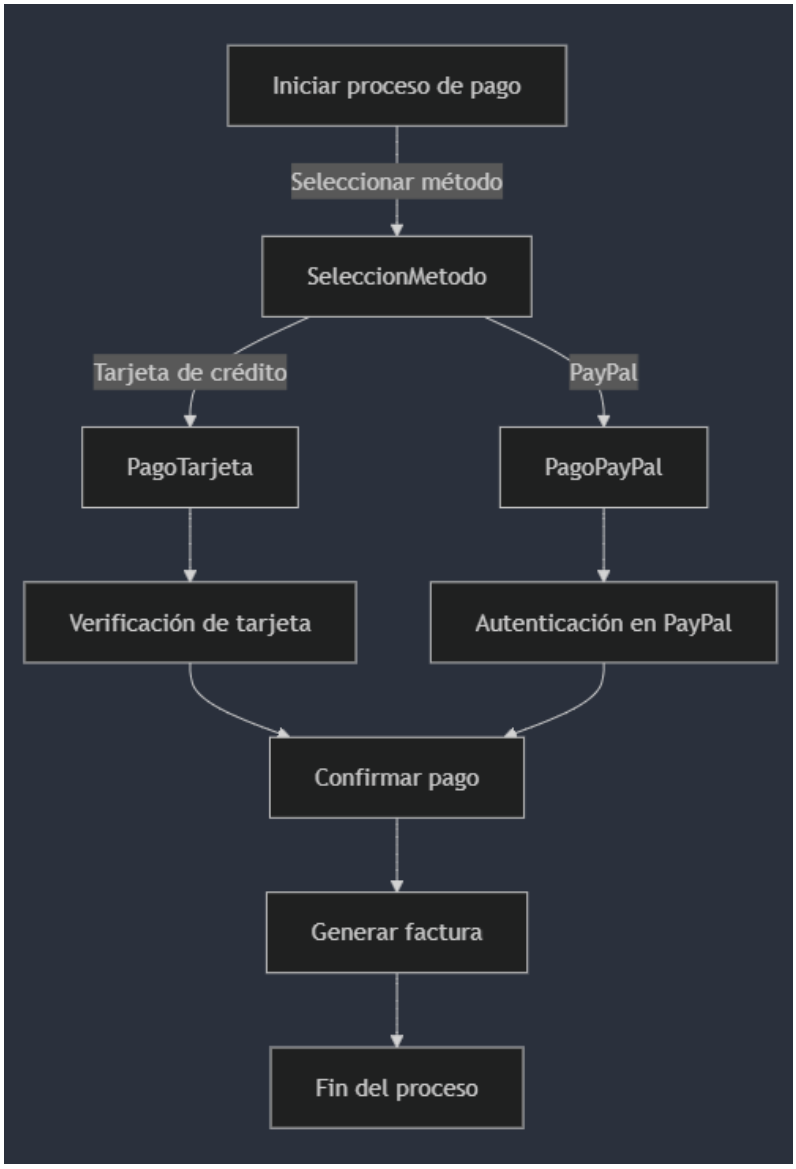
4.1.2. Modelo dinámico: diagramas de interacción y de actividad, diagramas de casos de uso y diagramas de estado.

El **modelo dinámico** describe los procesos y la forma en que los componentes se comportan en tiempo de ejecución. Entre las representaciones más **importantes** para este enfoque se encuentran los **diagramas de interacción**, que incluyen los de secuencia o colaboración.

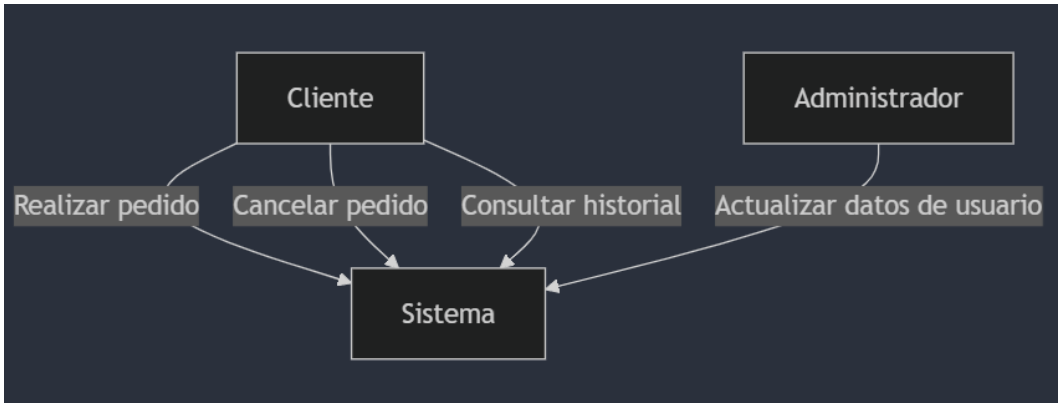
En un **diagrama de secuencia**, puede verse cómo cada componente recibe y envía mensajes, paso a paso, para llevar a cabo una funcionalidad. Esto aporta claridad acerca de quién inicia una operación, en qué orden se ejecutan las peticiones y cuál es la dependencia temporal entre módulos. El siguiente diagrama representa cómo un usuario inicia sesión en una aplicación, pasando por el servicio de autenticación y la base de datos:



Por su parte, los **diagramas de actividad** representan flujos de trabajo que involucran múltiples estados y decisiones, siendo ideales para describir algoritmos o procesos de negocio compuestos por pasos sucesivos. Por ejemplo, el siguiente diagrama representa el flujo de pago en un sistema de comercio electrónico:

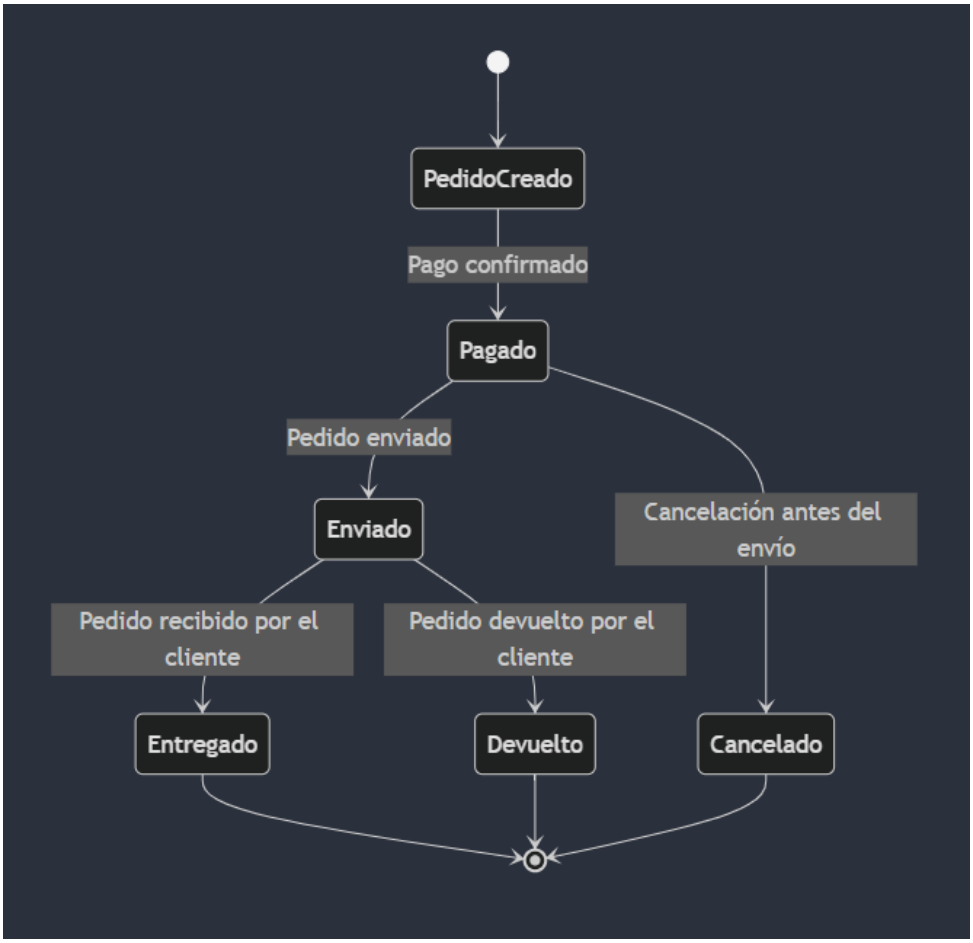


Para reflejar la relación del sistema con los usuarios o con otros actores, los **diagramas de casos de uso** resultan de gran ayuda, pues describen las acciones principales de cada actor y la funcionalidad que el sistema debe proveer sin entrar en detalles de implementación. El siguiente diagrama muestra las interacciones entre usuarios y el sistema en un portal de comercio electrónico:



Por su parte, los **diagramas de estado** son útiles para aquellos componentes que manejan ciclos de vida complejos, cambiando de un estado a otro según eventos o condiciones específicas. Esto cobra

relevancia cuando se requiere modelar transiciones y restricciones con precisión, como ocurre en sistemas de reserva de vuelos o de gestión de producción. Por ejemplo, el siguiente diagrama representa el ciclo de vida de un pedido en una tienda en línea:



5. Prueba de autoevaluación.

¿Cuál de las siguientes afirmaciones describe mejor un componente software en la orientación a componentes?

a) Es una unidad funcional autónoma con una interfaz bien definida.

b) Es un objeto que solo puede existir dentro de una aplicación específica.

c) Es un módulo de código que no requiere una interfaz para interactuar con otros módulos.

¿Cuál es una diferencia clave entre un componente y un objeto en la orientación a objetos?

a) Los componentes dependen directamente de la aplicación en la que se crean, mientras que los objetos son autónomos.

b) Los componentes pueden desplegarse y operar independientemente, mientras que los objetos suelen depender del entorno de ejecución.

c) Los objetos siempre tienen una interfaz pública, mientras que los componentes no necesitan interfaces definidas.

¿Cuál de las siguientes características define a una arquitectura basada en middleware?

a) No permite la comunicación entre distintos sistemas o tecnologías.

b) Facilita la interoperabilidad entre componentes a través de una capa intermedia.

c) Requiere que todos los componentes estén en el mismo entorno de ejecución.

¿Qué función cumplen las interfaces en el diseño de componentes?

a) Permiten la comunicación entre componentes asegurando consistencia y compatibilidad.

b) Evitan que los componentes puedan escalar en entornos distribuidos.

c) Hacen que los módulos dependan unos de otros sin importar su propósito.

¿Qué ventaja proporciona el escalado de componentes en una arquitectura basada en la nube?

a) Permite aumentar la capacidad de procesamiento sin modificar el resto del sistema.

b) Reduce la modularidad del sistema al centralizar los procesos.

c) Impide que diferentes instancias de un componente interactúen entre sí.

En la orientación a componentes, un _____ encapsula lógica de negocio y datos, permitiendo su reutilización en diferentes aplicaciones.

A diferencia de los objetos, los componentes pueden operar de manera _____, lo que facilita su integración en entornos distribuidos.

Un sistema basado en _____ permite que distintos componentes se comuniquen a través de una capa intermedia sin necesidad de estar directamente acoplados.

El versionado de _____ es fundamental para garantizar la compatibilidad con versiones previas y evitar interrupciones en los clientes.

Un componente sin estado (stateless) es más fácil de escalar horizontalmente porque no almacena _____ entre diferentes instancias.