Diseño y arquitectura de microservicios

Priyanka Vergadia
Developers Advocate, Google Cloud

Hola, soy Priyanka Vergadia, Developers Advocate de Google Cloud. En este módulo, presentamos la arquitectura de aplicaciones y el diseño de microservicios.

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Específicamente, aprenderá sobre las arquitecturas de microservicios y cómo desglosar aplicaciones monolíticas en microservicios. Los beneficios de una arquitectura de microservicios para aplicaciones nativas de la nube se analizan y contrastan con una aplicación monolítica.

Se investigarán los desafíos de desglosar aplicaciones en microservicios con límites claros para admitir unidades que se puedan implementar independientemente.

Aprenderá a diseñar arquitecturas para algunos de los desafíos técnicos más importantes de las arquitecturas de microservicios, como la administración de estados, la confiabilidad y la escalabilidad.

Una vez que se elija una arquitectura de microservicios nativa de la nube, se presentarán las prácticas recomendadas para el desarrollo y la implementación en torno a las prácticas recomendadas de 12 factores ampliamente reconocidas.

Al final del módulo, veremos un componente básico de una arquitectura de microservicios, es decir, el diseño de interfaces de servicios coherentes con acoplamiento bajo.

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Específicamente, aprenderá sobre las arquitecturas de microservicios y cómo desglosar aplicaciones monolíticas en microservicios. Los beneficios de una arquitectura de microservicios para aplicaciones nativas de la nube se analizan y contrastan con una aplicación monolítica.

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Se investigarán los desafíos de desglosar aplicaciones en microservicios con límites claros para admitir unidades que se puedan implementar independientemente.

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Aprenderá a diseñar arquitecturas para algunos de los desafíos técnicos más importantes de las arquitecturas de microservicios, como la administración de estados, la confiabilidad y la escalabilidad. .

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Una vez que se elija una arquitectura de microservicios nativa de la nube, se presentarán las prácticas recomendadas para el desarrollo y la implementación en torno a las prácticas recomendadas de 12 factores ampliamente reconocidas.

Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

Al final del módulo, veremos un componente básico de una arquitectura de microservicios,

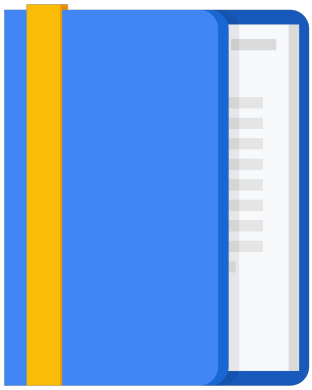
Objetivos de aprendizaje

- Desglosar las aplicaciones monolíticas en microservicios
- Reconocer los límites adecuados de los microservicios
- Diseñar servicios con y sin estado para optimizar la escalabilidad y la confiabilidad
- Implementar servicios siguiendo las prácticas recomendadas de 12 factores
- Crear servicios con acoplamiento bajo implementando una arquitectura de REST bien diseñada
- Diseñar API de servicio RESTful estándares y coherentes

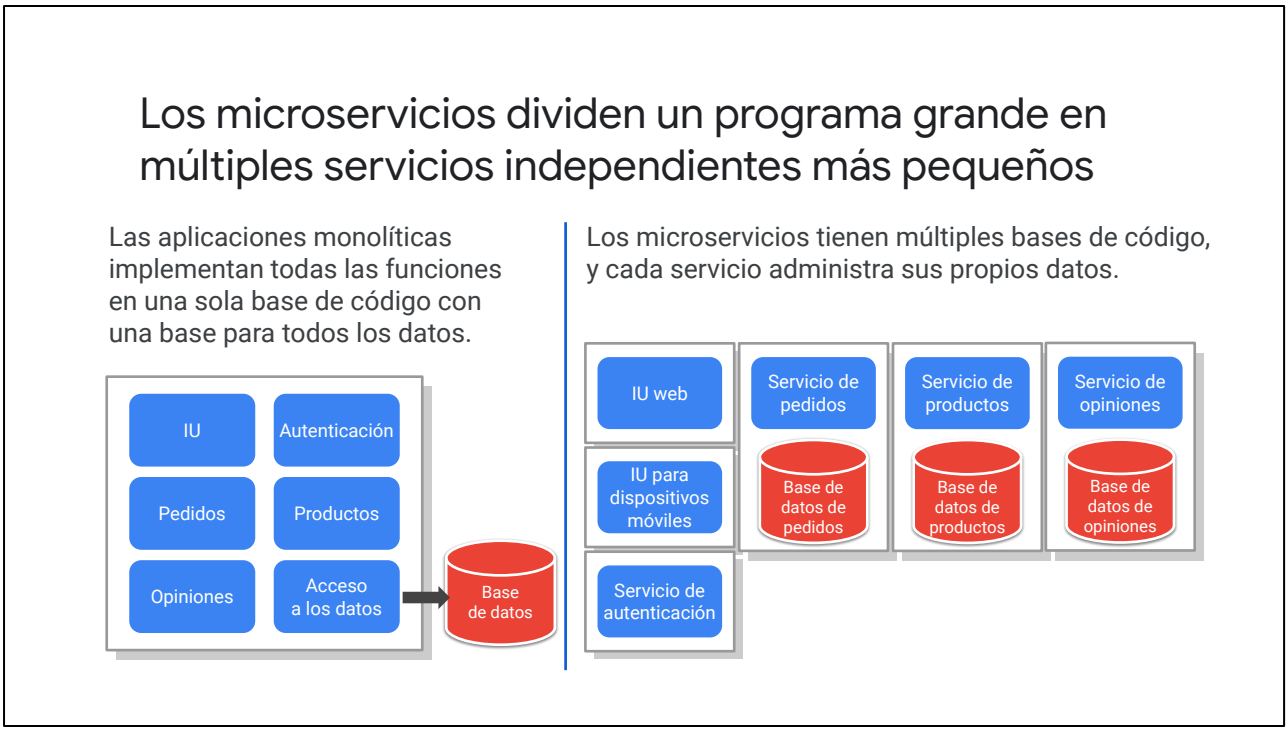
es decir, el diseño de interfaces de servicios coherentes con acoplamiento bajo.

Temario

- Microservicios
- Prácticas recomendadas sobre microservicios
- REST
- API



Comencemos por ver los microservicios en más detalle.



Los microservicios dividen un programa grande en diversos servicios independientes más pequeños, como se muestra a la derecha, a diferencia de una aplicación monolítica, que implementa todas las funciones en una sola base de código con una base para todos los datos, como se muestra a la izquierda.

Los microservicios son la tendencia actual de la industria; sin embargo, es importante garantizar que exista un buen motivo para seleccionar esta arquitectura. El motivo principal es facultar a los equipos para que trabajen de forma independiente y envíen su trabajo a producción a su propio ritmo. Esto es compatible con el escalamiento de la organización, es decir, agregar más equipos para incrementar la velocidad. También hay un beneficio adicional: poder escalar los microservicios de forma independiente según los requisitos.

A nivel de la arquitectura, una aplicación diseñada de forma monolítica o en torno a microservicios debe consistir en componentes modulares con límites definidos claramente. Con una aplicación monolítica, todos los componentes se empaquetan al momento de la implementación y se aplican en conjunto. Con los microservicios, los componentes individuales se pueden implementar. Google Cloud proporciona varios servicios de procesamiento que facilitan la implementación de microservicios. Estos incluyen App Engine, Cloud Run, GKE y Cloud Functions. Cada uno ofrece diferentes niveles de detalle y control, y se analizarán más adelante en el curso.

Para lograr la independencia en los servicios, cada uno de ellos debe tener su propio almacén de datos. Esto permite seleccionar la mejor solución de almacén de datos

para ese servicio y también mantiene la independencia de los servicios. No queremos introducir el acoplamiento entre servicios mediante un almacén de datos.

Ventajas y desventajas de las arquitecturas de microservicios

- Son fáciles de desarrollar y mantener
- Representan un riesgo reducido durante la implementación de versiones nuevas
- Los servicios escalan de forma independiente para optimizar el uso de infraestructura
- Son más rápidas para innovar y agregar funciones nuevas
- Pueden usar lenguajes y frameworks diferentes para distintos servicios
- Se puede elegir el entorno de ejecución adecuado para cada servicio

- Aumenta la complejidad cuando se establece la comunicación entre servicios
- Aumenta la latencia entre los límites de servicios
- Existen inquietudes en cuanto a la protección del tráfico entre servicios
- Implican diversas implementaciones
- Se debe garantizar que los cambios de versiones no afecten a los clientes
- Se debe mantener la retrocompatibilidad con los clientes a medida que evolucionan los microservicios

Un microservicio diseñado apropiadamente puede ayudar a lograr los siguientes objetivos:

- Definir contratos sólidos entre los diversos microservicios
- Permitir ciclos de implementación independientes, incluida la reversión
- Facilitar pruebas de actualización simultáneas y A/B en subsistemas
- Reducir al mínimo la automatización de pruebas y la sobrecarga de garantía de calidad
- Mejorar la claridad de los registros y la supervisión
- Proporcionar contabilidad de costos detallada
- Aumentar la escalabilidad y la confiabilidad de las aplicaciones a través del escalamiento de unidades más pequeñas

Sin embargo, las ventajas deben compensar los desafíos que plantea este estilo de arquitectura. Entre estos desafíos se incluyen los siguientes:

- Puede ser difícil definir límites claros entre servicios para admitir el desarrollo y la implementación independientes
- Mayor complejidad de infraestructura con servicios distribuidos que registran más puntos de fallas
- La latencia que introducen los servicios de redes es mayor y es necesario crear resiliencia para controlar posibles fallas y retrasos
- Debido a las redes involucradas, es necesario ofrecer seguridad para la comunicación de servicio a servicio, lo que aumenta la complejidad de la infraestructura
- Gran necesidad de administrar interfaces de servicios y aplicarles control de

- versiones. Con servicios que se pueden implementar independientemente, se incrementa la necesidad de mantener la retrocompatibilidad.

La clave para diseñar la arquitectura de aplicaciones de microservicios es reconocer los límites del servicio

Desglose las aplicaciones por función para minimizar las dependencias	Organice los servicios por capa de arquitectura	Aísle los servicios que ofrezcan funciones compartidas
• Servicio de opiniones • Servicio de pedidos • Servicio de productos • Etcétera	• Interfaces de usuario de iOS, de Android y web • Servicios de acceso a los datos	• Servicio de autenticación • Servicio de informes • Etcétera

Ahora bien, desglosar aplicaciones en microservicios es uno de los desafíos técnicos más grandes del diseño de aplicaciones. En este caso, las técnicas como el diseño basado en dominios son extremadamente útiles para identificar grupos funcionales lógicos.

El primer paso es desglosar la aplicación por función o grupo funcional para minimizar las dependencias. Por ejemplo, considere una aplicación de venta minorista en línea. Los grupos funcionales lógicos podrían ser: administración de productos, opiniones, cuentas y pedidos. Estos grupos luego forman miniaplicaciones que exponen una API. Posiblemente, múltiples microservicios implementarán cada una de estas miniaplicaciones a nivel interno. En ese mismo nivel, estos microservicios se organizan por capa de arquitectura, y cada uno debería poder implementarse y escalarse de forma independiente.

Cualquier análisis también identificará los servicios compartidos, como la autenticación, que luego se aíslan e implementan de forma independiente de las miniaplicaciones.

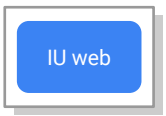
Los servicios con estado plantean desafíos diferentes que aquellos sin estado

Los servicios con estado administran datos almacenados con el tiempo

- Son más difíciles de escalar.
- Son más difíciles de actualizar.
- Requieren crear copias de seguridad.

Los servicios con estado obtienen sus datos del entorno, o bien de otros servicios con estado

- Son fáciles de escalar agregando instancias.
- Son fáciles de migrar a versiones nuevas.
- Son fáciles de administrar.



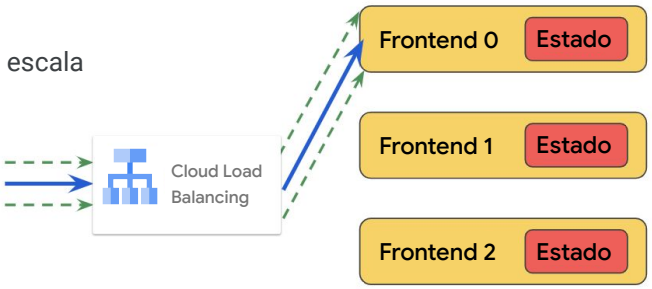
Cuando diseña microservicios, los servicios que no mantienen un estado, sino que lo obtienen del entorno o de servicios sin estado, son más fáciles de administrar. Es decir, son más fáciles de escalar y administrar, así como de migrar a versiones nuevas gracias a su falta de estado.

Sin embargo, por lo general, no es posible evitar usar servicios con estado en algún punto en una aplicación basada en microservicios. Por lo tanto, es importante entender las implicaciones de tener servicios con estado en la arquitectura del sistema. Esto incluye plantear desafíos significativos con respecto a la capacidad de escalar y mejorar los servicios.

En las primeras etapas del diseño de aplicaciones de microservicios, es importante estar conscientes de cómo se administrará el estado. Le presentaré algunas sugerencias y prácticas recomendadas para lograrlo.

Evite almacenar el estado compartido en memoria en sus servidores

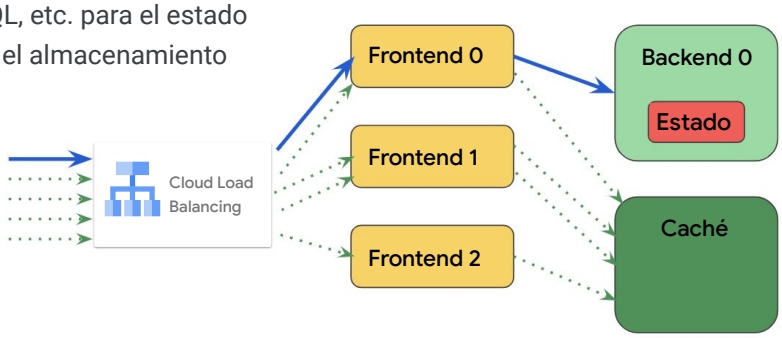
- Requiere sesiones fijas para configurarse en el balanceador de cargas.
- Obstaculiza el ajuste de escala automático elástico.



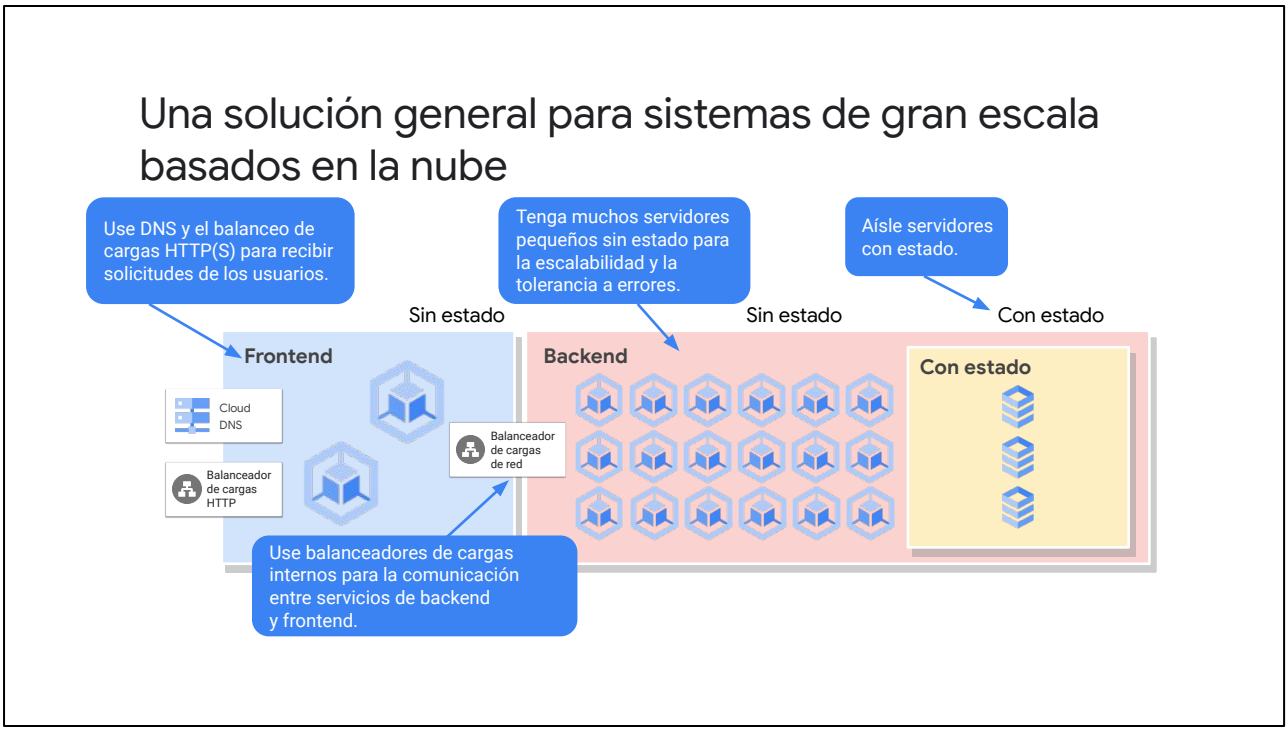
El estado compartido en memoria tiene implicaciones que afectan y anulan muchos de los beneficios de una arquitectura de microservicios. El potencial del ajuste de escala automático de microservicios individuales se ve obstaculizado debido a que deben enviarse solicitudes posteriores de clientes al mismo servidor al que se hizo la solicitud inicialmente. Además, esto requiere la configuración de los balanceadores de cargas para usar sesiones fijas, lo que en Google Cloud se conoce como afinidad de sesión.

Almacene el estado con los servicios de almacenamiento de backend que comparte el servidor de frontend

- Almacene en caché los datos de estado para obtener un acceso más rápido.
- Aproveche los servicios de datos administrados por Google Cloud.
  - Firestore, Cloud SQL, etc. para el estado
  - Memorystore para el almacenamiento en caché



Una práctica recomendada reconocida para diseñar servicios con estado es usar servicios de almacenamiento de backend que comparten los servicios sin estado de frontend. Por ejemplo, para el estado persistente, pueden ser adecuados los servicios de datos administrados por Google Cloud, como Firestore o Cloud SQL. Posteriormente, para mejorar la velocidad del acceso a los datos, estos últimos se pueden almacenar en caché. Memorystore, que es un servicio con alta disponibilidad basado en Redis, es ideal para esta tarea.



En este diagrama, aparece una solución integral en la que se muestra la separación de las etapas de procesamiento del backend y frontend. Un balanceador de cargas distribuye la carga entre los servicios de backend y frontend. Esto permite que el backend escale si necesita estar al nivel de la demanda del frontend. Además, los servidores o servicios con estado también se aíslan. Los servicios con estado pueden usar servicios de almacenamiento persistente y el almacenamiento en caché, como se analizó previamente.

Este diseño permite que una gran parte de la aplicación use la escalabilidad y la tolerancia a errores de servicios de Google Cloud como servicios sin estado. Por el aislamiento de los servidores y servicios con estado, los desafíos de escalar y actualizar se limitan a un subconjunto del conjunto general de servicios.

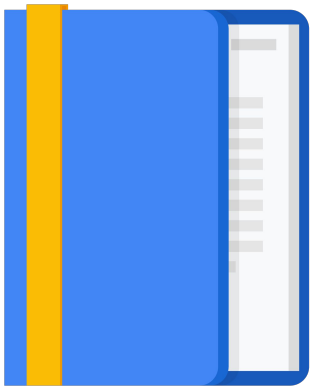
Temario

Microservicios

Prácticas recomendadas sobre microservicios

REST

API



Hablemos de las prácticas recomendadas sobre microservicios.

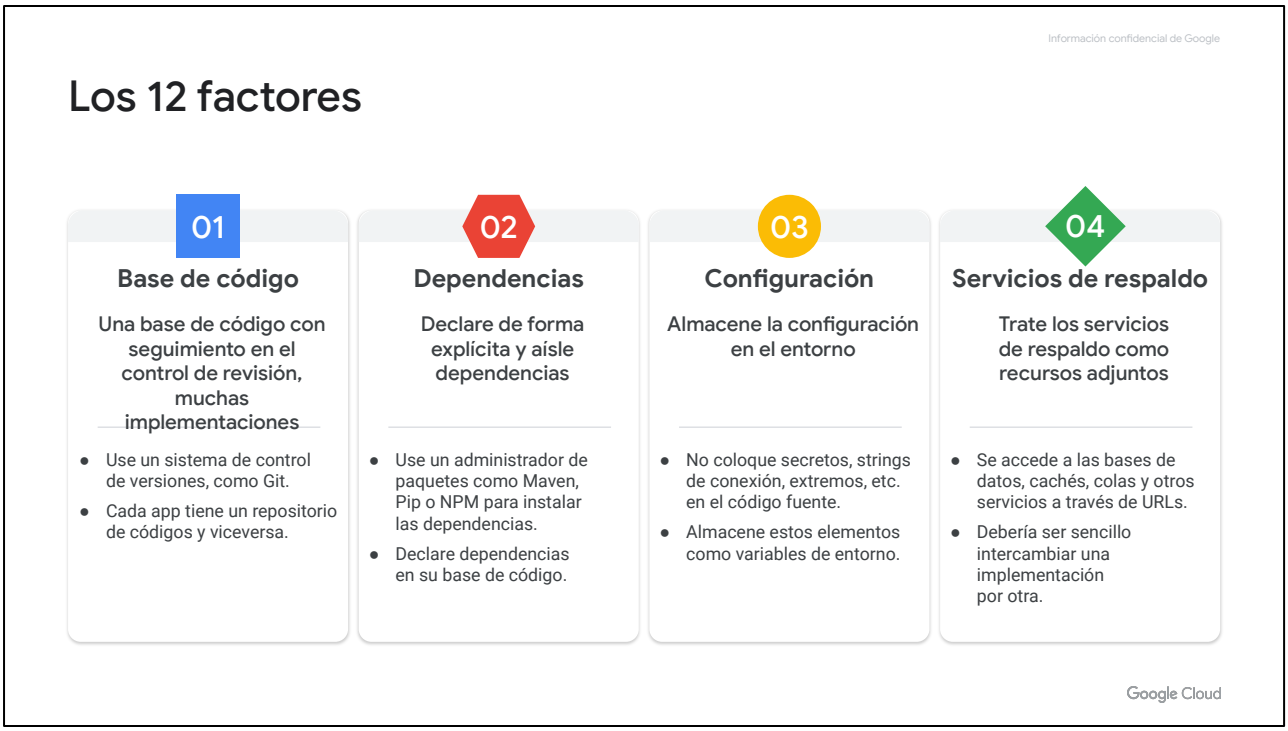
La app de 12 factores es un conjunto de prácticas recomendadas para crear aplicaciones web o de software como servicio

- Maximice la portabilidad.
- Implemente en la nube.
- Permita la implementación continua.
- Escale con facilidad.

THE TWELVE-FACTOR APP

La app de 12 factores es un conjunto de prácticas recomendadas para crear aplicaciones web o de software como servicio. El diseño de doce factores ayuda a desacoplar componentes de la aplicación, de modo que cada uno de ellos se pueda implementar en la nube mediante la implementación continua y se escale verticalmente sin inconvenientes. Los principios del diseño también ayudan a maximizar la portabilidad a diferentes entornos. Debido a que los factores son independientes de cualquier lenguaje de programación o pila de software, el diseño de 12 factores se puede aplicar a una amplia variedad de aplicaciones. Examinemos estas prácticas recomendadas.

[https://12factor.net]
[Artículo de ejemplo: https://content.pivotal.io/blog/beyond-the-twelve-factor-app]



El primer factor es la base de código. Se debe realizar un seguimiento de la base de código en un control de versiones como Git. Cloud Source Repositories proporciona repositorios privados con todas las funciones.

El segundo factor son las dependencias. Hay dos grandes consideraciones en términos de dependencias para las apps de 12 factores: la declaración de dependencia y el aislamiento de dependencia. Las dependencias se deben declarar explícitamente y almacenarse en el control de versiones. El seguimiento de dependencias se realiza con herramientas de lenguaje específico, como Maven para Java y Pip para Python. Una app y sus dependencias se pueden aislar empaquetándolas en un contenedor. Se puede usar Container Registry para almacenar las imágenes y proporcionar un control de acceso detallado.

El tercer factor es la configuración. Cada aplicación tiene una configuración para diferentes entornos, como los de prueba, producción y desarrollo. Esta configuración debe ser externa al código y suele mantenerse en las variables de entorno para la flexibilidad de la implementación.

El cuarto factor son los servicios de respaldo. Cada servicio de respaldo, como una base de datos, una caché o un servicio de mensajería, debe ser accesible a través de URLs y definirse según la configuración. Los servicios de respaldo actúan como abstracciones para el recurso subyacente. El propósito es poder intercambiar un servicio de respaldo por una implementación diferente de manera sencilla.

Para obtener más detalles, consulte:
https://cloud.google.com/solutions/twelve-factor-app-development-on-gcp

Los 12 factores

01

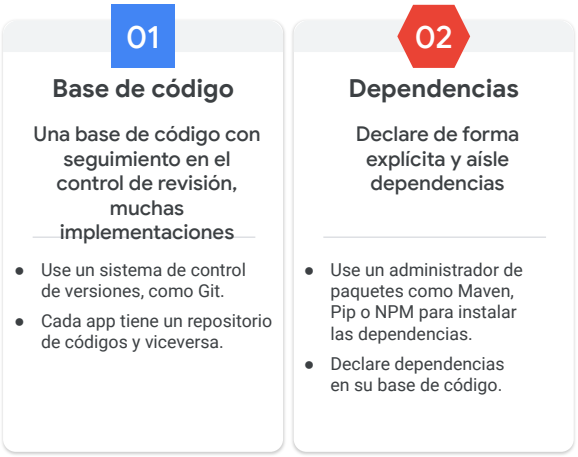
Base de código

Una base de código con seguimiento en el control de revisión, muchas implementaciones

- Use un sistema de control de versiones, como Git.
- Cada app tiene un repositorio de códigos y viceversa.

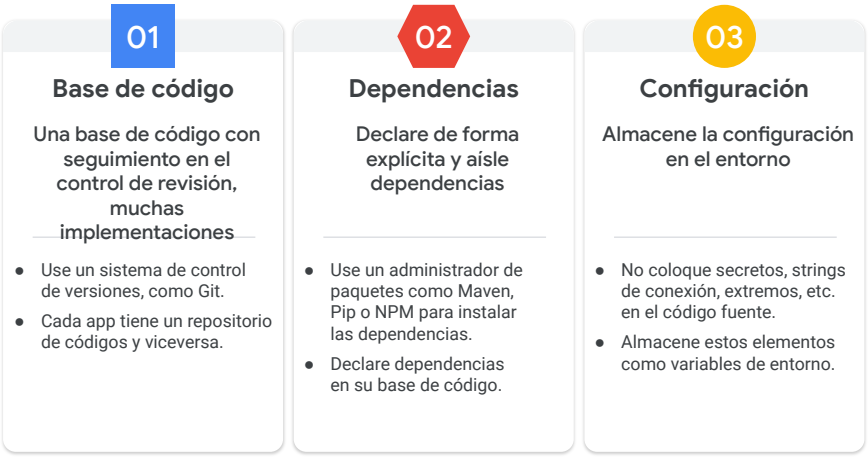
El primer factor es la base de código. Se debe realizar un seguimiento de la base de código en un control de versiones como Git. Cloud Source Repositories proporciona repositorios privados con todas las funciones.

Los 12 factores



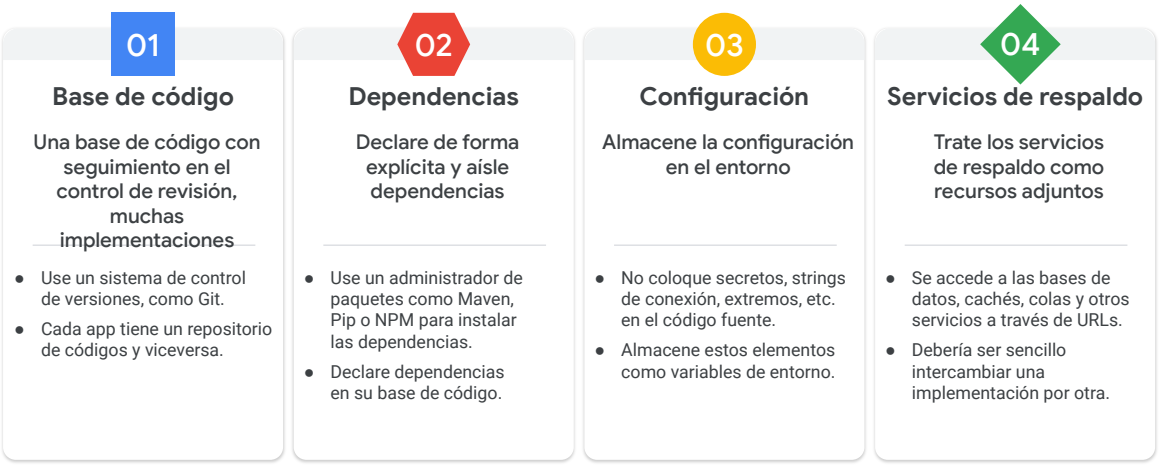
El segundo factor son las dependencias. Hay dos grandes consideraciones en términos de dependencias para las apps de 12 factores: la declaración de dependencia y el aislamiento de dependencia. Las dependencias se deben declarar explícitamente y almacenarse en el control de versiones. El seguimiento de dependencias se realiza con herramientas de lenguaje específico, como Maven para Java y Pip para Python. Una app y sus dependencias se pueden aislar empaquetándolas en un contenedor. Se puede usar Container Registry para almacenar las imágenes y proporcionar un control de acceso detallado.

Los 12 factores



El tercer factor es la configuración. Cada aplicación tiene una configuración para diferentes entornos, como los de prueba, producción y desarrollo. Esta configuración debe ser externa al código y suele mantenerse en las variables de entorno para la flexibilidad de la implementación.

Los 12 factores



El cuarto factor son los servicios de respaldo. Cada servicio de respaldo, como una base de datos, una caché o un servicio de mensajería, debe ser accesible a través de URLs y definirse según la configuración. Los servicios de respaldo actúan como abstracciones para el recurso subyacente. El propósito es poder intercambiar un servicio de respaldo por una implementación diferente de manera sencilla.

Los 12 factores (continuación)

El quinto factor es compile, libere y ejecute**.** El proceso de implementación de software debe dividirse en tres etapas distintas: compilar, liberar y ejecutar. Cada etapa debe dar como resultado un artefacto que se pueda identificar de manera inequívoca. La compilación creará un paquete de implementación a partir del código fuente. Se debe vincular cada paquete de implementación a una liberación específica que sea el resultado de combinar un entorno de ejecución con una compilación. Esto permite realizar reversiones fáciles y brinda un registro de auditoría visible del historial de cada implementación de producción. Luego, la etapa de ejecución simplemente efectúa la aplicación.

El sexto factor son los procesos. Las aplicaciones se ejecutan como uno o más procesos sin estado. Si se requiere el estado, se debe usar la técnica que se abordó anteriormente en este módulo para la administración de estados. Por ejemplo, cada servicio debe tener su propio almacén de datos y cachés con Memorystore para almacenar datos en caché y compartirlos entre los servicios que se usan.

El séptimo factor es la vinculación de puertos. Los servicios deben exponerse mediante un número de puerto. Las aplicaciones empaquetan el servidor web como parte de la aplicación y no requieren un servidor independiente como Apache. En Google Cloud, esas apps se pueden implementar en servicios de plataforma como Compute Engine, GKE, App Engine o Cloud Run.

El octavo factor es la simultaneidad. La aplicación debe ser capaz de escalar horizontalmente. Para ello, debe iniciar procesos nuevos y reducir la escala según

sea necesario a fin de cumplir con la demanda o carga.

Los 12 factores (continuación)

05

Compile, libere y ejecute

Separe la etapas de compilación y ejecución de manera estricta

- La compilación crea un paquete de implementación a partir del código fuente.
- La liberación combina la implementación con la configuración en el entorno de ejecución.
- La ejecución efectúa la aplicación.

El quinto factor es compile, libere y ejecute**.** El proceso de implementación de software debe dividirse en tres etapas distintas: compilar, liberar y ejecutar. Cada etapa debe dar como resultado un artefacto que se pueda identificar de manera inequívoca. La compilación creará un paquete de implementación a partir del código fuente. Se debe vincular cada paquete de implementación a una liberación específica que sea el resultado de combinar un entorno de ejecución con una compilación. Esto permite realizar reversiones fáciles y brinda un registro de auditoría visible del historial de cada implementación de producción. Luego, la etapa de ejecución simplemente efectúa la aplicación.

Los 12 factores (continuación)

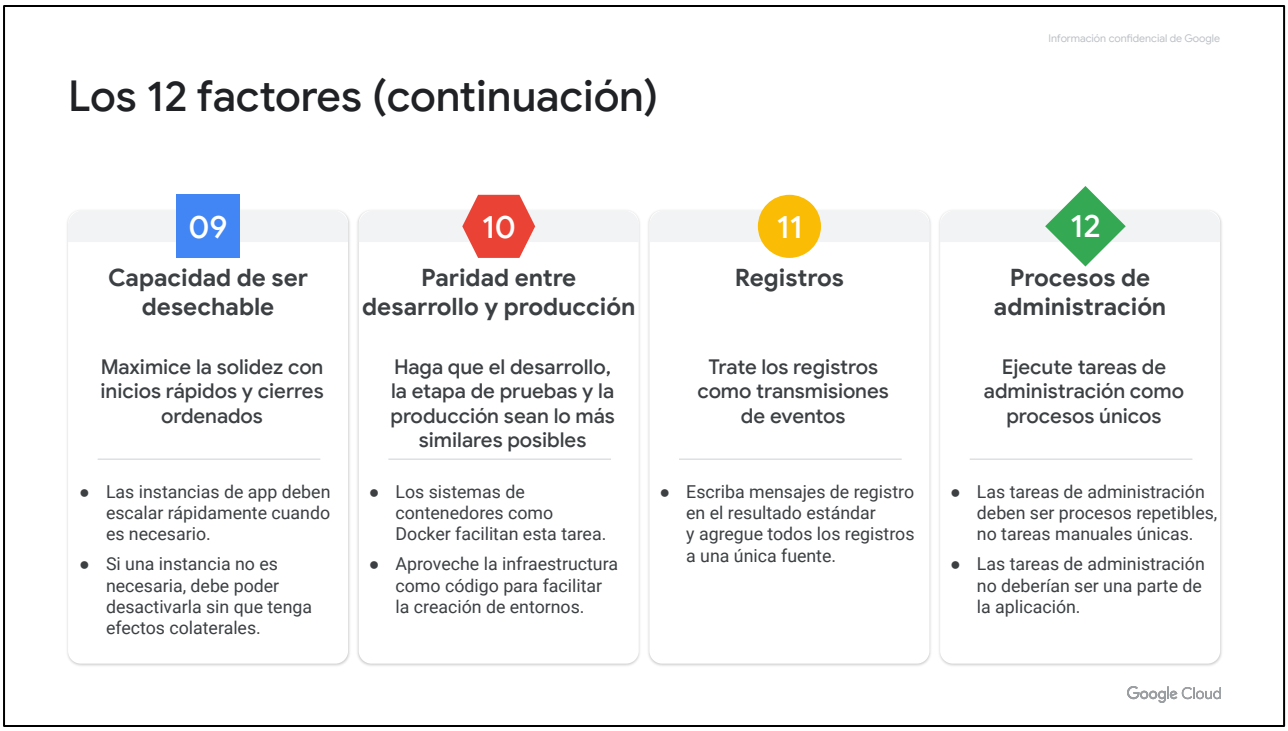
El sexto factor son los procesos. Las aplicaciones se ejecutan como uno o más procesos sin estado. Si se requiere el estado, se debe usar la técnica que se abordó anteriormente en este módulo para la administración de estados. Por ejemplo, cada servicio debe tener su propio almacén de datos y cachés con Memorystore para almacenar datos en caché y compartirlos entre los servicios que se usan.

Los 12 factores (continuación)

El séptimo factor es la vinculación de puertos. Los servicios deben exponerse mediante un número de puerto. Las aplicaciones empaquetan el servidor web como parte de la aplicación y no requieren un servidor independiente como Apache. En Google Cloud, esas apps se pueden implementar en servicios de plataforma como Compute Engine, GKE, App Engine o Cloud Run.

Los 12 factores (continuación)

El octavo factor es la simultaneidad. La aplicación debe ser capaz de escalar horizontalmente. Para ello, debe iniciar procesos nuevos y reducir la escala según sea necesario a fin de cumplir con la demanda o carga.



El noveno factor es la **capacidad de ser desechable.** Las aplicaciones se deben escribir de tal manera que sean más confiables que la infraestructura subyacente en la que se ejecutan. Esto significa que deben ser capaces de controlar fallas temporales en la infraestructura subyacente, además de cerrarse en orden y reiniciarse con rapidez. Las aplicaciones también deben ser capaces de escalar verticalmente con rapidez, adquiriendo y liberando recursos según sea necesario.

El décimo factor es la **paridad entre desarrollo y producción.** El objetivo debe ser utilizar en el desarrollo y la etapa de pruebas los mismos entornos que se usan en la producción. La infraestructura como código y los contenedores de Docker facilitan esta tarea. Los entornos se pueden aprovisionar y configurar rápida y coherentemente a través de variables de entorno. Google Cloud ofrece varias herramientas que se pueden usar para crear flujos de trabajo que mantengan la coherencia de los entornos. Estas herramientas incluyen Cloud Source Repositories, Cloud Storage, Container Registry y Terraform. Terraform usa las API subyacentes de cada servicio de Google Cloud para implementar sus recursos.

El decimoprimer factor son los **registros.** Los registros proporcionan un reconocimiento del estado de sus apps. Es importante separar la recopilación, el procesamiento y el análisis de registros de la lógica central de sus apps. Los registros deben escribirse en el resultado estándar y agregarse en una sola fuente. Esto es particularmente útil cuando sus apps requieren un escalamiento dinámico y se ejecutan en nubes públicas porque elimina la sobrecarga de administrar la ubicación del almacenamiento para los registros y la agregación de VM o

contenedores distribuidos (y, a menudo, efímeros). Google Cloud ofrece un paquete de herramientas que lo ayudan con la recopilación, el procesamiento y el análisis estructurado de los registros.

El decimosegundo factor son los **procesos de administración** que, por lo general, son procesos únicos y deben separarse de la aplicación. Deben ser procesos automatizados y repetibles, no manuales. Según su implementación en Google Cloud, hay muchas opciones para ello, incluidas las siguientes: trabajos cron en GKE, tareas en la nube en App Engine y Cloud Scheduler.

Los 12 factores (continuación)

09

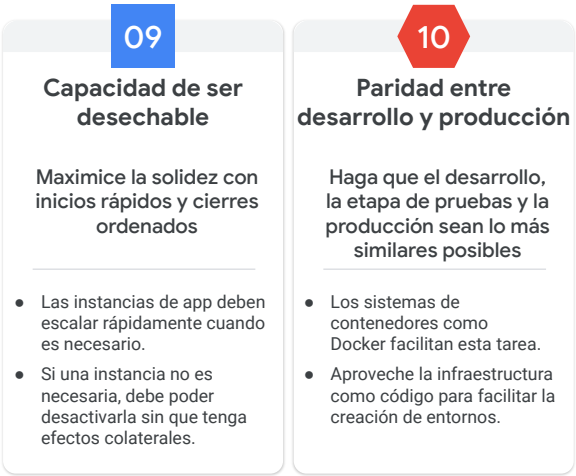
Capacidad de ser desechable

Maximice la solidez con inicios rápidos y cierres ordenados

- Las instancias de app deben escalar rápidamente cuando es necesario.
- Si una instancia no es necesaria, debe poder desactivarla sin que tenga efectos colaterales.

El noveno factor es la **capacidad de ser desechable**. Las aplicaciones se deben escribir de tal manera que sean más confiables que la infraestructura subyacente en la que se ejecutan. Esto significa que deben ser capaces de controlar fallas temporales en la infraestructura subyacente, además de cerrarse en orden y reiniciarse con rapidez. Las aplicaciones también deben ser capaces de escalar verticalmente con rapidez, adquiriendo y liberando recursos según sea necesario.

Los 12 factores (continuación)



El décimo factor es la **paridad entre desarrollo y producción**. El objetivo debe ser utilizar en el desarrollo y la etapa de pruebas los mismos entornos que se usan en la producción. La infraestructura como código y los contenedores de Docker facilitan esta tarea. Los entornos se pueden aprovisionar y configurar rápida y coherentemente a través de variables de entorno. Google Cloud ofrece varias herramientas que se pueden usar para crear flujos de trabajo que mantengan la coherencia de los entornos. Estas herramientas incluyen Cloud Source Repositories, Cloud Storage, Container Registry y Terraform. Terraform usa las API subyacentes de cada servicio de Google Cloud para implementar sus recursos.

Los 12 factores (continuación)

El decimoprimer factor son los **registros.** Los registros proporcionan un reconocimiento del estado de sus apps. Es importante separar la recopilación, el procesamiento y el análisis de registros de la lógica central de sus apps. Los registros deben escribirse en el resultado estándar y agregarse en una sola fuente. Esto es particularmente útil cuando sus apps requieren un escalamiento dinámico y se ejecutan en nubes públicas porque elimina la sobrecarga de administrar la ubicación del almacenamiento para los registros y la agregación de VM o contenedores distribuidos (y, a menudo, efímeros). Google Cloud ofrece un paquete de herramientas que lo ayudan con la recopilación, el procesamiento y el análisis estructurado de los registros.

Los 12 factores (continuación)

El decimosegundo factor son los **procesos de administración** que, por lo general, son procesos únicos y deben separarse de la aplicación. Deben ser procesos automatizados y repetibles, no manuales. Según su implementación en Google Cloud, hay muchas opciones para ello, incluidas las siguientes: trabajos cron en GKE, tareas en la nube en App Engine y Cloud Scheduler.

Actividad 4: Diseñe microservicios para su aplicación

Consulte el Cuaderno de ejercicios de Design and Process.

- Realice un diagrama de los microservicios requeridos para la aplicación de su caso de éxito.

En esta actividad de diseño, trabajará en la actividad 4 del cuaderno de ejercicios de diseño.

En él, diseñará microservicios para su aplicación. El propósito principal es crear un diagrama de los microservicios que requiere la aplicación de su caso de éxito. Algunos de los elementos que debe considerar son los límites de los microservicios y la administración de estados, al igual que los servicios comunes. Use los principios que abordamos en este módulo hasta ahora.

Actividad 4: Diseñe microservicios para su aplicación

Consulte el Cuaderno de ejercicios de Design and Process.

- Realice un diagrama de los microservicios requeridos para la aplicación de su caso de éxito.

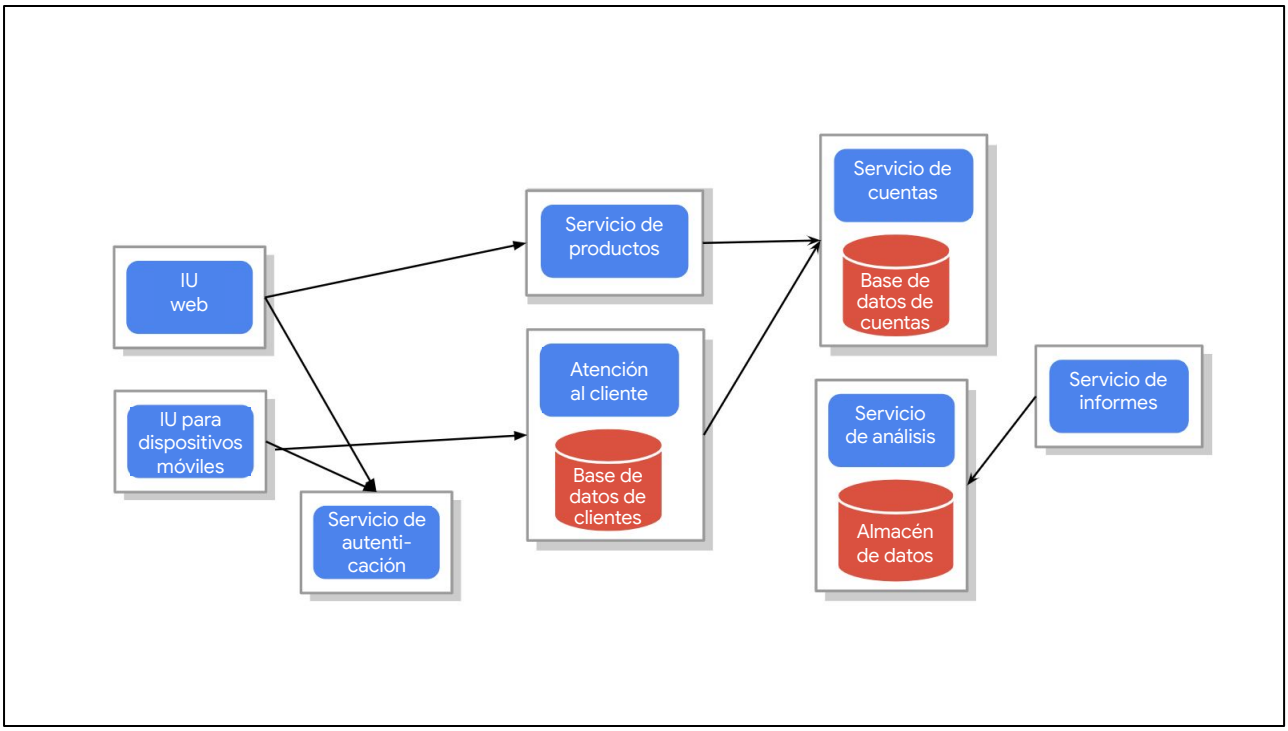
En esta actividad de diseño, trabajará en la actividad 4 del cuaderno de ejercicios de diseño.

Actividad 4: Diseñe microservicios para su aplicación

Consulte el Cuaderno de ejercicios de Design and Process.

- Realice un diagrama de los microservicios requeridos para la aplicación de su caso de éxito.

En él, diseñará microservicios para su aplicación. El propósito principal es crear un diagrama de los microservicios que requiere la aplicación de su caso de éxito. Algunos de los elementos que debe considerar son los límites de los microservicios y la administración de estados, al igual que los servicios comunes. Use los principios que abordamos en este módulo hasta ahora.



Este es un ejemplo del diagrama de microservicios para el sitio web y la aplicación para teléfono celulares de un servicio de banca en línea.

Dibuje un diagrama similar al que se muestra para su caso de éxito.

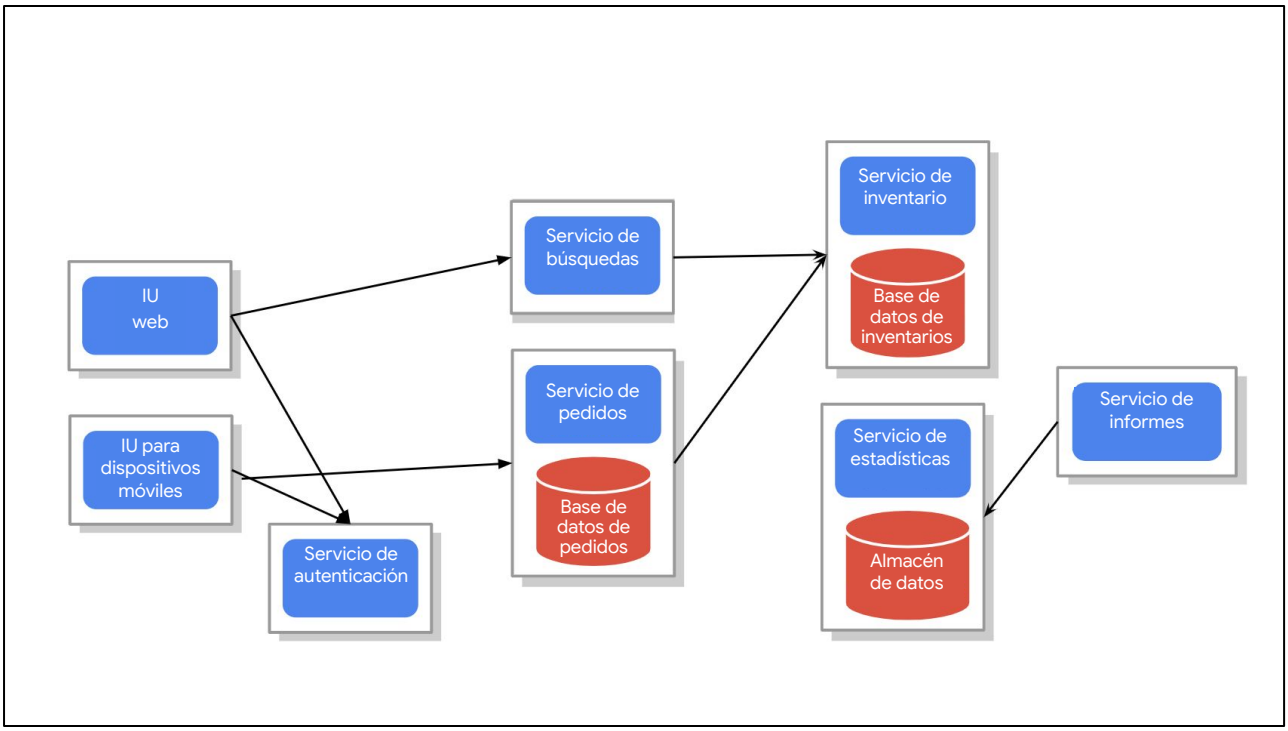
Revise la actividad 4: Diseñe microservicios para su aplicación

- Realice un diagrama de los microservicios requeridos para la aplicación de su caso de éxito.

En esta actividad, se le solicitó crear un diagrama de la aplicación de su caso de éxito con una arquitectura de estilo de microservicio. Conocer la cantidad de microservicios adecuada para su aplicación y reconocer los límites de los microservicios no es algo obvio. Puede que dos programas se parezcan, pero sus arquitecturas podrían ser considerablemente diferentes según la cantidad de usuarios, el tamaño de los datos, la seguridad y muchos otros factores.

Algunos servicios podrían facilitar la implementación y la comunicación entre servicios, pero también dificultar el desarrollo y la adición de funciones nuevas. Tener una cantidad mayor de servicios más pequeños permite que sea más sencillo entender e implementar cada servicio, pero complica la arquitectura de su programa.

Como en muchas otras situaciones en la vida, lo que se busca es encontrar el equilibrio correcto compensando un tipo de complejidad con otro diferente y desarrollar un sistema que, en general, sea lo más sencillo y fácil de manejar posible.



Aquí tenemos un diagrama de ejemplo en el que se representan los microservicios de nuestro portal de viajes en línea. Supongo que podemos diseñar esto de muchas formas diferentes. Realmente, no hay una sola forma correcta de diseñar una aplicación.

Observe que tenemos servicios independientes para nuestras IU web y para dispositivos móviles. Hay un servicio de autenticación compartido y tenemos microservicios para búsquedas, pedidos, inventarios, informes y análisis. Recuerde que cada uno de estos servicios se implementará como una aplicación independiente. En la medida de lo posible, queremos utilizar servicios sin estado, pero los de inventario y pedidos deberán tener bases de datos. Además, el servicio de análisis proporcionará un almacén de datos.

Este podría ser un buen punto de partida, ya que se podrán realizar ajustes según sea necesario cuando comencemos a implementar la aplicación.

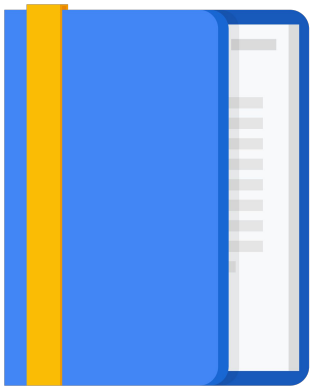
Temario

Microservicios

Prácticas recomendadas sobre microservicios

REST

API



Hablemos sobre el diseño de microservicios basados en REST y HTTP para lograr servicios independientes con acoplamiento bajo.

Un buen diseño de microservicio tiene acoplamiento bajo

- Los clientes no necesitan saber tantos detalles de los servicios que usan.
- Los servicios se comunican a través de HTTPS con cargas útiles basadas en texto.
  - El cliente hace solicitudes GET, POST, PUT o DELETE.
  - El cuerpo de la solicitud tiene formato JSON o XML.
  - Los resultados se muestran en formato JSON, XML o HTML.
- Los servicios deben agregar funciones sin afectar a los clientes existentes.
  - Agregue, pero no quite, elementos de las respuestas.

Si los microservicios no tienen acoplamiento bajo, terminará con una aplicación monolítica bastante complicada.

Uno de los aspectos más importantes de las aplicaciones basadas en microservicios es la capacidad de implementar microservicios que sean completamente independientes entre sí. Para lograr esa independencia, cada microservicio debe proporcionar un contrato bien definido con control de versiones a sus clientes, que son otros microservicios o aplicaciones. Los servicios no deben poner fin a estos contratos con control de versiones hasta que se determine que no hay ningún microservicio que depende de algún contrato con control de versiones específico. Recuerde que puede ser necesario revertir otros microservicios a una versión anterior del código que requiera un contrato previo; por eso, es importante tener en cuenta esa posibilidad en sus políticas de baja.
Lograr una cultura en torno a contratos bien definidos con control de versiones es probablemente el aspecto organizativo más desafiante de una aplicación estable basada en microservicios.

En términos generales, los servicios se comunican con HTTPS y cargas útiles basadas en texto, como JSON o XML, y usan los verbos HTTP, como GET y POST, para proporcionar significados a las acciones solicitadas. Los clientes solo necesitan conocer los detalles mínimos para usar el servicio, es decir, el URI, la solicitud y los formatos de los mensajes de respuesta.

La arquitectura de REST admite el acoplamiento bajo

- REST significa *Representational State Transfer (transferencia de estado representacional)*.
- No depende de protocolos.
  - El protocolo HTTP es el más común.
  - Hay otras opciones posibles, como gRPC.
- Los extremos de servicio que admiten REST se llaman *RESTful*.
- Los clientes y el servidor se comunican con el procesamiento de solicitudes y respuestas.

La arquitectura de REST admite el acoplamiento bajo. REST significa *Representational State Transfer (transferencia de estado representacional)*, y no depende de protocolos. El protocolo HTTP es el más común, pero gRPC también se usa ampliamente.

REST admite el acoplamiento bajo, pero igualmente requiere prácticas de ingeniería robustas para mantener ese acoplamiento. Un punto de partida es tener un contrato bien definido. Las implementaciones basadas en HTTP pueden usar un estándar como OpenAPI, y gRPC proporciona los búferes de protocolo. Para mantener el acoplamiento bajo, es crucial mantener la retrocompatibilidad del contrato y diseñar una API en torno a un dominio y no a casos de uso o clientes particulares. Si hace esto último, cada caso de uso o aplicación nuevos requerirán otra API de REST con propósito especial, independientemente del protocolo.

Si bien el procesamiento de solicitudes y respuestas es el caso de uso típico, es posible que también se requieran transmisiones, y estas pueden influir en la elección del protocolo. Por ejemplo, gRPC admite las transmisiones.

Los servicios de RESTful se comunican por la Web con HTTP(S)

- Los URI (o extremos) identifican recursos.
  - Las respuestas muestran una representación inmutable de la información del recurso.
- Las aplicaciones de REST proporcionan interfaces coherentes y uniformes.
  - La representación puede tener vínculos a recursos adicionales.
- El almacenamiento en caché de representaciones inmutables es adecuado.

Los URI o extremos identifican los recursos, y las respuestas a las solicitudes muestran una representación inmutable de la información del recurso.

Las aplicaciones de REST deben proporcionar interfaces coherentes y uniformes, y pueden vincular a recursos adicionales. Hipermedia como motor del estado de la aplicación (o HATEOAS) es un componente de REST que permite que el cliente requiera poco conocimiento previo de un servicio, ya que se ofrecen vínculos a recursos adicionales como parte de las respuestas.

Es importante que el diseño de API sea parte del proceso de desarrollo. Idealmente, se implementa un conjunto de reglas de diseño de API, lo que ayuda a las API de REST a proporcionar una interfaz uniforme; por ejemplo, cada servicio informa errores de forma constante, y la estructura de las URLs y el uso de las páginas son coherentes.

Además, considere el almacenamiento en caché para optimizar los recursos inmutables y mejorar el rendimiento.

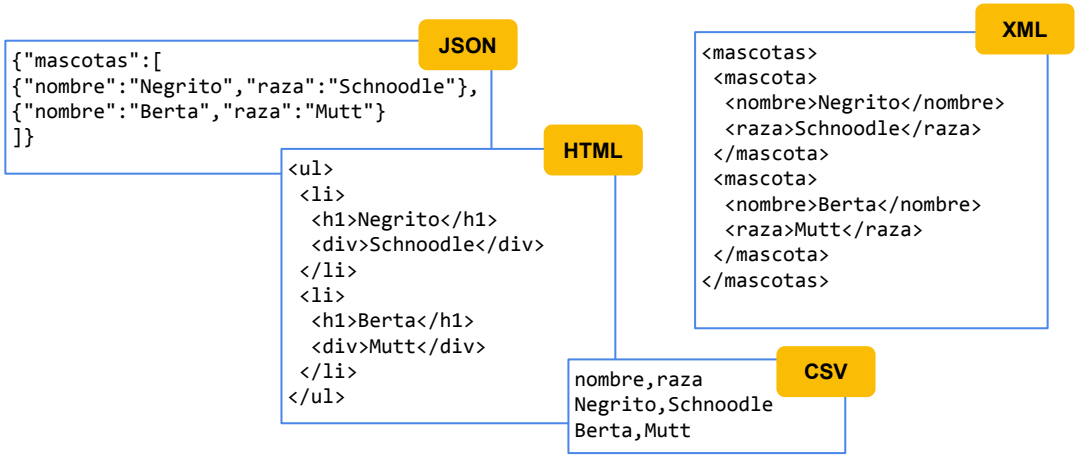
Recursos y representaciones

- Un recurso es una noción abstracta de información.
- Una representación es una copia de la información del recurso.
  - Las representaciones pueden ser elementos únicos o un conjunto de elementos.

En REST, un cliente y un servidor intercambian representaciones de un recurso. Un recurso es una noción abstracta de información. La representación de un recurso es una copia de la información de este. Por ejemplo, un recurso puede representar un perro. La representación de un recurso corresponde a los datos reales de un perro en específico; por ejemplo, Negrito, que es un schnoodle, o Berta, que es una mutt. Dos representaciones diferentes de un recurso.

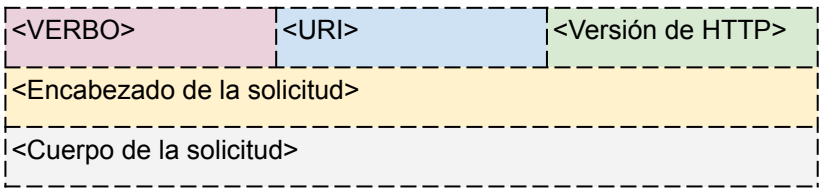
El URI brinda acceso a un recurso. Si realiza una solicitud para ese recurso, en la respuesta se muestra una representación de ese recurso, por lo general, en formato JSON. Los recursos solicitados pueden ser elementos únicos o un conjunto de elementos. Por motivos de rendimiento, puede ser beneficioso mostrar conjuntos de elementos en lugar de elementos individuales. Estos tipos de operaciones suelen llamarse *API en lotes*.

Para pasar representaciones entre servicios, se usan formatos estándar basados en texto



Las representaciones de un recurso entre un cliente y un servicio, por lo general, se logran con formatos estándar basados en texto. JSON es la norma para los formatos basados en texto, aunque se puede usar XML. En cuanto a las API orientadas al público o externas, JSON es el formato estándar. Y en lo que respecta a los servicios internos, se puede usar gRPC, especialmente si el rendimiento es clave.

Los clientes acceden a servicios mediante solicitudes HTTP



- VERBO: GET, PUT, POST o DELETE
- URI: Identificador de recursos uniforme (extremo)
- Encabezado de la solicitud: Metadatos sobre el mensaje
  - Formatos de representación preferidos (p. ej., JSON y XML)
- Cuerpo de la solicitud: Estado de la solicitud (opcional)
  - Representación (JSON o XML) del recurso

Cuando un cliente accede a servicios de HTTP, se forma una solicitud HTTP. Este tipo de solicitud se crea en tres partes: la línea de solicitud, las variables de encabezado y el cuerpo de la solicitud.

La línea de solicitud tiene el verbo HTTP (GET, POST, PUT, etc.), el URI solicitado y la versión del protocolo.

Las variables de encabezado contienen pares clave-valor. Algunas de ellas son estándar, como el usuario-agente, que ayuda al receptor a identificar el agente de software solicitante. En el caso de los servicios de REST basados en HTTPS, esta variable también incluye metadatos sobre el formato del mensaje o los formatos de mensaje preferidos. Aquí, puede agregar encabezados personalizados.

El cuerpo de la solicitud contiene datos que se enviarán al servidor, y solo es relevante para comandos HTTP que envían datos como POST y PUT.

Las solicitudes HTTP son simples y se basan en texto

```
GET / HTTP/1.1
Host: pets.drehnstrom.com
```

```
POST /add HTTP/1.1
Host: pets.drehnstrom.com
Content-Type: json
Content-Length: 35

{"nombre":"Negrito","raza":"Schnoodle"}
```

Aquí, vemos dos ejemplos de mensajes basados en texto de cliente HTTP. En el primer ejemplo, se muestra una solicitud GET HTTP a la URL / con la versión 1.1 de HTTP

Hay una variable de encabezado de la solicitud llamada Host con el valor pets.drehnstrom.com

En el segundo ejemplo, se muestra una solicitud POST HTTP a la URL /add con la versión 1.1 de HTTP

Hay tres variables de encabezado de la solicitud:
Host:
Content-Type: establecido en JSON
Content-Length: establecido en 35 bytes

Allí, se ve el cuerpo de la solicitud que tiene el documento JSON: {"nombre":"Negrito","raza":"Schnoodle"}. Esta es la representación de la mascota que se agrega.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

Como parte de una solicitud, el verbo HTTP le indica al servidor la acción que debe realizar en un recurso.

HTTP como protocolo proporciona nueve verbos, pero solo los cuatro que se indican aquí son los que suele usar REST.

- GET se usa para recuperar recursos.
- POST se usa para solicitar la creación de un recurso nuevo. El servicio luego crea el recurso y, por lo general, muestra al cliente el ID único generado para el recurso nuevo.
- PUT se usa para crear un recurso nuevo o hacer modificaciones a uno existente. Las solicitudes PUT deben ser idempotentes, lo que significa que, sin importar cuántas veces el cliente haga la solicitud a un servicio, los efectos en el recurso siempre serán exactamente los mismos.
- Finalmente, una solicitud DELETE se usa para quitar un recurso.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

Como parte de una solicitud, el verbo HTTP le indica al servidor la acción que debe realizar en un recurso.

HTTP como protocolo proporciona nueve verbos, pero solo los cuatro que se indican aquí son los que suele usar REST.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

GET se usa para recuperar recursos.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

POST se usa para solicitar la creación de un recurso nuevo. El servicio luego crea el recurso y, por lo general, muestra al cliente el ID único generado para el recurso nuevo.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

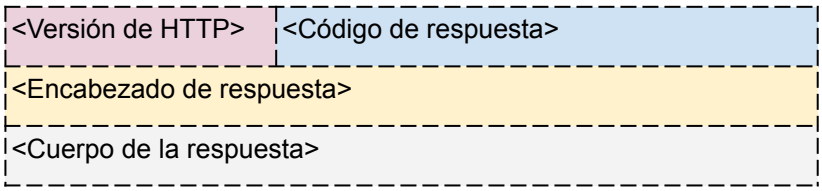
PUT se usa para crear un recurso nuevo o hacer modificaciones a uno existente. Las solicitudes PUT deben ser idempotentes, lo que significa que, sin importar cuántas veces el cliente haga la solicitud a un servicio, los efectos en el recurso siempre serán exactamente los mismos.

El verbo HTTP le indica al servidor qué hacer

- **GET** se usa para recuperar datos.
- **POST** se usa para crear datos.
  - Genera el ID de entidad y se lo muestra al cliente.
- **PUT** se usa para crear datos o alterar los existentes.
  - El ID de entidad debe ser conocido.
  - *PUT debe ser* idempotente*, lo que significa que aunque la solicitud se realice una o varias veces, los efectos en los datos serán exactamente los mismos.*
- **DELETE** se usa para quitar datos.

Finalmente, una solicitud DELETE se usa para quitar un recurso.

Los servicios muestran respuestas HTTP



- Código de respuesta: Código de estado HTTP de 3 dígitos
  - Códigos de numeración 200 para ejecuciones correctas
  - Códigos de numeración 400 para errores de clientes
  - Códigos de numeración 500 para errores de servidor
- Cuerpo de la respuesta: Contiene representación de recursos
  - JSON, XML, HTML, etcétera.

Los servicios de HTTP muestran respuestas en un formato estándar que define HTTP.
Estas respuestas HTTP se crean en tres partes: la línea de respuesta, las variables de encabezado y el cuerpo de la respuesta.

La línea de respuesta tiene la versión HTTP y un código de respuesta. Los códigos de respuesta se ven afectados en los límites cercanos a 100.

- El rango 200 indica que la solicitud tuvo éxito. Por ejemplo, 200 significa que la solicitud tuvo éxito, y 201 significa que se creó un recurso.
- El rango 400 significa que la solicitud del cliente se encuentra en un estado de error. Por ejemplo, 403 significa “prohibido debido a que el solicitante no tiene los permisos necesarios”. 404 significa “no se encontró el recurso solicitado”.
- El rango 500 significa que el servidor encontró un error y no puede procesar la solicitud. Por ejemplo, 500 significa “error interno del servidor”. 503 se refiere a “no disponible”, a menudo porque el servidor está sobrecargado.

El encabezado de respuesta es un conjunto de pares clave-valor, como tipo de contenido, que le indica al receptor el tipo de contenido que incluye el cuerpo de la respuesta.

El cuerpo de la respuesta tiene la representación de recurso solicitada en el formato especificado en el encabezado Content-Type, y puede tener formato JSON, XML,

HTML, etcétera.

Todos los servicios necesitan URI (identificadores de recurso uniformes)

- Use nombres en plural para conjuntos (colecciones).
- Use nombres en singular para recursos individuales.
- Intente tener nombres coherentes.
- Los URI no distinguen mayúsculas de minúsculas.
- No use verbos para identificar un recurso.
- Incluya información de la versión.

Los lineamientos que se enumeran aquí se enfocan en lograr coherencia en la API.

Los nombres en singular se deben usar para recursos individuales, y los nombres en plural para conjuntos o colecciones.

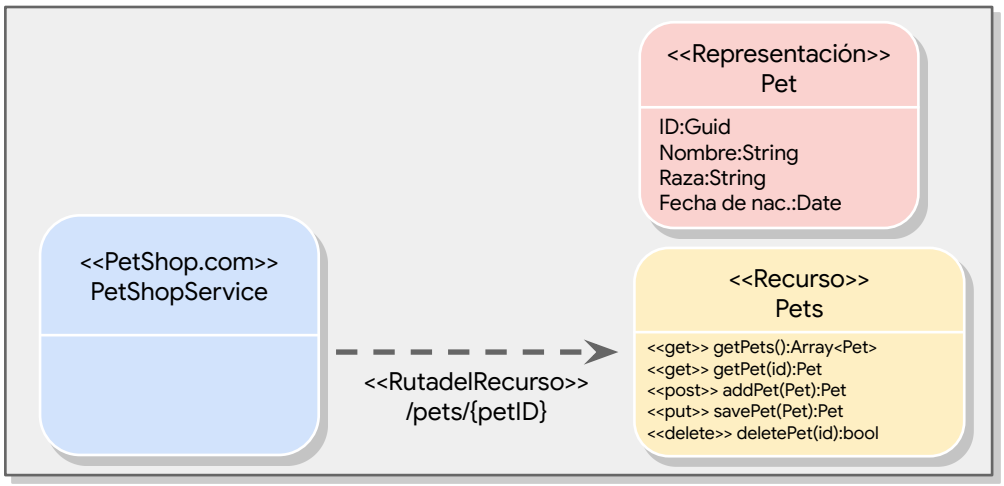
Por ejemplo, considere el siguiente URI /pet

Luego:
Una solicitud GET para el URI /pet/1 debería buscar una mascota con el ID 1, mientras que GET /pets busca todas las mascotas.

No use los URI como GET /getPets El URI debe referirse al recurso, no a la acción en el recurso (es decir, el rol del verbo).

Recuerde que los URI no distinguen mayúsculas de minúsculas y que incluyen información de la versión.

Diagrame un servicio de ejemplo



Diagramar servicios es una práctica recomendada.

En este diagrama, se muestra que hay un servicio que brinda acceso a un recurso conocido como Pets. La representación del recurso es Pet. Cuando se hace una solicitud para un recurso a través del servicio, se muestran una o más representaciones de Pet.

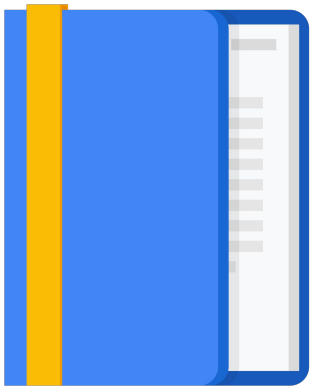
Temario

Microservicios

Prácticas recomendadas sobre microservicios

REST

API



Ahora, hablemos sobre el diseño de API.

Es importante diseñar API coherentes para servicios

- Cada servicio de Google Cloud expone una API de REST.
  - Las funciones se encuentran en el siguiente formato: `service.collection.verb`
  - Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.
- Por ejemplo, la API de Compute Engine tiene lo siguiente:
  - Un extremo de servicio en: https://compute.googleapis.com
  - Colecciones que incluyen `instances`, `instanceGroups`, `instanceTemplates`, etcétera.
  - Verbos que incluyen insert, list, get, etcétera.
- De modo que para ver todas sus instancias, debe realizar una solicitud GET al siguiente vínculo:
  `https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances`

Es importante diseñar API coherentes para servicios.

Google ofrece una guía de diseño de API con recomendaciones sobre elementos, como nombres, documentación de manejo de errores, control de versiones y compatibilidad. En las diapositivas, hay vínculos a la guía y la guía de estilo de las API.
[https://cloud.google.com/apis/design/]
[http://apistylebook.com/design/guidelines/google-api-design-guide]

Si desea consultar ejemplos de prácticas recomendadas, es útil revisar las API de Google Cloud.

Cada servicio de Google Cloud expone una API de REST. Las funciones se definen en el formato service.collection.verb. El servicio representa el extremo del servicio; p. ej., para la API de Compute Engine API, el extremo del servicio es https://compute.googleapis.com.

Las colecciones incluyen instances, instanceGroups e instanceTemplates. Los verbos incluyen LIST, INSERT y GET, por ejemplo.

Para ver todas sus instancias de Compute Engine, realice una solicitud GET al vínculo que se muestra en la diapositiva. Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.

Es importante diseñar API coherentes para servicios

- Cada servicio de Google Cloud expone una API de REST.
  - Las funciones se encuentran en el siguiente formato: `service.collection.verb`
  - Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.
- Por ejemplo, la API de Compute Engine tiene lo siguiente:
  - Un extremo de servicio en: https://compute.googleapis.com
  - Colecciones que incluyen `instances`, `instanceGroups`, `instanceTemplates`, etcétera.
  - Verbos que incluyen insert, list, get, etcétera.
- De modo que para ver todas sus instancias, debe realizar una solicitud GET al siguiente vínculo:
  `https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances`

Es importante diseñar API coherentes para servicios.

Google ofrece una guía de diseño de API con recomendaciones sobre elementos, como nombres, documentación de manejo de errores, control de versiones y compatibilidad. En las diapositivas, hay vínculos a la guía y la guía de estilo de las API.
[https://cloud.google.com/apis/design/]
[http://apistylebook.com/design/guidelines/google-api-design-guide]

Si desea consultar ejemplos de prácticas recomendadas, es útil revisar las API de Google Cloud.

Es importante diseñar API coherentes para servicios

- Cada servicio de Google Cloud expone una API de REST.
  - Las funciones se encuentran en el siguiente formato: `service.collection.verb`
  - Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.
- Por ejemplo, la API de Compute Engine tiene lo siguiente:
  - Un extremo de servicio en: https://compute.googleapis.com
  - Colecciones que incluyen `instances`, `instanceGroups`, `instanceTemplates`, etcétera.
  - Verbos que incluyen insert, list, get, etcétera.
- De modo que para ver todas sus instancias, debe realizar una solicitud GET al siguiente vínculo:
  `https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances`

Cada servicio de Google Cloud expone una API de REST.

Las funciones se definen en el formato service.collection.verb. El servicio representa el extremo del servicio;

Es importante diseñar API coherentes para servicios

- Cada servicio de Google Cloud expone una API de REST.
  - Las funciones se encuentran en el siguiente formato: `service.collection.verb`
  - Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.
- Por ejemplo, la API de Compute Engine tiene lo siguiente:
  - Un extremo de servicio en: https://compute.googleapis.com
  - Colecciones que incluyen `instances`, `instanceGroups`, `instanceTemplates`, etcétera.
  - Verbos que incluyen insert, list, get, etcétera.
- De modo que para ver todas sus instancias, debe realizar una solicitud GET al siguiente vínculo:
  `https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances`

p. ej., para la API de Compute Engine API, el extremo del servicio es https://compute.googleapis.com.

Las colecciones incluyen instances, instanceGroups e instanceTemplates. Los verbos incluyen LIST, INSERT y GET, por ejemplo.

Es importante diseñar API coherentes para servicios

- Cada servicio de Google Cloud expone una API de REST.
  - Las funciones se encuentran en el siguiente formato: `service.collection.verb`
  - Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.
- Por ejemplo, la API de Compute Engine tiene lo siguiente:
  - Un extremo de servicio en: https://compute.googleapis.com
  - Colecciones que incluyen `instances`, `instanceGroups`, `instanceTemplates`, etcétera.
  - Verbos que incluyen insert, list, get, etcétera.
- De modo que para ver todas sus instancias, debe realizar una solicitud GET al siguiente vínculo:

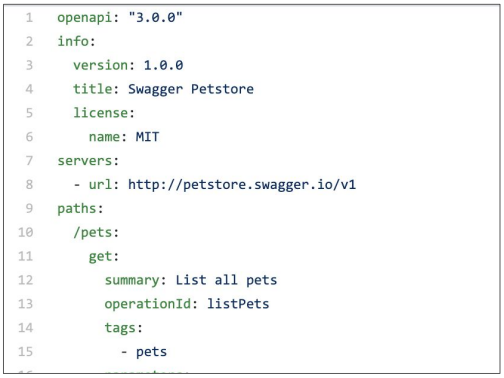
```
https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances
```

Para ver todas sus instancias de Compute Engine, realice una solicitud GET al vínculo que se muestra en la diapositiva.

Los parámetros se pasan ya sea en la URL o en el cuerpo de la solicitud en formato JSON.

OpenAPI es un estándar de la industria para exponer API a clientes

- Tiene un formato de descripción de interfaz estándar para las API de REST.
  - No depende del lenguaje.
  - Es de código abierto (basado en Swagger).
- Permite que las herramientas y las personas entiendan cómo usar un servicio sin necesitar su código fuente.



```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
```

OpenAPI es un estándar de la industria para exponer API a clientes. La versión 2.0 de la especificación se conocía como Swagger. Ahora, Swagger es un conjunto de herramientas de código abierto diseñadas en torno a OpenAPI que, con las herramientas asociadas, admite diseñar, crear, consumir y documentar API. OpenAPI admite un enfoque centrado en las API. Diseñar la API a través de OpenAPI puede proporcionar una única fuente de información a partir de la cual se pueden generar automáticamente código fuente para bibliotecas cliente y stubs de servidor, además de documentación de usuarios de API. Cloud Endpoints y Apigee admiten OpenAPI.

En el documento de ejemplo, aparece una muestra de una especificación de OpenAPI de un servicio de tienda de mascotas. El URI es http://petstore.swagger.io/v1. Observe la versión en el URI que se muestra aquí. Luego, en el ejemplo, se muestra un extremo, /pets, al que se accede con el verbo HTTP GET y el cual proporciona una lista de todas las mascotas.

gRPC es un protocolo ligero para una comunicación binaria rápida entre servicios o dispositivos

- Desarrollado en Google
  - Compatible con muchos lenguajes
  - Fácil de implementar
- Los servicios de Google admiten gRPC
  - Balanceador de cargas global (HTTP/2)
  - Cloud Endpoints
  - Posibilidad de exponer servicios de gRPC con un proxy Envoy en GKE

gRPC, que se desarrolló en Google, es un protocolo binario extremadamente útil para la comunicación interna de microservicios. Proporciona asistencia para muchos lenguajes de programación, tiene gran compatibilidad con el acoplamiento bajo a través de contratos definidos mediante búferes de protocolo y ofrece alto rendimiento por ser un protocolo binario. Se basa en HTTP/2 y admite transmisiones de clientes y servidores.

El protocolo es compatible con muchos servicios de Google Cloud, como el balanceador de cargas global y Cloud Endpoints para microservicios, además de GKE con un proxy Envoy.

Google Cloud ofrece dos herramientas para administrar API: Cloud Endpoints y Apigee

Ambas opciones proporcionan herramientas para lo siguiente:

- Autenticar usuarios
- Supervisar
- Proteger API
- Etcétera

Ambas admiten OpenAPI y gRPC

Plataforma de API de Apigee

Cloud Endpoints

Google Cloud ofrece dos herramientas para administrar API: Cloud Endpoints y Apigee.

Cloud Endpoints es una puerta de enlace de administración de API que lo ayuda a desarrollar, implementar y administrar API en cualquier backend de Google Cloud. Se ejecuta en Google Cloud y aprovecha mucha de la infraestructura subyacente de Google.

Apigee es una plataforma de administración de API diseñada para empresas, con opciones de implementación en la nube, a nivel local o de forma híbrida. El conjunto de funciones incluye un portal personalizable con puerta de enlace de API para integrar socios y desarrolladores, monetización y análisis profundos en torno a API. Puede usar Apigee para cualquier backend HTTP o HTTPS sin importar dónde se ejecutan (a nivel local, en alguna nube pública, etcétera).

Ambas soluciones proporcionan herramientas para servicios como la autenticación de usuarios, la supervisión y la protección, y también para OpenAPI y gRPC.

Actividad 5: Diseñe API de REST

Consulte el Cuaderno de ejercicios de Design and Process.

- Diseñe las API para los microservicios de su caso de éxito.

Ahora, diseñará las API de los microservicios identificados para su aplicación. El propósito de esta actividad es ganar experiencia en el diseño de API y considerar los aspectos como la estructura de URL de API, los formatos de respuesta a solicitudes de mensaje y el control de versiones.

Actividad 5: Diseñe API de REST

Consulte el Cuaderno de ejercicios de Design and Process.

- Diseñe las API para los microservicios de su caso de éxito.

Ahora, deberá realizar lo siguiente:

Actividad 5: Diseñe API de REST

Consulte el Cuaderno de ejercicios de Design and Process.

- Diseñe las API para los microservicios de su caso de éxito.

Diseñar las API de los microservicios identificados para su aplicación. El propósito de esta actividad es ganar experiencia en el diseño de API y considerar los aspectos como la estructura de URL de API, los formatos de respuesta a solicitudes de mensaje y el control de versiones.

Nombre del servicio	Colecciones	Métodos
Opinión	*opiniones*	*list* *add* *get*
Producto	*inventario*	*add* *get* *update* *Delete* *search*
IU web	*productos*	*list* *get* *review*
	cuentas	*get* *update* *delete* *add*

Supongamos que definimos una API para una tienda en línea. La API puede verse similar a lo que se muestra aquí.

Ya sea que un servicio proporcione una interfaz de usuario o alguna funcionalidad de backend, este requiere una interfaz programática a la que se puede llamar por HTTPS. La única diferencia entre un sitio web y un servicio web es el formato de los datos que se muestran. Para un servicio de IU, es posible que mostremos HTML, pero un servicio de backend podría mostrar JSON o XML.

Use los principios que abordamos en este módulo hasta ahora y consulte la actividad 5 en el cuaderno de ejercicios de diseño.

Revise la actividad 5: Diseñe API de REST

- Diseñe las API para los microservicios de su caso de éxito.

En esta actividad, se le solicitó diseñar una API de RESTful para los microservicios que se usan en su caso de éxito.

Nombre del servicio	Colecciones	Métodos
Búsqueda	Viajes (combinación de vuelos + hotel)	find save
Inventario	Artículos (vuelos + hoteles)	add search get remove
Análisis	ventas	analyze get list
Procesamiento de pedidos	pedidos	add get list update

Este es un ejemplo de nuestro portal de viajes en línea. Desde luego, nuestra API sería más grande, pero, en cierto modo, todas son similares. Cada servicio administra y pone a disposición algunos conjuntos de datos, y realizamos algunas operaciones típicas con ellos.

Lo mismo ocurre con las API de Google Cloud. Por ejemplo, en Google Cloud tenemos un servicio llamado Compute Engine que se usa para crear y administrar máquinas virtuales, redes, etc. La API de Compute Engine tiene colecciones, como instancias, grupos de instancias, redes, subredes y mucho más. En cada colección, se utilizan diferentes métodos para administrar los datos.

Recurso de REST: v1.firewalls

Métodos	
delete	`DELETE /compute/v1/projects/{project}/global/firewalls/{resourceId}` Borra el firewall especificado.
get	`GET /compute/v1/projects/{project}/global/firewalls/{resourceId}` Muestra el firewall especificado.
insert	`POST /compute/v1/projects/{project}/global/firewalls` Crea una regla de firewall en el proyecto especificado con los datos incluidos en la solicitud.
list	`GET /compute/v1/projects/{project}/global/firewalls` Recupera la lista de reglas de firewall disponibles para el proyecto especificado.
patch	`PATCH /compute/v1/projects/{project}/global/firewalls/{resourceId}` Actualiza la regla de firewall especificada con los datos incluidos en la solicitud.
update	`PUT /compute/v1/projects/{project}/global/firewalls/{resourceId}` Actualiza la regla de firewall especificada con los datos incluidos en la solicitud.

Por ejemplo, estos son métodos para agregar, administrar y borrar firewalls.

Cuando diseña sus API, debe esforzarse por ser lo más coherente posible. Así, facilitará el desarrollo, lo que también hará más sencillo para los clientes aprender a usar sus API.

Repaso

Diseño y arquitectura de microservicios

En este módulo, nos enfocamos en el diseño y la arquitectura de microservicios. Comenzamos por definir qué es una arquitectura de microservicios y las ventajas y desventajas de usar microservicios. También enumeramos algunas prácticas recomendadas sobre microservicios. Luego, abordamos cómo diseñar API de servicio y cómo implementar una arquitectura de estilo REST.

Cuestionario

Enumere algunas ventajas y desventajas de las arquitecturas de microservicios.

Enumere algunas ventajas y desventajas de las arquitecturas de microservicios.

Cuestionario

Enumere algunas ventajas y desventajas de las arquitecturas de microservicios.

Ventajas	Desventajas
Fáciles de programar y probar Escalan de forma independiente Implementaciones menos riesgosas Resulta más fácil agregar funciones nuevas Etcétera	Comunicación entre servicios Implican diversas implementaciones Latencia Control de versiones Etcétera

Entre los aspectos que podrían considerarse, se incluyen los siguientes:

Ventajas:
Escalan la organización, agregan más equipos y aceleran el desarrollo, ya que pueden funcionar de forma independiente en comparación con una aplicación monolítica, en la que, a menudo, agregar más equipos ralentiza el desarrollo.

Permita que los microservicios usen la tecnología más relevante, incluso versiones diferentes de la misma tecnología. Por ejemplo, un servicio usa Java 8 y otro usa Java 11.

Desventajas:
Dificultan la identificación de límites claros para unidades de implementación independiente.
Implican muchos más puntos de falla potenciales, por lo que es posible sufrir fallas en cascada.
Es muy necesario supervisar y observar para ayudar a diagnosticar rápidamente los errores, ya que hay muchos elementos en movimiento.

Partamos por las ventajas:
Los ejemplos incluyen que las unidades pequeñas con responsabilidades únicas son más fáciles de programar y probar. Se pueden escalar de forma independiente. Las implementaciones son menos riesgosas, ya que se aplican cambios pequeños en

lugar de grandes en aplicaciones más monolíticas. Agregar funciones nuevas también es más fácil.

También está la capacidad de escalar fácilmente la organización, agregar más equipos y acelerar el desarrollo, ya que pueden funcionar de forma independiente en comparación con una aplicación monolítica, en la que, a menudo, agregar más equipos ralentiza el desarrollo.

Además, los microservicios admiten el uso de tecnología más relevante, incluso diferentes versiones de la misma tecnología. Por ejemplo, un servicio usa Java 8 y otro usa Java 11.

Ahora, veamos las desventajas:
La comunicación entre los servicios genera sobrecarga y complejidad en comparación con una implementación monolítica, lo que puede afectar la latencia para los usuarios. El control de versiones puede ser difícil de administrar y es posible que se deban implementar muchas versiones para satisfacer diferentes clientes.

Puede resultar difícil identificar límites claros para unidades de implementación independiente.

Los canales de comunicación distribuidos proporcionan muchos más puntos de falla potenciales, por lo que es probable que haya fallas en cascada.

Existe una gran necesidad de supervisar y observar para ayudar a diagnosticar rápidamente errores debido a la complejidad de la implementación.

Cuestionario

Suponga que rediseñó una aplicación web monolítica para que el estado no se almacene en la memoria de los servidores web, sino en una base de datos. Sin embargo, esto hace que el rendimiento sea lento cuando se recuperan las sesiones de los usuarios. ¿Cuál puede ser la mejor forma de solucionar este inconveniente?

A. Transferir el estado de la sesión de vuelta a los servidores web y utilizar sesiones fijas en el balanceador de cargas

B. Usar un servicio de almacenamiento en caché, como Redis o Memorystore

C. Aumentar la cantidad de CPU en el servidor de la base de datos

D. Asegurarse de que todos los servidores web se encuentren en la misma zona que la de la base de datos

Suponga que rediseñó una aplicación web monolítica para que el estado no se almacene en la memoria de los servidores web, sino en una base de datos. Sin embargo, esto hace que el rendimiento sea lento cuando se recuperan las sesiones de los usuarios. ¿Cuál puede ser la mejor forma de solucionar este inconveniente?

A. Transferir el estado de la sesión de vuelta a los servidores web y utilizar sesiones fijas en el balanceador de cargas
B. Usar un servicio de almacenamiento en caché, como Redis o Memorystore
C. Aumentar la cantidad de CPU en el servidor de la base de datos
D. Asegurarse de que todos los servidores web se encuentren en la misma zona que la de la base de datos

Cuestionario

Suponga que rediseñó una aplicación web monolítica para que el estado no se almacene en la memoria de los servidores web, sino en una base de datos. Sin embargo, esto hace que el rendimiento sea lento cuando se recuperan las sesiones de los usuarios. ¿Cuál puede ser la mejor forma de solucionar este inconveniente?

A. Transferir el estado de la sesión de vuelta a los servidores web y utilizar sesiones fijas en el balanceador de cargas

B. Usar un servicio de almacenamiento en caché, como Redis o Memorystore

C. Aumentar la cantidad de CPU en el servidor de la base de datos

D. Asegurarse de que todos los servidores web se encuentren en la misma zona que la de la base de datos

A. La respuesta no es correcta. Se necesitan servicios sin estado para permitir la escalabilidad y la alta disponibilidad.

B. Esta respuesta es correcta. Se necesitan servicios sin estado, y un servicio como Redis o Memorystore proporciona un servicio de almacenamiento en caché rápido para almacenar estados. Esto permite usar servicios sin estado y hace posibles el escalamiento y la alta disponibilidad.

C. La respuesta no es correcta. No hay indicios sobre dónde se origina el problema. Podría ser la latencia de la red, pero la solución debería ser escalar horizontalmente, no de forma vertical.

D. Esta respuesta no es correcta. Cuando los recursos están en la misma zona, se genera un impacto potencial en la disponibilidad, pero también se ve afectada la experiencia del usuario con posibles demoras innecesarias para acceder al servicio.

Cuestionario

¿Cuál de las siguientes opciones ***infringe*** las prácticas recomendadas de una aplicación de 12 factores?

A. Almacenar la información de configuración en su repositorio de código fuente para un control de versiones sencillo

B. Tratar los registros como transmisiones de eventos y agregarlos en una única fuente

C. Hacer que el desarrollo, las pruebas y la producción sean tan similares como sea posible

D. Declarar y aislar dependencias de forma explícita

¿Cuál de las siguientes opciones ***infringe*** las prácticas recomendadas de una aplicación de 12 factores?

A. Almacenar la información de configuración en su repositorio de código fuente para un control de versiones sencillo

B. Tratar los registros como transmisiones de eventos y agregarlos en una única fuente

C. Hacer que el desarrollo, las pruebas y la producción sean lo más similares posibles

D. Declarar y aislar dependencias de forma explícita

Cuestionario

¿Cuál de las siguientes opciones ***infringe*** las prácticas recomendadas de una aplicación de 12 factores?

A. Almacenar la información de configuración en su repositorio de código fuente para un control de versiones sencillo

B. Tratar los registros como transmisiones de eventos y agregarlos en una única fuente

C. Hacer que el desarrollo, las pruebas y la producción sean tan similares como sea posible

D. Declarar y aislar dependencias de forma explícita

A es la respuesta correcta. El código y la configuración deberían separarse, ya que la configuración cambia de una implementación a otra, pero el código no varía. La verdadera prueba es si el repositorio se puede configurar con código abierto sin poner en riesgo ninguna credencial.

Las respuestas B, C y D se enumeran explícitamente como factores en https://12factor.net/ y, por lo tanto, no son correctas.

Cuestionario

Suponga que está escribiendo un servicio y necesita manejar un cliente que le envía datos no válidos en la solicitud. ¿Qué respuesta debería mostrar el servicio?

A. Una excepción XML

B. Un código de error 200

C. Un código de error 400

D. Un código de error 500

Suponga que está escribiendo un servicio y necesita manejar un cliente que le envía datos no válidos en la solicitud. ¿Qué respuesta debería mostrar el servicio?
A. Una excepción XML
B. Un código de error 200
C. Un código de error 400
D. Un código de error 500

Cuestionario

Suponga que está escribiendo un servicio y necesita manejar un cliente que le envía datos no válidos en la solicitud. ¿Qué respuesta debería mostrar el servicio?

A. Una excepción XML

B. Un código de error 200

C. Un código de error 400

D. Un código de error 500

A. Esta respuesta no es correcta. Con REST, las excepciones o los errores se deben informar mediante códigos de error, no excepciones.

B. Esta respuesta no es correcta. El código de estado 200 indica que todo está bien.

C. Esta respuesta es correcta. El código de estado HTTP 400 indica que una solicitud no se pudo procesar debido a un error aparente del cliente.

D. Esta respuesta no es correcta. El código de estado HTTP 500 indica un error interno del servidor.

Cuestionario

Suponga que está creando un microservicio de RESTful. ¿Cuál sería un formato de datos válido para mostrar los datos al cliente?

A. JSON

B. XML

C. HTML

D. Todas las opciones anteriores

Suponga que está creando un microservicio de RESTful. ¿Cuál sería un formato de datos válido para mostrar los datos al cliente?
A. JSON
B. XML
C. HTML
D. Todas las opciones anteriores

Cuestionario

Suponga que está creando un microservicio de RESTful. ¿Cuál sería un formato de datos válido para mostrar los datos al cliente?

A. JSON

B. XML

C. HTML

D. Todas las opciones anteriores

Todas las opciones anteriores son correctas. Tienen un tipo de contenido estándar que puede configurarse en el encabezado de respuesta, y se basan en texto. Se suele usar JSON, pero XML también es válido.

Más recursos

Guía de diseño de API

https://cloud.google.com/apis/design/

Autentique llamadas de servicio a servicio con Google Cloud Endpoints

https://youtu.be/4PgX3yBJEyw

Estos dos recursos son útiles para consultar sobre el diseño y la protección de API.