Arquitectura Software

"La arquitectura es un nivel de diseño que hace foco en aspectos "más allá de los algoritmos y estructuras de datos de la computación; el diseño y especificación de la estructura global del sistema es un nuevo tipo de problema "

— "An introduction to Software Architecture" de David Garlan y Mary Shaw

---

"La Arquitectura de Software se refiere a las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos. "

— Software Engineering Institute (SEI)

Arquitectura Software

"El conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de software, relaciones entre ellos, y las propiedades de ambos. "

— Documenting Software Architectures: Views and Beyond (2nd Edition), Clements et al, AddisonWesley, 2010

Arquitectura Software

"La arquitectura de software de un programa o sistema informático es la estructura o estructuras del sistema, que comprenden elementos de software, las propiedades visibles externamente de esos elementos y las relaciones entre ellos. "

— Software Architecture in Practice (2nd edition), Bass, Clements, Kazman; AddisonWesley 2003

Arquitectura Software

"La arquitectura se define como la organización fundamental de un sistema, encarnada en sus componentes, sus relaciones entre sí y con el entorno, y los principios que rigen su diseño y evolución "

— ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of SoftwareIntensive Systems

"Una arquitectura es el conjunto de decisiones importantes sobre la organización de un sistema de software, la selección de los elementos estructurales y sus interfaces mediante las cuales se compone el sistema "

— Rational Unified Process, 1999

Arquitectura Software

"La arquitectura de software es el diseño de más alto nivel de la estructura de un sistema "

— Wikipedia

"La arquitectura del software es "la estructura de los componentes de un programa/sistema, sus interrelaciones y los principios y directrices que rigen su diseño y evolución en el tiempo".

— Garlan & Perry, 1995

Arquitectura Software

Nuevo concepto: Arquitectura de software
La arquitectura de software es el diseño de más alto nivel de la estructura de un sistema, el cual consiste en un conjunto de patrones y abstracciones que proporcionan un marco claro para la implementación del sistema.

Patrones de diseño

¿Qué son los patrones de diseño?

Es común que cuando hablamos de arquitectura de software salgan términos como patrones, sin embargo, existen dos tipos de patrones, los patrones de diseño y los patrones arquitectónicos, los cuales no son lo mismo y no deberían ser confundidos por ninguna razón.

En principio, un patrón de diseño es la solución a un problema de diseño, el cual debe haber comprobado su efectividad resolviendo problemas similares en el pasado, también tiene que ser reutilizable, por lo que se deben poder usar para resolver problemas parecidos en contextos diferentes.

Patrones de diseño

Nuevo concepto: Patrones de diseño
es la solución a un problema de diseño, el cual debe haber comprobado su efectividad resolviendo problemas similares en el pasado, también tiene que ser reutilizable, por lo que se deben poder usar para resolver problemas parecidos en contextos diferentes.

Patrones de diseño

Los patrones de diseño tienen su origen en la Arquitectura (construcción), cuando en 1979 el Arquitecto Christopher Alexander publicó el libro *Timeless Way of Building*, en el cual hablaba de una serie de patrones para la construcción de edificios, comparando la arquitectura moderna con la antigua y cómo la gente había perdido la conexión con lo que se considera calidad.

Él utilizaba las siguientes palabras: "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."

Patrones de diseño

Tipos de patrones de diseño

Los patrones de diseño se dividen en tres tipos, las cuales agrupan los patrones según el tipo de problema que buscan resolver, los tipos son:

Patrones Creacionales: Son patrones de diseño relacionados con la creación o construcción de objetos. Estos patrones intentan controlar la forma en que los objetos son creados, implementando mecanismos que eviten la creación directa de objetos.

Patrones de diseño

¿Cómo diferenciar un patrón de diseño?

Una de las grandes problemáticas a la hora de identificar los patrones de diseño, es que se suelen confundir los patrones arquitectónicos que más adelante analizaremos.

Como regla general, los patrones de diseño tienen un impacto relativo con respecto a un componente, esto quiere decir que tiene un impacto menor sobre todo el componente. Dicho de otra forma, si quisiéramos quitar o remplazar el patrón de diseño, solo afectaría a las clases que están directamente relacionadas con él, y un impacto imperceptible para el resto de componentes que conforman la arquitectura.

Patrones de diseño

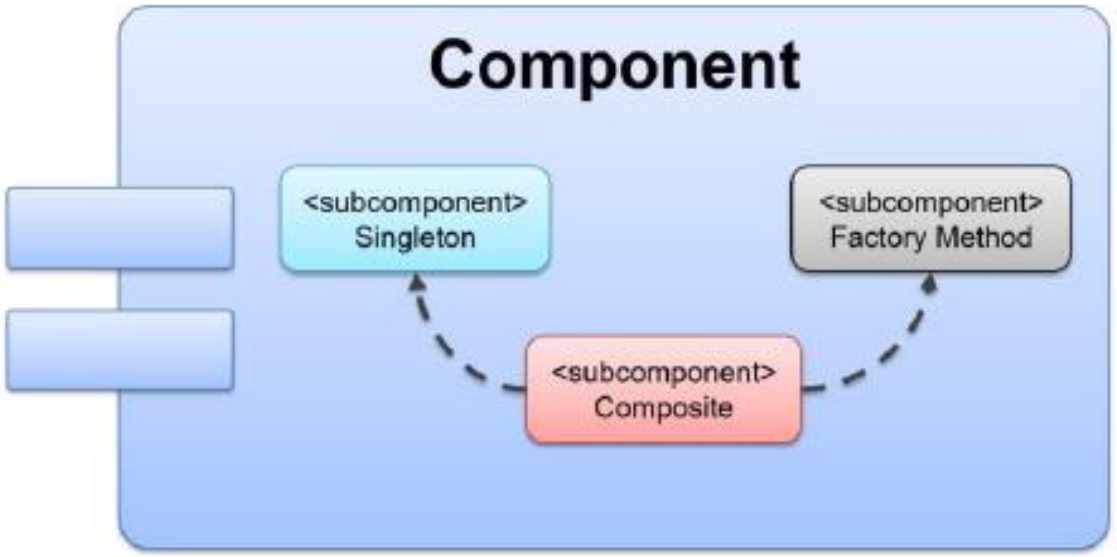

Fig 1: Muestra un componente con 3 patrones de diseño implementados.

Patrones de diseño

Como podemos ver en la imagen anterior, un componente puede implementar varios patrones de diseño, sin embargo, estos quedan ocultos dentro del componente, lo que hace totalmente transparente para el resto de módulos los patrones implementados, las clases utilizadas y las relaciones que tiene entre sí.

Como regla general, los patrones de diseño se centran en cómo las clases y los objetos se crean, relacionan, estructuran y se comportan en tiempo de ejecución, pero siempre, centrado en las clases y objetos, nunca en componentes.

Otra de las formas que tenemos para identificar un patrón de diseño es analizar el impacto que tendría sobre el componente que caso de que el patrón de diseño fallara, en tal caso, un patrón de diseño provocaría fallas en algunas operaciones del componente, pero el componte podría seguir operando parcialmente.

Patrones de diseño

Otra de las formas que tenemos para identificar un patrón de diseño es analizar el impacto que tendría sobre el componente que caso de que el patrón de diseño fallara, en tal caso, un patrón de diseño provocaría fallas en algunas operaciones del componente, pero el componte podría seguir operando parcialmente.

Cuando un arquitecto de software analiza un componente, siempre deberá crear abstracciones para facilitar su análisis, de tal forma, que eliminará todos aquellos aspectos que no son relevantes para la arquitectura y creará una representación lo más simple posible.

Patrones de diseño

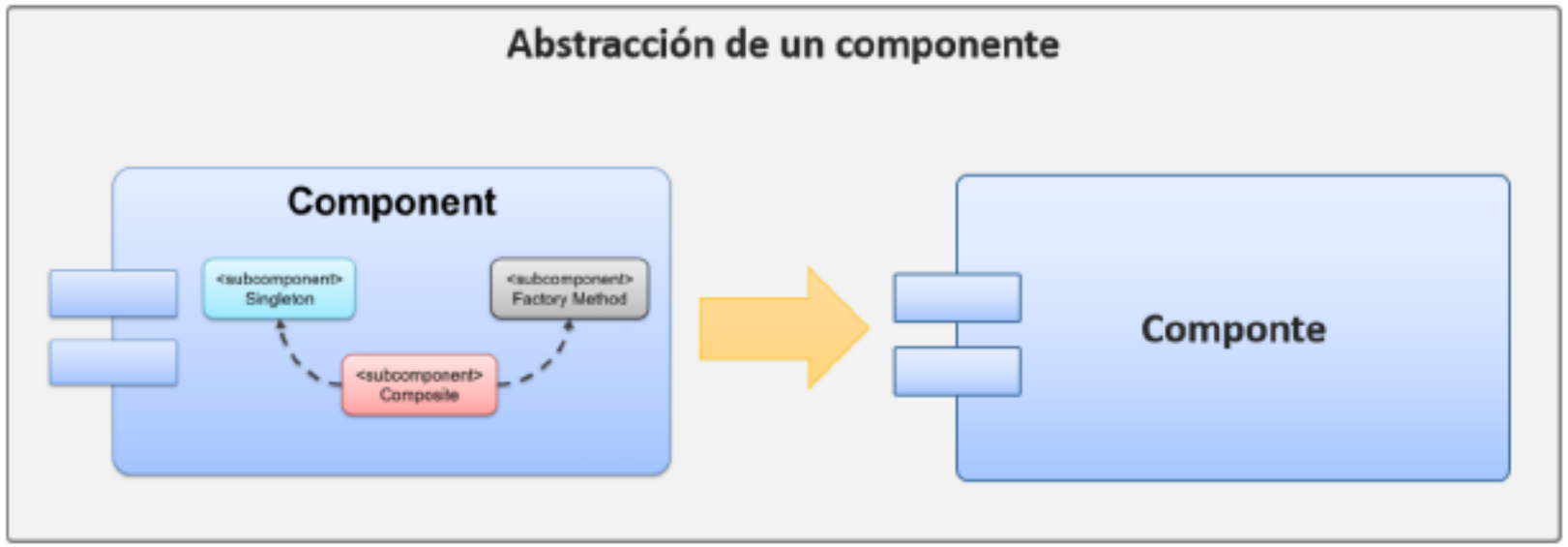

Fig 2: Abstracción de un componente

Patrones de diseño

Nuevo concepto: Abstracción
Es el proceso por medio del cual aislamos a un elemento de su contexto o resto de elementos que lo acompañan con el propósito de crear una representación más simple de él, el cual se centra en lo que hace y no en como lo hace.

Como podemos ver en la imagen anterior, un arquitecto debe de centrarse en los detalles que son realmente relevantes para la arquitectura y descartar todos aquellos que no aporten al análisis o evaluación de la arquitectura.

Sé que en este punto estarás pensando, pero un arquitecto también debe de diseñar la forma en que cada componente individual debe de trabajar, y eso es correcto, sin embargo, existe otra etapa en la que nos podremos preocupar por detalles de implementación, por hora, recordemos que solo nos interesan los componentes y la forma en que estos se relacionan entre sí.

Patrones arquitectónicos

¿Qué son los patrones arquitectónicos?

A diferencia de los patrones de diseño, los patrones arquitectónicos tienen un gran impacto sobre el componente, lo que quiere decir que cualquier cambio que se realice una vez construido el componente podría tener un impacto mayor.

> *"Un patrón arquitectónico es una solución general y reutilizable a un problema común en la arquitectura de software dentro de un contexto dado. Los patrones arquitectónicos son similares a los patrones de diseño de software, pero tienen un alcance más amplio. Los patrones arquitectónicos abordan diversos problemas en la ingeniería de software, como las limitaciones de rendimiento del hardware del equipo, la alta disponibilidad y la minimización de un riesgo empresarial. "*
>
> *— Wikipedia*

Patrones arquitectónicos

"Los patrones de una arquitectura expresan una organización estructural fundamental o esquema para sistemas complejos. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades únicas e incluye las reglas y pautas que permiten la toma de decisiones para organizar las relaciones entre ellos. El patrón de arquitectura para un sistema de software ilustra la estructura de nivel macro para toda la solución de software. Un patrón arquitectónico es un conjunto de principios y un patrón de grano grueso que proporciona un marco abstracto para una familia de sistemas. Un patrón arquitectónico mejora la partición y promueve la reutilización del diseño al proporcionar soluciones a problemas recurrentes con frecuencia. Precisamente hablando, un patrón arquitectónico comprende un conjunto de principios que dan forma a una aplicación."

— Achitectural Patterns - Pethuru Raj, Anupama Raman, Harihara Subramanian

Patrones arquitectónicos

"Los patrones de arquitectura ayudan a definir las características básicas y el comportamiento de una aplicación."

— Software Architecture Patterns - Mark Richards

"Los patrones arquitectónicos son un método para organizar bloques de funcionalidad para satisfacer una necesidad. Los patrones se pueden utilizar a nivel de software, sistema o empresa. Los patrones bien expresados le dicen cómo usarlos, y cuándo. Los patrones se pueden caracterizar según el tipo de solución a la que se dirigen. "

— MITRE

Patrones arquitectónicos

Nuevamente podemos observar que existen diversas definiciones para lo que es un patrón arquitectónico, incluso, hay quienes utilizan el termino patrón para referirse a los patrones de diseño y patrones arquitectónicos, sin embargo, queda claro que existe una diferencia fundamental, los patrones de diseño se centran en las clases y los objetos, es decir, como se crean, estructuran y cómo se comportan en tiempo de ejecución, por otra parte, los patrones arquitectónicos tiene un alcance más amplio, pues se centra en como los componentes se comportan, su relación con los demás y la comunicación que existe entre ellos, pero también abordar ciertas restricciones tecnológicas, hardware, performance, seguridad, etc.

Patrones arquitectónicos

Los patrones de diseño son diferentes a los patrones arquitectónicos.
Tanto los patrones de diseño cómo los patrones arquitectónicos pueden resultar similares ante un arquitecto inexperto, pero por ninguna razón debemos confundirlos, ya son cosas diferentes.

Patrones arquitectónicos

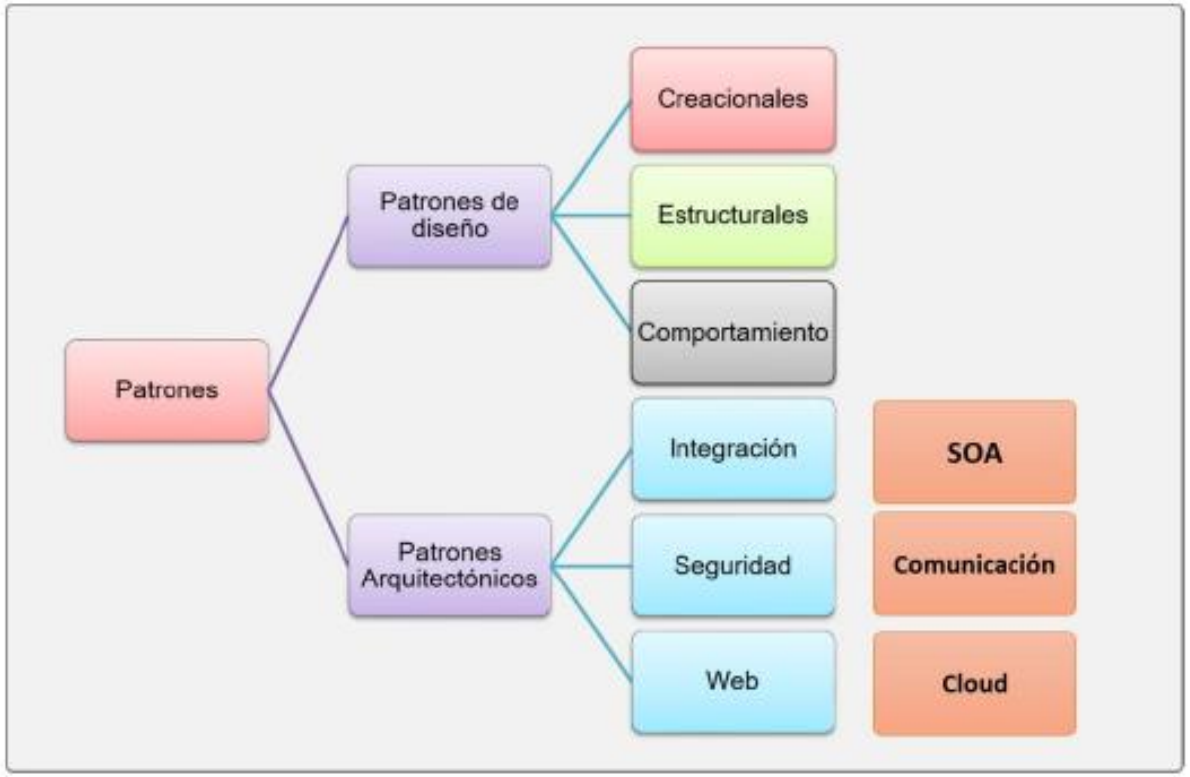

Fig 3: Tipos de patrones

Patrones arquitectónicos

¿Cómo diferenciar un patrón arquitectónico?

Los patrones arquitectónicos son fáciles de reconocer debido a que tiene un impacto global sobre la aplicación, e incluso, el patrón rige la forma de trabajar o comunicarse con otros componentes, es por ello que a cualquier cambio que se realice sobre ellos tendrá un impacto directo sobre el componente, e incluso, podría tener afectaciones con los componentes relacionados.

Patrones arquitectónicos

Un ejemplo típico de un patrón arquitectónico es el denominado "Arquitectura en 3 capas", el cual consiste en separar la aplicación 3 capas diferentes, las cuales corresponden a la capa de presentación, capa de negocio y capa de datos:

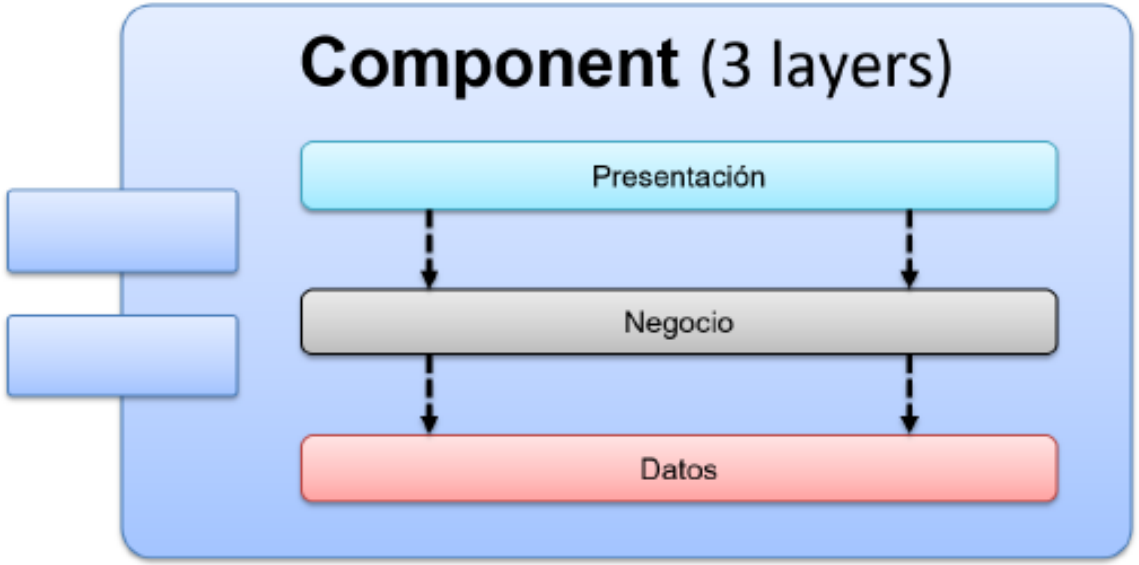

Fig 4: Arquitectura en 3 capas

Patrones arquitectónicos

En la imagen anterior podemos apreciar un componte que implementa la arquitectura de 3 capas, dicho patrón arquitectónico tiene la finalidad de separar la aplicación por capas, con la finalidad de que cada capa realiza una tarea específica. La capa de presentación tiene la tarea de generar las vistas, la capa de negocio tiene la responsabilidad de implementar la lógica de negocio, cómo implementar las operaciones, realizar cálculos, validaciones, etc, por último, la capa de datos tiene la responsabilidad de interactuar con la base de datos, como guardar, actualizar o recuperar información de la base de datos.

Cuando el usuario entra a la aplicación, lo hará a través de la capa de presentación, pero a medida que interactúe con la aplicación, este requerirá ir a la capa de negocio para consumir los datos, finalmente la capa de datos es la que realizará las consultas a la base de datos.

Patrones arquitectónicos

Dicho lo anterior, si alguna de las 3 capas falla, tendremos una falla total de la aplicación, ya que, si la capa de presentación falla, entonces el usuario no podrá ver nada, si la capa de negocios falla, entonces la aplicación no podrá guardar o solicitar información a la base de datos y finalmente, si la capa de datos falla, entonces no podremos recuperar ni actualizar información de la base de datos. En

Patrones arquitectónicos

cualquiera de los casos, en usuario quedará inhabilitado para usar la aplicación, tendiendo con ello una falla total de la aplicación.

Por otra parte, si decidimos cambiar el patrón arquitectónico una vez que la aplicación ha sido construida, tendremos un impacto mayor, pues tendremos que modificar todas las vistas para ya no usar la capa de negocios, la capa de negocio tendrá que cambiar para ya no acceder a la capa de datos, y la capa de datos quizás tenga que cambiar para adaptarse al nuevo patrón arquitectónico, sea como sea, la aplicación tendrá un fuerte impacto.

Además del impacto que tendrá el componente, hay patrones que su modificación podría impactar a otros componentes, incluso, compontes externos que están fuera de nuestro dominio, lo que complicaría aún más las cosas.

Estilos arquitectónicos

¿Qué son los estilos arquitectónicos?

El último término que deberemos aprender por ahora son los estilos arquitectónicos, los cuales también son diferentes a los patrones arquitectónicos.

Para comprender que es un estilo arquitectónico, es necesario regresarnos un poco a la arquitectura tradicional (construcción), para ellos, un estilo arquitectónico es un **método específico de construcción, caracterizado por las características que lo hacen notable y se distingue por las características que hacen que un edificio u otra estructura sea notable o históricamente identificable.**

Estilos arquitectónicos

En el software aplica exactamente igual, pues un estilo arquitectónico determina las características que debe tener un componente que utilice ese estilo, lo cual hace que sea fácilmente reconocible. De la misma forma que podemos determinar a qué periodo de la historia pertenece una construcción al observar sus características físicas, materiales o método de construcción, en el software podemos determinar que estilo de arquitectura sigue un componente al observar sus características.

Entonces, ¿Qué son los estilos arquitectónicos?, veamos algunas definiciones:

Estilos arquitectónicos

"Un estilo arquitectónico define una familia de sistemas en términos de un patrón de organización estructural; Un vocabulario de componentes y conectores, con restricciones sobre cómo se pueden combinar. "

— M. Shaw and D. Garlan, Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996.

"Un estilo de arquitectura es una asignación de elementos y relaciones entre ellos, junto con un conjunto de reglas y restricciones sobre la forma de usarlos"

—Clements et al., 2003

Estilos arquitectónicos

"Un estilo arquitectónico es una colección con nombre de decisiones de diseño arquitectónico que son aplicables en un contexto de desarrollo dado, restringen decisiones de diseño arquitectónico que son específicas de un sistema particular dentro de ese contexto, y obtienen cualidades beneficiosas en cada uno sistema resultante. "

—R. N. Taylor, N. Medvidović and E. M. Dashofy, Software architecture: Foundations, Theory and Practice. Wiley, 2009.

"La arquitectura del software se ajusta a algún estilo. Por lo tanto, como cada sistema de software tiene una arquitectura, cada sistema de software tiene un estilo; y los estilos deben haber existido desde que se desarrolló el primer sistema de software. "

—Giesecke et al., 2006; Garlan et al., 2009.

Estilos arquitectónicos

Nuevo concepto: Estilo arquitectónico
Un estilo arquitectónico establece un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos.

Como siempre, siéntete libre de adoptar la definición que creas más acertada.

Los estilos arquitectónicos son diferentes a los patrones arquitectónicos.
A pesar de que puedan parecer similares por su nombre, los patrones arquitectónicos y los estilos arquitectónicos son diferentes y por ningún motivo deben de confundirse.

Estilos arquitectónicos

La relación entre patrones de diseño, arquitectónicos y estilos arquitectónicos

Para una gran mayoría de los arquitectos, incluso experimentados, les es complicado diferenciar con exactitud que es un patrón de diseño, un patrón arquitectónico y un estilo arquitectónico, debido principalmente a que, como ya vimos, no existe definiciones concretas de cada uno, además, existe una línea muy delgada que separa a estos tres conceptos.

Para comprender mejor estos conceptos será necesario de apoyarnos de la siguiente imagen:

Estilos arquitectónicos

La relación entre patrones de diseño, arquitectónicos y estilos arquitectónicos

Para una gran mayoría de los arquitectos, incluso experimentados, les es complicado diferenciar con exactitud que es un patrón de diseño, un patrón arquitectónico y un estilo arquitectónico, debido principalmente a que, como ya vimos, no existe definiciones concretas de cada uno, además, existe una línea muy delgada que separa a estos tres conceptos.

Para comprender mejor estos conceptos será necesario de apoyarnos de la siguiente imagen:

Estilos arquitectónicos

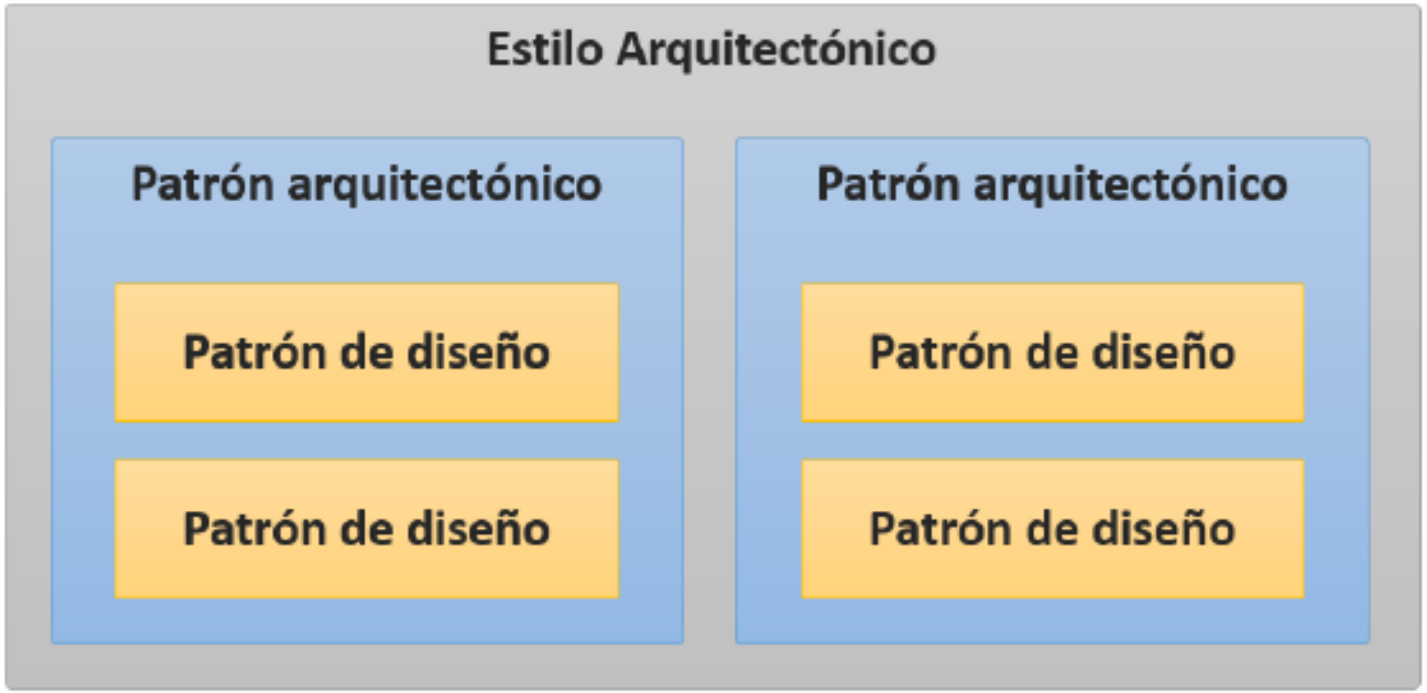

Fig 5: La relación entre patrones de diseño, patrones arquitectónicos y estilos arquitectónicos

Estilos arquitectónicos

Como podemos ver en la imagen, los estilos arquitectónicos son los de más alto nivel, y sirve como base para implementar muchos de los patrones arquitectónicos que conocemos, de la misma forma, un patrón arquitectónico puede ser implementado utilizando uno o más patrones de diseño. De esta forma, tenemos que los patrones de diseño son el tipo de patrón más específico y que centra en resolver como las clases se crean, estructura, relacionan y se comportan en tiempo de ejecución, por otra parte, los patrones arquitectónicos se enfocan en los componentes y como se relacionan entre sí, finalmente, los estilos arquitectónicos son marcos de referencia mediante los cuales es posible basarse para crear aplicaciones que compartan ciertas características.

Aun con esta explicación, siempre existe una especial confusión entre los patrones arquitectónicos y los estilos arquitectónicos, es por ello que debemos de recordar que un patrón arquitectónico existe para resolver un problema recurrente, mientras que los estilos arquitectónicos no existen para resolver un problema concreto, si no que más bien sirve para nombrar un diseño arquitectónico recurrente.

Conceptos importantes

Encapsulación

El encapsulamiento es el mecanismo por medio del cual ocultamos el estado de uno objeto con la finalidad de que actores externos no puedan modificarlos directamente, y en su lugar, se le proporcionan métodos para acceder (GET) o establecer los valores (SET), pero de una forma controlada.

Si bien, el encapsulamiento es algo que ya conocemos desde la programación orientada a objetos (POO), es un concepto que aplica de la misma forma en la arquitectura de software, pues el encapsulamiento es uno de los principales principios que debemos aprender al momento de desarrollar componentes de software.

De la misma forma que un objeto tiene métodos get y set para establecer y recuperar los valores, los componentes proporcionan métodos, funciones, servicios o API's que permite recuperar datos de su estado o incluso modificarlos.

Conceptos importantes

De la misma forma que un objeto tiene métodos get y set para establecer y recuperar los valores, los componentes proporcionan métodos, funciones, servicios o API's que permite recuperar datos de su estado o incluso modificarlos.

La importancia de encapsular un componente es evitar que cualquier agente externo, ya sea una persona u otro sistema modifique los datos directamente sin pasar por una capa que podamos controlar. Analicemos el siguiente ejemplo:

Tenemos dos sistemas (A y B), el sistema A requiere consultar y modificar datos de la aplicación B, por lo que el desarrollador de la aplicación A decide que lo más fácil es comunicarse directamente con la base de datos de la aplicación B.

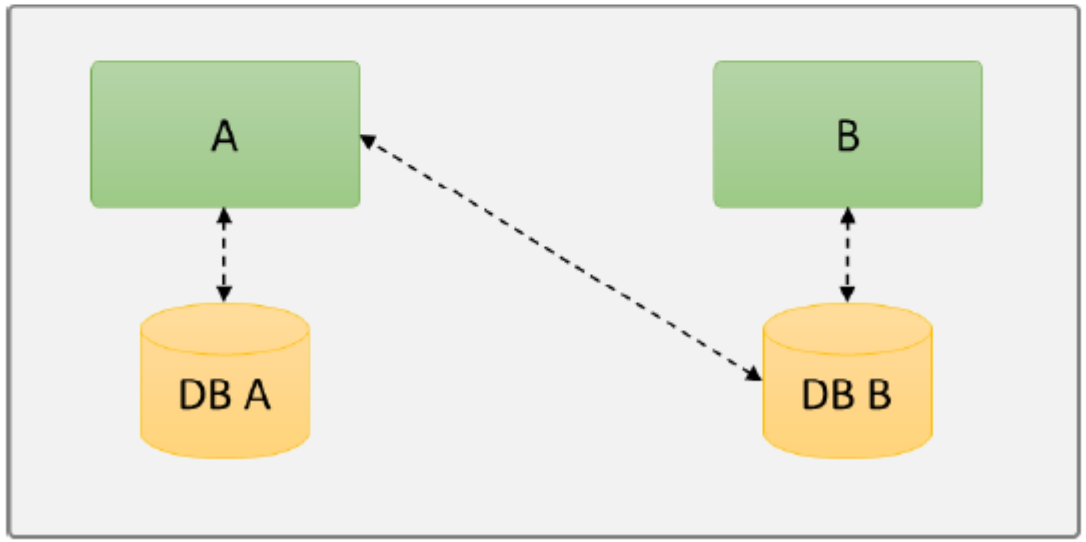

Fig 6 - Encapsulamiento omitido.

Conceptos importantes

Para muchos, este tipo de arquitectura es súper normal, incluso la llegan a justificar diciendo que lo hacen porque no tiene tiempo o porque es más rápido. Sin embargo, este tipo de arquitectura es hoy en día muy mal vistas, ya que la aplicación A podría modificar cualquier dato de B sin que la aplicación B se entere, pudiendo provocar desde inconsistencia en la información, hasta la eliminación o actualización de registros vitales para el funcionamiento de B. Por otra parte, también podría leer datos sensibles que no deberían salir de la aplicación, como datos de clientes o tarjetas de crédito.

Es por este motivo que B debe de impedir a toda costa que aplicaciones externas (como es el caso de A), accedan directamente a su estado (datos), y en su lugar deberá proporcionar funciones, método, servicios o un API que permite acceder al estado de una forma segura.

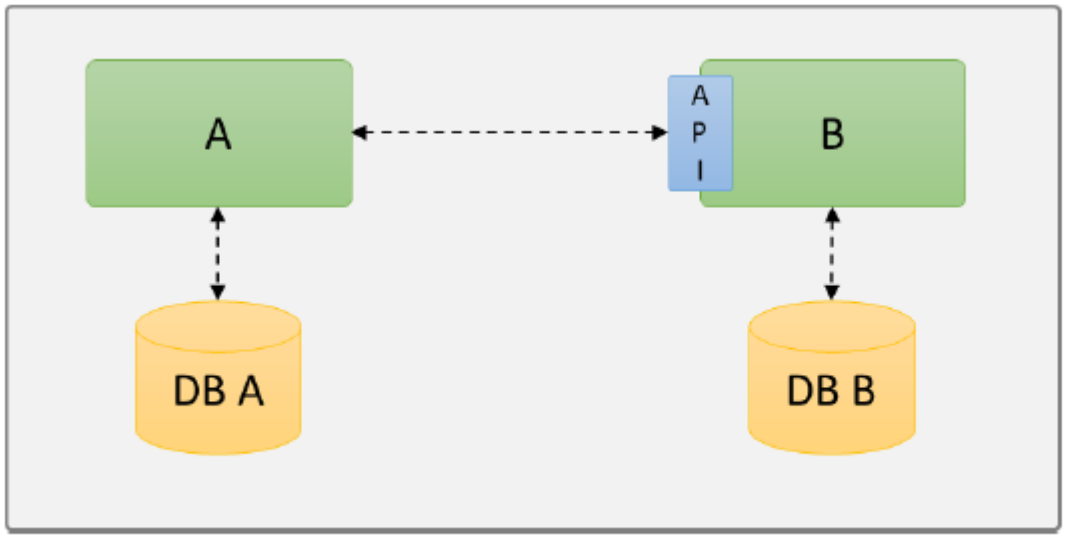

Fig 7 - Cumpliendo con el encapsulamiento.

Conceptos importantes

En esta nueva arquitectura podemos ver qué, A se comunica con B por medio de un API, y de esta forma, B puede controlar el acceso a los datos, de tal forma que podría negar el acceso a cierta información o rechazar una transacción que no cumpla con las reglas de negocio, de esta forma, B se protege de que le extraigan información privilegiada al mismo tiempo que se protege de una inconsistencia en los datos.

Pero el encapsulamiento no todo se trata de acceder a los datos, sino que también permite crear **Abstracciones** de un componente para poder ser representado de

la forma más simple posible, ocultado todos aquellos detalles que son irrelevantes para la arquitectura.

Conceptos importantes

Acoplamiento

El acoplamiento es nivel de dependencia que existe entre dos unidades de software, es decir, indica hasta qué grado una unidad de software puede funcionar sin recurrir a la otra.

Entonces podemos decir que el acoplamiento es el nivel de dependencia que una unidad de software tiene con la otra. Por ejemplo, imagina una aplicación de registro de clientes y su base de datos, dicho esto, ¿qué tan dependiente es la aplicación de la base de datos? ¿podría seguir funcionando la aplicación sin la base de datos? Seguramente todos estaremos de acuerdo que sin la base de datos la aplicación no funcionará en absoluto, pues es allí donde se guarda y consulta la información que vemos en el sistema, entonces podríamos decir que existe un **Alto acoplamiento**.

Conceptos importantes

Cuando un arquitecto inexperto diseña una solución, por lo general crea relaciones de acoplamiento altas entre los módulos de la solución, no porque sea bueno o malo, sino porque es la única forma que conoce y que al mismo tiempo es la más fácil. Por ejemplo, si te pidiera que comunicaras dos sistemas, en el cual, la aplicación A debe mandar un mensaje a B por medio de un Webservices asíncrono, ¿cómo diseñarías la solución? La gran mayoría haría esto:

Fig 8 - Alto acoplamiento.

En la imagen anterior podemos ver cómo A manda un mensaje directamente a B. Esta arquitectura podría resultar de los más normal, pues si B expone un servicio, entonces lo consumimos y listo, sin embargo, si B está apagado o fallando al momento de mandarle el mensaje lo más seguro es que todo el flujo en A falle, debido a que la comunicación con B fallara, lo que lanzará una excepción haciendo un Rollback de todo el proceso.

Conceptos importantes

Ahora, ponte a pensar que el proceso en A era una venta en línea y que el mensaje enviado a B era solo para mandar un mail de agradecimiento al cliente. Esto quiere decir que acabas de perder una venta solo por no poder mandar un mail. Entonces, podemos decir que el componente A y B están altamente acoplados y el hecho de que B falle, puede llevar a un paro de operaciones en A, a pesar de que el envío del mail no era tan importante.

Entonces, ¿el acoplamiento es bueno o es malo? Dado el ejemplo anterior, podríamos deducir que es malo, sin embargo, el acoplamiento siempre existirá, ya sea con un componente u otro, por lo que el reto del arquitecto no es suprimir todas las dependencias, si no reducirlas al máximo, lo que nos lleva al punto central de esta sección, el **Bajo acoplamiento**.

Conceptos importantes

El bajo acoplamiento se logra cuando un módulo no conoce o conoce muy poco del funcionamiento interno de otros módulos, evitando la fuerte dependencia entre ellos. Como buena práctica siempre debemos buscar el bajo acoplamiento, sin embargo, no siempre será posible y no deberemos forzar una solución para lograrlo.

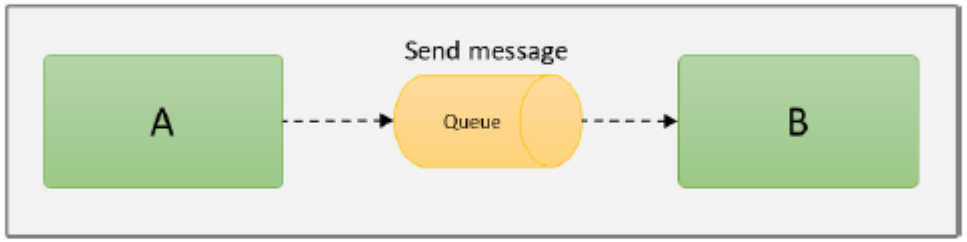

Fig 9 - Bajo acoplamiento

La imagen anterior propone una nueva arquitectura, en la cual utilizamos un Cola (Queue) para almacenar los mensajes, los cuales serán recuperados por B de forma asíncrona. De esta forma, si B está apagado, A podrá seguir dejando los mensajes en la Queue y cuando B esté nuevamente disponible podrá recuperar los mensajes enviados por A.

Esta es una de las muchas posibles soluciones para bajar el nivel de acoplamiento, incluso, desacoplarlas por completo.

Conceptos importantes

Bajo acoplamiento
Como arquitectos siempre deberemos buscar el bajo acoplamiento entre los componentes (siempre y cuando sea posible).

Conceptos importantes

Cohesión

la cohesión es una medida del grado en que los elementos del módulo están relacionados funcionalmente, es decir, se refiere al grado en que los elementos de un módulo permanecen juntos.

En otras palabras, la cohesión mide que tan relacionados están las unidades de software entre sí, buscando que todas las unidades de software busquen cumplir un único objetivo o funcionalidad.

Conceptos importantes

En la práctica, se busca que todos los compontes de software que construyamos sean altamente cohesivos, es decir, que el módulo esté diseñado para resolver una única problemática.

Imaginemos que estamos por construir una nueva aplicación web para un cliente; esta aplicación además de cumplir con todos los requerimientos de negocio, requiere de un módulo de seguridad en donde podamos administrar a los usuarios.

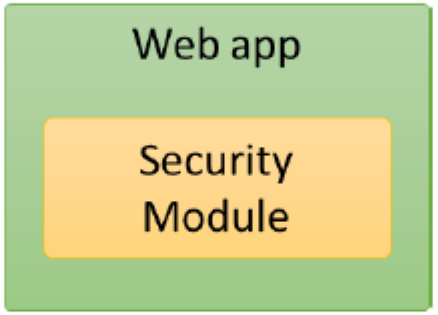

Fig 10 - Baja cohesión

Para muchos, implementar el módulo de seguridad dentro de la aplicación es lo más normal, pues al final, necesitamos asegurar la aplicación, sin embargo, puede que en principio esto no tenga ningún problema, pero estamos mesclando la responsabilidad de negocio que busca resolver la aplicación, con la administración de los accesos, lo que hace que además de preocuparse por resolver una

Conceptos importantes

problemática de negocio, también nos tenemos que preocupar por administrar los accesos y privilegios.

Si esta es la única aplicación que requiere autenticación, entonces podríamos decir que está bien, sin embargo, las empresas por lo general tienen más de una aplicación funcionando y que además requiere de autenticación. Entonces, ¿qué pasaría si el día de mañana nos piden otra aplicación que requiere autenticación? Lo más seguro es que terminemos copiando el código de esta aplicación y pegándolo en la nueva, lo que comprueba que la aplicación es bajamente cohesiva, pues buscó resolver más de una problemática.

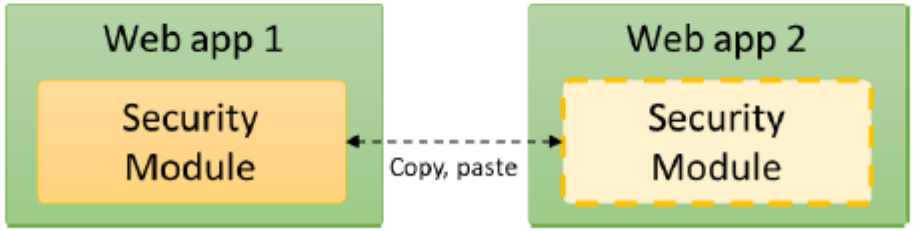

Fig 11 - Copy-Paste code

Conceptos importantes

Ahora bien, que pasaría si nos damos cuenta que el módulo de seguridad es una problemática distinta a la que busca resolver la aplicación y que, además, podría ser utilizada por otras aplicaciones:

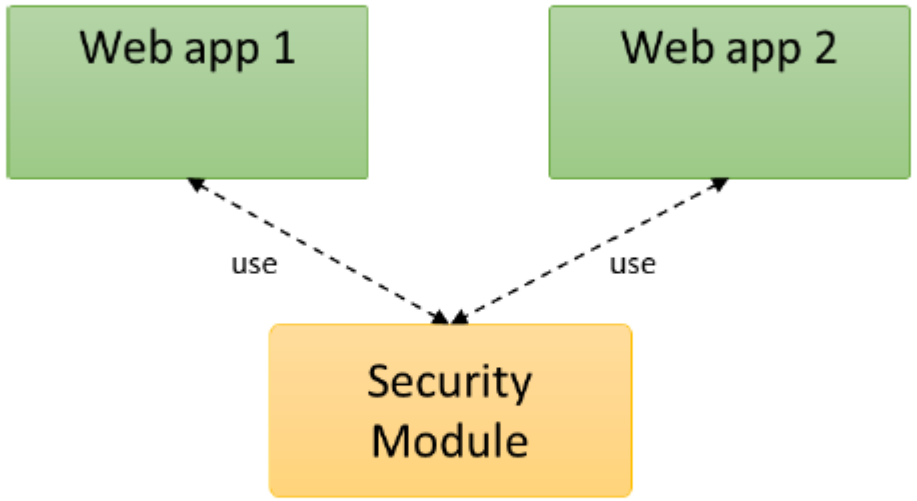

Fig 12 - Alta cohesión

En esta nueva arquitectura podemos ver que hemos decidido separar las responsabilidades de forma que sean más cohesivas, de tal forma que la Web App

Conceptos importantes

1 y 2 solo se dedican a resolver un problema concreto de negocio y el módulo de seguridad solo se dedica a la seguridad, de tal forma que cualquier aplicación que requiere habilitar la seguridad, puede utilizar el módulo.

Alta cohesión
Como arquitectos siempre deberemos buscar construir componentes con Alta cohesión.

Finalmente, la cohesión y el acoplamiento son principios que están muy relacionados, y de allí viene la famosa frase "**Bajo acoplamiento y Alta cohesión**", lo que significa que los componentes que desarrollemos deberán ser lo más independiente posibles (bajo acoplamiento) y deberán ser diseñados para realizar una sola tarea (alta cohesión).

Conceptos importantes

Don't repeat yourself (DRY)

Este es un principio que se infringe con regularidad por los principiantes, el cual se traduce a "No te repitas" y lo que nos dice es que debemos evitar duplicar código o funcionalidad.

No repetir código o funcionalidad puede resultar lógico, sin embargo, es muy común ver programadores que copian y pegan fragmentos de código, o arquitectos que diseñan componentes que repiten funcionalidad en más de un módulo, lo que puede complicar mucho la administración, el mantenimiento y la corrección de bugs.

Recordemos el caso que acabamos de exponer para explicar la Cohesión, decíamos que es muy común ver aplicaciones que implementan el módulo se seguridad como parte de su funcionalidad, y que, si nos pedían otra aplicación, terminábamos copiando y pegando el código de seguridad en la nueva aplicación. Este ejemplo además de los problemas de Cohesión que ya analizamos tiene otros problemas, como el mantenimiento, la configuración y la corrección de errores.

Analizamos los siguiente, del ejemplo anterior, imagina que necesitamos instalar las dos aplicaciones desde cero, esto provocaría que tengamos que configurar el

Conceptos importantes

módulo de seguridad dos veces, lo que implica doble trabajo y una posible inconsistencia, pues puede que, durante la captura de los usuarios, pongamos diferentes privilegios a algún usuario por error, o que capturemos un dato erróneamente en alguna de las dos aplicaciones.

Por otra parte, si nos piden cambiar los privilegios a un usuario, tendremos que acceder a las dos aplicaciones y aplicar los cambios. Y finalmente, si encontramos un Bug en el módulo de seguridad, tendremos que aplicar las correcciones en las dos aplicaciones y desplegar los cambios.

Entonces, cuando el principio DRY se aplica de forma correcta, cualquier cambio que realicemos en algún lugar, no debería de requerir aplicarlo en ningún otro lugar. Por el contrario, cuando no aplicamos correctamente este principio, cualquier cambio que realicemos, implicará que tengamos que replicar los cambios en más de un sitio.

Nunca repitas
Si detectas que un fragmento de código, método o funcionalidad completa se repite o requieres repetirla, quiere decir que es una funcionalidad que puede ser reutilizada, por lo que lo mejor es separarla en métodos de utilidad o un módulo independiente que pueda ser reutilizado.

Conceptos importantes

Separation of concerns (SoC)

El principio SoC o "Separación de preocupaciones" o "separación de conceptos" o "separación de intereses" en español, propone la separación de un programa en secciones distintas, de tal forma que cada sección se enfoque en resolver una parte delimitada del programa.

En este sentido, un interés o una preocupación es un conjunto de información que afecta al código de un programa el cual puede ser tan general como los detalles del hardware para el que se va a optimizar el código, el acceso a los datos, o la lógica de negocios.

Conceptos importantes

El principio SoC puede ser confundido con la cohesión, pues al final, los dos principios promueven la agrupación de funcionalidad por el objetivo que buscan conseguir, sin embargo, en SoC, puede tener varios módulos o submódulos separados que buscan conseguir el mismo objetivo, un ejemplo clásico de este principio son las arquitecturas por capas, donde la arquitectura en 3 capas es la más famosa y que analizaremos con detalle en este libro. Esta arquitectura propone la separación de la aplicación en 3 capas:

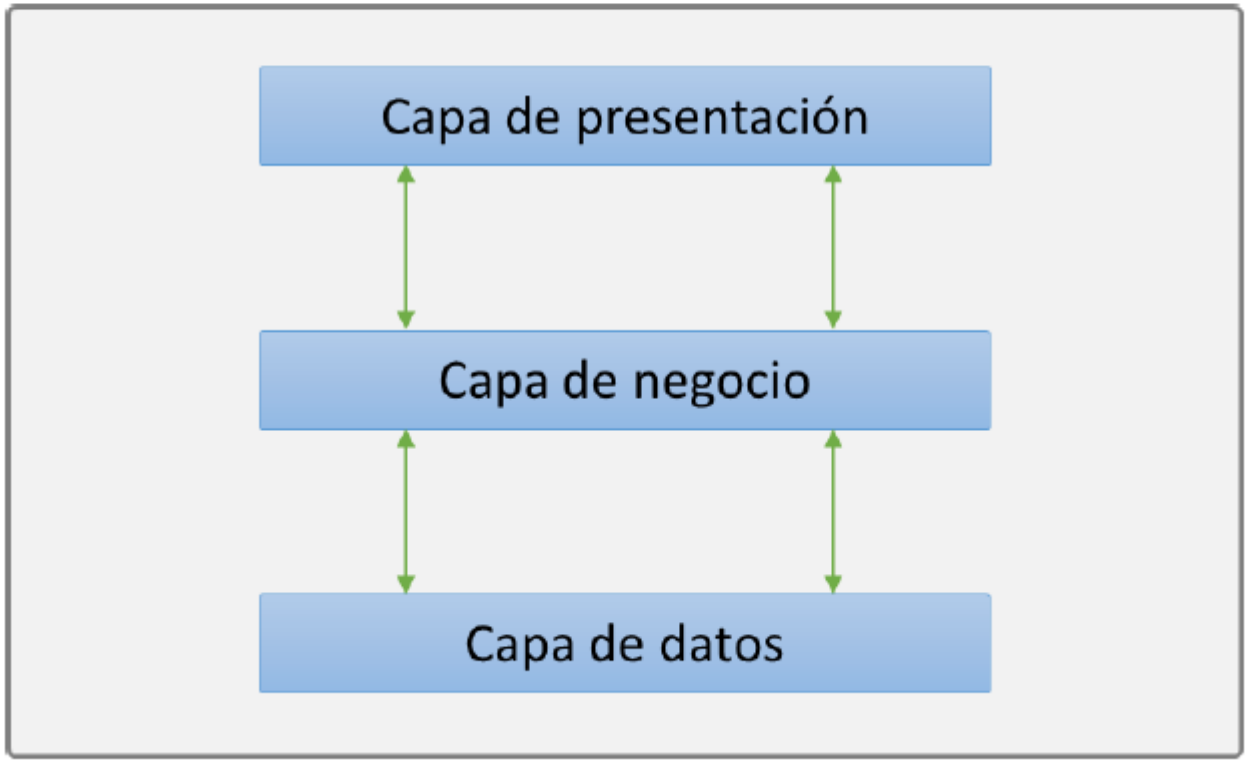

Fig 13 - Arquitectura de 3 capas

Conceptos importantes

1. Capa de presentación: en esta se implementa toda la lógica para generar las vistas de usuario.
2. Capa de negocio: en esta presentación se implementa toda la lógica de negocio, es decir, cálculos, validaciones, orquestación del flujo de negocio, etc.
3. Capa de datos: esta capa se encarga únicamente del acceso a datos.

A pesar de que las capas separan el código por responsabilidades, la realidad es que las 3 capas buscan satisfacer el mismo objetivo. Otro punto importante, es

Conceptos importantes

que para cumplir este principio no es necesario separar las responsabilidades en módulos, si no que puede ser el mismo módulo, pero con la funcionalidad encapsulada, de tal forma que cada responsabilidad no puede modificar al estado de la otra.

La separación por responsabilidades no siempre podrá ser posible, pues depende del tipo de aplicación que estemos desarrollando, sin embargo, es un buen principio que siempre tendremos que tener en mente.

Conceptos importantes

Inversion of control (IoC)

Seguramente muchos de nosotros ya hemos escuchado el término Inversion of control (Inversión de control) más de una vez, incluso otros, ya los están utilizando sin saberlo. Lo cierto es que cada vez es más común escuchar este término y es que ha revolucionado por completo la forma trabajar con los Frameworks.

Inversion of control (IoC) es una forma de trabajar que rompe con el formato tradicional, en donde el programar es el encargado de definir la secuencia de operaciones que se deben de realizar para llegar a un resultado. En este sentido el programador debe conocer todos los detalles del framework y las operaciones a realizar en él, por otra parte, el Framework no conoce absolutamente nada de nuestro programa. Es decir, solo expone operaciones para ser ejecutada. La siguiente imagen muestra la forma de trabajo normal.

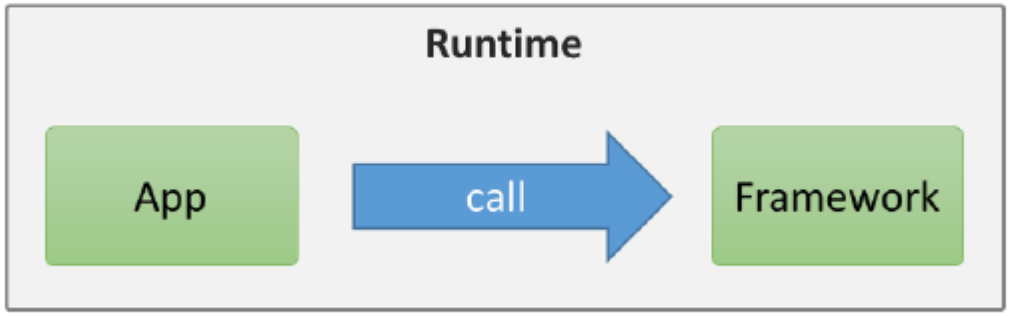

Fig 16 - Invocación tradicional

Conceptos importantes

Veamos que nuestra aplicación es la que llama en todo momento al Framework y este hace lo que la App le solicita. Por otra parte, al Framework no le interesa en absoluto la App.

Pero qué pasaría si invirtiéramos la forma de trabajo, y que en lugar de que la App llame funciones del Framework, sea el Framework el que ejecute operaciones sobre la App, ¿suena extraño no? Pues en realidad a esta técnica es a la que se le conoce como Inversion of Control. Veamos en la siguiente imagen como es que funciona IoC:

Conceptos importantes

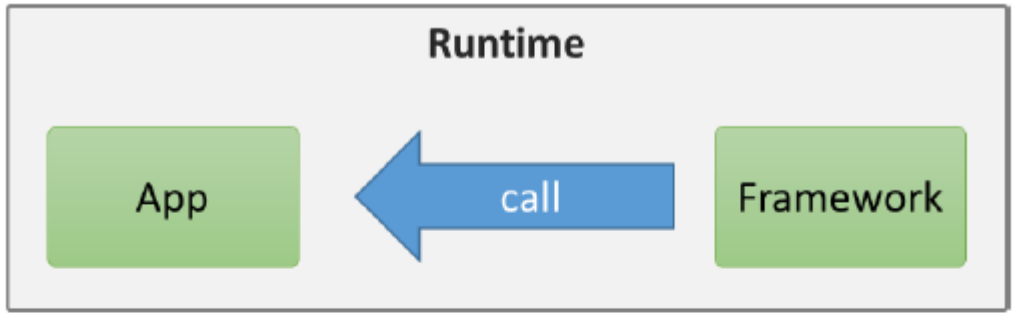

Fig 17 - Inversión de control

Esta última imagen se ve bastante parecida a la anterior, sin embargo, tiene una gran diferencia, y es que las llamadas se invirtieron, ahora es el Framework es el realiza operaciones sobre nuestra App.

Seguramente a estas alturas te pregunta cómo es que un Framework que no conoce tu App y que incluso se compilo desde mucho antes puede realizar operaciones sobre nuestra App. La respuesta es simple pero complicada, y es que utilizar IoC es mucho más simple de lo que parecería, pero implementar un Framework de este tipo tiene un grado alto de complejidad, dicho de otra manera, si tú eres la App puedes implementar muy fácilmente al Framework, pero si tu estas desarrollando el Framework puede ser un verdadero dolor de cabeza.

Conceptos importantes

Pues bien, IoC se basa en la introspección (en Java llamado Reflection) el cual es un procedimiento por el cual leemos metadatos de la App que nos permita entender cómo funciona. Los metadatos pueden estar expresados principalmente de dos formas, la primera es mediante archivos de configuración como XML o con metadados directamente definidos sobre las clases de nuestro programa, en Java estos metadatos son las `@Annotations` o anotaciones y es por medio de estos metadatos y con técnicas de introspección que es posible entender el funcionamiento de la App.

Nuevo concepto: Introspección
La introspección son técnicas para analizar clases en tiempo de ejecución, mediante la cual es posible saber las propiedades, métodos, clases, ejecutar operaciones, settear u obtener valores, etc. y todo esto sin saber absolutamente nada de la clase.

Este principio es también conocido como el principio Hollywood, haciendo referencia a lo que les dicen los productores a los actores en los castings, "**No nos**

Conceptos importantes

llame, nosotros lo llamaremos", y es que la inversión de control hace exactamente eso, en lugar de que nosotros llamemos las operaciones del framework, es el framework en que terminan llamando las operaciones de nuestras clases.

Microservicios

El estilo arquitectónico de Microservicios se ha convertido en el más popular de los últimos años, y es que no importa la conferencia o eventos de software te dirijas, en todos están hablando de los Microservicios, lo que la hace el estilo arquitectónico más relevante de la actualidad.

El estilo de Microservicios consiste en crear pequeños componentes de software que solo hacen una tarea, la hace bien y son totalmente autosuficientes, lo que les permite evolucionar de forma totalmente independiente del resto de componentes.

El nombre de "Microservicios" se puede mal interpretar pensando que se trata de un servicio reducido o pequeño, sin embargo, ese es un concepto equivocado, los Microservicios son en realidad pequeñas aplicaciones que brindan un servicio, y observa que he dicho "**brinda un servicio**" en lugar de "exponen un servicio", la diferencia radica en que los componentes que implementan un estilo de Microservicios no tiene por qué ser necesariamente un Web Services o REST, y en su lugar, puede ser un procesos que se ejecuta de forma programada, procesos que mueva archivos de una carpeta a otra, componentes que responde a eventos, etc. En este sentido, un Microservicios no tiene porqué exponer necesariamente un servicio, sino más bien, proporciona un servicio para resolver una determinada tarea del negocio.

Microservicios

Un Microservicios es un pequeño programa que se especializa en realizar una pequeña tarea y se enfoca únicamente en eso, por ello, decimos que los Microservicios son Altamente Cohesivos, pues toda las operaciones o funcionalidad que tiene dentro está extremadamente relacionadas para resolver un único problema.

En este sentido, podemos decir que los Microservicios son todo lo contrario a las aplicaciones Monolíticas, pues en una arquitectura de Microservicios se busca desmenuzar una gran aplicación en muchos y pequeños componentes que realizar de forma independiente una pequeña tarea de la problemática general.

Una de las grandes ventajas de los Microservicios es que son componentes totalmente encapsulados, lo que quiere decir que la implementación interna no es de interés para los demás componentes, lo que permite que estos evolucionen a la velocidad necesaria, además, cada Microservicios puede ser desarrollado con tecnologías totalmente diferentes, incluso, es normal que utilicen diferentes bases de datos.

Microservicios

Como se estructura un Microservicios

Para comprender los Microservicios es necesario regresar a la arquitectura Monolítica, donde una solo aplicación tiene toda la funcionalidad para realizar una tarea de punta a punta, además, una arquitectura Monolítica puede exponer servicios, tal como lo vemos en la siguiente imagen:

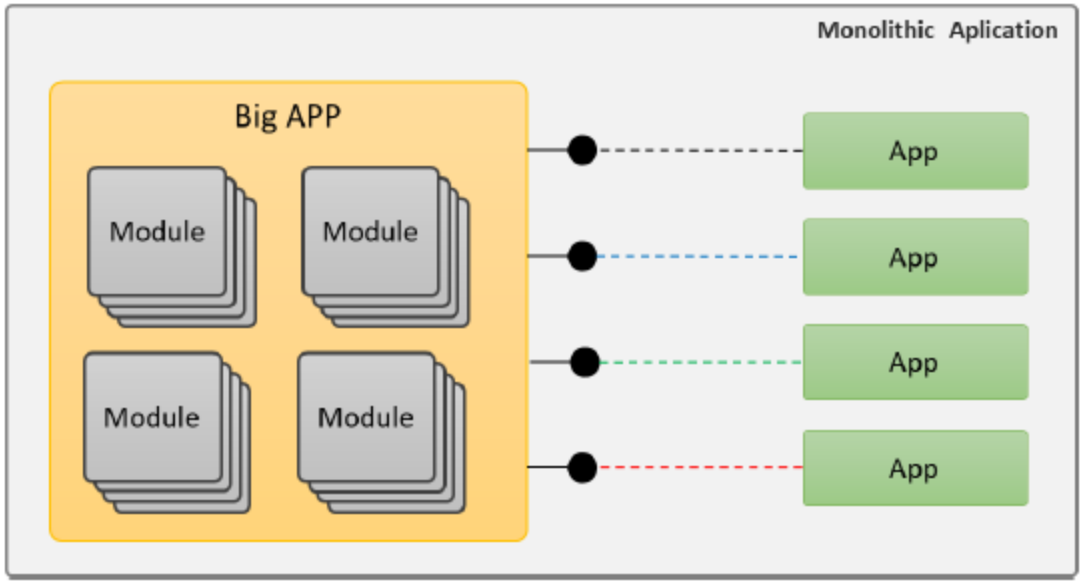

Fig 67 - Arquitectura Monolítica.

Microservicios

Una aplicación Monolítica tiene todos los módulos y funcionalidad necesario dentro de ella, lo que lo hace una aplicación muy grande y pesada, además, en una arquitectura Monolítica, es Todo o Nada, es decir, si la aplicación está arriba, tenemos toda la funcionalidad disponible, pero si está abajo, toda la funcionalidad está inoperable.

La arquitectura de Microservicios busca exactamente lo contrario, dividiendo toda la funcionalidad en pequeños componentes autosuficientes e independientes del resto de componentes, tal y como lo puedes ver en la imagen anterior.

Microservicios

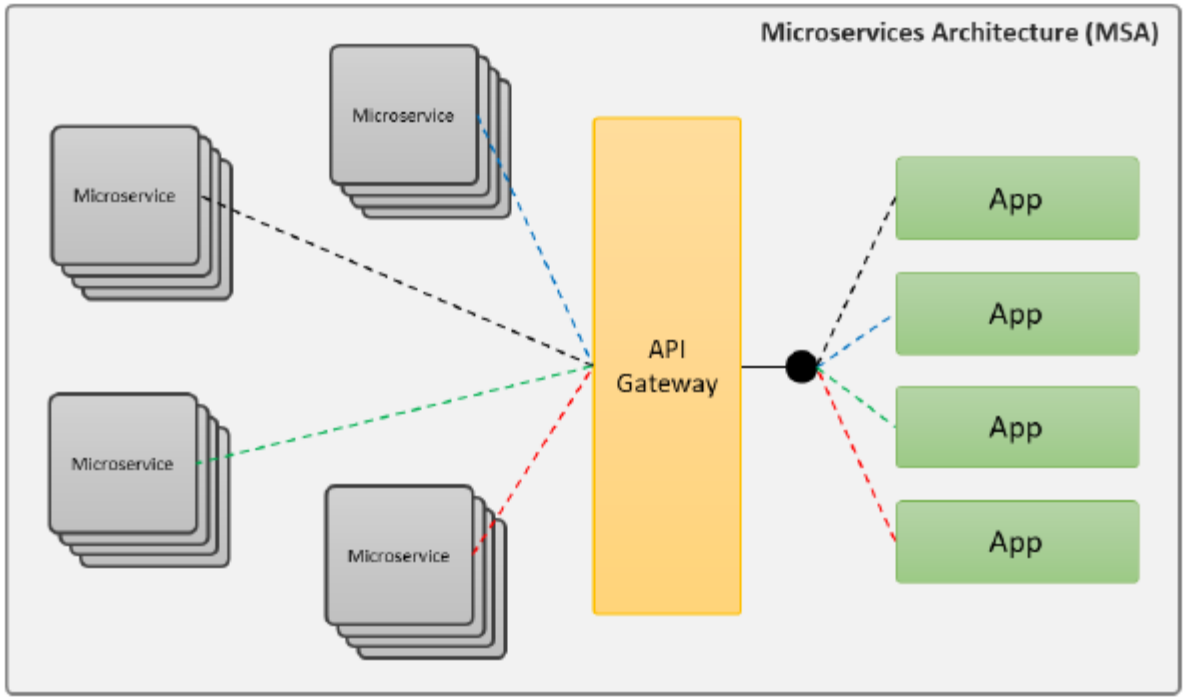

Fig 68 - Arquitectura de Microservicios

En la arquitectura de Microservicios es común ver algo llamado *API Gateway*, el cual es un componente que se posiciona de cara a los microservicios para servir como puerta de entrada a los Microservicios, controlando el acceso y la funcionalidad que deberá ser expuesta a una red pública (Más adelante retomaremos el API Gateway para analizarlo a detalle).

Microservicios

Debido a que los Microservicios solo realizan una tarea, es imposible que por sí solos no puedan realizar una tarea de negocio completa, por lo que es común que los Microservicios se comuniquen con otros Microservicios para delegar ciertas tareas, de esta forma, podemos ver que todos los Microservicios crean una red de comunicación entre ellos mismos, incluso, podemos apreciar que diferentes Microservicios funcionan con diferentes bases de datos.

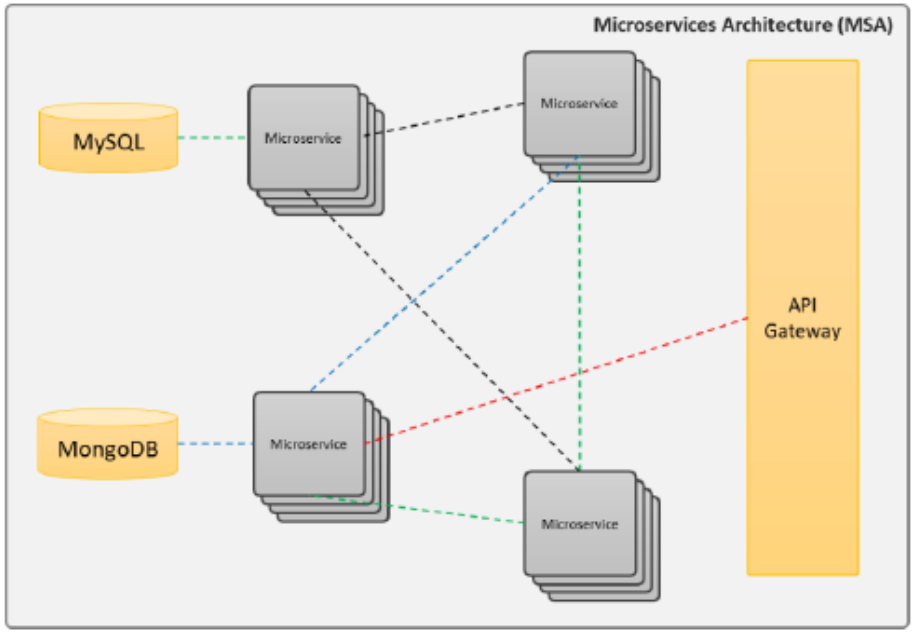

Fig 69 - Comunicación entre Microservicios

Microservicios

Algo a tomar en cuenta es que los Microservicios trabajan en una arquitectura distribuida, lo que significa todos los Microservicios son desplegados de forma independiente y están desacoplados entre sí. Además, deben ser accedidos por medio de protocolos de acceso remoto, como colas de mensajes, SOAP, REST, RPC, etc. Esto permite que cada Microservicio funcione o pueda ser desplegado independientemente de si los demás están funcionando.

Microservicios

Escalabilidad Monolítica vs escalabilidad de Microservicios

Una de las grandes problemáticas cuando trabajamos con arquitecturas Monolíticas es la escalabilidad, ya que son aplicaciones muy grandes que tiene mucha funcionalidad, la cual consume muchos recursos, se use o no, estos recursos tienen un costo financiero pues hay que tener el hardware suficiente para levantar todo, ya que recordemos que una arquitectura Monolítica es TODO o NADA, pero hay algo más, el problema se agudiza cuando necesitamos escalar la aplicación Monolítica, lo que requiere levantar un nuevo servidor con una réplica de la aplicación completa.

Microservicios

En una arquitectura Monolítica no podemos decidir qué funcionalidad desplegar y que no, pues la aplicación viene como un TODO, lo que nos obliga a escalar de forma Monolítica, es decir, escalamos todos los componentes de la aplicación, incluso si es funcionalidad que casi no se utiliza o que no se utiliza para nada.

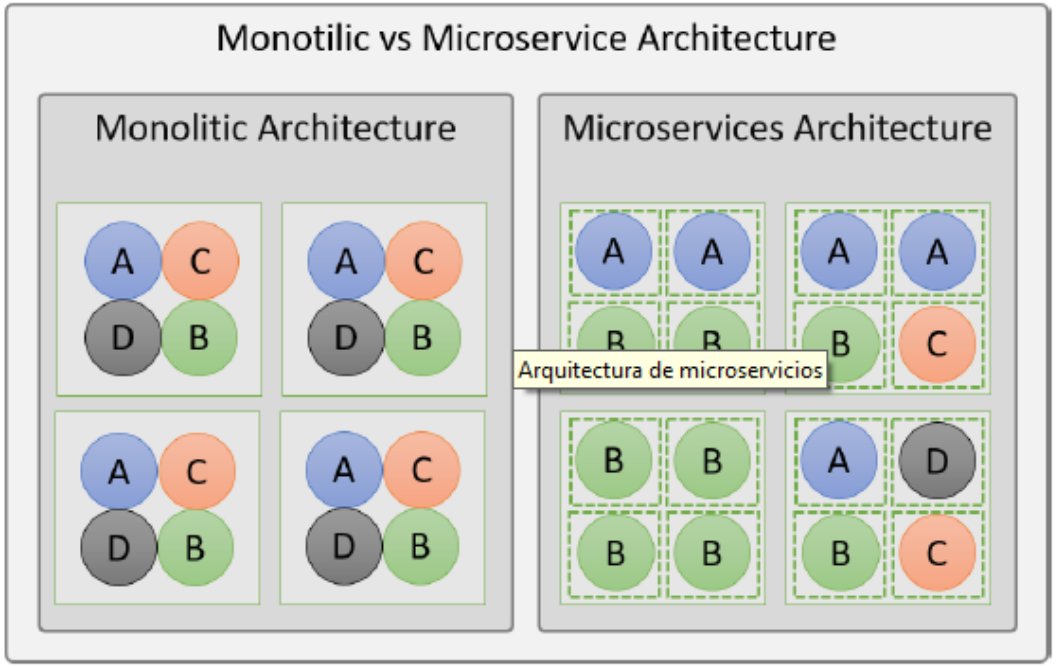

Fig 70 - Escalamiento Monolítica vs Microservicios

Microservicios

Del lado izquierdo de la imagen anterior podemos ver como una aplicación Monolítica es escalada en 4 nodos, lo que implica que la funcionalidad A, B, C y D sean desplegadas por igual en los 4 nodos. Obviamente esto es un problema si hay funcionalidad que no se requiere, pero en un Monolítico no tenemos otra opción.

Del lado derecho podemos ver cómo es desplegado los Microservicios. Podemos ver que tenemos exactamente las mismas funcionalidades A, B, C y D, con la diferencia de que esta vez, cada funcionalidad es un Microservicio, lo que permite que cada funcionalidad sea desplegada de forma independiente, lo que permite que decidamos que componentes necesitan más instancias y cuales menos. Por ejemplo, pude que la funcionalidad B sea la más crítica, que es donde realizamos las ventas, entonces creamos 8 instancias de ese Microservicio, por otro lado, tenemos el componente que envía Mail, el cual no tiene tanda demanda, así que solo dejamos una instancia.

Microservicios

Como podemos ver, los Microservicios permite que controlemos de una forma más fina el nivel de escalamiento de los componentes, en lugar de obligarnos a escalar toda la aplicación de forma Monolítica.

En la siguiente unidad hablaremos de los patrones arquitectónicos *Service Registry* y *Service Discovery*, los cuales nos permiten hacer un auto descubrimiento de los servicios y lograr un escalamiento dinámico que permite crecer o disminuir el número de instancias a medida que crece o baja la carga de trabajo sin necesidad de modificar la configuración de la aplicación.

Microservicios

Características de un Microservicio

En esta sección analizaremos las características que distinguen al estilo de Microservicios del resto. Las características no son ventajas o desventajas, si no que más bien, nos sirven para identificar cuando una aplicación sigue un determinado estilo arquitectónico.

Las características se pueden convertir en ventajas o desventajas solo cuando analizamos la problemática que vamos a resolver, mientras tanto, son solo características:

- Son componentes altamente cohesivos que se enfocan en realizar una tarea muy específica.
- Son componentes autosuficientes y no requieren de ninguna otra dependencia para funcionar.
- Son componentes distribuidos que se despliegan de forma totalmente independiente de otras aplicaciones o Microservicios.
- Utilizan estándares abiertos ligeros para comunicarse entre sí.
- Es normal que existan múltiples instancias del Microservicios funcionando al mismo tiempo para aumentar la disponibilidad.

Microservicios

- Los Microservicios son capaces de funcionar en hardware limitado, pues no requieren de muchos recursos para funcionar.
- Es común que una arquitectura completa requiere de varios Microservicios para ofrecer una funcionalidad de negocio completa.

- Es común ver que los Microservicios están desarrollados en tecnologías diferentes, incluido el uso de bases de datos distintas.

Microservicios

Ventajas y desventajas

Ventajas

- **Alta escalabilidad**: Los Microservicios es un estilo arquitectónico diseñado para ser escalable, pues permite montar numerosas instancias del mismo componente y balancear la carga entre todas las instancias.
- **Agilidad**: Debido a que cada Microservicios es un proyecto independiente, permite que el componente tenga ciclo de desarrollo diferente del resto, lo que permite que se puedan hacer despliegues rápidos a producción sin afectar al resto de componentes.
- **Facilidad de despliegue**: Las aplicaciones desarrolladas como Microservicios encapsulan todo su entorno de ejecución, lo que les permite ser desplegadas sin necesidad de dependencias externas o requerimientos específicos de Hardware.
- **Testabilidad**: Los Microservicios son especialmente fáciles de probar, pues su funcionalidad es tan reducida que no requiere mucho esfuerzo, además, su naturaleza de exponer o brindar servicios hace que sea más fácil de crear casos específicos para probar esos servicios.

Microservicios

- **Fácil de desarrollar**: Debido a que los Microservicios tiene un alcance muy corto, es fácil para un programador comprender el alcance del componente, además, cada Microservicios puede ser desarrollado por una sola persona o un equipo de trabajo muy reducido.
- **Reusabilidad**: La reusabilidad es la médula espinal de la arquitectura de Microservicios, pues se basa en la creación de pequeños componentes que realice una única tarea, lo que hace que sea muy fácil de reutilizar por otras aplicaciones o Microservicios.
- **Interoperabilidad**: Debido a que los Microservicios utilizan estándares abiertos y ligeros para comunicarse, hace que cualquier aplicación o componente pueda comunicarse con ellos, sin importar en que tecnología está desarrollado.

Microservicios

Desventajas

- **Performance**: La naturaleza distribuida de los Microservicios agrega una latencia significativa que puede ser un impedimento para aplicaciones donde el performance es lo más importante, por otra parte, la comunicación por la red puede llegar a ser incluso más tardado que el proceso en sí.
- **Múltiples puntos de falla**: La arquitectura distribuida de los Microservicios hace que los puntos de falla de una aplicación se multipliquen, pues cada comunicación entre Microservicios tiene una posibilidad de fallar, lo cual hay que gestionar adecuadamente.
- **Trazabilidad**: La naturaleza distribuida de los Microservicios complica recuperar y realizar una traza completa de la ejecución de un proceso, pues cada Microservicio arroja de forma separa su traza o logs que luego deben de ser recopilados y unificados para tener una traza completa.
- **Madurez del equipo de desarrollo**: Una arquitectura de Microservicios debe ser implementada por un equipo maduro de desarrollo y con un tamaño adecuado, pues los Microservicios agregan muchos componentes que deben ser administrados, lo que puede ser muy complicado para equipo poco maduros.

Microservicios

Cuando debo de utilizar un estilo de Microservicios

Debido a que los Microservicios se han puesto muy moda en estos años, es normal ver que en todas partes nos incentivan a utilizarlos, sin embargo, su gran popularidad ha llegado a segar a muchos, pues alientan y justifican la implementación de microservicios para casi cualquier cosa y ante cualquier condición, muy parecido al anti-patrón Golden Hammer.

Microservicios

Nuevo concepto: Gold Hammer Anti-pattern
El anti patrón Gold Hammer nos habla del uso excesivo de la misma herramienta para resolver cualquier tipo de problema, incluso si hay otra que lo resuelve de mejor manera. Se suele utilizar la siguiente frase para describirlo "**Si todo lo que tienes es un martillo, todo parece un clavo**".

Los Microservicios son sin duda una excelente arquitectura, pero debemos tener en claro que no sirve para resolver cualquier problema, sobre todos en los que el performance es el objetivo principal.

Un error común es pensar que una aplicación debe de nacer desde el inicio utilizando Microservicios, lo cual es un error que lleva por lo general al fracaso de la aplicación, tal como lo comenta Martin Fowler en uno de sus artículos. Lo que él recomienda es que una aplicación debe nacer como un Monolítico hasta que llegue el momento que el Monolítico sea demasiado complicado de administrar y todos los procesos de negocio están muy maduros, en ese momento es cuando Martin Fowler recomienda que debemos de empezar a partir muestro Monolítico en Microservicios.

Microservicios

Puede resultar estúpido crear un Monolítico para luego partirlo en Microservicios, sin embargo, tiene mucho sentido. Cuando una aplicación nace, no se tiene bien definido el alcance y los procesos de negocio no son maduros, por lo que comenzar a fraccionar la lógica en Microservicios desde el inicio puede llevar a un verdadero caos, pues no se sabe a ciencia cierta qué dominio debe atacar cada Microservicio, además, Empresas como Netflix, que es una de las que más promueve el uso de Microservicios también concuerda con esto.

Dicho lo anterior y analizar cuando NO debemos utilizar Microservicios, pasemos a analizar los escenarios donde es factible utilizar los Microservicios. De forma general, los Microservicios se prestan más para ser utilizadas en aplicaciones que:

- Aplicaciones pensadas para tener un gran escalamiento
- Aplicaciones grandes que se complica ser desarrolladas por un solo equipo de trabajo, lo que hace complicado dividir las tareas si entrar en constante conflicto.
- Donde se busca agilidad en el desarrollo y que cada componente pueda ser desplegado de forma independiente.

Microservicios

Hoy en día es común escuchar que las empresas más grandes del mundo utilizan los Microservicios como base para crear sus aplicaciones de uso masivo, como es el caso claro de Netflix, Google, Amazon, Apple, etc.

Conclusiones

En esta sección hemos aprendido que los Microservicios son pequeños componentes que se especializan en realizar una sola tarea, lo hace bien y son totalmente desacoplados. Hemos analizado y comparado el estilo Monolítico con los Microservicios y hemos analizado las diferencias que existen entre estos dos estilos, sin embargo, el estilo de Microservicios es uno de los más complejos por la gran cantidad de componentes y patrones arquitectónicos necesario para implementarlo correctamente, es por ello que en la siguiente unidad hablaremos de varios de los patrones arquitectónicos que hacen posible este estilo arquitectónico.

Sin lugar a duda, este estilo arquitectónico requiere de un análisis mucho más profundo de lo que podríamos cubrir en este libro, pues este tema da para escribir varios libros solo sobre este tema, por lo que mi recomendación es que, una vez finalizado este libro, si quieres profundizar en este tema, busques alguna literatura especializada que te ayude a aterrizar muchos de los conceptos que trataré de transmitirte en este libro.

Respresentational State Transfer (REST)

REST se ha convertido sin lugar a duda en uno de los estilos arquitectónicos más utilizados en la industria, ya que es común ver que casi todas las aplicaciones tienen un API REST que nos permite interactuar con ellas por medio de servicios. ¿Pero qué es exactamente REST y por qué se ha vuelto tan popular?

Lo primero que tenemos que saber es que REST es un conjunto de restricciones que crean un estilo arquitectónico y que es común utilizarse para crear aplicaciones distribuidas. REST fue nombrado por primera vez por Roy Fielding en el año 2000 donde definió a REST como:

Nuevo concepto: REST

REST proporciona un conjunto de restricciones arquitectónicas que, cuando se aplican como un todo, enfatizan la escalabilidad de las interacciones de los componentes, la generalidad de las interfaces, la implementación independiente de los componentes y los componentes intermedios para reducir la latencia de interacción, reforzar la seguridad y encapsular los sistemas heredados.

— Roy Fielding

Respresentational State Transfer (REST)

Algo muy importante a tomar en cuenta es que **REST ignora los detalles de implementación** del componente y la sintaxis del protocolo para enfocarse en los roles de los componentes, las restricciones sobre su interacción con otros componentes y la representación de los datos. Por otro lado, abarca las restricciones fundamentales sobre los componentes, conectores y datos que definen la base de la arquitectura web, por lo tanto, podemos decir que la esencia de REST es comportarse como una aplicación basada en la red.

Como se estructura REST

REST describe 3 conceptos clave, que son: Datos, Conectores y Componentes, los cuales trataremos de definir a continuación.

Respresentational State Transfer (REST)

Datos

Uno de los aspectos más importantes que propone REST es que los datos deben de ser transmitidos a donde serán procesados, en lugar de que los datos sean transmitidos ya procesados, pero ¿qué quiere decir esto exactamente?

En cualquier arquitectura distribuida, son los componentes de negocio quien procesa la información y nos responde la información que se produjo de tal procesamiento, por ejemplo, una aplicación web tradicional, donde la vista es construida en el backend y nos responde con la página web ya creada, con los elementos HTML que la componente y los datos incrustados. En este sentido, la aplicación web proceso los datos en el backend, y como resultado nos arroja el resultado de dicho procesamiento, sin embargo, REST lo que propone es que los datos sean enviados en bruto para que sea el interesado de los datos el que los procese para generar la página (o cualquier otra cosa que tenga que generar), en este sentido, lo que REST propone no es generar la página web, si no que mandar los datos en bruto para que el consumidor sea el responsable de generar la página a través de los datos que responde REST.

Respresentational State Transfer (REST)

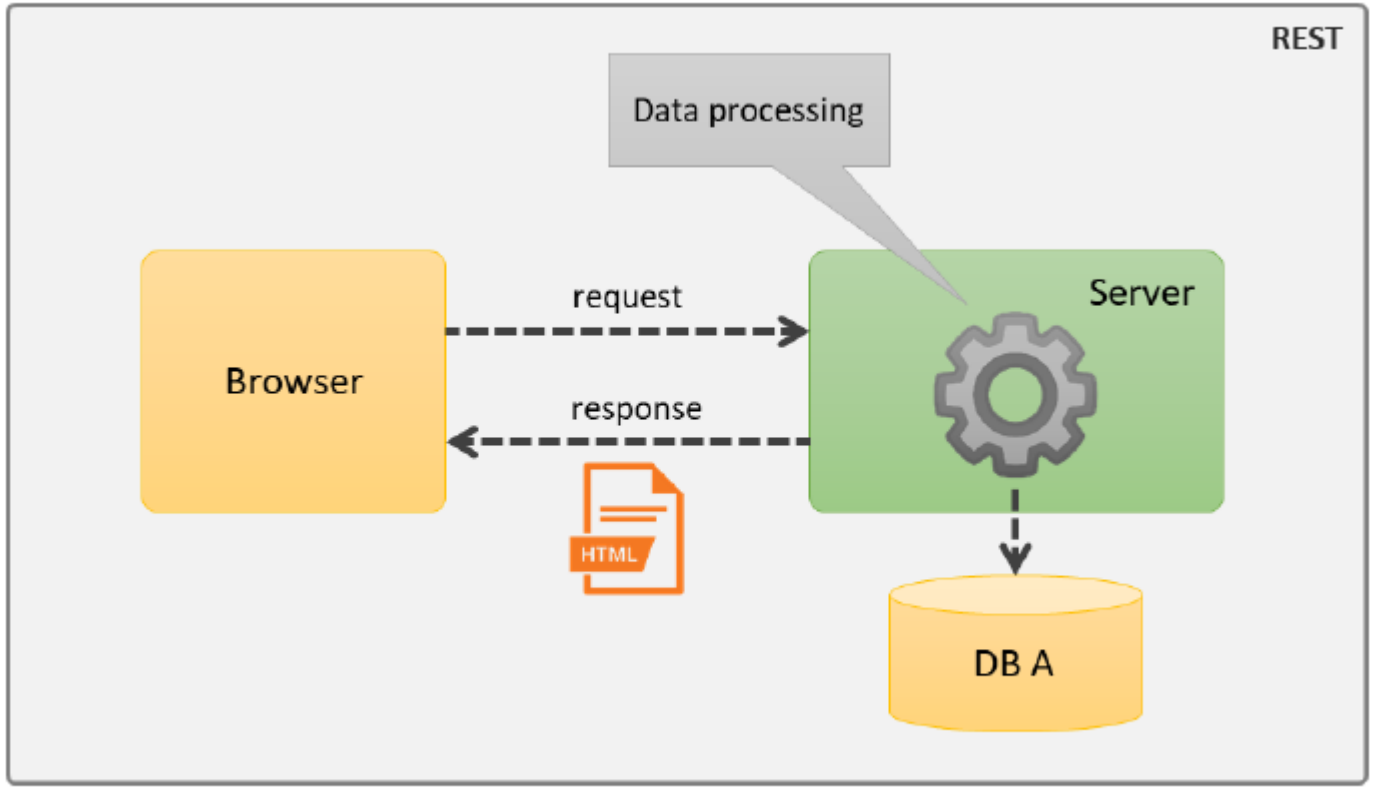

Fig 76 - Procesamiento del lado del servidor.

Respresentational State Transfer (REST)

En la imagen anterior podemos apreciar más claramente lo que comentábamos anteriormente, en el cual, los datos son procesados por el servidor y como resultado se obtiene un documento HTML procesado con todos los datos a mostrar.

Este enfoque fue utilizado durante mucho tiempo, pues las aplicaciones web requerían ir al servidor a consultar la siguiente página y este les regresa el HTML que el navegador solamente tenía que renderizar, sin embargo, que pasa con nuevas tecnologías como Angular, Vue, o React que se caracterizan por generar la siguiente página directamente en el navegador, lo que significa que no requieren ir al servidor por la siguiente página y en su lugar, solo requieren los datos del servidor presentarlo al usuario.

Bajo este enfoque podemos entender que REST propone servir los datos en crudo para que sea el consumidor quien se encargue de procesar los datos y mostrarlo como mejor le convenga.

Respresentational State Transfer (REST)

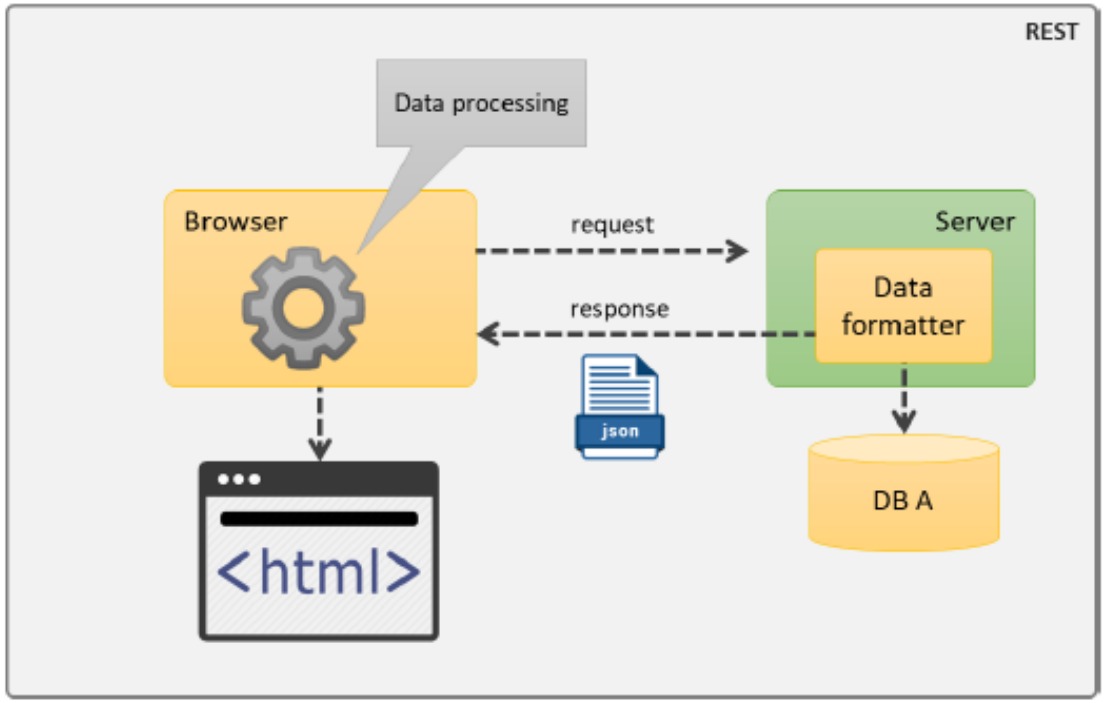

Fig 77 - Procesamiento del lado del cliente

Respresentational State Transfer (REST)

La imagen anterior representa mejor lo que acabamos de decir, pues ya vemos que el servidor solo se limita a servir la información en un formato esperado por el cliente, el cual por lo general es JSON, aunque podría existen más formatos. Por otra parte, el cliente recibe esa información, la procesa y genera la vista a partir de los datos proporcionados por el servidor, de esta forma, varias aplicaciones ya sean Web, Móviles, escritorio, o en la nube pueden solicitar esa misma información y mostrarla de diferentes formas, sin estar limitados por el formato procesado que regresa el servidor, como sería el caso de una página HTML.

- Ahora bien, el caso de la página HTML es solo un ejemplo, pero se podría aplicar para cualquier otra cosa que no sea una página, como mandar los datos a una app móvil, imágenes o enviarlos a otro servicio para su procesamiento, etc.

Respresentational State Transfer (REST)

Recursos

En REST se utiliza la palabra Recurso para hacer referencia a la información, y de esta forma, todos lo que podamos nombrar puede ser un recurso, como un documento, imagen, servicios, una colección de recursos, etc. En otras palabras, un recurso puede ser cualquier cosa que pueda ser recuperada por una referencia de hipertexto (URL)

Cada recurso tiene una dirección única que permite hacerle referencia, de tal forma que ningún otro recurso puede compartir la misma referencia, algo muy parecido con la web, donde cada página tiene un dominio y no pueden existir dos páginas web en el mismo dominio, o dos imágenes en la misma URL, de la misma forma, en REST cada recurso tiene su propia URL o referencia que hace posible recuperar un recurso de forma inconfundible.

Respresentational State Transfer (REST)

Representaciones

Una de las características de los recursos es que pueden tener diferentes representaciones, es decir, que un mismo recurso puede tener diferentes formatos, por ejemplo, una imagen que está representada por un recurso (URL) puede estar en formato PNG o JPG en la misma dirección, con la única diferencia de que los metadatos que describen el recurso cambian según la representación del recurso.

En este sentido, podemos decir que una representación consta de datos y metadatos que describen los datos. Los metadatos tienen una estructura de nombre-valor, donde el nombre describe el metadato y el valor corresponde al valor asignado al metadato. Por ejemplo, un metadato para representar una representación en formato JSON sería "Content-Type"="application/json" y el mismo recurso en formato XML sería "Content-Type"="application/xml".

Respresentational State Transfer (REST)

Conectores

Los conectores son los componentes que encapsulan la actividad de acceso a los recursos y transferencia, de esta forma, REST describe los conectores como **interfaces abstractas** que permite la comunicación entre compontes, mejorando la simplicidad al proporcionar una separación clara de las preocupaciones y

Respresentational State Transfer (REST)

ocultando la implementación subyacente de los recursos y los mecanismos de comunicación.

Por otro lado, REST describe que todas las solicitudes a los conectores sean sin estado, es decir, que deberán contener toda la información necesaria para que el conector comprenda la solicitud sin importar las llamadas que se hallan echo en el pasado, de esta forma, ante una misma solicitud deberíamos de obtener la misma respuesta. Esta restricción cumple cuatro funciones:

1. Eliminar cualquier necesidad de que los conectores retengan el estado de la aplicación entre solicitudes, lo que reduce el consumo de recursos físicos y mejora la escalabilidad.
2. Permitir que las interacciones se procesen en paralelo sin requerir que el mecanismo de procesamiento entienda la semántica de la interacción.
3. Permite a un intermediario ver y comprender una solicitud de forma aislada, lo que puede ser necesario cuando los servicios se reorganizan dinámicamente.
4. Facilita el almacenamiento en caché, lo que ayuda a reutilizar respuestas anteriores.

Respresentational State Transfer (REST)

En REST **existen dos tipos de conectores principales, el cliente y el servidor**, donde el cliente es quien inicial la comunicación enviando una solicitud al servidor, mientras que el servidor es el que escucha las peticiones de los clientes y responde las solicitudes para dar acceso a sus servicios.

Otro tipo de conector es el conector de caché, el cual puede estar ubicado en la interfaz de un conector cliente o servidor, el cual tiene como propósito almacenar las respuestas y reutilizarlas en llamas iguales en el futuro, evitando con ello, solicitar recursos consultados anteriormente al servidor. El cliente puede evitar hacer llamadas adicionales al servidor, mientras que el servidor puede evitar sobrecargar a la base de datos, disco o procesador con consultas repetitivas. En ambos casos se mejora la latencia.

Respresentational State Transfer (REST)

Componentes

Finalmente, los componentes con software concretos que utilizan los conectores para consultar los recursos o servirlos, ya sea una aplicación cliente como lo son los navegadores (Chrome, Firefox, etc) o Web Servers como Apache, IIS, etc. Pero también existen los componentes intermedios, que actúan como cliente y servidor al mismo tiempo, como es el caso de un proxy o túnel de red, los cuales actúan como servidor al aceptar solicitudes, pero también como cliente a redireccionar las peticiones a otro servidor.

Los componentes se pueden confundir con los conectores, sin embargo, un conector es una interface abstracta que describe la forma en que se deben de comunicar los componentes, mientras que un componente es una pieza de software concreta que utiliza los conectores para establecer la comunicación.

Respresentational State Transfer (REST)

Características de REST

En esta sección analizaremos las características que distinguen al estilo REST del resto. Las características no son ventajas o desventajas, si no que más bien, nos sirven para identificar cuando una aplicación sigue un determinado estilo arquitectónico.

Las características se pueden convertir en ventajas o desventajas solo cuando analizamos la problemática que vamos a resolver, mientras tanto, son solo características:

1. Todos los recursos deben de poder ser nombrados, es decir, que tiene una dirección (URL) única que la diferencia de los demás recursos.
2. Un recurso puede tener más de una representación, donde un recurso puede estar representado en diferentes formatos al mismo tiempo y en la misma dirección.
3. Los datos están acompañados de metadatos que describen el contenido de los datos.
4. Las solicitudes tienen toda la información para poder ser entendidas por el servidor, lo que implica que podrán ser atendidas sin importar las invocaciones pasadas (sin estado) ni el contexto de la misma.
5. Debido a la naturaleza sin estado de REST, es posible cachear las peticiones, de tal forma que podemos retornar un resultado previamente consultado sin necesidad de llamar al servidor o la base de datos.

Respresentational State Transfer (REST)

6. REST es interoperable, lo que implica que no está casado con un lenguaje de programación o tecnología especifica.

RESTful y su relación con REST

Sin lugar a duda, uno de los conceptos que más se confunden cuando hablamos de arquitecturas REST, es el concepto de RESTful, pues aún que podrían resultar similares, tiene diferencias fundamentales que las hacen distintas.

Como ya habíamos mencionado antes, REST parte de la idea de que tenemos datos que pueden tener diferentes representaciones y los datos siempre están acompañados de metadatos que describen el contenido de los datos, además, existen conectores que permiten que se pueda establecer una conexión, los cuales

Respresentational State Transfer (REST)

son abstractos desde su definición, finalmente, existen los componentes que son piezas de software concretas que utilizan los conectores para transmitir los datos.

Por otra parte, RESTful toma las bases de REST para crear servicios web mediante el protocolo HTTP, de esta forma, podemos ver que los conectores ya no son abstractos, si no que delimita que la comunicación siempre tendrá que ser por medio de HTTP, además, define que la forma para recuperar los datos es por medio de los diferentes métodos que ofrece HTTP, como lo son GET, POST, DELETE, PUT, PACHT. En este sentido, podemos observar que REST jamás define que protocolo utilizar, y mucho menos que debemos utilizar los métodos de HTTP para realizar las diferentes operaciones.

Entonces, podemos decir que RESTful promueve la creación de servicios basados en las restricciones de REST, utilizando como protocolo de comunicación HTTP.

También podemos observar que los servicios RESTful mantiene las propiedades de los datos que define REST, ya que en RESTful, todos los recursos deben de poder ser nombrados, es decir, deberán tener un URL única que pueda ser alcanzada por HTTP, por otra parte, define una serie de metadatos que describen el contenido de los datos, es por ello que con RESTful podemos retornar un XML, JSON, Imagen, CSS, HTML, etc. Cualquier cosa que pueda ser descrita por un metadato, puede ser retornada por REST.

Respresentational State Transfer (REST)

Ventajas y desventajas

Ventajas

- **Interoperable**: REST no está casado con una tecnología, por lo que es posible implementarla en cualquier lenguaje de programación.
- **Escalabilidad**: Debido a todas las peticiones son sin estado, es posible agregar nuevos nodos para distribuir la carga entre varios servidores.
- **Reducción del acoplamiento**: Debido a que los conectores definen interfaces abstractas y que los recursos son accedidos por una URL, es posible mantener un bajo acoplamiento entre los componentes.
- **Testabilidad**: Debido a que todas las peticiones tienen toda la información y que son sin estado, es posible probar los recursos de forma individual y probar su funcionalidad por separado.
- **Reutilización**: Una de los principios de REST es llevar los datos sin procesar al cliente, para que sea el cliente quién decida cuál es la mejor forma de representar los datos al usuario, por este motivo, los recursos de REST pueden ser reutilizado por otros componentes y representar los datos de diferentes maneras.

Respresentational State Transfer (REST)

Desventajas

- **Performance**: Como en todas las arquitecturas distribuidas, el costo en el performance es una constante, ya que cada solicitud implica costos en latencia, conexión y autenticaciones.
- **Más puntos de falla**: Nuevamente, las arquitecturas distribuidas traen con sigo ciertas problemáticas, como es el caso de los puntos de falla, ya que al haber múltiples componentes que conforman la arquitectura, multiplica los puntos de falla.
- **Gobierno**: Hablando exclusivamente desde el punto de vista de servicios basados en REST, es necesario mantener un gobierno sobre los servicios implementados para llevar un orden, ya que es común que diferentes equipos creen servicios similares, provocando funcionalidad repetida que tiene que administrarse.

Respresentational State Transfer (REST)

Cuando debo de utilizar el estilo REST

Actualmente REST es la base sobre la que están construidas casi todas las API de las principales empresas del mundo, con Google, Slack, Twitter, Facebook, Amazon, Apple, etc, incluso algunas otras como Paypal han migrado de los Webservices tradicionales (SOAP) para migrar a API REST.

En este sentido, REST puede ser utilizado para crear integraciones con diferentes dispositivos, incluso con empresas o proveedores externos, que requieren información de nuestros sistemas para opera con nosotros. Por ejemplo, mediante

API REST, Twitter logra que existan aplicaciones que puedan administrar nuestra cuenta, pueda mandar Tweets, puedan seguir personas, puedan retwittear y todo esto sin entrar a Twitter.

Otro ejemplo, es Slack, que nos permite mandar mensaje a canales, enviar notificaciones a un grupo si algo pasa en una aplicación externa, como Trello o Jira.

En Google Maps, podemos ir guardando nuestra posición cada X tiempo, podemos analizar rutas, analizar el tráfico, tiempo de llega a un destino, todo esto mediante el API REST, y sin necesidad de entrar a Google Maps para consultar la información.

Respresentational State Transfer (REST)

En otras palabras, REST se ha convertido prácticamente en la Arquitectura por defecto para integrar aplicaciones, incluso ha desplazado casi en su totalidad a los Webservices tradicionales (SOAP).

Entonces, cuando debo de utilizar REST, la respuesta es fácil, cuando necesitamos crear un API que nos permite interactuar con otra plataformas, tecnologías o aplicaciones, sin importar la tecnología y que necesitemos tener múltiples representaciones de los datos, como consultar información en formato JSON, consultar imágenes en PNG o JPG, envías formularios directamente desde un formulario HTML o simplemente enviar información binaria de un punto a otro, ya que REST permite describir el formato de la información mediante metadatos y no está casado un solo formato como SOAP, el cual solo puede enviar mensajes en XML.

Respresentational State Transfer (REST)

Conclusiones

Hemos hablado de que REST no es como tal una arquitectura de software, sino más bien es un conjunto de restricciones que en conjunto hacen un estilo de arquitectura de software distribuido que se centra en la forma en la que transmitimos los datos y las diferentes representaciones que estos pueden tener.

También hablamos de que REST no especifica ningún protocolo, sin embargo, HTTP es el protocolo principal de transferencia y el más utilizado por REST, concretamente los servicios RESTful.

REST no es para nada una estilo de arquitectura nuevo, pues la misma WEB es un ejemplo de REST, pero sí que se ha popularizado enormemente debido a la creciente necesidad de integrar aplicaciones de una forma simple, pero sobre todo, se ha disparado su uso debido a los servicios RESTful que llegaron para reemplazar los Webservices tradicionales (SOAP), los cuales si bien funciona bien, está limitado a mensajes SOAP en formato XML y que son en esencia mucho más difíciles de comprender por su sintaxis en XML y los namespace que un simple JSON.

Respresentational State Transfer (REST)

Por otra parte, los servicios RESTful en combinación con un estilo de Microservicios está dominando el mundo, ya que hoy en día todas las empresas buscan migrar a Microservicios o ya lo están, lo que hace que REST sea una de las arquitecturas más de moda de la actualidad.