INGENIERÍA EN INFORMÁTICA

Tesis de Ingeniería

Sistema de mensajería instantánea basado en arquitectura de microservicios

Eric Cristian Loza Montaño

Departamento de Tecnología y Administración
Universidad Nacional de Avellaneda

Avellaneda, Provincia de Buenos Aires - República Argentina

INGENIERÍA EN INFORMÁTICA

Sistema de mensajería instantánea basado en arquitectura de microservicios

Eric Cristian Loza Montaño

.....
Fecha de Exposición

.....	.....
Eric Loza Montaño	Fernando Asteasuain
Tesista	Director

.....
Calificación

.....	.....	.....
.....	.....	.....
Jurado	Jurado	Jurado

Agradecimientos

A mis padres, mis hermanos y mi sobrina.

A la Universidad Nacional de Avellaneda,

su personal y sus autoridades.

A mi director de tesis, Fernando Asteasuain,

y a mi coordinador, Roberto Mayer.

A mis amigos y compañeros de estudio.

Contenidos

1. Introducción

El objetivo principal de este trabajo es realizar un análisis y diseño de un sistema de mensajería instantánea. Este tipo de sistemas es fundamental en un mundo interconectado y globalizado como el actual, y particularmente para la comunicación interna y externa de las empresas. Se utilizará una arquitectura orientada a microservicios, ya que se adapta a muchos de los desafíos de las aplicaciones modernas.

Quan-Haase et al. (2005) consideran que, en muchas organizaciones, los empleados colaboran mediante mensajería instantánea, ya sea como complemento del email o como reemplazo. La adopción de la misma se suele originar en los mismos trabajadores porque están acostumbrados a usarla en su vida personal. Lo que hace popular a la mensajería instantánea es la rapidez y facilidad de la comunicación porque evita coordinar llamadas telefónicas y reuniones físicas.

De igual forma, Maina (2013) afirma:

> La mensajería instantánea permite conectar personas, sin importar dónde estén localizadas. Dentro de la compañía, colegas pueden enviar y responder mensajes en tiempo real, sin una interacción cara a cara. Además, los empleados pueden comunicarse con clientes o proveedores de esta misma manera.

Por otra parte, el impacto de la tecnología en la vida cotidiana reformuló muchas de las expectativas de los usuarios con respecto a las aplicaciones. Según Davis (2019), ya no es aceptable que el software no esté disponible o provea una mala calidad de servicio. Aún más, el usuario espera que la aplicación tenga novedades de forma constante y provea una experiencia personalizada. Es imprescindible tener en cuenta estos hechos para el diseño de una solución moderna.

El enfoque de este trabajo es el diseño inicial del denominado *backend* de este sistema. En pocas palabras, esto significa el componente donde reside la lógica de negocio y la interacción con la base de datos. Se analizarán las alternativas disponibles para este aspecto de la aplicación, teniendo en cuenta las buenas prácticas y las múltiples novedades surgidas en los últimos años. En trabajos posteriores, se priorizará el desarrollo completo del *backend* y de la interfaz gráfica, además de un análisis detallado de los requerimientos funcionales.

Estructura

- Introducción: es una comparación entre aplicaciones monolíticas y orientadas a microservicios. Se define en mayor detalle el significado de backend y frontend.
- Requerimientos de una aplicación moderna: se enumeran aspectos no funcionales fundamentales en el diseño inicial de la aplicación.
- Microservicios: Varios de los desafíos de este tipo de arquitectura son analizados a lo largo de este capítulo. Como es sabido, todas las decisiones técnicas conllevan aspectos positivos y negativos, y estos deben distinguirse claramente para lograr un correcto diseño.
- Diseño la solución: se diseña un esquema de microservicios a partir de las funcionalidades iniciales del sistema de mensajería. Se retoman muchos de los apartados del capítulo anterior y se profundizan consideraciones relacionadas a la implementación. Sobre el final, se desarrolla una *proof of concept* del diseño propuesto.
- Conclusión: se analizan opciones alternativas a las abordadas en capítulos anteriores y se definen los pasos a seguir en futuros trabajos.

Alcance

El alcance de esta tesis es el diseño inicial del sistema de mensajería. Este diseño incluye aspectos relacionados a la infraestructura, la seguridad, el modelo de datos, las APIs, el monitoreo, entre otros. También, se realizará un desarrollo de software, donde se implementan muchas de las decisiones propuestas. Los siguientes tópicos no están dentro del alcance de esta tesis:

- El desarrollo de la totalidad de las funcionalidades del sistema, ya que el desarrollo mencionado sólo tendrá lo mínimo y necesario para plasmar el diseño inicial.
- Una investigación sobre la usabilidad y el diseño de una interfaz gráfica.
- Un detallado análisis de requerimientos funcionales.

2. Arquitectura

2.1. Sistemas monolíticos

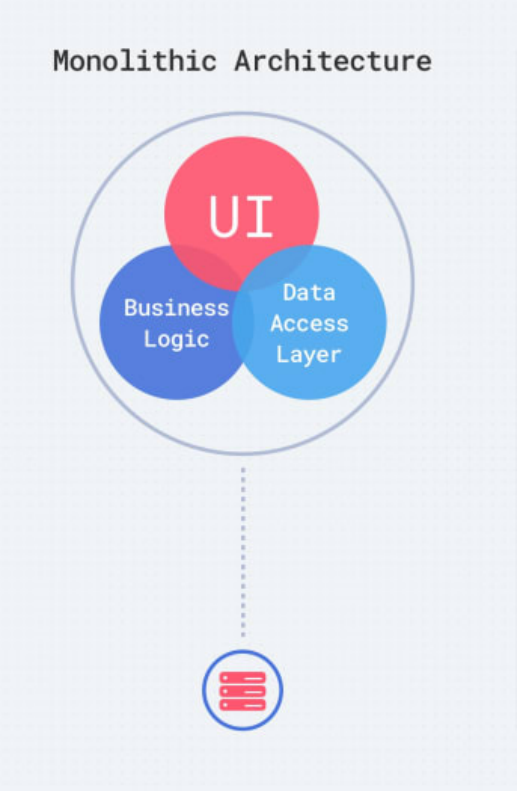
Este tipo de sistemas se caracteriza por centrar sus funcionalidades en una sola unidad. La comunicación entre las partes se realiza mediante mecanismos internos de la propia aplicación. En etapas iniciales de desarrollo, este tipo de arquitecturas tiene una serie de ventajas, debido a su simpleza. Sin embargo, con el paso del tiempo, la complejidad suele ser cada vez mayor y se torna inmanejable para los equipos de desarrollo.

Según Tilkov (2015), las partes de un monolito están extremadamente acopladas entre sí y esa es la clave para definirlo como tal. Todas las partes se comunican con la base de datos u otras aplicaciones mediante abstracciones compartidas. Aún más, los objetos de dominio y modelo de persistencia también están compartidos. En la figura 2-1, podemos observar que la interfaz de usuario, la lógica de negocio y la capa de acceso a los datos conviven en el mismo componente.

Este paradigma fue muy común hace unos años y aún se sigue utilizando en casos de uso adecuados. Muchas empresas actuales empezaron con sistemas monolíticos, entre las cuales podemos nombrar las reconocidas Amazon, Mercadolibre y eBay[1]. Posteriormente, estas compañías migraron a esquemas de microservicios.

[1] eBay empezó con una arquitectura monolítica y, a partir del 2005, comenzó una migración a microservicios: https://www.slideshare.net/tcng3716/ebay-architecture .

Figura 2-1



Fuente: https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m

Richardson (2018) enumera distintos beneficios y limitaciones de este tipo de arquitectura:

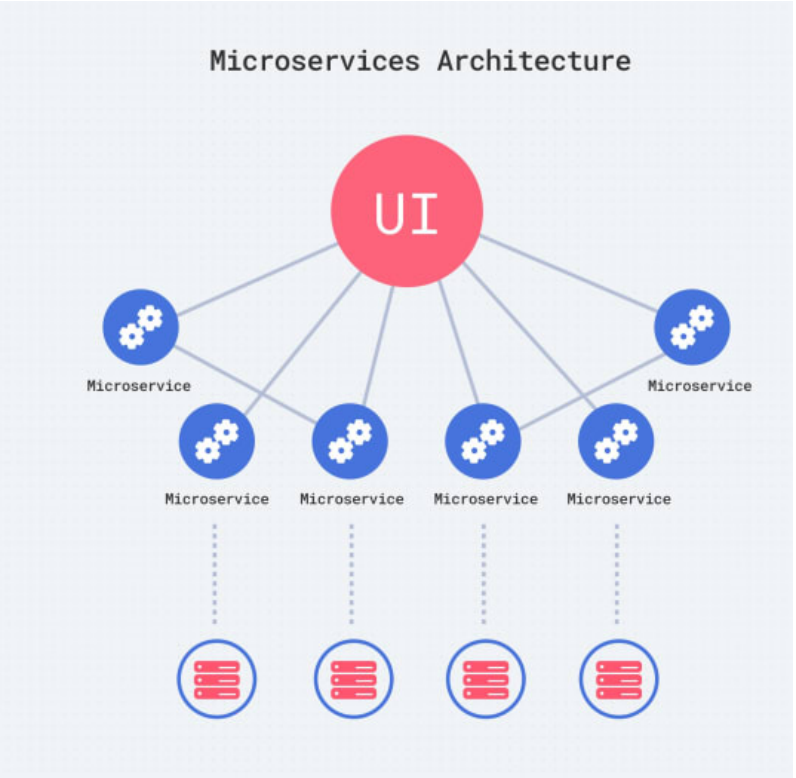
- Beneficios
    - Simplicidad para desarrollar: los entornos de desarrollo integrados y otras herramientas están optimizadas para trabajar con una sola aplicación
    - Facilidad para realizar cambios radicales: al estar todo centralizado en un solo lugar, se pueden hacer cambios de fondo, tanto en el código como en el esquema de base de datos.
    - El *testing* es sencillo: los desarrolladores pueden crear pruebas donde se verifican flujos completos dentro del sistema. Además, es posible seguir y comprender completamente una funcionalidad a través del código.
    - El cambio de versiones es simple: sin importar la tecnología y el lenguaje de programación, siempre es más fácil actualizar una aplicación que varias.
    - Facilidad para escalar: la aplicación puede replicarse fácilmente en otro servidor, si es que necesitamos más capacidad de procesamiento.

    - Robusta interacción entre módulos: se usan mecanismos internos del lenguaje de programación para la comunicación.
- Limitaciones que surgen a partir de la creciente complejidad de la aplicación
    - Dificultad para comprenderlo: no es trivial tener una visión general del sistema y cada una de sus funcionalidades.
    - El desarrollo es lento: el entorno de desarrollo integrado necesita cada vez más recursos para administrar la aplicación. Esto causa que tanto el proceso de compilación como el de testing local sean más lentos.
    - Dificultad para cambiar versiones aplicativas: muchos equipos modifican el mismo repositorio de código y, de esta forma, realizar pruebas completas del sistema para verificar todos estos cambios lleva mucho tiempo.
    - Escalar es difícil: cada módulo tiene distintas necesidades de recurso. Algunos pueden hacer uso intensivo de CPU y otros usar más memoria. Dado que todos los módulos conviven en la misma aplicación, no es posible optimizar sus recursos individualmente.
    - Inestabilidad: cada versión de la aplicación puede contener muchos cambios en distintos módulos. Todos estos cambios dificultan las pruebas necesarias para verificar la correctitud del sistema. Adicionalmente, un error en un módulo puede generar problemas en toda la aplicación.
    - Imposibilidad de adoptar nuevas tecnologías: cada módulo está obligado a compartir las mismas tecnologías con el resto. Cambiar versiones del lenguaje de programación o de alguna dependencia suele ser una tarea muy compleja de realizar.

2.2. Microservicios

Las limitaciones de las aplicaciones monolíticas obligaron a repensar la forma en que se diseñan las aplicaciones. Newman (2019) define los microservicios como servicios autónomos que se modelan alrededor de un dominio de negocio. Los microservicios encapsulan el guardado de sus datos y la lógica para modificarlos. Además, cada uno expone representaciones de sus datos y funcionalidades mediante interfaces bien definidas. Su autonomía les brinda la posibilidad de evolucionar de forma independiente. En pocas palabras, este tipo de arquitectura consiste en múltiples servicios que colaboran entre sí.

Figura 2-2



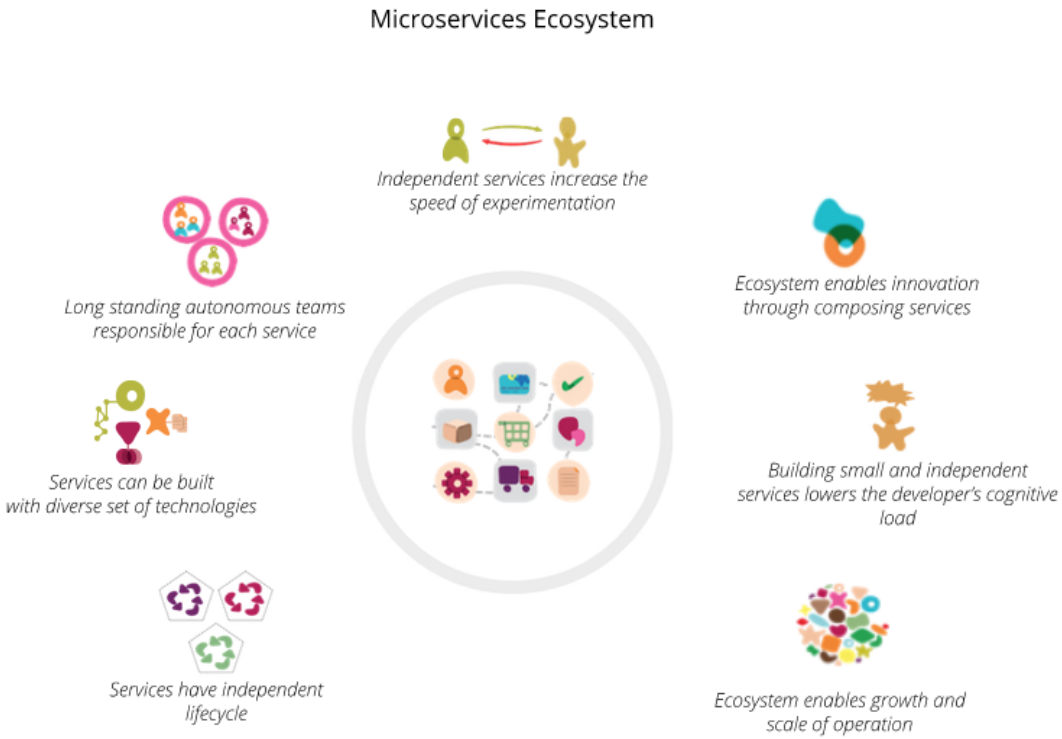
Fuente: https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m

Estas son las características de las arquitecturas orientadas a microservicios según Fowler (2014):

- La aplicación está descompuesta en componentes llamados microservicios. Cada servicio es una unidad de software que puede actualizarse o reemplazarse de forma independiente. Un equipo está a cargo del mismo.

- Cada servicio está creado a partir de una responsabilidad de negocio. De esta forma, los equipos administran productos, no proyectos de una duración determinada.
- La comunicación se realiza mediante mecanismos simples y estándar, como el protocolo HTTP.
- Los servicios deben ser pensados de forma tal que puedan ser reemplazados sin afectar al resto. Dicho de otra forma, es inusual que un cambio en un servicio necesite un cambio en otro para funcionar adecuadamente.
- Están diseñados para ser tolerantes a errores a nivel red, infraestructura o relacionados a otros servicios.
- Cada equipo administra todas las etapas de diseño, desarrollo y puesta en producción. Además es responsable del modelo de datos y las tecnologías necesarias para el correcto funcionamiento de su servicio.
- Hay un proceso que abarca todas las etapas para llegar a producción, partiendo de un cambio en el repositorio de código. Las etapas son definidas y su ejecución es obligatoria y automática.

Figura 2-3



Fuente: https://www.martinfowler.com/articles/break-monolith-into-microservices.html

Por su parte, Richardson (2018) distingue las siguientes ventajas y desventajas:

- Ventajas
    - Cada servicio es, por definición, reducido. Su equipo responsable puede comprenderlo en su totalidad y, de esta manera, se facilita la mantención y la evolución.
    - Los servicios tienen la capacidad de escalar de forma independiente.
    - Cada servicio puede modificar su versión aplicativa, aislado del resto. Su equipo administra el ciclo de desarrollo.
    - Permite automatizar el proceso de testing y generación de nuevas versiones y, de esta forma, poner en producción cambios incrementales en cada servicio.
    - Mejora la autonomía y agilidad de cada equipo.
    - Facilita la experimentación y adopción de nuevas tecnologías, según el caso de uso a resolver.
    - Existe un mejor aislamiento de los errores.
- Desventajas
    - Definir cuántos servicios son necesarios y la responsabilidad de cada uno no es trivial.
    - Los sistemas distribuidos son complejos e implican muchos desafíos nuevos.
    - La comunicación debe ser considerada con especial atención porque se transforma en un nuevo punto de falla.
    - Cada servicio tiene su propia infraestructura, con todos los desafíos que esto implica.
    - Un cambio que involucre varios servicios es complejo de desarrollar y de coordinar. La puesta en producción tiene que ser cuidadosamente realizada en estos casos.

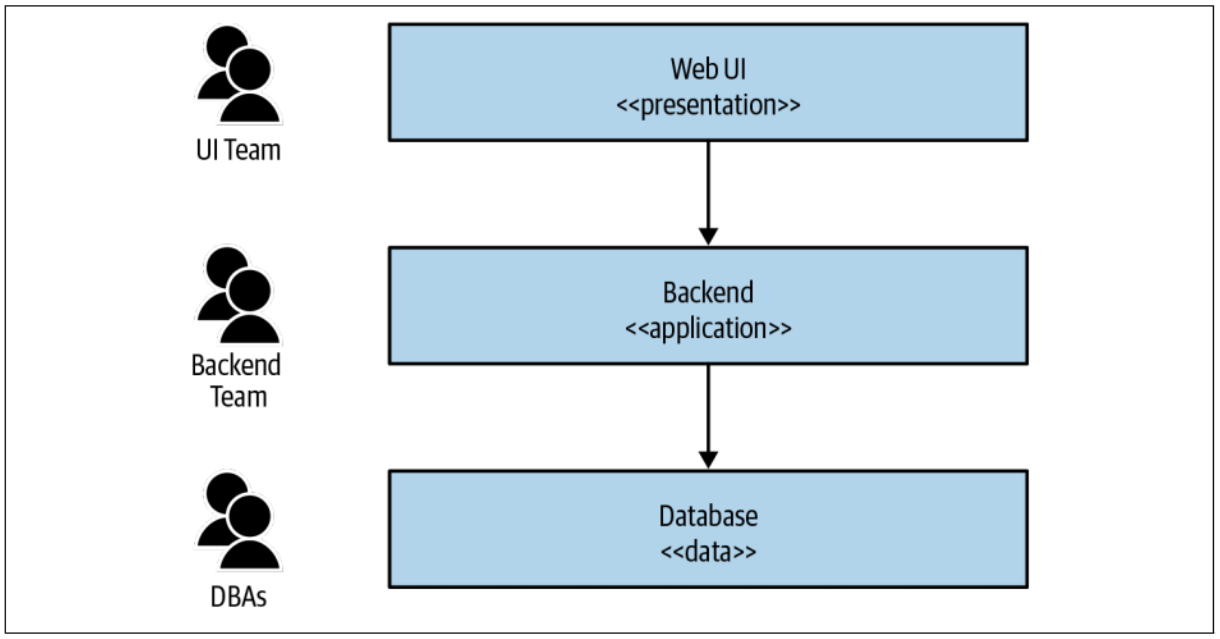
2.3. Backend, Frontend y Micro Frontend

Según Newman (2019), en la actualidad, es muy común que las aplicaciones web sean divididas en tres componentes bien diferenciados:

- *Frontend*: el usuario final se comunica directamente con este componente. Se encarga de mostrar una interfaz mediante una página web.
- *Backend*: es donde reside la lógica de negocio.
- Base de datos: su responsabilidad es guardar todas las entidades relevantes y permitir consultarlas.

En la figura 2-4, podemos observar esta arquitectura de tres capas, donde cada una tiene responsabilidades diferentes y es administrada por equipos especializados. De esta manera, cada equipo se crea a partir de las competencias principales de sus integrantes.

Figura 2-4



Fuente: Newman (2019)

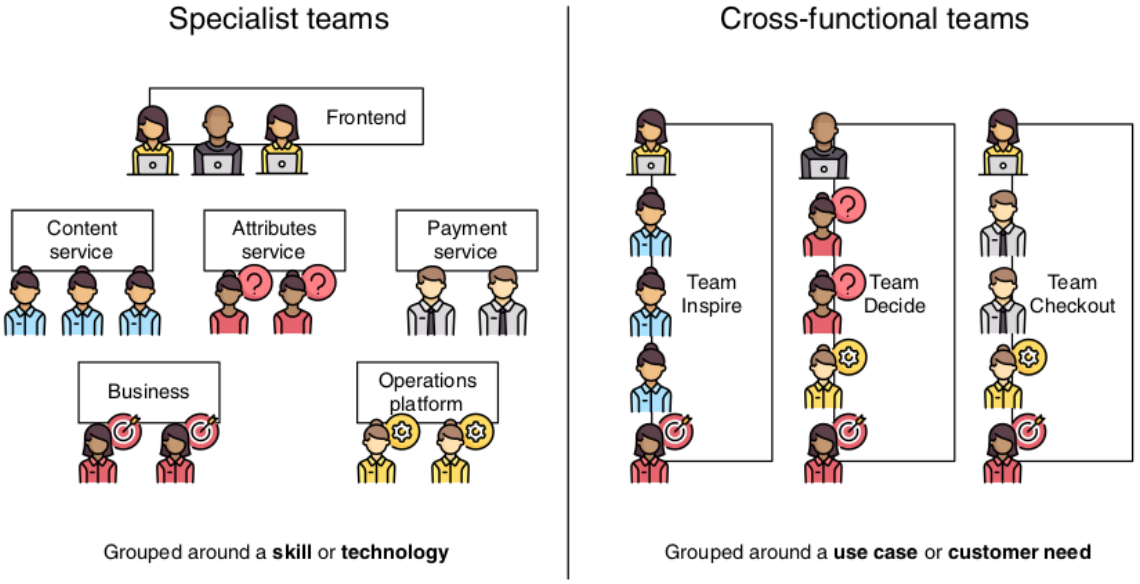
En este tipo de arquitecturas, los cambios suelen involucrar a todos los equipos. Newman (2019) considera el caso donde se quiere modificar una aplicación de búsqueda y reproducción de música. El cambio consiste en dar la posibilidad al usuario final de definir su género musical preferido.

Cada capa debe modificarse de la siguiente forma:

- Frontend: mostrar una lista de géneros disponibles.
- Backend: administrar los cambios en el género.
- Base de datos: guardar el género preferido

En contraposición a esta arquitectura, Geers (2020) afirma que los equipos deben crearse alrededor de un objetivo de negocio, no a partir de las competencias de sus integrantes. Desde el aspecto técnico, plantea adoptar los denominados *Micro Frontends* que, en pocas palabras, permiten dividir aplicaciones en fragmentos administrados por equipos diferentes. En la figura 2-5, se comparan ambas formas de organización.

Figura 2-5



Fuente: Geers (2020)

Adicionalmente, Geers (2020) resalta la importancia de un enfoque multidisciplinario, ya que llega a soluciones más creativas y efectivas a los problemas. Debido a la forma en que se conforman los equipos, todos los integrantes tienen claramente definidas sus responsabilidades y objetivos. Otra ventaja de este enfoque es que cualquier cambio se diseña y realiza dentro del mismo equipo, no implica costosas coordinaciones entre grupos con diferentes intereses.

3. Requerimientos de una aplicación moderna

Hace muchos años, las aplicaciones dejaron de ser un complemento para nuestra vida y pasaron a jugar un rol fundamental en la mayor parte de nuestras actividades diarias. Esta importancia modificó las expectativas de los usuarios con respecto al software. En pocas palabras, ellos esperan que siempre esté disponible, se actualice constantemente con nuevas funcionalidades y provea una experiencia personalizada. Durante este capítulo, vamos profundizar estos puntos que se consideran fundamentales en una aplicación moderna, como la propuesta en el presente trabajo.

Davis (2019) considera que, desde la etapa inicial de diseño de la aplicación, es necesario tener en cuenta estos aspectos y enumera los requerimientos clave para las aplicaciones modernas:

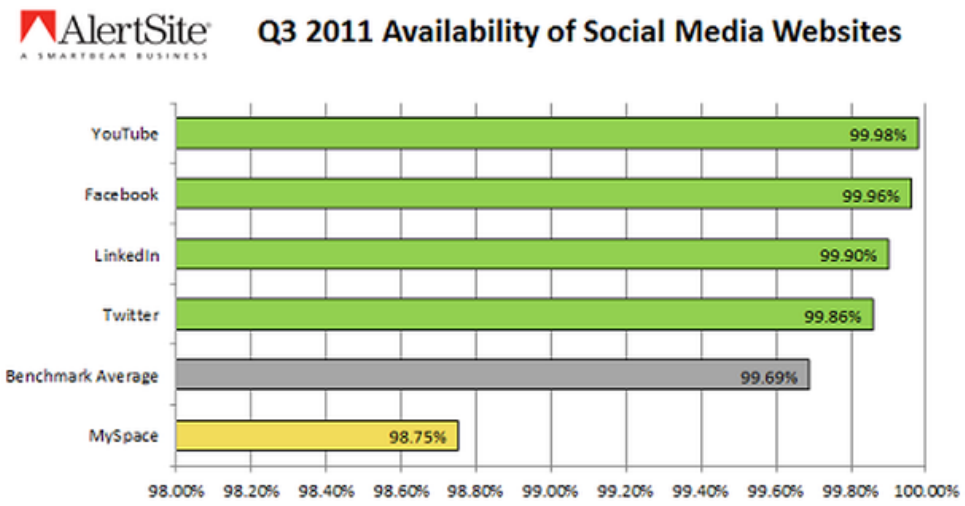
- Siempre disponible: una caída del sistema genera descontento entre los usuarios y, en aplicaciones de gran escala, implica una gran pérdida de ingresos. [2]
- Ciclos de retroalimentación cortos: las expectativas de los usuarios y la alta competencia aceleran el ritmo de cambios aplicativos. Es importante tener un seguimiento en tiempo real de la efectividad de cada funcionalidad.
- Soporte para celulares y otros dispositivos: la masificación de los teléfonos celulares y su mejora tecnológica causó un gran aumento del tráfico proveniente de estos dispositivos. Asimismo, es cada vez más común que los productos hogareños tengan la capacidad de conectarse a internet y, de esta forma, sea posible monitorearlos y controlarlos.
- *Data-driven*: el volumen de datos crece constantemente y sus orígenes se diversifican. Todos estos datos permiten brindar más valor a los usuarios finales mediante aplicaciones inteligentes, pero es desafiante administrarlos de forma eficiente.

[2] En 2013, Amazon tuvo una caída del sistema durante 30 minutos. Se estima que sus pérdidas totalizaron alrededor de 66.000 dólares por minuto.
https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/?sh=556e6982495c

3.1. Disponibilidad

Como hemos mencionado, las aplicaciones están tomando un rol cada vez más importante en la vida cotidiana de las personas. A lo largo de los últimos años, este proceso se fue profundizando cada vez más y todo indica que seguirá así. Las aplicaciones modernas deben diseñarse de tal manera que estén disponibles cerca del 100%[3] del tiempo De otra forma, no cumplirían las exigentes expectativas de sus usuarios. Ya en 2011, muchos sitios web tenían una disponibilidad cercana a este número, como podemos observar en la figura 3-1.

Figura 3-1



Fuente: Protalinski (2011)

Disponibilidad implica que la aplicación continuará funcionando tal como esperan sus usuarios, sin importar los errores que ocurran en los diferentes componentes del sistema. Para lograrlo, es necesario entender las expectativas que genera en cuanto a su funcionamiento. Según Kleppmann (2017), el usuario espera que:

- Se realice correctamente la acción solicitada
- Se toleren errores humanos y se indiquen de forma explícita
- La interfaz gráfica y sus operaciones se realicen en un tiempo acotado
- Se prevenga el uso no autorizado de la aplicación

[3] Probablemente, un valor más preciso sea 99.9999%.
https://seekingalpha.com/news/3102246-netflix-nears-four-nines-uptime-target

En un entorno dinámico como el actual, tolerar errores y ser resiliente a los mismos es fundamental. Estos pueden ser fallos humanos o a nivel hardware, software o red. En un mundo ideal, no querríamos lidiar con todos los posibles problemas, pero en la realidad estamos obligados a considerarlos. Los errores pueden y van a ocurrir. El proceso de *testing* y las distintas validaciones para llegar a producción son un aspecto relevante para evitarlos. Es deseable que estos procesos sean automatizados y rápidos para no ralentizar el desarrollo de software.

Otra de las claves para lograr disponibilidad es evitar puntos únicos de falla que tengan como consecuencia la caída general del sistema. Esto se puede realizar mediante la modularización y la replicación de cada módulo. En pocas palabras, dividir nuestro sistema en pequeñas partes independientes, donde el fallo de alguna no afecte al resto.

Por último, ¿Qué pasa si el error ya ocurrió y afectó a los usuarios? En primer lugar, es prioritario solucionarlo lo antes posible. Después de la resolución del incidente, es necesario realizar un análisis de lo sucedido, usualmente denominado *postmortem*.[4] En este análisis, se incluye el impacto del evento y su mitigación, en conjunto con la causa de lo ocurrido y los pasos a seguir para evitar que se repita. El postmortem no busca culpables, su objetivo es entender lo que pasó y accionar a partir de ello.

3.2. Escalabilidad

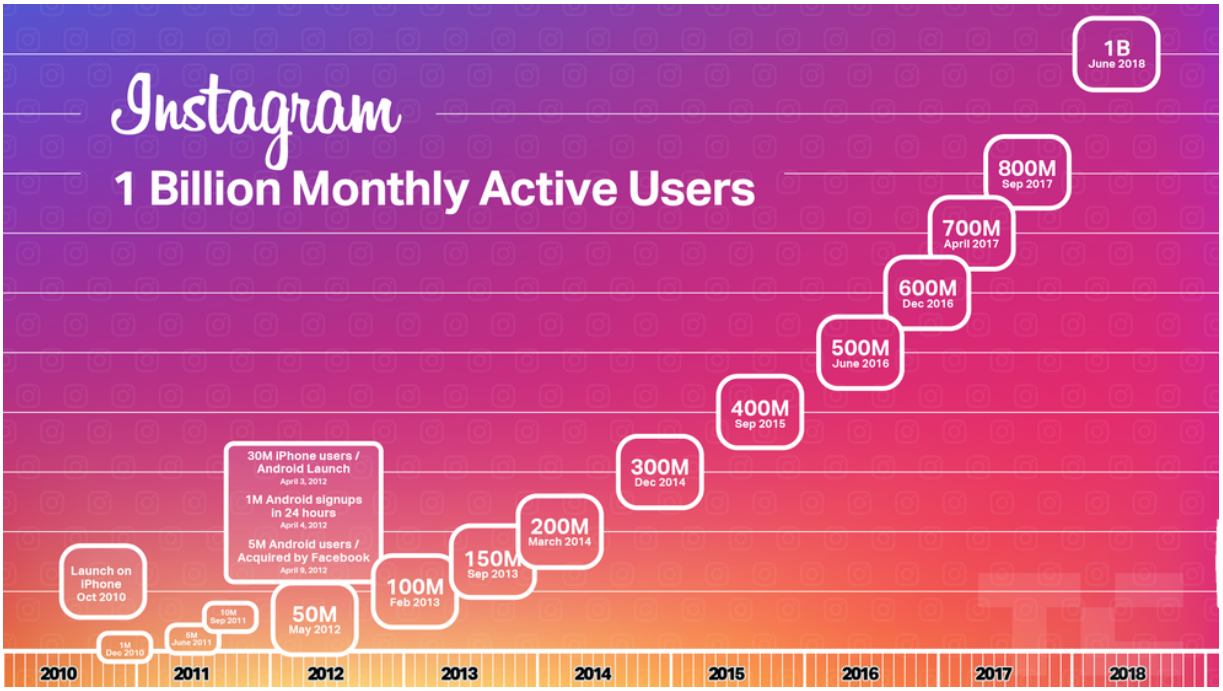
Debido a la masividad que adquirió Internet, con la difusión de las computadoras personales primero y con los smartphones después, el crecimiento de la cantidad de usuarios totales y concurrentes en las aplicaciones es cada vez mayor. En la actualidad, cualquier sitio web orientado al público debe estar preparado para adaptarse a este potencial crecimiento. En la figura 3-2, observamos el crecimiento en cantidad de usuarios activos de Instagram, una reconocida red social.

Dado este crecimiento potencialmente enorme, una aplicación puede funcionar correctamente en la actualidad, pero no es posible asegurar que lo hará de la misma forma en el futuro. Es imprescindible que sea escalable. Pero ¿A qué nos referimos exactamente con

[4] https://www.pagerduty.com/resources/learn/incident-postmortem/

escalable? A la capacidad de realizar una mayor cantidad de operaciones en simultáneo, sin perder calidad de servicio y con eficiencia en el uso de recursos.

Figura 3-2



Fuente: https://techcrunch.com/2018/06/20/instagram-1-billion-users

Los límites de nuestra aplicación en términos de escalabilidad pueden estar determinados por distintos componentes: la base de datos, la red, la aplicación, entre otros. En este sentido, las pruebas de carga, donde se exige al sistema hasta el límite, resultan muy útiles para detectar y resolver problemas de rendimiento. En particular, en este apartado, nos centraremos en el escalamiento de la aplicación.

Narumoto (2017) enumera dos formas formas principales de escalar aplicaciones:

Escalamiento vertical

Significa cambiar la capacidad de un recurso. En este caso, nos referimos a los servidores donde está alojada la aplicación. Suele requerir que la aplicación no esté disponible temporalmente durante el transcurso de la migración a los nuevos servidores, que tienen mayores prestaciones de hardware. Dependiendo de la necesidad de recursos, esto puede ser más o menos costoso, pero hay un límite donde ya no es posible realizarlo.

En general, el uso de una aplicación no es homogéneo a lo largo del día. Durante la noche se suele ver una baja en el uso y, en consecuencia, de los recursos necesarios. El problema es que nuestros servidores ya están adecuados al uso intensivo y reducir sus prestaciones no es trivial. Por esta razón, probablemente, soportemos toda la carga del sistema durante el día, pero vamos a tener un uso ineficiente de recursos en la noche.

Escalamiento horizontal

Otra forma de aumentar la capacidad de nuestra aplicación es mantener los recursos individuales de cada servidor pero aumentar su cantidad. El sistema puede seguir funcionando de forma ininterrumpida mientras nuevas instancias idénticas a las actuales se crean y procesan el exceso de carga.

Un beneficio adicional del escalamiento horizontal es que es flexible y nos permite reducir la cantidad de servidores, si es que estamos en un período del día donde no los necesitamos. Esto nos ahorra costos, pero también nos obliga a que la creación de nuevas instancias sea lo más rápida posible y el procesamiento correcto de operaciones no dependa de un servidor en particular.

3.3. Agilidad

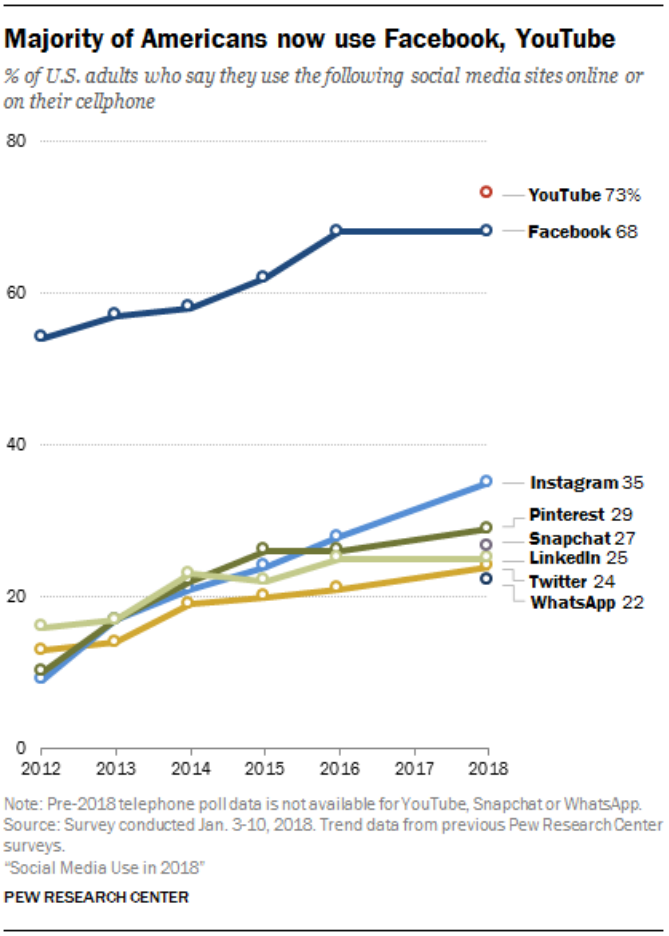
Como mencionamos anteriormente, la masividad de internet generó un ambiente propicio para el desarrollo de una infinidad de aplicaciones, que compiten entre sí. Esta competencia constante puede notarse en las redes sociales, donde hace unos pocos años Facebook era el líder indiscutible. Sin embargo, en el último tiempo, surgieron fuertes competidores como Instagram o Snapchat. Esta información se observa en la encuesta realizada por el Pew Research Center (2018) que vemos en la figura 3-3.

Está claro que el tiempo desde que una nueva funcionalidad es pensada hasta que está disponible en el mercado debe ser lo más breve posible. Esto nos permite ser uno de los primeros en introducirla y, así, tener una ventaja competitiva. La manera de lograrlo es mediante ciclos de desarrollo y puesta en producción cortos, donde se modifica la aplicación de forma constante e incremental.

Google (2019) realizó una encuesta a miles de ingenieros de distintas empresas y detectó que el 20% pone en producción nuevas versiones aplicativas más de una vez al día. Otro dato

interesante es que, en caso de un error crítico en su sistema, este se suele recuperar en menos de una hora.[5] Teniendo en cuenta la tendencia, el primer número debería seguir creciendo y el segundo reduciéndose.

Figura 3-3



Fuente: Pew Research Center (2018)

Organización

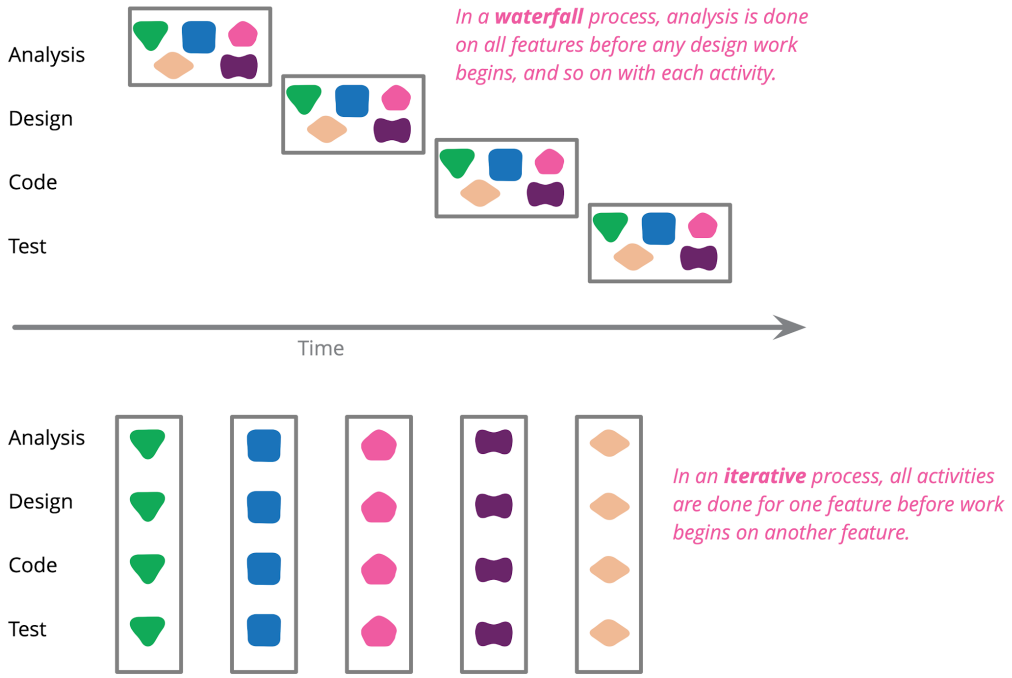
El hecho de lidiar con la vorágine de las aplicaciones modernas también generó cambios a nivel organizativo. Las nuevas funcionalidades y la mantención de las actuales debe planificarse y ejecutarse de una forma cualitativamente diferente. Fowler (2019b) contrapone dos tipos de proceso de desarrollo de software:

5 https://www.zdnet.com/article/devops-leaders-deliver-software-200-times-more-frequently-than-their-peers-study-shows

- *Waterfall*: se caracteriza por descomponer el trabajo en etapas diferenciadas por actividad. Todas las funcionalidades atraviesan estas etapas: análisis, diseño, desarrollo y *testing*. Cada una requiere la finalización de la anterior para poder comenzar.
- Iterativo o ágil: divide el proceso en funcionalidades. Cada etapa implica el diseño y desarrollo de una única funcionalidad que atraviesa diferentes pasos.

En pocas palabras, waterfall significa “hacer una actividad a la vez para todas las funcionalidades” e iterativo es “hacer todas las actividades para una funcionalidad a la vez”.

Figura 3-4



Fuente: Fowler (2019b)

En líneas generales, waterfall implica que es posible tener una planificación acertada de la duración de cada etapa y, como podemos suponer, esto es sumamente complejo de realizar. Además, los errores se verifican en las últimas etapas y esto implica que una mala planificación sólo es detectada cuando es demasiado tarde. Por estas razones, durante las últimas décadas y con el surgimiento de los microservicios, el uso de metodologías ágiles aumentó considerablemente.

3.4. Mantenibilidad

Es sabido que el mayor costo del software no es en su desarrollo inicial, sino en su mantenimiento posterior. Estamos hablando de solucionar errores, mantener el sistema operando correctamente, disminuir deuda técnica, desarrollar nuevas funcionalidades y modificar las actuales. En este sentido, Kleppmann (2017) considera que es necesario priorizar estos puntos:

- Fácil de operar: desde el punto de vista operativo, es necesario monitorear regularmente el estado general del sistema, hacer un seguimiento de problemas de software y hardware que aparecen, evitar el uso de software de terceros antiguo que puede ser vulnerable o tener *bugs*, anticipar problemas futuros a nivel rendimiento, entre otras cosas. Estas tareas deben facilitarse a los operadores del sistema.
- Entendible: los desarrolladores que trabajan en una aplicación van cambiando a lo largo del tiempo. Es fundamental que el código sea entendible para agilizar este proceso de recambio.
- Modificable: está fuertemente relacionada con la calidad del código y la existencia de de pruebas y validaciones que cubran los distintos casos de uso, incluyendo el uso habitual y los posibles errores.

Gran bola de lodo

Este particular concepto surgió a partir de un estudio donde no se mencionan los patrones ideales para el desarrollo de software, sino que se hace hincapié en los que surgen en la realidad. En su *paper*, Foote & Yoder (1999) afirman “la gran bola de lodo es la arquitectura de software que predomina en la práctica” y analizan las distintas causas de su existencia.

Una “gran bola de lodo” es un repositorio de código que parece estructurado al azar, como si hubiera crecido sin ninguna planificación. Probablemente, distintos parches permiten su aparente funcionamiento, pero dificultan de sobremanera entenderlo. Además, la información suele compartirse entre los distintos módulos, de tal forma que se transforma en información global o, peor aún, se replica en las diferentes partes del sistema.

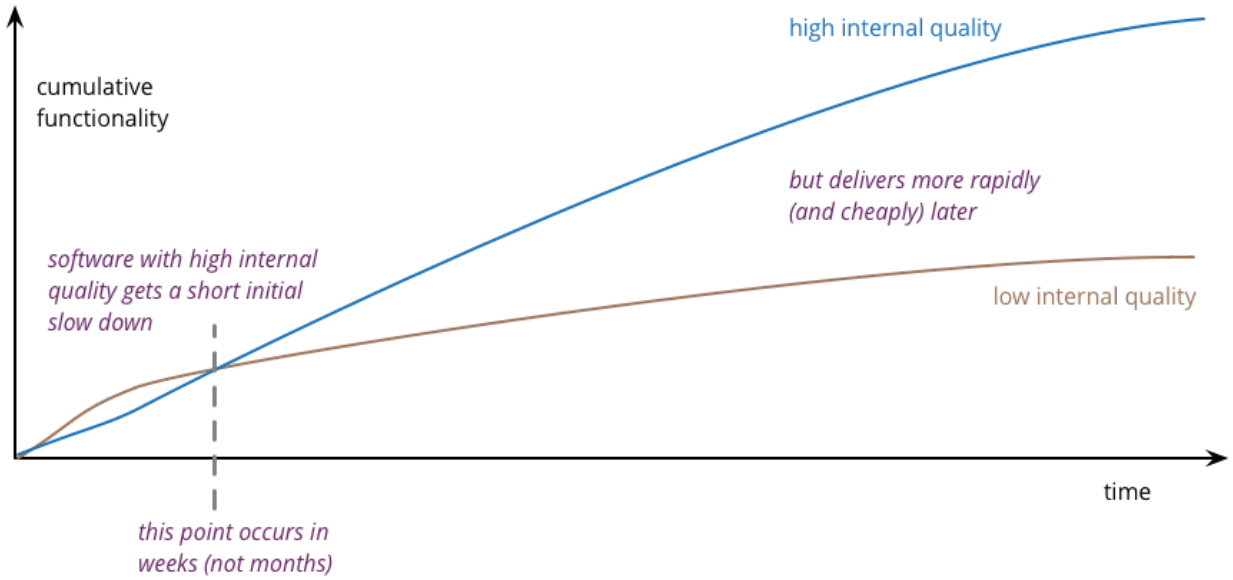
En relación a las causas de este patrón, todos los proyectos de desarrollo de software tienen un límite de recursos disponibles y de tiempo. Cuando este límite es demasiado bajo,

genera que el equipo se centre en funcionalidades y deje en segundo lugar la arquitectura, el rendimiento y las buenas prácticas. Esto también puede suceder si no hay una correcta apreciación del significado de la calidad de código. Es una inversión a mediano plazo. De hecho, en principio genera mayor carga de trabajo, sin un beneficio inmediato.

En ese sentido, Fowler (2019a) afirma:

> "Cuando la calidad del software es baja, el progreso inicial es rápido. No obstante, con el pasar del tiempo, es cada vez más complejo agregar nuevas funcionalidades. Incluso cambios menores requieren que los programadores tengan un entendimiento de grandes partes de código, que a su vez es difícil de comprender. Cualquier cambio puede generar efectos inesperados y traducirse en largos procesos de prueba."

Figura 3-5



Fuente: Fowler (2019a)

3.5. Integrable

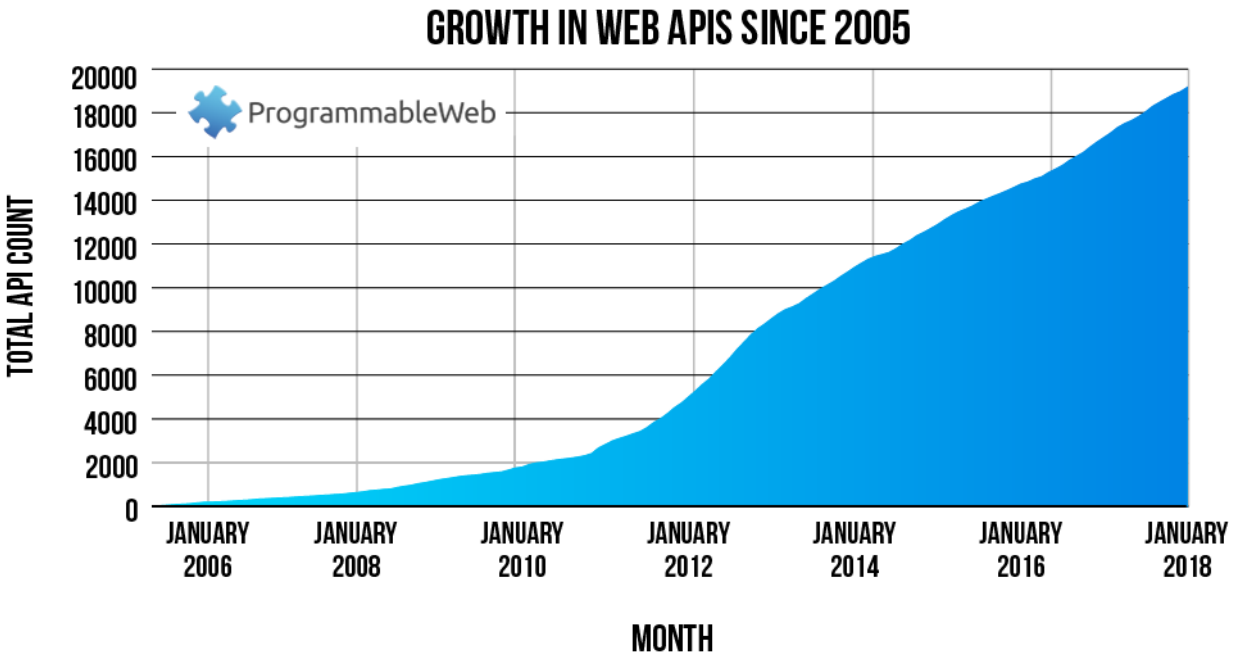
En los últimos años, distintas empresas a nivel global comenzaron a exponer públicamente formas de interactuar con sus sistemas.[6] En general, lo realizan mediante una interfaz denominada *Application Programming Interface* (API), la cual provee un conjunto de

[6] Podemos citar distintos ejemplos de empresas, pero uno bastante conocido a nivel local es Mercadolibre. https://developers.mercadolibre.com.ar

operaciones para consultar y modificar información. De esta forma, es posible desarrollar softwares que se integren fácilmente con estos sistemas.

Programmable Web (2019) es un sitio web con un amplio directorio de APIs web. Recientemente, realizaron un análisis sobre su crecimiento a lo largo de los últimos años, que podemos observar en la figura 3-6. Claramente, hay una tendencia hacia el desarrollo de APIs públicas y todo indica que continuará en el próximo tiempo.

Figura 3-6



Fuente: Programmable Web (2019)

Higginbotham (2015) recopila distintas causas para el surgimiento de la denominada "economía de las APIs":

- Mayor demanda: las computadoras de escritorio, los teléfonos celulares y cada vez más dispositivos de uso hogareño pueden conectarse a internet e integrarse con otros sistemas.
- Simplicidad: muchas de las APIs web actuales usan *Hypertext Transfer Protocol* (HTTP) como medio para comunicarse. Este protocolo es ampliamente conocido y se caracteriza por ser simple, a diferencia de otras soluciones previamente utilizadas.
- Nuevos modelos de negocios: muchas compañías ofrecen servicios especializados para resolver necesidades específicas. Esta modalidad se denomina *Software as a Service*

(Saas). De esta manera, al momento de diseñar una nueva aplicación, es posible integrar servicios de terceros a través de sus APIs.

Probablemente, una de las primeras empresas que notó la importancia de las APIs es Amazon. Mason (2017) afirma que, aproximadamente en 2002, Jeff Bezos envió esta serie de puntos a sus empleados.

- De ahora en adelante, todos los equipos expondrán los datos y la funcionalidad de sus servicios a través de interfaces. Los sistemas deben comunicarse entre sí a través de estas interfaces.
- La única forma permitida de comunicación entre procesos es vía interfaces y a través de la red.
- No importa la tecnología usada por cada equipo.
- Cada equipo debe planificar y diseñar sus interfaces con el objetivo de ser expuestas al mundo exterior. Sin excepciones.
- Cualquier empleado que no haga esto será despedido.

Actualmente, Amazon provee decenas de APIs de una amplia variedad de servicios propios. En el caso de Amazon Web Services, los mismos van desde infraestructura hasta *machine learning*.[7]

[7] https://aws.amazon.com/es/products

4. Microservicios

En Arquitectura, hicimos un breve repaso de las ventajas y desventajas de una arquitectura orientada a microservicios. En este capítulo, vamos a profundizar la definición de servicio y cómo diseñarlo correctamente. Además, se analizará el rol preponderante que cumple la comunicación entre los servicios y se indagarán diferentes tópicos técnicos relacionados a este tipo de arquitectura como la seguridad, la infraestructura, el modelo de datos, entre otros. Sobre el final de este apartado, veremos que la implementación de cualquier arquitectura es inviable si la organización de los equipos no es adecuada y que los microservicios no son siempre la solución.

4.1. Definición de servicio

La correcta definición de cada servicio y sus responsabilidades son la clave para potenciar los beneficios de este tipo de arquitectura, pero también es uno de sus aspectos más complejos. Según Newman (2015), deben considerarse las siguientes características:

- Bajo acoplamiento: cambiar un servicio no debería requerir un cambio en otro. La gran ventaja de los microservicios es que pueden evolucionar independientemente y con una mínima coordinación entre ellos. Adicionalmente, la interacción entre servicios debe ser lo más simple y reducida posible.
- Alta cohesión: las funcionalidades relacionadas deben estar en el mismo servicio. Esto es debido a que necesariamente van a cambiar en algún momento y es mucho más beneficioso modificar en un solo lugar que en varios. A su vez, esto evita complejas coordinaciones entre equipos responsables de distintos servicios.

Newman (2015) propone que, para cumplir estos objetivos, cada servicio debe representar un *bounded context.* Este concepto fue introducido por Eric Evans en 2003 y está relacionado al denominado *Domain-Driven Design*.[8] Para definir esta idea, primero debemos recordar que todos los sistemas, en última instancia, representan problemas del mundo real. Estos pueden ser una red social, un sistema administrativo o un sistema de mensajería como el del presente trabajo. Hay una gran variedad de tipos de problemas, o dominios, que pueden resolverse a partir de un software que los modele. A su vez, cada dominio de un problema específico está

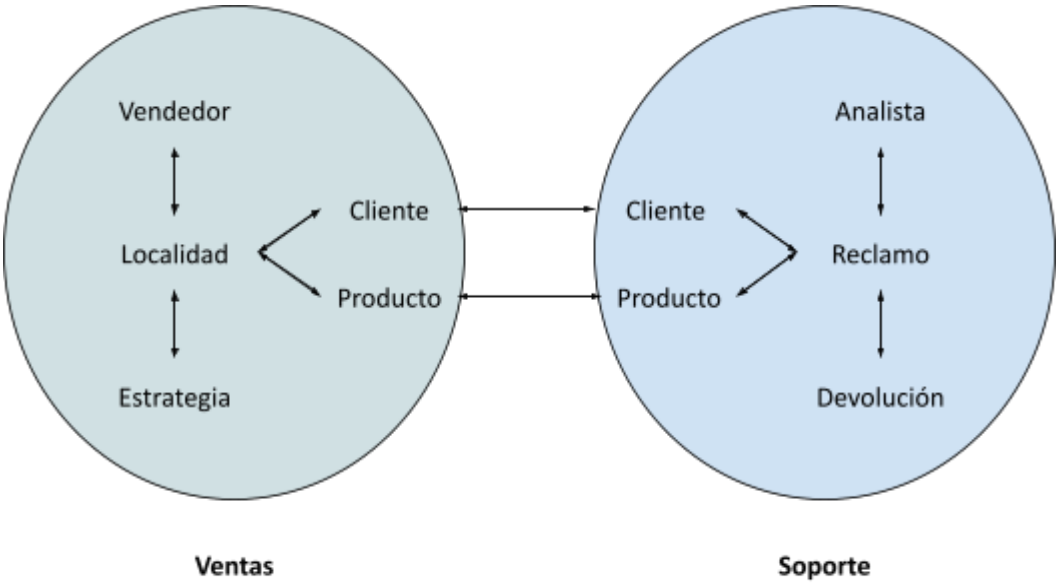
[8] https://martinfowler.com/bliki/DomainDrivenDesign.html

compuesto por *bounded contexts.* Los contexts son partes delimitadas que contienen modelos internos y modelos compartidos con otros.

En la figura 4-1, podemos observar un ejemplo de dos bounded contexts, uno relacionado al soporte y otro a las ventas de una empresa. Ambos comparten los modelos cliente y producto, y esto permite su interacción. Además, existen modelos internos como vendedor o reclamo que sólo son parte de un context y no se exponen a otros.

Por último, Newman (2015) resalta que no sólo tienen que tenerse en cuenta los datos para definir los bounded contexts de un dominio. Cada uno también debe representar un conjunto de responsabilidades de negocio. De esta forma, cada servicio está alineado a un objetivo de negocio particular.

Figura 4-1



4.2. Comunicación

Los microservicios, al ser aplicaciones independientes, deben comunicarse mediante mecanismos genéricos y agnósticos de la tecnología de cada uno. A lo largo de los años, se experimentaron distintas formas. En esta sección, vamos a repasar algunos de estos formatos y decidir cuál es el más adecuado para la aplicación del presente trabajo. Adicionalmente, se detallarán los diferentes problemas en la comunicación, y cómo abordarlos.

4.2.1. Application Programming Interface (API)

Una API es una interfaz expuesta por un software, que permite la interacción con otros sistemas. Esta interfaz provee un conjunto de operaciones disponibles y es una abstracción de la implementación interna de la aplicación. Particularmente, nos centraremos en las APIs web, donde se utiliza HTTP como protocolo de comunicación.

En Integrable, analizamos las causas del surgimiento de la denominada “economía de las APIs”. A continuación, vamos a ver algunas consideraciones para el diseño inicial. Lauret (2019) afirma que el diseño debe poner el foco en el usuario o consumidor. Esto puede traducirse en las siguientes preguntas:

- ¿Quién va a usar la API?
- ¿Qué van a poder hacer?
- ¿Cómo lo van a hacer?
- ¿Qué necesitan para hacerlo?
- ¿Qué información obtiene a partir de la API?

¿Por qué es importante prestarle atención al consumidor? En primer lugar, para facilitarle los flujos de trabajo y, así, reducir las posibilidades de que realicen un mal uso de nuestro sistema. Además, se agiliza el desarrollo de su integración y la consecuente adopción de la API.

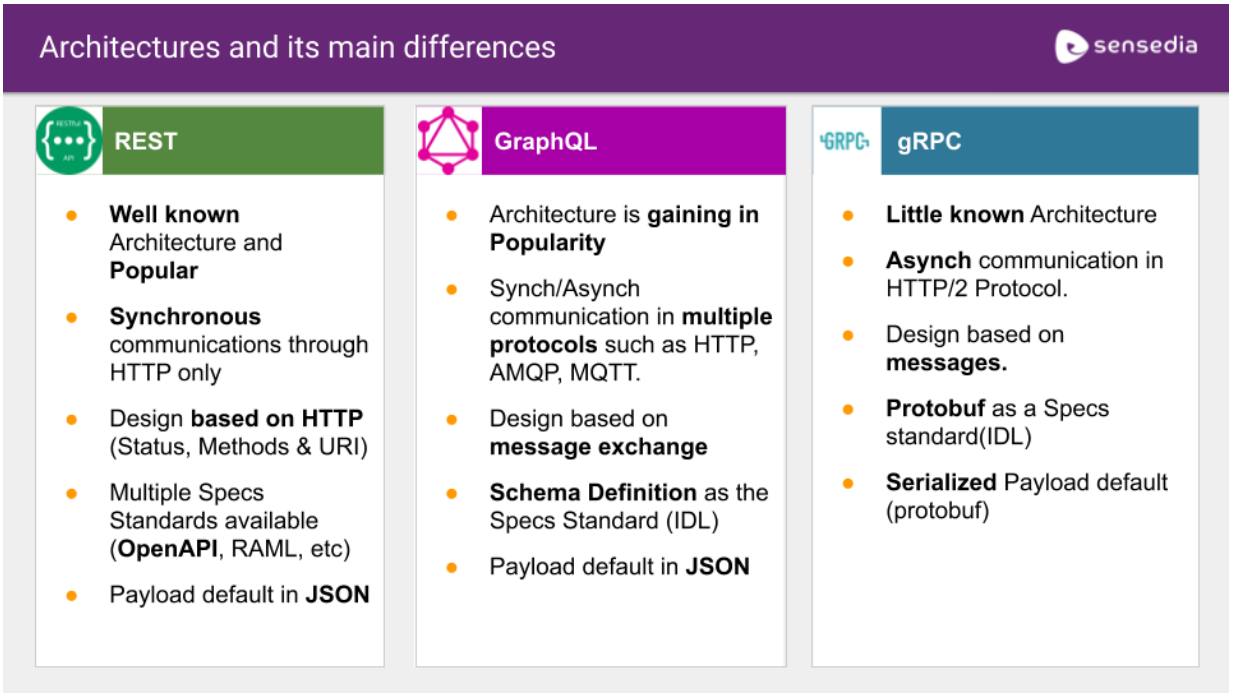
Conociendo el significado de las APIs y su importancia, podemos analizar distintas tecnologías relacionadas. Los formatos de comunicación más usados:

- JSON: es la representación que usa Javascript para definir objetos. Es legible para cualquier desarrollador, dinámico y con amplio soporte en los distintos lenguajes de programación.
- XML: hace unos años, era muy usado para interactuar entre servicios, pero su rigidez y su complejidad causaron que entrara en desuso.
- Binario: A diferencia de JSON y XML, no está basado en texto. Es decir, no es legible a simple vista. Además, requiere la existencia de un esquema compartido por el cliente y el servidor. Su gran ventaja es la optimización en términos de tamaño resultante y velocidad de decodificación y codificación de los mensajes.

Según Reselman (2020), las arquitecturas más comunes actualmente son:

- REST: está basado en recursos y operaciones sobre los mismos. En general, se usa en combinación con HTTP. De esta forma, los recursos son identificados mediante *Uniform Resource Identifier* (URI) y las operaciones son los métodos disponibles en este protocolo. La comunicación es sincrónica.
- GraphQL: los clientes tienen la capacidad de definir exactamente la información que necesitan obtener desde la API, a diferencia de REST, donde los campos de los recursos son previamente establecidos. Está basado en un esquema definido y la comunicación puede ser sincrónica o asincrónica.
- gRPC: la comunicación es mediante un formato binario. Soporta comunicaciones bidireccionales y asincrónicas, ya que está implementado a partir de la versión 2 de HTTP. Es la menos popular de las tecnologías mencionadas.

Figura 4-2



Fuente: https://www.sensedia.com/post/apis-rest-graphql-or-grpc-who-wins-this-game

El enfoque predominante en la actualidad es diseñar APIs REST usando JSON como formato, aunque claramente el resto de las tecnologías emergentes mencionadas está creciendo en popularidad.

4.2.2. Representational State Transfer (REST)

Hace varios años, REST es un estándar de facto al momento de modelar una API. Es una arquitectura desarrollada por Fielding (2000) con las siguientes características:

- Separación entre cliente y servidor: cada uno tiene responsabilidades diferentes y puede evolucionar de forma autónoma.
- Sin estado: las interacciones entre cliente y servidor son autocontenidas. Es decir, individualmente contienen toda la información necesaria para realizarse. En caso de ser necesario información relacionada al contexto, la misma es enviada por el cliente en cada interacción con el servidor.
- *Cache*: los clientes pueden guardar las respuestas obtenidas y es responsabilidad del servidor indicar si las mismas pueden ser guardadas o no y bajo qué criterios. Esto reduce la cantidad de interacciones entre ambas partes.
- Sistema de capas: entre el cliente y el servidor pueden existir intermediarios, pero los mismos no afectan en absoluto la comunicación. Un servidor, a su vez, puede comunicarse con otros para procesar lo que el cliente necesita
- Interfaz uniforme: El cliente envía un identificador del recurso y el servidor devuelve una representación del mismo. A partir de la representación, el cliente tiene toda la información necesaria para comunicar un posterior cambio en el recurso. En pocas palabras, el servidor debe proveer todos los recursos disponibles y las acciones asociados a los mismos a través de hipervínculos.

Este último aspecto es el más representativo de una arquitectura REST. Es remarcable que, aunque las APIs REST suelen implementarse en conjunto con HTTP, no es un requisito obligatorio.

4.2.3. Hypertext Transfer Protocol (HTTP)

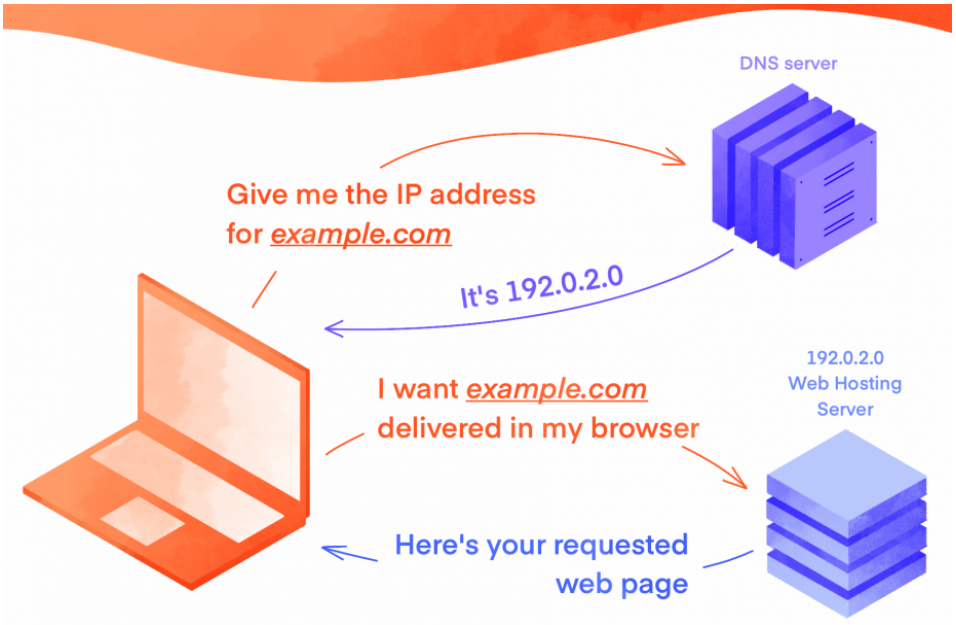
Es el protocolo que sustenta buena parte de lo que conocemos como *World Wide Web*. Está diseñado para facilitar el intercambio de información entre aplicaciones mediante

internet. Los navegadores web utilizan este protocolo para comunicarse con el servidor y, de esta forma obtener la página web que el usuario solicitó. Y no sólo cumple esta funcionalidad, cualquier acción que realicemos en la página web, como enviar un formulario o subir un archivo, se realiza a través de HTTP. A continuación, se detalla el funcionamiento de este protocolo, en particular de la versión 1.1, publicada por The Internet Society (1999).

A modo de ejemplo, enumeremos los pasos que realiza un navegador web cuando se solicita una página:

- Un usuario escribe una URL en la barra de dirección. Por ejemplo, www.example.com
- La URL es traducida a una IP. Esto se realiza a través del *Domain Name System* (DNS).
- El navegador web solicita esa página web a los servidores, donde está alojada la página web, mediante un *request*.
- El servidor devuelve un *response* al navegador, donde indica si fue obtenida exitosamente y el contenido de la página web.

Figura 4-3



Fuente: https://seranking.com/blog/www-vs-non-www/

La base de este protocolo son los requests y responses entre un cliente y un servidor. En este caso, el cliente es el navegador web y el servidor es el servicio de alojamiento de la página web. En la tabla 4-1 podemos observar los distintos campos de los requests y responses. Como observamos, comparten muchos campos, pero no todos.

Es importante resaltar que no se mantiene un estado, ni un contexto, entre requests. Cada interacción entre el servidor y el cliente es auto contenida. Sin embargo, muchas páginas web necesitan identificar a sus usuarios o mantener algún tipo de estado. En general, el cliente es responsable de enviar información adicional en los headers o en algún otro campo de cada request y, de esta forma, mantener ese estado. Por ejemplo, los navegadores webs realizan esta tarea automáticamente mediante *cookies.*

Tabla 4-1

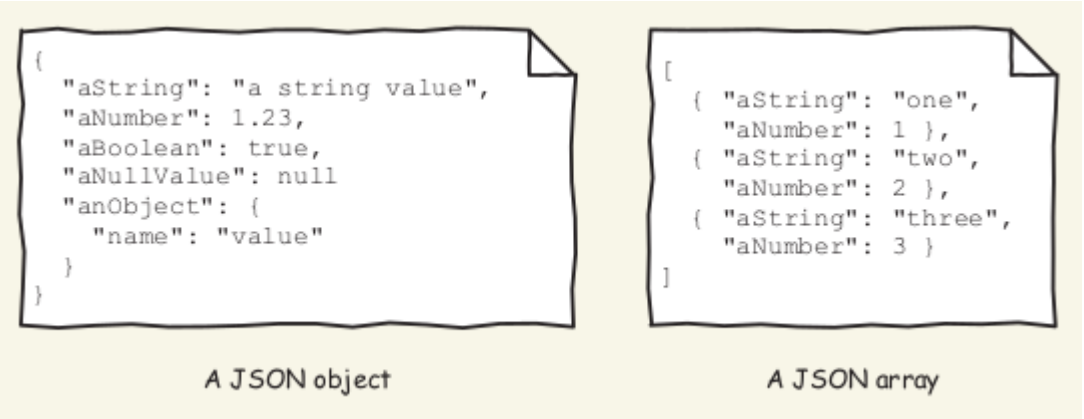
Campo	**Descripción**	**Valores posibles**
Method	El método define el tipo de acción que se va a realizar	GET, POST, PUT, DELETE, TRACE, CONNECT, HEAD, OPTIONS
Headers	Definen información relacionada al contexto del request.	Son pares clave/valor. Su uso no está restringido y pueden usarse para diversos casos de uso. Por ejemplo, el header Content-Type indica el tipo de dato que se quiere obtener.
URI (Uniform Resource Identifier)	Es la ruta parcial que se quiere obtener.	Por ejemplo, si accedemos a example.com/productos, la URI sería **/productos**. Además, en la URI, se puede enviar información adicional del request mediante parámetros.
Versión	Es la versión del protocolo HTTP que se usará en la comunicación	1.0 , 1.1 , 2.0
Body (opcional)	Es información asociada al request o response.	El tipo de dato del body no está definido. Puede utilizarse cualquier tipo, pero es necesario que el servidor y el cliente sean compatibles.
Status code (sólo en response)	El status code indica si el procesamiento del request fue exitoso o erróneo. Está definido por el servidor.	Los más comunes son: 200 (ok), 400 (bad request), 404 (not found), 500 (internal server error), entre otros.

4.2.4. Java Script Object Notation (JSON)

Es un formato de texto basado en cómo Javascript describe sus objetos. Buena parte de su popularidad se debe a la simpleza para usarlo y entenderlo. Además, es muy flexible, ya que en su versión original no tiene un esquema de campos definidos. Esta característica también lo hace menos robusto.

El formato JSON permite describir objetos que contienen pares nombre/valor desordenados y también listas conteniendo valores ordenados. El nombre del campo siempre debe ir entre comillas y separado por dos puntos de su valor. En la figura 4-4, se puede notar la flexibilidad y expresividad de este formato.

Figura 4-4



Fuente: Lauret (2019)

Tipos de valores disponibles:

- Boolean: *true* o *false*
- Número entero o con decimales
- Cadena de texto: es necesario que el valor esté entre comillas
- Objeto: está delimitado por llaves
- Lista: está delimitado por corchetes
- Nulo: es el valor *null*

Lauret (2019) afirma que el uso del formato JSON está bastante extendido no sólo para APIs, sino también como campos en bases de datos o archivos de configuración.

Prácticamente, todos los lenguajes tienen implementaciones nativas para decodificarlos. Como ya se mencionó, es relativamente fácil leerlo y escribirlo por parte de los desarrolladores y esto masificó su uso.

4.2.5. Ejemplo práctico de API Rest

Teniendo un conocimiento general de HTTP, REST y JSON, podemos combinarlos y aplicarlos a un ejemplo práctico. Supongamos que queremos modelar algunos procesos de una librería y, a partir de ellos, desarrollar un sistema de administración simple. Un requisito de este hipotético sistema es la posibilidad de hacer un seguimiento de libros prestados.

En primer lugar, definamos los recursos REST:

- Usuario (o cliente)
- Libro
- Préstamo

Para cada recurso, debemos establecer los campos que nos interesan, teniendo en cuenta la problemática que queremos resolver. No olvidemos que, en última instancia, el software es un modelado de la realidad enfocado en ciertos aspectos.

- Usuario: identificador, nombre, apellido, email
- Libro: identificador, título, autor, fecha de impresión
- Préstamo: identificador del usuario y del libro, fecha de devolución, estado (vigente o vencido)

Hay campos calculados automáticamente, como el identificador, y hay otros que se calculan a partir de otros campos, como el estado. También, existen campos ingresados por los operarios del sistema, como el título del libro o el nombre y apellido del cliente. Recordemos que todos los campos de los recursos son representados mediante JSON por la API. Esto incluye el request body y el response body de cada operación disponible. En la figura 4-5, podemos observar ejemplos de cada uno de los recursos.

Figura 4-5

El siguiente paso es enumerar las acciones permitidas en la API de cada recurso. En este ejemplo, aplican a todos los recursos, pero claramente esto puede ser diferente en otros casos de uso. En particular para los préstamos, nos interesa tener una forma de buscar los que estén vencidos y los que no.

- Crear
- Eliminar
- Modificar
- Búsqueda (sólo para préstamos)

Ya definidos los campos y las acciones, debemos traducirlos a HTTP mediante un method y una URI. Esta es la gran utilidad que tiene este protocolo, ya que estandariza las acciones comunes que poseen la mayoría de las APIs y también la respuesta a cada una por parte del servidor.

Los methods HTTP se usan, en general, de esta forma:

- GET: obtener un recurso
- POST: crear un recurso
- PUT: modificar un recurso
- DELETE: eliminar un recurso

Los status code más comunes son:

- 200: el recurso se obtuvo correctamente
- 201: el recurso se creó exitosamente
- 400: se envió información errónea al servidor al momento de modificar o crear un recurso
- 403: el acceso al recurso no está permitido
- 404: el recurso no existe
- 500: hubo un error al obtener, modificar o eliminar el recurso

Como notaron, cada status code está asociado a una acción específica. Esto no significa que no se pueda usar en otras, pero este es el uso habitual. Otro aspecto relevante son los rangos de status code porque permiten entender su significado. De esta forma, si nos topamos con un status code que no conocemos, podemos tener una idea general.[9]

Tabla 4-2

Rango	**Significado**
100 - 199	Respuesta informativa
200 - 299	Respuesta exitosa
300 - 399	Redirección
400 - 499	Error asociado al cliente
500 - 599	Error asociado al servidor

[9] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Finalmente, definimos cada *endpoint* de nuestra API. Nótese que, en el caso del buscador de préstamos, usamos parámetros definidos en la URI. Esta es una práctica habitual para enviar información adicional en los requests.

Tabla 4-3

Recurso	**Acción**	**Método**	**URI**	**Body**
Libro	Obtener	GET	/libros/{ID}	-
	Crear	POST	/libros	Información del libro
	Modificar	PUT	/libros/{ID}	Nuevos valores
	Eliminar	DELETE	/libros/{ID}	-
Usuario	Obtener	GET	/usuarios/{ID}	-
	Crear	POST	/usuarios	Información del usuario
	Modificar	PUT	/usuarios/{ID}	Nuevos valores
	Eliminar	DELETE	/usuarios/{ID}	-
Préstamo	Obtener	GET	/prestamos/{ID}	-
	Crear	POST	/prestamos	Información del préstamo
	Modificar	PUT	/prestamos/{ID}	Nuevos valores
	Eliminar	DELETE	/prestamos/{ID}	-
	Buscar	GET	/prestamos/buscar?estado={ESTADO}	-

Repasemos algunos ejemplos de errores que podemos tener y el status code correspondiente a cada uno. En todos los casos, deben especificarse más detalles en el body:

- Se intenta crear un préstamo con una fecha de devolución inválida. El status code debería ser 400 (Bad Request).
- Se intenta crear un préstamo con un ID de libro inexistente. Nuevamente, el status code que corresponde es 400 (Bad Request).

- Se quiere obtener un usuario con un ID inexistente. En este caso, es 404 (Not Found).
- Se quiere modificar un libro pero hubo un error inesperado del lado del servidor. Por ejemplo, se detectó un error en la base de datos. Corresponde un 500.

En conclusión, el flujo habitual de nuestro ejemplo sería:

- Creamos un libro: POST /libros
- Creamos un usuario: POST /usuarios
- A partir de los IDs obtenidos, creamos un préstamo: POST /prestamos
- Regularmente, vamos a buscar todos los préstamos que estén vencidos: GET /prestamos/buscar?estado=vencido

Como mencionamos, en cada paso puede haber diversidad de errores pero, a modo de ejemplificar, nos centramos en el flujo donde no hubo problemas del lado del cliente ni del lado del servidor.

4.2.6. Evolución de las APIs

Una API inevitablemente va a cambiar a lo largo del tiempo, ya sea por nuevas funcionalidades o por mejoras a las existentes. Podemos diferenciar dos tipos de cambios:

- Incrementales: son cambios menores de la API donde los clientes actuales no se ven afectados. Un caso típico es agregar un campo a la respuesta.
- Rotundos: denominados *breaking changes*. En este caso, los clientes deben modificarse porque pueden dejar de funcionar correctamente. Por ejemplo, esto sucede si cambiamos algún tipo de dato o eliminamos un campo en la respuesta.

Lauret (2019) detalla los breaking changes posibles en las APIs:

- Resultado de las operaciones:
    - Renombrar o mover cualquier campo
    - Eliminar un campo obligatorio
    - Cambiar el tipo de dato, la representación o el significado de un campo
- Entrada y los parámetros de las operaciones:
    - Renombrar o mover cualquier campo
    - Eliminar cualquier campo

    - Transformar un campo opcional en obligatorio o crear nuevos campos obligatorios
    - Cambiar el tipo de dato, la representación o el significado de un campo
    - Modificar las restricciones de cualquier campo
- Mensajes de error o informativos: además de responder recursos específicos, la API puede responder mensajes que indiquen detalles de los errores o informen si la operación fue exitosa. Estos mensajes deben estar estandarizados y los clientes se apoyan en los mismos para entender mejor lo que sucedió.
- Flujos: hay procesos que involucran varios pasos en la misma o en distintas APIs. Agregar, modificar o eliminar etapas en ellos suele causar problemas en los clientes.
- Acuerdos: todas las APIs tienen convenciones implícitas. Por ejemplo, puede existir una lista que, por defecto, está ordenada según cierto criterio. Los clientes tienen en cuenta estas convenciones y probablemente dependan de las mismas para funcionar adecuadamente.

Es importante comprender el rol que tienen los clientes al momento del diseño inicial y durante los posteriores cambios. Una API sólo será exitosa si cada decisión es pensada del lado del cliente y se considera su posible impacto.

Versionado de APIs

En algunos escenarios, introducir breaking changes es la mejor solución, ya sea porque la API se torna muy difícil de comprender o porque realmente es imposible adaptar el cambio al esquema actual. Richardson (2018) plantea que la mejor estrategia para abordar este tipo de modificaciones es mediante el versionado semántico. El mismo consiste en una serie de reglas que especifican cómo usar y cambiar las versiones de acuerdo a la magnitud de la modificación.

Cada versión consiste de parte, como vemos en la figura 4-6:

- Major: cambios incompatibles en la API
- Minor: mejoras que mantienen la retrocompatibilidad
- Patch: pequeños cambios retrocompatibles donde se soluciona algún bug

Figura 4-6

La API es responsable de exponer la versión usada mediante headers, URI o algún otro mecanismo del protocolo HTTP. De esta forma, el cliente conoce los cambios de versiones e incluso podría tener la posibilidad de elegir cuál quiere usar. En el caso de breaking changes, por un período de tiempo, conviven varias versiones de la API y eventualmente las versiones anteriores son desactivadas. Cada versión debe ser soportada por un tiempo prudencial para que cada cliente tenga la posibilidad de adaptarse.

4.2.7. Tipos de interacción

Como vimos anteriormente, es recomendable que cada servicio sea lo más autónomo posible del resto, pero, por otra parte, la existencia de dependencias es inevitable. Si estas dependencias no se administran correctamente, podemos enfrentar el escenario donde problemas en un servicio afectan el funcionamiento del resto. En este apartado, vamos a analizar los distintos tipos de interacciones entre servicios y las consideraciones a tener en cuenta en cada una.

Richardson (2018) identifica dos dimensiones de la interacción entre servicios. La primera dimensión está relacionada a la cantidad de servicios involucrados en la comunicación. La segunda tiene que ver con la sincronicidad o asincronicidad de la interacción. El autor combina estas dos dimensiones y establece los siguientes tipos de interacción:

- Interacciones uno a uno:
    - Request/response: un servicio realiza un request a otro y espera la respuesta. El cliente supone que la misma se recibirá en un tiempo limitado y se bloquea hasta que pueda ser obtenida. Esto resulta en una fuerte dependencia y acoplamiento entre servicios.
    - Request/response asincrónico: Un cliente envía un request, que es respondido asincrónicamente. Es decir, el cliente no se bloquea porque supone que el otro servicio puede no responder en un tiempo adecuado.

    - Notificaciones dirigidas: un cliente envía un request y no se espera ninguna respuesta.
- Interacción uno a muchos:
    - Publicar/suscribir: un cliente publica un mensaje de notificación que puede ser consumido por varios servicios interesados. De esta forma, están desacoplados el emisor del mensaje y sus receptores.
    - Publicación con respuesta asincrónica: Un cliente publica un mensaje y espera la respuesta de los servicios suscritos durante un tiempo acotado.

Cada uno de nuestros servicios probablemente use una combinación de esta formas de interacción para comunicarse con el resto. La estabilidad y el correcto funcionamiento del sistema va a depender en gran parte de la correcta valoración de este aspecto. En particular, la diferenciación importante es si la interacción entre servicios es sincrónica o asincrónica. En la tabla 4-4, se analizan más en detalle estos dos tipos.

Tabla 4-4

Tipo	**Ventajas**	**Desventajas**
Sincrónica	El cliente conoce el resultado y puede accionar a partir del mismo.	El cliente debe esperar el resultado. Ambos servicios tienen que estar disponibles.
Asincrónica	No hay una dependencia fuerte entre servicios.	El tiempo que tardará el servidor en procesar el mensaje es incierto. El cliente no obtiene el resultado inmediatamente. Seguramente, sea necesario tener un servicio adicional que administre el envío y recepción de mensajes.

En la figura 4-6 podemos observar un ejemplo donde un cliente externo realiza un request al Servicio 1 y esté, a su vez, realiza otro al Servicio 2. Mientras el cliente externo espera una respuesta sincrónica con un resultado definido, el Servicio 1 intenta minimizar la dependencia

con el Servicio 2. Esto genera un dilema que debe abordarse para tener un sistema resiliente con servicios independientes. En este sentido, Richardson (2018) afirma que la comunicación sincrónica entre microservicios reduce la disponibilidad de los mismos y recomienda minimizar su uso. El autor recomienda dos formas de mejorar este aspecto:

- Replicar datos localmente: si el Servicio 1 tiene una copia de los datos que necesita del Servicio 2, puede consultar directamente esa copia. Es necesario tener en cuenta que los datos deben mantenerse actualizados mediante algún mecanismo fiable. Por otro lado, si la cantidad de datos es muy grande, la replicación podría ser impracticable. De una u otra forma, estos datos pueden no estar totalmente actualizados.
- Responder de forma parcial: el Servicio 1 podría responder parcialmente, sin tener en cuenta el resultado del Servicio 2. Esto genera una complejidad mayor en el cliente externo porque probablemente necesite requests adicionales para conocer el resultado final.

Figura 4-7

4.3. Resiliencia

Newman (2015) considera que los errores pueden y van a suceder en distintos niveles de los sistemas distribuidos. Cuestiones relacionadas al hardware, el software, la infraestructura o la red pueden fallar o funcionar lentamente, entre otros ejemplos. Por ello, es necesario adaptarse a este hecho y diseñar cada servicio teniendo en mente esta premisa. En este sentido, es relevante diferenciar errores intermitentes de errores generales y detectar si son a nivel red o a nivel servicio.

Newman (2015) propone distintas formas para mejorar la resiliencia general del sistema:

- Comunicación[10]
    - *Timeouts:* el servidor puede no responder en un tiempo adecuado. Un timeout permite definir un tiempo máximo de espera por parte del cliente. En caso de superarse, se considera que el servidor no es capaz de procesar el request en ese momento.
    - Reintentos: Para minimizar el impacto de errores intermitentes, un request puede intentarse múltiples veces. Usualmente, los reintentos no son inmediatos y se espera un tiempo definido entre cada uno. Las operaciones deben ser idempotentes para poder reintentarse de forma segura.
    - *Circuit breakers:* la duración total de los requests tiene un impacto directo en el uso de CPU y memoria del cliente. Como se mencionó, los timeouts y los reintentos son beneficiosos, pero también aumentan potencialmente esta duración. El circuit breaker aborda este escenario de la siguiente manera: si la cantidad de errores en un período de tiempo es muy alta, considera al servidor como no disponible. Cualquier request falla inmediatamente. De esta forma, la duración total de los requests no aumenta y el cliente no se ve afectado..
    - *Bulkheads:* Consideremos el caso donde el servidor está funcionando lentamente. Esto puede causar que todo los *threads* del cliente estén bloqueados porque esperan la finalización de los requests. La consecuencia directa es que el cliente, en algún momento, se quedará sin threads disponibles y dejará funcionar. Los *bulkheads* aíslan los requests que van dirigidos a cada servidor, creando un conjunto de threads independiente para cada caso. En consecuencia, un servidor, que funcione incorrectamente, no afecta al resto de la aplicación.
- Infraestructura
    - Dividir infraestructura: cada servicio debe tener un conjunto de servidores separado del resto y, a su vez, dentro de cada servicio, es necesario identificar subdivisiones basadas en el tipo de procesamiento de cada flujo de trabajo.
    - Escalamiento horizontal: si tenemos varios servidores con capacidad de procesar requests, reducimos la posibilidad de que errores en instancias puntuales afecten al servicio. No tenemos un único punto de falla.

[10] Denominamos cliente al servicio que realiza los requests y servidor al que los procesa.

- Diseño
    - Funcionalidad degradada: la interacción entre servicios es inevitable, pero en muchos casos es posible reducir la criticidad de la misma. Por ejemplo, si un servicio tiene una dependencia no disponible, puede limitar sus funcionalidades, en lugar de fallar.
    - Idempotencia: significa que las operaciones pueden aplicarse varias veces sin afectar el resultado final. Esto permite implementar reintentos.

Schmaus (2011) coincide en el riesgo que implica que un error en un servicio pueda propagarse al resto e identifica distintos principios para evitar este escenario:

- Cada servicio debe tener una acción correctiva automática en el caso de que una dependencia falle.
- Cada servicio debe ser capaz de mostrarnos lo que está sucediendo en este momento y lo que sucedió en el último periodo de tiempo.
- Un error en cualquier servicio no debe perjudicar la experiencia del usuario final.

4.4. Seguridad

La seguridad informática es fundamental en el diseño e implementación porque de ella depende la confianza de los usuarios y la reputación del sistema. Garg & Kohnfelder (1999) crearon un modelo para identificar amenazas durante la etapa de diseño. Lo denominaron S.T.R.I.D.E.[11] y está compuesto por:

- Suplantación de identidad: un atacante logra autenticarse como un usuario válido del sistema y actuar en su nombre.
- Corrupción de datos: se modifican datos del sistema o los usuarios de forma no autorizada.
- Repudio: usuarios, maliciosos o no, niegan una acción realizada y no hay posibilidad de verificar que esta afirmación sea cierta, debido a la falta de auditoría.

[11] Spoofing of user identity, Tampering with data, Repudiability, Information disclosure, Denial of Service, Elevation of privilege.

- Filtración de información: un atacante tiene acceso a información privada de los usuarios o de la compañía y la expone a individuos que no tienen permisos de obtenerla.
- Denegación de servicio: el objetivo es que el sistema no esté disponible o no sea usable por un período de tiempo. En general, se realizan mediante un uso excesivo de los recursos del sistema por parte de los atacantes.
- Elevación de privilegios: un atacante logra obtener privilegios superiores a los que le corresponden. Esto puede derivar en acciones dañinas e irreversibles aplicadas en el sistema.

Estas amenazas pueden transformarse en realidad por diversidad de razones: descuido por parte de los usuarios, fallas en el diseño inicial, vulnerabilidades en la implementación, entre otros. Es imprescindible tenerlas en cuenta en nuestro sistema y abordarlas individualmente y en conjunto. En general, el punto más débil del sistema es el que, en última instancia, lo termina comprometiendo. Además, es importante notar que los atacantes suelen usar una combinación de estas amenazas para lograr sus objetivos.

En este apartado, vamos a centrarnos en la seguridad de las APIs. Es común que cada una deba procesar operaciones de usuarios con distinto nivel de permisos. Incluso, es posible que otros sistemas se integren a la API y realicen operaciones en nombre de un usuario. Estas cuestiones son resueltas mediante la autenticación y la autorización. Madden (2020) las considera fundamentales porque permiten lograr:

- Confidencialidad: la información sólo debe ser leída por quién está autorizado a hacerlo.
- Integridad: prevenir la creación, modificación o eliminación de información por personal no autorizado.
- Disponibilidad: sólo usuarios legítimos pueden acceder al sistema y lo pueden hacer siempre que lo necesiten.

Autenticación

Es el proceso que verifica si un usuario es quien dice ser. En otras palabras, el usuario indica quién es y el sistema verifica que esta afirmación sea cierta.

Los factores más comunes de autenticación son:

- Algo que el usuario conoce (por ejemplo, una contraseña)
- Algo que el usuario tiene físicamente (una llave o un dispositivo físico)
- Algo que el usuario es (huella digital o algún factor biométrico)

Todo estos métodos son falibles. La contraseña puede ser muy fácil de deducir, la llave puede perderse y la huella digital puede ser erróneamente analizada. Por ello, últimamente, los sistemas solicitan dos o más factores de autenticación a sus usuarios para minimizar este riesgo. Esto se denomina *Multi-Factor Authentication* (MFA).

Identificar efectivamente a los usuarios de nuestro sistema es útil para:

- Registrar las acciones que realizó el usuario, por cuestiones de auditoría
- Definir los permisos que tiene dentro del sistema
- Diferenciar usuarios reales de usuarios anónimos
- Limitar la cantidad de acciones que puede realizar el usuario en cierto rango de tiempo

Autorización

La autenticación no alcanza por sí sola para tener un sistema seguro, también es necesario asignarle permisos a cada usuario para controlar quién tiene acceso a cierta información y qué acciones puede realizar.

Madden (2020) identifica dos formas de asignar permisos a usuarios:

- Basado en la identidad. El permiso es asignado individualmente a partir de la identidad del usuario. Por ejemplo, cualquier red social necesita controlar que un usuario sólo pueda realizar acciones en su propio perfil, como enviar mensajes o publicar imágenes. En otros casos, una práctica usual es agrupar a los usuarios en grupos y asignar permisos determinados a cada uno. De esta forma, los nuevos usuarios se suman a grupos ya existentes y se facilita la operatoria.
- Basado en la capacidad. Un *special token* o *key* tiene permisos para realizar acciones determinadas. Con el auge de las APIs, es común que los sistemas se comuniquen entre sí a través de ellas. Por ejemplo, supongamos que se quiere compartir una publicación de un sitio web en una red social. El sitio necesitaría permisos para poder

usar la API de esta red social en nombre del usuario. Esto puede realizarse mediante un token que lo autorice a realizar esta acción específica por única vez.

4.5. Infraestructura

El lanzamiento de *Elastic Compute Cloud* (EC2), por parte de Amazon, en 2006, modificó radicalmente la administración de la infraestructura.[12] Ya no era necesario tener servidores propios y configurarlos para obtener nuevos recursos. A sólo unos clicks de distancia y en una fracción de tiempo, Amazon proveía servidores listos para usar. Estos nuevos servidores residían en la denominada nube. Con el paso del tiempo, nuevos proveedores como Google y Microsoft se sumaron y surgieron servicios auto administrados que cubrían distintos casos de uso a nivel técnico.

La nube jugó un rol fundamental en el cambio tecnológico del último tiempo. El National Institute of Standards and Technology (2011) considera las siguientes características de la nube:

- Servicio bajo demanda: los consumidores pueden obtener recursos computacionales de forma automática, en caso de necesitarlo.
- Amplio acceso a través de la red: los servicios están expuestos mediante la red y son accedidos mediante mecanismos estándar.
- Recursos compartidos: el proveedor administra la forma de asignar los recursos disponibles a cada consumidor.
- Elasticidad: los recursos pueden ser asignados y liberados según la demanda de procesamiento.
- Servicio medido: el proveedor determina límites de uso para cada consumidor. Además, facilita el monitoreo de los aspectos principales de cada recurso.

Como observamos, muchas de las complejidades de la infraestructura son ajenas al consumidor. Este considera que siempre es posible pedir más recursos y no conoce el detalle

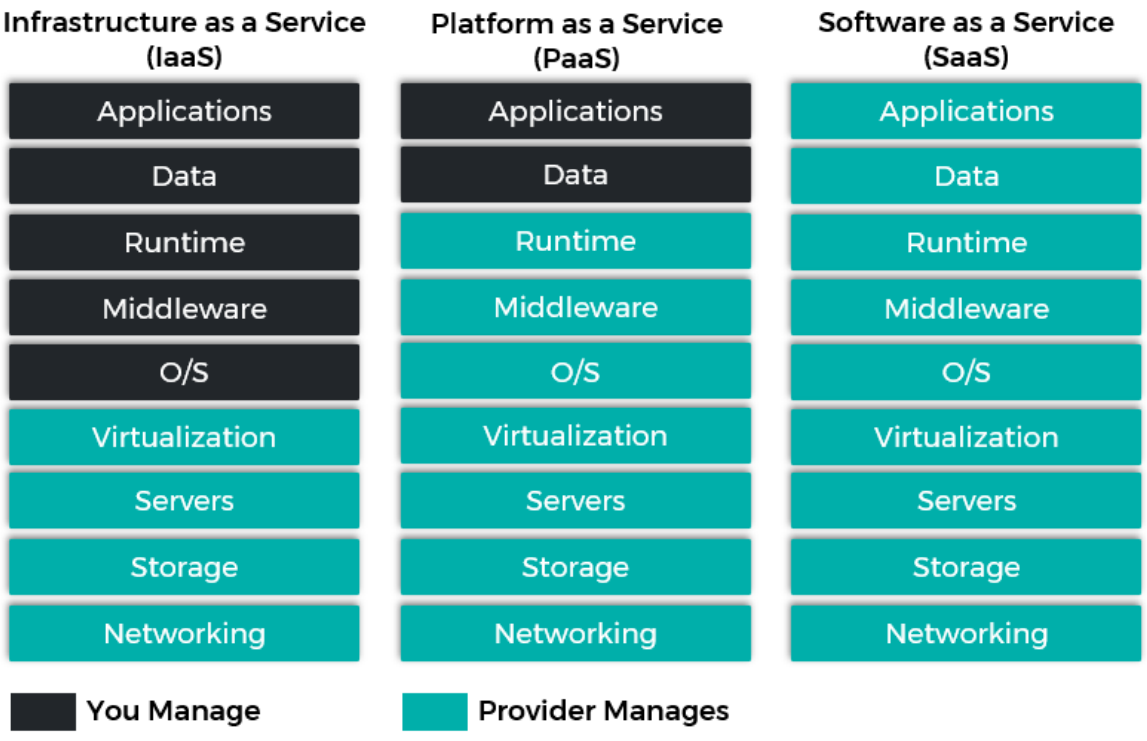
[12] https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

de la asignación de los mismos. Principalmente, no debe lidiar con servidores propios y toda la complejidad operativa que eso implica.

Adicionalmente, el National Institute of Standards and Technology (2011) enumera distintos tipos de modelo de servicios para la nube:

- Software as a Service (SaaS): los clientes tienen acceso directo al software. El proveedor se encarga de todo lo relacionado a su correcto funcionamiento.
- Platform as a Service (PaaS): se provee un entorno que facilita el desarrollo de aplicaciones. El consumidor se encarga de la aplicación propiamente dicha y de configurar los componentes necesarios para su operatoria.
- Infrastructure as a Service (IaaS): el proveedor ofrece recursos básicos como red, almacenamiento, servidores, entre otros. El consumidor es responsable de administrar el sistema operativo, las aplicaciones y demás aspectos relevantes.

Figura 4-8



Fuente: https://www.inap.com/blog/iaas-paas-saas-differences

4.6. Monitoreo

Las aplicaciones distribuidas son complejas por naturaleza porque constan de diversidad de componentes que interactúan entre sí. Los componentes y la comunicación entre ellos puede tener errores de diversa índole y naturaleza. Por ello, es imprescindible tener un monitoreo constante para poder identificar y resolver los distintos problemas.

El monitoreo está basado en métricas y logs. Estos representan distintos indicadores y eventos del sistema. Reichert (2018) los define de la siguiente manera:

- Logs: representan eventos que sucedieron durante la ejecución de la aplicación. Cada tipo de evento tiene una información asociada diferente. Suelen dividirse según su criticidad: informativos, errores, advertencias, *debug* y alertas. En esencia, permiten ver cronológicamente los distintos hechos importantes que ocurrieron.
- Métricas: son indicadores de algún aspecto del sistema en un momento determinado. De esta manera, se suelen representar mediante series de tiempo. Cada valor tiene un contexto específico que facilita su posterior análisis. Por ejemplo, podemos definir si la métrica medida surgió a partir de una operación exitosa o no. Mientras los logs son esporádicos, las métricas se recolectan en cada instante.

S. Fowler (2017) define las siguientes características de un sistema monitoreado adecuadamente:

- Sus métricas clave están identificadas y monitoreadas, tanto a nivel infraestructura como aplicativo
- Tiene *logs* apropiados que reflejan precisamente el estado reciente del servicio
- Los distintos gráficos de las métricas son fáciles de interpretar y contienen todas las métricas clave
- Sus alertas son representativas y tienen acciones de mitigación claras
- Hay un equipo o una persona de guardia responsable de responder cualquier incidente
- Hay un procedimiento claro, definido y estandarizado que sigue la guardia durante el incidente.

Por su parte, Richardson (2018) detalla una serie de recomendaciones:

- Métricas de la aplicación centralizadas: las instancias de cada servicio deben enviar sus métricas a un sistema unificado que tenga capacidades de graficarlas y generar alertas.
- Agregación de logs: al igual que las métricas, un sistema debe recopilar los logs y permitir búsquedas y alertas.
- Health check: el servicio expone un endpoint en su API que indica su estado. De esta forma, durante el deployment o en el escalamiento horizontal, es posible saber si una instancia es capaz de procesar tráfico.
- Distributed tracing: como es sabido, suelen existir flujos que involucran varios servicios. En otras palabras, hay varios requests asociados a la misma operación. Los errores en estos flujos son difíciles de analizar porque implican detectar todos estos requests y relacionarlos de alguna forma. Una manera de hacerlo es mediante un ID que defina unívocamente la operación y todos sus requests. Un sistema se encarga de recopilar esta información desde todos los servicios y permitir el posterior análisis.
- Exception tracking: las excepciones significan un error importante en la aplicación y deben tener un adecuado seguimiento mediante logs o un servicio diferenciado.

Un requisito implícito que atraviesa la temática del monitoreo es la necesidad de tener un punto de comparación que indique si los valores actuales son normales o no. Se suele comparar el estado actual con algún estado previo donde sabemos que la aplicación funcionaba correctamente y estaba en la misma versión. En el caso donde la aplicación es nueva y no tiene información histórica, se deben ejecutar pruebas de carga que servirán como base para definir el estado saludable. Las pruebas de carga, además, nos permiten saber si una nueva versión aplicativa tiene problemas de rendimiento antes de que llegue a producción.

4.7. Modelo de datos

Kleppman (2017) considera que los modelos de datos son probablemente la parte más importante del desarrollo de software porque tienen un profundo efecto en cómo se diseña el software y cómo se piensan los problemas que surgen. En este apartado, vamos a enumerar los modelos más populares en la actualidad para guardar y consultar datos. Como suele suceder, cada uno tiene ventajas y desventajas, y se adapta mejor a ciertos casos de uso.

4.7.1. Relacional

Surgió en 1970 a partir de una propuesta de Edgar Codd[13]. En principio, se pensó que no iba a ser factible implementarlo de forma eficiente, dadas las limitaciones de hardware de la época. Sin embargo, el aumento exponencial de la potencia computacional hizo posible que, en sólo unas décadas, sea el modelo predominante. A lo largo del tiempo, demostró ser flexible y aplicable a una gran diversidad de aplicaciones. El reconocido *Structured Query Language* (SQL) está basado en el modelo relacional.

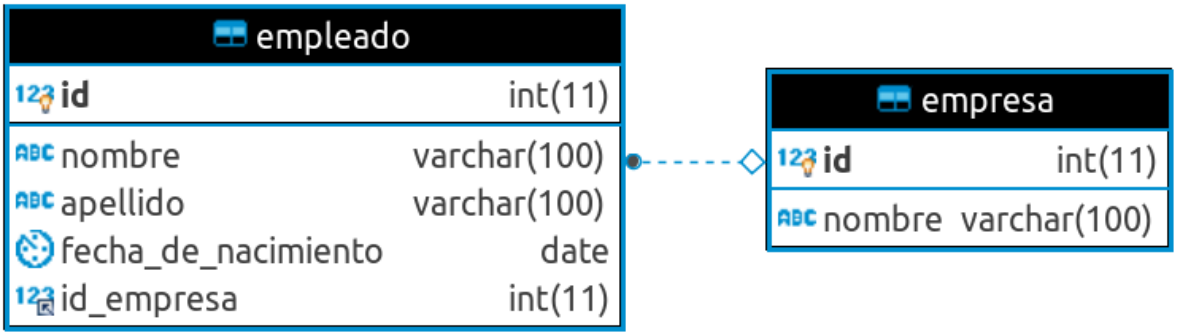
El concepto principal de este modelo es la relación (o tabla), que tiene las siguientes características:

- Contiene un conjunto de registros.
- Un esquema define su nombre, los atributos que posee y el tipo de dato de cada uno de ellos.
- Todos los registros poseen valores definidos para los atributos del esquema.
- Pueden especificarse distintas restricciones que todos los registros deben cumplir.

En la figura 4-9, vemos un esquema de tablas muy simple, donde se representan empleados y sus respectivas empresas.

[13] https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

Figura 4-9



La tabla Empleado hace referencia a la tabla Empresa mediante un identificador y, de esta forma, no repetimos la información de la empresa para cada uno. Esto evita duplicar datos en diferentes tablas y permite agregar validaciones que aseguran la integridad de datos. Es posible implementar una amplia variedad de restricciones, como la mencionada, y este es uno de los grandes beneficios del modelo relacional.

4.7.2. No relacional

En el siglo XXI, el modelo relacional comenzó a mostrar sus limitaciones y surgieron alternativas que resolvían de mejor forma los problemas modernos. Con la aparición de la Web 2.0[14], las aplicaciones comenzaron a generar cada vez más datos a partir de sus usuarios. Y este no sólo fue un cambio cuantitativo, también lo fue cualitativamente. Había una mayor necesidad de modelar relaciones complejas en los datos y de consultarlos de una manera diferente.

Kleppman (2017) considera la siguientes causas para el nacimiento de las bases de datos NoSQL:

- Una tendencia hacia la escalabilidad horizontal que las bases de datos relacionales no podían alcanzar fácilmente. Muchas aplicaciones debían soportar una gran cantidad de datos y una alta tasa de modificación de los mismos.
- Una amplia preferencia por software libre y abierto por sobre las bases de datos comerciales
- Ciertas consultas especializadas no eran posibles o eran muy complejas de implementar en el modelo relacional

[14] https://es.wikipedia.org/wiki/Web_2.0

- Frustración por las restricciones que imponía el modelo relacional y deseo de tener un modelo más dinámico y expresivo.

noSQL abarca una gran variedad de modelos diferentes. A continuación, mencionamos los más relevantes:

- Documento: la información se agrupa en documentos que suelen ser autocontenidos. Estos documentos no tienen un esquema rígido y pueden cambiar a lo largo del tiempo. A diferencia del modelo relacional, la lógica de estos cambios y la retrocompatibilidad reside en la aplicación.
- Grafo: se enfoca en las distintas relaciones entre las entidades y, como su nombre lo indica, consiste en vértices que se conectan entre sí a través de aristas. El modelo relacional tiene limitaciones al expresar una gran cantidad de este tipo de relaciones.
- Clave-valor: los datos son guardados y obtenidos mediante un identificador. Sus capacidades de búsqueda son reducidas, pero brindan una escalabilidad muy alta.

4.7.3. Transacciones

Hay diversos problemas que pueden ocurrir durante la interacción entre la aplicación y la base de datos. Podemos mencionar los siguientes:

- Un error de hardware afecta a algunas de las dos.
- La aplicación deja de funcionar en el medio de una serie de operaciones a causa de algún bug.
- La red no funciona correctamente.
- Un proceso lee datos que no tienen sentido porque hay una modificación en curso en la base de datos.

Estas son algunas de las cuestiones que pueden generar problemas difíciles de detectar y analizar. Kleppmann (2017) plantea que, durante décadas, las transacciones fueron la solución porque permitían simplificar los tipos de problema previstos. En pocas palabras, una transacción consiste en agrupar lecturas y escrituras a objetos de la base de datos en una unidad lógica. El autor, además, enumera un conjunto garantías que ofrecen las transacciones:

- Atomicidad: asegura que todas las operaciones se ejecutan como una sola unidad. Pueden fallar todas o ninguna, pero no hay errores parciales.
- Consistencia: implica que hay varias afirmaciones sobre los datos que siempre deben ser verdaderas. Por ejemplo, la cantidad disponible de un producto nunca puede ser negativa. De esta forma, si las transacciones realizan modificaciones válidas a una base de datos consistente, esta siempre va a estar en estado consistente.
- Aislamiento: permite que muchos clientes puedan modificar la misma información, sin causar problemas de concurrencia. Es decir, garantiza que, cuando varias transacciones terminaron, el resultado sea el mismo que si hubieran sido ejecutadas secuencialmente.
- Durabilidad: asegura que las modificaciones de una transacción exitosa no se van a perder, incluso si existen problemas a nivel hardware o software.

La mayoría de las bases de datos relacionales implementan transacciones, tal como las definimos. Sin embargo, las transacciones son complejas de implementar y requieren una fuerte coordinación. Estas limitaciones causan que las bases de datos no relacionales opten por tener transacciones con menos garantías o directamente no las soporten. Como suele suceder, una mejora en un aspecto va en detrimento de otro. En este caso, las bases de datos no relacionales son escalables y flexibles, pero no tienen la fuerte consistencia de las relacionales.

4.7.4. Consistencia

Las aplicaciones monolíticas tienen la posibilidad de delegar el manejo de las transacciones a la base de datos relacional que utilicen. En cambio, los microservicios tienen bases de datos independientes y el mantenimiento de la consistencia a través de ellos debe afrontarse a nivel aplicativo. Este es uno de los aspectos más desafiantes de este tipo de arquitectura.

Richardson (2018) propone el uso de *sagas* para abordar esta problemática. Los define como mecanismos que aseguran la consistencia entre microservicios, usando una secuencia de transacciones locales coordinadas de forma asincrónica. Cada flujo que actualiza datos en diferentes servicios debe tener un saga asociado. Esta consistencia, claro está, no tiene tantas garantías como la soportada por los modelos relacionales que vimos anteriormente.

Imaginemos un esquema simple donde existen tres servicios: órdenes de compra, clientes e inventario. El primero tiene la responsabilidad de manejar todos los cambios de estado de la orden, el segundo administra la cantidad disponible de cada producto y el tercero se encarga de los clientes. Al momento de realizar una nueva compra, deben verificarse una serie de condiciones en el cliente y en el inventario. En este ejemplo, tenemos cinco transacciones locales:

- 1- Crear orden con estado pendiente en Orden.
- 2- Verificar *stock* disponible en Inventario.
- 3- Verificar cliente
- 4- Reservar *stock* en Inventario.
- 5- Aprobar orden en Orden.

Cada transacción es local porque es interna de cada servicio. Al finalizar cada una, se emite un mensaje para que empiece la siguiente. El problema es cuando falla una de estas operaciones. Por ejemplo, si no tenemos la cantidad disponible del producto que se compró o el cliente no tiene la capacidad de realizar la compra. Para estos escenarios, lo que se propone son operaciones compensatorias. Todas las transacciones locales, exceptuando la última y las que son de sólo lectura, deben tener este tipo de operación. Si sucede un error, las operaciones compensatorias de las transacciones previas se ejecutan en orden inverso y deshacen todo lo realizado. En la tabla 4-5, observamos esta información.

Tabla 4-5

Transacción local	**Operación compensatoria**	**Detalles**
Crear orden pendiente	Rechazar orden	
Verificar stock disponible	-	Sólo lectura
Verificar cliente	-	Sólo lectura
Reservar stock	Liberar stock	
Aprobar orden	-	Última transacción

En conclusión, los sagas son un mecanismo que mantiene la consistencia a lo largo de los servicios. Su naturaleza asincrónica nos otorga resiliencia durante el avance de las transacciones locales y también en su desandar, en caso de error. De todas formas, se debe remarcar la posibilidad de que, por un período de tiempo, los servicios no sean consistentes. Si es necesaria una consistencia más robusta, las operaciones deben convivir en el mismo servicio para ser capaces de usar la misma base de datos.

4.8. Organización

A diferencia del resto de este trabajo, en este apartado se analizarán algunos tópicos relacionados a la organización de los equipos. El aspecto organizacional es fundamental para diseñar y desarrollar una aplicación exitósamente, usando microservicios o cualquier otro tipo de arquitectura. En este sentido, Conway (1968) afirma:

> "Las organizaciones que diseñan sistemas (definidos en términos generales) están limitadas a producir diseños que son copias de la estructura comunicacional de cada organización."

Con respecto a cómo se organizan los equipos relacionados a la informática en el contexto empresarial, Newman (2019) considera:

> "Históricamente, las organizaciones estuvieron estructuradas alrededor de sus competencias principales. Los desarrolladores de software estaban en equipos junto a otros desarrolladores de software. De la misma forma, los *testers* y los analistas de bases de datos integraban grupos, cuyos miembros tenían conocimientos similares. Un cambio en el software involucraba a todos estos equipos. Progresivamente, estamos viendo que esto está cambiando. Muchas empresas actuales optan por equipos independientes y autónomos, que están enfocados en áreas de producto, en lugar de tecnologías específicas."

En pocas palabras, más allá de que los aspectos técnicos se cumplan rigurosamente, si la organización no se adapta, irremediablemente van a surgir complicaciones. Como mencionamos anteriormente, la agilidad es un aspecto clave en el desarrollo de aplicaciones modernas, ya que permite innovar y competir exitosamente en el mercado.

A lo largo de los últimos años, muchas compañías se enfrentaron al dilema de ser ágiles a gran escala. Más allá de que existían metodologías para equipos, como Kanban o Scrum, estas tuvieron que ser reformuladas para ser aplicables a esquemas de mayor tamaño. De esta forma, surgieron distintas alternativas, entre las cuáles podemos mencionar el modelo Spotify [15], Scrum@Scale, Large Scale Scrum, entre otros. Sin embargo, más allá de que son útiles como punto de partida, cada organización debe hacer una análisis particular de su contexto tecnológico, humano y de negocio para poder llegar a un cambio beneficioso, y no simplemente copiar lo que otra empresa hizo.

En este sentido, Atlassian (2020) plantea distintos puntos que mejoran la agilidad, más allá de la metodología que se decida usar:

- Cada integrante del equipo debe tener un rol definido. Asignar más de una responsabilidad a cualquiera de ellos es perjudicial porque no le permite enfocarse.
- Los productos aportan valor a los usuarios de diversas maneras. Los equipos deben estructurarse a partir de estos valores y ser capaces de administrarlos de principio a fin.
- La agilidad comienza a nivel equipo. Todos deben seguir una metodología ágil definida, ya sea Kanban, Scrum o alguna otra.
- La transición a un esquema ágil lleva tiempo y debe ser planificada. Extender la transición es mejor que forzar a todos los equipos a parecer ágiles.
- Las dependencias entre equipos siempre van a existir y, debido a esto, es importante detectarlas y abordarlas, ya sea mediante la unión de equipos interdependientes o planteando procesos de mejora claros.
- Todas las personas involucradas deben estar convencidas de los beneficios del cambio. Esto incluye todos los niveles de autoridad.
- Las compañías y sus integrantes son únicos. Cualquier metodología debe ser adaptada al contexto donde se implementa. En conjunto con el cambio, debe crearse una cultura que lo potencie.

[15] En 2012, Spotify publicó la forma en que organiza sus equipos. https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf

4.9. Conclusión

A lo largo de este capítulo, hemos visto las enormes potencialidades de los microservicios, aunque también notamos sus múltiples desafíos. Y no sólo sucede con los microservicios, prácticamente todas las decisiones técnicas tienen aspectos positivos y negativos. Dicho de otra forma, “no hay balas de plata”.[16]

Newman (2019) enumera distintas razones para optar o no por este tipo de arquitectura, como vemos en la tabla 4-6. En particular, la capacidad de los microservicios para adaptarse a los requerimientos de las aplicaciones modernas es lo que fundamenta la elección de esta arquitectura para la aplicación de la presente tesis.

Tabla 4-6

Razones para adoptar microservicios	**Razones para no hacerlo**
Mejorar la autonomía de los equipos	No son claras cuáles serían las responsabilidades de cada microservicio
Reducir el tiempo de desarrollo y puesta en producción de cambios aplicativos	La empresa está en una etapa inicial, donde el equipo es reducido, y es necesario que la aplicación evolucione rápidamente.
Aumentar la robustez del sistema	Si la aplicación es instalada por el cliente, como es el caso de aplicaciones de escritorio.
Soportar el crecimiento de carga en el sistema de forma eficiente	
Mejorar el manejo de los equipos y permitir que crezcan más rápido	
Tener la posibilidad de usar nuevas tecnologías	

[16] https://www.cgl.ucsf.edu/Outreach/pc204/NoSilverBullet.html

5. Diseño de la solución

En este capítulo, se avanzará con el diseño del sistema de mensajería. Muchos de los tópicos ya vistos se van revisar nuevamente y aplicar a este caso de uso concreto. Esto nos va a permitir tener una visión más clara de cada uno de estos temas. A su vez, se conocerán más en detalle los distintos desafíos que aparecen al momento de implementarlos.

En primer lugar, se definen las funcionalidades básicas del sistema, los servicios y sus respectivas API. Luego, se analizan las interacciones entre servicios y su seguridad, abordando los potenciales problemas de escalabilidad y disponibilidad. La segunda mitad de este capítulo consta de consideraciones sobre la infraestructura y el monitoreo, y finaliza con una prueba de concepto, donde se aplica buena parte de lo aprendido a lo largo de este trabajo.

5.1. Diseño de servicios

El punto de partida del análisis es el conjunto de funcionalidades que queremos soportar en la aplicación. Particularmente, las principales son:

- Creación, modificación y eliminación de usuarios
- Creación, modificación y eliminación de grupos de usuarios
- Recepción y envío de mensajes de texto, que pueden tener imágenes asociadas
- Búsqueda de mensajes de texto por distintos criterios

Como primer paso, indaguemos más a fondo estas funcionalidades y agrupemoslas por casos de uso. Este mayor detalle nos va a posibilitar tener una visión clara de las distintas entidades y flujos de nuestro sistema. En la tabla 5-1, podemos observar el conjunto de operaciones que van a ser la base para el armado de los diferentes servicios y sus respectivas APIs. Adicionalmente, en la tabla 5-2, analizamos las entradas y salidas que corresponde a cada una.

En Definición de servicios, enumeramos criterios para definirlos correctamente. En particular, se propone basarse en bounded contexts. Si observamos las diferentes operaciones, podemos notar que algunas entidades se repiten constantemente: Grupo, Usuario y Mensaje. En general, este es un buen indicio para considerarlos bounded contexts y transformarlos en recursos independientes. Es importante notar que este proceso es iterativo,

no necesariamente estos van a ser los servicios definitivos. Si aparecen complejidades durante la implementación, o surgen nuevas funcionalidades que cambian la naturaleza de nuestro sistema, esta definición debe analizarse de nuevo. Finalmente, en la figura 5-1, agrupamos las operaciones por recurso.

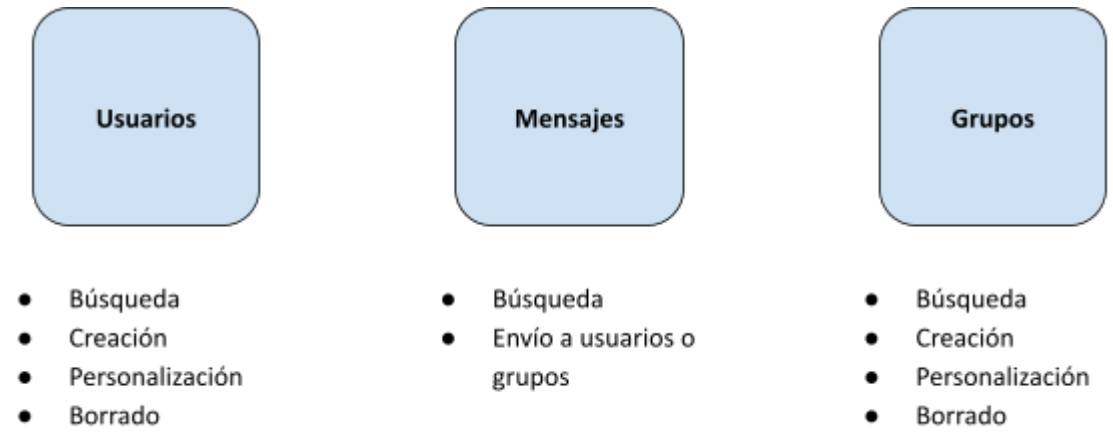
Tabla 5-1

Rol	**Caso de uso**	**Operación**
Usuario final	Comunicarse con otros usuarios	Buscar usuario
		Enviar mensajes a otro usuario
		Enviar mensajes a un grupo
		Buscar mensajes en mi bandeja de entrada
		Buscar mis grupos
	Administrar grupos	Crear un grupo
		Agregar un usuario a un grupo
		Personalizar grupo
		Eliminar grupo
	Personalizar perfil	Modificar datos personales
		Modificar imagen de perfil
Administrador	Administrar usuarios	Crear usuario
		Eliminar usuario

Tabla 5-2

Operación	**Entrada**	**Salida**
Buscar usuario	Nombre o email del usuario	Usuario
Enviar mensajes a otro usuario	Usuario receptor y mensaje	
Enviar mensajes a un grupo	Grupo y mensaje	
Buscar mensajes en mi bandeja de entrada	Usuario	Mensajes
Buscar mis grupos	Usuario	Grupos
Crear un grupo	Información del grupo	Grupo
Agregar un usuario a un grupo	Usuario que se va a agregar	
Personalizar grupo	Grupo	
Eliminar grupo	Grupo	
Modificar datos personales	Nuevos datos personales	
Modificar imagen de perfil	Nueva imagen	
Crear usuario	Información del usuario	Usuario
Eliminar usuario	Usuario	

Figura 5-1



5.1.1. Modelo de datos

En este apartado, se detallan los campos necesarios en cada recurso. La idea es tener un equilibrio entre información imprescindible e información potencialmente útil. Esta tarea no es trivial, ya que muchos campos complejizan el recurso y pocos dificultan el desarrollo de futuras funcionalidades. En este caso, optamos por mantener simple cada recurso. En la tabla 5-3, definimos los campos, sus tipos de datos y los dividimos en dos tipos: ingresados por el cliente y automáticos.

Inmediatamente, después de definir los campos, surge el dilema de qué tipo de modelo de datos debemos usar. En particular, la escalabilidad y la calidad de servicio son fundamentales en las aplicaciones de mensajería. Pocos usarían una aplicación que no envíe mensajes en tiempo y forma. Un modelo no relacional parecería el más adecuado, teniendo en cuenta este aspecto. Sin embargo, hay muchas otras consideraciones a tener en cuenta para esta elección y claramente puede pensarse de nuevo a futuro.

Tabla 5-3

Recurso	**Información proveniente del cliente**	**Información automática**
Usuario	Nombre Apellido *Nickname* (único) Email (único) Año de nacimiento Link de la imagen de perfil	• Identificador: definen unívocamente a un recurso.[17] • Eliminado: indica si el recurso fue borrado o no.[18] • Fecha de creación: es el momento en el cual se creó el recurso • Fecha de modificación: es el instante donde se modificó por última vez • Fecha de borrado: es el momento en el cual se eliminó el registro
Mensaje	ID del usuario emisor ID del usuario receptor Contenido Link de la imagen adjunta Visto por el receptor	
Grupo	Nombre ID del usuario que lo creó Lista de usuarios que pertenecen al grupo	

[17] En el modelo relacional, es común usar números enteros auto incrementales como IDs. Últimamente, se popularizaron los UUIDs, que no requieren un proceso centralizado para generarlos y, por ende, escalan mejor.

[18] Usualmente se usa un borrado lógico, no físico.

5.1.2. APIs

La información que tenemos hasta el momento es útil para tener una sólida visión general de los servicios de nuestro sistema. El siguiente paso es transformar estas ideas generales en una implementación de sus APIs.

Los formatos de descripción permiten especificar todo lo relacionado a la API de una forma estándar. Esto facilita la tarea de compartir el diseño con otras personas y equipos. También, pueden usarse herramientas complementarias, que procesan esta información estructurada y generan distintos resultados útiles.

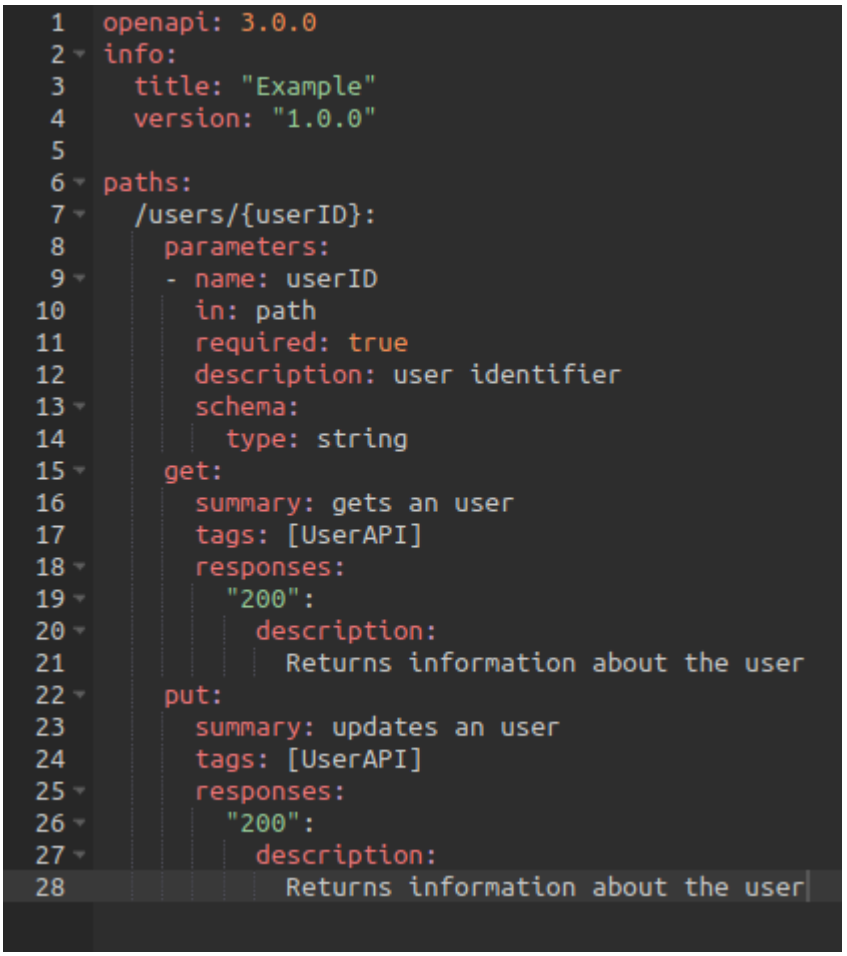
Especificar los detalles de cada operación disponible no sólo facilita el diseño de las APIs y el debate del mismo, además es una forma ideal para documentar cómo funciona. Recordemos que una API es un contrato implícito entre dos partes y la manera en que debe usarse tiene que ser lo más clara posible para evitar problemas y malentendidos futuros. La documentación debe incluir, además de la especificación, ejemplos prácticos y consideraciones al momento de utilizar la API. También, debemos tener en cuenta los flujos que incluyen distintas APIs.

Puntualmente, en esta sección se usará OpenAPI Specification (OAS)[19]. Esta especificación es ampliamente usada para APIs REST. OAS surgió a partir de Swagger y actualmente está disponible su versión 3.0. Posee distintas configuraciones que permiten llegar a un gran nivel de detalle de las distintas facetas de la API. [20]

A continuación, vamos a hacer un pequeño ejemplo para mostrar la facilidad de uso que tiene este formato y las distintas herramientas online existentes. En la figura 5-2, vemos una definición parcial del recurso Usuario. Como podemos observar , en primer lugar se define la versión de OAS utilizada, el nombre de la API y su versión. Después, establecemos cada una de las operaciones a partir de sus *paths* (URIs) y sus respectivos parámetros. Finalmente, definimos los distintos métodos disponibles. Como notamos, los campos que conforman el body del request y de su respectivo response, aún, no están definidos. Asimismo, no están presentes todos los paths, methods y status code del recurso.

[19] https://www.openapis.org
[20] En https://openapi-map.apihandyman.io se pueden ver todas las posibilidades.

Figura 5-2



```yaml
openapi: 3.0.0
info:
  title: "Example"
  version: "1.0.0"

paths:
  /users/{userID}:
    parameters:
    - name: userID
      in: path
      required: true
      description: user identifier
      schema:
        type: string
    get:
      summary: gets an user
      tags: [UserAPI]
      responses:
        "200":
          description:
            Returns information about the user
    put:
      summary: updates an user
      tags: [UserAPI]
      responses:
        "200":
          description:
            Returns information about the user
```

El formato nos da la posibilidad de establecer, a nivel documento, los componentes que usaremos en las distintas operaciones de la API. Esto nos facilita su reutilización y evita errores. De esta manera, en la figura 5-3, definimos el parámetro User ID. Por su parte, en la figura 5-4, están los campos del recurso Usuario, que se configuran mediante dos schemas. El primero corresponde a campos modificables y el segundo incluye los generados automáticamente. Esto va a ser de mucha utilidad para la posterior definición del request body y el response body de cada operación.

Como vimos en Ejemplo Práctico, en general, si una operación se ejecutó de forma exitosa, la respuesta es el recurso creado o actualizado. En el caso de que exista un error, la respuesta debe describirlo. En la figura 5-5, definimos los campos correspondientes a estos mensajes de error e informativos.

Figura 5-3

```yaml
components:
  parameters:
    userID:
      name: userID
      in: path
      required: true
      description: user identifier
      schema:
        type: string
```

Figura 5-4

```yaml
schemas:
  userCustomFields:
    type: object
    required:
      - first_name
      - last_name
      - nickname
      - email
      - birth_date
    properties:
      first_name:
        type: string
      last_name:
        type: string
      nickname:
        type: string
      email:
        format: email
        type: string
      birth_date:
        format: date
        type: string
  user:
    allOf: [$ref: "#/components/schemas/userCustomFields"]
    type: object
    properties:
      id:
        type: string
      profile_image:
        type: string
      date_created:
        format: dateTime
        type: string
      last_updated:
        format: dateTime
        type: string
      deleted:
        type: boolean
```

Figura 5-5

```yaml
    info:
      type: object
      required:
        - message
      properties:
        message:
          type: string
        details:
          type: string
    error:
      allOf: [$ref: "#/components/schemas/info"]
      type: object
      required:
        - code
      properties:
        code:
          type: integer
```

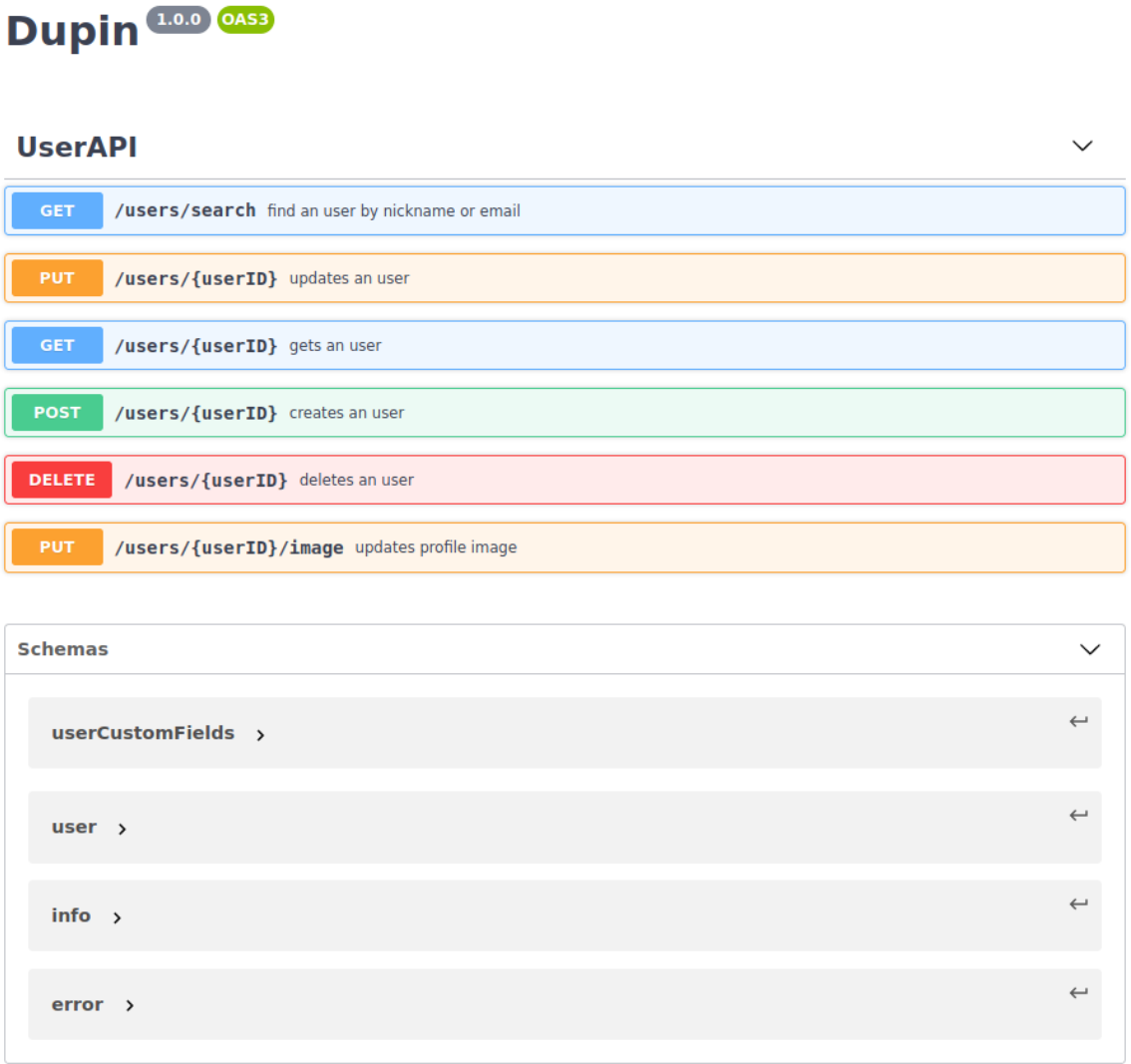
Figura 5-6

```yaml
/users/{userID}:
  parameters:
    - $ref: "#/components/parameters/userID"
  put:
    summary: updates an user
    tags: [UserAPI]
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/userCustomFields"
    responses:
      "200":
        description: updated user
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/user"
      "400":
        description: invalid values
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/error"
      "404":
        description: user not found
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/error"
      "500":
        description: internal server error
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/error"
```

En la figura 5-6, quedan establecidos los posibles parámetros, request bodies, response bodies y status codes del método PUT del recurso Usuario.[21] La especificación terminada nos permite usar las diferentes herramientas disponibles. Por ejemplo, en la figura 5-7, generamos automáticamente una documentación web de esta API.

Las APIs de Mensajes y Grupos tendrán una estructura similar a la de Usuarios. A partir de este ejemplo, y conociendo sus campos y operaciones, especificarlas no debería implicar mayor complejidad.

Figura 5-7



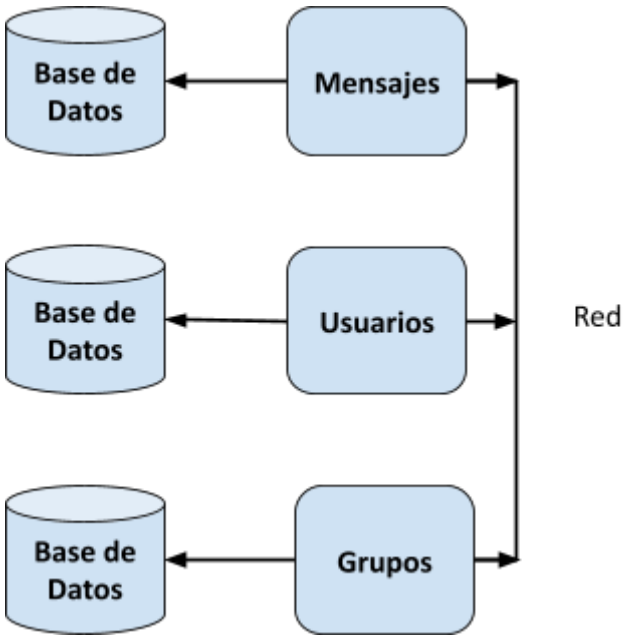
Fuente: https://editor.swagger.io/

[21] Definición completa de todas las operaciones de la API de Usuarios: https://github.com/lozaeric/dupin/blob/v1.2.1/users-api/api_description.yml

5.2. Comunicación entre servicios

A lo largo de esta tesis, notamos la importancia que tiene la comunicación entre los distintos servicios. Con una visión clara de cada recurso individual, podemos avanzar al siguiente paso, que consta de analizar sus interacciones. La acertada apreciación de las mismas va a definir buena parte de la estabilidad y el adecuado funcionamiento del sistema

Figura 5-8

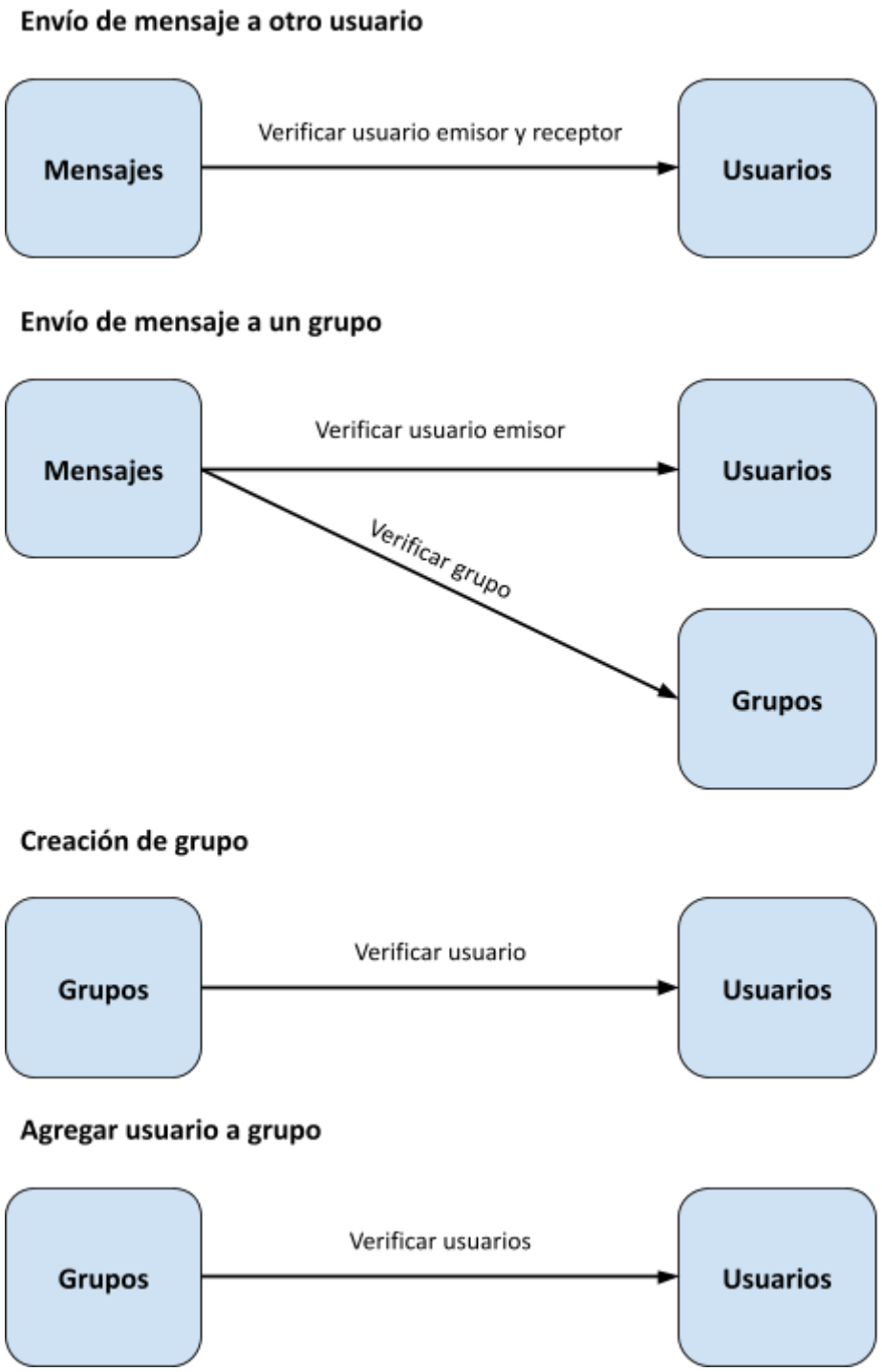


Christensen (2012) resalta un punto interesante con respecto a la dependencia entre servicios y su impacto en la disponibilidad. Imaginemos un escenario ideal donde cada servicio está disponible el 99% del tiempo. Si tenemos un servicio A que depende de cinco servicios, A sólo estará disponible el 95% (99^5) del tiempo, ya que requiere que todo el resto funcione correctamente. Es remarcable cómo las dependencias reducen drásticamente el *uptime* de un servicio, incluso teniendo todas una alta disponibilidad.

En principio, analicemos los distintos flujos existentes y qué servicios implican. En la figura 5-9, podemos ver las dependencias al momento de realizar operaciones en los servicios Mensajes y Grupos. Es claro que el servicio de Usuarios cumple un rol crítico en estos casos, ya que los flujos principales de los otros servicios necesariamente deben consultarlo. Como sabemos, la dependencia fuerte entre servicios debe evitarse para mantener un sistema saludable y resiliente. Y no sólo hablamos de escenarios donde Usuarios deje de funcionar,

también puede suceder que funcione lento o incluso que el problema no esté a nivel servicio, sino a nivel red. Hay varias estrategias para administrar esta situación y las describiremos a continuación.

Figura 5-9



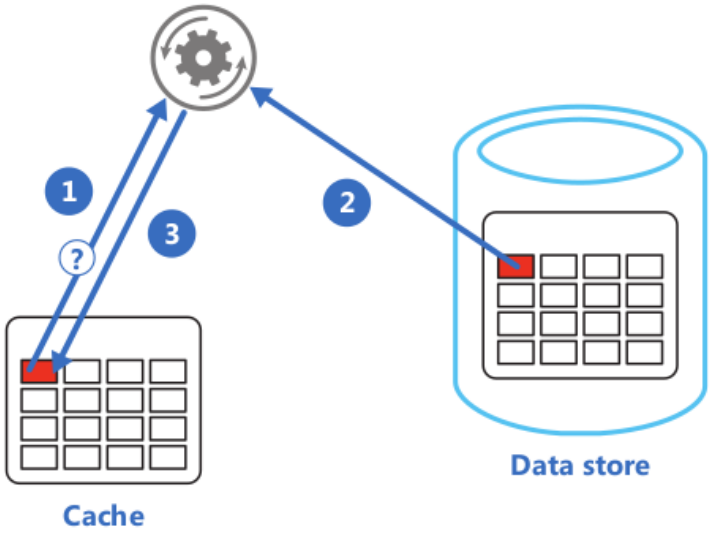
5.2.1. Cache

Una técnica muy común para mejorar la disponibilidad y el rendimiento de una aplicación es usar un *cache*. El mismo guarda temporalmente información proveniente de una base de datos, de otro servicio o de cualquier otro tipo de sistema que almacene datos. Obtener valores desde el cache suele ser más rápido que consultarlos desde el origen de los datos. Además, debido a que se replica la información, podemos utilizarla incluso si el origen no está disponible.

En la figura 5-10, detallamos el rol del cache al momento de consultar un valor::

- 1) Determinar si el valor existe en el cache. En caso de existir, la aplicación usa este valor.
- 2) Si el valor no existe, se obtiene desde el origen.
- 3) Se guarda una copia del valor en el cache para futuras consultas.

Figura 5-10



Fuente: Homer et al. (2014)

Hay distintas cuestiones a tener en cuenta al momento de implementar un cache. Por un lado, agrega complejidad al sistema, ya que es otro componente más. También, es necesario mantener sus valores actualizados y lidiar con escenarios donde temporalmente podrían no estarlo.

Homer et al. (2014) mencionan distintas consideraciones relevantes:

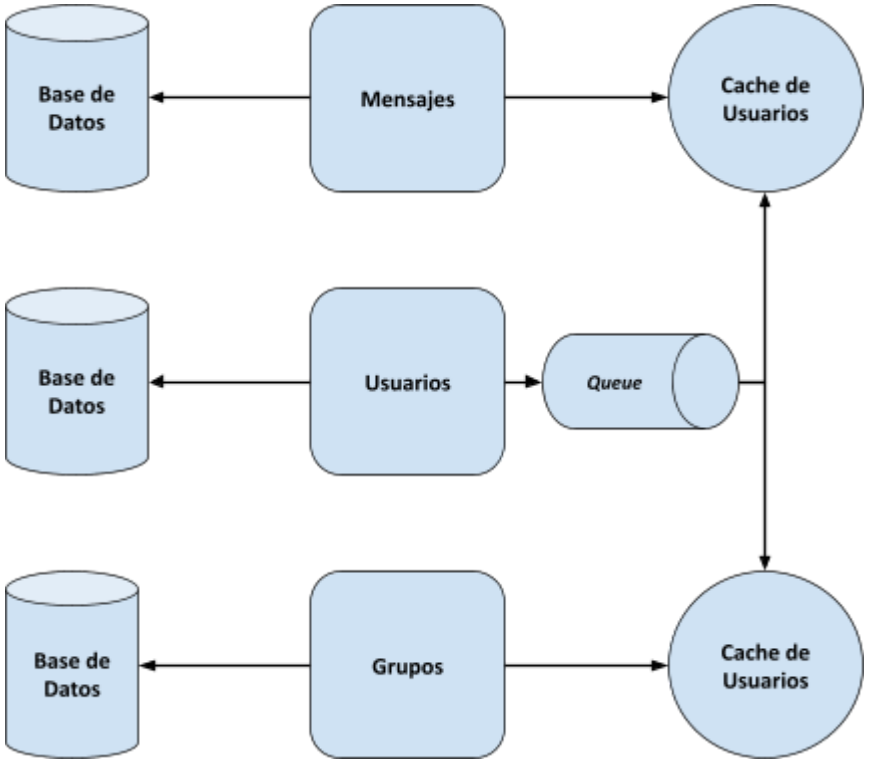
- Ciclo de vida de los datos: muchos caches permiten definir un período de tiempo durante el cual un valor es válido. Si el valor no fue accedido durante este período, es invalidado y removido del cache. Es difícil definir un tiempo adecuado porque, si es muy corto, el valor se tendrá que obtener de nuevo desde el origen y, si es muy largo, tendremos datos desactualizados por un tiempo mayor.
- *Eviction*: el tamaño del cache es limitado y probablemente mucho menor a la cantidad total de datos que potencialmente puede tener. Se denomina eviction al proceso que elimina datos en el cache para permitir guardar nuevos. Hay distintas estrategias que pueden usarse y cada una debe analizarse en el contexto de cada caso de uso particular.
- Consistencia: La consistencia entre el cache y el origen no está garantizada.
- Cache local y global: cada instancia de nuestro servicio puede guardar una réplica del cache y, de esta forma, agilizar su acceso. Sin embargo, el tamaño total va a estar limitado por la cantidad de memoria disponible por instancia. Una alternativa es que el cache sea un componente diferenciado y todas las instancias lo consulten, pero esto lo transforma en otra parte del sistema que debe monitorearse y mantenerse.
- Carga de datos: en algunos escenarios, puede ser necesario cargar datos al cache antes de que inicie la aplicación y así evitar enviar una gran cantidad de consultas al origen.

En nuestro caso particular, vamos a implementar un cache de Usuarios para el servicio de Mensajes y otro para Grupos. De esta forma, ambos servicios no van a depender totalmente de Usuarios. Sin embargo, la consistencia eventual nos obliga a considerar ciertos escenarios que surgen a partir de esta decisión. Por ejemplo, ¿Qué pasaría si un usuario eliminado intenta mandar un mensaje, pero ese usuario sigue siendo válido en el cache?. Esto claramente generaría mensajes inconsistentes. Estas situaciones son inevitables pero podemos reducir su impacto si minimizamos el tiempo durante el cual el cache está desactualizado. Aunque, en conjunto con esta estrategia, probablemente sea necesario implementar procesos que compensen estos errores. Por ejemplo, cuando un usuario es borrado, Mensajes puede eliminar toda la información relacionada y, de esta manera, solucionar las inconsistencias. Por otro lado, podrían existir flujos donde no es posible

soportar consistencia eventual. En estos casos, debe consultarse la API de Usuarios directamente.

Una manera eficiente y adecuada de actualizar el cache es mediante eventos. M. Fowler (2017) resalta su importancia como medio de comunicación entre los distintos microservicios. Los eventos son por naturaleza asincrónicos y el servicio que los produce no espera una respuesta definida a los mismos. De esta manera, facilitan la integración de una forma desacoplada. En concreto, la API de Usuarios va a emitir eventos cuando un usuario es creado, modificado o eliminado y, a partir de esta notificación, Mensajes y Grupos actualizarán sus respectivos caches, tal como indica la figura 5-11. También se pueden realizar otras acciones, como el proceso que compensa inconsistencias mencionado anteriormente. Los eventos son enviados mediante una comunicación del tipo publicar/suscribir y esto requiere agregar una *queue* con capacidad para procesarlos.

Figura 5-11



5.2.2. Timeout y Circuit Breaker

En Resiliencia, abordamos distintas estrategias para resolver errores durante la comunicación entre microservicios. Adicionalmente, en Ejemplo Práctico, observamos el uso de los status code para informar tipos de errores. En esta sección, se aplican estas buenas prácticas a nuestro diseño.

En principio, debemos diferenciar algunos tipo de errores que pueden generarse al momento de consultar una API:

- Errores en el cliente: en esta categoría, encontramos errores relacionados a la creación de recursos con valores inválidos, a la obtención de recursos inexistentes, entre otros. Se indica mediante un status code mayor o igual a 400 y menor a 500.
- Errores en el servidor: el request del cliente puede ser totalmente válido, pero igualmente puede fallar si el servidor no es capaz de procesarlo correctamente, ya sea por un error en el propio servicio o en algún otro componente relacionado. Se informa mediante un status code mayor o igual a 500 y menor a 600.

Estos errores son cualitativamente diferentes. En el primer caso, si el request se ejecuta de nuevo, probablemente el error vuelva a ocurrir. En cambio en el segundo, si se reintenta, seguramente la operación sea exitosa. Esta distinción es la que va a determinar nuestra estrategia de reintentos. Sin embargo, no siempre va a ser posible usar el status code para determinar la naturaleza del error, porque el mismo puede estar a nivel red. Estos casos también debemos tenerlos en cuenta, pero usando otros criterios.

El timeout nos posibilita mitigar la situación donde existen problemas a nivel red o el servicio no respondió a tiempo, aunque implica complejidades como la definición correcta de su valor y sus efectos colaterales. Analicemos el siguiente ejemplo:

- Supongamos que en condiciones normales, la API de Usuarios responde el 99% de los requests en 10 ms o menos.[22]

[22] Los servicios internos deben tener establecidos response times esperados para facilitar la definición del timeout. Para servicios externos, generalmente tenemos disponible el SLA. En última instancia, el timeout se puede calcular de forma empírica.

- El valor del timeout lo definimos en 12ms para tener en cuenta la inestabilidad de la red y, además, agregamos un reintento que se ejecutará 5 ms después del primer request. Mensajes se configura con estos valores.

Parece un valor de timeout razonable, pero comparemos el caso ideal y el peor caso:

- Caso ideal: duración total ≤ 12 ms
- Peor caso: duración total = 29 ms (12 ms + 5 ms + 12 ms)

A gran escala, esta diferencia puede incrementar muchísimo el uso de CPU y de memoria de cada instancia de Mensajes por la cantidad de requests concurrentes que debe procesar. En pocas palabras, una degradación del servicio de Usuarios va a degradar Mensajes o, incluso peor, generar que no esté disponible. Y no sólo Mensajes se verá afectado, los reintentos van a dificultar la recuperación de Usuarios. Este círculo vicioso puede eliminarse mediante un circuit breaker que, a partir de su activación, va a considerar a Usuarios como no disponible. De esta forma, Mensajes puede fallar inmediatamente, sin realizar el request y sin consumo extra de recursos. A su vez, Usuarios no recibe tráfico excesivo, debido a los reintentos.

En resumen, los reintentos evitan que errores intermitentes afecten el funcionamiento normal de cada servicio. Por su parte, el timeout y el circuit breaker nos dan la posibilidad de abordar situaciones donde la red o el servicio dejan de funcionar o lo hacen lentamente. Estas herramientas se pueden implementar a nivel aplicativo o como un componente de la infraestructura.[23]

Figura 5-12

[23] https://www.nginx.com/blog/what-is-a-service-mesh/

5.3. Seguridad

Hasta el momento, no hablamos de la seguridad en el conjunto de servicios que estamos diseñando. Imaginemos el caso donde Mensajes procesa un mensaje nuevo. El servicio debería tener una forma fiable de determinar que el usuario emisor realmente quiere realizar esta acción. En Seguridad, ya mencionamos la importancia de estos aspectos y en esta sección vamos a implementarlos.

5.3.1. Open Authorization (OAuth)

OAuth 2.0 es un reconocido protocolo para administrar la autorización en aplicaciones.[24] En pocas palabras, permite que un usuario final pueda autorizar a una aplicación para acceder a recursos de un sistema en nombre de ese usuario. Mencionemos algunos términos clave de este protocolo:

- Scope: Es un conjunto de recursos disponibles.
- *Access token*: Es una cadena de letras, números o símbolos, que representa un permiso a un scope definido.

OAuth 2.0 además define 4 roles:

- Dueño del recurso: es el usuario propietario del recurso..
- Servidor dueño del recurso: el servidor que efectivamente posee el recurso. Tiene capacidad de procesar requests con access token.
- Cliente: una aplicación que quiere acceder a un recurso y actúa en representación del dueño del mismo.
- Servidor de autorización: es el encargado de administrar los flujos de autorización disponibles en el protocolo OAuth. Además, genera los access tokens.

En la figura 5-13, podemos ver el rol que cumple cada una de las partes en este esquema. El flujo de autorización consta de 3 etapas:

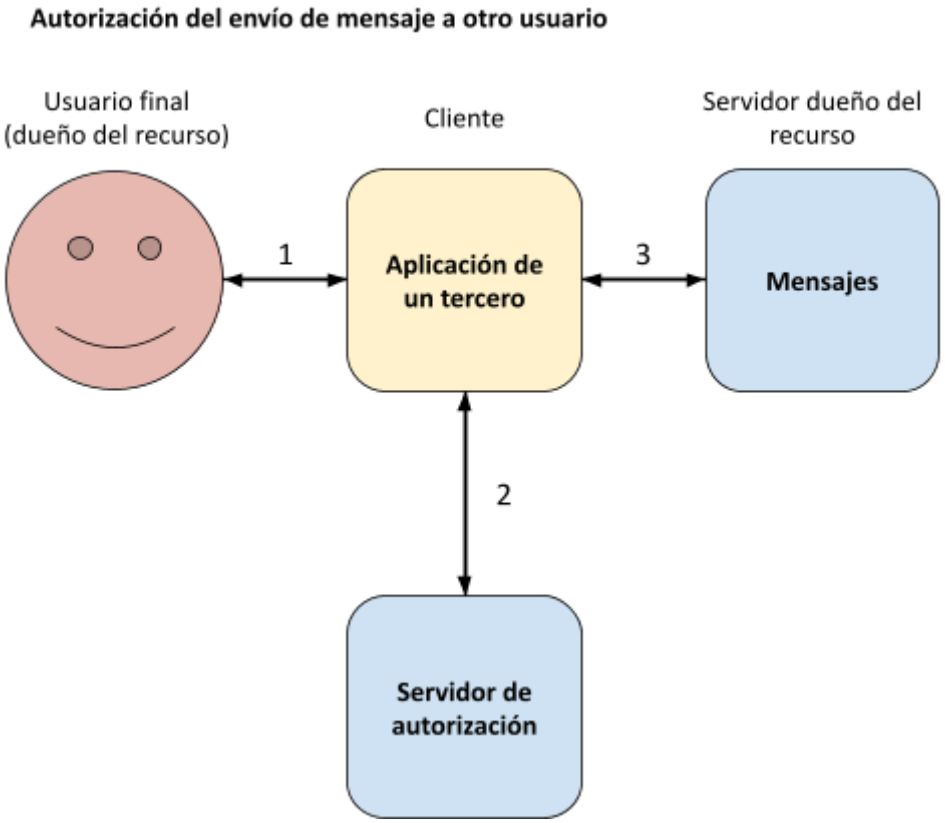
- El cliente le pide una autorización al dueño del recurso. Lo puede pedir directamente o mediante el servidor de autorización. Si el dueño lo autoriza, el cliente recibe un permiso de autorización

[24] Una definición completa del protocolo puede encontrarse en https://tools.ietf.org/html/rfc6749.

- El cliente se identifica ante el servidor de autorización y pide un access token a partir del permiso previamente obtenido.
- El cliente accede al recurso protegido enviando el access token al servidor dueño del recurso y este verifica su validez.

OAuth 2.0 define estos términos y flujos en profundidad y analiza distintos escenarios, pero conociendo los fundamentos básicos de este protocolo, es posible realizar la implementación adecuada para nuestro caso.

Figura 5-13



5.3.2. JSON Web Token (JWT)

En la sección anterior, notamos la importancia de los access tokens para la seguridad de nuestro sistema. Podemos diferenciar dos tipos de tokens, según Richardson (2018):

- Opacos: Son una cadena de caracteres, que no tienen ningún significado por sí mismo. Por ello, es necesario consultar un servicio adicional para obtener la información relacionada al token. Este nuevo servicio puede impactar en la disponibilidad y rendimiento del sistema.

- Transparentes: El token es auto contenido o, dicho de otra forma, posee toda la información necesaria para ser usado. En este caso, su contenido tiene que estar encriptado por cuestiones de seguridad.

Una opción popular para tokens transparentes son los denominados *JSON Web Tokens* (JWT), que codifican la información necesaria en formato JSON. Esta parecería una opción sólida para evitar los aspectos negativos de los tokens opacos. Sin embargo, debemos analizar algunos escenarios adicionales que aplican a ambos tipos de tokens:

- Validez temporal: los tokens pueden ser vulnerados y, dado que son almacenados por los clientes, su completa seguridad está fuera del alcance del servicio. Una forma de mitigar esta situación es hacerlos válidos por un período de tiempo determinado.
- Revocar acceso: suele suceder que el dueño del recurso quiera revocar el acceso a un cliente o que el mismo sea restringido del sistema. En ambos casos, el token debe perder su validez inmediatamente.

JWT puede ser válido de forma temporal, sin embargo, no puede ser invalidado inmediatamente porque no depende de un servicio externo. De esta forma, los servicios que autoricen mediante JWT no tienen forma de conocer la validez real del token. Dependiendo de la naturaleza de la aplicación, esto puede ser inaceptable, aunque en nuestro caso de uso, donde priorizamos la escalabilidad y disponibilidad, nos inclinaremos por esta opción.

5.3.3. Implementación

En OAuth, los scopes cumplen un papel muy importante, ya que agrupan los permisos para acceder a las distintas operaciones. En nuestro caso particular, vamos a priorizar un esquema de seguridad simple donde cada scope incluye operaciones de similar impacto. Este criterio es similar al que utilizaremos al momento de dividir nuestra infraestructura en clusters.

Es necesario crear un servicio adicional que va a cumplir las funciones del Servidor de Autorización, según la definición de OAuth. Dado que usamos JWT, este servidor no va a verificar la validez de los tokens en cada request, sino que se va a encargar de la generación de tokens, la autenticación del usuario y el manejo del resto de los flujos de seguridad.

Tabla 5-4

Operación	**Recurso responsable**	**Scope**
Buscar usuario	Usuario	Usuario - lectura
Modificar datos personales	Usuario	Usuario - escritura
Modificar imagen de perfil	Usuario	Usuario - escritura
Crear usuario	Usuario	Usuario - admin
Eliminar usuario	Usuario	Usuario - admin
Enviar mensajes a otro usuario	Mensaje	Mensaje - escritura
Enviar mensajes a un grupo	Mensaje	Mensaje - escritura
Buscar mensajes en su bandeja de entrada	Mensaje	Mensaje - lectura
Buscar mis grupos	Grupo	Grupo - lectura
Crear un grupo	Grupo	Grupo - admin
Agregar un usuario a un grupo	Grupo	Grupo - escritura
Personalizar un grupo	Grupo	Grupo - escritura
Eliminar grupo	Grupo	Grupo - admin

Además de la autenticación y la autorización, debemos tener en cuenta otros mecanismos que aborden el resto de las amenazas existentes. Esto incluye procesos de auditoría, utilización de Hypertext Transfer Protocol Secure (HTTPS), monitoreo de tráfico inusual, limitación en la cantidad de operaciones por usuario, entre otras cuestiones relevantes.

5.4. Infraestructura

Como detallamos en Microservicios, una de sus grandes ventajas es la autonomía entre servicios y la posibilidad de dividir su infraestructura para mejorar la eficiencia en el uso de recursos. Ya tenemos definidos los servicios que conforman nuestro sistema, es momento de traducir estos componentes a conjuntos de servidores que van a escalar horizontalmente para

procesar el tráfico. Además, se van a mencionar formas de llevar el software a producción en este tipo de infraestructura.

Tanto las queues como las bases de datos serán servicios externos, provistos por algún proveedor. La razón para tomar esta decisión es enfocarnos en nuestra aplicación y no en componentes complementarios. Con la aparición de la nube y todos los servicios de infraestructura que brinda, sumado a la proliferación de APIs, ya no es necesario reinventar la rueda para resolver problemas puntuales.

5.4.1. Clusters

En Resiliencia, mencionamos distintas estrategias para administrar la tolerancia a errores en el sistema. Con respecto a la infraestructura, está claro que cada servicio debe estar aislado del resto. A su vez, los flujos de trabajo de cada uno también deben agruparse en distintos conjuntos o *clusters* de instancias. Esto nos va a permitir mejorar la resiliencia, la escalabilidad y el monitoreo. Además, es posible replicar los cluster principales en varias regiones geográficas para aumentar la disponibilidad general y reducir la latencia de la red.

En particular, el criterio propuesto para separar la infraestructura es la carga esperada y la homogeneidad del tráfico. De esta forma, las operaciones con necesidades de recursos similares estarán en el mismo cluster. Partimos de las siguientes premisas:

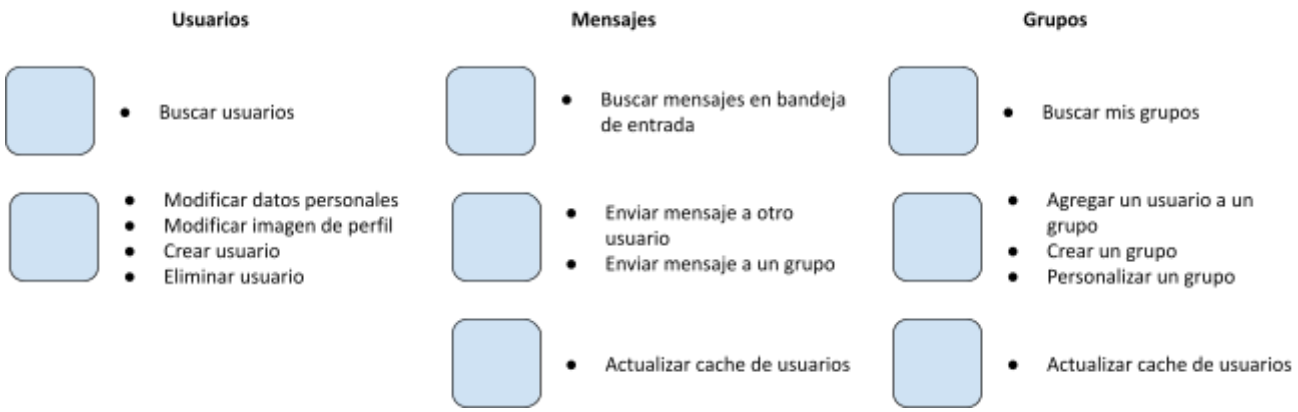
- Los requests que implican una escritura a la base de datos son intrínsecamente diferentes de los que realizan una lectura. Los primeros modifican el estado del recurso y suelen ser más lentos.
- Las lecturas son mucho más frecuentes que las escrituras.
- Las caches se actualizan mediante eventos de forma asincrónica. Por definición, este flujo es diferente de las operaciones sincrónicas mencionadas en puntos anteriores.

En la tabla 5-5, describimos cada una de las operaciones en base al criterio propuesto. Adicionalmente, en la figura 5-14, enumeramos los ocho clusters definidos y los flujos que contendrá cada uno. Esta división inicial debe ser repensada durante el avance de la implementación del sistema, ya que nuevas funcionalidades o servicios pueden modificarla. También, puede cambiar si se detecta un flujo crítico, que requiere un alto rendimiento o disponibilidad, o se observan dificultades durante el monitoreo de cada cluster.

Tabla 5-5

Operación	**Recurso responsable**	**Tipo**
Buscar usuario	Usuario	Lectura
Modificar datos personales	Usuario	Escritura
Modificar imagen de perfil	Usuario	Escritura
Crear usuario	Usuario	Escritura
Eliminar usuario	Usuario	Escritura
Enviar mensajes a otro usuario	Mensaje	Escritura
Enviar mensajes a un grupo	Mensaje	Escritura
Buscar mensajes recibidos	Mensaje	Lectura
Actualizar cache de usuarios	Mensaje	Escritura asincrónica
Buscar mis grupos	Grupo	Lectura
Crear un grupo	Grupo	Escritura
Agregar un usuario a un grupo	Grupo	Escritura
Personalizar un grupo	Grupo	Escritura
Eliminar grupo	Grupo	Escritura
Actualizar cache de usuarios	Grupo	Escritura asincrónica

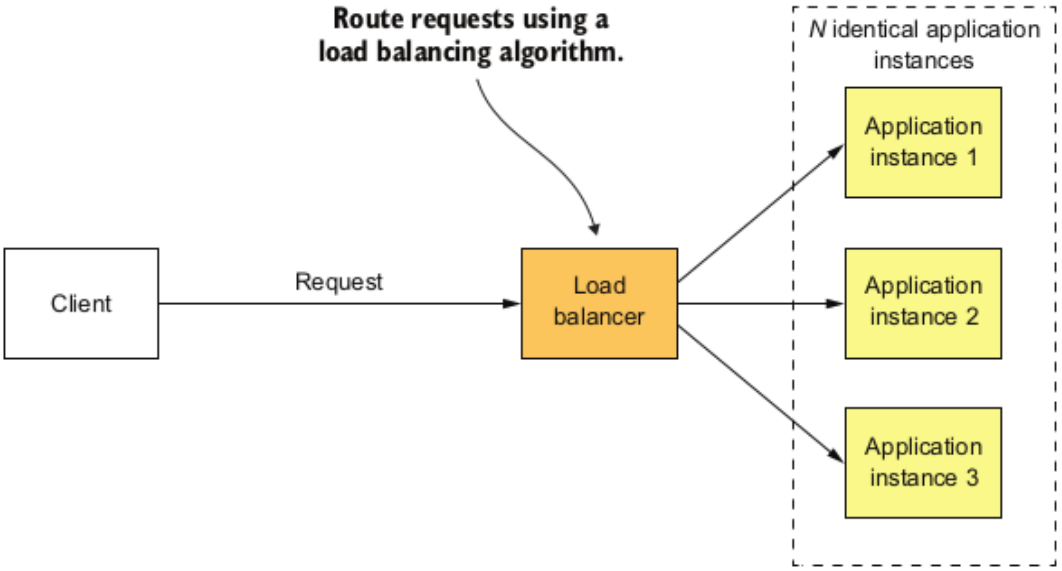
Figura 5-14



5.4.2. Instancias

Es usual que cada cluster de una arquitectura de microservicios utilice escalamiento horizontal por todos los beneficios que aporta. En la figura 5-15, podemos ver más en detalle el funcionamiento del cluster. Un *load balancer* reparte el tráfico entre el conjunto de instancias y controla el estado de cada una. Asimismo, es responsable de crear o eliminar servidores a partir de la cantidad de tráfico actual y del uso de recursos.

Figura 5-15



Fuente: Richardson (2018)

Algunas consideraciones importantes al momento de usar escalamiento horizontal:

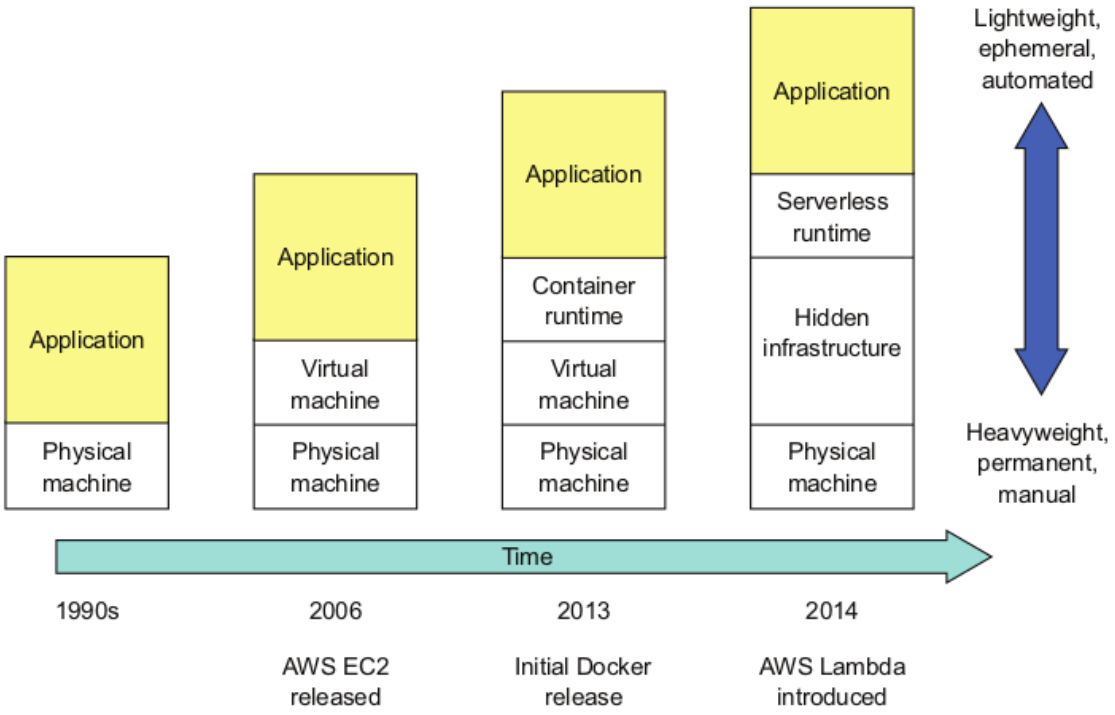
- La creación de las nuevas instancias debe ser automática y rápida. De otra forma, no sería posible escalar el cluster en tiempo y forma.
- El criterio para definir cuándo eliminar o agregar instancias debe ser analizado con sumo cuidado. Hay distintas alternativas disponibles, como uso promedio de memoria, uso de CPU, tiempos de respuesta, entre otras. Según el tipo de operaciones que el cluster realice, se puede optar por una u otra alternativa.

Richardson (2018) afirma que la necesidad de escalar horizontalmente y la migración a la nube causaron que las aplicaciones residan en contextos más efímeros, livianos y automatizados. Este profundo cambio a nivel infraestructura puede observarse en la figura

5-16. Cada instancia debe ser efímera porque puede eliminarse o crearse en cualquier momento, según se necesite.

Recordemos que la comunicación entre servicios se realiza mediante HTTP. Este protocolo no mantiene un estado entre requests, ya que cada uno es auto contenido. De esta manera, debería ser indiferente que un request sea procesado por una instancia nueva o una existente hace mucho tiempo.

Figura 5-16



Fuente: Richardson (2018)

McCance (2012) coincide en que hay una idea cada vez más extendida de considerar a los servidores "como ganado y no como mascotas". Para entender mejor esta particular comparación, enumeremos las características de las mascotas: hay una sola o unas pocas, son muy queridas e importantes para el hogar, reciben muchos cuidados y si se enferman nos afecta. En cambio, el ganado está formado por muchos animales, que necesitan cuidados mínimos. Cada uno es muy similar al otro e individualmente no son relevantes o, dicho de otra forma, son fácilmente reemplazables. En resumen, es preferible tener muchos servidores irrelevantes que pocos muy importantes porque esto fortalece la estabilidad general.

5.4.3. Deployment

Richardson (2018) define *deployment* como una combinación de dos conceptos relacionados: proceso y arquitectura. El proceso consiste en etapas que deben ejecutarse para poder tener el software en producción.[25] La arquitectura del deployment es el entorno en el cual se ejecuta la aplicación. En este apartado, se analiza el proceso..

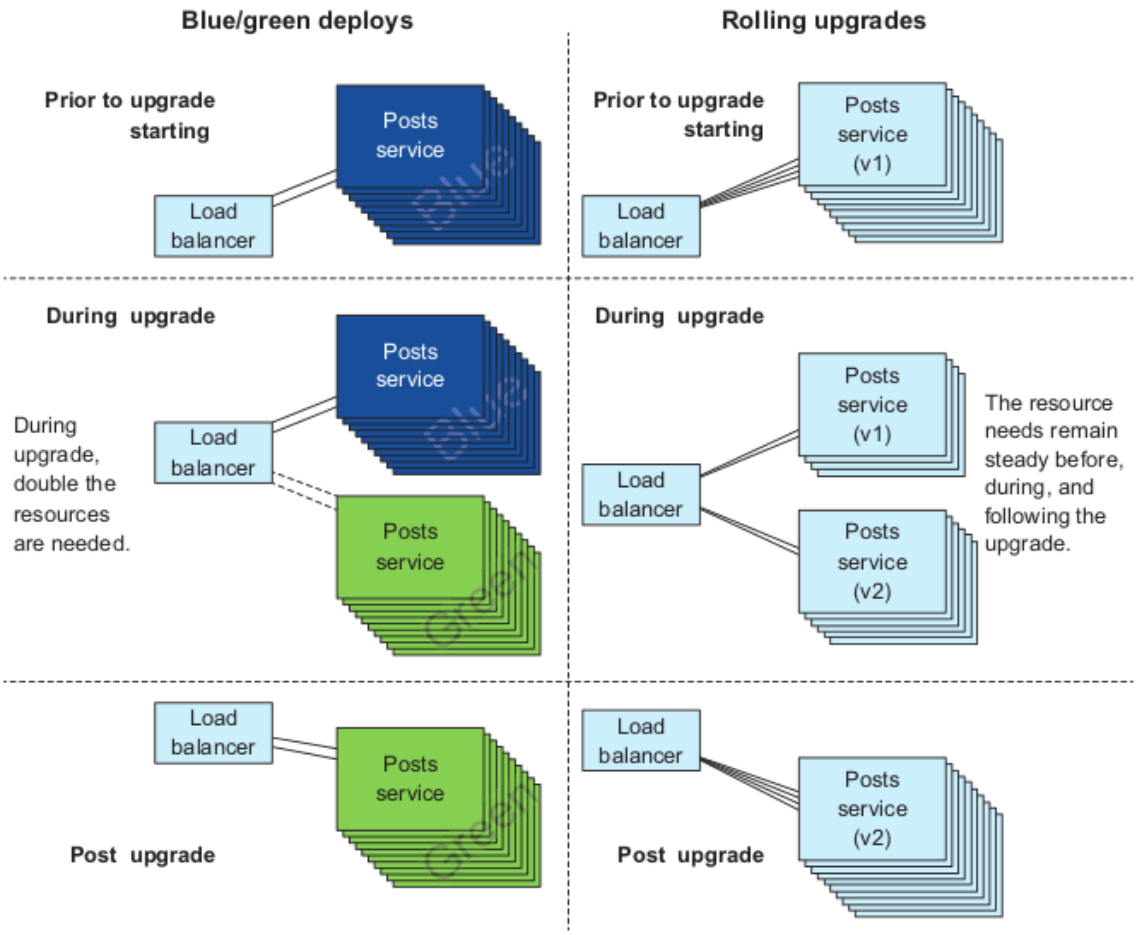
Por definición, los microservicios constan de varias aplicaciones que funcionan en conjunto. Esto le da más independencia a cada equipo para controlar el ciclo de desarrollo y de puesta en producción de su aplicación. A su vez, cada servicio consta de uno o más clusters que pueden actualizarse de forma independiente y su naturaleza es distribuida. Davis (2019) analiza dos maneras de realizar deployments en este tipo de infraestructura:

- *Blue/green*: la versión actual del cluster se denomina *blue* y la nueva se llama *green*. Este tipo de deployment implica la existencia de dos clusters, uno para cada versión. Como primer paso, parte del tráfico actual se envía a green para verificar que la nueva versión se comporte como se espera. Si el resultado fue positivo, green se transforma en la versión actual y, posteriormente, se elimina blue.
- *Rolling*: en este caso, tendremos un sólo cluster a lo largo del deployment. Progresivamente, cada instancia es actualizada con la nueva versión hasta que todo el cluster esté finalizado. Si se verifica que la nueva versión tiene algún error, es necesario modificar todas las instancias que se hayan actualizado.

Cada tipo de deployment tiene sus puntos positivos y negativos. El blue/green genera un mayor uso de recursos durante el deployment por la existencia de dos clusters. Sin embargo, tenemos la posibilidad de volver rápidamente a la versión blue, en caso de necesitarlo. Por el contrario, el rolling no permite que este tipo de cambio se realice rápidamente porque sólo existe un cluster en cada momento. Asimismo, el tiempo durante el cual tenemos dos versiones procesando tráfico productivo es mucho mayor para el caso del rolling y esto debe ser considerado. La existencia de dos versiones simultáneas puede dificultar el análisis de errores transcurridos durante el deployment.

[25] Este punto se analizará en CI y CD.

Figura 5-17



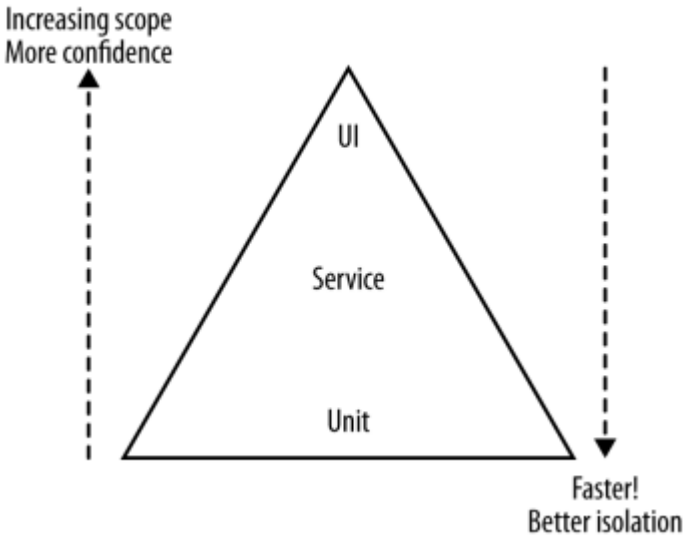
Fuente: Davis (2019)

5.4.4. Continuous Integration (CI) y Continuous Delivery (CD)

En Requerimientos de una aplicación moderna, mencionamos la importancia de agilizar el proceso de modificación de cada uno de nuestro servicios. Las exigencias de los usuarios finales nos obligan a implementar mecanismos automatizados que aseguren la correctitud del código y aceleren su proceso de puesta en producción. Antes de describir CI y CD, se debe ahondar en la definición de *testing* porque va a ser una parte fundamental en estos procesos. El testing de cada servicio permite asegurarnos que la aplicación funciona correctamente, tanto en particular como en general. S. Fowler (2017) define distintos tipos de tests:

- Tests unitarios: son pruebas pequeñas e independientes, que verifican cada unidad de código. Estas unidades se prueban de forma aislada y pueden ser funciones, clases o cualquier componente individual de la aplicación.
- Tests de integración: validan que las distintas partes del servicio funcionan adecuadamente cuando trabajan en conjunto. Una forma habitual de hacerlo es ejecutando requests definidos contra un entorno de prueba y analizando su resultado.
- Tests *end-to-end*: Es la prueba más abarcativa porque verifica que un servicio, junto a los servicios y componentes de los que depende, funcione correctamente.

Figura 5-18



Fuente: Newman (2015)

Los tests mencionados están enfocados en el aspecto funcional de la aplicación, pero también es posible analizar otras características como la seguridad, la usabilidad, el rendimiento, entre otras. Todas estas pruebas deben automatizarse y ser parte del proceso de puesta en producción de cada versión de la aplicación. Según el tipo de servicio que diseñamos, podemos necesitar pruebas más rigurosas o no.

Ya teniendo definido el testing, podemos avanzar con la definición de *Continuous Integration* (CI) y *Continuous Delivery* (CD). Newman (2015) los define de la siguiente forma:

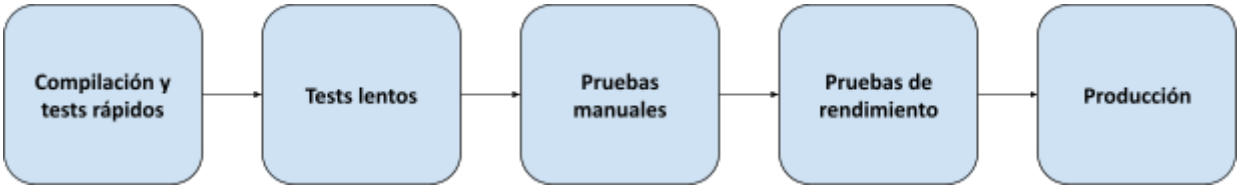
- Continuous Integration: el objetivo es asegurar que el nuevo código se integre adecuadamente al código existente. Un servidor CI detecta cambios en el repositorio de código y, a partir de los mismos, realiza una serie de verificaciones. En general, el

proceso consiste en generar un único artefacto ejecutable y realizar tests sobre el mismo.

- Continuous Delivery: permite modelar y automatizar la ejecución de la serie de pasos necesarios para llegar desde un cambio en el código a producción. Cada paso debería ser automático, pero también es posible definir etapas manuales. En primer lugar, se suelen ejecutar los que menos tiempo y recursos requieren. En la figura 5-19, podemos ver un ejemplo.

Sin mecanismos automatizados, como los mencionados, sería imposible agilizar el desarrollo en sistemas distribuidos de gran escala, donde hay infinidad de modificaciones constantemente. Otro beneficio de *deployar* cambios pequeños e incrementales es que reduce enormemente la posibilidad de tener errores mayúsculos en producción. Cada *deployment* es mínimo y es realizado por el equipo responsable del servicio.

Figura 5-19



Por último, la existencia de varios servicios independientes dificulta la creación de ambientes de prueba. Esto es debido a lo complejo que resultaría mantenerlos actualizados con respecto al entorno productivo. Y, como sabemos, un ambiente de prueba que difiere de producción no es útil para verificar funcionalidades. De igual forma, crear tests end-to-end es sumamente complejo en esquemas donde existen decenas o cientos de servicios. Más allá de los diversos tests implementados, las pruebas definitivas terminan siendo en producción. Por ello, es imprescindible realizarlas de forma controlada y progresiva para no interferir con el correcto funcionamiento del sistema

5.5. Monitoreo

Es importante saber lo que está haciendo cada servicio en cada momento porque nos da la posibilidad de tener un panorama general del sistema. Esta información se complementa con la relacionada a la infraestructura y la red. A partir del monitoreo, podemos generar alertas que nos indican cuándo un componente no está funcionando como debería. Muchas de las

métricas de infraestructura, en conjunto con alertas y consideraciones, son brindadas por el proveedor de servicios de la nube utilizado. En cambio, las métricas de cada servicio deben ser definidas por su equipo responsabe y dependen de la naturaleza del mismo. En este apartado, nos centramos en las métricas clave del sistema y en la relevancia del rendimiento en nuestro sistema.

5.5.1. Métricas clave

Las distintas métricas y logs nos facilitan la investigación eficiente de problemas en producción. Cada operación debería tener asociada una métrica con la cantidad de veces que se ejecutó y el tiempo de esa ejecución. Cada interacción entre dos servicios o entre un servicio y la base de datos también debería ser medida de esta manera. Todas estas métricas van a ser útiles para encontrar la causa de un problema, pero no para identificarlo rápidamente, debido a su enorme cantidad. Aquí yace la importancia de definir un conjunto de métricas relevantes del sistema. Las denominaremos métricas clave.

Las métricas clave deben representar las funcionalidades principales y darnos la posibilidad de conocer rápidamente el estado de la aplicación en cada instante. Particularmente, los mensajes enviados y vistos son dos operaciones centrales en nuestro sistema. Por su lado, las operaciones de creación de grupos y usuarios también deben ser tenidas en cuenta para tener un panorama general de la actividad de los clientes. En la tabla 5-6, definimos las operaciones clave, sus métricas e información adicional sobre cada una. Esto último facilita el posterior análisis de estos indicadores.

Tabla 5-6

Operaciones	**Métricas**	**Información adicional**
Mensajes enviados	Cantidad Duración	Status code Tipo (usuario o grupo)
Mensajes vistos	Cantidad Duración	Status code Tipo (usuario o grupo)
Nuevos usuarios	Cantidad Duración	Status code
Nuevos grupos	Cantidad Duración	Status code

Todas las métricas clave suelen representarse de dos maneras: cantidad y duración. Con respecto a la cantidad, puede cambiar a lo largo del día, especialmente durante la noche. Por su parte, la duración o los tiempos pueden medirse de distintas formas: promedio, percentil, mediana, entre otros. En particular, Tong (2014) recomienda utilizar percentiles. En pocas palabras, los mismos indican que un porcentaje de las muestras es menor o igual a determinado tiempo. Por ejemplo, si el percentil 99 es 100 ms, podemos deducir que el 99% de las operaciones duró ese tiempo o menos. En cambio, los promedios y medianas no expresan con claridad los valores muy altos o muy bajos.

Los errores, que alteran la operatoria de los servicios, pueden tener orígenes variados, pero necesariamente van a afectar las métricas clave y, de esta forma, podemos detectarlos. En base a esta detección, es posible indagar en el resto de las métricas y deducir la causa. Finalmente, va a ser posible realizar acciones de mitigación y plantear mejoras a futuro para evitar repetir este tipo de problemas.

5.5.2. Rendimiento

La calidad de servicio es un punto clave en el diseño de un sistema de mensajería. Entregar los mensajes en tiempo y forma es un requisito clave, con todo lo que esto implica. Hay dos dimensiones relevantes relacionadas a este tema: conocer los límites de escalabilidad del sistema y asegurarnos que las nuevas versiones aplicativas tengan el rendimiento esperado. Realizar pruebas controladas, que estresan la aplicación, nos va a permitir tener un mejor acercamiento a su rendimiento.

Según S. Fowler (2017), una prueba de carga es una forma de conocer el comportamiento de un sistema bajo una cantidad de tráfico determinada. Durante la prueba, debe realizarse un detallado monitoreo, ya que los resultados van a facilitar el análisis de diferentes optimizaciones. Además, esta prueba es una etapa muy importante en el proceso de puesta en producción de una nueva versión aplicativa. Los resultados de la prueba de carga posibilitan:

- Analizar los límites de escalabilidad de nuestro sistema y detectar dónde está el cuello de botella.
- Conocer el estado normal o saludable. Esto es clave para definir correctamente las alertas en aplicaciones nuevas.

- Predecir la necesidad futura de recursos.
- Detectar *bugs* o malas prácticas que estén haciendo un mal uso de los recursos.
- Verificar el cumplimiento del *Service Level Agreement* (SLA).[26]

Las pruebas de carga, además, tienen que ser repetibles. Es decir, ejecutarlas reiteradas veces sobre la misma versión debe generar resultados con mínimas diferencias. De otra forma, no serían fiables. Para garantizar esta característica, es necesario usar una infraestructura similar y aislada en cada ejecución, además de utilizar el mismo operaciones en cada prueba. Ningún proceso ajeno debería interferir durante la realización de la prueba.

Ya hemos definido los indicadores más importantes. La prueba debería ejecutar estas operaciones con distintos niveles de carga y, de esta manera, simular un escenario real. Pasemos a un ejemplo concreto:

- Crear 10.000 usuarios con valores aleatorios
- Crear 500 grupos con 50 usuarios cada uno
- Cada usuario envía un mensaje a otros 10 usuarios y a 2 grupos. Esto es un total de 120.00 mensajes enviados.
- Cada usuario marca como leídos 4 mensajes recibidos..

Esta serie de pasos puede repetirse cada minuto, cada 5 minutos o el tiempo que corresponda, según la cantidad de tráfico que queremos probar. El ejemplo mencionado ejecuta operaciones en varios servicios y se aproxima a un escenario real. Es adecuado para ser parte del proceso de CD. Adicionalmente, serán necesarias pruebas que estresen individualmente a cada servicio y nos faciliten la detección de los límites de cada uno.

5.6. Proof of Concept (PoC)

En apartados anteriores, no se especificaron lenguajes de programación, bases de datos, caches o métodos de virtualización. Esto es debido a que, durante la etapa de diseño, no es deseable elegir una implementación definida porque implicaría adelantarse en la toma de decisiones. Sin embargo, en el desarrollo final del sistema, será necesario elegir soluciones

[26] El SLA es un acuerdo entre un proveedor y un cliente donde el primero ofrece distintas garantías sobre la aplicación como tiempos de respuesta, porcentaje de disponibilidad, entre otras.

concretas, teniendo en cuenta las capacidades del equipo y un minucioso análisis de ventajas y desventajas.

Cordeiro (2020) define una *proof of concept* como una evidencia de que la solución propuesta resuelve el problema planteado. De esta forma, la PoC aumenta la posibilidad de éxito de la solución. El autor define las siguientes características:

- Está enfocada en aspectos relevantes de la solución adoptada
- Solamente se usa como demostración, no está preparada para producción
- Usualmente consta de un conjunto limitado de funcionalidades
- Puede ser realizada en una o múltiples tecnologías

En la tabla 5-7, se observan los servicios y las operaciones implementadas en la PoC desarrollada. Teniendo en cuenta que la operación fundamental del sistema es el envío de mensajes, todo el resto se pensó partiendo de ese punto y priorizando la simpleza. Por esta razón, no se implementan las operaciones del servicio Grupos, la búsqueda de usuarios, ni el envío de mensajes con imágenes adjuntas, entre otros flujos.

Tabla 5-7

Servicio	**Operaciones**
Usuarios	Crear Obtener Actualizar
Mensajes	Crear Obtener Actualizar Buscar
Auth	Crear contraseña de un usuario Obtener access token

En la PoC, se utilizaron las siguientes tecnologías:

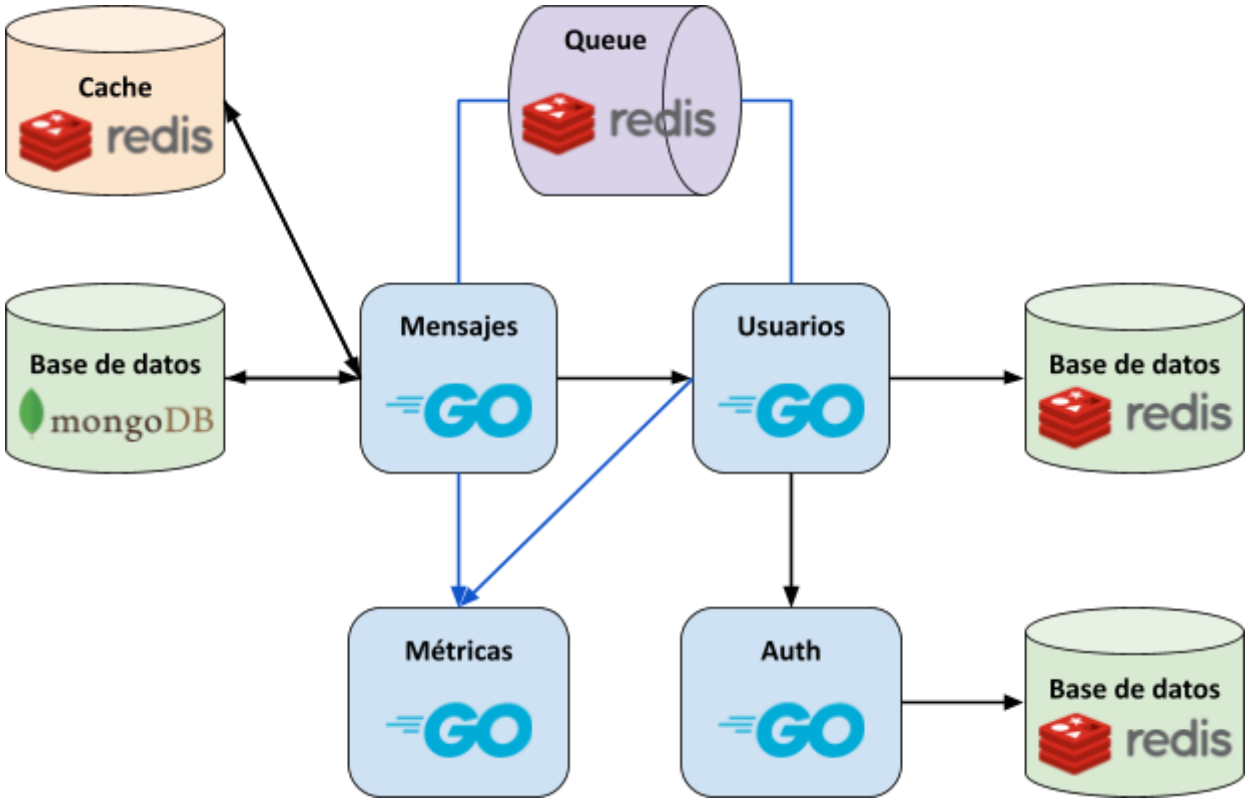
- Go[27]: es un lenguaje de programación surgido en 2012. Su objetivo es crear aplicaciones con tipado estático y alta eficiencia, en conjunto con una sintaxis clara y

[27] https://golang.org

facilidades para multiprocesamiento. Estas características lo convierten en una opción interesante para desarrollar microservicios.

- MongoDB[28]: es una base de datos no relacional basada en documentos con formato JSON. Es ampliamente usada y provee varias funcionalidades relevantes como escalamiento horizontal y agregaciones.
- Docker[29]: posibilita la creación de entornos livianos y automatizables. Estos entornos, denominados *containers,* constan de todo lo necesario para que la aplicación pueda ejecutarse correctamente, como el sistema operativo, compiladores, libraries o configuraciones.
- Redis[30]: es una base de datos de tipo clave-valor que reside en memoria. Puede utilizarse como cache o queue, y soporta escalamiento horizontal.

Figura 5-20



Las flechas azules indican interacciones asincrónicas.

[28] https://www.mongodb.com
[29] https://www.docker.com
[30] https://redis.io

En la figura 5-20, se muestran los diferentes componentes del sistema. En total, existen 6 containers: dos corresponden a cada servicio, dos a componentes complementarios, uno a MongoDB y otro a Redis. Dado que es una PoC, estos últimos son compartidos por todos los servicios y componentes. Claramente, en un entorno productivo, esto no es recomendable.

La posibilidad de traducir los entornos de los componentes en containers de Docker nos facilita la ejecución de pruebas locales y la automatización de los tests. En la PoC realizada,, cada uno de los servicios posee tests de integración. Asimismo, dado que el repositorio de código está alojado en Github, podemos usar su funcionalidad de CI para verificar la correctitud de cada cambio en el código.[31]

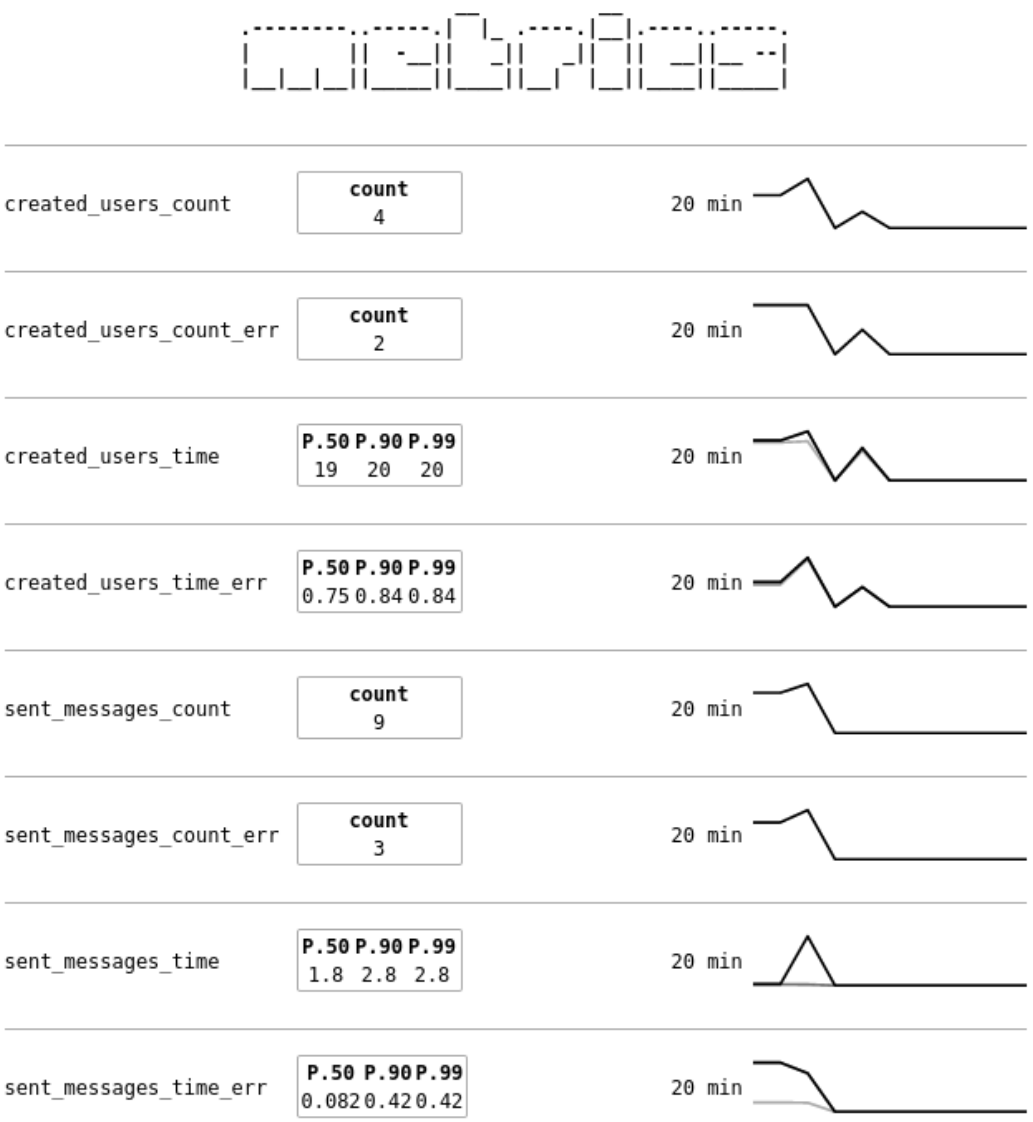
Analicemos de forma más detallada cada servicio y componente:

- Usuarios: utiliza Redis como base de datos porque, como mencionamos previamente, no son necesarias capacidades de búsqueda de usuarios. Además, se comunica con el servicio Auth cuando es necesario guardar la contraseña de un usuario nuevo.
- Mensajes: en este caso, la base de datos es Mongo para permitirnos búsquedas más complejas. La interacción con el servicio Usuarios es imprescindible para realizar el envío de los mensajes, ya que permite verificar que el receptor sea válido. Por ello, existe una cache de usuarios que mejora la disponibilidad y el rendimiento general. La misma se actualiza mediante eventos enviados por Usuarios.
- Auth: este componente guarda las contraseñas de los usuarios y nos da la posibilidad de generar access tokens a partir de esta información. Recordemos que cada servicio tiene la posibilidad de verificar localmente estos tokens, ya que son JSON Web Tokens.
- Métricas: es el encargado de recolectar métricas sobre el envío de mensajes y la creación de usuarios. Cada vez que se realiza una de estas operaciones, el servicio responsable envía el valor de la métrica a este componente de forma asincrónica y mediante una API. En la figura 5-21, se puede observar un ejemplo de métricas recolectadas.

[31] Las últimas ejecuciónes pueden observarse en https://github.com/lozaeric/dupin/actions

Esta prueba de concepto muestra varias de las decisiones de diseño y buenas prácticas adoptadas a lo largo del presente trabajo. El código completo de cada servicio y componente, junto a la configuración de los containers y documentación adicional, puede encontrarse en https://github.com/lozaeric/dupin .[32]

Figura 5-21



[32] Auguste Dupin es un detective de ficción, creado por Edgar Allan Poe. Además, es un nombre breve y fácil de recordar.

6. Conclusión

Es sabido que hay muchas formas técnicas de resolver el mismo problema y cada una tiene sus ventajas y desventajas. En el comienzo de este capítulo, se analizan algunas alternativas relevantes a las decisiones tomadas a lo largo de esta tesis. Posteriormente, se mencionan consideraciones a tener en cuenta en futuros trabajos. Sobre el final, encontramos reflexiones relacionadas a la presente tesis y al impacto de la tecnología en el mundo actual.

6.1. Nuevas tendencias

Podemos considerar a la arquitectura adoptada en este trabajo como un esquema clásico de microservicios. Se utilizaron APIs REST que se comunican mediante HTTP y en formato JSON. La infraestructura consta de clusters de instancias que escalan horizontalmente y se recomienda una base de datos no relacional. No obstante, esta no es la única manera de abordar estas cuestiones. En este apartado, vamos a enumerar algunas alternativas.

Event sourcing

Los modelos de datos mencionados constan de entidades con campos definidos y relacionadas entre sí. Cuando una de ellas cambia, simplemente se modifican sus valores y, de esta manera, siempre tenemos disponible la última versión de cada entidad. Como veremos, existe otro enfoque que se centra en los cambios propiamente dichos.

Richardson (2018) define *event sourcing* como una forma diferente de estructurar la lógica de negocios y la persistencia de los datos. Cada evento representa un cambio de estado en una entidad y, por ello, es inmutable. De esta manera, distintos conjuntos de eventos conforman la información disponible en el sistema.

En event sourcing, la aplicación sólo puede obtener el estado actual de cada entidad a partir del procesamiento de todos sus eventos históricos. En otras palabras, obtener el último estado requiere la existencia de vistas materializadas, usualmente implementadas mediante caches, que contienen esta información.

Este tipo de arquitectura cuenta con varios beneficios. La capacidad de preservar la historia de los cambios nos facilita su auditoría. Además, es posible procesarlos una y otra vez, en caso de que una nueva versión de la aplicación así lo necesite. Asimismo, el hecho de realizar la

comunicación exclusivamente mediante eventos reduce el acoplamiento entre las partes y facilita la integración de nuevos componentes.

Más allá de las ventajas mencionadas, pensar y diseñar un sistema basado en eventos no es trivial e implica un cambio de mentalidad importante con respecto a los modelos clásicos. Además, las vistas materializadas suelen ser eventualmente consistentes y, en ciertos casos de uso, esto puede no ser adecuado.

Apache Avro

JSON es un formato con varios aspectos positivos como su legibilidad, su versatilidad y su popularidad. Sin embargo, carece de un esquema de campos definido y consume mucho espacio en comparación con otros formatos disponibles. En particular, vamos a analizar un formato binario basado en esquemas denominado Avro, surgido en 2009.

Kleppmann (2017) realiza una comparación entre los distintos formatos, donde codifica los mismos valores y compara los tamaños resultantes. Puntualmente, Avro utiliza 32 bytes mientras JSON necesita 81 bytes para la misma información. Esto representa un ahorro de casi el 60% en tamaño. Las implicancias de esta diferencia en sistema de gran escala es enorme, ya que reduce drásticamente el uso de red. Avro logra esta reducción mediante el guardado de los datos en formato binario. Claramente, esto también genera que no sea legible, a diferencia de JSON.

El uso de un esquema con los campos y tipos de datos definidos es obligatorio en Avro porque permite la decodificación de la información. Este esquema siempre está actualizado y se transforma en una buena forma de documentar la estructura de datos. Asimismo, nos posibilita hacer validaciones de compatibilidad hacia atrás y hacia adelante al momento de modificarlo.

Por último, es importante reconocer el contexto en el cual se puede usar una solución como Avro. En líneas generales, podemos diferenciar dos tipos de APIs: públicas e internas. Las primeras tienen que usar soluciones estándar como JSON o GraphQL para facilitar la integración por parte de los clientes. En cambio, las internas no tienen esta obligación y es posible implementar formatos como el descrito.

Serverless

En Instancias, realizamos un breve repaso histórico de cómo fue cambiando el entorno donde se ejecuta la aplicación. Hay una tendencia a que este contexto sea cada vez más abstracto y efímero. En conjunto con este cambio, los proveedores de servicios en la nube mejoraron enormemente su capacidad para administrar infraestructura de forma automática y, en la actualidad, proporcionan una amplia variedad de servicios nuevos.[33] Estos dos procesos derivaron en el surgimiento de las arquitecturas *serverless*. Sbarski (2017) las define de la siguiente manera:

- Se divide la aplicación en un conjunto de funciones que tienen un sólo propósito y no mantienen un estado
- Se usan servicios de la nube con capacidad de ejecutar código bajo demanda.
- Las funciones que componen el sistema se comunican mediante eventos.
- Los frontends tienen más responsabilidad y, por ende, son más complejos
- Siempre que sea posible, se utilizan servicios de terceros para resolver funcionalidades puntuales.

Uno de los grandes beneficios de serverless es la simpleza para escalar. Es decir, no necesitamos hacer un diseño de clusters de instancias donde cada uno tiene flujos de trabajo determinados y escalan horizontalmente. Toda esta complejidad la aborda el proveedor que se contrate. Asimismo, el uso de recursos es mucho más eficiente en este tipo de arquitectura porque se paga por el tiempo durante el cual la función se ejecutó, no por instancia individual.

Como desventaja, podemos nombrar la complejidad potencial del sistema debido a que las funciones pueden ser cientos o miles. Esto dificulta la comprensión de todos los flujos y también el proceso de testing y deployment. Adicionalmente, se genera una dependencia fuerte con el proveedor del servicio que utilicemos para configurar y ejecutar las funciones.

[33] https://aws.amazon.com/es/lambda/features/

6.2. Próximos pasos

Considero que el trabajo futuro debe enfocarse en los siguientes puntos:

- Análisis de requerimientos funcionales
- Implementación completa de cada servicio
- Diseño y desarrollo del frontend

Las funcionalidades actuales cumplen perfectamente los requerimientos mínimos de un sistema de mensajería, pero probablemente deban analizarse más en detalle. Es necesario una investigación más rigurosa de los requerimientos funcionales de una aplicación de esta naturaleza. Seguramente, se requiera una comparación con sistemas de mensajería ya existentes en la actualidad.

En el diseño propuesto, se plantearon lineamientos generales del backend del sistema. Todas las decisiones fueron tomadas pensando en los requerimientos de las aplicaciones modernas y las buenas prácticas actuales. Sin embargo, esto no quita la posibilidad de modificar estas decisiones durante el desarrollo final. Ninguna etapa es definitiva, puede y debe repensarse a lo largo de todo el ciclo de desarrollo.

Uno de los requisitos principales de una aplicación de este tipo es que sea simple e intuitiva de usar. La experiencia de usuario debe ser analizada con sumo detalle para lograrlo. Además, considero que la arquitectura del frontend debe ser pensada a partir de los lineamientos generales de este trabajo, priorizando la agilidad, la escalabilidad, la disponibilidad y la mantenibilidad.

6.3. Reflexiones finales

A lo largo de esta tesis, hemos visto una amplia variedad de temas, desde la composición de una instancia hasta el cambio organizacional que implican los microservicios. Este trabajo puede pensarse como una introducción a todo estos tópicos, ya que cada uno en sí tiene una complejidad inabarcable. Y no sólo es profuso el conocimiento actual, todo el tiempo surgen nuevas soluciones y se reinventan las existentes.

La tecnología cambia de forma acelerada y, así también, su impacto en la vida cotidiana de las personas. Implícitamente, esta tesis narra una parte de ese proceso, donde se modificó la forma en la que pensamos, diseñamos e implementamos las aplicaciones. Este cambio continúa y es uno de los hechos más desafiantes y apasionantes de la informática.

> “La tecnología es una fuerza poderosa en nuestra sociedad. Los datos, el software y las comunicaciones pueden ser usados para el mal: consolidar estructuras de poder injustas, socavar los derechos humanos y proteger intereses establecidos. Pero también pueden ser usados para el bien: hacer que se escuchen las voces de las personas subrepresentadas, crear oportunidades para todos y prevenir desastres.” (Kleppmann, 2017)

Bibliografía

Atlassian. (2020). *Beyond the basics of scaling agile*. https://pages.eml.atlassian.com/rs/594-ATC-127/images/Beyond-the-basics-of-scaling-agile-white-paper.pdf

Christensen, B. (2012). *Fault Tolerance in a High Volume, Distributed System*. https://netflixtechblog.com/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a

Conway, M. (1968). How Do Committees Invent. *Datamation*. https://www.melconway.com/Home/pdf/committees.pdf

Cordeiro, R. H. (2020). *Proof of Concept or pilot*. https://docs.microsoft.com/en-us/azure/architecture/serverless-quest/poc-pilot

Davis, C. (2019). *Cloud Native Patterns*. Manning.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Foote, B., & Yoder, J. (1999). *Big Ball of Mud*. http://laputan.org/mud/mud.html

Fowler, M. (2014). *Microservices*. https://www.martinfowler.com/articles/microservices.html

Fowler, M. (2017). *What do you mean by "Event-Driven"?* https://martinfowler.com/articles/201701-event-driven.html.

Fowler, M. (2019). *Is High Quality Software Worth the Cost?* https://martinfowler.com/articles/is-quality-worth-cost.html

Fowler, M. (2019). *WaterfallProcess*. https://martinfowler.com/bliki/WaterfallProcess.html

Fowler, S. (2017). *Production-Ready Microservices*. O'Reilly.

Garg, P., & Kohnfelder, L. (1999). *The threats to our products*.
https://www.microsoft.com/security/blog/2009/08/27/the-threats-to-our-products/

Geers, M. (2020). *Micro Frontends in Action*. Manning.

Google - Devops Research & Assesment. (2019). *Accelerate State of DevOps Report*.
https://services.google.com/fh/files/misc/state-of-devops-2019.pdf

Higginbotham, J. (2015). *Designing Great Web APIs*. O'Reilly.

Homer, A., Sharp, J., Narumoto, M., & Swanson, T. (2014). *Cloud Design Patterns*. Microsoft.

The Internet Society. (1999). *Hypertext Transfer Protocol -- HTTP/1.1*.
https://tools.ietf.org/html/rfc2616

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly.

Lauret, A. (2019). *The Design of Web APIs*. Manning.

Madden, N. (2020). *API Security in Action*. Manning.

Maina, T. M. (2013). *Instant messaging an effective way of communication in workplace*.
https://arxiv.org/pdf/1310.8489.pdf

Mason, R. (2017). *Have you had your Bezos moment? What you can learn from Amazon*.
https://www.cio.com/article/3218667/have-you-had-your-bezos-moment-what-you-can-learn-from-amazon.html

McCance, G. (2012). *CERN Data Centre Evolution*.
https://www.slideshare.net/gmccance/cern-data-centre-evolution

Narumoto, M. (2017). *Autoscaling*.
https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

National Institute of Standards and Technology. (2011). *The NIST Definition of Cloud Computing*.
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Newman, S. (2015). *Building Microservices*. O'Reilly.

Newman, S. (2019). *Monolith to Microservices*. O'Reilly.

Pew Research Center. (2018). *Social Media Use in 2018*. https://www.pewresearch.org/internet/2018/03/01/social-media-use-in-2018/

Programmable Web. (2019). *APIs show Faster Growth Rate in 2019 than Previous Years*. https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17

Protalinski, E. (2011). *Facebook the fastest social network, second in uptime (Q3 2011)*. https://www.zdnet.com/article/facebook-the-fastest-social-network-second-in-uptime-q3-2011/

Quan-Haase, A., Cothrel, J., & Wellman, B. (2005). *Instant Messaging for Collaboration: a Case Study of a High-Tech Firm*. https://doi.org/10.1111/j.1083-6101.2005.tb00276.x

Reichert, D. (2018). *Logs and Metrics: What are they, and how do they help me?* https://www.sumologic.com/blog/logs-metrics-overview/

Reselman, B. (2020). *An Architect's guide to APIs: SOAP, REST, GraphQL, and gRPC*. https://www.redhat.com/architect/apis-soap-rest-graphql-grpc

Richardson, C. (2018). *Microservices Patterns*. Manning.

Sbarski, P. (2017). *Serverless Architectures on AWS*. Manning.

Schmaus, B. (2011). *Making the Netflix API More Resilient*. https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d

Tilkov, S. (2015). *Don't start with a monolith*. https://martinfowler.com/articles/dont-start-monolith.html

Tong, Z. (2014). *Averages Can Be Misleading: Try a Percentile*. https://www.elastic.co/blog/averages-can-dangerous-use-percentile