



INTRODUCCIÓN A LA ARQUITECTURA DE SOFTWARE **UN ENFOQUE PRÁCTICO**

Oscar Javier Blancarte Iturralde

PRIMERA EDICIÓN



Introducción a la arquitectura de software – Un enfoque práctico

“La arquitectura de software no es el arte de hacer diagramas atractivos, sino el arte de diseñar soluciones que funcionen adecuadamente durante un periodo de tiempo razonable”

— Oscar Blancarte (2020)

Datos del autor:

Ciudad de México.

e-mail: oscarblancarte3@gmail.com

Autor y edición:

© **Oscar Javier Blancarte Iturralde**

Queda expresamente prohibida la reproducción o transmisión, total o parcial, de este libro por cualquier forma o medio; ya sea impreso, electrónico o mecánico, incluso la grabación o almacenamiento informáticos sin la previa autorización escrita del autor

Composición y redacción:

Oscar Javier Blancarte Iturralde

Edición:

Oscar Javier Blancarte Iturralde

Portada:

Arq. Jaime Alberto Blancarte Iturralde

Primera edición:

Se publicó por primera vez en enero del 2020

Acerca del autor

Oscar Blancarte es originario de Sinaloa, México donde estudió la carrera de Ingeniería en Sistemas Computacionales y rápidamente se mudó a la Ciudad de México donde actualmente radica.



Oscar Blancarte es Arquitecto de software con más de 15 años de experiencia en el desarrollo y arquitectura de software. Certificado como Java Programmer (Sun microsystems), Análisis y Diseño Orientado a Objetos (IBM) y Oracle IT Architect (Oracle). A lo largo de su carrera ha trabajado para diversas empresas del sector de TI, entre las que destacan su participación en diseños de arquitectura de software y consultoría para clientes de los sectores de Retail, Telco y Health Care. Oscar Blancarte es además autor de su propio blog <https://www.oscarblancarteblog.com> desde el cual está activamente publicando temas interesantes sobre Arquitectura de software y temas relacionados con la Ingeniería de Software en general. Desde su blog ayuda a la comunidad a resolver dudas y es por este medio que se puede tener una interacción más directa con el autor.

Además, es un apasionado por el emprendimiento, lo que lo ha llevado a emprender en diversas ocasiones, como es el caso de [Codmind](#), una plataforma de educación online, [Snipping Code](#), una plataforma de productividad para desarrolladores donde pueden guardar pequeños fragmentos de código repetitivos y [Reactive Programming](#), la plataforma en la cual publica sus libros e invita a otros desarrollar sus propias obras.

Otras obras del autor

Aplicaciones reactivas con React, NodeJS & MongoDB



Es un libro enfocado a enseñar a construir aplicaciones reactivas utilizando React, que va desde crear una aplicación WEB, hasta construir tu propio API REST utilizando NodeJS + Express y persistiendo toda la información con MongoDB, la base de datos NoSQL más popular de la actualidad.

Con este libro no solo aprenderás a construir aplicaciones WEB, sino que aprenderás todo lo necesario desde desarrollo hasta producción.

Durante todo el libro vamos desarrollando un proyecto final llamado Mini Twitter, el cual es una réplica de la red social Twitter, en donde utilizamos absolutamente todos los conceptos que se enseñan durante la lectura del libro.

[Ir al libro](#)

Introducción a los patrones de diseño



Es el libro más completo para aprender patrones de diseño, pues nos enfocamos en enseñar los 25 patrones más utilizados y desde una filosofía del mundo real, es decir, que todos los patrones están acompañados de un problema que se te podría presentar en alguno de tus proyectos.

Todos los patrones se explican con un escenario del mundo real, en el cual se plantea la problemática, como se solucionaría sin patrones de diseño y cómo sería la mejor solución utilizando patrones, finalmente, implementaremos la solución y analizaremos los resultados tras ejecutar cada aplicación.

[Ir al libro](#)

Agradecimientos

Este libro tiene una especial dedicación a mi esposa Liliana y mi hijo Oscar, quienes son la principal motivación y fuerza para seguir adelante todos los días, por su cariño y comprensión, pero sobre todo por apoyarme y darme esperanzas para escribir este libro.

A mis padres, quien con esfuerzo lograron sacarnos adelante, darnos una educación y hacerme la persona que hoy soy.

A todos los lectores anónimos de mi blog y todas aquellas personas que, de buena fe, compraron y recomendaron mis libros y fueron en gran medida quienes me inspiraron para escribir este tercer libro.

Finalmente, quiero agradecerte a ti, por confiar en mí y ser tu copiloto en esta gran aventura para aprender el Arte de la arquitectura de software y muchas cosas más :)

Prefacio

A lo largo de mi experiencia, me ha tocado convivir con muchas personas que se hacen llamar así mismas "*Arquitectos de software*" (las comillas son a propósito), aludiendo a su experiencia en el desarrollo de software o simplemente por haber aguantado muchos años en una empresa, lo que los hace sentirse merecedores de ese título, ya que según ellos, son los únicos que conocen todo el sistema, sin embargo, esto puede ser muy peligroso, pues un arquitecto no es una persona que tiene mucho tiempo en la empresa o el líder de desarrollo o una persona con mucha experiencia en el desarrollo de software.

Un verdadero arquitecto de software no es solo una persona con fuertes conocimientos técnicos, sino que debe de tener muchos de los denominados "*Soft Skills*" o habilidades blandas, que van desde el liderazgo, la toma de decisiones, innovación y toma de riesgos. Un arquitecto es una persona capaz de ofrecer soluciones innovadoras que solucionen problemas complejos, aun así, no es solo eso, un Arquitecto debe de poder justificar y defender sus argumentos, llegando al grado de tener que confrontar a altos directivos y hacerles ver la importancia de sus decisiones.

Si bien, este es un libro introductorio, es, sin duda, el libro en donde he puesto la mayor parte de mis conocimientos, pues son conocimientos que me tomó muchos años adquirir, madurar y perfeccionar, pues la arquitectura de software no es un conocimiento que se adquiera con una simple lectura o curso, sino que se van adquiriendo con la experiencia, experiencia que yo quiero compartirles y que es sin lugar a duda, el libro que yo hubiera querido tener cuando inicié en la arquitectura de software.

No quisiera entrar en más detalle en este momento, ya que a medida que profundicemos en el libro, analizaremos el rol de un arquitecto de software y su relación con la arquitectura de software. Analizaremos la diferencia que existe entre patrones de diseño, patrones arquitectónicos, los tipos de arquitectura y cómo es que todo esto se combina para crear soluciones de software capaces de funcionar durante grandes periodos de tiempo sin verse afectadas drásticamente por nuevos requerimientos.

Cómo utilizar este libro

Este libro es en lo general fácil de leer y digerir, su objetivo es enseñar todos los conceptos de forma simple y asumiendo que el lector tiene poco o nada de conocimiento del tema, así, sin importar quien lo lea, todos podamos aprender.

Como parte de la dinámica de este libro, hemos agregado una serie de tipos de letras que hará más fácil distinguir entre los conceptos importantes, código, referencias a código y citas. También hemos agregado pequeñas secciones de tips, nuevos conceptos, advertencias y peligros, los cuales mostramos mediante una serie de íconos agradables para resaltar a la vista.

Texto normal:

Es el texto que utilizaremos durante todo el libro, el cual no enfatiza nada en particular.

Negritas:

El texto en negritas es utilizado para enfatizar un texto, de tal forma que buscamos atraer tu atención, ya que se trata de algo importante.

Cursiva:

Lo utilizamos para hacer referencia a fragmentos de código como una variable, método, objeto o instrucciones de líneas de comandos. También se usa para resaltar ciertas palabras técnicas.

Código

Para el código utilizamos un formato especial, el cual permite colorear ciertas palabras especiales, crear el efecto entre líneas y agregar el número de línea que ayude a referenciar el texto con el código.

```
1. ReactDOM.render(  
2.   <h1>Hello, world!</h1>,  
3.   document.getElementById('root')  
4. );
```

El texto con fondo verde, lo utilizaremos para indicar líneas que se agregan al código existente.

~~Mientras que el rojo y tachado, es para indicar código que se elimina de un archivo existente.~~

Por otra parte, tenemos los íconos, que nos ayudan para resaltar algunas cosas:



Nuevo concepto: <concepto>

Cuando mencionamos un nuevo concepto o término que vale la pena resaltar.



Tip

Esta caja la utilizamos para dar un tip o sugerencia que nos puede ser de gran utilidad.



Importante

Esta caja se utiliza para mencionar algo muy importante.



Error común

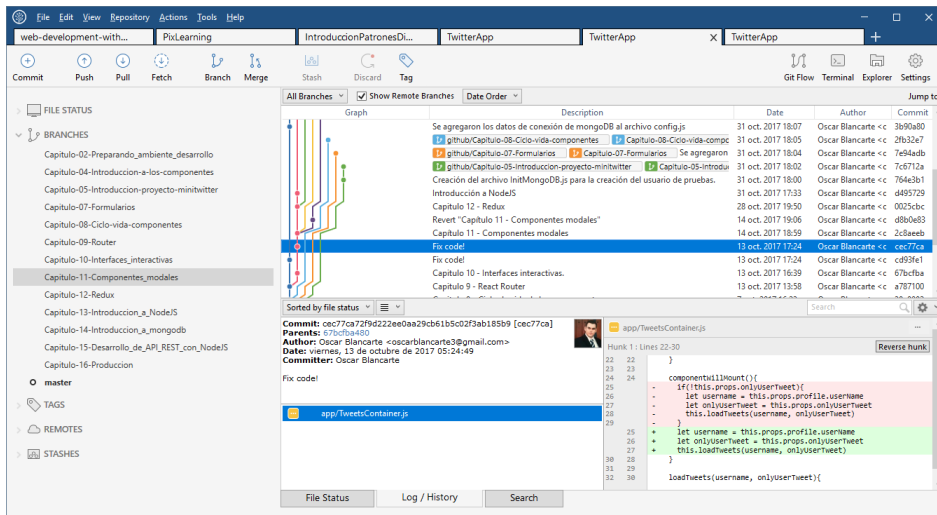
Esta caja se utiliza para mencionar errores muy comunes que pueden ser verdadero dolor de cabeza o para mencionar algo que puede prevenir futuros problemas.

Código fuente

Todo el código fuente de este libro está disponible en GitHub y lo puedes descargar en la siguiente URL:

<https://github.com/oscarjb1/introduction-to-software-architecture>

La segunda y más recomendable opción es, utilizar un cliente de Git, como lo es Source Tree y clonar el repositorio:



Requisitos previos

Si bien, he escrito este libro lo más simple y claro posible, cabe resaltar que este es un libro que requiere conocimientos sólidos en programación y el proceso de construcción de software en general.

Para comprender mejor todos los conceptos que explicaremos a lo largo de este libro también es necesario conocimiento sólido en patrones de diseño.

INTRODUCCIÓN

Desde los inicios de la ingeniería software, los científicos de la computación lucharon por tener formas más simples de realizar su trabajo, ya que las cuestiones más simples, como: imprimir un documento, guardar un archivo o compilar el código, eran tareas que podría tardar desde un día hasta una semana.

Solo como anécdota, yo recuerdo que uno de mis maestros de programación me comentó que ellos para poder ejecutar un programa que solo sumara dos números, era necesario escribir una gran cantidad de código, grabarlo en tarjetas perforadas y llevarlo al departamento de informática y esperar desde 1 día hasta una semana para que dieran el resultado de la compilación, y si faltaba una coma o un punto, se tenía que repetir todo el proceso.

Hoy en día, existen herramientas que van indicando en tiempo real si hay errores de sintaxis en el código, pero no solo eso, además, son lo suficientemente inteligentes para autocompletar lo que se va escribiendo, incluso, nos ayudan a detectar posibles errores en tiempo de ejecución.

La realidad es que, a medida que la tecnología avanza, tenemos cada vez más herramientas a nuestra disposición, como: lenguajes de programación, IDE's, editores de código, frameworks, librerías, plataformas en la nube y una gran cantidad de herramientas que hacen la vida cada vez más simple, así que, los retos de hoy en día no son compilar el código, imprimir una hoja, guardar en una base de datos, tareas que antes eran muy difíciles. Lo curioso es que, hoy en día hay tantas alternativas para hacer cualquier cosa, y por increíble que parezca, **el reto de un programador hoy en día es, decidirse por qué tecnología irse**, eso es increíble, tenemos tantas opciones para hacer lo que sea, que el reto no es hacer las cosas, sino decidir con que tecnologías lo construiremos.

Ahora bien, yo quiero hacerte una pregunta, ¿crees que hacer un programa hoy en días es más fácil que hace años?

Seguramente a todos los que les haga esta pregunta concordarán con que hoy en día es más fácil, sin embargo, y por increíble que parezca, el hecho de que las tecnologías sean cada vez más simples, nos trae nuevas problemáticas y justo aquí donde quería llegar.

A medida que las tecnologías son más simples y más accesibles para todas las personas del mundo, las aplicaciones se enfrentan a retos que antes no existían, como la concurrencia, la seguridad, la alta disponibilidad, el performance, la usabilidad, la reusabilidad, testabilidad, funcionalidad, modificabilidad, portabilidad, integridad, escalabilidad, etc, etc. Todos estos son conceptos que prácticamente no existían en el pasado, porque las aplicaciones se desarrollaban para una audiencia muy reducida y con altos conocimientos técnicos, además, se ejecutaban en un Mainframe, lo que reducía drásticamente los problemas de conectividad o intermitencia, pues todo se ejecuta desde el mismo servidor.



Nuevo concepto: Mainframe

Los Mainframe son súper computadores capaces de realizar millos de instrucciones por segundo (MIPS). Son utilizadas para el procesamiento de grandes volúmenes de datos y pueden estar varios Mainframe conectados entre sí en la misma ubicación física (site).

Seguramente has escuchado alguna vez eso que dicen las mamás de hoy en día, “¡Ay!, los niños de hoy son más inteligentes, desde bebés aprender a utilizar las tablets o las computadoras”. La realidad no es que los niños de ahora son más inteligentes, ya que el ser humano siempre ha sido inteligente, lo que pasa es que, cada vez se desarrollan aplicaciones con una usabilidad mejor y son altamente intuitivas, lo que ocasiona que cualquier persona pueda utilizarlas, incluso si conocimientos.

Pues bien, debido a todos estos cambios, más la llegada de nuevas arquitecturas como Cloud computing y SOA, Microservicios, REST, es que nace la necesidad del arquitecto de software, el cual tiene como principal responsabilidad asegurarse de que una aplicación cumpla con los **atributos de calidad**, entre otras responsabilidades más, como dictar los estándares de codificación, herramienta, plataformas y tecnologías, entre otras cosas que analizaremos más adelante.

Para los que no les quede claro que es un atributo de calidad, son por lo general los requerimientos no funcionales.

¿Alguien aquí ha escuchado que son los requerimientos funcionales y no funcionales?

**Nuevo concepto: Requerimientos funcionales**

los requerimientos funcionales son todos aquellos que nacen de la necesidad de los usuarios para cubrir un requerimiento de negocio y son solicitados explícitamente, como, por ejemplo, que el sistema me permite registrar clientes, crear facturas, administrar el inventario, etc.

**Nuevo concepto: Requerimientos no funcionales**

los requerimientos no funcionales son aquellos que no son solicitados explícitamente por los usuarios, pero van implícitos en un requerimiento, por ejemplo, que la aplicación sea rápida, segura y que pueda escalar si el número de usuarios aumenta, etc.

**TIP: Los atributos de calidad son por lo general los requerimientos no funcionales**

Si bien, analizaremos con detalles los atributos de calidad más adelante, imagina que los atributos de calidad son los requerimientos no funcionales, y que si bien, el cliente no te los solicitará, su ausencia podría implicar el fracaso de proyecto.

Entonces, un arquitecto deberá siempre cuidar que la aplicación cumpla con los atributos de calidad, y será su responsabilidad que una solución no solo cumpla los requerimientos solicitados explícitos (requerimientos funcionales), sino que, además, deberá cuidar los requerimientos no funcionales, incluso, si el cliente no los ha solicitado, pues su cumplimiento garantizará el funcionamiento de la aplicación por un largo periodo de tiempo.

Índice

Agradecimientos.....	7
Prefacio	8
Cómo utilizar este libro.....	9
Código fuente	11
Requisitos previos	12
INTRODUCCIÓN	13
Índice.....	16
Por dónde empezar	22
¿Qué es la arquitectura de software?	23
¿Qué son los patrones de diseño?	27
Tipos de patrones de diseño.....	29
¿Cómo diferenciar un patrón de diseño?	29
¿Dónde puedo aprender patrones de diseño?	31
¿Qué son los patrones arquitectónicos?.....	33
¿Cómo diferenciar un patrón arquitectónico?	35
¿Qué son los estilos arquitectónicos?	37
La relación entre patrones de diseño, arquitectónicos y estilos arquitectónicos	39
Comprendiendo algunos conceptos.....	42
Encapsulación.....	43
Acoplamiento	45
Cohesión	47
Don't repeat yourself (DRY).....	49
Separation of concerns (SoC).....	50
La Ley de Demeter.....	52
Keep it simple, Stupid! (KISS).....	54
Inversion of control (IoC)	55

<i>S.O.L.I.D</i>	57
Single responsibility principle (SRP).....	57
Open/closed principle (OCP)	58
Liskov substitution principle (LSP)	59
Interface segregation principle (ISP)	61
Dependency inversion principle (DIP)	64
Atributos de calidad	68
<i>Importancia de los atributos de calidad</i>	70
<i>Clasificación de los atributos de calidad</i>	70
<i>Atributos de calidad observables</i>	71
Performance (rendimiento).....	72
Security (Seguridad)	76
Availability (disponibilidad)	79
Functionality (funcionalidad).....	82
Usabilidad	83
<i>No observables</i>	85
Modificabilidad.....	85
Portabilidad	87
Reusabilidad	89
Testabilidad	93
Escalabilidad	97
Estilos arquitectónicos	104
<i>Monolítico</i>	107
Como se estructura un Monolítico	108
Características de un Monolítico	109
Ventajas y desventajas	110
Cuando debo de utilizar un estilo Monolítico	112
Conclusiones.....	112
<i>Cliente-Servidor</i>	114
Como se estructura un Cliente-Servidor	116
Flujo de Comunicación entre el cliente y el servidor.....	117
Características de Cliente-Servidor	121
Ventajas y desventajas	121
Cuando debo de utilizar un estilo Cliente-Servidor.....	123
Conclusiones.....	124
<i>Peer-to-peer (P2P)</i>	126
Como se estructura P2P	127
Características de una arquitectura P2P	138
Ventajas y desventajas	139
Cuando debo de utilizar un estilo P2P.....	140

Conclusiones.....	143
<i>Arquitectura en Capas.....</i>	<i>144</i>
Como se estructura una arquitectura en capas.....	145
Capas abiertas y cerradas.....	148
Arquitectura en 3 capas.....	150
Características de una arquitectura en capas.....	155
Ventajas y desventajas	155
Cuando debo de utilizar una arquitectura en capas.....	156
Conclusiones.....	158
<i>Microkernel</i>	<i>160</i>
Como se estructura una arquitectura de Microkernel	161
Características de una arquitectura de Microkernel	165
Ventajas y desventajas	167
Cuando debo de utilizar una arquitectura de Microkernel	168
Conclusiones.....	169
<i>Service-Oriented Architecture (SOA)</i>	<i>171</i>
Como se estructura una arquitectura de SOA	172
Características de SOA.....	181
Ventajas y desventajas	181
Cuando debo de utilizar un estilo SOA	182
Conclusiones.....	183
<i>Microservicios.....</i>	<i>185</i>
Como se estructura un Microservicios	186
Escalabilidad Monolítica vs escalabilidad de Microservicios	188
Características de un Microservicio.....	190
Ventajas y desventajas	191
Cuando debo de utilizar un estilo de Microservicios.....	192
Conclusiones.....	194
<i>Event Driven Architecture (EDA).....</i>	<i>195</i>
Como se estructura una arquitectura EDA	198
Características de una arquitectura EDA.....	201
Ventajas y desventajas	202
Cuando debo de utilizar un estilo EDA	203
Conclusiones.....	205
<i>Representational State Transfer (REST).....</i>	<i>207</i>
Como se estructura REST.....	208
Características de REST	214
RESTful y su relación con REST	214
Ventajas y desventajas	215
Cuando debo de utilizar el estilo REST	216
Conclusiones.....	218

Proyecto E-Commerce.....	220
El Proyecto E-Commerce	221
Instalación	234
Iniciar la aplicación	264
Cómo utiliza la aplicación	278
Patrones arquitectónicos	284
<i>Data Transfer Object (DTO)</i>	286
Problemática	286
Solución	291
Converter pattern.....	293
DTO en el mundo real.....	300
Conclusiones.....	309
<i>Data Access Object (DAO)</i>	312
Problemática	312
Solución	314
DAO y el patrón Abstract Factory	318
DAO en el mundo real	323
Conclusiones.....	333
<i>Polling</i>	334
Problemática	334
Solución	335
Polling sobre un servidor FTP	337
Polling sobre una base de datos.....	338
Consulta del status de un proceso	339
Monitoreo de servicios.....	340
Polling en el mundo real.....	342
Conclusiones.....	354
<i>Webhook</i>	355
Problemática	355
Solución	356
Dos formas de implementar Webhook	359
Reintentos	360
Webhook en el mundo real.....	361
Conclusiones.....	374
<i>Load Balance</i>	375
Problemática	375
Solución	376
Principales algoritmos de balanceo de cargas.....	380
Productos para balanceo de carga	389
Load Balance en el mundo real	389
Conclusiones.....	403

<i>Service Registry</i>	404
Problemática.....	404
Solución	405
Service Registry en el mundo real	408
Conclusión	418
<i>Service Discovery</i>	419
Problemática.....	419
Solución	423
Service Discovery en el mundo real.....	427
Conclusiones.....	432
<i>API Gateway</i>	433
Problemática.....	433
Solución	437
API Gateway en el mundo real	441
Conclusiones.....	452
<i>Access token</i>	453
Problemática.....	453
Solución	455
Access Token en el mundo real	464
Conclusiones.....	481
<i>Single Sign On (Inicio de sesión único)</i>	482
Problemática.....	482
Solución	483
SSO en el mundo real	487
Comentarios adicionales	493
Conclusiones.....	493
<i>Store and forward</i>	495
Problemática.....	495
Solución	496
Store and Forward en el mundo real.....	500
Conclusiones.....	509
<i>Circuit Breaker</i>	510
Problemática.....	510
Solución	511
Circuit Breaker en el mundo real	516
Conclusiones.....	529
<i>Log aggregation</i>	530
Problemática.....	530
Solución	533
Log aggregation en el mundo real	537
Conclusiones.....	548

Conclusiones 549

Por dónde empezar

Capítulo 1

Para muchos, dominar la arquitectura de software es uno de los objetivos que han buscado durante algún tiempo, sin embargo, no siempre es claro el camino para dominarla, ya que la arquitectura de software es un concepto abstracto que cada persona lo interpreta de una forma diferente, dificultando con ello comprender y diseñar con éxito una arquitectura de software.

Para iniciar este libro y comprender mejor todos los conceptos que analizaremos será necesario que abramos la mente y nos borremos esa tonta idea de que un arquitecto es aquella persona que tiene más tiempo en la empresa, o la que conoce todo el sistema, o el que desarrolla la solución, todas esas cosas nos hacen caer en nuestros laureles y nos hacen pensar que ya somos arquitectos y que nadie más puede hacerlo mejor. Dicho lo anterior, aprendamos desde cero los conceptos básicos:

¿Qué es la arquitectura de software?

Dado que este es un libro de arquitectura de software, lo más normal es preguntarnos; entonces **¿Qué es la arquitectura de software?** Sin embargo, temo decirte que no existe un consenso para definir con exactitud que es la arquitectura de software, por el contrario, existe diversas publicaciones donde se dan definiciones diferentes, lo que hace complicado decir con exactitud que es la arquitectura de software, por ello, vamos a analizar algunas definiciones para tratar de buscar la definición que usaremos en este libro.

Cabe mencionar que las definiciones que analizaremos a continuación son solo alguna de las muchísimas definiciones que existen, te invito a que veas el artículo publicado por el Software Engineering Institute titulado *"What Is Your Definition of Software Architecture"* el cual recopila una gran cantidad de definiciones realizadas por personas, publicaciones e instituciones. Te dejo el link a continuación:

<https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513807>

Veamos alguna de las definiciones más relevantes sobre que es la arquitectura de 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)

"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

"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

"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

"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

Como podemos observar, existen muchas definiciones sobre que es la arquitectura de software, lo que hace complicado desde un inicio dar una definición exacta y que deje conformes a todos, sin embargo, como arquitectos de software estamos forzados a poder dar una definición, lo que nos deja dos opciones, adoptar una de las existente o creamos nuestra propia definición basado en las anteriores.

Antes de elegir la que sería la definición que utilizaremos en este libro, quiero que analicemos las definiciones anteriores, podrás observar que, si bien todas son diferentes, todas coinciden en que la arquitectura se centra en la estructura del sistema, los componentes que lo conforman y la relación que existe entre ellos. Esto quiere decir que sin importar que definición utilicemos, debe de quedar claro que la arquitectura de software se centra en **la estructura del sistema, los componentes que lo conforman y la relación que existe entre ellos**.

Dicho lo anterior, a mí en lo particular me agrada la definición de Wikipedia, pues es sumamente minimalista, clara y concisa. Sin embargo, yo le agrego algo más para dejarla de la siguiente manera:



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.

Cabe mencionar que esta es la definición que para mí más se acerca a lo que es la arquitectura de software y que estará utilizado a lo largo de este libro para

definirla, sin embargo, siéntete libre de adoptar una existente o hacer tu propia definición.

¿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.



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.

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."

Más tarde, Christopher Alexander y sus colegas publicaron el volumen *A Pattern Language* en donde intentaron formalizar y plasmar de una forma práctica las generaciones de conocimientos arquitectónicos. En la obra se refieren a los patrones arquitectónicos de la siguiente manera:

Los patrones no son principios abstractos que requieran su redescubrimiento para obtener una aplicación satisfactoria, ni son específicos a una situación particular o cultural; son algo intermedio. Un patrón define una posible solución correcta para un problema de diseño dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones. (1977).

Entre las cosas que se describen en el libro se encuentra la forma de diseñar la red de transporte público, carreteras, en qué lugar deben ir las perillas de las puertas, etc.

Hasta ese momento los patrones conocidos tenían un enfoque arquitectónico y hablan de cómo construir estructuras, pero fue hasta 1987 cuando Ward Cunningham y Kent Beck, motivados por el pobre entrenamiento que recibían los nuevos programadores en programación orientada a objetos, se dieron cuenta de la gran semejanza que existían entre una buena arquitectura propuesta por Christopher Alexander y la buena arquitectura de la programación orientada a objetos. De tal manera que utilizaron gran parte del trabajo de Christopher para diseñar cinco patrones de interacción hombre-máquina y lo publicaron en el artículo OOPSLA-87 bajo el título *Using Pattern Languages for OO Programs*.

Sin embargo, fue hasta principios de la década de 1990 cuando los patrones de diseño tuvieron su gran debut en el mundo de la informática a partir de la publicación del libro *Design Patterns*, escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, en el que se recogían 23 patrones de diseño comunes que ya se utilizaban sin ser reconocidos como 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 Estructurales: Son patrones que tiene que ver con la forma en que las clases se relacionan con otras clases. Estos patrones ayudan a dar un mayor orden a nuestras clases ayudando a crear componentes más flexibles y extensibles.

Patrones de Comportamiento: Son patrones que están relacionados con procedimientos y con la asignación de responsabilidad a los objetos. Los patrones de comportamiento engloban también patrones de comunicación entre ellos.

¿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.

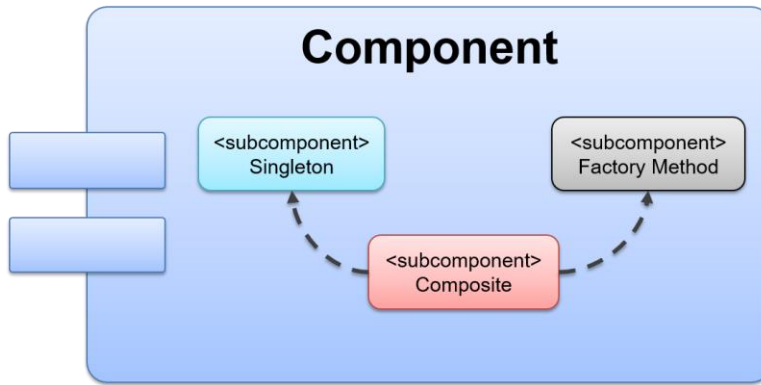


Fig 1: Muestra un componente con 3 patrones de diseño implementados.

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.

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.

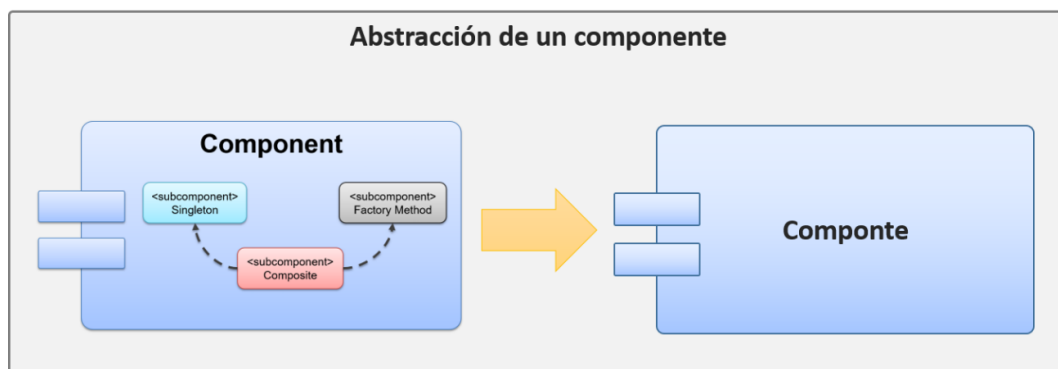


Fig 2: Abstracción de un componente



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.

¿Dónde puedo aprender patrones de diseño?

Como comentamos hace un momento, este libro no se centra en la enseñanza de los patrones de diseño, sino al contrario, es un pre-requisito para poder comprender mejor este libro.

Si quieres regresar un paso atrás y aprender de lleno los patrones de diseño, te invito a que veas mi otro libro “**Introducción a los patrones de diseño – Un enfoque práctico**” en el cual explico de lleno los principales patrones de diseño con ejemplos del mundo real, los cuales podrás descargar y ejecutar directamente en tu equipo.

El libro lo podrás ver en: <https://reactiveprogramming.io/books/design-patterns/es>



Introducción a los **patrones** de diseño

— Un enfoque práctico

El único libro que te enseña patrones de diseño con ejemplos del mundo real.

¿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

"Expresa una organización estructural fundamental o esquema para sistemas de software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y pautas para organizar las relaciones entre ellos. "

— TOGAF

"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."

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

"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

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.



Los patrones de diseño son diferentes a los patrones arquitectónicos.

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

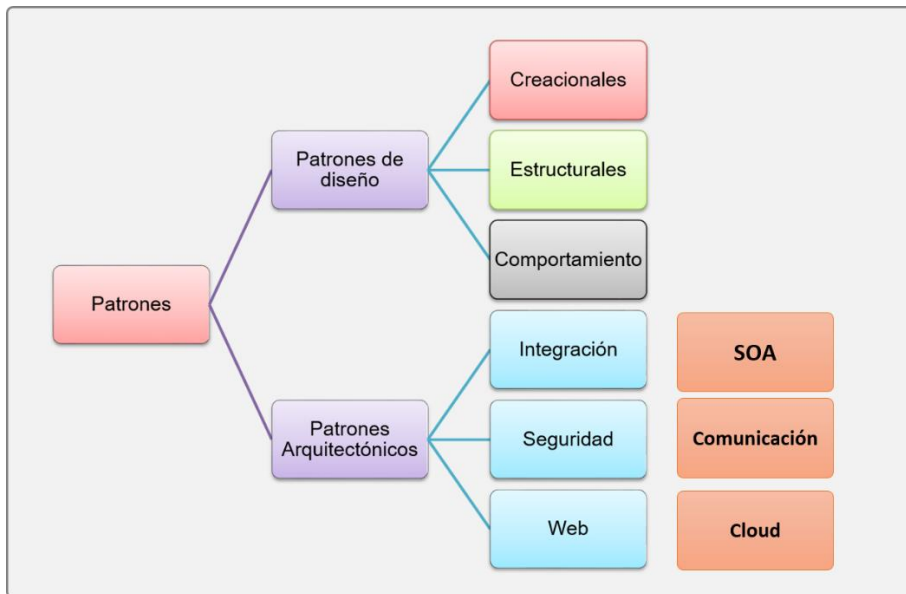


Fig 3: Tipos de patrones

La siguiente imagen nos ayudará a entender un poco mejor los distintos patrones de diseño y patrones arquitectónicos que existen para ayudarnos a diferenciarlos mejor. Cabe mencionar que existen muchos más tipos de patrones arquitectónicos que los que hay en la imagen.

Más adelante tendremos un capítulo dedicado exclusivamente para analizar los principales 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.

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

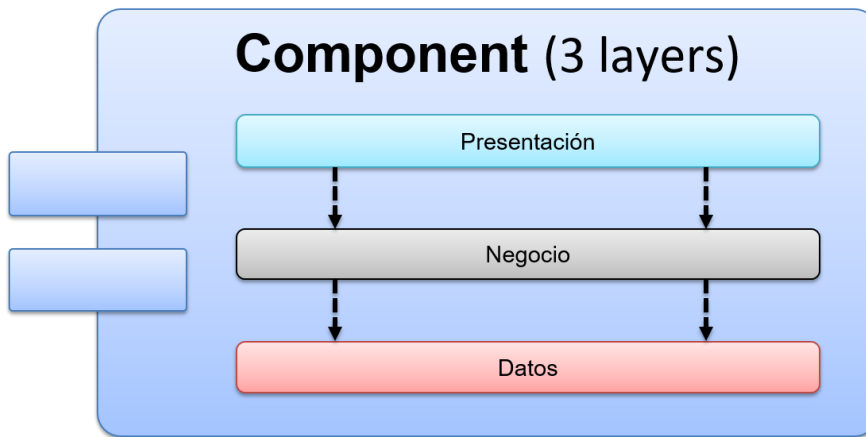


Fig 4: Arquitectura en 3 capas

En la imagen anterior podemos apreciar un componente 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.

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

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, componentes externos que están fuera de nuestro dominio, lo que complicaría aún más las cosas.

¿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.**

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:

"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

"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.

Nuevamente podemos observar que no existe una única definición para describir lo que es un estilo arquitectónico, lo que hace difícil tener una definición de referencia. En mi caso, me gusta decir que un estilo arquitectónico es:



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.

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:

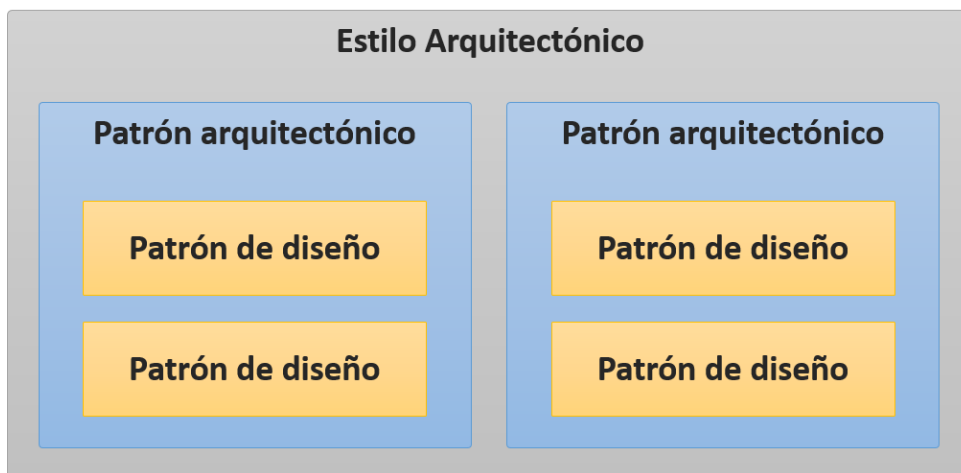


Fig 5: La relación entre patrones de diseño, patrones arquitectónicos y estilos arquitectónicos

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.



Estilo arquitectónico

Los estilos arquitectónicos no existen para resolver un problema de diseño, en su lugar, sirven para nombrar un diseño arquitectónico recurrente.

A pesar de que estos 3 conceptos tengan un propósito diferente (aunque relacionado), es importante que cualquier arquitecto de software entienda la diferencia, pues confundirlos puede ser un error grave, es por ello que en este libro nos centraremos en aprender de lleno los principales estilos y patrones arquitectónicos, dejando de lado los patrones de diseño, pues asumimos que a estas alturas ya los dominas.

Si quieres profundizar en los patrones de diseño, te puedo recomendar que revises mi libro de “**Introducción a los patrones de diseño – un enfoque práctico**”, en el cual explicamos los 25 principales patrones de diseño utilizando ejemplos del mundo real.

Comprendiendo algunos conceptos

Capítulo 2

Este capítulo lo dedicaremos a analizar algunos de los conceptos básicos que todo arquitecto debe de conocer, estos conceptos pueden resultar para muchos, algo muy básico, pero son los principales principios en los que deberíamos basarnos para crear mejores arquitecturas.

Los principios que explicaremos son importantes, debido a que muchos de los patrones y estilos arquitectónicos que analizaremos en este libro se basan en estos principios, por lo que es importante dominarlos antes de pasar temas más avanzados.

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.

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.

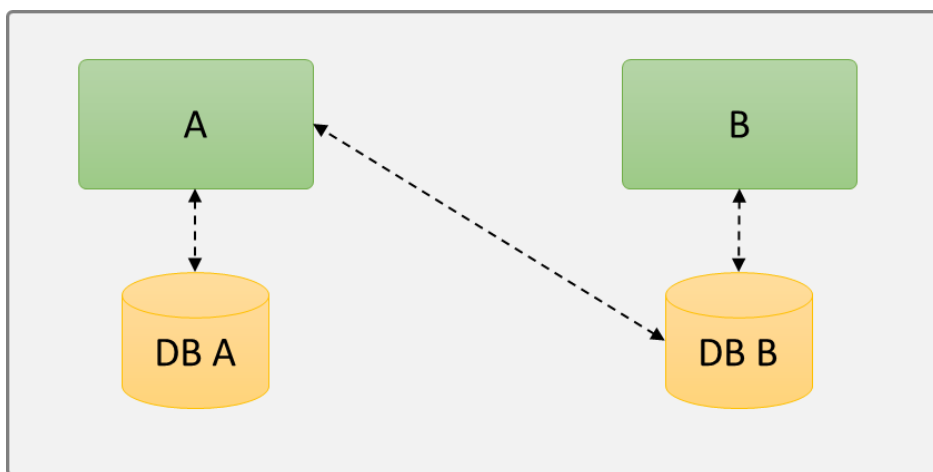


Fig 6 - Encapsulamiento omitido.

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.

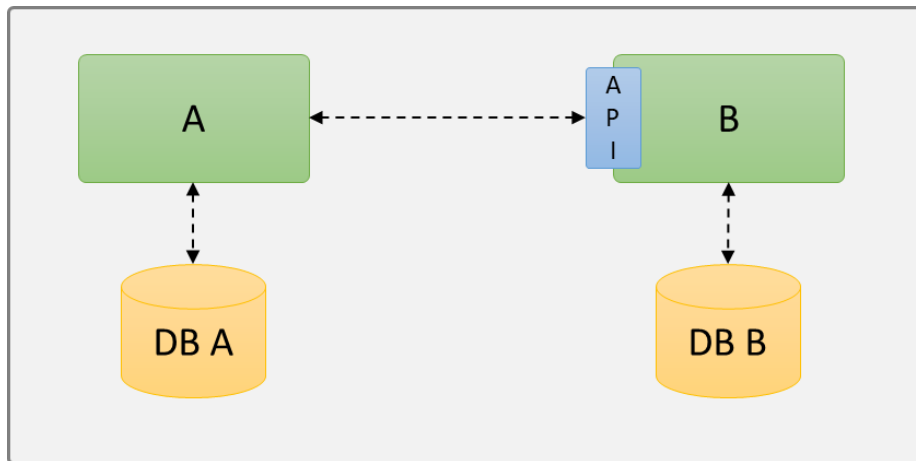


Fig 7 - Cumpliendo con el encapsulamiento.

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, ocultando todos aquellos detalles que son irrelevantes para la arquitectura.

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**.

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 lo 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.

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**.

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.

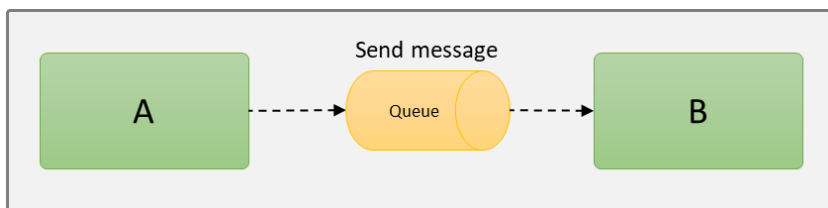


Fig 9 - Bajo acoplamiento

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.



Bajo acoplamiento

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

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.

En la práctica, se busca que todos los componentes 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.

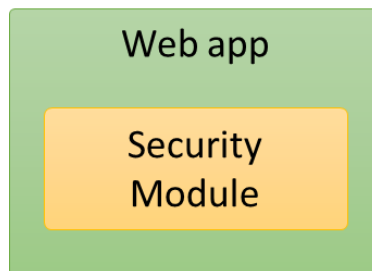


Fig 10 - Baja cohesión

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 mezclando 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

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.

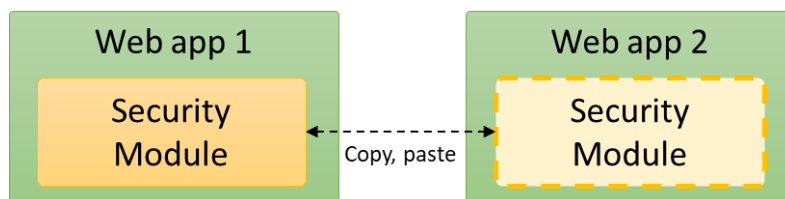


Fig 11 - Copy-Paste code

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:

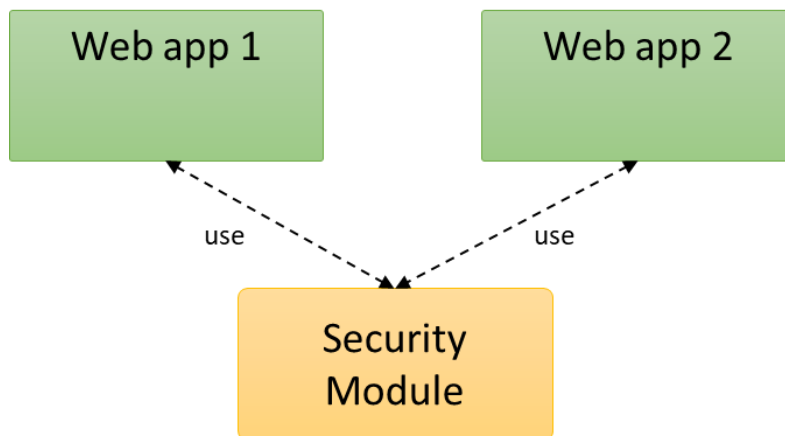


Fig 12 - Alta cohesión

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

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).

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 de 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

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.

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.

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:

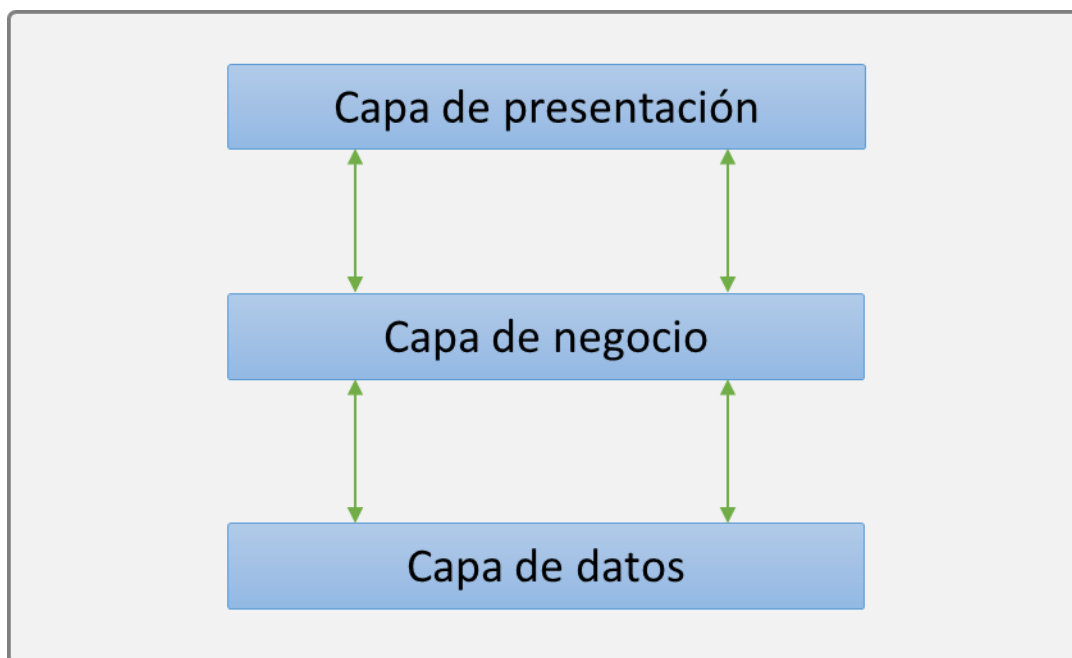


Fig 13 - Arquitectura de 3 capas

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

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.

La Ley de Demeter

La ley de Demeter o “Principio de menor conocimiento” es uno de los principios básicos de la programación orientada a objetos, la cual nos dice que debemos conocer lo menor posible de una clase, de tal forma que evitemos conocer las entrañas del componente al cual nos estamos acoplando.

La ley de demeter es muy parecida a la famosa frase que nos decían nuestros padres “**No hables con extraños**”, y es que lo que promueve es que solo hablemos con las propiedades directas del componente a comunicarnos, sin introducirnos demasiado en sus propiedades internas. Veamos el siguiente ejemplo:

```
1. String country = invoice.getCustomer().getAddress().getCountry();
```

En este ejemplo, vemos que tenemos un objeto factura (invoice), el cual tiene dentro un cliente (customer), el cual, a su vez, tiene un domicilio (address), el cual a su vez tiene un país (country). En este sentido, podríamos decir que la factura es un conocido, y al cliente lo conocemos por la factura; hasta este punto podemos decir que estamos hablando con un conocido, pero al momento que solicitamos la dirección al cliente, estamos hablando con el conocido de un conocido, es decir, para nosotros ya es un extraño; de la misma forma, el país también sería un desconocido. Por esta razón la ley de demeter nos dice que debemos evitar hablar con los desconocidos, pues a medida que más conocemos la estructura del componente, más nos acoplamos a él.

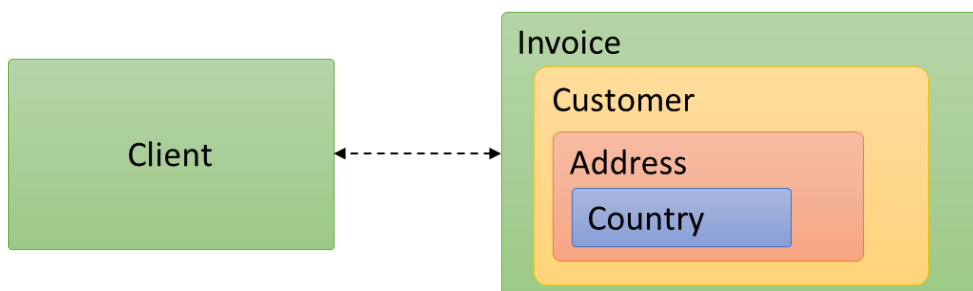


Fig 14 - Infracción de la ley de demeter

Solo piénsalo, entre más dependemos de la estructura de la factura, somos más susceptibles a cualquier cambio en el funcionamiento de esta, muy diferente a que, si solo habláramos con los "conocidos", pues nuestro nivel de acoplamiento disminuiría.

Esta ley es como la famosa frase que usan en la mafia, "Entre menos sepas mejor" o "Entre menos sepas estarás más seguro" y es que entre más conozcamos la estructura, más implicados estaremos y acoplados estaremos.

Ahora bien, en su lugar, siempre será mejor conocer lo menos posible y tener una representación más minimalista del componente:

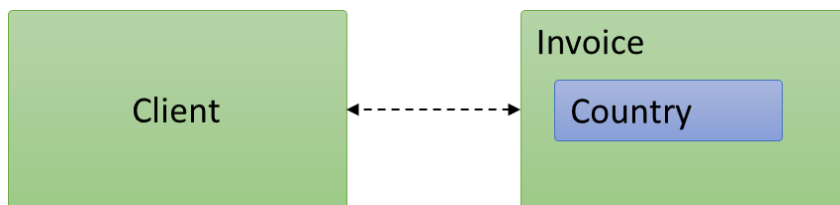


Fig 15 - Ley de demeter

Esta vista más simple del componente puede evitar conocer lo menos posible del componente. El patrón DTO puede ser una buena forma de hacer representaciones más simples (hablaremos de el más adelante).

Sé que este ejemplo aplica solo para código, sin embargo, en la arquitectura de software pasa algo similar, ya que entre más conocemos los detalles de su implementación, más nos acoplamos a ellos, por ejemplo, si utilizamos un protocolo específico, utilizamos librerías propietarias o infraestructura del proveedor, más nos acoplamos a ellos y con ello, nos volvemos más dependientes.

Keep it simple, Stupid! (KISS)

Por increíble que parezca, tendemos a pensar que entre más complejo y elabora sea el software será mucho mejor, sin embargo, este principio nos dice lo contrario. Este principio que se traduce a **¡Mantenlo sencillo, estúpido!** Puede resultar ofensivo, sin embargo, lo que nos quiere decir es que “La mejor solución no es la más elaborada o compleja, sino la más simple”, y trata de darnos a entender que la simplicidad es la clave del éxito.

Muchas veces, cuando pensamos en una solución, tratamos de hacer en arquitectura muy elaboradas, que usen la nube y luego estos se conectan a una base de datos NoSQL, para terminar descargando los resultados en un archivo plano que importamos en otra aplicación, que, a su vez, manda un mail cuando algo sale mal. Pero tal vez el usuario solo requería pasar una información de un punto A al B y listo, sin embargo, como arquitectos, queremos lucirnos un poco y proponer las tecnologías más nuevas o innovadoras, cuando quizás una app en Visual Basic podía haber resuelto el problema perfectamente.

Este no es un principio que tenga una serie de pasos o acciones a seguir, sino más bien nos invita a reflexionar si lo que estamos haciendo es realmente necesario y si no estamos complicando las cosas más de lo que deberíamos.

Ahora bien, hacer las cosas simples, no es fácil, pues hacerlo simple no significa que dejemos de hacer cosas o que quitemos funcionalidad que el usuario requiera, sino más bien, idear como hacer que todo sea lo más fácil e intuitivo para el usuario, como reducir el número de pasos para terminar una transacción, reducir el número de campo solicitados, reducir el número de actores involucrados en un proceso, reducir el número de trámites, etc.

Dos de los ejemplos más exitoso de KISS son Google y Apple. Google invento un buscador súper minimalista, al entrar solo aparecía una barra de búsqueda y el logotipo, y eso era todo, los usuarios podían encontrar lo que buscaban si saber absolutamente nada de buscadores.

Por su parte Steve Jobs invento un teléfono súper intuitivo, con teclado integrado, multitouch y navegador integrado, lo que nos demuestra que hacer las cosas simples no es fácil, si no requiere conocer a tus usuarios para ofrecerles lo que buscan lo más simple posible.

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 programador 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.

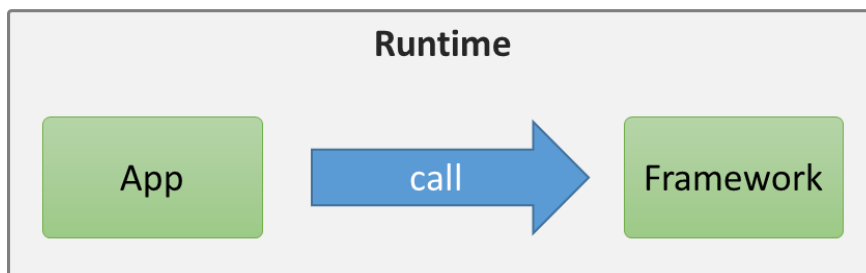


Fig 16 - Invocación tradicional

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:

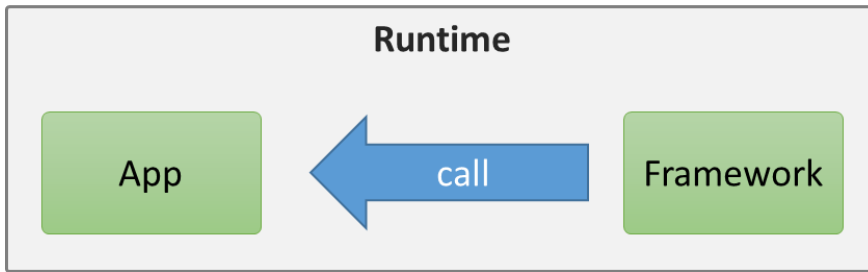


Fig 17 - Inversión de control

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.

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 metadatos 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**

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.

S.O.L.I.D

Hemos dejado para el final el acrónimo SOLID, el cual es en realidad de unión de 5 principios que para muchos son los más importantes de todos, este acrónimo fue introducido por primera vez a principios de la década del 2000 por Robert C. Martin (mejor conocido por muchos como "el tío Bob"). El acrónimo SOLID hace referencia a:

- **S**: Single responsibility principle (SRP)
- **O**: Open/closed principle (OCP)
- **L**: Liskov substitution principle (LSP)
- **I**: Interface segregation principle (ISP)
- **D**: Dependency inversion principle (DIP)

Todos estos principios los analizaremos a continuación:

Single responsibility principle (SRP)

Este principio establece que cada función, clase o módulo debe de tener una única responsabilidad dentro del software al que pertenece, al mismo tiempo que toda su funcionalidad debe de estar encapsulada.

Este principio previene que desvirtuemos el propósito para el cual hemos creado una función, clase o módulo, agregando elementos que no están relacionados a resolver el problema para el cual existe.

Es muy común entre los arquitectos inexpertos diseñar componentes que tiene más de una responsabilidad o en su defecto, desvían el objetivo que buscaba resolver para terminar teniendo más de una responsabilidad.

Robert C. Martin define una responsabilidad como una razón para cambiar, y concluye que una clase o módulo debe tener una, y solo una, razón para ser cambiada (es decir, reescrita), y expone el siguiente ejemplo:

Considere un módulo que compila e imprime un informe. Imagina que un módulo de este tipo se puede cambiar por dos razones. Primero, el contenido del informe podría cambiar. En segundo lugar, el formato del informe podría cambiar. Estas dos cosas cambian por causas muy diferentes; Una sustantiva, y una cosmética. El principio de responsabilidad única dice que estos dos aspectos del problema son en realidad dos responsabilidades separadas y, por lo tanto, deben estar en clases o módulos separados. Sería un mal diseño juntar dos cosas que cambian por diferentes razones en diferentes momentos.

En pocas palabras, un componente deberá ser pensado y diseñado basado en su responsabilidad, evitando que un componente tenga más de una responsabilidad.

Dividir un componente en responsabilidades es importante porque permite que cada uno cambie de forma independiente sin afectar el funcionamiento del otro, además y más importante, permite que sea más fácil su reutilización.

Open/closed principle (OCP)

Este principio dice que toda función, clase, modulo que creamos, deberá estar abierto para su extensión, pero cerrado para las modificaciones.

Básicamente lo que dice este principio es que cualquier unidad de software que creamos deberá poder extenderse para ampliar su funcionalidad, es decir, mejorar con ciertas restricciones el software, pero debemos evitar (cerrar) que lo modifiquen, es decir, debemos evitar que el programador altere el propósito para el cual esta echo o modifique el funcionamiento core del componente.

Para que quede más claro este ejemplo, imagina un Smartphone, todos sabemos que podemos desarrollar aplicaciones para ejecutarse dentro del sistema operativo, con lo cual, le damos más funcionalidad a nuestros usuarios. Ahora bien, el API que proporciona Android o IOS nos permite realizar ciertas acciones sobre el teléfono, sin embargo, las aplicaciones pueden realizar acciones limitadas, pues el OS se protege a sí mismo para que una aplicación no altere el funcionamiento del teléfono como tal, por lo tanto, todas las cosas que realice estarán acotadas a la app en cuestión, por lo que un fallo en una app no debería de afectar a otro.

En este contexto, podríamos decir que el API permite que extendamos la funcionalidad, pero al mismo tiempo, el OS nos cerrara ciertas acciones que puedan modificar o dañar el funcionamiento del sistema. Es por eso que podemos decir que está abierto para ser extendido pero cerrado para ser modificado.

Los Framework de programación son otro claro ejemplos de este principio, pues por un lado nos permiten implementarlos y extender su funcionalidad por default para adecuarlos a nuestras necesidades, sin embargo, no podemos afectar el funcionamiento central de este, el cual dicta la forma en que trabaja o el ciclo de vida.

Sin lugar a duda, este es uno de los principios más difíciles de implementar, pues requiere de mucho análisis para determinar hasta qué punto nuestro componente será extensible y hasta qué punto estará cerrado. Permitir que un componente sea sumamente extensible puede provocar la inestabilidad del sistema, pero hacerlo demasiado cerrado puede resultar poco práctico y difícilmente reutilizable, es por ello que debemos de buscar un punto medio.

Liskov substitution principle (LSP)

Este sin duda uno de mis principios favoritos y que está profundamente relacionado con muchos de los patrones de diseño.

Este principio dice lo siguiente: Sea $\Phi(x)$ una propiedad demostrable sobre los objetos x de tipo T . Entonces $\Phi(y)$ debe ser cierta para los objetos y de tipo S donde S es un subtipo de T .

Más claro ni el agua... Es broma, seguramente quedaste a cuadros con esta definición, y es que esta fue la definición matemática que dio Barbara Liskov sobre este principio. Sin embargo, vamos a aterrizarla a lenguaje natural para que todos podamos entender.

El principio nos quiere decir que: Un objeto de tipo T puede sustituirse por cualquier objeto de un subtipo S . Esta definición es más simple, pero vamos a aterrizarla todavía más:

Si tenemos una clase base, y luego remplazamos esa clase base por un subtipo (que extiende a la clase base), el programa debería de seguir trabajando. Veamos un ejemplo:

```
1. interface Figure {
2.     public void draw();
3. }
4.
5. class Circle implements Figure{
6.     public void draw(){
7.         System.out.println("Circle");
8.     }
9. }
10.
11. class Square implements Figure{
12.     public void draw(){
13.         System.out.println("Square ");
14.     }
15. }
16.
17. public class Main{
18.     public static void main(String[] args) {
19.         Figure figure = new Circle();
20.         figure.draw();
21.
22.         Figure figure2 = new Square();
23.         figure2.draw();
24.     }
25. }
```

En este, podemos ver cómo podemos asignar las variables *figure* y *figure2* (que son de tipo *Figure*) a cualquiera de sus subclases, como *Circle* y *Square*, y la aplicación seguirá trabajando sin problema.

Básicamente este principio nos dice que todo subtipo es polimórficamente compatible con su padre, por lo que remplazarlos no debería de tener problemas para el programa.

Si tuviste la oportunidad de leer mi libro de "Introducción a los patrones de diseño" verás que casi la mayoría de los patrones se basan en este principio.

Interface segregation principle (ISP)

El Principio de Segregación de la Interfaz indica que el cliente de un programa solo debería de conocer del otro solo los métodos que realmente va a utilizar, evitando conocer una gran cantidad de método que no son necesarios.

La idea que propone es separar los métodos de una interfaz con una gran cantidad de métodos, en varias interfaces más pequeñas y cohesivas, de tal forma que cada interface tenga un rol específico, permitiendo al cliente ignorar al resto de interfaces y centrarse únicamente en la que necesita, reduciendo el número de método y la complejidad de la misma. A este tipo de interface reducidas se les conoce como "**Interfaces de rol**".

El objetivo que se busca con este principio es mantener desacoplado a los sistemas con respecto a los sistemas de los que depende. A mayor número de métodos mayor será la dependencia, lo que hará más difícil refactorizarlo, modificarlo y desplegarlo.

Para comprender mejor este principio imaginemos que tenemos un sistema con una gran cantidad de servicios expuestos por nuestra aplicación, estos servicios abarcan desde clientes, proveedores, productos, ordenes, facturas, etc. Y para cada uno de los anteriores puede haber varios servicios, como alta, modificación, borrado y consulta, lo que hace que el número de servicios crezca de forma exponencial.

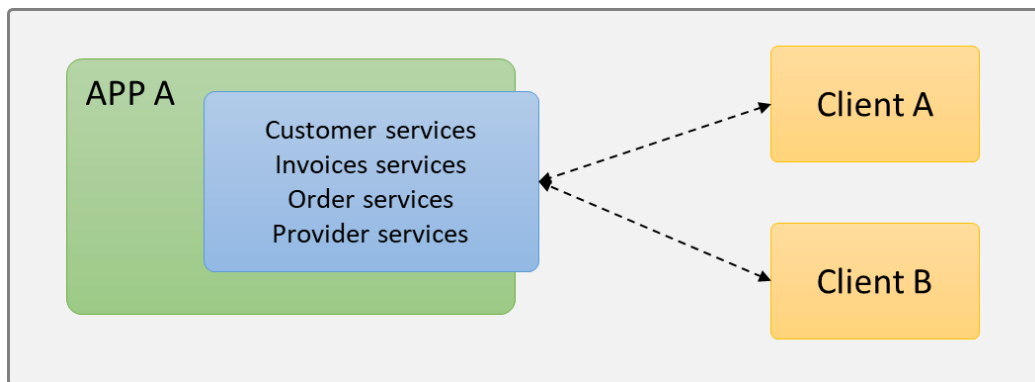


Fig 18 - Aglomeración de métodos

En la figura anterior podemos observar dos clientes que intenta comunicarse con un App que tiene una gran cantidad de servicios, lo que hace muy complejo para los clientes entender los servicios expuestos, a la vez que crea un fuerte acoplamiento a medida que utiliza más servicios de la aplicación. En este ejemplo, el Cliente A solo utiliza los servicios de órdenes, mientras que el Cliente B solo utiliza los servicios de clientes

Ahora bien, imagina que nuestra empresa crece y se ha decidido que separaremos la administración de los clientes en una aplicación independiente:

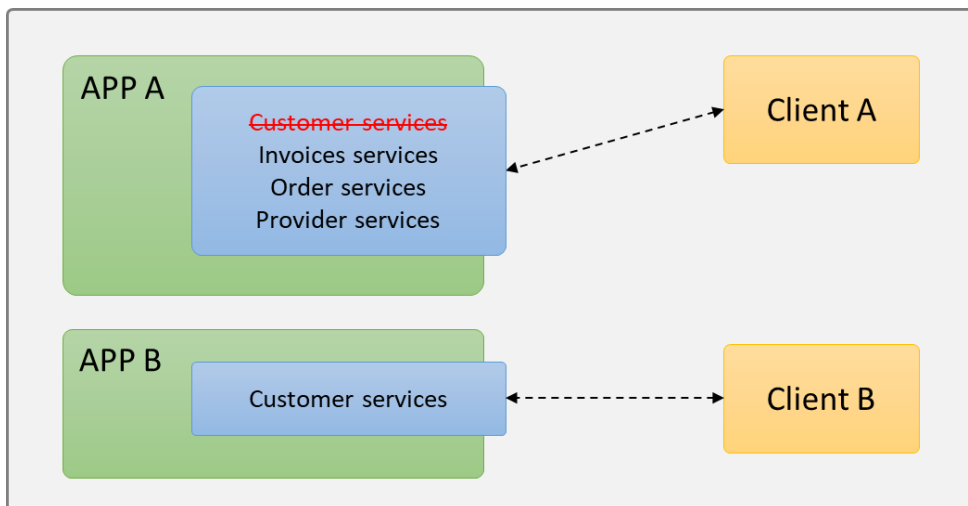


Fig 19 - Separación de responsabilidades

En esta nueva imagen hemos separado la responsabilidad de gestión de los clientes en una nueva aplicación, lo que nos lleva a separar los servicios de clientes de la App A y publicarlos en la nueva Aplicación B. A primera vista esto no tiene ningún problema, pues la App A no utiliza los servicios de los clientes, por lo que no debería de tener ningún impacto, sin embargo, esto no es así, pues la interfaz de la App A ha cambiado, lo que rompe la compatibilidad con A, lo que obligará a hacer un refactor en el Cliente A y un redespigie.

Entonces, el principio ISP lo que propone es separar los métodos de las interfaces por roles, de tal forma que cada interface tenga solo los métodos relacionados entre sí:

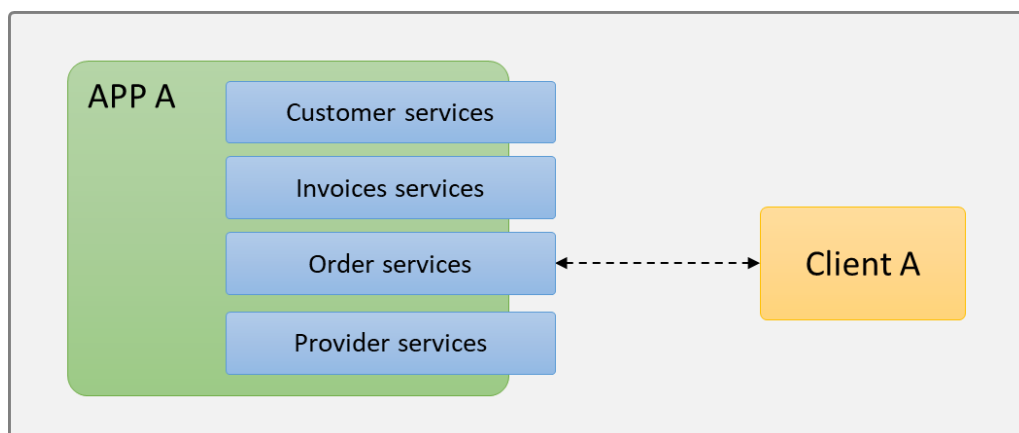


Fig 20 - Principio de Segregación de la Interfaz

Como vemos, ya no tenemos un interfaz gigante, y en su lugar tenemos varias pequeñas.

Ahora repliquemos lo que hicimos hace un momento de separar los clientes:

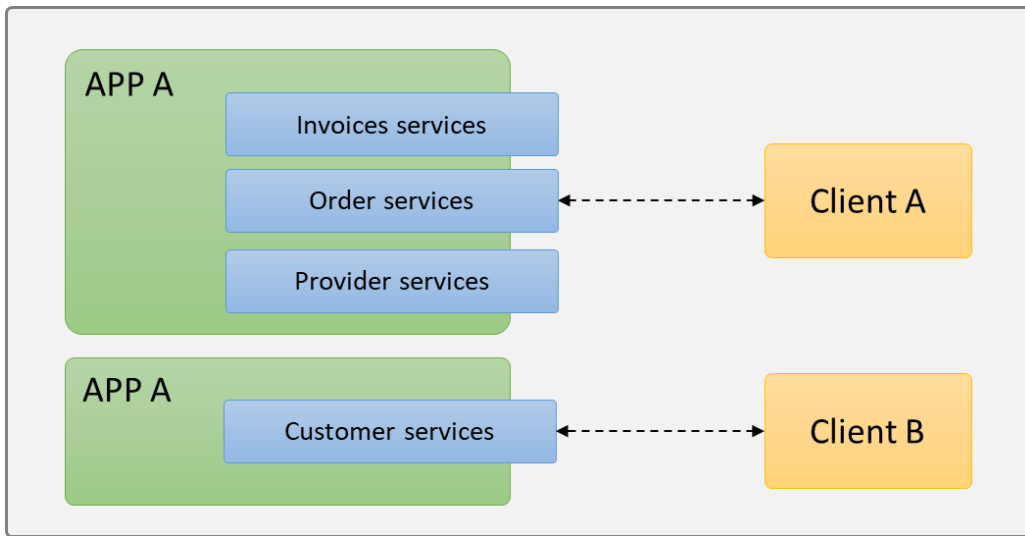


Fig 21 - Separación exitosa de interfaces

En este nuevo ejemplo, el Cliente A no se ve afectado pues jamás utilizó la interfaz de clientes, por lo que para él es indistinto si el servicio se mueve, modifica o elimina.

Dependency inversion principle (DIP)

Normalmente cuando desarrollamos software, solemos hacer que los software de más alto nivel, dependa de los más bajo nivel, de tal forma que para que un módulo completo funcione, requiere que otros más pequeños estén presentes para funcionar, muy parecido a la clásica arquitectura de 3 capas, en la que la capa de presentación depende de la capa de negocio y la capa de negocio depende de la capa de datos, sin embargo, el principio DIP viene a movernos un poco la jugada, pues lo que propone es lo siguiente:

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de las abstracciones.

- Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

Como podemos observar, en estas oraciones se mencionan 3 cosas, 1) el módulo de alto nivel, 2) el módulo de bajo nivel, 3) abstracciones, pero ¿qué quiere decir esto? Para comprenderlo, analicemos primero como funciona una dependencia normalmente:

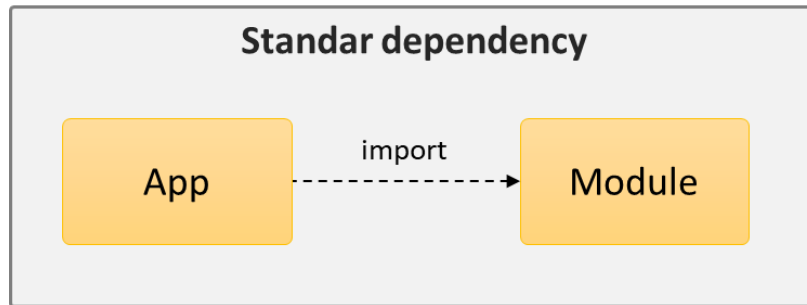


Fig 22 - Forma habitual de dependencia.

En esta imagen, podemos ver como la aplicación principal realiza una importación directa del módulo que requiere para su funcionamiento, lo cual es lo que hacemos normalmente, sin embargo, esto va en contra de este principio, pues lo que nos dice, es que los módulos superiores no deben depender los módulos de bajo nivel, y es allí donde entra la abstracción:

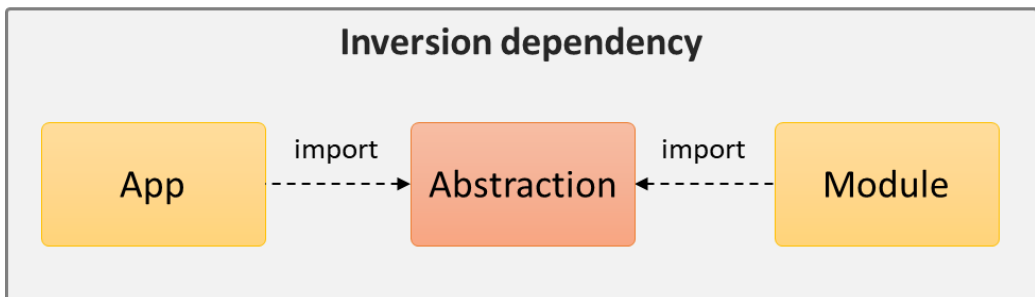


Fig 23 - Inversión de dependencias.

La abstracción es un módulo abstracto que solo implementa las interfaces o clases base que luego el módulo de bajo nivel (Module en el diagrama) deberá implementar y posteriormente el módulo de Alto nivel (App en el diagrama) deberá utilizar. Cabe mencionar que el módulo de bajo nivel deberá tener toda la funcionalidad que define la abstracción y siempre alineado a las interfaces o especificaciones de la abstracción, de tal forma que cuando el módulo de Alto necesite utilizar el módulo de bajo nivel lo haga a través de la abstracción, de esta forma, el Módulo de alto nivel no necesita saber la implementación concreta del módulo de bajo nivel, logrando con esto, la variación de cualquier parte sin afectar el funcionamiento, incluso, mediante este patrón, es posible tener múltiples implementaciones del módulo de bajo nivel.

A pesar de que la App (alto nivel) termina utilizando las clases y objetos del módulo (bajo nivel) no las utiliza directamente, si no que las utiliza encapsuladas detrás de las clases e interfaces que define la abstracción, de esta forma, utilizamos el polimorfismo para ejecutar las operaciones o funcionalidad que nos da el módulo de bajo nivel sin saber la implementación exacta, pues estará encapsulada detrás de la abstracción. Esto tiene que ser así, pues, si la App (alto nivel) utiliza el operador *new* para instanciar cualquier clase concreta del módulo de bajo nivel estaría rompiendo la primera regla del principio, la cual dice *"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de las abstracciones"* y al realizar un *new* estamos automáticamente dependiendo del módulo de bajo nivel.



Nuevo concepto: Polimorfismo

El polimorfismo es una característica de los lenguajes orientados a objetos que permite a los objetos adquirir muchas formas, de tal manera que se puede comportar de forma diferente en tiempo de ejecución sin romper la compatibilidad.

Ahora bien, te estarás preguntando, como diablos puedo instanciar las clases concretas del módulo de bajo nivel si no puedo instanciarlas, pues bien, existen dos opciones, la primera es que la abstracción defina clases Fábricas (patrón Abstract Factory o Factory Method) para crear las instancias, o bien, utilizando el patrón arquitectónico Dependency Injection (Inyección de dependencias).

**Nuevo concepto: Instancia (Instanciar)**

Se le conoce como Instanciar al proceso por medio del cual se crea un nuevo objeto a partir de una clase concreta determinada, por lo tanto, una instancia es un objeto creado a partir de una clase determinada.

El principio de inversión de dependencias tiene muchas ventajas, pero tiene la desventaja de que es muy complicado de implementar, pero una vez implementado, es muy fácil de utilizar, por lo tanto, es complicado para el programador que lo desarrolla, pero es muy fácil para el programador que lo utiliza.

Atributos de calidad

Capítulo 3

Es increíble lo fácil que es hacer software en nuestros tiempos, pues existen tantas herramientas, lenguajes y frameworks que permite que casi cualquier persona, incluso con pocos conocimientos puede construir un software. Combinado con la gran demanda de empresas que requieren desarrollo de software, ha provocado el surgimiento de muchas empresas que se dedican a satisfacer esta demanda, sin embargo, y a pesar de que crear software es muy fácil, la realidad es que crear software de calidad es muy difícil, lo que provoca que una gran cantidad del software desarrollado no cumple con las cuestiones básicas de calidad exigidas por el cliente, lo que provoca un gran trabajo de corrección o incluso, el fracaso total del proyecto.

La pregunta aquí es, ¿por qué construir software es fácil, pero construir software de calidad es muy difícil? La respuesta es más complicada de lo que podría parecer, sin embargo, la clave del éxito de un software nace desde las etapas más tempranas, es decir, desde la toma de requerimientos, que es cuando nuestro cliente nos plantea la problemática que busca resolver.

En esta etapa, el cliente nos indicará los escenarios de negocio o problemáticas que tiene el negocio y como quiere resolverlas, es decir, *"quiero que el sistema haga esto, quiero que el sistema haga el otro"*. En esta etapa nos enfocamos en entender los requerimientos del cliente y vamos apuntando todo lo que nos va pidiendo. Todos estos requerimientos que nos va solicitando el cliente de forma explícita y que está relacionados directamente a resolver un problema del negocio los conocemos como **requerimientos funcionales**.



Nuevo concepto: Requerimientos funcionales

Un requerimiento funcional representa una característica que debe tener el sistema, la cual es por lo general solicitada de forma explícita.

Regresando a la pregunta de por qué es difícil crear software de calidad, podemos decir que, por lo general, cuando se toman los requerimientos, solo se toman en cuenta los requerimientos funcionales, es decir, lo que solicita explícitamente el cliente. Sin embargo, pasamos por alto todos aquellos requerimientos implícitos que se leen entre líneas y que son claves para que un software cumpla con la calidad adecuada, estos requerimientos son los **requerimientos no funcionales**.



Nuevo concepto: Requerimientos no funcionales

Los requerimientos no funcionales son aquellos que no se refieren directamente a una funcionalidad específica del sistema, si no a las propiedades del sistema, como la seguridad, performance, usabilidad, etc.

Los requerimientos no funcionales son rara vez mencionados por los usuarios, sobre todo porque no forman parte del proceso de negocio en sí, aunque son claramente claves para el éxito del sistema.

Pero entonces, ¿qué tiene que ver esto de los requerimientos funcionales y no funcionales con la arquitectura de software?, pues bien, en la arquitectura de software, los requerimientos no funcionales son por lo general los **atributos de calidad**, lo que nos lleva a otra duda, ¿que son los atributos de calidad?:

Los atributos de calidad son todos aquellos requerimientos que tiene un impacto en la arquitectura del software. En otras palabras, son **requisitos arquitectónicamente significativos** que requieren la atención de los arquitectos. Pero esta definición nos deja otra duda, ¿Qué son los requerimientos arquitectónicamente significativos? Pues bien, se ha demostrado que no todos los requerimientos no funcionales afectan a la arquitectura, pero, por otro lado, algunos requerimientos funcionales pueden tener un impacto sobre la arquitectura, entonces podríamos definir a los atributos de calidad como:



Nuevo concepto: Atributos de calidad

Son todos aquellos requerimientos funcionales o no funcionales que tiene un impacto directo sobre la

arquitectura del sistema (*requisitos arquitectónicamente significativos*) y que requieren la atención de un arquitecto.

Con la definición de lo que es un atributo de calidad, ya nos va quedando clara la relación de los atributos de calidad con la arquitectura de software, pues los atributos de calidad son aquellos requerimientos que tiene un impacto directo sobre la arquitectura del software, y es por ese motivo que son tan importante para un arquitecto de software.

Importancia de los atributos de calidad

Los atributos de calidad son importantes porque sirve como un punto de referencia que nos ayuda a evaluar la calidad de un sistema, ya que a medida que cumplimos con los atributos de calidad, vamos cumpliendo con la calidad general del sistema. Además, los atributos de calidad pueden tener un impacto directo sobre el diseño de la arquitectura, por lo que es de gran interés para los arquitectos identificarlos y diseñar soluciones que cumplan con todos o la gran mayoría de ellos. Debido a esto, es importante que los atributos de calidad puedan ser medibles y comprobables, en otro caso, no se podría medir la calidad del sistema.

Puedes pensar que el rol del arquitecto es cumplir con la totalidad de los atributos de calidad, sin embargo, esto no siempre será posible, ya que existen casos en los que un atributo de calidad entra en conflicto directamente con otro, por ejemplo, que la aplicación sea rápida, sin embargo, por seguridad, nos hacen pasar por varios procesos de auditoría que afectan el performance.

En estos casos, es responsabilidad del arquitecto proponer soluciones equilibradas que ayuden a satisfacer a medida de lo posible todos los atributos de calidad, ya sea total o parcialmente, pero siempre buscando una solución aceptable.

Clasificación de los atributos de calidad

Al igual que muchas de las definiciones que hemos visto a lo largo de este libro, no existe un consenso sobre como clasificar los atributos de calidad, pues en diversas literaturas podrás ver que los atributos de calidad son agrupados o clasificados de forma diferente, como medibles y no medibles, externos o internos, o visibles y no visibles. En mi caso, me gusta agruparlos como visibles o no visibles, porque suelen ser más fáciles de explicar.

Los atributos de calidad visibles o no visibles se centran en su capacidad de ser observados simplemente al utilizar el sistema, de esta forma, los atributos visibles son todos aquellos que puedes observar al utilizar el sistema, como el performance, seguridad, disponibilidad, funcionalidad y usabilidad o dime tú, si el sistema tarda un minuto en cargar, no te das cuenta, o si se cae cada 3 horas, o si cualquiera puede acceder a la información privada, etc. Todos estos son visibles fácilmente por cualquier usuario, no solo por los desarrolladores.

Por otra parte, los atributos de calidad no visibles son aquellos que no se pueden observar simplemente con solo utilizar el sistema, si no que requieren ser puesto a pruebas o a un análisis completo para comprobar su cumplimiento. Entre estos atributos tenemos la portabilidad, reusabilidad, integridad, testabilidad o la escalabilidad. Todos estos atributos no pueden ser validados por un usuario con tan solo utilizar el sistema, si no que requiere ponerlas a prueba para poder comprobar su cumplimiento.

Estudiar todos los atributos de calidad que existen nos podría llevar un libro completo, sin embargo, no quisiera aburrirte con toda esa teoría, y en su lugar, quiero darte una guía sobre los atributos de calidad más importantes que existen y que puedes ver en casi en cualquier proyecto, y como buen arquitecto de software que eres, deberías de conocer.

Atributos de calidad observables

Los atributos de calidad observables son (como ya lo mencionamos) todos aquellos que pueden ser observados y validados simplemente por un usuario final con tan solo utilizar el sistema. Entre los más importantes están:

Performance (rendimiento)

El performance es el atributo de calidad relacionado con el tiempo en que tarda un sistema en responder a un estímulo o evento, de esta forma, el performance busca determinar si el tiempo de respuesta de un sistema es adecuado según los requerimientos.

Medir el rendimiento puede ser fácil de observar, por ejemplo, ¿te ha pasado alguna vez que entras a una página y tarda mucho en cargar, tanto que decides mejor cancelar y busca en otra página?, bueno, esto es un claro ejemplo de que el performance se puede observar fácilmente, sin embargo, no basta con determinar si un sistema es rápido o lento según nuestra propia perspectiva, ya que para cada persona la sensación de tiempo de carga puede ser diferente, es por eso que el performance se debe de medir de tal forma que no quede a la interpretación del usuario.

Debido a que el performance debe de ser medible, es necesario definir claramente que es para nosotros que una aplicación tenga buen rendimiento, ya que no solo basta que la aplicación responda rápido con un solo usuario conectado, si no que deberá responder rápido aun cuando tengamos varios usuarios o procesos ejecutándose al mismo tiempo, es por eso que la forma correcta para medir el performance es, **número de transacciones por unidad de tiempo**, es decir, cuantas transacciones podemos procesar en un tiempo determinado, donde la unidad de tiempo podrían ser milisegundos, segundos, minutos, horas, etc.

Un error común es creer que el performance es solo el tiempo de respuesta de la aplicación, pues esto solo mediría cuánto tarda el sistema en responder una solicitud individual, lo cual no garantiza que el sistema tenga buen rendimiento al momento de meterle carga o múltiples usuarios simultáneos.

Cuando el tiempo de respuesta debe de ser medido con exactitud, solemos implementar lo que se conoce como SLA (service level agreement) Acuerdo de Nivel de Servicio, el cual mediante herramientas podemos monitorear para determinar cuando el sistema está tardando más del tiempo del esperado por transacción individual, de esta forma, en el SLA podemos determinar el tiempo exacto en el cual el sistema debe de responder, y nos alerta cuando el sistema tarda más de este tiempo.



Nuevo concepto: Service Level Agreeegment (SLA)

Es un acuerdo escrito entre un proveedor de servicio y su cliente con objeto de fijar el nivel acordado para la calidad de dicho servicio. SLA es una herramienta que ayuda a ambas partes a llegar a un consenso en términos del nivel de calidad del servicio, en aspectos tales como tiempo de respuesta, disponibilidad horaria, documentación disponible, personal asignado al servicio, etc.

— Wikipedia

Como hemos de esperar, el performance es uno de los atributos de calidad más críticos, debido a que tiempos largos de respuesta pueden llevar al desespero del usuario o incluso declinar de su uso, es por ello que los arquitectos de software deben de tener especial atención en este atributo de calidad.

Determinar la causa por las que un sistema tiene mal performance puede ser complicado, ya que existen muchas razones que lo puede provocar, como la latencia, operaciones de lectura y escritura, velocidad de red/internet, servicios externos, sobre carga de usuarios/procesos, hardware, etc.



Nuevo concepto: Latencia

La latencia es el tiempo que transcurre desde que se envía una petición al sistema, hasta que se recibe el primer bit de respuesta.

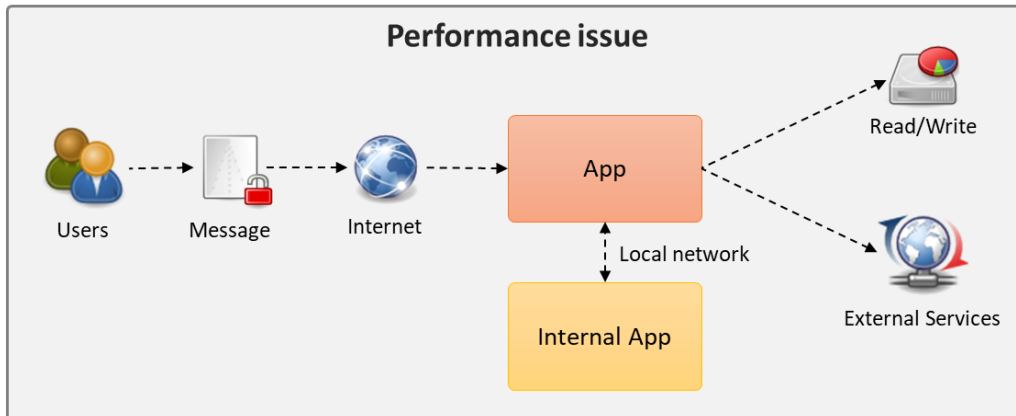


Fig 24 - Problemas de rendimiento

La imagen anterior intenta dar un panorama de las diferentes problemáticas que puede ocasionar problemas en el rendimiento de la aplicación, los cuales se detallan a continuación:

- **Usuarios:** Todas las aplicaciones deben de ser diseñadas para soportar una cierta cantidad de usuario simultáneos, así como un crecimiento a lo largo del tiempo, no tomar en cuenta esto puede llevar a problemas serios de rendimiento que van incrementando con el tiempo.
- **Mensaje:** Transmitir datos por internet son de las tareas más lentas, por lo que hay que tener cuidado con la cantidad de información que transmitimos, así como considerar su compresión para mejorar el rendimiento.
- **Internet:** El internet es el canal por el cual un mensaje es transmitido desde el cliente al servidor, por lo que debemos tener en cuenta el tipo de conexión que tenemos o que tendrán nuestros usuarios, ya que no es lo mismo enviar datos por un internet de 100mb/s que por una red 2G. Es importante hacer pruebas de velocidad con diferentes dispositivos y conexiones para asegurar tiempos de respuesta adecuados.
- **App:** La app representa la aplicación que estamos desarrollando, en ella, es importante utilizar algoritmos y estructuras de datos adecuadas que permiten procesar y responder lo mejor posible, también es importante diseñarla para escalar a medida que la demanda crezca.
- **Red local:** en ocasiones utilizamos redes locales para comunicar aplicaciones internas, por lo que hay que tener cuidado con la latencia de los servidores y evitar el redireccionamiento que pueda incrementar la latencia, por otra parte,

es importante tener una red lo suficientemente rápida y estable para comunicar las aplicaciones internas.

- **Servicios externos o aplicaciones internas:** Los servicios externos o aplicaciones internas son todas aquellas que brindan servicios a nuestra aplicación para poder realizar una determinada tarea en la App, las cuales pueden generar un cuello de botella en el rendimiento de nuestra aplicación, es por ello que es importante tener SLA en los servicios externos o internos para asegurar los tiempos de respuesta de la App. Cabe mencionar que estos servicios salen de nuestro control, por lo que es importante tomarlo en cuenta.
- **Lectura y escritura:** La gran mayoría de las aplicaciones escriben o leen de la base de datos, escriben logs, guardan/leen en FTP o el File System, etc. Todos estos procesos que escriben en disco son críticos, pues degradan el tiempo de repuesta del disco duro, el cual, es el componente más lento de una computadora, es por ello que debemos tener cuidado de cuanta información estamos escribiendo o leyendo en disco, para reducirla al máximo. Algunas estrategias de mitigación son: utilizar discos o servidores independientes para las tareas de escritura/lectura o mantener en memoria la información más utilizada (Caché).



Nuevo concepto: Caché

El caché es un componente de hardware o software que almacena datos para que las solicitudes futuras de esos datos se puedan atender con mayor rapidez; los datos almacenados en un caché pueden ser el resultado de un cálculo anterior o el duplicado de datos almacenados en otro lugar, generalmente, da velocidad de acceso más rápido.

— Wikipedia

Algo muy importante a tomar en cuenta es que, existen diferentes patrones de llegada de solicitudes, como lo son las periódicas o estocásticas. Las periódicas son aquellas solicitudes que sabemos que siempre llegan y tiene un patrón conocido, como un sensor que registra la temperatura cada minuto, en tal caso sabemos que cada minuto tendremos una llamada. Por otro lado, tenemos las estocásticas, que son aquellas que no siguen un patrón determinado, pero si podemos determinar su llegada por medio de una distribución probabilística, por ejemplo, sabemos por datos estadísticos que las personas usan más la tarjeta de crédito los fines de semana, por lo que podemos predecir una carga mayor durante

estos días. Por otro lado, existe un tercer patrón que es el más peligroso, y son las solicitudes esporádicas, las cuales pueden llegar sin una lógica aparente y que no obedecen a los patrones periódicos o estocásticos, un ejemplo podría ser un video que se hace viral y que trae tráfico extremo a nuestro sitio de forma inesperado y sin forma de predecirlo. Como arquitectos debemos de tener en cuenta todo esto para evitar que la aplicación se vea afectada.

A pesar de todo esto, al final, al usuario lo único que le interesará es cuánto tiempo tardar nuestra aplicación en responder/cargar, por lo que debemos poner mucho foco en ello, pues esto está directamente relacionado a la experiencia de usuario (UX) que le estamos brindando.

Security (Seguridad)

La seguridad es la capacidad de un sistema para resistir al uso no autorizado, al mismo tiempo que proporciona sus servicios a usuarios legítimos. Cualquier intento no autorizado de acceder al sistema es considerado como un ataque, y un ataque es cualquier intento de violar la seguridad del sistema.

En la práctica no todos los ataques son mal intencionados o ejecutados con la intención de burlar la seguridad del sistema, si no que en ocasiones son usuarios que quizás olvidaron su contraseña, lanzan peticiones desde orígenes sospechosos, realizan simultáneamente varias solicitudes o intentan acceder por error a datos a lo que no tienen privilegios, es por ello que el sistema debe de ser lo suficientemente listo como para distinguir entre un ataque real y un simple error del usuario. Un ejemplo claro de estos es, cuando nuestro banco bloquea ciertas transacciones de nuestra tarjeta de crédito, debido a que identifica un comportamiento inusual en el uso, al mismo tiempo nos notifica que se realizó una transacción sospechosa y nos solicita que la reconozcamos o la rechazemos y de esta forma el sistema va aprendiendo.

Podemos pensar que la seguridad es solo pedir un usuario y contraseña o que pongamos un certificado en nuestra página para habilitar la comunicación segura, sin embargo, la seguridad va muchos más allá de eso. Por desgracia, la seguridad tiene muchas aristas y cada vez es más difícil hacer aplicaciones seguras debido

a las nuevas arquitecturas donde casi todo pasa por internet. Algunos de los ataques más frecuentes son:

- **Robo de datos:** Consiste en robar datos confidenciales de nuestros clientes para poder ser explotados. Ejemplos de estos son: datos de las tarjetas de crédito para realizar compras fraudulentas, robo de datos personales para extorsionar o venderse, conocer información privilegiada.
- **Denegación de servicio (DoS):** Consiste en que el atacante lanza masivas solicitudes sobre una aplicación para saturarla e impedir que usuarios legítimos puedan utilizarla.
- **Borrado de datos:** Consiste en ataques dirigidos a hacer un daño directo sobre la compañía, borrando datos críticos para su operación, como base de datos o archivos del sistema.
- **Secuestro de información (Ransomware):** Son virus que encriptan la información de nuestro disco con la finalidad de que paguemos un rescate para desencriptar la información.
- **Intermediario (Main-in-the-middle):** Este ataque consiste en interceptar los mensajes entre dos entidades, de tal forma que el atacante puede ver los mensajes que se envían entre sí, o incluso, enviar mensajes en nombre de alguno de ellos.
- **Robo de identidad:** Es cuando el atacante se hace pasar por un usuario legítimo y realiza acciones en su nombre.
- **Software malicioso:** Los softwares maliciosos son todos aquellos virus, troyanos, keyloggers, spyware, etc, los cuales tiene como objetivo desde solo espiar al usuario, hasta tomar el control total del equipo. Suelen ser los más peligrosos, pues tiene una conexión directa con el atacante, el cual puede explotar la vulnerabilidad en cualquier momento sin necesidad de realizar alguna acción específica por parte del usuario.

Está demostrado que la mayoría de los ataques exitosos son causados por descuidos de los usuarios y no por un super hacker que quiebra toda la seguridad (aunque desde luego que hay casos), esto consiste en que los usuarios dejan sus contraseñas a la vista, las guardan en texto plano, utilizan la misma para todas sus aplicaciones/servicios, introducen las contraseñas en redes no seguras, no

utilizan múltiples factores de autenticación, o simplemente responden mails de orígenes dudosos con datos importantes.

Hablar de todas las formas en que un sistema podría ser atacado y como prevenirlo es un tema muy extenso, tanto, que existen certificaciones, especialidades, libros completos sobre el tema, por lo que sería imposible cubrir el tema en este libro, en su lugar, quisiera invitarte a que, si quieres saber más sobre cómo crear aplicaciones más seguras, te dirijas a una literatura especializada en el tema, por lo que me gustaría hablarte de las características que un sistema seguro debería de tener:

- **No repudio:** En una comunicación entre dos partes (emisor y receptor), ninguna de las partes podrá negar una transacción, de tal forma que el emisor no podrá negar que envió un mensaje al receptor, porque el receptor tendrá pruebas de que si lo hizo, mientras que el receptor no podrá negar que recibió el mensaje, porque el emisor tendrá pruebas de la recepción del mensaje. Esto es muy importante, ya que incrementa la confianza entre las partes en las comunicaciones.
- **Autenticidad:** La autenticidad es la capacidad de un sistema para detectar quien está del otro lado, y asegurarse de que la persona que está del otro lado es quien dice ser, de esta forma, prevenimos enviar información sensible a un impostor. Para lograr esto, es necesario utilizar algún tipo de autenticación, como usuarios/password, intercambio de llaves simétricas (certificado), token, etc.
- **Confidencialidad:** La confidencialidad consiste en que la información, ya sea almacenada, transmitida por la red o almacenada en memoria, solo esté disponible para aquellas personas que estén autorizada para acceder a dicha información. Incluso, si la información cayese en manos no autorizadas, esta no debería de poder ser interpretada, es decir, la información debería estar encriptada para no poder ser interpretada. Si la confidencialidad se rompe, la información estaría expuesta y perderíamos la confianza de nuestros usuarios.
- **Integridad:** La integridad es básicamente la capacidad de un sistema para detectar cuando un mensaje ha sido alterado durante su transmisión. En aplicaciones que se comunican por internet, es común que el emisor mande mensajes al receptor, los cuales podrían ser interceptados, modificados y enviados nuevamente al receptor haciéndose pasar por el emisor, por ello, el

sistema deberá poder validar que el mensaje recibido no ha sido alterado desde su creación. Las firmas digitales es una forma adecuada de validar la integridad de un mensaje.

- **Auditoria:** La auditoría es la capacidad de un sistema para rastrear las transacciones realizadas una vez que estas sucedieron para poder analizarlas más adelante. La auditoría es por lo general una de las fases más avanzadas en la maduración de un sistema, pues no todos lo implementan desde el inicio, aunque se va haciendo necesario a medida que la aplicación crece o dependiendo la criticidad de la información que se está moviendo, o incluso, si las leyes lo exigen.



Tip: Lee más sobre la Confidencialidad, Integridad y Autenticidad en mensajes

En mi blog [he escrito un artículo muy completo](#) sobre estos conceptos por si quieres profundizar más en el tema.

La seguridad no es algo que se pueda medir con una formula, lo que complica saber exactamente qué tan segura es nuestra aplicación, por lo que se hace a menudo es, crear un check list de todas las características de seguridad que deberían de tener nuestras aplicaciones, al mismo tiempo, se le asigna una criticidad, la cual permite evaluar mejor cuales son más importantes que otras, sobre este check list, se puede asignar un puntaje, el cual podrá ser la calificación de seguridad que tiene nuestra aplicación.

Finalmente, recuerda una cosa, una vez que los datos de un usuario han sido expuestos, ya no hay marcha atrás, estarán expuestos para siempre. Datos como tu nombre, dirección, teléfono, número de seguridad social, nacionalidad, fecha de nacimiento, genero, enfermedades, alergias, familiares, etc, son datos que quizás nunca cambien durante toda tu vida, por lo que una vez expuestos, no habrá nada que hacer.

Availability (disponibilidad)

Este atributo de calidad hace referencia al tiempo al que está operable un sistema para realizar la tarea para la que fue diseñada. En pocas palabras, es el tiempo que la aplicación está funcionando. La disponibilidad también se puede medir mediante la fórmula 100% - la indisponibilidad (unavailability).

Un sistema debe de ser capaz de responder una solicitud en cualquier momento, es decir, puede recibir aleatoriamente cualquier número de peticiones a cualquier hora o día y el sistema debe de responder, sin embargo, esto no es así en el mundo real, pues todo sistema, por más complejo que sea, se ha caído en algún momento, como es el caso de Facebook, Whatsapp, Google, Amazon o Twitter, todas las empresas en algún momento de su vida han fallado y seguirán fallando, pues las fallas son inevitables.

Por este motivo, la disponibilidad debe de indicar en términos reales cuanto tiempo estará disponible nuestra aplicación en un determinado periodo de tiempo, por ejemplo, "la aplicación tendrá una disponibilidad del 99% al año", esto quiere decir que tenemos considerado que la aplicación fallará un promedio de 3.65 días por año, o estará disponible por 361.35 días por año.

Como podemos ver, ya estamos dando una cifra medible que podemos evaluar, con la cual podemos determinar si atributo de calidad se cumple o no, o incluso, podemos darnos cuenta si nuestro umbral de fallo es muy grande. Solo imagina que el sistema que procesa los pagos de las tarjetas de crédito, ¿Te puedes dar el lujo de fallar 365 días al año? Seguramente no, porque estarías perdiendo ventas.

La fórmula para medir la disponibilidad (Availability) es:

$$availability = \frac{uptime}{uptime + downtime}$$

Donde:

- **uptime:** es el tiempo promedio entre dos fallas del sistema, es decir, es el tiempo promedio que hay entre una falla y otro.
- **downtime:** Es tiempo promedio para solucionar una falla y dejar nuevamente operativo el sistema, tomando en cuenta solo las interrupciones no planificadas.

Es importante que al momento de diseñar nuestro software evaluemos que tanta disponibilidad debe de tener nuestro sistema, pues una disponibilidad muy baja puede ser un grave problema para los usuarios, mientras que una disponibilidad innecesariamente alta puede solo representar gastos y derroche de recursos que podrían ser utilizados para otras cosas, como agregar nuevos features o corregir otros problemas. Solo ponte a pensar, tu usuario notará la diferencia entre 99.99% y 99.999% lo más seguro es que no, pero claro, todo dependerá del tipo de aplicación que estés diseñando. Un dato a tomar en cuenta es que el Alta disponibilidad se logra cuando alcanzamos los cinco nueves, es decir 99.999%.



Nuevo concepto: Alta disponibilidad

La alta disponibilidad es una característica del software para mantenerse disponible por largos periodos de tiempo sin interrumpir su servicio. La disponibilidad deberá estar garantizada sin importar que la falla se origine por un bug, falla en la red o desastre natural.

La disponibilidad es un atributo de calidad difícil de lograr, pues necesitamos crear software que fue diseñado para escalar desde el inicio, ya que hacerlo sobre la marcha puede tener un impacto grande sobre el sistema. Algunas de las estrategias más importantes para lograr tener una buena disponibilidad son:

- **Load balancer (balanceo de cargas):** Consiste en tener múltiples instancias de la aplicación corriendo en diferentes servidores, con la finalidad de que la carga de trabajo se puede dividir entre el número de servidores que conforman el cluster.
- **Redundancia:** La redundancia es la capacidad de replicar los datos, software o hardware crítico, de tal forma que pueda soportar la carga en caso de que otro componente fallara.
- **Monitoreo:** El monitoreo es una herramienta preventiva que nos puede arrojar advertencias antes de que el sistema falle por completo, dándole tiempo a los administradores del sistema para realizar las acciones correctivas.
- **Failover:** Consiste en tener una infraestructura paralela, la cual pueda soportar toda la carga en caso de que la infraestructura principal falle.

- **Maximizar el rendimiento:** No todas las fallas son por la infraestructura, muchas veces los algoritmos implementados no están dando el desempeño adecuado, provocando la saturación del sistema y si eventual colapso. Lo que aplica es analizar nuestro sistema para ver que podemos optimizar.
- **Minimizar el impacto de mantenimiento:** Todos los sistemas requieren mantenimiento, sin embargo, es indispensable planear correctamente nuestras ventanas de mantenimiento en las fechas y horarios menos críticos, así como juntar la mayor cantidad de parches a aplicar en una sola ventana, evitando tener que programar más ventanas de mantenimiento. Otra alternativa es, diseñar nuestra aplicación para que permita el mantenimiento sin necesidad de detener la operación, es más complicado, pero posible.

Un dato interesante es que, la mayor parte del tiempo que un sistema está fuera de servicio es por fallas humanas, ya sea una característica que no fue probada correctamente, una configuración incorrecta o ejecución errónea de algunos comandos, es por ello que además de las estrategias anteriores, es importante tener una buena fase de pruebas que garantice que lo que estamos llevando a producción no fallará, al mismo tiempo, es importante reducir el riesgo humano, automatizando la mayor cantidad de tareas y reduciendo así el margen para que un humano cometa un error.

Functionality (funcionalidad)

La funcionalidad es la capacidad del sistema para realizar el trabajo para el cual fue diseñado, así de fácil. En este sentido, a este atributo no le interesa como es que se construyamos el sistema, siempre y cuando cumpla con su trabajo, esto quiere decir que la aplicación podría ser un solo componente (monolítico), sin una estructura concreta o una división de responsabilidades.

Es curioso, pero la funcionalidad no tiene un impacto sobre la arquitectura, pues este no nos impone restricciones, o pautas que debemos cumplir, sino más bien, nos indica el objetivo buscado y no el medio para alcanzarlo.

Medir la funcionalidad es relativamente sencillo, pues solo bastaría con comparar los requerimientos del sistema contra lo realmente implementado, lo cual nos daría el total de cobertura de requerimientos implementados.

Este atributo podría resultar trivial, pues es obvio que un sistema, debe de implementar todos los requerimientos, sin embargo, es impresionante la cantidad de proyectos que concluyen con algún requerimiento no implementado o mal interpretado que lleva al incumplimiento de la funcionalidad.

Usabilidad

La usabilidad mide el grado en el que un usuario específico puede usar un software para lograr una serie de objetivos cuantificados.

A diferencia de otros atributos de calidad, este atributo es más complicado de medir, pues no podemos decir que el sistema tendrá un 99.99% de usabilidad, pues la usabilidad no funciona de esta manera. En su lugar, la usabilidad se debe de medir mediante otros factores, por ejemplo, cuanto tiempo tarde un usuario promedio en aprender a utilizar la herramienta, gráficamente que tan bien le pareció el sistema, con que eficiencia o eficacia logra un usuario su objetivo, pero en general, mide la satisfacción y utilidad para el usuario.

En este sentido, no hay una fórmula exacta para medir la usabilidad, la forma en que la evaluamos cambia según los objetivos que buscamos, por ejemplo, en video juegos, usan una algo llamado "acceso temprano" el cual consiste en dar acceso anticipado a ciertos usuarios para que utilicen el juego y den su feedback para mejorar el juego. En aplicaciones web, hay diseños evolutivos, en el cual se libera una determinada interfaz gráfica y se encuesta a los usuarios para determinar si les gusto, Otra estrategia es mediante software analizar como los usuarios utilizan el sistema, para determinar cuánto tiempo permanecen en la aplicación, en qué punto desisten, o cual es el flujo por el que navegó el usuario.

Otra forma de medir la usabilidad es medir cuantas transacciones puede realizar un usuario en un tiempo determinado, lo cual podría medir, que tan productivo es el usuario utilizando la herramienta, así como que tan fácil de utilizar.

La usabilidad es uno de los atributos clave para el éxito de nuestro proyecto, pues de alguna forma mide el nivel de satisfacción del usuario. A pesar de esto, es impresionante el número de proyectos en el cual es ignorado este atributo, asumiendo que el usuario entenderá la idea que le queremos transmitir.

No observables

Los atributos no observables son todos aquellos que no es posible distinguirlos a simple vista, pues corresponde a características más internas del software que requieren otro tipo de observación o validación para poder ser observadas y medidas. Los atributos que entran dentro de esta categoría son:

Modificabilidad

La Modificabilidad corresponde al **costo y riesgo de realizar un cambio** sobre un sistema existente.

Es normal que los sistemas cambien o sean modificados durante toda su vida útil, ya sea para adaptarse a las nuevas necesidades del negocio, cambio en las tendencias del mercado, adaptar nuevas tecnologías, implementar nuevos protocolos, corregir issues de seguridad, mejorar la usabilidad, ajustar el performance, etc. Es por esta razón que desde el inicio de la construcción del software es necesario pensar en que en algún momento este sistema deberá ser modificado, pero no solo eso, sino que seguramente, será modificado por personas que no estuvieron desde el inicio, incluso, podrían ser modificado por personal externo a nuestra organización, y es por esta razón que la Modificabilidad es un atributo muy importante al momento de desarrollar software.

Pensar que nuestro sistema está terminado una vez que lo liberamos a producción es un gran error, error que nos puedes costar mucho tiempo o dinero de corregir en el futuro, y es por ello que los arquitectos deben de tener un especial instinto para predecir cómo, cuándo y por qué un sistema tendría que ser modificado. Por ejemplo, si hacemos un sistema que venda helados de vainilla y chocolate, no hace falta ser un genio para predecir que en el futuro agreguemos más sabores.

Cuando un sistema se planea, no basta con tener conocimientos técnicos de como el sistema podría evolucionar en el futuro, si no que hace falta tener cierto conocimiento en la industria para entender como los usuarios usarán nuestro sistema y los cambios que podrían darse en el futuro.

Si bien, es importante pensar de qué forma podría evolucionar el software, también es importante delimitar hasta donde nuestro sistema puede ser modificado, ya que pensar en un sistema que puede modificarse para adaptarse a lo que sea, también es un error, es por ello que un arquitecto deberá plantearse las siguientes preguntas con la finalidad de planificar la modificabilidad del sistema:

- ¿Qué puede cambiar? Lo primero que el arquitecto debe de analizar es, cuáles son las partes del sistema más propensas a cambiar, ya sea por nuevas tecnologías, nuevos requerimientos o la evolución natural de la industria.
- ¿Cuál es la probabilidad del cambio? Una vez que determinamos lo que puede cambiar, es necesario evaluar que tan probable es que pueda cambiar, pues esto nos dará un análisis de los problemas que tendríamos en el futuro y poder preparar al sistema para estos cambios.
- ¿Quién debe hacer el cambio? Cuando hablamos de cambios, siempre pensamos en desarrolladores que modifican el código, compilan y suben a producción una nueva versión del software, sin embargo, en estos tiempos, también hay cambios que podría realizar un usuario final, como mover la configuración del sistema, cambiar el idioma, el formato de hora, moneda, etc. todos estos cambios podrían ser deseables por los usuarios a medida que nos internacionalizamos, es por ello que debemos considerar quien realizaría el cambio y habilitar las herramientas adecuadas para que lo puede hacer según su nivel técnico.
- ¿Cuál es el costo del cambio? Una vez que hemos evaluado que puede cambiar, cual es la probabilidad del cambio y quien estaría realizando el cambio, tenemos todos los elementos para determinar cuál será el costo de realizar ese cambio.

Plantearnos adecuadamente todas estas preguntas nos llevaría a conocer los principales puntos de cambio del sistema y poder actuar en consecuencia. Desde luego que no podremos visualizar todos lo que podrá cambiar, pero al menos podremos identificar qué es lo más propenso a cambiar y desarrollar mecanismos que nos ayuden a mitigar los riesgos de realizar cambios desde el diseño de la solución.

Portabilidad

La portabilidad es la capacidad de un software para adaptarse a una plataforma diferente para la que se diseñó originalmente.

En primera instancia podríamos pensar que la portabilidad es la capacidad de ejecutar un programa en los diferentes sistemas operativos, como lo son Windows, IOS, Linux, Android, etc. lo cual es correcto, sin embargo, la portabilidad se puede dar a otro nivel, por ejemplo, que nuestras aplicaciones corran en más de un servidor de aplicaciones, o que puede ser interpretado por diferentes máquinas virtuales, o protocolos de comunicación independientes de la plataforma, etc.

Existe herramientas que nos permite crear software portable, como es el caso de Java o Python, el cual nos permite programar una vez y ejecutar en diferentes sistemas operativos, ese es el caso más fácil, por otro lado, existen otras plataformas como C/C++ que, si bien, no cuenta con ese nivel de portabilidad, sí que podemos aprovechar mucho del desarrollo y recompilarlo en el sistema operativo deseado.

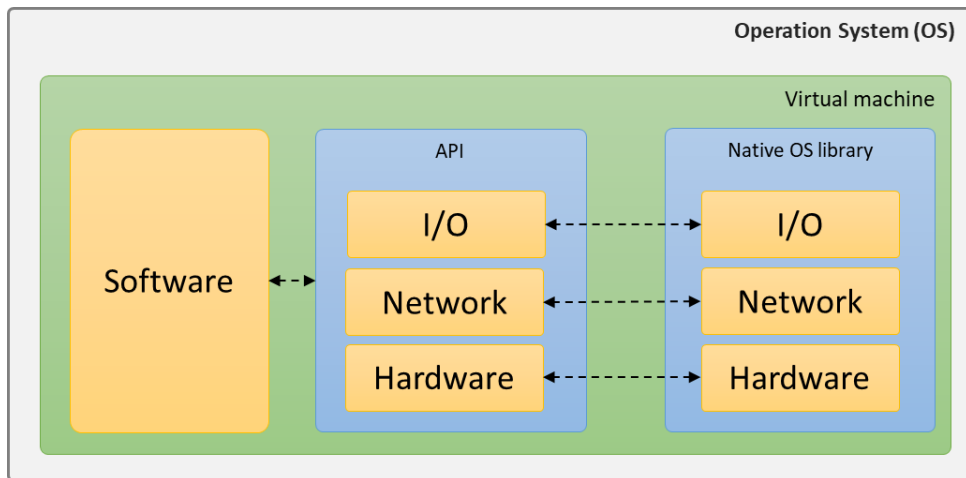


Fig 25 - Máquina virtual

En la imagen anterior podemos apreciar la arquitectura que siguen plataformas con máquinas virtuales, en la cual se componen de 5 componentes:

- **Sistema operativo:** Representa al sistema operativo sobre el cual está funcionando nuestra aplicación.
- **Software:** representa la aplicación que estamos desarrollando, la cual utiliza una capa de abstracción para hacer referencia a las librerías nativas del OS.
- **API:** El API es un conjunto de librerías que proporciona la máquina virtual para abstraer la implementación concreta del sistema operativo, de tal forma que expone un API multiplataforma que no hace referencia a cuestiones nativas del OS (recuerda el principio Dependency inversión principle (DIP), pues exactamente esto).
- **Native OS library:** Para que el API funcione necesita una serie de librerías del sistema operativo, las cuales son las que hacen la comunicación real con el OS. Estas librerías se tienen que implementar para cada sistema operativo específico.
- **Máquina Virtual:** Esta tiene la responsabilidad de ejecutar nuestro software, el cual está escrito en un lenguaje que puede ser interpretado. Se requiere una máquina virtual específica para cada plataforma.

En esta arquitectura podrás ver el Software solo se comunica con el API, pero no tiene dependencias directas con el OS, en su lugar, delegamos la responsabilidad de la comunicación a las librerías nativas del OS. Esto permite que el software sea portable, sin importar en que sistema operativo se ejecute, con la única limitante que deberá existir una máquina virtual compatible con el OS.

La portabilidad no es un atributo de calidad que todo el software deba de tener, por lo que es importante analizar nuestro requerimiento y entender cuál será el alcance de nuestro software, el crecimiento que tendrá, y los futuros usuarios que podrá tener la aplicación. Hay desarrollos muy específicos que no podrán ser portables, como software para dispositivos embebidos, aplicaciones móviles nativas o librerías para el sistema operativo, lo cual está bien, ya que probablemente no tenemos pensado que se ejecuten en otra plataforma.

Lograr la portabilidad se logra reduciendo las dependencias con el sistema operativo, como usar al mínimo características propias de un sistema operativo o separando esa funcionalidad en librerías que podremos implementar en cada sistema operativo por separado, desacoplando el funcionamiento de nuestros

sistemas de las funciones nativas del OS, la otra es utilizar máquinas virtuales como es el caso de Java, que se encarga de aislar las dependencias del OS y brindándonos un entorno de ejecución propio para cada sistema operativo.

Finalmente, ser portable no significa que nuestro software debe ejecutar en todos los sistemas operativos, si no que podemos limitar el número de sistemas operativos a los cuales queremos llevar la portabilidad.

Reusabilidad

La reusabilidad es la capacidad de una unidad de software para poder ser reutiliza en más de un programa de software.

La reutilización es uno de los atributos de calidad que más de moda están hoy en día, ya que reutilizar software implica ahora de tiempo, dinero y muchos errores en producción. Reutiliza no significa que tomemos un software existente y lo modifiquemos para que nos funcione, en su lugar, la reutilización es crear pequeños fragmentos de software que hagan una solo tarea y que esta tarea nos sirve sin modificarla.

Existe una diferencia muy grande entre reutilizar y modificar algo para adaptarlo a nuestra necesidad, cuando modificamos algo existente le llamamos apalancamiento, mientras que cuando utilizamos una librería o servicio existente tal cual está, lo llamamos reutilizar.



Nuevo concepto: Apalancamiento

El apalancamiento es cuando partimos de un componente existente para desarrollar una nueva versión que se adapte a nuestras necesidades, un apalancamiento requiere forzosamente de una nueva versión del componente, la cual contiene los cambios específicos que nos sirve solo a nuestro sistema.

Para que un componente de software sea reutilizable debe de ser diseñado desde el principio para eso, pues tenemos de pensar en los diferentes escenarios en los que se podría utilizar y no solo para el escenario que queremos resolver en ese

momento. Un componente reutilizable debe de tener ciertas características para considerarse reutilizable, como lo son:

- **Modular:** el componente en si debe de ser diseñado para ser solo una parte de la solución, lo cual implica que pueda ser importado como librería o ser consumido como un servicio.
- **Responsabilidad única:** Debe de cumplir el principio de responsabilidad única, es decir, debe ser diseñado para realizar una sola tarea, o un conjunto de tareas relacionadas.
- **Alta cohesión:** El componente debe agrupar solo clases, métodos u operaciones que estén enfocadas únicamente a resolver la misma problemática.
- **Bajo acoplamiento:** Debemos evitar al máximo el acoplamiento de nuestro componente con otras dependencias, pues recordemos que las dependencias de nuestro módulo se pueden convertirán en dependencias del sistema que utilice nuestro componente.
- **Fácil de utilizar:** Debido a que la idea es que sea reutilizable, debemos de pensar que gente no involucrada con el módulo querrá implementarlo, por lo que debemos de diseñarlo para que sea fácil de utilizar, tenga documentación y las operaciones y objetos expuestos sean lo bastante descriptivos.
- **Encapsulado:** La encapsulación es muy importante, pues evita que gente sin conocimiento modifique propiedades que podrían afectar el funcionamiento, por lo que, en su lugar, debemos exponer operaciones de alto nivel que puedan ser consumidas fácilmente por los usuarios.

Además de las características anteriores, también es importante considerar que, si hacemos los componentes con tecnologías interoperables como REST o SOA podríamos aumentar su reutilización, pues permite que puedan ser consumidos sin importar el lenguaje de programación con el que fue desarrollado nuestro componente.

Estilos arquitectónicos como SOA, REST y Microservicios (analizaremos los estilos arquitectónicos más adelante) motivan el desarrollo de componentes de software que expongan servicios reutilizables, utilizando protocolos de comunicación que no dependen de la tecnología ni el lenguaje de programación, como es el caso de HTTP e intercambio de documentos como JSON o XML que pueden ser procesados por cualquier lenguaje de programación sin problemas.



Nuevo concepto: Interoperabilidad

La interoperabilidad es la habilidad de organizaciones y sistemas dispares y diversos para interaccionar con objetivos consensuados y comunes, con la finalidad de obtener beneficios mutuos. La interacción implica que las organizaciones involucradas compartan información y conocimiento a través de sus procesos de negocio, mediante el intercambio de datos entre sus respectivos sistemas de tecnología de la información y las comunicaciones.

— Comisión Europea

Un componente se puede reutilizar básicamente de dos formas, importarlos como librería, o consumiéndolo como un recurso externo, en los dos casos tenemos una reutilización, sin embargo, tiene diferentes implicaciones.

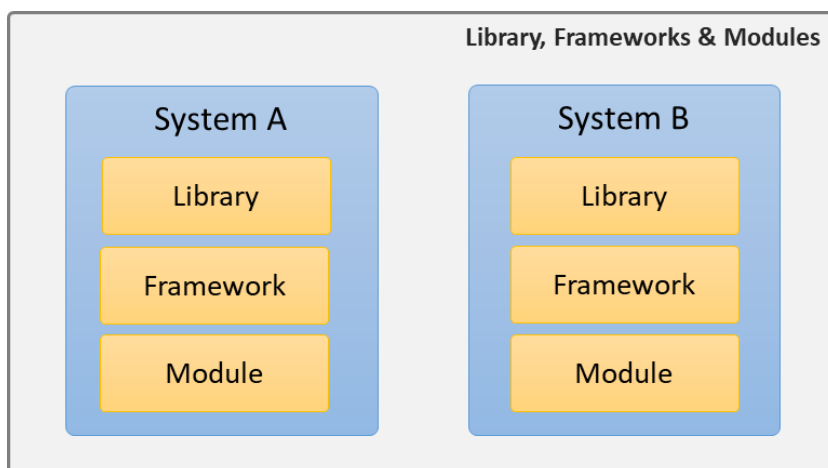


Fig 26 - Dependencias importadas

La imagen anterior representa cuando desde nuestros sistemas (System A y System B) importamos directamente las dependencias que son necesarias para la compilación del sistema, en este caso, es necesario así porque nos proporciona método, operaciones y clases que podemos utilizar directamente desde nuestro sistema. Por otra parte, una copia del módulo es compilado junto con el sistema,

por lo que cualquier actualización en alguna librería requerirá una nueva compilación y versión del sistema.

En este tipo de dependencias tenemos un acoplamiento alto, pues dependemos directamente de las dependencias desde la compilación, además, estamos atados a la versión exacta del componente sobre el cual construimos el componente, pues versiones superiores o inferiores podrían romper la compatibilidad. Además, este tipo de dependencias son un poco más inseguras, pues puede inyectar puertas traseras a nuestros sistemas que pueden ser explotadas para atacar el sistema, por lo que hay que tener cuidado de lo que instalamos y el proveedor que proporciona esa librería, framework o módulo.

Una segunda forma de reutilizar es consumir los módulos como servicios:

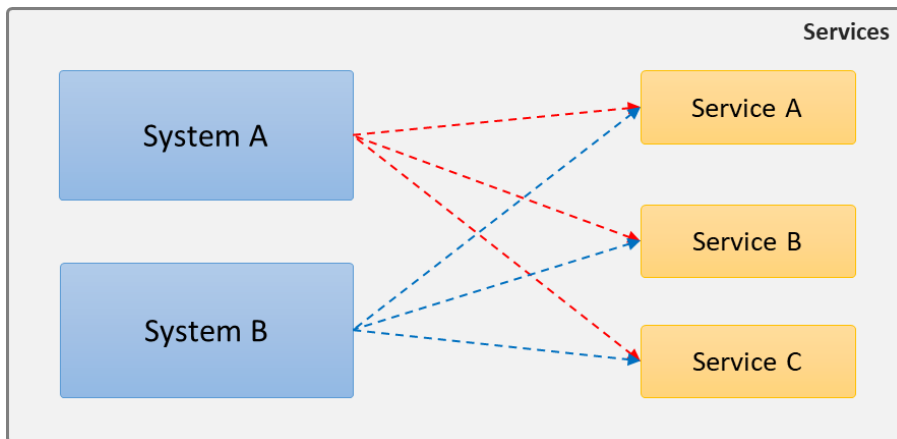


Fig 27 - Dependencia con servicios

La dependencia con servicios externos tiene la ventaja de que está débilmente acoplado, pues la única dependencia es sobre un recurso en internet, que puede estar o no disponible, pero el hecho de que no esté disponible no es algo que impida que nuestro programa compile o incluso se ejecute, por otra parte, el servicio puede tener cuantas dependencias o cambios de versión y no impactar a nuestros sistemas, siempre y cuando no modifique los contratos de comunicación.

Un contrato se podría definir como el formato que se acordó previamente entre las dos partes, de tal forma que los dos sepan que mensaje y formato se enviarán y recibirán.

Una de las desventajas, sobre todo si es un componente externo, es que no sabemos dónde guarda la información o que protocolos de seguridad tiene para cuidar los datos que le enviamos. Otra de las desventajas es que un fallo implicará que todos los sistemas dependientes comiencen a fallar, pues todo harán referencia a un solo servicio, en lugar de tener copias del código como es el caso de las librerías.

Testabilidad

La testabilidad o capacidad de prueba del software es la capacidad para hacer fallar lo más simplemente posible un componente de software por medio de pruebas.

Partiendo de la idea de que todos los softwares fallan, podemos decir entonces que la testabilidad es la capacidad para hacer que esos errores salgan a la luz lo más rápido posible durante la fase de pruebas, evitando tener el menor número de errores en producción.

La testabilidad se logra desarrollando pequeñas unidades de software que puedan ser probadas de forma individual, de tal forma que en lugar de tener que ejecutar una serie de pasos para llegar a una funcionalidad determinada, debemos de tener la capacidad de enviar parámetros directamente a esa unidad de software para comprobar su funcionalidad.

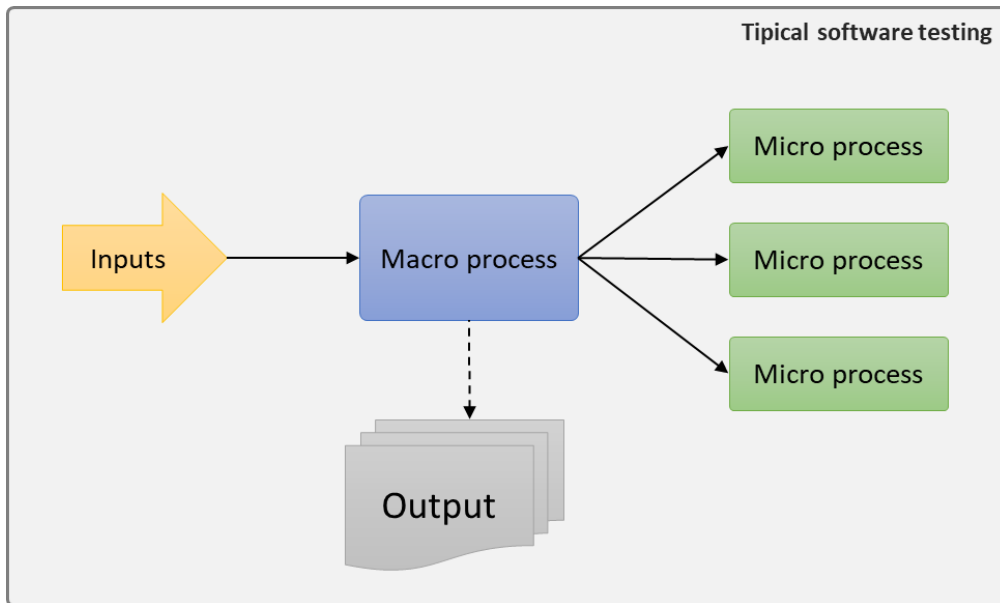


Fig 28 - Pruebas típicas del software.

La imagen anterior muestra la forma en que típicamente se realizan las pruebas, en el cual un usuario le envía ciertos parámetros a un macro proceso, el cual podría ser, por ejemplo, crear una factura, levantar un pedido, etc. En este escenario, el macro proceso recibe los parámetros de entrada y ejecuta varios micro procesos que en su conjunto terminan haciendo la tarea principal y dando como resultado una salida, sin embargo, un error es creer que este esquema de pruebas es adecuado, pues solo probamos el macro proceso en su conjunto y no tenemos forma de probar de forma individual cada micro proceso.

Para realizar una prueba adecuada del software, es necesario tener una suite de pruebas que estén dirigidas a cada micro proceso, de esta forma, podremos validar cómo se comporta ante parámetros que el Macro proceso no enviaría normalmente, pero que en algún momento se podría dar.

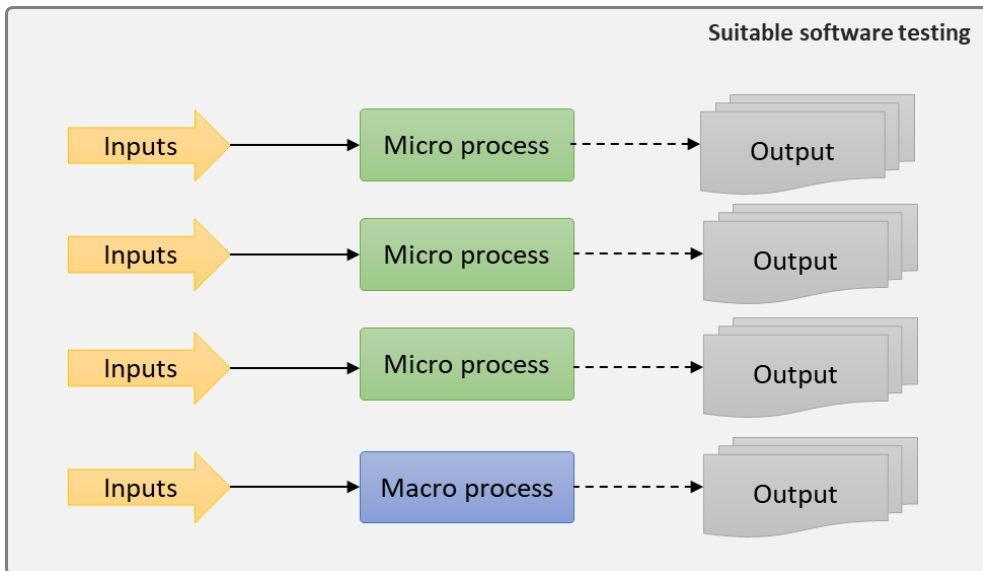


Fig 29 - Adecuada ejecución de las pruebas.

En la imagen anterior se representa la forma adecuada en la que un software debería ser probado, en la cual, cada unidad de software por más pequeña que sea, debe ser probada de forma individual y exponerla ante parámetros que quizás no se presentarían de forma habitual, pero que igual podrían hacer que fallen.

En este nuevo esquema es un hecho que un usuario no podrá realizar las pruebas, porque ya no se trata de que entre a una pantalla y genere una factura o una orden de compra, en su lugar, estamos realizando pruebas a nivel de objetos, operaciones o procedimientos, los cuales requieren un nivel técnico, es por ello que muchas de estas pruebas las tendrá que realizar el mismo desarrollador y son conocidas como pruebas unitarias (Unit Test).



Nuevo concepto: Pruebas unitarias (Unit Test)

Una prueba unitaria es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejemplo, en diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo

de lo que se devuelve. Si el estado inicial es válido, entonces el estado final es válido también.

— Wikipedia

Las pruebas de unidad se pueden hacer con herramientas especializadas como es el caso de JUnit para Java, CUnit para C, PHPUnit para PHP PyUnit para Python, etc. La idea es que el programador sea el responsable de asegurarse de que todas las unidades de software liberadas hayan pasado por una serie de pruebas y con ello garantizar la calidad del software liberado.

Probar las unidades de software garantiza que el software funcionará coherentemente sin importar los inputs que reciba, incluso en desarrollos futuros para los cuales quizás no fue pensado en el inicio.

Otra de las ventajas de las pruebas unitarias es que son programadas y se pueden ejecutar en cualquier momento, incluso como una tarea posterior a la compilación o el despliegue, de esta forma, no tenemos que repetir las pruebas de forma manual cada vez que hagamos un cambio, en su lugar, solo basta ejecutar el set de pruebas unitarias después de realizar un cambio para garantizar que lo que ya estaba funcionando, continúe funcionando, o quien no ha escuchado el dicho de “arreglo una cosa pero descompuso dos”, pues bien, las pruebas unitarias buscan mitigar precisamente eso.

La siguiente imagen representa la pirámide de pruebas, la cual es una de las gráficas más aceptadas con respecto a la importancia y tipos de pruebas:

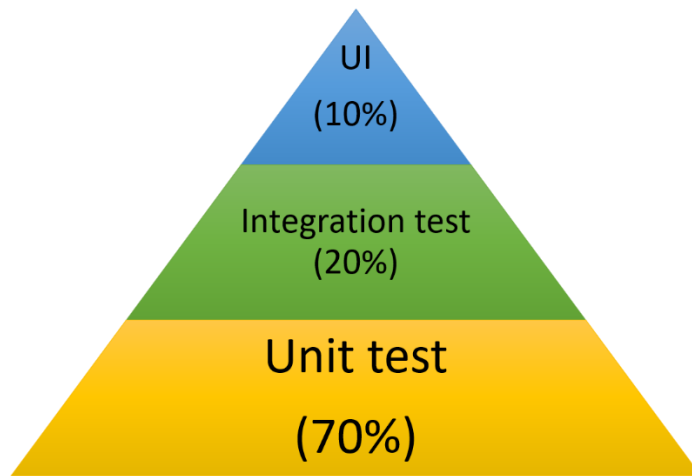


Fig 30 - Pirámide de pruebas.

Cómo podemos ver en la gráfica, el 70% de los errores se pueden detectar en las pruebas unitarias, mientras que en las pruebas de integración se encuentran el 20%, finalmente, en las pruebas realizadas sobre el sistema terminado (UI) solo se detecta el 10%.

A pesar de estas métricas, la realidad es que mucho del software que se desarrollan no tiene pruebas unitarias, lo que provoca una gráfica invertida o también llamada de "cono de nieve", en la cual terminamos detectando la gran cantidad de las fallas en la UI, lo que tiene un costo altísimo, tanto económico como de tiempo, y ni hablar del desgaste de los usuario o inconformidad del cliente.

Escalabilidad

La escalabilidad es la capacidad del software para adaptarse a las necesidades de rendimiento a medida que el número de usuarios crece, las transacciones aumentan y la base de datos empieza a sufrir desgaste del performance.

Hoy en día es muy común escuchar a los arquitectos de software decir que su solución es **robusta, segura y escalable** (Si, ¡cómo no!), pero la realidad es que pocas aplicaciones están realmente preparadas para ser escaladas, ya que desde su diseño de arquitectura no fueron concebidas para soportarlo o realmente no está muy claro que es realmente escalamiento.

Como ya dijimos, el escalamiento es la capacidad del software para adaptarse al creciente número de usuario, transacciones, etc. pero ¿cómo un sistema se puede preparar para crecer indeterminadamente? La realidad es que las aplicaciones no pueden crecer infinitamente, por lo que siempre es clave determinar desde el diseño, el grado de escalamiento que una aplicación podrá soportar, ya que por más estrategias que utilicemos, el sistema llegará el momento que simplemente no dará más.

Puede que los algoritmos utilizados no se puedan optimizarse más, o la arquitectura no fue diseñada para un volumen tan alto de carga, por otro lado, podríamos agregar más memoria al servidor, más disco duro, cambiar a SSD, poner más cores, mejorar el enfriamiento, etc, etc. Sin embargo, llegara un momento que el hardware no puede crecer más y tus sistemas simplemente explotaran.

Escalabilidad Horizontal y Vertical

En la práctica existen muchas formas de hacer que un software sea escalable, ya que podemos combinar técnicas de software y hardware e incluso arquitecturas de RED (por qué no), pero en esta ocasión me quiere centrar en la escalabilidad horizontal y vertical, porque sin duda es una de las características más importantes para sistemas de alta demanda o uso crítico.

Bien, como ya dijimos, existen dos tipos de escalamiento, Horizontal y Vertical, pero ¿qué quiere decir cada uno?

Escalabilidad vertical

La escalabilidad vertical o escalamiento hacia arriba, este es el más simple, pues significa crecer el hardware de uno de los servidores, es decir aumentar el hardware por uno más potente, como disco duro, memoria, procesador, etc. pero también puede ser la migración completa del hardware por uno más potente. El esfuerzo de este crecimiento es mínimo, pues no tiene repercusiones en el software, ya que solo será respaldar y migrar los sistemas al nuevo hardware.

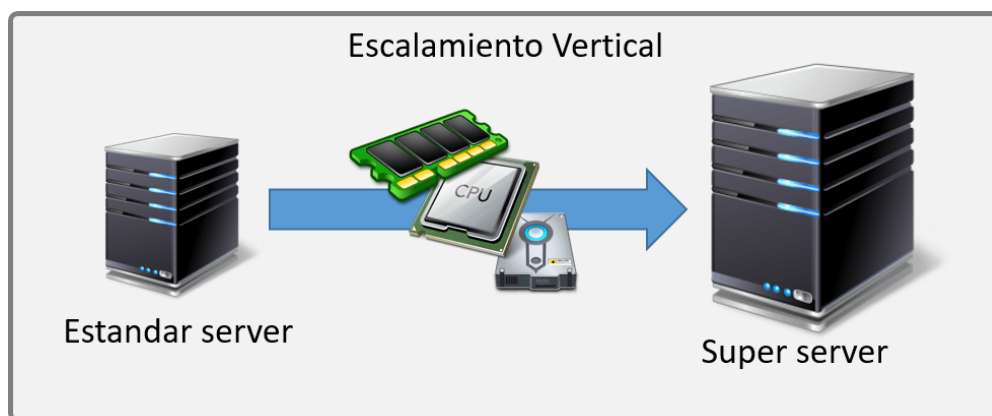


Fig 31 - Escalamiento vertical (hacia arriba)

¿Bastante fácil no?, la realidad es que este tipo de escalamiento tiene algunos aspectos negativos, ya que nuestro crecimiento está ligado al hardware, y este; tarde o temprano tendrá un límite, llegara el momento que tengamos el mejor procesador, el mejor disco duro, la mejor memoria y no podamos crecer más, o podríamos comprar el siguiente modelo de servidores que nos costara un ojo de la cara y el rendimiento solo mejorar un poco, lo que nos traerá el mismo problema el próximo año.

Ahora bien, no significa que este modelo de escalamiento sea malo, ya que lo podemos combinar con el escalamiento horizontal para obtener mejores resultados.

Ventajas:

- No implica un gran problema para las aplicaciones, pues todo el cambio es sobre el hardware
- Es mucho más fácil de implementar que el escalamiento horizontal.

- Puede ser una solución rápida y económica (compara con modificar el software)

Desventajas:

- El crecimiento está limitado por el hardware.
- Una falla en el servidor implica que la aplicación se detenga.
- No soporta la Alta disponibilidad.
- Hacer un upgrade del hardware al máximo puede llegar a ser muy caro, ya que las partes más nuevas suelen ser caras con respecto al rendimiento de un modelo anterior.
- Actualizar el hardware implica detener la aplicación.

Escalabilidad horizontal

El escalamiento horizontal es sin duda el más potente, pero también el más complicado. Este modelo implica tener varios servidores (conocidos como Nodos) trabajando como un todo. Se crea una red de servidores conocida como Cluster, con la finalidad de repartirse el trabajo entre todos nodos del cluster, cuando el performance del cluster se ve afectada con el incremento de usuarios, se añaden nuevos nodos al cluster, de esta forma a medida que es requeridos, más y más nodos son agregados al cluster.



Nuevo concepto: Clúster

El término clúster (del inglés cluster, que significa grupo o racimo) se aplica a los conjuntos o conglomerados de computadoras unidos entre sí normalmente por una red de alta velocidad y que se comportan como si fueran una única computadora.

— Wikipedia

Para que el escalamiento horizontal funcione deberá existir un servidor primario desde el cual se administra el cluster. Cada servidor del cluster deberá tener un software que permite integrarse al cluster, por ejemplo, para las aplicaciones Java,

tenemos los servidores de aplicaciones como Weblogic, Widfly, Websphere, etc. y sobre estos se montan las aplicaciones que queremos escalar.

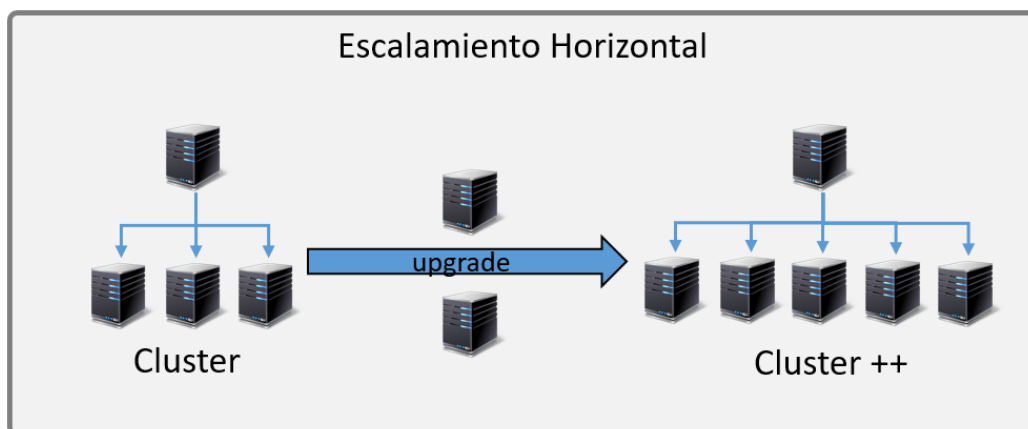


Fig 32 - Escalamiento horizontal

Ventajas:

- El crecimiento es prácticamente infinito, podríamos agregar cuantos servidores sean necesarios
- Es posible combinarse con el escalamiento vertical.
- Soporta la alta disponibilidad
- Si un nodo falla, los demás sigue trabajando.
- Soporta el balanceo de cargas.

Desventajas:

- Requiere de más mantenimiento
- Es difícil de configurar
- Requiere de grandes cambios en las aplicaciones (si no fueron diseñadas para trabajar en cluster)
- Requiere de una infraestructura más grande.

Como estaremos viendo más adelante, existen arquitecturas como los microservicios que fomentan el uso de contenedores como Docker para virtualizar

nuestras aplicaciones en un mismo servidor, lo que implica que no necesariamente necesitamos más servidores sino más bien más contenedores.

Failover

A pesar de que escalamiento vertical no soporta de forma natural la alta disponibilidad, sí que es posible habilitarla mediante una estrategia llamada Failover, la cual consiste en que cuando el servidor falla, mediante hardware se redireccionan las peticiones a un servidor secundario que es un espejo del principal, este servidor normalmente se encuentra en stand by, y solo se habilita cuando el primario falla, de la misma manera, cuando el primario se recupera, el secundario vuelve a pasar a stand by.

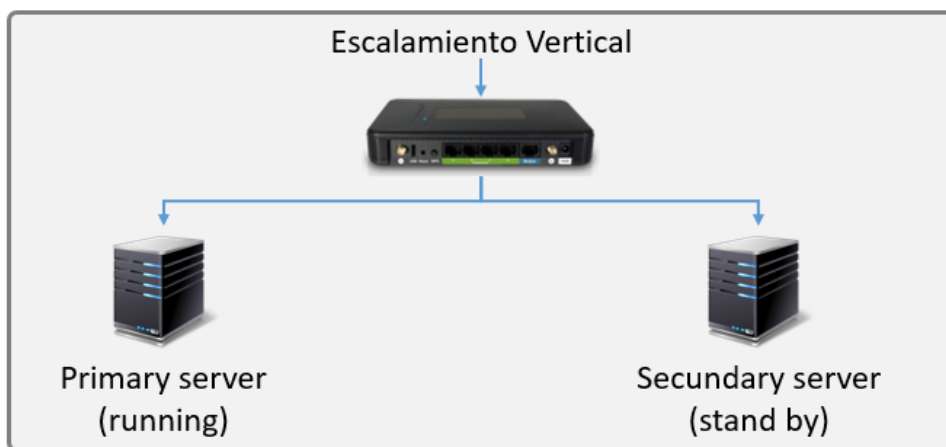
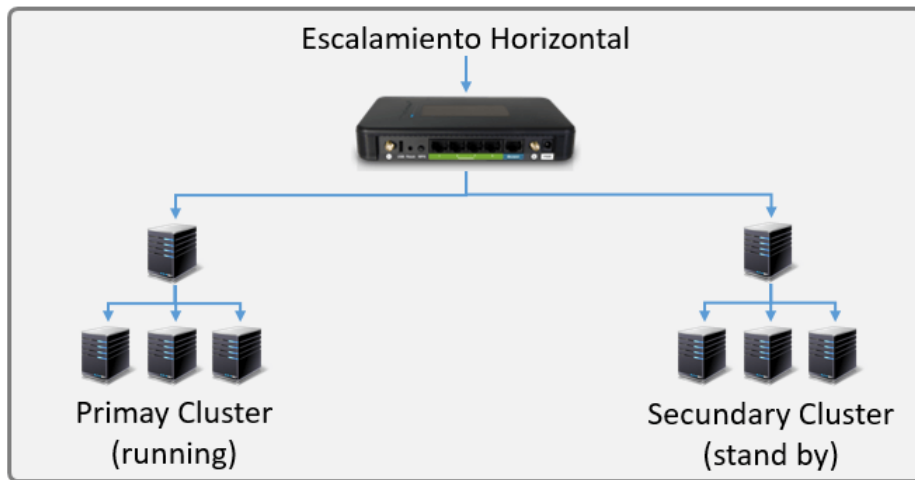


Fig 33 - Failover vertical

Por otra parte, la escalabilidad horizontal ya utiliza de forma natural el Failover, ya que, al haber varios nodos, es posible pasar la carga al resto de nodos activos en caso de que uno falle, ahora bien, esto no garantiza una alta disponibilidad total, ya que, si el servidor primario falla puede tumbar todo el cluster, por lo que también se suele utilizar un ambiente espejo del cluster para garantizar la alta disponibilidad. Cabe mencionar que esto solo lo hacen empresas gigantes, ya que administrar dos cluster completos es un verdadero reto.



Hoy en día, las aplicaciones son parte fundamental de las empresas y una falla en ellas puede incluso detener la operación de la empresa, llevándolas a perder grandes cantidades de dinero, es por eso que es sumamente importante tener en mente el escalamiento de las aplicaciones y siempre tener un plan B en caso de que estas fallen, ¿Te imaginas la cantidad de dinero que perdería por hora, si la página de Amazon se callera? probablemente millones. Es por eso que debemos de estar preparados para escalar nuestras aplicaciones.

Si en este punto aun no estás seguro que escalamiento te va mejor, puedes analizar lo siguiente, si tu aplicación es de baja demanda y el número de usuarios está pensado para crecer poco a poco, la mejor solución puede ser el escalamiento vertical, ahora, si esta misma aplicación a pesar de su poco crecimiento es crítica para el negocio, entonces la puedes combinar con Failover para asegurar la alta disponibilidad.

Por otra parte, si tu aplicación esta pensaba para un crecimiento acelerado y crees que un servidor pueda quedar chico en un corto o mediano plazo, lo mejor será que empieces a pensar en escalamiento horizontal. Ahora bien, si además del crecimiento la aplicación es de uso crítico, lo mejor será implementar un Failover. Si ya está utilizando escalamiento horizontal, recuerda que también puedes realizar escalamiento vertical a cada nodo del cluster, siempre y cuando el escalamiento vertical resulte más económico que agregar un nodo más.

Estilos arquitectónicos

Capítulo 4

En la primera unidad tocamos brevemente que eran los estilos arquitectónicos y mencionábamos que 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, que cuenta con ciertas características que lo hacen notable y hacen que un edificio u otra estructura sea notable o históricamente identificable.**

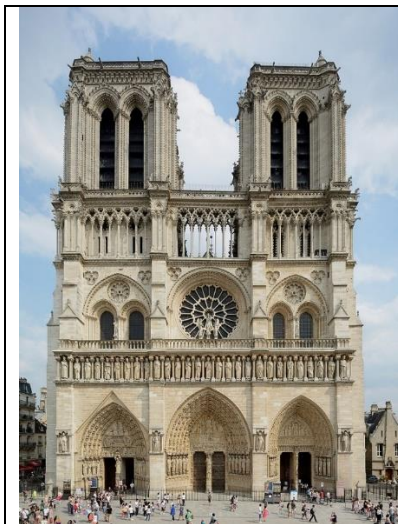
También mencionábamos que esta misma analogía aplica exactamente igual para el software, pues un estilo arquitectónico determina las características que debe tener un componente que utilice ese estilo, lo cual hace que sea fácilmente reconocible. De la misma forma que podemos determinar a qué periodo 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.



Recordemos la definición de “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.

Para comprender que son los estilos arquitectónicos, me gustaría basarme en algunas construcciones importantes, con la finalidad de poder comprender mejor un concepto algo abstracto, para ello, quiero que vean las siguientes imágenes:



Catedral de Nuestra Señora de París (*Notre-Dame*)



catedral de Amiens



Abadía de Westminster

Las imágenes anteriores son algunas de las construcciones góticas más emblemáticas, las cuales además de ser hermosas y que son patrimonio de la humanidad, comparten algunas características similares. A pesar de que estas estructuras no son iguales, sí que podemos ver una extraordinaria similitud, por ejemplo, todas tienen grandes arcos en las entradas, tienen puertas muy grandes y en pares, tienen techos muy altos, tienen vitrales redondos y muy similares, están construidas con piedra, etc, etc, etc. No necesitamos ser unos expertos para darnos cuenta que estas tres estructuras comparten un estilo similar.

A sí cómo es posible identificar y clasificar las estructuras basadas en sus atributos, también es posible identificar y clasificar los estilos arquitectónicos, sin embargo, el software no es muy fotogénico que digamos, por lo que no podremos sacar una foto e identificar rápidamente a qué estilo arquitectónico pertenece, pero sí que es posible analizar sus características para clasificarlos adecuadamente. En el caso del software, podemos darnos cuenta a qué estilo arquitectónico pertenece al ver sus diagramas de arquitectura o analizar su funcionamiento.

Por otra parte, en la construcción existen muchos estilos arquitectónicos, como el Gótico, Románico, Clásico, Barroco, Neoclásico, Bauhaus, Moderno, Post-moderno, etc, de la misma forma, en el software existen muchos estilos arquitectónicos, los cuales tiene una serie de características que hace fácil de identificarlos. En este capítulo profundizaremos en los estilos arquitectónicos más importantes y aprenderemos a analizarlos.

Cabe mencionar que los estilos arquitectónicos no determinan la tecnología en la cual está construido el software, ni determina los detalles técnicos de cómo debe de construirse, en su lugar, solo da ciertos lineamientos y características que debe de cubrir un software para considerarse que sigue un determinado estilo arquitectónico.



Los estilos arquitectónicos no determinan la tecnología ni los detalles de implementación

Un error común es creer que los estilos arquitectónicos determinan las tecnologías a utilizar o que definen de alguna forma como los componentes deben de ser construidos. Recuerda que en su lugar, solo brindan ciertos lineamientos que ayudan a clasificar al software por medio de sus características.

Comprender los estilos arquitectónicos nos colocan en una mejor posición como arquitectos, pues podemos identificar y diseñar soluciones que sigan determinados estilos arquitectónicos, además, podremos identificar que estilos se aplican mejor para resolver ciertos problemas, podremos identificar sus puntos fuertes y sus puntos débiles. Los estilos arquitectónicos más importante son:

Monolítico

El estilo arquitectónico monolítico consiste en crear una aplicación autosuficiente que contenga absolutamente toda la funcionalidad necesaria para realizar la tarea para la cual fue diseñada, sin contar con dependencias externas que complementen su funcionalidad. En este sentido, sus componentes trabajan juntos, compartiendo los mismos recursos y memoria. En pocas palabras, **una aplicación monolítica es una unidad cohesiva de código.**

Un monolítico podrías estar construido como una sola unidad de software o creada a partir de varios módulos o librerías, pero lo que la distingue es que **al momento de compilarse se empaqueta como una sola pieza**, de tal forma que todos los módulos y librerías se empaquetarán junto con la aplicación principal.

El estilo monolítico no es algo que haya sido planeado o ideado por alguien en particular, si no que todas las aplicaciones al inicio de la computación nacían con este estilo arquitectónico. Solo hace falta recordar los sistemas antiguos, donde todo funcionaba en una súper computadora, la cual realizaba todas las tareas. Recordemos que al inicio no existían el internet, por lo que no había forma de consumir servicios externos para realizar determinadas tareas, en su lugar, el sistema monolítico tenía que implementar absolutamente toda la funcionalidad necesaria para funcionar, y de esta forma ser auto suficiente.

Con el tiempo, llegó el internet y con ello la posibilidad de consumir servicios externos, llegaron arquitecturas modulares que permitían separar el código en unidades de software más manejables, cohesivas y fáciles de administrar, sin embargo, con todos estos avances, siguen existiendo las aplicaciones Monolíticas, las cuales son vistas por los inexpertos como algo malo o incluso como un Anti patrón, pero la realidad es que esto está muy alejado de la realidad.

A pesar de todo el estigma que tienen las aplicaciones monolíticas, la realidad es que, hasta el día de hoy, las aplicaciones monolíticas siguen teniendo un protagonismo muy importante y siguen existiendo caso donde son totalmente indispensables para mantener la operación de las empresas.

Puede parecer un poco tonto el solo hecho de pensar en hacer una aplicación monolítica hoy en día, sin embargo, todavía hay escenarios donde son totalmente necesarias, solo imagina el sistema de venta y facturación de una pequeña empresa,

el software de los equipos médicos, programas de escritorio, como procesadores de texto o incluso sistemas más completos como los clásicos CRM o ERP.

Todos estos sistemas muchas veces funcionan de forma independiente, sin acceso a internet y necesitan una autonomía total, solo imagina que un equipo médico no funcione si no se puede conectar a internet o que necesite de servicios externos para operar, eso podría costar vidas, o que el cajero de una tienda no pueda vender o administrar su inventario porque una dependencia no está disponible, eso podría costar la pérdida de ventas. Lo cierto es que las aplicaciones monolíticas son cada vez menos atractivas, pero hasta el día de hoy, tiene aplicaciones donde difícilmente podrán ser remplazadas.

Una falsa creencia es que, una aplicación monolítica es un caos por dentro, donde todo el código está amontonado, no hay una estructura clara y que por lo general tiene miles de clases u objetos, sin embargo, esto es solo una mala fama que se le ha dado, si bien es verdad que se podía dar el caso, recordemos que eso también se podría dar en cualquier estilo de arquitectura, pues eso dependen más bien del programador y no del estilo arquitectónico.

Otra falsa creencia es creer que las aplicaciones Monolíticas son solo las aplicaciones grandísimas que hacen un montón de cosas, pero lo cierto es que un monolítico puede ser de una sola clase, o de miles, lo que define un estilo monolítico no es el número de clases, archivos o líneas de código, lo que lo define es que es autosuficiente, es decir, que tiene toda la funcionalidad para operar por sí mismo y sin depender de nadie más.



Tip: Independencia

Quizás la palabra que mejor define a un monolítico es la Independencia, pues es la capacidad más notable que tiene.

Como se estructura un Monolítico

En los Monolítico podemos tener una serie de paquetes bien organizados y un código muy claro, donde cada paquete puede tener cierta parte de la funcionalidad

y están desacoplados uno de otro (recordemos que eso es independiente del estilo arquitectónico). Sin embargo, al momento de compilarse el código, todo se empaqueta como un solo software. Cabe mencionar que cualquier librería que sea requerida, será exportada como parte del archivo compilado de salida.

Algo a tomar en cuenta es que cuando los módulos son compilados por separado y son instalados como complementos ya no se trata de un Monolítico y pasar a ser un estilo arquitectónico de Microkernel, el cual analizaremos más adelante.

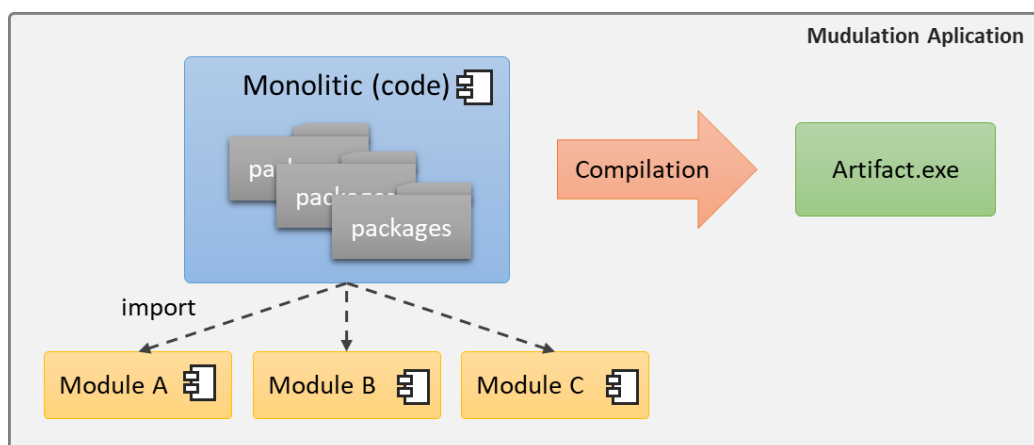


Fig 34 – Proceso de compilación.

En la imagen podemos apreciar cómo funciona el proceso de compilación de una aplicación Monolítica, el cual todos los paquetes junto con sus dependencias son compilados y da como resultado un solo artefacto, el cual incluye todo el código junto con las dependencias. En este ejemplo decimos que hemos creado un EXE, pero se pudo haber creado un Jar en el caso de Java o un JS en el caso de JavaScript, dependiendo la tecnología utilizada tendremos un artefacto diferente, pero al final, todos contendrán todo el código con sus dependencias.

Características de un Monolítico

En esta sección analizaremos las características que distinguen al estilo Monolítico 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. Son aplicaciones autosuficientes (no requieren de nada para funcionar).
2. Realizan de punta a punta todas las operaciones para terminar una tarea.
3. Son por lo general aplicaciones grandes, un que no es un requisito.
4. Son por lo general silos de datos privados, es decir, cada instalación administra si propia base de datos.
5. Todo el sistema corre sobre una solo plataforma.

Ventajas y desventajas

Ventajas

- **Fácil de desarrollar:** Debido a que solo existe un componente, es muy fácil para un equipo pequeño de desarrollo iniciar un nuevo proyecto y ponerlo en producción rápidamente.
- **Fácil de escalar:** Solo es necesario instalar la aplicación en varios servidores y ponerlo detrás de un balanceador de carga.
- **Pocos puntos de fallo:** El hecho de no depender de nadie más, mitiga gran parte de los errores de comunicación, red, integraciones, etc. Prácticamente los errores que pueden salir son por algún bug del programador, pero no por factores ajenos.
- **Autónomo:** Las aplicaciones Monolíticas se caracterizan por ser totalmente autónomas (auto suficientes), lo que les permite funcionar independientemente del resto de aplicaciones.

- **Performance:** Las aplicaciones Monolíticas son significativamente más rápidas debido que todo el procesamiento lo realizan localmente y no requieren consumir procesos distribuidos para completar una tarea.
- **Fácil de probar:** Debido a que es una sola unidad de código, toda la funcionalidad está disponible desde el inicio de la aplicación, por lo que es posible realizar todas las pruebas necesarias sin depender de nada más.

desventajas

- **Anclado a un Stack tecnológico:** Debido a que todo el software es una sola pieza, implica que utilicemos el mismo Stack tecnológico para absolutamente todo, lo que impide que aprovechemos todas las tecnologías disponibles.
- **Escalado Monolítico:** Escalar una aplicación Monolítica implica escalar absolutamente toda la aplicación, gastando recursos para funcionalidad que quizás no necesita ser escalada (en el estilo de Microservicios analizaremos como solucionar esto).
- **El tamaño sí importa:** sin albur, las aplicaciones Monolíticas son fáciles de operar con equipo pequeños, pero a medida que la aplicación crece y con ello el equipo de desarrollo, se vuelve cada vez más complicado dividir el trabajo sin afectar funcionalidad que otro miembro del equipo también está moviendo.
- **Versión tras versión:** Cualquier mínimo cambio en la aplicación implicará realizar una compilación del todo el artefacto y con ello una nueva versión que tendrá que ser administrada.
- **Si falla, falla todo:** A menos que tengamos alta disponibilidad, si la aplicación Monolítica falla, falla todo el sistema, quedando totalmente inoperable.
- **Es fácil perder el rumbo:** Por la naturaleza de tener todo en un mismo módulo es fácil caer en malas prácticas de programación, separación de responsabilidades y organización de las clases del código.
- **Puede ser abrumador:** En proyectos muy grandes, puede ser abrumador para un nuevo programador hacer un cambio en el sistema.

Cuando debo de utilizar un estilo Monolítico

Como te comenté hace un momento, las aplicaciones Monolíticas han sido víctimas de una campaña de desprestigio, desde hablar mal de ella, hasta clasificarlo como un Anti patrón, sin embargo, esto es causa de una opinión desinformada o poco profesional, pues como lo comenta el mismísimo Martin Fowler (uno de los arquitectos más reconocidos) en uno de sus artículos, **"no considerar los microservicios a menos que tenga un sistema que sea demasiado complejo para administrarlo como un monolito"** (<https://martinfowler.com/bliki/MicroservicePremium.html>)

Quizás en este momento no sepas que son los Microservicios (los cuales veremos más adelante), pero quiero que te quedes con la idea de que quizás, deberías comenzar la aplicación como un Monolítico y luego ir pensando en otros estilos más sofisticados a medida que la aplicación comience a crecer, ya que arquitecturas más sofisticadas traen consigo cierto grado de complejidad que quizás no valga la pena para empezar.

En resumen, yo diría que utilizar un estilo de Monolítico puede ser una buena opción para aplicaciones que apenas comienza, pues te permitirá comenzar rápido, validar tu producto y sacarlo al mercado, sin embargo, a medida que la aplicación va creciendo, puedes ir pensando en migrar tu aplicación en arquitecturas más modulares como los Microservicios.

Un error frecuente entre los novatos es querer implementar la arquitectura más compleja desde el inicio, haciendo que las ventajas que trae esta arquitectura se conviertan en una carga para el equipo de desarrollo, por la misma complejidad que traen arquitecturas más sofisticadas.

Conclusiones

A pesar de la mala fama que se les ha dado a las aplicaciones Monolíticas, la realidad es que tiene ventajas que hasta el día de hoy son difíciles de igualar, entre las que destacan **su total independencia el performance que puede llegar a alcanzar.**

Utilizar un estilo Monolítico no es para nada malo, incluso en nuestra época, lo malo sería quedarnos siempre atascados en este estilo arquitectónico y no ir migrando a otros estilos más sofisticados a medida que nuestra aplicación va creciendo. Algunos síntomas de que el estilo arquitectónico ya nos comienza a quedar chico sería cuando, cada vez es más difícil dividir el trabajo entre los desarrolladores sin que tengan conflictos con el código, existen una necesidad de escalamiento más quirúrgica y se necesita escalar ciertas funcionalidades y no todo el proyecto, cuando el proyecto es tan grande que es complicado para los desarrolladores comprender el funcionamiento por la complejidad y la absurda cantidad de clases o archivos, y finalmente, cuando necesitamos empezar a diversificarnos con respecto al Stack tecnológico a utilizar.

Cliente-Servidor

Cliente-Servidor es uno de los estilos arquitectónicos distribuidos más conocidos, el cual está compuesto por dos componentes, el proveedor y el consumidor. El proveedor es un servidor que brinda una serie de servicios o recursos los cuales son consumido por el Cliente.

En una arquitectura Cliente-Servidor existe un servidor y múltiples clientes que se conectan al servidor para recuperar todos los recursos necesarios para funcionar, en este sentido, el cliente solo es una capa para representar los datos y se detonan acciones para modificar el estado del servidor, mientras que el servidor es el que hace todo el trabajo pesado.

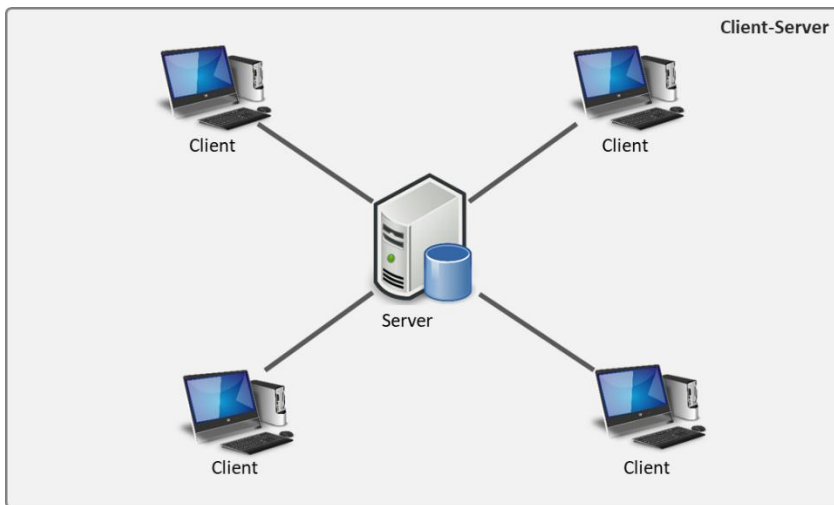


Fig 35 - Arquitectura Cliente-Servidor

En esta arquitectura, el servidor deberá exponer un mecanismo que permite a los clientes conectarse, que por lo general es TCP/IP, esta comunicación permitirá una comunicación continua y bidireccional, de tal forma que el cliente puede enviar y recibir datos del servidor y viceversa.

Creo que es bastante obvio decir que en esta arquitectura el cliente no sirve para absolutamente nada si el servidor no está disponible, mientras que el servidor por sí solo no tendría motivo de ser, pues no habría nadie que lo utilice. En este

sentido, las dos partes son mutuamente dependientes, pues una sin la otra no tendría motivo de ser.

Cliente-Servidor es considerada una arquitectura distribuida debido a que el servidor y el cliente se encuentran distribuidos en diferentes equipos (aunque podrían estar en la misma máquina) y se comunican únicamente por medio de la RED o Internet.

En esta arquitectura, el Cliente y el Servidor son desarrollados como dos aplicaciones diferentes, de tal forma que cada una puede ser desarrollada de forma independiente, dando como resultado dos aplicaciones separadas, las cuales pueden ser construidas en tecnologías diferentes, pero siempre respetando el mismo protocolo de comunicación para establecer comunicación.

La idea central de separar al cliente del servidor radica en la idea de centralizar la información y la separación de responsabilidades, por una parte, el servidor será la única entidad que tendrá acceso a los datos y los servirá solo a los clientes del cual el confía, y de esta forma, protegemos la información y la lógica detrás del procesamiento de los datos, además, el servidor puede atender simultáneamente a varios clientes, por lo que suele ser instalado en un equipo con muchos recursos. Por otro lado, el cliente suele ser instalado en computadoras con bajos recursos, pues desde allí no se procesa nada, simplemente actúa como un visor de los datos y delega las operaciones pesadas al servidor.

Quizás uno de los puntos más característicos de la arquitectura Cliente-Servidor es la centralización de los datos, pues el server recibe, procesa y almacena todos los datos provenientes de todos los clientes. Si bien los clientes por lo general solo se conectan a un solo servidor, existen variantes donde hay clientes que se conectan a múltiples servidores para funcionar, tal es el caso de los navegadores, los cuales, para consultar cada página establece una conexión a un servidor diferentes, pero al final es Cliente-Servidor:

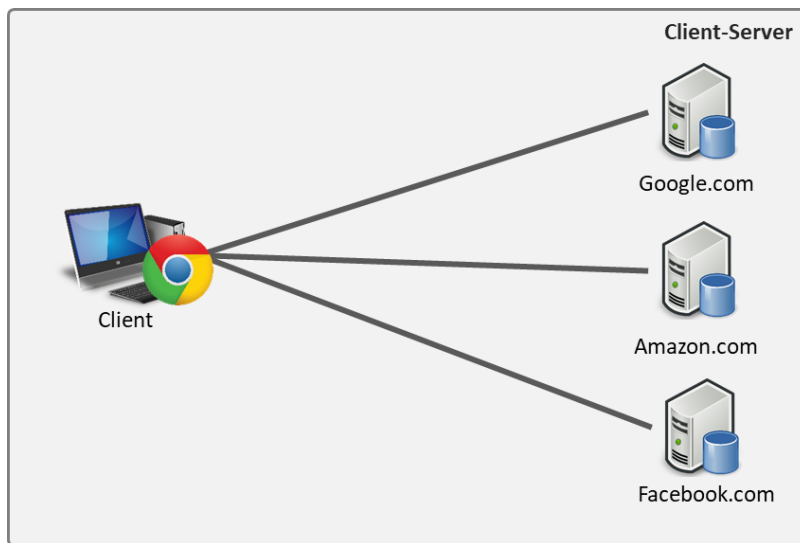


Fig 36 - Múltiples servidores por cliente.

Como podemos ver en la imagen, el navegador actúa como un cliente, pero en lugar de conectarse a un solo servidor, puede conectarse a múltiples servidores. Por otro lado, el navegador no sirve para absolutamente nada sin no podemos acceder a un servidor.

Como se estructura un Cliente-Servidor

Como ya lo mencionamos, el cliente y el servidor son aplicaciones diferentes, por lo que pueden tener un ciclo de desarrollo diferente, así como usar tecnologías y un equipo diferente entre sí. Sin embargo, en la mayoría de los casos, el equipo que desarrolla el servidor también desarrolla el cliente, por lo que es normal ver que el cliente y el servidor están contruidos con las mismas tecnologías.

Como podemos ver en la imagen de abajo, el cliente y el servidor son contruidos en lo general como Monolíticos, donde cada desarrollo crea su propio ejecutable único y funciona sobre un solo equipo, con la diferencia de que estas aplicaciones no son autosuficientes, ya que existe una dependencia simbiótica entre los dos.

En este sentido, es normal tener 3 artefactos, el Cliente, el Servidor y una tercera librería que contiene Objetos comunes entre el servidor y el cliente, esta librería tiene por lo general los Objetos de Entidad, DTO, interfaces y clases base que se

usan para compartir la información, es decir, objetos que se utilizan en las dos aplicaciones y se separan para no repetir código (Principio DRY – Don't repeat yourself), sin embargo, este tercer componente no es obligatorio que exista, sobre todo si el cliente y el servidor utilizan tecnologías diferentes o son implementados con diferentes proveedores.

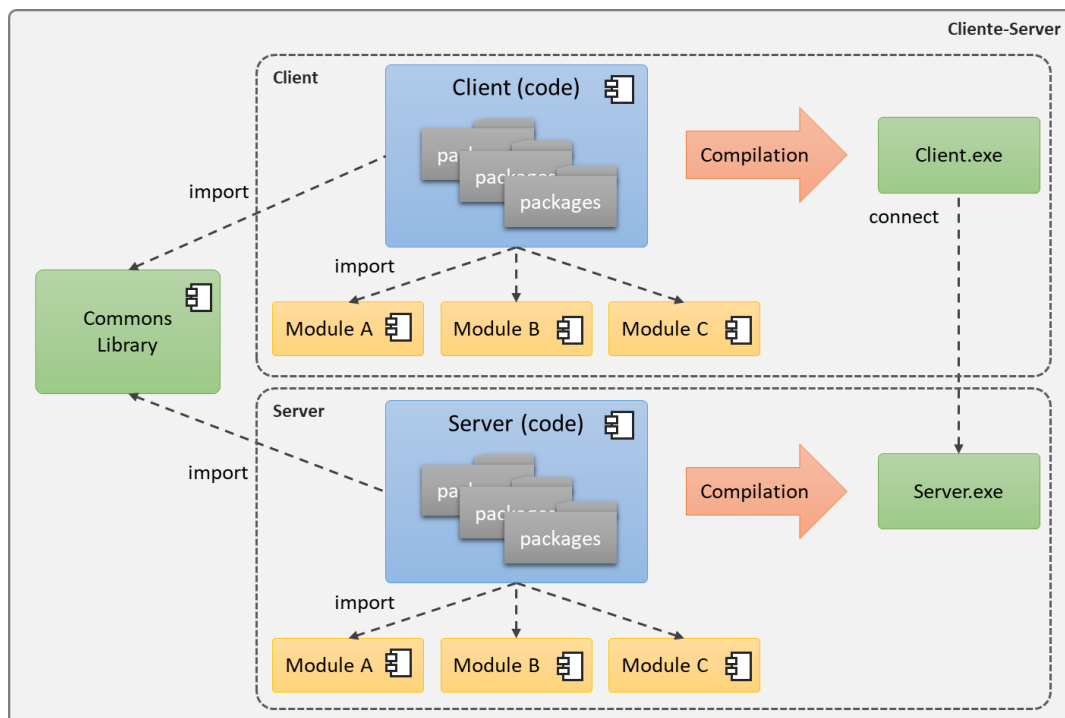


Fig 37 - Estructura de una aplicación Cliente-Servidor.

Sin importar como desarrollamos el cliente y el servidor, lo importante es notar que siempre existirán un cliente y un servidor, donde el cliente expone la funcionalidad y el cliente la consume.

Flujo de Comunicación entre el cliente y el servidor

Quizás una de las dudas más importantes de este estilo arquitectónico es, ¿cómo se hace para que el cliente y el servidor se comuniquen?, ya que suele ser la parte medular de este estilo arquitectónico, así como algo de lo más complicado.

La forma más pura de realizar una comunicación entre un cliente y un servidor es mediante un canal de comunicación directo, por medio de un protocolo de comunicación en común, como podría ser el TCP/IP, el cual abre un Socket de comunicación directo y se realiza una comunicación de muy bajo nivel, en el cual los mensajes deben de ser serializados para ser enviados como pequeños paquetes para ser enviados de un extremo al otro.

Utilizar protocolos de comunicación estándares como es TCP o UDP permite que cualquier cliente, sin importar la plataforma o lenguaje de programación pueden establecer una comunicación directa con el servidor sin ayuda de un intermediario:

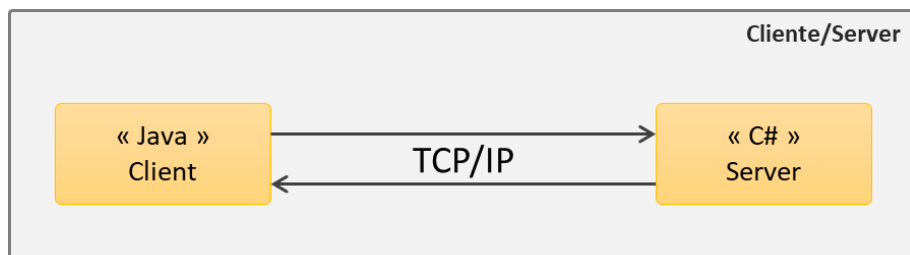


Fig 38 - Comunicación directa entre Cliente y Servidor.

La imagen anterior representa un cliente que se comunica directamente con un servidor, un ejemplo claro de este escenario es un Navegador de internet como Chrome, Firefox, o Edge comunicándose con una página web. Te podrás dar cuenta que el navegador puede estar escrito en cualquier lenguaje de programación y comunicarse con un servidor web el cual también puede estar construido en cualquier lenguaje de programación y sobre cualquier sistema operativo, esa es la flexibilidad que nos brinda establecer comunicación directa mediante protocolos estándar. Sin embargo, este tipo de comunicación suele ser muy complicado, por lo que muchos de los fabricantes de plataformas cliente-servidor brindan API que se encargan de realizar toda la comunicación por nosotros.

Los fabricantes de plataformas cliente-servidor conocen la complejidad de establecer una comunicación a bajo nivel, por lo que optan por crear una serie de

API's para que los clientes puedan utilizarlas y conectarse con el servidor con un API de alto nivel:

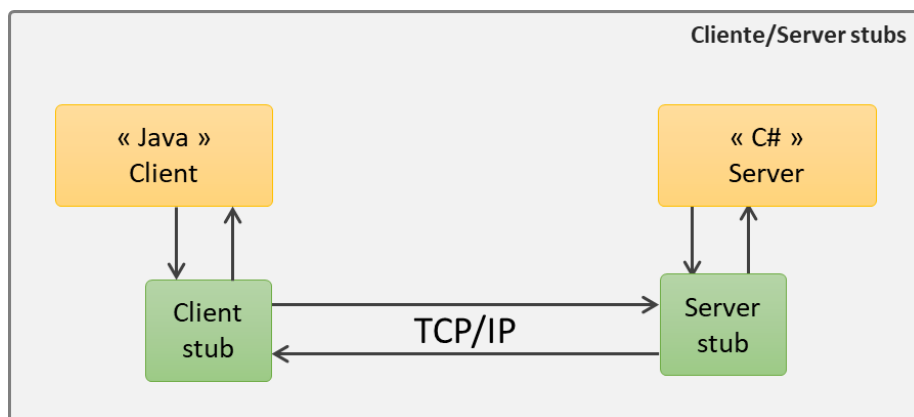


Fig 39 - Comunicación Cliente-Servidor con Stubs.

Como puedes ver en la imagen, los proveedores crean algo llamado Stubs, los cuales son una serie de clases nativas para cada lenguaje de programación. Estas clases tienen una serie de interfaces y objetos que hacen referencia a métodos y objetos del servidor, de tal forma que cuando ejecutamos uno de los métodos del stubs, el API internamente serializa los parámetros, los manda al servidor y los direcciona a una operación que existe en el servidor, cuando el servidor retorna, el Stub del servidor serializa los datos y los envía al cliente; del lado del cliente se recibe la respuesta serializada y el Stub convierte esa respuesta binaria en un Objeto nativo para el lenguaje de programación. Ha esto le llamamos Remote Procedure Call (RPC) o Llamada a Procedimiento Remoto.



Nuevo concepto: Remote Procedure Call (RPC)

En computación distribuida, la llamada a procedimiento remoto (en inglés, Remote Procedure Call, RPC) es un programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas

— Wikipedia

La llamada a procedimientos remotos tampoco es un concepto fácil de dominar, pues requiere profundizar en temas como CORBA (Common Object Request Broker Architecture), los cuales quedan fuera de alcance de este libro.



Nuevo concepto: Common Object Request Broker Architecture (CORBA)

Es un estándar definido por OMG que permite que diversos componentes de software escritos en múltiples lenguajes de programación y que corren en diferentes computadoras, puedan trabajar juntos; es decir, facilita el desarrollo de aplicaciones distribuidas en entornos heterogéneos.

— Wikipedia

Como podemos ver en este ejemplo, el API se encarga de absolutamente toda la comunicación y la complejidad que esto conlleva, pero nada de esto es gratis, pues el proveedor del sistema cliente servidor se tendrá que encargar de construir y mantener el API para cada lenguaje o plataforma a la cual quiere dar soporte.

Uno de los ejemplos más simples para comprender como funciona una conexión por API son los Driver de comunicación con la Base de datos, por ejemplo, en Java tenemos los famosos JDBC Driver y en C# están los Driver OLE, estos lo que hace es simplificar la comunicación entre la base de datos y nuestra aplicación, de tal forma que nosotros solo tenemos que decir a que servidor nos queremos conectar, el usuario y el API se encargará de todo lo demás.

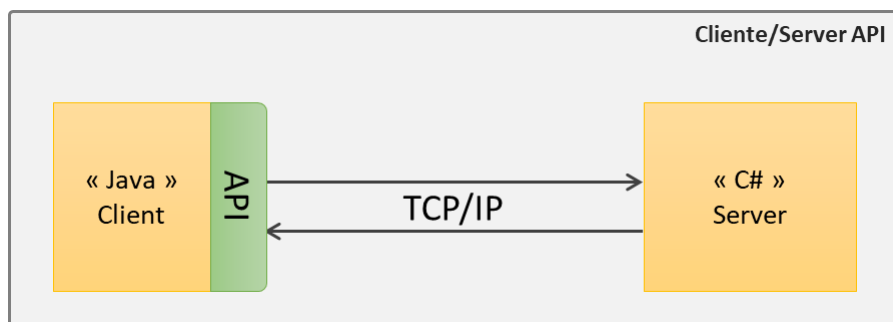


Fig 40 - Comunicación Cliente-Servidor con API.

En este caso, la API no hace llamada a un procedimiento remoto (RPC) sino que simplemente se encarga de gestionar la conexión con el servidor por nosotros.

La realidad es que hoy en día casi el total de la comunicación cliente-servidor se hace mediante API's, pues pocos dominan la complejidad de una comunicación cliente-servidor a bajo nivel.

Características de Cliente-Servidor

En esta sección analizaremos las características que distinguen al estilo Cliente-Servidor 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:

- Este estilo se distingue por que siempre existe una aplicación que es un cliente y otra aplicación que actúa como servidor y el cliente es siempre el que inicia la conexión con el servidor.
- Requieren una conexión activa mediante todo el tiempo que estén trabajando.
- Este estilo permite que los componentes estén montados en equipos diferentes, incluso, fuera de la red local y accedido por medio de internet.
- El cliente y el servidor pueden evolucionar a velocidades diferentes y ser desarrollados por equipos y tecnologías diferentes.

Ventajas y desventajas

Ventajas

- **Centralización:** El servidor fungirá como única fuente de la verdad, lo que impide que los clientes conserven información desactualizada.
- **Seguridad:** El servidor por lo general está protegido por firewall o subredes que impiden que los atacantes pueden acceder a la base de datos o los recursos sin pasar por el servidor.
- **Fácil de instalar (cliente):** El cliente es por lo general una aplicación simple que no tiene dependencias, por lo que es muy fácil de instalar.
- **Separación de responsabilidades:** La arquitectura cliente-servidor permite implementar la lógica de negocio de forma separada del cliente.
- **Portabilidad:** Una de las ventajas de tener dos aplicaciones es que podemos desarrollar cada parte para correr en diferentes plataformas, por ejemplo, el servidor solo en Linux, mientras que el cliente podría ser multiplataforma.

desventajas

- **Actualizaciones (clientes):** Una de las complicaciones es gestionar las actualizaciones en los clientes, pues puede haber muchos terminales con el cliente instalado y tenemos que asegurar que todas sean actualizadas cuando salga una nueva versión.
- **Concurrencia:** Una cantidad no esperada de usuarios concurrentes puede ser un problema para el servidor, quien tendrá que atender todas las peticiones de forma simultánea, aunque se puede mitigar con una estrategia de escalamiento, siempre será un problema que tendremos que tener presente.
- **Todo o nada:** Si el servidor se cae, todos los clientes quedarán totalmente inoperables.
- **Protocolos de bajo nivel:** Los protocolos más utilizados para establecer comunicación entre el cliente y el servidor suelen ser de bajo nivel, como Sockets, HTTP, RPC, etc. Lo que puede implicar un reto para los desarrolladores.
- **Depuración:** Es difícil analizar un error, pues los clientes están distribuidos en diferentes máquinas, incluso, equipos a los cuales no tenemos acceso, lo que hace complicado recopilar la traza del error.

Cuando debo de utilizar un estilo Cliente-Servidor

Hoy en día no es muy común desarrollar aplicaciones que sigan el estilo arquitectónico Cliente-Servidor, no porque sea malo, sino porque recientemente han salido nuevos estilos que se adaptan a las necesidades de desarrollo modernos, sin embargo, el estilo Cliente-Servidor sigue teniendo terrenos en los que es difícil de superar y que a pesar de que surjan nuevas arquitecturas, esta no ha podido ser remplazada.

Hoy en día hay dos tipos de escenarios donde sigue siendo ideal:

Aplicaciones en tiempo real

Las aplicaciones en tiempo real son todas aquellas que necesitan reflejar los datos al instante, es decir, que no existe un tiempo de espera para actualizar la página o la pantalla.

Las aplicaciones en tiempo real necesitan mecanismos de comunicación orientados a la conexión, es decir que mantengan un canal de comunicación ininterrumpido que les permitan recibir o enviar datos al momento, sin tener que esperar un tiempo para ir a buscar los nuevos datos.

Hoy en día tenemos tecnologías como los Sockets y Web Sockets que permiten establecer una comunicación direccional con el servidor mediante el protocolo TCP/IP, el cual es el más utilizado para este estilo arquitectónico.

La ventaja de usar Sockets es que es posible enviar datos al cliente o al servidor de forma inmediata cuando algún cambio ocurra, de tal forma que se evita tener que ir al servidor cada X tiempo a buscar si hay nuevas actualizaciones.

Aplicaciones que muestran la fluctuación de la bolsa de valores o el sistema de radares aéreos son ejemplos claros donde es necesario tener los datos actualizados todo el tiempo, pues tener datos desfasados podría hacer que tomemos decisiones equivocadas.

Aplicaciones centralizadas de alto rendimiento

Como ya mencionamos al inicio de esta sección, el estilo Cliente-Servidor se distingue por que permite centralizar los datos, la seguridad y la lógica de negocio en el servidor, por ello, las aplicaciones donde requieren tener un solo origen confiable de datos podrían ser candidatos para una arquitectura cliente servidor.

Ejemplos claros de estos son los motores de base de datos, los cuales centralizan la seguridad y la información, de tal forma que, al conectarnos, estamos seguros que son la única fuente de información confiable y que nadie podrá tener un dato diferente a ella.

Desde luego que otras arquitecturas podrán cumplir la tarea de centralizar los datos, pero la arquitectura cliente-servidor permite mantener comunicación constante lo que mejora el performance, por no requerir establecer conexión cada vez que es necesario consultar o guardar datos.

Por otro lado, base de datos en tiempo real como Firebase (de Google), permite recibir eventos en tiempo real al navegador a medida que los datos cambian en la base de datos, mediante triggers que son comunicados por WebSockets.

Otros ejemplos podrían ser los sistemas de mensajería, los cuales requiere de conexiones activas para recibir notificaciones cada vez que un nuevo mensaje es entregado a una cola de mensajes para procesarlo lo antes posible.

Conclusiones

A pesar de que el estilo Cliente-Servidor no es muy popular entre los nuevos desarrolladores, la realidad es que sigue siendo parte fundamental un muchas de las arquitecturas de hoy en día, solo basta decir que todo el internet está basado en Cliente-Servidor, sin embargo, no es común que como programadores o arquitectos nos encontremos ante problemáticas que requieran implementar un Cliente-Servidor, ya que estas arquitecturas están más enfocadas a aplicaciones CORE o de alto rendimiento, que por lo general es encapsulado por un Framework o API.

A pesar de que puede que no te toque implementar una arquitectura Cliente-Servidor pronto, sí que es importante entender cómo funciona, pues muchas de las herramientas que utilizamos hoy en día implementan este estilo arquitectónico y ni nos damos cuenta, como podrían ser la base de datos, internet, sistemas de mensajería (JMS, MQ, etc), correo electrónico, programas de chat tipo Skype, etc.

La realidad es que Cliente-Servidor es la base sobre la que está construida gran parte de la infraestructura tecnológica que hoy tenemos, pero apenas somos capaces de darnos cuenta.

Peer-to-peer (P2P)

El estilo arquitectónico Red entre iguales (Peer-to-peer, P2P) es una red de computadoras donde todos los dispositivos conectados a la red actúan como cliente y servidor al mismo tiempo. En esta arquitectura no es necesario un servidor central que administre la red (aunque puede existir), si no que todos los nodos de la red pueden comunicarse entre sí.

La arquitectura P2P es para muchos solo una variante de la arquitectura Cliente-Servidor, sin embargo, tiene una diferencia importante que hace que la podamos clasificar como un estilo arquitectónico independiente, y es que la arquitectura Cliente-Servidor tiene como punto medular la centralización, mientras la arquitectura P2P busca la descentralización.

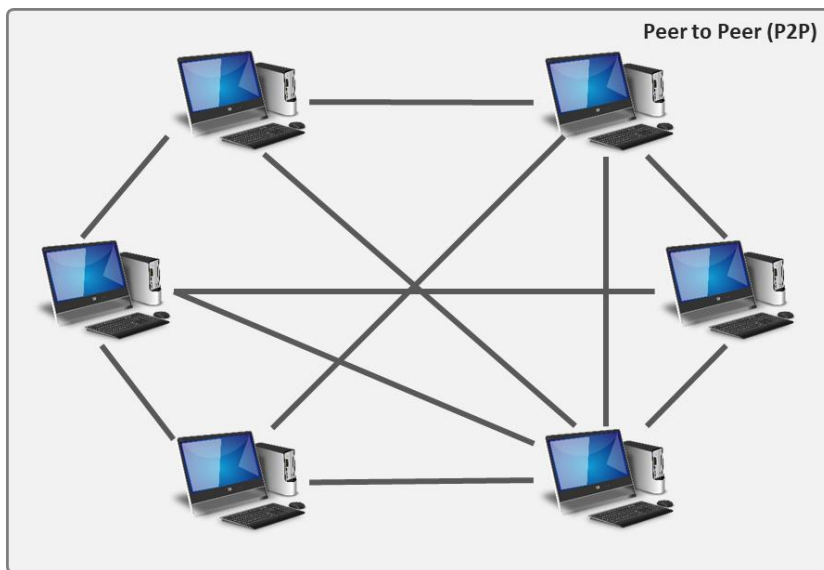


Fig 41 - Arquitectura Peer to Peer (P2P)

La arquitectura P2P es como si uniéramos el Cliente y el Servidor en una sola aplicación, lo que permite conectarse a otras computadoras de la red para consumir los recursos expuestos por los otros nodos de la red, pero al mismo

tiempo, funciona como un Servidor, lo que permita que otros nodos se conecten a nuestro software para leer los recursos que nosotros exponemos.

Algo interesante de esta arquitectura es que el software puede funcionar como Cliente y Servidor al mismo tiempo, por lo que podríamos estar compartiendo recursos al mismo tiempo que consumir recursos de otro nodo de la red.

Si bien, P2P utiliza los mismos conceptos que una arquitectura Cliente-Servidor para conectar a todos los nodos de la red, la verdad es que buscan solucionar problemas diferentes. Por una parte, la arquitectura Cliente-Servidor solo tiene un servidor, el cual centraliza los datos, recursos, seguridad, lógica de negocio, etc. De tal forma que, si el servidor se cae, se cae todo el sistema, por otra parte, la arquitectura P2P busca que no exista un servidor central, si no que cada computadora dentro de la red funcione como un Cliente y un Servidor al mismo tiempo, de tal forma que, entre más usuarios conectados a la red, más servidores se unen también a la red.

En una arquitectura Cliente-Servidor entre más usuario conectados existan, más carga se acumula sobre el servidor, mientras que en una arquitectura P2P, entre más computadoras conectadas existan en la red, más poder de procesamiento se agrega, lo que lo vuelve una red con una increíble capacidad de escalamiento.

Como se estructura P2P

Para comprender mejor como es que una aplicación P2P se estructura, es importante conocer las diferentes formas de implementarlo. A pesar de que la arquitectura P2P busca la descentralización y que mencionamos que no es necesario un servidor central, existen variantes que lo requieren, por lo que la arquitectura P2P puede variar en función del objetivo que estamos buscando resolver. Para esto, podemos clasificar una arquitectura P2P basado en su grado de centralización:

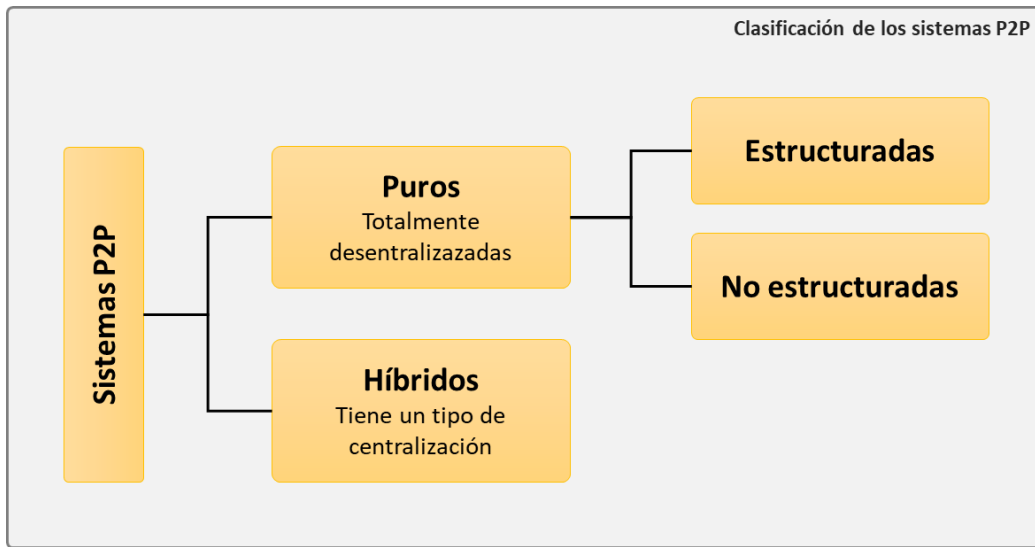


Fig 42 - Clasificación de los sistemas P2P.

Arquitectura P2P Puro no estructuradas

Este tipo de arquitectura conforman una red totalmente descentralizada, donde no existe un servidor central ni roles donde algún nodo de la red tenga más privilegios o responsabilidades, además, todos los pares actúan como cliente y servidor al mismo tiempo, estableciendo una comunicación simétrica entre sí.

En esta arquitectura un par solo se puede comunicar con los pares que conozca, es decir, que conoce la ubicación exacta. Ha estos pares conocidos se le conoce como vecinos, por lo que cada par puede tener un número de vecinos diferentes:

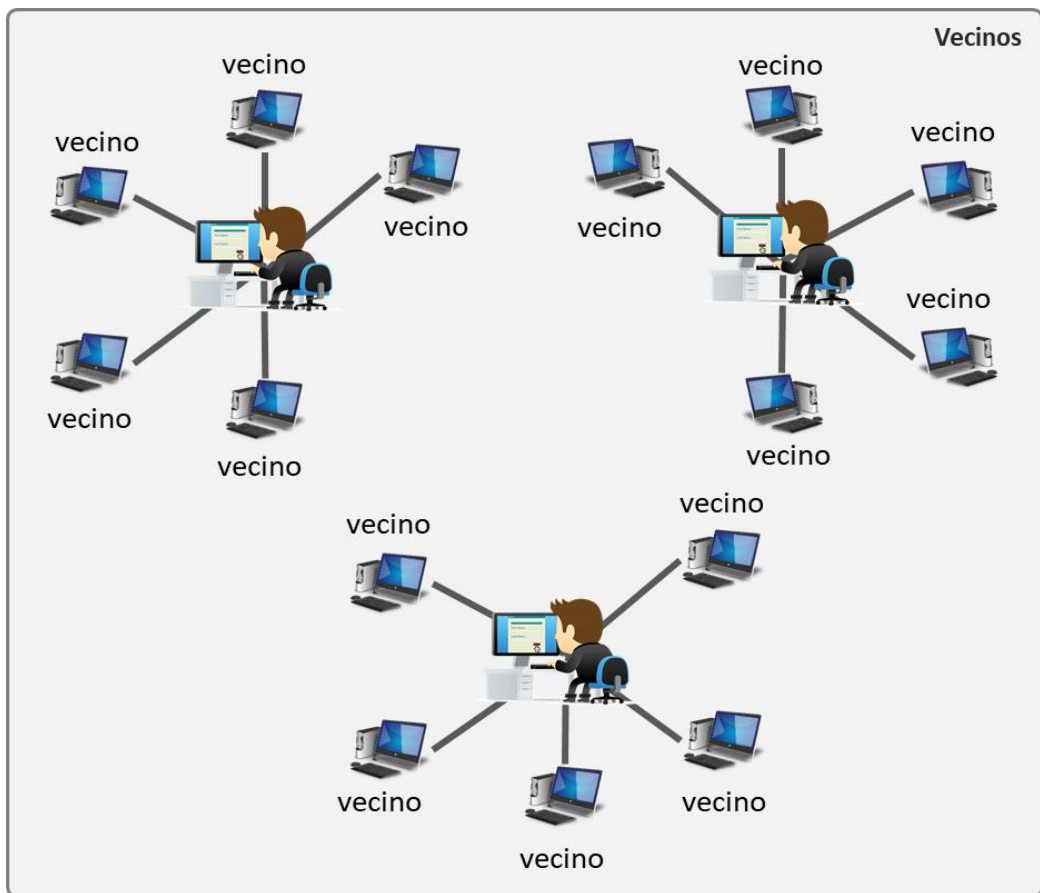


Fig 43 - Vecinos en un sistema P2P.

Como podemos ver en la imagen, pueden existir muchas sub-redes donde un par conoce a algunos vecinos y otras sub-redes conocen a otros vecinos diferentes, y a pesar de que todos son partes de la red P2P no todos los pares se conocen entre sí, por lo que un par solo podrá localizar y buscar recursos sobre los vecinos que conozco, lo que provoca que sea imposible llegar a los pares que no conocemos.

Limitar la búsqueda de recursos solo a los pares que conocemos es una gran desventaja de esta arquitectura, pues estaríamos perdiendo la oportunidad de acceder a todos los demás recursos que ofrecen los pares desconocidos, por lo que esta arquitectura implementa un algoritmo llamado "inundación".

La inundación es el proceso por medio de cual, un vecino que recibe una solicitud de búsqueda, replica la solicitud de búsqueda sobre sus vecinos, los cuales harán

los mismo con sus otros vecinos. Para comprender mejor esto, un par que desea localizar un recurso envía una petición a sus vecinos, quienes a su vez la reenviarán a todos sus otros vecinos, logrando con esto llegar a nodos más alejados de la red inicial de vecinos.

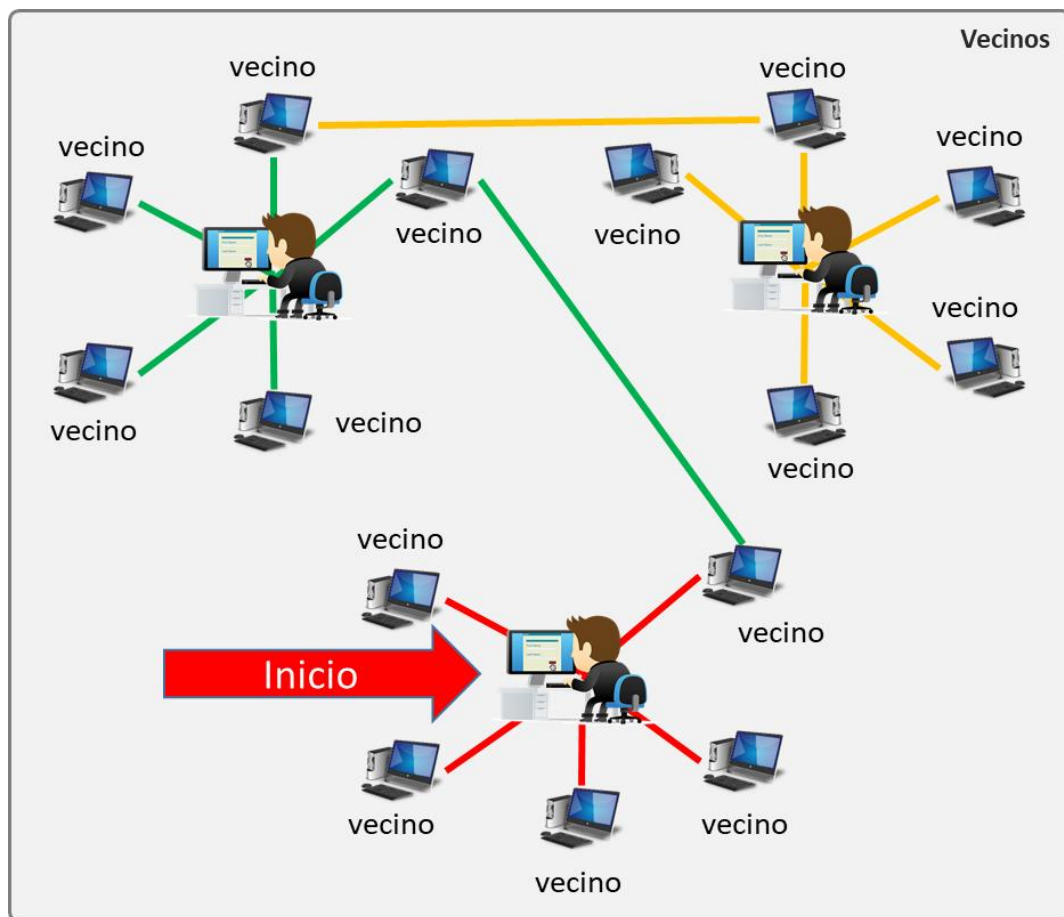


Fig 44 - Inundación de la red.

Analicemos la imagen anterior. Imaginemos que nosotros somos la persona que inicia la búsqueda de un recurso (fecha roja), lo que haremos primero será realizar una petición de búsqueda sobre los únicos pares que conocemos, es decir, nuestros vecinos (líneas rojas), esto provocará que nuestros vecinos repliquen esa búsqueda sobre sus otros vecinos (líneas verdes), esto provocará que alcancemos

pares más allá de nuestra red de vecinos. Ahora bien, resulta que los vecinos de nuestros vecinos también tienen otros vecinos, por lo que replican nuevamente la búsqueda a sus otros vecinos (líneas amarillas), de esta forma estamos inundando la red para alcanzar la mayor cantidad de pares sin necesidad de un servidor central.

Una importante desventaja de la búsqueda por inundación es que genera una gran cantidad de tráfico en la red por la replicación de las peticiones, lo que provoca que una búsqueda crezca exponencialmente a medida que se va replicando por la red, por lo que este algoritmo tiene un candado para evitar esto, y es mediante un campo llamado TTL (time to live), que es básicamente un timeout para evitar replicar la búsqueda una vez el tiempo de espera termine. Esto desde luego tiene el problema de que hace que sea imposible alcanzar los nodos más alejados de nuestra red, pero a cambio de no colapsar la red completa.

Algo a tomar en cuenta es que en una red P2P pura no estructurada, todos los nodos de la red son sometidos a una misma carga de trabajo, pues todos los nodos son tratados como iguales y la carga de trabajo es equilibrada entre todos los nodos de la red.

Debido a la naturaleza totalmente distribuida hace que sea casi o imposible detenerla, pues no existe un único punto al cual podamos golpear para que toda la red caiga, al contrario, entre más usuario se conectan, más crece la red y más difícil de hacer de parar, al mismo tiempo, como las búsquedas se hacen por inundación, es casi imposible rastrear cual fue el nodo origen de la petición lo que tiene un grado de privacidad muy grande, y más si a esto le sumamos una conexión cifrada.

En las arquitecturas P2P la escalabilidad suele ser una de las mayores ventajas, pues a mayor cantidad de pares mayor será la cantidad de recursos y poder de procesamiento se unen a la red, sin embargo, para las redes P2P puras sin estructura es todo lo contrario, debido al excesivo tráfico que genera la llegada masiva de nuevos nodos a la red que realizan búsquedas. Como ejemplo, tenemos la aplicación Gnutella, la cual ya no puede escalar a los niveles que son demandados hoy en día para la búsqueda de archivos por internet.

Finalmente, hablábamos de que la búsqueda por inundación permite localizar nodos más allá de nuestros vecinos, pero limitados al campo TTL o timeout de la búsqueda, por lo que buscar archivos escasos o raros en la red puede llegar a ser

imposible, pues muchas veces el timeout se produce antes de llegar al recurso buscado, para resolver este tipo de problemas tenemos la **arquitectura P2P pura estructurada**, la cual veremos a continuación.

Algunos ejemplos de aplicaciones que utilizan esta arquitectura son Gnutella y FreeNet

Arquitectura P2P pura estructurada

La arquitectura P2P estructurada es una variante de la P2P no estructurada, por lo que mucho de la teoría planteada anteriormente aplica exactamente igual, sin embargo, la arquitectura P2P estructurada cambia la forma de buscar los recursos. En lugar de usar el método por inundación, busca los recursos basados en Tablas Hash Distribuidas (DHT por sus siglas en inglés).

Esta tabla hash mantiene un registro de todos los recursos disponibles en la red en un formato Clave-Valor (Key-Value), donde la clave es un hash del recurso y el valor corresponde a la ubicación (nodo) donde se encuentra el recurso, de esta forma, solo es necesario buscar el clave hash del recurso en la tabla para conocer su ubicación, sin embargo, no están fácil como parece. Debido a que la tabla hash es distribuida, no existe físicamente un solo servidor, sino que está distribuida entre todos los nodos de la red, y cada nodo guarda un fragmento del índice total de los recursos compartidos.

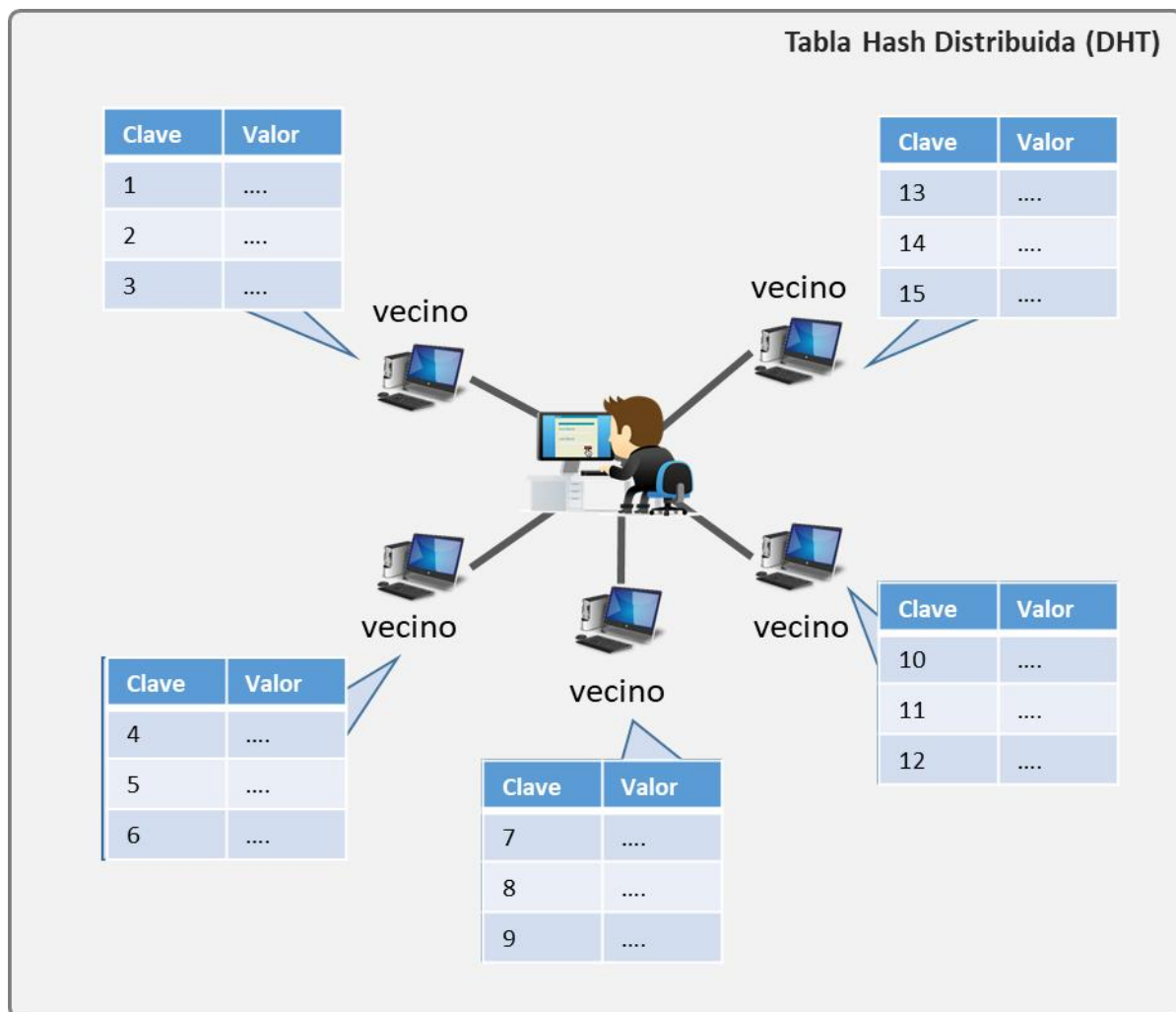


Fig 45 - Búsqueda por medio de tablas hash distribuidas.

Como podemos ver en la imagen, cada nodo es responsable de almacenar y tener actualizada una fracción de la tabla hash, evitando que un solo nodo tenga toda la tabla o que todos los nodos tengan toda la tabla. Sin embargo, todavía queda una pregunta clave por resolver, y es ¿cómo sabe cada nodo que parte debe administrar?

Cuando un nuevo nodo entra a la red se le asigna un ID y es colocado en una estructura de anillo, donde es colocado en orden por medio del ID que se le asignó,

de esta forma, cada nodo se compromete a administrar todos los hashes cuyo valor sea mayor al nodo anterior del anillo y menor o igual a su ID:

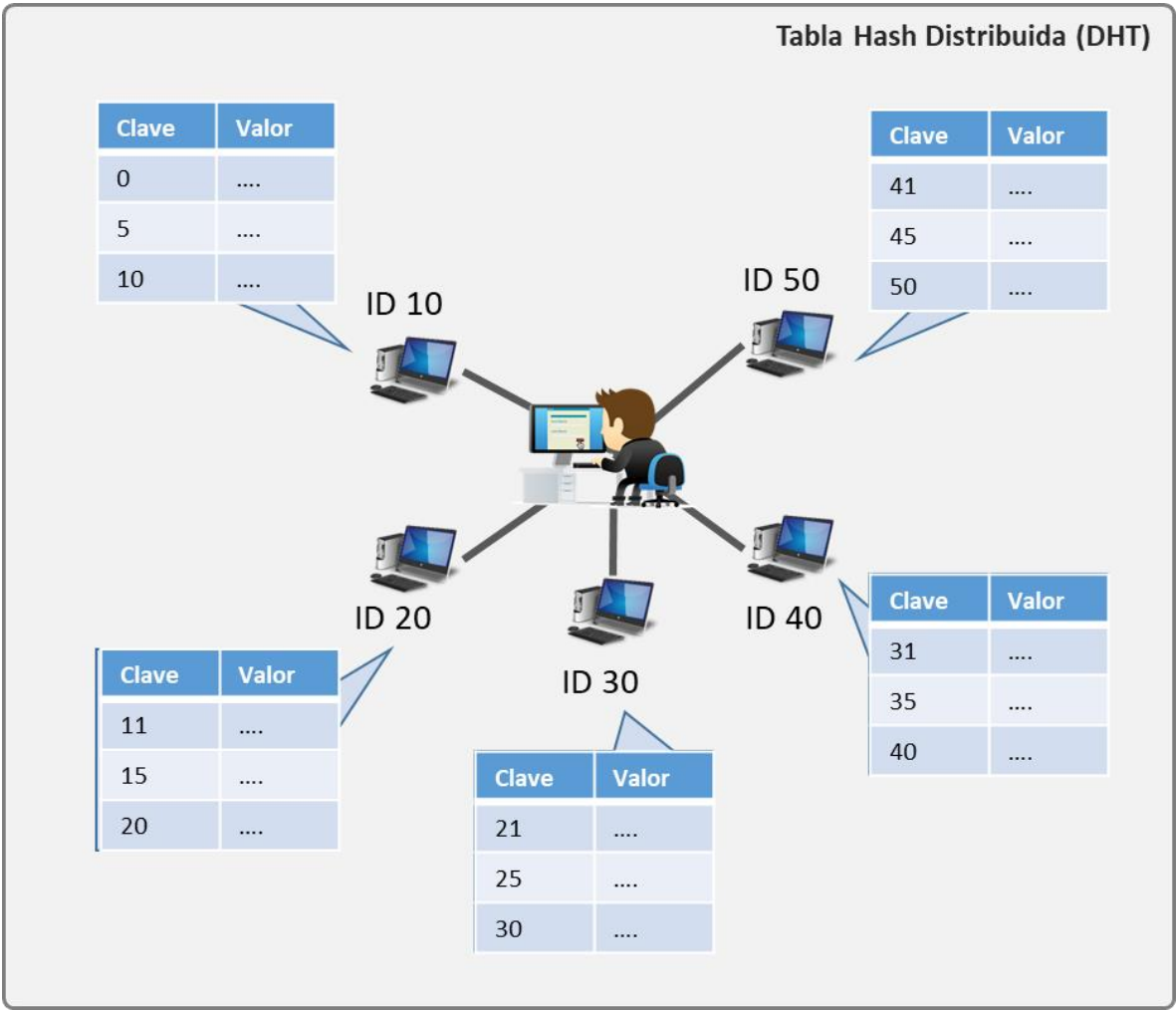


Fig 46 - Asignación de Hash por medio del ID

Analicemos la imagen anterior para entender cómo se administra la tabla hash distribuida, por ejemplo, supongamos que nuestro nodo se le asigna el ID 10, esto significa que debemos almacenar todos los ID del 0 al 10, 0 porque no existe un nodo anterior, y 10 porque es su ID, por otra parte, si somos el ID 20, debemos administrar los hashes del 11 al 20, 11 porque es un número mayor al ID anterior

(que es 10) y 20 porque es ID del nodo. Esta misma regla se aplica para todos los nodos de la red.

Una vez que tenemos lista la tabla hash distribuida, solo basta resolver como es que buscamos un recurso, ya que decíamos que la inundación no es eficiente. La respuesta es fácil, en lugar de replicar la consulta a todos nuestros vecinos, solo le solicitamos la búsqueda a nuestro vecino que tenga el ID más cercano al hash que estamos buscando, de la misma, forma, ese vecino validará si el hash que se está buscando está dentro de su rango, si no es así, entonces replicará nuevamente la búsqueda al vecino con el ID más cercano al hash del recurso buscado, hasta que finalmente se llega al nodo que tiene el hash buscado y nos retornará la ubicación del recurso.

Cuando un nodo entre a la red deberá buscar a su sucesor y predecesor en la red utilizando el mismo mecanismo mencionado anteriormente, de esta forma podrá saber los rangos de llaves que deberá administrar, por otra parte, si una parte del rango a administrar está siendo administrado por otro nodo, se inicia un proceso por medio del cual el predecesor y el sucesor le envían las llaves que quedarán ahora en resguardo del nuevo nodo y posteriormente el predecesor y el sucesor eliminarán de su tabla.

De la misma forma que existe un proceso al entrar a la red, existe otro proceso para salir de ella, pues en caso de salir, deberá enviar su tabla hash al predecesor, sin embargo, no siempre se ejecuta este proceso, sobre todo cuando hay salidas súbitas del sistema, como un reinicio o apagón, en tal caso, la red sigue funcionando, pero pierde eficiencia, pues el bloque de hash no estará disponible hasta que el nodo se conecte de nuevo.

En esta arquitectura podemos destacar la eficiencia con la que un recurso es encontrado, incluso si solo existe una sola copia en toda la red, pues el algoritmo de hash permitirá llegar a él, sin embargo, esta arquitectura tiene algunas desventajas, cómo el costoso trabajo de mantenimiento de las tablas hash, ya que mucho de los usuarios de hoy en día, entran y salen con mucha frecuencia, lo que trae muchos problemas para mantener actualizada la tabla hash distribuida, por otra parte, no es posible realizar búsquedas complejas, pues la tabla hash solo guarda el hash del archivo, que por lo general es el hash del nombre del archivo, lo cual limita la búsqueda solo por nombre.

Algunos ejemplos de programas que utilizan esta arquitectura son: Chord, CAN, Tapestry, Pastry y Kademlia

Arquitectura P2P Híbridos

La arquitectura P2P híbrida o también conocida como Centralizada se caracteriza por tener un servidor central que sirve de enlace entre los nodos de la red, de tal forma que cualquier solicitud para consumir los recursos de otros nodos deberán pasar primero por este servidor.

La diferencia fundamental que tiene esta arquitectura con el Cliente-Servidor es que, este servidor no es el que proporciona los recursos, sino que solo sirve de enlace para conectar con otros nodos. Un ejemplo claro de esta arquitectura es Napster, un programa muy famoso para descargar música en formato MP3. En este sentido, el servidor central de Napster no es quien almacena la información, sino que solo tiene el registro de todos los nodos conectados a la red y que archivos tiene cada nodo para compartir, de esta forma, cuando lanzamos una búsqueda, el servidor central de Napster sabe que nodos tiene ese archivo y nos dirige para que comencemos a descargar el archivo directamente sobre el nodo que tiene el archivo sin necesidad de un intermediario, de esta forma, Napster nunca tiene los archivos, pero sabe quién sí.

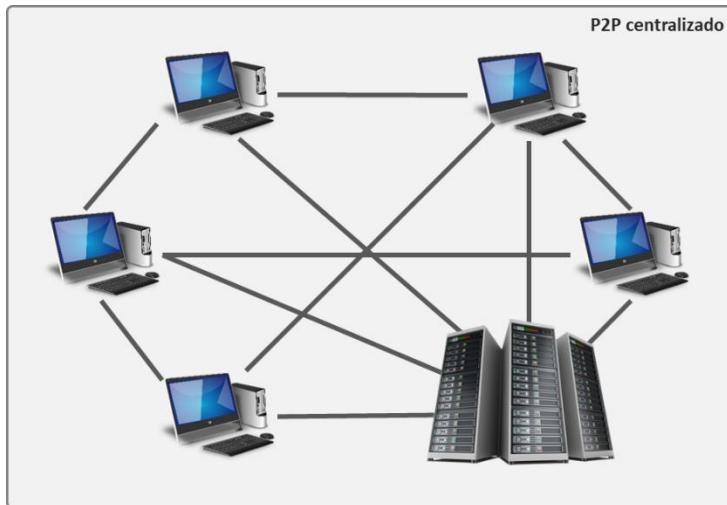


Fig 47 - Arquitectura P2P centralizada

Como podemos apreciar en la imagen anterior, todos los nodos de la red (computadoras) pueden comunicarse entre sí, sin embargo, siempre hay un servidor central que servirá de vínculo para todos los demás nodos.

Tener un servidor tiene algunas ventajas, como poder tener un índice completo de todos los usuarios conectados y los recursos que puede compartir cada nodo, haciendo las búsquedas más eficientes para los usuarios, de la misma forma, el usuario final no se tiene que preocupar por saber dónde está el nodo que tiene el archivo y configurar manualmente la conexión con este otro nodo.

A pesar de las ventajas que ofrece esta arquitectura, también tiene algunas desventajas importantes a tener en cuenta. Las desventajas de esta arquitectura se centran precisamente en el servidor central, el cual implica tener un solo punto de fallo, por lo que, si este servidor falla, toda la red queda inservible, lo que da pie a potenciales problemas de escalabilidad, pues entre más usuarios se conectan a la red, más carga le damos al servidor central.

Detener este tipo de arquitecturas es muy simple, pues solo basta con inhabilitar a quien administra el servidor central, lo cual fue lo que le paso a Napster, cuando el gobierno de EU le exigió detener sus operaciones por los cargos de complicidad en la piratería de música, algo imposible de lograr en redes P2P puras.

La privacidad de los usuarios es otro de los inconvenientes en esta arquitectura, pues el servidor central podría almacenar y analizar la actividad de todos los usuarios en la red.

Un aspecto importante a tener en cuenta con arquitecturas P2P centralizadas es que hay que tener mucho cuidado de no almacenar información con derechos de autor en nuestro servidor central, porque estaríamos violando los derechos, por lo que debemos delegar la responsabilidad a los nodos la tarea de compartir los archivos.

Ejemplos de aplicaciones que utilizan esta arquitectura son: Napster, eDonkey y BitTorrent, Skype

Características de una arquitectura P2P

En esta sección analizaremos las características que distinguen al estilo P2P 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:

- La aplicación funciona como una red, donde cada nodo puede funcionar como un cliente y servidor al mismo tiempo.
- Pueden o no tener un servidor central para ayudar en la búsqueda de los recursos.
- El colapso de cualquiera de los nodos de la red no compromete la red.
- La llegada de nuevos nodos incrementa el poder de procesamiento de la red.
- Crea una red descentralizada, donde todos los nodos de la red tienen la misma responsabilidad y son tratados como iguales.

Ventajas y desventajas

Ventajas

- **Alta escalabilidad:** Las redes P2P permiten escalar fácilmente, pues a mayor número de nodos, más recursos hay disponibles y la carga de trabajo se divide entre todos los participantes.
- **Tolerancia a fallas:** Tener los recursos distribuidos por varios nodos permite que los recursos estén disponibles sin importar si algunos de los nodos se caen.
- **Descentralización:** Los sistemas P2P puros tienen una autonomía completa, por lo que pueden funcionar sin un servidor central que organice toda la red.
- **Privacidad:** Debido al sistema de búsqueda por inundación o tablas hash, es posible tener un alto nivel de privacidad, pues no es tan fácil saber que nodo fue el que realizó la petición inicial.
- **Equilibrio de carga:** En una arquitectura P2P, todos los nodos de la red tienen el mismo rol y responsabilidades, por lo que toda la carga de trabajo se equilibra entre todos los nodos de la red.

Desventajas

- **Alta complejidad:** Las arquitecturas P2P tienen una complejidad muy alta para desarrollarse, ya que tenemos que crear aplicaciones que funcionen como cliente y servidor, al mismo tiempo que tenemos que garantizar que las búsquedas de los recursos sean eficientes.
- **Control:** A menos que tengamos un servidor central para administrar los recursos que se buscan y los usuarios que se conectan a la red, es muy fácil perder el control sobre el contenido que se puede compartir en la red.
- **Seguridad:** Esta es una de las características menos implementadas en este tipo de arquitecturas, para evitar la interceptación de las comunicaciones, nodos maliciosos, contenido falso o adulterado o la propagación de virus o programas maliciosos.

- **Tráfico:** En redes no optimizadas como las no estructuradas, podemos saturar la red con una gran cantidad de tráfico para propagar las peticiones a los nodos adyacentes.

Cuando debo de utilizar un estilo P2P

Compartir archivos

Sin lugar a duda, las aplicaciones para compartir archivos como Napster, Ares, Kazaa, eDonkey, BitTorrent entre otras muchas más son las que han ganado la atención de los usuarios finales. La salida de Napster fue el inicio de una revolución y un cambio cultural entre los usuarios, pues permitía descargar música de forma gratuita, aunque lógicamente, contribuía a la piratería, sin embargo, dio pie a que los usuarios supieran que este tipo de tecnologías existían para finalmente se ampliara para pasar de compartir música a compartir cualquier tipo de archivos, desde películas, programas, documentos, etc.

A pesar de que las aplicaciones para compartir archivos son las más populares, se estima que este tipo de aplicaciones solo representa el 10% de todas las aplicaciones que implementan una arquitectura P2P.

Streaming (video en vivo)

En el mundo de la televisión en vivo se ha extendido el termino P2PTV (Peer to Peer TeleVision). P2PTV permite compartir el ancho de banda entre todos los nodos de la red, de esta forma, cada nodo que recibe la señal, ayuda a compartirla a los demás nodos.

Este modelo de P2PTV tiene grandes ventajas contra el modelo tradicional, en el que un solo servidor transmite la señal a todos los espectadores, ya que P2PTV permite tener una señal mucho más fluida y con uso de ancho de banda mucho

menor, ya que el servidor principal solo transmitirá la señal a unos cuantos nodos, los cuales se encargarán de replicarla a otros y estos otros la replicarán a otros más.

Aplicaciones como PPLive, PPStream, SopCast, Tvants son ejemplos de aplicaciones para transmitir señales de televisión en vivo. También tenemos aplicaciones que se especializan en la transmisión de Voz por IP (VoIP), como lo son, F-Talk, P2P VoIP y Skype.

Otro de las cosas que se puede hacer es la transmisión de Voz por IP (VoIP),

Procesamiento distribuido

Hemos hablado en este capítulo que una de las grandes ventajas que tiene una red P2P es que entre más nodos se conectan a la red, más podemos de procesamiento se reúne, pero que pasaría si en lugar de utilizar esta red para compartir archivos, solo prestamos el poder de procesamiento perezoso de nuestra computadora o Smartphone para ayudar a un proyecto científico, a resolver un problema complejo o simplemente buscar vida extraterrestre.

El proyecto SETI (Búsqueda de Inteligencia Extraterrestre) es un proyecto que tiene como objetivo detectar vida inteligente fuera de la tierra, utilizando radio telescopios para escuchar señales de radio de banda estrecha proveniente del espacio. Las señales son captadas por estaciones de televisión, radares, satélites, etc, sin embargo, procesar esta información es sumamente difícil y se requiere de un gran poder de procesamiento, y es allí donde entre P2P, pues mediante una red de nodos es posible distribuir el trabajo entre todos los nodos de la red, y de esta forma repartir el trabajo de procesamiento entre todas las computadoras conectadas. Los nodos conectados a esta red son personas que ofrecen voluntariamente el poder de sus dispositivos para ayudar con esta tarea titánica.

Puedes leer más de este proyecto en <https://www.seti.org/>

BOINC es otro ejemplo de esta arquitectura, donde se utiliza el poder de procesamiento de todos los nodos de la red para resolver problemas relacionados con la física, matemáticas, medicina, astrofísica, electrónica, criptografía, sismología, climatología, etc.

Puedes leer más de este proyecto en <https://boinc.berkeley.edu/>

Criptomonedas

Las criptomonedas han tenido un gran auge en estos últimos años, pues permite realizar transacciones sin necesidad de un banco como intermediario, lo que permite que una persona envíe dinero a otra de forma directa, sin embargo, una transacción es válida solo cuando toda la red que conforman el blockchain puede avalar la transacción, por este motivo, las criptomonedas basadas en blockchain utilizan el protocolo P2P para difundir la transacción entre todos los nodos de la red, de esta forma, todos los nodos de la red pueden constatar que la transacción fue efectuada.

El modelo P2P permite que no exista una sola fuente de la verdad, si no que una transacción es validada por todos los nodos de la red, lo que hace casi imposible crear transacciones falsas, pues tendrías que, literalmente, hackear a todos los nodos de la red para que todos tengan la misma información, lo que lo hace casi imposible.

En una transacción tradicional (bancaria) el banco es el que valida la veracidad de una transacción, lo que lo hace una arquitectura centralizada, donde el banco es el único con la autoridad de realizar o validar una transacción, en un esquema de blockchain, no existe una entidad centralizada, si no que todos los nodos de la red actúan como una única entidad capaz de validar o realizar una transacción.

Aplicaciones descentralizadas

Finalmente, y la más obvia, esta arquitectura es una de las más potentes para crear redes de aplicaciones descentralizadas y que no dependan de un servidor central o infraestructura específica. Además, es de las más difíciles de detener o supervisar, por lo que es muy interesante para aplicaciones anónimas donde no queremos que los gobiernos o terceros rastren nuestra actividad dentro de la red.

Conclusiones

Como hemos podido analizar, la arquitectura P2P se caracteriza por su capacidad de crecer a medida que más nodos entran a la red, de tal forma que entre más nodos se conectan, más poder de procesamiento se agrega a la red, al mismo tiempo que la hace más difícil de detener.

No es común que se nos presente la necesidad de diseñar una aplicación con esta arquitectura, sin embargo, es importante entender cómo funcionan, pues seguramente utilizamos o estamos utilizando alguna aplicación que la utilice.

Arquitectura en Capas

La arquitectura en capas es una de las más utilizadas, no solo por su simplicidad, sino porque también es utilizada por defecto cuando no estamos seguros que arquitectura debemos de utilizar para nuestra aplicación.

La arquitectura en capas consta en dividir la aplicación en capas, con la intención de que cada capa tenga un rol muy definido, como podría ser, una capa de presentación (UI), una capa de reglas de negocio (servicios) y una capa de acceso a datos (DAO), sin embargo, este estilo arquitectónico no define cuantas capas debe de tener la aplicación, sino más bien, se centra en la separación de la aplicación en capas (Aplica el principio Separación de preocupaciones (SoC)).

En la práctica, la mayoría de las veces este estilo arquitectónico es implementado en 4 capas, presentación, negocio, persistencia y base de datos, sin embargo, es habitual ver que la capa de negocio y persistencia se combinan en una solo capa, sobre todo cuando la lógica de persistencia está incrustada dentro de la capa de negocio.

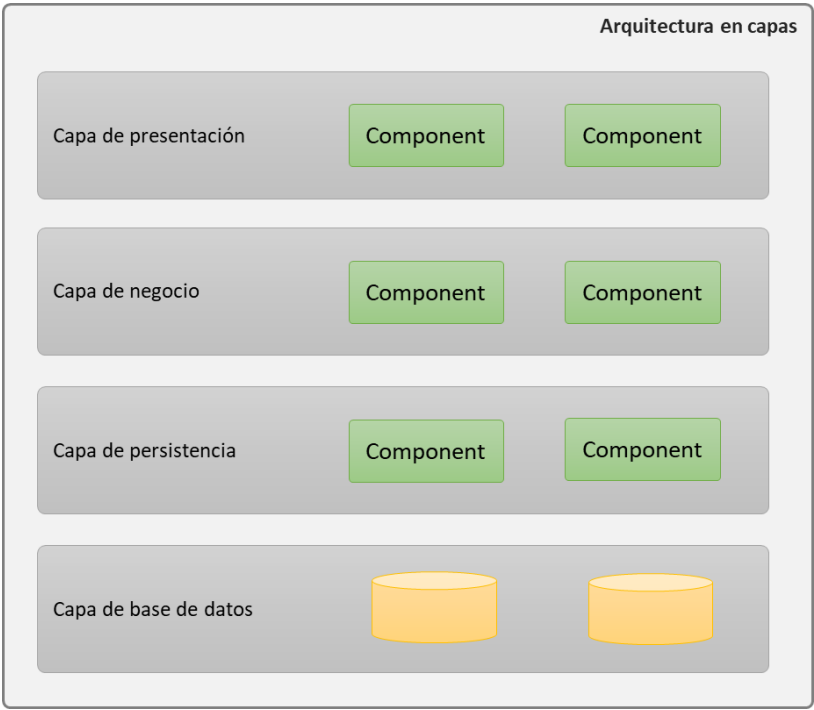


Fig 48 - Arquitectura en capas

Como se estructura una arquitectura en capas

En una arquitectura en capas, todas las capas se colocan de forma horizontal, de tal forma que cada capa solo puede comunicarse con la capa que está inmediatamente por debajo, por lo que, si una capa quiere comunicarse con otras que están mucho más abajo, tendrán que hacerlo mediante la capa que está inmediatamente por debajo. Por ejemplo, si la capa de presentación requiere consultar la base de datos, tendrá que solicitar la información a la capa de negocio, la cual, a su vez, la solicitará a la capa de persistencia, la que a su vez, la consultará a la base de datos, finalmente, la respuesta retornará en el sentido inverso hasta llegar la capa de presentación.

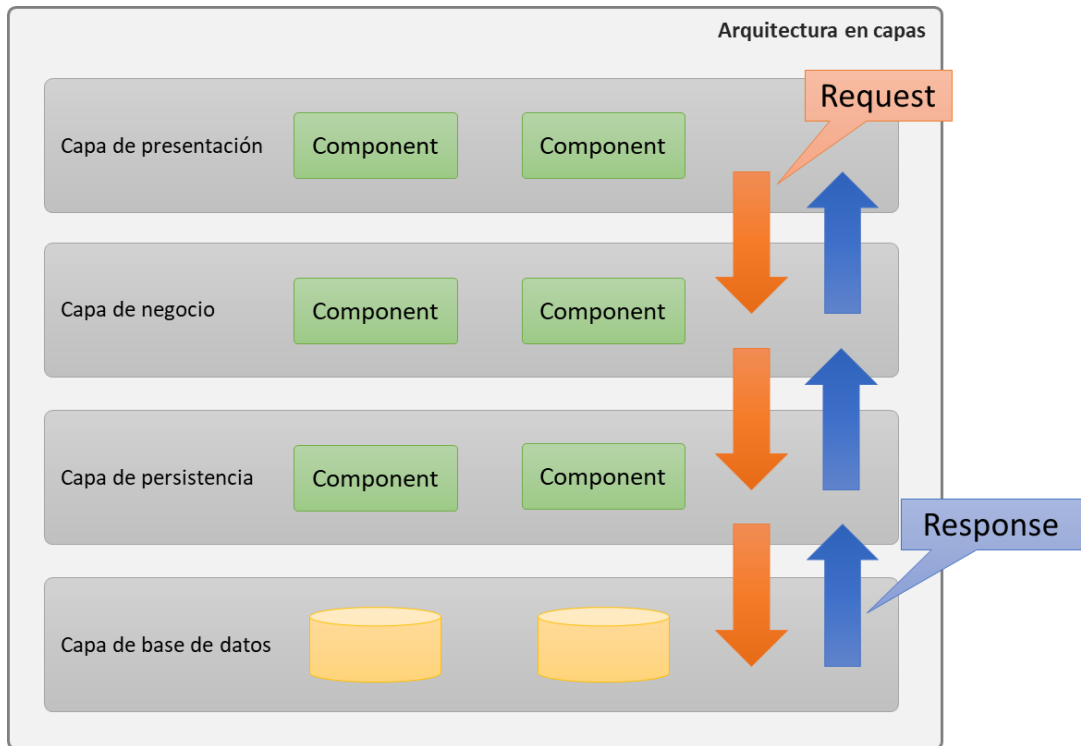


Fig 49 - Comunicación entre capas.

Un detalle a tener en cuenta en esta arquitectura, es que cada capa debe de ser un componente independiente, de tal forma que se puedan desplegar por separado, incluso, es habitual que estos componentes residan en servidores separados pero que se comunican entre sí.

La separación de la aplicación en capas busca cumplir con el principio de separación de preocupaciones, de tal forma que cada capa se encargue una tarea muy definida, por ejemplo, la capa de presentación solo se preocupa por presentar la información de forma agradable al usuario, pero no le interesa de donde viene la información ni la lógica de negocio que hay detrás, en su lugar, solo sabe que existe una capa de negocio que le proporcionará la información. Por otra parte, la capa de negocio solo se encarga de aplicar todas las reglas de negocio y validaciones, pero no le interesa como recuperar los datos, guardarlos o borrarlos, ya que para eso tiene una capa de persistencia que se encarga de eso. Por otro lado, la capa de persistencia es la encargada de comunicarse a la base de datos, crear las instrucciones SQL para consultar, insertar, actualizar o borrar registros y retornarlos en un formato independiente a la base de datos. De esta forma, cada capa se preocupa por una cosa y no le interesa como le haga la capa de abajo para servirle los datos que requiere.

Respetar el orden de las capas es muy importante, ya que brincarnos una capa para irnos sobre una más abajo suele ser un grave error, ya que bien es posible hacerlo, empezamos a crear un desorden sobre el flujo de comunicación, lo que nos puede llevar al típico anti patrón conocido como código espagueti, el cual consiste en que empezamos a realizar llamadas desde cualquier capa a otra capa, haciendo que el mantenimiento se vuelva un verdadero infierno.

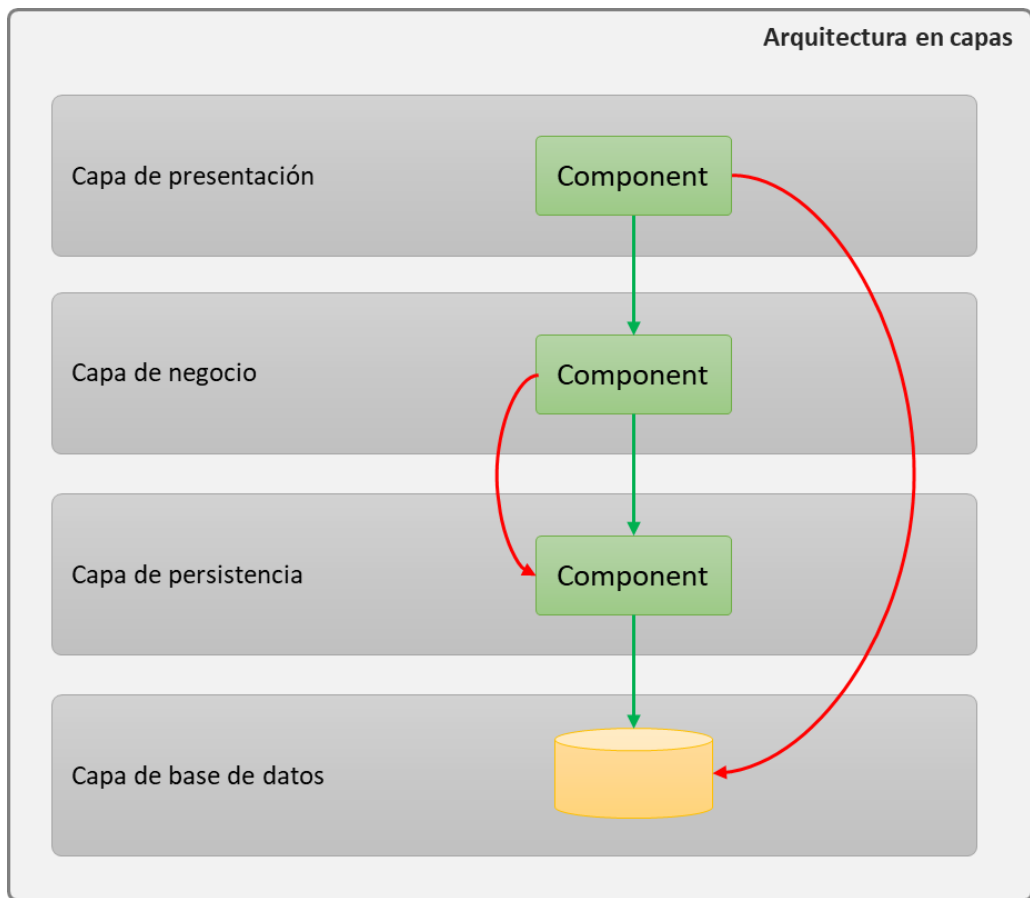


Fig 50 - Salto indebido entre capas.

Respetar el orden de las capas es importante por un término llamado **Aislamiento de capas**, el cual consiste en que cualquier cambio realizado en una capa no debería de afectar a otras capas. Permitir que una capa se comuniqué con otras no debidas, puede provocar que el cambio en una capa tenga un impacto sobre esta. Por ejemplo, si permitimos que la capa de presentación se comuniqué con la capa de persistencia y esta cambia su modelo de datos, tendríamos un impacto directo sobre la capa de presentación, pues está ya esperando una estructura, la cual acabamos de cambiar. Ahora te preguntarás, si cambia el modelo de datos, seguramente también afecta a la capa de negocio, entonces cual es el caso de usar capas, pues bien, la ventaja es que mediante las capas podemos controlar hasta donde se propaga el impacto de un cambio, es decir, si cambia el modelo de datos, solo impactamos a la capa de negocio, pero esta podrá controlar el

cambio para respetar el forma con el que la presentación recibe los datos, por lo que controlamos el impacto solo hasta la capa superior inmediata. En otro caso, si permitimos que una capa acceda a cualquier capa, los cambios se propagarían por todas las capas.

Es importante resaltar que las capas funcionan con contratos bien definidos, donde sabemos exactamente que reciben y que responde, esto hace que, si hacemos un cambio, pero no afectamos el contrato, aislamos el cambio y la capa superior ni se enterará del cambio.



Nuevo concepto: Contrato

Un contrato describe la interfaz que expone un servicio o componente, el cual consiste en las operaciones brindadas, parámetros de entrada y salida, estructura de los objetos y la localización de los servicios.

Capas abiertas y cerradas

Hace un momento dejamos muy en claro que una capa solo se puede comunicar con la capa que está inmediatamente por debajo, sin embargo, hay algunas excepciones que en esta sección abordaremos.

Si bien una arquitectura en capas se construye para satisfacer las necesidades de una aplicación, estas en ocasiones hacen uso de servicios genéricos o de utilidad, que por lo general son compartidos por varias aplicaciones, estos servicios podrían representar una nueva capa de servicios, lo que nos trae un problema, la capa de presentación debe de pasar por la capa de servicios o la capa de negocio, en este punto creo que todos coincidimos en que debería de pasar por la capa de negocio, lo que automáticamente nos indica que entonces la capa de negocio debería de llamar a la capa de servicios para consultar la capa de persistencia:

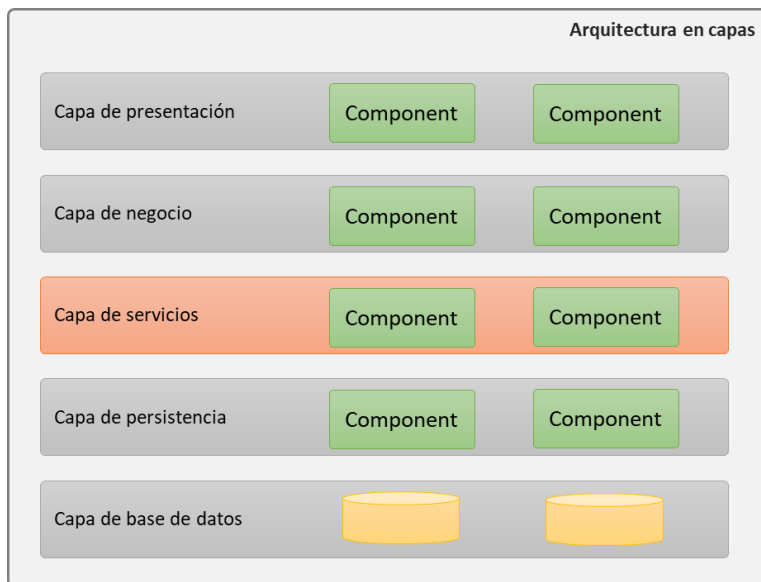


Fig 51 - Capas genéricas o compartidas.

Si prestas un poco de atención, verás que esta arquitectura es totalmente ilógica, pues la capa de servicios por ningún motivo tomará en cuenta la capa de nuestra aplicación, ya que fueron diseñados para usos generales y no para cumplir con las reglas de nuestro proyecto. Esta nueva capa ha venido a complicar todo ¿cierto? Por suerte tenemos el concepto de capas abiertas y cerradas.

Las capas cerradas es lo que hemos estado analizando todo este tiempo, es decir, son capas que no podemos brincar por ningún motivo, mientras que las capas abiertas son aquellas que está justificado brincarlas.

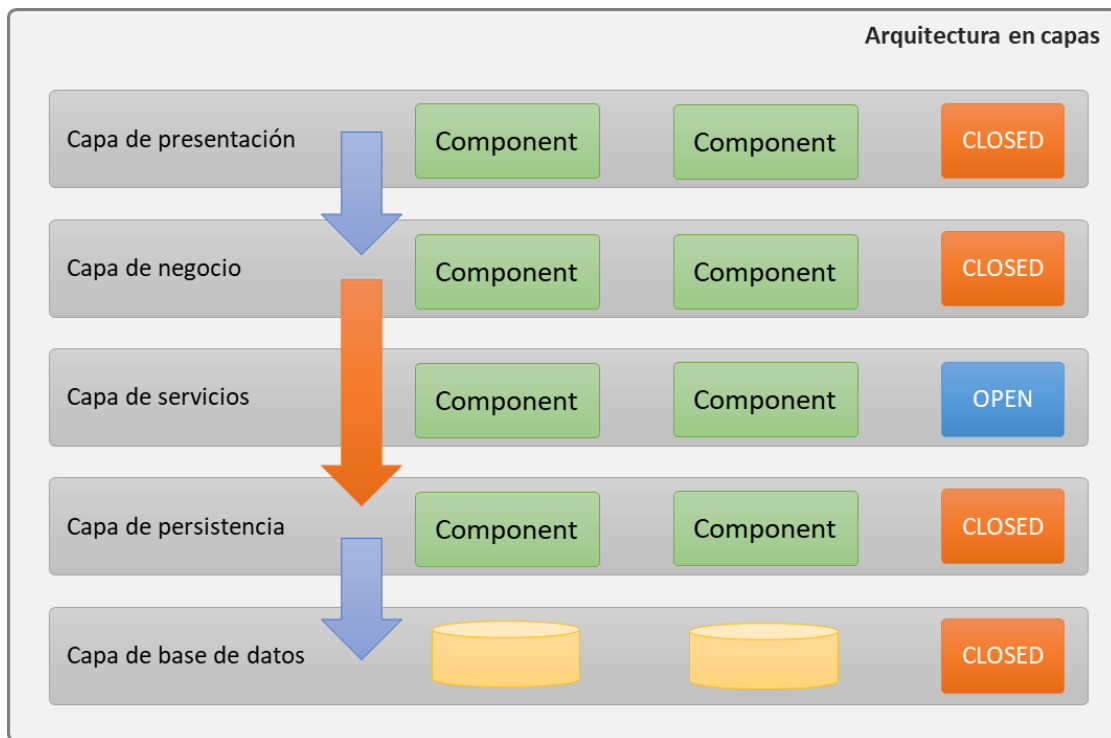


Fig 52 - Capas abiertas y capas cerradas

En este nuevo diagrama hemos dejado en claro que la capa de servicios es una capa opcional, la cual podemos saltarnos para pasar de la capa de negocio a la capa de persistencia sin necesidad de pasar por la capa de servicios. Ahora bien, el hecho de que tengamos este nuevo termino de capas abiertas y cerradas significa que tenemos luz verde para saltarnos todas las capas con la justificación de que es una capa abierta, ya que todas las capas que sean clasificadas como abiertas deberán estar perfectamente justificadas, de lo contrario entraríamos incumpliendo el aislamiento de las cajas.

Arquitectura en 3 capas

Acabamos de mencionar que la arquitectura en capas solo propone la construcción de capas, pero no define cuales deben de ser el número de capas o lo que tiene que hacer cada una, lo que nos deja un sabor de boca medio extraño, pues

sabemos que es bueno crear capas, pero al mismo tiempo, no sabemos con certeza cuantas capas crear y como dividir de responsabilidad de la aplicación entre estas capas.

Pues bien, la arquitectura de 3 capas viene a aterrizar mejor el concepto del estilo arquitectónico en capas, definiendo cuales son las capas y que responsabilidad deberá tener cada una. Cabe mencionar que la arquitectura de 3 capas es solo una definición más concreta del patrón arquitectónico en capas.

Uno de los principales problemas al construir aplicaciones, es que vamos construyendo todo el software como una enorme masa sin forma, donde un solo componente se encarga de la presentación, la lógica de negocio y el acceso a datos, lo que hace que las aplicaciones sean sumamente difíciles de mantener y escalar, al mismo tiempo es complicado para los desarrolladores dividir el trabajo, ya que muchas clases están relacionadas con otras y modificar una clase puede tener un gran impacto sobre muchas más.

Esta problemática era muy común sobre todos con tecnologías web antiguas, como PHP, ASP o JSP, donde era común encontrar la lógica para crea la vista, las reglas de negocio e incluso la conexión a la base de datos sobre el mismo archivo, por ejemplo:

```
1. <html>
2.   <body>
3.     <sql:setDataSource var="connection" driver="com.mysql.jdbc.Driver"
4.       url="jdbc:mysql://localhost/crm"
5.       user="root" password="1234"/>
6.
7.     <sql:query dataSource="${connection}" var="results">
8.       select * from products;
9.     </sql:query>
10.
11.    <table>
12.      <tr>
13.        <th>Id</th>
14.        <th>Name</th>
15.        <th>Price</th>
16.      </tr>
17.
18.      <c:forEach var="row" items="${results.rows}">
19.        <tr>
20.          <td><c:out value="${row.id}"/></td>
21.          <td><c:out value="${row.name}"/></td>
22.          <td><c:out value="${row.price}"/></td>
23.        </tr>
```

```
24.         </c:forEach>
25.     </table>
26. </body>
27. </html>
```

Esto que estamos viendo, es código en JSP (Java Server Page), el cual permitía definir como se vería la página e incrustas fragmentos de código Java o TagLibs (librerías de tags), en las cuales podíamos hacer casi cualquier cosa.

Cuando tecnologías como JSP, PHP, ASP llegaron al mercado, era muy común ver este tipo de páginas, las cuales eran sumamente difícil de mantener y prácticamente no se podía reutilizar nada de código, por lo general, lo que muchos desarrolladores hacían era copiar el código de una página y pegarlo en otra, lo cual era sumamente ineficiente, y al detectarse un bug en ese código se tenía que arreglar todas las páginas donde se había copiado el código (Infringían el principio Don't repeat yourself (DRY)) . No estoy diciendo que JSP, PHP o ASP fueran malos lenguajes, o que no hubiera forma de solucionar esto, solo quiere contarte lo que hacían muchos de los programadores inexpertos.

Pero a pesar de lo que muchos puedan creer, este problema no es exclusivo de aplicaciones web, también se presenta en aplicaciones de escritorio o móviles, ya que al final, todas estas cuentan con presentación, lógica de negocios y acceso a datos, es por ello que la arquitectura de 3 capas se presenta como una solución general para resolver este tipo de problemas.

Como hemos analizando, separar la aplicación en unidades de código cohesivas es necesario para una mejor organización del proyecto, pero las ventajas que ofrecen no son solo las de organizar código, si no que ayuda mucho a la reutilización del mismo.

La arquitectura en 3 capas está basada en el estilo de arquitectura en capas, donde cada capa se encarga de una parte del programa, y el orden de ejecución siempre va de la capa de más arriba a la de más abajo, sin embargo, la arquitectura de 3 capas define exactamente las capas que debe de tener la aplicación y la responsabilidad de cada una.

En resumen, la arquitectura define que una aplicación debe de estar diseñada en 3 capas, las cuales son:

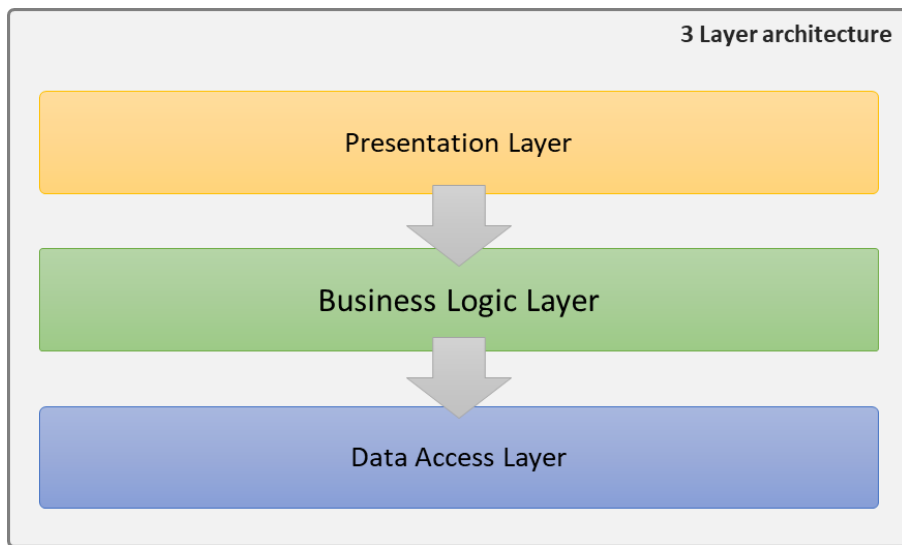


Fig 53 - Arquitectura de 3 capas.

1. **Capa de presentación:** Es la encargada de crear la vista o interface gráfica de usuario, puede ser una aplicación web, una app de escritorio o una app móvil. Generalmente escrito en HTML, Javascript, CSS, Android/Swift, etc.
2. **Capa de lógica de negocio:** Esta capa contiene la lógica de negocio y las operaciones de alto nivel que la capa de presentación puede utilizar. Generalmente escrito en Java, Python, .NET, NodeJS, etc.
3. **Capa de acceso a datos:** Esta capa corresponde a la capa donde están los datos, por ejemplo, MySQL, Oracle, PostgreSQL, MongoDB, etc.

Como ya lo habíamos comentado, en una arquitectura en capas, es necesario que la ejecución inicie de las capas superiores a las inferiores, de tal forma que el usuario siempre tendrá que ejecutar la aplicación por la capa de presentación, la cual deberá hacer llamadas a la capa de negocio para recuperar o actualizar los datos, la cual, a su vez, tendrá que comunicarse a la capa de datos.

Como regla general, en una arquitectura de 3 capas, no se utiliza el concepto de capa abierta, ya que no tiene sentido que la vista se comunique directamente con la capa de datos.

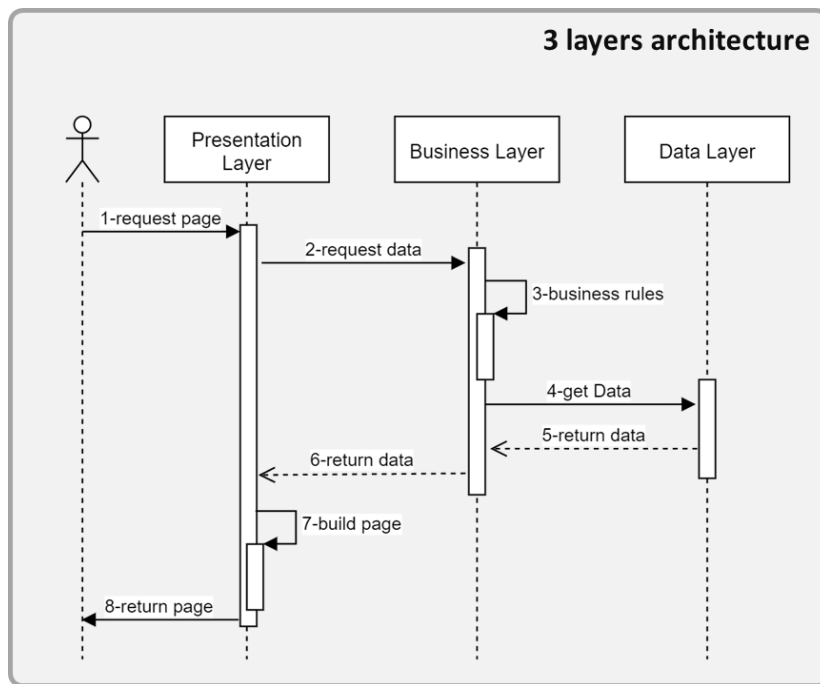


Fig 54 - Diagrama de secuencia de la ejecución en una arquitectura de 3 capas.

El diagrama anterior ilustra mejor como se da la ejecución en una arquitectura de 3 capas, el cual simula consulta de una nueva página

1. Un usuario solicita a la aplicación una nueva página
2. La capa de presentación determina cual es la página que solicita el usuario y los datos que requiere para renderizar la página, por lo que solicita a la capa de negocios los datos.
3. La capa de negocio realiza algunas reglas de negocio, como validar el usuario, o filtrar la información según el usuario.
4. La capa de negocio realiza la consulta a la capa de datos.
5. La base de datos retorna los datos solicitados por la capa de negocio.
6. La capa de negocio convierte los datos a un formato amigable y la regresa a la capa de presentación.
7. La capa de presentación recibe los datos y construye la vista a partir de estos.
8. La capa de presentación retorna la nueva página al cliente.

Características de una arquitectura en capas

En esta sección analizaremos las características que distinguen a una arquitectura en capas 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:

- La aplicación se compone de capas, en la cual cada capa tiene una sola responsabilidad dentro de la aplicación.
- Las capas de la aplicación son totalmente independientes de las demás, con la excepción de la capa que está inmediatamente debajo.
- Deben de existir al menos 3 capas para considerar una aplicación por capas.
- Toda la comunicación se hace siempre de forma descendente, pasando desde las capas superiores a las más inferiores.

Ventajas y desventajas

Ventajas

- **Separación de responsabilidades:** Permite la separación de preocupaciones (SoC), ya que cada capa tiene una sola responsabilidad.
- **Fácil de desarrollar:** Este estilo arquitectónico es especialmente fácil de implementar, además de que es muy conocido y una gran mayoría de las aplicaciones la utilizan.
- **Fácil de probar:** Debido a que la aplicación construida por capas, es posible ir probando de forma individual cada capa, lo que permite probar por separada cada capa.

- **Fácil de mantener:** Debido a que cada capa hace una tarea muy específica, es fácil detectar el origen de un bug para corregirlo, o simplemente se puede identificar donde se debe aplicar un cambio.
- **Seguridad:** La separación de capas permite el aislamiento de los servidores en subredes diferentes, lo que hace más difícil realizar ataques.

Desventajas

- **Performance:** La comunicación por la red o internet es una de las tareas más tardadas de un sistema, incluso, en muchas ocasiones más tardado que el mismo procesamiento de los datos, por lo que el hecho de tener que comunicarnos de capa en capa genera un degrado significativo en el performance.
- **Escalabilidad:** Las aplicaciones que implementan este patrón por lo general tienden a ser monolíticas, lo que hace que sean difíciles de escalar, aunque es posible replicar la aplicación completa en varios nodos, lo que provoca un escalado monolítico.
- **Complejidad de despliegue:** En este tipo de arquitecturas es necesario desplegar los componentes de abajo arriba, lo que crea una dependencia en el despliegue, además, si la aplicación tiende a lo monolítico, un pequeño cambio puede requerir el despliegue completo de la aplicación.
- **Anclado a un Stack tecnológico:** Si bien no es la regla, la realidad es que por lo general todas las capas de la aplicación son construidas con la misma tecnología, lo que hace amarrarnos la comunicación con tecnologías propietarias de la plataforma utilizada.
- **Tolerancia a los fallos:** Si una capa falla, todas las capas superiores comienzan a fallar en cascada.

Cuando debo de utilizar una arquitectura en capas

La arquitectura por capas es una de las arquitecturas más versátiles, incluso es considerada por mucho como el estilo arquitectónico por defecto, pues es tan versátil que se puede aplicar a muchos de las aplicaciones que construimos día a

día. Es común ver que los arquitectos utilizan esta arquitectura cuando no están seguros de cual utilizar, lo que la hace una de las arquitecturas más comunes.

Entorno empresarial

Sin lugar a duda, la arquitectura en capas es muy utilizada en aplicaciones empresariales, donde desarrollamos los típicos sistemas de ventas, inventario, pedidos o incluso sitios web empresariales que tienden a ser aplicaciones monolíticas, pues permite crear aplicaciones con cierto orden que se adapta muy bien a este tipo de empresas muy ordenadas, donde todo tiene que tener un orden muy definido y justificado.

En este tipo de entorno hay poca innovación y mucha burocracia, lo que impide proponer arquitecturas mucho más elaboradas que se salga de los esquemas típicos, además, no requiere de mucho mantenimiento o perfiles muy especializados, lo que favorece la proliferación de este tipo de arquitecturas.

Por otra parte, las aplicaciones empresariales son por lo general de uso interno, lo que implica que no tiene una proyección de crecimiento exponencial, lo que ayuda a compensar el hecho de que este estilo arquitectónico tiene un escalamiento limitado.

Aplicaciones web

Las aplicaciones por lo general siguen el patrón arquitectónico conocido como MVC (Modelo-Vista-Controlador), el cual consiste en separar la aplicación en tres partes, la vista (páginas web), los datos (Modelo) y el controlador (lógica de negocio) en partes separadas, lo cual se adapta muy bien con la arquitectura en capas.

Es común ver que las aplicaciones de hoy en día se desarrollan en capas, donde por una parte se construye la interface gráfica con tecnologías 100% del FrontEnd como Angular, React o Vue, la cual corresponde a la capa de la vista, luego, llaman a un API REST el cual contiene la capa de negocio y persistencia y finalmente tenemos la base de datos, haciendo que nuestra aplicación tenga 3 o 4 capas, según como se construya la aplicación

Arquitectura de uso general

Como mencionamos al inicio, esta arquitectura es una buena opción cuando no estamos muy seguros que de arquitectura utilizar, incluso, es una buena arquitectura para arquitectos principiantes, pues tiene una estructura muy clara y que no tiene muchos problemas de implementación. Sin embargo, es importante analizar las desventajas y características de esta arquitectura para no implementarla en una aplicación para la cual no dará el ancho, al final, es nuestra responsabilidad como arquitectos que el sistema funcione y garantizar su funcionamiento durante un tiempo considerable.

Conclusiones

Como hemos podido observar, el estilo arquitectónico es uno de los más fácil de implementar, lo que lo hace unos de los patrones más versátiles y más ampliamente utilizados, lo que lo convierte en uno de los estilos arquitectónicos de referencia para muchas aplicaciones. Además, es un estilo que no representa mucha carga de mantenimiento para las empresas, lo que hace que pueda funcionar durante mucho tiempo de forma interrumpida.

Por otra parte, hemos visto que este estilo tiene algunas desventajas que son importantes a tener en cuenta, sobre todo el performance y la escalabilidad, por lo que no es muy recomendable para aplicaciones de alto nivel de procesamiento de datos o que tiene una proyección de alto crecimiento a corto plazo.

Si eres un arquitecto novato y quieres comenzar a diseñar arquitecturas de software, esta puede ser un buen inicio, pues es muy clara, fácil de diseñar y explicar en un comité de arquitectura.

Microkernel

El estilo arquitectónico de Microkernel o también conocido como arquitectura de Plug-in, permite crear aplicaciones extensibles, mediante la cual es posible agregar nueva funcionalidad mediante la adición de pequeños plugins que extienden la funcionalidad inicial del sistema.

En una arquitectura de Microkernel las aplicaciones se dividen en dos tipos de componentes, en sistema Core (o sistema central) y los plugins (o módulos), el sistema Core contiene los elementos mínimos para hacer que la aplicación funcione y cumpla el propósito para el cual fue diseñada, por otra parte, los módulos o plugins con componentes periféricos que se añaden o instalan al componente Core para extender su funcionalidad. En este sentido, solo puede haber un componente Core y muchos Plugins.

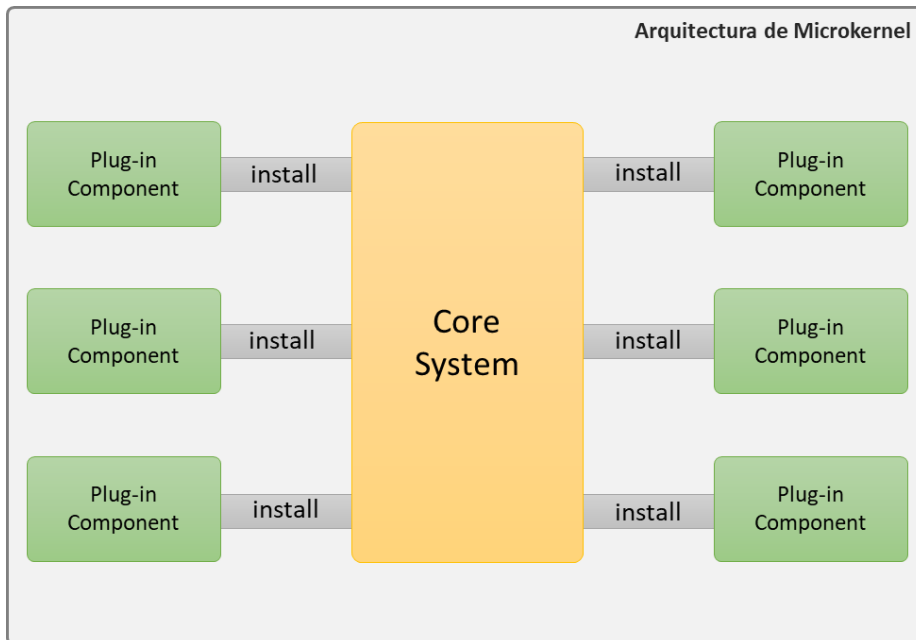


Fig 55 - Estructura de la arquitectura de Microkernel.

La idea central de este estilo arquitectónico es permitir la extensión de su funcionalidad o personalización, pero respetando el principio Open-Closed, es

decir, está abierto para extender la funcionalidad, pero cerrado para modificar su funcionalidad principal. De esta forma, se logra que los desarrolladores pueden crear plugins para agregar nueva funcionalidad o extender la existente, pero sin alterar la funcionalidad Core del sistema.

Los ejemplos más claros de esta arquitectura son los IDE's de desarrollo como Eclipse, Netbeans, Visual Studio, Visual Studio Code o los sistemas de ofimática como Word, Powerpoint, Excel, etc. Todas estas aplicaciones permiten que los desarrolladores creen nueva funcionalidad, la cual se instala para extender o agregar nuevas características. En el caso de los IDE's de desarrollo, tenemos plugins para permitir el uso de servidores de aplicaciones, soportar nuevos lenguajes de programación, para editar archivos de determinada extensión, para conectarnos a la base de datos, utilizar terminales, para conectarnos con sistemas de control de versiones, consumir servicios, etc. En el caso de los programas de ofimática tenemos la opción de conectarnos con servicios como Sharepoint, almacenamiento en la nube con Google Drive, DropBox, One Drive, integración con Slack, plugins para insertar firmas electrónicas, traducción, etc.

Como se estructura una arquitectura de Microkernel

Los sistemas que utilizan una arquitectura de Microkernel no son fáciles de desarrollar, pues necesitamos crear aplicaciones que son capaces de agrandar dinámicamente su funcionalidad a medida que nuevos plugins son instalados, al mismo tiempo que debemos de tener mucho cuidado de que los plugins no modifiquen o alteren la esencia de la aplicación.

El solo hecho de lograr que un sistema acepte plugins ya es complicado, sobre todo porque el sistema Core y los plugins son desarrollados por lo general por equipos separados, por lo que el sistema Core debe de dejar muy en claro como los plugins deben de desarrollados y deben de tener un archivo descriptor que le diga al sistema Core como debe de instalarse o debería de mostrar el plugin ante el usuario.

Para permitir la construcción de plugins, el sistema Core debe de proporcionar un API o una definición la cual el plugin debe de implementar, de esta forma tenemos el sistema Core y el API que provee para que los desarrolladores creen los plugins.

El API es una serie de clases o interfaces que deben ser implementadas por el Plug-in, las cuales serán analizadas por el sistema Core al momento de la instalación.

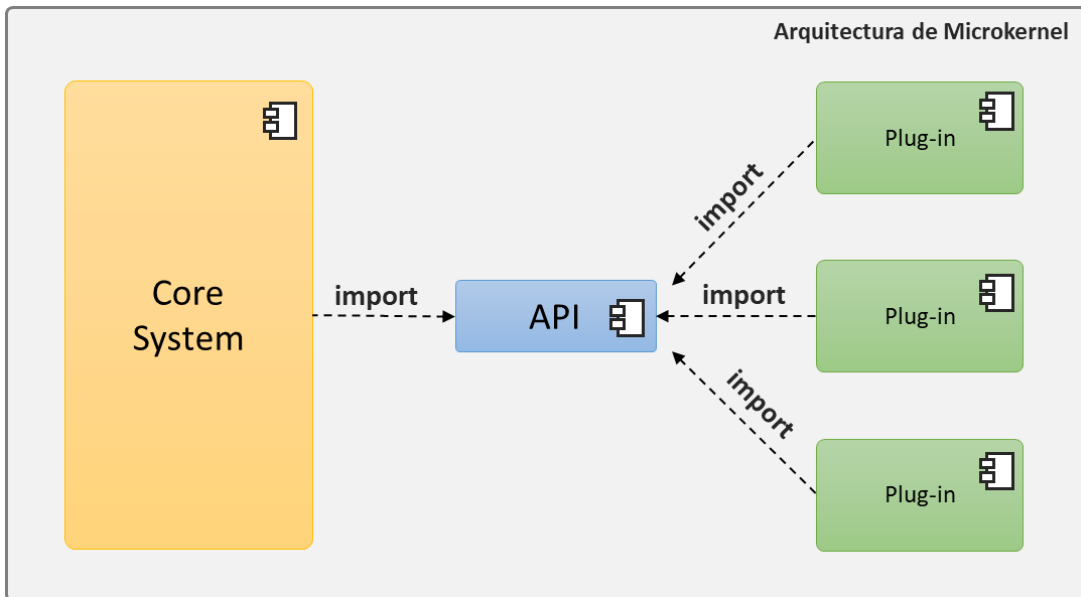


Fig 56 - API del Microkernel.

En la imagen anterior podemos ver el rol que tiene el API en la arquitectura. El API en realidad es un componente independiente del sistema Core, el cual contiene solamente las clases e interfaces que deberán conocer tanto el sistema Core como los Plugins.

Por lo general hay una interface o clase abstracta llamada Plugin la cual debe de implementar una clase del componente Plugin, la cual servirá como punto de entrada para el sistema Core, la cual le dará toda la información del Plugin al sistema Core.

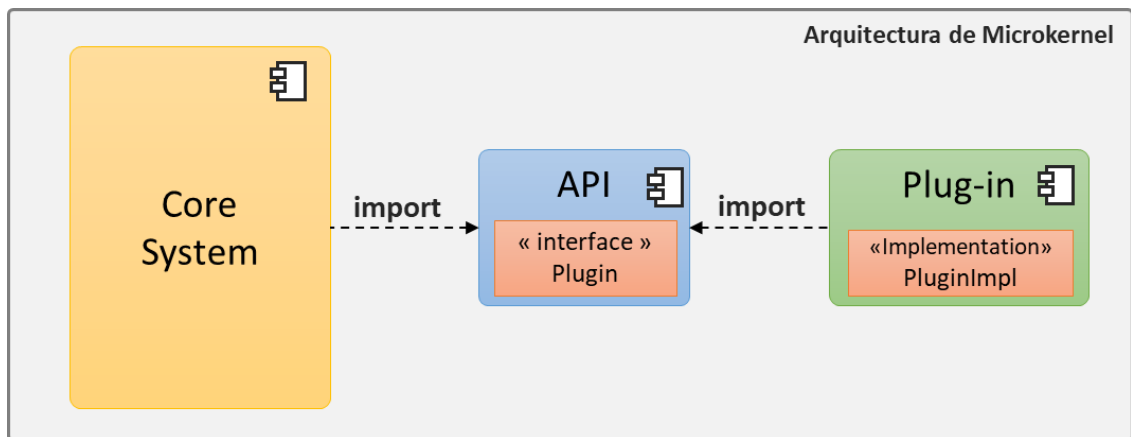


Fig 57 - Implementando un plug-in.

El siguiente paso consiste en que el sistema Core sepa cómo analizar el Plug-in para su instalación, para ello existen dos escenarios:

Instalación por descriptores

El primero y más simple es mediante un archivo descriptor, el cual contiene una estructura previamente definida, en la cual se encuentre listado el nombre de la clase que implementa la interface Plugin, el nombre, la versión y toda la Metadata necesaria para agregar las nuevas pantallas, opciones del menú, etc. La idea es que cuando instalemos un Plug-in, el sistema Core lea primero este archivo y posteriormente instale el Plug-in con toda la Metadata obtenida de este archivo.

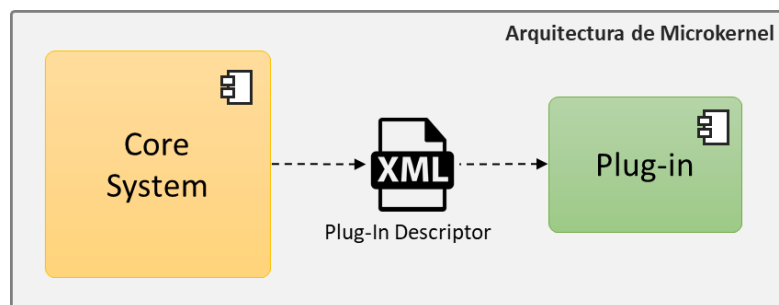


Fig 58 - Archivo descriptor del plug-in

El uso de archivos descriptores es la forma más simple de implementar del lado del sistema Core, pues deja del lado del desarrollador del plug-in crear correctamente el archivo, lo cual puede ser un poco tedioso para el desarrollador del plug-in, debido a que al final, es un archivo de texto propenso a errores de datos o formato.

Instalación por metadatos

La segunda forma es utilizar el principio de Inversión de control (Inversion of control - IoC), la cual consiste que en lugar de utilizar archivos para describir el Plug-in, podemos utilizar metadatos en las clases, de tal forma que el sistema Core se encargará de introspectar el código del Plug-In para descubrir todos los metadatos necesarios para instalar y activar el Plug-In.



Nuevo concepto: Introspección

la introspección es la capacidad de algunos lenguajes orientados a objetos para para inspeccionar los metadatos de los objetos, como atributos, propiedades, tipos de datos, nombre de propiedades y método en tiempo de ejecución.

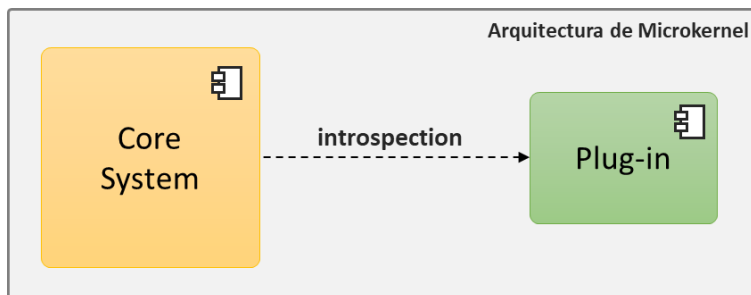


Fig 59 - Introspectando un Plug-In

La introspección es un mecanismo más complicado de implementar del lado del sistema Core, pues tenemos que leer y analizar dinámicamente todas las clases de un Plug-In para obtener todos los metadatos necesarios para la instalación, sin embargo, es mucho más fácil para el desarrollador del Plug-In, pues a medida que

programa sus clases les va agregando los metadatos necesarios y se olvida de crear un archivo descriptor.

Repositorio de Plugins

Si bien es posible que el usuario descargue el Plug-In lo instale manualmente, por lo general, las aplicaciones más grandes manejan un repositorio de Plugins, el cual es básicamente una plataforma en la cual los desarrolladores registren sus Plugin para que puedan ser descargados e instalados desde el sistema Core. Si bien, esta característica es deseable, no es necesaria para implementar una arquitectura de Microkernel.

Características de una arquitectura de Microkernel

En esta sección analizaremos las características que distinguen al estilo de Microkernel 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:

- La aplicación se compone de un sistema Core que tiene la mínima funcionalidad y un conjunto de plugins que complementan o extienden la funcionalidad base.
- Los plugins agregan o extienden la funcionalidad del sistema Core, pero por ningún motivo puede modificar la forma de trabajar del sistema Core.
- Desarrolladores externos al desarrollo de la aplicación pueden desarrollar Plugins
- Por lo general es posible ver un registro de plugins disponibles.

- Los plugins no pueden funcionar de forma independiente, si no que requieren ser ejecutados por medio del sistema Core.

Ventajas y desventajas

Ventajas

- **Testabilidad:** Debido a que los Plugins y el sistema Core son desarrollados de forma por separado, es posible probarlos de forma aislada.
- **Performance:** En cierta forma, muchas de las aplicaciones basadas en Microkernel trabajan de forma Monolítica una vez que el Plug-in es instalado, lo que hace que todo el procesamiento se haga en una sola unidad de software.
- **Despliegue:** Debido a la naturaleza de Plugins es posible instalar fácilmente todas las características adicionales que sea necesarias, incluso, pueden ser agregar en tiempo de ejecución, lo que en muchos casos ni siquiera requiere de un reinicio del sistema.
- **Dinamismo:** Las aplicaciones basadas en Plugins pueden habilitar o deshabilitar características basadas en perfiles, lo que ayuda que solo los plugins necesarios sean activados, incluso, pueden ser activados solo cuando son utilizados por primera vez (hot deploy), lo que hace que los módulos que nunca se utilizan, no se activen nunca, ahorrando una gran cantidad de recursos.
- **Construcción modular:** El sistema de Plugins permite que diferentes equipos puedan trabajar en paralelo para desarrollar los diferentes Plugins.
- **Reutilización:** Debido a que los Plugins puede ser instalados en cualquier instancia del sistema Core, es posible reutilizar los módulos en varias instancias, incluso, es posible comercializarlas de forma independiente. Solo como ejemplo, existe empresas que se dedican exclusivamente a desarrollar Plugins para venderlos, como es el caso de los Plugins de Wordpress.

Desventajas

- **Escalabilidad:** Las aplicaciones basadas en Microkernel son generalmente desarrolladas para ser ejecutadas en modo Standalone. Si bien existen aplicaciones que implementan el estilo de Microkernel que son altamente escalables como algunos servidores de aplicaciones, la realidad es que este

no es un estilo arquitectónico que se distinga por crear aplicaciones altamente escalables.

- **Alta complejidad:** Las aplicaciones basadas en Microkernel son difíciles de desarrollar, no solo por la habilidad técnica para soportar la agregación de funcionalidad adicional por medio de Plugins, si no que requiere un análisis muy elaborado para identificar hasta qué punto puede ser extendida la aplicación sin afectar la esencia del sistema Core.



Nuevo concepto: Standalone

Término del inglés utilizado para llamar a las aplicaciones que se ejecutan de forma independiente o autónomas.

Cuando debo de utilizar una arquitectura de Microkernel

Si bien existen muchas aplicaciones basadas en Microkernel, la realidad es que es poco probable que te encuentres en la necesidad de implementar un Microkernel, pues son utilizado por lo general para productos que se intentan comercializar a gran escala, dando la posibilidad a desarrolladores de todo el mundo crear su propios plugins.

La arquitectura de Microkernel se suele utilizar para construir aplicaciones que den la posibilidad a terceros de extender la funcionalidad de la aplicación mediante módulos que se puedan activas o desactivar, si la necesidad de tener que hacer un desarrollo sobre el sistema base.

Plataformas

Las plataformas son software diseñado para servir como base para el desarrollo de otras aplicaciones, algunos se especializan en ciertas áreas, pero hay otros de propósito general como el Netbeans Platform (<https://netbeans.org/kb/trails/platform.html>). Como programadores nos hemos dado cuenta que todos los IDE's soportan plugins, y es que esto se debe a que la

tecnología avanza tan rápido que nuevas cosas van saliendo, por lo que los IDE's deben poder soportar agregar soporte a estas nuevas tecnologías a medida que van saliendo al mercado, es por ellos que los plugins son una excelente idea, ya que, por lo general, es la misma compañía que desarrolla la nueva tecnología, la que termina desarrollando los plugins.

Productos

Los productos son sistemas que están empaquetados para ser distribuidos a gran escala, como es el caso de la suite de Office, CMS como Wordpress, Drupal, Joomla, Herramientas de desarrollo como Jenkins, etc. Todos estos productos son distribuidos a gran escala, lo que permite que desarrolladores de todo el mundo creen nuevos plugins y mejoren las características del producto.

Conclusiones

Como hemos podido analizar a lo largo de esta sección, el estilo arquitectónico de Microkernel permite extender la funcionalidad de sistema mediante la adición de Plugins, dichos plugins pueden ser desarrollados por terceros, lo que amplía las posibilidades de la aplicación Core si necesidad de costear los desarrollos, al mismo tiempo que permite que puedan personalizar al máximo el sistema.

Por otra parte, podemos observar que un sistema basado en Microkernel permite tener múltiples equipos de desarrollo que construyen los módulos en paralelo sin interferir unos con otros, al mismo tiempo que permite que estos componentes sean probados de forma independientes, por lo que podemos decir que este estilo es fácil de probar.

Sin embargo, hemos visto que uno de las principales problemáticas de este estilo arquitectónico es su alta complejidad de desarrollar y su escalabilidad, lo cual puede ser una limitante para sistemas que deben ser diseñados para un alto escalamiento.

Service-Oriented Architecture (SOA)

Hace cerca de 20 años cuando todavía no existían tantos productos de software comerciales para gestionar las diferentes áreas de una compañía, las empresas por lo general construían sus propios sistemas adecuados para satisfacer sus necesidades específicas, por lo que casi toda la información residía en un solo punto, lo que hacía que las necesidades de integrar información entre diferentes sistemas eran casi nulas.

Sin embargo, con el paso de los años, empezaron a salir sistemas especializados que ayudaban a gestionar diferentes áreas de la compañía de una forma más eficiente y con un costo mucho menor que desarrollarlos por ellos mismo, como es el caso de los sistemas de nómina, inventarios, distribución, contabilidad, almacenes, ERP, CRM, CMS, etc. Aunado a esto, todos estos productos eran desarrollados por empresa y tecnologías diferentes, lo que hacía que enviar o consultar información de un sistema a otro era un verdadero dolor de cabeza.

Una de las problemáticas más importantes para la transferencia de información de un sistema a otra era la tecnología por sí misma, ya que una aplicación desarrollada en C no se podía comunicar naturalmente con una en Java, o una aplicación de Visual Basic, esto provocaba que los desarrolladores buscaran alternativas para la transmisión de información, la cual consistía en intercambiar archivos de texto plano o compartir una base de datos desde la cual leyeran y escribieran información, que como veremos más adelante, era otro problema.

Para solventar esta problemática salieron tecnologías como DCOM (Distributed Component Object Model) y CORBA (Common Object Request Broker Architecture) los cuales buscaban resolver los problemas de comunicación entre las diferentes tecnologías, y lo lograron, sin embargo, son hasta la fecha, tecnologías sumamente complejas. DCOM se implementa mediante la creación RPC (Remote Procedure Call – “Llamada a procedimientos remotos”) el cual consiste en crear procedimientos expuestos por un servidor que pueden ser ejecutados por otra aplicación, con la limitante que solo soporta parámetros básicos, es decir, no soporta estructuras u objetos. Por otra parte, CORBA, si permite en envío de mensajes completos u objetos, sin embargo, CORBA requiere de la definición de cada objeto en un lenguaje descriptor de interfaz (IDL) y la creación de Stubs, los cuales son representaciones de los objetos y operaciones remotos del lado del consumidor del servicio.

Debido a toda esta problemática, nace la arquitectura SOA (Service-Oriented Architecture), la cual incentiva a que las empresas creen servicios interoperables

que puedan ser aprovechados por toda la organización. Dichos servicios deben de ser contruidos utilizando estándares abiertos, lo que permita que pueden ser consumidos por cualquier aplicación sin importar en que tecnología esté desarrollada.

SOA solo promueva la creación de servicios para lograr la integración de aplicaciones, por lo que se podría decir que CORBA es el precursor de SOA, aunque al momento de crear CORBA no existía el término de SOA. Por su parte, el nacimiento de SOA como arquitectura no solo trae el concepto de crear servicios si no un nuevo paradigma que revoluciono la forma en que construimos software de hoy en día.

Como se estructura una arquitectura de SOA

Lo primero que tenemos que entender al momento de implementar una arquitectura SOA es que, todas las aplicaciones que construimos están diseñadas para ser integradas con otras aplicaciones, por lo que deben de exponer como servicios todas las operaciones necesarias para que otras aplicaciones pueden interactuar con ella.

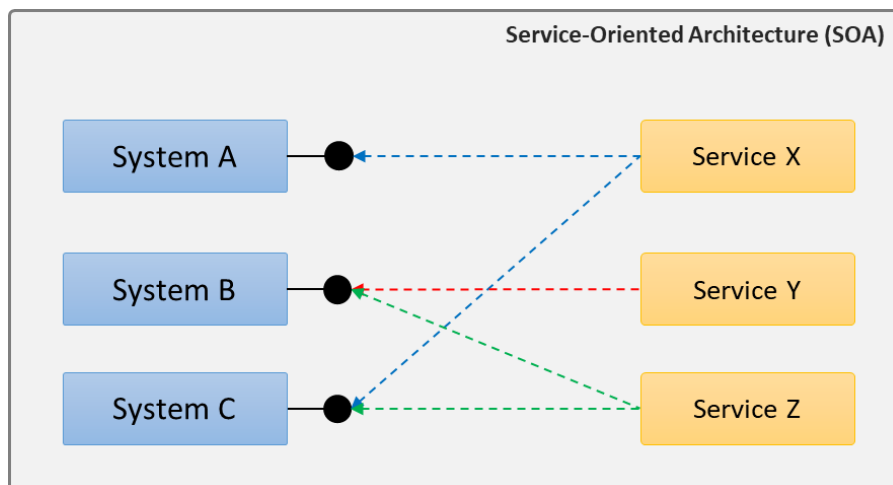


Fig 60 - Consumo de servicios

En la imagen anterior podemos ver como las aplicaciones del lado izquierdo (A, B, C) exponen una serie de servicios que estarán disponibles por la red, los cuales son consumidos por las aplicaciones del lado derecho (X, Y, Z), aun que podrían ser consumidos por cualquier otra aplicación que este dentro de la misma red, o en su defecto, por internet.

La idea central de construir servicios va más allá de solventar la problemática de la integración con aplicaciones heterogéneas, si no que brinda una serie de ventajas que ayudan a la reutilización de los servicios, encapsulamiento de las aplicaciones, seguridad, una larga lista de características.

Independiente de la tecnología

Lo primero que debemos de analizar es que, los servicios deben ser contruidos con estándares abiertos, lo que quiere decir que es independiente del sistema operativo o del lenguaje de programación. Hoy en día conocemos a estos servicios como Web Services.



Nuevo concepto: Web Service

Un servicio web (en inglés, web service) es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet.

— Wikipedia

Para que un servicio sea considerado un Web Service, debe de publicar un WSDL (Web Services Description Language), el cual es un contrato que describe las operaciones que expone el servicio, los tipos de datos esperados en formato XSD (XML Schema).

Este contrato (WSDL) será utilizado por el consumidor para crear mensajes en formato XML para enviar la información, los cuales deberán de cumplir al pie de la letra la estructura definida en el contrato.

Si bien el protocolo HTTP es el más utilizado para la transmisión de los datos, no es la única forma, como muchos creen, pues en realidad SOA no define los protocolos a utilizar, es por ello que es posible transmitir los mensajes por protocolos como FTP o SMTP, lo importante es que el mensaje (XML) cumpla con el contrato (WSDL). Finalmente, el mensaje es envuelto en el protocolo SOAP (Simple Object Access Protocol), el cual es otro protocolo estándar que divide el mensaje en dos secciones, el header y el payload, el primero son una serie de parámetros de cabecera de control, que ayudan a los sistemas a entender el payload, por otra parte, el payload es el mensaje transmitido como tal (XML).

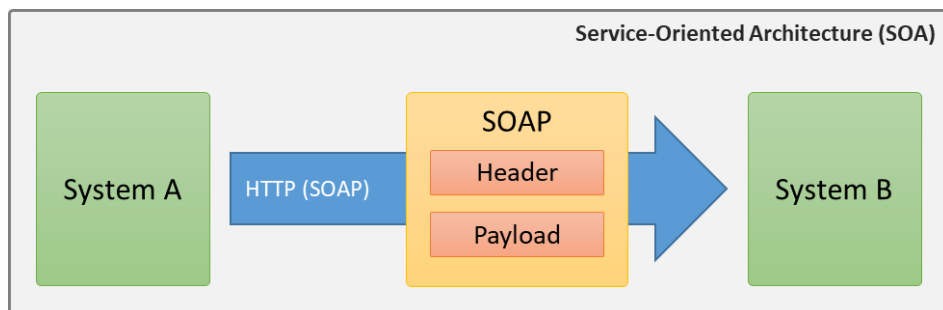


Fig 61 - Consumo de un Web Service.

La utilización de todos estos estandartes y protocolos como HTTP, SOAP, WSDL, XML, XSD hacen que las aplicaciones se puedan comunicar de una forma totalmente homogénea, pues ya no importa en qué tecnología este desarrollado el sistema, sino más bien, cumplir con el formato del mensaje, el cual está perfectamente definido en el WSDL.

Reutilización

Cuando nació DCOM y CORBA no existía una cultura de reutilización, por lo que lo que se exponía eran por lo general operaciones de negocio complejas que realizaban varios pasos, lo que hacía muy difícil que una operación se pudiera ser reutilizada por otras aplicaciones, en este sentido, una operación era construida específicamente para una integración específica.

Por el contrario, SOA promueve la construcción de servicios que hagan una sola tarea, lo que hace que en lugar de tener 1 servicio que haga 10 cosas, mejor hacemos 10 servicios que hagan 1 cosas. En principio se puede escuchar estúpido tener que crear más servicios, sin embargo, al construir servicios que hagan una sola tarea los hace mucho más reutilizables, por otra parte, es posible crear nuevos servicios basado en los anteriores, lo que conocemos como servicios compuestos.

Imagina el siguiente escenario. Eres el programador de un Banco y te han pedido que realices una serie de servicios para operar un cajero automático, en el cual puedes consultar el saldo y realizar retiros. Como regla, realizar un retiro concluye con mostrarle al usuario su nuevo saldo. La respuesta obvia sería crear dos servicios, uno que consulte el saldo y otro que haga el retiro y luego consulte el saldo, sin embargo, para que consultamos nuevamente el saldo en el retiro si ya tenemos un servicio que lo hace, entonces mejor realizamos el retiro y luego llamamos al servicio de consulta de saldo:

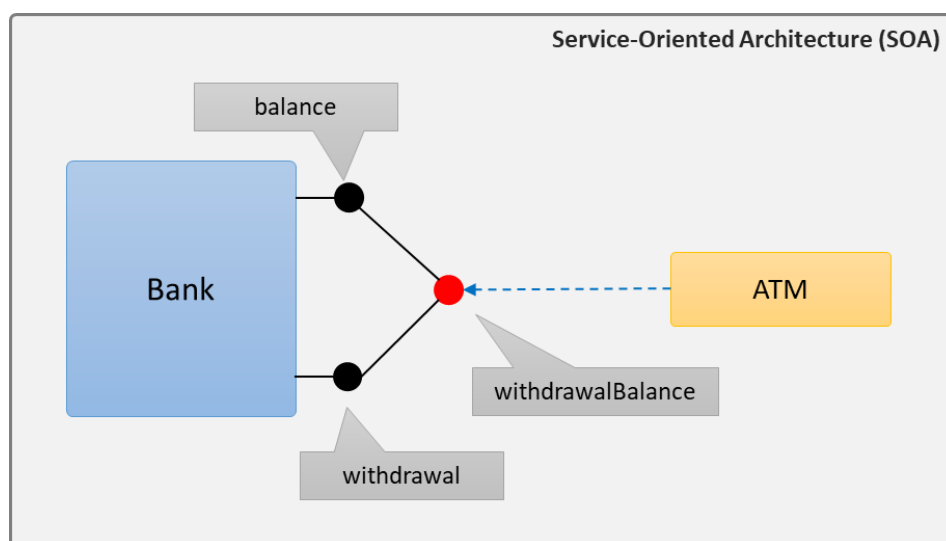


Fig 62 - Composición y reutilización de servicios.

Como podemos ver en la imagen, el banco crea un servicio para consultar el saldo (balance) y otro para realizar retiros (withdrawal) los cuales pueden ser utilizados por cualquier aplicación, no solo el cajero automático, por ejemplo, cuando vas a la ventanilla a retirar efectivo o consultar tu saldo con el agente del banco. Por

otra parte, el cajero necesita realizar las dos operaciones al mismo tiempo, por lo que podemos crear un nuevo servicio basado en los dos anteriores para crear uno más poderoso, el cual retira efectivo y consulta el saldo en un solo paso, de esta forma, quien quiere solo hacer un retiro o consultar el saldo lo podrá hacer. En SOA es común ver una amalgama de servicios creados a partir de otros, lo que hace que aprovechemos al máximo la reutilización de servicios y construir mucho más rápido a partir de servicios existentes.

Cabe mencionar que SOA pone foco en la flexibilidad por encima de la optimización, lo que quiere decir que esta arquitectura no es especialmente rápida con respecto a otras, ya que en ocasiones un servicios tiene que consumir a otros para completar un proceso de negocio, lo que hace que cada brinco a otro servicio provoque una pequeña latencia, además, los servicios por lo general son distribuidos, lo que quiere decir que los diferentes servicios pueden estar en diferentes redes o regiones geográficas, sin embargo, esta arquitectura se caracteriza por su flexibilidad, ya que permite la construcción de nuevos servicios utilizando la infraestructura de servicios existentes.

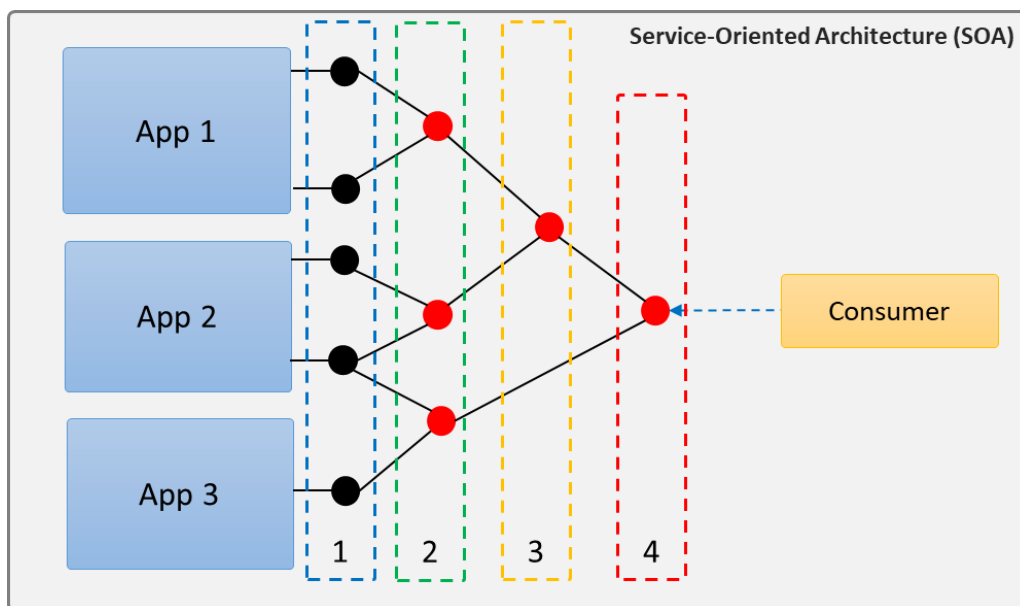


Fig 63 - Composición de servicios.

En la imagen anterior podemos ver claramente como nuevos servicios se van construyendo a partir de los servicios existentes, pero al mismo tiempo, a la vez que vamos subiendo en el nivel de composición, los servicios se van haciendo de más alto nivel, es decir, puede que el primer nivel sean servicios básicos de CRUD (Altas, Bajas, Cambios, Borrado) y los servicios de más nivel son operaciones completas como reservar un vuelo con hospedaje y la renta de un carro, al mismo tiempo que a más alto nivel, diferentes aplicaciones pueden ser impactadas en una sola ejecución.

Encapsulamiento

La encapsulación es una de las características más importantes de SOA, pues permite ocultar la complejidad de un sistema detrás de una serie de servicios de alto nivel. Solo imagina, que es más fácil, pedirle a un servicio que te consulte un pedido con solo el número de pedido, o tener que ir a la base de dato del sistema, saber todas las tablas y campos que tenemos que consultar y retornarlas. Creo que no queda duda que el servicio es mucho más simple, pues no nos tenemos que preocupar en donde están los datos ni como los tenemos que retornar, pues le servicio lo hace por ti.

Mediante los servicios es posible que cualquier persona con pocos conocimientos de la otra aplicación logre una integración, pues los servicios retornarán toda la información necesaria y en un estructura muy organizada, por otra parte, si queremos enviarle datos a la aplicación, los servicios también exponen operaciones de alto nivel, los cuales solo nos piden la información a guardar y ellos se encargan de saber que procesos y tablas afectar, al mismo tiempo que validan toda la información para no permitir introducir valores incorrectos.

Anteriormente veíamos que las aplicaciones se conectaban directamente a las tablas de los otros sistemas, creando un caos total, pues si el programador de la otra aplicación no sabía cómo guardar los datos, crear una serie de registros que terminaba corrompiendo la integridad de los datos:

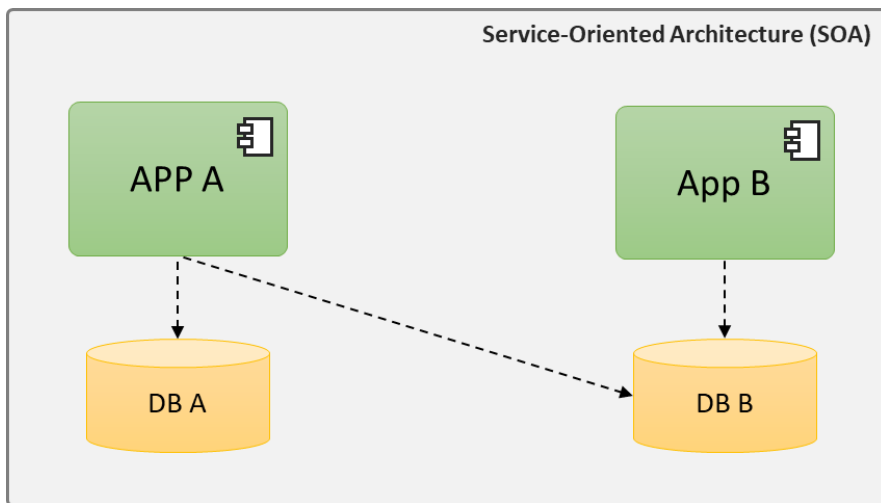


Fig 64 - Compartir una base de datos.

En la imagen podemos ver dos aplicaciones, cada una con su base de datos, sin embargo, cuando la aplicación A quiere consultar o enviar datos a B, lo hace afectando directamente su base de datos, lo cual **es considerado una muy mala práctica hoy en día**.



No compartas las bases de datos

Hoy en día es muy mal visto que una aplicación ajena modifique directamente la base de datos, pues puede llevar a problemas serios de integridad de los datos.

Para solventar este problema, cada aplicación debe de exponer una serie de servicios que permite consultar o enviar información al sistema:

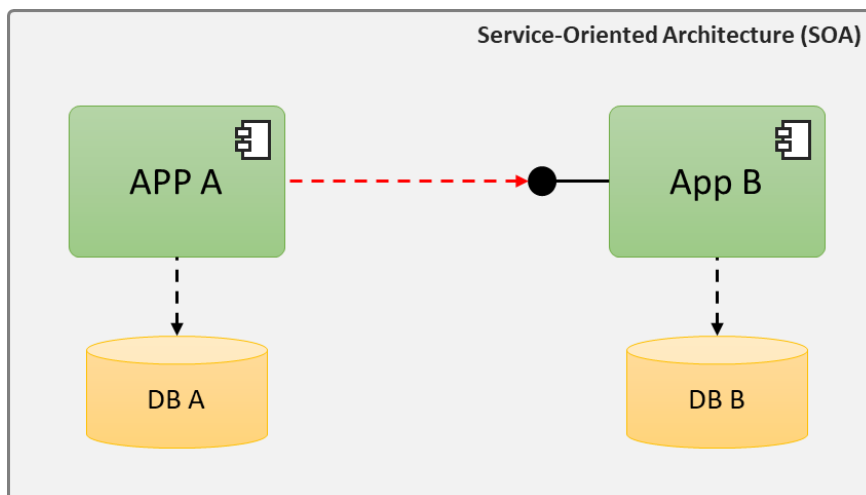


Fig 65 - Integración por servicios.

Enterprise Service Bus (ESB)

Uno de los componentes más importantes cuando hablamos de SOA son los Services Bus, el cual es un componente que permite crear abstracciones de los servicios expuestos de una aplicación para evitar el acoplamiento entre los dos sistemas.

Una de las principales problemáticas cuando comenzamos a trabajar con servicios es que, se comienza a crear una serie de conexiones de punto a punto entre las aplicaciones, lo que hace difícil entender que aplicaciones se comunica con cual, además de que se crear un fuerte acoplamiento entre las aplicaciones, llegando a los extremos que una aplicación no puede ser desplegada si los servicios requeridos no están disponibles, lo que ocasiona una falla en cascada de las aplicaciones. Para solventar este problema, El Service Bus permite crear abstracciones de los servicios expuestos por las aplicaciones, lo que hace que las aplicaciones se comuniquen con el Service Bus, el cual, a su vez, se comunicará con la aplicación final.

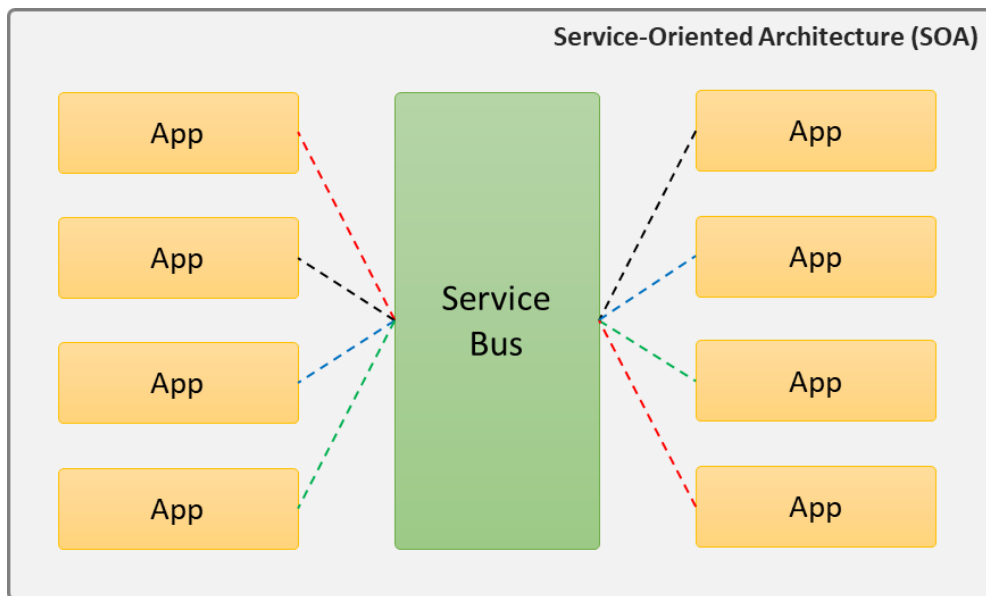


Fig 66 - Arquitectura de un Enterprise Service Bus.

Si estás familiarizado con los patrones de diseño, quizás esta arquitectura te resulte familiar, ya que un Service Bus funciona como un Mediador, el cual se coloca en medio de la arquitectura para mediar la comunicación entre los diferentes sistemas, de esta forma, logramos controlar la comunicación entre las aplicaciones, eliminar el acoplamiento e incluso, agregar lógica adicional como la comprobación de seguridad o auditoría de la comunicación para medir número de invocaciones, tiempos de respuesta, detectar y notificar errores, etc.

En realidad, un Service Bus es un componente muy grande y complejo, del cual existe libros enteros dedicados a enseñar su funcionamiento, por lo que en esta sección solo queremos dar una pequeña introducción.

En el mercado existen diferentes productos ya desarrollados que podemos utilizar, como es el caso de Oracle Service Bus, IBM Message Broker, JBoss ESB, Mulesoft, etc.

Características de SOA

En esta sección analizaremos las características que distinguen al estilo SOA 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:

- Las aplicaciones son pensadas desde el inicio para ser integradas con otras, por lo que exponen una serie de servicios para exponer su funcionalidad al resto de aplicaciones.
- Existe un grado muy alto de desacoplamiento entre las aplicaciones, debido a que trabajan con contratos y no referencias directas a las otras aplicaciones.
- Las aplicaciones se comunican mediante un Enterprise Service Bus (No es la regla, pero es lo más común).
- La comunicación se puede dar por diferentes protocolos (aunque HTTP es el más utilizado).
- Los mensajes intercambiados entre las diferentes aplicaciones siguen un formato específico el cual está definido en el contrato del servicio (WSDL).
- Los servicios son distribuidos, lo que quiere decir que los servicios pueden estar en cualquier parte de la red, ya sea local o remota y en cualquier parte geográfica.

Ventajas y desventajas

Ventajas

- **Reduce el acoplamiento:** La comunicación basada en contratos permite desacoplar las aplicaciones, ya que no existe referencias a la aplicación, si no a un servicio que puede estar en cualquier parte.
- **Testabilidad:** Las aplicaciones basadas en SOA son muy fáciles de probar, pues es posible probar de forma individual cada servicio.
- **Reutilización:** Los servicios SOA son desarrollados para hacer una sola tarea, por lo que facilita que otras aplicaciones reutilicen los servicios que existen.
- **Agilidad:** Permite crear rápidamente nuevos servicios a partir de otros existentes.
- **Escalabilidad:** Los servicios SOA son muy fáciles de escalar, sobre todo porque permite agregar varias instancias de un servicio y balancear la carga desde el ESB.
- **Interoperable:** La interoperabilidad es uno de los factores por lo que existe esta arquitectura y es que permite que aplicaciones heterogéneas se comuniquen de forma homogénea mediante el uso de estándares abiertos.

Desventajas

- **Performance:** La arquitectura SOA no se caracteriza por el performance, pues la comunicación por la red y la distribución de los servicios agregar una latencia significativa en la comunicación.
- **Volumetría:** SOA no se lleva con grandes volúmenes por petición, por lo que para procesar mucha información es recomendable utilizar otras tecnologías.
- **Gobierno:** Para que SOA pueda ser implementado correctamente, es necesario crear un Gobierno que ponga orden sobre el uso y la creación de nuevos servicios, lo que puede representar una carga adicional a la empresa.
- **Más puntos de falla:** Debido a que SOA es una arquitectura distribuida, es necesario implementar estrategias de falla, pues se incrementa el número de puntos de falla.

Cuando debo de utilizar un estilo SOA

Antes que nada, SOA es un estilo arquitectónico compatibles con otras arquitecturas, pues este estilo solo menciona como las aplicaciones deben de exponer servicios para comunicarse con otras aplicaciones, por lo que una vez dentro del sistema, este puede ser un Monolítico, un Microservicio, una aplicación en Capas, etc.

Si viéramos SOA como una cebolla, SOA sería la capa superior, es decir, es la arquitectura de cara a los usuarios, mientras que por dentro, podría implementar cualquier otra arquitectura.

Dicho lo anterior, se podría decir que SOA es utilizada en aplicaciones que están pensadas para ser integradas con otras aplicaciones, o que tiene la necesidad de brindar servicios de negocio a la compañía la cual les permita crear servicios mucho más fáciles y reutilizando lo más posible.

En realidad, SOA es muy flexible en cuanto a su implementación, pues puede ser utilizada en casi cualquier tipo de aplicaciones, por lo que en lugar de detallar en cuales podríamos utilizarlo, me gustaría centrarnos en cuales no deberíamos utilizar SOA.

Básicamente existen tres situaciones en las cuales no deberíamos utilizar SOA:

1. Para aplicaciones que tiene que procesar o enviar masivas cantidades de datos. Es importante diferenciar entre grandes volúmenes de datos y grandes volúmenes de peticiones, con SOA podemos procesar grandes volúmenes de mensajes, pues podemos clusterizar los servicios para atender muchas peticiones, pero es diferente cuando estas peticiones tienen en solitud o respuesta una masiva cantidad de datos, este último es un inconveniente para SOA pues puede poner en jaque a la arquitectura.
2. Cuando el performance es lo más importante en nuestra aplicación, SOA no será una buena idea, pues la arquitectura SOA agrega muchas capas que hace que el procesamiento de los datos no sea tan rápido.
3. Implementar SOA puede resultar atractivo, sin embargo, hay que tomar en cuenta que su implementación requiere de administración y gobierno, por lo que para empresas pequeñas o que no tiene procesos maduros puede llegar a ser un obstáculo en lugar de una ventaja. Si no te sientes listo para asumir estas responsabilidades, será mejor esperar un poco.

Conclusiones

Como hemos podido analizar en este capítulo, SOA es una arquitectura muy flexible que permite la interoperabilidad entre las diferentes aplicaciones empresariales, ya que utiliza una serie de estándares abiertos que permite que sean independientes de la tecnología o la plataforma, sin embargo, SOA es uno de los estilos arquitectónicos más complejos por la cantidad de componentes que interviene, el gobierno y toda una serie de lineamientos que se deben de cumplir para implementarla correctamente, por lo que te sugiero que si quieres aprender SOA de lleno, busques una literatura especializada en el tema, ya que esta arquitectura tiene libros completos sobre cómo implementarla, que hacen que sea imposible de cubrir en este libro.

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.

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.

Como se estructura un Microservicios

Para comprender los Microservicios es necesario regresar a la arquitectura Monolítica, donde una sola 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:

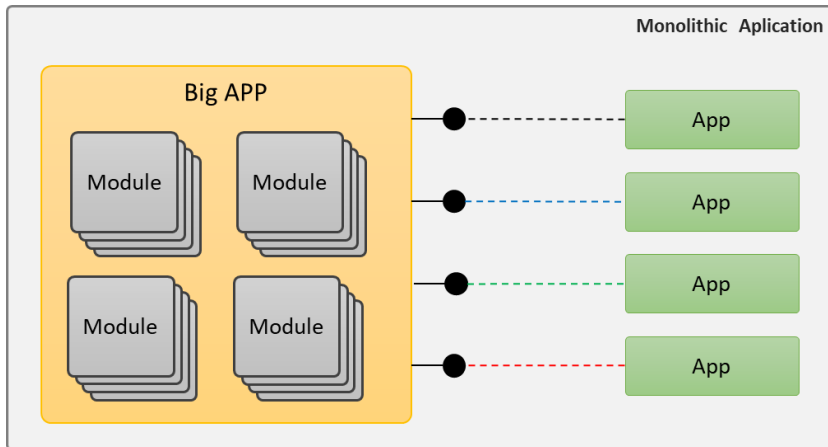


Fig 67 - Arquitectura Monolítica.

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.

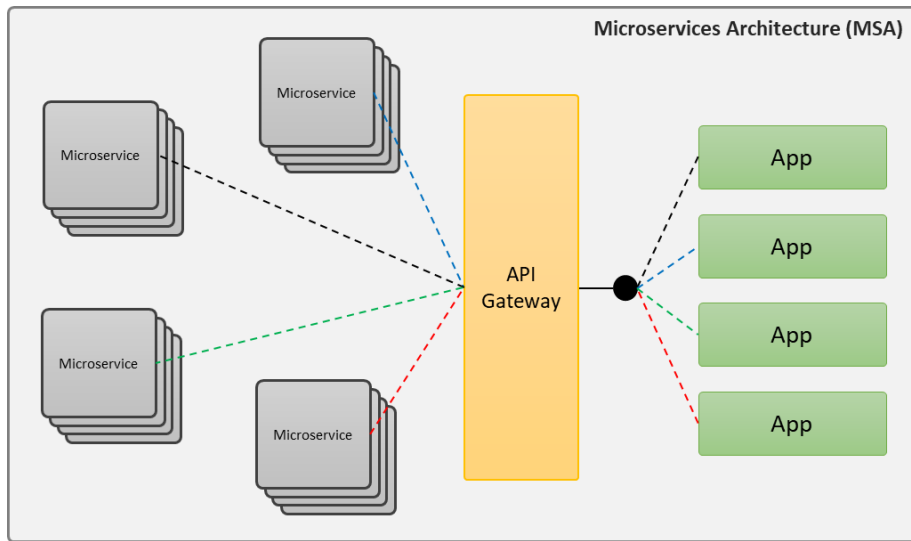


Fig 68 - Arquitectura de Microservicios

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).

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.

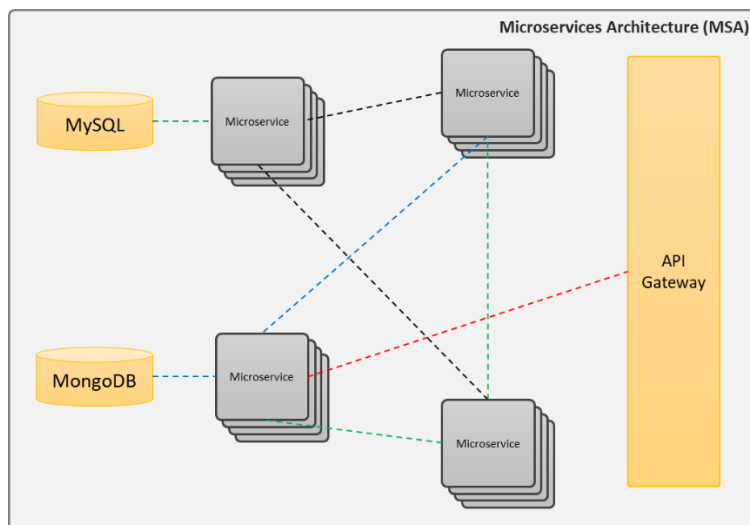


Fig 69 - Comunicación entre Microservicios

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.

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.

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.

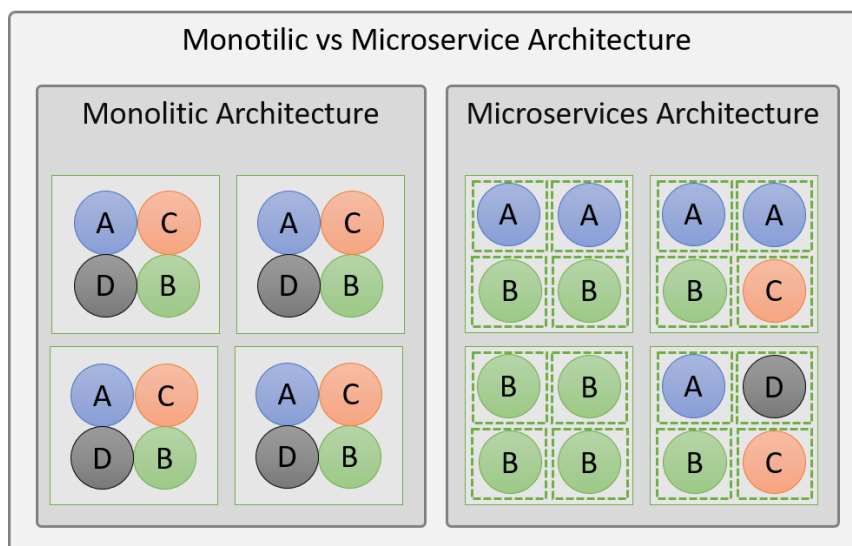


Fig 70 - Escalamiento Monolítica vs Microservicios

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 tanta demanda, así que solo dejamos una instancia.

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.

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.
- 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.

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.
- **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.

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.

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.



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 nuestro Monolítico en 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.

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 transmitirti en este libro.

Event Driven Architecture (EDA)

La arquitectura dirigida por eventos o simplemente EDA (por sus siglas en inglés) es una arquitectura asíncrona y distribuida, pensada para crear aplicaciones altamente escalables.

En una arquitectura EDA los componentes no se comunican de forma tradicional, en la cual se establece comunicación de forma síncrona, se obtiene una respuesta y se procede con el siguiente paso. En esta arquitectura, se espera que las aplicaciones lancen diversos “eventos” para que otros componentes puedan reaccionar a ellos, procesarlos y posiblemente generar nuevos eventos para que otros componentes continúen con el trabajo.



Nuevo concepto: Evento

Un evento lo podemos definir como un cambio significativo en el estado de la aplicación, ya sea por un dato que cambio o alguna acción concreta en el sistema que merece la pena ser observada para tomar acciones en consecuencia.

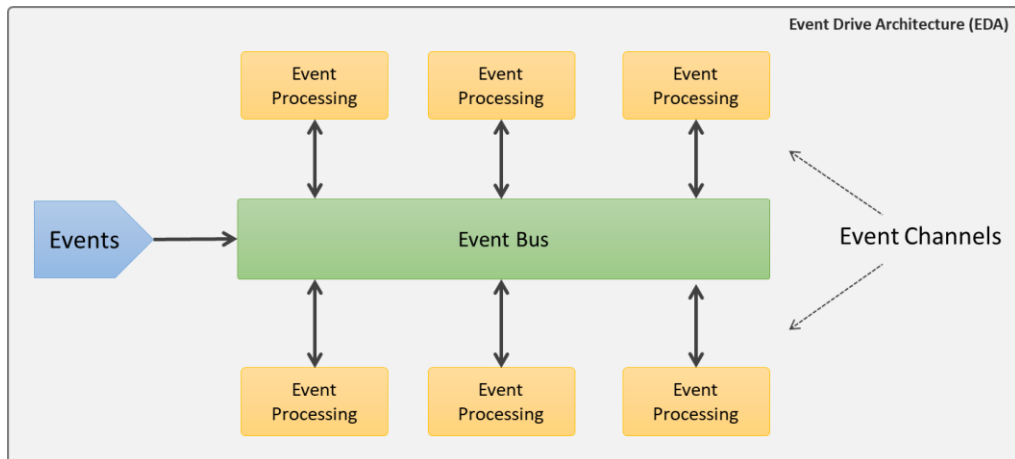


Fig 71 - Arquitectura EDA

Como podemos ver en la imagen anterior, todo inicia con la llegada de un nuevo evento a algo que llamaremos por ahora *Event Bus*, el cual es un componente que se encarga de administrar todos los eventos de la arquitectura. El *Event Bus* se encargará de recibir el evento y colocarlo en los *Event Channels*, las cuales son básicamente una serie de Colas (*Queues*) o Temas (*Topics*) sobre los cuales habrá ciertos componentes a la escucha de nuevos mensajes.

Por otra parte, los *Event Processing* son componentes de negocio que está a la escucha de nuevos eventos depositados en los *Event Channels*. Los *Event Processing* son los encargados de procesar los eventos, es decir, realizar acciones concretas sobre el sistema, como crear, actualizar o borrar algún registro, finalmente, el proceso puede terminar de forma silenciosa, o puede generar un nuevo evento para que otro componente lo tome y continúe con el procesamiento.

Un punto importante a tomar en cuenta es que todos los *Event Processing* son componentes distribuidos, lo que quiere decir que están totalmente desacoplados del resto y que el funcionamiento de uno no afecta al resto. De esta forma, cada *Event Processing* administra sus propias transacciones, lo que hace difícil mantener la atomicidad de la transacción en una serie de eventos, por lo que es un punto importante a tomar en cuenta al momento de trabajar con esta arquitectura.

En esta arquitectura, cada *Event Processing* solo tiene conocimiento de la tarea que el realiza y no tiene conocimiento de la existencia de otros *Event Processing*, lo cual permite que trabajen de forma autónoma.



Tip

De forma general, podemos comprar la arquitectura EDA con el patrón de diseño Observable, ya que EDA está a la escucha de ciertos eventos sobre un determinado *Event Channel* en lugar de escuchar los eventos directamente del objeto observado.

Ya que EDA es una arquitectura asíncrona, no podemos darnos el lujo de ejecutar una operación y esperar una respuesta, por lo que debemos dejar la solicitud y esperar hasta que alguien la atienda, durante este proceso, podemos llamar directamente a la operación asíncrona del otro componente, sin embargo, esto es una mala idea:

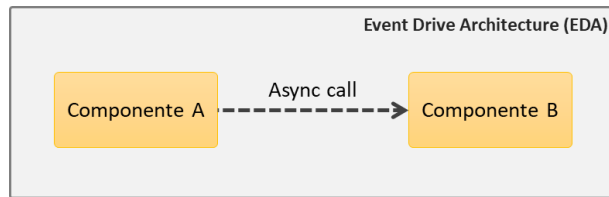


Fig 72: Llamada directa a una operación asíncrona.

En la imagen anterior podemos ver claramente un fuerte acoplamiento entre los dos componentes, lo cual va en contra de la arquitectura EDA, la cual busca un bajo acoplamiento entre los componentes, además, de que la operatividad de un componente no debe depender de la operatividad del otro. En este escenario, el componente A depende de que B esté funcionando al momento de realizar la llamada, ya que, si no, podrían suceder dos cosas, que el proceso completo falle porque no pudo realizar la llamada a B o la segunda, que el proceso continúe ante la falla, pero perdamos el mensaje enviado a B. Sea cual sea el caso, no es el esperado, porque al final tenemos un fallo total o parcial, lo que nos deja en un estado inconsistente.

Para solucionar este tipo de problemas, debemos hacer uso de un Intermediario que tomará las solicitudes, las almacenará y las entregará al consumidor adecuando cuando este esté disponible, de esta forma, el productor del mensaje podrá dejar el mensaje en el Intermediario y continuar con el proceso, pues el Mediador garantizará que el mensaje será entregado al consumidor. Por otra parte, el consumidor (Componente B) estará a la escucha de nuevos mensajes en el Intermediario para procesarlos. Si el consumidor está fuera de servicio cuando el productor (Componente A) deja el mensaje, este no se perderá, pues el Intermediario lo guardará y lo entregará cuando el Consumidor esté nuevamente operativo.

En EDA, el Intermediario son conocidos como Message-Oriented-Middleware (MOM) o Middleware Orientado a Mensajes por su traducción al español. Los MOM son aplicaciones especializadas en el transporte confiable de mensajes. Entre los más conocidos están WebSphere MQ, Microsoft Message Queuing, Oracle Textudo, Apache Kafka, etc. Más adelante analizaremos como los MOM juegan un papel fundamental en la implementación de una arquitectura EDA.

Como se estructura una arquitectura EDA

En la sección anterior hablábamos de un componente llamado *Event Bus*, sin embargo, este fue un nombre genérico que utilizamos para poder explicar la arquitectura, sin embargo, en la práctica, existen dos topologías diferentes para implementarlo, las cuales tiene diferencias sustanciales en su implementación.

Topología con Mediador

La topología de Mediador es utilizada para procesar Eventos complejos, donde el éxito de la operación requiere de una serie de pasos definidos y un orden de ejecución exacto, lo que en EDA conocemos como Orquestación.



Nuevo concepto: Orquestación

La orquestación es un proceso central que coordina los pasos y secuencia de ejecución para llevar a cabo una tarea más grande.

El Mediator es un componente que toma el evento primario y fracciona un evento en eventos más pequeños para ser procesados por varios *Event Processing*, de esta forma, cada *Event Processing* realiza una pequeña parte de la tarea de forma Asíncrona. Adicionalmente, el *Mediador*, podrá ejecutar todos los pasos en forma Asíncrona, o podrá requerir que ciertos eventos sean procesados antes de proceder con el resto de pasos. En este sentido, el *Mediador* no implementa nada de lógica de negocio ni transacciona con los sistemas, sino que solo se encarga de orquestar los eventos para llevar un orden adecuado de ejecución.

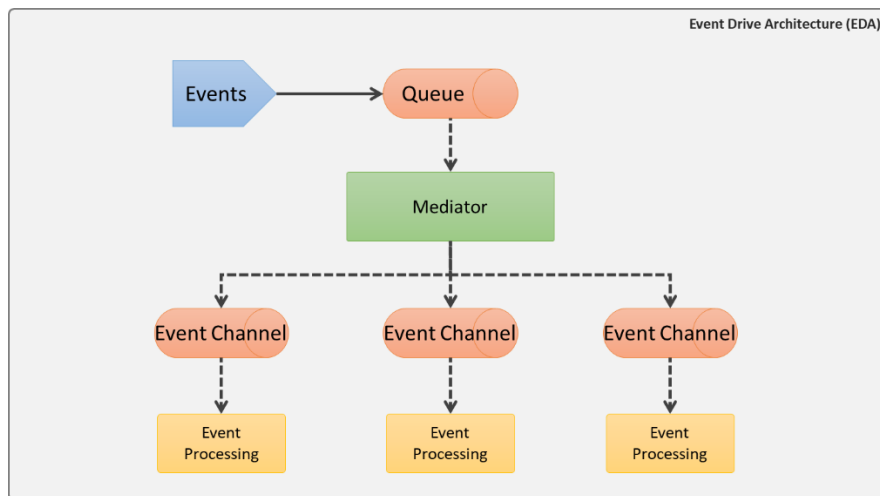


Fig 73 - Topología con Mediator

En la imagen anterior podemos observar como el Mediator toma el Evento inicial y luego lo distribuye a varios *Event Channels* para que varios *Event Processing* pueda procesar el evento. Cabe mencionar que todos los *Event Processing* puede recibir el mismo Evento, sin embargo, lo más normal es que el Mediator fraccione el Evento inicial en Eventos más pequeños para ser procesados por los diferentes *Event Processing*.

Si bien, la imagen anterior deja claro cómo es que el Evento se distribuye entre los diferentes *Event Processing*, no se logra observar cómo es que se lleva a cabo la orquestación, esto se debe a que el *Mediator* por sí mismo es un componente que tiene que ser desarrollado para orquestar adecuadamente los eventos. Algunas tecnologías utilizadas para este propósito son los Services Bus, BPEL y BPM, los cuales son tecnologías utilizadas para la orquestación de servicios y Eventos.

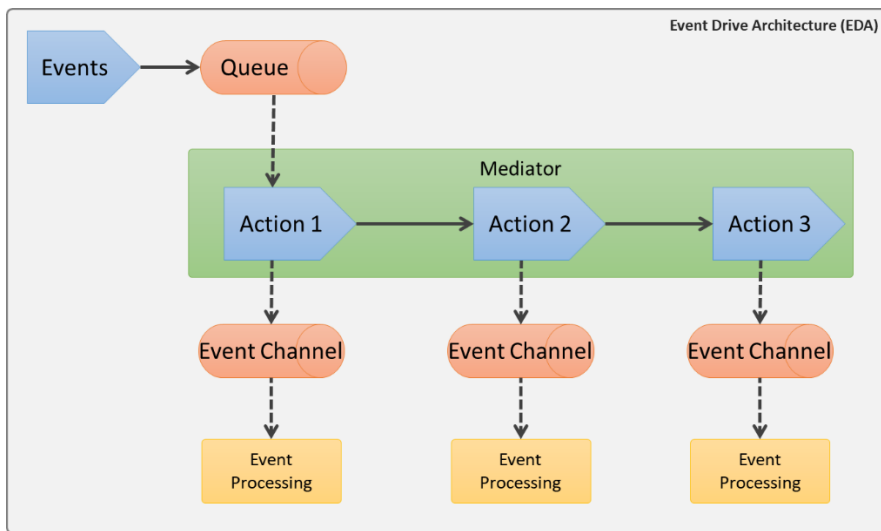


Fig 74 - Orquestación de un Mediator

En esta nueva imagen podemos ver con detalle cómo es que el Mediator orquesta la ejecución de los eventos. El Mediator puede ejecutar todos los pasos de forma Asíncrona o puede esperar a que algún paso termine antes de proceder con el siguiente, todo dependerá de cómo implementemos el Mediator.

La topología de Mediator es muy utilizada en ambientes empresariales donde tiene proceso de integración muy complejos y que por lo general tiene Middleware especializados para realizar estas tareas.

Topología con Broker

La topología de Broker es utilizada para procesar Eventos simples, en los cuales no es necesario orquestar una serie de pasos para llegar al objetivo, en su lugar, los *Event processor* toman los mensajes directamente de los *Event Channels* y pueden publicar nuevos eventos para que otros *Event processor* continúen procesando los siguientes eventos.

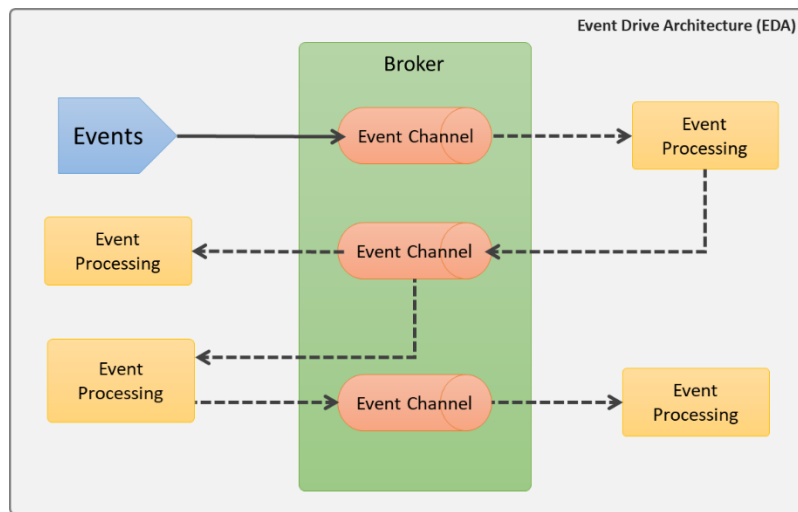


Fig 75 - Topología con Broker.

En este sentido, podemos comparar la topología de Broker con una carrera de relevos, en donde cada *Event Processing* corre una parte de la carrera, y al finalizar deja otro evento en el Broker para que otro *Event Processing* lo tome y continúe. En este sentido, no existe un orden de ejecución definido, si no que cada *Event Processing* está a la espera de un determinado tipo de evento para tomarlo y procesarlo.

Esta topología es muy común en empresas más pequeñas o dinámicas, donde lo que buscan es una mejor agilidad y poder responder más rápido a los cambios, pues al no tener un orquestador, nos permite cambiar rápidamente el flujo de ejecución o adecuarnos a los cambios sin tener que actualizar el Mediador.

Características de una arquitectura EDA

En esta sección analizaremos las características que distinguen al estilo EDA 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:

- Se conforma de varios componentes distribuidos que reaccionan únicamente a Eventos.
- Usa sistemas de mensajería asíncrona como Colas (Queues) y Tópicos (Topics).
- Existe un componte intermediario para administrar los mensajes de forma segura, como son los Message-Oriented-Middleware (MOM) o Service Bus.
- Son con frecuencia utilizados en arquitecturas SOA y Microservicios.
- Completar una operación de negocio requiere por lo general del involucramiento de varios componentes que realizan una pequeña parte de la tarea global.
- Tanto el productor cómo los consumidores están totalmente desacoplados entre sí, incluso, no tiene conocimiento de si existe tal consumidor.

Ventajas y desventajas

Ventajas

- **Escalabilidad:** La escalabilidad es uno de los puntos más fuertes de esta arquitectura, pues permite que cada consumidor (*Event Consumer*) puede escalar de forma independiente y reduce al máximo el acoplamiento entre los componentes.
- **Despliegue:** Debido al bajo acoplamiento entre los componentes, es posible el despliegue sin preocuparse por dependencias o precondiciones, al final, los componentes solamente se suscriben para recibir eventos y reaccionar ante ellos.
- **Performance:** Esta es que ventaja cuestionable, ya que al igual que en REST o SOA, EDA necesita pasar por una serie de pasos para completar una tarea, agregando retrasos en cada paso, desde colocar el Evento por parte del

productor, esperar a que el consumidor lo tome, y generar nuevos Eventos, sin embargo, la naturaleza Asíncrona de EDA hace que esta desventaja se supere mediante el procesamiento en paralelo.

- **Flexibilidad:** EDA permite responder rápidamente a un entorno cambiante, debido a que cada componente procesador de eventos tiene una sola responsabilidad y está completamente desacoplado de los demás, de esta forma, si ocurre un cambio, se puede aislar en un solo componente sin afectar al resto, además, si un nuevo requerimiento es requerido, solo es necesario regresar un nuevo tipo de procesador de eventos que escuche un determinado tipo de evento.

Desventajas

- **Testabilidad:** Una arquitectura distribuida y asíncrona agrega cierta complejidad a las pruebas, pues no es posible generar un evento y esperar un resultado para validar el resultado, en su lugar, es necesario crear pruebas más sofisticadas que creen los eventos y esperen la finalización del procesamiento del evento inicial más toda la serie de eventos que se podrían generar como consecuencia del evento inicial para validar el resultado final.
- **Desarrollo:** Codificar soluciones asíncronas es complicado, pero aún más, es la necesidad de crear manejadores de errores más avanzados que permitan recuperarse en una arquitectura asíncrona, donde la falla en un procesador no significa la falla en el resto, lo que puede ocasionar la inconsistencia entre los sistemas.

Cuando debo de utilizar un estilo EDA

EDA es una arquitectura muy robusta que permite crear soluciones altamente escalables, pero tiene como principal problemática su dificultad para programar, la cual puede ser mayor a las ventajas que podría ofrecer si se implementa de forma inadecuada, es por ello que debemos pensar en EDA solo cuando sabes que nuestra aplicación está planeada para un rápido crecimiento o para un volumen de transacciones muy alto, en otro caso, podríamos estar haciendo una sobre ingeniería que nos podría salir muy caro.

En este sentido podríamos decir que debemos utilizar EDA cuando:

1. Pensamos crear aplicaciones altamente escalables.
2. Necesitamos explotar al máximo la capacidad de procesamiento paralelo, concurrente y asíncrono.
3. Donde una respuesta síncrona no es obligatoria.
4. En arquitecturas donde es necesario producir eventos en tiempo real para que otros suscriptores puedan actuar en consecuencia en tiempo real.

Hoy en día existen varias herramientas que nos permite utilizar EDA, entre las que sobre salen Apache Kafka, IBM Message Broker, Mulesoft, RabbitMQ, Oracle SOA Suite, entre otras muchas alternativas más.

Aplicaciones altamente escalables

Las aplicaciones que utilizan EDA como arquitectura central, tiene la capacidad de escalar de una forma extraordinaria, ya que solo hace falta agregar nuevos consumidores para extender el poder de procesamiento de la solución completa, incluso, es posible registrar dos o más instancias del mismo consumidor para una red de consumidores que permitan procesar en paralelo una gran cantidad de eventos.

Paralelismo, concurrencia y asincronismo

La capacidad de explotar el paralelismo y la concurrencia ayuda a que el procesamiento de una transacción de negocio completa se lleve a cabo mucho más rápido, ya que varios consumidores pueden trabajar de forma independiente y por separado, lo que permite que la tarea termine más rápido que si se ejecutara de forma secuencial.



Nuevo concepto: Concurrencia

La concurrencia es la capacidad del CPU para procesar más de un proceso al mismo tiempo.



Nuevo concepto: Paralelismo

El paralelismo sigue la filosofía de “divide y vencerás”, ya que consiste en tomar un único problema, y mediante concurrencia llegar a una solución más rápido.

Puedes comprender más sobre lo que es la [concurrencia y paralelismo](#) en mi artículo.

Aplicaciones en tiempo real

Si bien EDA no es una arquitectura puramente en tiempo real, sí que permite crear eventos que pueden ser procesados casi en tiempo real, es decir, un productor crear un evento y un consumidor lo puede procesar justo cuando está listo para hacerlo, lo que permite que pueda recibir eventos con un retraso casi nulo y reaccionar en consecuencia, de esta forma, nuestra aplicaciones pueden crear eventos a medida que transaccionamos en la aplicación y notificar a todos los interesados en tiempo real, en lugar que las aplicaciones tengan que ir a consultar los datos cada X tiempo para obtener las últimas novedades.

Conclusiones

EDA es sin lugar a duda una de los estilos arquitectónicos más escalables que existe y que permite que empresa globales puedan atender a millones de solicitudes sin colgar sus sistemas, ya que permite dividir el trabajo en diversos procesadores de mensajes que trabajan de forma totalmente asiladas y desacopladas del resto, sin embargo, implementar EDA no es fácil, pues requiere un esfuerzo mucho mayor que una arquitectura síncrona, y requiere una mejor preparación de los ingenieros que hay detrás de la solución.

Otra de las cosas a considerar al implementar EDA es que, debemos tener un conocimiento más profundo de la solución de negocio que vamos a desarrollar, pues necesitamos ese conocimiento para saber cómo dividir la aplicación en los

diversos consumidores, ya que de lo contrario podemos crear módulos de software que no sean lo suficientemente desacoplados del resto.

Finalmente, debemos de entender que en una arquitectura EDA no existe el concepto de transacciones, pues cada consumidor manejará de forma independiente sus propias transacciones, por lo que si un paso de la cadena de procesamiento falla, no implicará el rollback en todos los consumidores, es por ello que debemos de tener manejadores de errores muy avanzados que permiten a un proceso recuperarse en caso de fallo o de plano que levante una serie de nuevos eventos para hacer el rollback en los otros consumidores. Y toma en cuenta que en EDA no hay garantía que un evento sea procesado, ya sea porque no hay consumidores para un determinado evento o por un error en el diseño.

Representational 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

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.

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.

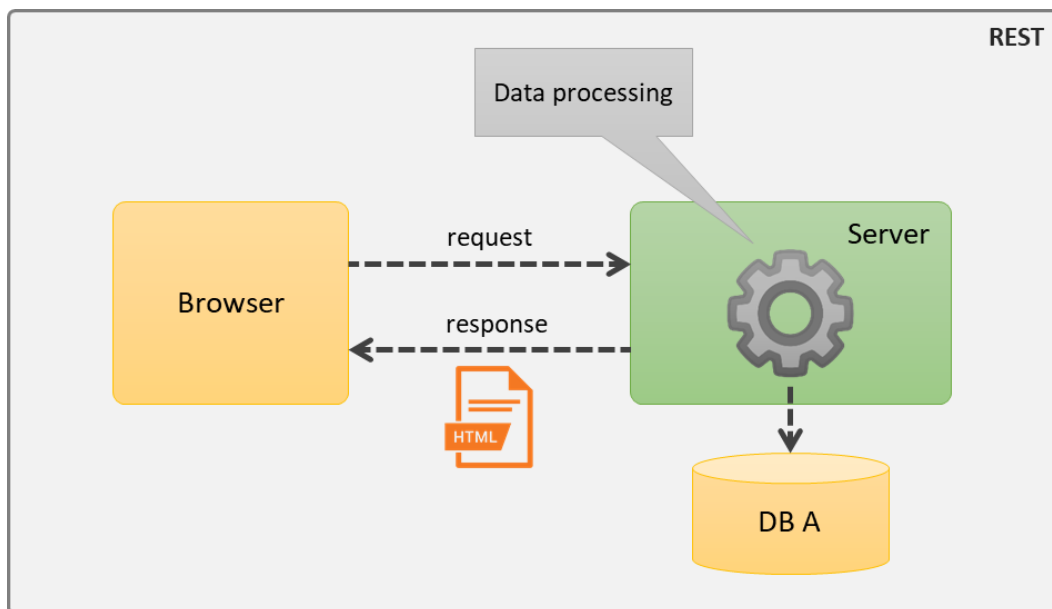


Fig 76 - Procesamiento del lado del servidor.

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.

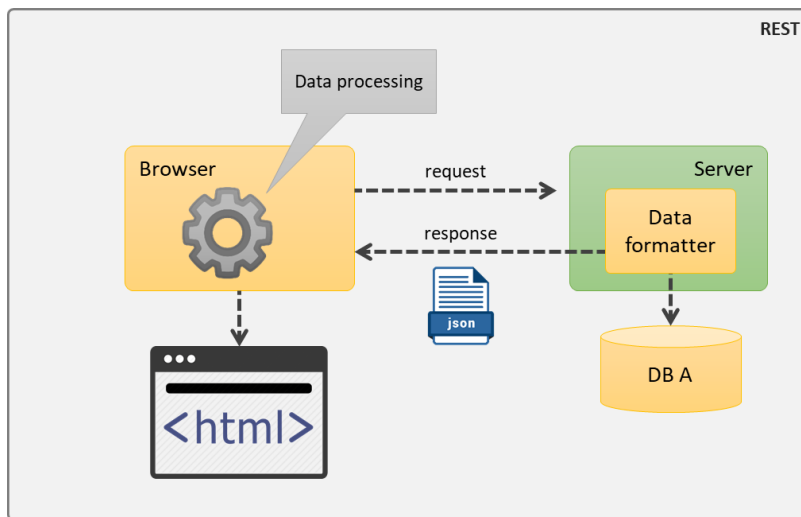


Fig 77 - Procesamiento del lado del cliente

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.

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.

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".

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 componentes, mejorando la simplicidad al proporcionar una separación clara de las preocupaciones y

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.

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.

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.

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

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

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, PATCH. 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.

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.

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 consigo 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.

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.

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.

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.

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.

Proyecto E-Commerce

Capítulo 5

Hablar de arquitectura de software sin ponerlo en práctica es fácil, cualquiera puede alardear de saber arquitectura de software cuando se trata de crear diagramas y unas cuantas flechas que indican como se “supone” que debe de funcionar la aplicación, sin embargo, cuando te pones a analizar con detalle cada componente, su responsabilidad, tu tolerancia a fallas y como podría escalar en el futuro, te das cuenta que en realidad pocos de los que se hacen llamar arquitectos, lo son realmente, por ese motivo, he decidido crear esa sección, donde vamos a explicar un proyecto completo de E-Commerce, desde el diseño hasta la implementación, para demostrar que todo lo que hemos estado hablando y de lo que hablaremos más adelante es perfectamente posible implementar y obtener buenos resultados.

En esta unidad explicaremos a nivel arquitectónico como la aplicación de E-Commerce se compondrá, y en la siguiente unidad abordaremos como cada patrón arquitectónico va encajando en la arquitectura, de esta forma, busco que veas que todo lo que explico en este libro es totalmente implementable y demostrar que todo lo que menciono en este libro no es solo un conjunto de buenas intenciones que en la práctica no se implementa.



Este no es un libro de programación

Algo que quisiera dejar muy en claro antes de comenzar con el proyecto, es que este no es un libro de programación, ni está enfocado a enseñar a utilizar alguna tecnología en específico, por lo que solo nos detendremos a explicar los aspectos relevantes con la arquitectura y no como funciona toda la aplicación

Por otra parte, si tu no desarrollas en Java o React que son las tecnologías que utilizaremos, no deberías de tener problemas en entender su funcionamiento si realmente eres una aspirante a Arquitecto de software o incluso si ya tienes experiencia avanzada en el desarrollo de software en cualquier tecnología. Este es un libro de Arquitectura de Software y no de programación, por lo que como arquitectos debemos comenzar a comprender el funcionamiento incluso si no dominamos la tecnología en cuestión.

El Proyecto E-Commerce

El Proyecto E-Commerce es una aplicación completa de comercio electrónico, donde los clientes pueden realizar la compra de diversos productos, los cuales puede agregar a su carrito de compras e introducir su tarjeta de crédito para completar la compra, sin embargo, la aplicación web o lo que ve el usuario es solo la punta del Iceberg de toda nuestra aplicación, pues tendremos un super Backend distribuido que procesará los pedidos en una arquitectura de Microservicios.

Pero antes de contarte más, es importante que vemos el diagrama de componentes que conforman todo el ecosistema de la plataforma E-Commerce:

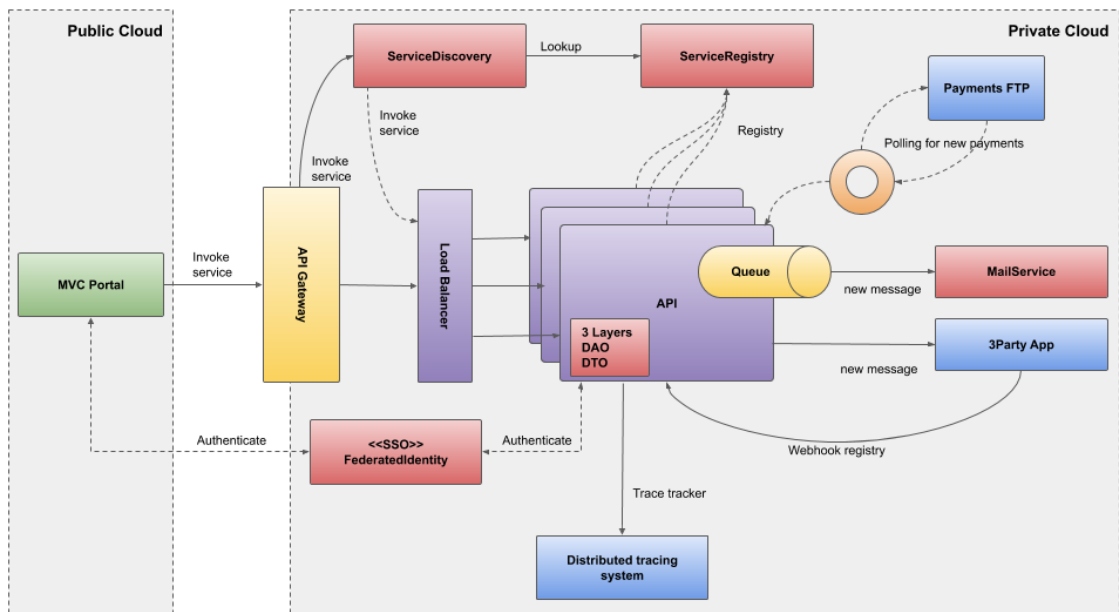


Fig 78 - E-Commerce architecture

Puedo asegurarte que esta imagen no hace justicia a todo lo que aprenderemos en este libro, pues son tantas cosas, tantos patrones que es difícil resaltarlos en el diagrama, sin embargo, trataré de explicar cómo funciona la arquitectura a un alto nivel y en la siguiente unidad iremos profundizando en cada aspecto a menudo que vamos explicando los patrones arquitectónicos.

Lo primero que podemos observar son dos cajas a alto nivel, una nube pública (izquierda) y una nube privada (derecha) las cuales delimitan lo que los usuarios pueden ver desde internet y lo que no pueden ver, por ejemplo, en la nube pública tenemos la aplicación web del E-Commerce, que es donde un usuario normal puede entrar para realizar sus pedidos, de esta forma, cualquiera con una conexión a internet podría acceder. Del otro lado (derecha) tenemos la nube privada, la cual alberga todos los Microservicios que en su conjunto conforman toda la infraestructura que hace posible el funcionamiento de la aplicación web, sin embargo, al ser una red privada impide que cualquiera pueda acceder a los servicios de la red, incluyendo la misma aplicación web del E-Commerce, si leíste bien, ni la aplicación web puede acceder a los servicios o recursos de la red privada, por ello, creamos una API Gateway, la cual podemos ver como una caja amarilla al borde de la nube privada.

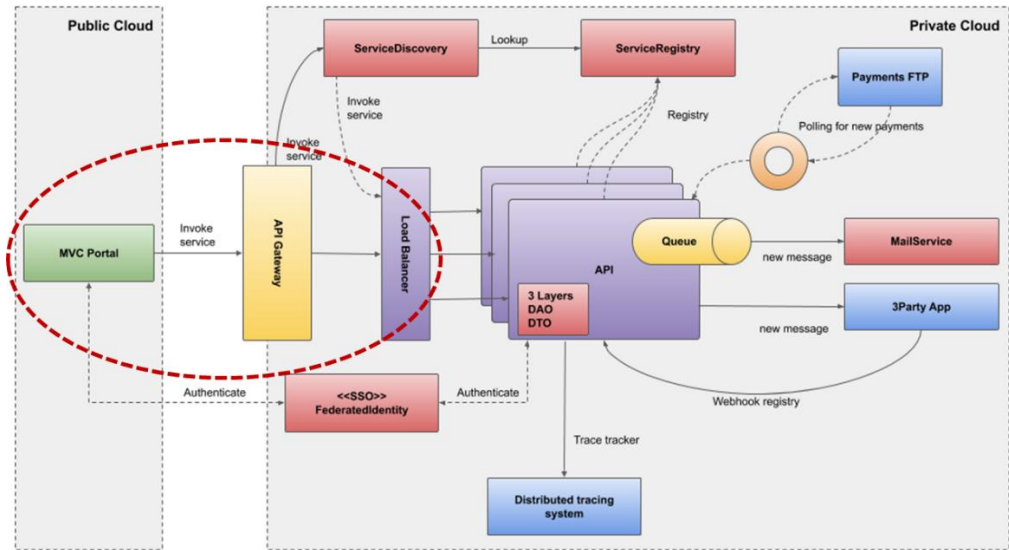


Fig 79 - API Gateway.

El API Gateway es una compuerta que permite controlar que recursos o servicios de nuestra infraestructura estarán disponibles en la red pública, de esta forma, exponemos solo los recursos necesarios y ocultamos el resto.

A pesar de que los recursos expuestos por el API Gateway son visibles por cualquiera, siguen siendo delicados, por lo que debemos de controlar el acceso.

API Central (E-Commerce)

Si bien la arquitectura es un conjunto de Microservicios y cada uno tiene su responsabilidad concreta, existe un componente que podríamos decir que es el Microservicio central o principal, el cual realiza las tareas más importantes, cómo procesa los pedidos y administra los productos, el componente lo hemos llamado simplemente API.

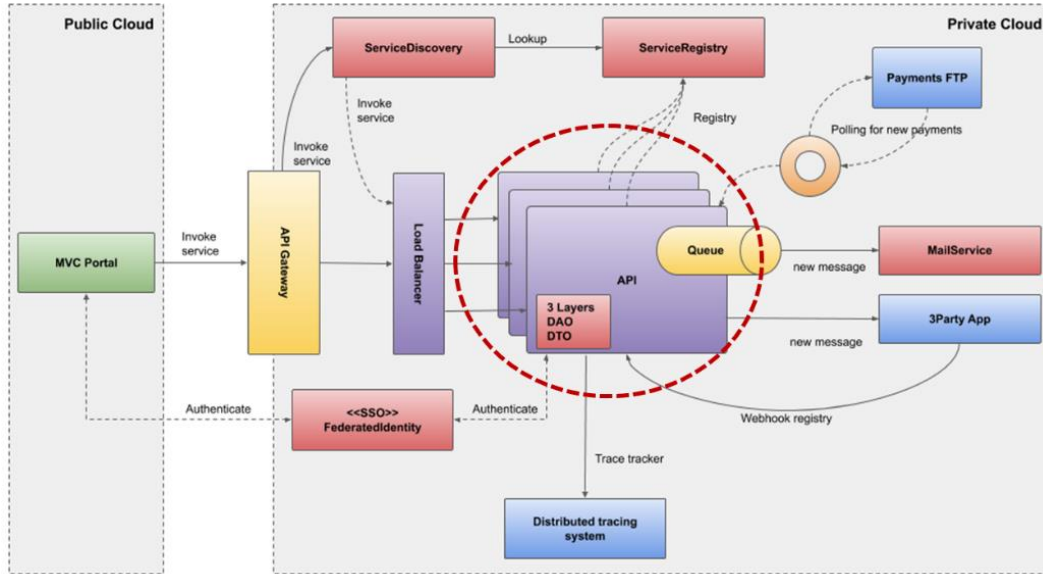


Fig 80 - API Central.

Este Microservicio lo utilizaremos como laboratorio para implementar diversos patrones arquitectónicos, que van desde separar la lógica de datos (Data Access Object – DAO) de la de servicios, objetos de transferencia de datos (Data Transfer Object), una arquitectura de 3 capas de separar la lógica de datos, negocio, servicio y presentación, finalmente, implementaremos el patrón Circuite Braker para tratar los errores del procesamiento de pedidos y no perder ni una sola venta.

Seguridad basada en Tokens

La aplicación de E-Commerce utiliza un sistema de Inicio de Sesión único (Single Sign On – SSO), el cual consiste en implementar una aplicación independiente que tiene como objetivo autenticar a los usuarios y emitir tokens, lo cuales deberán ser enviados a los servicios en lugar de un usuario y contraseña. Dicho token lo validaremos desde el API Gateway para centralizar la autenticación y evitar que autenticar al usuario en cada punto o servicios de la arquitectura.

De esta forma, cuando un usuario quiera acceder a la aplicación web, tendrá que autenticarse en el Single Sign On (Componente Federated Identity del diagrama), para esto, tendrá que mandar su usuario y contraseña para obtendrá un token como respuesta que tendrá que guardar en un lugar seguro. Dicho token deberá ser enviado en las posteriores invocaciones al API Gateway para autenticarse.

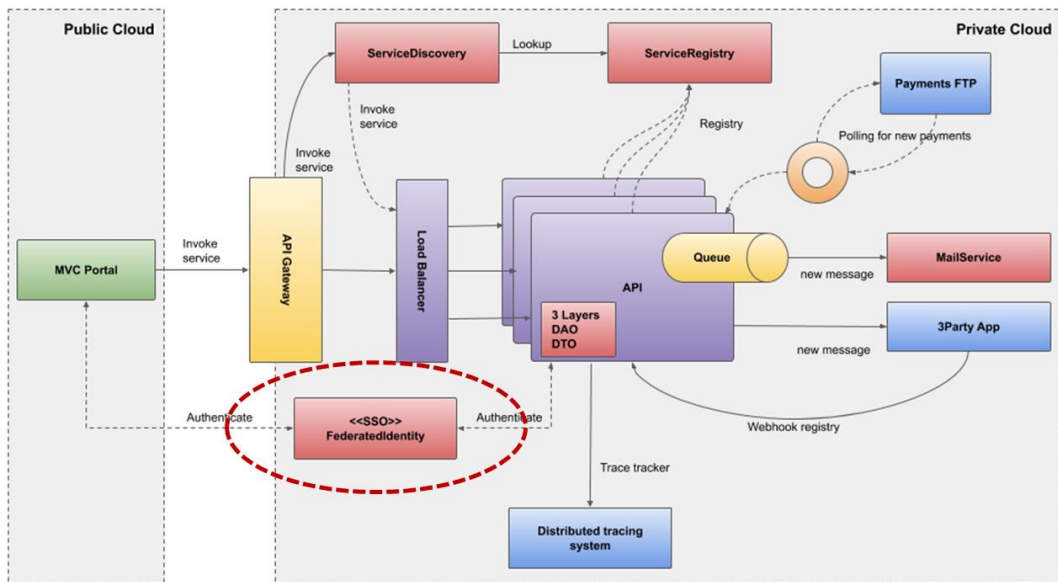


Fig 81 - Single Sign On.

Registro de servicios y autodescubrimiento

Debido a que todos los componentes de la arquitectura son servicios independientes que pueden funcionar en servidores y puertos diferentes, es complicado gestionar la configuración de donde está cada Microservicio, por lo que en lugar de configurar cada servicio, le indicamos en donde se tiene que registrar una vez que se inicie, de esta forma, tanto el servidor como el puerto puede variar de forma aleatoria sin afectar el funcionamiento de los demás servicios, los cuales podrán ir al registro y buscar la nueva ubicación de los servicios disponibles para ejecutarlos.

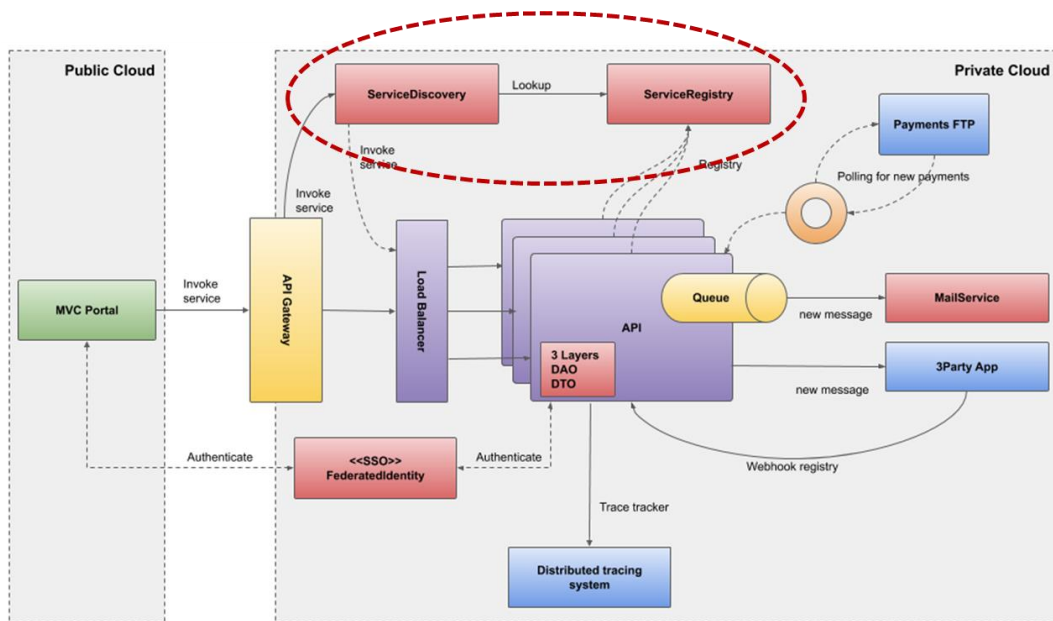


Fig 82 - Service Registry & Service Discovery.

La ventaja de esto es que no necesitamos configurar en cada servicio la ubicación de los demás, sino más bien, solo le decimos donde registrarse y donde recuperar el registro de servicios disponibles, los cuales se actualizan a medida que más servicios se agregan a la red o se desconectan.

Balanceo de cargas

El balanceo de cargas nos permitirá atender una demanda creciente de solicitudes a medida que nuevas instancias de un mismo componente se levanta y se registran en el registro de servicios mencionado anteriormente, de esta forma, a podemos ejecutar un mismo servicio en más de un servidor y permitir que la carga de trabajo se divida entre el total de instancias registradas:

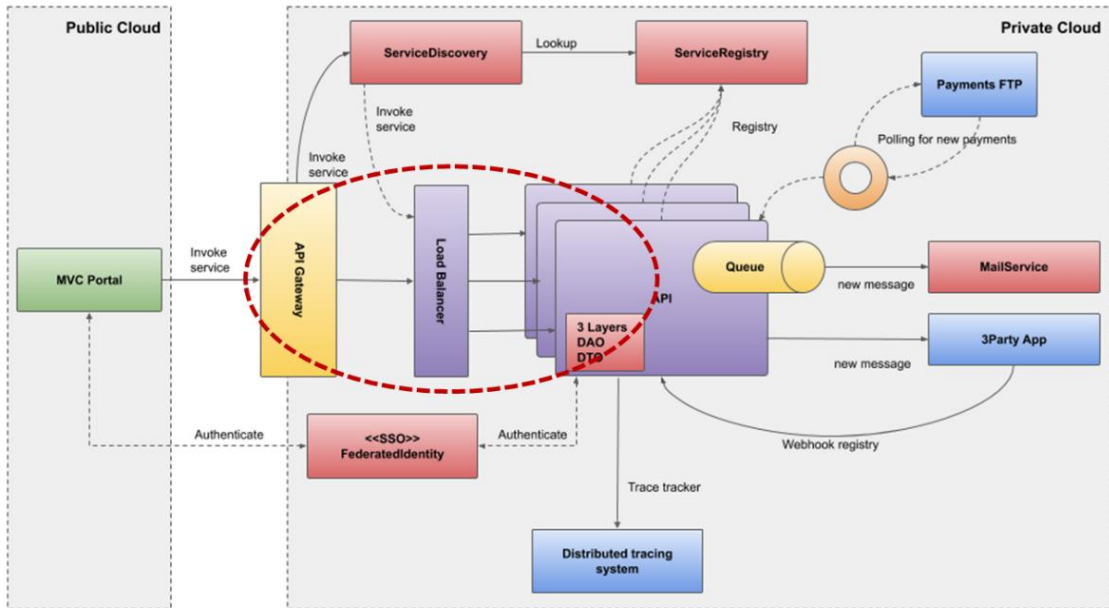


Fig 83 - Balanceo de cargas.

Una de las ventajas del autodescubrimiento de los servicios es que podemos agregar nuevas instancias que serán detectadas por el balanceador de cargas, lo que permite agregar o quitar instancias dinámicamente sin necesidad de reiniciar los servicios o cambiar la configuración para agregar el nuevo servidor al balanceador.

Notificaciones distribuidas y persistentes

Una de las funcionalidades del E-Commerce es que envía correos electrónicos a los clientes una vez que su orden es procesada, con la intención de hacer saber al cliente que hemos recibido su pedido y que ya lo estamos trabajando.

Lo más obvio es configurar el envío de correos electrónicos directamente sobre el componente que procesa el pedido, de esta forma procesamos el pedido y enviamos el correo, sin embargo, hemos detectado que el envío de correos es un proceso repetitivo que puede que varios de nuestros componentes requieran en el futuro, por lo que lo más inteligente es separarlo y administrarlo como un Microservicio independiente que se encargue únicamente del envío de los correos. Esta idea tiene dos ventajas, crear un componente reutilizable y que la configuración del correo solo se administre en un punto y no en cada componente donde sea necesario mandar un correo electrónico.

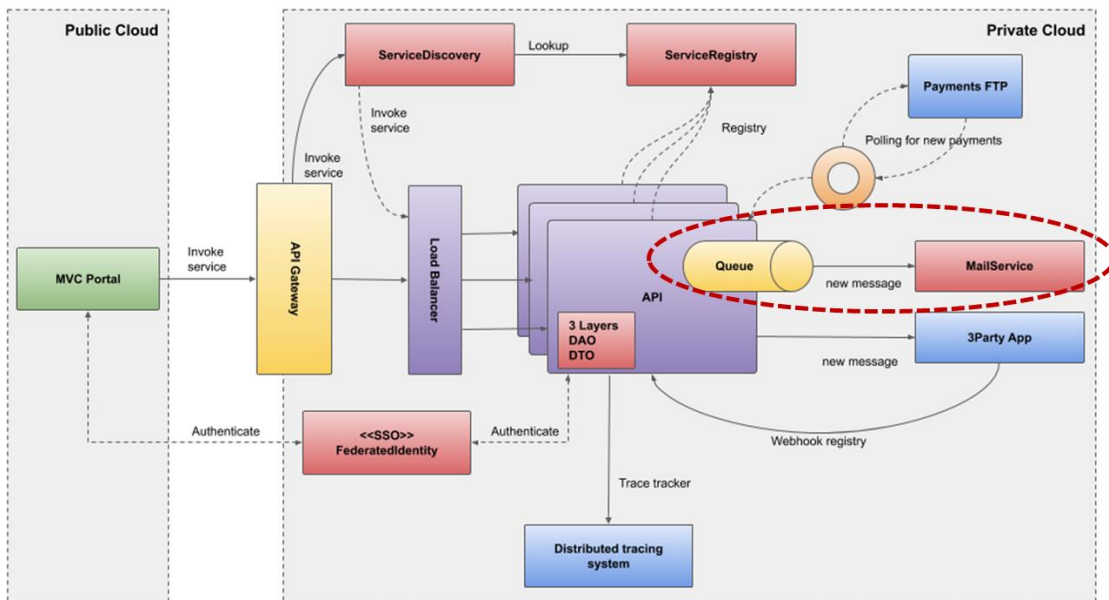


Fig 84 - Store and forward.

Sin embargo, este enfoque tiene un problema, y es que al depender de un servicios externo, estamos expuestos a que este no esté disponible al momento de querer enviar el correo, lo que podría provocar el fallo de todo el proceso o que el cliente no reciba su correo, sea cual sea el caso, tenemos un problema, por ello, implementamos una cola de mensajes intermedio que permite persistir las solicitudes de correo, para que de esta forma, el componente pueda atender los

mensajes cuando esté disponible y cuando no le esté, se puedan ir almacenando y ser atendidos cuando el servicio se restablezca, de esta forma, no perdemos ningún correo y el proceso de procesamiento de pedidos puede funcionar sin importarles si el mail ha sido enviado o no.

Notificaciones proactivas

Hoy en día las aplicaciones requieren integrarse con otras, ya sea que consultemos o enviemos datos, es necesario implementar mecanismos que permita que la información viaje de un punto a otro, y un caso específico son los eventos que se producen en las aplicaciones. En nuestro caso, tenemos la necesidad de notificar a otras aplicaciones cuando una nueva orden sea creada. En realidad, no nos interesa que hagan con la notificación, pero es importante hacer llegar esa notificación, porque puede que existe un sistema que se encargue de la facturación, otro que se encargue enviar los productos al cliente, etc. Nuestro trabajo es enviar las notificaciones a todos los interesados, para ello, hemos implementado un Microservicios que notifica los eventos de la aplicación a todos los que se registren.

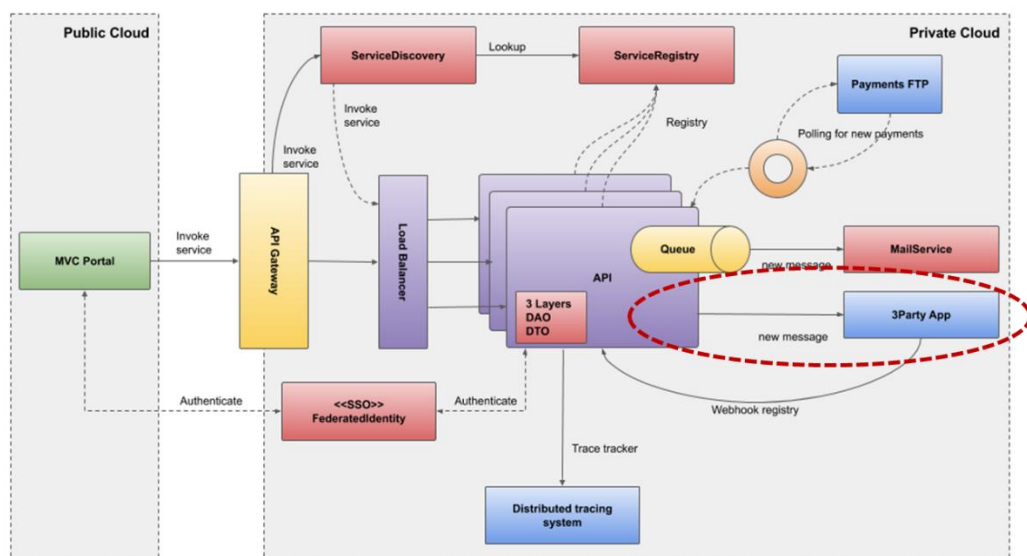


Fig 85 – Notificaciones mediante un Webhook.

Este enfoque permite evitar que los clientes realicen consultas repetitivas en búsqueda de nueva información, las cuales van degradando el performance gradualmente, por lo que en lugar de que ellos nos consulten, nosotros le hacemos llegar las notificaciones casi en tiempo real.

Sondeo de pagos

La aplicación de E-Commerce soporta dos formas de pago, mediante tarjeta de crédito y depósito bancario. El pago por tarjeta es el más simple, pues solicitamos los datos de la tarjeta, realizamos el cargo y procesamos el pedido, sin embargo, el segundo escenario, es cuando el cliente no cuenta con una tarjeta y le damos la opción de que realice su pago directamente en la ventanilla de su banco, para ello, les generamos un número de referencia el cual tendrá que proporcionar al momento del pago.

Este método de pagos nos trae otro problema, y es como le hacemos para detectar los pagos, por suerte, el banco nos envía un archivo de texto plano con los pagos realizados por los clientes, el cual tiene dos valores separados por comas (CSV), los cuales corresponde al número de referencia y el monto pagado.

Debido a esto, hemos creado un componente que se encarga únicamente al sondeo de estos archivos y aplicarlos en el Microservicio del E-Commerce (API). Debido a que el banco puede poner el archivo en cualquier momento, tenemos que diseñar un componente que esté preguntando constantemente por nuevos archivos en un servidor FTP.

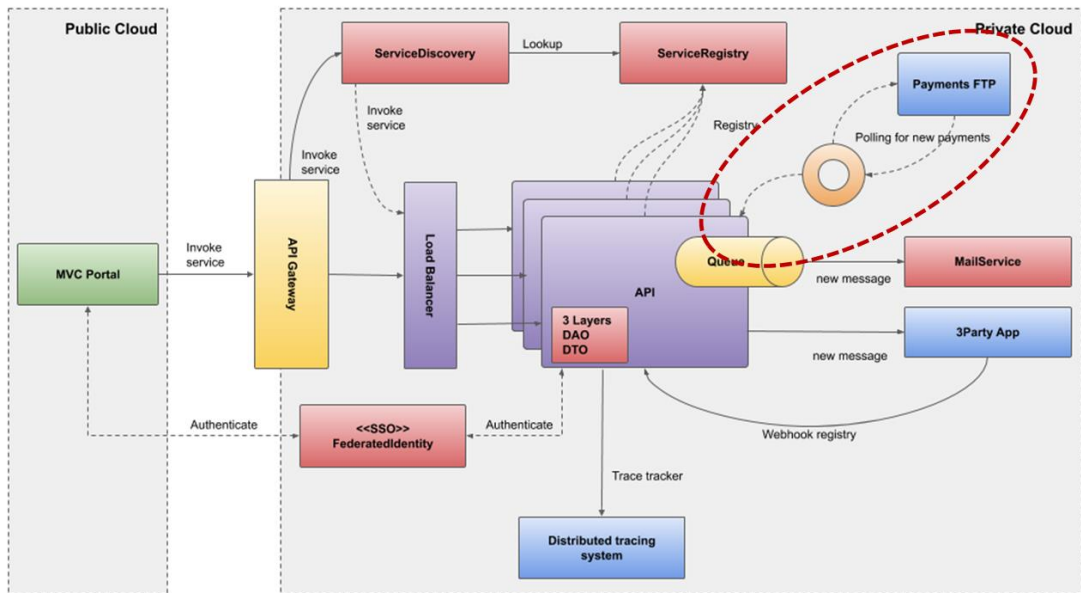


Fig 86 - File pooling.

Traza distribuida

Uno de los principales problemas en una arquitectura distribuida es la trazabilidad de la ejecución de un proceso, pues una sola llamada puede repercutir en la llamada de varios servicios, lo que implica que tengamos que ir recuperando el log por partes, es decir, tenemos que sacar cada parte del log en cada microservicio, y luego unirlos, lo cual puede ser una tarea titánica y complicado, por este motivo, implementaremos un sistema de traza distribuido que permite unificar los logs en un solo punto y agruparlos por ejecución:

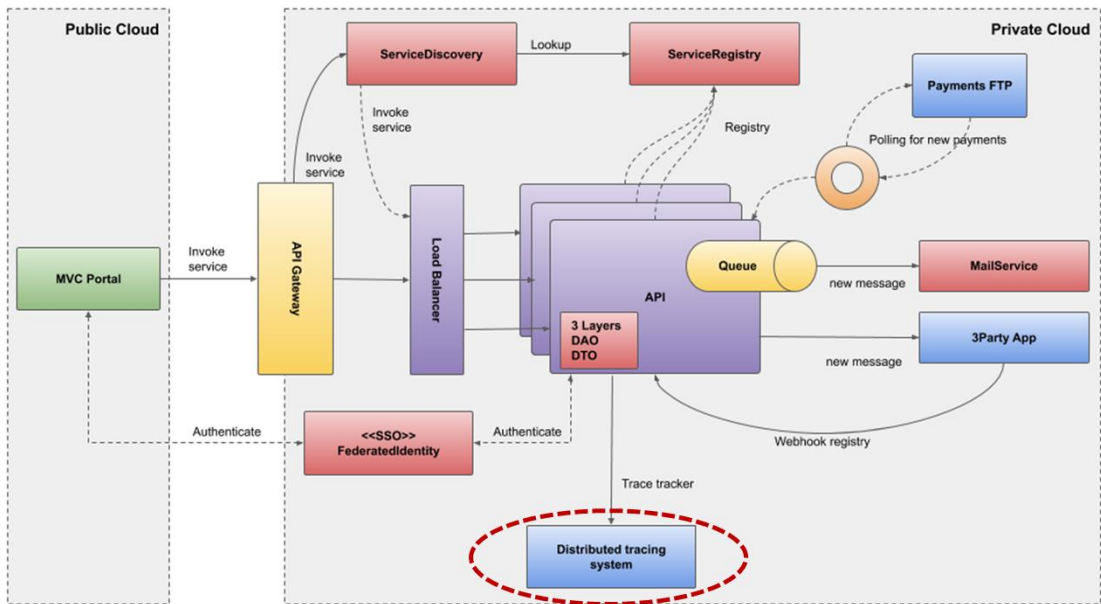


Fig 87 - Distributed tracing system

Interface gráfica de usuario

Para la interface gráfica de usuario utilizaremos React, una librería desarrollada por Facebook basada en Javascript que permite crear interfaces gráficas de usuario super dinámicas. Si bien React no es una librería propiamente MVC (Modelo Vista Controlador) sí que tiene algunos elementos de este patrón que nos permitirán explicar el patrón con este ejemplo:

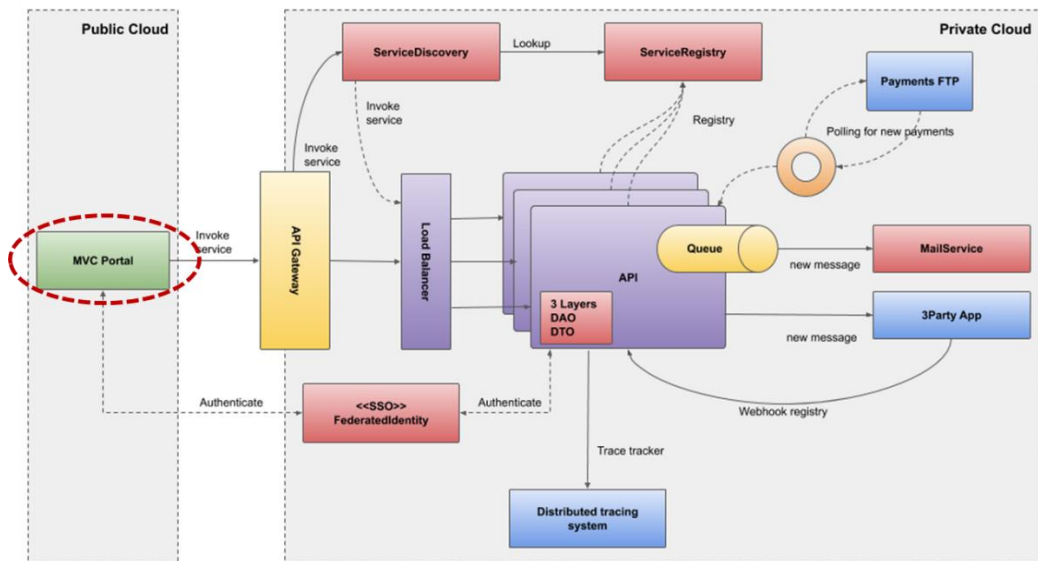


Fig 88 - Aplicación E-Commerce.

Instalación

El objetivo de esta sección es instalar por completo la aplicación y dejarla funcionando para que puedas ir analizando su funcionamiento a medida que explicamos los diferentes patrones arquitectónicos.

Antes de comenzar quiero avisarte que la instalación y ejecución de la aplicación puede ser algo tediosa la primera vez, no porque la aplicación este mal diseñada, si no que hemos creado una aplicación basada en un estilo de Microservicios, lo que nos obliga a crear varias piezas de software independientes y totalmente desacopladas, lo que puede dar la sensación de que estamos instalando demasiadas cosas, pero verás que cuando comencemos a explicar el porqué de todas estas cosas quedarás muy contento con el resultado y abras aprendido muchas cosas que podrás implementar en tus futuros proyectos, así que si más, comencemos con la instalación.

A modo de resumen, tendremos que instalar lo siguiente:

- **Descargar código fuente:** Descargaremos el código fuente de la aplicación desde Github.
- **Java Developer Kit (JDK8):** Incluye el compilador y entorno de ejecución para Java
- **Spring Tool Suite:** IDE con el que analizaremos y ejecutaremos el código.
- **NodeJS & NPM:** Entorno de ejecución de Javascript del lado del servidor y gestor de paquetes (necesario para React)
- **Visual Studio Code:** El editor de código más famoso para desarrollo WEB.
- **MySQL:** La base de datos Open Source más popular.



Nota importante acerca de la instalación

Debido a que las páginas de las diversas herramientas cambian con frecuencia, puede que algunas página o links que aquí se mencionan cambien o ya no estén disponibles. Por lo que confió en que podrás sobrellevar este tipo de problemas. (de igual forma podrás reportarnos cualquier inconveniente)

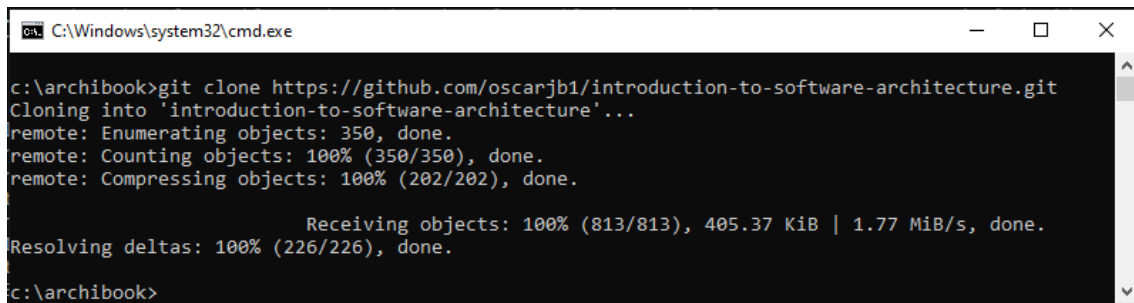
Descargar código fuente

El código fuente de la toda la aplicación lo podemos descargar desde Github, por lo que existen dos formas, clonar el repositorio con un cliente Git o descargarlo como un archivo comprimido.

Descargar el repositorio con un cliente Git

Puedes descargar el instalador de Git desde su página oficial (<https://git-scm.com/downloads>). Una vez instalado, abrimos la terminal y nos ubicamos en la carpeta en la que queremos descargar el código fuente, una vez allí, ejecutamos el comando:

```
> git clone https://github.com/oscarjb1/introduction-to-software-architecture.git
```



```
C:\Windows\system32\cmd.exe

c:\archibook>git clone https://github.com/oscarjb1/introduction-to-software-architecture.git
Cloning into 'introduction-to-software-architecture'...
remote: Enumerating objects: 350, done.
remote: Counting objects: 100% (350/350), done.
remote: Compressing objects: 100% (202/202), done.

Receiving objects: 100% (813/813), 405.37 KiB | 1.77 MiB/s, done.
Resolving deltas: 100% (226/226), done.
c:\archibook>
```

Fig 89 - Clonando el repositorio de GitHub.

Al terminar, navegamos con el explorador de archivos a la ubicación donde clonamos el repositorio y podremos ver todos los proyectos.

Descargar el repositorio como un archivo comprimido

Esta segunda forma nos permite descargar el repositorio sin necesidad de un cliente Git y obtendremos exactamente el mismo resultado que descargarlo con un cliente Git.

En este caso podemos descargar el código como un archivo ZIP descargándolo desde la siguiente URL:

<https://github.com/oscarjb1/introduction-to-software-architecture/archive/master.zip>

La descarga comenzará de inmediato una vez que pongamos la URL en el navegador, y ya solo tendremos que descomprimir el ZIP.

Después de descargar el código

Sea cual sea el método para descargar el código, el resultado será el mismo, por lo que al final deberemos de poder ver los proyectos como se muestra en la siguiente imagen:

Nombre	Fecha de modificación	Tipo
.git	25/10/2019 04:55 p. m.	Carpeta de archivos
3pary-app	25/10/2019 04:55 p. m.	Carpeta de archivos
api-gateway	25/10/2019 04:55 p. m.	Carpeta de archivos
commons	25/10/2019 04:55 p. m.	Carpeta de archivos
crm-api	25/10/2019 04:55 p. m.	Carpeta de archivos
ecommerce-app	25/10/2019 04:55 p. m.	Carpeta de archivos
ftp-payment-pooling	25/10/2019 04:55 p. m.	Carpeta de archivos
mail-sender	25/10/2019 04:55 p. m.	Carpeta de archivos
RemoteSystemsTempFiles	25/10/2019 04:55 p. m.	Carpeta de archivos
security	25/10/2019 04:55 p. m.	Carpeta de archivos
service-registry	25/10/2019 04:55 p. m.	Carpeta de archivos
sql	25/10/2019 04:55 p. m.	Carpeta de archivos
webhook-notif	25/10/2019 04:55 p. m.	Carpeta de archivos
.gitignore	25/10/2019 04:55 p. m.	Documento de te...

Fig 90 - Validando la descarga del repositorio.

Por el momento solo hay que validar que la carpeta se ve como en la imagen y no hace falta preocuparnos por el contenido de cada una de las carpetas, ya que a medida que avancemos en el libro vamos a ir explicando su contenido e iremos comprendiendo cómo funciona la aplicación de forma incremental.

Instalando MySQL

El siguiente paso será instalar la base de datos MySQL, la cual la utilizaremos para persistir toda la información del API.

Vamos a instalar la versión Open Source de MySQL o también llamada Community Edition, la cual no tiene un costo de licencia y podemos descargar libremente, para ello, nos dirigimos a la página oficial (<https://dev.mysql.com/downloads/mysql/>) y descargamos la versión 8:

General Availability (GA) Releases

MySQL Community Server 8.0.18

Select Operating System:
Microsoft Windows

Looking for previous GA versions?

Recommended Download:

MySQL Installer for Windows

All MySQL Products. For All Windows Platforms. In One Package.

Starting with MySQL 5.6 the MySQL Installer package replaces the standalone MSI packages.

Windows (x86, 32 & 64-bit), MySQL Installer MSI

[Go to Download Page >](#)

Other Downloads:

Windows (x86, 64-bit), ZIP Archive (mysql-8.0.18-winx64.zip)	8.0.18	272.3M	Download
MD5: 3c1fc0bc3368639d968f8e5bf8afa23d Signature			
Windows (x86, 64-bit), ZIP Archive Debug Binaries & Test Suite (mysql-8.0.18-winx64-debug-test.zip)	8.0.18	402.6M	Download
MD5: 8d56a0f2418d86598495b411dc29d3b9 Signature			

We suggest that you use the [MD5 checksums](#) and [GnuPG signatures](#) to verify the integrity of the packages you download.

Fig 91 - Seleccionando la versión adecuada de MySQL.

MySQL también se puede instalar en Windows, Mac y Linux, por lo que no deberías de tener problemas para instalarlo. Selecciona la versión que se adapte a tu sistema operativo y comenzamos la descarga. Seguramente nos pedirá que nos

autentiquemos antes de comenzar la descarga, pero lo podemos ignorar con pequeño link bastante escondido en la parte inferior que dice *"No thanks, just start my download"*.

A diferencia de las instalaciones anteriores, con MySQL es importante realizar los pasos con cuidado, pues tendremos que definir el puerto de la base de datos y la contraseña del usuario root, así que mi sugerencia es que mantengas el puerto por default que es 3306 y establezcas una contraseña simple y que puedas recordar para el usuario root, ya que más adelante necesitaremos estos datos para configurar el API.

Dependiendo del método de instalación, nos podrá incluir o no el Workbench, el cual es una aplicación cliente para conectarnos a la base de datos y probar la conexión, por lo que si no se instaló, lo podemos descargarlo de forma independiente en <https://www.mysql.com/products/workbench/>.

Una vez que hemos instalado la base de datos y el Workbench, procederemos a comprobar que nos podamos conectar, para ello, tendremos que abrir el cliente Workbench y crear una nueva conexión:

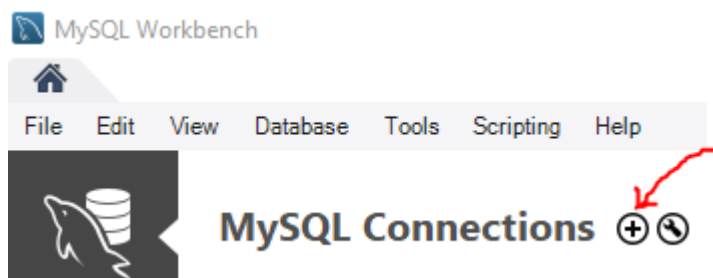


Fig 92 - Crear una nueva conexión a MySQL.

Al dar click en el botón + nos saldrá una nueva venta en la que tendremos que configurar los datos de conexión a la base de datos:

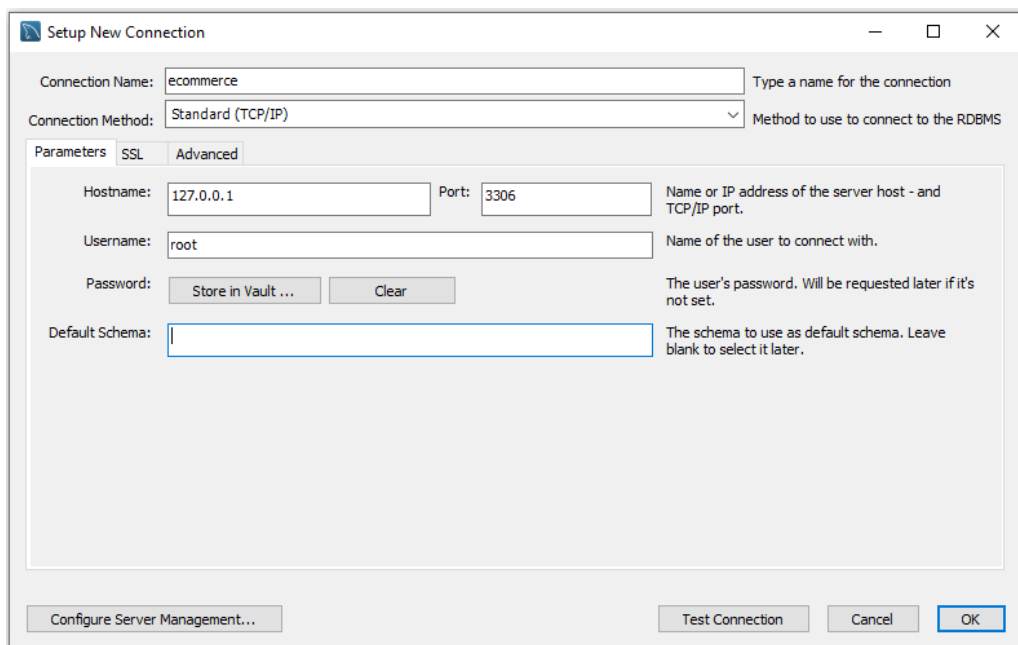


Fig 93 - Estableciendo los datos de conexión a MySQL.

Definimos “ecommerce” como nombre la conexión, y dejamos los demás campos tal y como está precargados, a menos que hubiéramos cambiado algún dato durante la instalación de MySQL. Una vez capturados los datos presionamos “Test Connection” para comprobar la instalación. Si todo sale bien veremos el siguiente mensaje:

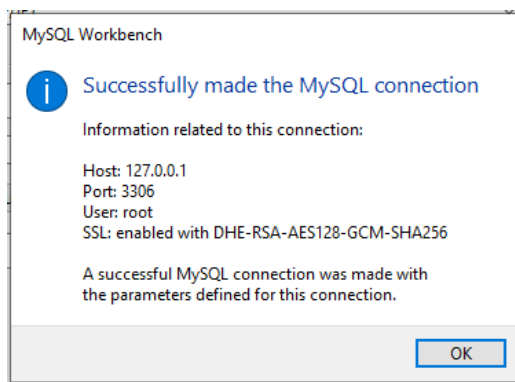


Fig 94 - Conexión exitosa a MySQL.

En la siguiente sección continuaremos hablando de cómo crear las bases de datos para nuestra aplicación.

Preparando la base de datos

Ya con la instalación y conexión exitosa, procederemos a crear las bases de datos e insertar los datos iniciales para la aplicación. Para ello, regresaremos a la carpeta donde descargamos el proyecto de GitHub y nos abriremos la carpeta llamada *sql*, la cual tendrá el script *ecommerce.sql*.

Nombre	Fecha de modificación	Tipo
.git	25/10/2019 05:17 p. m.	Carpeta de archivos
.metadata	22/10/2019 05:12 p. m.	Carpeta de archivos
.recommenders	15/10/2019 02:09 p. m.	Carpeta de archivos
3pary-app	10/09/2019 11:44 a. m.	Carpeta de archivos
api-gateway	10/09/2019 11:44 a. m.	Carpeta de archivos
commons	10/09/2019 11:44 a. m.	Carpeta de archivos
crm-api	21/10/2019 12:49 p. m.	Carpeta de archivos
ecommerce-app	10/09/2019 11:49 a. m.	Carpeta de archivos
ftp-payment-pooling	15/10/2019 07:06 p. m.	Carpeta de archivos
mail-sender	10/09/2019 11:49 a. m.	Carpeta de archivos
security	10/09/2019 11:56 a. m.	Carpeta de archivos
service-registry	10/09/2019 11:56 a. m.	Carpeta de archivos
sql	25/10/2019 01:17 p. m.	Carpeta de archivos
webhook-notif	10/09/2019 11:58 a. m.	Carpeta de archivos
.gitignore	17/10/2019 10:11 a. m.	Documento de te...
.project	25/10/2019 05:50 p. m.	Archivo PROJECT
zipkin-server-2.18.0-exec.jar	16/10/2019 04:44 p. m.	Executable Jar File

Fig 95 - Abrir la carpeta *sql*.

Abrimos el script con Workbench y lo ejecutamos con el icono del rayo:

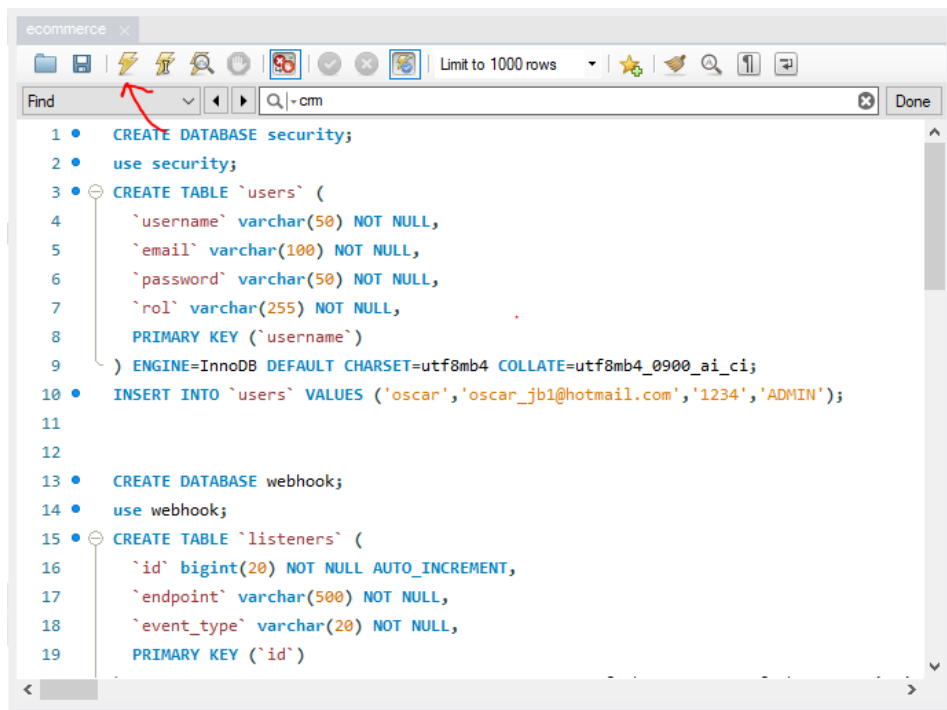


Fig 96 - Creando las bases de datos.

Como resultado de la ejecución del Script habremos creados 3 bases de datos, llamadas crm, security y webhook con sus respectivas tablas, las cuales podemos ver en el navegador del lado izquierdo tras darle actualizar.

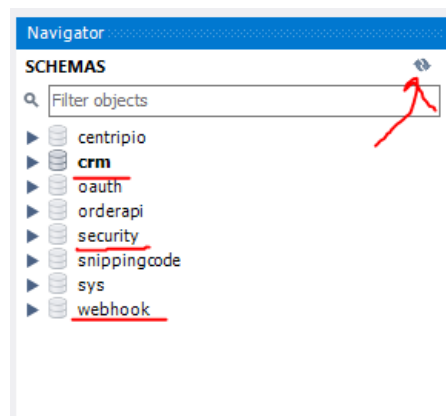


Fig 97 - Comprobando la DB creadas.

No nos preocupemos por el momento por analizar el contenido de las bases de datos, ya que más adelante regresaremos a ellas para analizarlas con detalle.

Instalando el JDK

Ahora procederemos con la instalación del JDK en su versión 8. Debido a que las versiones posteriores han pasado a ser licenciadas por Oracle y tiene un costo, nos conformaremos con esta versión, la cual tiene todo lo que necesitamos para trabajar, sin embargo, podrías utilizar cualquier versión superior o incluso la versión del OpenJDK, el cualquiera de los casos, tendría que ser una versión 8 u superior.

Para instalar la versión de Oracle Java nos iremos a la siguiente página <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> y aceptaremos los términos de licencia y seleccionamos la versión más adecuada para nuestro sistema operativo.

En nuestro caso seleccionamos la versión para Windows x64, al seleccionarla nos pedirá que nos autentiquemos en Oracle, así que, si tienes una cuenta, simplemente colocamos nuestro usuarios y listo, en otro caso, tendremos que crear una nueva cuenta. Una vez autenticados, la descarga comenzará de forma inmediata.

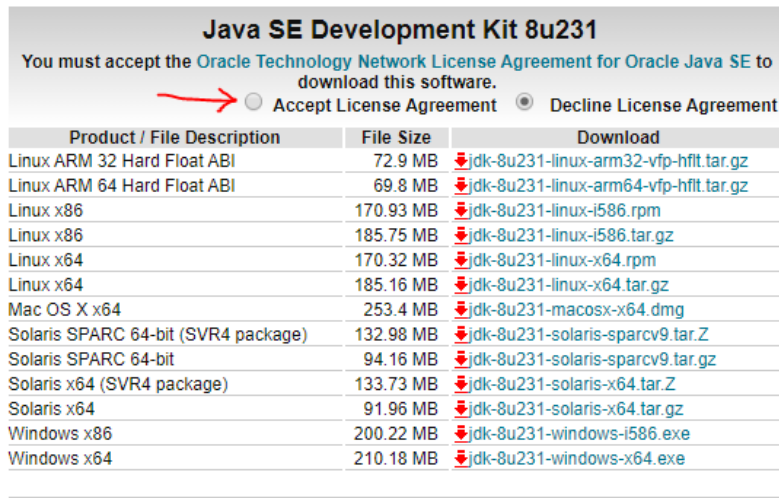


Fig 98 - Selección del JDK

En el caso de Windows o Mac la instalación deberá ser muy simple, prácticamente abrir el instalador y seguir el wizard, así que no entraremos mucho en los detalles. En el caso de Linux puede ser más complicado, ya que cambia ligeramente según la distribución que estés utilizando, así que si tiene problemas con la instalación, dejo la liga a la documentación: https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Si la instalación fue exitosa, deberás de poder ver la versión de Java instalada al ejecutar el comando "java -version" en la terminal:

```

C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 10.0.18362.418]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\oscar>java -version
java version "1.8.0_221"
Java(TM) SE Runtime Environment (build 1.8.0_221-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.221-b11, mixed mode)

C:\Users\oscar>

```

Fig 99 - Comprobando la instalación del JDK.

Si vemos algo parecido a lo anterior quiere decir que hemos hecho todo bien, en otro caso, tendrás que revisar la instalación hasta poder ver un mensaje como el anterior.

Instalando NodeJS & NPM

NodeJS quizás sea la herramienta menos conocida en este punto, pero la necesitaremos para correr el FrontEnd, ya que crearemos un servidor JavaScript que nos permite servir la aplicación web en el navegador.

NodeJS & NPM los podemos descargar desde <https://nodejs.org/es/download/>, allí seleccionaremos la versión que se adapte mejor a nuestro sistema operativo:

Descargas

Versión actual: 12.13.0 (includes npm 6.12.0)

Descargue el código fuente de Node.js o un instalador pre-compilado para su plataforma, y comience a desarrollar hoy.

LTS
Recomendado para la mayoría

Actual
Últimas características


Windows Installer
node-v12.13.0-x64.msi


macOS Installer
node-v12.13.0.pkg


Source Code
node-v12.13.0.tar.gz

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

Source Code

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
64-bit	
ARMv7	ARMv8
node-v12.13.0.tar.gz	

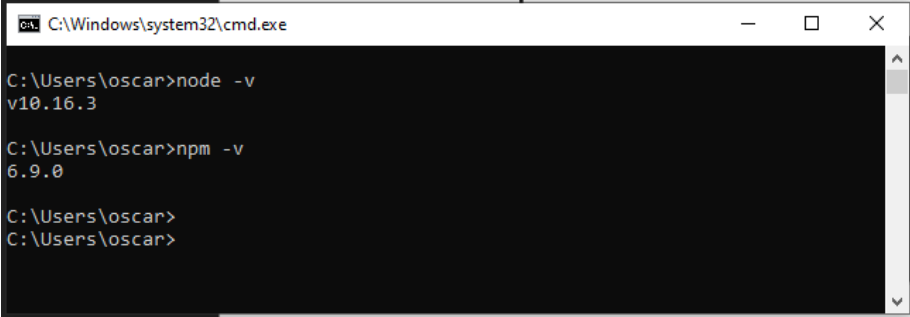
Plataformas adicionales

Fig 100 - Seleccionando la versión de NodeJS.

Recomendamos seleccionar siempre la versión LTS ya que es la más estable y la que se recomienda para producción, ya que la versión “Actual” tiene todavía componentes en Beta o que no han probados del todo en ambientes productivos.

La descarga comenzará de forma inmediata una vez que seleccionemos la versión, el cual deberá ser un archivo instalable en el caso de Windos o Mac, así que simplemente lo ejecutamos y seguimos las instrucciones. En el caso de Linux puede ser un poco más complicado según la distribución de Linux (Tendrás que consultar la documentación).

Una vez finaliza la instalación deberemos poder comprobar si la instalación fue exitosa ejecutando el comando “`node -v`” y “`npm -v`” desde nuestra terminal:



```
C:\Windows\system32\cmd.exe

C:\Users\oscar>node -v
v10.16.3

C:\Users\oscar>npm -v
6.9.0

C:\Users\oscar>
C:\Users\oscar>
```

Fig 101 - Comprobando la instalación de NodeJS.

El primer comando comprueba la instalación de NodeJS, mientras que el segundo comprueba la versión instalada de NPM. Si el resultado es como el de la imagen anterior, quiere decir que la instalación fue exitosa.



Instalación en Linux

En Linux es posible que se tenga que instalar NodeJS y NPM por separado, pero dependerá de la distribución y el método utilizado para la instalación, es por ello que podemos tener instalado NodeJS, pero que no nos detecte NPM.

Instalando Visual Studio Code

Visual Estudio Code es el editor de código más popular de la actualidad para el desarrollo web y que utilizaremos para analizar y ejecutar la aplicación web desarrollada en React.



Importante

Visual Studio Code es diferente a Visual Studio, el primero es un editor de código minimalista y ligero desarrollado por Microsoft y es Open Source multipropósito, pero fuertemente orientado a tecnologías web, mientras que el segundo es un IDE licenciado orientado más que nada a desarrollos Microsoft.

Para descargarlo nos tendremos que ir a su página oficial <https://code.visualstudio.com/> y lo primero que nos saldrá será la opción de descargar, por suerte, Visual Studio Code funciona en Windows, Mac y Linux, así que no tendremos problemas es instalarlo.

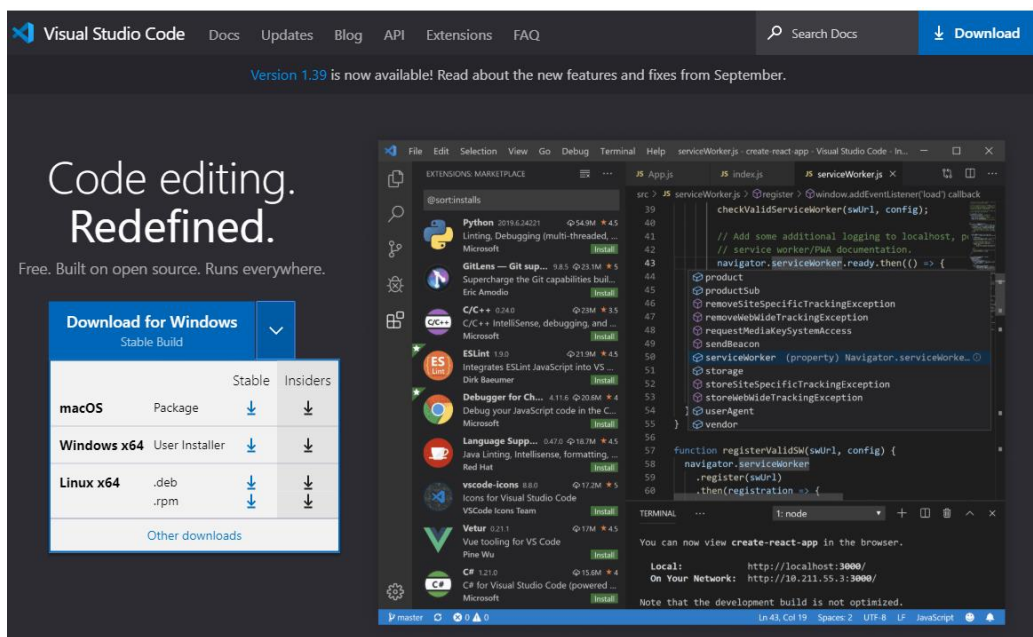


Fig 102 - Seleccionando la versión de Visual Studio Code.

La instalación no debería de ser un problema, ya que es solo un Wizard con los típicos pasos de aceptar la licencia, siguiente, siguiente y finalizar, así que no nos detendremos en explicar cómo instalarlo.

Una vez terminada la instalación se nos creará un acceso directo en el escritorio o lo podemos buscar en los programas instalados, lo ejecutamos y nos abrirá una pantalla como la siguiente:

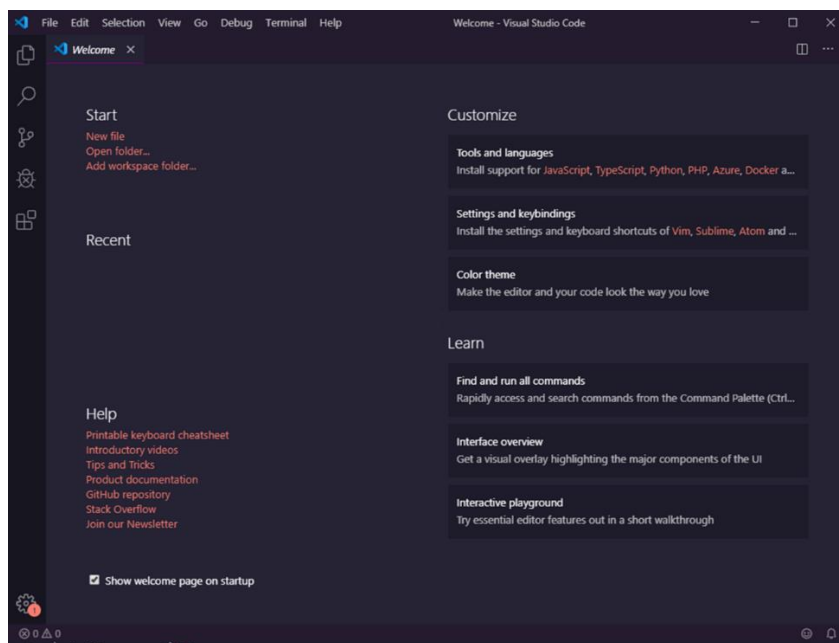


Fig 103 - Visual Studio Code.

Desde Visual Studio Code vamos a dirigirnos a la opción *File* y luego seleccionamos *Open FoLder* para abrir la carpeta *ecommerce-app* que se encuentra en la carpeta donde descargamos el código de GitHub. Una vez seleccionado el proyecto se abrirá y veremos algo similar a esto:

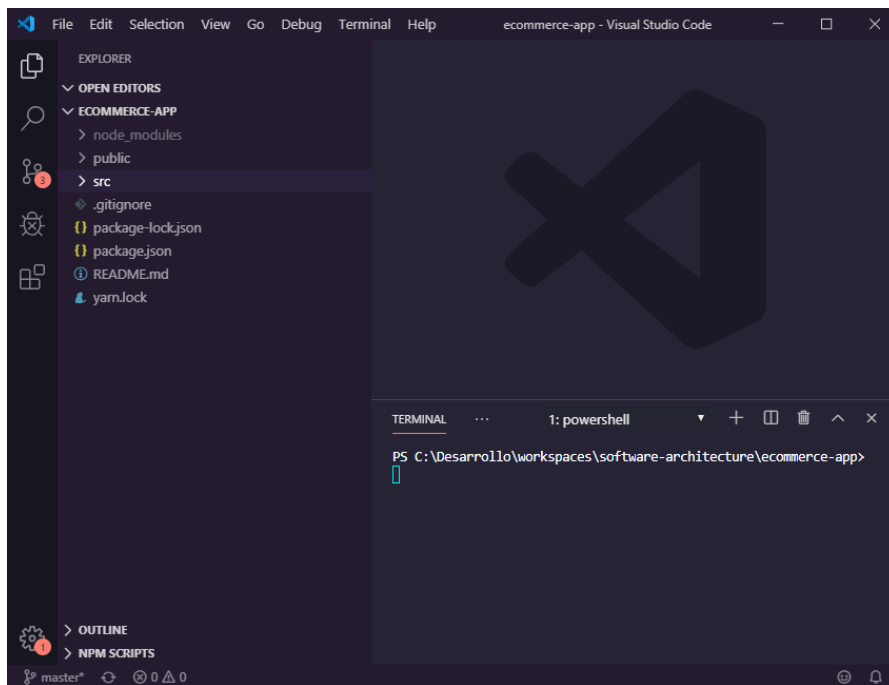


Fig 104 - Estructura del proyecto Ecommerce.

Vamos a dejar abierto Visual Studio Code y regresaremos al más tarde para ejecutar la aplicación.

Instalando Spring Tool Suite (STS)

Para instalar Spring Tool Suite nos iremos a la página oficial (<https://spring.io/tools>) y descargaremos la versión correspondiente a nuestro sistema operativo. Por suerte, existe una versión para Windows, Mac y Linux:

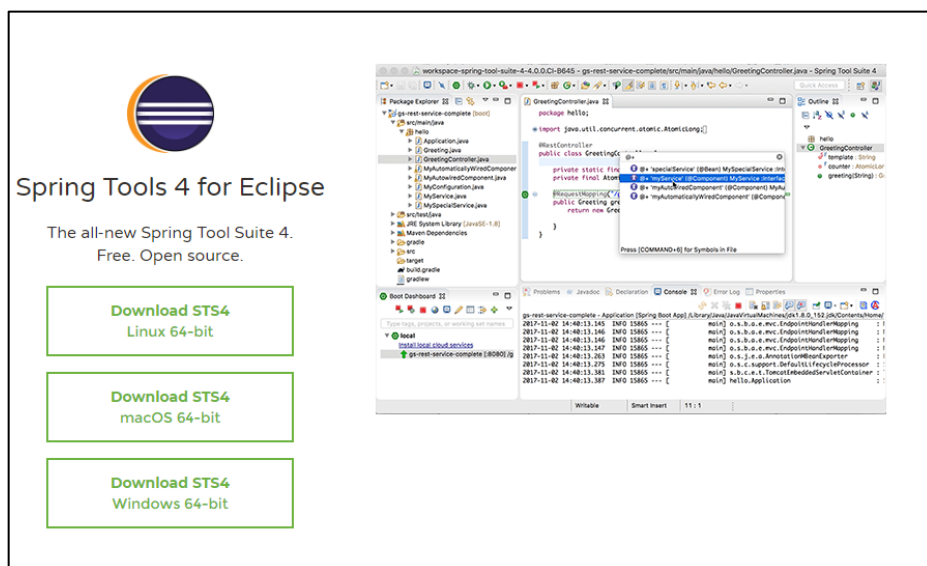


Fig 105 - Seleccionando la versión adecuada de STS

Una vez que seleccionamos el sistema operativo, la descarga iniciará automáticamente. Lo descomprimos y estaremos listo para ejecutar el IDE, ya que no requiere instalación. Para ejecutarlo nos ubicamos en la carpeta donde lo descomprimos y nos ubicamos en la carpeta `<STS_HOME>/sts-bundle/sts-<VERSION>.RELEASE` y ejecutamos el archivo `STS.exe` en Windos, o su equivalente en otro sistema operativo.

Al iniciar nos pedirá que seleccionemos una carpeta en donde se creará nuestra área de trabajo. Podemos seleccionar la que nos pone por default o seleccionar una personalizada. Finalmente presionamos *Launch* y el IDE iniciará.

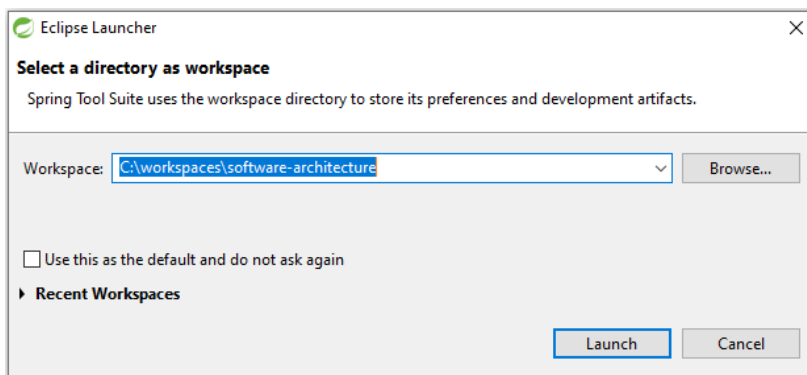


Fig 106 - Seleccionando el espacio de trabajo.

Una vez iniciado el IDE, vamos a ubicar la opción *File* del menú superior, y seleccionaremos la opción *Import*, lo que nos arrojará la siguiente pantalla:

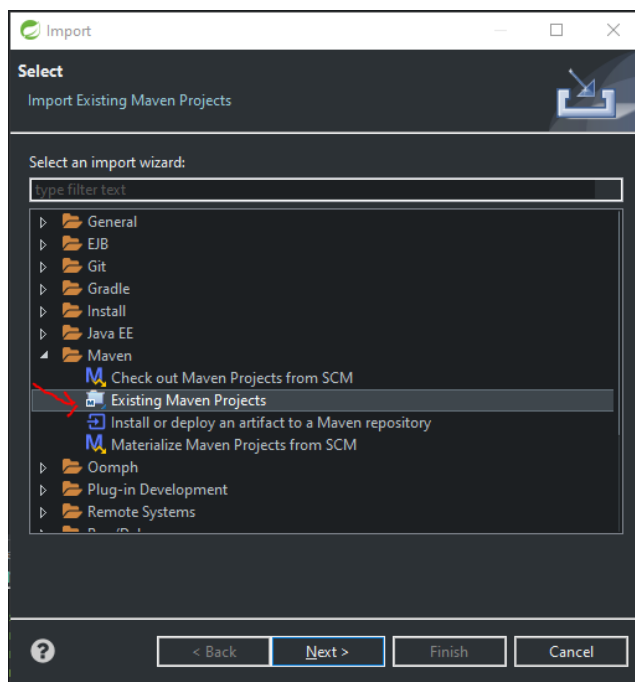


Fig 107 - Importando los proyectos de Spring boot parte 1.

En la pantalla anterior expandiremos la opción *Maven* y seleccionaremos *Existing Maven Projects*, finalmente presionamos *Next*.

En la siguiente pantalla nos posicionaremos en el campo *Root Directory* e introduciremos la dirección en donde descargamos el código fuente de GitHub y presionamos *Enter* para que inicie el proceso de escaneo de las carpetas. El IDE comenzará a analizar las carpetas en búsqueda de los proyectos, por lo que este proceso podría tomar un par de minutos. Tras finalizar, nos arrojará todos los proyectos que encontró, tal y como lo podemos ver en la siguiente imagen:

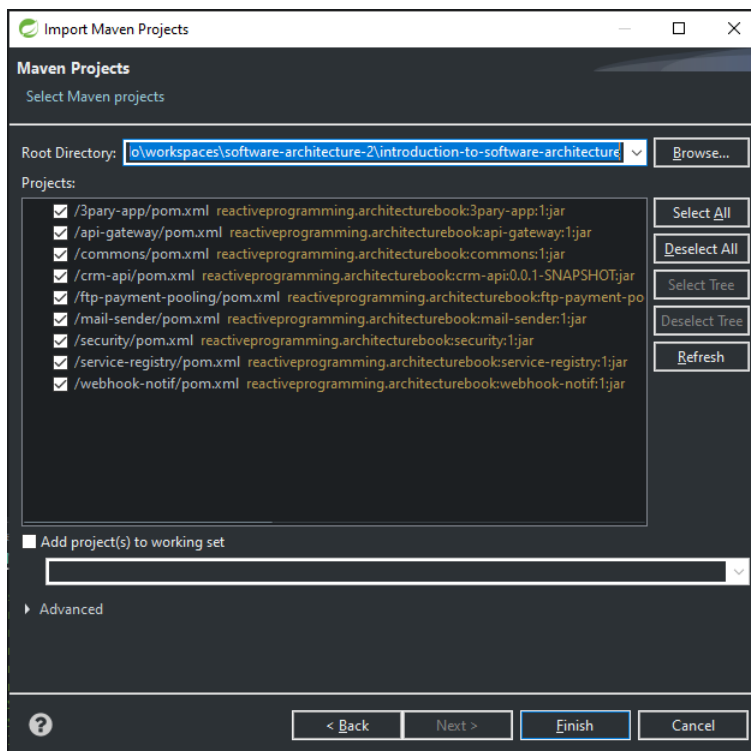


Fig 108 - Importando los proyectos

Nos aseguramos de seleccionar todos los proyectos y presionamos *Finish*. Finalmente veremos que todos los proyectos han sido importados a nuestra área de trabajo:

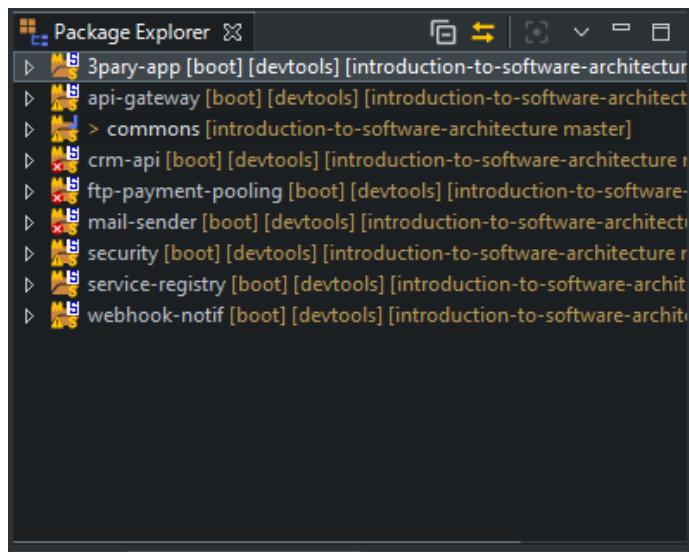


Fig 109 - Proyectos importados a nuestra área de trabajo.

Tras importar los proyectos el IDE comenzará a descargar todas las dependencias, por lo que podemos ver una barra de progreso en la parte inferior derecha. Este proceso puede tardar varios minutos dependiendo la velocidad de internet, por lo que hay que ser pacientes y esperar hasta que finalice.

Configurando RabbitMQ

Nuestra aplicación requiere de un *Message Broker* que permite intercambiar mensajes entre los diferentes componentes de una forma asíncrona, por lo que vamos a utilizar *RabbitMQ*, el *Message Broker* Open Source más popular del mercado.

RabbitMQ se presenta en dos versiones, la versión On-Premise, la cual podemos descargar e instalar en nuestros propios servidores, o la versión cloud que la podemos utilizar desde la nube sin una instalación y es totalmente administrada por ellos. Para no preocuparnos por la instalación, hemos decidido utilizar la versión cloud, la cual tiene un plan gratuito que nos servirá perfectamente para nuestra aplicación.

Para comenzar a utilizar *RabbitMQ* en su versión cloud será necesario ir a su página oficial (<https://www.cloudamqp.com/>) y crear una nueva cuenta. Puedes utilizar el login de Google para crearla más rápido o puedes llenar el formulario, cómo sea, una vez creada la cuenta, verás una página como la siguiente:

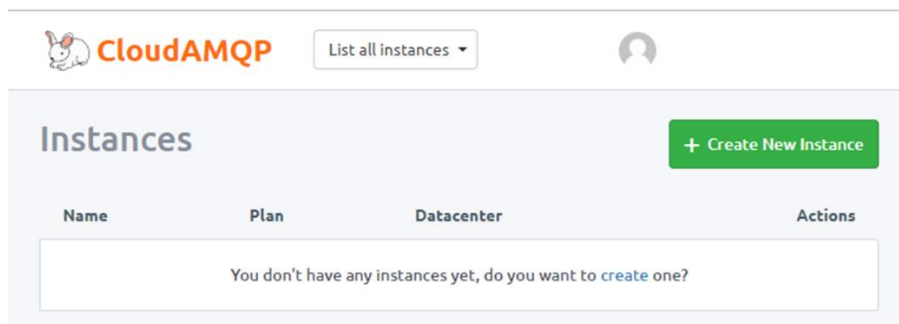


Fig 110 - Página de inicio de RabbitMQ.

Lo primero que haremos será dar click en el botón verde que dice *Create New Instance* para comenzar la configuración de *RabbitMQ*. Lo que nos llevará a la siguiente página:

Fig 111 - Configurando la instancia de RabbitMQ.

Como nombre de la instancia ponemos "Software Architecture pattern" y seleccionamos el plan *Little Lemur (Free)* y presionamos el botón verde "Select Region", lo que nos llevará a la siguiente página:

The screenshot shows the 'Select a data center and type' step (Step 2 of 4) in the AWS console. At the top, a progress bar indicates the current step is 'Region', with 'Plan' completed and 'Configure' and 'Confirm' pending. Below the progress bar, the title 'Select a data center and type - Step 2 of 4' is displayed. A 'Data center' dropdown menu is set to 'US-East-1 (Northern Virginia)'. The AWS logo is visible below the dropdown. At the bottom, there are three buttons: '« Back', 'Cancel', and a green 'Review' button.

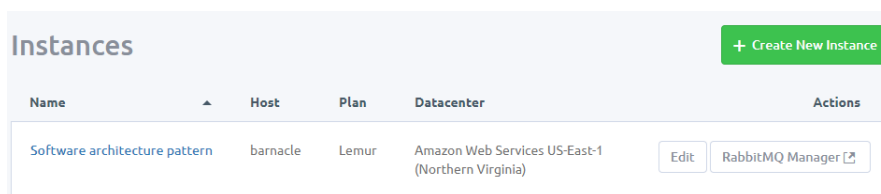
Fig 112 - Configurando la instancia de RabbitMQ.

Dejamos el Data center que viene por defecto y presionamos el botón verde "Review".

The screenshot shows the 'Confirm new instance' step (Step 4 of 4) in the AWS console. The progress bar at the top shows 'Plan', 'Region', and 'Configure' completed, with 'Confirm' pending. The title is 'Confirm new instance - Step 4 of 4'. A central box displays the 'Plan' details: a cartoon lemur, the name 'Little Lemur', and 'Total: \$0/month'. Below this, a table lists the instance details: Name (Software architecture pattern), Provider (Amazon Web Services), Region (US-East-1 (Northern Virginia)), Backend (RabbitMQ), and Tags. At the bottom, there are three buttons: '« Back', 'Cancel', and a green 'Create Instance' button with a checkmark.

Fig 113 - Fig 110 - Configurando la instancia de RabbitMQ.

Finalmente presionamos el botón verde *Create Instance*. Y la instancia estará creada y lista para comenzar a configurarla:



The screenshot shows the 'Instances' page in the AWS IAM console. At the top right is a green button labeled '+ Create New Instance'. Below it is a table with columns: Name, Host, Plan, Datacenter, and Actions. One instance is listed with the name 'Software architecture pattern', host 'barnacle', plan 'Lemur', and datacenter 'Amazon Web Services US-East-1 (Northern Virginia)'. The Actions column for this instance contains an 'Edit' button and a 'RabbitMQ Manager' link with an external icon.

Name	Host	Plan	Datacenter	Actions
Software architecture pattern	barnacle	Lemur	Amazon Web Services US-East-1 (Northern Virginia)	Edit RabbitMQ Manager

Fig 114 - Nueva instancia de RabbitMQ creada.

Para comenzar a configurar la instancia de RabbitMQ daremos click en el botón *RabbitMQ Manager*, lo que nos llevará a la siguiente página:

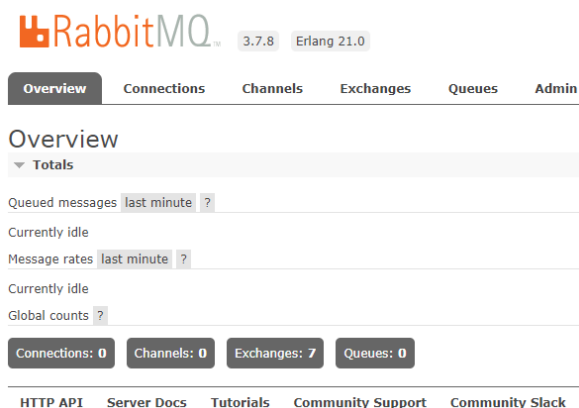


Fig 115 - Administrador de RabbitMQ.

En esta nueva página vamos a dar click en la opción *Queues* del menú superior:

The screenshot shows the RabbitMQ web interface. At the top, the RabbitMQ logo is displayed with version 3.7.8 and Erlang 21.0. Below the logo is a navigation bar with tabs: Overview, Connections, Channels, Exchanges, Queues (selected), and Admin. The main heading is 'Queues'. Below it, a dropdown menu shows 'All queues (0)'. A pagination section indicates 'Page 1 of 0 - Filter: []' with a 'Regex' checkbox. A message states '... no queues ...'. The 'Add a new queue' section contains a form with the following fields: 'Name' (containing 'emails'), 'Durability' (set to 'Durable'), 'Auto delete' (set to 'No'), and 'Arguments' (empty). Below these fields are several configuration options: 'Message TTL', 'Auto expire', 'Max length', 'Max length bytes', 'Overflow behaviour', 'Dead letter exchange', 'Dead letter routing key', 'Maximum priority', 'Lazy mode', and 'Master locator'. At the bottom of the form is a button labeled 'Add queue'. Red arrows point to the 'Name' field and the 'Add queue' button.

Fig 116 - Creando las colas de mensajes.

En esta nueva página vamos a crear dos *Queues*, por lo cual, vamos a poner el valor **emails** en el campo Name y presionaremos el botón **Add queue**. Repetimos el procedimiento para crear otra Queue llamada **newOrders**. Al finalizar habremos creados las dos Queues que podremos visualizar de la siguiente manera:

Queues

▼ All queues (2)

Pagination

Page **1** ▼ of 1 - Filter: ☐ Regex ?

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
emails	D HA	idle	0	0	0				
newOrders	D HA	idle	0	0	0				

Fig 117 - Visualizando las Queues creadas.

El siguiente paso será irnos a la opción *Exchanges* del menú superior:

 RabbitMQ 3.7.8 Erlang 21.0

Overview Connections Channels **Exchanges** Queues Admin

Exchanges

▼ All exchanges (9)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D	0.00/s		
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
emails	direct	D	0.00/s	0.00/s	
newOrders	direct	D	0.00/s	0.00/s	

→ ▼ Add a new exchange

→ Name: *

Type: direct ▼

Durability: Durable ▼

Auto delete: ? No ▼

Internal: ? No ▼

Arguments: = String ▼

Add Alternate exchange ?

→ Add exchange

Fig 118 - Creando nuestros Exchanges.

Una vez en la página de *Exchanges*, vamos a expandir la opción **Add a new exchange** y vamos a llenar el campo *Name* con el valor **emails** y presionamos **Add Exchange**, repetimos el proceso, pero esta vez pones **newOrders** en el campo *Name*. Al finalizar el proceso deberíamos de ver los dos *Exchanges* de la siguiente forma:

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
emails	direct	D			
newOrders	direct	D			

Fig 119 - Nuevos Exchanges creados.

De la tabla anterior, vamos a dar click en **emails** para configurar el Exchange:

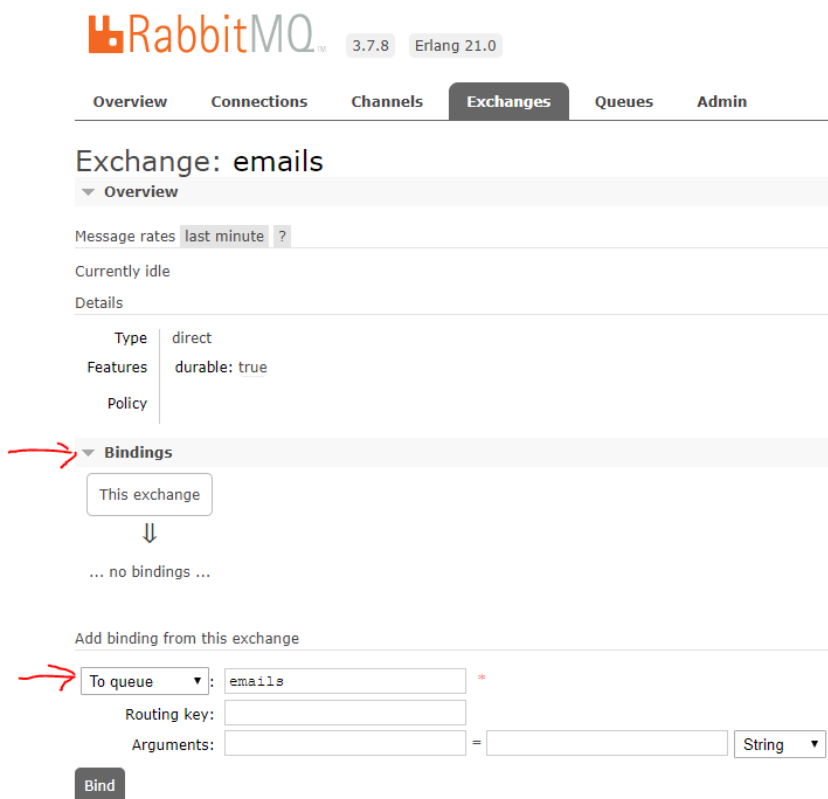


Fig 120 - Configure Exchanges.

Expandimos la sección Bindings y agregamos el valor **emails** en el campo que dice To queue y presionamos el botón **Bind**. Como resultado habremos vinculado la Queue emails con el Exchange emails:

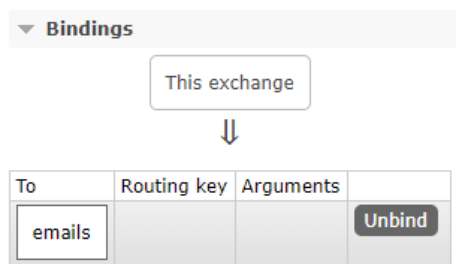


Fig 121 - Binding emails Exchange.

Vamos a repetir el proceso para configurar el *Exchange* **newOrders**, para lo cual regresaremos a la sección de los exchange (figura 117) y damos click en el Exchange **newOrders** y en el campo To que vamos a capturar el valor **newOrders**:

To	Routing key	Arguments
newOrders		Unbind

Fig 122 - Binding newOrders Exchange.

Ya para concluir con la configuración, vamos a regresar a la página de bienvenida de Rabbit, donde se encontraban las instancias, y daremos click en el nombre de la instancia.

Instances

Create New Instance

Name	Host	Plan	Datacenter	Actions
Software architecture pattern	barnacle	Lemur	Amazon Web Services US-East-1 (Northern Virginia)	Edit RabbitMQ Manager

Fig 123 - Seleccionar la instancia.

Esto nos llevará a los detalles de la instancia, donde recuperamos los datos para conectarnos desde nuestra aplicación:

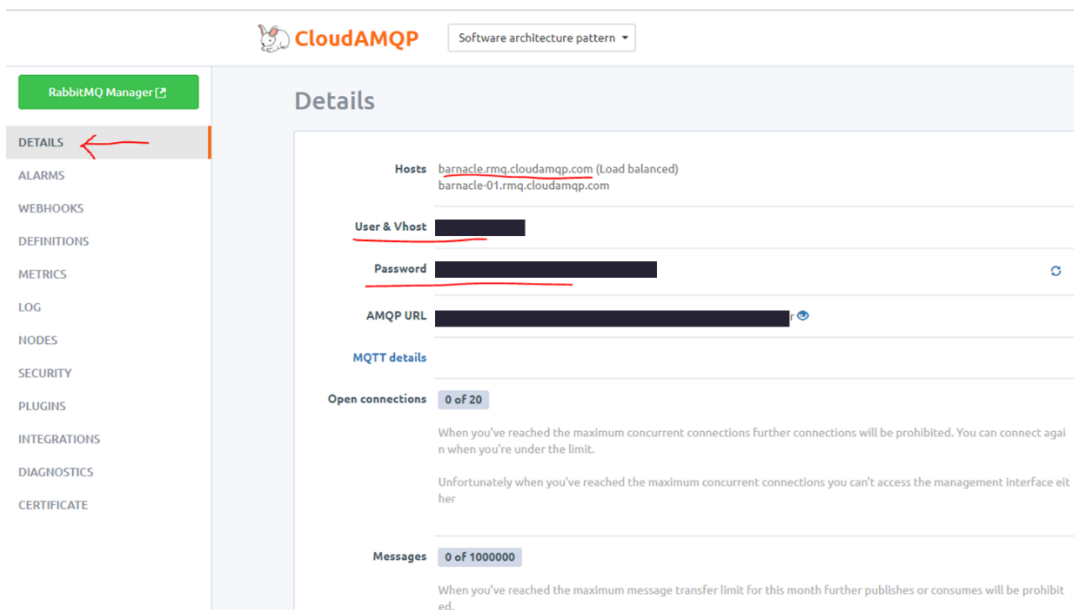


Fig 124 - Recuperar los datos de conexión.

En esta nueva página nos vamos a ir la sección *DETAILS* y vamos a tomar nota de los campos Hosts, User & Vhost y Password, ya que los vamos a necesitar más adelante cuando configuremos la aplicación.

Respecto al campo Hosts, podemos ver que tiene dos valores, por lo que hay que tener cuidado de seleccionar el primero (Load balancer). Mantén estos valores a la vista, porque los vamos a necesitar en la siguiente sección.

Iniciar la aplicación

En esta sección describiremos los pasos necesarios para echar a andar la aplicación, por lo que es importante haber concluido todos los pasos de la sección anterior antes de continuar.

Para llevar una correcta secuencia en la forma en que echamos a andar la aplicación, vamos a comenzar con el Backend, es decir, como los Microservicios que montamos en Spring Tool Suite.

El backend se compone de varios Microservicios que iremos analizando a medida que vamos avanzando en el libro, por lo que por ahora nos centraremos en echarlos a andar e iremos analizándolos a medida que avancemos en el libro.

Sin más preámbulos, regresemos al Spring Tool Suite y ubiquemos la pestaña **Boot Dashboard** que se ubica en la esquina inferior izquierda:

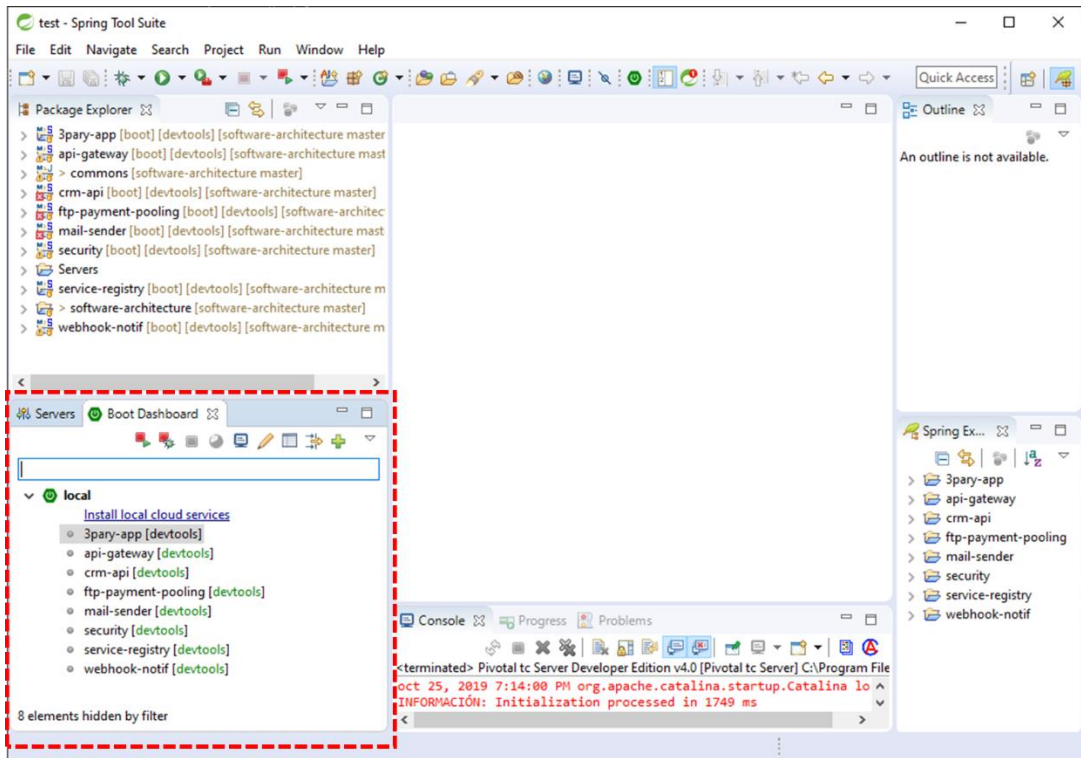


Fig 125 - Boot Dashboard.

Si no vez la pestaña o la cerraste por error, puedas abrirla nuevamente desde la opción Window → Show view → Other del menú superior, y luego seleccionamos la opción **Boot Dashboard** de la carpeta **Spring**.

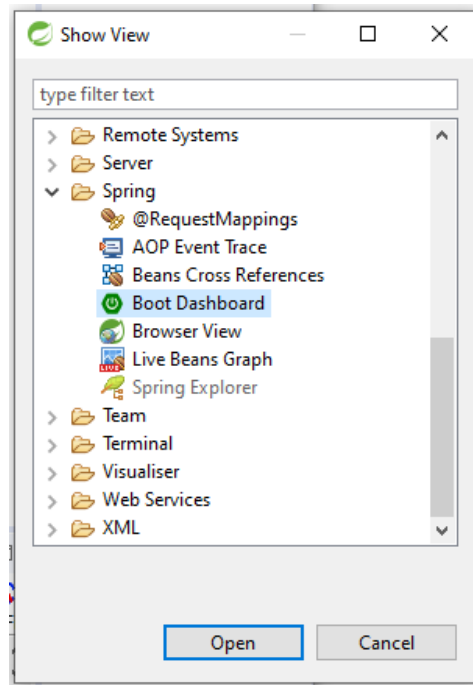


Fig 126 - Cómo abrir la sección Boot Dashboard.

Desde la sección *Boot Dashboard* podemos iniciar o detener los Microservicios, o incluso, podemos ver su estado, es decir, podemos ver si están apagados o encendidos, incluso, nos indica el puerto en el cual están aceptando peticiones. Para arrancar un Microservicio solo hay que seleccionarlo y presionar el botón de start (), el cual sirve para iniciar el Microservicio o Reiniciarlo en caso de que ya está encendido.

Antes de arrancar cada Microservicio es importante entender que cada uno levante su propio servidor Tomcat integrado, por lo que requiere un puerto para funcionar, por lo tanto, hay que estar seguros de los puertos necesarios para cada aplicación están disponibles, o de lo contrario tendríamos que cambiarlos para que puedan funcionar.

Cada proyecto tiene un archivo llamado *application.yml* (un archivo por proyecto) en el cual se guarda la configuración de la aplicación, y la propiedad "*server.port*" indica el puerto que tomará la aplicación al iniciar, por lo que podemos cambiarlo desde allí, sin embargo, te recomiendo no cambiar los puertos, ya que en el libro

aremos referencia a estos puertos y puede que si los modifiques te puedas confundir.

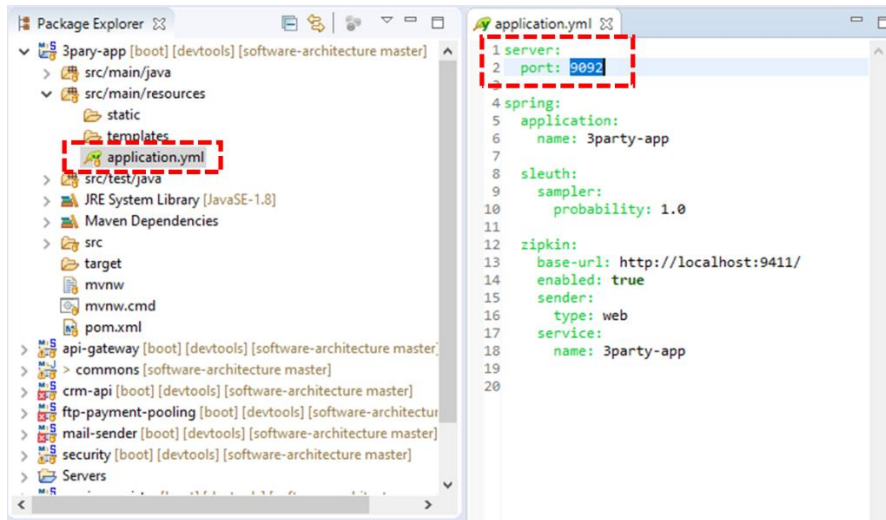


Fig 127 - Configurar el puerto de la aplicación.

Iniciando el Microservicio service-registry

Dicho lo anterior, vamos a comenzar iniciando la aplicación “**service-registry**”, para lo cual, vamos a regresar al Boot Dashboard y presionar el botón de iniciar, al mismo tiempo de iniciar la aplicación, se abrirá una **consola** donde se mostrará el log de la aplicación. Si todos sale bien, deberemos ver que la aplicación inicio correctamente y está escuchando solicitudes en el puerto 8761:

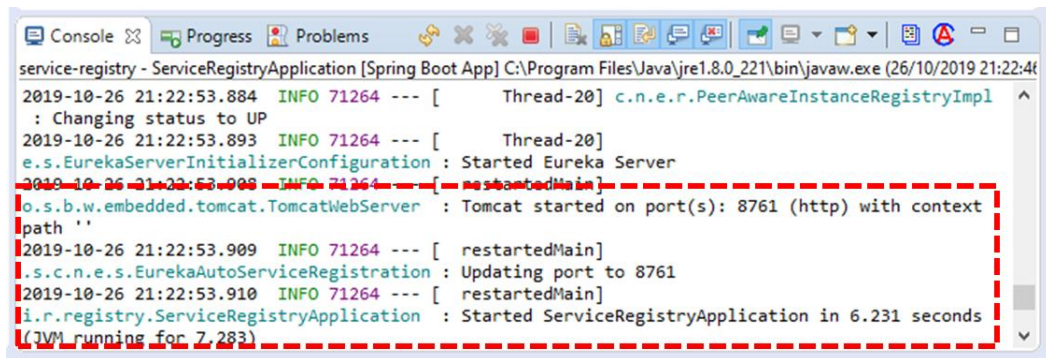


Fig 128 - Inicio exitoso del Microservicio service-registry.

Por otra parte, si regresamos al Boot Dashboard podremos ver que el ícono de la aplicación cambio a una fecha verde (↑) y se muestra el puerto en el que está aceptando peticiones.

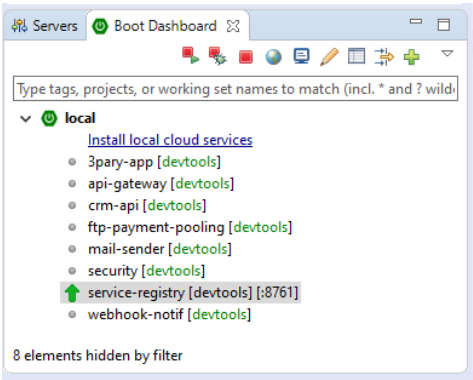


Fig 129 - Iniciando el Microservicio service-registry.

Finalmente, podemos comprobar que el Microservicio está funcionando si abrimos la url <http://localhost:8761> en el navegador y vemos una página como la siguiente:

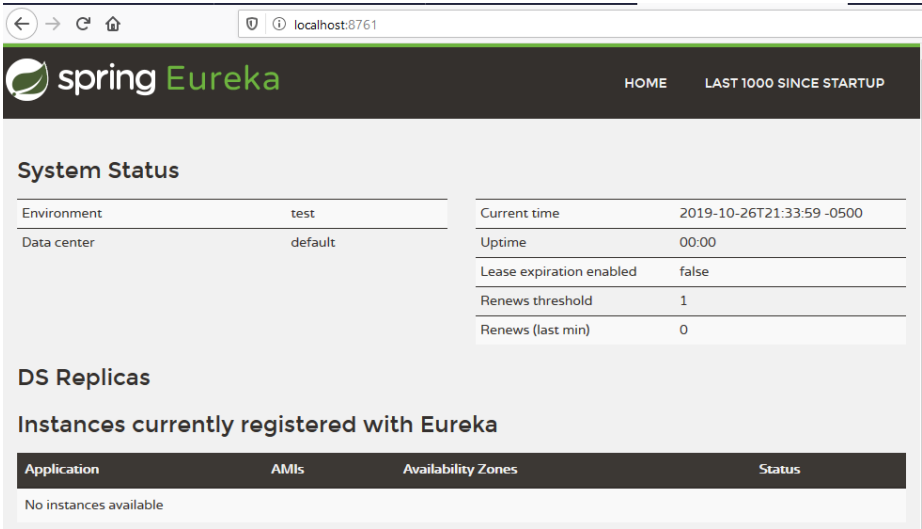


Fig 130 - Eureka server.

Mas adelante veremos que es esta página, por lo pronto nos servirá como prueba de que el Microservicio **service-registry** está funcionando correctamente.

Iniciando el Microservicio api-gateway y 3part-app

Bien, hemos iniciado el primer Microservicio, ahora vamos a repetir el proceso para iniciar los Microservicios **api-gateway** y **3party-app**:

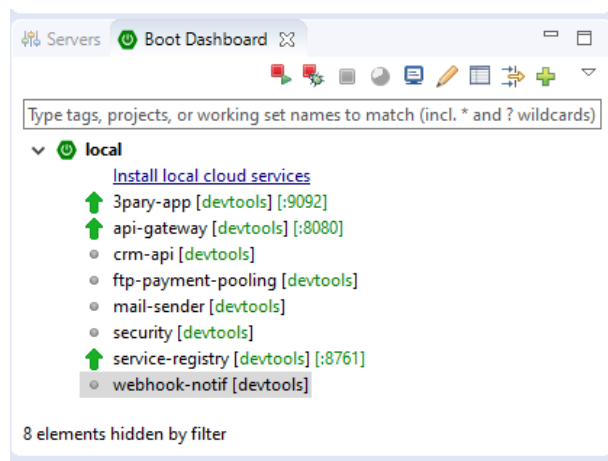


Fig 131 - Iniciando 3part-app y api-gateway.

Recordemos validar que la aplicación cambio de status a iniciada en el *Boot Dashboard* y revisar la consola para asegurarnos que no hay errores.

Iniciando el Microservicio security

Los Microservicios restantes requieren de algunas configuraciones adicionales para echarlas a andar, como configurar la conexión a la base de datos, configurar una cuenta de correo electrónico para enviar mails o configurar la conexión a RabbitMQ. Por lo que vamos a ir explicando como configurar cada Microservicio e iniciarlo.

Comenzaremos con el Microservicio “**security**”, el cual requiere configurar la conexión a la base de datos, para ello, vamos a expandir la carpeta del proyecto y ubicar el archivo *application.yml*, lo abrimos y buscaremos la sección de *mysql* para configurarlo según nuestra instalación de MySQL. **Es importante conectarnos a la base de datos “security”.**

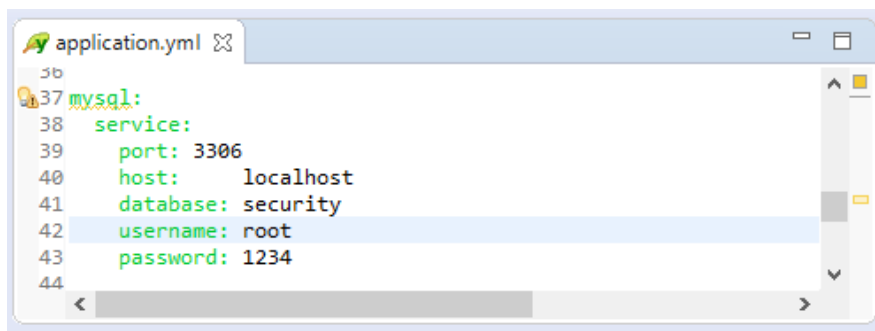


Fig 132 - Configurando la conexión a la base de datos.

NOTA: Si dejaste la configuración por default de MySQL solo vas a necesitar cambiar el password, en otro caso, cambia lo datos que veas necesario.

Una vez aplicados los cambios, guardamos el archivo y arrancamos el Microservicio desde el *Boot Dashboard*:

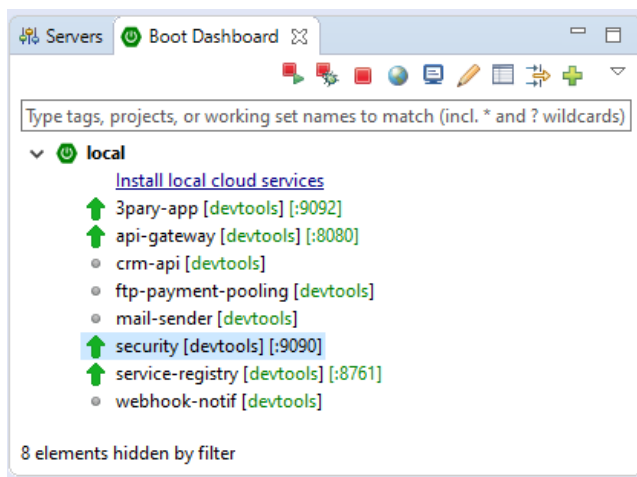


Fig 133 - Iniciando el Microservicio security.

Nuevamente, validamos se vea como encendido en el *Boot Dashboard* y validamos la consola para validar que no existan errores.

Iniciando el Microservicio webhook-notify

Vamos a continuar configurando el Microservicio **webhook-notify**, en el cual tendremos que configurar la conexión a la base de datos de la misma forma que lo hicimos con el Microservicio **security**, con la diferencia de que esta vez nos conectaremos a la base de datos webhook, por lo cual, vamos a abrir el archivo *application.yml* y editar los datos de conexión, guardamos ejecutamos el Microservicio desde el *Boot Dashboard*:

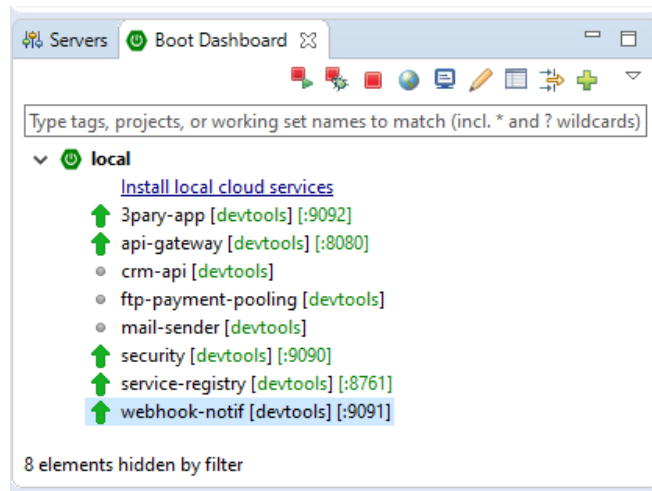


Fig 134 - Iniciando el Microservicio webhook-notify.

Iniciando el Microservicio crm-api

El siguiente Microservicio por iniciar es **crm-api**, en el cual también tenemos que abrir el archivo *application.yml* y configurar los datos de conexión a la base de

datos como lo hicimos en los Microservicios anteriores, además, es necesario configurar la conexión a *RabbitMQ*, por lo cual, debemos tener a la mano los datos de conexión que recuperamos en la sección de “**Configurando RabbitMQ**”, si no lo recuerdas puedes regresar a esta sección para ver como recuperarlo.

```
32 rabbitmq:
33   host: [REDACTED]
34   port: 5672
35   username: [REDACTED]
36   password: [REDACTED]
37   virtual-host: [REDACTED]
38 mysql:
39   service:
40     port: 3306
41     host: localhost
42     database: crm
43     username: root
44     password: 1234
```

Fig 135 - Configurando el Microservicio crm-api.

NOTA: El campo *username* y *virtual-host* corresponde al campo “**User & Vhost**” que recuperamos de la consola de *RabbitMQ*, los dos deben contener el mismo valor.

Una vez que hemos terminado de aplicar los cambios, guardamos los cambios y ejecutamos el Microservicio **crm-api** desde el *Boot Dashboard*. No olvidemos validar que el ícono del servicio cambio a encendido y revisamos la consola para validar que no existan errores de conexión a *RabbitMQ*, al final, deberíamos de ver el servicio encendido:

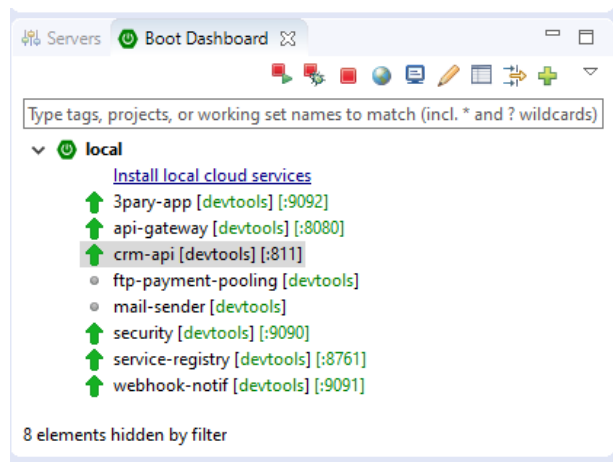


Fig 136 - Iniciando el Microservicio *crm-api*.

Iniciando el Microservicio mail-sender

El siguiente Microservicio a iniciar es **mail-sender**, el cual lo utilizaremos para el envío de correos electrónicos, por lo cual vamos a necesitar una cuenta de Gmail. Puedes utilizar tu cuenta personal o puedes crear una nueva, lo único importante es que configures la cuenta para habilitar el “*acceso de aplicaciones menos seguras*”. No te preocupes por el nombre de la opción, no significa que vamos a dejar nuestra cuenta desprotegida, simplemente la habilita para que las aplicaciones externas se puedan autenticar, al final, puedes deshabilitar esta opción una vez que termines de leer este libro.

No voy a explicar cómo crear una cuenta de Gmail porque es bastante obvio, pero si explicaré como configurar la cuenta para permitir el acceso a aplicaciones, para ello, lo primero que tenemos que hacer es ir a Gmail y autenticarnos. Una vez en el buzón, nos dirigimos a la sección de “Cuenta de Google” la cual se puede ver al darle click a nuestro avatar en la esquina superior derecha.

Una vez allí, nos dirigimos a la sección de seguridad y buscamos la sección “Acceso de apps menos seguras”, una vez allí, damos click “Habilitar el acceso” y listo, eso debería ser suficiente para que nuestra aplicación se pueda autenticar.

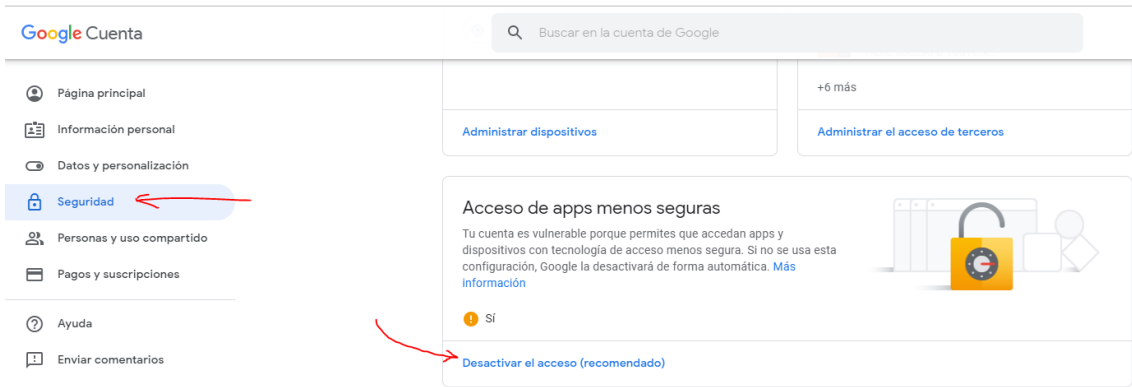


Fig 137 - Habilitando el acceso de app menos seguras.

Ya configurada la cuenta de correo, vamos a abrir el archivo *application.yml* del proyecto **mail-sender** y configurar la conexión a Gmail y RabbitMQ.

```
20 main:
21   allow-bean-definition-overriding: true
22 mail:
23   host: smtp.gmail.com
24   username: [REDACTED]
25   password: [REDACTED]
26   port: 587
27   properties:
28     mail:
29       smtp:
30         auth: true
31         connectiontimeout: 5000
32         timeout: 5000
33         writettimeout: 5000
34         starttls:
35           enable: true
36         ssl:
37           trust: smtp.gmail.com
38 rabbitmq:
39   host: [REDACTED]
40   port: 5672
41   username: [REDACTED]
42   password: [REDACTED]
43   virtual-host: [REDACTED]
```

Fig 138 - Configurando el servicio mail-sender.

Finalmente, guardamos los cambios y ejecutamos el Microservicio desde el Boot Dashboard. Nos aseguramos que el servicio esté encendido y que no existan errores en los logs de la consola.

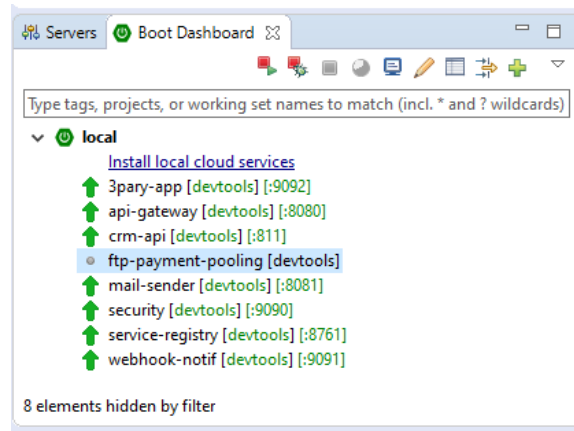


Fig 139 - Ejecutar el Microservicio mail-sender.

En este punto, deberíamos de tener casi todos los servicios encendidos y el *Boot Dashboard* se debería de ver como en la imagen anterior. Si algún servicio de los anteriores no está ascendido, deberás de revisar la consola en búsqueda de algún error y corregirlo, seguramente alguna configuración está incorrecta.

Ya solo nos quedaría iniciar el Microservicio **ftp-payment-pooling**, pero este lo dejaremos para después, ya que no es requerido de momento.



Tip

En el futuro puede iniciar todos los servicios al mismo tiempo o en cualquier orden. Aquí lo hemos hecho en un orden específico solo para enseñarte a configurarlos, pero una vez configurados, ya se pueden iniciar en cualquier orden.

En este punto ya tenemos todo el backend funcionando, es decir, todos los Microservicios ya están funcionando y están listos para aceptar peticiones, por lo que en la siguiente sección describiremos los pasos necesarios para ejecutar la aplicación web.

Iniciando la aplicación web (e-commerce)

Para iniciar la aplicación web será necesario regresar al Visual Studio Code tal y como lo habíamos dejado en la sección de "[Instalando Visual Studio Code](#)", en el cual aviamos abierto la carpeta del proyecto ecommerce y nuestro editor se veía de la siguiente manera:

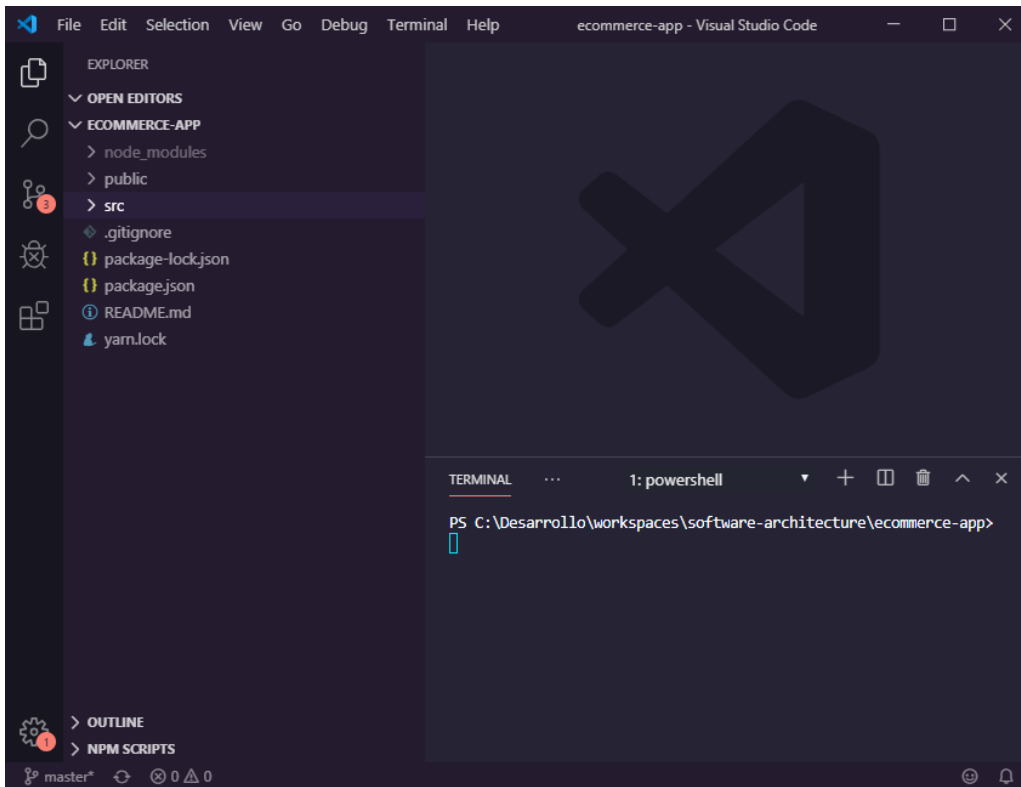
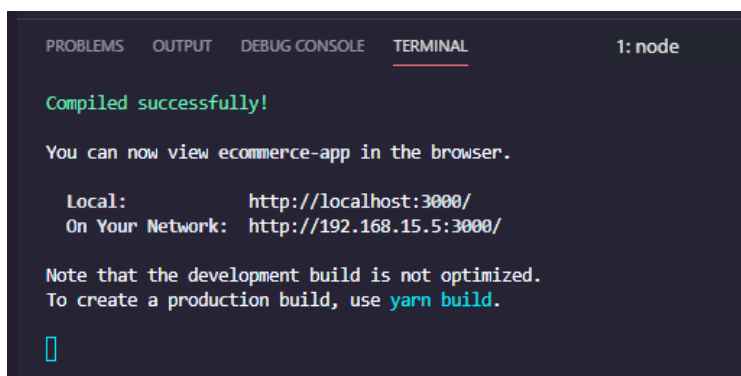


Fig 140 - Estado actual de Visual Studio Code.

Una vez allí, podemos presionar `Ctrl + ñ` para abrir una nueva terminal, o también puedes ir a `View → Terminal` del menú principal. Por defecto, la terminal se encuentre en la carpeta del proyecto, por lo que simplemente podemos ejecutar el comando `"npm install"` para que se comiencen a descargar todas las librerías necesarias. La instalación de las librerías puede tardar varios minutos, dependiendo de la conexión a Internet, así que seamos pacientes.

Una vez que la instalación termine, ejecutaremos el comando `"npm start"`, lo que hará que la aplicación web esté por fin disponible y lista para ser ejecutada en la URL <http://localhost:3000>.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  1: node

Compiled successfully!

You can now view ecommerce-app in the browser.

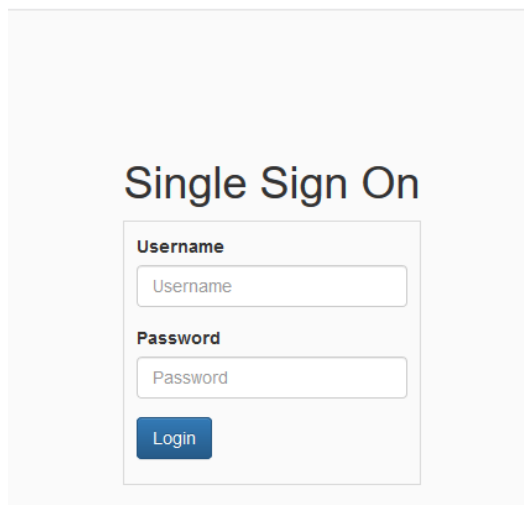
Local:      http://localhost:3000/
On Your Network:  http://192.168.15.5:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.

[]
```

Fig 141 - Ejecución exitosa de la aplicación ecommerce.

En la imagen anterior se puede observar que la aplicación fue construida exitosamente y que está disponible en el puerto 3000, por lo que nos vamos al navegador y abrimos la URL que nos arroja:



Single Sign On

Username

Password

Login

Fig 142 - Página inicial de la aplicación.

El login será la primera página que veamos al momento de entrar a la aplicación, por lo que tendremos que introducir un usuario válido. Por defecto, el único

usuario que existirá será **"oscar"** con password **"1234"**. Puedes agregar más usuario en la tabla **"users"** de la base de datos **"security"**, aunque de momento no será necesario. Al introducir las credenciales podremos ver la aplicación web:

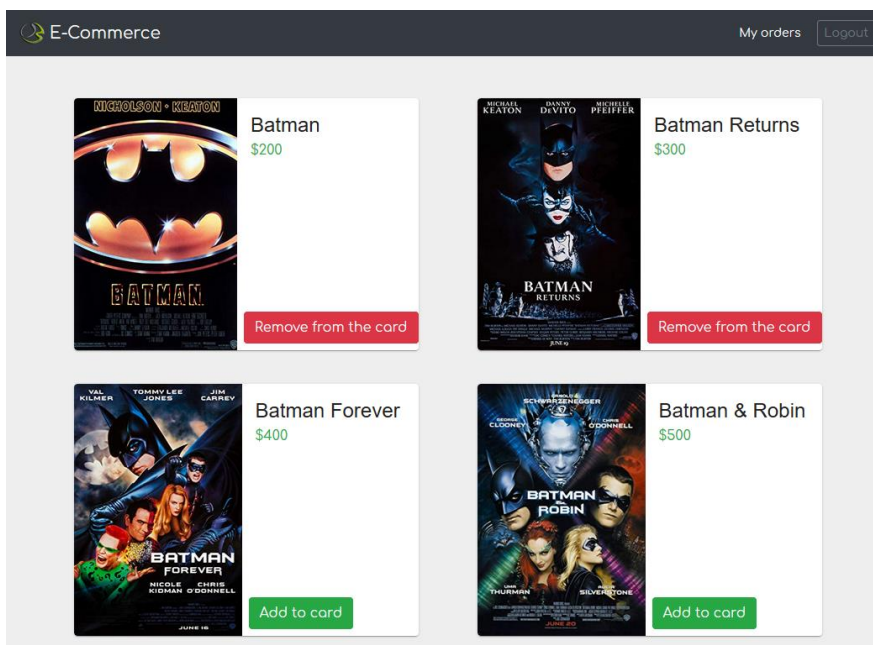


Fig 143 - Página principal de la aplicación e-commerce.

Si toda la instalación fue correcta, deberías de poder ver la aplicación como se muestra en la imagen anterior, en otro caso, será necesario revisar los pasos anteriores y asegurarnos de que todo está correctamente instalado y todos los componentes han encendido exitosamente.

En la siguiente sección analizaremos algunos casos de uso de la aplicación y veremos cómo funciona.

Cómo utiliza la aplicación

La aplicación básicamente nos permite agregar ciertos productos a nuestro carrito de compra y realizar la compra, lo cual puede parecer una aplicación bastante simple, y lo es, al menos en funcionalidad para el usuario, pero recordemos que la gran cantidad de trabajo está en el Backend, por lo que podemos subestimar la aplicación basado en lo que podemos ver como usuarios de la aplicación, sin embargo, a medida que avancemos por el libro veremos como todos los patrones arquitectónicos aquí definidos están contemplados en este ejemplo práctico.

Para comenzar, vamos a entrar a la aplicación (<http://localhost:3000>), si es la primera vez que entras te pedirá que te autentiques, así que recuerda que el usuario válido es oscar/1234, una vez dentro, veremos los productos disponibles en la aplicación, que en este caso son películas:

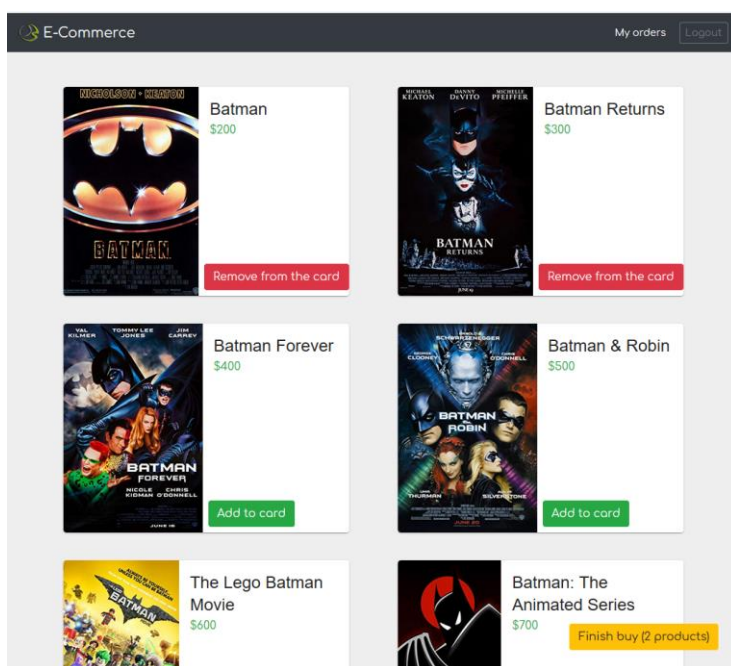


Fig 144 - Página principal de la aplicación.

Vamos a comenzar con realizar una compra, para ello, vamos a agregar dos productos a nuestro carrito de compras, para esto, deberemos de presionar el botón **"Add to card"** que se encuentra en cada película, una vez que agreguemos las películas, vamos a ir al carrito de compras para realizar el pago, para esto, presionamos el botón amarillo que dice **"Finish buy"** posicionado en la esquina inferior derecha.

Esto nos llevará al carrito de compras donde podremos ver los productos agregados y un formulario de pago:

E-Commerce

My ordersLogout

Revisa tu pedido y realiza el pago

#	Product	Price
1	Product A	100
2	Product B	200
Total		\$300

Seleccione forma de pago

Tarjeta de créditoDepósito bancario

Introduzca los datos de su tarjeta de crédito

**** *
YOUR NAME HERE

valid thru
/

Número de tarjeta

Nombre del cliente

Fecha vencimiento

CVC

Pagar

Fig 145: Carrito de compras.

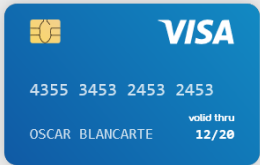
La aplicación soporta dos métodos de pago, tarjeta de crédito o depósito bancario, el primero nos permite capturar los datos de una tarjeta de crédito para finalizar la compra, lo que disparará la creación de la orden y un correo de confirmación del pago recibido, lo que dará como resultado una orden pagada y lista para entregar los productos comprados. Por otro lado, el pago mediante depósito bancario solo creará la orden, pero estará pendiente de pago y nos generará un número de referencia que tendremos que utilizar para pagar en el banco, por lo que la orden no se liberará para su entrega hasta que se reciba el pago. Por ahora solo nos centraremos en el pago por tarjeta de crédito y retomaremos el depósito bancario cuando hablemos del patrón *Polling*.

Retomando el pago con tarjeta de crédito, tendremos que llenar los datos de la tarjeta para finalizar la compra, por lo que vamos a proceder a llenar los datos. Los datos de la tarjeta son ficticios y no realizaremos un cargo real, así que podemos poner cualquier dato, siempre y cuando cumpla con el formato, por ejemplo, número de tarjeta debe de ser de entre 15 y 16 dígitos, el nombre cualquier valor alfanumérico (no blanco), la fecha de nacimiento a 4 dígitos, dos para el mes y dos para el año y el CVC puede ser de entre 3 y 4 dígitos:

Seleccione forma de pago

Tarjeta de crédito Depósito bancario

Introduzca los datos de su tarjeta de crédito



4355345324532453

oscar blancarte

1220 1233

Pagar

Fig 146 - Capturando los datos de la tarjeta de crédito.

Una vez capturado los datos de la tarjeta vamos a presionar el botón **"Pagar"**, lo que nos dará como resultado la confirmación de la compra:

Pedido realizado con éxito

Tu pedido ha sido registrado con éxito, un email llegará a tu correo con los detalles de tu compra

Pedido No: 57f1ddce-88c4-49d7-b7a0-99bcd2ec0e6

Ver mi compra

Fig 147 - Confirmación exitosa de la compra.

Si presionamos el botón **"Ver mi compra"** nos llevará al detalle del pedido:

 E-Commerce
 My orders
Logout

Datos de la compra

Usuario	oscar
User Email	oscar_jb1@hotmail.com
Número de orden	57f1ddce-88c4-49d7-b7a0-99bcd82ec0e6
Fecha de creación	31/10/2019 02:48:57
Estatus	PAYED

Datos del pago

Fecha de pago	31/10/2019 02:48:57
Método de pago	Tarjeta de crédito

Fig 148 - Detalle de la compra.

Otra forma de ver todas las ordenes creadas por nosotros es ir a la sección “My orders”, dando click en el botón ubicado en la barra de navegación, en la esquina superior derecha:

 E-Commerce
 My orders
Logout

Todas tus órdenes

ID	User	Ref Number	Total	Date	Actions
123	oscar	57f1ddce-88c4-...	300	31/10/2019 02:48:...	Ver
122	oscar	6a1658de-0e56-...	200	31/10/2019 02:29:...	Ver
121	oscar	8a601824-eff7-4...	200	31/10/2019 02:10:03	Ver
120	oscar	7461e2ab-f649-4...	200	31/10/2019 02:09:...	Ver
119	oscar	67a7b53a-80b2-...	300	27/10/2019 12:30:...	Ver

Previous
Page 1 of 3
5 rows
Next

Fig 149 - Mis órdenes.

Desde el lado de la aplicación es básicamente todo lo que hay por mostrar, por lo que en la siguiente sección analizaremos todos los componentes que componen el backend y como estos interactúan para hacer funcionar la aplicación.

En las siguientes secciones comenzaremos a explicar los patrones arquitectónicos y analizaremos como estos se implementan en la aplicación web.

Patrones arquitectónicos

Capítulo 6

Hasta antes de esta unidad nos hemos estado preparando para llegar a esta unidad, pues desde el inicio hemos venido aprendiendo los conceptos básicos de la arquitectura de software, los principales principios de la ing. de software y hemos aprendido a identificar y clasificar los principales estilos arquitectónicos de software que existen.

Si te has dado cuenta, todo lo que hemos visto hasta ahora es pura teoría, pues hemos desarrollado y reafirmado las principales capacidades de un arquitecto, que es conocer la tecnología de forma **agnóstica**, lo cual quiere decir que somos capaces de comprender como el software funciona sin comprender exactamente como está desarrollado ni las tecnologías empleadas para hacerlo funcionar, pero debido a nuestro conocimiento como arquitectos, somos capaces de entender e identificar sus características, ventajas, y sus vulnerabilidades.



Nuevo concepto: Agnóstico

En el software utilizamos el termino Agnóstico para hacer referencia a conceptos o componentes genéricos que no hacen referencia a una marca o tecnología particular.

En esta unidad saltaremos de la teoría o cosas más reales, pues exploraremos los patrones arquitectónicos que desde mi punto de vista son los más importantes y útiles para desarrollar aplicaciones modernas, ya sean aplicaciones de escritorio, web, en la nube o distribuidas.

Como mencionamos al inicio de este libro, los patrones arquitectónicos se distinguen de los patrones de diseño debido a que tiene un alcance global sobre

los componentes, ya que afectan su funcionamiento, su integración o la forma en que se comunican con otros componentes. Sin embargo, la principal diferencia que vamos a ver entre los patrones de diseño y los estilos arquitectónicos es que, en los patrones arquitectónicos si se define como las cosas de deben de funcionar, se definen los componentes mínimos y como estos deberían de funcionar para cumplir el patrón, es por esta razón que veo importante que desarrollemos un proyecto completo donde exploremos al máximo todos los patrones de arquitectónicos.

Debido a esto, esta unidad tendrá una sección para describir el patrón arquitectónico, y luego otra para explicar cómo este patrón arquitectónico se utiliza dentro de nuestra aplicación.

Data Transfer Object (DTO)

Hoy en día vivimos en una época donde prácticamente todas las aplicaciones tienen necesidad de intercambiar mensajes con terceros, ya sea componente que integran información con otros sistemas o servicios que disponibilizamos para que cualquiera pueda interactuar con nuestro sistema, sea cual sea el caso, tenemos la necesidad de enviar mensajes, dichos mensajes por lo general tiene un correlación directa con las entidades de nuestra aplicación, por lo que es común utilizar dichas entidades de cómo respuestas de nuestros servicios.

Problemática

Una de las problemáticas más comunes cuando desarrollamos aplicaciones, es diseñar la forma en que la información debe viajar desde una capa de la aplicación a otra capa, ya que muchas veces por desconocimiento o pereza, utilizamos las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios, o incluso, tengamos que ir en más de una ocasión a la capa de servicios para recuperar los datos requeridos.

El patrón DTO tiene como finalidad la creación de objetos planos (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en **una sola invocación**, de tal forma que un DTO puede contener información de **múltiples fuentes** o tablas y concentrarlas en una única clase simple.



Nuevo concepto: Entidad

Las entidades son las clases que mapean directamente contra la base de datos, es este sentido, una Entidad hace referencia a una tabla y tienen un atributo por cada columna de la tabla. Son muy utilizadas en los Frameworks ORM (Object-Relational mapping).

Para comprender mejor este patrón imaginemos que tenemos dos tablas en la base de datos, una con Clientes (*Customers*) y otra con su dirección (*Address*). La tabla *Address* tiene una columna que hace referencia al Cliente, pero desde el cliente no tenemos una columna para llegar a su dirección. En pocas palabras tenemos una relación "One To One".

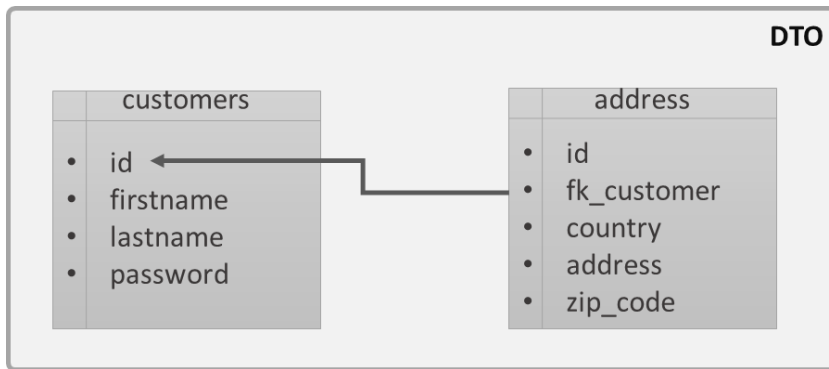


Fig 150 - Estructura de las tablas Customers y Adress.

Dadas las tablas anteriores, imagina que tiene una Entidad para cada tabla:

```

1. public class Customer {
2.     private Long id;
3.     private String firstname;
4.     private String lastname;
5.     private String password;
6. }
  
```

Y

```

1. public class Address {
2.     private Long id;
3.     private Customer customer;
4.     private String country;
5.     private String address;
6.     private String zipCode;
7. }
  
```

Ahora imagina que tienes que construir un servicio que regresa todos los datos de la Entidad *Customer*, seguramente lo que primero que te pasará por la mente será crear un servicio que retorne la entidad como tal, al fin y al cabo, que quieren todos los datos del cliente, entonces termino haciendo lo siguiente:

```

1. public class CustomerService {
2.
3.     public Customer findCustomerById(Long id) {
4.         //Simulate db connection
5.         Customer customer = customerDB.getCustomer(id);
  
```

```
6.         return customer;
7.     }
8. }
```

Vemos que hemos creado un método llamado *findCustomerById*, el cual es el que se encarga de buscar el cliente en la base de datos (línea 5) y luego retorna la Entidad obtenida. **Pon atención en el tipo de dato que estamos retornando porque es la clave de esto.**

NOTA: Estamos asumiendo que existe un método llamado *getCustomer* (línea 5) que va a la base de datos recupera al cliente y crear la Entidad por nosotros.

Esta solución es más común de lo que parece, porque no implica nada de esfuerzo adicional, simplemente retornamos la Entidad que ya construí en otro lado y listo, sin embargo, eso es precisamente lo que trata de evitar el patrón *Data Transfer Object*, el cual nos propone la creación de objetos planos utilizados únicamente para la transmisión de los datos, logrando con ello tener un mejor control sobre lo que estamos retornando.

Muy bien, supongamos que ya hemos implementado el método como se ve en el fragmento de código anterior y estamos muy orgullosos de eso, y de repente llega un nuevo requerimiento que nos solicita que este servicio debe de retornar también la dirección del cliente.

Mmmm, esto ya nos pone a pensar un poco la situación, pues la entidad como tal no tiene una propiedad de tipo *Address*, lo que hace imposible retornar la dirección, o al menos no con la estructura actual de la Entidad, pero cómo buen Ing en Software decimos, pues muy fácil, agreguemos la propiedad *Address* a la Entidad *Customer* y listo:

```
1. public class Customer {
2.     private Long id;
3.     private String firstname;
4.     private String lastname;
5.     private String password;
6.     private Address address;
7. }
```

A primera vista esto parece coherente, al final un cliente tiene una dirección, por lo tanto, no estaría mal agregar dicha relación y con esto en mente nos justificamos a nosotros mismos que es lo correcto, pero seamos sinceros, tú y yo sabemos que algo comienza a apestar con esta solución, aunque no sabemos exactamente por qué.

Pero entonces llega otro requerimiento donde nos piden que retornemos al cliente con todos sus pedidos, y luego sus teléfonos y luego con todas las compras que ha hecho, etc, etc.

Cuando menos nos damos cuenta, la Entidad *Customer* que su única responsabilidad era la de mapear con la base de datos comienza a verse comprometida por las reglas de negocio y los requerimientos que no obedecen al contexto de datos, por lo tanto, terminamos con una Entidad que mezclan datos de la base de datos con datos que necesitábamos al vuelo para solucionar un problema de negocio y no de persistencia.

Otro problema importante es que la forma en que persistimos los datos no es exactamente como los queremos mostrar a los consumidores de los servicios, por ejemplo, es posible que queramos cambiar el nombre o el tipo de los datos para adaptarse mejor a los consumidores, por ejemplo, es posible que tengamos una fecha de registro de tipo *Calendar* (en Java) y el status de un tipo de dato Enumeración:

```
1. public class Customer {  
2.     private Long id;  
3.     private String firstname;  
4.     private String lastname;  
5.     private String password;  
6.     private Address address;  
7.     private Calendar regDate;  
8.     private Status status;  
9. }
```

Y

```
1. public enum Status {  
2.     ACTIVE,  
3.     INNACTIVE  
4. }
```

Imagina que el servicio que estamos construyendo es de tipo REST y que los datos deben de viajar en formato JSON, esto nos trae un problema, y es que el tipo de dato *Calendar* (Un tipo de datos usado en Java para representar la fecha y hora) no puede ser serializado correctamente al pasar a JSON y si lo fuera, no tenemos el control del formato que tomará la fecha y hora al momento de convertirse en una cadena de texto, por otro lado, la Enumeración tampoco nos queda claro cómo será convertida a *String*, ahora imagina que el campo *regDate* te han solicitado

se exponga con el nombre de *date*. Podemos ver que empezamos a tener muchos problemas con la Entidad, pues no nos da el control que pensábamos tener el comienzo, por lo que otra mala práctica es comenzar a crear propiedades paralelas para solucionar este problema:

```
1. public class Customer {  
2.     private Long id;  
3.     private String firstname;  
4.     private String lastname;  
5.     private String password;  
6.     private Address address;  
7.     private Calendar regDate;  
8.     private Status statusDB;  
9.  
10.    private String date;  
11.    private String status;  
12. }
```

Lo que hacemos para salir rápido del problema es crear dos propiedades nuevas llamadas *date* y *status* de tipo *String*, con la intención de solucionar el problema del requerimiento de cambiar el nombre propiedad *regDate* por *date* y el tipo de datos del status de *Calendar* a *String*, pero ahora nos damos cuenta que el nombre de la propiedad *status* ya está definida por la propiedad de tipo Enumeración, lo que nos lleva renombrar la propiedad para dejar el nombre status disponible para la propiedad de tipo *String*. Bueno, a lo que quiero llegar con esto es que la Entidad no nos da la flexibilidad necesaria para adaptarnos a los cambios.

Otro problema es que al retornar la entidad completa nos estamos arriesgando a retornar datos de más o que son sensibles, por ejemplo, vemos que la Entidad *Customer* tiene una propiedad llamada *password*, la cual puede corresponder a su contraseña, pero como retornamos toda la Entidad, estamos regresando todos los datos, incluido el password, el cual no deberíamos de regresar por ningún motivo.

Otro problema es que, si por cualquier motivo necesitamos cambiar la Entidad, estaríamos impactando directamente la respuesta del servicio y con ello, romperíamos la compatibilidad del servicio con los consumidores que ya contaban con un determinado formato de la respuesta.

Solución

Como vemos, utilizar una *Entity* o cualquier otro objeto que allá sido creado para otro propósito diferente que el de ser usado para transmisión de datos puede tener complicaciones, por lo que el Patrón Data Transfer Object (DTO) propone que en lugar de usar estas clases, creamos clases especiales para transmitir los datos, de esta forma, podemos controlar los datos que enviamos, el nombre, el tipo de datos, etc, además, si estos necesitan cambiar, no tiene impacto sobre la capa de servicios o datos, pues solo se utilizan para transmitir la respuesta. Dicho lo anterior, retornemos al ejemplo anterior, pero utilizando un DTO:

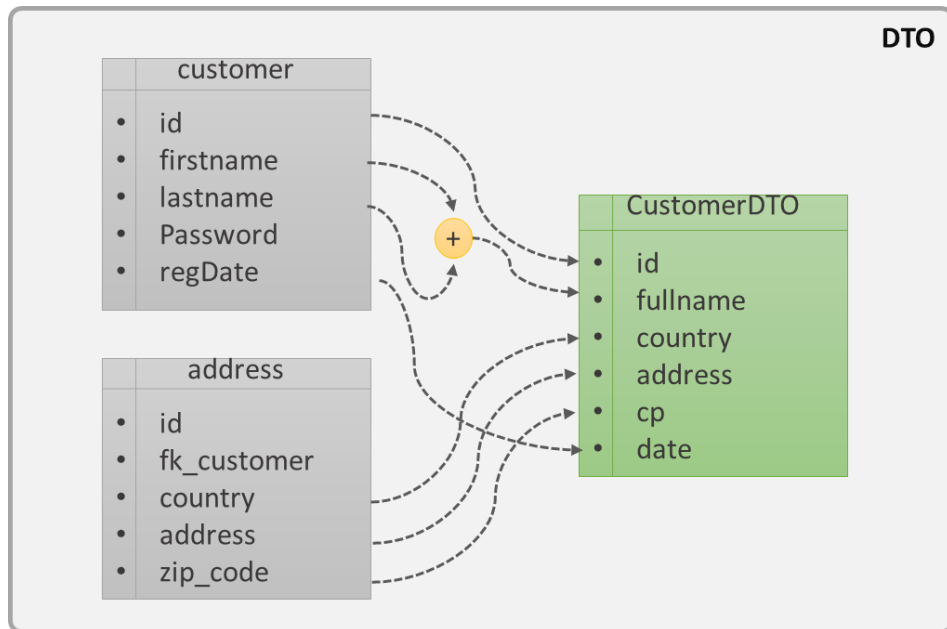


Fig 151 - Implementando un DTO.

En este nuevo ejemplo podemos ver que hemos creado un nuevo objeto llamado *CustomerDTO*, en el cual podemos agregar libremente cuanto atributo requiramos, incluso, podemos asignarle valores de diferentes fuentes de datos.

Debido a que el DTO es una clase creada únicamente para una determinada respuesta, es posible modificarla sin mucho problema, pues no tiene un impacto

en la capa de servicios o de datos, ya que en estas capas se trabaja con las Entidades.

Ahora bien, una vez que tenemos el DTO, podemos modificar el servicio para que quede de la siguiente manera:

```
1. public class CustomerService {
2.
3.     public CustomerDTO findCustomerById(Long id) {
4.         //Simulate db connection
5.         Customer customer = customerDB.getCustomer(id);
6.         Address address = addressDB.getAddressByUser(id);
7.
8.         SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
9.         CustomerDTO dto = new CustomerDTO();
10.
11.        //Data from customer
12.        dto.setId(customer.getId());
13.        dto.setFullname(customer.getFirstname() + " "
14.            + customer.getLastname());
15.        dto.setCountry(customer.getCountry());
16.
17.        //Data from address
18.        dto.setDate(dateFormat.format(customer.getRegDate().getTime()));
19.        dto.setAddress(address.getAddress());
20.        dto.setCp(address.getZipCode());
21.
22.        return dto;
23.    }
24. }
```

Si observar el fragmento de código anterior, podrás observar que el tipo de dato que retornamos ha cambiado de la entidad *Customer* a *CustomerDTO*, pero también te habrás dado cuenta que ahora tenemos que construir el DTO “seteando” cada uno de sus valores.

Seguramente en este punto te estés imaginando que es mucho trabajo tener que construir el DTO cada vez que queramos retornar algún valor, y eso multiplicado por todas las entidades de nuestro proyecto o peor aún, por todos los métodos o servicios que tengamos en nuestro proyecto, sin embargo, podemos utilizar algo llamado *Converter*, las cuales son clases dedicadas únicamente para conversión bidireccional de las Entidades a DTO y de DTO a *Entity*.



Nuevo concepto: Converter

Converter es un patrón de diseño que se utiliza para realizar conversiones entre dos tipos de clases de forma Bidireccional, de tal forma que podemos crear un DTO a partir de una Entidad y una Entidad a partir de un DTO.

En la siguiente sección analizaremos a detalle que es un *converter*, pero por ahora imagina que es una clase que tiene un método para convertir nuestra Entidad en un DTO, y de esta forma, en lugar de hacer la conversión directamente sobre nuestro servicio, delegamos esa responsabilidad a una clase *converter*:

```
1. public class CustomerService {
2.
3.     public CustomerDTO findCustomerById(Long id) {
4.         //Simulate db connection
5.         Customer customer = customerDB.getCustomer(id);
6.
7.         CustomerConverter converter = new CustomerConverter();
8.         CustomerDTO dto = converter.convert(customer);
9.
10.        return dto;
11.    }
12. }
```

Observa cómo en la línea 7 creamos una instancia del *converter* para después, y de un solo paso, convertir la Entidad a DTO mediante la línea 8. Esto reduce drásticamente la cantidad de líneas necesarias para conversión, hace más limpio el código y al tener una clase *converter* tenemos la ventaja de que podemos reutilizar la lógica de conversión en cualquier otro punto de la aplicación, reduciendo los puntos de error al convertir y de esta forma, si hay algún cambio en el DTO o en la Entidad, solo tendremos que afectar el *converter* en lugar de tener que ir método por método modificando la lógica de conversión.

Converter pattern

Si bien el patrón *Converter* es algo que debería quedar fuera de este libro, he decidido agregar una sección pues este patrón complementa a la perfección el patrón DTO.

En cualquier lenguaje de programación, es común encontrarnos con la necesidad de realizar conversión de tipos de datos, sobre todo, aquellos tipos de datos de Entidad que tienen una relación directa con un DTO que utilizamos para enviar los datos del servidor a un cliente o aplicación web, lo que hace que tengamos que convertir la Entidad a DTO para enviarla al cliente y de DTO a Entidad para persistirla en la base de datos, lo cual es una tarea cansada y repetitiva que podemos evitar mediante el uso del patrón *Converter*.

Pues como lo acabo de decir, convertir tipos de datos es una tarea que se presenta mucho en las aplicaciones, pero sobre todo en la capa de negocio o servicios, donde tenemos que enviar un tipo de dato amigable para el cliente o navegador, pero en el Backend necesitamos otros tipos de datos. Para evitar esta tarea tan repetitiva se puede utilizar el patrón *Converter*, el cual permite encapsular la lógica de conversión de dos tipos de datos de forma bidireccional, de esta forma, evitamos tener que repetir la lógica de conversión de los tipos de datos en todas las partes del programa donde sea requerido, y en su lugar, delegamos esta responsabilidad a una clase externa.

Para analizar esta problemática, analizaremos el caso típico de la Entidad Usuario, la cual tiene por lo general un *id*, *username*, *password* y los *roles*, tal como podemos ver a continuación:

```
1. public class User {  
2.  
3.     private Long id;  
4.     private String username;  
5.     private String password;  
6.     private List<Role> roles;  
7.  
8.     /** GET & SET */  
9. }
```

Esta entidad es utilizada por el Backend para persistir la información en la base de datos, sin embargo, no es la que utilizamos para enviar a los clientes, ya que por ejemplo, el campo *roles* es de un tipo de Enumeración, el cual no puede interpretar el cliente que quizás consuma el backend por medio de un API REST,

por lo que es posible que en lugar de enviarle una lista de Roles, le tendremos que mandar una lista de *Strings*, lo que nos obliga a crear un DTO que se ve de la siguiente manera:

```
1. public class UserDTO {  
2.     private Long id;  
3.     private String username;  
4.     private String password;  
5.     private List<String> roles;  
6.  
7.     /* GET & SET */  
8. }
```

Este DTO ya es más amigable para el cliente, pero ahora tenemos un problema, como crear el DTO a partir del Entity, y luego, cuando lo manden de regreso para guardarlo o actualizarlo, hay que hacer el proceso inverso. Por simple que parezca, esta es una tarea que se puede repetir muchas veces en la aplicación, creando mucho código redundante y que cada repetición puede inyectar errores, además, un cambio en la Entity o en el DTO, nos obliga a refactorizar todas las secciones donde habríamos realizado la conversión, por este motivo, siempre es mejor crear una clase que se encargue exclusivamente de convertir entre Entity y DTO.

Para solucionar este problema y tener un mejor orden en nuestro código, deberemos crear una clase base de la cual extiendan todos los *Converters* de nuestra aplicación:

```
1. public abstract class AbstractConverter<E,D> {  
2.  
3.     public abstract E fromDto(D dto);  
4.     public abstract D fromEntity(E entity);  
5. }
```

Esta clase abstracta define dos métodos, *fromEntity* que convierte una *Entidad* a DTO y *fromDto* que hace lo inverso, adicional, la clase define dos tipos genéricos *E* para representar al tipo Entidad y *D* para representar al DTO.

El siguiente paso es crear implementaciones concretas del *Converter*, por que deberá existir una implementación por cada par Entidad-DTO. En este caso solo tenemos la clase *User*, por lo que crearemos el Convertidor para esta clase:

```

1. public class UserConverter extends AbstractConverter<User, UserDTO>{
2.
3.     @Override
4.     public User fromDto(UserDTO dto) {
5.         User user = new User();
6.         user.setId(dto.getId());
7.         user.setUsername(dto.getUsername());
8.         user.setPassword(dto.getPassword());
9.
10.        // Prevent Null Exception
11.        if(dto.getRoles()!=null) {
12.            user.setRoles(dto.getRoles().stream()
13.                .map(rol -> Role.valueOf(rol)).collect(Collectors.toList()));
14.        }
15.        return user;
16.    }
17.
18.    @Override
19.    public UserDTO fromEntity(User entity) {
20.        UserDTO user = new UserDTO();
21.        user.setId(entity.getId());
22.        user.setUsername(entity.getUsername());
23.        user.setPassword(entity.getPassword());
24.
25.        // Prevent Null Exception
26.        if(entity.getRoles()!=null) {
27.            user.setRoles(entity.getRoles().stream()
28.                .map(rol -> rol.name()).collect(Collectors.toList()));
29.        }
30.        return user;
31.    }
32. }

```

Esta nueva clase se encargará de convertir los dos tipos de datos dejando en un solo lugar la lógica de conversión y evitando tener que repetir esta lógica en todo nuestro código. Podrás observar que esta clase no tiene gran ciencia, pues es básicamente tomar los valores de un objeto y pasarlos al otro objeto, nada del otro mundo.

Ahora bien, ya con esto, ¿cómo le haríamos para convertir los datos?, pues ya solo falta instanciar al convertidor y utilizarlo. Imaginemos que tenemos un método que consulta un usuario:

```

1. public UserDTO findUserByUsername(String username) {
2.     User user = userDao.findByUsername(username);
3.     UserConverter converter = new UserConverter();
4.     return converter.fromEntity(user);
5. }

```

En este método ya podemos observar las bondades del *converter*, pues nos deja un código muy limpio y no tenemos que preocuparnos por convertir la *Entidad* a DTO. Pero qué pasaría si ahora nos mandan el Usuario como DTO para actualizarlo en la base de datos:

```
1. public void save(UserDTO userDto) {  
2.     UserConverter converter = new UserConverter();  
3.     User userEntity = converter.fromDto(userDto);  
4.     userDao.save(userEntity);  
5. }
```

Nuevamente vemos como el *Converter* nos ha salvado de nuevo, pues nos olvidamos de la lógica de conversión. Pero, que pasaría ahora si en lugar de buscar un solo usuario, necesitamos que nos regrese todos los usuarios, bueno, estos nos obligaría a retornar una lista en lugar de un solo usuario, por lo que tendríamos que implementar una lógica para convertir listas. Lo primero que se nos puede ocurrir es crear un *for* en el servicio anterior y convertir cada uno de los objetos, crear una nueva lista a partir de los resultados y retornar la lista, pero esto traerá el mismo problema que hablamos al inicio, pues tendríamos que repetir este proceso cada vez que necesitemos convertir una lista. Por suerte contamos con una clase base llamada *AbstractConverter*, ¿la recuerdas?

Que te parece si entonces aprovechamos esta clase base para agregar los métodos de conversión de listas y de esta forma, se podrán utilizar en todos los *Converters*:

```
1. public abstract class AbstractConverter<E,D> {  
2.  
3.     public abstract E fromDto(D dto);  
4.  
5.     public abstract D fromEntity(E entity);  
6.  
7.     public List<E> fromDto(List<D> dtos){  
8.         if(dtos == null) return null;  
9.         return dtos.stream()  
10.        .map(dto -> fromDto(dto)).collect(Collectors.toList());  
11.    }  
12.  
13.    public List<D> fromEntity(List<E> entities){  
14.        if(entities == null) return null;  
15.        return entities.stream()  
16.        .map(entity -> fromEntity(entity)).collect(Collectors.toList());  
17.    }  
18. }
```

Observa que hemos agregado dos nuevos métodos, *fromDto* y *fromEntity* los cuales se encargan de convertir las listas aprovechando los métodos abstractos previamente definidos, de esta forma, en cuanto creamos una implementación de *AbstractConverter* tendremos de forma automática los métodos para convertir listas. Ahora veamos como quedaría el ejemplo de un servicio de consulta de todos los usuarios:

```
1. public List<UserDTO> findAllUsers() {  
2.     List<User> users = userDAO.findAllUsers();  
3.     UserConverter converter = new UserConverter();  
4.     return converter.fromEntity(users);  
5. }
```

Observemos que esta vez solo hace falta pasar la lista al *Converter* para que este se encargue de iterar la lista y crearnos una nueva con todos los datos convertidos.

Ya solo nos quedaría hacer un ejemplo de convertir una lista de DTO a Entity, pero creo que ya está de más, creo que con lo que hemos explicado se demuestra perfectamente el punto.

Composición de convertidores

Otra de las ventajas de los convertidores es que podemos utilizar un *Converter* dentro de otro, por ejemplo, imagina que tienes una Entidad *Invoice* (factura) la cual tiene asociado al usuario que la creo:

```
1. public class Invoice {  
2.  
3.     private Long id;  
4.     private User user;  
5.     private List<Lines> lines;  
6. }
```

No te voy a aburrir nuevamente con toda la explicación, así que solo nos enfocaremos en las novedades. Dicho esto, observar que entre todos los campos que puede tener una factura, tiene una referencia al Usuario, por lo que en el converter de la factura podríamos implementar nuevamente la lógica para convertir al usuario, pero eso sería estúpido, pues ya tenemos una clase que lo

hace, por lo tanto, podríamos utilizar el converter del usuario dentro del converter de la factura:

```
1. public class InvoiceConverter extends AbstractConverter<Invoice, InvoiceDTO>{
2.
3.     private final UserConverter userConverter = new UserConverter();
4.
5.     @Override
6.     public Invoice fromDto(InvoiceDTO dto) {
7.
8.         Invoice invoice = new Invoice();
9.         invoice.setUser(userConverter.fromDto(dto.getUser()));
10.        // ... other setters
11.        return invoice;
12.    }
13.
14.    @Override
15.    public InvoiceDTO fromEntity(Invoice entity) {
16.        InvoiceDTO invoice = new InvoiceDTO();
17.        invoice.setUser(userConverter.fromEntity(entity.getUser()));
18.        // ... other setters
19.        return invoice;
20.    }
21. }
```

En este nuevo *converter* podemos ver que hemos instanciado a *UserConverter* y lo hemos utilizado para convertir el usuario en lugar de volver a implementar la lógica de conversión aquí.

DTO en el mundo real

En este punto ya nos debería de quedar claro cómo funciona el patrón DTO, o al menos en la teoría, por lo que en esta sección analizaremos como hemos utilizado el patrón DTO en nuestra aplicación e-commerce.

Login y validación de los tokens

Puede que no lo parezca, pero estamos utilizando el DTO desde el primer momento en que utilizamos la aplicación, y me refiero al *login*. Cuando entramos por primera vez o si cerramos la sesión, la aplicación nos llevará a la página de inicio de sesión, la cual se ve de la siguiente forma:

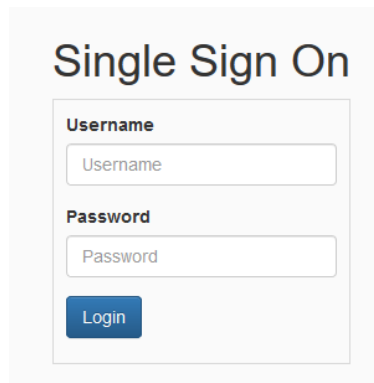
A screenshot of a web form titled "Single Sign On". The form is contained within a light gray border. It has two input fields: the first is labeled "Username" and the second is labeled "Password". Below these fields is a blue button with the text "Login" in white.

Fig 152 - Formulario de inicio de sesión.

Observa que cuando entramos a la página del e-commerce (<http://localhost:3000>) esta nos lleva de forma automática a otra URL:

`http://localhost:8080/api/security/sso?redirect=http://localhost:3000`

Ahora bien, si observamos que microservicio está utilizando este puerto, nos podremos dar cuenta es que el *api-gateway*:

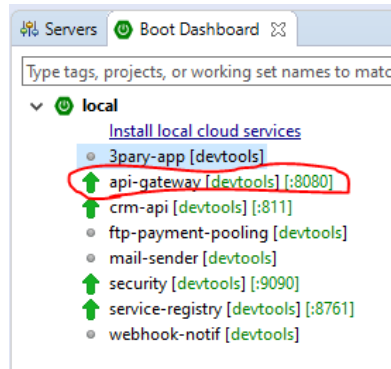


Fig 153 - api-gateway.

No quiero entrar en detalles del *api-gateway* en este momento, porque lo abordaremos de lleno más adelante, por lo que solo te contaré que este microservicio es la puerta de entrada a todos los microservicios de la aplicación, por lo tanto, el contexto */api/security* lo que hace es redireccionar las llamadas al microservicio *security*.

Dicho lo anterior, entonces podemos observar claramente que hemos salido de la aplicación e-commerce y para ser atendidos por el microservicio *security*. Una vez que queda claro este punto, quiero que abramos el archivo *sso.jsp* ubicado en este mismo microservicio:

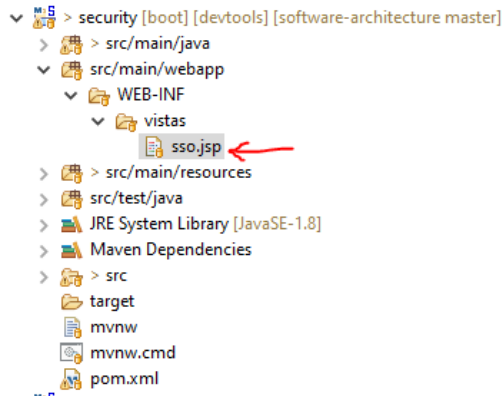


Fig 154 - Archivo sso.jsp

Dentro de este archivo tenemos un formulario que envía los datos del *login* al microservicio *security*:

```

1. <form id="login" method="POST"
2.     action="http://localhost:8080/api/security/login" >
3.     <div class="form-group">
4.         <label for="username">Username</label>
5.         <input type="text" class="form-control"
6.             id="username" name="username" placeholder="Username">
7.     </div>
8.     <div class="form-group">
9.         <label for="password">Password</label>
10.        <input type="password" class="form-control"
11.            id="password" name="password" placeholder="Password">
12.    </div>
13.    <button type="submit" class="btn btn-primary">Login</button>
14. </form>

```

Quiero que observes que cuando damos click en el botón "Login", se realiza una petición POST a la URL <http://localhost:8080/api/security/login>, la cual es atendida por el método `login` de la clase `SecurityREST`:

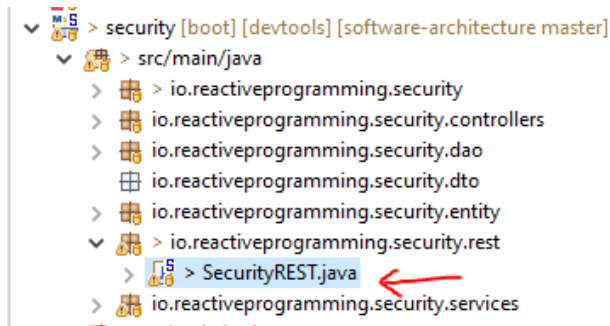


Fig 155 - Clase SecurityREST .

```

1. @PostMapping(path="login")
2. public ResponseEntity<WrapperResponse<LoginResponseDTO>> login(
3.     @RequestBody LoginDTO loginDTO) {
4.     try {
5.         LoginResponseDTO response = this.securityService.login(loginDTO);
6.         return ResponseEntity.ok(new WrapperResponse(true, "", response));
7.     } catch (Exception e) {
8.         return ResponseEntity.ok(new WrapperResponse(false, e.getMessage()));
9.     }
10. }

```

No quiero entrar en tecnicismos de Java, pues no es el objetivo de este libro, por lo que solo centrémonos en el parámetro de entrar que es de tipo *LoginDTO* y la respuesta de tipo *LoginResponseDTO*.

La definición de estos DTO's es la siguiente:

```
1. public class LoginDTO {  
2.  
3.     private String username;  
4.     private String password;  
5.     private String token;  
6. }
```

Y

```
1. public class LoginResponseDTO {  
2.     private String username;  
3.     private String rol;  
4.     private String token;  
5.     private String email;  
6. }
```

Quiero que observes que hemos utilizado dos DTO diferentes, uno para la entrada y otro para la respuesta, esto con la finalidad de tener un mejor control sobre los parámetros de entra y los campos que regresaremos al cliente.

Ahora bien, si observas los campos *username* y *password* del DTO *LoginDTO* te podrás dar cuenta que coinciden con los nombres de los *inputs* del archivo *sso.jsp*, lo que hace que se "seten" automáticamente al momento de llegar la petición al método *Login*.

También te podrás dar cuenta que esta misma clase tiene un atributo *token*, el cual lo utilizamos cuando ya nos hemos autenticado con la aplicación y en lugar de manda un usuario/password, enviamos un token que valida nuestra identidad (Abordaremos el tema de los tokens más adelante).

Internamente el microservicio *security* tiene una Entidad llamada *User*, la cual tiene la siguiente estructura:

```
1. @Entity  
2. @Table(name="USERS")  
3. public class User {
```

```

4.
5.     @Id
6.     @Column(name="USERNAME", length=50)
7.     private String username;
8.
9.     @Column(name="PASSWORD", nullable=false, length=50)
10.    private String password;
11.
12.    @Enumerated(EnumType.STRING)
13.    @Column(name="ROL", nullable=false)
14.    private Rol rol;
15.
16.    @Column(name="EMAIL", nullable=false, length=100)
17.    private String email;
18. }

```

Quiero que observes que esta entidad tiene una serie de metadatos que le indican contra que tabla mapea (*USERS*) y cada propiedad de la clase indica contra que columna mapea. También quiero que observes la pureza de esta clase, es decir, no está contaminada con ninguna propiedad que no obedezca a las necesidades de persistencia.

Si vamos a la tabla *USERS* de la base de datos *security* podremos observar que todos los atributos de la tabla corresponden con una columna de la base de datos:

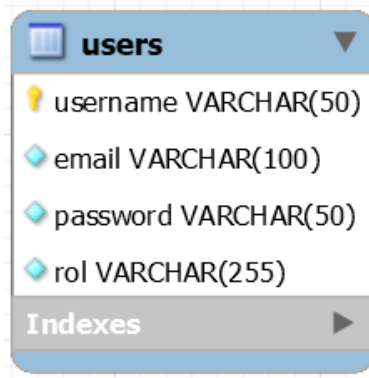


Fig 156 - Tabla users

Respecto a la respuesta del método *Login*, podemos observar que la respuesta es de tipo *LoginResponseDTO*, el cual es casi idéntico en estructura con la Entidad *User*, con la única diferencia de que el atributo *rol*, pasa de ser una enumeración a un String en el DTO.

Consulta de productos

Otro punto donde utilizamos DTO es para consultar los productos de la página principal, por lo cual, cuando entramos a la aplicación se cargan todos los productos desde la URL:

`http://localhost:8080/api/crm/products`

Nuevamente, los productos son consultados por medio del *api-gateway* (puerto 8080) pero esta vez, las peticiones son redireccionadas al microservicio *crm-api*. La última parte de la URL (*/products*) indica que la petición debe de ser atendida por el método *getALLProducts* de la clase *ProductREST*:

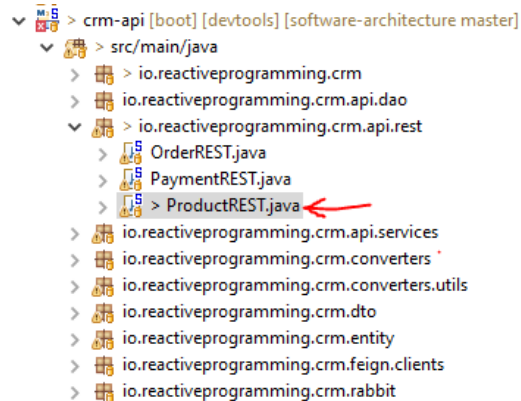


Fig 157 – Clase *ProductREST*

```
1. @GetMapping
2. public ResponseEntity<WrapperResponse<ProductDTO>> getAllProducts() {
3.     try {
4.         List<ProductDTO> products = productService.getAllProducts();
5.         WrapperResponse response =
6.             new WrapperResponse(true, "Consulta exitosa", products);
7.         return new ResponseEntity<>(response, HttpStatus.OK);
8.     } catch (Exception e) {
9.         e.printStackTrace();
10.        logger.error(e.getMessage());
11.        WrapperResponse response =
12.            new WrapperResponse(false, "Error al consultar los productos");
13.        return new ResponseEntity<>(response, HttpStatus.OK);
14.    }
```

15. }

Podemos observar que en la línea 4 hacemos una llamada a otro método con el mismo nombre, pero del objeto *productService*, el cual es la clase que encapsula la lógica de negocio, si abrimos este método veremos lo siguiente:

```
1. public List<ProductDTO> getAllProducts() throws Exception{
2.     try {
3.         ProductConverter converter= new ProductConverter();
4.         return converter.toDtoList(productDAO.findAll());
5.     } catch (Exception e) {
6.         e.printStackTrace();
7.         throw new Exception(e.getMessage(),e);
8.     }
9. }
```

Podrás observar que en la línea 3 estamos creando una instancia de la clase *ProductConverter*, la cual es un convertidor entre la entidad *Product* y el DTO *ProductDTO*. Finalmente, en la línea usamos un DAO (lo analizaremos en el siguiente capítulo) para ir a la base de datos y obtener todos los productos, finalmente, el resultado lo convertimos con el *converter*.

Consulta de mis ordenes

Otro ejemplo de donde utilizamos un DTO es para retornar las ordenes, ya sea una orden en particular o todas las ordenes de un usuario determinado, en cualquier caso, utilizamos el mismo DTO.

Cuando consultamos una orden en específico utilizamos el método *getOrder* y cuando consultamos todas las ordenes utilizamos el método *getAllOrders*, ambos métodos de la clase *OrderREST*.

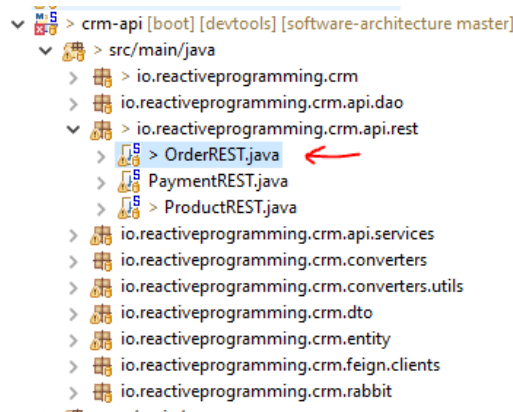


Fig 158 - Clase OrderREST.

```
1. @RequestMapping(value = "{orderId}", method = RequestMethod.GET)
2. public WrapperResponse<SaleOrderDTO> getOrder(
3.     @PathVariable("orderId") Long orderId) {
4.     try {
5.         SaleOrderDTO newOrder = orderService.findSaleOrderById(orderId);
6.         WrapperResponse response = new
7.             WrapperResponse(true, "success", newOrder);
8.         return response;
9.     } catch(ValidateServiceException e) {
10.        return new WrapperResponse(false, e.getMessage());
11.    } catch (Exception e) {
12.        e.printStackTrace();
13.        return new WrapperResponse(false, "Internal Server Error");
14.    }
15. }
16.
17. @GetMapping()
18. public WrapperResponse getAllOrders() {
19.     try {
20.         List<SaleOrderDTO> orders = orderService.getAllOrders();
21.         return new WrapperResponse(true, "success", orders);
22.     } catch(ValidateServiceException e) {
23.         return new WrapperResponse(false, e.getMessage());
24.     } catch (Exception e) {
25.         e.printStackTrace();
26.         return new WrapperResponse(false, "Internal Server Error");
27.     }
28. }
```

Como la orden es un objeto complejo, creamos un *converter* que nos ayude a convertir la orden, y ese *converter* lo reutilizamos para convertir ya sea una sola orden o una lista de ordenes. El *converter* se ve de la siguiente manera:

```
1. public class SaleOrderConverter extends AbstractConverter<SaleOrder, SaleOrderD
   TO> {
2.
3.     private DateFormat dateFormat =
4.         new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");
5.
6.     @Override
7.     public SaleOrder toEntity(SaleOrderDTO dto) {
8.
9.         SaleOrder saleOrder = new SaleOrder();
10.        saleOrder.setCustomerEmail(dto.getCustomerEmail());
11.        saleOrder.setCustomerName(dto.getCustomerName());
12.        saleOrder.setId(dto.getId());
13.        saleOrder.setRefNumber(dto.getRefNumber());
14.        saleOrder.setTotal(dto.getTotal());
15.        saleOrder.setStatus(OrderStatus.valueOf(dto.getStatus()));
16.
17.        try {
18.            Calendar registDate = Calendar.getInstance();
19.            registDate.setTime(dateFormat.parse(dto.getRegistDate()));
20.            saleOrder.setRegistDate(registDate);
21.        } catch (Exception e) {
22.            e.printStackTrace();
23.        }
24.
25.
26.        ProductConverter productConverter = new ProductConverter();
27.        if(dto.getOrderLines() != null) {
28.            saleOrder.setOrderLines(new HashSet<>());
29.            for(OrderLineDTO lineDTO : dto.getOrderLines()) {
30.                OrderLine line = new OrderLine();
31.                line.setId(lineDTO.getId());
32.                line.setProduct(
33.                    productConverter.toEntity(lineDTO.getProduct()));
34.                line.setQuantity(lineDTO.getQuantity());
35.                line.setSaleOrder(saleOrder);
36.                saleOrder.getOrderLines().add(line);
37.            }
38.        }
39.
40.        return saleOrder;
41.
42.    }
43.
44.    @Override
45.    public SaleOrderDTO toDTO(SaleOrder entity) {
46.        SaleOrderDTO saleOrderDTO = new SaleOrderDTO();
```

```

47.     saleOrderDTO.setCustomerEmail(entity.getCustomerEmail());
48.     saleOrderDTO.setCustomerName(entity.getCustomerName());
49.     saleOrderDTO.setId(entity.getId());
50.     saleOrderDTO.setRefNumber(entity.getRefNumber());
51.     saleOrderDTO.setTotal(entity.getTotal());
52.     saleOrderDTO.setRegistDate(
53.         dateFormat.format(entity.getRegistDate().getTime()));
54.     saleOrderDTO.setStatus(entity.getStatus().toString());
55.
56.     ProductConverter productConverter = new ProductConverter();
57.     if(entity.getOrderLines() != null) {
58.         saleOrderDTO.setOrderLines(new HashSet<>());
59.         for(OrderLine line : entity.getOrderLines()) {
60.             OrderLineDTO lineDTO = new OrderLineDTO();
61.             lineDTO.setId(line.getId());
62.             lineDTO.setProduct(
63.                 productConverter.toDTO(line.getProduct()));
64.             lineDTO.setQuantity(line.getQuantity());
65.             saleOrderDTO.getOrderLines().add(lineDTO);
66.         }
67.     }
68.
69.     DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");
70.
71.     Payment payment = entity.getPayment();
72.     if(payment != null) {
73.         PaymentDTO paymentDTO = new PaymentDTO();
74.         paymentDTO.setId(payment.getId());
75.         paymentDTO.setPaymentMethod(payment.getPaymentMethod().name());
76.         if(payment.getPaydate() != null)
77.             paymentDTO.setPaydate(
78.                 dateFormat.format(payment.getPaydate().getTime()));
79.
80.         saleOrderDTO.setPayment(paymentDTO);
81.     }
82.
83.     return saleOrderDTO;
84. }
85. }

```

Puedes observar en las líneas 26 y 56 que estamos reutilizando el *ProductConverter* para convertir todos los productos que contiene la orden, demostrando que podemos reutilizar los convertidores en más de una parte de nuestro proyecto.

Conclusiones

Como hemos podido demostrar, DTO es un patrón muy efectivo para transmitir información entre un cliente y un servidor, pues permite crear estructuras de datos independientes de nuestro modelo de datos (Entidades), lo que nos permite crear cuantas “*vistas*” sean necesarias de un conjunto de tablas u orígenes de datos. Además, nos permite controlar el formato, nombre y tipos de datos con los que transmitimos los datos para ajustarnos a un determinado requerimiento. Finalmente, si por alguna razón, el modelo de datos cambio (y con ello las entidades) el cliente no se afectará, pues seguirá recibiendo el mismo DTO.

Por otro lado, analizamos como el patrón *Converter* es de gran ayuda para evitar la repetición de código que puede ser tedioso y propenso a errores, además, vemos que mediante la composición podemos reutilizar los convertidores para crear convertidores más avanzados.

Solo me resta mencionar un detalle importante, y es que debemos de tener cuidado que datos regresamos al cliente, ya que, por ejemplo, regresar el password de nuestros usuarios a nuestros clientes puede ser una mala idea, así que podemos omitir este campo en el *converter* o eliminarlo de los objetos una vez que ha sido convertido, todo dependerá de que te funcione mejor.

Data Access Object (DAO)

Prácticamente todas las aplicaciones de hoy en día, requiere acceso al menos a una fuente de datos, dichas fuentes son por lo general base de datos relacionales, por lo que muchas veces no tenemos problema en acceder a los datos, sin embargo, hay ocasiones en las que necesitamos tener más de una fuente de datos o la fuente de datos que tenemos puede variar, lo que nos obligaría a refactorizar gran parte del código. Para esto, tenemos el patrón Arquitectónico Data Access Object (DAO), el cual permite separar la lógica de acceso a datos de los Objetos de negocios (Business Objects), de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Problemática

Una de las grandes problemáticas al momento de acceder a los datos, es que la implementación y formato de la información puede variar según la fuente de los datos, además, implementar la lógica de acceso a datos en la capa de lógica de negocio puede ser un gran problema, pues tendríamos que lidiar con la lógica de negocio en sí, más la implementación para acceder a los datos, adicional, si tenemos múltiples fuentes de datos o estas pueden variar, tendríamos que implementar las diferentes lógicas para acceder las diferentes fuentes de datos, como podrían ser: bases de datos relacionales, No SQL, XML, archivos planos, servicios web o REST, etc).

Un programador inexperto o con pocos conocimientos sobre arquitectura podría optar por crear en un solo método la lógica de negocio y la lógica de acceso a datos, creando un método muy difícil de entender y mantener.

Solo imagina que necesitas hacer un servicio que cree un nuevo usuario, para lo cual, se tendrá que validar que el nombre de usuario no esté repetido antes de permitir la creación del usuario:

```
1. public class UserService {  
2.  
3.     public void createUser(NewUserRequestDTO dto) {  
4.         Connection connection =  
5.             DriverManager.getConnection("connection url ...");  
6.
```

```

7.      Statement queryStm = connection.createStatement();
8.      PreparedStatement usernameStm = connection.prepareStatement(
9.          "select 1 from users u where u.username = ?");
10.     usernameStm.setString(1, dto.getUsername());
11.     ResultSet usernameResult = usernameStm.executeQuery();
12.     if(usernameResult.next()) {
13.         throw new RuntimeException("Username already exists");
14.     }
15.
16.     connection.setAutoCommit(false);
17.     PreparedStatement stm = connection.prepareStatement(
18.         "insert into users(id,username,password) values (?,?,?)");
19.     stm.setLong(1, dto.getId());
20.     stm.setString(2, dto.getUsername());
21.     stm.setString(3, dto.getPassword());
22.     stm.executeUpdate();
23.     connection.commit();
24. }
25. }

```

Observa que las únicas líneas que obedecen a la lógica de negocio son de la 12 a la 14, donde validamos que no esté repetido el usuario, y podríamos decir que la 22 donde guardamos el nuevo usuario, todo lo demás es solo para establecer la conexión a la base de datos, y eso que solo estamos consultando un registro e insertando otro, imagina si esta operación fuera una transacción más grande que involucrar varias validaciones y entidades; seguramente este método crecería exponencialmente de tamaño y la complejidad para comprenderlo y darle mantenimiento también crecería.

Ahora bien, imagina ahora que la aplicación puede cambiar la fuente de datos o esta puede cambiar dinámicamente mediante la configuración, lo que nos llevaría primero que nada determinar que fuente de datos estamos utilizando y según esa fuente, realizar la persistencia:

```

1.  public void createUser(NewUserRequestDTO dto) {
2.      if(CONFIG.DB_TYPE == "MySQL") {
3.          Connection connection =
4.              DriverManager.getConnection("connection url ...");
5.
6.          Statement queryStm = connection.createStatement();
7.          PreparedStatement usernameStm = connection.prepareStatement(
8.              "select 1 from users u where u.username = ?");
9.          usernameStm.setString(1, dto.getUsername());
10.         ResultSet usernameResult = usernameStm.executeQuery();

```

```

11.         if(usernameResult.next()) {
12.             throw new RuntimeException("Username already exists");
13.         }
14.
15.         connection.setAutoCommit(false);
16.         PreparedStatement stm = connection.prepareStatement(
17.             "insert into users(id,username,password) values (?,?,?)");
18.         stm.setLong(1, dto.getId());
19.         stm.setString(2, dto.getUsername());
20.         stm.setString(3, dto.getPassword());
21.         stm.executeUpdate();
22.         connection.commit();
23.     }else if(CONFIG.DB_TYPE == "MongoDB"){
24.         //MongoDB logic
25.     }
26. }

```

Observa cómo en la línea 2 preguntamos qué origen de datos estamos utilizando y basado en eso, realizamos una acción diferente. Podrás ver que esto ya comienza a salirse de control y eso que la lógica de negocio es muy simple.

Solución

Dado lo anterior, el patrón Data Access Object (DAO) propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

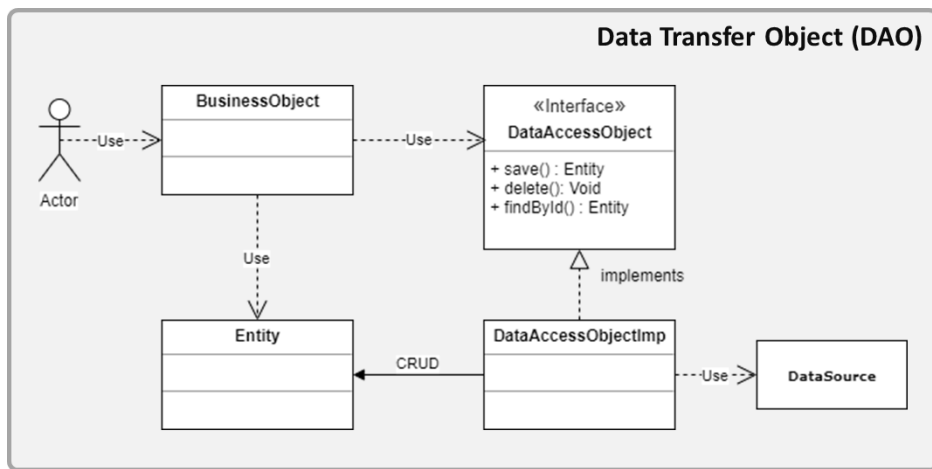


Fig 159 - Diagrama de clases del patrón DAO.

Los compones que conforman el patrón son:

- **Actor**: es el usuario o sistema que inicia la solicitud para consultar o guardar algo en la base de datos.
- **BusinessObject**: representa un objeto con la lógica de negocio, este cuenta con operaciones de alto nivel que el usuario puede ejecutar.
- **DataAccessObject**: representa la interface que los DAO tendrán que implementar, con la intención de ocultar al *BusinessObject* la instancia concreta que estamos utilizando.
- **DataAccessObjectImp**: representa una implementación concrete de la interface *DataAccessObject*. Debería de existir una implementación concreta por cada Entidad y fuente de datos existente.
- **Entidad**: representa una clase que mapea contra un objeto de dominio de la aplicación, el cual necesita ser persistente, ya sea en una base de datos SQL o No SQL.
- **DataSource**: representa de forma abstracta una conexión a la fuente de datos (comúnmente una base de datos).

Ya conocemos los componentes que forman el patrón, pero ahora debemos de comprender cual es la secuencia.

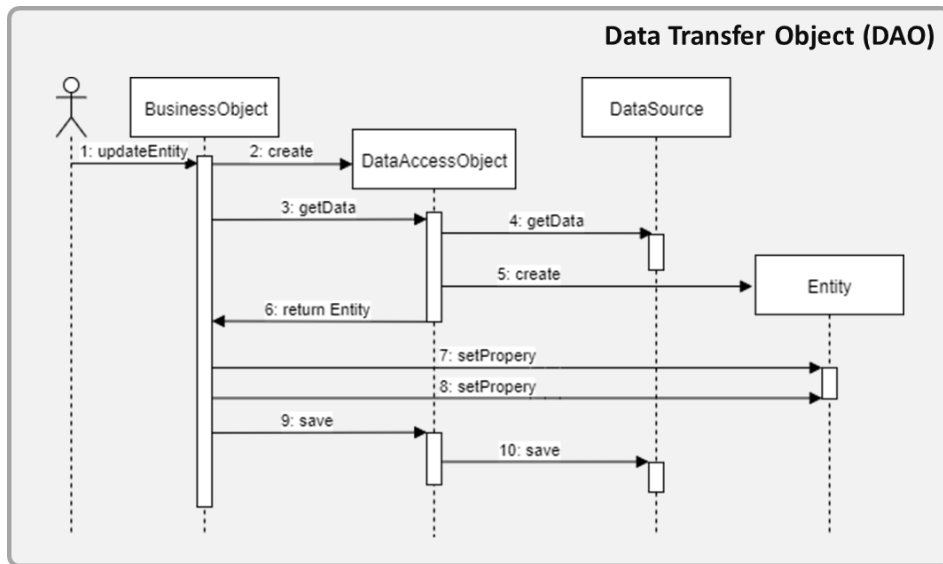


Fig 160 - Diagrama de secuencia del patrón DAO.

El diagrama anterior representa la secuencia de pasos para actualizar una Entidad existente mediante el uso del patrón DAO.

1. Un usuario o sistema solicita la actualización de una *Entidad*
2. El *BusinessObject* crea una instancia concreta del DAO según la fuente de datos utilizada (recuerda que pueden existir un DAO por cada fuente de datos diferente).
3. El *BusinessObject* consulta la Entidad a actualizar mediante el *DataAccessObject*.
4. El *DataAccessObject* solicita la información al *DataSource*, el cual según su implementación concreta sabe contra que fuente de datos realizar la consulta.
5. El *DataAccessObject* crea una instancia del *Entidad* con los datos recuperados del *DataSource*.
6. El *DataAccessObject* retorna los datos como una Entidad.
7. El *BusinessObject* actualiza algún valor de la Entidad.
8. Más actualizaciones
9. El *BusinessObject* solicita el guardado de los datos actualizados al *DataAccessObject*.
10. El *DataAccessObject* guarda los datos en el *DataSource*.

Como hemos podido ver, el *BusinessObject* no se preocupa de donde vengan los datos ni cómo deben de ser guardados, el solo se preocupa por saber que método se deben de utilizar para consultar o actualizar la Entidad, pues será el DAO quien se preocupe por persistir adecuadamente la información. Por otro lado, el que desarrolla la capa de servicio no se tiene que preocupar por el API de bajo nivel necesario para realizar esta persistencia.

Un error común al implementar este patrón es no utilizar Entidades y en su lugar, regresar los objetos que regresan las mismas API's de las fuentes de datos, ya que esto obliga al *BusinessObject* tener una dependencia con estas librerías, además, si la fuente de datos cambia, también cambiarán los tipos de datos, lo que provocaría una afectación directa al *BusinessObject*.

Si aplicamos el nuevo conocimiento adquirido para actualizar nuestro servicio de creación de usuario, podemos ver una clara diferencia:

```
1. public void createUser(NewUserRequestDTO dto) {  
2.  
3.     UserDao userDao = new UserDaoImpl();  
4.  
5.     if(userDao.findByUsername(dto.getUsername()) != null) {  
6.         throw new RuntimeException("Username already exists");  
7.     }  
8.  
9.     UserConverter usrConverter = new UserConverter();  
10.    User user = usrConverter.fromDto(dto);  
11.  
12.    userDao.save(user);  
13. }
```

Observa el cambio significativo que hemos logrado utilizando un DAO, podemos ver que en lugar de preocuparnos por cómo crear la conexión a la base de datos y como consultar y guardar los datos, solo nos centramos en las reglas de negocio, por lo tanto, el código se hace mucho más fácil de entender y mantener.

Si solo tenemos una fuente de datos esta podría ser la solución definitiva, sin embargo, seguimos teniendo un problema, y es como sabemos exactamente que instancia del DAO instanciar, por ejemplo, en la línea 3 estamos creando una instancia de *UserDaoImpl* en código duro, por lo que si queremos cambiar de fuente de datos tendríamos que cambiar el código y con ello, un nuevo despliegue forzado, por ello, es inteligente utilizar el DAO en conjunto con el patrón Abstract Factory para que nos ayude a determinar y crear la instancia correcta del DAO según nuestra configuración.

DAO y el patrón Abstract Factory

Hasta este punto solo hemos analizado como trabajaríamos si solo tuviéramos una fuente de datos, sin embargo, existe ocasiones donde requerimos obtener datos de más de una fuente, y es allí donde entra el patrón de diseño Abstract Factory el cual explico perfectamente en mi libro "[Introducción a los patrones de diseño](#)".

Mediante el patrón Abstract Factory podemos definir una serie de familias de clases que permitan conectarnos a las diferentes fuentes de datos. Para esto, examinaremos un sistema de autenticación de usuarios, el cual puede leer los usuarios en una base de datos o sobre un XML, adicional, el sistema generara registros de login que podrán ser utilizados para auditorias.

Lo primero sería implementar las familias de clases para acceder de las dos fuentes:

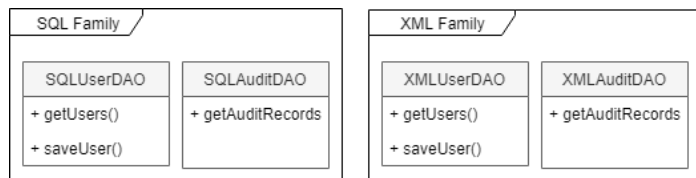


Fig 161 - Familias de clases.

En la imagen anterior podemos apreciar dos familias de clases, con las cuales podemos obtener los Usuarios y los registros de auditoria, sin embargo, estas clases por separado no ayudan mucho, pues no implementan una misma interface que permita la variación entre ellas, por lo que el siguiente paso es crear estas interfaces:

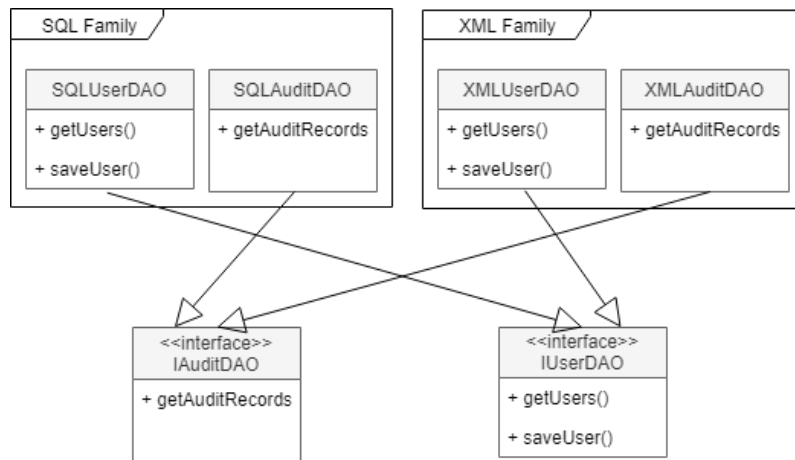


Fig 162 - Implementando interfaces en común entre las familias.

En este punto, los DAO ya implementan una interfaz común, lo que permite intercambiar la implementación sin afectar al *BusinessObject*. Sin embargo, ahora solo falta resolver la forma en que el *BusinessObject* obtendrá la familia de interfaces, es por ello que deberemos crear un Factory para cada familia de interfaces:

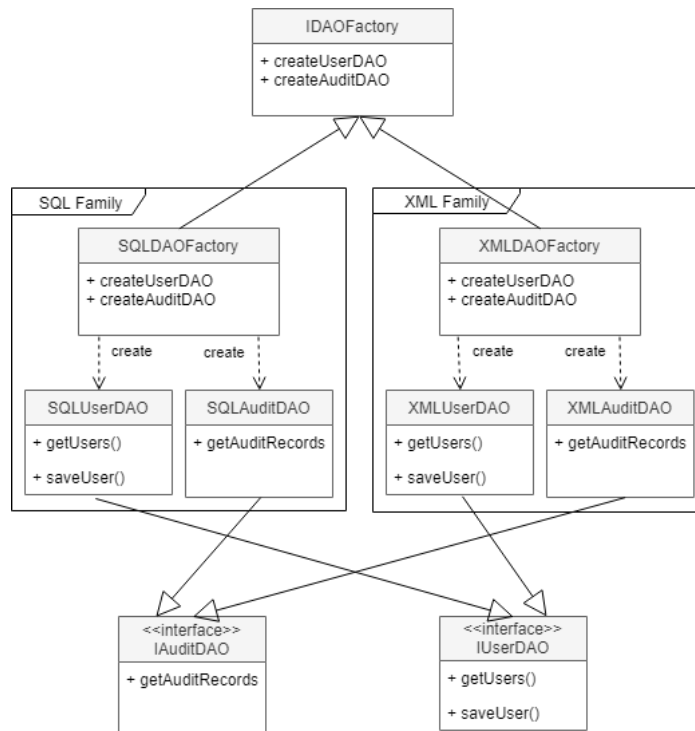


Fig 163 - Implementado factorías concretas para cada familia.

En esta nueva configuración, podemos ver que tenemos un Factory para cada familia, y los dos factories implementan una interfaz en común, adicional, tenemos la interface *IDAOFactory* necesaria para que el factory de cada familia implementen una interface en común.

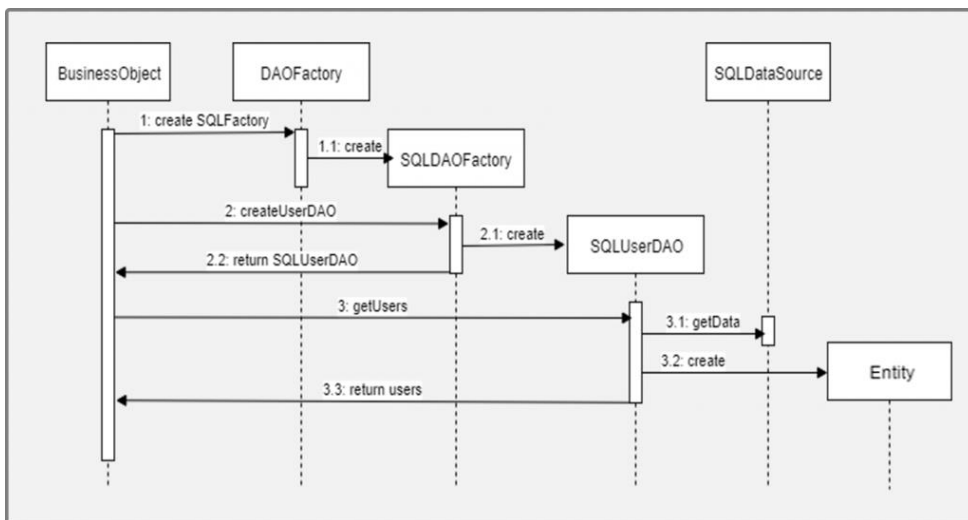


Fig 164 - Diagrama de secuencia con Abstract Factory.

Analicemos como quedaría la secuencia de ejecución

1. El *BusinessObject* solicita la creación de un *DAOFactory* para SQL
2. El *DAOFactory* crea una instancia de la clase *SQLDAOFactory* y la retorna
3. El *BusinessObject* solicita al *SQLDAOFactory* la creación del *SQLUserDAO* para interactuar con los usuarios.
4. El *SQLDAOFactory* crea una nueva instancia del *SQLUserDAO*
5. El *SQLDAOFactory* retorna la instancia creada del *SQLUserDAO*
6. El *BusinessObject* solicita el listado de todos los usuarios registrados al *SQLUserDAO*
7. El *SQLUserDAO* recupera los usuarios del *SQLDataSource*
8. El *SQLUserDAO* crea un *Entidad* con los datos recuperados del paso anterior.
9. El *SQLUserDAO* retorna la *Entidad* creado en el paso anterior.
10. Adicional a los pasos que hemos listado aquí, podríamos solicitar al *SQLDAOFactory* la creación del *SQLAuditDAO* o incluso, solicitar al

DAOFactory la creación del *XMLFactory* para interactuar con la fuente de datos en XML.

DAO en el mundo real

En la sección anterior analizábamos como el patrón DAO nos permite separar la capa de datos de la lógica de negocio, al mismo tiempo que nos permite variar la fuente de datos sin afectar a la capa de negocio, mediante la implementación de una interface común que comparten los DAO's.

En esta sección analizaremos como implementamos el patrón DAO dentro de nuestro proyecto e-commerce y como este nos da una ventaja sustancia y un código más limpio.

Por fortuna, *Spring Boot*, la tecnología que estamos utilizando para crear nuestro proyecto cuenta con un framework muy potente para crear los DAO, lo que nos permite implementar una gran cantidad de funcionalidad con unas cuantas líneas de código, por lo que te sorprenda si un DAO lo podemos implementar con solo una interface.

La teoría dice que deberíamos tener al menos un DAO por cada Entidad de nuestro proyecto, con la finalidad de que el DAO controle el acceso a los datos únicamente de esa Entidad, así que analizaremos las Entidades que tenemos en alguno de nuestros Microservicios.

Microservicio security

Si recordamos, el Microservicio *security* sirve para controlar el acceso de los usuarios, por lo tanto, es el encargado de administrar la Entity *User*, la cual se ve de la siguiente manera:

```
1. @Entity
2. @Table(name="USERS")
3. public class User {
4.
5.     @Id
6.     @Column(name="USERNAME", length=50)
7.     private String username;
8.
9.     @Column(name="PASSWORD", nullable=false, length=50)
10.    private String password;
11.
12.    @Enumerated(EnumType.STRING)
```

```

13.     @Column(name="ROL", nullable=false)
14.     private Rol rol;
15.
16.     @Column(name="EMAIL", nullable=false, length=100)
17.     private String email;
18.
19.     /** GETs and SETs */
20. }

```

La Entidad User por lo tanto debería de tener una clase DAO que permite interactuar con la base de datos:

```

1. @Repository
2. public interface IUserDAO extends JpaRepository<User, String>{
3.
4. }

```

No te dejes engañar por la interface anterior, ya que la Interface *JpaRepository* tiene una gran cantidad de método que permite interactuar con las Entidad, entre los que destacan las operaciones CRUD (Altas, Bajas, Cambios, Borrado) sobre un determinado tipo de Entidad.

Para que la interface *JpaRepository* funcione, requiere de definir dos tipos genéricos, los cuales podemos ver entre *<User, String>*, el primer parámetro corresponde al tipo de datos de la Entidad y el segundo, corresponde el tipo de datos de la llave primaria, así que si regresamos a la Entidad User podrás ver que la llave primaria es el campo *username*, marcado con el metadato *@Id*.

Por otro lado, el metadato *@Repository* se utiliza para que Spring Boot pueda detectar que se trata de un DAO al momento de construir la aplicación y pueda inyectar las dependencias, como es el caso de las conexiones y el contexto de persistencia (Analizaremos la inyección de dependencias más adelante).

Podrás ver la definición de estas dos clases dentro del proyecto *security*:

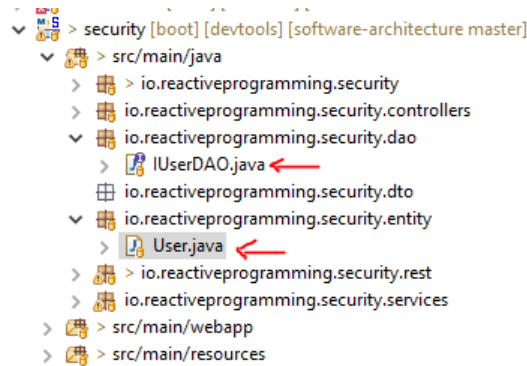


Fig 165 - Entidad User y DAO IUserDAO.

Con el DAO *IUserDAO* implementado, procederemos a utilizarlo desde el servicio de autenticación, el cual podemos ubicar en la clase *SecurityService* del mismo proyecto:

```
1. @Service
2. public class SecurityService {
3.
4.     private static final Logger logger = Logger.getLogger(SecurityService.class);
5.
6.     private static final String TOKEN = "1234";
7.     private static final int TOKEN_VALIDA_TIME = 86_400_000;
8.     private static final String SUBJECT = "CRM-API";
9.
10.    @Autowired
11.    private Tracer tracer;
12.
13.    @Autowired
14.    private IUserDAO userDAO;
15.
16.    public LoginResponseDTO login(LoginDTO request)
17.        throws ValidateServiceException, GenericServiceException {
18.        try {
19.            LoginResponseDTO response = null;
20.            if(request.getToken() != null) {
21.                response = decryptToken(request.getToken());
22.                response.setToken(createJWT(response.getUsername()));
23.                return response;
24.            } else {
25.                String username = request.getUsername();
26.
27.                Optional<User> userOpt = userDAO.findById(username);
28.                if (!userOpt.isPresent()) {
29.                    throw new ValidateServiceException(
```

```

30.         "Invalid user or password");
31.     }
32.
33.     User user = userOpt.get();
34.     if(!user.getPassword().equals(request.getPassword())) {
35.         throw new ValidateServiceException(
36.             "Invalid user or password");
37.     }
38.
39.     String token = createJWT(user.getUsername());
40.
41.     response = new LoginResponseDTO();
42.     response.setUsername(user.getUsername());
43.     response.setRol(user.getRol().name());
44.     response.setEmail(user.getEmail());
45.     response.setToken(token);
46.
47.     logger.info(user.getUsername() + " authenticated");
48.     tracer.currentSpan().tag(
49.         "login", user.getUsername() + " authenticated");
50. }
51. return response;
52.
53. } catch(ValidateServiceException e) {
54.     logger.info(e.getMessage());
55.     tracer.currentSpan().tag("validate", e.getMessage());
56.     throw e;
57. } catch (Exception e) {
58.     logger.error(e.getMessage(), e);
59.     tracer.currentSpan().tag("validate",
60.         "Error al autenticar al usuario" + e.getMessage());
61.     throw new GenericServiceException("Error al autenticar al usuario");
62. }
63.
64. }
65.
66. // Other methods
67. }

```

Quiero que prestes atención en las líneas 12 y 14, pues en ellas estamos inyectando el DAO *IUserDAO* con la finalidad de poder utilizarlo más adelante.



Tip

El termino de inyección de dependencias lo analizaremos más adelante, por lo que, por ahora, puedes imaginars como que es algo mágico que instancia la clase por nosotros y luego la asigna a la variable sin necesidad de que nosotros la creamos.

Por otro lado, tenemos el método *Login*, encargado de validar las credenciales del usuario o el token, si fuera el caso. Cuando un usuario se autentica por primera vez, necesita enviar el *username* y *password*, pero de allí en adelante, solo requiere enviar el token, es por ello que vemos un *if* en la línea 20. Por ahora nos centraremos en la sección de la validación del usuario por medio del *username* y *password*, es decir, la sección del *else*.

Quiero ve observes la línea 27, en ella utilizamos el DAO para recuperar al usuario por medio del ID, es decir su *username*. Este método no salió de la nada, recuerda que la interface *IUserDAO* implementa la interface *JpaRepository*, la cual provee este método por defecto.



Tip

JpaRepository es parte del Framework de Spring, más precisamente, Spring Data, un framework ORM para controlar el acceso a datos, por lo que no encontraremos la definición de la clase en el proyecto, ya que es una librería.

Regresando al *Login*, podemos apreciar que podemos obtener el Usuario en una sola línea de código, lo cual nos permite centrarnos en la lógica de negocios.

Lo que resta es validar el password y crear el token para ser retornado como parte de una login exitoso, en otro caso, lanzamos una Excepción para regresar el error encontrado.

Solo una cosa más, te estarás preguntando como el *IUserDAO* sabe a qué base de datos conectarse, pues es simple, Spring Data utiliza por debajo JPA (Java Persistence API), el cual se auto configura mediante el archivo *application.yml*, en el cual definimos la conexión a la base de datos:

```
1. datasource:
2.   url: jdbc:mysql://${mysql.service.host}:${mysql.service.port}/${mysql.service
   .database}
3.   username: ${mysql.service.username}
4.   password: ${mysql.service.password}
5.   driver-class-name: com.mysql.jdbc.Driver
6. jpa:
7.   hibernate:
8.     ddl-auto: update
```

```

9.   generate-ddl: true
10.  show-sql: false
11.  database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
12.
13.
14. sql:
15. service:
16.   port: 3306
17.   host:   localhost
18.   database: security
19.   username: root
20.   password: 1234

```

Esto que ves arriba es solo la parte relevante para Spring Data, lo que corresponde a los datos de conexión a la base de datos, los parámetros de configuración de JPA y el datasource necesario para crear las conexiones. Con todo esto y magia (Inyección de dependencias), hacemos que le DAO funcione con el menor esfuerzo posible.

Microservicio *crm-api*

El caso del microservicio *crm-api* idéntico al anterior, solo cambian las Entitys en cuestión, en este caso tenemos varias Entidades, las cuales son: *Order*, *OrderLine*, *Payment*, *Product* y *SaleOrder*, sin embargo, solo vamos a requerir un DAO para dos de ellas:

```

1. @Repository
2. public interface IOrderDAO extends JpaRepository<SaleOrder, Long>{
3.     public SaleOrder findByRefNumber(String refNumber);
4. }

```

Y

```

1. @Repository
2. public interface IProductDAO extends JpaRepository<Product, Long>{
3.
4. }

```

Ya te estás dando cuenta que las Entidad con DAO son solamente *SaleOrder* y *Product*. Las demás entidades no requieren un DAO debido a que serán gestionada como Entidades dependientes de la Entidad *SaleOrder*, por lo tanto, al afectar esta Entidad se afectarán las demás en cascada.

Comencemos con el servicio de consulta de los productos, el cual es utilizado para mostrar todos los productos disponibles desde la página principal de la aplicación, el método es *getAllOrders* que se encuentra en la clase *OrderService*:

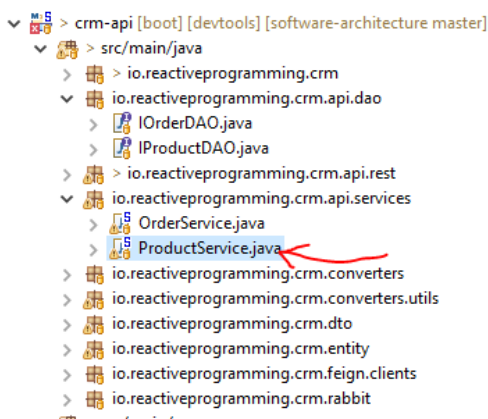


Fig 166 - Clase *OrderService*.

```
1. @Component
2. public class ProductService {
3.
4.     @Autowired
5.     private IProductDAO productDAO;
6.
7.
8.     public List<ProductDTO> getAllProducts() throws Exception{
9.         try {
10.             ProductConverter converter= new ProductConverter();
11.             return converter.toDtoList(productDAO.findAll());
12.         } catch (Exception e) {
13.             e.printStackTrace();
14.             throw new Exception(e.getMessage(),e);
15.         }
16.
17.     }
18. }
```

Observa lo limpio que es este servicio, pues en dos líneas hemos consultado y retornado una lista completa de Productos, incluyendo su conversión de Entity a DTO.

El método *findAll* que utilizamos en la línea 11 es provisto por la interface *JpaRepository*, la cual recupera todos los registros de una determinada Entidad, finalmente el resultado lo convertimos en DTO para ser enviado al consumidor del servicio.

En el caso del DAO para las ordenes, también tenemos un servicio que consulta todos los registros, solo que este está en la clase *OrderService*:

```
1. public List<SaleOrderDTO> getAllOrders() throws GenericServiceException, Valida
   teServiceException{
2.     try {
3.         List<SaleOrder> orders = orderDAO.findAll();
4.         SaleOrderConverter orderConverter = new SaleOrderConverter();
5.         List<SaleOrderDTO> ordersDTO = orderConverter.toDtoList(orders);
6.         return ordersDTO;
7.     } catch (Exception e) {
8.         throw new GenericServiceException("Internal Server Error");
9.     }
10. }
```

Nuevamente utilizamos el método *findAll* para recuperar todas las ordenes y convertirlas en DTO antes de retornarlas.

Dentro de la misma clase tenemos el método *createOrder*, es cual es ejecutado al finalizar la compra:

```
1. @HystrixCommand(fallbackMethod="queueOrder",ignoreExceptions= {ValidateServiceExcept
   ion.class})
2. public SaleOrderDTO createOrder(NewOrderDTO order) throws ValidateServiceException,
   GenericServiceException {
3.
4.     logger.info("New order request ==>");
5.     try {
6.         if(order.getOrderLines() == null || order.getOrderLines().isEmpty()) {
7.             throw new ValidateServiceException("Need one or more order lines");
8.         }
9.         PaymentMethod paymentMethod = null;
10.        try {
```



```

11.         paymentMethod = PaymentMethod.valueOf(order.getPaymentMethod());
12.     } catch (Exception e) {
13.         throw new ValidateServiceException("Invalid payment menthod");
14.     }
15.
16.     SaleOrder saleOrder = new SaleOrder();
17.
18.     Payment payment = new Payment();
19.     payment.setPaymentMethod(paymentMethod);
20.     payment.setSaleOrder(saleOrder);
21.     saleOrder.setPayment(payment);
22.
23.     if(paymentMethod == PaymentMethod.CREDIT_CARD) {
24.         CardDTO card = order.getCard();
25.         String cvc = card.getCvc();
26.         String name = card.getName();
27.         String number = card.getNumber();
28.         String expiry = card.getExpiry();
29.
30.         if(cvc == null || cvc.isEmpty()
31.            || cvc.length() < 3 || cvc.length() > 4) {
32.             throw new ValidateServiceException(
33.                 "CVC is required, is need 3 or 4 numbers");
34.         }
35.         if(expiry == null || expiry.isEmpty()
36.            || expiry.length() < 4 || expiry.length() > 4) {
37.             throw new ValidateServiceException(
38.                 "Expiry date id required, formato mm/yy");
39.         }
40.         if(name == null || name.isEmpty() || name.length() > 30) {
41.             throw new ValidateServiceException(
42.                 "Named is requiered, 30 characters max");
43.         }
44.         if(number == null || number.isEmpty()
45.            || number.length() < 15 || number.length() > 16) {
46.             throw new ValidateServiceException(
47.                 "Invalid credit card number, is need 15 or 16 numbers")
48.         ;
49.     }
50.     payment.setPaydate(Calendar.getInstance());
51.     saleOrder.setStatus(OrderStatus.PAYED);
52. }else {
53.     saleOrder.setStatus(OrderStatus.PENDING);
54. }
55.
56.     saleOrder.setCustomerEmail(order.getCustomerEmail());
57.     saleOrder.setCustomerName(order.getCustomerName());
58.     saleOrder.setRefNumber(UUID.randomUUID().toString());
59.     saleOrder.setRegistDate(Calendar.getInstance());
60.
61.     Set<OrderLine> lines = new HashSet<>();

```

```

62.         for(NewOrderLineDTO lineDTO : order.getOrderLines()) {
63.             Optional<Product> productOpt =
64.                 productDAO.findById(lineDTO.getProductId());
65.             if(!productOpt.isPresent()) {
66.                 throw new ValidateServiceException(
67.                     String.format("Product %s not found", lineDTO.getProductId()));
68.             }
69.             Product product = productOpt.get();
70.
71.             OrderLine line = new OrderLine();
72.             line.setProduct(product);
73.             line.setQuantity(lineDTO.getQuantity());
74.             line.setSaleOrder(saleOrder);
75.
76.             lines.add(line);
77.         }
78.         saleOrder.setOrderLines(lines);
79.
80.         SaleOrder newSaleOrder = orderDAO.save(saleOrder);
81.
82.         /* lines no relevance */
83.
84.         return returnOrder;
85.     } catch(ValidateServiceException e) {
86.         e.printStackTrace();
87.         throw e;
88.     } catch (Exception e) {
89.         e.printStackTrace();
90.         throw new GenericServiceException(e.getMessage(),e);
91.     }
92. }

```

Este es un método más complejo y quizás no tenga mucho sentido analizarlo por completo, por lo que solo nos centraremos en los realmente relevante, por lo que las primeras líneas son básicamente validaciones, entre las que se encuentran validar que un producto existe en la base de datos, para ello, utilizamos el método *findById* de DAO *IProductDAO* (línea 64), el cual busca un producto por medio del ID.

Una vez validados todos los campos y creadas la Entity *SaleOrder*, la guardamos con el método *save* de la clase *IOrderDAO* en la línea 80.

No quisiera explicar cada uno de los servicios que utilizan un DAO porque sería muy repetitivo, pero si quieres, puedes seguir analizando las clases

ProductService y *OrderService* para que veas como utilizamos el DAO para todos los servicios del microservicio crm-api.

Conclusiones

El patrón DAO es sin lugar a duda, uno de los más utilizados en la actualidad, ya que es fácil de implementar y proporciona claros beneficios, incluso, si solo tenemos una fuente de datos y esta no cambia, pues permite separar por completo la lógica de acceso a datos en una capa separada, y así, solo nos preocupamos por la lógica de negocio sin preocuparnos de donde viene los datos o los detalles técnicos para consultarlos o actualizarlos.

Por otro lado, mediante el patrón Abstract Factory es posible determinar en tiempo de ejecución que instancia concreta del DAO debemos instanciar, desacoplando la instancia concreta del DAO del *BusinessObject*, lo que permite cambiar la instancia concreta del DAO sin afectar el funcionamiento del *BusinessObject*.

Polling

Prácticamente todas las aplicaciones modernas requieren comunicarse con aplicaciones externas, ya sea para consultar o enviar datos a otra aplicación, pero que pasa cuando una aplicación necesita saber si hay alguna novedad en un sistema externo, como podemos saber si algo ha pasado de lo cual necesitamos enterarnos.

Problemática

Normalmente, cuando algo pasa en un sistema externo no tenemos forma de saberlo, si no hasta que vamos al sistema y lo consultamos, pero ¿cómo saber exactamente qué fue lo que cambio? no podemos simplemente ir y consultar toda la base de datos para comparar registro contra registro para identificar el cambio, lo cual se convierte en un problema.

O que pasa cuando un servidor se encuentra de tras de un firewall que impide la comunicación bidireccional, por ejemplo, el navegador puede establecer conexión con el servidor, pero el servidor no puede iniciar una comunicación con el navegador, en ese caso, como logramos saber si hay algún cambio en el servidor, quizás la única solución sea actualiza la página para volver a realizar una nueva consulta al servidor.

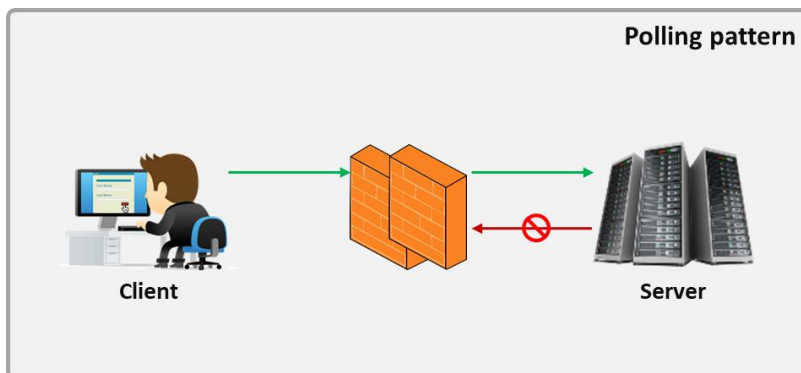


Fig 167 - Comunicación bloqueada por el firewall.

O simplemente el servidor no cuenta con mecanismos para notificar de algún cambio a los interesados, lo que obliga al sistema interesado en realizar consultas repetitivas para traer los últimos cambios.

Solución

Una de las formas más rudimentarias, pero aun eficiente forma de obtener las actualizaciones de un sistema es mediante el *Polling*, el cual consiste en **realizar una serie de consultar repetitivas y con una periodicidad programada** en búsqueda de nueva información, de esta forma, el sistema interesado tendrá que ir al servidor y preguntar si hay nuevas actualizaciones, si las hay, el servidor las retornará, en otro caso, el cliente seguirá preguntando cada x tiempo hasta que encuentre nuevas actualizaciones. Un dato importante es que el *Polling* no se detiene nunca, es decir, encuentre o no actualizaciones, este seguirá preguntando por más actualizaciones.

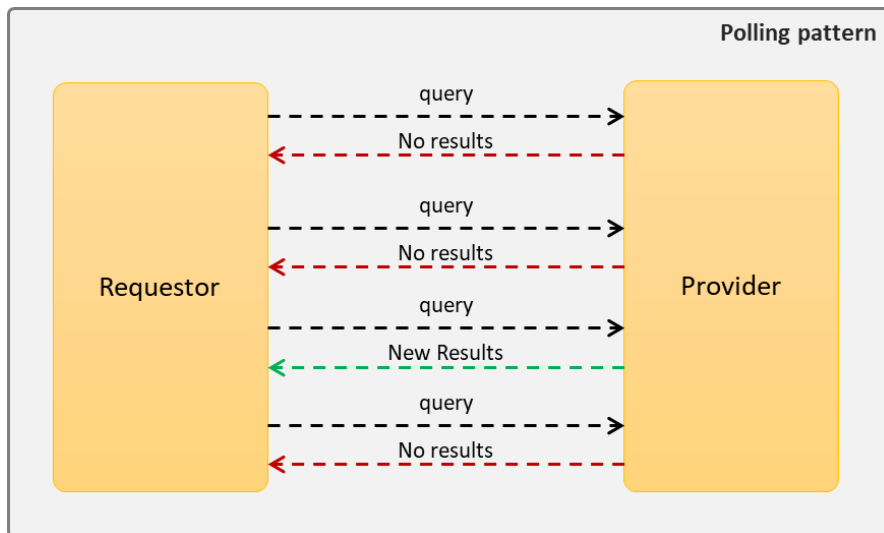


Fig 168 - Polling pattern

En el patrón *Polling* se conoce como *Requestor* al componente que realiza las consultas recurrentes, mientras que el *Provider* corresponde al servidor que contiene los datos que pueden cambiar.

En este sentido, podemos ver en la imagen anterior, como el *Requestor* realizar una serie de consultas al *Provider* en búsqueda de nueva información, y no se detiene, incluso cuando encuentra nuevas actualizaciones, es por ello que, la única forma en que un Polling deja de realizar consultas es cuando lo apagamos manualmente, en otro caso, seguirá pregunta día y noche.

El **Polling es un patrón que se debe de utilizar con cuidado**, pues la consulta recurrente de nuevas actualizaciones trae un desgaste en el **performance** del *Provider*, ya que cada solicitud implica, por lo general, una consulta a la base de datos o al sistema de archivos, lo que hace que un abuso en la frecuencia de consulta sea un tema grave a considerar.

Por otro lado, es importante controlar el número de *Requestor* que estarán solicitando actualizaciones, pues no podemos permitir que cualquiera llegue y comience a tirarle consultas a *Provider*. Un ejemplo claro de esto es, una página web que requiere mostrar cierta información en tiempo real, la cual actualiza un reporte cada x tiempo, sin embargo, es una página web la cual muchos usuarios puede entrar al mismo tiempo, esto provocará que un número indeterminado de *Requestor* comience a realizar consultas al *Provider*, lo que puede llevar al proceso de consulta de actualizaciones a competir por los recursos de la misma operación.

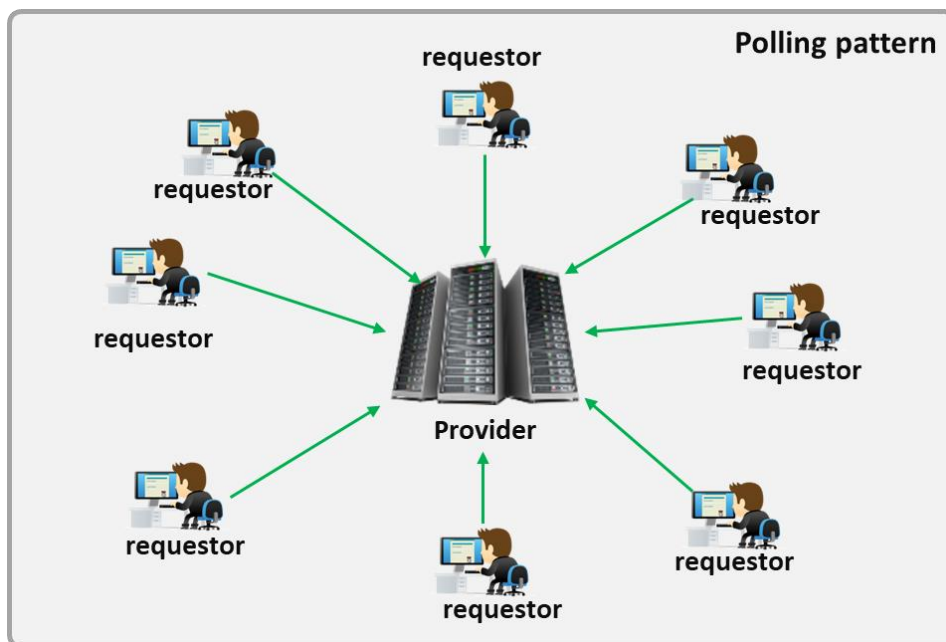


Fig 169 - Polling fuera de control.

Por ningún motivo debemos diseñar una arquitectura donde no podamos controlar el número de *Requestor*, si llegamos a un escenario como el de la imagen anterior, quiere decir que hemos hecho las cosas mal.

El patrón Polling se utiliza para consultar las actualizaciones de un sistema externo cuando es la única alternativa, es decir, cuando el *Provider* está fuera de nuestro control y no tenemos un mecanismo más eficiente para consultar las actualizaciones, como es el caso de los Webhook (los analizaremos más adelante).

Pueden existir diversos motivos por los cuales sea necesario utilizar un *Polling*, pero los más comunes son:

1. Consulta de nuevos archivos de texto, como facturas, pagos o catálogos diversos, como proveedores, clientes, cuentas, etc. Generalmente por miedo de FTP
2. Consultas recurrentes sobre una base de datos en búsqueda de nuevos registros.
3. Consulta recurrente a un servicio de aplicación para saber el status de un determinado proceso.
4. Monitoreo del estado de salud de servicios e infraestructura.

Polling sobre un servidor FTP

En el mundo real, muchas de las aplicaciones empresariales generan archivos de texto para que otros sistemas los tomen e integren la información en sus sistemas, sobre todo cuando son grandes volúmenes de datos.

Los pagos de recibos como, luz eléctrica, agua, internet, etc son ejemplo claro del Polling, ya que muchas empresas dan a sus clientes la facilidad de que realicen los pagos de sus servicios en las tiendas de conveniencia. Estas tiendas lógicamente no nos notifican inmediatamente cuando se ha recibido un pago, al contrario, agrupan todos los pagos del día y los agrupan en un archivo de texto que nos dejan en un servidor FTP.

Estos archivos son depositados con cierta periodicidad, pueden ser diarios, cada 6 horas o cada par de días, todo dependerá del acuerdo comercial. Lo importante es que ellos no nos dicen cuando dejan estos archivos, solo los dejan allí para que nosotros los tomemos.

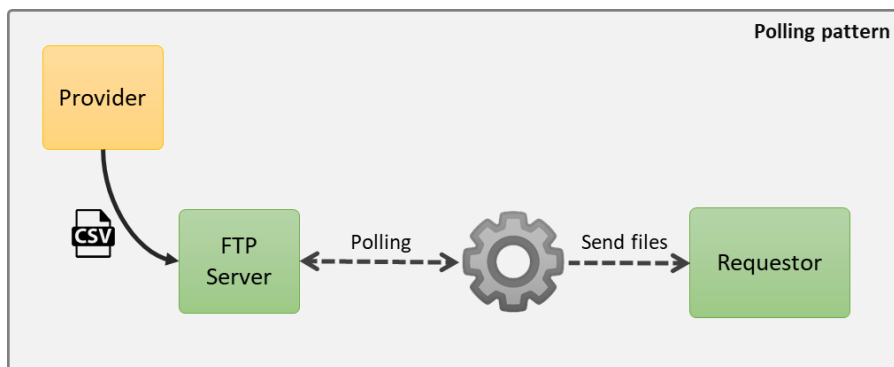


Fig 170 - FTP Polling

Una arquitectura cómo la anterior es muy común, donde el *Provider* dejan los archivos cada X tiempo en un servidor FTP, luego, tenemos un componente que se encarga de consultar si existe nuevos archivos en el servidor de FTP.

Cuando el *Polling* detecta un nuevo archivo lo lleva al *Requestor*, quien será el que los termine procesando, finalmente, el *Polling* puede borrar, renombrar o cambiar de carpeta el archivo una vez procesado, lo cual impide que sea procesado más de una vez.

El componente que hace el *Polling* puede ser la misma aplicación que requiere de los archivos o podría ser un componente independiente que tiene como único objetivo la consulta de los archivos.

Polling sobre una base de datos

Otro de los *Pollings* más frecuentes es directamente sobre una base de datos, en el cual el componente que realiza el *Polling* realiza una determinada consulta cada X tiempo sobre una o varias tablas en búsqueda de nuevos registros, de esta forma, es el *Polling* el encargado de conectarse a la base de datos y buscar las actualizaciones, traerlas al *Requestor* y actualizar/borrar los registros recuperados directamente sobre la base de datos que alimenta el *Provider*.

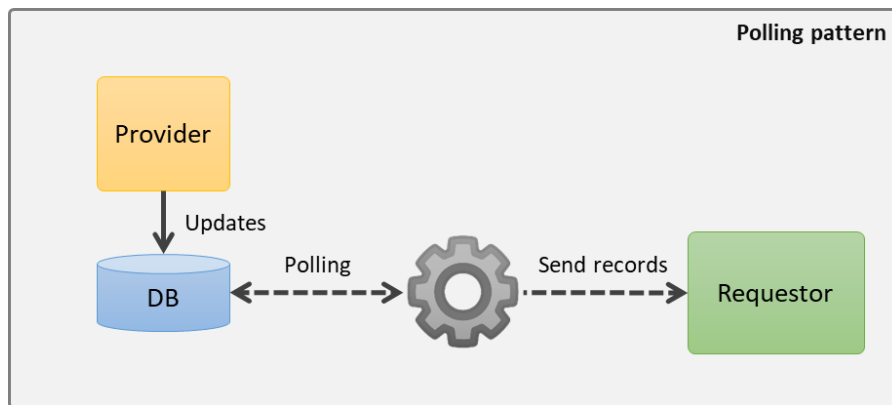


Fig 171 - Polling sobre una base de datos.

En esta nueva arquitectura el *Provider* puede crear tablas especiales de replicación, o notificaciones, las cuales son ajenas a las tablas estándares de la aplicación, con el único objetivo de crear registros cuando nuevas actualizaciones relevantes fueron realizadas en el *Provider*, de esta forma, el *Polling* está en constante consulta de estas tablas, en búsqueda de nuevos registros o registros que cumplan con ciertas condiciones, para llevarlos al *Requestor*.

Nuevamente, el *Polling* deberá borrar o actualizar los registros que fueron recuperados y enviados al *Requestor*, con el objetivo de no procesar más de una vez el mismo registro.

Este tipo de arquitectura es común cuando el *Provider* y el *Requestor* están dentro de nuestra infraestructura, lo cual nos permite, obviamente, tener una conexión directa a la base de datos del *Provider*.

Consulta del status de un proceso

Existe ocasiones donde necesitamos que pase algo en otra aplicación antes de poder continuar con nuestro proceso, desafortunadamente, esta aplicación no nos dará actualizaciones, por lo que estamos obligados a realizar un *Polling* hasta que cierta condición se cumpla en la otra aplicación.

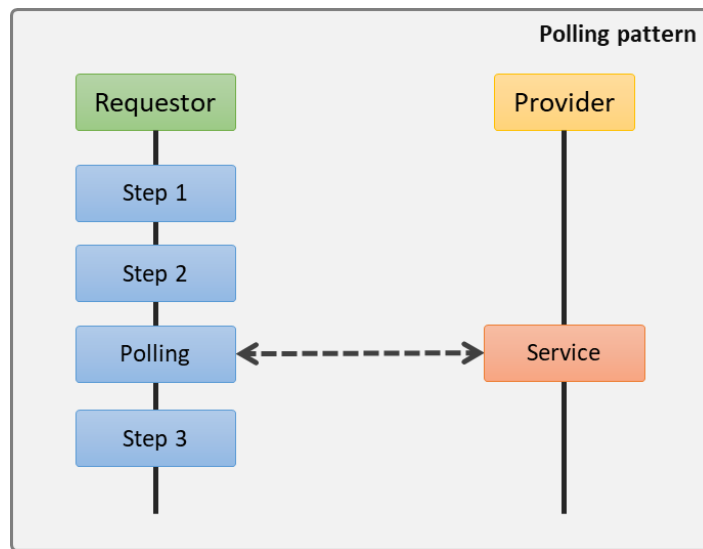


Fig 172 - Polling sobre el status de un proceso.

En el ejemplo anterior podemos ver como el *Requestor* es en realidad un proceso de negocio que realiza una serie de pasos, sin embargo, llega un punto donde no puede continuar hasta que cierta condición se cumpla en el *Provider*, para esto, el *Requestor* inicia un *Polling* sobre el *Provider* y esperará hasta que la condición se cumpla en el *Provider*, cuando la condición se cumple, entonces se ejecuta el Paso 3 (*Step 3*), antes no.

Monitoreo de servicios

El *Polling* es una técnica muy utilizada para comprobar la disponibilidad de la infraestructura o servicios, ya que mediante este es posible hacer *pings* a los servidores para comprobar que estén encendidos, o realizar consulta a los servicios de un API para comprobar si están respondiendo adecuadamente.

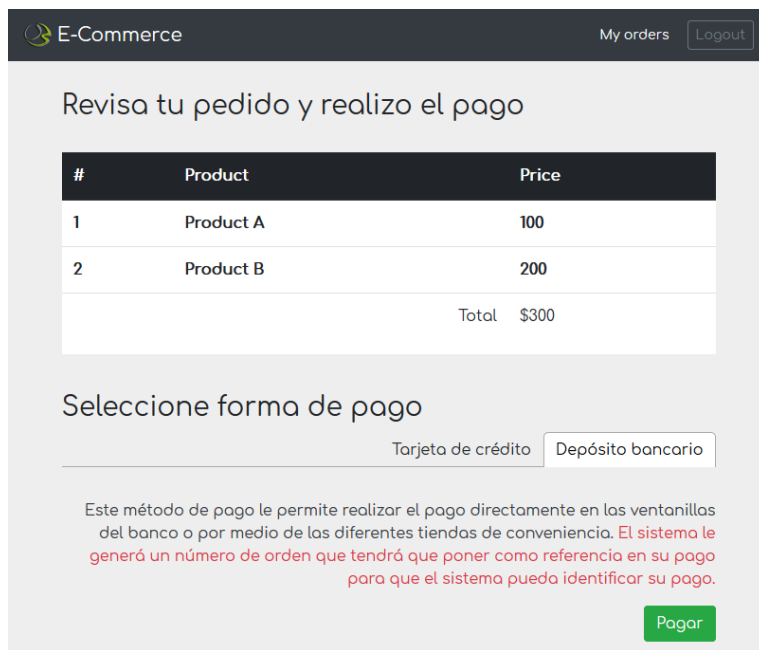
Si el *Polling* es exitoso quiere decir que todos está bien y no se realiza ninguna acción, sin embargo, si un ping o una llamada a algún servicio falla, entonces se notifica a los administradores para solucionar el problema lo antes posible.

Polling en el mundo real

En la sección anterior analizamos la teoría que hay detrás de patrón Polling, pero ha llegado el momento de ponerlo en práctica, pero antes de eso, analicemos como es que utilizamos el patrón Polling dentro de nuestro proyecto e-commerce.

Si recordarás, la aplicación e-commerce soporta dos formas de pago, tarjeta de crédito y depósito bancario; pues bien, en esta ocasión nos centraremos en el pago por depósito bancario.

Para analizar cómo funciona, agregaremos algunos productos al carrito de compras, finalizaremos la compra y seleccionaremos la forma de depósito bancario.



The screenshot shows a web interface for an E-Commerce application. At the top, there is a dark header with the 'E-Commerce' logo on the left, and 'My orders' and 'Logout' links on the right. The main content area has a light gray background. The heading 'Revisa tu pedido y realiza el pago' is centered. Below it is a table with three columns: '#', 'Product', and 'Price'. The table contains two rows of products and a total row. Below the table, there is a section titled 'Seleccione forma de pago' with two radio buttons: 'Tarjeta de crédito' and 'Depósito bancario'. The 'Depósito bancario' option is selected. Below the radio buttons, there is a paragraph of text explaining the payment process. At the bottom right, there is a green 'Pagar' button.

#	Product	Price
1	Product A	100
2	Product B	200
Total		\$300

Seleccione forma de pago

☐ Tarjeta de crédito ☒ Depósito bancario

Este método de pago le permite realizar el pago directamente en las ventanillas del banco o por medio de las diferentes tiendas de conveniencia. El sistema le generará un número de orden que tendrá que poner como referencia en su pago para que el sistema pueda identificar su pago.

Fig 173 - Pago por medio de depósito bancario.

Observa que para finalizar el pago no nos solicita ningún dato, esto es porque al momento de finalizar la compra se nos generará un número de referencia, el cual deberemos utilizar al momento de realizar el pago en el banco, por lo tanto, la

compra no estará finalizada hasta no recibir el pago, así que presionamos el botón “Pagar” para finalizar la compra:

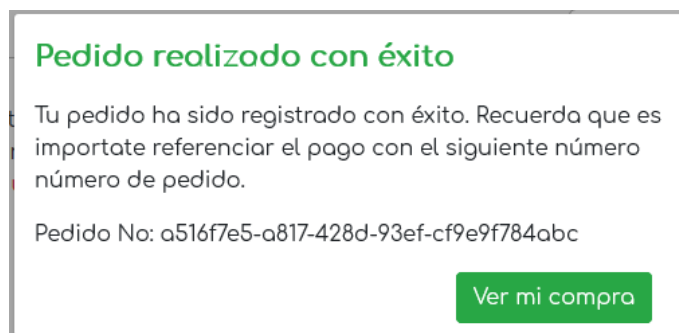


Fig 174 - Finalización de la compra y referencia generada.

Observa que nos ha generado un número de pedido, el cual deberemos utilizar como referencia al momento de realizar el pago. Ahora damos click en “Ver mi compra” para ver el detalle de la compra:

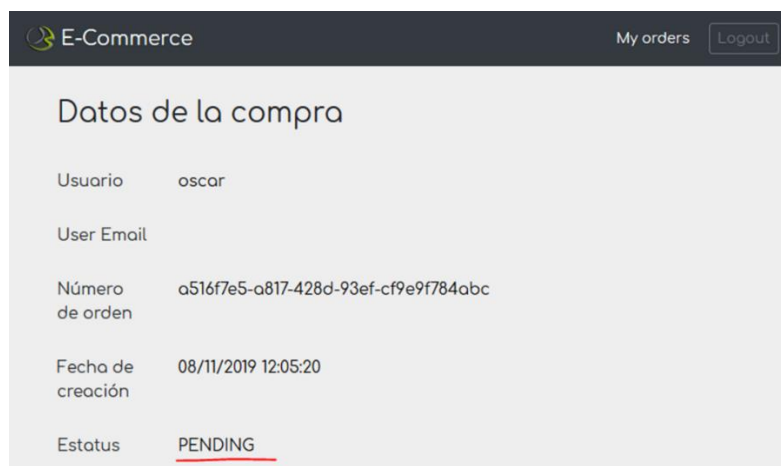


Fig 175 - Detalle de mi pedido.

Observa que el status del pedido es *PENDING*, lo que significa que está pendiente de pago.

En este punto nuestra aplicación estará haciendo un *Polling* contra un FTP que nos proporciona el banco, en busca de nuevos archivos de pagos, para ellos, tenemos

un Microservicio que se encarga exclusivamente de hacer un *Polling* con el FTP del banco, el cual es *ftp-payment-pooling*.

Este Microservicio espera que el banco le deje los archivos en formato de texto plano, donde cada fila corresponde a un pago, y cada pago deberá tener dos campos, el número de referencia y el monto pagado, por ejemplo:

1. 8c908d56-b74b-4d0a-b2a7-03b88fde7ce7,300
2. a516f7e5-a817-428d-93ef-cf9e9f784abc,500

Cada campo está separado con una coma, la cual será utilizada para identificar el número de referencia y el monto pagado.

Cuando el Microservicio *ftp-payment-pooling* detecte un nuevo archivo, lo tomará, aplicará todos los pagos encontrados en el Microservicio *crm-api* y finalmente, borrar el archivo, con la finalidad de no volverlo a procesar.

Instalación previa

Antes de continuar con el ejemplo, debemos de configurar un servidor FTP para simular el que nos proporcionaría el banco, para ello, puedes bajar cualquier servidor FTP de tu preferencia, como FileZilla server que es open Source, pero en mi caso me he inclinado a utilizar *Xlight FTP Server* por su simplicidad, pero siente libre utilizar con el que te sientas más cómodo.

Para instalarlo solo tenemos que descargarlo de <https://www.xlightftpd.com/>, seguimos el wizard y listo, una vez instalado lo abrimos y veremos una pantalla como la siguiente:

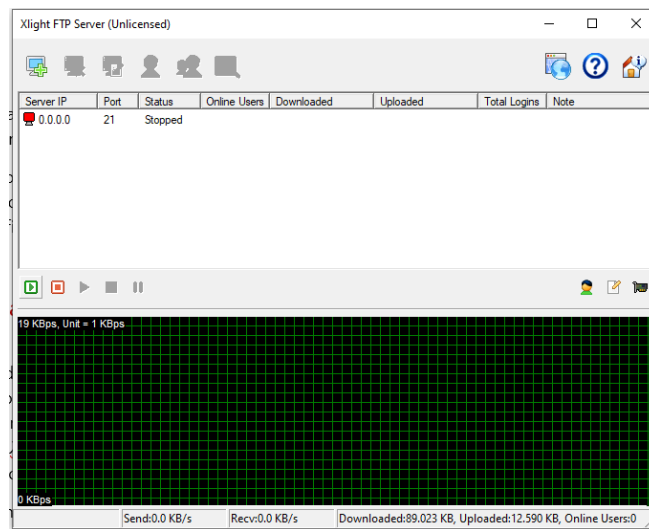


Fig 176 - Xlight FTP Server.

Una vez allí, tendremos que crear un nuevo servidor virtual, desde el ícono (), esto nos arrojará una nueva pantalla:

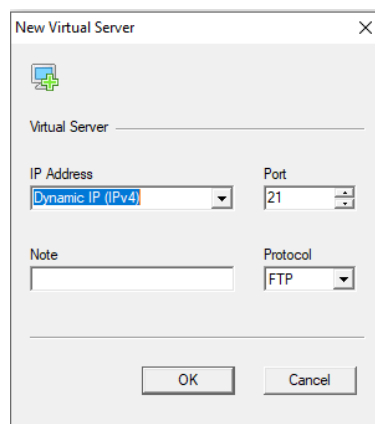


Fig 177 - Configuración del servidor virtual.

Vamos a dejar los valores por default, lo que nos creará un servidor FTP que responde en el puerto 21. El siguiente paso será prender el servidor, lo cual aremos desde el botón de “play”:

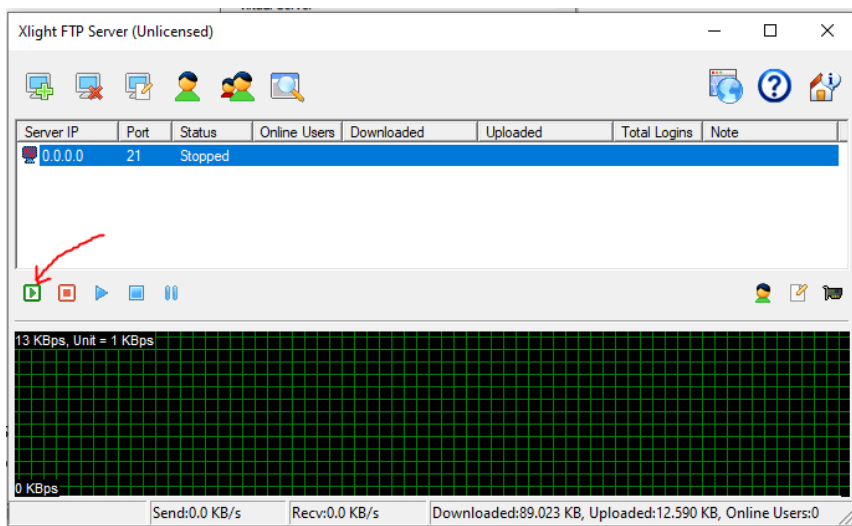


Fig 178 - Encendido del servidor FTP.

El siguiente paso será crear un nuevo usuario para podernos conectar, para esto vamos a click en el ícono (👤) y en la siguiente pantalla damos click en nuevo usuario:

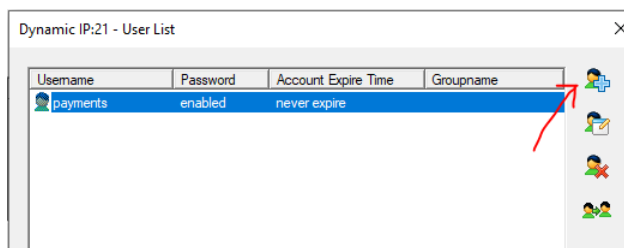


Fig 179 - Nuevo usuario.

En la nueva pantalla tendremos que poner el nombre del usuario y el password con el cual nos autenticaremos:

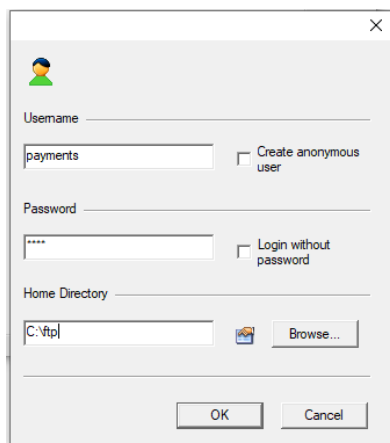


Fig 180 - Nuevo usuario de FTP.

En este caso hemos puesto el usuario **"payments"** con el password **"1234"**, además debemos definir el directorio raíz del FTP. Este último campo es muy importante, pues es donde deberemos de dejar los archivos de pagos, por lo que debe de ser un directorio válido.

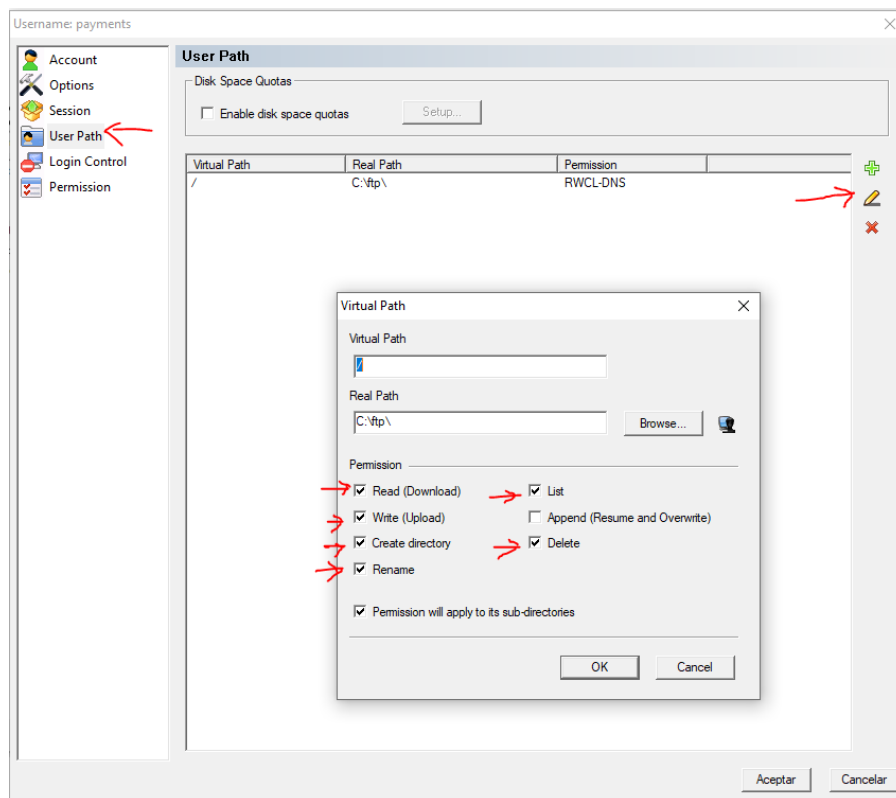


Fig 181 - Agregar privilegios al usuario.

El siguiente paso es dar privilegios al usuario sobre la carpeta, para lo cual, vamos a editar el usuario que acabamos de crear y nos dirigiremos a la sección de “*User Path*”, editamos la entrada que hay y agregar los privilegios que se ven en la imagen anterior.

Ya con el FTP configurado, nos aseguramos que la configuración que acabamos de realizar coincida con la configuración del archivo *application.yml* del Microservicio *ftp-payment-polling*:

1. ftp:
2. enable: **true**
3. host: localhost
4. port: **21**
5. user: payments
6. password: **1234**

Una vez que todo está correctamente configurado, procedemos con encender el servicio desde el *Boot Dashboard*.

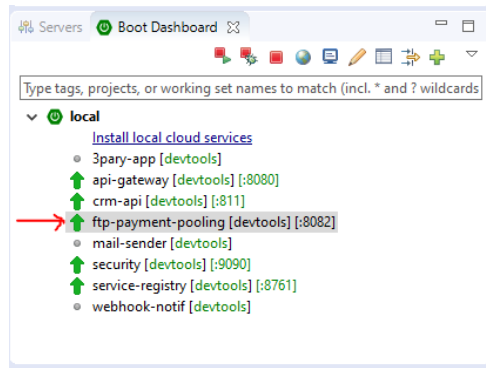


Fig 182 - Iniciando el microservicio ftp-payment-pooling.

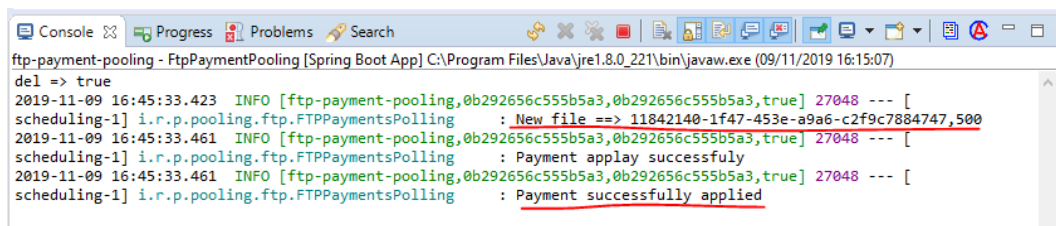
Utilizando el Polling para aplicar los pagos

Ya iniciado el microservicio será necesario revisar el log para asegurarnos de que no existan errores al conectarse al FTP. Una vez validado que todo está bien, tenemos que regresar a la app y crear un nuevo pedido y recuperar el número de orden que utilizaremos como referencia, ya con ese número y el total de la orden crearemos un nuevo archivo plano (txt) como el siguiente:

1. 8c908d56-b74b-4d0a-b2a7-03b88fde7ce7,300

El archivo va a contener una sola línea, con el número de orden y el total de la compra, separados con una coma, tal cual puedes ver en el ejemplo anterior. Guardaremos el archivo con el nombre *payment.txt* y lo guardaremos en la carpeta del FTP que configuramos en la sección anterior. El número de referencia anterior es solo un ejemplo, por lo que debemos de tener cuidado de agregar el número de pedido que nos generó la aplicación y el total de la misma.

Si todo está correctamente configurado, veremos en el log el mensaje de que el pago ya ha sido aplicado, además, deberás de recibir un email con la confirmación del pago:



```
ftp-payment-pooling - FtpPaymentPooling [Spring Boot App] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (09/11/2019 16:15:07)
del => true
2019-11-09 16:45:33.423 INFO [ftp-payment-pooling,0b292656c555b5a3,0b292656c555b5a3,true] 27048 --- [
scheduling-1] i.r.p.pooling.ftppaymentspolling : New file ==> 11842140-1f47-453e-a9a6-c2f9c7884747,500
2019-11-09 16:45:33.461 INFO [ftp-payment-pooling,0b292656c555b5a3,0b292656c555b5a3,true] 27048 --- [
scheduling-1] i.r.p.pooling.ftppaymentspolling : Payment apply successfully
2019-11-09 16:45:33.461 INFO [ftp-payment-pooling,0b292656c555b5a3,0b292656c555b5a3,true] 27048 --- [
scheduling-1] i.r.p.pooling.ftppaymentspolling : Payment successfully applied
```

Fig 183 - Confirmación del pago aplicado exitosamente.

En la imagen anterior puedes observar como el Polling automáticamente detecta el archivo tras unos segundos, esto se debe a que tenemos la clase *FTPPaymentPolling*, la cual se encarga de hacer un Polling sobre el FTP:

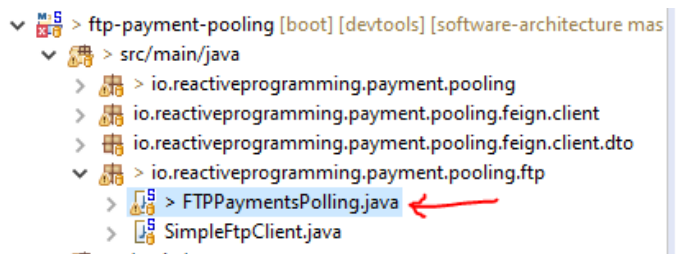


Fig 184 - Clase FTPPaymentPolling.

La clase se ve a sí.

```
1. @Component
2. public class FTPPaymentsPolling {
3.
4.     private static final Logger logger =
5.         LoggerFactory.getLogger(FTPPaymentsPolling.class);
6.
7.     @Autowired
8.     private Tracer tracer;
9.
10.    @Value("${ftp.enable}")
11.    private String enable;
12.
13.    @Autowired
14.    private SimpleFtpClient ftpClient;
15.
16.    @Autowired
17.    private OrderServiceFeignClient orderService;
18.
19.}
```

```

20.     @Scheduled(fixedRate = 10000)
21.     public void pollingFile() {
22.         if(!Boolean.valueOf(enable)) {
23.             return;
24.         }
25.         logger.debug("pollingFile ==>");
26.         try {
27.             ftpClient.connect();
28.             FTPFile[] files = ftpClient.listFiles("/");
29.
30.             for(FTPFile file : files) {
31.                 if(file.getType() != FTPFile.FILE_TYPE) continue;
32.
33.                 String fileContent = ftpClient.readFileAndRemove("/", file);
34.                 logger.info("New file ==> " + fileContent);
35.                 tracer.currentSpan().tag("file.new", file.getName());
36.
37.                 String[] lines = fileContent.split("\n");
38.                 for(String line : lines) {
39.                     String[] linesField = line.split(",");
40.                     if(linesField.length != 2) {
41.                         tracer.currentSpan()
42.                             .tag("file.error", "Invalid file format");
43.                         throw new RuntimeException("Invalid file format");
44.                     }
45.                     String refNumber = linesField[0];
46.                     float ammount = Float.parseFloat(linesField[1]);
47.
48.                     ApplyPaymentRequest payment = new ApplyPaymentRequest();
49.                     payment.setAmmount(ammount);
50.                     payment.setRefNumber(refNumber);
51.                     WrapperResponse response =
52.                         orderService.applyPayment(payment);
53.                     logger.info(response.getMessage());
54.                     tracer.currentSpan()
55.                         .tag("file.process", response.getMessage());
56.                     if(response.isOk()) {
57.                         logger.info("Payment successfully applied");
58.                     }else {
59.                         logger.error("Error " + response.getMessage());
60.                     }
61.                 }
62.             }
63.             ftpClient.disconnect();
64.         } catch (Exception e) {
65.             e.printStackTrace();
66.             System.out.println(e.getMessage());
67.         }
68.     }
69. }

```

La clase *FTPPaymentPolling* del proyecto *ftp-payment-polling* es la encargada de realizar el *Polling* al FTP. Observa cómo en la línea 14 inyecto una referencia a la clase *SimpleFtpClient*, la cual estaremos utilizando para establecer conexión con el servidor de FTP. Por suerte, *Spring boot* toma la configuración de la conexión directamente del archivo *application.yml*, por lo que no hay que preocuparnos por configurarlo.

El siguiente punto interesante es la línea 20, ya que definimos el método *pollingFile* y le agregamos el metadato *@Scheduled*, el cual le dice a *Spring boot* que se trata de un método que se debe de ejecutar cada 10,000 milisegundos, es decir, 10 segundos, por lo tanto, este método estará leyendo los archivos de nuestro servidor cada 10 segundo.

El siguiente punto de interés son las líneas 27 y 28, pues es donde nos conectamos al servidor FTP y listamos todos los archivos para procesarlos. Si se encuentran archivos, estos son cargados y luego por cada línea aplica el pago en el Microservicio *crm-api*, lo cual podemos ver en las líneas 48 a 52.



Error común

Debido a que este es solo un ejemplo, hemos configurados el Polling para que busque nuevos archivos cada 10 segundos, con la finalidad de ver el resultado rápidamente, pero en el mundo real, esto puede ser excesivo, así que hay que cuidar el tiempo con el que leemos los archivos.

Quiero que observar que este es un ejemplo muy básico, pues solo intenta demostrar cómo podemos utilizar el Polling, por lo que hemos dejado de lado algunos controles básicos, cómo un manejo adecuado de los errores para no perder los archivos que fallaron, realizar alguna acción en caso de que una o todas las líneas del archivo no tenga el formato esperado, o como controlar registros o archivos repetidos, todos esto sería parte de la lógica de las reglas de negocio de la aplicación, por lo que lo hemos omitido para no desviarnos mucho del tema central.

Conclusiones

Polling es un patrón que hasta el día de hoy se sigue utilizando mucho, sin embargo, es un patrón que debemos utilizar cuando no tenemos una mejor forma de consultar las actualizaciones de un sistema externo, ya que su uso puede afectar el rendimiento del *Provider*. Por otro lado, tenemos que tener un control cuidadoso sobre todos los *Requester* que podrán realizar consultas, para evitar que cualquier pueda comenzar a realizar un número de consultas excesivas que afecten la operación del sistema.

Por otro lado, el *Provider* podrá contener (no obligatorio) controles que ayuden a determinar el número de peticiones de los *Requester* con la intención de cortar el servicio en caso de un abuso en el número de consultas y la periodicidad de las mismas.

Finalmente, el patrón *Polling* es una excelente alternativa cuando debemos monitorear los cambios en sistemas de terceros, donde no tenemos control sobre ellos o que simplemente no fueron diseñados para notificar sus cambios de una mejor manera.

Webhook

En el capítulo anterior analizábamos como mediante el *Polling*, era posible sondear los cambios de un componente externo mediante una serie de consultas repetitivas, pero que pasa cuando necesitamos hacer todo lo contrario, y en lugar de estar preguntado cada rato por nuevas actualizaciones, sea el otro sistema quien nos notifique al instante cuando un nuevo cambio sucede, pasando de una arquitectura proactiva a una pasiva.

Problemática

En la sección pasada analizábamos que el *Polling* es un patrón inadecuado cuando tenemos grandes cantidades de *Requesters*, ya que cada uno comienza a realizar consultas repetitivas que puedan afectar el rendimiento de nuestra aplicación.

Este problema se acentúa cuando tenemos cientos o miles de *Requesters* interesados en saber si existen nuevas actualizaciones, lo que puede llegar a colapsar al *Provider* por las miles o millones de peticiones que se podrían acumular durante todo el día, lo que hace que el *Polling* sea una solución inaceptable.

Para comprender esta problemática imaginemos el clásico escenario de la pareja con sus hijos que van por la carretera y a los 10 minutos de haber comenzado el viaje los niños comienzan a preguntar cada 5 minutos, ¿Cuánto falta para llegar?, ¿Ya vamos a llegar? Quizás esta comparación es un poco burda, pero en ese mismo escenario, imagina que vas en un camión escolar y todos los niños comienzan a preguntar y tú necesitas estar concentrado en el camino. Seguramente después de un rato te vas a volver loco respondiendo a cada niño que pregunte, y en una de esas te puede distraer y causar un accidente.

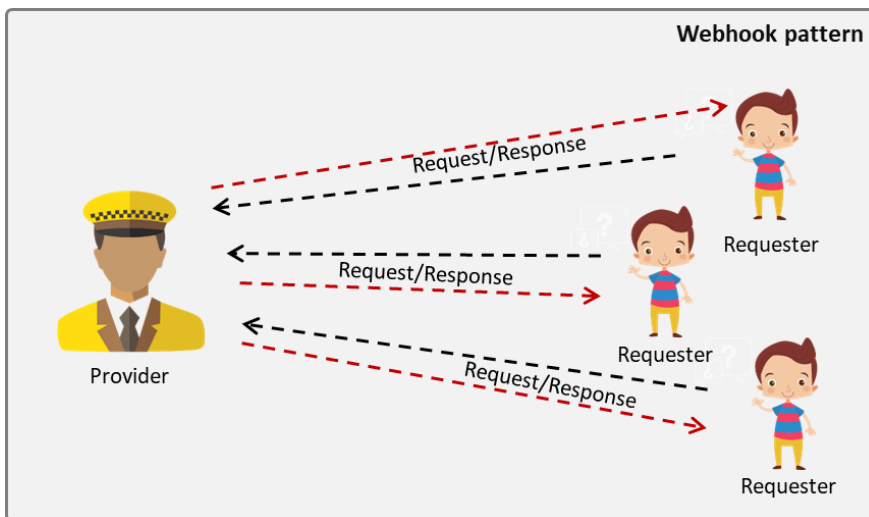


Fig 185 - Cómo funciona el Polling.

El *Polling* es algo similar, cuando las preguntas son pocas es fácil controlarlas, pero a medida que estas suben de intensidad se va haciendo imposible controlarlas, lo que hace que, nuestra operación se vea comprometida por la necesidad de responder a todos los que pregunta. En este sentido, los niños serían los *Requester*, mientras llegas a tu destino es la operación, la cual no se puede detener por nada y si por estar respondiendo a los niños nuestra operación se puede ver comprometida, entonces quiere decir que algo anda mal.

Solución

Una vez que sabemos que el *Polling* no es la mejor solución para resolver este tipo de problemas pasamos a analizar cuál sería la mejor solución.

Basado en el ejemplo anterior, que pasaría si en lugar de responder a los niños cada vez que pregunte, cambiamos la estrategia y mejor usamos un megáfono para decirles a los niños cuanto falta cada vez que vemos un cartelón en la carretera que nos indica los kilómetros restantes. De esta forma los niños podrán saber cuánto falta para llegar de una forma regular y no distraerá mucho al conductor, pues solo tendrá que hacer unos cuantos anuncios en comparación con responder a cada niño.

Los Webhook trabajan de una forma similar, pues la responsabilidad de notificar los cambios relevantes pasa a ser la responsabilidad del *Provider*, pero el *Requester* deberá contar con un mecanismo para recibir estas notificaciones. De esta forma, el *Provider* notificará cuando alguna actualización relevante suceda, pero no le interesará que haga el *Requester* con ella.

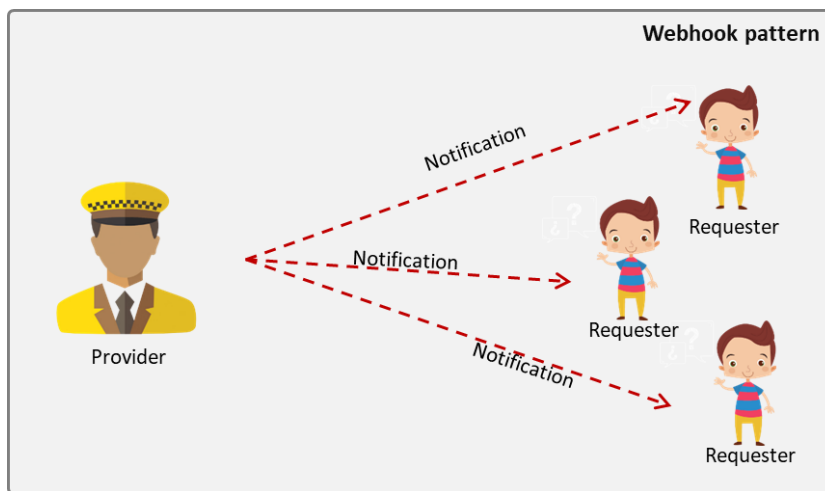


Fig 186 - Cómo funciona un Webhook.

En el caso de una aplicación real, lo primero que se tiene que hacer es definir los eventos de la aplicación que deberán ser notificados a los interesados, de esta forma, es el *Provider* el que determina qué información será notificada y cuál no, por otro lado, los *Requester* se tendrán que suscribir a los eventos del *Provider*, para esto, será necesario que el *Requester* le indique al *Provider* qué eventos quiere recibir y proporcionarle una dirección (URL) donde el *Provider* deberá enviar las notificaciones.

Este registro se hace mediante los mecanismos que proporciona el *Provider*, los cuales pueden ser diversos, como una página web, un servicio web, REST, etc. En realidad, el patrón no determina la forma en que se debe de hacer este registro.

Una vez registrado, el *Requester* comenzará a recibir las notificaciones de los eventos a los que se ha registrado, directamente a la URL que configuró. Estas notificaciones tienen la ventaja de que pueden ser muy rápidas, incluso con unos

pocos segundos de diferencia entre que el evento se disparó y el *Requester* recibe la notificación.

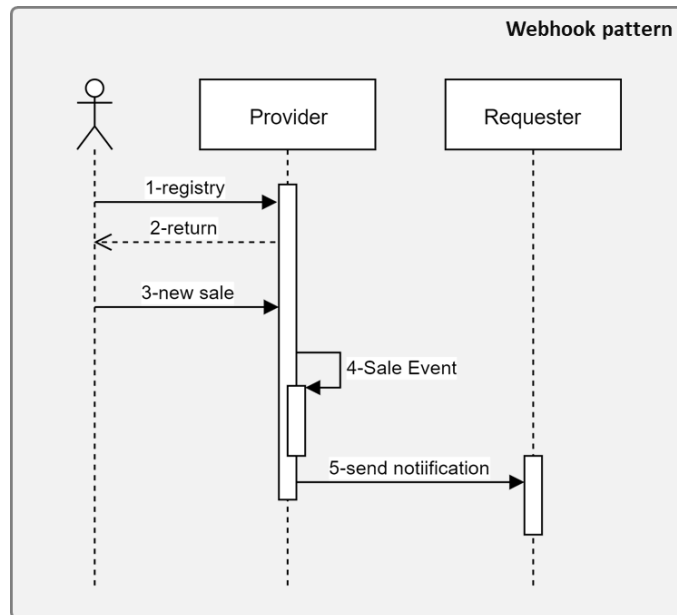


Fig 187 - Diagrama de secuencia del webhook.

En el diagrama anterior podemos observar cual es la secuencia de pasos de un *Webhook*, el cual se lee de la siguiente forma:

1. Un usuario administrador realiza el registro del *Requester* para recibir notificaciones.
2. El *Provider* retorna con la confirmación del registro.
3. Un nuevo usuario entra a la aplicación del *Provider* y realiza una nueva compra.
4. El *Provider* procesa la compra y genera un nuevo evento que requiere ser notificado.
5. El *Provider* busca en su registro para saber cuáles son los *Requester* interesados en ser notificados y les envía una nueva notificación por medio de la URL provista en el primer paso.

Algo a tomar en cuenta es que, las notificaciones no se envían precisamente a todos los *Requester* registrados, sino más bien, es necesario determinar por medio de los privilegios, quien tiene acceso a cierta información, por ejemplo, si es una aplicación donde varios clientes pueden acceder, no les vamos a mandar las

notificaciones de información a la cual no deberían de tener acceso, entonces, tendríamos que tomar en cuenta esto en caso de que sea aplicaciones para múltiples clientes.

Dos formas de implementar Webhook

Un *Webhook* puede ser implementado de dos formas, las cuales consisten en enviar en el payload toda la información referente al evento producido, de tal forma que el remitente tenga toda la información necesaria para actuar en consecuencia, o la otra es, enviar la información mínima para que el remitente sepa únicamente que cambio y luego consumir por servicio el detalle de la actualización.

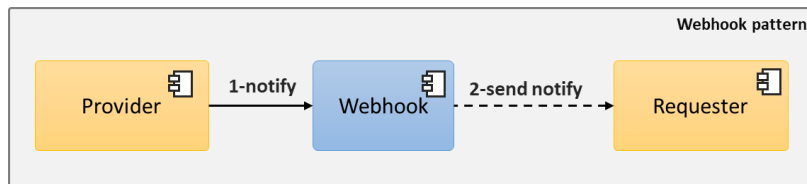


Fig 188 - Notificaciones de una sola ida.

En la imagen anterior podemos observar cómo sería el escenario donde el mensaje contiene toda la información para actuar en consecuencia. Observa que el *Requester* no necesita ir nuevamente al *Provider* para comprobar la información, sin embargo, este tipo de escenarios es más común en entornos cerrados, donde no cualquiera puede inyectar un mensaje malicioso a nuestra aplicación, porque al final, el *Webhook* notifica mediante un servicio expuesto por el *Requester*, que podría ser fácilmente consumido por un agente externo, inyectando información maliciosa.

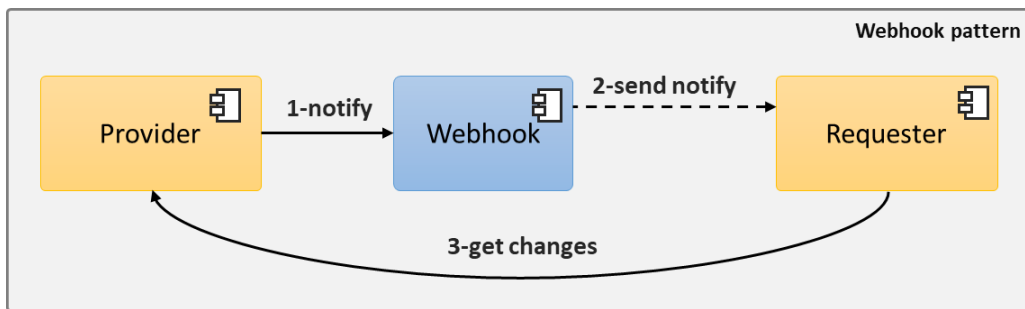


Fig 189 - Notificaciones con comprobación.

Una segunda forma de implementar un *Webhook* es como se ve en la imagen anterior, donde ante cada eventos recibido, el *Requester* comprueba nuevamente la información desde el *Provider*, de esta forma se logran dos cosas, la primera es, poder consultar de primera mano la última información del *Provider*, asegurándonos de tener los últimos cambios, por otro lado, tenemos una mayor seguridad, ya que la notificación nos asegura que algo sucedió en el *Provider*, pero mediante una consulta podemos comprobar que efectivamente la información está como dicen estar por la notificación, lo que nos permite exponer fácilmente el servicio de notificación por Internet sin preocuparnos que nos inyecten datos. En lo particular, yo siempre utilizo la segunda opción, pues la considero más fiable.

Reintentos

Algo muy importante a tomar en cuenta es que, es posible que al momento de mandar las notificaciones el destinatario puede estar offline, lo que haría imposible entregar las notificaciones a ese destinatario, por lo que es normal implementar algún sistema de reintentos que permita reenviar esas notificaciones cada X tiempo, hasta un máximo de N reintentos, con esto, nos aseguramos de hacer todos lo posible por notificar al destinatario, pero tampoco nos podemos comprometer a reintentar indeterminadamente, por lo que es importante pensar en una buena estrategia de reintentos que garantice lo mejor posible la entrega de los mensajes.

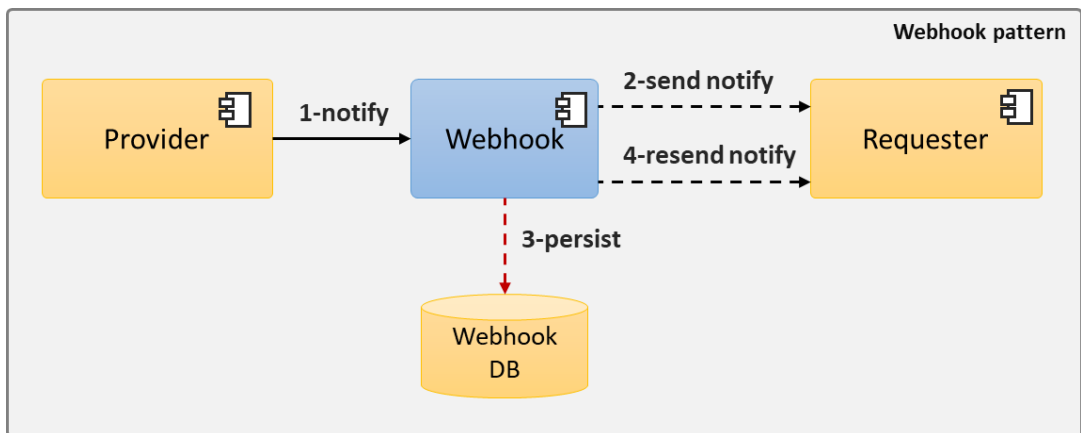


Fig 190 - Arquitectura de reintentos.

En la imagen anterior podemos ver una arquitectura típica de como implementar un sistema de reintentos, la cual está compuesta por tres partes, el Provider que es básicamente la aplicación que dispara los eventos, el componente Webhook que puede ser un Microservicio que reciba los eventos y se encargue de canalizarlos al receptor adecuado y finalmente el Requester, el cual es el interesado en recibir los eventos.

Podrás ver como el Provider no envía directamente los eventos, en su lugar, se apoya de un componente encargado exclusivamente del envío de las notificaciones (puede ser un componente independiente o dentro del Provider), de esta forma, el componente del Webhook puede tomar exclusivamente la responsabilidad del envío de las notificaciones y controlar los reintentos, es por ello que tiene una base de datos asociada, ya que le permite guardar los eventos que no pudieron ser entregados y luego tomarlos más adelante para su reenvío.

Para el reenvío podrías utilizar un Polling, que esté constantemente leyendo las tablas de eventos y reenviar los mensajes que no pudieron ser entregados.

Webhook en el mundo real

Una de las características de nuestra API es la posibilidad de notificar a otros componentes cuando una nueva venta se ha realizado, para ello hemos

implementado un *Webhook* que nos permita notificar a todas las aplicaciones previamente registradas.

Para lograr esto, hemos creado un Microservicio llamado *webhook-notify*, el cual tiene la única responsabilidad de enviar las notificaciones a los componentes registrados, pero es el Microservicio *crm-api* el que crea los eventos cuando se crea una nueva venta.

Vamos a comenzar a analizar en el orden en que se dan los eventos, por lo que comenzaremos analizando el servicio donde se transacciona la compra y se crea el evento de la nueva venta, para ello, regresaremos al método *createOrder* de la clase *OrderService* del Microservicio *crm-api*:

```
1. @HystrixCommand(fallbackMethod="queueOrder",
2.     ignoreExceptions= {ValidateServiceException.class})
3. public SaleOrderDTO createOrder(NewOrderDTO order) throws ValidateServiceException,
4.     GenericServiceException {
5.     logger.info("New order request ==>");
6.     try {
7.         /* suppressed lines */
8.
9.         SaleOrder newSaleOrder = orderDAO.save(saleOrder);
10.
11.         logger.info("New order created ==>" + newSaleOrder.getId() +
12.             ", refNumber > " + newSaleOrder.getRefNumber());
13.         tracer.currentSpan().tag("order.new", newSaleOrder.getRefNumber());
14.         tracer.currentSpan().name(saleOrder.getRefNumber());
15.
16.         SaleOrderConverter orderConverter = new SaleOrderConverter();
17.         SaleOrderDTO returnOrder = orderConverter.toDTO(newSaleOrder);
18.
19.         EmailDTO mail = new EmailDTO();
20.         mail.setFrom("no-reply@crm.com");
21.         mail.setSubject("Hemos recibido tu pedido");
22.         mail.setTo(newSaleOrder.getCustomerEmail());
23.         if(paymentMethod==PaymentMethod.CREDIT_CARD) {
24.             mail.setMessage(String.format("Hola %s,...",
25.                 newSaleOrder.getCustomerName(), newSaleOrder.getRefNumber()));
26.         }else {
27.             mail.setMessage(String.format("Hola %s,...",
28.                 newSaleOrder.getCustomerName(), newSaleOrder.getRefNumber()));
29.         }
30.
31.         sender.send( "emails", null, mail);
32.     }
```



```

33.         // Send Webhook notification
34.         try {
35.             if(paymentMethod==PaymentMethod.CREDIT_CARD) {
36.                 MessageDTO message = new MessageDTO();
37.                 message.setEventType(MessageDTO.EventType.NEW_SALES);
38.                 message.setMessage(returnOrder);
39.                 WrapperResponse response = restTemplate.postForObject(
40.                     "http://webhook/push", message, WrapperResponse.class);
41.             }
42.             catch (Exception e) {
43.                 logger.error(e.getMessage(),e);
44.                 tracer.currentSpan().tag("webhook.fail", e.getMessage());
45.                 System.out.println("No webhook service instance up!");
46.             }
47.             return returnOrder;
48.         } catch (ValidateServiceException e) {
49.             e.printStackTrace();
50.             throw e;
51.         } catch (Exception e) {
52.             e.printStackTrace();
53.             throw new GenericServiceException(e.getMessage(),e);
54.         }
55.     }

```

Este método ya lo habíamos analizado anteriormente, pero habíamos omitido las líneas relacionadas a las notificaciones del *Webhook*, por ello, nos centraremos únicamente en los puntos relevantes para esta unidad.

Lo primero que vamos a resaltar es la creación o el guardado de la Orden de venta en la línea 9, después validamos el tipo de pago, ya que solo notificaremos si la orden fue pagada, por lo que solo notificaremos en caso de que el método de pago sea Tarjeta de crédito, por ello, en la línea 35 validamos el método de pago. Una vez que sabemos la orden fue pagada, creamos el request para enviar la notificación en las líneas 36 a 38. Quiero que prestes especial atención en la línea 37, pues allí definimos mediante una Enumeración el tipo de evento que vamos crear, en este caso, el evento es de tipo *NEW_SALE*, presta atención porque retomaremos este tipo de evento más adelante.

Finalmente, solicitamos el envío de la notificación en la línea 39-40. Con estas dos últimas líneas hacemos una llamada al Microservicio *webhook-notify*, con lo que ejecutamos el siguiente servicio:

```

1. PostMapping(path="push")
2. public WrapperResponse<Void> pushMessage(@RequestBody MessageDTO message) {
3.     try {
4.         eventListenerService.pushMessage(message);
5.         return new WrapperResponse<>(true, "Message sent successfully");
6.     } catch (ValidateServiceException e) {

```

```

7.         return new WrapperResponse<Void>(false, e.getMessage());
8.     }catch(GenericServiceException e) {
9.         e.printStackTrace();
10.        return new WrapperResponse<Void>(false, "Internal server error");
11.    }
12. }

```

El método anterior lo puedes encontrar en la clase *WebhookREST* del microservicio *webhook-notify*:

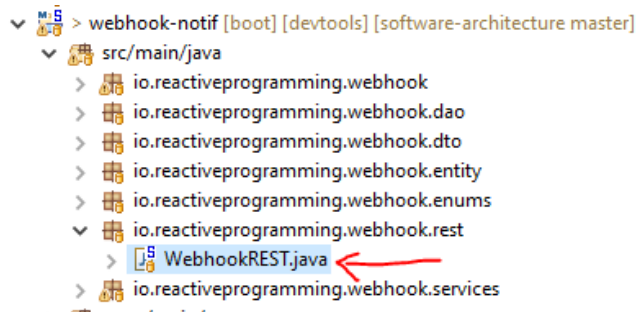


Fig 191 - La clase *WebhookREST*.

Este servicio ejecuta a la vez al método de negocio *pushMessage* de la clase *EventListenerService* del mismo Microservicio:

```

1. public void pushMessage(MessageDTO message)
2. throws ValidateServiceException, GenericServiceException {
3.     try {
4.         List<Listener> listeners =
5.             listenerDAO.findByEventType(message.getEventType());
6.         for(Listener listener : listeners) {
7.             try {
8.                 RestTemplate template = new RestTemplate();
9.                 Map map = new HashMap();
10.                 System.out.println("Send notify to ==> "
11.                     + listener.getEndpoint());
12.                 System.out.println("With body ==> "
13.                     + message.getMessage());
14.                 template.postForObject(listener.getEndpoint(),
15.                     message.getMessage(), Void.class, map);
16.             } catch (Exception e) {
17.                 e.printStackTrace();
18.                 throw new ValidateServiceException(
19.                     "Error to send message to " + listener.getEndpoint());
20.             }

```

```
21.     }
22.   } catch (Exception e) {
23.     e.printStackTrace();
24.     throw new ValidateServiceException("Error to send messages");
25.   }
26. }
```

El método *pushMessage* es el que se encargará finalmente de enviar las notificaciones a los componentes registrados, para esto, hace uso del método *findByEventType* (líneas 4 y 5), el cual busca a todos los componentes registrados para el tipo de evento que se envía como parámetro. ¿Recuerdas que te dije que prestaras atención en el tipo de evento? Pues bien, en este caso, el tipo de evento es *NEW_SALE*, entonces, el método *findByEventType* buscará todos los componentes registrados para recibir notificaciones de este tipo de eventos.

Ahora bien, en las líneas 14 y 15 hacer el envío de la notificación a la URL definida por la propiedad *listener.getEndpoint()* que es la URL que el componente registra para recibir las notificaciones.

Pero solo ahora queda solo una duda por aclarar, ¿cómo se registran los componentes al *Webhook* para comenzar a recibir las notificaciones? Pues la respuesta es simple, tenemos un servicio RESTful que los interesados pueden consumir para registrarse, el cual podemos ver en la URL <http://localhost:9091/swagger-ui.html>, solo recuerda que el componente *webhook-notifiy* debe de estar encendido para poder ver la página. Al entrar veremos la siguiente página:

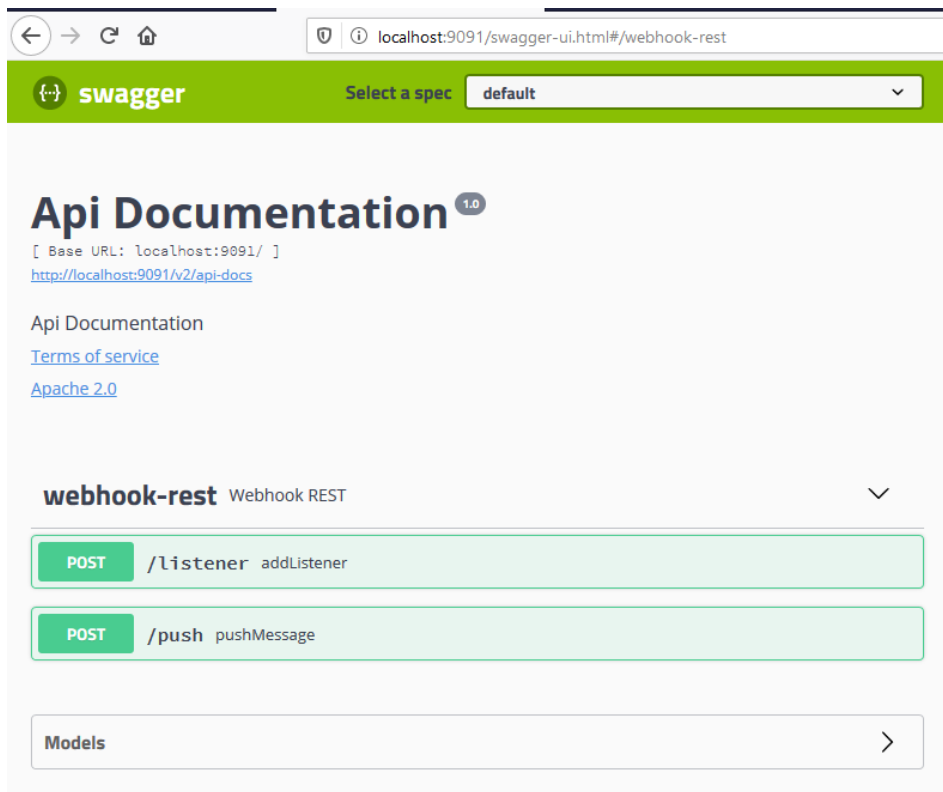


Fig 192 - Documentación del API RESTful de webhook-notify.

En la imagen anterior podemos ver los métodos disponibles por el API, donde el método *push* es el que usamos desde *crm-api* para enviar la notificación y el método *Listener* es que usaremos para registrar a los componentes que quieran recibir notificaciones.

Si damos click en la operación */listener* podremos ver el detalle de la operación, así como los parámetros de entrada que requiere para registrar un nuevo componente:

POST /listener addListener

Parameters Try it out

Name	Description	Example Value	Model
listener * required (body)	listener	<pre>{ "endpoint": "string", "eventType": "string", "id": 0 }</pre>	

Parameter content type
application/json

Fig 193 - Detalle de la operación listener.

Observa en la imagen anterior que nos muestra un request de ejemplo, el cual requiere de 3 parámetros, de los cuales solo requerimos de los primeros para registrar un nuevo componente, el *endpoint* y *eventType*, primero corresponde a la URL a la cual será enviada la notificación, que deberá corresponder a un servicio HTTP que responde en el método POST, el segundo es el tipo de evento al que nos queremos registrar, que por el momento solo tenemos NEW_SALE.

Desde esa misma pantalla podríamos registrar un nuevo componente, pero antes de eso, quiero que echemos un vistazo a la tabla *listeners* de la base de datos *webhook*:

```

11 • use webhooks;
12 • select * from listeners;

```

id	endpoint	event_type
1	http://localhost:9092	NEW_SALES
NULL	NULL	NULL

Fig 194 - Tabla listeners.

Observarás que por default ya tenemos un componente registrado, el cual quiere que sea notificado en la URL <http://localhost:9092>, así que si prestamos atención en nuestros proyectos, verás que el Microservicio *3party-app* responde en ese puerto, por lo que esa será la aplicación que reciba la notificación cuando una nueva venta se realice.

Ahora bien, regresando al servicio *listener*, podrás ver que si presionamos el botón “try it out” podremos ejecutar el servicio:

POST

/listener addListener

Parameters

Try it out

Name	Description
listener * required (body)	listener

Example Value

Model

```

{
  "endpoint": "string",
  "eventType": "string",
  "id": 0
}

```

Parameter content type

application/json

Fig 195 - Probando el servicio listener.

Como resultado, nos saldrá un campo de texto donde podremos introducir el request y desde aquí podremos registrar cualquier componente que queramos que sea notificado.

The screenshot shows a REST client interface with a green header bar. On the left, a green button labeled 'POST' is next to the endpoint '/listener' and the method 'addListener'. Below this, a 'Parameters' section is visible with a 'Cancel' button. The main area is divided into two columns: 'Name' and 'Description'. Under 'Name', there is a field labeled 'listener' with a red asterisk and the text 'required' next to it. Below this, '(body)' is indicated. Under 'Description', the word 'listener' is present. To the right of the 'Name' field, there are tabs for 'Example Value' and 'Model'. The 'Example Value' tab is active, showing a JSON body:

```
{  "endpoint": "http://dummyspage.com/webhook",  "eventType": "NEW_SALES"}
```

. Below the JSON editor, there is another 'Cancel' button. At the bottom, there is a dropdown menu for 'Parameter content type' with 'application/json' selected. A large blue 'Execute' button is at the very bottom.

Fig 196 - Registrando un nuevo componente.

En este momento no registraremos ninguna aplicación pues no tenemos otro Microservicios al cual registrar, pero si quieres puedes crear uno nuevo y registrarlo.

Bueno, ya tenemos claro cómo es que los eventos se crean y se envían, pero ahora falta analizar cómo los componentes interesados reciben esta notificación, por lo que ahora analizaremos el componente *3party-app*, la cual intenta simular ser una aplicación externa que quiere recibir notificaciones para hacer algo cuando una nueva venta se realice.

Este Microservicio expone un servicio RESTful que responde en la raíz y en el método POST, es decir, recibe peticiones en la URL <http://localhost:9092>, es decir la misma URL que tenemos registrada en la tabla *Listener*.

Ahora bien, cuando la notificación sea enviada al Microservicio 3party-app será procesada por el método *salesOrderEndpoint* definida en la clase *WebhookEndpoint*:

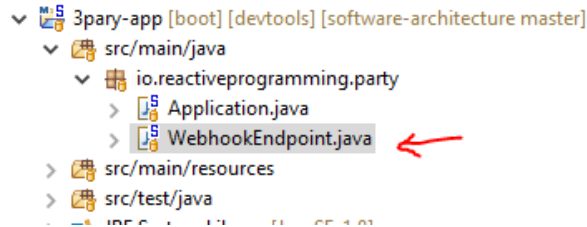


Fig 197 - Clase *WebhookEndpoint*.

```
1. @RestController
2. public class WebhookEndpoint {
3.
4.     @PostMapping
5.     public void salesOrderEndpoint(@RequestBody Object message) {
6.         System.out.println("New message ==> " + message );
7.     }
8. }
```

Podrás observar en la línea 4 que usamos el metadato *@PostMapping* para indicar que este método atiende solicitudes en el método POST, y al no definir una path, asume que atenderá las peticiones en la raíz. Finalmente, en la línea 6 imprimimos en la consola la notificación recibida.

Hemos decidido simplemente imprimir la notificación, pues el objetivo de esta unidad es comprobar como las notificaciones pueden ser enviadas y recibidas sin centrarnos en qué hacer con ellas. Desde luego que si esta fuera una aplicación real quisieras hacer algo más que solo imprimirla, como preparar el envío del pedido, o reponer del inventario las unidades vendidas, etc.

Enviando una notificación

Bueno, ya hemos explicado cómo funciona todo, por lo que ahora realizaremos un ejemplo real desde la aplicación, por lo tanto, deberemos asegurarnos de tener encendidos los siguientes Microservicios:

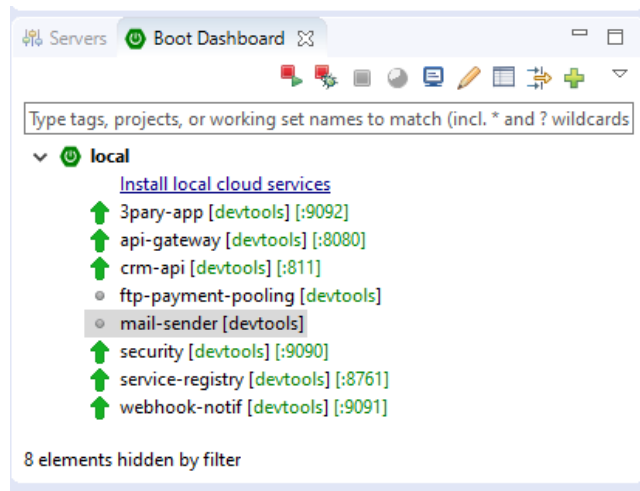


Fig 198 - Microservicios requeridos para las notificaciones.

Una vez comprobado que los Microservicios están disponibles, vamos a crear una nueva venta desde la aplicación web:

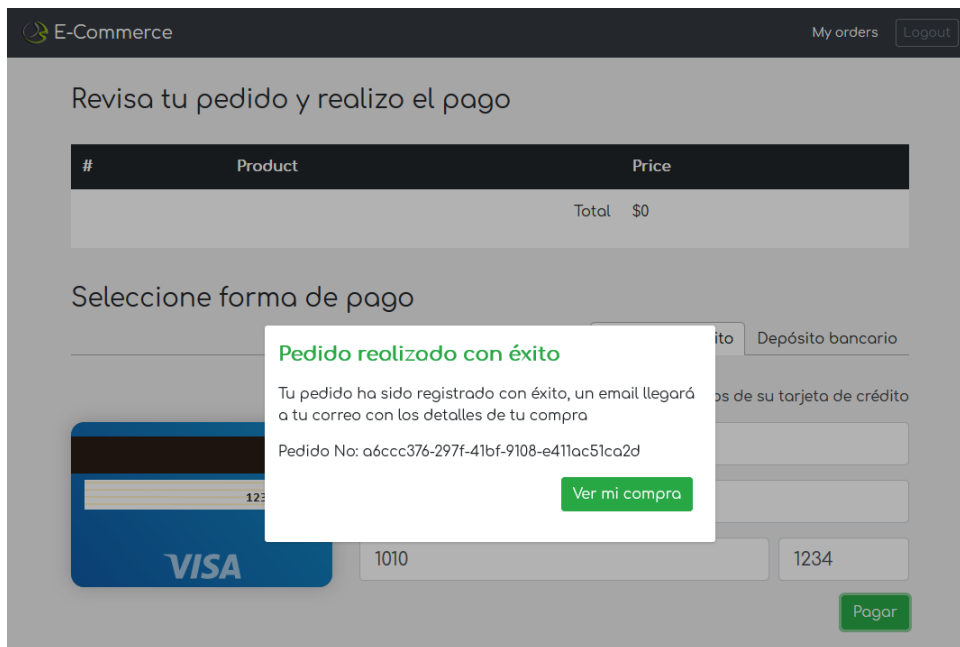


Fig 199 - Nueva venta.

Es importante pagar con tarjeta de crédito, pues los depósitos bancarios no son notificados. Una vez completado el pago, nos vamos al log del Microservicio *3party-app*:

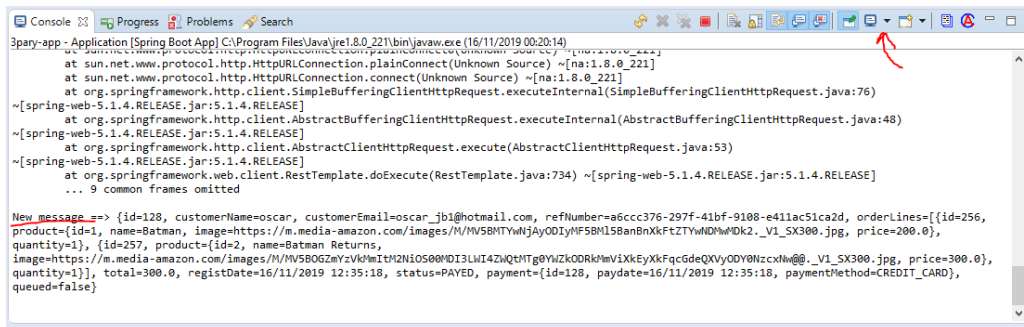


Fig 200 - Nueva notificación recibida.

En la imagen anterior puedes observar el log del Microservicio *3party-app*, en el cual se observa la notificación recibida junto con todos los campos que contiene la notificación.

Si no vez el log puede dar click en la fecha al lado del monitor que se ve en la imagen anterior para que te muestre los logs de todos los Microservicios ejecutándose.

Y bueno, con esto hemos comprobado que mediante los *Webhooks* es posible notificar a todos los componentes interesados al instante y sin la necesidad de tener que hacer un *Polling* para recuperar la nueva información.

Conclusiones

Como hemos podido observar, lo Webhook son una excelente opción cuando necesitamos crear un mecanismo para notificar a otros componentes sobre los eventos producidos en nuestra aplicación, de esta forma, brindamos una mejor experiencia al evitar que los componentes externos tengan que estar preguntando por las actualizaciones a cada rato, y en su lugar, como nosotros los que los notificamos.

Además, hemos analizado como el uso de los Webhook evitan el uso de los Polling, un patrón que puede afectar el performance a medida que el número de Requester aumenta, lo que hace que no sea una arquitectura escalable y que incluso, puede llegar a afectar la operación.

Load Balance

Cualquier aplicación diseñada para ser escalable requiere implementar mecanismos que le ayuden a agregar más poder de procesamiento de una forma fácil y que no afecta la operación. Por otro lado, las aplicaciones deben de estar diseñadas para ser robustas y con una alta disponibilidad, lo que quiere decir que si un servidor falla no debería de comprometer la operación. En este sentido, cuando una aplicación crece, eventualmente un solo servidor no podrá con la carga de trabajo, lo que nos obligará a requerir de un mayor poder de procesamiento.

Problemática

El principal problema que tienen las aplicaciones de hoy en día son las limitaciones del hardware, lo que hace imposible que un solo servidor sea capaz de atender todas las solicitudes, incluso si compramos el mejor servidor, llegar el momento en que este se vea rebasado por la gran cantidad de operaciones, lo que llevaría a un colapso del servidor. Pero entonces ¿si el hardware es limitado que podemos hacer? Una respuesta obvia es el **escalamiento vertical** el cual consiste en agregar más hardware al servidor actual, como más RAM, más procesador, más almacenamiento, etc.

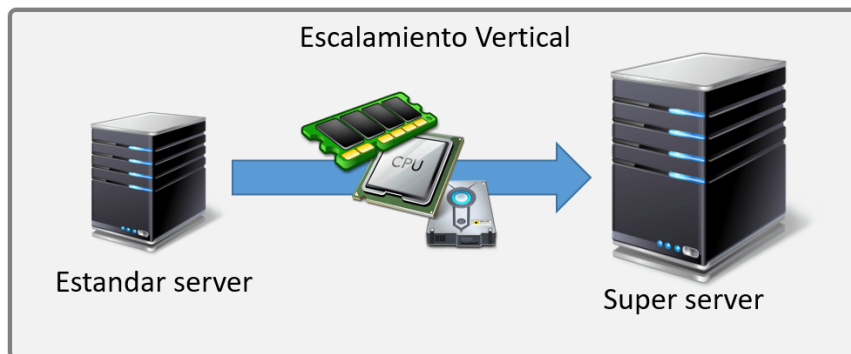


Fig 201 - Escalamiento vertical.



Nuevo concepto: Escalamiento Vertical

Término utilizado para hacer referencia al crecimiento del hardware existente, el cual consiste en agregar más procesador, almacenamiento o memoria RAM.

Pero el escalamiento vertical tiene un gran problema, por un lado, todo hardware tiene un límite de crecimiento, por lo que llegará un punto en el que ya no podremos crecer más, pero, por otro lado, toda nuestra aplicación depende de ese único servidor, por lo que, si llegara a fallar, toda la aplicación se afectaría.

Solución

Para solucionar este tipo de problema contamos con el escalamiento horizontal, el cual consiste en agregar más de un servidor a nuestro cluster, con la finalidad de que la carga de trabajo se divida en partes iguales entre todos los servidores del cluster, de esta forma, cuando necesitamos más poder de procesamiento, simplemente agregamos un nuevo servidor al cluster y dividimos la carga entre más servidores.



Nuevo concepto: Cluster

El término clúster (del inglés cluster, que significa grupo o racimo) se aplica a los conjuntos o conglomerados de servidores unidos entre sí normalmente por una red de alta velocidad y que se comportan como si fuesen un único servidor.

— *Wikipedia*



Nuevo concepto: Escalamiento Horizontal

Es la capacidad de una aplicación para dividir el trabajo entre varios servidores de una red. Se dice que una aplicación es escalable horizontalmente si al agregar nuevos servidores al cluster, esta mejora su rendimiento.

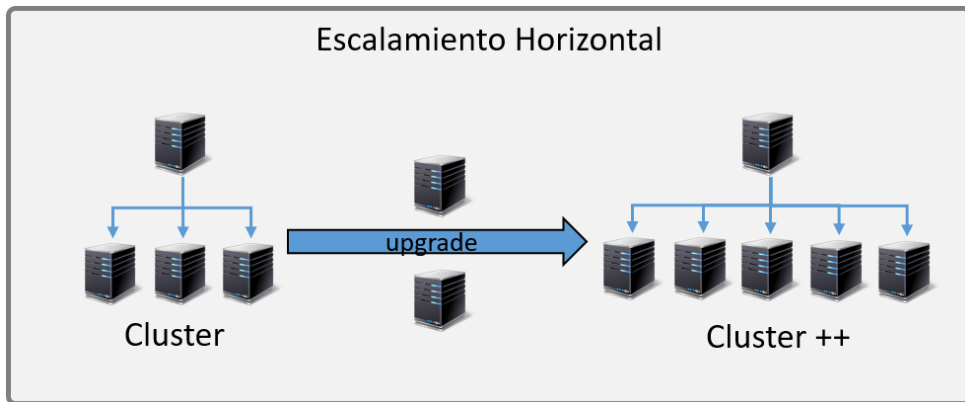


Fig 202 - Escalabilidad Horizontal.

En la imagen anterior podemos ver claramente el concepto de escalabilidad Horizontal, donde un cluster de servidores puede ser ampliado agregando más servidores, lo que mejora el rendimiento significativamente al dividir el trabajo entre más servidores.

Pero en este punto tenemos un problema, como le hacemos para que la carga de trabajo se divida entre todos los servidores de la aplicación, pues al final, cada servidor atenderá en diferente IP y puerto.

Balanceo de cargas del lado del cliente

Una primera forma de hacer esto es que el cliente sepa exactamente dónde está cada instancia de los servicios y que internamente haga el balanceo a los servidores, pero esto no es nada eficiente, pues si agregamos un nuevo servidor, hay que actualizar a todos los clientes para agregar al nuevo servidor, por otro lado, tendríamos que exponer todos nuestros servidores en la red para que pueda ser accedidos, lo cual no es muy seguro:

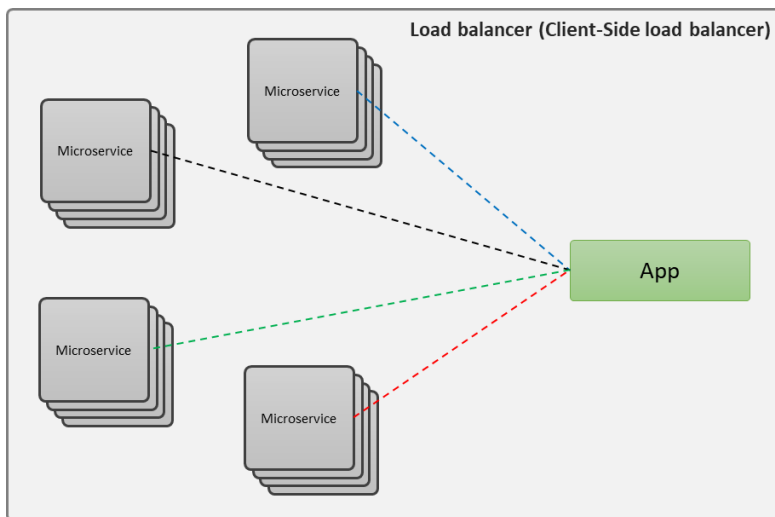


Fig 203 - Balanceo de cargas del lado del cliente.

Otra desventaja es que no tenemos el control para saber si el cliente está realizando un balanceo de cargas eficiente, es decir, nosotros ponemos las diferentes instancias de los servicios, pero no sabemos si el cliente está realizando un verdadero balanceo de cargas o simplemente siempre se va por el primer servidor.

Balanceo de cargas de lado del servidor

Otra alternativa mucho más eficiente es la proporcionar una capa de balanceo de cargas del lado del servidor, de esta forma, el cliente solo conoce la URL del servidor de balanceo de cargas e internamente de realiza el balanceo:

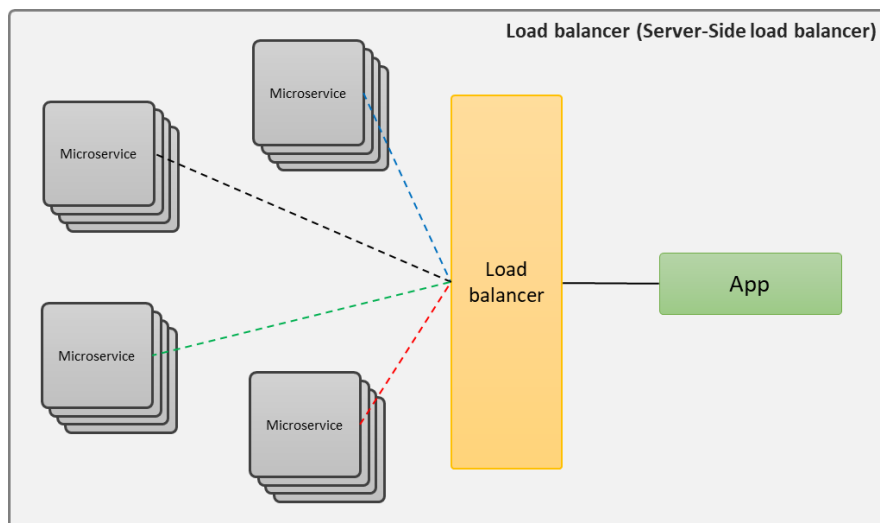


Fig 204 - Balanceo de cargas del lado del servidor.

En esta nueva arquitectura podemos ver claramente las ventajas, ya que en primer lugar el cliente solo ejecuta un URL lo cual es muy bueno, pues si agregamos una nueva instancia al cluster solo tenemos que actualizar el Load Balancer y las aplicaciones no se enteran del cambio, por otro lado, tenemos forma de controlar la forma en que se balancee la carga, es decir, podemos determinar el mejor algoritmo para realizar el Balanceo de la carga y de esta forma estamos seguros que la carga se distribuye adecuadamente entre todos los servidores.

Por otro lado, incrementamos la seguridad, pues solo exponemos a internet el balanceador de cargas, mientras que el resto de los servicios pueden estar en una red sin salida a Internet, lo que hace que sea significativamente más difícil que ocurra un ataque:

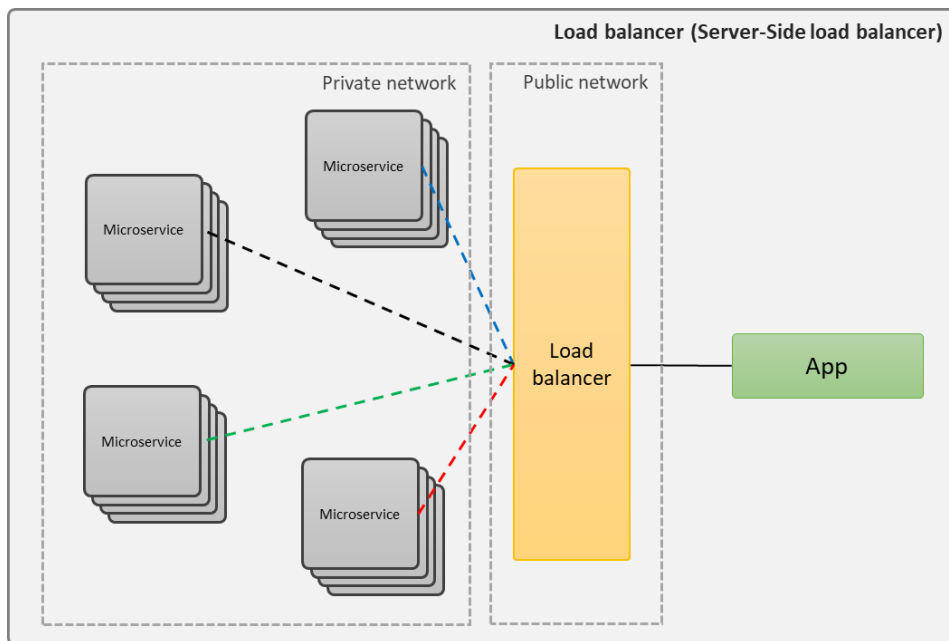


Fig 205 - Mejorando la seguridad mediante la separación de redes.

Principales algoritmos de balanceo de cargas

Parte fundamental de realizar el balance de cargas es determinar el algoritmo que utilizaremos para distribuir la carga entre los diferentes servidores, pues una correcta distribución de la carga puede liberar a los servidores de una carga desproporcionada respecto a las características del servidor, por lo que conocer algunos de los algoritmos más importantes no nos vendría nada mal.

Round-Robin

Este es el algoritmo más utilizado y el que se aplica de forma predeterminada en la mayoría de los casos si no establecemos uno diferentes, el cual consiste en repartir las peticiones de forma pareja entre todos los servidores, es decir, envía una petición a cada servidor y cuando llega al último servidor entonces vuelve a comenzar.

Este algoritmo es especialmente útil cuando todos los servidores tienen exactamente las mismas características y proporcionan el mismo servicio, pues de esta forma se puede predecir los tiempos de respuesta de todos los servidores y saber que todos están en la misma capacidad de responder al mismo número de peticiones.

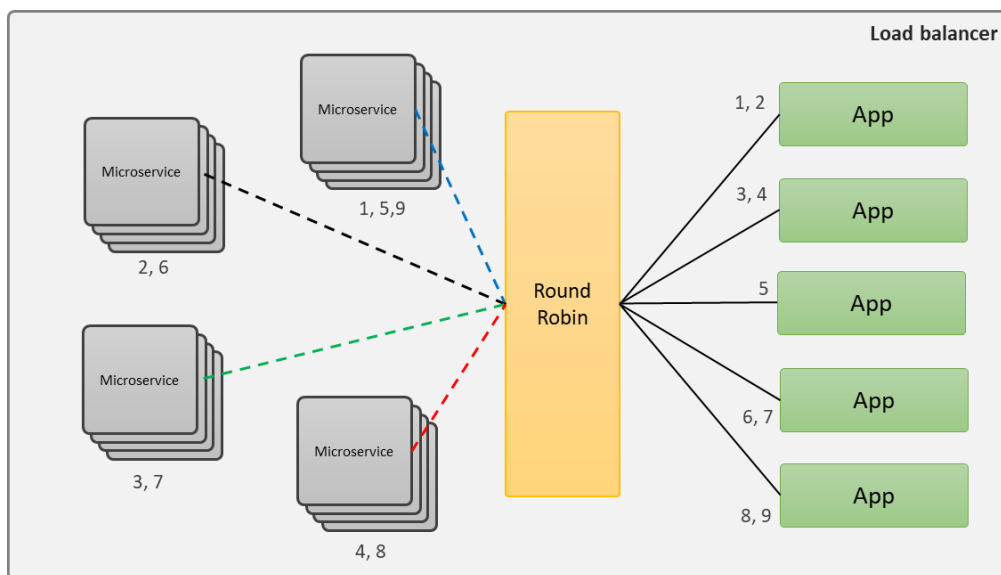


Fig 206 - Round Robin

Weighted Round Robin

Este algoritmo es prácticamente igual al Round Robin anterior, con la diferencia de que a cada servidor se le asigna una ponderación numérica estática, la cual sirve para determinar que servidor deberá de recibir más carga. A mayor ponderación, mayor será la carga de trabajo recibida.

Este algoritmo es especialmente útil cuando hay una evidente diferencia de poder de procesamiento entre los servidores que conforma el cluster, ya que un servidor con menor poder de procesamiento no podrá procesar la misma carga que uno con mayor poder, por lo tanto, se asigna una mayor ponderación a los servidores que son capaces de procesar un número mayor de solicitudes, al mismo tiempo que desahogamos a los que no pueden con esa carga.

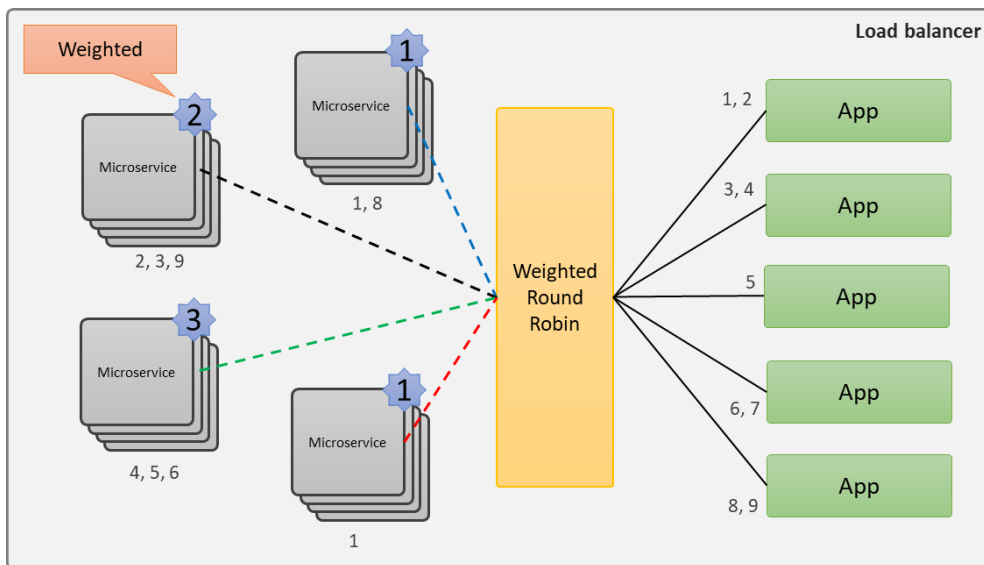


Fig 207 - Weighted Round Robin

Least connected

Este algoritmo toma en cuenta el número de conexiones actuales a los servidores, de tal forma que puede determinar cuál es que tiene menos carga basado en el número de conexiones activas al momento de enviar la solicitud, de esta forma, el balanceador no reparte la carga de forma uniforme, sino que se fija basado en las conexiones, cual está más desocupado.

Este algoritmo es utilizando cuando no queremos solo repartir la carga de forma pareja, si no que queremos tomar en cuenta la carga de trabajo actual del servidor para hacer una repartición más equitativa entre los servidores, ya que recibir el mismo número de peticiones no quiere decir exactamente que reciban la misma carga de trabajo, ya que hay solicitudes computacionalmente más complejas que otras, por lo que recibir una serie de solicitudes más complejas que otras puede tener una carga adicional sobre el servidor.

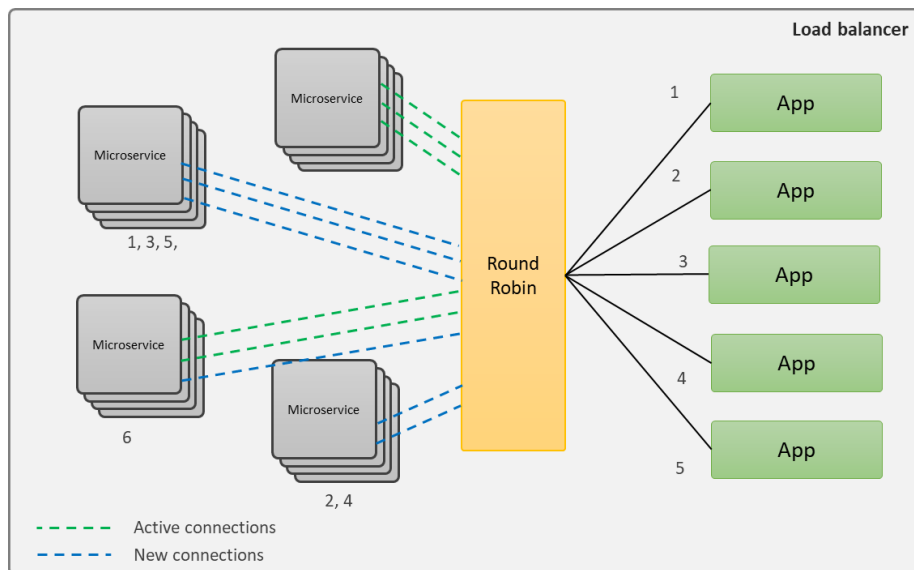


Fig 208 - Least connected

Weighted Least connection

Funciona exactamente que *Weighted Round Robin*, aplicando una ponderación a cada servidor para determinar el número de solicitudes que debe de recibir, con la diferencia que este reparte la carga basado en el número de conexiones abiertas al momento de recibir la petición.

Es utilizado en las mismas circunstancias que *Last connection*, con la diferencia de que tenemos servidores significativamente más poderosos que otros.

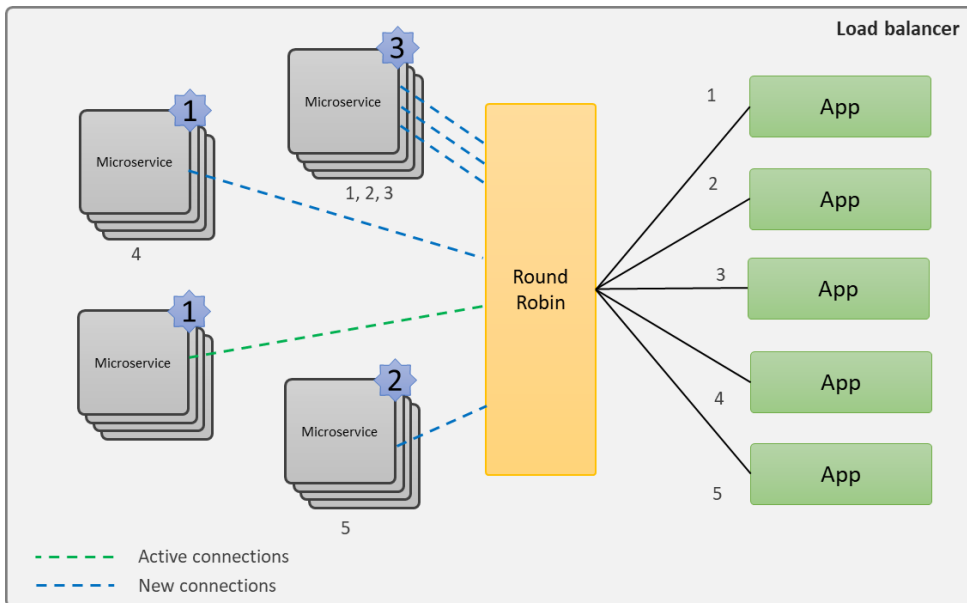


Fig 209 - Weighted Least connection

Chained Failover

Este algoritmo define una cadena de servidores a los cuales se les tendrá que enviar las peticiones, sin embargo, las peticiones no son distribuidas entre todos los servidores, sino más bien, todas son enviadas al primer servidor, si este no está disponible, las manda al segundo, y si el segundo no está disponible las manda al tercero, y así sucesivamente hasta llegar al último servidor.

Este algoritmo como su nombre lo indica, está diseñado para los casos de *Failover*, es decir, que nos permite recuperarnos ante un fallo y es normalmente utilizado en arquitecturas de alta disponibilidad, donde podemos tener más de una infraestructura de respaldo, por si una no responde, podemos tener otra lista para hacerse cargo de la operación.

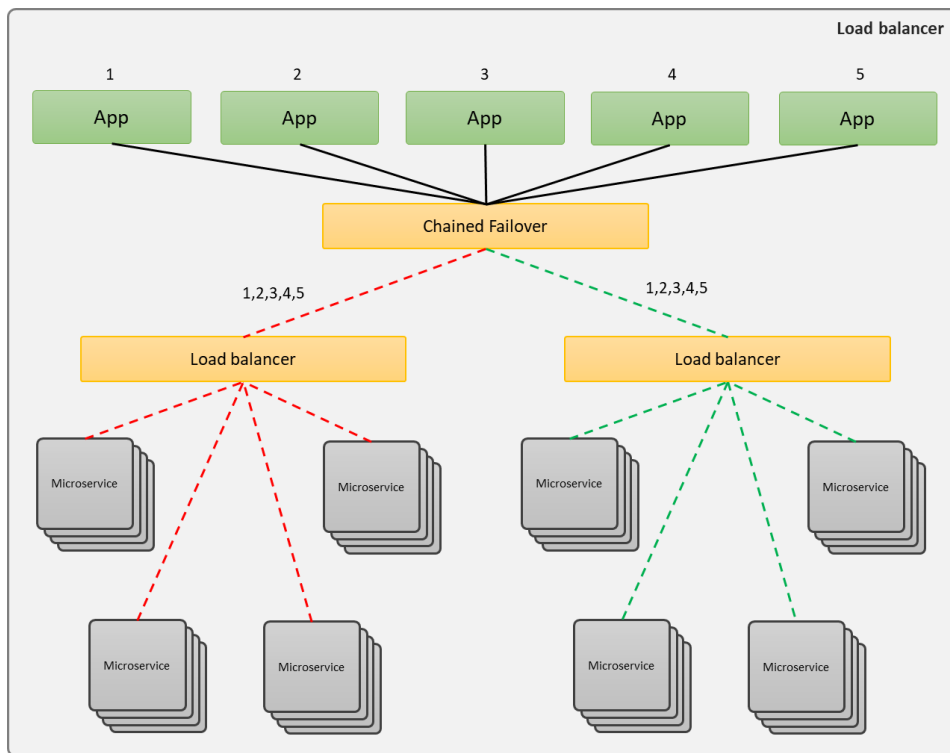


Fig 210 - Chained Failover.

En la imagen anterior puedes ver que tenemos 3 balanceadores en lugar de solo uno, y esto se debe a que podemos tener dos o más infraestructuras de respaldo, de tal forma que si la primera (izquierda) se cae, todo el tráfico recae sobre nuestra infraestructura secundaria (derecha).

De esta misma forma, cada infraestructura tiene su propio balanceador de carga para repartir el tráfico entre las diferentes instancias de los servicios.

Como comenté anteriormente, este tipo de arquitecturas es usada para aplicaciones críticas, donde mantenernos operativos es lo más importante, sin importar el costo y la administración de infraestructura adicional.

Weighted Response Time

Este algoritmo utiliza los tiempos de respuesta de los servidores para determinar a donde enviar las siguientes peticiones, de esta forma, puede determinar que

servidores están respondiendo más rápido para enviarles las siguientes peticiones, de esta forma se logra que los servidores que estén tardando más en responder no se saturen con nuevas peticiones.

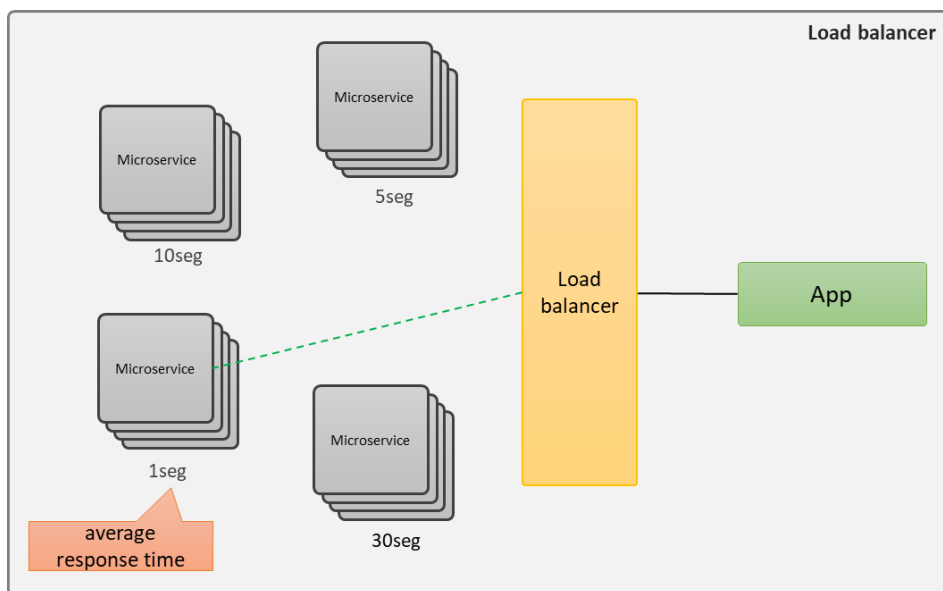


Fig 211 - Weighted Response Time.

Algo a tomar en cuenta es que el tiempo promedio de respuesta se va actualizando con cada petición, por lo que el servidor que más rápido responde comenzará a recibir más solicitudes, hasta que llegue el momento en que por tanta carga su tiempo de respuesta disminuya, y con eso, bajará el número de solicitudes que reciba, de esta forma, los servidores se van equilibrando automáticamente.

Esta es una excelente estrategia donde queremos balancear la carga mediante un método que realmente refleja la carga de trabajo de los servidores y no solo reparte peticiones según un orden definido.

Hash

Este algoritmo crea un Hash a partir de cada recurso y lo vincula a un servidor dentro del cluster, de tal forma que todas las peticiones a un mismo recurso serán atendidas por el mismo servidor, de esta forma, cada recurso disponible tendrá un Hash diferente.

Digamos que tenemos dos recursos, el recurso */resourceA* y */resourceB*, a cada uno se le asignará un Hash que se guardará en el balanceador de cargas, luego, se asignará un servidor a cada recurso, entonces podríamos tener lo siguiente:

- */resourceA* = 0fdce565929068f915e85abb9881b410 = server1
- */resourceB* = fc8ad628fcf50bc7f1e4dbfd301e36af = server2

Con ayuda de estos *Hash's*, el balanceador redireccionará las peticiones, de tal forma que todas las peticiones al recurso */resourceA* siempre se irán al servidor 1, mientras que todas las peticiones al */resourceB* se redireccionarán al servidor 2.

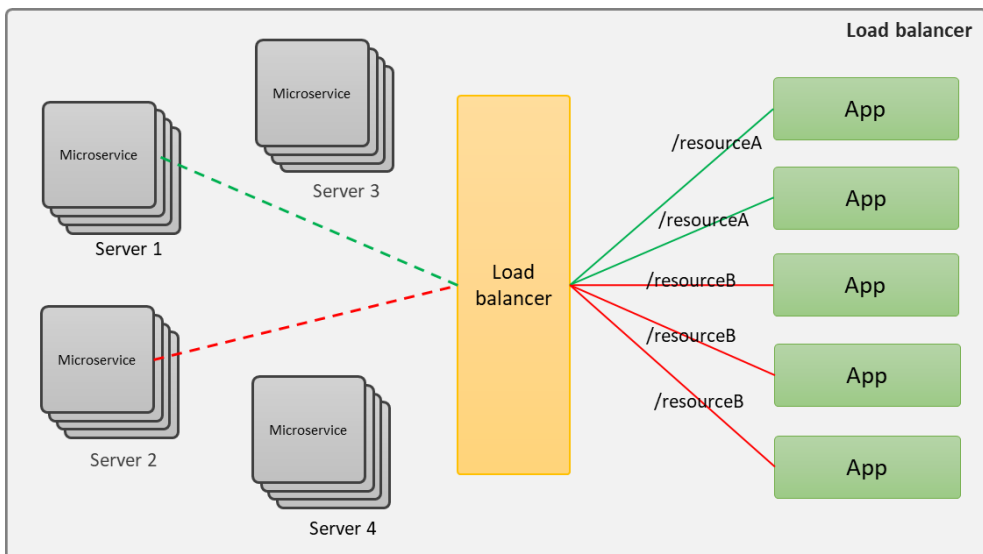


Fig 212 – Hash

Este algoritmo garantiza que el mismo servidor servirá la misma petición, incluso, si la conexión con el servidor se pierde, podremos garantizar que el mismo servidor nos responda cuando nos reconectemos. Esto es especialmente útil cuando trabajamos con caché o servicios con estado, donde el servidor tiene en memoria el contexto de nuestra actividad.

Imaginemos que el */resourceA* realiza una consulta muy compleja y tardada en la base de datos y el resultado lo guarda en caché, esto quiere decir que solo la primera persona que consuma ese recurso tendrá que esperar a que el query termine, pero los demás usuarios tendrán la respuesta en caché, y como sabemos que ante la misma URL nos responderá el mismo servidor, nos aseguramos que un solo servidor tenga el cache de esa consulta, en lugar de todos los servidores tengan que cachear la respuesta la primera vez que lo consultan, haciendo que tengamos cache duplicado en cada servidor.

Sin embargo, esta estrategia tiene un inconveniente importante, y es que no podemos distribuir la carga de forma equitativa entre los servidores, ya que puede que ciertos servidores reciban más carga que otros, todo depende de los recursos que sean asignados a un servidor determinado y la demanda que tenga esos recursos, por lo tanto, puede que los recursos más complejos sean asignados a un servidor y que además, sean los que más carga tengan, por lo que puede inclinar la carga a algunos servidores.

Random

Como su nombre lo indica, esta estrategia asigna los request de forma aleatoria, por lo que cada request que llega es redirigido a un servidor aleatorio, lo que hace impredecible a que servidor se enviara la petición.

En la práctica este algoritmo no tiene muchas aplicaciones, porque no distribuye de forma adecuada, pues no utiliza ningún criterio para determinar cuál es el mejor servidor para enviar la petición, sin embargo, el único lugar donde puede tener un poco de sentido es cuando utilizamos el balanceo del lado del cliente, fuera de eso, no es una estrategia muy adecuada y deberíamos evitar a menos que tengamos una buena razón para utilizarla.

Productos para balanceo de carga

En la práctica es posible que no tengas que desarrollar un balanceador de cargas, pues ya existen muchos productos que lo hacen de una forma muy profesional y que nos puede ahorrar mucho tiempo y esfuerzo, porque a menos que tengas un requerimiento muy concreto que requiere que lo desarrolles, no veo la necesidad de que lo hagas.

Los productos más populares para hacer balance de cargas son:

- F5 (<https://www.f5.com/services/resources/glossary/load-balancer>)
- Nginx (<https://www.nginx.com/>)
- HAProxy (<https://www.haproxy.org/>)
- LoadMaster (<https://freeloadbalancer.com/>)

En la industria los más utilizados son F5 y Nginx, ya que son los más completos y potentes de todos, por un lado, F5 tiene costo de licencia y Nginx tiene una versión Open Source.

Otra cosa a tomar en cuenta es que el balance de cargas no solo se puede hacer mediante software, si no que existe hardware especializado para hacer el balance, aunque esto solo aplica si nosotros somos los que administramos la infraestructura.

Load Balance en el mundo real

Una característica importante de cualquier API es poder distribuir la carga entre diferentes instancias para poder escalar fácilmente, pero, por otro lado, es importante el balanceo de cargas para poder ofrecer una alta disponibilidad, la cual nos permite seguir operando incluso si alguna instancia del servicio falla.

Por suerte, nuestra arquitectura fue diseñada para soportar el balanceo de carga de una forma natural y transparente, por lo que solo hace falta prender más de una instancia de un mismo Microservicio para que el balance de cargas de lleve a cabo. Para empezar, vamos a iniciar una instancia de cada Microservicio, tal y como se ve en la siguiente imagen:

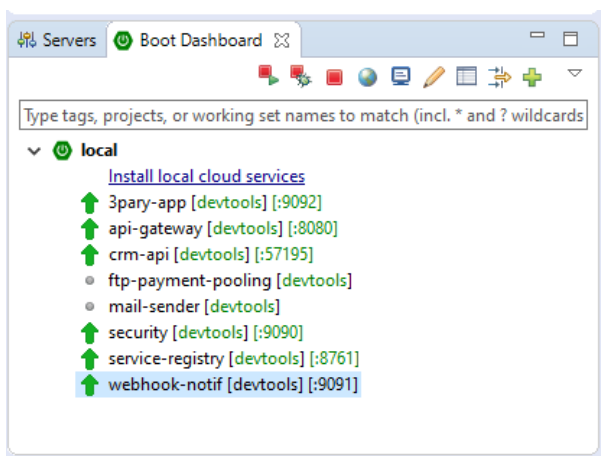


Fig 213 - Iniciando los microservicios necesarios.

Para este ejemplo vamos a prender todos los Microservicios con excepción de *ftp-payment-pooling* y *mail-sender*, lo cuales son opcionales.

Una vez que todos los Microservicios está encendidos nos dirigiremos a <http://localhost:8761>, donde veremos la siguiente pantalla:

The screenshot shows the Eureka server web interface in a browser window. The address bar shows 'localhost:8761'. The page has a dark header with the 'spring Eureka' logo and navigation links 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2019-11-18T09:34:55 -0600
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	8
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CRM	n/a (1)	(1)	UP (1) - 192.168.15.5:crm:811
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080
SECURITY	n/a (1)	(1)	UP (1) - 192.168.15.5:security:9090
WEBHOOK	n/a (1)	(1)	UP (1) - 192.168.15.5:webhook:9091

Fig 214 - Eureka server.

Esto que estamos viendo en pantalla es Eureka Server un componente desarrollado por Netflix que permite el Registro de servicios, Autodescubrimiento y el balance de carga, sin embargo, el registro de servicio (*Service Registry*) y el Autodescubrimiento de servicios (*Service Discovery*) son temas que todavía no hemos analizado, por lo que solo los explicaremos brevemente para comprender que está pasando y en las siguientes unidades regresaremos aquí para explicar todo esto a detalle.

Regresando a la imagen anterior, vemos que nos muestra una tabla con todos los Microservicios que están encendidos actualmente y lo más importante, nos muestra entre paréntesis el número de instancias encendidas de cada Microservicio. Este número es importante porque la carga de trabajo se balanceará entre todas las instancias encendidas.

Por el momento verás que existe solo una instancia de cada Microservicio, lo cual es correcto, porque son las que hemos encendido, pero hagamos una prueba, ¿recuerdas que dejamos dos Microservicios sin encender? Me refiero a los

Microservicios *ftp-payment-pooling* y *mail-sender*. Verás que no aparecen en Eureka Server, pues bien, procederemos a encenderlos:

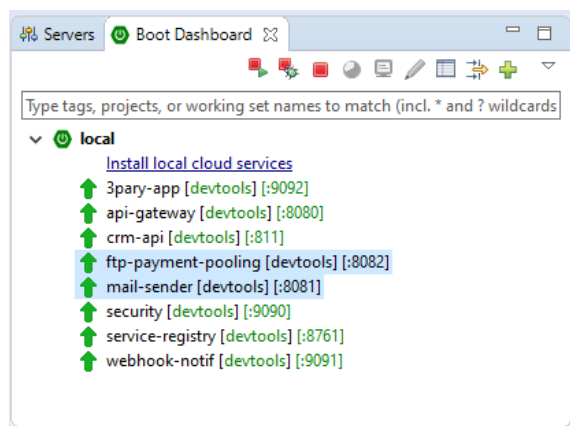


Fig 215 - Iniciando todos los Microservicios.

Una vez encendidos, actualizamos Eureka Server y podremos ver los nuevos servicios registrados:

The screenshot shows the Spring Eureka server interface in a web browser. The browser tab is labeled 'Eureka' and the address bar shows 'localhost:8761'. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2019-11-18T09:47:40 -0600
Data center	default	Uptime	00:13
		Lease expiration enabled	true
		Renews threshold	11
		Renews (last min)	12

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CRM	n/a (1)	(1)	UP (1) - 192.168.15.5:crm:811
FTP-PAYMENT-POOLING	n/a (1)	(1)	UP (1) - 192.168.15.5:ftp-payment-pooling:8082
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080
MAIL-SENDER	n/a (1)	(1)	UP (1) - 192.168.15.5:mail-sender:8081
SECURITY	n/a (1)	(1)	UP (1) - 192.168.15.5:security:9090
WEBHOOK	n/a (1)	(1)	UP (1) - 192.168.15.5:webhook:9091

Fig 216 - Eureka server.

Lo que acabas de ver no es magia, es lo que conocemos como *Service Registry*, el cual es un patrón arquitectónico que permite que todas las instancias se registren al momento de iniciar, pero no quiero entrar en los detalles ahora, pues lo veremos con detenimiento en su momento.

Lo siguiente que haremos será registrar una segunda instancia del Microservicio *crm-api*, por lo cual tendremos que hacer un pequeño ajuste a su configuración, ya que estamos corriendo en localhost y esto nos puede tener un conflicto, pues la idea de los Microservicios es que se ejecuten en diferentes máquinas o contenedores, por lo que haremos un pequeño *hack* para que esto funcione en el localhost.

Lo primero que vamos a hacer es, irnos al archivo *application.yml* del Microservicio *crm-api* y cambiar el puerto por el siguiente valor:

```

1. server:
2.   port: 811
3.   port: ${PORT:0}

```

Es decir, cambiamos el puerto 811 por el valor `${PORT:0}`, lo que hará que el puerto se asigne dinámicamente al momento de iniciar la aplicación. Si guardamos los cambios veremos que el Microservicio se reiniciará y cambia a un puerto aleatorio. Este puerto cambia con cada reinicio por lo que seguramente te asignará un puerto distinto al que puedes ver en la imagen, pero no es relevante:

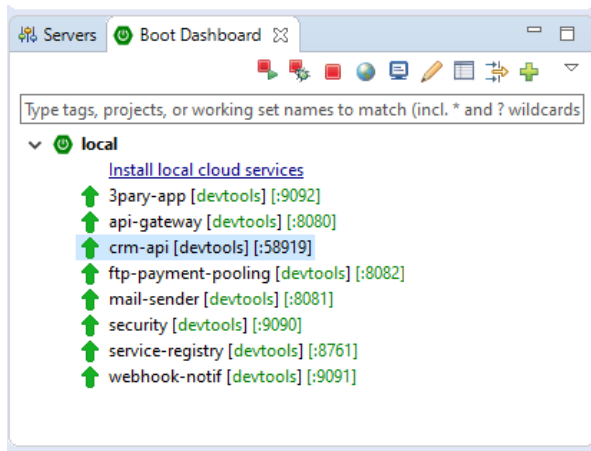


Fig 217 - Puerto asignado aleatoriamente.

El siguiente paso será agregar las siguientes líneas al mismo archivo:

```

1. eureka:
2.   client:
3.     registerWithEureka: true
4.     fetchRegistry: true
5.     serviceUrl:
6.       defaultZone: http://localhost:8761/eureka/
7.   instance:
8.     instance-id: ${spring.application.name}:${server.port}-${
   ${spring.application.instance_id:${random.value}}

```

Toma en cuenta que el valor del campo `instance-id` es solo una línea a pesar de que aquí se vea un salto de línea por el espacio disponible.

Este cambio hará que el nombre de la instancia se genera de forma aleatoria tomando como entrada el nombre de la aplicación, el puerto y un número

aleatorio. Guardamos los cambios y la aplicación se reiniciará nuevamente, por lo que el puerto asignado cambiará nuevamente.

Una vez termine de reiniciar el Microservicio, regresaremos a Eureka Server (<http://localhost:8761>) y actualizaremos la pantalla:

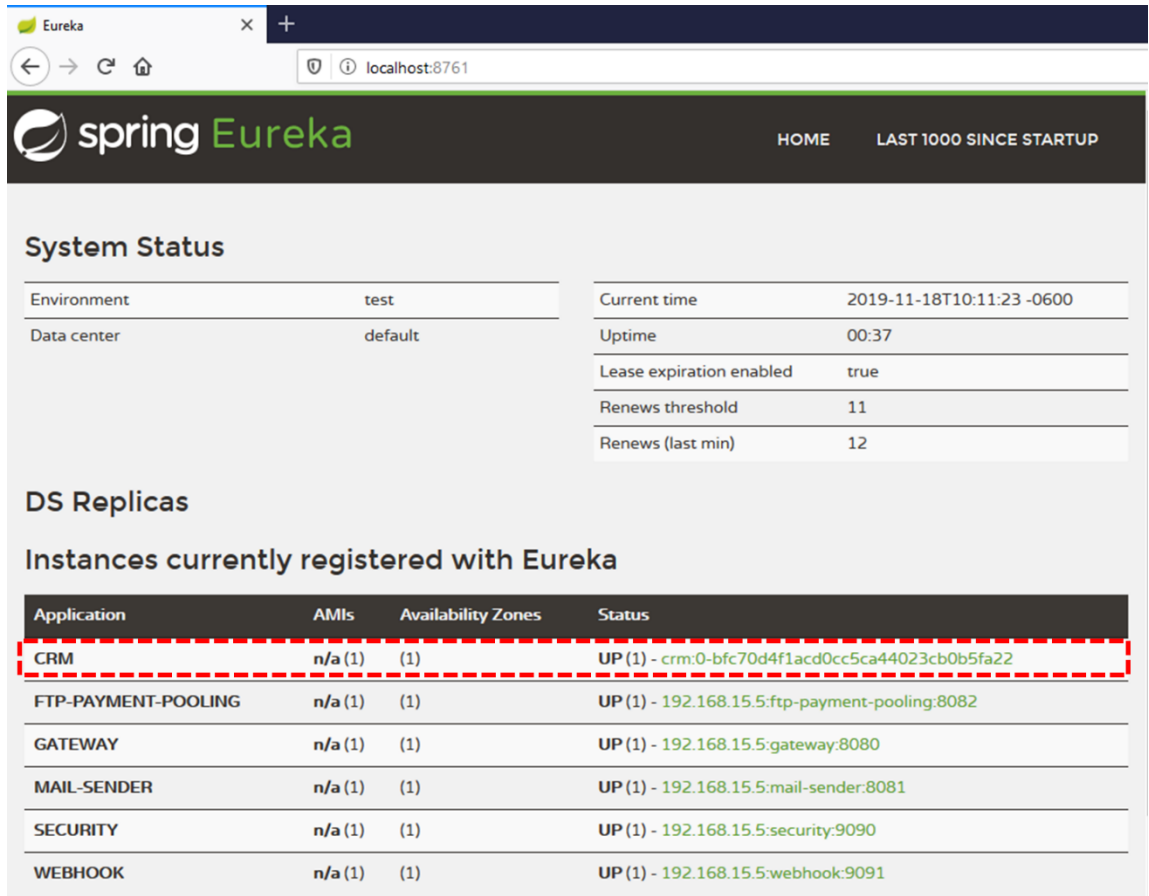


Fig 218 - Actualizando Eureka Server.

Observa que el nombre de la instancia de *crm-api* se ha generado de forma diferente al resto de instancias. Esto es importante porque nos permitirá registrar otra instancia del mismo Microservicio.

Los siguientes pasos son importantes, así que presta atención, debido a que Spring Tool Suite (IDE) es una herramienta de desarrollo, no nos mostrará más de una instancia de un Microservicio en el *Boot Dashboard*, así que tendremos que

asegurarnos de ver el número de terminales abiertas (una por instancia), para esto, damos click en la pequeña fecha que está a un lado del monitor de la barra de herramientas:

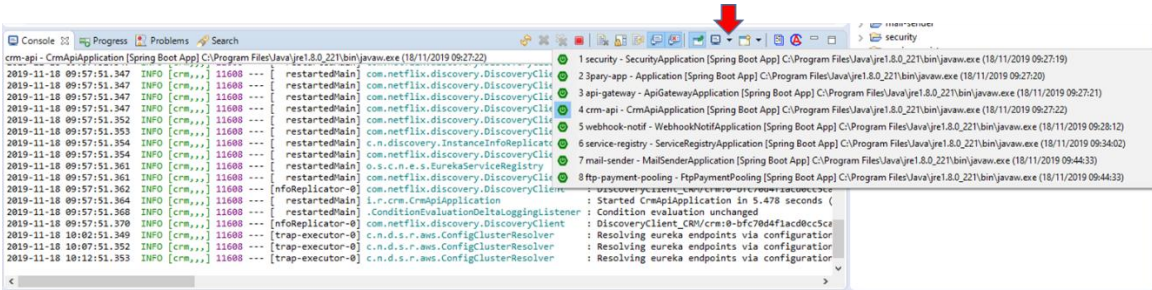


Fig 219 - Desplegando las terminales activas.

Podrás observar que existe una terminal por cada Microservicio encendido, así que ahora que sabemos cuántas instancias están encendidas, vamos a proceder a levantar otra instancia para el Microservicio *crm-api*, por lo que vamos a darle click derecho al proyecto y dar click en *Run As* → *Spring Boot App*, lo que comenzará a iniciar una nueva instancia:

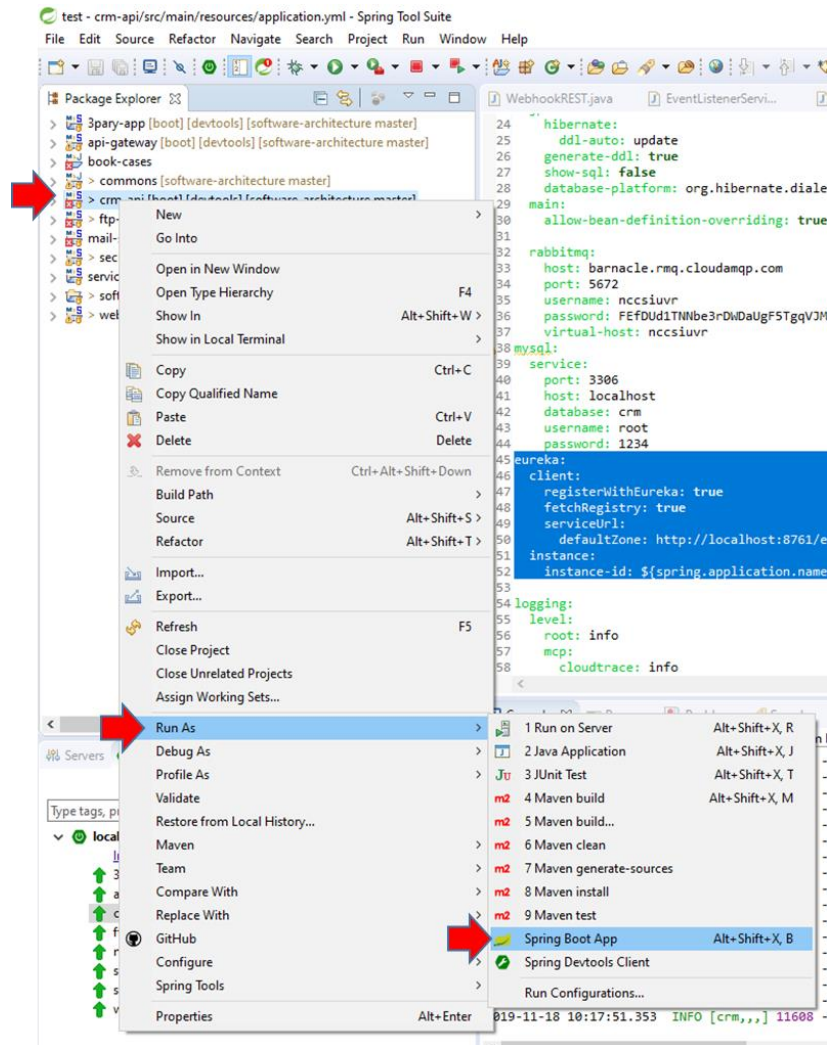


Fig 220 - Iniciando una nueva instancia del Microservicio crm-api.

Como te comenté antes, parecerá que el IDE no arranco la instancia, pues no veremos una nueva instancia en el *Boot Dashboard*, por lo que nos tendremos que ir a las terminales abiertas como te mostrar hace un momento:

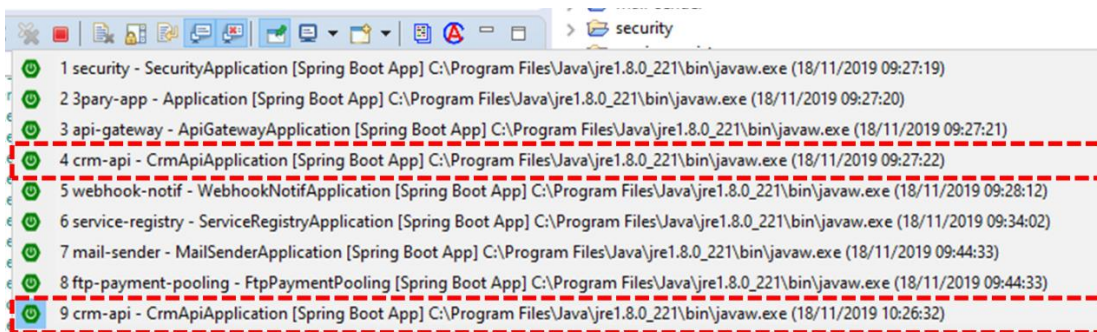


Fig 221 - Comprobando la ejecución de las dos instancias de crm-api.

Podrás observar que existen dos terminales de *crm-api*, una por cada instancia en ejecución, también puedes dar click en ellas para ver el log.

Con las dos instancias corriendo, regresamos a Eureka Server y actualizamos la pantalla:

The screenshot shows the Spring Eureka web interface in a browser window. The address bar shows 'localhost:8761'. The page has a dark header with the 'spring Eureka' logo and navigation links 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2019-11-18T10:29:36 -0600
Data center	default	Uptime	00:55
		Lease expiration enabled	true
		Renews threshold	13
		Renews (last min)	14

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CRM	n/a (2)	(2)	UP (2) - <u>crm-0-bfc70d4f1acd0cc5ca44023cb0b5fa22</u> , crm-0-bfaff55bd550ca23fc894f5dc52aa829
FTP-PAYMENT-POOLING	n/a (1)	(1)	UP (1) - 192.168.15.5:ftp-payment-pooling:8082
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080
MAIL-SENDER	n/a (1)	(1)	UP (1) - 192.168.15.5:mail-sender:8081
SECURITY	n/a (1)	(1)	UP (1) - 192.168.15.5:security:9090
WEBHOOK	n/a (1)	(1)	UP (1) - 192.168.15.5:webhook:9091

Fig 222 - Comprobando las instancias en Eureka Server.

Observa como ahora aparecen dos instancias del Microservicio *crm-api*, pero con diferente puerto. Ya con esto y gracias a nuestro *Service Discovery* y *Service Registry* podemos auto detectar las instancias y lograr el balance de carga entre las diferentes instancias del Microservicio.

El siguiente paso será entrar a la consola de cada instancia del microservicio *crm-api* y borrar todo el log para asegurarnos de ver solo lo nuevo, para ello, vamos damos click en cada terminal como te explique hace un rato y sobre la consola damos click derecho y luego en clear, como resultado, la terminal se debería de ver totalmente limpia:

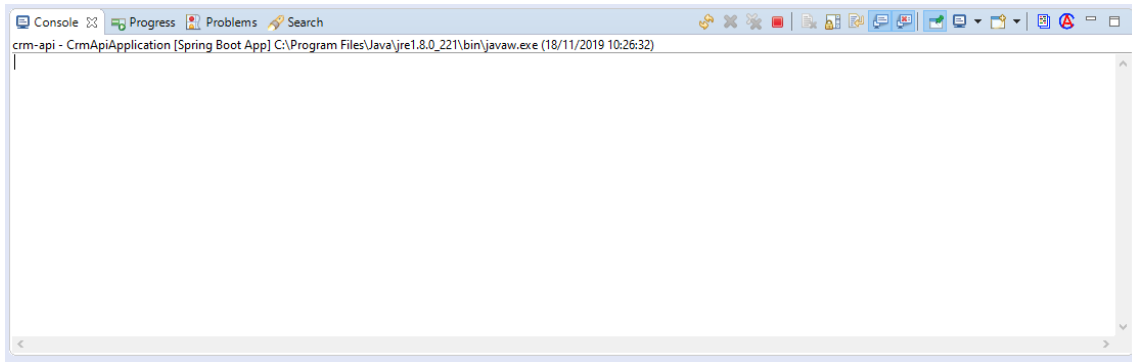


Fig 223 - Limpiar la terminal.

Repetimos el proceso para las dos instancias y luego nos vamos a la aplicación web y realizamos unas 5 compras, pagamos con tarjeta de crédito y luego regresamos a los logs de las dos terminales, allí podrás observar cómo cada instancia recibió peticiones, las peticiones que recibió cada una puede variar, por ejemplo, una puede recibir las peticiones de la consulta de los productos, otra la de agregar productos al carrito, otra de la consulta del carrito de compras y otra podría a ver procesado las compras, lo importante aquí es que si vemos que se está generando log, es por que esa instancia está recibiendo peticiones.

Algo a tomar en cuenta es que nuestra aplicación web no se conecta directamente a las instancias del Microservicio *crm-api*, sino que se conecta al *API Gateway*, el cual por medio del *Service Discovery* reconoce las instancias disponibles y hace el balance de cargas del lado del servidor, por lo que el cliente (aplicación web) nunca se entera de que del lado del servidor existe un balanceo de cargas y menos, cuantas instancias activas existen.

Tal vez no lo parezca, pero una solo llamada al API hace uso de todos los compontes dentro del cuadro rojo de nuestra arquitectura general:

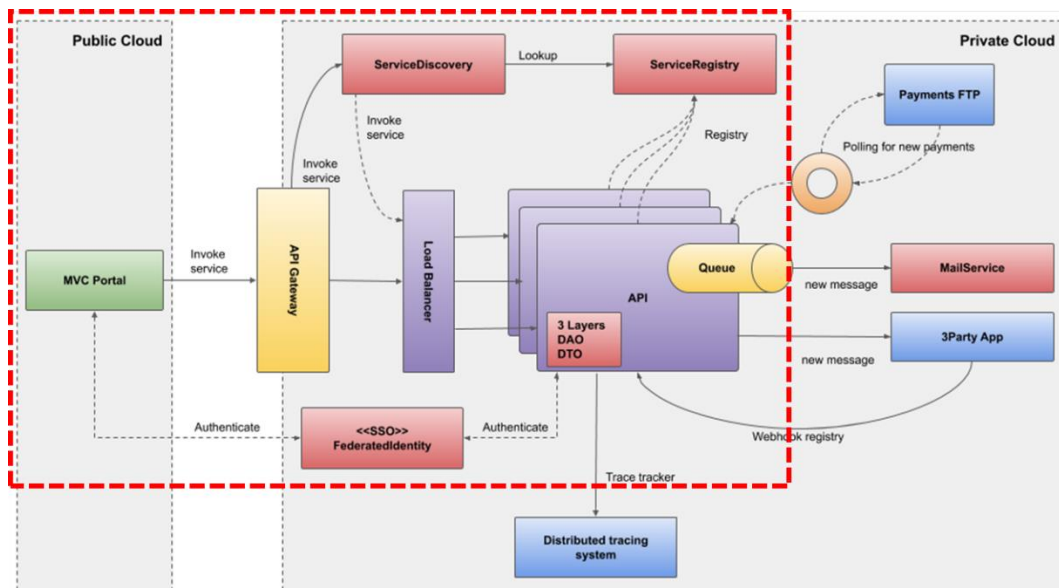


Fig 224 - Componentes utilizados en una llamada al API.

La ejecución se hace de la siguiente manera, la aplicación web (e-commerce) hace una llamada al *API Gateway*, quien, basado en el recurso solicitado, sabe a qué Microservicio redireccionar la solicitud, pero antes de eso, llama al Microservicio de seguridad (*FederatedIdentity*) para validar el token, si las credenciales son válidas, el API Gateway recupera la ubicación de todas las instancias registradas en el *ServiceDiscovery* (Eureka Server) y luego, internamente hace el balanceo de la carga entre todas las instancias disponible, finalmente, se hace la llamada al API del crm (API).

En este ejemplo solo hemos registrado más de una instancia para el Microservicio *crm-api*, pero podrías repetir el proceso para crear más instancias de cualquier otro Microservicio, con la única excepción de los Microservicios *services-registry* y *api-gateway*, ya que estos si necesitan un puerto predefinido para que la aplicación funcione, aunque desde luego se pueden poner en alta disponibilidad, pero necesitamos algunas herramientas adicionales que no analizaremos en este libro.

De esta forma, comenzamos a darnos cuenta como una arquitectura de Microservicios funciona para mezclar todos los servicios para dar una única funcionalidad.

**Importante**

Una vez que probemos las capacidades de balanceo de cargas, te sugiero que regresas la configuración inicial de todos los Microservicios para que levante en el puerto original, esto debido a que en el resto del libro haremos referencia a los puertos predefinidos, esto con la intención de poder explicarlo mejor.

Conclusiones

Hemos analizado como el balance de cargas permite que las aplicaciones escalen de forma fácil y casi ilimitada, con tan solo agregar nuevos servidores al cluster y repartiendo la carga entre todos los servidores.

El balance de cargas es hoy en día algo indispensable en las aplicaciones modernas donde la disponibilidad es algo fundamental y la llegada de la nube, que permite aprovisionar rápidamente nuevos servidores que se agregan dinámicamente a la red.

Service Registry

Trabajar con servicios es fácil, ya que nos permite exponer servicios de alto nivel que podemos fácilmente reutilizar y permite ejecutarlos con tecnologías interoperables, es decir que podemos ejecutarlas sin importar la tecnología en que estén echas, pero que pasa cuando estos servicios se van haciendo numerosos y cada vez es más complicado saber en dónde vive cada servicio o peor aún, en que puerto corren, sobre todo cuando el puerto y la IP en que se ejecuta es aleatorio.

Problemática

A medida que las empresas crecen, lo hace también la cantidad de servicios que se requieren para darle soporte a la operación, lo que provoca que cada vez sea más difícil saber exactamente en donde está cada servicio (IP, puerto), sumando a esto, es común que, para balancear la carga de trabajo, despluguemos múltiples instancias de un mismo componente, lo que hace que la cantidad de servicio publicados crezca de forma exponencial, dificultando aún más la administración de todos los servicios publicados.

Uno de los problemas más comunes cuando trabajamos con servicios, es que necesitamos saber forzosamente donde está cada servicio, y peor aún, necesitamos saber exactamente qué servicio apunta a desarrollo, cual a QA y cual producción, lo que va a haciendo que trabajar con servicios sea una tarea complicado, además, cada vez que agregamos una nueva instancia, es necesario modificar el balanceador de carga para registrar la nueva instancia.

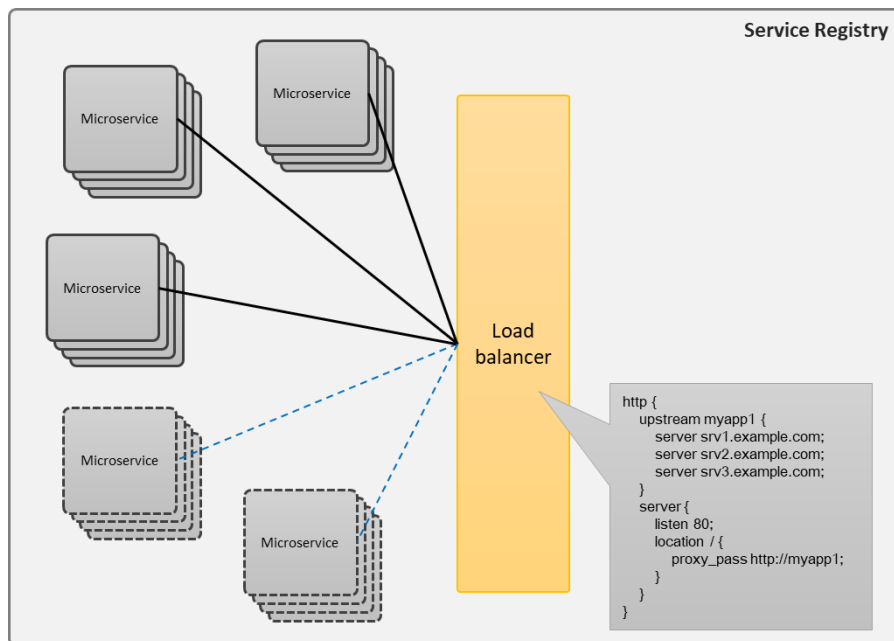


Fig 225 - Nuevas instancias de un mismo servicio.

En la imagen anterior puedes ver más claramente este problema, donde tenemos una configuración de Nginx (Balanceado de cargas) donde están registradas 3 instancias de un Microservicio, sin embargo, cuando encendemos dos nuevas instancias, al balanceador de cargas no sabrá de sus existencia, por lo que tendremos que modificar la configuración del balanceador de cargas y reiniciarlo para que este tome la nueva configuración, lo cual no es para nada bueno en un arquitectura cloud, donde podemos estirar la nube para agregar o quitar instancias dinámicamente a medida que la carga de trabajo crece o disminuye, por lo que no podemos tener a una persona que esté registrando y eliminado los servidores en el balanceado de cargas.

Solución

El patrón *Service Registry* propone crear un servidor centralizado donde todos los servicios se registren al momento de encender, de esta forma, cada servicio le tendrá que enviar la dirección IP, el puerto en el que responde al servidor y finalmente, el identificador del servicio, que por lo general es un nombre

alfanumérico que ayude a identificarlo, de esta forma, el servidor central o registro, sabrá exactamente dónde está cada servicio disponible.

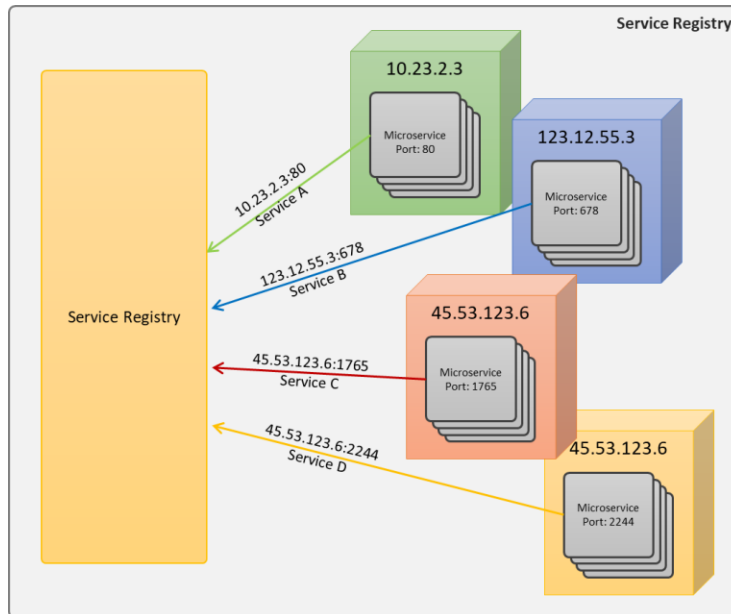


Fig 226 - Registro de servicios.

Esto lo que provoca es que mediante el Service Registry sepamos exactamente qué servicios están disponibles y su ubicación física, ya que por lo general proporciona una interface gráfica que nos permite ver gráficamente todos los servicios activos.

Otra de las características de este patrón es que los servicios que se registren, deberán estar enviando constantemente una señal al registro, la cual le indica que los servicios siguen disponibles al pasar del tiempo. Ha esta señal se le conoce como heartbeat (latidos).

Mediante los latidos el Registro puede identificar qué servicios han dejado de funcionar, ya que, si deja de recibir los latidos, el Registro lo tomará como que ese servicio se ha caído, lo que puede disparar una alerta al administrador del sistema.

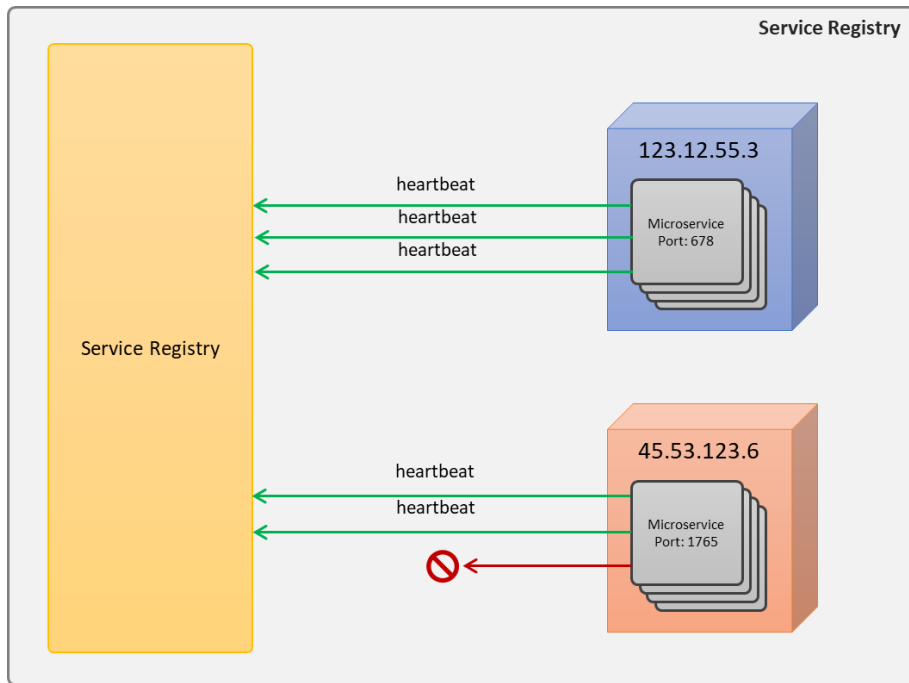


Fig 227 - Heartbeat.

Como podemos apreciar en la imagen, tenemos un par de servicios ya registrados, los cuales comienzan a mandar los latidos al Registro, sin embargo, el segundo (abajo) comienza a fallar y el servidor se cae, por lo que el servicio se apaga abruptamente, por lo tanto, el Registro no sabe de momento del servicio dejó de responder, sin embargo, después de un tiempo, el Registro detecta que no ha recibido los latidos de ese servicio, por lo que automáticamente lo marca como fuera de servicio y puede mandar en ese momento una notificación a algún administrador o DevOps que administre el servicio.

Service Registry en el mundo real

Para implementar un *Service Registry* vamos a utilizar Eureka, el cual es parte del Stack tecnológico open Source de Netflix, y que es parte de *Open Source Software* (OOS). Eureka es una tecnología super poderosa que nos permite levantar un *Service Registry* con unas cuantas líneas, además, incluye capacidades de *Service Discovery* (lo analizaremos en la siguiente sección) y al mismo, permite el balance de carga entre los diferentes servicios registrados.

Para comenzar a explicar cómo funciona nuestro *Service Registry* vamos a comenzar a pagando todos nuestros Microservicios, para comenzar desde cero:

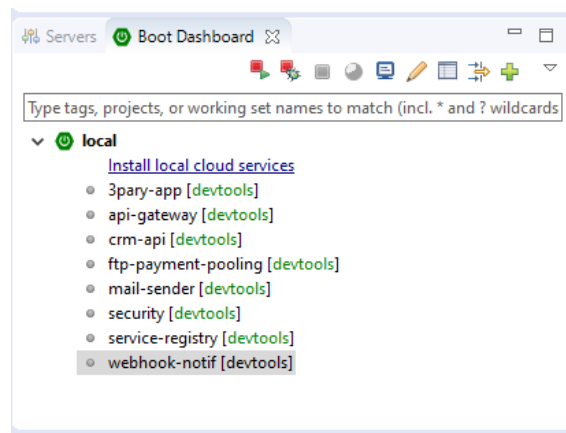


Fig 228 - Apagando todos nuestros Microservicios.

Una vez que hemos apagado todos los Microservicios, encenderemos el Microservicio *service-registry* y esperamos a que termine para analizar el log:

```
known remote regions: []
2019-11-22 13:19:44.871 INFO 12000 --- [ restartedMain] c.n.eureka.DefaultEurekaServerContext : Initialized
2019-11-22 13:19:44.881 INFO 12000 --- [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'
2019-11-22 13:19:44.968 INFO 12000 --- [ restartedMain] o.s.c.n.e.s.EurekaServiceRegistry : Registering application UNKNOWN with eureka with
status UP
2019-11-22 13:19:44.971 INFO 12000 --- [ Thread-20] o.s.c.n.e.server.EurekaServerBootstrap : Setting the eureka configuration..
2019-11-22 13:19:44.972 INFO 12000 --- [ Thread-20] o.s.c.n.e.server.EurekaServerBootstrap : Eureka data center value eureka.datacenter is not
set, defaulting to default
2019-11-22 13:19:44.972 INFO 12000 --- [ Thread-20] o.s.c.n.e.server.EurekaServerBootstrap : Eureka environment value eureka.environment is not
set, defaulting to test
2019-11-22 13:19:44.985 INFO 12000 --- [ Thread-20] o.s.c.n.e.server.EurekaServerBootstrap : isAws returned false
2019-11-22 13:19:44.985 INFO 12000 --- [ Thread-20] o.s.c.n.e.server.EurekaServerBootstrap : Initialized server context
2019-11-22 13:19:44.985 INFO 12000 --- [ Thread-20] c.n.e.r.PeerAwareInstanceRegistryImpl : Got 1 instances from neighboring DS node
2019-11-22 13:19:44.985 INFO 12000 --- [ Thread-20] c.n.e.r.PeerAwareInstanceRegistryImpl : Renew threshold is: 1
2019-11-22 13:19:44.985 INFO 12000 --- [ Thread-20] c.n.e.r.PeerAwareInstanceRegistryImpl : Changing status to UP
2019-11-22 13:19:44.996 INFO 12000 --- [ Thread-20] e.s.EurekaServerInitializerConfiguration : Started Eureka Server
2019-11-22 13:19:45.024 INFO 12000 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8761 (http) with context
path ''
2019-11-22 13:19:45.025 INFO 12000 --- [ restartedMain] o.s.c.n.e.s.EurekaAutoRegistration : Updating port to 8761
2019-11-22 13:19:45.027 INFO 12000 --- [ restartedMain] i.r.registry.ServiceRegistryApplication : Started ServiceRegistryApplication in 7.316 seconds
(JVM running for 15.327)
2019-11-22 13:20:44.986 INFO 12000 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:21:44.987 INFO 12000 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
```

Fig 229 - Iniciando el Microservicio service-registry.

Si revisamos el log, podremos ver que la aplicación inicio en el puerto 8761, por lo tanto, iremos al navegador y abriremos la URL <http://localhost:8761>.

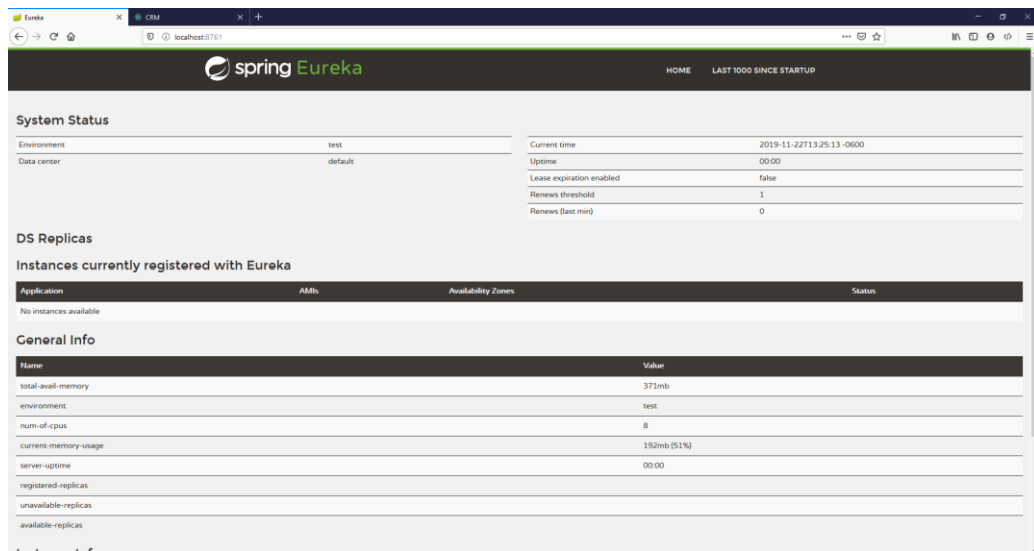
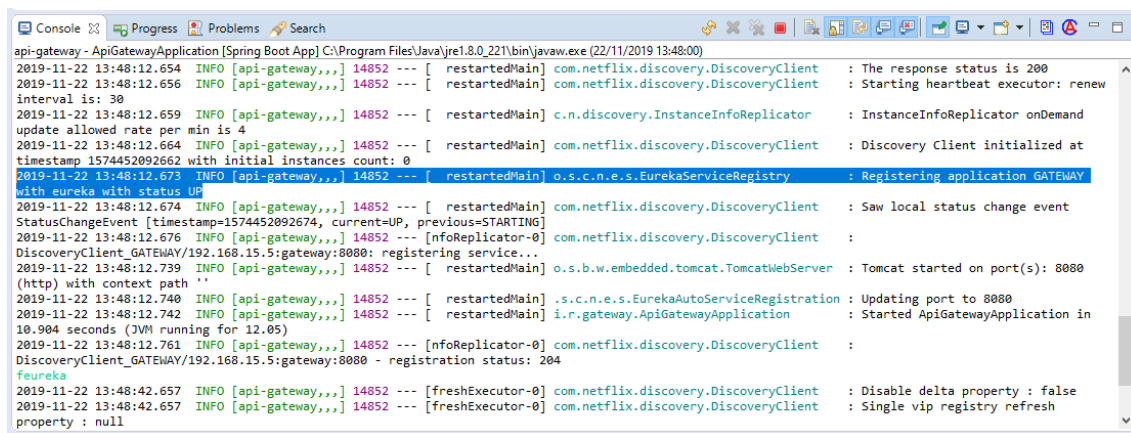


Fig 230 - Eureka server.

Lo que estás viendo ahora en pantalla es nuestro Service Registry, en el cual podremos ver todas las instancias de nuestros Microservicios encendidos, sin embargo, como tenemos todo apagado no podemos ver ningún Microservicio.

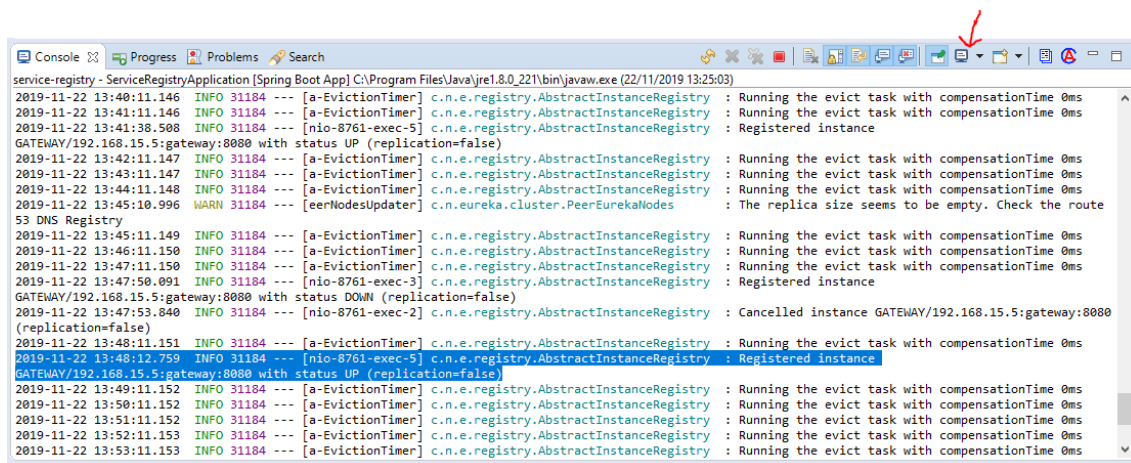
Lo que haremos ahora es ir al *Boot Dashboard* y encender el Microservicio *api-gateway*. Como consecuencia se iniciará una nueva terminal con la salida del *api-gateway*:



```
api-gateway - ApiGatewayApplication [Spring Boot App] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (22/11/2019 13:48:00)
2019-11-22 13:48:12.654 INFO [api-gateway,,,] 14852 --- [ restartedMain] com.netflix.discovery.DiscoveryClient : The response status is 200
2019-11-22 13:48:12.656 INFO [api-gateway,,,] 14852 --- [ restartedMain] com.netflix.discovery.DiscoveryClient : Starting heartbeat executor: renew
interval is: 30
2019-11-22 13:48:12.659 INFO [api-gateway,,,] 14852 --- [ restartedMain] c.n.discovery.InstanceInfoReplicator : InstanceInfoReplicator onDemand
update allowed rate per min is 4
2019-11-22 13:48:12.664 INFO [api-gateway,,,] 14852 --- [ restartedMain] com.netflix.discovery.DiscoveryClient : Discovery Client initialized at
timestamp 1574452092662 with initial instances count: 0
2019-11-22 13:48:12.673 INFO [api-gateway,,,] 14852 --- [ restartedMain] o.s.c.n.e.s.EurekaServiceRegistry : Registering application GATEWAY
with eureka with status UP
2019-11-22 13:48:12.674 INFO [api-gateway,,,] 14852 --- [ restartedMain] com.netflix.discovery.DiscoveryClient : Saw local status change event
StatusChangeEvent [timestamp=1574452092674, current=UP, previous=STARTING]
2019-11-22 13:48:12.676 INFO [api-gateway,,,] 14852 --- [ nfoReplicator-0] com.netflix.discovery.DiscoveryClient :
DiscoveryClient_GATEWAY/192.168.15.5:gateway:8080: registering service...
2019-11-22 13:48:12.739 INFO [api-gateway,,,] 14852 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080
(http) with context path
2019-11-22 13:48:12.740 INFO [api-gateway,,,] 14852 --- [ restartedMain] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8080
2019-11-22 13:48:12.742 INFO [api-gateway,,,] 14852 --- [ restartedMain] i.r.gateway.ApiGatewayApplication : Started ApiGatewayApplication in
10.984 seconds (JVM running for 12.05)
2019-11-22 13:48:12.761 INFO [api-gateway,,,] 14852 --- [ nfoReplicator-0] com.netflix.discovery.DiscoveryClient :
DiscoveryClient_GATEWAY/192.168.15.5:gateway:8080 - registration status: 204
feureka
2019-11-22 13:48:42.657 INFO [api-gateway,,,] 14852 --- [ freshExecutor-0] com.netflix.discovery.DiscoveryClient : Disable delta property : false
2019-11-22 13:48:42.657 INFO [api-gateway,,,] 14852 --- [ freshExecutor-0] com.netflix.discovery.DiscoveryClient : Single vip registry refresh
property : null
```

Fig 231 - Inicio del Microservicio *api-gateway*.

Quiero que observes la línea subrayada de la imagen anterior, en la cual podemos leer lo siguiente: “*Registering application GATEWAY with eureka with status UP*”. Esto nos indica que el Microservicio se ha registrado de forma automática justo al momento de iniciar. Por otro lado, si cambiamos a la terminal del Microservicio *service-registry*, veremos que Eureka también detecto que el Microservicio *api-gateway* se ha registrado:

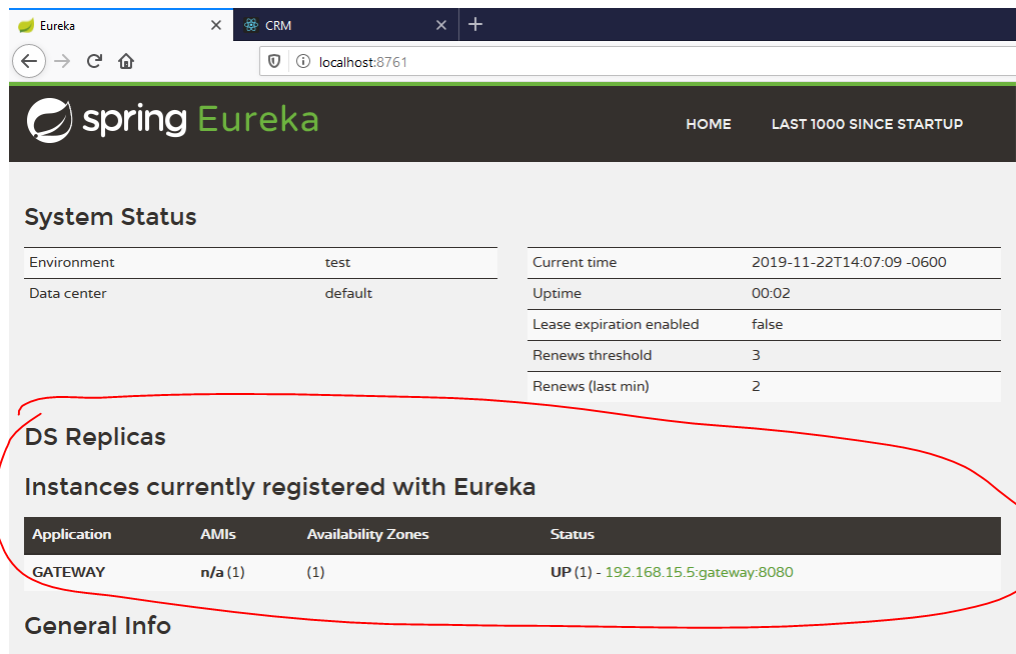


```
service-registry - ServiceRegistryApplication [Spring Boot App] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (22/11/2019 13:25:03)
2019-11-22 13:40:11.146 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:41:11.146 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:41:38.508 INFO 31184 --- [nio-8761-exec-5] c.n.e.registry.AbstractInstanceRegistry : Registered instance
GATEWAY/192.168.15.5:gateway:8080 with status UP (replication=false)
2019-11-22 13:42:11.147 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:43:11.147 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:44:11.148 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:45:10.996 WARN 31184 --- [eureka-cluster-0] c.n.eureka.cluster.PeerEurekaNodes : The replica size seems to be empty. Check the route
53 DNS Registry
2019-11-22 13:45:11.149 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:46:11.150 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:47:11.150 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:47:50.091 INFO 31184 --- [nio-8761-exec-3] c.n.e.registry.AbstractInstanceRegistry : Registered instance
GATEWAY/192.168.15.5:gateway:8080 with status DOWN (replication=false)
2019-11-22 13:47:53.840 INFO 31184 --- [nio-8761-exec-2] c.n.e.registry.AbstractInstanceRegistry : Cancelled instance GATEWAY/192.168.15.5:gateway:8080
(replication=false)
2019-11-22 13:48:11.151 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:48:12.759 INFO 31184 --- [nio-8761-exec-5] c.n.e.registry.AbstractInstanceRegistry : Registered instance
GATEWAY/192.168.15.5:gateway:8080 with status UP (replication=false)
2019-11-22 13:49:11.152 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:50:11.152 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:51:11.152 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:52:11.153 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2019-11-22 13:53:11.153 INFO 31184 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
```

Fig 232 - Terminal del *service-registry*.

En la terminal del Microservicio *service-registry* podemos leer la siguiente línea *"Registered instance GATEWAY/192.168.15.5:gateway:8080 with status UP"*. Esta línea nos da mucha información del servicio registrado, ya que nos dice el nombre del Microservicio, la ip y el puerto, con lo cual ya podríamos localizarlo más fácil y posteriormente ejecutarlo.

Pero para comprobar que efectivamente el Microservicio *api-gateway* se registró, actualizaremos la página de Eureka (<http://localhost:8761>) y veremos que ahora si aparece el Microservicio:



The screenshot shows the Spring Eureka web interface in a browser. The address bar shows `localhost:8761`. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into sections. The 'System Status' section contains two tables. The first table lists 'Environment' as 'test' and 'Data center' as 'default'. The second table lists 'Current time' as '2019-11-22T14:07:09 -0600', 'Uptime' as '00:02', 'Lease expiration enabled' as 'false', 'Renews threshold' as '3', and 'Renews (last min)' as '2'. Below this is the 'DS Replicas' section, which is circled in red. It contains the heading 'Instances currently registered with Eureka' and a table with the following data:

Application	AMIs	Availability Zones	Status
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080

Below the table is a 'General Info' section.

Fig 233 - Validando el registro en Eureka.

Con esto demostramos que el registro de servicios es totalmente posible, sin embargo, regresaremos al *Boot Dashboard* he iniciaremos todos los Microservicios:

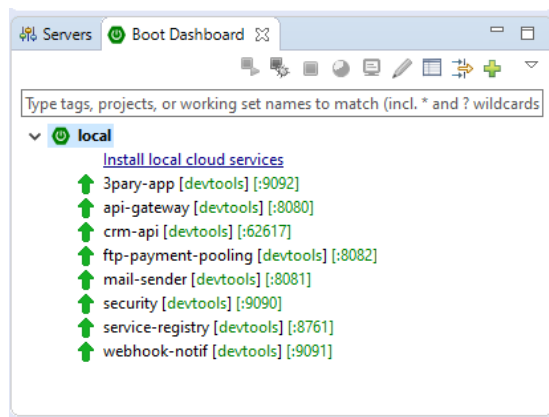


Fig 234 - Encender todos los Microservicios.

Una vez que todos los Microservicios aparecen como encendidos, regresaremos a Eureka y actualizaremos la página:

The screenshot shows the Spring Eureka web interface in a browser window. The browser tabs include 'Eureka' and 'CRM'. The address bar shows 'localhost:8761'. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2019-11-22T14:32:22 -0600
Data center	default	Uptime	00:28
		Lease expiration enabled	true
		Renews threshold	11
		Renews (last min)	12

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CRM	n/a (1)	(1)	UP (1) - 192.168.15.5:crm:811
FTP-PAYMENT-POOLING	n/a (1)	(1)	UP (1) - 192.168.15.5:ftp-payment-pooling:8082
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080
MAIL-SENDER	n/a (1)	(1)	UP (1) - 192.168.15.5:mail-sender:8081
SECURITY	n/a (1)	(1)	UP (1) - 192.168.15.5:security:9090
WEBHOOK	n/a (1)	(1)	UP (1) - 192.168.15.5:webhook:9091

Fig 235 - Comprobando el registro en Eureka.

Observa como todas las instancias de nuestros Microservicios se registraron correctamente, y sabemos desde aquí donde están físicamente cada servicio, ya que nos proporciona la IP y el puerto en el que están respondiendo.

Como implementar Eureka Server

Ya que comprobamos cómo funciona el *Service Registry*, analicemos como se implementó en nuestra API.

Lo primero que vimos es que el Microservicio *service-registry* es el que levanta el *Service Registry* con Eureka, por lo que comenzaremos analizado la estructura del proyecto:

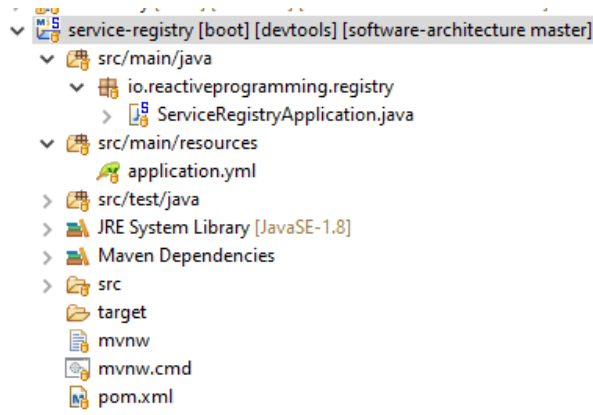


Fig 236 - Proyecto *service-registry*.

Quiero que observes que el proyecto solo tiene una clase y el archivo de configuración *application.yml*, así que con solo una clase hemos podido levantar todo un Registro, pero ahora de seguro estarás pensando, esa clase ha de tener cientos de líneas. Veamos la clase *ServiceRegistryApplication*:

```
1. package io.reactiveprogramming.registry;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5. import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6.
7. @SpringBootApplication
8. @EnableEurekaServer
9. public class ServiceRegistryApplication {
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(ServiceRegistryApplication.class, args);
13.     }
14. }
```

Así como lo vez, 14 líneas de código han sido suficientes para levantar un registro con Eureka, y es que el metadato *@EnableEurekaServer* es lo único que necesitamos para que el servidor se levante. Además, requerimos una configuración mínima en el archivo *application.yml*:

```
1. server:
2.     port: 8761
```

```

3. eureka:
4.   instance:
5.     hostname: localhost
6.   client:
7.     registerWithEureka: false
8.     fetchRegistry: false
9.     serviceUrl:
10.    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

```

Básicamente, lo que configuramos en el hostname y el puerto en el que va a funcionar Eureka Server, además tenemos la propiedad *defaultZone*, en la cual no quisiera entrar en detalles, pues sería entrar en cuestiones avanzadas de Eureka, pero básicamente nos permite definir zonas para separar los servicios en distintas zonas.

Ya explicado cómo funciona Eureka Server, pasaremos a explicar cómo funcionan los clientes, es decir los Microservicios que se registran en Eureka, para esto, podemos analizar el Microservicio que sea, al final todos funcionan igual, por lo que agarraremos uno al azar, en este caso analizaremos el Microservicio *security*:

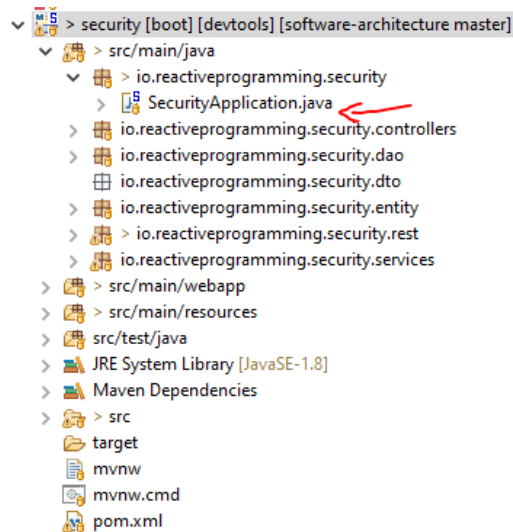


Fig 237 - Estructura del microservicio security.

Dentro de este proyecto abriremos el archivo *SecurityApplication*, el cual se ve la siguiente forma:

```
1. package io.reactiveprogramming.security;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5. import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
6.
7.
8. @SpringBootApplication
9. @EnableEurekaClient
10. public class SecurityApplication {
11.
12.     public static void main(String[] args) {
13.         SpringApplication.run(SecurityApplication.class, args);
14.     }
15. }
```

Observa que hemos anotado la clase con la *@EnableEurekaClient*, la cual permite que el componente se auto registre con Eureka, sin una sola línea de código, aunque si debemos de realizar algunas configuraciones en el archivo *application.yml*:

```
1. ...
2.
3. eureka:
4.     instance:
5.         prefer-ip-address: true
6.     client:
7.         registerWithEureka: true
8.         fetchRegistry: true
9.         serviceUrl:
10.             defaultZone: http://localhost:8761/eureka/
```

Dentro de la configuración, la línea 10 es la más importante, pues le indicamos en que zonas de Eureka nos queremos registrar, la cual corresponde con la que configuramos de Eureka Server.

Solo con esta pequeña configuración hacemos que cualquier Microservicio se registre de forma automática a nuestro registro.

En este punto podemos ver claramente las ventajas de tener un registro que nos ayude a administrar nuestras instancias, pero de poco sirve si no le sacamos provecho de este registro, por ello, en la siguiente sección analizaremos el patrón *Service Discovery*, el cual se complementa con el *Service Registry* para detectar las instancias registradas y balancear la carga entre todas estas.

Conclusión

El Service Registry es hoy en día un componente indispensable en arquitectura de Microservicios o aplicaciones desarrolladas nativamente para la nube, pues permite que los servicios se puedan registrar independientemente de su ubicación física, lo que hace que podamos saber fácilmente su ubicación y posteriormente utilizar técnicas de auto descubrimiento para localizarlos y balancear la carga.

A pesar de que es uno de los elementos más importantes en una arquitectura de Microservicios, por desgracia muchos arquitectos la olvidan al momento de diseñar sus aplicaciones, lo que hace aplicaciones dependientes de las ubicaciones físicas de los Microservicios y finalmente, difíciles de escalar.

Service Discovery

En la sección pasada hablábamos del patrón *Service Registry*, y analizábamos como mediante este patrón era posible tener un catálogo actualizado de todos los Servicios disponibles, sin embargo, poco o nada de provecho podemos tener de un registro si no tomamos ventaja de este registro para crear aplicaciones mucho más robusta, es por ello que en esta sección analizaremos el patrón *Service Discovery* el cual hace complemento con el patrón *Service Registry*, por lo que si te saltaste la unidad anterior, te recomiendo que regreses antes de continuar con esta unidad, ya haremos uso de los conceptos aprendidos.

Problemática

Uno de los mayores problemas cuando trabajamos con servicios, es saber su ubicación física, ya que cada servicio responde en un dirección y puerto específico, por lo que recurrimos en técnicas como archivos de configuración que nos permite guardar la dirección de cada servicio por ambiente, de esta forma, tenemos un archivo con la propiedades de desarrollo, de calidad y producción, lo cual es una estrategia efectiva si trabajamos con un hardware específico, es decir, tenemos el control del servidor sobre el cual desplegamos nuestros componentes. Por ejemplo:

Desarrollo:

```
1. #config.dev.properties
2. service.salesorders.url=localhost
3. service.salesorders.port=8080
```

Calidad:

```
1. #config.test.properties
2. service.salesorders.url=http://test.myorders.com
3. service.salesorders.port=80
```

Producción:

```
1. #config.prod.properties
2. service.salesorders.url=https://myorders.com
```

3. `service.salesorders.port=443`

Pero que pasa en arquitecturas Cloud, donde cada componente puede cambiar dinámicamente de IP o puerto, ya sea por fallas o por agregar o quitar instancias según la demanda de nuestra aplicación (Nube elástica)



Nuevo concepto: Elasticidad

En arquitecturas Cloud, la elasticidad es la capacidad de la nube para crecer o disminuir en recursos a medida que la demanda aumenta o disminuye, por lo que es posible aprovisionar nuevos servidores o ampliar la capacidad de procesamiento de los existentes a demanda, sin necesidad la intervención humana.

Entonces, si la nube es elástica, implica que nuevas instancias de un servicio se den de alta o las existentes se apaguen, además, cuando la nube se estira, crea nuevas instancias en IP random, lo cual hace que un archivo de configuración fijo se vuelva obsoleto, ya que tendríamos que tener a una persona que este actualizando ese archivo cada vez que una instancia se de alta o de baja, lo cual es obviamente ineficiente y altamente propenso a errores.

Para comprender mejor este problema analicemos la siguiente imagen:

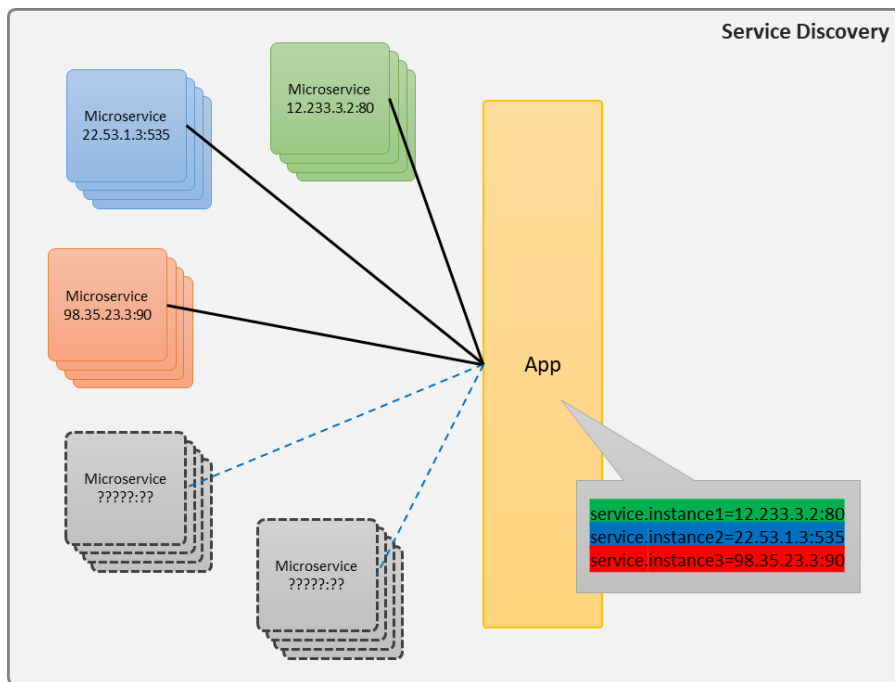


Fig 238 - Problemática de no usar Service Discovery.

En la imagen anterior podemos observar el problema que describimos hace un momento, es decir, una aplicación que tiene un archivo de configuración para ubicar la dirección física de cada instancia, sin embargo, hablábamos de la elasticidad de la nube, que permitía que nuevas instancias se aprovisionara, lo que provocaría que estas nuevas instancias fueran desconocidas por la aplicación hasta que alguien manualmente agregara estas nuevas instancias al archivo de configuración.

Otro problema que también se puede dar es que una instancia deje de responder o simplemente se elimina como parte de la elasticidad de la nube, lo que dejaría a la aplicación apuntando a una instancia del servicio que no está en capacidades de responder:

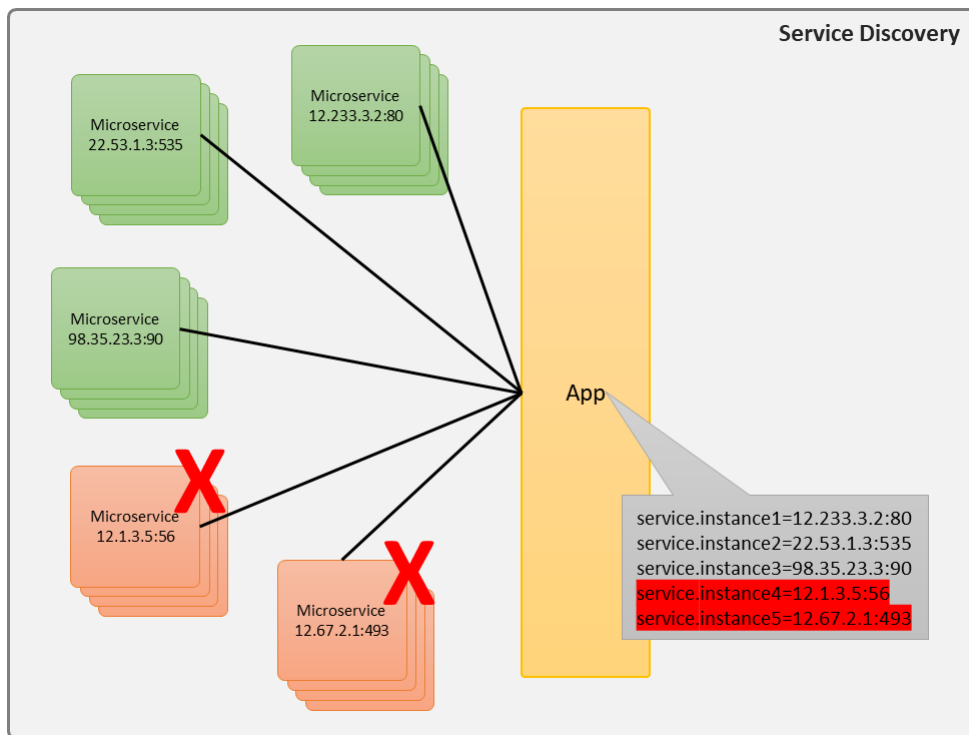


Fig 239 - Contracción de la nube.

Como puedes ver en la imagen, dos instancias del servicio han dejado de responder o simplemente se han dado de baja, dando como resultado que la aplicación nuevamente esté desactualizada, e incluso termine fallando, por intentar ejecutar servicios que no estén disponibles.

Estos dos ejemplos que te acabo de mostrar ocurren mucho en la vida real, sobre todo en temporadas de ventas altas, como el Black Friday, Navidad, o días festivos como día de la madre o incluso, lanzamientos de nuevos productos, por lo tanto, cualquiera de las configuraciones anteriores garantizará la operatividad con demanda fluctuante.

Solución

Para solucionar el problema de la ubicación de los servicios, es indispensable desacoplar la configuración de la ubicación de los servicios a un componente externo llamado *Service Discovery*, el cual es el encargado de determinar todas las instancias activas de los servicios por medio de un Registro central, el cual es nada menos que el *Service Registry* que analizamos en la sección pasado.

El *Service Discovery* es un componente que se encarga de recuperar del *Service Registry* todas las instancias de los servicios disponibles y realizar el balance de cargas, sin embargo, existen dos formas en la que este descubrimiento se pueda dar, del lado del cliente y del lado del servidor.

Descubrimiento del lado del cliente

En el descubrimiento del lado del cliente, es el cliente el encargado de consultar al registro de los servicios disponibles, para posteriormente, el mismo realizar el balanceo entre las instancias disponibles:

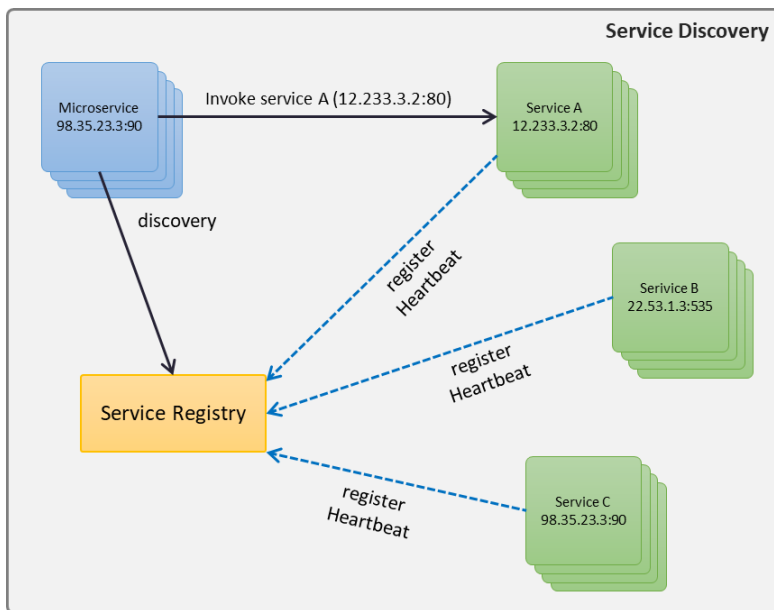


Fig 240 - Descubrimiento del lado del cliente.

Para comprender la imagen anterior, es necesario listar como se dan los eventos. Lo primero que sucede es que los todos los servicios (lado derecho) se registran con el *Service Registry*, después del registro, estos continuaran mandando los latidos (*heartbeat*) para que el *Service Registry* los detecte como activos. En un segundo momento, el cliente (azul) requiere consumir el *Service A*, sin embargo, este no conoce la ubicación física del servicio, por lo que realiza una consulta al *Service Registry* para que le informe de todas las instancias disponibles (a este paso le conocemos como descubrimiento de servicios). Finalmente, el cliente tiene la dirección de todas las instancias disponibles, por lo que hace un balanceo de cargas de forma local y determina a que instancia realiza la invocación.

Un detalle a tomar en cuenta es que el cliente no requiere consultar al Registro la ubicación de los servicios cada vez que requiere hacer una llamada, en su lugar, el cliente puede guardar en cache la ubicación de todas las instancias disponibles, evitando tener que ir al registro con cada llamada, sin embargo, ese cache es válido solo por un corto tiempo, ya que después de un tiempo configurable, este consulta nuevamente al registro, para de esta forma tener actualizado el registro de servicios.

La principal ventaja de tener el descubrimiento del lado del cliente es que el cliente puede utilizar el algoritmo de balanceo que mejor se ajuste a sus necesidades, sin

tener que depender de alguien que lo haga por él, por otro lado, ya que el cliente tiene en cache todas las instancias disponibles, podría seguir operando por un tiempo si se cae el Service Register, ya que no existe un solo punto de fallo, sin embargo, también tiene sus desventajas, y es que es necesario desarrollar la lógica de descubrimiento de servicios directamente en el cliente, por lo que será necesario desarrollar el descubrimiento para cada tecnología o framework específico.

Descubrimiento del lado del servidor

Como su nombre lo indica, en esta variación, el descubrimiento de los servicios se hace del lado del servidor, ocultando al cliente esta capacidad, de esta forma, el cliente se comunica a una única dirección la cual internamente hace el descubrimiento y el balanceo de cargas.

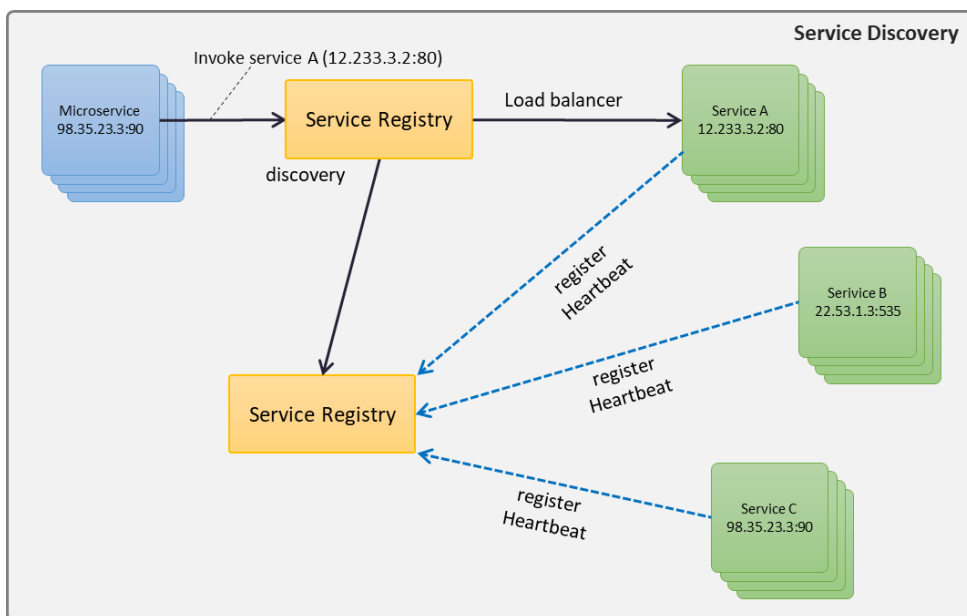


Fig 241 - Descubrimiento del lado del servidor.

Quiero que observes que en esta arquitectura interviene un elemento adicional, que es el balanceador de cargas, pues este permite, en primer lugar, proporcionar una dirección única, la cual pueda consumir el cliente, por otro lado, balancea la carga. Para lograr el balance, es el balanceador de cargas el que tiene que comunicarse con el *Service Registry*, para de esta forma conocer todas las instancias activas de un servicio.

La principal ventaja de esta variante es que el cliente no tiene que implementar la lógica para el descubrimiento de servicios ni la del balanceo de cargas, pues el solo ejecuta una sola URL y se olvida del resto. Sin embargo, tiene como desventajas que hay que implementar un balanceador de cargas, lo que hace que tengamos un elemento más que administrar y que creamos un único punto de falla, de esta forma, si se cae el balanceador de cargas perdemos toda comunicación con los servicios.

Service Discovery en el mundo real

Una de las capacidades de Eureka es que tiene dos componentes, el Eureka server que es realmente el *Service Registry* y el Eureka Client que tiene dos funcionalidades, por una parte permite el registro con el *Service Registry*, pero otro lado, funciona como un *Service Discovery* del lado del cliente, de esta forma, cuando un Microservicio se enciende por primera vez, este se conecta al *Service Registry* y al mismo tiempo recupera la información de todas las instancias registradas para almacenarlas localmente en cache, de esta forma, cuando hacemos la llamada a un servicio, este lo busca en el cache interno y realizar el balanceo de cargas allí mismo.

Ya en la sección pasada hablábamos de cómo es que los Microservicios se registraban al Registro, incluso, hablamos que como parte del registro y los latidos se realizaba para la sincronización de las instancias disponibles en cache por parte del cliente, sin embargo, solo queda un punto por analizar, y es como utilizamos el Service Discovery para hacer las llamadas a los Microservicios, por lo en esta sección nos centraremos en analizar eso.

Lo primero que debemos de saber es que con Eureka tenemos el descubrimiento del lado del cliente, por lo tanto, cada instancia tiene en cache el registro de todas las demás instancias y este registro se actualiza con cada latido (heartbeat), con esto en claro, debemos saber que cada instancia tiene una dirección única (IP y puerto) por lo que no podemos hacer la llamada a una dirección de red concreta, ya que estaríamos brincándonos el balanceo de carga, es por ello que Eureka nos permite hacer la llamada a los Microservicios por medio de su nombre y no su dirección.

Me explico, cuando un servicio se registra en Eureka Server, este le envía su dirección (IP y puerto) y su nombre, y es por medio de este nombre que se realiza el balanceo de cargas. Veamos esto en Eureka server:

The screenshot shows the Spring Eureka web interface in a browser. The browser tabs include 'Eureka', 'CRM', and 'React App'. The address bar shows 'localhost:8761'. The Eureka logo and 'HOME' link are at the top. The 'LAST 1000 SINCE STARTUP' link is also visible.

System Status

Environment	test	Current time	2019-11-23T11:02:37 -0600
Data center	default	Uptime	20:58
		Lease expiration enabled	true
		Renews threshold	11
		Renews (last min)	12

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CRM	n/a (1)	(1)	UP (1) - 192.168.15.5:crm:811
FTP-PAYMENT-POOLING	n/a (1)	(1)	UP (1) - 192.168.15.5:ftp-payment-pooling:8082
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.15.5:gateway:8080
MAIL-SENDER	n/a (1)	(1)	UP (1) - 192.168.15.5:mail-sender:8081
SECURITY	n/a (1)	(1)	UP (1) - 192.168.15.5:security:9090
WEBHOOK	n/a (1)	(1)	UP (1) - 192.168.15.5:webhook:9091

Fig 242 - Analizando el nombre de los Microservicios.

Quiero que observes en la imagen anterior, como cada instancia tiene un nombre, dicho nombre es el que definimos en el archivo *application.yml* de cada Microservicio. Pues bien, este nombre es utilizado para hacer las llamadas a los servicios.

Un ejemplo claro de esto que estoy diciendo está en la clase *OrderService* del Microservicio *crm-api*:

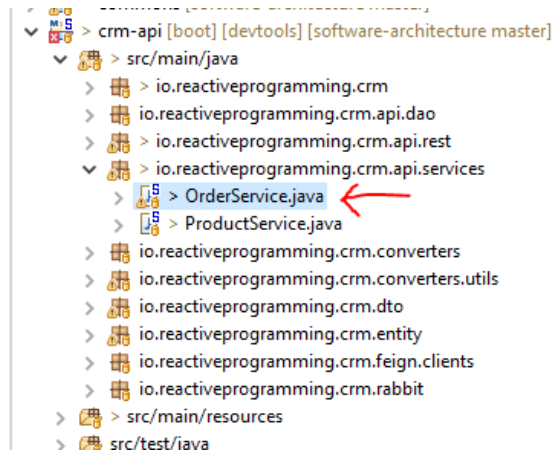


Fig 243 - Clase OrderService.

Dentro de esta clase tenemos el método *createOrder* que ya habíamos analizado en repetidas ocasiones, este método es el que procesa las compras, es decir, cuando le damos pagar desde la aplicación web, este es el método que ejecutamos. Si recuerdas, este método lanza una notificación mediante un *Webhook*, me refiero a las siguientes líneas:

```
1. public SaleOrderDTO createOrder(NewOrderDTO order)
2.     throws ValidateServiceException, GenericServiceException {
3.     logger.info("New order request ==>");
4.     try {
5.         // líneas suprimidas
6.
7.         // Send Webhook notification
8.         try {
9.             if(paymentMethod==PaymentMethod.CREDIT_CARD) {
10.                 MessageDTO message = new MessageDTO();
11.                 message.setEventType(MessageDTO.EventType.NEW_SALES);
12.                 message.setMessage(returnOrder);
13.                 WrapperResponse response = restTemplate.postForObject(
14.                     "http://webhook/push", message, WrapperResponse.class);
15.             }
16.         } catch (Exception e) {
17.             logger.error(e.getMessage(),e);
18.             tracer.currentSpan().tag("webhook.fail", e.getMessage());
19.             System.out.println("No webhook service instance up!");
20.         }
21.         return returnOrder;
22.     } catch(ValidateServiceException e) {
```

```
23.         e.printStackTrace();
24.         throw e;
25.     } catch (Exception e) {
26.         e.printStackTrace();
27.         throw new GenericServiceException(e.getMessage(),e);
28.     }
29. }
```

En las líneas 13 y 14 podrás observar cómo hacemos una llamada a la URL “<http://webhook.push>” la cual obviamente no es una URL válida, sin embargo, esta no es un URL a un servicio concreto, sino más bien es un URL especial que Eureka pueda descomponer para hacer el descubrimiento y el balanceo de cargas. La primera sección (en rojo) es simplemente el protocolo de comunicación, por lo que no tiene nada de especial, la segunda sección corresponde al nombre del Microservicio, en este caso al Microservicio *webhook-notify*. Si no vamos al archivo *application.yml* de este servicio podremos corroborar esta información:

```
1. server:
2.   port: 9091
3. spring:
4.   application:
5.     name: webhook
```

Finalmente, la tercera sección corresponde al “*path*” del servicio a ejecutar, dicho path deberá ser un servicio válido dentro del Microservicio *webhook-notity*, el cual puedes ver dentro de la case *WebhookREST*.

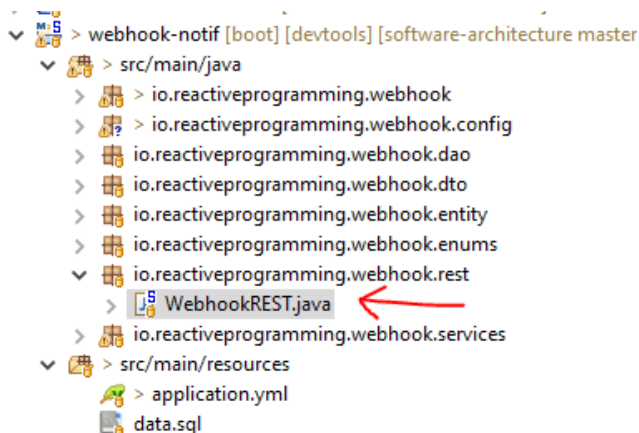


Fig 244 - Clase WebhookREST.

```

1. @PostMapping(path="push")
2. public WrapperResponse<Void> pushMessage(@RequestBody MessageDTO message) {
3.     try {
4.         eventListenerService.pushMessage(message);
5.         return new WrapperResponse<>(true, "Message sent successfully");
6.     } catch (ValidateServiceException e) {
7.         return new WrapperResponse<Void>(false, e.getMessage());
8.     } catch (GenericServiceException e) {
9.         e.printStackTrace();
10.        return new WrapperResponse<Void>(false, "Internal server error");
11.    }
12. }

```

Observa como el método *pushMessage* está respondiendo en el path "*push*" por lo que la URL que analizamos hace un momento está correctamente ligada a este servicio.

Finalmente, lo que hará Eureka con esta URL será, tomar el nombre del Microservicio y buscar todas las instancias disponibles, luego se seleccionará la instancia más adecuada mediante un balanceo de cargas (Round Robin), y finalmente se hará la llamada al servicio.

Observa que en ningún momento hacemos referencia una instancia concreta del microservicio, pues hacerlo implicaría que nos estamos atando a una dirección concreta, que es justo lo que intentamos evitar con el patrón *Service Discovery*.

Conclusiones

Como hemos podido constatar, utilizar una estrategia de auto descubrimiento de servicios nos permite crear aplicaciones mucho más escalables, pues no dependemos de un servidor específico ni la IP o puerto que este nos asigne, sino todo lo contrario, ahora, podemos desplegar donde sea y como sea, al final, todos los servicios podrán ser localizados gracias al *Service Discovery* que por medio de *Service Registry* es posible ubicar y balancear la carga entre todas las instancias disponibles.

Al igual que el *Service Registry*, el *Service Discovery* es uno de los patrones más importantes a la hora de implementar arquitecturas de Microservicios o nativas para la nube, porque nos permite primero que nada desacoplarnos de los servidores físicos y, en segundo lugar, porque nos permite escalar rápidamente con tan solo levantar nuevas instancias.

API Gateway

La gran mayoría de las aplicaciones modernas, exponen recursos a internet con la intención de que otras aplicaciones o usuarios puedan accederlos, sin embargo, mantener una línea clara entre los recursos públicos y privados es una tarea complicada, pero, sobre todo, es clave para la seguridad, por lo que mantener una certeza clave de que recursos son expuestos a internet es fundamental.

Problemática

Uno de los problemas más olvidados al momento de construir Microservicios es, la forma de exponer estos servicios a los clientes, pues a pesar de que una arquitectura de Microservicios es muy buena en el Backend, puede tener varios inconvenientes para los clientes o consumidores de estos servicios, pues cada Microservicio proporciona una pequeña cantidad de servicios, lo que requiere que una aplicación requiere de varios Microservicios para brindar una funcionalidad completa al usuario, lo que hace que cada aplicación requiere conocer la ubicación de cada Microservicio y los servicios que expone cada uno, lo cual genera una experiencia muy mala para los clientes.

Otro de los problemas que se presenta con frecuencia es la seguridad, pues no podemos exponer todos nuestros servicios a internet, por lo que debemos de controlar la forma en que los clientes los consumen y la forma en lo que hacen, por lo que dar un acceso directo a un Microservicio implica dar acceso a todos sus recursos o nos obliga a implementar la seguridad en cada método de cada Microservicio lo cual es una tarea titánica.

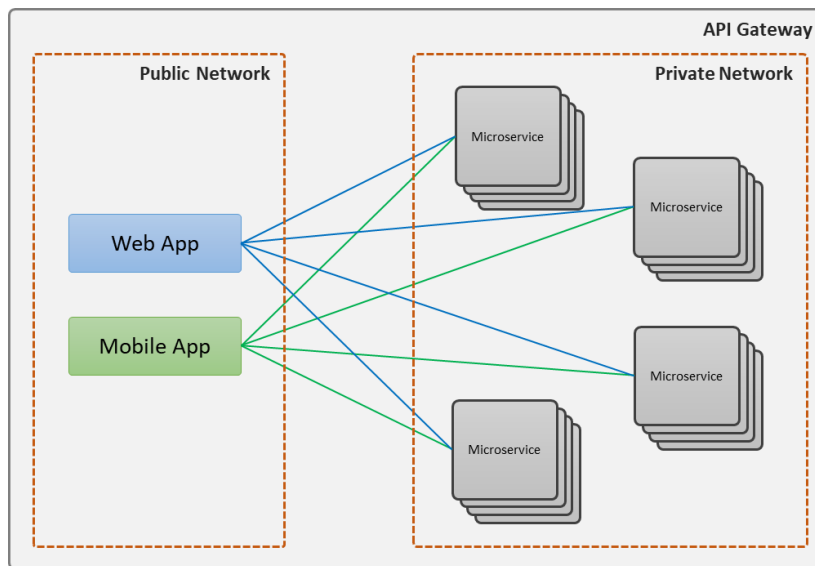


Fig 245 - Comunicación directa a los Microservicios.

Finalmente, cada cliente requiere de una experiencia diferente, por lo tanto, habrá clientes que requieran los datos en formatos diferentes o con políticas de acceso diferentes, por lo que exponer los Microservicios tal cual están desarrollados puede ser un problema para algunos clientes, por lo que es indispensable proporcionar una capa que les permite tener una mejor experiencia.

En la imagen anterior se puede apreciar más claro el problema que mencionaba antes, donde cada aplicación tiene que conocer la ubicación física de cada Microservicio, además, si el servicio se expone sin más, puede tener graves problemas de seguridad, pues estamos exponiendo todos los recursos del Microservicio a internet, y aun cuando estos recursos tenga seguridad, siempre corremos el riesgo de que alguien logre acceder, entonces, si hay recursos que no son necesario exponer, ¿para qué exponerlos?, lo mejor será mantenerlos privado.

Respecto al problema de la ubicación de los servicios, seguramente estás pensando, pues fácil, utiliza un *Service Discovery*, sin embargo, el *Service Registry* es un componente privado de nuestra arquitectura, por lo que no podemos darle acceso a quien sea, por lo tanto, el *Service Registry* debería quedar encerrado en nuestra red privada:

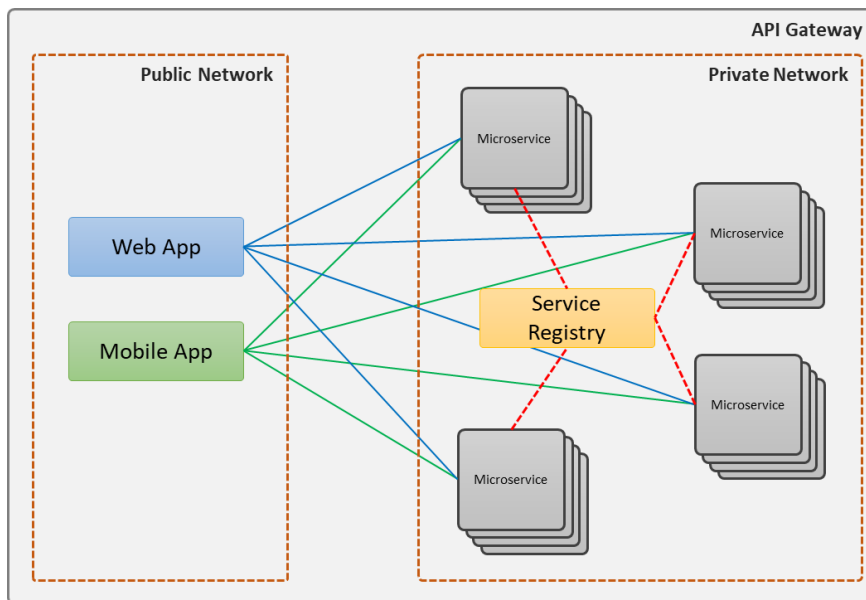


Fig 246 - Como encaja el Service Registry en la arquitectura.

Entonces, si el *Service Registry* es un recurso privado, ya no podemos utilizar el *Service Discovery*, por lo que nuevamente regresamos al problema inicial, la de saber dónde están todos los servicios. Entonces podemos pensar, bueno, entonces pongo un balanceador de cargas:

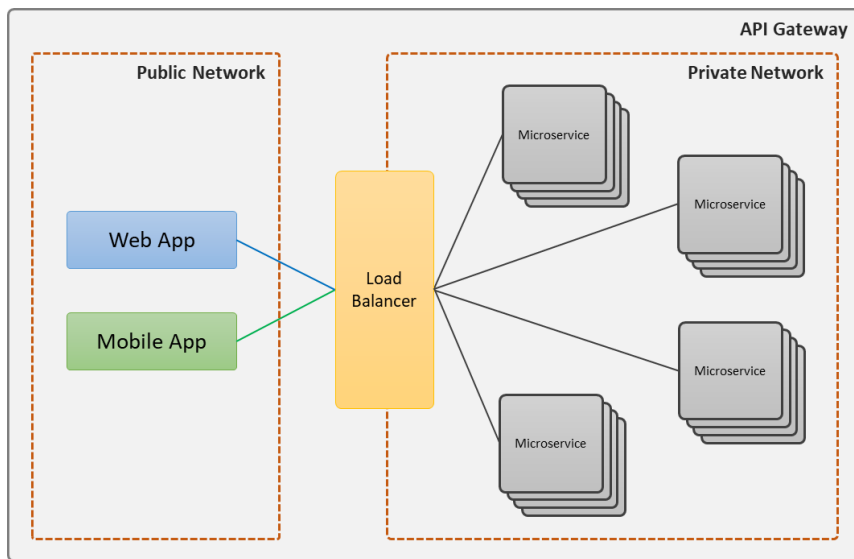


Fig 247 - Implementando un balanceador de cargas.

Esta estrategia puede resultar un poco mejor, ya que evitamos exponer los servicios directamente a Internet, pues los podemos ocultar de la red y hacer que toda la comunicación pase por balanceador de cargas, pero nuevamente hay un problema, el balanceador puede contralar hasta cierto punto el acceso, como restringir el acceso a ciertos recursos, pero ¿qué pasa con la autenticación? Los servicios expuestos seguirán requiriendo de autenticación, por lo tanto, tenemos que implementar la seguridad en cada método de nuestros Microservicios, o en otro caso cualquiera podría acceder, pero implementar la seguridad en cada método de cada Microservicio es una tarea inmensa.

Y finalmente, con un balanceador de cargas no podemos personalizar la experiencia de cada cliente, ya que todos los clientes tendrán los mismos recursos, con los mismos mecanismos de seguridad y los datos en el mismo formato, pues el balanceador solo balancea la carga.

Solución

Para solucionar estos problemas, podemos utilizar un API Gateway, el cual, y como su nombre lo dice, es una compuerta para nuestra API, desde la cual podemos exponer nuestros Microservicios personalizando la experiencia para cada cliente, de esta forma, podemos indicar que servicios exponer, el formato de los datos retornados, e incluso, controlar el acceso, además, nos permite proporcionar un único punto de acceso para toda el API, lo que podemos aprovechar para agregar la seguridad que sea necesaria.

En pocas palabras, **el API Gateway es la cara que damos a los clientes**, y es la forma en que los clientes externos se comunicarán con nosotros por lo que es común que el API Gateway ofrezca servicios simples y de alto nivel que oculten la complejidad de nuestra arquitectura.

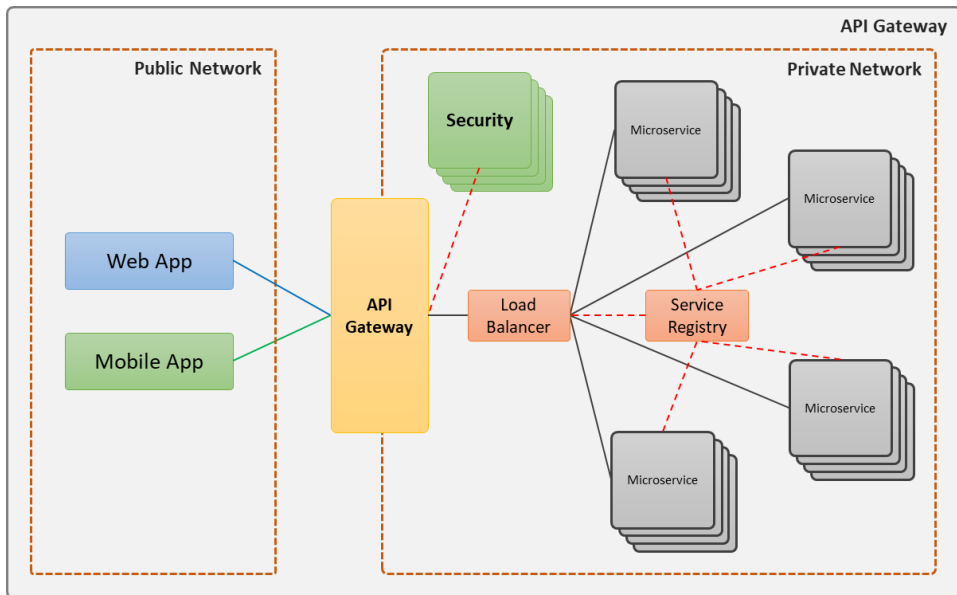


Fig 248 - Implementando un API Gateway.

Este enfoque es especialmente útil cuando los clientes son aplicaciones de terceros, los cuales no necesitan saber los detalles de cómo están compuestos nuestros servicios, como están divididos y los Endpoints internos de nuestros Microservicios.

En la imagen anterior podemos ver cómo los clientes solo se comunican con el API Gateway y no necesitan preocuparse por si existen múltiples instancias o si tenemos un balanceador, o incluso, el API está compuesta de múltiples Microservicios, porque al final, el cliente solo accede a todos los servicios por una URL base.

Múltiples API Gateway para múltiples clientes

Lo más común cuando construimos API's es que expongamos una sola cara para nuestros clientes, algo muy parecido a la imagen anterior, sin embargo, existen situaciones donde debemos exponer diferentes rostros a diferentes clientes, de esta forma, puede que cada cliente requiere servicios diferentes, con seguridad diferente o incluso, que los datos sean servidos en formatos diferentes, por lo tanto, podemos crear múltiples API Gateway con servicios y configuración diferente:

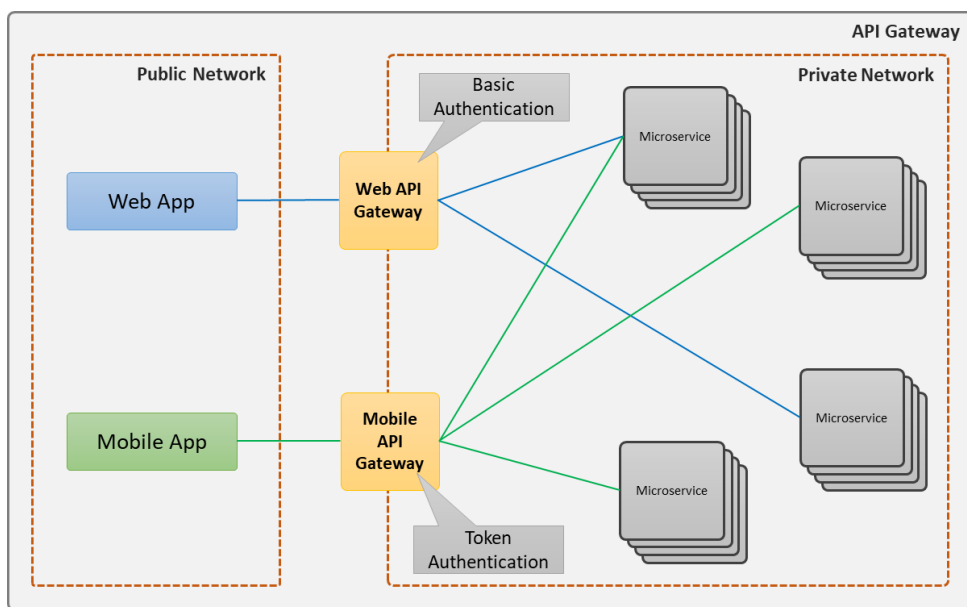


Fig 249 - Múltiples API Gateway.

Observa la imagen anterior, tenemos dos API Gateway, pero cada uno de estos accede a diferentes Microservicios, lo que hace que podamos ofrecer una

experiencia de uso diferente a cada cliente. Con esto logrados varias cosas, por una parte, observa que usamos los mismo Microservicios para los dos API Gateway, lo que hace que los podamos reutilizar, por otro lado, solo exponer los servicios que cada aplicación requiere, sin darle acceso a recursos de más, finalmente, podemos configurar políticas de seguridad diferentes para cada API Gateway, de esta forma, podemos crear métodos más fuertes para aplicaciones menos seguras o más simples para aplicaciones más seguras.



Error común

El API Gateway no es lo mismo que un balanceador de cargas, ya que un balanceador tiene como principal objetivo equilibrar la carga entre todos los servidores, mientras que un API Gateway funciona como una compuerta al mundo, donde podemos controlar los accesos a los recursos que exponemos y personalizar la experiencia para cada cliente. Incluso, lo normal es que detrás de un API Gateway existe un balanceador de carga para que internamente se equilibre la carga entre todos los Microservicios, sin embargo, el cliente no es consciente de esto.

Cómo funciona internamente un API Gateway

Para comprender como funciona el API Gateway podríamos compararlo con el Patrón de diseño *Fachada* y *Proxy* al mismo tiempo, por una parte, se comporta como una fachada ya que brinda una interfaz simple que oculta a los clientes la complejidad de nuestra arquitectura, el número real de Microservicios microservicio y su ubicación. Por otro lado, el API Gateway se comporta como un Proxy, ya que permite realizar ciertas acciones previas y posteriores a la llamada a uno de nuestro Microservicios.



Nuevo concepto: Patrón Facade (fachada)

El patrón *Facade* (fachada) tiene la característica de ocultar la complejidad de interactuar con un conjunto de subsistemas proporcionando una interface de alto nivel, la cual se encarga de realizar la comunicación con todos los subsistemas necesarios. La fachada es una buena estrategia cuando requerimos interactuar

con varios subsistemas para realizar una operación concreta ya que no se necesita tener el conocimiento técnico y funcional para saber qué operaciones de cada subsistema tenemos que ejecutar y en qué orden, lo que puede resultar muy complicado cuando los sistemas empiezan a crecer demasiado.



Nuevo concepto: Patrón Proxy

Este es un patrón de diseño que centra su atención en la mediación entre un objeto y otro. Se dice mediación porque este patrón nos permite realizar ciertas acciones antes y después de realizar la acción deseada por el usuario. El *Proxy* se caracteriza por que el cliente ignora totalmente que una mediación se está llevando a cabo debido a que el cliente recibe un objeto idéntico en estructura al esperado, y no es consciente de la implementación tras la interface ejecutada, de esta manera el cliente interactúa con el *Proxy* sin saberlo.

Dadas las dos definiciones anteriores, podemos decir entonces que el *API Gateway* ofrece una interfaz de alto nivel al cliente, pero al mismo tiempo, la hace de *Proxy*, entonces, mediante el API Gateway podemos personalizar el ruteo a nuestros Microservicios y exponer cada servicio en la URL que queramos, y por otro lado, podemos hacer acciones previas y posteriores a la llamada del servicio final, y es allí donde está la magia, ya que podemos, por ejemplo, validar las credenciales previo a permitir la llamada a los servicios, o incluso podemos medir el número de peticiones de un usuario, ya sea para cobrar por el uso o para restringir el acceso en caso de un exceso de llamadas al API.

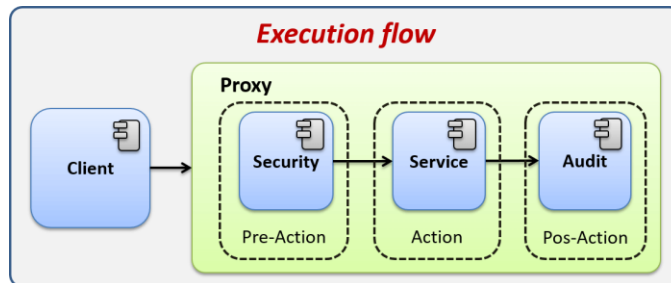


Fig 250 - Cómo funciona un Proxy.

En la imagen anterior vemos como un API Gateway puede realizar ciertas acciones previas a la acción solicitada, y como se ve en este ejemplo, podemos validar las credenciales como una acción previa y luego podemos realizar una auditoría una vez que finaliza el servicio.

API Gateway en el mundo real

Para entender cómo encaja el API en nuestro proyecto, vamos a regresar al diagrama de arquitectura de la aplicación:

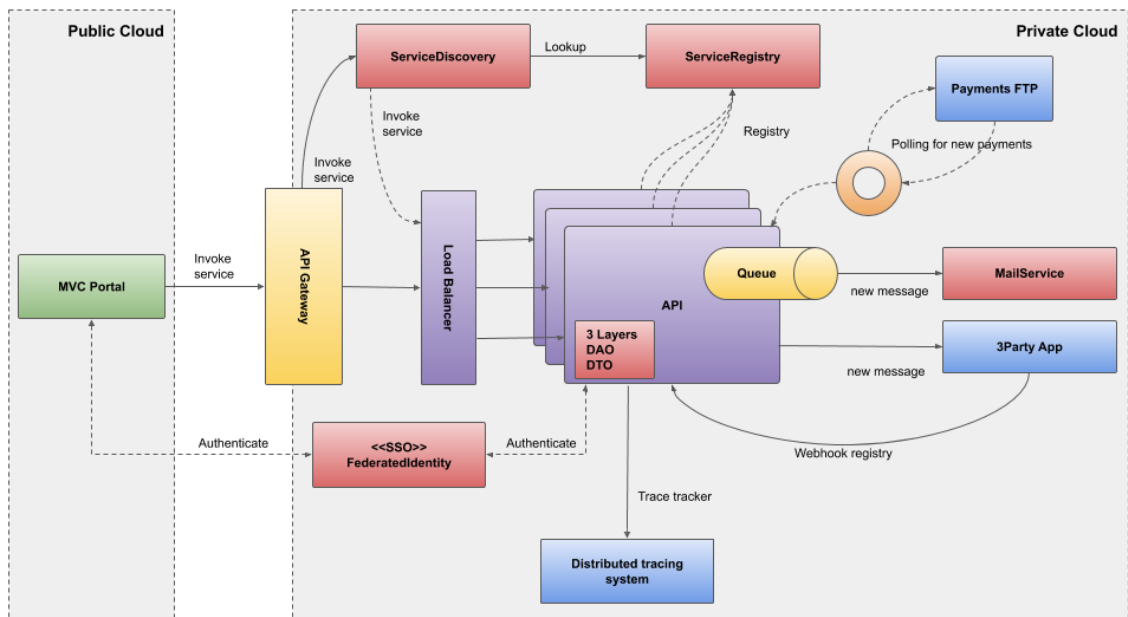


Fig 251 - Diagrama de arquitectura de nuestra API.

Para comprender el juego que cumple nuestro API Gateway es importante notar la separación de las redes, ya que en esto radica la principal ventaja que nos ofrece, del lado izquierdo tenemos la nube pública donde cualquiera puede acceder, es decir, cualquiera con una conexión a Internet podría acceder a la aplicación web del ECommerce, por otro lado, tenemos la nube privada, donde tenemos todos nuestros servicios, servidores, infraestructura, etc. A esta red solo

deberían de poder acceder ciertos miembros de nuestra organización, pues todos los servicios que están allí no están protegidos, lo que quiere decir que cualquiera dentro de esa red podría alcanzar nuestros servidores o ejecutar nuestros servicios directamente sin una capa de seguridad.

Por otra parte, fíjate que en medio de la nube pública y privada tenemos el API Gateway, el cual está colocado estratégicamente en medio de los dos, ya que este si es posible ser accedido desde la nube pública, pero está dentro de nuestra nube privada, es decir, este es el único componente que debería tener salida a Internet.

La importancia de este API Gateway es simple, le ofrece a la aplicación de ECommerce un acceso simple a los servicios, ya que este solo conoce una URL, la del API Gateway y no se tiene que preocupar por la complejidad de la arquitectura interna. Por otro lado, el API Gateway agrega mecanismos de seguridad que evita que cualquier los puede ejecutar, lo que restringe el acceso de dos formas, en primer lugar, solo exponemos a la nube pública los servicios que son realmente necesario exponer, dejando privados el resto, por otro lado, los servicios expuestos pueden ser asegurados con mecanismos de seguridad, como en este caso, que utilizamos una seguridad por Tokens (Analizaremos los tokens más adelante.), lo que hace mucho más fuerte la autenticación que un simple usuario y password.

Cómo funciona el API Gateway

Para implementar nuestro API Gateway utilizaremos Zuul, una tecnología muy simple y poderosa que forma parte del Netflix Open Source Software (Netflix OSS). Mediante *Zuul* podemos implementar un API Gateway completo con apenas una cuantas líneas de código, además, podemos realizar todas las configuraciones mediante el ya conocido archivo *application.yml*.

Por su nombre, creo que debe de quedar lo bastante claro que el API Gateway de nuestro API es el Microservicio *api-gateway*, el cual, si analizamos su estructura, podremos notar lo pequeño que es:

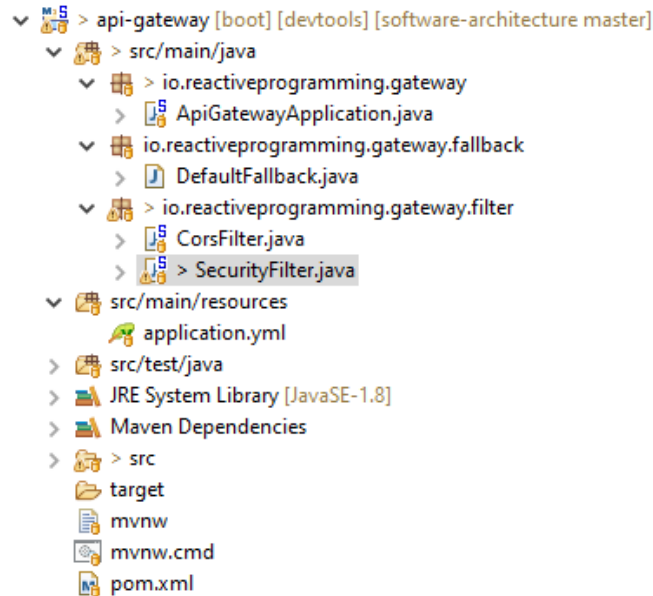


Fig 252 - Estructura del proyecto api-gateway.

Zuul se comienza configurando con el metadato `@EnableZuulProxy` en la clase `ApiGatewayApplication`:

```
1. @EnableZuulProxy
2. @SpringBootApplication
3. @EnableEurekaClient
4. public class ApiGatewayApplication {
5.
6.     public static void main(String[] args) {
7.         SpringApplication.run(ApiGatewayApplication.class, args);
8.     }
9.
10.    @Bean
11.    public SecurityFilter preFilter() {
12.        return new SecurityFilter();
13.    }
14.
15.    @LoadBalanced
16.    @Bean
17.    public RestTemplate restTemplate() {
18.        return new RestTemplate();
19.    }
20. }
```

También podrás notar que lo estamos conectado con Eureka Server, es por eso que vemos la anotación `@EnableEurekaClient`. Esto es muy importante que por *Zuul* utiliza Eureka para balancear la carga entre los Microservicios, es por esta misma razón que tenemos las líneas 15 a 19, donde creamos una instancia de *RestTemplate*, una clase utilizada en *Spring Boot* para hacer llamadas a servicios Rest. También observa el metadato `@LoadBalanced` que permite hacer ese balanceo.

El siguiente paso es configurar *Zuul* para que redireccione las llamadas a los Microservicios correspondientes, por lo que nos iremos al archivo *application.yml* para ver cómo funciona:

```
1. server:
2.   port: 8080
3.
4. spring:
5.   application:
6.     name: gateway
7.   sleuth:
8.     sampler:
9.       probability: 1.0
10.  zipkin:
11.    base-url: http://localhost:9411/
12.    enabled: true
13.    sender:
14.      type: web
15.    service:
16.      name: api-gateway
17.
18. eureka:
19.   instance:
20.     prefer-ip-address: true
21.   client:
22.     registerWithEureka: true
23.     fetchRegistry: true
24.     serviceUrl:
25.       defaultZone: http://localhost:8761/eureka/
26.
27. zuul:
28.   ignoredServices: '*'
29.   prefix: /api
30.   routes:
31.     crm:
32.       path: /crm/**
33.       serviceId: crm
34.   security:
35.     path: /security/**
36.     serviceId: security
```

```
37.     webhook:
38.         path: /webhook/**
39.         serviceId: webhook
40.     sensitiveHeaders: Cookie,Set-Cookie
41.     host:
42.     socket-timeout-millis: 30000
```

Del archivo anterior, solo nos centraremos en las líneas 27 a 42, que es donde se configura *Zuul*, el resto de líneas son mera configuración de *Spring Boot* y *Eureka* que ya hemos analizado antes.

La primera propiedad que analizaremos es *ignoreServices*, la cual utilizamos para decirle que cualquier URL no definida más abajo deberá ser restringida, con esto, evitamos exponer servicios por error, por lo que solo se expondrá lo que definamos implícitamente.

La propiedad *prefix* define el "Base URL", es decir, la URL base sobre la estarán todos los servicios, de esta forma le decimos que el API Gateway responderá a partir de `"/api"`.

La sección *routes* es la más importante, porque es donde definimos los servicios que expondremos al público. Esta parte de la configuración tiene una estructura variable, ya que debemos de definir un nombre para la ruta y dentro de esta, el *serviceId* que corresponde al nombre del Microservicio con el que está registrado en *Eureka* y que casualmente hemos definido en el archivo *application.yml* de cada Microservicio. Por otra parte, la propiedad *path* define la URL base con que sabrá contra que Microservicio mapear. Se que esta parte es confusa, así que analicemos el caso del Microservicio *crm*:

```
1. routes:
2.     crm:
3.         path: /crm/**
4.         serviceId: crm
```

Del ejemplo anterior, la línea 2 es solo un nombre con el que identificaremos la ruta, pero no tiene por qué corresponder con el nombre de un Microservicio, así que podríamos poner cualquier nombre, respecto a la línea 4, esa si muy importante, ya que corresponde con el nombre del Microservicio al que vamos a redireccionar las peticiones y este debe de coincidir con la propiedad *spring.application.name* del Microservicio en cuestión.

La propiedad `path` es una expresión que *Zuul* evaluará para saber a qué Microservicio redireccionar la petición, de esta forma, cada vez que una petición llegue, *Zuul* la comparará contra todos los `path` definidos y redireccionará la llamada al primer `path` que haga match con la URL ejecutada. Por ejemplo, cuando desde la aplicación web (eCommerce) consultamos los productos, ejecutamos la URL: <http://localhost:8080/api/crm/products>

La primera parte (Azul) de la URL corresponde al host donde está el API Gateway, la cual obviamente está en el localhost y responde el puerto 8080. Si observas el *Boot Dashboard* podrás observar que el *api-gateway* responde en el puerto 8080, lo que quiere decir que nuestra aplicación está consumiendo los servicios a través del API Gateway, y no directamente al Microservicio CRM.

La segunda parte (Verde) de la URL corresponde a la propiedad *prefix* que analizamos hace un momento, lo que establece que todos los servicios expuestos estarán a partir de la URL base */api*.

De aquí en adelante, la tercera (Amarillo) y cuarta parte (Rojo) de la URL será evaluada contra todos los `path`, y si hace match, entonces será ejecutado el servicio definido en la propiedad *serviceId*, por lo tanto */crm/products* hace match con la expresión */crm/***, donde **** indica que cualquier URL que venga después de */crm* hace match, por lo tanto */crm/products* hace match con este `path` y la petición es redireccionada al Microservicio *crm*.

Esto quiere decir que cuando ejecutamos la URL <http://localhost:8080/api/crm/products>, el API Gateway hace un Ruteo a <http://localhost:811/products>, la cual corresponde a la URL directa al Microservicio *crm*.

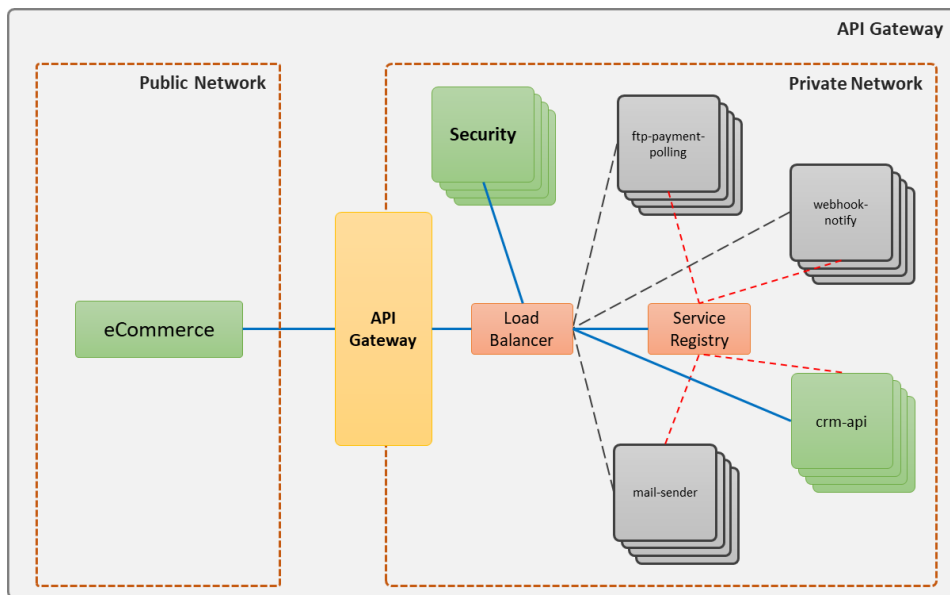


Fig 253 - Traza de ejecución de la consulta de productos

Puede que estemos haciendo una simple consulta al API Gateway, pero internamente esta petición tiene un grado de complejidad bastante grande, pues no solo implica que el API Gateway redirija la llamada al Microservicio *crm-api*, si no que hace muchos pasos adicionales, tal y como vemos en la imagen anterior. Con líneas Azules podrás observar todos los pasos que se tiene que hacer para hacer una simple consulta a los productos.

Lo primero que hace nuestro API Gateway es registrarse al *Service Registry*, con la intención de obtener información de todas las instancias activas de los Microservicios, seguido, tiene que habilitar un balanceador de cargas sobre esas instancias, ya con esto, ya está preparado para recibir peticiones, seguido, cuando llega una petición, valida las credenciales consumiendo al servicio de seguridad, apoyándose claramente del balanceador de cargas, seguido, si el Token es correcto, nuevamente redirige la llamada al Microservicio *crm-api* por medio de un balanceo de cargas. Podemos observar que toda esta complejidad queda oculta para la aplicación (*eCommerce*), pues esta solo ejecuta un URL y listo.

Ahora bien, hagamos una prueba, abramos la URL <http://localhost:811/products> directamente en el navegador a ver qué pasa:

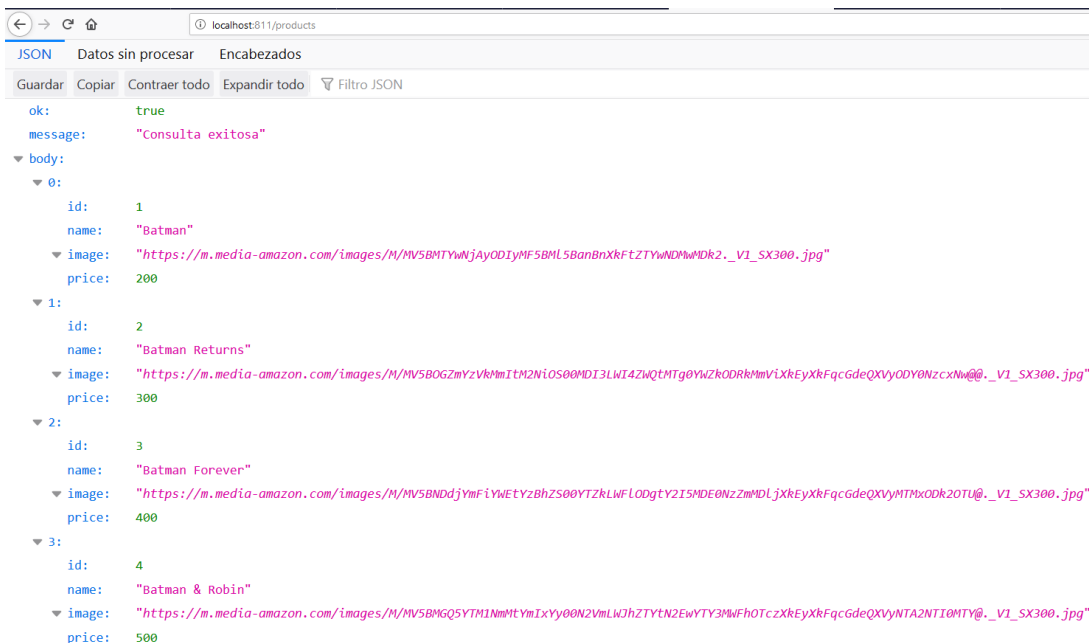


Fig 254 - Consultando los productos directamente desde crm-api.

Esto que estamos viendo es la respuesta directa del Microservicio *crm-api*, pero ¿qué pasaría si ahora consultamos los productos desde el API Gateway? Para comprobar esto, ahora abramos la siguiente URL en el navegador <http://localhost:8080/api/crm/products>.

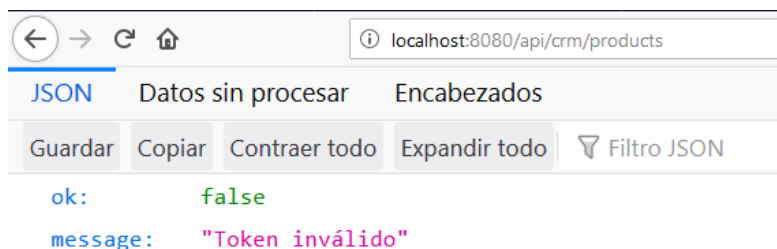


Fig 255 - Consultando los productos por medio del API Gateway.

Observa que el *API Gateway* no nos ha permitido consultar los productos, ya que no nos hemos autenticado, y esto nos lleva al siguiente punto importante del *API*

Gateway, y es que nos permite asegurar los servicios para no exponerlos libremente, y para hacer esto, *Zuul* utiliza Filtros, los cuales son acciones que se pueden ejecutar previas o posteriores a la llamada al servicio. Por lo que, para habilitar la seguridad, es importante crear un Filtro que se ejecute previo a la llamada de los servicios:

```
1. public class SecurityFilter extends ZuulFilter {
2.
3.     @Autowired
4.     private RestTemplate restTemplate;
5.
6.     @Override
7.     public Object run() throws ZuulException {
8.         RequestContext ctx = RequestContext.getCurrentContext();
9.         HttpServletRequest request = ctx.getRequest();
10.
11.         String path = request.getRequestURI();
12.         if("/api/security/login".equals(path)
13.             || "/api/security/login".equals(path)
14.             || "/api/security/sso".equals(path)
15.             || "/api/security/loginForm".equals(path)
16.             || path.startsWith("/api/crm/products/thumbnail")
17.         ) {
18.             //Handle all public resources
19.         }else {
20.             String token = request.getHeader("Authorization");
21.             if (token == null) {
22.                 ctx.setResponseBody("{\"ok\": false,\"message\": \"
23.                     +\"Zuul Unauthorized - Required token\"}");
24.                 ctx.getResponse().setContentType("application/json");
25.                 ctx.setResponseStatusCode(401);
26.             }
27.
28.             Map<String,String> queryParams = new HashMap<>();
29.             queryParams.put("token", token);
30.
31.             class LoginWrapper extends WrapperResponse<LoginResponseDTO>{};
32.             WrapperResponse<LoginResponseDTO> result =
33.                 restTemplate.exchange(
34.                     "http://security/token/validate?token={token}",
35.                     HttpMethod.GET, null,
36.                     new ParameterizedTypeReference
37.                         <WrapperResponse<LoginResponseDTO>>() {},
38.                     queryParams).getBody();
39.
40.             if(!result.isOk()) {
41.                 ctx.setResponseBody("{\"ok\": false,\"message\": \"
42.                     +result.getMessage()+"\"}");
43.                 ctx.getResponse().setContentType("application/json");
44.                 ctx.setResponseStatusCode(401);
45.             }
46.         }
47.     }
48. }
```

```

46.     }
47.     return null;
48. }
49.
50. @Override
51. public boolean shouldFilter() {
52.     return true;
53. }
54.
55. @Override
56. public int filterOrder() {
57.     return 0;
58. }
59.
60. @Override
61. public String filterType() {
62.     return "pre";
63. }
64. }

```

La clase anterior es un Filtro *Zuul*, el cual podemos distinguir porque es una clase que extiende de *ZuulFilter*. Por otra parte, en las líneas 61 a 63 definimos el método *filterType* que regresa el valor “pre”, que será interpretado por *Zuul* como un Filtro que se ejecute previo a la llamada a los Microservicios.

Finalmente, en la línea tenemos el método *run*, que contendrá la lógica del Filtro y que será ejecutado previo a la llamada de los Microservicios. Este método recupera el Token del *header* de la petición (analizaremos los Tokens más adelante), el cual tendrá que ser validado para comprobar la autenticidad del usuario, y para validar el token será necesario hacer una llamada al Microservicio *security*, cómo podemos ver en las líneas 32 a 38.

Si el token no logra ser validado retornamos un código 401 (línea 44), que significa que no tienes los privilegios necesarios para consumir el servicio, y esa es la razón por la cual el API Gateway nos regresa error cuando intentamos consumir los servicios sin un Token.

No quisiera entrar en detalle acerca de cómo se lleva a cabo la autenticación por medio de los tokens, pues son conceptos que vamos a analizar con detalle cuando hablemos de los Tokens y la Identidad Federada, por ahora, imagina que ese token es como un usuario y password que validamos desde el Microservicio *security*.

En las siguientes secciones analizaremos como crear un token para consumir el servicio, por ahora solo nos quedaremos con la idea de que no podemos consumirlo sin un token.

Finalmente me gustaría aclarar un punto sumamente importante, observa que todos los servicios expuestos por nuestros Microservicios no tienen seguridad, por lo que cualquier persona mal intencionada podría ejecutarlos para inyectar o borrar datos de nuestras API, es por ello que debemos asegurarnos de todos estos Microservicios vivan en una red privada, donde solo unos cuantos tengan acceso y de una forma controlable y auditable. Dentro de nuestra red privada tenemos que evitar que cualquiera de nuestros Microservicios sea expuesto a internet, incluso, es deseable que tampoco se puedan comunicar a internet directamente, si no por medio de Proxy inverso (*Reverse Proxy*).



Nuevo concepto: Reverse Proxy

Un proxy inverso es un tipo de servidor proxy que recupera recursos en nombre de un cliente desde uno o más servidores. Estos recursos se devuelven al cliente, apareciendo como si se originaran en el servidor proxy. A diferencia de un proxy directo, que es un intermediario para que sus clientes asociados se comuniquen con cualquier servidor, un proxy inverso es un intermediario para que cualquier cliente se comunique con sus servidores asociados. En otras palabras, un proxy actúa en nombre de los clientes, mientras que un proxy inverso actúa en nombre de los servidores.

— *Wikipedia*

El único Microservicio que debería ser expuesto a internet es el API Gateway, pues el que recibirá las peticiones de los usuarios.

Conclusiones

Ocultar la complejidad de nuestra API a nuestros clientes es super importante, porque reducimos la complejidad para utilizarla, logrando con esto, tener una mejor experiencia en su utilización, pero al mismo tiempo, nos permite controlar la forma y los servicios que exponer al público, creando una barrera clara entre los servicios públicos y privado, obteniendo con esto, claras ventajas de seguridad para el API.

Por otra parte, el usuario solo conocerá un solo punto de acceso, lo que nos permitirá aplicar políticas más fácilmente, como monitoreo de solicitudes, seguridad, auditoría, medir SLA's, etc.

Access token

El control de acceso es algo que está presente en casi cualquier aplicación, pues de una u otra manera debemos de ser capaces de detectar a los usuarios que intentan acceder a los recursos de nuestra aplicación, por lo que las aplicaciones siempre han contado con diversas estrategias para protegerse y negar el acceso a todas aquellas personas no autorizadas.

Lo que es un hecho es que, cada vez es más difícil garantizar la seguridad de las aplicaciones, pues con los años han salido diversas herramientas especializadas para quebrar la seguridad, sumado a esto, cada vez hay más profesionales que se dedican exclusivamente a quebrar la seguridad, por lo que los métodos de autenticación también deben de evolucionar.

Problemática

Durante años nos enseñaron que el usuario y password era lo suficientemente seguro para autenticarnos en una aplicación, sin embargo, esto no es del todo cierto, y es que en la actualidad la autenticación mediante este método es considerado el más vulnerable de todos.

El problema con este método es que tenemos que enviar el usuario y password en cada petición, lo que hace que pueda ser capturado por terceros de muchas formas, desde un simple *Keylogger*, ser capturado durante su viaje por la red o ser recuperado de algún medio persistente del lado del servidor, cómo por ejemplo, que el programador imprima por error el usuario y password en los logs, los guarde sin encriptar en la base de datos o simplemente, quiere hacer un mal uso de la información por lo que guarda los password en algún otro lugar.

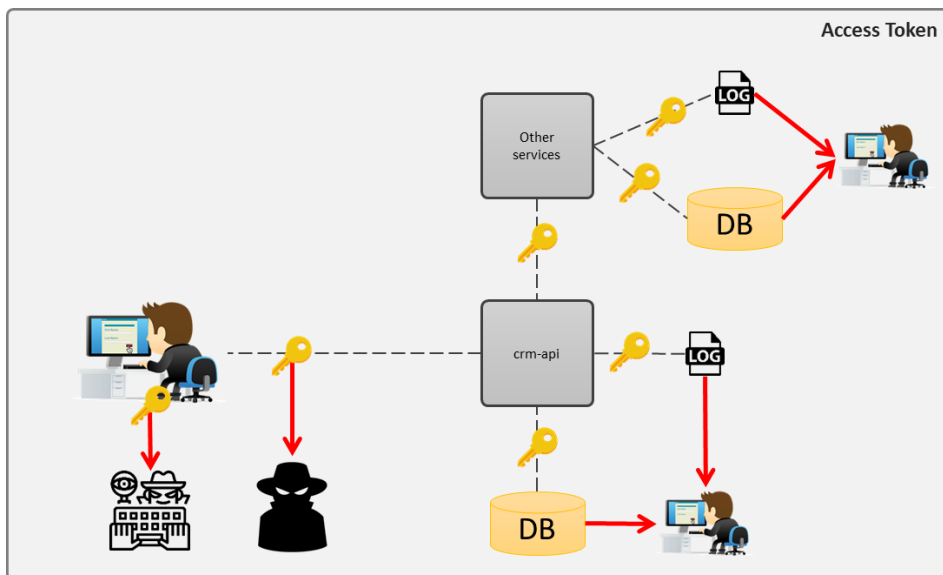


Fig 256 - Puntos de captura del password.

La imagen anterior muestra los puntos en que un password puede ser vulnerado, donde las líneas punteadas representan por donde el password podría viajar, mientras que las líneas rojas representan donde el password podrá ser capturado. En primer lugar, el usuario captura el password desde su equipo, donde un software de *Keylogger* podría guardar el password, luego el password es enviado al servidor, donde podría ser capturado por alguien que este escuchado la red, sobre todo en comunicaciones inseguras. Luego, el password llegar al servidor, donde creemos que el password ya está a salvo, sin embargo, el riesgo de ser vulnerado baja considerablemente, pero siempre existen factores de riesgo, como que el usuario guarde el password en los logs o lo guarde en las base de datos, por lo que eventualmente un usuario del sistema podría revisar el log o la base de datos y bingo, tiene el password, otro caso es que ese mismo password lo replicamos a otro sistema, lo que repite el riesgo anterior.

El problema de enviar los password es precisamente ese, que una vez que el password se envía al servidor no sabemos por cuantas personas o servicios pasará, además, un password siempre será válido hasta que el usuario lo cambie, pero si el usuario no sabe que ya fue vulnerado, porque habría de cambiarlo, por lo que la persona maliciosa tendrá acceso a sus sistemas por un tiempo indeterminado.

Otro problema es que con frecuencia utilizamos el mismo password para muchas de nuestras aplicaciones, como el correo electrónico, redes sociales, equipo de cómputo o incluso otros sistemas de la compañía, por lo que, en muchos de los casos, vulnerar una contraseña implica vulnerar varios de los accesos de ese mismo usuario.

Solución

La solución es clara, dejar de enviar el usuario y password en cada petición al servidor y en su lugar enviar un Token, el cual es una cadena de caracteres sin aparente significado, pero que el servidor puede descifrar y comprobar la autenticidad del usuario. Pero, antes que nada, describamos que es un Token.

Qué es un Token

Un token es una cadena alfanumérica con caracteres aparentemente aleatorios, como el siguiente:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.TTVA950rM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

O el siguiente:

```
3454241234
```

Estas cadenas de texto, pueden no aparentar un significado, sin embargo, tiene un significado real para el servidor o institución que lo emitió, el cual puede entender y así, validar al usuario que intenta acceder a la información, e incluso, puede tener datos adicionales.

Un caso simple de tokens, es el dispositivo que dan los bancos para realizar transacciones desde internet, este token te genera un valor numérico que luego tenemos que ingresar al sistema, y de esta forma, el portal pueda asegurarse de que efectivamente somos nosotros y no un impostor.

- Firma (Amarillo): La tercera parte es la firma digital, esta parte es un Hash generado a partir de la Payload, la cual tiene como objetivo validar la autenticidad del Token.

Ejemplo del Header

```
1. {  
2.   "alg": "HS256",  
3.   "typ": "JWT"  
4. }
```

Ejemplo del payload:

```
1. {  
2.   "id": "1234567890",  
3.   "name": "Oscar Blancarte",  
4.   "rol": "admin"  
5. }
```

La firma por otra parte es el payload y el header en base64 y después encriptado.



Error común

A pesar de que la sección del header y payload se ven como encriptadas, la realidad es que solo pasan por un proceso de codificación en base64, lo que hace que cualquiera puede obtener su contenido, por lo que por ningún motivo debemos almacenar datos sensibles, como el password del usuario, datos de tarjetas de crédito, etc.

Para vivir un poco mejor la experiencia de que es un token JWT, podemos ir a su página oficial <https://jwt.io/> y veremos lo siguiente:

ALGORITHM

HS256

Encoded

PASTE A TOKEN HERE

```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMekKF2QT4fwpMeJf36P0k6yJV_adQssw5c

```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```

{
  "alg": "HS256",
  "typ": "JWT"
}

```

PAYLOAD: DATA

```

{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}

```

Wed Jan 17 2018 19:30:22 GMT-0600 (hora estándar central)

VERIFY SIGNATURE

```

HMACSHA256(
  base64UrlEncode(header) + ".",
  base64UrlEncode(payload),
  your-256-bit-secret
)

```

secret base64 encoded

Fig 258 - Creando un token en línea.

Eso que estamos viendo en pantalla es una herramienta que ofrece JWT para que probemos como funciona un token. Del lado derecho podemos ver las tres secciones del token que te mencionaba, e incluso, las podemos editar, solo recuerda que los valores deberán cumplir con la sintaxis de JSON. Finalmente, del lado izquierdo podrás ver cómo se va formando el token a partir de la información que vamos proporcionando.

Cómo funciona la autenticación con JWT

El primer paso es que el usuario obtenga un Token, para esto, se puede usar un sistema de login tradicional como usuario y password o incluso uno más fuerte, la idea es que de alguna forma el usuario se identifique con la aplicación.

Una vez, que el sistema valida al usuario, este le retorna un token. Una vez generado el token, es el cliente el responsable de guardarlo, pues de aquí en adelante, todas las peticiones que realice al servidor, deberán llevar el token.

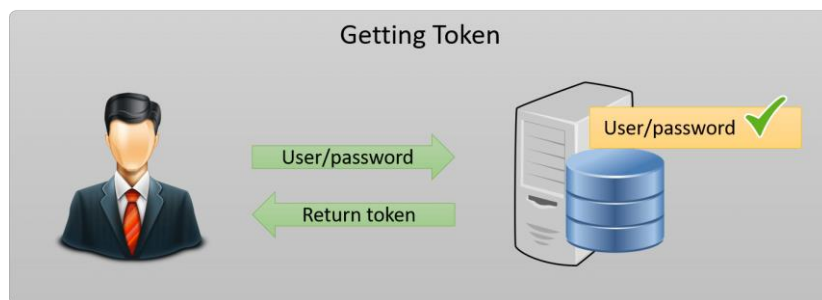
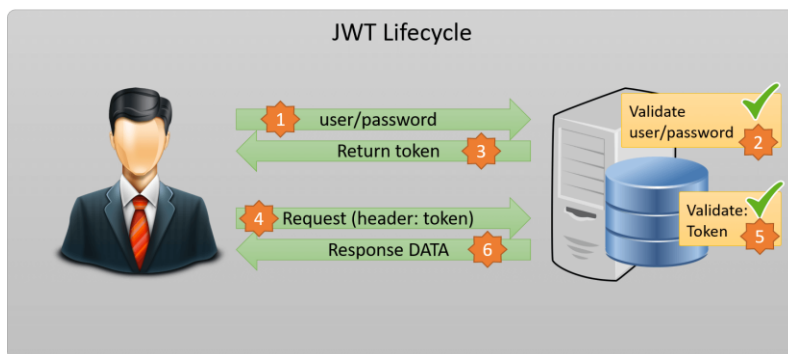


Fig 259 - Obteniendo un Token.

No existe una regla de donde deba de ser guardado el token, ya que puede ser almacenado como una Cookie o en Local Storage si es que es un cliente web, o puede ser en la base de datos, si es que el usuario que consume el API es otra aplicación, la idea es que ese token lo deberemos conservar para futuras invocaciones al API.

Ya hemos hablado de como JWT funciona, pero ahora entraremos a ver cómo es la interacción que tiene un usuario al autenticarse por medio de JWT, para lo cual veamos la siguiente imagen:



Los pasos son los siguientes:

1. El usuario requiere de una autenticación tradicional con el servidor, es decir usuario y password (o cualquier otro tipo de autenticación).
2. El servidor validará que los datos introducidos sean correctos y generará un Token.
3. El servidor enviará el token al usuario y este lo tendrá que almacenar.
4. Una vez con el token, el usuario realiza una petición al servidor, enviando el token previamente generado.
5. El servidor validará que el token sea correcto.
6. Si el token es correcto, entonces el servidor retornará los datos solicitados.

Cabe mencionar que los puntos 4, 5 y 6 se pueden repetir indeterminado número de veces hasta que el token caduque, cuando esto pase, entonces será necesario reiniciar desde el paso 1.

Seguramente en este punto te preguntas, ¿de qué sirve el token si al final necesito enviar el usuario y password para crear el token? Y es una pregunta muy válida, pero existe una diferente enorme, para empezar, solo vamos a capturar el usuarios y password una vez, lo que reduce en gran medida la posibilidad de que nos intercepten las credenciales, en segundo lugar, las credenciales las mandamos solo al servicios que crea el token y no a todos los servicios del API, lo que limita la posibilidades de ser capturado a un solo servicio, por lo que podemos tener una mejor auditoría sobre ese servicio. Finalmente, cuando queramos consumir otro servicio, mandaremos el token y no el usuario y password, por lo que el token podrá validar nuestra identidad, pero no hay forma de saber con qué password se generó ese token.

Cómo se valida un token

Durante el proceso de creación del token, es posible agregar cierta información en la sección del *payload*, cómo el usuario, el rol y una fecha de vigencia, entre otros datos más, estos valores se utilizan para generar una firma, que corresponde a la tercera parte del token que analizamos hace un momento. Esa firma es creada mediante un password que solo el que creo el token conoce, y mediante ese password es posible saber si la firma corresponde con el payload, lo que da una certeza de que el token no ha sido alterado y que el contenido de este es válido.

En aplicaciones monolíticas donde solo tenemos un componente como Backend es normal que el mismo Monolítico valide el token, por lo que la comunicación se vería de la siguiente forma:

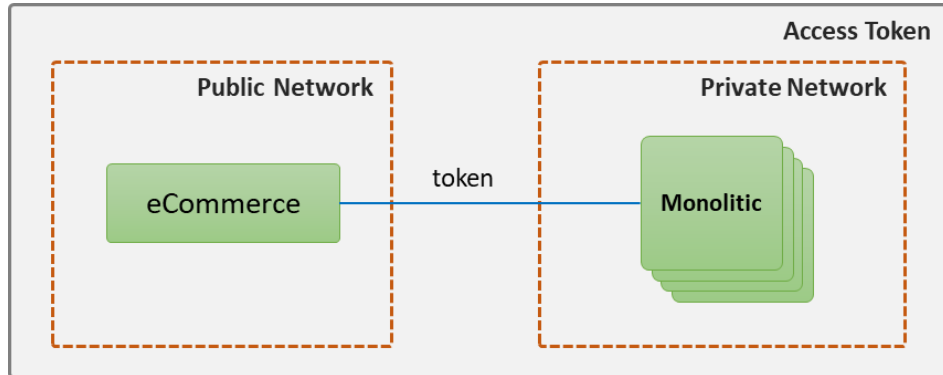


Fig 260 - Validación del token en una aplicación Monolítica.

Puedes observar que esta arquitectura es muy simple, solamente mandamos el token a la aplicación monolítica y como el mismo lo creo, el mismo puede validarlo, lo cual lo hace la forma más simple de hacerlo.

Pero que pasa en arquitecturas distribuidas como la arquitectura SOA, EDA o Microservicios, donde tenemos más de un componente distribuido y que requieren autenticar al usuario, eso se convierte rápidamente en un problema, porque cada componente distribuido tendría que implementar la lógica para validar los tokens, sin embargo, repetir la lógica en cada componente tampoco es lo más sabio del mundo, por lo que la solución obvia es desacoplar esa lógica en un Microservicio independiente que se encargue únicamente de la seguridad, lo que da como resultado una arquitectura como la siguiente:

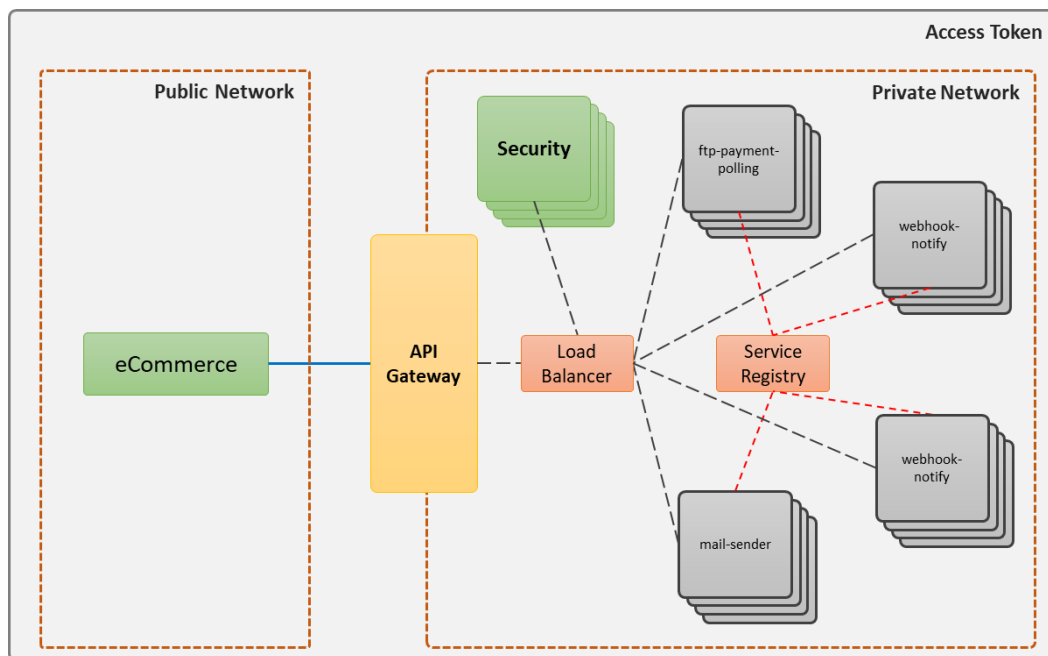


Fig 261 - Delegando la seguridad a un componente externo.

En la imagen podemos apreciar ahora sí, un componente que se encargará de la seguridad, el cual tendrá como responsabilidad almacenar a los usuarios, sus contraseñas y será el único facultado para crear tokens y validarlos, por lo que, de ahora en adelante, cuando recibamos un token, mandaremos a llamar a este servicio para que valide ese token, incluso, podríamos solicitar cierta información del usuario.

Autenticación de aplicaciones

Si bien, lo normal es autenticar usuarios del sistema, existe otros casos donde necesitamos que otras aplicaciones se firmen en el sistema para interactuar con el API, por ejemplo, puede que otros sistemas consulten los productos, o que desde otros sistemas se creen ordenes de compra, etc. Por lo que se puede crear tokens especiales para estas aplicaciones, lo que implicaría que nuestra aplicación debería de tener alguna interfaz o API especial para crear estos tokens, y de esta

forma, el sistema cliente deberá guardar ese token en su base de datos y utilizarlo en cada llamada.

Access Token en el mundo real

Para demostrar cómo utilizar los tokens, hemos implementado la creación de Tokens JWT en nuestra API, de tal forma que cualquier usuario que quiera consumir los servicios deberá contar con un Token, de la misma forma, la aplicación web (eCommerce) también requiere de un token para validar que estamos autenticados en la aplicación.

Para comprobar cómo funcionan los Tokens nos aseguraremos que nuestros Microservicios de API REST están encendidos al igual que la aplicación web (eCommerce), y nos dirigiremos a la aplicación web, si actualmente estás autenticado, será necesario cerrar la sesión para ubicarnos en la página de Login:

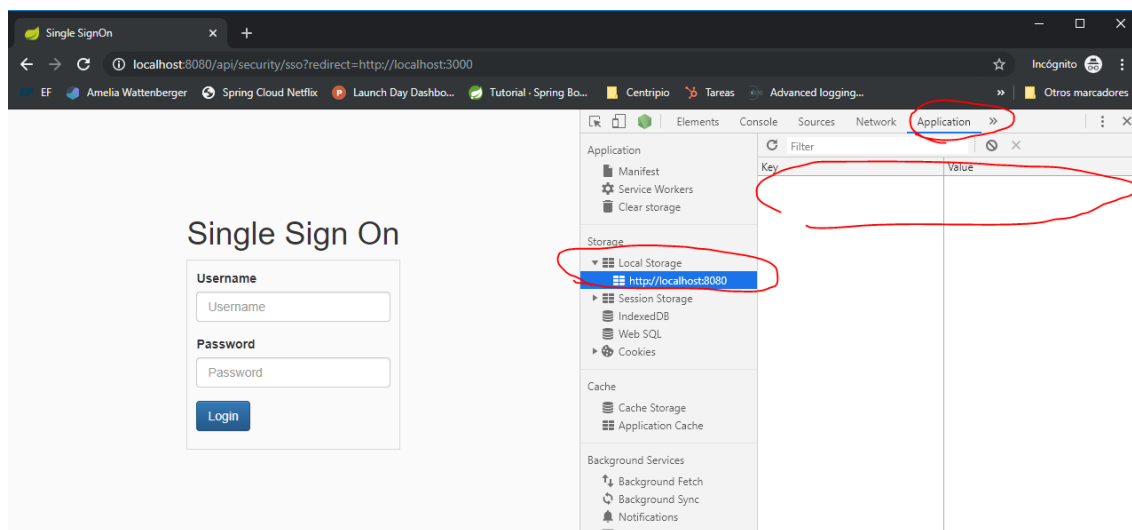


Fig 262 - Login de la aplicación.

Estando en la página de login vamos a dar click derecho sobre la página y luego seleccionaremos la opción "Inspeccionar" o presionamos Ctrl + Mayús + I para abrir la sección de debugger del navegador. Desde allí ubicaremos la sección "Application" y luego expandiremos la sección de "Local Storage", tal cual puedes ver en la imagen anterior. Podrás ver del lado izquierdo una tabla con dos columnas (Key, Value) la cual está vacía. Esta tabla corresponde al Local Storage del navegador, el cual es una característica del navegador para almacenar datos del lado del cliente.

Una vez que hemos comprobado que la tabla está vacía, procederemos a autenticarnos en la aplicación:

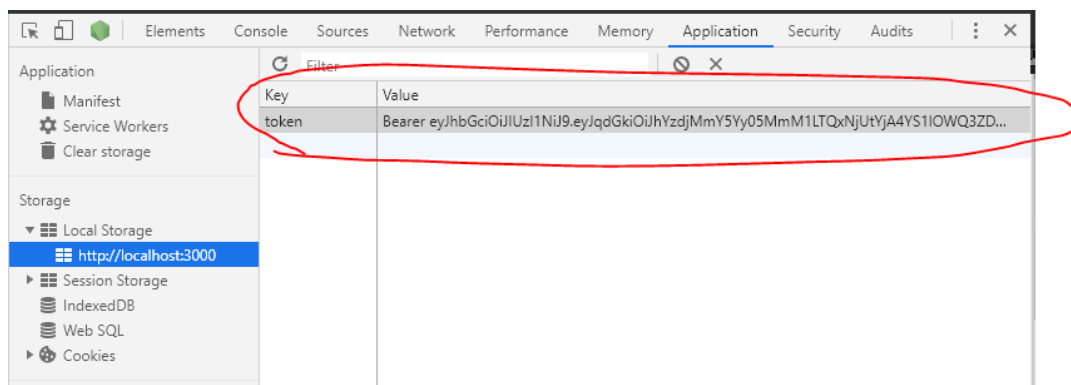


Fig 263 - Comprobando el Local Storage.

Una vez autenticados, la aplicación nos redirigirá nuevamente a la página de inicio de la aplicación, pero, además, podrás ver que el Local Storage se ha actualizado y ahora vemos un nuevo registro llamado Token.

Este token ha sido creado por nuestro API una vez que las credenciales que le hemos enviado han sido validadas, por lo que lo hemos guardado en el Local Storage para usarlo en las siguientes llamadas al API.

Ahora bien, antes de pasar a analizar cómo se creó este token, me gustaría que analizáramos el token como tal, por lo cual vamos a seleccionar el valor que tenemos en la columna Value del Local Storage y luego nos iremos a <https://jwt.io> y pegaremos su contenido del lado izquierdo de la herramienta que nos proporciona, asegurándonos de borrar la primera parte del token "bearer ":

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJhYzdjMmY5Yy05MmM1LTQxNjUyYjA4YS10WQ3ZDV1Mjg3MDQiLCJpYXQiOiE1NzUyMjI0TA5InN1YiI6IiN1Y3VyaXR5U2VydmljZSI6Im9zY2FyIiwiaXhwaWJoxNTc1MzA4OTkwfQ.RyoCAY04Y5-_i4nF44GMKGj0pAsfoNXG52CdSv9uLEo
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256"}
```

PAYLOAD: DATA

```
{  "jti": "ac7cd2f9c-92c5-4165-b08a-e9d7d5e28704",  "iat": 1575222590,  "sub": "SecurityService",  "iss": "oscar",  "exp": 1575308990}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  your-256-bit-secret)
```

☐ secret base64 encoded

Fig 264 - Validando un token JWT.

Quiero que observes como del lado derecho han aparecido todos los datos con el que se creó que Token, podemos ver el header con el algoritmo utilizado (H256) y en el payload podemos apreciar toda la información que utilice para crear el token, en este caso, he agregado las valores *jti* que representa el identificador del token, *iat* que indica la fecha en que fue creado el token, *sub* que indica quien creo el token, *iss* el dueño de este token, y *exp*, que indica la fecha de expiración del token. En este punto el único dato que nos interesa es el campo *iss*, pues le indica al API cual es el usuario que está intentando autenticarse.

En este punto hay algo muy interesante, y es que, si bien la estructura del token es correcta, nos está mostrando los datos con el que fue creado el token, la página nos dice que la firma del token es incorrecta y nos pone en rojo la caja de texto del lado izquierdo.

Si recuerdas, mencionamos que la tercera parte del token es la Firma, es una campo que se calcula tomando como entrada las otras secciones (payload y header), luego es procesada con la llave privada para generar la firma, de tal forma que mediante la firma es posible saber si el contenido del token cambio, entonces, para validar que el token es correcto necesitamos la llave privada, la cual para fines de pruebas es "1234", por lo tanto haremos lo siguiente para validar el token.

Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJhYzdjMmY5Yy05MmM1LTQxNjUtYjA4YS11OWQ3ZDV1Mjg3MDQiLCJpYXQiOiE1NzUyMjI0TAsInN1YiI6IiN1Y3VyaXR5U2VydmljZSI6Im9zY2FyIiwiaXhwaWJoxNTc1MzA4OTkwfQ.RyoCAY84Y5-14nF44GMKGj0pAsfoNXG52CdSv9uLEo

Subject (whom the token refers to)

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

{
 "alg": "HS256"
}

PAYLOAD: DATA

{
 "jti": "ac7c2f9c-92c5-4165-b08a-e9d7d5e28704",
 "iat": 1575222590,
 "sub": "SecurityService",
 "iss": "oscar",
 "exp": 1575300990
}

VERIFY SIGNATURE

HMACHSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 1234
)
☒ secret base64 encoded

SHARE JWT

Signature Verified

Fig 265 - Validando un token JWT.

Primero que nada y muy importante será borrar todo el token del lado izquierdo (es importante borrarlo), luego, en la sección de "Verify signatura" del lado derecho, introduciremos la llave privada, que es "1234" y seleccionaremos el checkbox "secret base64 encoded", finalmente, pegaremos nuevamente el token sin la sección "Bearer ". Como resultado veremos en la parte inferior que el token ha sido validado exitosamente, eso quiere decir que el contenido del token corresponde con la firma digital, lo que también quiere decir que el contenido de ese token es fiable, por lo que si el API recibe ese token puede saber con certeza que el usuario autenticado es "oscar".

Implementando los Tokens en nuestra API

Ya que sabemos bien que es un Token y como lo utilizamos para autenticarnos en la aplicación, pasaremos a explicar el proceso por medio del cual el Token es creado y validado por el API.

Lo primero que debemos de saber es que el Microservicio encargado de la seguridad es *security*, el cual solo tiene una responsabilidad en toda la arquitectura, que es la de emitir los tokens, validarlos y proporcionar información del usuario.

Dicho lo anterior analicemos el proceso por medio del cual un usuario se autentifica:

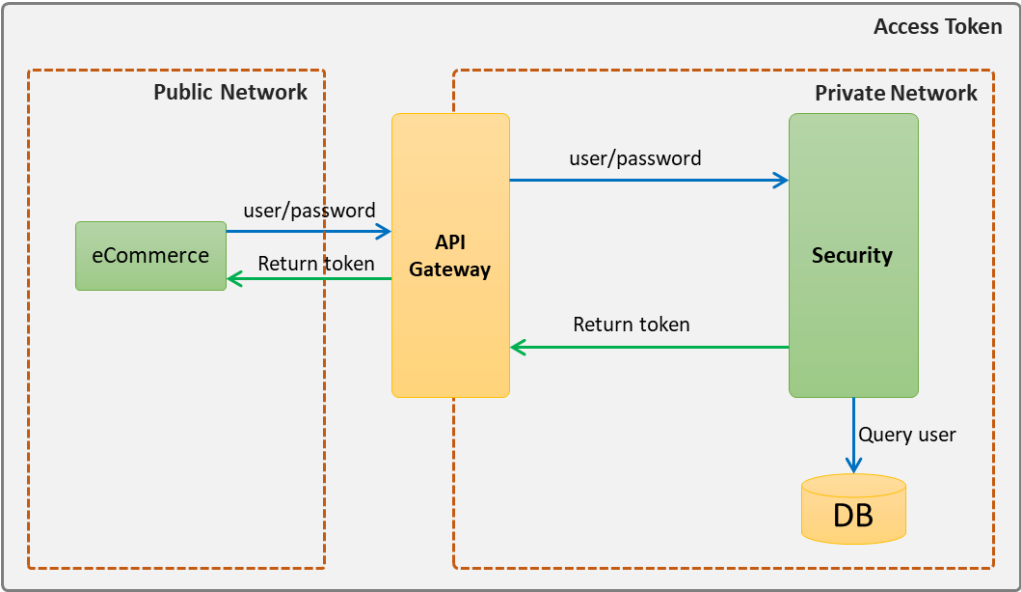


Fig 266 - Proceso de autenticación.

1. El usuario introduce su usuario y password desde el formulario de inicio de sesión.
2. El usuario y password llegan al Microservicio de seguridad (security) por medio del API Gateway.
3. El Microservicio Security valida el usuario y password con los registrados en la base de datos.
4. Si las credenciales son correctas, entonces el Microservicio security crea un Token con una vigencia de un día.
5. El Token generado es retornado al cliente por medio del API Gateway.
6. El cliente recupera el Token y lo almacena en el Local Storage.
7. Fin.

En este punto el usuario tiene el token y lo tiene almacenado en su Local Storage, por lo que en las siguientes peticiones al API ya no necesita enviar nuevamente el usuario/password, y en lugar envía el Token.

Para comprobar esto que te estoy diciendo, vamos a crear un token manualmente, por lo que vamos a necesitar un cliente REST para probar el servicio de Autenticación. Yo recomiendo usar Postman, ya que es uno de los mejores clientes REST y tiene una versión gratuita. Puedes instalar Postman desde <https://www.getpostman.com/>

Una vez instalado Postman, vamos a crear un nuevo request, para esto, nos damos click en el botón "New" en la esquina superior izquierda y seleccionamos la opción "Request". En la siguiente pantalla pones "Authentication" como nombre del request y listo, nos creará el nuevo request.

En el nuevo request tenemos que indicar que es una petición POST a la URL `http://localhost:8080/api/security/login`, luego, nos vamos a la pestaña de "Headers" y agregamos la propiedad "Content-Type=application/json" tal cual puedes ver en la siguiente imagen:

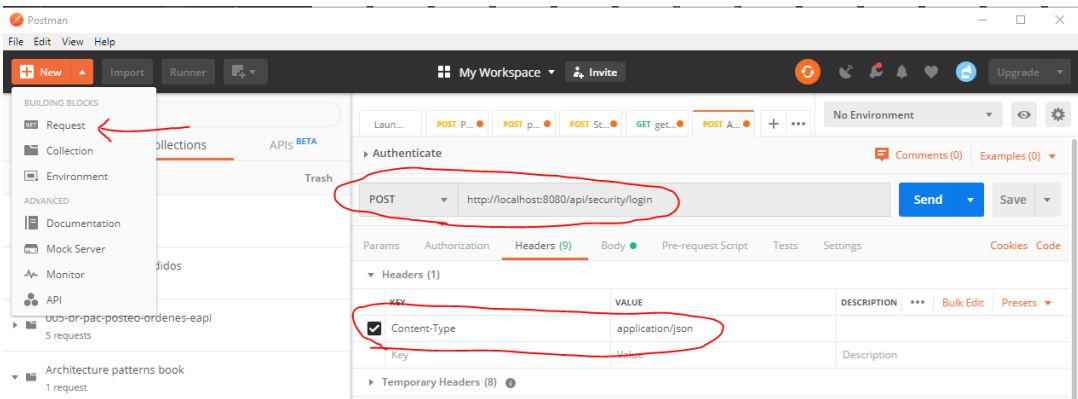


Fig 267 - Preparando el Request para la Autenticación.

Por último, nos vamos a la sección de "Body" y agregamos como payload el siguiente contenido:

```
1. {  
2.   "username": "oscar",  
3.   "password": "1234"  
4. }
```

Finalmente, presionamos el botón "Send" para realizar la autenticación:

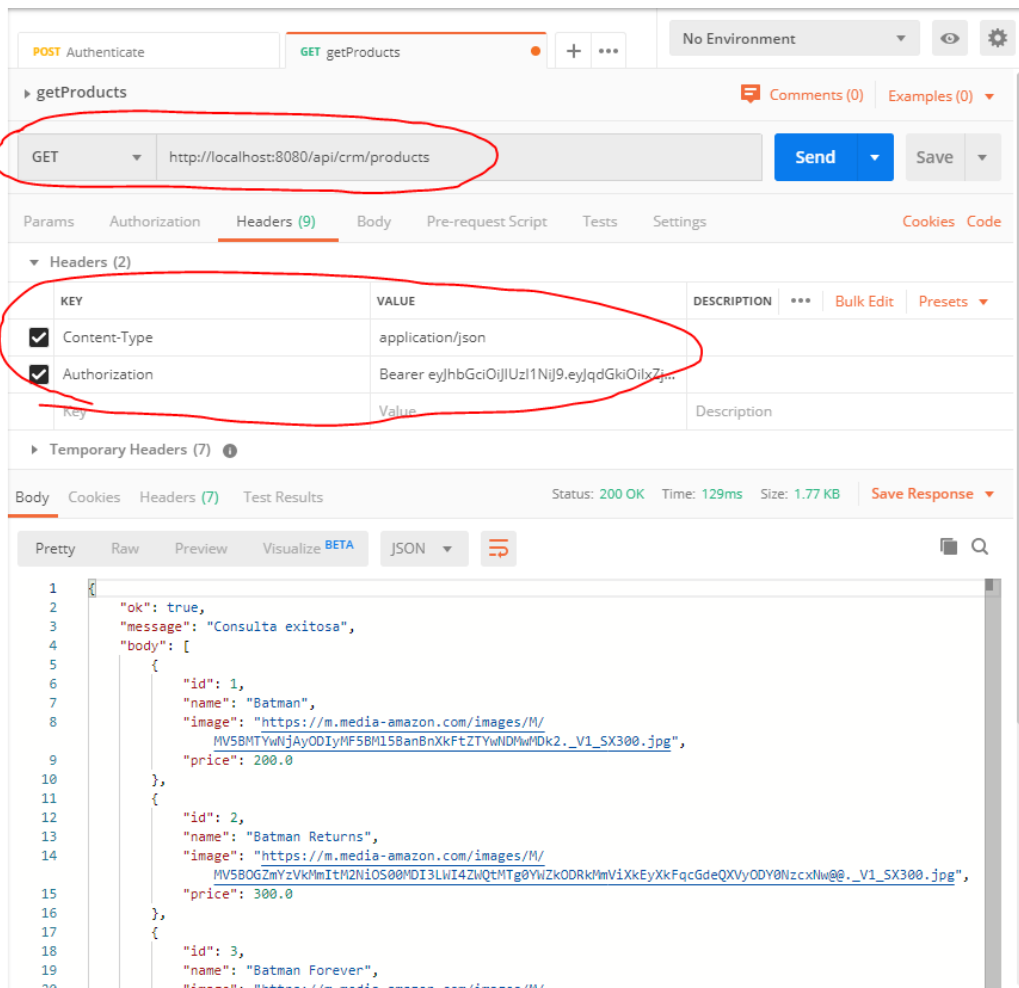


Fig 269 - Probando el token

Como podrás observar, el API ha podido validar el token y nos ha regresado todos los productos, pero para salir de dudas de si el token realmente está funcionando, podemos hacer dos cosas, la primera es borrar el header "Authorization" y ejecutar nuevamente el servicio, para que veas que este lanzará un error:

Cuando ejecutamos el servicio de autenticación se manda llamar al método *Login* de la clase *SecurityService* del Microservicio *securit*, y el cual se ve de la siguiente manera:

```
1. public LoginResponseDTO login(LoginDTO request)
2.     throws ValidateServiceException, GenericServiceException {
3.     try {
4.         LoginResponseDTO response = null;
5.         if(request.getToken() != null) {
6.             response = decryptToken(request.getToken());
7.             response.setToken(createJWT(response.getUsername()));
8.             return response;
9.         }else {
10.            String username = request.getUsername();
11.
12.            Optional<User> userOpt = userDAO.findById(username);
13.            if (!userOpt.isPresent()) {
14.                throw new ValidateServiceException("Invalid user or password");
15.            }
16.
17.            User user = userOpt.get();
18.            if(!user.getPassword().equals(request.getPassword())) {
19.                throw new ValidateServiceException("Invalid user or password");
20.            }
21.
22.            String token = createJWT(user.getUsername());
23.
24.            response = new LoginResponseDTO();
25.            response.setUsername(user.getUsername());
26.            response.setRol(user.getRol().name());
27.            response.setEmail(user.getEmail());
28.            response.setToken(token);
29.
30.            logger.info(user.getUsername() + " authenticated");
31.            tracer.currentSpan().tag("login",
32.                user.getUsername() + " authenticated");
33.        }
34.        return response;
35.    } catch(ValidateServiceException e) {
36.        logger.info(e.getMessage());
37.        tracer.currentSpan().tag("validate", e.getMessage());
38.        throw e;
39.    }catch (Exception e) {
40.        logger.error(e.getMessage(), e);
41.        tracer.currentSpan().tag("validate",
42.            "Error al autenticar al usuario" + e.getMessage());
43.        throw new GenericServiceException("Error al autenticar al usuario");
44.    }
45. }
```


Algo interesante sobre el servicio de login es que permite la autenticación de dos formas, la primera es mediante usuario y contraseña, pero una segunda forma es mediante token. Por ejemplo, si eres un usuario que ya tiene un token, puede mandar el token para obtener un nuevo token con una nueva fecha de vigencia adicional de los datos del usuario. Esa lógica la puedes ver en las líneas 5 a 8.

Cuando mandamos el usuario y password para la autenticación, en este caso, lo primero que haremos será buscar al usuario por medio del *username* que nos envían como parámetro (línea 12), luego en las siguientes líneas validamos que el usuario exista, seguido de validar la contraseña contra la obtenida como parámetro.



Error común

Para este ejemplo hemos almacenado las contraseñas en texto plano, sin embargo, el estándar de la industria es procesar el password por un algoritmo de digestión no reversible como es el caso de MD5, con la intención de que nadie pueda saber que input genero ese hash. Por otro lado, la comprobación del password es simple, realizamos el mismo proceso de digestión sobre el password que nos envía para el login y lo comparamos contra el password digerido de la base de datos, al final, si la contraseña enviada es correcta, dará como resultado el mismo hash que tenemos en la base de datos, pero al mismo tiempo, no habrá forma de saber el password real del usuario.

Seguido de esto, viene la parte interesante (línea 22)., donde ya hemos validado al usuario y creamos el token. Para la creación del token hacemos una llamada al método *createJWT* que recibe como único parámetro el *username* y se ve de la siguiente manera:

```
1. private String createJWT(String username) {  
2.     //The JWT signature algorithm we will be using to sign the token  
3.     SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256;  
4.  
5.     long nowMillis = System.currentTimeMillis();  
6.     Date now = new Date(nowMillis);  
7.  
8.     //We will sign our JWT with our ApiKey secret
```

```

9.     byte[] apiKeySecretBytes = DatatypeConverter.parseBase64Binary(TOKEN);
10.    Key signingKey = new SecretKeySpec(apiKeySecretBytes,
11.        signatureAlgorithm.getJcaName());
12.
13.    //Let's set the JWT Claims
14.    JwtBuilder builder = Jwts.builder().setId(UUID.randomUUID().toString())
15.        .setIssuedAt(now)
16.        .setSubject("SecurityService")
17.        .setIssuer(username)
18.        .signWith(signatureAlgorithm, signingKey);
19.
20.    //if it has been specified, let's add the expiration
21.    long expMillis = nowMillis + TOKEN_VALIDA_TIME;
22.    Date exp = new Date(expMillis);
23.    builder.setExpiration(exp);
24.
25.    //Builds the JWT and serializes it to a compact, URL-safe string
26.    return "Bearer " + builder.compact();
27. }

```

No te quisiera aburrir con los detalles técnicos que hay en este método, por lo que nos centraremos en la parte donde se establece el *payload* del token. Entre las líneas 14 y 18 podrás ver como utilizamos el patrón de diseño *Builder* para crear el token, donde podemos establecer el id del token (línea 14), fecha de generación del token (línea 15), entidad que emite el token (línea 16), usuario al que se emite el token (línea 17) y finalmente, en la línea 18 aplicamos el algoritmo de cifrado y la llave privada. Finalmente, en la línea 23 se establece la vigencia del token y en la línea 26 se construye el token y se retorna.

Ya con el token construido, se retorna el token junto con algunos datos adicionales del usuario autenticado.

Proceso de validación del token

En este punto ya nos queda más claro cómo es que el token se construye y el proceso por medio del cual se realiza la autenticación, sin embargo, falta analizar cómo es que el token se valida al momento de ejecutar un servicio:

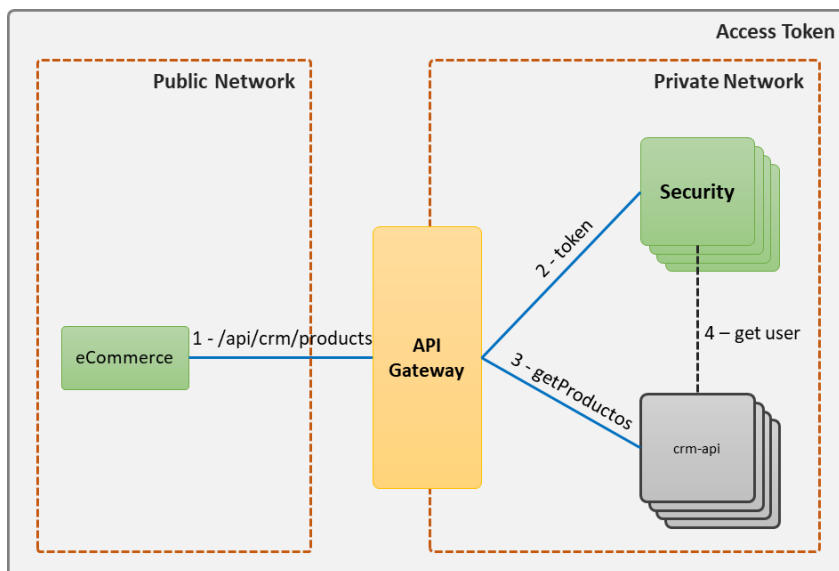


Fig 272 - Flujo de para consumir un servicio mediante un Token.

El flujo comienza cuando el usuario desde el eCommerce ya cuenta con un Token válido en el Local Storage, el cual utiliza para hacer la llamada al API, por lo tanto, el API Gateway recibe la petición con la URL del recurso a consultar y el Token, así que el API Gateway determina si el recurso que intenta ser accedido es restringido, en caso afirmativo, entonces el API Gateway toma el token y lo envía nuevamente al servicio de login que analizamos hace un momento (recordarás que puede autenticar por token y usuarios/password). Entonces, si el token es válido, la petición se redirecciona al servicio de consulta de productos, en otro caso, la petición es rechazada.

Cuando el API Gateway redirecciona la llamada al servicio de consulta de productos, envía también el token como parte de la solicitud, con la finalidad de que el Microservicio *crm-api* sepa quien lo está llamado. El Microservicio *crm-api* no requiere el token para comprobar que sea válido, ni tampoco para autenticar al usuario, porque eso ya lo ha hecho el API Gateway antes. En realidad, el token se propaga para que los Microservicios puedan tener el contexto del usuario que los está llamando, por lo que, si requirieran el detalle completo del usuario, tendrían que hacer una nueva llamada al Microservicio *security* para obtener la información del usuario a partir del token. Este último paso no es obligatorio, sobre todo si el servicio a consumir no requiere saber el usuario que realiza la petición, como es el caso de la consulta de los productos, pero si es requerido, por

ejemplo, cuando creamos la orden, ya que debemos saber a qué usuario asociar la nueva orden creada.

Dicho lo anterior, analicemos el proceso por medio del cual el token es validado, para esto, es necesario regresarnos al API Gateway (*api-gateway*), más precisamente, a la clase *SecurityFilter*:

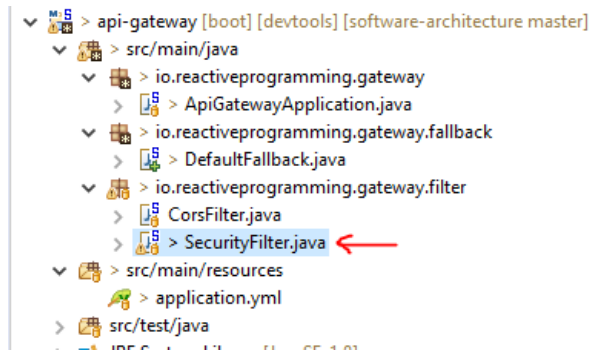


Fig 273 - Clase SecurityFilter.

```
1. public class SecurityFilter extends ZuulFilter {
2.
3.     @Autowired
4.     private RestTemplate restTemplate;
5.
6.     @Override
7.     public Object run() throws ZuulException {
8.         RequestContext ctx = RequestContext.getCurrentContext();
9.         HttpServletRequest request = ctx.getRequest();
10.
11.         String path = request.getRequestURI();
12.         if("/api/security/login".equals(path)
13.             || "/api/security/login".equals(path)
14.             || "/api/security/sso".equals(path)
15.             || "/api/security/loginForm".equals(path)
16.             || path.startsWith("/api/crm/products/thumbnail")
17.         ) {
18.             //Handle all public resources
19.         } else {
20.             String token = request.getHeader("Authorization");
21.             if (token == null) {
22.                 ctx.setResponseBody("{\"ok\": false,\"message\": \" "
23.                     + "\"Zuul Unauthorized - Required token\\\"}\"");
24.                 ctx.getResponse().setContentType("application/json");
25.                 ctx.setResponseStatusCode(401);
```

```

26.     }
27.
28.     Map<String,String> queryParams = new HashMap<>();
29.     queryParams.put("token", token);
30.
31.     class LoginWrapper extends WrapperResponse<LoginResponseDTO>{};
32.     WrapperResponse<LoginResponseDTO> result =
33.         restTemplate.exchange(
34.             "http://security/token/validate?token={token}",
35.             HttpMethod.GET, null,
36.             new ParameterizedTypeReference
37.                 <WrapperResponse<LoginResponseDTO>>() {},
38.             queryParams).getBody();
39.
40.     if(!result.isOk()) {
41.         ctx.setResponseBody("{\"ok\": false,\"message\": \""
42.             +result.getMessage()+"\""});
43.         ctx.getResponse().setContentType("application/json");
44.         ctx.setResponseStatusCode(401);
45.     }
46. }
47. return null;
48. }
49.
50. @Override
51. public boolean shouldFilter() {
52.     return true;
53. }
54.
55. @Override
56. public int filterOrder() {
57.     return 0;
58. }
59.
60. @Override
61. public String filterType() {
62.     return "pre";
63. }
64. }

```

En realidad, esta clase ya la habíamos analizado en una sección antes, cuando tocamos el patrón *API Gateway*, pero no profundizamos en la parte de la autenticación, por lo que esta vez regresaremos para realizar algunas explicaciones.

Para empezar, esta clase es un Filtro Zuul, lo que quiere decir que cualquier llamada al API será interceptada por esta clase antes de que la llamada al servicio real se lleve a cabo. Dicho esto, todas las llamadas al API serán interceptadas por

esta clase y el método `run` será ejecutado, el cual tiene como responsabilidad validar el token (solo para las URL que no sean públicas).

A partir de la línea 20 se llevará a cabo la validación del token, donde precisamente en esta línea se recupera el header `Authorization` y luego en la línea 21 se valida si está presente, por lo que un error es retornado si no está presente el token.

Luego en las líneas 32 a 38 se hace una llamada al servicio de seguridad para validar el token, si no es exitosa la validación entonces se regresa un código 401 y la descripción del error correspondiente.

Finalmente, si el token es correcto, simplemente terminará el método y Zuul lo interpretará como que todo está bien y continuará con la llamada al servicio.

Por otra parte, es importante resaltar que el token es propagado al servicio destino, por lo que siempre podrá hacer una llamada al servicio de *security* para validar el token y obtener la información del usuario asociada al token.

Conclusiones

Hoy en día el uso de Tokens se ha venido popularizando rápidamente, no solo por la seguridad que implica, si no que permite mantener servicios sin estado del lado del servidor, lo que ha favorecido mucho a las API de tipo REST, pues ya no es necesario mantener una sesión del lado del servidor, ahorrando muchos recursos.

Por otro lado, no comprometemos la contraseña del usuario, pues un token puede ser validado e incluso, saber a qué usuario fue emitido, al mismo tiempo que podemos agregar datos, como privilegios fecha de caducidad, etc.

Single Sign On (Inicio de sesión único)

La autenticación de los usuarios es uno de los procedimientos más normales de una aplicación, pues prácticamente todas las aplicaciones necesitan saber la identidad del usuario que lo está utilizando, pero a medida que las empresas crecen, lo hacen también el número de aplicaciones necesarias para soportar la operación, lo que hace que los usuarios tengan que recordar las credenciales para cada aplicación, lo que reduce la productividad y es molesto para los usuarios.

Problemática

En un entorno empresarial es común que las empresas tengan múltiples aplicaciones y en cada una de estas, el usuario tenga una cuenta diferente, ya sea que tenga un password diferente o incluso, un nombre de usuario diferente.

En la práctica es común ver que los usuarios se autentican con su correo empresarial, mientras que en otros, necesitan un nombre de usuario generado por el sistema o por el departamento de sistemas, lo que hace confuso para los usuarios que cuenta utilizar en cada una de las aplicaciones, sumado a esto, el password varía según la aplicación.

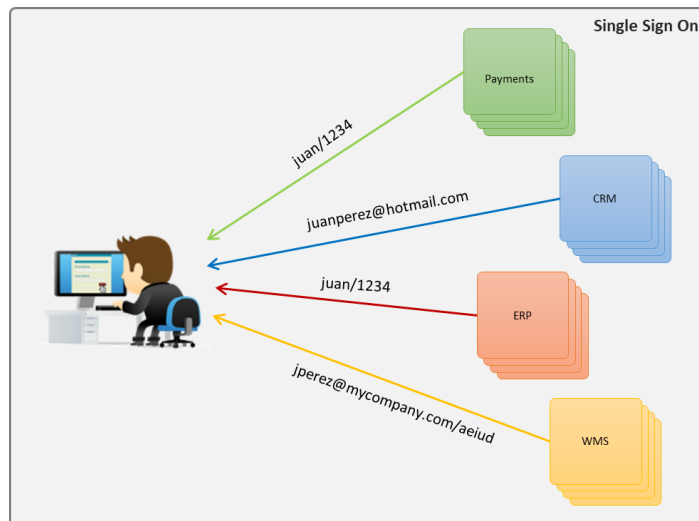


Fig 274 – Inicio de sesión por aplicación.

Como podemos ver en la imagen anterior, cada usuario necesita introducir credenciales diferentes para cada aplicación, lo que lo obliga a recordar su nombre de usuario y password para cada aplicación.

Por otro lado, cada aplicación deberá contar con un equipo de administración especializado para dar acceso a cada aplicación, lo que hace que dar acceso o desbloquear usuario sea una tarea complicada, además del costo de tener un equipo de soporte especializado para cada aplicación.

Solución

Para solucionar este problema es necesario separar la lógica de autenticación de la aplicación como un componente independiente, que tenga como única responsabilidad la de autenticar a los usuarios.

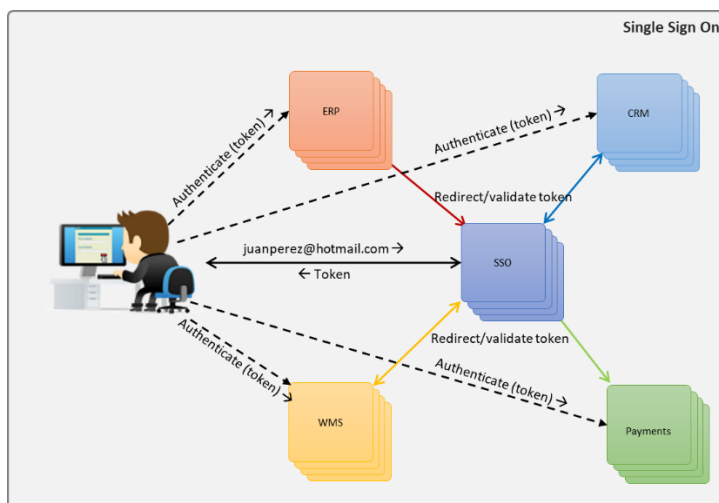


Fig 275 - Implementando un SSO.

La imagen anterior ilustra como quedaría una arquitectura que implementa SSO (Single Sign On), en el cual podemos apreciar cómo hemos agregado un componente central encargado únicamente de la autenticación.

La imagen anterior puede resultar sumamente confusa, pues existen demasiados flujos que parecen no tener sentido, además, la ejecución de los pasos puede variar según si el usuario ya está autenticado o no, por lo que comenzaremos analizando el escenario donde un usuario no autenticado intenta entrar a cualquier aplicación.

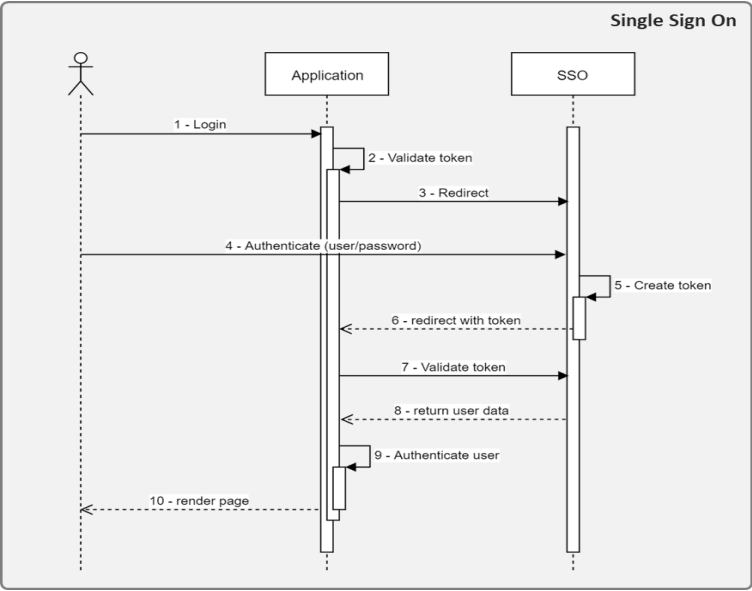


Fig 276 - Diagrama de secuencia de la autenticación.

El flujo de autenticación se interpreta de la siguiente manera:

1. El usuario intenta autenticarse en alguna aplicación (cualquiera).
2. La aplicación valida si como parte de la petición para acceder a la aplicación viene un Token (los tokens los analizamos en la sección pasada).
3. La aplicación determina que el usuario no cuenta con un token, por lo que redirecciona al usuario al SSO (Single Sign On) para que se autentique.
4. El usuario ahora se encuentra en el SSO y le solicita que introduzca sus credenciales.
5. El SSO valida las credenciales y le emite un Token que valida que sus credenciales son correctas
6. El usuario es redireccionado nuevamente del SSO a la aplicación que lo redirecciono al SSO, pero como parte del redireccionamiento se envía el Token.
7. Ya en la aplicación, esta toma el token creado por el SSO y lo valida contra el SSO para comprobar que es válido.
8. El SSO valida el Token y le regresa los datos del usuario, como ID, nombre de usuario, email, etc.
9. La aplicación recibe los datos del SSO y autentica al usuario dentro de la aplicación.
10. La aplicación renderiza la página como un usuario ya autenticado.

Este es el flujo más largo, pues el usuario intenta entrar a la aplicación sin un token, lo que lo lleva a una serie de redireccionamientos, sin embargo, existe el segundo caso, en donde el usuario intenta entrar a la aplicación ya autenticado, es decir, que ya cuenta con un token.

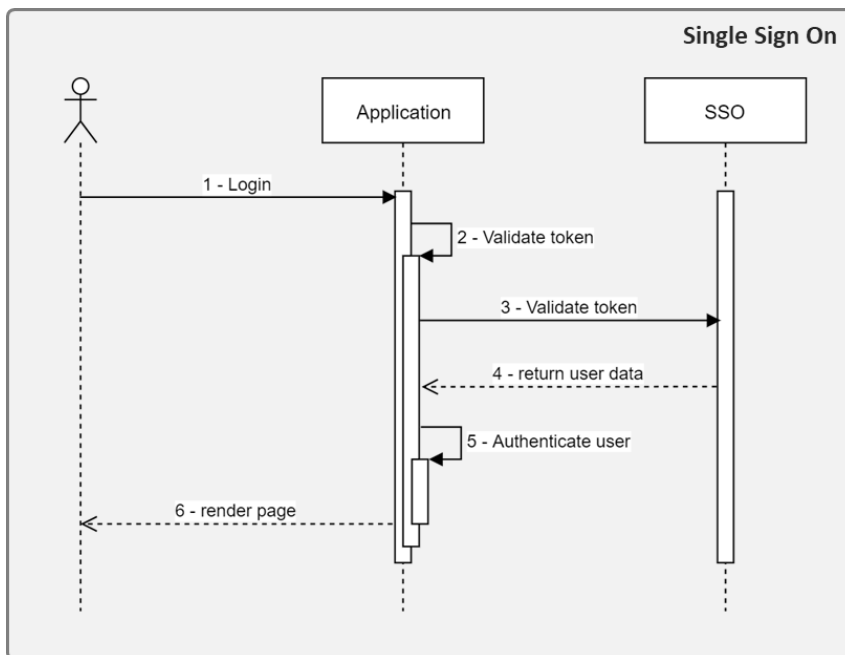


Fig 277 - Autenticación con token.

Este segundo escenario es mucho más simple y se interpreta de la siguiente manera:

1. El usuario intenta autenticarse en alguna aplicación (cualquiera).
2. La aplicación valida si como parte de la petición para acceder a la aplicación viene un Token.
3. La aplicación envía el token al SSO para su validación.
4. El SSO valida el Token y le regresa los datos del usuario, como ID, nombre de usuario, email, etc.
5. La aplicación recibe los datos del SSO y autentica al usuario dentro de la aplicación.
6. La aplicación renderiza la página como un usuario ya autenticado.

Puedes pensar que todo este proceso de redireccionamientos pueda ser más complejo para el usuario, pero la realidad es que es apenas perceptible, incluso para usuarios técnicos como nosotros.

SSO en el mundo real

Para comprender como encaja el SSO en nuestra aplicación basta con solo entrar a la aplicación y observar la barra de navegación. Si recuerdas, la aplicación responde en la URL <http://localhost:3000>, por al momento de entrar a la página nos redirecciona a la página <http://localhost:8080/api/security/sso?redirect=http://localhost:3000>



Importante

Para que este comportamiento suceda debes de tener la sesión cerrada, de lo contrario te llevará directo a la aplicación.

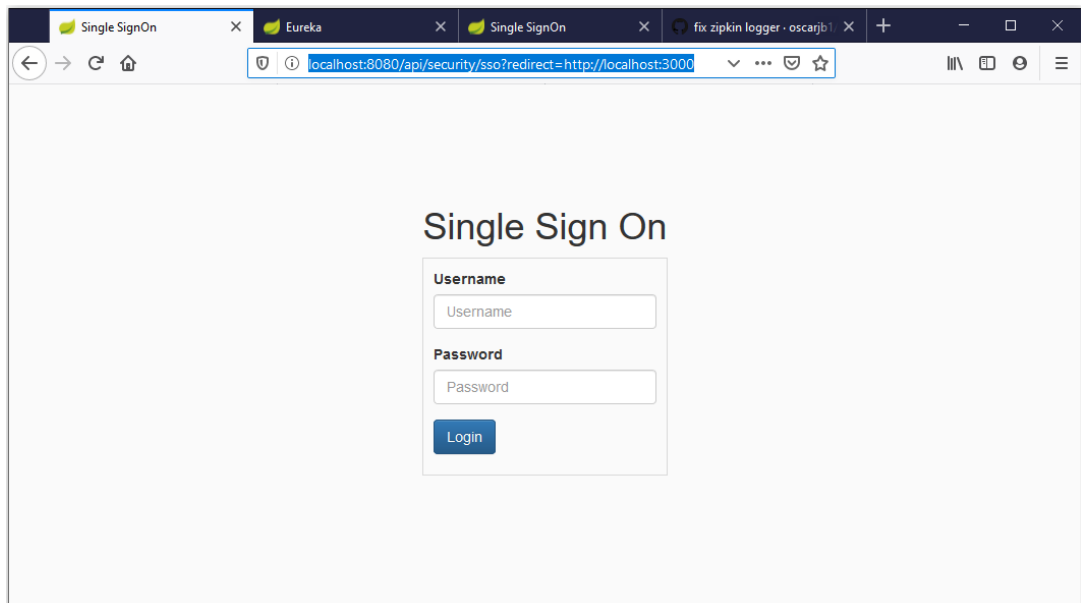


Fig 278 - Redireccionamiento del SSO.

Si descomponemos la URL a la cual nos está redirigiendo, podremos analizar mejor lo que está pasando:

<http://localhost:8080/api/security/sso?redirect=http://localhost:3000>

Si nos fijamos bien, la primera parte (rojo) de la URL corresponde a nuestro API Gateway, es decir el puerto 8080, lo que quiere decir que hemos pasado de la App del ecommerce (puerto 3000) al API Gateway. La segunda parte de la URL nos indica el recurso que estamos accediendo, que, en este caso, nos dice que estamos consumiendo el recurso *sso* del Microservicio *security*. Finalmente, en la tercera parte (verde) de la URL estamos definiendo a donde queremos que el SSO no redireccione una vez que la autenticación sea exitosa, y si pones atención, nos está redirigiendo al puerto 3000, es decir, nuevamente a la App.

Dicho lo anterior, el siguiente paso será introducir las credenciales para acceder, lo que nos dará como resultado la autenticación y el redireccionamiento automático a la aplicación:

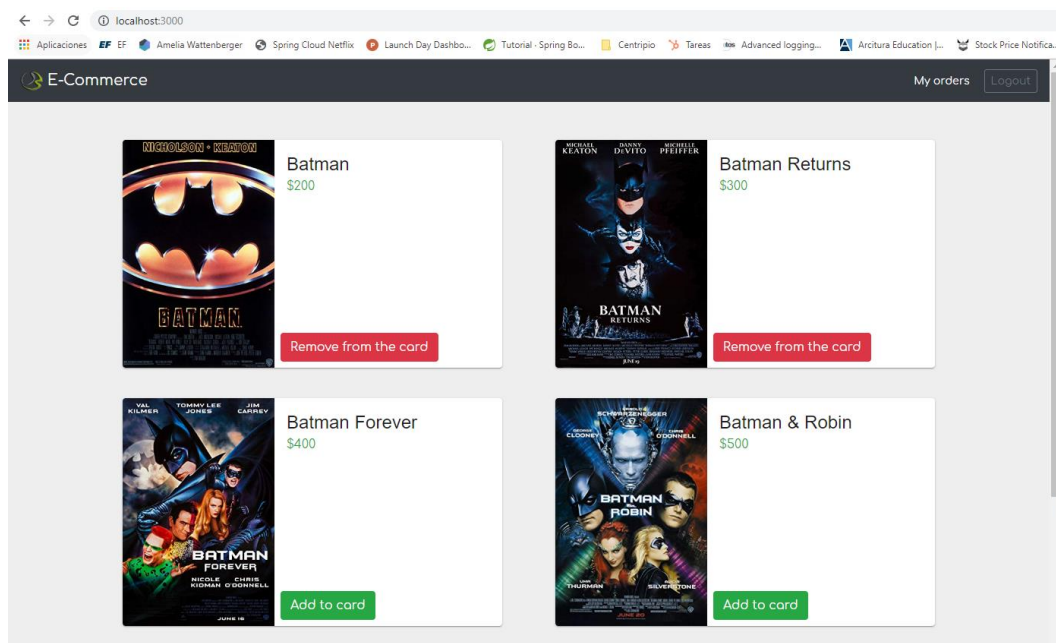


Fig 279 - Autenticación exitosa.

Ahora bien, entre la autenticación en el SSO y que nos muestre la página de la imagen anterior, sucedió un paso adicional que no alcanzamos a ver por la rápida que pasa, y es que si observamos la URL del navegador, veremos que estamos en <http://localhost:3000>, sin embargo, el SSO no nos direcciona exactamente a esta página, si no que nos redirecciona pero con el token para que la aplicación nos pueda autenticar.

Entonces para ver que paso en medio tendremos que hacer algunos pasos adicionales para poder *debugear* lo que está pasando, para esto, será necesario que cerremos sesión nuevamente, lo que nos redireccionará nuevamente a la página del SSO, una vez allí, tendremos que abrir el inspector de elementos de Google Chrome (*Ctrl + Mayus + I*) y nos ubicaremos en la pestaña "Network".

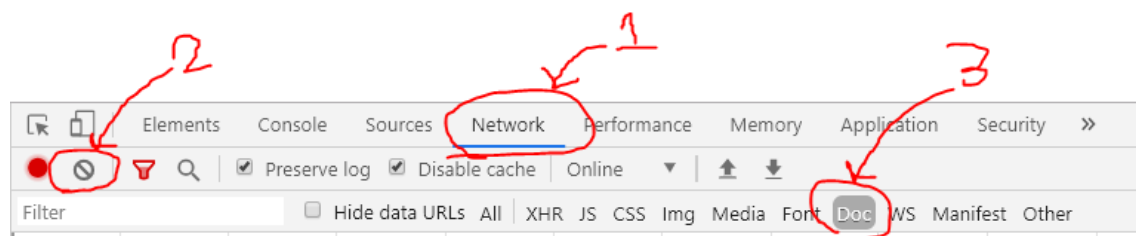


Fig 280 - Preparar el debugger.

Lo siguiente será limpiar los registros (paso 2 de la imagen) y filtraremos los registros a solo documentos (paso 3 de la imagen). En este punto ya estamos listos para realizar el análisis, por lo que lo sigue será introducir tu usuario y password y observar detenidamente la barra de navegación, verás cómo esta cambia rápidamente antes de llevarnos a la aplicación, y es por eso que estamos *debugeando* la página. Ahora bien, si realizamos correctamente los pasos anteriores, veremos en el registro lo siguiente:

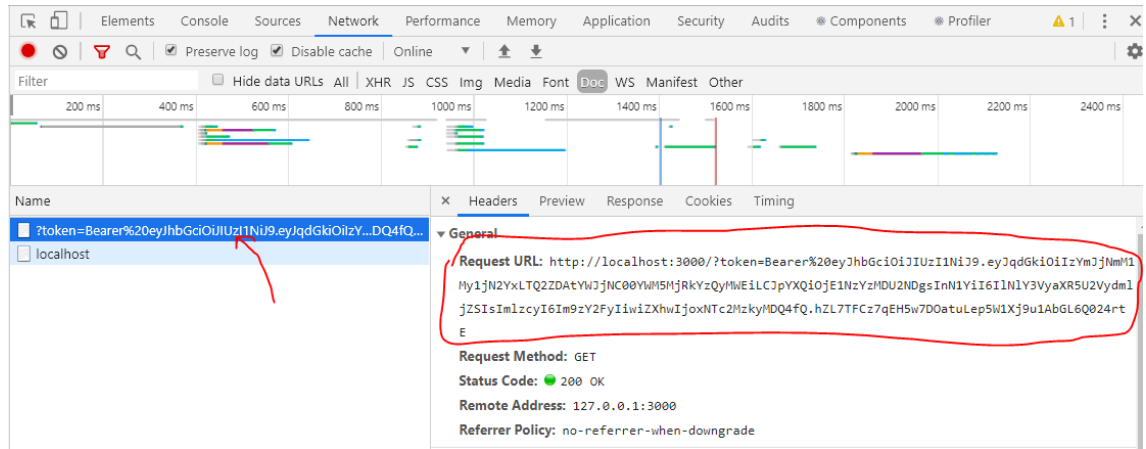


Fig 281 - Analizando el debugger.

Dentro de los registros seleccionaremos el que comienza con *?token=* y del lado derecho veremos la URL real a la que el SSO nos redireccionó, la cual es la siguiente:

`http://localhost:3000/?token=Bearer%20eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJmM1My1jN2YxLTQ2ZDAtYWJjNC00YWw5MjRkYzQyMWEiLCJpYXQiOiJlNzYzMDU2NDgsInN1YiI6IiNlY3VyaXR5U2VydmljZSI6ImZcyI6Im9zY2FyIiwiaXhwIjoxNTc2MzkyMDQ4fQ.hZL7TFCz7qEH5w7D0atuLep5W1Xj9u1AbGL6Q024rtE`

Te podrás dar cuenta que, en efecto, el SSO nos está redireccionando a la App (puerto 3000) pero adicional, nos ha agregado el Query Param token, el cual es nada menos que el Token JWT que nos ha generado el Microservicio Security.

Ahora bien, necesitamos guardar el token para utilizarlo en las siguientes llamadas al API, por lo que el siguiente paso por parte de la aplicación será guardar el token

en el Local Storage, tal y como lo podemos ver en la pestaña "Application" del mismo inspector:

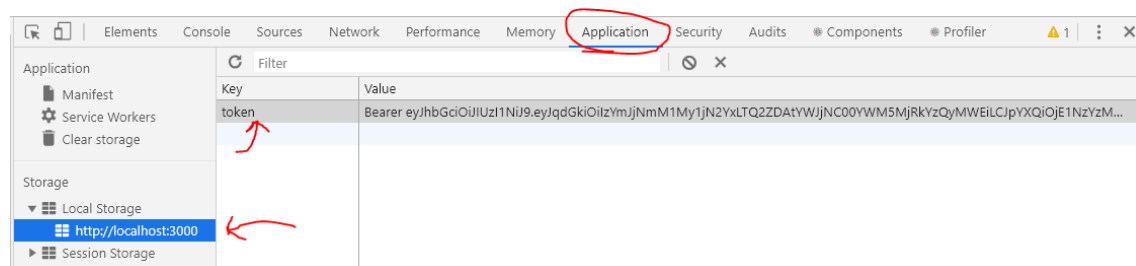


Fig 282 - Verificando el Local Storage.

Observa que el Local Storage ahora tiene un registro llamado token, el cual tiene el token que nos regresó el SSO.

Para analizar como la App guarda el token, podemos regresar al Visual Studio Code y abrir el archivo *Themplete.js*:

```
1. export default class Themplete extends React.Component{
2.
3.   ....
4.
5.   componentDidMount(){
6.     const urlParams = new URLSearchParams(window.location.search);
7.     const newToken = urlParams.get('token');
8.     if(newToken){
9.       window.localStorage.setItem("token", newToken)
10.      window.location = '/'
11.    }
12.
13.    let token = window.localStorage.getItem("token")
14.    if(!token){
15.      window.location = config.sso
16.    }
17.
18.    APIInvoker.invokeGET(`/security/token/validate?token=${token}`, response=>{
19.      this.setState({
20.        login: true
21.      })
22.      this.props.setUser(response.body)
23.    }, error => {
24.      window.location = config.sso
25.    })
26.  }
27.  ....
```

28. }

Esta clase es mucho más grande, por lo que solo nos centraremos en el método *componentDidMount* el cual se ejecuta solo una vez cuando el componente es montado (renderizado en pantalla).

Este método tiene una doble función, la primera es que, si detecta que el token está en los query params, lo guardaremos en el Local Storage (línea 9) y luego nos redirecciona a la raíz de la App (línea 10), esto con el fin de quitar de la URL el token. Ahora bien, como el redireccionamiento se hace de forma inmediato, las siguientes líneas ya no se ejecutan. Sin embargo, como el redireccionamiento es la misma aplicación, la página se vuelve a cargar, lo que hace que este método sea ejecutado nuevamente, pero esta vez, el token ya no estará en la URL, pero si en el Local Storage, lo que hace que se ejecuten las líneas 18 a 25, que lo que hacen es básicamente tomar el Token y validarlo contra el Microservicio *security*, el cual nos dirá si el Token es válido y nos regresará toda la información del usuarios.

Para comprobar esto, podemos regresar al Inspector de Chrome y regresar a la pestaña "Network" y cambiar el filtro a XHR para ver solo las llamadas AJAX al servidor:

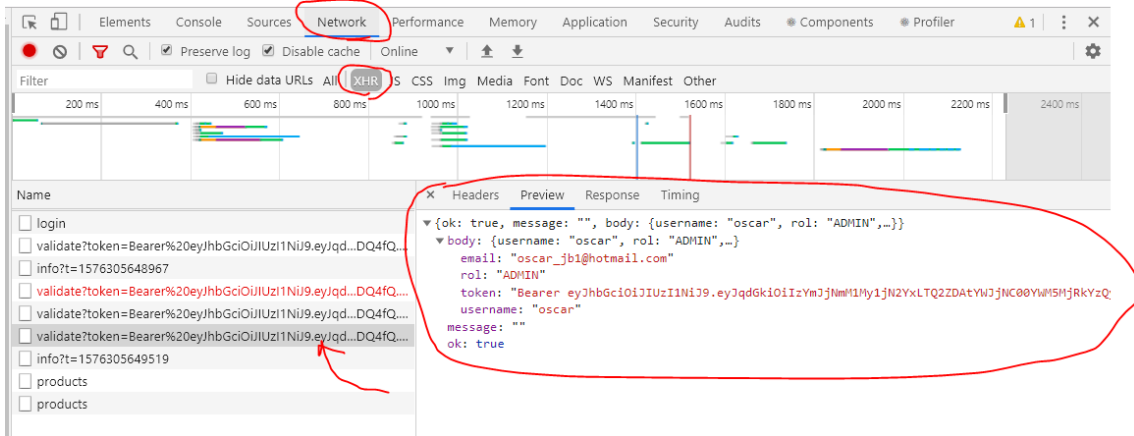


Fig 283 - Analizando la autenticación del usuario por Token.

Seleccionamos el registro que comience con "validate" y nos ubicamos en la pestaña "Preview" del lado derecho y podremos ver cómo hemos enviado el token para autenticar al usuario. Con esta respuesta la App puede finalmente autenticar

al usuario y mostrar el catálogo de las películas para comenzar a realizar las compras.

Ya para terminar, falto agregar un comentario respecto a la clase *Themeplete.js*, y es que en las líneas 13 a 16 se hace el redireccionamiento al SSO en caso de que no tengamos un Token, es por ellos que cuando entramos a la aplicación sin estar firmando, esta nos redirecciona de forma automática.

Comentarios adicionales

Un punto importante que no cubrimos con este ejemplo, es la autenticación de múltiples aplicaciones para comprobar efectivamente el funcionamiento del SSO, pero lo que podrías hacer de ahora en adelante es conectar el resto de aplicaciones a este SSO para generar tus tokens, y de esta forma, hacer que todas las aplicaciones si firmen mediante el SSO.

Un ejemplo de SSO que usamos con frecuencia es conocido como Social Login, el cual es cuando nos autenticamos con Google, Facebook, Twitter o Github, ya que estos utilizan algo muy parecido a lo que estamos haciendo asíendo aquí, y de esta forma, podemos autenticarnos en múltiples plataformas con las mismas credenciales.

Por otra parte, hemos dejado de lado la sección registro de usuarios y la recuperación del password, pues creemos que, si bien es importante para un ambiente productivo, no es necesario para explicar el SSO y cómo es que se integra con el resto de aplicaciones, por lo que podrías intentar agregar esta sección el Microservicio *security* y exponerlo por medio del API Gateway como tarea.

Conclusiones

Como hemos podido observar, la configuración del SSO puede ser algo complicado, sobre todo por los redireccionamientos necesario que tenemos que hacer y la coordinación para que estos funcionen correctamente, pero una vez que lo logramos proporciona una experiencia muy agradable para el usuario, el cual muchas veces ni se entera de que salió de la aplicación para la autenticación.

Store and forward

Prácticamente todas las aplicaciones modernas requieren del envío de mensajes, ya sea para aplicaciones externas o internas, donde forzosamente tendrá que pasar por la red, ya sea local o internet, lo que siempre puede tener problemas de comunicación y eventualmente la pérdida de algún mensaje, lo que nos lleva a crear estrategias de comunicación robustas que aseguren la entrega de los mensajes.

Problemática

El principal problema de las aplicaciones distribuidas como SOA o Microservicios es la comunicación, ya que, al tener múltiples componentes distribuidos, estamos en riesgo de que algún mensaje no llegue a su destino. Esto puede ocurrir por varias razones, pero las principales pueden ser fallas en la red, en las aplicaciones o que estas no estén disponibles.

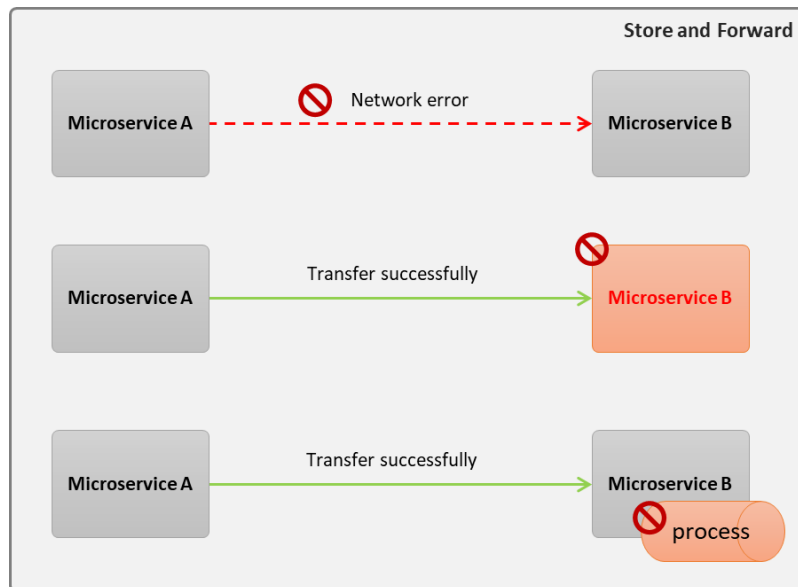


Fig 284 - Errores de comunicación más comunes.

Si bien existen muchos motivos por los cuales un mensaje puede no ser entregado, los tres más importantes son los que se ilustran en la imagen anterior.

El primer caso (arriba) es cuando existe un problema de comunicación entre los dos componentes, en este caso se puede dar por ejemplo por una falla en la red o en el internet que impida que los dos componentes se comuniquen entre sí, lo que provocara que el Microservicio A no pueda entregar el mensaje. En este escenario al menos el Microservicio A sabrá que el mensaje no pudo ser entregado y podrá actuar en consecuencia, pero tendremos que hacer echar atrás la transacción en A, porque no logramos terminar todo el proceso, para finalmente, mostrar algún error al usuario.

En el segundo caso (en medio), la red está funcionando correctamente, pero el componente B no, lo que hace que nuevamente el mensaje no pueda ser entregado, lo que implicaría nuevamente echar a atrás la transacción en A y notificar al usuario.

Sin embargo, el tercer caso (abajo) es el más peligroso, ya que tanto la red como el componente B están operando correctamente, por lo que A puede entregar el mensaje a B, sin embargo, una vez que B recibe el mensaje, este falla por algún motivo, lo que provoca un problema grave, ya que A está seguro que B recibió el mensaje, sin embargo B no lo pudo procesar, e incluso, como parte de la falla, este perdió el mensaje, lo que provoca un estado inconsistente entre las dos aplicaciones.

Para comprender la magnitud de este problema, imagina que el componente A es el que procesa el pedido y el B es el que se encarga del embarque (envío) del producto al cliente. El componente A habrá confirmado el pedido y realizado el cobro, pero B que es el encargado del envío del paquete, nunca lo enviará.

Solución

Una de las soluciones más utilizadas es implementar lo que se conoce como Store and Forward, que consiste en el almacenamiento local de los mensajes para su envío a un servidor remoto, de esta forma, el mensaje se almacena primeramente de forma local para impedir que se pierda, luego, el mensaje local es reenviado al servidor remoto cuando este puede aceptar el mensaje, de esta forma se garantiza

la entrega del mensaje incluso si la aplicación destino no está en condiciones de recibirlo en ese momento.

Para lograr una arquitectura que implemente Store and Forward es indispensable eliminar las llamadas directas entre componentes, pues en una arquitectura distribuidas las conexiones de punta a punta son las más propensas a fallar. Entonces, eliminamos las llamadas directas y en su lugar, debemos de poner algo conocido como MOM (Middleware Orientado a Mensajes), los cuales son sistemas especializados en la entrega de mensajes. El MOM recibirá los mensajes del remitente y los persistirá, evitando que una falla o un reinicio produzca la pérdida del mensaje. Por otro lado, el receptor se registrará con el MOM para que este le envíe los mensajes cuando esté disponible.

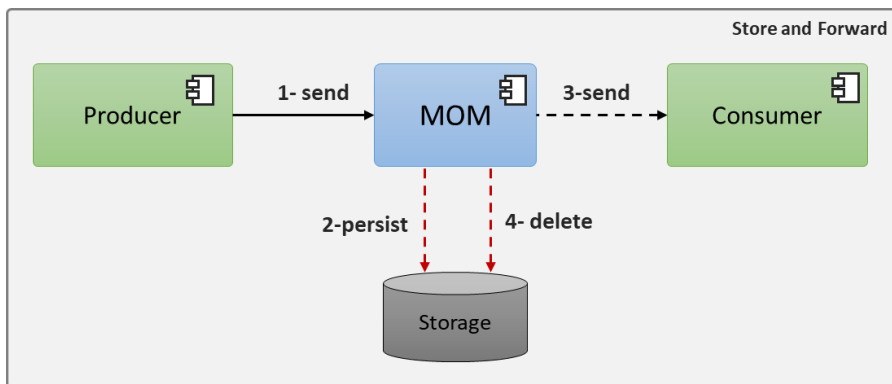


Fig 285 - Implementando Store and Forward.

En la imagen anterior se muestra como quedaría la arquitectura más básica para implementar un Store and Forward, el cual consiste en un Productor (el que produce el mensaje), el MOM que es el sistema especializado en la transmisión de mensajes de alta fidelidad, y el Consumidor (el destinatario de los mensajes). La secuencia de ejecución es la siguiente:

7. El Producer genera un nuevo mensaje que es transmitido al MOM
8. El MOM acepta el mensaje y lo persisten para garantizar que no se pierda ante una falla o reinicio del sistema.
9. El MOM envía el mensaje al consumidor
- 10.El MOM borrar el mensaje una vez que fue entregado.

Algo a tomar en cuenta es que todo este proceso es asíncrono, por lo que el *producer* solo deja el mensaje y se olvida de él, lo que pase de allí en adelante ya no es de su incumbencia y puede seguir trabajando con la garantía de que será entregado.

Si bien, la imagen anterior es la forma más básica de lograr el Store and Forward, existe una variante que es considerada la más robusta, pues permite persistir los mensajes en cada etapa de la transmisión de los mensajes, la cual luce de la siguiente manera:

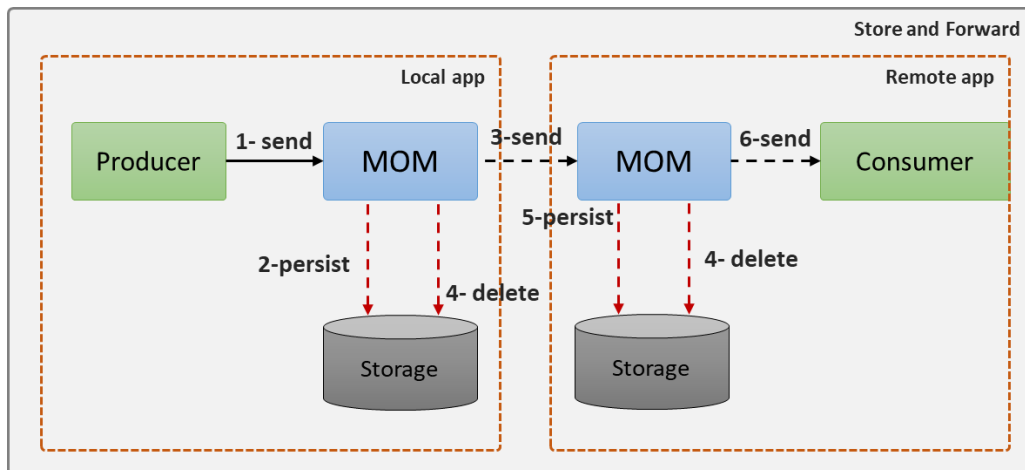


Fig 286 - Store and forward avanzado.

La imagen anterior muestra como algunos productos más avanzado logran el Store and *forward*, el cual consiste en ir persistiendo el mensaje en todos los puntos por lo que se transmite, y en este caso en particular, podemos ver como el mensaje se transmite de MOM a MOM, para garantizar que el mensaje será entregado.

Un ejemplo rápido de algo que utiliza esta arquitectura es el correo electrónico, ya que este funciona replicando los mensajes de un servidor a otros, es decir, cuando enviamos un mail, este se almacena en el servidor SMTP de nuestro proveedor de correo y luego este lo replica al servidor SMTP del destinatario, para finalmente, ser entregado en el cliente de correo de nuestro destinatario, en este caso, nuestro Outlook sería *producer*, el servidor SMTP sería el MOM y el Outlook del destinatario sería el *consumer*.

Store and Forward en el mundo real

Para nuestro proyecto hemos implementado la versión más básica del store and forward debido principalmente a las limitaciones de un equipo local para correr diferentes Microservicios junto con múltiples instancias de un sistema de mensajería, por lo que hemos implementado una arquitectura con un solo MOM.

Si recuerdas, cada vez que realizamos una venta, enviamos un correo electrónico a nuestros clientes con la confirmación del pedido. Este correo es importante porque le confirma al cliente que hemos recibido su pedido, por lo que tenemos que garantizar que se entregue, incluso, si el Microservicio de envío de correos no está disponible, es por eso, que en lugar solicitar el envío del correo electrónico directamente al Microservicio de envío de correos (*mail-sender*) lo hacemos por medio un MOM, es este caso, vamos a utilizar RabbirMQ, una de las plataformas más utilizadas para transmitir mensajes y que permite instalarse localmente o utilizarla en su versión SaaS (Software as a Service).



Importante

La configuración de RabbitMQ la vimos en la unidad pasada, donde explicamos el proyecto E-Commerce y como instalarlo, por lo que sería bueno que regreses a esa sección para comprender lo que hicimos.

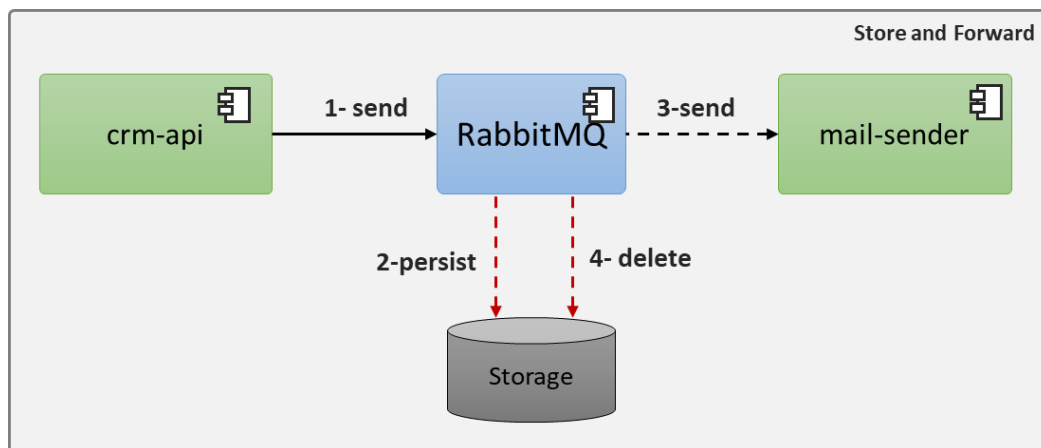


Fig 287 - Arquitectura de nuestro proyecto.

La imagen anterior muestra la arquitectura que tenemos para el envío de correos, en donde el Microservicio *crm-api* enviará la solicitud a una Queue en RabbitMQ, la cual la persistirá e intentará enviarla al destinatario. Algo a tomar en cuenta es que como RabbitMQ persiste el mensaje, este lo retendrá hasta que lo pueda entregar al Microservicio *mail-sender*, incluso si este está apagado, esperará hasta que esté disponible para entregar el mensaje, por lo que no dependeremos de si *mail-sender* está operativo al momento de solicitar el envío del mail.

Vamos a hacer una primera prueba, vamos a entrar a nuestra aplicación y vamos a realizar una prueba, por lo que nos autenticamos con nuestra cuenta y realizamos un pedido, si todo está correctamente configurado, veremos que nos llegará un correo cómo el siguiente:

Hemos recibido tu pedido



oscarblancarte3@gmail.com <oscarblancarte3@gmail.com>



09:21 p. m.

Hola oscar,

Hemos recibido tu pedido #0c658a1d-beca-4756-8984-3cb772eab2d9, tambien hemos confirmado tu pago, por lo que estarás recibiendo tus productos muy pronto

Fig 288 - Ejemplo del email de confirmación de la compra.

Ahora bien, dijimos que con Store and forward deberíamos de recibir el correo incluso si la aplicación de envío está fallando, por lo vamos a apagar el Microservicio *mail-sender* y vamos realizar una nueva compra.

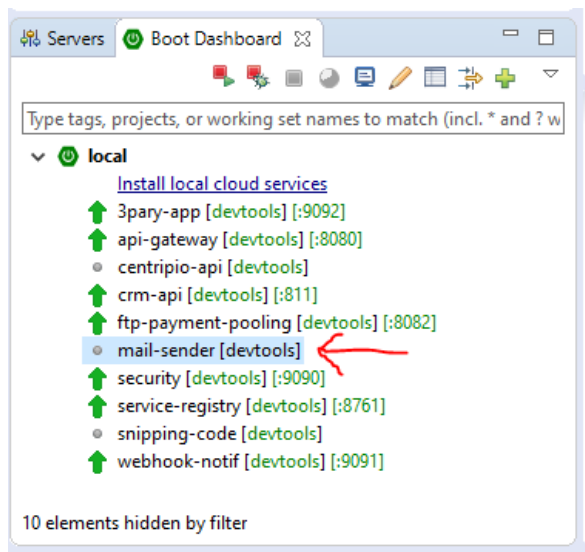


Fig 289 - Apagando el microservicio mail-sender.

Al terminar la compra verás que esta termina correctamente, incluso si el Microservicio *mail-sender* está apagado, y esto se debe a que hemos desacoplado el Microservicio *crm-api* del *mail-sender* poniendo a RabbitMQ entre medio.

Ahora bien, vamos a esperar un momento más para comprobar que el mail de la confirmación de la compra no ha llegado. Tras esperar unos minutos, vamos a iniciar nuevamente el Microservicio *mail-sender* y veremos que tras unos segundos de haberlo iniciado mágicamente habrá llegado el correo que creíamos pedido.

Una vez que hemos explicado que esto realmente funciona, pasaremos a explicar cómo es que esto funciona, por lo que regresaremos al ya conocido método *createOrder* de la clase *OrderService* del Microservicio *crm-api*:

```
1. public SaleOrderDTO createOrder(NewOrderDTO order)
2. throws ValidateServiceException, GenericServiceException {
3.     logger.info("New order request ==>");
4.     try {
5.
6.         ...
7.
8.         EmailDTO mail = new EmailDTO();
9.         mail.setFrom("no-reply@crm.com");
```

```

10.     mail.setSubject("Hemos recibido tu pedido");
11.     mail.setTo(newSaleOrder.getCustomerEmail());
12.     if(paymentMethod==PaymentMethod.CREDIT_CARD) {
13.         mail.setMessage(String.format("Hola %s,<br> " +
14.             "Hemos recibido tu pedido <strong>#%s</strong>, " +
15.             "tambien hemos confirmado tu pago, por lo que " +
16.             "estarás recibiendo tus productos muy pronto",
17.             newSaleOrder.getCustomerName(),
18.             newSaleOrder.getRefNumber()));
19.     }else {
20.         mail.setMessage(String.format("Hola %s,<br> Hemos " +
21.             "recibido tu pedido <strong>#%s</strong>, " +
22.             "debido a que has seleccionado la forma de pago " +
23.             "por depósito bancario, esperaremos hasta tener " +
24.             "confirmado el pago para enviar tus productos.",
25.             newSaleOrder.getCustomerName(),
26.             newSaleOrder.getRefNumber()));
27.     }
28.
29.     sender.send( "emails", null, mail);
30.
31.     ...
32.
33.     return returnOrder;
34. } catch(ValidateServiceException e) {
35.     e.printStackTrace();
36.     throw e;
37. }catch (Exception e) {
38.     e.printStackTrace();
39.     throw new GenericServiceException(e.getMessage(),e);
40. }
41. }

```

Para hacer más clara la explicación vamos a dejar solo la parte del método relacionada con el envío del mail, que es lo que estamos viendo en el fragmento de código de arriba.

Podrás ver como en la línea 8 iniciamos una instancia del DTO *EmailDTO*, el cual utilizaremos para definir para quien deberá ir el email, el título y el cuerpo del correo. Seguido, en la línea 29 hacemos el envío de ese DTO.

Para el envío del DTO nos apoyamos de la clase *RabbitSender* que es donde tenemos configurado como se envía el mensaje a RabbitMQ:

```

1. @Component
2. public class RabbitSender {
3.
4.     @Autowired

```

```

5.     private RabbitTemplate template;
6.
7.     public void send(Object message) {
8.         template.convertAndSend("emails", "*", message);
9.     }
10.
11.    public void send(String exchange, String routingKey, Object payload) {
12.        template.convertAndSend(exchange, routingKey, payload);
13.    }
14.
15. }

```

No quisiera entrar en los detalles técnicos de RabbitMQ, por lo que solo quiero que veas como en el método *send* (línea 7) hacemos el envío del DTO y pasamos otros dos parámetros, de los cuales solo el primero es el importante, observa que dice *"emails"*, y eso hace referencia a la *Queue* a la cual estamos enviando el mensaje.

Por otra parte, si revisamos el archivo *application.yml* del proyecto *crm-api*, podremos ver la configuración para conectarnos a RabbitMQ:

```

1. rabbitmq:
2.   host: barnacle.rmq.cloudamqp.com
3.   port: 5672
4.   username: xxxxxxxx
5.   password: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
6.   virtual-host: xxxxxx

```

Gracias a esta configuración es que nos podemos conectar a RabbitMQ (recuerdo que en la unidad pasada hablamos de como configurar RabbitMQ)

Ya que hemos visto cómo es que se hace el envío, vamos hacer una nueva venta, pero antes de eso vamos a volver a apagar el Microservicio *mail-sender*, una vez apagado, ahora si realizamos la nueva venta. Nuevamente veremos que el mail no llega, pero ahora quiero que veamos que está pasando en RabbitMQ, por lo que regresaremos a la página donde creamos nuestra cuenta (cloudamqp.com), iniciaremos sesión y daremos click en el botón que dice "RabbitMQ Manager"

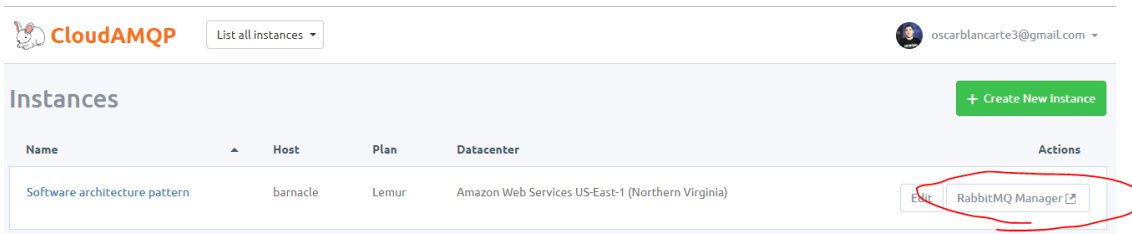


Fig 290 - Entrando al administrador de RabbitMQ.

Esto nos llevará a la página de administración de RabbitMQ, y daremos click en la pestaña que dice "Queues":

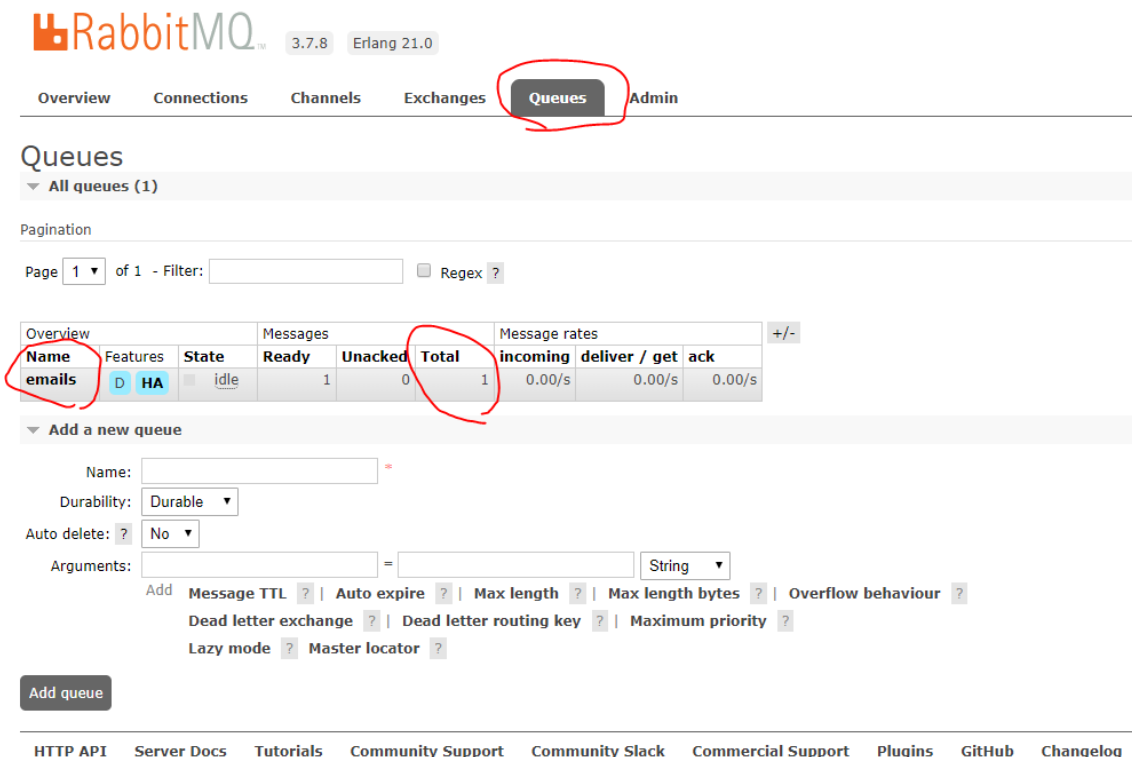


Fig 291 - Analizando las colas de RabbitMQ.

Quiero que prestes atención a la tabla que aparece, principalmente a la columna "emails" y la columna "Total", esta primera corresponde al nombre de la Queue,

la cual corresponde con lo que vimos en la clase *RabbitSender* hace un momento, y la columna "Total" corresponde al total de mensaje que tenemos encolados en la Queue, quiere decir que tenemos 1 mensaje que no ha sido entregado, el cual corresponde a la venta que acabamos de realizar.

Ahora bien, quiero que realices unas cuantas ventas más con el Microservicio *mail-sender* apagado, y luego actualiza la página de RaabbitMQ para que veas que la columna total se irá incrementando. Esto es importante, porque quiere decir que los mensajes se están persistiendo para ser entregados en cuanto prendamos el *mail-sender*.

Antes de encender nuevamente el Microservicio *mail-sender*, vamos a dar click en el nombre de la Queue (emails), es decir, en la columna "Name" para ver lo que contiene la Queue en este momento:

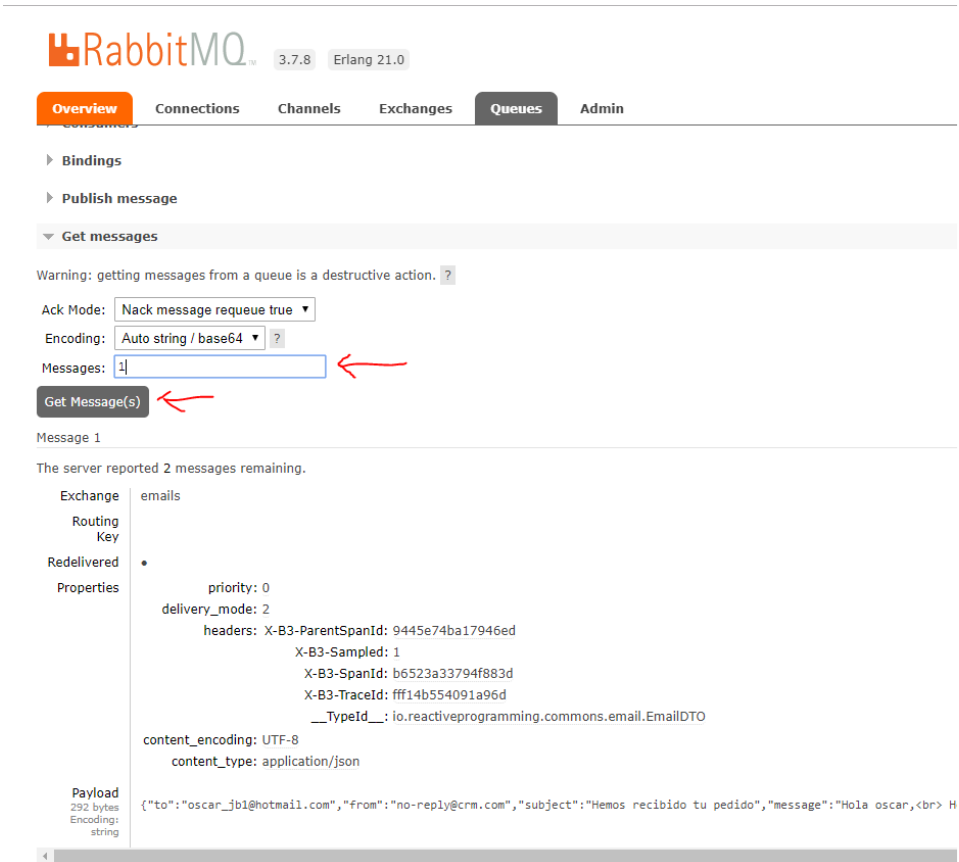


Fig 292 - Consultando los mensajes de la Queue.

En esta nueva pantalla podremos consultar los mensajes que está actualmente en la *Queue*, por lo que primero tendremos que indicar en el campo “*Messages*” el número de mensajes que queremos ver, seguido, presionamos el botón “*Get Messages*” y en la parte de abajo veremos los mensajes que tiene la *Queue*, en nuestro caso, podremos observar que es un JSON que tiene la misma estructura que el DTO *EmailDTO*.

Finalmente, y una vez que hemos analizado como funciona todo esto, encenderemos nuevamente el Microservicio *mail-sender* para ver como todos los mails comienzan a llegar nuevamente.

Una cosa más, no hemos visto cómo funciona el Microservicio *mail-sender* para recibir los mensajes, por lo que abriremos la clase *RabbitReceiver* de este mismo servicio y veremos lo siguiente:

```
1. package io.reactiveprogramming.mail.rabbit;
2.
3. import io.reactiveprogramming.commons.email.EmailDTO;
4. import io.reactiveprogramming.mail.services.MailSenderService;
5.
6. @Service
7. public class RabbitReceiver {
8.
9.     @Autowired
10.    private MailSenderService mailService;
11.
12.    @RabbitListener(queues = "emails")
13.    public void receive1(EmailDTO message) throws InterruptedException {
14.        System.out.println("newMessage => " +
15.            ReflectionToStringBuilder.toString(message,
16.                ToStringStyle.MULTI_LINE_STYLE));
17.        mailService.sendSimpleMessage(message);
18.    }
19. }
```

Nuevamente no quiero entrar en detalles técnicos de cómo funciona RabbitMQ, por lo que solo me limitaré a decir que el metadato de la línea 12 le dice a Spring boot que deberá leer todos los mensajes de la *Queue* “*emails*”, por lo que una vez que el mensaje es recuperado, es convertido nuevamente en el DTO *EmailDTO* y seguido, el email es enviado en la línea 17.

Conclusiones

Si bien no hemos podido implementar un Store and Forward más completo que persista los mensajes de ambos lados, sí que hemos demostrado las ventajas que ofrece este patrón, pues permite que el productor de los mensajes envíe el mensaje y se olvide de él, pero con la seguridad de que sus mensajes serán entregados tarde o temprano, lo que le da la seguridad de continuar con el proceso.

En la práctica, este patrón es altamente utilizado, no solo porque garantiza la entrega de los mensajes, si no que permite que el consumidor de los mensajes pueda ir extrayendo y procesando los mensajes a su propio ritmo, algo así como un embudo, donde los productores pueden enviar miles o millones de mensajes y el consumidor los irá tomando a medida que pueda procesarlos, de esta forma, evitamos asfixiar al consumidor con tantos mensajes.

Circuit Breaker

La llegada de nuevas arquitecturas como SOA o Microservicios han traído grandes ventajas, pero con ello, han surgido nuevas problemáticas que pocas veces se saben resolver con precisión, uno de estos casos, es identificar cuando un servicio ha dejado de funcionar para dejarle de enviar peticiones, pero por otro lado, identificar el fallo, reportarlo y hacer algo en consecuencia, por suerte, la patrón *Circuit Breaker* (Corto circuito) permite cortar la comunicación con un determinado servicio cuando se ha detectado que está fallado, evitando así, que el sistema continúe fallando.

Problemática

Uno de los problemas más común en arquitecturas distribuidas es que, algunos de los componentes comiencen a fallar, lo que esto puede implicar desde que dejemos de realizar ventas, hasta detener la operación por completo, por lo que tener un plan B en caso de que algo falle siempre es recomendable, sobre todo en procesos críticos.

En la práctica es común que cuando algo falla, simplemente le mostremos un error al usuario de que algo salió mal y que lo intente más tarde, pero que pasa que si eso que salió mal es precisamente una venta, ¿es acaso que estamos dispuestos a dejar pasar esa venta? Lo más seguro es que no, he intentaremos hacer lo posible por completarla, pero ¿cómo si el sistema está fallando?

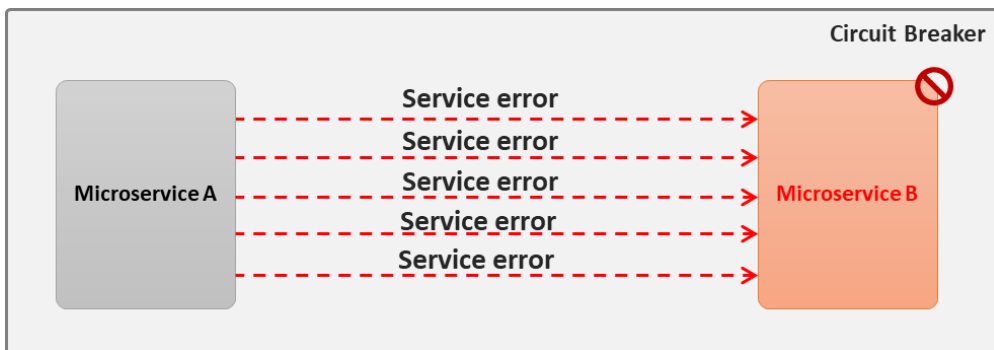


Fig 293 - Invocaciones condenadas a fallar.

Ahora bien, si este componente está fallando, cual es el sentido de seguirle enviando peticiones, si ya sabemos de antemano que va a fallar, además, si ya está fallando, el enviarle más peticiones puede hacer que agrave la situación.

Imagina un servicio que está dejando de responder y nos arroja tiempo de espera agotado, este servidor puede que dejó de responder por una carga excesiva de peticiones, lo que está haciendo que responda muy lento. Ahora bien, si sabemos que el servicio está saturado, y le seguimos enviando peticiones, eventualmente lo terminaremos de rematar, lo que puede hacer que termine callándose. Entonces, si ya sabemos que el servicio va a fallar, ¿no sería más inteligente dejar de mandarle peticiones y ejecutar un plan B en lo que el servicio se repone?

Solución

El patrón *Circuit Breaker* es muy parecido a un fusible, el cual se funde para evitar que una descarga eléctrica afecte al circuito, con la única diferencia de que el software, este fusible se puede restaurar cuando el problema allá pasado. Esto puede resultar un poco confuso, pero analicemos cómo funciona el patrón con el siguiente diagrama:

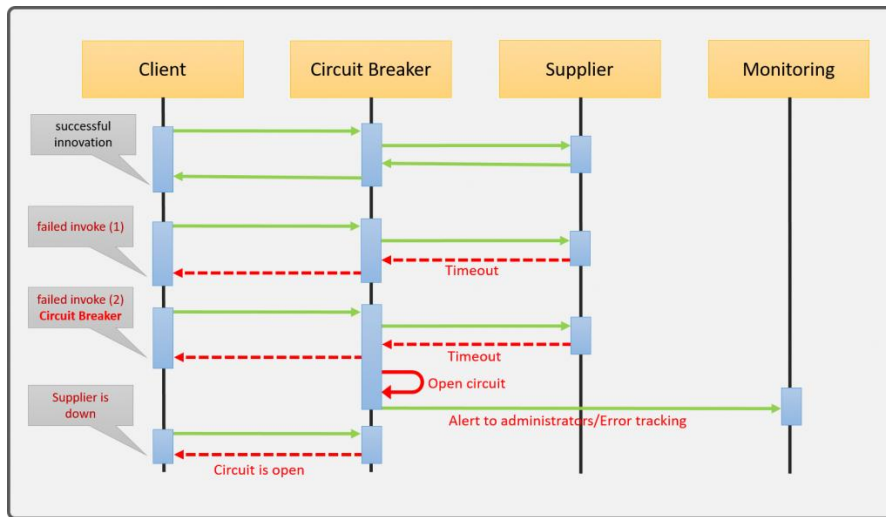


Fig 294 - Diagrama de secuencia del Circuit Breaker.

Antes de explicar cómo funciona el diagrama, debemos de comprender que son y el rol que tiene cada uno de los componentes involucrados:

- **Supplier:** representa el servicio de negocio remoto, el cual expone una serie de funcionalidad para ser consumida por el Client.
- **Client:** representa al cliente que intenta consumir los servicios del *Supplier*, sin embargo, este lo hace por medio del *Circuit Breaker*.
- **Circuit Breaker:** es el componente que se encarga de la comunicación con el *Supplier*, el cual, mantiene un registro del estado de salud del servicio que usará para abrir el circuito cuando detecte que el *Supplier* está fallando.

Una vez que entendemos esto, pasemos a explicar cómo funciona el patrón *Circuit Breaker*:

1. El Client realiza una primera llamada al *Supplier* por medio del *Circuit Breaker*.
 - 1.1.El *Circuit Breaker* determina que el *supplier* está funcionando correctamente y redirecciona la petición al *Supplier*.
 - 1.2.El *Supplier* responde correctamente y envía la respuesta al *Circuit Breaker* para que sea este quien finalmente reenvíe la respuesta al Client.

2. El *Client* envía nuevamente una petición al *Supplier* por medio del *Circuit Breaker*
 - 2.1.El *Circuit Breaker* redirecciona la petición al *Supplier*
 - 2.2.En esta ocasión el *Supplier* responde con error o lanza un *Timeout*
 - 2.3.El *Circuit Breaker* recibe la respuesta de error y toma nota del error, el cual tiene un contador que determina cuantas veces ha fallado el *Supplier* de forma simultánea. En este caso, el contador de errores se establece en 1.
 - 2.4.El *Circuit Breaker* redirecciona el error al *Client*.
3. Seguido de un tiempo, el *Client* envía nuevamente una petición al *Supplier* por medio el *Circuit Breaker*.
 - 3.1.El *Circuit Breaker* redirecciona la petición al *Supplier*
 - 3.2.El *Supplier* nuevamente responde con error o *Timeout*
 - 3.3.El *Circuit Breaker* recibe el error e incrementa el contador de errores a 2
 - 3.3.1. Dada una configuración, el *Circuit Breaker* puede determinar que 2 es el número máximo de errores simultáneos que puede tolerar antes de realiza el corto circuito y abre el circuito con el *Supplier*.
 - 3.3.2. Un paso opcional pero muy deseable es tener algún componente de monitoreo que permite notificar a los administradores en tiempo real sobre los errores para poder tomar cartas en el asunto.
 - 3.4.El *Circuit Breaker* redirecciona el error al *Client*.
4. El *Client* nuevamente envía una nueva petición al *Supplier* por medio del *Circuit Breaker*.
 - 4.1.El *Circuit Breaker* determina que el servicio no está respondiendo debido a los dos errores anteriores, por lo que retorna un error al *Client* indicándole que el *Supplier* está fuera de servicio.

Como podemos observar en la explicación anterior, el *Circuit Breaker* puede concluir que un determinado servicio (*Supplier*) está fuera de servicio al detectar que ha respondido con error de forma consecutiva un número determinado de veces y abrir el circuito para impedir que nuevas peticiones condenadas a fallar sea enviadas al *supplier*.

Abriendo el circuito

En este punto te estarás preguntando, bien, ya he abierto el circuito, pero no me puedo quedar todo el tiempo así, necesito cerrarlo nuevamente una vez que el problema se ha resuelto. Pues bien, para eso, tenemos una funcionalidad adicional, en la cual configuramos cuanto tiempo debe esperar el *Circuit Breaker* antes de mandar una solicitud al *Supplier*. En este caso, por ejemplo, podríamos esperar una hora desde que se cerró el circuito para enviar una nueva petición, si esta petición retorna exitosamente, el *Circuit Breaker* cerrará nuevamente el circuito con el *Supplier*, pero, por el contrario, si este nuevamente retorna con error o *Timeout*, entonces el tiempo de espera se reiniciará y esperará otra hora antes de mandar una nueva petición para comprobar su disponibilidad.

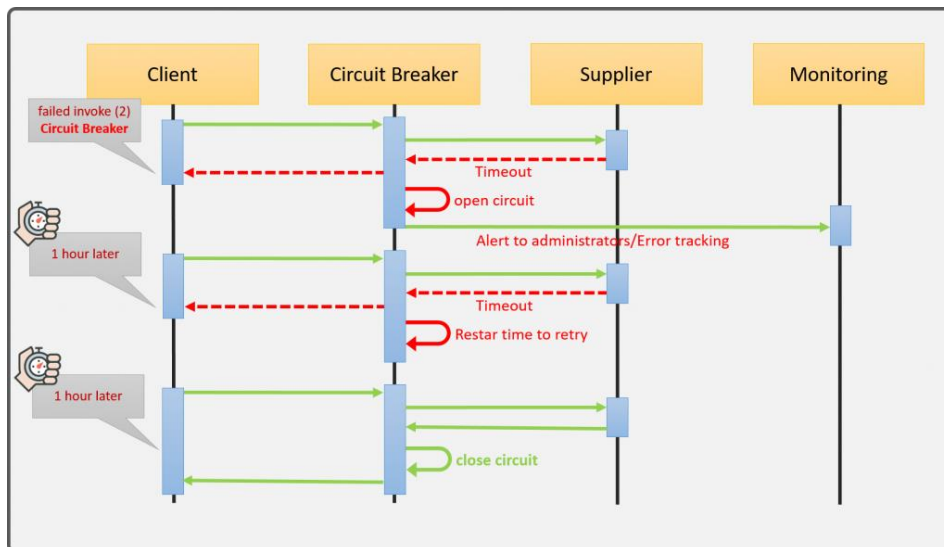


Fig 295 - Diagrama de secuencia para abrir el circuito.

Dicho lo anterior, veamos cómo quedaría el diagrama de secuencia para la reapertura del circuito:

1. En la parte superior partimos del escenario donde hemos abierto el circuito, lo cual explicamos anteriormente, por lo que nos saltaremos esta explicación. Solo asumamos que, en este punto, el circuito se encuentra abierto.
2. Una hora después de que el circuito se abrió, el *Circuit Breaker* redirecciona la siguiente petición al *Supplier* que estaba fallando para comprobar si ya se encuentra en funcionamiento.
 - 2.1. Tras redireccionar la petición al *Supplier*, esta falla nuevamente y redirecciona el error al *Circuit Breaker*.
 - 2.2. El *Circuit Breaker* al darse cuenta que sigue fallando, reinicia el contador para esperar otra hora antes de volver a intentar consumir los servicios del *Supplier*.
3. Tras una hora desde el paso anterior, el *Circuit Breaker* redirecciona una nueva petición al *Supplier* para comprobar si ya se encuentra en funcionamiento.
4. En este caso, el *Supplier* ya se encuentra operando correctamente, por lo que recibe la petición, la procesa y retornar exitosamente.
5. El *Circuit Breaker* detecta la respuesta exitosa del *Supplier* y cierra nuevamente el circuito.

Ciclo de vida del Circuit Breaker

El *Circuit Breaker* se puede ver como una máquina de estados, la cual puede pasar por 3 estados diferentes, y cada estado afectará la forma en que funciona, los estados son:

- **Closed:** Es el estado inicial, el cual indica que el servicio destino está respondiendo correctamente. En caso de alcanzar un umbral de errores, pasará al estado *Open*.
- **Open:** Este estado indica que el servicio destino está fallando, por lo que toda invocación regresará un error de inmediato. Tras un tiempo de espera, pasará a estado *Half-Open*.
- **Half-Open:** (medio abierto): Tras un tiempo en estado *Open*, el componente pase a *Half-Open*, lo que indica que puede recibir una pequeña cantidad de solicitudes para validar si el servicio está nuevamente activo. Si las solicitudes pasan exitosamente, el componente pasa a *Closed*, en otro caso, regresa a *Open*.

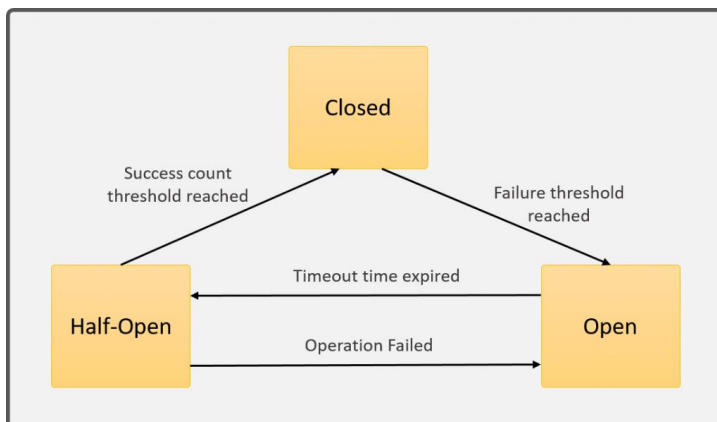


Fig 296 - Ciclo de vida de Circuit Breaker.

Circuit Breaker en el mundo real

Uno de los procesos más críticos de nuestra aplicación es el proceso de ventas, ya que es la forma en que obtenemos los ingresos, por lo que tenemos que hacer todo lo posible por procesar todas las ventas, incluso, si el sistema está fallando en ese momento, por ese motivo, vamos a implementar un *Circuit Breaker* en nuestro proceso de ventas, de tal forma que si este comienza a fallar, abriremos el circuito y realizaremos un Plan B para no perder la venta.

Nuestra aplicación puede procesar las ventas de dos formas, la primera y más simple, es cuando todo funciona correctamente, en tal caso, procesamos la venta de forma síncrona y confirmamos el pedido mediante un Email, dicho mail es el que hemos estado viendo todo este tiempo:

Hemos recibido tu pedido



oscarblancarte3@gmail.com <oscarblancarte3@gmail.com>

09:21 p. m.



Hola oscar,

Hemos recibido tu pedido **#0c658a1d-beca-4756-8984-3cb772eab2d9**, tambien hemos confirmado tu pago, por lo que estarás recibiendo tus productos muy pronto

Fig 297 - Confirmación exitosa de la venta.

Observa como el correo nos indica el número de pedido y nos dice que el pago de ha realizado exitosamente, he incluso, le dices que si producto llegará pronto.

Ahora bien, que pasaría si algún error ocurriera durante la compra, lo más seguro es que simplemente perdamos la venta, pues le diremos al cliente que hay un error y que intente más tarde, pero eso sería estúpido, pues estaríamos dejando ir a un cliente que probablemente ya no regrese, incluso, quedará con la duda de si le realizaron el cargo a su tarjeta, por lo que seguramente esperará al día siguiente para verificar con su banco, y si tenemos suerte regresará el próximo día a termina la compra.

Entonces, si el cliente ya introdujo todos los datos para hacer la compra, incluso sus datos de la tarjeta, solo basta que el sistema esté nuevamente operativo para terminar de procesar la compra, por lo que utilizaremos el *Circuit Breaker* para abrir el circuito y mandar la compra al plan B.

El plan B consiste en persistir la solicitud de la compra en RabbitMQ y procesarla cuando el error se esté solucionado, y en el inter, le mandamos el siguiente correo:

Hemos recibido tu pedido



03:25 p. m.



Para

Hola oscar,

Hemos recibido tu pedido #002cb8e5-75c9-490a-917a-b8e69b60cfb4, en este momento tu pedido está siendo procesado y una vez validado te contactaremos para darle seguimiento a tu pedido

Fig 298 - Correo del Circuit Breaker.

Observa que en este correo cambiamos un poco la gramática, pues no le estamos confirmando la compra, en su lugar, le estamos diciendo que estamos procesando su compra. En este punto el cliente creerá que este es el flujo normal de compra y no se imaginará que por dentro, nuestra aplicación ha fallado, lo que le da la confianza de que su pedido realmente está siendo atendido.

Dicho lo anterior, pasaremos a explicar cómo es que hemos implementado el *Circuit Breaker* dentro de nuestro proyecto. Para comenzar, debo decir que para implementar este patrón hemos utilizado Hystrix, otra tecnología de Netflix OSS (Open Source Software), el cual es una herramienta muy potente que nos permite implementar el patrón de una forma muy simple, e incluso, nos permite configurar una acción en caso de que el circuito de abra.

Para ver cómo funciona, regresaremos al ya conocido método *createOrder* de la clase *OrderService* del Microservicio *crm-api*, pero esta vez no nos centraremos en el cuerpo del método, si no en su definición:

```
1. @HystrixCommand(  
2.     fallbackMethod="queueOrder",  
3.     ignoreExceptions= {ValidateServiceException.class})  
4. public SaleOrderDTO createOrder(NewOrderDTO order)  
5.     throws ValidateServiceException, GenericServiceException {  
6.     ...  
7. }
```

Quiero que observes que el método *createOrder* tiene el metadato *@HystrixCommand*, con el cual le estamos diciendo al método que debemos habilitar el *Circuit Breaker*, pero, además, le estamos diciendo por medio de la propiedad *fallbackMethod*, el método que debe de ejecutar en caso de que la ejecución de este método falle o el circuito este abierto, lo que quiere decir que debemos de tener un método llamado *queueOrder* en esta misma clase.

```
1. private SaleOrderDTO queueOrder(NewOrderDTO order)  
2.     throws ValidateServiceException, GenericServiceException {  
3.     tracer.currentSpan().tag("queue.order",  
4.         "Queue order from the customer " + order.getCustomerName());  
5.     logger.error("Queue order from the customer " +  
6.         order.getCustomerName());  
7.     try {  
8.         order.setRefNumber(UUID.randomUUID().toString());  
9.         sender.send("newOrders", null, order);  
10.   
11.         SaleOrderDTO response = new SaleOrderDTO();  
12.         response.setQueued(true);  
13.         response.setRefNumber(order.getRefNumber());  
14.   
15.         EmailDTO mail = new EmailDTO();  
16.         mail.setFrom("no-reply@crm.com");  
17.         mail.setSubject("Hemos recibido tu pedido");
```

```

18.         mail.setTo(order.getCustomerEmail());
19.         mail.setMessage(String.format("Hola %s,<br> Hemos recibido " +
20.             "tu pedido <strong>#%s</strong>, en este momento tu pedido " +
21.             "está siendo procesado y una vez validado te contactaremos " +
22.             "para darle seguimiento a tu pedido",order.getCustomerName(),
23.             order.getRefNumber()));
24.
25.         sender.send( "emails", null, mail);
26.
27.         return response;
28.     } catch (Exception e) {
29.         logger.error(e.getMessage());
30.         tracer.currentSpan().tag("queue.order.error",
31.             "Error to queue a new order from the customer " +
32.             order.getCustomerName());
33.         throw new GenericServiceException(e.getMessage(),e);
34.     }
35. }

```

Antes de comenzar a analizar el método *queueOrder*, quiero que prestes atención en que el parámetro de entrar y la respuesta es idéntica al método *createOrder*, esto se debe que si cuando el circuito se abre, este método será ejecutado en su lugar, y la respuesta será replicada por el *Circuit Breaker* al consumir del servicio, por lo tanto, el cliente espera una respuesta compatible con la esperada, ya que recuerda que para el cliente es transparente si tenemos o no un *Circuit Breaker*. Entonces, si el método *createOrder* falla, el parámetro de este método (*NewOrderDTO*) será enviado al método *queueOrder*, luego, la respuesta de este último será retornada al cliente.

Algo a tomar en cuenta es que el método *queueOrder* se puede ejecutar por dos razones, la primera es que, el servicio *createOrder* falle por arriba del umbral de fallas permitido, lo que hará que el circuito de habrá y todas las llamadas sean redireccionadas al método *queueOrder* de forma automática sin ejecutar el cuerpo del método *createOrder*, el segundo motivo es que el método *createOrder* termine en excepción, es decir que falle.

Si regresamos a la definición del método *createOrder*, podrás ver que tiene la propiedad *ignoreExceptions*, en esta propiedad le decimos que excepciones no deberán ser tomadas en cuenta para el *Circuit Breaker*, ya que no todas las excepciones significan que el método este fallando, por ejemplo, la excepción *ValidateServiceException* la utilizamos para lanzar error de validación de datos,

lo que quiere decir que el servicio está funcionando correctamente, aunque termine en error.

Un ejemplo claro de esto es, cuando los datos de la tarjeta de crédito son incorrectos, en tal caso lanzamos la excepción *ValidateServiceException*, pero esta excepción se debe a los datos, no a que el servicio este fallando, por ese motivo debemos de indicarle al *Circuit Breaker* que excepciones deberá de ignorar, por tal motivo indicamos que esta excepción deberá ser ignorada en el atributo *ignoreExceptions*.

Abrir el circuito

Ya explicado cómo hemos implementado el patrón, vamos a simular algunos errores para ver cómo se abre el circuito y se llama al *fallbackMethod*, para esto vamos a lanzar una excepción dentro del método *createOrder*:

```
1. @HystrixCommand(  
2.     fallbackMethod="queueOrder",  
3.     ignoreExceptions= {ValidateServiceException.class})  
4. public SaleOrderDTO createOrder(NewOrderDTO order)  
5.     throws ValidateServiceException, GenericServiceException {  
6.     logger.info("New order request ==>");  
7.     try {  
8.         if(order.getId() == null) {  
9.             throw new GenericServiceException("Dummy error");  
10.        }  
11.        ...  
12.    } catch(ValidateServiceException e) {  
13.        e.printStackTrace();  
14.        throw e;  
15.    } catch (Exception e) {  
16.        e.printStackTrace();  
17.        throw new GenericServiceException(e.getMessage(),e);  
18.    }  
19. }
```

Lo primero que observe dentro del método *createOrder* es que tiene un *logger*, que imprimirá el texto "New order request ==>", este mensaje es importante porque demostrará el orden de ejecución. Finalmente, en las líneas 8 a 10 vamos a lanzar una excepción, la cual se lanzará en todas las ejecuciones, pues el ID de la orden siempre será null para la creación de una nueva orden.

Si nos pasamos al método *queueOrder*, también tenemos un *Logger* que imprimirá la leyenda "Queue order from the customer + nombre del usuario", esto también es importante, porque demostrará el orden de ejecución.

```
1. private SaleOrderDTO queueOrder(NewOrderDTO order)
2.     throws ValidateServiceException, GenericServiceException{
3.     tracer.currentSpan().tag("queue.order",
4.         "Queue order from the customer " + order.getCustomerName());
5.     logger.error("Queue order from the customer " + order.getCustomerName());
6.
7.     ...
8. }
```

El siguiente paso será crear una nueva venta en la aplicación, lo que nos arrojará un mensaje como el siguiente:

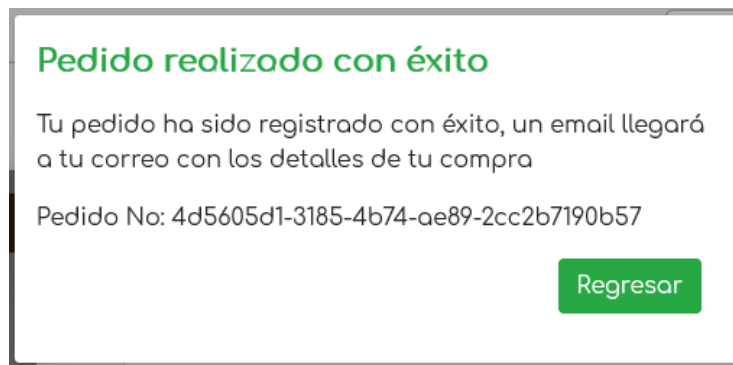


Fig 299 - Pedido registrado con éxito.

Observa que el mensaje que lanza la aplicación al usuario hace referencia a que el pedido se ha registro y recibiremos un email con el detalle de la compra, y en ningún momento sospechas que algo salió mal, pero si nos vamos a los logs, veremos el detalle de la traza:

```
1. 2019-12-18 03:10:04.084 INFO [crm,53af4e4cd5ac77a8,c5447d40f20031d4,true] 55596 --- [-OrderService-
2. 3] i.r.crm.api.services.OrderService : New order request ==>
3. io.reactiveprogramming.commons.exceptions.GenericServiceException: Dummy error
4. at io.reactiveprogramming.crm.api.services.OrderService.createOrder(OrderService.java:106)
5. at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
6. at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
7. at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
8. at java.lang.reflect.Method.invoke(Unknown Source)
```

```

8.    at com.netflix.hystrix.contrib.javanica.command.MethodExecutionAction.execute(MethodExecutionAction.java
:116)
9.    at com.netflix.hystrix.contrib.javanica.command.MethodExecutionAction.executeWithArgs(MethodExecutionAct
ion.java:93)
10.   at com.netflix.hystrix.contrib.javanica.command.MethodExecutionAction.execute(MethodExecutionAction.java
:78)
11.   at com.netflix.hystrix.contrib.javanica.command.GenericCommand$1.execute(GenericCommand.java:48)
12.   at com.netflix.hystrix.contrib.javanica.command.AbstractHystrixCommand.process(AbstractHystrixCommand.ja
va:145)
13.   at com.netflix.hystrix.contrib.javanica.command.GenericCommand.run(GenericCommand.java:45)
14.   at com.netflix.hystrix.HystrixCommand$2.call(HystrixCommand.java:302)
15.   at com.netflix.hystrix.HystrixCommand$2.call(HystrixCommand.java:298)
16.   at rx.internal.operators.OnSubscribeDefer.call(OnSubscribeDefer.java:46)
17.   at rx.internal.operators.OnSubscribeDefer.call(OnSubscribeDefer.java:35)
18.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:48)
19.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:30)
20.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:48)
21.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:30)
22.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:48)
23.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:30)
24.   at rx.Observable.unsafeSubscribe(Observable.java:10327)
25.   at rx.internal.operators.OnSubscribeDefer.call(OnSubscribeDefer.java:51)
26.   at rx.internal.operators.OnSubscribeDefer.call(OnSubscribeDefer.java:35)
27.   at rx.Observable.unsafeSubscribe(Observable.java:10327)
28.   at rx.internal.operators.OnSubscribeDoOnEach.call(OnSubscribeDoOnEach.java:41)
29.   at rx.internal.operators.OnSubscribeDoOnEach.call(OnSubscribeDoOnEach.java:30)
30.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:48)
31.   at rx.internal.operators.OnSubscribeLift.call(OnSubscribeLift.java:30)
32.   at rx.Observable.unsafeSubscribe(Observable.java:10327)
33.   at rx.internal.operators.OperatorSubscribeOn$SubscribeOnSubscriber.call(OperatorSubscribeOn.java:100)
34.   at com.netflix.hystrix.strategy.concurrency.HystrixContextSchedulerAction$1.call(HystrixContextSchedulerAc
tion.java:56)
35.   at com.netflix.hystrix.strategy.concurrency.HystrixContextSchedulerAction$1.call(HystrixContextSchedulerAc
tion.java:47)
36.   at org.springframework.cloud.sleuth.instrument.async.TraceCallable.call(TraceCallable.java:69)
37.   at com.netflix.hystrix.strategy.concurrency.HystrixContextSchedulerAction.call(HystrixContextSchedulerActi
on.java:69)
38.   at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.java:55)
39.   at java.util.concurrent.Executors$RunnableAdapter.call(Unknown Source)
40.   at java.util.concurrent.FutureTask.run(Unknown Source)
41.   at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
42.   at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
43.   at java.lang.Thread.run(Unknown Source)
44. 2019-12-18 03:10:04.108 ERROR [crm,53af4e4cd5ac77a8,c5447d40f20031d4,true] 55596 --- [-OrderService-
3] i.r.crm.api.services.OrderService      : Queue order from the customer oscar

```

Este log puede resultar un poco caótico por el montón de líneas, pero lo resumiremos a 3 líneas. En la primera línea podemos observar el texto “New order request ==>”, el cual indica que está dentro del cuerpo del método *createOrder*.

Este mensaje nos indica que el método si se ejecutó, pero luego, en el la línea 2 del log podemos ver la leyenda “*Dummy error*”, la excepción que lanzamos a propósito, y como consecuencia, esto hace que el método *fallback* se ejecute, es decir, el método *queueOrder*, y es por eso que podemos ver el mensaje “*Queue order from the customer oscar*” es la línea 44. Este log nos demuestra que lo primero que se ejecutó fue el método *createOrder* y luego el método *queueOrder* tras lanzarse la excepción.

Lo que nos resta es verificar en RabbitMQ si la orden realmente se guardó en la cola, por lo que tendremos que ir a la consola de administración de RabbitMQ y abrir la *queue newOrders*:

OverviewConnectionsChannelsExchangesQueuesAdmin

Queues

▼ All queues (2)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview			Messages			Message rates				+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver	/ get	ack	
emails	D HA	idle	1	0	1	0.00/s		0.00/s	0.00/s	
newOrders	D HA	idle	3	0	3	0.00/s		0.00/s	0.00/s	

▼ Add a new queue

Fig 300 - Analizando la queue newOrders.

Podremos ver que la columna Total se incrementa con cada orden que pasa por el *fallback*, incluso, si abrimos el detalle de la *Queue*, podremos ver el request completo de la orden:

RabbitMQ 3.7.8 Erlang 21.0

Overview Connections Channels Exchanges **Queues** Admin

Get messages

Warning: getting messages from a queue is a destructive action. ?

Ack Mode:

Encoding:

Messages:

Get Message(s)

Message 1

The server reported 2 messages remaining.

Exchange	newOrders
Routing Key	
Redelivered	•
Properties	priority: 0 delivery_mode: 2 headers: X-B3-ParentSpanId: 3819e0ccafdeb164 X-B3-Sampled: 1 X-B3-SpanId: c04763216c5c3a84 X-B3-TraceId: 11cc26bdfeaf50c4 __TypeId__: io.reactiveprogramming.crm.dto.NewOrderDTO content_encoding: UTF-8 content_type: application/json
Payload	335 bytes Encoding: string <pre>{ "id": null, "customerName": "oscar", "customerEmail": "oscar_jb1@hotmail.com", "refNumber": "002cb8e5-75c9-490a-917a-b8e69b60cfb4",</pre>

Fig 301 - Analizando la Queue newOrders.

Si bien, en este punto hemos demostrado como el método *fallback* es ejecutado cuando un error se produce en el método *createOrder*, todavía tenemos algo que analizar, pues en este caso, el método *fallback* se está ejecutando como si fuera un try catch, donde el catch sería el método *queueOrder*, por lo que no queda tan obvio la ventaja de hacerlo así, pues bien, el detalle es que hasta este momento no hemos explotado el *Circuit Breaker*, pues el objetivo de este es evitar que el método que falle se ejecute por completo, es decir, que en lugar de pasar por el método *createOrder* y luego brincar al *queueOrder*, el objetivo es ir directo al método *queueOrder*.

Para que el circuito se habrá realmente es necesario ejecutar el servicio repetidas veces en un tiempo corto de tiempo, por lo que tenemos dos opciones, o tenemos muchos usuarios conectados haciendo fallar al servicio o tendremos que simular las llamadas al API para simular varias invocaciones simultaneas. En nuestro caso,

solo podemos inclinarnos por la segunda opción, por lo que vamos a utilizar la herramienta Runner que viene incluida dentro de Postman.

Para crear el Runner tenemos que tener un request preparado en Postman, por lo que vamos a crear un nuevo request:

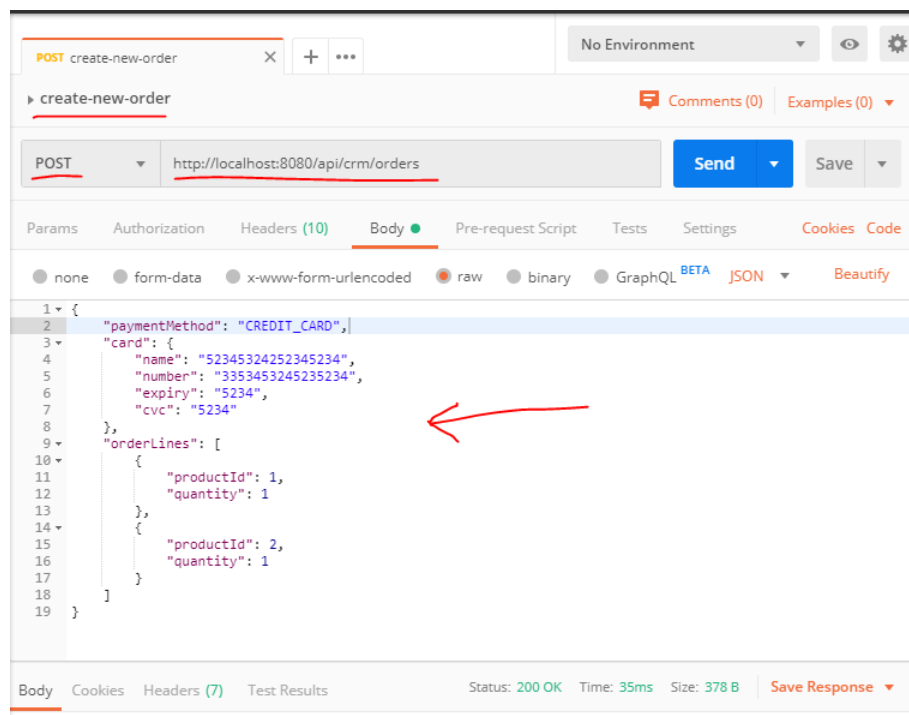


Fig 302 - Preparando un request en Postman.

Ha este request lo hemos llamado "create-new-order", y hemos definido que realizaremos una llamada al a URL `http://localhost:8080/api/crm/orders` por el método POST, también hemos agregado el header `Authorization` con el Token y el payload es el siguiente:

```
1. {
2.   "paymentMethod": "CREDIT_CARD",
3.   "card": {
4.     "name": "52345324252345234",
5.     "number": "3353453245235234",
6.     "expiry": "5234",
7.     "cvc": "5234"
8.   },
9.   "orderLines": [
```

```

10.    {
11.        "productId": 1,
12.        "quantity": 1
13.    },
14.    {
15.        "productId": 2,
16.        "quantity": 1
17.    }
18. ]
19. }

```

Ya con el request creado, nos vamos a la sección Runner de Postman:

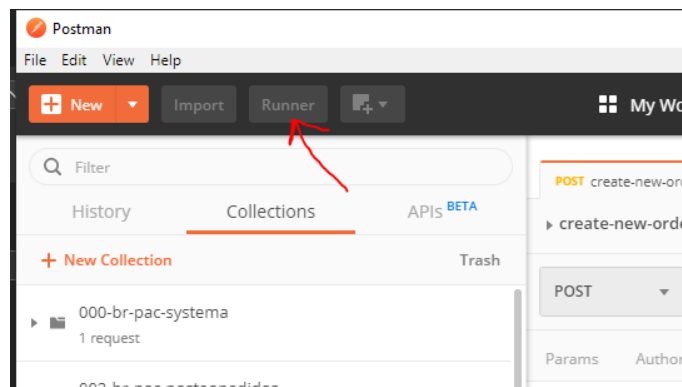


Fig 303 - Abriendo el Runner.

En la siguiente pantalla tendremos que seleccionar el request que acabamos de crear e indicar que queremos 30 request seguidos, finalmente presionar el botón "Run Architecture":

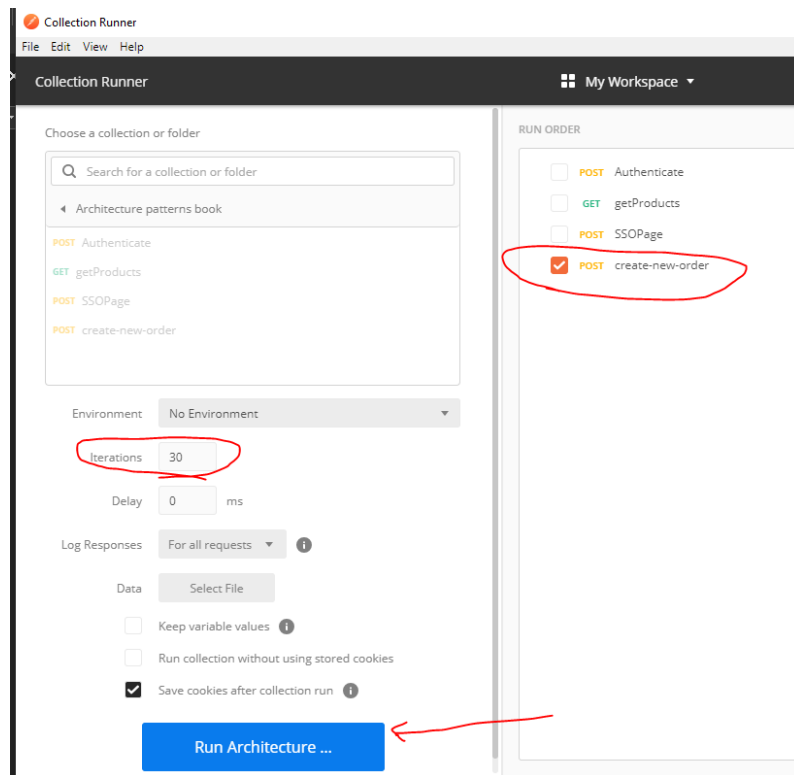


Fig 304 - Preparando el Runner.

Una vez que ejecutemos el Runner, 30 request serán lanzado en simultaneo al servicio de creación de ordenes, por lo que hora solo nos queda revisar el log para comprender que está pasando.

Antes que nada, el log es muy largo como para ponerlo aquí, por lo que nos centraremos en lo más relevante. Lo primero que vas a ver es que los request van a entrar al servicio `createOrder` y después se pasarán al método `queueOrder`, pero ya para el final del log, podrás ver algo parecido a lo siguiente:

```

1. 2019-12-19 13:20:20.683 ERROR [crm,8273a8e20fc33f63,667839138dfb1383,true] 55596 --- [-nio-811-exec-
4] i.r.crm.api.services.OrderService : Queue order from the customer oscar
2. 2019-12-19 13:20:20.792 ERROR [crm,253a7b1598a90aac,70e606d961f9704e,true] 55596 --- [-nio-811-exec-
5] i.r.crm.api.services.OrderService : Queue order from the customer oscar
3. 2019-12-19 13:20:20.891 ERROR [crm,26486c57799b3d66,a7bf934ca1459e45,true] 55596 --- [-nio-811-exec-
6] i.r.crm.api.services.OrderService : Queue order from the customer oscar
4. 2019-12-19 13:20:20.982 ERROR [crm,3e46768bbe5265fc,7e20c29bbd32a00f,true] 55596 --- [-nio-811-exec-
7] i.r.crm.api.services.OrderService : Queue order from the customer oscar

```

```
5. 2019-12-19 13:20:21.068 ERROR [crm,a883d71649e37c23,b5104ebae094d605,true] 55596 --- [-nio-811-exec-8] i.r.crm.api.services.OrderService : Queue order from the customer oscar
6. 2019-12-19 13:20:21.157 ERROR [crm,58fe71d84a129140,03c7a83dd43f855c,true] 55596 --- [-nio-811-exec-9] i.r.crm.api.services.OrderService : Queue order from the customer oscar
7. 2019-12-19 13:20:21.248 ERROR [crm,d5b4e36a3636d9c6,5b55e493424c1f6a,true] 55596 --- [nio-811-exec-10] i.r.crm.api.services.OrderService : Queue order from the customer oscar
8. 2019-12-19 13:20:21.336 ERROR [crm,b7e7fa96cf44748e,987dc187df52581f,true] 55596 --- [-nio-811-exec-1] i.r.crm.api.services.OrderService : Queue order from the customer oscar
```

Lo que estás viendo en log anterior es como el método *queueOrder* se está ejecutando directamente sin pasar por el método *createOrder*, lo que indica que el circuito se ha abierto por tantos errores y ahora el método *createOrder* ya ni siquiera se intenta ejecutar.

Este comportamiento es diferente a un Try Catch tradicional, ya que en un Try se intenta ejecutar el código y en caso de falla, se va al catch, en nuestro caso, es como si brincáramos directamente al catch sin pasar por el try. Esto tiene grandes ventajas, pues impide que el código del método que va a fallar se ejecute, lo que impide que terminemos de rematar al servicio que está fallando o dejemos nuestro sistema en un estado inconsistente.

Finalmente, si dejamos pasar un par de minutos y ejecutamos nuevamente el servicio, veremos nuevamente que se intentará ejecutar el método *createOrder*, pues estará en estado semi abierto, lo que indica que deja pasar algunas peticiones para ver si el servicio está nuevamente funcionando.

Conclusiones

Como hemos analizado, utilizar el patrón *Circuit Breaker* impide que inundemos nuestra aplicación con una gran cantidad de solicitudes que sabemos de antemano que van a fallar, por lo que, en lugar de eso, evitamos la llamada e incluso, podemos hacer una acción en consecuencia.

Este patrón es muy utilizado en procesos críticos, donde no podemos darnos el lujo de cancelar la operación y regresar un error al cliente, sino al contrario, tratamos de proporcionar el servicio, aunque tengamos de una forma diferente.

Log aggregation

Tener una traza clara de la ejecución de la aplicación es sin duda una de las cosas más importantes a tener en cuenta cuando diseñamos la aplicación, pues de eso depende encontrar errores y poder dar soluciones más rápidas a los clientes, pero sobre todo, nos permite saber con claridad que es lo que está haciendo y por donde está pasando la aplicación.

Problemática

Uno de los problemas más frecuentes cuando hablamos de arquitecturas distribuidas es obtener una trazabilidad de la ejecución de un servicio, ya que en este tipo de arquitecturas la ejecución de un servicio pasa por múltiples servicios, lo que hace complicado comprender lo que está pasando y tener un log detallados de lo que está pasando.

Además, en el caso de arquitecturas como Microservicios, es común tener múltiples instancias de un mismo componente, lo que hace que recuperar la traza de ejecución sea un verdadero dolor de cabeza.

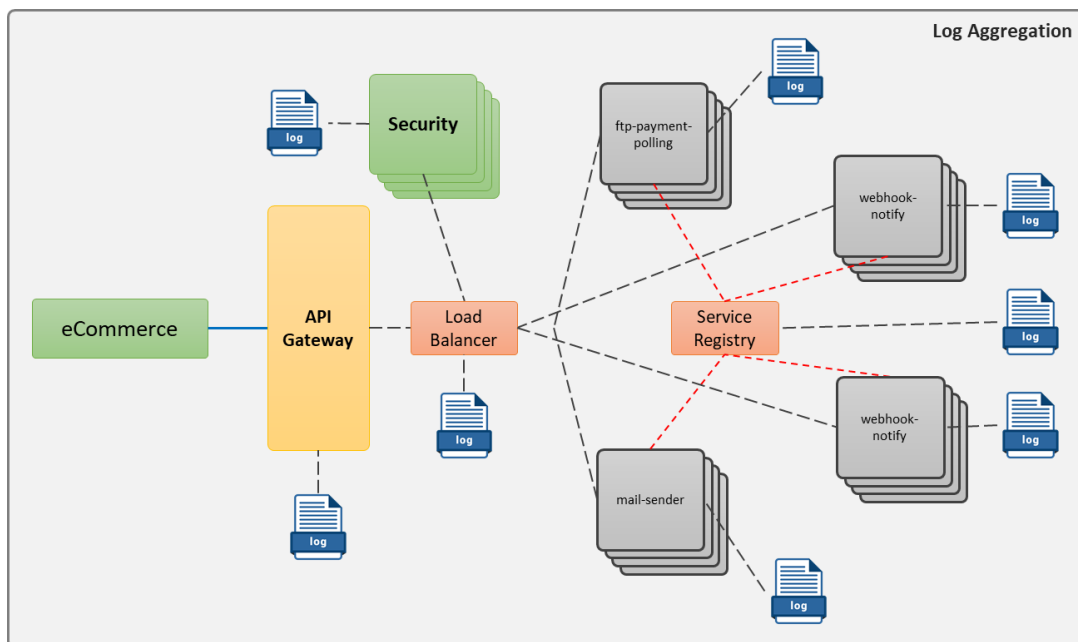


Fig 305 - Logs distribuidos.

En la imagen anterior podemos ver más claramente el problema que mencionaba anteriormente, en el cual, cada servicio va dejando de forma local un log con el registro de la ejecución, por lo que si queremos tener una traza completa de un proceso, necesitamos recuperar el log de cada Microservicios, buscar por medio de la hora el log que corresponda a la ejecución y luego unificarlo, lo que puede ser una tarea verdaderamente complicado, sumado a esto, si tenemos múltiples instancias de cada servicio, es necesario abrir el log de todas las instancias para saber cuál fue la que proceso nuestra petición, pues si recordamos, el balanceo de cargas hace que sea imposible saber cuál fue la instancia de cada Microservicio que proceso nuestra solicitud.

Solo para darnos una idea de lo complicado que puede ser esto, analicemos la ejecución necesaria para crear una nueva orden desde la aplicación eCommerce:

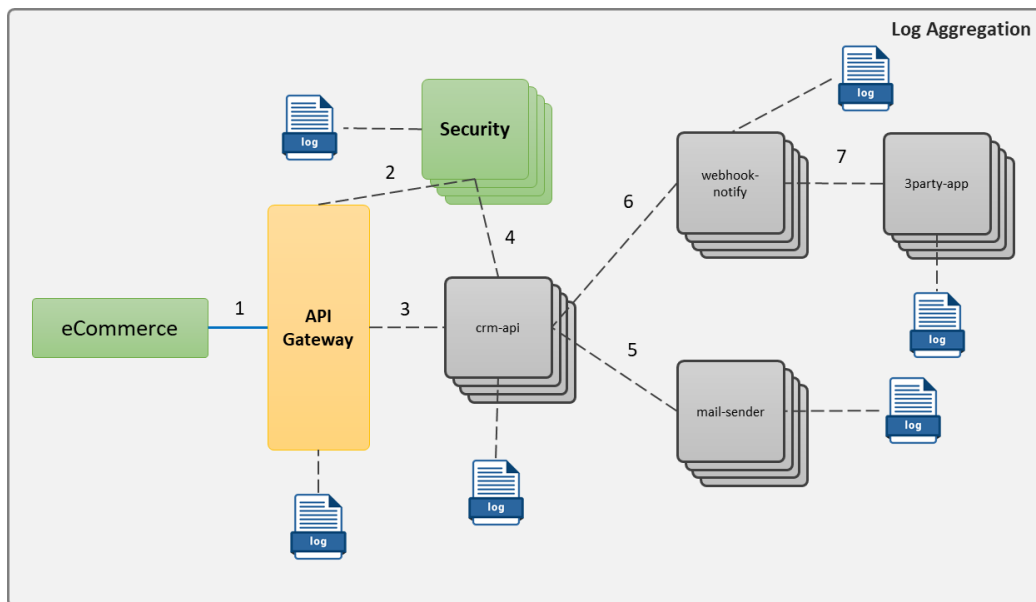


Fig 306 - Flujo de ejecución para la creación de una orden.

En la imagen podemos apreciar de forma enumerada los Microservicios que es necesario ejecutar para crear una sola orden, lo que implica que cada uno de estos cree un log por separado, si a esto le sumamos que cada Microservicios podría tener 3, 6, 10 o más instancias, nos llevaría un escenario caótico, donde analizar la traza completa de ejecución de una sola ejecución nos podría llevar a analizar decenas o cientos de logs, lo cual es una verdadera locura.

Otro de los problemas que se presenta es que, aunque tengamos ya todos los archivos concentrados en una sola carpeta para analizarlos, es complicado saber que fragmento del log corresponde con una ejecución concreta, lo que nos llevaría a tener que filtrar los archivos por rangos de fecha y hora para tener una aproximación de cómo se realizó la ejecución.

Solución

Para solucionar este problema tenemos el patrón Log aggregation, el cual nos permite que, por medio de un componente externo, podamos concentrando los logs en una sola fuente de datos que podemos consultar más adelante sin la necesidad de tener acceso físico a los servidores y sin importar cuantas instancias de cada componente tengamos.

Lo primero que tenemos que hacer para solucionar este problema es identificar qué información es relevante para el monitoreo de la aplicación, ya que la gran mayoría de la información que enviamos a los logs es basura, por lo que debemos ser cuidadosos para seleccionar que datos son necesario y cuáles no, la regla básicamente es que mandemos lo suficiente para comprender que está pasando, pero tampoco de menos que perdamos la trazabilidad. Una vez identificada la información a enviar, solo hay que enviarla al aggregator para que comience a persistir la información.

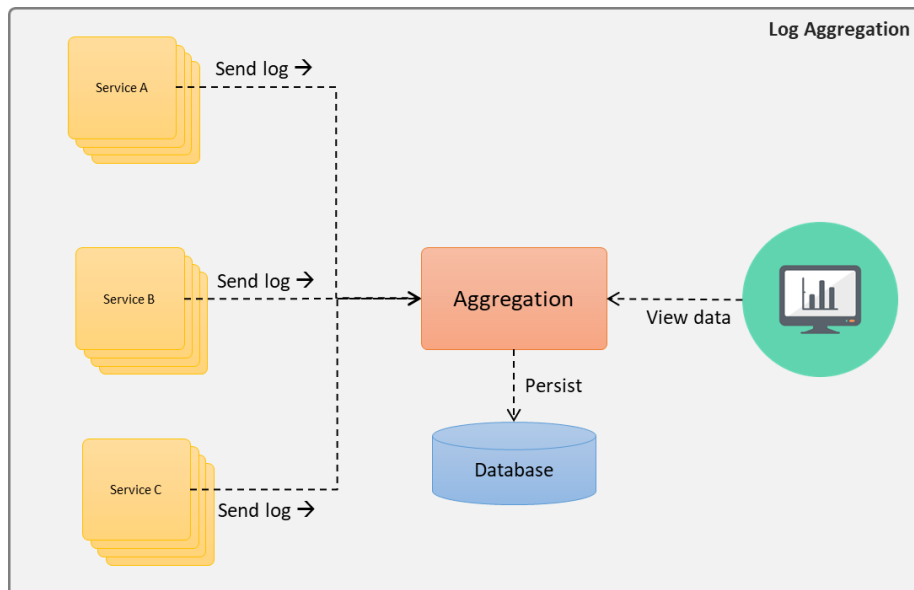


Fig 307 - Log aggregation

En la imagen anterior podemos apreciar el rol central que juega el *Log Aggregation* en la arquitectura, pues todos los Microservicios van enviando detalles de la ejecución a medida que son ejecutados.

Algo a tomar en cuenta es que para que el log sea efectivo, es importante identificar cada ejecución de cada Microservicio por separado, pero más importante aún es, identificar todos los Microservicios que se ejecutaron como parte de una sola invocación, por lo que cada Microservicio registra el log con dos valores que ayudan a rastrear la ejecución, el primero es un identificador global de la transacción, el cual comparte todos los Microservicios que se ejecutan como parte de la misma llamada, y el segundo valor es un identificador único por operación, de esta forma, podemos saber todo el log producido por una sola operación y el log producido por toda la transacción.

Una vez que tenemos toda la información concreada en un solo lugar, es posible consultarlas de forma gráfica, lo que permite que cualquier desarrollador puedan consultarla sin tener que tener acceso a los servidores o tener que solicitar a un administrador que le envíe los logs.

La forma en que mandamos la traza al *Log Aggregation* puede variar, pues hay productos ya desarrollados que permiten mandar los logs de forma síncrona por HTTP, otros permite el envío de información mediante colas de mensajes, otro, permiten enviar los archivos de log como tal, por lo que no existe una única forma de hacerlo, de la misma forma, la forma en que la información se guarda en el *Log Aggregation* también puede variar, hay algunos que lo guardan en bases de datos relacionales, otros en NoSQL, e incluso, hay casos que son en Memoria.

Formato de la información

Uno de los aspectos más importantes cuando trabajamos con logs distribuidos es el formato de la información, ya que es común que cada componente utilice librerías y formatos diferentes para almacenar el log, lo que causa un conflicto importante al momento de realizar una traza centralizada, es por ello, que parte del proceso de enviar los logs a *log Aggregation* consiste en convertir el log específico en un formato estandarizado, de esta forma, el *Log Aggregation* solo recibirá los logs en un formato específico.

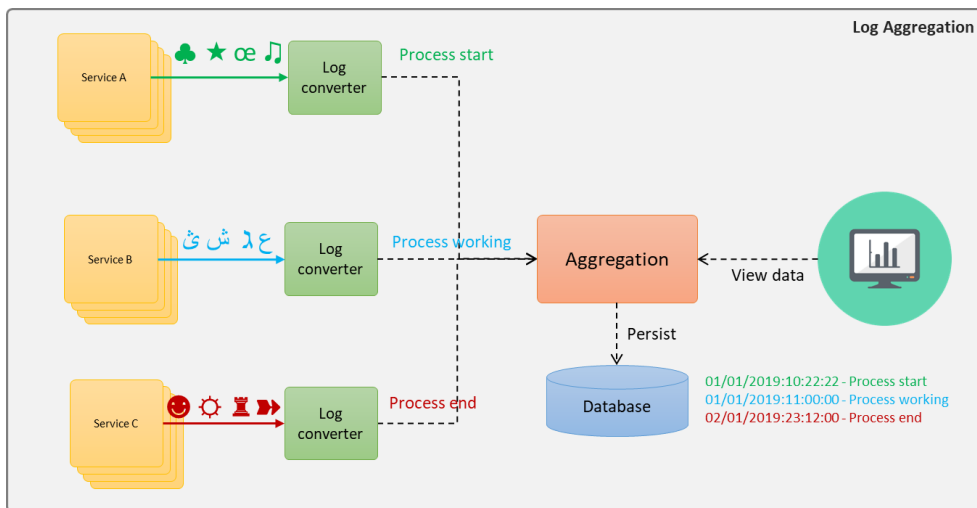


Fig 308 - Homogenizando el log.

Lo más normal es que el proveedor del *Log Aggregation* proporcionen herramientas para hacer esta conversión, por lo que hace todavía más fácil para el programador integrarlo con estas herramientas. Un ejemplo real de esto que menciono es Elastic Search, una de las plataformas más utilizadas para logs distribuidos, el cual funciona con la siguiente arquitectura:



Fig 309 - Arquitectura de Elastic Search.

Elastic Seach cuenta con varios productos que en su conjunto brinda una plataforma registro de logs distribuidos, mediante el cual es posible recopilar datos de nuestras aplicaciones en tiempo real y ayudar a monitorear las aplicaciones, por ejemplo, Beats es un producto que permite recuperar los datos de un servidor o aplicación para enviarla a Logstash. Logstash es el API para convertir los logs a un formato estándar y enviarlo a Elastic Search, Elastic Search es el producto principal, donde se almacena la información, y se realiza la búsqueda de toda la información, finalmente, Kibana es una aplicación que nos permite visualizar los datos de una forma más amigable.

Log aggregation en el mundo real

En este punto del libro ya te habrás dado cuenta que seguir la trazabilidad de la ejecución de un servicio es muy complicado, ya que nos tenemos que estar moviendo entre la terminal de cada Microservicio para ver cada parte del log. Si bien tenemos la ventaja de que siguen el mismo formato, todavía tenemos el problema de que una ejecución está distribuida entre el log de varios Microservicios, por lo que para comprender que es lo que está pasando es necesario analizar cada log por separado y luego tratar de interpretar que está pasando por medio de la fecha y hora de registro, lo cual es una tarea bastante tardada.

Para solucionar esta problemática vamos a utilizar dos tecnologías que se acoplan perfectamente a Spring Boot, las cuales son Sleuth y Zipkin. Sleuth es una solución de rastreo distribuido para Spring y tiene como principal ventaja que es auto configurado, por lo que no requiere de ninguna acción por parte del programador para comenzar a realizar las trazas distribuidas.

La segunda herramienta es Zipkin, el cual es una aplicación que se ejecuta por separado y tiene como propósito la recopilando toda la traza de los Microservicios y proporcionarnos una UI para consultar las trazas.

Debido a que Zipkin tiene como dependencia a Sleuth, solo es necesario importar Zipkin como librería en el archivo *pom.xml*:

```
1. <dependency>
2.   <groupId>org.springframework.cloud</groupId>
3.   <artifactId>spring-cloud-starter-zipkin</artifactId>
4. </dependency>
```

El primer cambio que veremos cuando agregamos esta librería, es que los logs comenzarán a tener un formato homogéneo, como el siguiente:

```
1. 2020-01-09 17:21:09.364 INFO [crm,fcff4a08d66aaedc,a87553d5b7ef7855,true] 17764 --- : Flipping property...
2. 2020-01-09 17:21:09.369 INFO [crm,fcff4a08d66aaedc,a87553d5b7ef7855,true] 17764 --- : Shutdown hook installed...
3. 2020-01-09 17:21:09.369 INFO [crm,fcff4a08d66aaedc,a87553d5b7ef7855,true] 17764 --- : Client: security instant...
4. 2020-01-09 17:21:09.369 INFO [crm,fcff4a08d66aaedc,a87553d5b7ef7855,true] 17764 --- : Using serverListUpdater Polling...
5. 2020-01-09 17:21:09.394 INFO [crm,fcff4a08d66aaedc,a87553d5b7ef7855,true] 17764 --- : Flipping property: security....
```

La primera sección del log contiene la fecha, hora y el nivel del log, lo que nos permite saber el momento en que se generó esa entrada y el nivel de criticidad del mensaje. La siguiente sección contiene metadatos para la trazabilidad, los cuales son utilizados para determinar que microservicio lo generó, y si forma parte de una transacción más grande, por ejemplo, el primer campo (remarcado en verde) indica el Microservicio que generó el log, por lo que en este caso vemos que *crm* fue quien generó el log, el segundo (marcado en azul) indica el *TraceID*, el cual es un ID que comparten todos los Microservicios que son ejecutados bajo la misma operación, y mediante el *TraceID* es posible agrupar los logs de otros Microservicios como parte de una misma operación. Finalmente, el tercer campo corresponde al *SpanID*, el cual hace referencia a una unidad de trabajo, es decir, un método o un servicio, de esta forma, a medida que un servicio mande a llamar a otro, el *TraceID* no cambiará, pero el *SpanID* si cambiará.

En la parte final de cada línea tenemos un texto libre que corresponde al mensaje que queremos guardar en el log, por lo que no tiene un formato predefinido.

Para ver cómo es que esto funciona podemos hacer una nueva compra en el e-commerce y vamos a ir analizando los logs para ver qué está pasando. Cabe mencionar que es posible que no todos los Microservicios tengan logs relacionados con la ejecución, pues no en todos imprimimos algo en la consola, así que analizaremos donde si hemos impreso algo para analizar la traza. El primer Microservicio que nos interesa es el *crm-api*, pues allí es donde se crea la orden:

```
1. 2020-01-10 13:31:35.351 INFO [crm,06d6a63c09224e01,7b1388209bb8eb64,true] 28552 --
- : New order request ==>
2. 2020-01-10 13:31:35.385 INFO [crm,06d6a63c09224e01,7b1388209bb8eb64,true] 28552 --
- : New order created ==>172, refNumber > 6e4e8cc1-bc3b-4981-a30d-e13a2b8176d5
```

Observa nuevamente cómo en verde tenemos el Microservicio que creó el log y luego en azul el *TraceID* para identificar la transacción completa, y en amarillo, el *SpanID*, el cual hace referencia solo al método de creación de la orden.

El siguiente log a analizar será el del servicio de *security*, que es desde donde validamos el token:

```
1. 2020-01-10 13:31:35.304 INFO [security,06d6a63c09224e01,0858c2bd43a152ca,true] 72800 --
- : Token decrypt ==>
2. 2020-01-10 13:31:35.333 INFO [security,06d6a63c09224e01,808418f27c9c3c0a,true] 72800 --
- : Token decrypt ==>
```


Observa como en este log se menciona que el que creo el log es el Microservicio *security*, además, observar que el *TraceID* (verde) es el mismo que tenemos en el log del *crm-api*, esto es importante porque agrupa las llamadas como parte de la misma transacción, la cual utilizaremos más adelante para dar seguimiento a una ejecución.

Finalmente, el *SpingID* si es diferente, pues este cambia por cada método que ejecutamos.

El siguiente log a analizar es el del Microservicio *mail-sender*:

```
1. 2020-01-10 13:31:35.453 INFO [mail-sender,06d6a63c09224e01,adb09dd88624db0c,true] 44540 --
- : mailSender => EmailDTO@6f355789[
2.   to=oscar_jb1@hotmail.com
3.   from=no-reply@crm.com
4.   subject=Hemos recibido tu pedido
5.   message=Hola oscar,<br> Hemos recibido tu pedido <strong>#6e4e8cc1-bc3b-4981-a30d-e13a2b8176d5</strong>, ....
6. ]
```

Lo primero a resaltar es el nombre del Microservicio *mail-sender*, luego, tenemos el mismo *TraceID* que corresponde con los dos logs anteriores y nos indica que es parte de la misma transacción, finalmente, el *SpinID* cambia porque se trata de un nuevo método.

Cabe mencionar que Sleuth es el que se encarga de generar los *TraceID* y el *SpinID*, por lo que nosotros como programadores no necesitamos hacer nada, lo cual es una gran ventaja, sin embargo, según la tecnología utilizada y la herramienta de *Log Aggregation* que utilicemos, quizás, esa tarea la tengamos que hacer manualmente, así que tenemos que documentarnos adecuadamente según la tecnología que vamos a utilizar.

Analices de la traza con Zipkin

Si bien Sleuth nos ayuda a agrupar las entradas de log por medio de los *TraceID*, todavía tenemos el problema de que necesitamos ir a cada log para recuperar cada parte de la traza, y es allí donde entra Zipkin, pues esta herramienta nos ayuda a centralizar toda la traza y visualizarla de una forma más gráfica, así que lo que haremos para comenzar a trabajar con Zipkin será ejecutarlo.

Zipkin es Open Source, así que podrás descargarlo y utilizarlo sin ningún tipo de licenciamiento o costo, así que puedes descargarlo libremente desde su página

web (<https://zipkin.io/>), sin embargo, yo he puesto el ejecutable en el repositorio de Github de este libro, para asegurarnos de trabajar con la misma versión y no tener problemas de compatibilidad. Para utilizar la versión de Zipkin que he puesto en el repositorio nos dirigiremos a la carpeta donde hemos descargado el repositorio y luego abriremos la carpeta llamada Zipkin, en la cual veremos un archivo con extensión .jar llamado zipkin-server-2.18.0-exec. Para ejecutarlo, tendremos que abrir la terminal y navegar hasta la carpeta donde se encuentra este archivo y ejecutaremos el comando:

```
1. java -jar zipkin-server-2.18.0-exec.jar
```

Lo que dará como resultado la siguiente salida:

[illegible]

Fig 310 - Ejecución de Zipkin.

Esperamos unos segundos hasta que termine de iniciar y después abrimos el navegador en la URL <http://127.0.0.1:9411/>, si es la primera vez que entras verás una página como la siguiente:

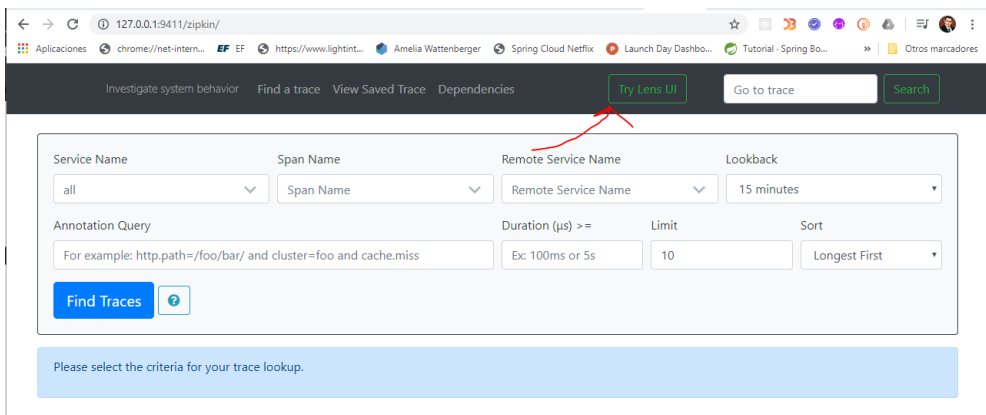


Fig 311 - Versión antigua de Zipkin.

Esta interface gráfica es antigua, por lo que daremos click en el botón “Try Lens UI” que se encuentra en la sección superior para abrir la nueva interface gráfica:

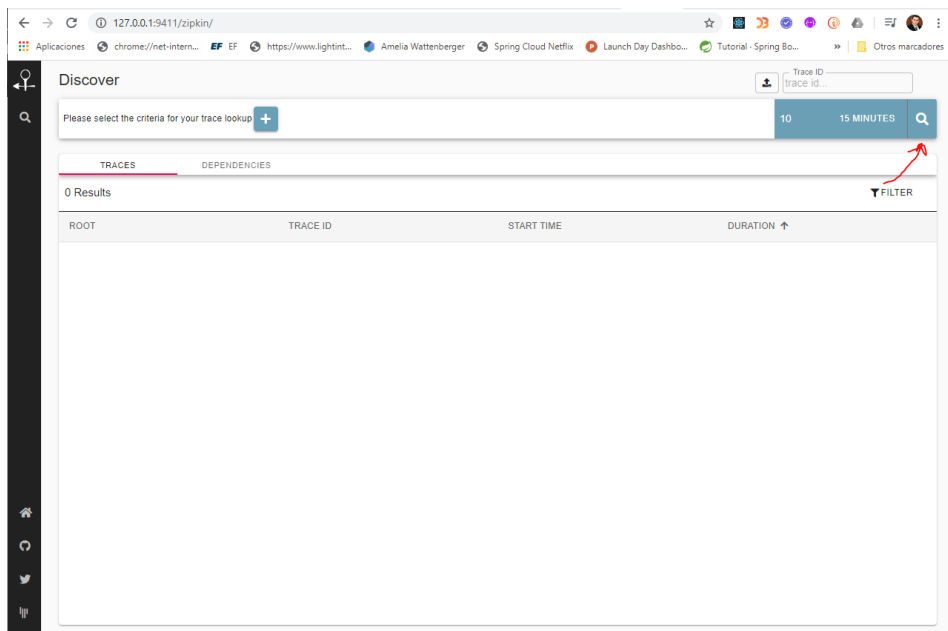


Fig 312 - Nueva interface gráfica de Zipkin.

Una vez que Zipkin está funcionando, solo nos restará hacer una nueva compra en la aplicación y actualizar Zipkin mediante el botón de la lupa ubicado en la parte superior derecha para ver la magia:

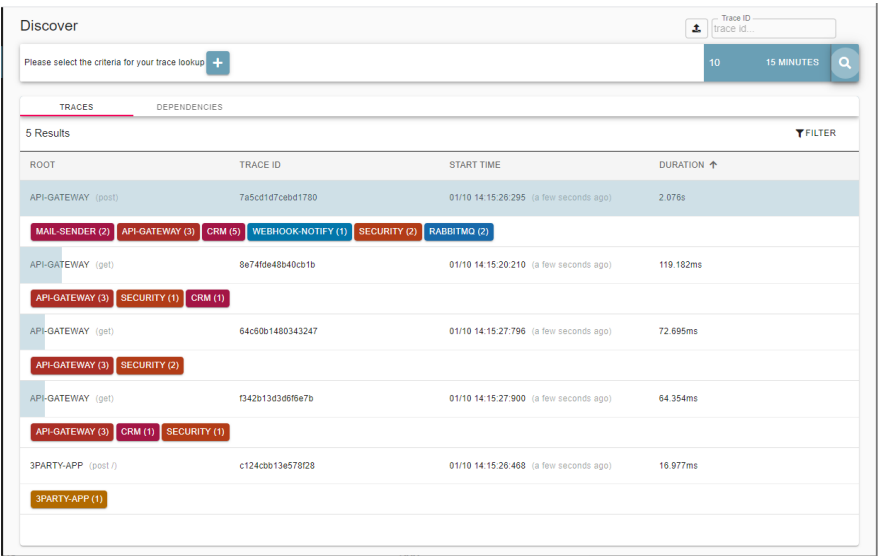


Fig 313 - Analizando la traza con Zipkin.

Cuando actualices la página verás varios registros de log, los cuales corresponde a todas las ejecuciones que hemos realizado al API, recuerda que una ejecución es por ejemplo, autenticarnos, consultar los productos o realizar una compra, por lo que según la operación que realicemos, serán los Microservicios involucrados, por ejemplo, en la imagen anterior, podemos ver que el primer registro corresponde a una nueva compra, y nos indica que los Microservicios involucrados fueron *mail-sender*, *api-gateway*, *crm*, *webhook*, *security* y *rabbitmq*.

También podemos dar click en la traza para ver el detalle completo:

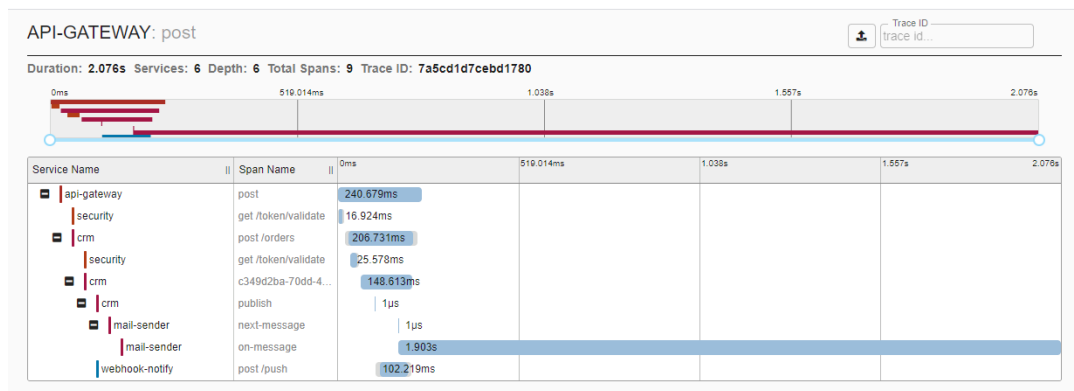


Fig 314 - Detalle de la traza.

Aquí es donde las cosas se comienzan a poner interesantes, pues Zipkin tiene la capacidad de no solo de agrupar los logs mediante el *TraceID*, si no que puede construir una jerarquía de ejecución, de tal forma que podemos ver qué servicios mandan a llamar a otros, lo que nos permite entender con detalle los pasos que se realizaron en la ejecución.

Además de los microservicios involucrados y su jerarquía, podemos apreciar el tiempo de ejecución de cada servicio, y la columna "*Span Name*" nos dice el nombre del método o la operación REST ejecutada, lo cual nos da más información de la ejecución.

Pero no es todo, si damos click en cualquier registro podemos expandir aún más información sobre cada Microservicio ejecutado, por ejemplo, si le damos click al Microservicio *security* veremos algo como esto:

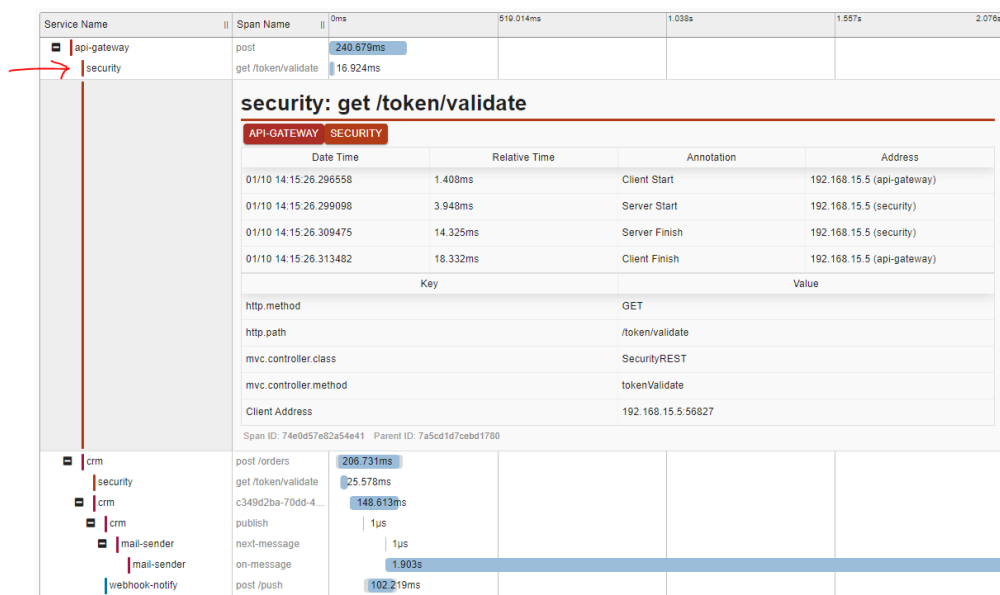


Fig 315 - Detalle de la ejecución.

En este detalle podemos ver el método HTTP utilizado (GET), el path (/token/validate), entre otros datos interesantes.

Algo a tomar en cuenta es que, todo el log que se imprime en la terminal no se envía a Zipkin, en su lugar, es necesario indicar manualmente que información es relevante y cual no. De forma predeterminado, Zipkin recolecta la información de la traza para saber el orden de ejecución, pero no guarda todo lo que imprimimos en la consola, por lo que para llevarlos a Zipkin es necesario crear *Tags*.

Un Tag es una propiedad llave-valor, donde la llave es el nombre del tag y el valor es un texto libre que podemos enviarle a Zipkin, de esta forma podemos agregar información complementaria a la traza. Un ejemplo de un Tag lo podemos ver si expandimos el detalle del Microservicio crm:

Service Name || Span Name || Oms || 519.014ms || 1.038s || 1.557s || 2.070s

api-gateway

security

crm

security

crm

post240.679msget /token/v...16.924mspost /orders206.731msget /token/v...25.578msc349d2ba-7...148.613ms

crm: c349d2ba-70dd-4d42-bbdc-624072dce636

CRM

Date Time	Relative Time	No rows found	Annotation	Address
Key				Value
order.new				c349d2ba-70dd-4d42-bbdc-624072dce636

Span ID: d5b2346e868df58d Parent ID: 84a6830316970d71

crm

mail-sender

mail-sender

webhook-notify

publish1µsnext-message1µson-message1.903s

post /push102.219ms

Fig 316 - Analizando un Tag.

En la imagen anterior podemos ver que hemos creado el Tag `order.new` que tiene como valor el UUID o identificar del pedido creado. Esto nos ayuda a identificar cual es el pedido que fue creado en esta operación.

Para crear un tag usamos la librería *Tracer*, la cual debemos inyectar en cualquier parte donde queramos utilizarla, por ejemplo, en la clase *OrderService* del microservicio *crm-api*, donde podemos ver las siguientes líneas al inicio de la clase:

```
1. @Autowired
2. private Tracer tracer;
```

Luego, simplemente creamos el tag de la siguiente forma:

```
1. tracer.currentSpan().tag("order.new", newSaleOrder.getRefNumber());
2. tracer.currentSpan().name(saleOrder.getRefNumber());
```

La primera línea crea el Tag y la segundo la utilizamos para establecer el nombre del *span*, en este caso, establecemos el número de referencia como nombre del *span*.

También podemos hacer búsquedas por varios criterios, como por ejemplo *TraceID* o un *Tag*, para esto, solo nos regresamos a la página principal de Zipkin y establecemos los criterios de búsqueda:

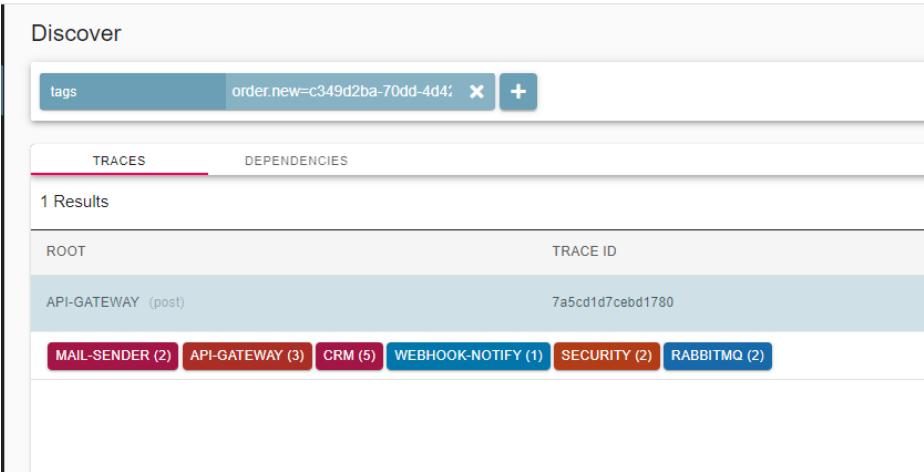
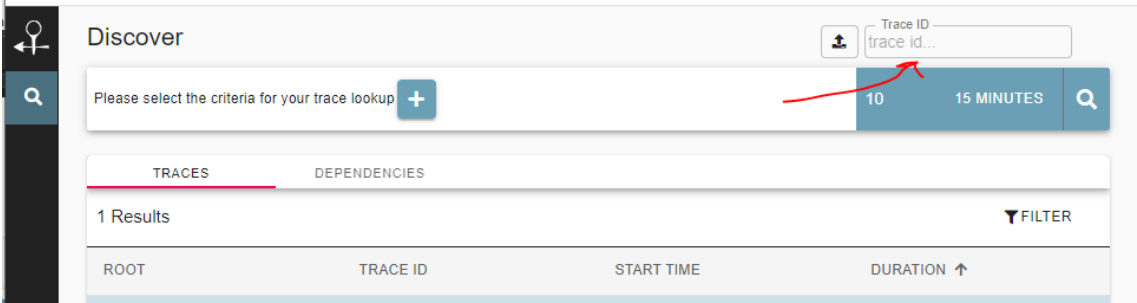


Fig 317 - Buscando por Tag.

En este ejemplo he realizado una búsqueda por el tag "order.new=c349d2ba-70dd-4d42-bbdc-624072dce636" el cual nos arrojará todas las tazas donde existe un tag llamado order.new con el valor c349d2ba-70dd-4d42-bbdc-624072dce636.

También podemos hacer una búsqueda por Trace ID utilizando el campo especial que se encuentra en la esquina superior derecha.



Conclusiones

Como hemos podido analizar en esta sección, contar con herramientas para centralizar las trazas nos puede dar grandes ventajas al momento de analizar lo que está pasando en nuestra aplicación, ya que nos permite encontrar los errores de una forma más eficiente, dándole una mejor atención a nuestros clientes.

Otra de las ventajas, es que los programadores no necesitan acceso al sistema operativo de todos los servidores para poder recuperar el log, lo que proporciona una mejor seguridad y hace que ellos mismo puedan analizar los errores sin necesidad de que alguien más les proporcione los archivos de logs.

Conclusiones

Si estás leyendo estas líneas es porque seguramente ya has concluido de leer este libro, lo que seguramente te dará nuevas habilidades que podrás dominar en tu día a día, sin embargo, recuerda que el camino a la arquitectura de software es un camino largo y complicado que solo unos cuantos son capaces de cruzar.

Este libro es solo una introducción a la arquitectura de software por lo que todavía quedan muchas cosas por aprender que solo la experiencia te dará, por lo que te invito a que continúes aprendiendo nuevas cosas y nunca parar de leer sobre tecnologías y arquitectura de software.

Si bien en este libro hemos explicado varios temas, la realidad es que apenas lo hemos tocado por la superficie, ya que existe libros completos sobre cada tema del que hemos hablado en este libro, pero ahora que tienes un panorama más abierto de lo que hay allí afuera, lo que sigue es especializarte más afondo en cada uno de estos patrones o estilos arquitectónicos.