Selección de componentes

Elegir los componentes adecuados para un sistema no es tan simple como descargar cualquier librería o herramienta de Internet. Cada software tiene requisitos específicos, y seleccionar las piezas correctas puede marcar la diferencia entre una aplicación fluida y escalable o un sistema lleno de problemas de compatibilidad y rendimiento. Son como piezas de un rompecabezas que hacen que una aplicación funcione de manera eficiente. No se trata solo de descargar un programa y usarlo, sino de asegurarse de que cada pieza encaje perfectamente con las demás. ¿Para qué sirven? Pues, básicamente, para facilitar el trabajo de los desarrolladores y mejorar el rendimiento de los sistemas sin tener que construirlo todo desde cero. Los componentes pueden ser comerciales (COTS), adaptables (MOTS) o de código abierto, cada uno con sus ventajas y limitaciones. Mientras que los comerciales suelen ofrecer soporte y estabilidad, los de código abierto permiten más personalización y flexibilidad.

No solo importa de dónde proviene un componente, sino también cómo encaja en el conjunto del sistema. La compatibilidad con entornos cloud, la facilidad de integración, el rendimiento y la seguridad son factores clave en esta decisión. Hoy en día, herramientas como Infrastructure as Code (IaC) facilitan la parametrización y configuración de componentes, mientras que la automatización ayuda a garantizar que las actualizaciones sean seguras y eficientes. En resumen, una buena selección de componentes puede hacer que un sistema sea más robusto y preparado para evolucionar con el tiempo.

Por ejemplo, imagina que necesitas que tu aplicación permita a los usuarios iniciar sesión con su cuenta de Google o Facebook. En lugar de programar todo el sistema de autenticación desde el principio, puedes usar un componente de seguridad, como OAuth o OpenID Connect, que ya tiene todo listo para verificar identidades y gestionar accesos. Así, tu aplicación se vuelve más segura sin que tengas que reinventar la rueda.

Otra función importante de los componentes es hacer que diferentes partes del sistema se comuniquen entre sí. Supongamos que tienes una tienda online y necesitas conectar el carrito de compras con un sistema de pagos como PayPal o Stripe. En lugar de programar toda la conexión manualmente, usas un componente que ya está diseñado para manejar estas transacciones de manera segura y rápida. Eso significa menos errores y una mejor experiencia para los usuarios.

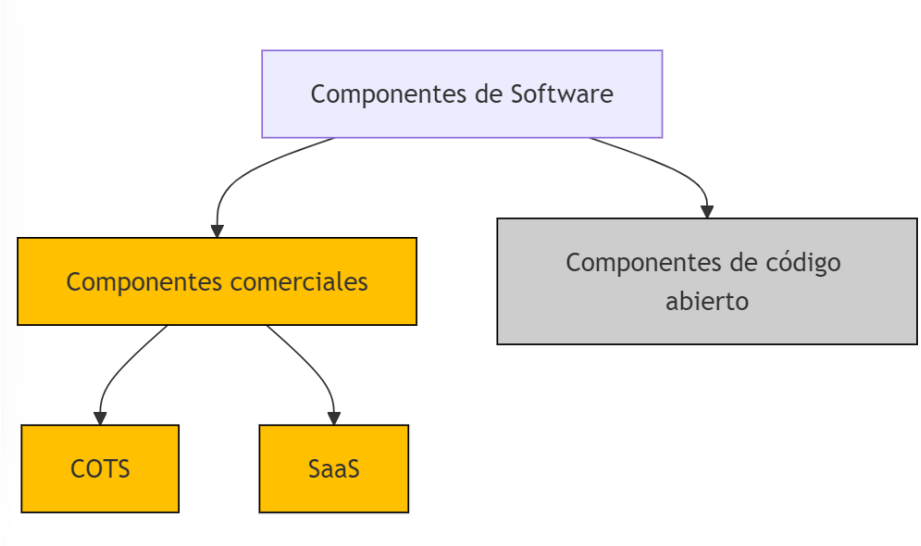
Además, los componentes ayudan a mantener un sistema estable y escalable. Si tu aplicación empieza a recibir más usuarios de los esperados, algunos componentes, como balanceadores de carga o servidores en la nube, permiten que la plataforma se adapte sin colapsar. Es como tener un restaurante con pocas mesas y, de repente, recibir una gran cantidad de clientes: si tienes un sistema que puede "agregar más mesas" automáticamente, podrás atender a todos sin problemas.

Los componentes comerciales, como software de gestión empresarial (ERP) o bases de datos de pago, que ofrecen soluciones listas para usar con soporte técnico. Son ideales para empresas que necesitan estabilidad y garantía de funcionamiento, aunque pueden tener menos flexibilidad que las opciones de código abierto, que permiten modificaciones según las necesidades.

Además, los componentes ahorran tiempo y dinero. Si un equipo de desarrollo tuviera que programar cada detalle desde cero, los proyectos tardarían mucho más y serían mucho más costesos. Al reutilizar componentes bien diseñados, se puede lanzar un producto más rápido y con menos errores, lo que beneficia tanto a los programadores como a los usuarios.

1. Tipos de componentes.

Los componentes de software pueden clasificarse en distintas categorías según su origen, su disponibilidad y su forma de distribución. Algunos son productos comerciales que se adquieren con licencias, mientras que otros son de código abierto y pueden utilizarse, modificarse y redistribuirse libremente.



La elección del tipo de componente depende de varios factores, como el presupuesto disponible, la flexibilidad que se necesita, el nivel de soporte técnico requerido y la compatibilidad con la infraestructura existente. Además, en entornos modernos, donde muchas aplicaciones se despliegan en la nube o en entornos híbridos, es importante considerar cómo se integran estos componentes con plataformas cloud y soluciones on-premise.

1.1. Componentes comerciales (COTS y SaaS).

Los **componentes comerciales** son soluciones de software desarrolladas y vendidas por empresas especializadas. Se diseñan para resolver problemas específicos sin necesidad de desarrollos a medida, lo que permite a las organizaciones reducir costes y tiempos de implementación. Dentro de esta categoría encontramos dos tipos principales:

COTS (Commercial Off-The-Shelf)

Son productos listos para usar, diseñados para cubrir necesidades estándar en diversas industrias. Ejemplos de COTS incluyen herramientas de gestión empresarial como **SAP ERP**, bases de datos como **Microsoft SQL Server** y plataformas de desarrollo como **OutSystems**. La ventaja de estos productos es que están ampliamente probados y cuentan con soporte técnico, aunque pueden tener limitaciones en cuanto a personalización y costes de licencias.

¿Cuándo elegir COTS (Commercial Off-The-Shelf)?

- **Empresas con requisitos de personalización estrictos:** Si una organización necesita adaptar el software a su estructura interna, los productos COTS pueden ser una opción viable. Por ejemplo, un banco que requiere un sistema de bases de datos con seguridad avanzada y compatibilidad con normativas financieras puede optar por un sistema como **Microsoft SQL Server** o **SAP ERP**.

- **Sistemas con regulación estricta:** En sectores como el sanitario, financiero o gubernamental, donde se exige el cumplimiento de normativas de seguridad y privacidad (como GDPR o ISO 27001), los COTS son preferidos debido a su estabilidad y certificaciones.
- **Empresas que quieren evitar problemas de compatibilidad:** Muchas organizaciones optan por COTS porque garantizan compatibilidad con otros productos comerciales. Por ejemplo, una empresa que usa **Windows Server** y Microsoft 365 podría elegir **Microsoft SQL Server** en lugar de una base de datos de código abierto para facilitar la integración.

SaaS (Software as a Service)

Es un modelo en el que las aplicaciones se ejecutan en la nube y los usuarios acceden a ellas a través de Internet. Ejemplos conocidos incluyen **Google Workspace, Microsoft 365, Salesforce y AWS Lambda**. Este modelo ofrece actualizaciones automáticas, escalabilidad y facilidad de acceso desde cualquier lugar, pero depende de la conectividad a Internet y puede implicar costes recurrentes.

¿Cuándo elegir SaaS (Software as a Service)?

- **Startups y empresas con equipos remotos:** Las empresas que necesitan herramientas de colaboración y gestión en la nube pueden beneficiarse de soluciones SaaS como **Google Workspace** o **Microsoft 365**. Al no requerir servidores propios, se reducen los costes de infraestructura.
- **Negocios con necesidades dinámicas:** Empresas que requieren escalabilidad sin preocuparse por mantenimiento pueden elegir SaaS, ya que estas plataformas se ajustan automáticamente a la demanda. Un **CRM como Salesforce** permite a las empresas gestionar clientes sin preocuparse por el crecimiento del sistema.
- **Empresas con presupuesto reducido:** SaaS permite pagar por suscripción mensual o anual sin necesidad de grandes inversiones iniciales. Para una pyme que busca una solución de almacenamiento sin gastar en servidores, **Dropbox Business** o **Google Drive** pueden ser opciones ideales.

En general, los componentes comerciales son una opción atractiva para empresas que buscan soluciones estables y con soporte garantizado. Sin embargo, pueden no ser la mejor alternativa para desarrollos que requieren un alto grado de personalización o para organizaciones con restricciones presupuestarias.

Actividad 4

Las empresas suelen enfrentarse a la decisión de utilizar soluciones comerciales para sus necesidades tecnológicas. En esta actividad, analizarás las diferencias entre los componentes COTS (Commercial Off-The-Shelf) y SaaS (Software as a Service) y reflexionarás sobre sus ventajas y limitaciones en distintos escenarios.

Parte 1:

Relaciona cada característica con el tipo de componente comercial al que pertenece (COTS o SaaS):

- Producto listo para usar que se instala en servidores locales.
- Se accede a través de un navegador sin necesidad de instalación.
- Puede requerir una inversión inicial alta en licencias.

- Actualizaciones automáticas gestionadas por el proveedor.
- Mayor posibilidad de personalización, pero con mayor complejidad en la implementación.
- Dependencia de conexión a Internet para su uso.
- Ejemplo: SAP ERP, Microsoft SQL Server.
- Ejemplo: Google Workspace, Salesforce.

Parte 2:

Imagina que trabajas como consultor/a tecnológico y debes asesorar a una empresa que busca una solución de software para mejorar su gestión interna. Analiza cada uno de los siguientes casos y responde con cuál opción recomendarías (COTS o SaaS) y por qué.

Caso 1: Una empresa manufacturera con estrictos requisitos de personalización y normativas de seguridad que necesita un software de gestión empresarial.

Caso 2: Una startup con un equipo remoto que requiere herramientas colaborativas para comunicación y almacenamiento de documentos sin preocuparse por infraestructura propia.

Caso 3: Un banco que busca un sistema de base de datos robusto, altamente seguro y con soporte técnico dedicado para manejar grandes volúmenes de transacciones.

1.2. Componentes de código abierto y frameworks modernos (Spring Boot, Quarkus, Express.js, NestJS).

El **software de código abierto** ha ganado una gran popularidad en los últimos años, gracias a su flexibilidad, la colaboración de la comunidad y la ausencia de costes de licenciamiento. A diferencia de los componentes comerciales, el código abierto permite a los desarrolladores inspeccionar, modificar y mejorar el software según sus necesidades.

En el ámbito del desarrollo de aplicaciones, muchos frameworks modernos de código abierto facilitan la creación de sistemas robustos y escalables. Algunos ejemplos destacados son:

- **Spring Boot:** Un framework basado en Java que simplifica la creación de aplicaciones empresariales, con un enfoque modular y una gran integración con bases de datos y servicios cloud.
- **Quarkus:** Diseñado para entornos cloud y contenedores, Quarkus es una opción eficiente para aplicaciones Java con tiempos de arranque rápidos y bajo consumo de memoria.
- **Express.js:** Un framework ligero para Node.js que facilita el desarrollo de aplicaciones web y APIs REST. Es ampliamente utilizado por su simplicidad y su gran compatibilidad con otras herramientas.
- **NestJS:** Un framework moderno para Node.js que utiliza TypeScript y sigue una arquitectura modular inspirada en Angular. Es ideal para el desarrollo de aplicaciones escalables y bien estructuradas.

¿Cuándo elegir software de código abierto?

- **Empresas con equipos de desarrollo avanzados:** Si una empresa cuenta con un equipo técnico capacitado, el código abierto permite adaptar soluciones sin pagar licencias costesas. Por ejemplo, **PostgreSQL** es una alternativa a bases de datos comerciales con alto rendimiento y sin costes de licenciamiento.

- **Proyectos de startups con presupuesto limitado:** Una startup que quiere construir su plataforma sin costes de software puede usar frameworks como **Express.js** para backend o **Vue.js** para frontend en lugar de herramientas comerciales.
- **Sistemas que necesitan compatibilidad y personalización:** Si una empresa busca adaptar su software sin restricciones de un proveedor, frameworks como **Spring Boot** en Java o **NestJS** en Node.js permiten construir aplicaciones personalizadas y modulares.

¿Cuándo no es recomendable el software de código abierto?

- **Empresas que requieren soporte técnico garantizado:** A diferencia del software comercial, los proyectos de código abierto dependen de la comunidad, lo que puede generar tiempos de respuesta más lentos ante problemas críticos.
- **Negocios con poca experiencia en seguridad:** Muchas vulnerabilidades surgen por mala gestión de software de código abierto. Empresas sin experiencia en ciberseguridad pueden beneficiarse más de soluciones comerciales con soporte y auditorías de seguridad regulares.

1.3. Ventajas e inconvenientes en entornos cloud y on-premise.

La infraestructura donde se ejecutan los componentes también influye en su selección. No todas las empresas tienen las mismas necesidades, y dependiendo de su modelo de negocio, seguridad y escalabilidad, pueden optar por soluciones locales (on-premise) o en la nube (cloud). Mientras que algunas organizaciones prefieren mantener sus datos y sistemas dentro de su propia infraestructura, otras aprovechan la flexibilidad y escalabilidad de la nube para optimizar recursos y reducir costes.

Las organizaciones con **requisitos de privacidad estrictos** suelen preferir entornos on-premise. Por ejemplo, empresas del sector financiero o gubernamental que manejan datos sensibles pueden optar por soluciones locales para evitar exponer información en la nube y garantizar un mayor control sobre la seguridad y el cumplimiento normativo.

Los **negocios con infraestructura propia** también pueden beneficiarse del modelo on-premise. Si una empresa ya ha invertido en servidores y equipos IT, puede ser más económico seguir utilizando esos recursos en lugar de migrar a la nube y asumir costes adicionales de suscripción o almacenamiento.

Por otro lado, cuando se trata de **aplicaciones con baja variabilidad en la demanda**, mantener servidores locales puede ser una opción más rentable. Si el uso del sistema es estable y predecible, pagar por servidores en la nube que escalen automáticamente puede no ser necesario y generar costes innecesarios.

Las **empresas con necesidades de escalabilidad** encuentran en la nube una solución ideal. Startups o aplicaciones con tráfico variable pueden aprovechar los servicios cloud para escalar dinámicamente sin la necesidad de comprar y mantener nuevos servidores físicos. De esta manera, pueden adaptar su infraestructura a la demanda sin grandes inversiones iniciales.

Los **equipos distribuidos** también se benefician de la nube. Si una empresa tiene empleados en distintas ubicaciones geográficas, usar soluciones como **AWS Lambda** o **Google Cloud Run** facilita el acceso a las aplicaciones sin depender de una oficina central o infraestructura local.

Además, la nube permite **reducir costes de mantenimiento**. En este modelo, las actualizaciones de software, la gestión de seguridad y la administración de servidores son manejadas por el proveedor, lo que ahorra recursos y tiempo en la administración interna de la infraestructura.

Con la adopción de prácticas **DevOps**, la selección de componentes debe considerar su compatibilidad con herramientas de automatización y despliegue continuo. La integración de estos elementos permite optimizar procesos, reducir errores humanos y mejorar la eficiencia operativa en entornos modernos.

Las **empresas que necesitan despliegues rápidos y frecuentes** deben optar por componentes compatibles con herramientas de automatización. Tecnologías como **Docker, Kubernetes y Terraform** permiten gestionar infraestructuras de manera programática, asegurando que cada actualización se implemente sin interrupciones y de forma eficiente.

En el caso de **organizaciones con arquitecturas basadas en microservicios**, es fundamental elegir componentes que faciliten la gestión de múltiples servicios en contenedores. Herramientas como **Helm Charts** y **Kubernetes Operators** permiten administrar aplicaciones complejas, asegurando escalabilidad y facilidad de mantenimiento.

Para los **equipos que buscan reducir errores manuales**, la infraestructura como código (**IaC**) es una gran ventaja. Soluciones como **Terraform o Ansible** permiten definir configuraciones de forma declarativa, asegurando que todas las instancias del sistema se desplieguen de manera uniforme y sin inconsistencias.

Como hemos visto, cada tipo de componente tiene ventajas y desventajas dependiendo del entorno en el que se despliegue. Hoy en día, muchas organizaciones utilizan infraestructuras híbridas, combinando soluciones en la nube con sistemas locales (on-premise). Es importante entender cómo los distintos tipos de componentes se adaptan a estos entornos.

Entornos cloud

En un entorno en la nube, los componentes pueden ejecutarse en plataformas como **AWS, Microsoft Azure o Google Cloud**. Esto ofrece varias ventajas:

- ✓ **Escalabilidad automática:** Los servicios en la nube permiten ajustar los recursos según la demanda.
- ✓ **Menor gestión de infraestructura:** No es necesario preocuparse por la administración de servidores físicos.
- ✓ **Acceso desde cualquier ubicación:** Los equipos pueden trabajar de forma remota sin problemas.

Sin embargo, también existen desventajas:

- × **Dependencia del proveedor:** Si se elige un componente muy ligado a un proveedor cloud, puede ser difícil migrar a otra plataforma en el futuro.
- × **Costes variables:** Aunque la nube permite pagar solo por el uso real, los costes pueden aumentar rápidamente si no se gestionan bien los recursos.

Entornos on-premise

En un entorno on-premise, los componentes se ejecutan en servidores locales dentro de la empresa. Este modelo tiene sus propios beneficios:

- ✓ **Mayor control sobre los datos y la seguridad.**
- ✓ **Personalización total de la infraestructura.**
- ✓ **Menos dependencia de proveedores externos.**

Pero también presenta desventajas:

- **Altos costes iniciales:** Se requiere una inversión en hardware y en equipos de administración.
- **Menor flexibilidad:** Adaptar la infraestructura a nuevas necesidades puede ser más complicado.

En muchos casos, las empresas optan por una solución híbrida, utilizando componentes en la nube para aplicaciones de alto tráfico y manteniendo sistemas críticos en entornos locales.

2. Métodos de personalización de componentes.

Cuando se despliegan componentes de software, rara vez se utilizan exactamente como vienen por defecto. En la mayoría de los casos, es necesario ajustarlos para que se adapten a los requisitos específicos de cada entorno y aplicación. La personalización de componentes permite mejorar su integración, optimizar el rendimiento y facilitar la administración a lo largo del tiempo.

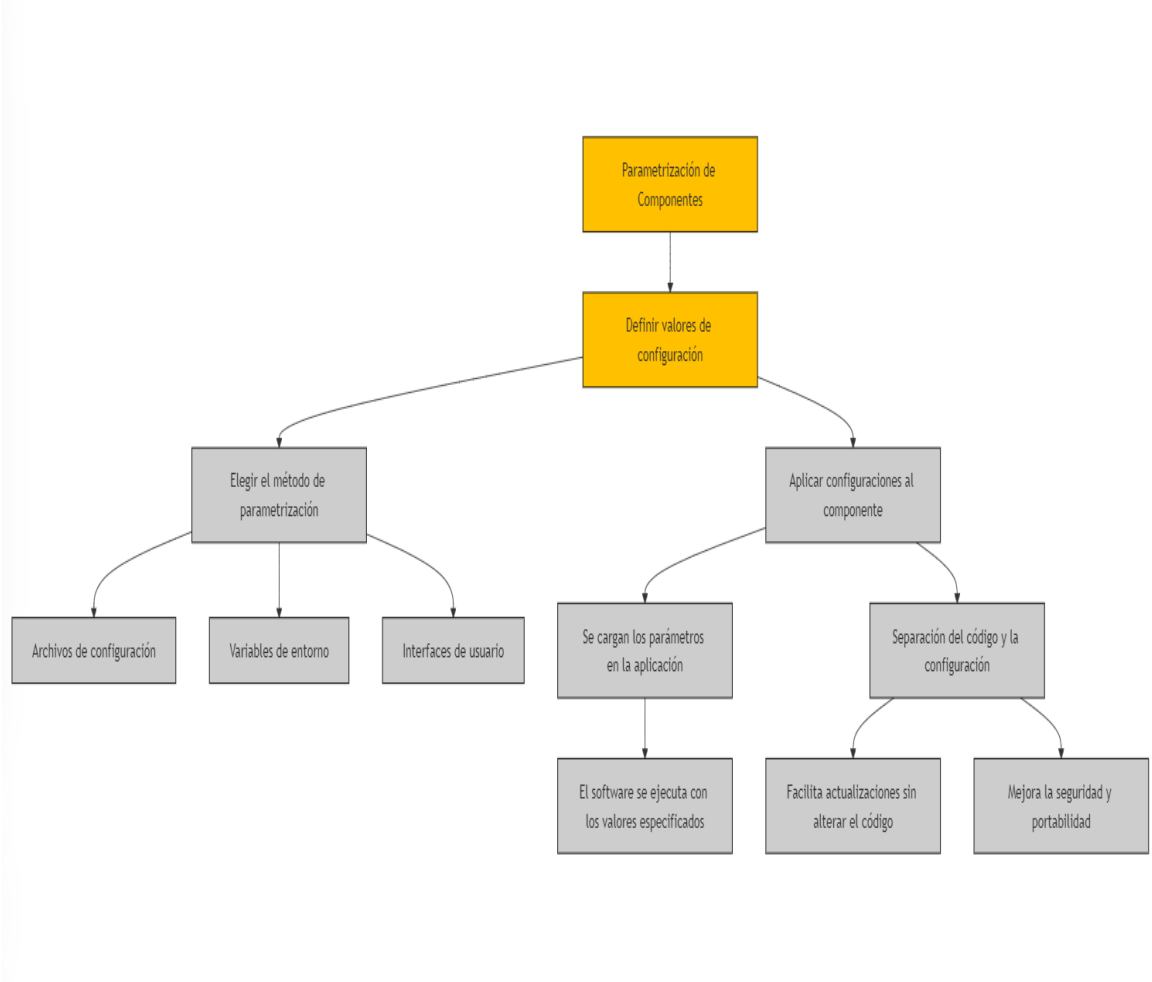
Existen varias formas de personalizar los componentes, dependiendo del nivel de control que se necesite. Algunos métodos permiten realizar cambios básicos sin alterar el código fuente, mientras que otros ofrecen mayor flexibilidad mediante configuraciones avanzadas. A continuación, veremos tres enfoques ampliamente utilizados en la industria: la **parametrización**, el **uso de extensiones o plugins** y las **configuraciones declarativas mediante Infrastructure as Code (IaC)**.

2.1. Parametrización.

La parametrización es una de las formas más sencillas y efectivas de personalizar un componente de software. En lugar de modificar directamente el código fuente, se configuran variables y ajustes que determinan su comportamiento sin alterar su estructura interna. Es decir, cuando hablamos de parametrización de software, nos referimos a la técnica de configurar un programa sin tocar su código fuente. Es como cuando ajustamos el brillo y el volumen de un televisor: no cambiamos el hardware, solo modificamos la forma en que funciona según nuestras necesidades.

Este método se basa en proporcionar valores de configuración a través de archivos, variables de entorno o interfaces de usuario. Por ejemplo, una base de datos como **PostgreSQL** permite ajustar parámetros como el tamaño del caché, la cantidad de conexiones concurrentes o la política de almacenamiento a través de su archivo postgresql.conf. De manera similar, en aplicaciones web, un archivo config.json puede definir el puerto de ejecución, la URL de la base de datos y otras opciones de personalización.

La parametrización es especialmente útil porque evita la necesidad de modificar el código fuente del software. Esto permite actualizar versiones sin perder configuraciones personalizadas y facilita la administración en entornos donde se requieren múltiples instancias del mismo componente, cada una con ajustes específicos. Además, al mantener la configuración separada del código, se mejora la seguridad y la portabilidad del software.



Esquema del proceso de parametrización.

El primer paso es identificar qué aspectos del software se pueden personalizar sin modificar el código. Imagina que tenemos una **aplicación web de reservas de hoteles**. Algunos parámetros que podríamos querer ajustar serían:

- **Número máximo de reservas por usuario** (ejemplo: 5 reservas simultáneas).
- **Hora de cierre del sistema** (ejemplo: 23:00 h).
- **Tipo de moneda utilizada** (ejemplo: euros o dólares).
- **Límite de habitaciones disponibles por hotel** (ejemplo: 50 habitaciones).

Estos valores deben estar bien definidos para que el sistema pueda funcionar correctamente sin necesidad de reprogramarlo cada vez que queramos cambiar una configuración.

Ahora que sabemos qué configuraciones queremos personalizar, necesitamos decidir **cómo y dónde almacenarlas**. Existen varias formas de hacerlo:

1. **A través de archivos de configuración:**

   Se crean archivos como config.json o settings.yaml, donde se guardan los valores de configuración. Por ejemplo:

```
{
  "max_reservas_por_usuario": 5,
  "hora_cierre": "23:00",
  "moneda": "EUR",
  "limite_habitaciones": 50
}
```

→ Aquí, el sistema leerá estos datos cada vez que arranque y usará esos valores. Si queremos cambiar la moneda de euros a dólares, solo editamos "moneda": "USD", guardamos el archivo y listo.

2. **Uso de variables de entorno:**

En servidores y contenedores como **Docker**, se configuran parámetros sin tocar archivos. Ejemplo real usando variables de entorno en un servidor Linux:

```
export MAX_RESERVAS=5
export HORA_CIERRE=23:00
export MONEDA=EUR
export LIMITE_HABITACIONES=50
```

→ Si el hotel cambia su política y ahora quiere permitir hasta 10 reservas por usuario, simplemente actualizamos MAX_RESERVAS=10 sin tocar el código.

3. **Interfaces de usuario (panel de configuración):**

- En aplicaciones avanzadas, se pueden ofrecer **paneles de configuración** donde los administradores cambian los parámetros sin necesidad de editar archivos o usar comandos. Por ejemplo, en una página web de administración, el gerente del hotel podría ver opciones como estas:

```
[ ] Número máximo de reservas: [ 5 ]
[ ] Hora de cierre: [ 23:00 ]
[ ] Moneda: [ EUR / USD / GBP ]
[ ] Límite de habitaciones: [ 50 ]
[ Guardar cambios ]
```

→ Al hacer clic en "Guardar cambios", el sistema actualiza la configuración y la aplica en tiempo real.

Una vez que hemos definido los valores y elegido el método de parametrización, llega el momento de hacer que el software **use esas configuraciones**.

Si usamos **archivos de configuración**, el software los leerá en cada arranque. Si usamos **variables de entorno**, se cargarán automáticamente cuando el sistema inicie. Si usamos **una interfaz de usuario**, los cambios se guardarán en una base de datos y se aplicarán dinámicamente:

```
import json


# Cargar configuración desde un archivo JSON
```

```
with open("config.json") as config_file:
    config = json.load(config_file)
# Usar la configuración en la aplicación
print(f"Máximo de reservas por usuario: {config['max_reservas_por_usuario']}")
print(f"La moneda seleccionada es: {config['moneda']}")
```

Si cambiamos el archivo config.json para aumentar el límite de reservas, la próxima vez que iniciemos el programa, el nuevo valor se aplicará automáticamente **sin necesidad de modificar el código fuente**.

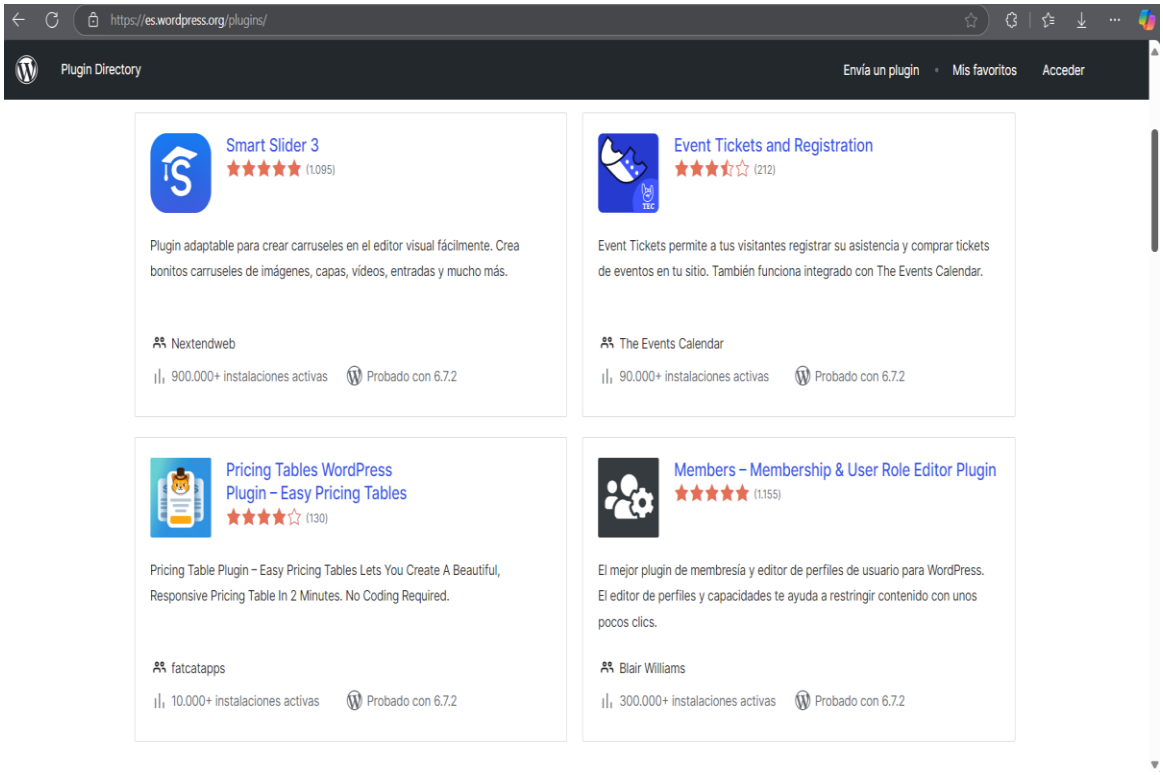
¿Por qué es tan importante parametrizar en lugar de meter los valores directamente en el código?

- Si queremos cambiar la moneda o los límites de reservas, no tenemos que editar el código del programa, solo los valores de configuración.
- No almacenamos contraseñas ni información sensible dentro del código, sino en archivos protegidos o en variables de entorno.
- Podemos usar el mismo código para distintos hoteles cambiando solo la configuración.

2.2. Uso de extensiones (plugins).

En muchos casos, la parametrización no es suficiente para adaptar un componente a todas las necesidades de un sistema. Para estos escenarios, las extensiones o **plugins** permiten agregar nuevas funcionalidades sin modificar el software original.

Un ejemplo muy común de este enfoque es el uso de **plugins en navegadores web**, que añaden funcionalidades extra sin cambiar el núcleo del programa. En el ámbito del desarrollo de software, plataformas como **WordPress** permiten ampliar sus capacidades mediante plugins, agregando desde sistemas de seguridad hasta herramientas de optimización de rendimiento.



Los servidores de aplicaciones y bases de datos también utilizan este modelo. En **NGINX** o **Apache**, los módulos y extensiones permiten habilitar funcionalidades como balanceo de carga, autenticación y compresión de datos sin modificar el código base. En el caso de **PostgreSQL**, existen extensiones como **PostGIS**, que añade soporte para datos geoespaciales.

El uso de plugins facilita la personalización y mejora la modularidad de los sistemas, permitiendo que cada instancia tenga solo las funcionalidades necesarias sin sobrecargar el software con características que no se utilizarán. Sin embargo, es importante gestionar bien las dependencias y asegurarse de que las extensiones sean compatibles con las versiones del software base, para evitar conflictos o problemas de seguridad.

2.3. Configuraciones declarativas con Infrastructure as Code (IaC), Ansible, Terraform.

En entornos modernos, especialmente en la nube y en infraestructuras escalables, la personalización de componentes no se realiza manualmente en cada servidor, sino que se define mediante código en archivos de configuración. Este enfoque se conoce como **Infrastructure as Code (IaC)** y permite describir toda la infraestructura y su configuración en archivos de texto legibles y reutilizables.

Herramientas como **Terraform, Ansible y CloudFormation** permiten definir, desplegar y gestionar infraestructuras de manera automatizada. En lugar de configurar servidores y aplicaciones manualmente, se crean archivos en los que se especifican los recursos necesarios y sus ajustes.

Por ejemplo, con **Terraform**, un archivo de configuración puede definir cuántas máquinas virtuales se necesitan en AWS, su tamaño, qué software deben tener instalado y cómo deben interactuar entre sí. Este archivo puede replicarse fácilmente en distintos entornos sin errores humanos.

Con **Ansible**, en lugar de conectarse a cada servidor y configurar manualmente los componentes, se pueden crear "playbooks" que describen en YAML qué paquetes instalar, qué archivos modificar y qué servicios activar. Esto es especialmente útil en despliegues masivos, donde cientos de servidores deben tener la misma configuración.

El uso de configuraciones declarativas tiene varias ventajas. Primero, permite mantener la infraestructura bajo control y evitar configuraciones inconsistentes entre servidores. Segundo, mejora la trazabilidad y facilita la recuperación ante fallos, ya que los cambios en la infraestructura quedan registrados en archivos de código. Finalmente, hace posible la automatización completa del despliegue, reduciendo tiempos de configuración y errores manuales.

Actividad 5

Imagina que trabajas en una empresa que debe desarrollar una nueva aplicación web y tienes la opción de elegir entre frameworks de código abierto como Spring Boot, Quarkus, Express.js o NestJS.

Investiga brevemente uno de estos frameworks y responde:

¿Qué ventajas ofrece en términos de rendimiento, escalabilidad y facilidad de uso?

¿Cuáles podrían ser sus desventajas o desafíos al implementarlo?

Tu empresa está considerando si utilizar un framework de código abierto o una solución comercial con soporte especializado. Reflexiona y responde:

¿Crees que el código abierto es siempre la mejor opción?

¿En qué casos una empresa podría preferir una solución comercial en lugar de un framework de código abierto?

3. Criterios de selección de componentes reutilizables.

Cuando se elige un componente para integrarlo en una aplicación o infraestructura, es importante considerar una serie de criterios que aseguren su funcionalidad, estabilidad y facilidad de mantenimiento. No se trata solo de encontrar una herramienta que cumpla con los requisitos técnicos, sino de asegurarse de que se adapte bien al ecosistema donde se desplegará, que sea segura y que su rendimiento sea adecuado a la carga de trabajo esperada.

Los componentes reutilizables pueden ser bibliotecas, frameworks, módulos de software o servicios en la nube que se integran en diferentes proyectos. Elegirlos correctamente puede ahorrar tiempo y recursos, facilitando la evolución y escalabilidad del sistema. Para tomar una decisión informada, es útil evaluar ciertos aspectos clave como la adaptabilidad, la auditabilidad, la estandarización y otros factores que influyen en su comportamiento en producción.

3.1. Adaptabilidad.

Un buen componente debe poder ajustarse a distintos entornos sin requerir grandes modificaciones. La capacidad de adaptación es clave en proyectos que pueden cambiar con el tiempo, ya sea por nuevas necesidades del negocio o por actualizaciones tecnológicas.

Por ejemplo, un framework de desarrollo debe permitir la personalización de configuraciones para ajustarse a distintos tipos de aplicaciones. Si un equipo usa **Spring Boot** para una API interna y más adelante necesita exponerla a clientes externos, la capacidad del framework para manejar distintos niveles de seguridad y protocolos de comunicación facilitará la transición sin necesidad de cambiar de herramienta.

También es importante evaluar si el componente puede integrarse con otras tecnologías ya existentes en el ecosistema de la empresa. Por ejemplo, si se trabaja con bases de datos relacionales, es recomendable elegir bibliotecas de acceso a datos que soporten distintos motores como **PostgreSQL, MySQL y SQL Server**, evitando dependencias rígidas que limiten futuras migraciones.

3.2. Auditabilidad.

La transparencia en el funcionamiento de un componente es fundamental para garantizar la seguridad y la trazabilidad de su uso. Un componente auditable permite rastrear cambios, registrar eventos importantes y diagnosticar problemas cuando ocurren fallos.

En sistemas críticos, como plataformas de pago o aplicaciones gubernamentales, la auditabilidad es un requisito obligatorio. Un componente reutilizable debe ofrecer mecanismos como **logs detallados, integración con herramientas de monitoreo y la posibilidad de registrar transacciones o modificaciones en su configuración**.

Si un componente no permite auditar lo que sucede en su interior, su integración en un sistema puede generar incertidumbre y dificultar la detección de errores o intentos de uso indebido. Por eso, muchas organizaciones priorizan soluciones que ofrecen registros detallados de actividad y compatibilidad con herramientas de observabilidad como **Prometheus, Grafana o ELK Stack**.

3.3. Estandarización.

El uso de componentes basados en estándares facilita su integración con otras herramientas y su mantenimiento a largo plazo. Elegir soluciones que sigan estándares abiertos evita depender de tecnologías propietarias que puedan quedar obsoletas o ser difíciles de migrar.

Por ejemplo, en el desarrollo de APIs, es recomendable seguir estándares como **REST o GraphQL**, en lugar de optar por soluciones propietarias que puedan limitar la interoperabilidad. En bases de datos, es preferible utilizar componentes que cumplan con **SQL ANSI** en lugar de formatos específicos de un solo proveedor, lo que facilita la portabilidad de los datos entre distintos sistemas.

El uso de estándares también impacta en la compatibilidad con entornos modernos, como la computación en la nube. Tecnologías como **OCI (Open Container Initiative)** garantizan que los contenedores sean compatibles entre diferentes plataformas, evitando bloqueos tecnológicos que limiten la flexibilidad de los despliegues.

3.4. Escalabilidad y elasticidad.

Un componente reutilizable debe poder manejar aumentos en la carga de trabajo sin comprometer su rendimiento. La **escalabilidad** se refiere a la capacidad de un sistema para aumentar su capacidad agregando más recursos, mientras que la **elasticidad** implica la capacidad de ajustar automáticamente los recursos según la demanda.

Por ejemplo, un servidor de bases de datos como **MongoDB** debe permitir escalar verticalmente (aumentando la capacidad del servidor) o escalar horizontalmente (añadiendo más nodos a un clúster). Si un componente no es escalable, su rendimiento puede degradarse a medida que crece el volumen de usuarios o datos.

La elasticidad es especialmente importante en entornos cloud, donde los recursos pueden asignarse dinámicamente. Un servicio como **AWS Lambda** permite ejecutar funciones bajo demanda, escalando automáticamente sin intervención manual, lo que lo hace ideal para cargas de trabajo variables.

3.5. Rendimiento.

El rendimiento de un componente impacta directamente en la eficiencia de todo el sistema. Un componente mal optimizado puede generar cuellos de botella que afecten la velocidad de respuesta y el consumo de recursos.

Para evaluar el rendimiento de un componente, es recomendable realizar pruebas de carga y revisar métricas como **latencia, throughput y consumo de CPU y memoria**. En bases de datos, se puede analizar la eficiencia de los índices y la velocidad de ejecución de consultas. En aplicaciones web, es útil medir el tiempo de respuesta de las API y la capacidad de procesamiento simultáneo de solicitudes.

Herramientas como **JMeter, Locust o k6** permiten simular tráfico real y evaluar el comportamiento de los componentes bajo distintas condiciones de carga.

3.6. Consumo de recursos.

Un componente eficiente debe aprovechar los recursos del sistema sin desperdiciar memoria, CPU o ancho de banda. En entornos cloud, donde el uso de recursos está directamente relacionado con el coste, elegir componentes optimizados puede generar ahorros significativos.

Por ejemplo, algunos frameworks de backend como **Quarkus o Micronaut** están diseñados para consumir menos memoria que alternativas más pesadas como Spring Boot, lo que los hace ideales para entornos con restricciones de hardware o para ejecutar aplicaciones en contenedores de tamaño reducido.

Es recomendable revisar los requisitos mínimos de cada componente y comparar su consumo de recursos en pruebas reales antes de integrarlo en un sistema.

3.7. Seguridad.

La seguridad es un aspecto clave en la selección de componentes, especialmente en aplicaciones que manejan datos sensibles o que están expuestas a Internet. Un componente inseguro puede abrir la puerta a vulnerabilidades que comprometan todo el sistema.

Al evaluar un componente, es importante revisar si sigue buenas prácticas de seguridad, como **cifrado de datos, control de acceso y protección contra ataques conocidos**. También es recomendable verificar si recibe actualizaciones de seguridad frecuentes y si tiene una comunidad activa que monitorea y soluciona vulnerabilidades.

En entornos empresariales, muchas organizaciones prefieren componentes que cumplen con certificaciones de seguridad, como **ISO 27001 o SOC 2**, que garantizan que han sido evaluados bajo estándares rigurosos:

3.8. Características de mantenimiento y actualización.

Un componente bien mantenido reduce los riesgos de quedarse obsoleto o generar problemas de compatibilidad en el futuro. Antes de elegir un componente, es útil revisar su historial de actualizaciones y si cuenta con documentación clara y soporte activo.

Un software que no recibe mantenimiento puede volverse incompatible con versiones más recientes del sistema operativo o de otras herramientas. En cambio, un componente con una comunidad activa o respaldo de una empresa garantiza que recibirá mejoras y correcciones de seguridad en el tiempo.

3.9. Compatibilidad con entornos cloud y contenedores.

Con el crecimiento de la computación en la nube y el uso de contenedores, es importante que los componentes seleccionados sean compatibles con estos entornos.

Por ejemplo, un sistema de bases de datos como **PostgreSQL** es altamente compatible con despliegues en contenedores y puede ejecutarse en servicios como **Amazon RDS** o **Google Cloud SQL**, facilitando su integración en arquitecturas modernas.

Elegir componentes diseñados para funcionar en entornos escalables y con soporte para orquestadores como **Kubernetes** permite mantener la flexibilidad y facilitar despliegues automatizados.

Actividad 6

Lee cada afirmación y marca si es verdadera (V) o falsa (F). Justifica tu respuesta en caso de que sea falsa.

- Un componente reutilizable debe adaptarse a distintos entornos sin necesidad de realizar modificaciones.

- La auditabilidad de un componente no es importante si se utiliza en aplicaciones internas sin acceso externo.
- Elegir componentes basados en estándares abiertos facilita la integración y el mantenimiento a largo plazo.
- La escalabilidad y la elasticidad son términos equivalentes en el contexto de la selección de componentes.
- Un componente con alto consumo de CPU y memoria siempre es la mejor opción porque garantiza mejor rendimiento.

Lee cada pista y escribe qué criterio de selección de componentes está describiendo.

Adivinanza 1:

Si me usas, puedes ver,

qué hizo el código ayer.

Errores, cambios y más,

te ayudo a rastrear.

Adivinanza 2:

Si tu software no es ligero,

y consume sin parar,

cuando escale tu proyecto,

problemas te va a causar.

Adivinanza 3:

Si tu sistema quiere crecer,

más nodos debe tener,

y si los recursos han cambiado,

ajustarme está asegurado.

Adivinanza 4:

Con REST o con GraphQL,

mejor es si sigo un papel,

para que todos me entiendan bien,

y no cambien su parecer.

Adivinanza 5:

Si el software quieres cuidar,

de ataques me debes blindar,

cifrado y permisos bien debes usar,

si a hackers no quieres invitar.

4. Proceso de selección de componentes.

Elegir los componentes adecuados para un sistema de software no es solo cuestión de encontrar el que mejor se ajuste a las necesidades inmediatas. También es importante evaluar su integración con otros sistemas, su rendimiento a largo plazo y su compatibilidad con prácticas de desarrollo modernas. Un buen proceso de selección garantiza que los componentes sean reutilizables, escalables y seguros, facilitando su mantenimiento y actualización con el tiempo.

El proceso de selección de componentes no es algo que se haga una sola vez y se olvide. A medida que las aplicaciones evolucionan, pueden surgir nuevas necesidades que requieran cambios en los componentes utilizados. Por eso, es fundamental seguir un enfoque estructurado que incluya evaluación de requisitos, integración, pruebas, mantenimiento y actualización continua.

4.1. Evaluación de componentes según requisitos.

Antes de incorporar un componente en un sistema, es necesario analizar si cumple con los requisitos técnicos y funcionales del proyecto. Esto implica evaluar aspectos como compatibilidad con la arquitectura existente, facilidad de integración, rendimiento esperado y nivel de soporte disponible.

Por ejemplo, si se necesita un sistema de mensajería entre microservicios, una de las primeras preguntas sería si conviene usar **RabbitMQ, Apache Kafka o NATS**. Para tomar una decisión informada, se pueden comparar características como el soporte para colas persistentes, el consumo de recursos y la facilidad de escalabilidad en entornos cloud.

También es importante considerar el grado de madurez del componente. Si se trata de un software de código abierto, se debe revisar la frecuencia de actualizaciones y la actividad de su comunidad. Un componente sin mantenimiento reciente puede convertirse en un problema de seguridad o compatibilidad en el futuro.

4.2. Diseño y codificación (código de enlace).

Una vez seleccionado el componente, hay que asegurarse de que pueda integrarse correctamente en el sistema. Esto se hace mediante el diseño de código de enlace, que permite la comunicación entre el nuevo componente y los demás elementos del software.

4.2.1. Enlace de componentes con otros sistemas.

Cuando un componente no ha sido diseñado específicamente para el sistema en el que se va a usar, es posible que sea necesario escribir código que actúe como intermediario. Este código de enlace puede ser una API, un adaptador o un conector que permita la comunicación entre distintas partes del software sin modificar su estructura interna.

Por ejemplo, si se necesita integrar un servicio de autenticación externo como **OAuth 2.0 o OpenID Connect**, es posible que se requiera un middleware que traduzca los tokens de autenticación en un formato compatible con la aplicación.

4.2.2. Integración.

La integración de un componente implica conectarlo con otros sistemas y también asegurarse de que interactúe de forma eficiente y sin generar conflictos. Una buena práctica es realizar pruebas de integración antes de implementarlo en producción, para detectar posibles incompatibilidades o errores de comunicación.

En entornos donde se usan microservicios, herramientas como **Istio o Linkerd** pueden facilitar la integración mediante la gestión de la comunicación entre servicios, aplicando control de tráfico y balanceo de carga de manera transparente.

4.2.3. Configuración.

Después de integrar el componente, es necesario configurarlo adecuadamente para que funcione según los requisitos del sistema. Dependiendo del tipo de componente, la configuración puede realizarse mediante archivos YAML, variables de entorno o ajustes dentro de una plataforma de administración.

Un caso típico es la configuración de bases de datos como **PostgreSQL o MySQL**, donde se pueden ajustar parámetros como el número de conexiones simultáneas, la memoria caché utilizada o las políticas de seguridad para proteger los datos.

4.3. Automatización de pruebas con CI/CD (Jenkins, GitHub Actions, GitLab CI/CD).

Para garantizar que un componente funciona correctamente en el sistema, es recomendable automatizar su prueba dentro del flujo de integración y entrega continua (**CI/CD**). Herramientas como **Jenkins, GitHub Actions y GitLab CI/CD** permiten definir pipelines de pruebas que verifican el correcto funcionamiento del componente en cada nueva versión del software.

Por ejemplo, si se integra una nueva librería en un backend basado en **Node.js**, se pueden configurar pruebas automatizadas con **Jest o Mocha**, de modo que cada vez que se haga un cambio en el código, se ejecuten test que validen su compatibilidad con el sistema.

4.4. Detección de fallos.

Un buen componente no solo debe funcionar bien en condiciones normales, sino que también debe ser capaz de manejar errores de manera eficiente. Para detectar fallos, se pueden emplear técnicas como **pruebas de carga**, **simulación de fallos** y **monitorización en tiempo real**.

Por ejemplo, en sistemas distribuidos, herramientas como **Chaos Monkey de Netflix** permiten probar la resiliencia del sistema desconectando aleatoriamente servicios y observando cómo reacciona la infraestructura ante fallos inesperados.

4.5. Mantenimiento y gestión de configuraciones.

Los componentes seleccionados deben poder mantenerse y actualizarse sin generar problemas en el sistema. Para ello, es recomendable gestionar su configuración de manera centralizada utilizando herramientas como **Ansible, Puppet o Terraform**, que permiten definir la configuración de los componentes como código, asegurando que todos los entornos tengan ajustes consistentes.

En sistemas modernos, los gestores de configuración también facilitan la actualización de software sin necesidad de intervención manual, reduciendo el riesgo de errores humanos y asegurando despliegues más confiables.

4.6. Actualización de componentes en entornos DevOps (Rolling Updates, Canary Releases, Blue-Green Deployments).

Actualizar un componente en un sistema en producción puede ser riesgoso si no se hace correctamente. Para minimizar el impacto de los cambios, en entornos **DevOps** se utilizan estrategias de despliegue que permiten introducir nuevas versiones de forma gradual:

- **Rolling Updates:** Se actualizan los componentes poco a poco, reemplazando las versiones antiguas por las nuevas sin detener el sistema.
- **Canary Releases:** Se despliega la nueva versión solo a un pequeño grupo de usuarios para detectar posibles errores antes de un lanzamiento completo.
- **Blue-Green Deployments:** Se mantienen dos versiones del sistema en paralelo (una activa y otra en espera), permitiendo hacer cambios sin afectar la disponibilidad del servicio.

Estas estrategias permiten actualizar componentes de manera segura, evitando interrupciones y reduciendo el impacto de posibles fallos en producción.

4.7. Métodos de selección de uso común.

Existen distintas metodologías para seleccionar componentes de manera eficiente y estructurada.

4.7.1. DevOps y metodologías ágiles.

El enfoque **DevOps** y las metodologías ágiles promueven la selección de componentes basándose en la capacidad de automatización, integración y entrega continua. En lugar de elegir componentes de manera aislada, se evalúa cómo encajan dentro del flujo de desarrollo y despliegue del sistema.

Por ejemplo, en una empresa que trabaja con **Scrum**, se puede definir un criterio de selección de componentes dentro del backlog del proyecto, asegurando que cada elección tenga en cuenta la compatibilidad con las herramientas de CI/CD y la infraestructura de despliegue.

4.7.2. SAST y DAST para evaluación de seguridad en components.

La seguridad es un aspecto clave en la selección de componentes. Para evaluar posibles vulnerabilidades, se utilizan técnicas como:

- **SAST (Static Application Security Testing):** Analiza el código fuente en busca de vulnerabilidades antes de ejecutarlo. Herramientas como **SonarQube o Checkmarx** permiten detectar errores de seguridad en las dependencias de un proyecto.
- **DAST (Dynamic Application Security Testing):** Evalúa la seguridad del componente en tiempo de ejecución, probando su comportamiento ante ataques simulados.

Utilizar estas metodologías ayuda a garantizar que los componentes seleccionados sean seguros y no introduzcan riesgos en el sistema.

Actividad 7

Responde a las siguientes preguntas en un breve párrafo, argumentando tu respuesta con ejemplos o experiencias previas.

Al elegir un nuevo componente para un sistema de software, ¿qué crees que es más importante: su rendimiento o su seguridad? Justifica tu respuesta considerando casos en los que un aspecto podría ser más prioritario que el otro.

Si una empresa quiere modernizar su infraestructura, pero aún depende de componentes antiguos, ¿qué pasos crees que debería seguir para asegurar una transición efectiva? Explica qué factores deberían evaluar antes de hacer el cambio.

Las metodologías ágiles y DevOps han cambiado la forma en que se seleccionan y actualizan los componentes de software. ¿Cómo crees que este enfoque ha mejorado la calidad y estabilidad de los sistemas?

Imagina que un equipo de desarrollo está evaluando dos componentes para integrar en su sistema: uno tiene mejor compatibilidad con su infraestructura actual, pero el otro tiene más soporte y actualizaciones frecuentes. ¿Cuál crees que deberían elegir y por qué?

Los errores en la selección de componentes pueden generar problemas en la escalabilidad y mantenimiento de un sistema. ¿Puedes pensar en un ejemplo donde una mala elección de un componente podría afectar a largo plazo el funcionamiento de una aplicación?

5. Prueba de autoevaluación.

¿Qué tipo de componente de software suele ofrecer soporte y estabilidad, pero puede tener restricciones de personalización?

a) Código abierto

b) Comercial (COTS)

c) Frameworks de desarrollo

¿Cuál de estos frameworks de código abierto es utilizado en el desarrollo de aplicaciones con Java?

a) Spring Boot

b) Express.js

c) Nginx

¿Qué ventaja ofrece un componente de código abierto en comparación con uno comercial?

a) Mayor flexibilidad y posibilidad de modificación

b) Siempre incluye soporte técnico garantizado

c) No requiere mantenimiento por parte de la comunidad

¿Cuál de los siguientes es un factor clave al seleccionar un componente reutilizable?

a) La disponibilidad de emojis en la documentación

b) La compatibilidad con entornos cloud y contenedores

c) Que tenga una interfaz gráfica atractiva

¿Qué herramienta permite definir infraestructura como código para gestionar la configuración y despliegue de componentes?

a) Terraform

b) PostgreSQL

c) GitHub

Un componente _____ permite su modificación y adaptación libremente sin pagar licencias.

Un software COTS es un componente _____ que se compra a un proveedor y generalmente no se puede modificar.

_____ es un framework de Node.js utilizado para desarrollar APIs y aplicaciones web.

La herramienta _____ permite gestionar la infraestructura de manera declarativa, automatizando la creación y configuración de recursos.

Un buen criterio de selección de componentes es que tenga alta _____, permitiendo su adaptación a diferentes entornos y necesidades.

Control de calidad de componentes