



[New Contributor Contest] Gana un iPad Mini, un Amazon Ech...

[Leer los detalles del concurso](#) ▶

# ¿Qué son los microservicios?

por **Justin Albano** MVB · 07 de febrero, 18 · Zona de microservicios

Aprenda a realizar un seguimiento de los microservicios implementados en infraestructura elástica, como contenedores o nubes, donde los nodos aumentan y disminuyen rápidamente.

Los microservicios han sido una parte importante de la comunidad de software desde 2011, pero al igual que con muchas de las otras filosofías de arquitectura y diseño, ha habido un hype puntuado que rodea este estilo arquitectónico desde sus inicios. Al igual que con muchos de estos hypes, ha habido una tendencia a convertir todo el software existente o exigir que todo el nuevo software se implemente utilizando este estilo. En respuesta, muchos han descartado este estilo como pura exageración superficial, esperando cínicamente que su importancia se destrone de la misma manera que las arquitecturas orientadas a servicios (SOA) y los protocolos de comunicación orientados a objetos multiplataforma (es decir, Common Object Request Broker Architecture, CORBA).

En algún lugar entre la exageración y el cinismo, hay ventajas importantes que los microservicios aportan a la mesa, no solo en términos de su reutilización y modularidad, sino también en términos de las herramientas y tecnologías de software que facilita en la práctica. De hecho, este estilo de arquitectura ha sido adoptado por muchos de los desarrolladores de aplicaciones empresariales más grandes, incluidos Amazon , Uber , Groupon , Capital One , Walmart y Netflix (que representaron el 35,2% de todo el tráfico de Internet de descarga en América del Norte en 2016

de Internet de descarga en América del Norte en 2010 ). Con una adopción tan monumental, cualquier ingeniero de software y técnico serio se beneficia al prestar atención a este enfoque práctico.

En este artículo, exploraremos los fundamentos de los microservicios, incluida una definición general que captura muchas de las características de los microservicios, una breve historia de cómo surgieron los microservicios y algunas de las ventajas y desventajas que presentan los microservicios. Aunque los microservicios son un tema extenso, tanto en términos de profundidad como de amplitud, nos centraremos en los principios generales de la arquitectura, en lugar de en las tecnologías de implementación específicas. También exploraremos algunos de los recursos más populares y prolíficos que el lector interesado puede utilizar para tomar la información de este resumen y ponerla en práctica.

## Una definición simplificada

Por lo general, buscamos una definición clara y nítida para las diferentes filosofías y arquitecturas, pero en el caso de los microservicios, no existe una definición universalmente aceptada. Un intento confiable fue realizado por Microservices.io (creado por Chris Richardson ), que proporciona la siguiente definición:

---

**Microservicios, también conocida como arquitectura de microservicio, es un estilo arquitectónico que estructura una aplicación como una colección de servicios débilmente acoplados, que implementan capacidades comerciales. La arquitectura de microservicio permite la entrega / implementación continua de aplicaciones grandes y complejas. También permite a una organización evolucionar su stack de tecnología.**

---

James Lewis y Martin Fowler proporcionan su propia definición en la misma línea:

---

**El estilo arquitectónico de microservicio es un enfoque para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos livianos, a menudo una API de recursos HTTP. Estos servicios se basan en las capacidades empresariales y se pueden implementar de forma independiente mediante una maquinaria de implementación completamente automatizada. Existe un mínimo de administración centralizada de estos servicios, que puede escribirse en diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos.**

---

Si bien estas dos definiciones pueden parecer vagas para algunos, hacen bien en capturar la esencia de una arquitectura de microservicios. En general, una arquitectura de microservicios implica la creación de servicios individuales que cumplen un objetivo empresarial establecido y están vinculados con interconexiones minimalistas, como las interfaces de programación de aplicaciones (API) de transferencia de estado representacional (REST). Si bien la verdadera definición de servicio individual puede variar entre las aplicaciones, la idea general es que cada servicio debe resolver un objetivo comercial y contar con otros servicios para cumplir objetivos fuera de su alcance.

Por ejemplo, supongamos que quisiéramos crear una librería en línea que permita a los usuarios registrados enviar pedidos, verificar y actualizar el inventario, y enviar libros utilizando el método menos costoso (si los libros están disponibles). Se debe poder acceder a esta aplicación usando una aplicación móvil, una aplicación de una sola página (SPA) del lado del cliente (ejecutada en un navegador) y directamente a través de una interfaz REST, con las interfaces móviles y SPA confiando en la API REST. Usando esta descripción, podemos dividir nuestro sistema en tres servicios principales, junto con sus respectivas responsabilidades:

1. **Servicio de usuario** : administra los usuarios registrados
2. **Servicio de inventario** : administra el stock de libro actualmente disponible
3. **Servicio de envío** : se encarga de encontrar el servicio de envío menos costoso y enviar una orden

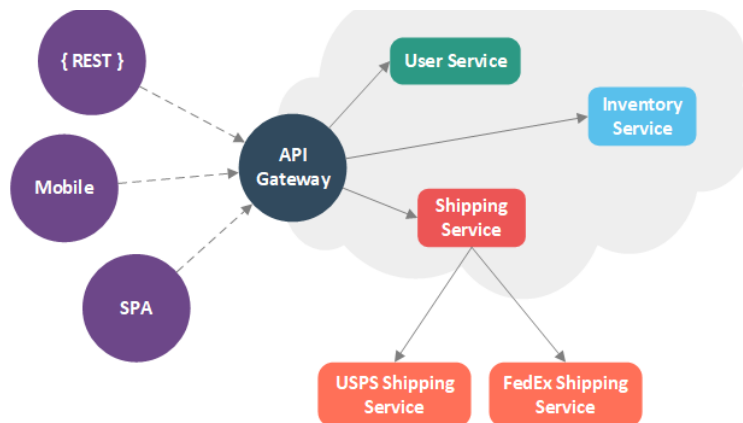
Del mismo modo, podemos clasificar tres interfaces principales en nuestro sistema:

1. **Aplicación móvil** : Consume la interfaz REST a través de una API Gateway / proxy
2. **SPA basado en navegador** : consume la interfaz REST a través de una API Gateway / proxy
3. **Interfaz REST** : una **interfaz REST de Hypertext Markup Language (HTTP)** que consume y produce recursos de JavaScript Object Notation (JSON); para nuestra implementación de microservicios, cada servicio tendrá su propia interfaz REST y se usará una puerta de enlace API para componer estas API individuales en una API REST aparentemente singular.

Combinando todas estas divisiones e interfaces en una sola aplicación, podemos diseñar el siguiente diagrama de sistema:

Production Environment





## La forma monolítica

Si se tratara de un sistema simple, podemos combinar cada uno de nuestros servicios en una sola aplicación y crear interfaces de software para facilitar las interacciones entre los servicios. A su vez, creamos una única interfaz REST que permite que las solicitudes sean manejadas por cada uno de los servicios. Del mismo modo, utilizaríamos un almacén de datos compartido. En esencia, crearíamos una aplicación nominal de tres niveles, con una interfaz REST en la parte superior, el dominio y la lógica de negocios (para usuarios, inventario y envío combinados) en el medio, y una capa de datos (que interactúa con nuestros datos compartidos) tienda y cada uno de los servicios de envío externos) en la parte inferior.

## El camino de Microservicios

En una aplicación de escala más realista, podemos beneficiarnos al dividir cada uno de los servicios en sus propias aplicaciones, con su propio espacio de proceso. Por ejemplo, podemos crear un servicio de usuario separado que tenga su propio almacén de datos, su propio negocio y lógica de dominio, y su propia interfaz REST. Asimismo, podemos repetir el mismo proceso para los servicios de inventario y envío. Esto resulta en servicios independientes y altamente cohesivos, como se ilustra en el diagrama del sistema anterior.

Esta división nos permite crear servicios en miniatura (o micro) bien encapsulados que hacen una cosa muy bien. Como cada uno tiene su propio espacio de proceso, también podemos implementarlos de forma independiente. Esto nos permite realizar cambios en un solo microservicio sin afectar a los demás. Esto

también nos permite escalar porciones del sistema que se sobrecargan. Por ejemplo, si tuviéramos que crear una sola aplicación, nos veríamos obligados a implementar nuevas instancias de todas las aplicaciones detrás de un equilibrador de carga para compartir la carga de trabajo. Si solo el servicio de envío está sobrecargado, nos vemos obligados a crear nuevas instancias del usuario y los servicios de inventario también, pero estas nuevas instancias probablemente no se utilizarán (o compartirán una carga que una sola instancia podría haber manejado sin la sobrecarga de carga). equilibrio).

En el enfoque de microservicios, podemos escalar la parte de la aplicación (el servicio de envío en este caso) que requiere más recursos. En esencia, los microservicios nos proporcionan un mayor nivel de granularidad en el despliegue, pero esta fineza tiene algunas desventajas. El principal entre ellos es la necesidad de una orquestación más matizada. Por ejemplo, si hubiéramos combinado nuestro sistema en una sola aplicación, habríamos compartido un espacio de proceso y las llamadas entre servicios serían tan triviales como hacer una llamada a otro objeto en la memoria. Con los microservicios, ahora somos responsables de garantizar que los mensajes entre servicios se mapeen correctamente, lo que permite que cada uno de los servicios se comuniquen entre sí a través de sus respectivas API REST.

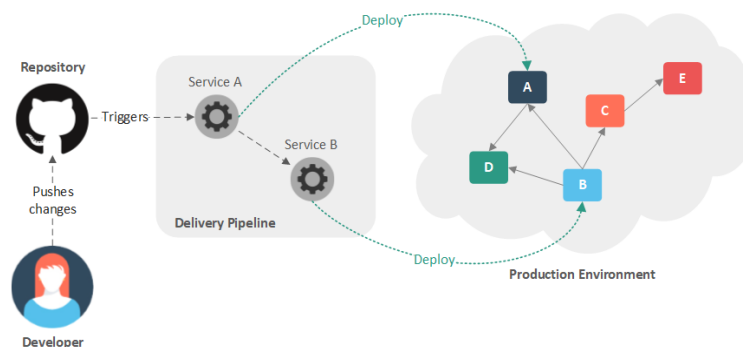
Si bien la orquestación puede ser una preocupación para los microservicios, las interconexiones entre los servicios deben ser tan simples como sea factible y constituir la capacidad suficiente para facilitar la comunicación mínima entre los servicios. A diferencia de los días de SOA con Enterprise Service Buses (ESB), la *inteligencia* en la arquitectura de microservicios está contenida dentro de los servicios, en lugar de dentro de las conexiones entre los servicios.

## El entorno ideal de microservicio

La flexibilidad de implementación y escalado no solo nos permite tomar decisiones más precisas sobre cómo se implementa una aplicación en el tiempo de ejecución, sino que también nos permite introducir

ejecución, sino que también nos permite introducir un alto grado de automatización en nuestro sistema de producción. Por ejemplo, teóricamente, podemos crear una canalización de entrega para cada uno de nuestros servicios, creando relaciones entre las canalizaciones que reflejan las relaciones de dependencia entre los servicios. Cuando se realiza un cambio en la base de código para un servicio, se inicia el proceso de entrega para ese servicio. A medida que se ejecuta la interconexión para el servicio alterado, todas las interconexiones dependientes se ejecutan posteriormente.

Una vez que el servicio ha sido creado y probado (todas las etapas de su pipeline han pasado), el servicio, junto con sus dependencias (cuyas tuberías también pasaron todas las etapas) se empacan en contenedores y se implementan en el entorno de producción. Este entorno idealista se ilustra en la figura a continuación.



Además del entorno de despliegue descrito anteriormente, se puede establecer una configuración similar para hacer girar nuevas instancias de microservicios y escalar automáticamente para satisfacer las necesidades del entorno de producción. Por ejemplo, una vez finalizada la canalización de un servicio, el servicio se contiene y se almacena en un repositorio. Si se necesitan más instancias del servicio, como durante los momentos de mucho estrés o de gran tráfico del día, se pueden implementar automáticamente más instancias del microservicio en contenedor para satisfacer estas necesidades inmediatas. Cuando disminuyó la necesidad inmediata de más instancias de servicio, los contenedores se pueden eliminar y el uso de recursos del sistema vuelve a sus niveles normales.

Si bien la combinación de tecnologías de distribución de entrega es un gran objetivo, las tecnologías en

de entrega es un gran objetivo, las tecnologías en contenedores y la filosofía de los microservicios, los matices de las aplicaciones del mundo real hacen que este sistema ideal sea difícil de implementar. Por ejemplo, supongamos que una aplicación consiste en veinte microservicios, todos con docenas de sus propias dependencias e interfaces requeridas. A medida que crece el tamaño del sistema, la orquestación de una gran cantidad de microservicios puede ser prohibitivamente difícil.

En aplicaciones pragmáticas, la automatización completa (desde el punto de que los desarrolladores revisen el código hasta un microservicio actualizado implementado y escalado en un entorno de producción) puede no ser posible, pero incluso implementaciones parciales de estos conceptos pueden reducir enormemente la fragilidad de un sistema de producción.

## Una definición simplificada revisitada

Con una comprensión del proceso de pensamiento detrás de los microservicios y algunas de las ventajas que presentan, podemos crear una definición simplificada para una arquitectura de microservicios:

---

**Una arquitectura de microservicios es aquella en la que los objetivos comerciales de un sistema se dividen en servicios independientes, encapsulados y desacoplados que interactúan entre sí a través de interfaces remotas bien definidas.**

---

Si bien esta definición puede carecer de algunos de los detalles y la especificidad de otras definiciones, proporciona una línea de base para comprender los microservicios. La restricción de la independencia y la encapsulación da como resultado servicios programados defensivamente que son capaces de



hacer frente a la falla, actualización o escalado con un impacto mínimo. En la práctica, la interfaz remota bien definida de los microservicios será algún tipo de tecnología basada en web, como WebSockets , Protocol Buffers o HTTP REST, aunque cualquier tecnología que pueda comunicarse entre espacios de proceso será suficiente.

Para obtener una comprensión más profunda de los microservicios, debemos explorar cómo surgieron los microservicios, específicamente qué tecnologías y filosofías suplantó la arquitectura de microservicios a medida que crecía y maduraba.

## La evolución de las aplicaciones empresariales

La modularidad y la componente han sido durante mucho tiempo una parte de la ingeniería de software, pero los microservicios han sido un concepto relativamente reciente que se deriva de una larga historia de arquitecturas. Si bien muchos de los conceptos subyacentes han existido desde la infancia de la ingeniería de software, la idea de crear una serie en red de servicios en miniatura con fines especiales tuvo su inicio en la más básica de todas las arquitecturas: la arquitectura monolítica.

### Arquitectura monolítica

Un monolito es una única estructura de piedra maciza y, desde los albores del desarrollo de software, la mayoría de los sistemas tienen un extraño parecido con estos edificios. En la mayoría de los softwares, simplemente unimos todo nuestro código en un único y gigantesco proyecto, con toda la lógica de negocios, interfaces, frameworks y data stores en una sola aplicación. Si bien esto puede parecer una forma pobre de estructurar una aplicación, es muy efectiva para una amplia gama de casos de uso.

Por ejemplo, los microservicios de usuario, inventario y envío pueden considerarse como monolitos. Visto en el panorama general de la aplicación de librería, son microservicios simples, pero si nos centramos solo en uno de estos servicios, se trata de una

aplicación singular de tres niveles con una interfaz de usuario (REST API), lógica de negocios y data store. .

Si bien los monolitos son suficientes para aplicaciones a pequeña escala o pequeños servicios, son totalmente inadecuados para la escala masiva de las aplicaciones que se desarrollan y despliegan en la actualidad. Para aplicaciones como Netflix, que procesa más de 400 mil millones de eventos al día , una arquitectura demasiado simplificada es deficiente tanto en su desarrollo como en su implementación. En los primeros días de la era de Internet, se concibió un nuevo tipo de arquitectura para resolver este enigma: SOA.

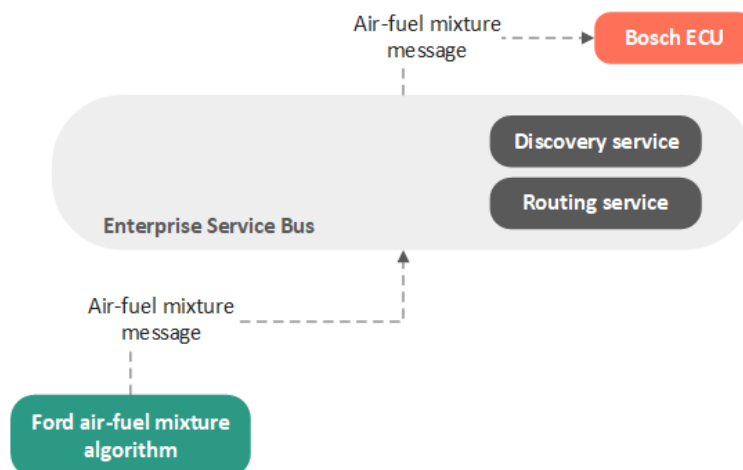
## Arquitectura orientada a Servicios

A diferencia de una arquitectura monolítica, SOA es una arquitectura mucho más estandarizada, casi hasta la avería. En lugar de tener una sola aplicación que maneje todos los tipos de servicios en un sistema, una SOA delinea cada una de las partes principales de un sistema en servicios con interfaces muy precisas. Más que solo una colección de servicios, SOA también incluye una gran pieza central: el ESB.

El objetivo detrás de muchas implementaciones de SOA es crear un conjunto estandarizado de servicios (que consumen y producen datos) y adjuntarlos a un ESB singular. El ESB es responsable de vincular a productores y consumidores y permitir que los servicios se descubran entre sí. Cabe señalar que la interfaz para cada servicio o clase de servicios está altamente estandarizada y es probable que sea contractual.

Por ejemplo, podemos crear un SOA para un automóvil creando un ESB centralizado que permita conectar diferentes dispositivos (servicios). Luego podemos agregar contratos para los dispositivos deseados, como una Unidad de control del motor (ECU). Para adjuntar una ECU al ESB, implementamos un servicio que se ajusta completamente a la interfaz contractual de una ECU y se registra con la ESB. Cuando otros servicios en el autobús desean realizar cambios en el motor (como cambiar la mezcla de aire y combustible), envían un

mensaje al ESB. El ESB luego reconoce que nuestro servicio ECU es capaz de realizar esta acción y enruta el mensaje a nuestro servicio ECU. Este esquema se ilustra en la figura a continuación.



Es importante tener en cuenta que el servicio que envió el mensaje de actualización de la mezcla de aire y combustible no sabe nada sobre nuestro servicio ECU y puede que ni siquiera haya sabido que nuestro servicio ECU se registró en el bus. En cambio, simplemente produjo un mensaje e instruyó al ESB para encontrar un consumidor para el mensaje. El ESB, que conocía nuestro servicio de ECU y reconoció que era un consumidor de mensajes de actualización de mezcla de aire y combustible (esto se descubrió cuando se registró como una implementación del contrato de interfaz de ECU), envió el mensaje a la ECU. Es importante tener en cuenta que la mayor parte del enrutamiento y la toma de decisiones de los mensajes tiene lugar en el ESB: en general, la *inteligencia* de la aplicación está contenida en el ESB.

En teoría, este enfoque permitiría implementaciones de servicios completamente ortogonales para conectarse al mismo bus e interoperar. Por ejemplo, podríamos crear una implementación de ESB dentro de un vehículo Ford que permitiera que un algoritmo de Ford enviara mensajes de actualización de la mezcla de aire y combustible al autobús y hacer que una ECU de Bosch realice los cambios necesarios en el motor. La belleza de esta arquitectura es que el algoritmo de Ford no tiene ningún concepto de la ECU de Bosch, solo que hay un suscriptor de los mensajes de mezcla de aire y combustible que produce.

Si bien esta idea era teóricamente sólida, requería un

conjunto prohibitivamente estricto de interfaces de contrato y crear un ESB muy complejo. En la práctica, esto significó numerosos y complejos contratos de lenguaje de descriptores de servicios web (WSDL) para aplicaciones en línea y un sinnúmero de contratos de interfaz propietarios para aplicaciones localizadas. Si bien algunas implementaciones de SOA fueron fructíferas, en general, se gastaron millones de dólares para crear ESB de SOA en vano. Lo que se necesitaba era un compromiso entre los monolitos y los ESB: pequeños servicios con interfaces de comunicación simples.

## Los microservicios son diferentes

La arquitectura de microservicios es similar a SOA, pero tiene algunas diferencias importantes. Al igual que SOA, la arquitectura de microservicio se compone de servicios encapsulados, pero los contratos de interfaz son mucho más relajados. En lugar de descripciones de contrato propietarias o complicadas, los microservicios utilizan interfaces minimalistas que utilizan REST a través de HTTP con cuerpos de solicitud y respuesta JSON predefinidos (u otros formatos de transferencia de información, como lenguaje de marcado extensible, XML). Esto no solo elimina la necesidad de contratos estrictos, sino que también elimina la necesidad del ESB.

Con las arquitecturas de microservicios, la inteligencia de la arquitectura está en las hojas y no en las ramas: los servicios contienen la lógica comercial y realizan solicitudes directamente a otros servicios (o a través de un proxy o una puerta de enlace) en lugar de utilizar enrutamiento o descubrimiento centralizado. En esencia, los microservicios son el equilibrio entre las oscilaciones de péndulo de arquitecturas monolíticas y SOA.

## Ventajas desventajas

Al igual que cualquier otra filosofía o enfoque en ingeniería de software, los microservicios están destinados a resolver problemas dentro de un contexto específico. Como tal, los microservicios tienen su parte justa de ventajas y desventajas.

## Ventajas

Algunas de las principales ventajas de usar una arquitectura de microservicios incluyen:

- Los servicios son relativamente pequeños y autónomos (pequeños en relación con el tamaño de una aplicación monolítica equivalente).
- Un cambio en un servicio no necesariamente provoca un cambio en otros servicios.
- Los servicios se pueden implementar y escalar independientemente de otros servicios, o incluso implementarse en contenedores separados utilizando una tecnología de contenedor, como Docker ; los conductos de compilación y entrega se pueden usar para automatizar la implementación, con un cambio en un servicio que causa una compilación, prueba e implementación automatizadas del servicio, junto con otros servicios que dependen de él.
- Los servicios se pueden desarrollar de forma independiente, posiblemente con idiomas diferentes y pilas de tecnología, diferentes desarrolladores o incluso con diferentes organizaciones.
- Partes del sistema pueden intercambiarse en caliente sin descargar toda la aplicación
- Una falla en un servicio en el sistema no necesariamente detiene toda la aplicación (los límites entre las partes del sistema están más definidos y cada parte está más aislada en una arquitectura de microservicio que en una aplicación monolítica).
- Probar los servicios de forma aislada es más fácil.

## Desventajas

Si bien los microservicios son efectivos para algunas aplicaciones, también tienen algunas desventajas, que incluyen:

- Los servicios pueden contener lógica duplicada o el esfuerzo de desarrollo puede duplicarse entre múltiples servicios.

- Se requiere una mayor orquestación.
- Probar la totalidad del sistema es más difícil debido a la necesidad de rutas de red y otras vías de comunicación entre servicios.
- Dividir los servicios limpiamente puede ser difícil (o, como lo dice SoapUI Pro : "Partir la aplicación en microservicios es en gran medida un arte").
- Las transacciones que encapsulan adecuadamente entre servicios pueden ser difíciles; por ejemplo, creamos un ID de pedido en el servicio de inventario y hacemos que cada artículo se agregue al pedido como mensajes separados (usando el ID del pedido) o esperamos un mensaje que contenga una lista completa de artículos en un pedido que se enviará a el servicio de inventario?
- Se pueden requerir más recursos, como memoria, cuando se implementan microservicios (en comparación con una aplicación monolítica equivalente).
- Las conexiones entre servicios a menudo son más complejas que los monolitos (no en términos de la complejidad de las conexiones individuales, como con SOA, sino en términos del número de conexiones entre los diversos microservicios).

## Más información

Aunque no hemos analizado los detalles de la creación de un microservicio a nivel de código, o cómo migrar una aplicación existente a una arquitectura de microservicio, hay muchos recursos valiosos disponibles sobre estos temas. Si bien la siguiente lista no es exhaustiva, contiene algunos de los libros y artículos más reconocidos y citados sobre microservicios y ayudará a comprender los microservicios a nivel práctico:

- Microservicios de James Lewis y Martin Fowler : James Lewis y Martin Fowler toman microservicios. Aunque han aparecido muchas nuevas tecnologías y marcos de microservicios

desde su publicación a principios de 2014, este artículo sigue capturando las principales características de la arquitectura de microservicios. Es un excelente recurso para desarrolladores de microservicios nuevos y experimentados y resume vívidamente los principios rectores de los microservicios sin enmascarar su visión general con tecnologías específicas y proyectos de código abierto.

- Building Microservices por Sam Newman : El libro de *facto* sobre microservicios que contiene una guía útil sobre los principios de los microservicios. Este libro no pretende enseñar a su lector cómo implementar microservicios en detalle (ya que depende de la aplicación individual y el dominio del problema), sino que arroja algo de luz sobre las técnicas utilizadas por los diseñadores y arquitectos de microservicios para crear soluciones prácticas en entornos de producción.
- Guía de microservicios de Martin Fowler : un compendio para cualquier diseñador de microservicios. Esta página contiene una gran variedad de información de microservicios, que incluye grandes citas, conferencias, presentaciones, entrevistas y una variedad de otros materiales de aprendizaje variados.
- Primeros pasos con Microservicios por Vineet Reynolds y Arun Gupta : la Ref. DZone oficial en microservicios. Esta Refcard es ideal para cualquier desarrollador o arquitecto que busque obtener una visión general rápida y concisa del mundo de los microservicios (incluida la migración de monolitos en microservicios y los patrones comunes de microservicios) sin verse inundado de detalles.
- The Netflix TechBlog : el blog técnico oficial para la comunidad de ingeniería de software de Netflix. Siendo que Netflix es uno de los principales post-hijos para microservicios a gran escala, Netflix TechBlog tiene numerosos artículos escritos por ingenieros de Netflix que principalmente comparten sus experiencias de primera mano con la implementación de microservicios para posiblemente la aplicación empresarial más grande jamás desarrollada

- **Microservicios.io** : Un compendio creado por Chris Richardson que incluye catálogos de patrones de microservicios, artículos y presentaciones en microservicios, y otros recursos de aprendizaje para ayudar a los desarrolladores a despegar.

---

Maximice su velocidad de desarrollo y profundice en las métricas de Docker para obtener información inmediata sobre su contenedor.

---

Temas: MICROSERVICIOS, MONOLITO,  
ARQUITECTURA DE SOFTWARE,  
ARQUITECTURA ORIENTADA A SERVICIOS

Las opiniones expresadas por los contribuidores de DZone son suyas.

## Recursos para socios de microservicios

Conozca los beneficios y principios de la arquitectura de microservicios

CA Technologies



EBook de Arquitectura de microservicios: descargue su copia aquí

CA Technologies



## Software Design Principles

by Hussein Terek MVB · Feb 05, 18 · Java Zone

Just released, a free O'Reilly book on Reactive Microsystems: The Evolution of Microservices at Scale. Brought to you in partnership with Lightbend.

---

Software design has always been the most important phase in the development cycle. The more time you put into designing a resilient and flexible architecture, the more time will save in the future when changes



the more time will save in the future when changes arise.

Requirements always change — software will become legacy if no features are added or maintained on regular basis — and the cost of these changes are determined based on the structure and architecture of the system. In this article, we'll discuss the key design principles that help in creating easily maintainable and extendable software.

## A Practical Scenario

Suppose that your boss asks you to create an application that converts Word documents to PDFs. The task looks simple — all you have to do is to look up a reliable library that converts Word documents to PDFs and plug it in inside your application. After doing some research, say you ended up using the *Aspose.words* framework and created the following class:

```

1  /**
2   * A utility class which converts a word document to PDF
3   *
4   * @author Hussein
5   */
6  public class PDFConverter {
7
8      /**
9       * This method accepts as input the document bytes and
10      * returns the converted one.
11      * @param fileBytes
12      * @throws Exception
13      */
14      public byte[] convertToPDF(byte[] fileBytes) throws Exception {
15
16          // We're sure that the input is always a valid Word document
17          // using the Aspose.words framework and do the conversion
18
19          // Create a new document
20          com.aspose.words.Document wordDocument = new com.aspose.words.Document();
21          wordDocument.save(pdfDocument, SaveFormat.Pdf);
22      }
23  }

```

```

        wordDocument.save(pdfDocument, SaveForm
21
        return pdfDocument.toByteArray();
22
    }
23
24 }

```

Life is easy and everything is going pretty well!

## Requirements Change, as Always

After a few months, some client asks you to support Excel documents as well. So you did some research and decided to use *Aspose.cells*. Then, you go back to your class, add a new field called *documentType*, and modify your method like the following:

```

1  public class PDFConverter {
    // we didn't mess with the existing func
2
    // the class will still convert WORD to PDF
3
    // this field to EXCEL.
4    public String documentType = "WORD";
5
6
7    /**
    * This method accepts as input the documen
8
    * returns the converted one.
9    * @param fileBytes
10   * @throws Exception
11   */
12   public byte[] convertToPDF(byte[] fileBytes
13
14       if (documentType.equalsIgnoreCase("WORD
15
16           InputStream input = new ByteArrayIr
17
18           com.aspose.words.Document wordDocun
19
20           ByteArrayOutputStream pdfDocument =
21
22           wordDocument.save(pdfDocument, Save
23
24           return pdfDocument.toByteArray();
25
26       } else {
27           InputStream input = new ByteArrayIr
28
29           Workbook workbook = new Workbook(ir

```

```
22 PdfSaveOptions saveOptions = new PdfSaveOptions();
23 saveOptions.setCompliance(PdfCompliance.PDF_A_1_4);
24 ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
25 workbook.save(pdfDocument, saveOptions);
26 return pdfDocument.toByteArray();
27 }
28 }
29 }
30 }
```

This code will work perfectly for the new client (and will still work as expected for the existing clients), but some bad design smells are starting to appear in the code. That means we're not doing this the perfect way, and we will not be able to modify our class easily when a new document type is requested.

1. **Code repetition:** As you see, similar code is being repeated inside the if/else block, and if we managed, someday, to support different extensions, then we will have a lot of repetitions. Also, if we decided later on, for example, to return a file instead of byte[], then we have to make the same change in all the blocks.
2. **Rigidity:** All the conversion algorithms are being coupled inside the same method, so there is a possibility that, if you change some algorithm, others will be affected.
3. **Immobility:** The above method depends directly on the *documentType* field. Some clients will forget to set the field before calling *convertToPDF()*, so they will not get the expected result. Also, we're not able to reuse the method in any other project because of its dependency on the field.
4. **Coupling between the high-level module and the frameworks:** If we decide later on, for some purpose, to replace the Aspose framework with a more reliable one, we will end up modifying the whole *PDFConverter* class —



```

        workbook.save(pdfDocument, saveOptions)
15
        return pdfDocument.toByteArray();
16
    };
17
18 }

1  /**
    * This class holds the algorithm for convertir
2
    * documents to PDFs.
3
    * @author Hussein
4
    *
5
    */
6
public class WordPDFConverter implements Conver
7
8
    @Override
9
    public byte[] convertToPDF(byte[] fileBytes
10
        InputStream input = new ByteArrayInputS
11
        com.aspose.words.Document wordDocument
12
        ByteArrayOutputStream pdfDocument = new
13
        wordDocument.save(pdfDocument, SaveForm
14
        return pdfDocument.toByteArray();
15
    }
16
17 }

1  public class PDFConverter {
2
3
    /**
    * This method accepts the document to be c
4
    * returns the converted one.
5
    * @param fileBytes
6
    * @throws Exception
7
    */
8
    public byte[] convertToPDF(Converter conver
9
        return converter.convertToPDF(fileBytes
10
    }
11
12 }

```

We force the client to decide which conversion algorithm to use when calling *convertToPDF()*.

# What Are the Advantages of Doing It This way?

1. **Separation of concerns (high cohesion/low coupling):** The *PDFConverter* class now knows nothing about the conversion algorithms used in the application. Its main concern is to serve the clients with the various conversion features regardless how the conversion is being done. Now that we're able to replace our low-level conversion framework, no one would even know as long as we're returning the expected result.
2. **Single responsibility:** After creating an abstract layer and moving each dynamic behavior to a separate class, we actually removed the multiple responsibilities that the *convertToPDF()* method previously had in the initial design. Now it just has a single responsibility, which is delegating client requests to the abstract conversion layer. Also, each concrete class of the *Converter* interface has now a single responsibility related to converting some document type to a PDF. As a result, each component has one reason to be modified, hence no regressions.
3. **Open/Closed application:** Our application is now opened for extension and closed for modification. Whenever we want to add support for some document type, we just create a new concrete class from the *Converter* interface and the new type will become supported without the need to modify the *PDFConverter* tool, since our tool now depends on abstraction.

## Design Principles Learned From This Article

The following are some best design practices to follow when building your application's architecture.

1. Divide your application into several modules and add an abstract layer at the top of each module.
2. Favor abstraction over implementation: Always make sure to depend on the abstraction layer.

make sure to depend on the abstraction layer.

This will make your application open for future extensions. The abstraction should be applied on the dynamic parts of the application (which are most likely to be changed regularly) and not necessarily on every part, since it complicates your code in case of overuse.


3. Identify the aspects of your application that vary and separate them from what stays the same.
4. Don't repeat yourself: Always put duplicate functionalities in some utility class and make it accessible through the whole application. This will make your modification a lot easier.
5. Hide low-level implementation through the abstract layer: Low-level modules have a very high possibility to be changed regularly, so separate them from high-level modules.
6. Each class/method/module should have one reason to be changed, so always give a single responsibility for each of them in order to minimize regressions.
7. Separation of concerns: Each module knows what another module does, but it should never know how to does it.

---

Strategies and techniques for building scalable and resilient microservices to refactor a monolithic application step-by-step, a free O'Reilly book. Brought to you in partnership with Lightbend.

---

Topics: DESIGN PRINCIPLES, JAVA, SINGLE RESPONSIBILITY PRINCIPLE, DRY PRINCIPLE, REFACTORING, ABSTRACTION, TUTORIAL

Published at DZone with permission of Hussein Terek, DZone MVB. [See the original article here.](#)  Opinions expressed by DZone contributors are their own.