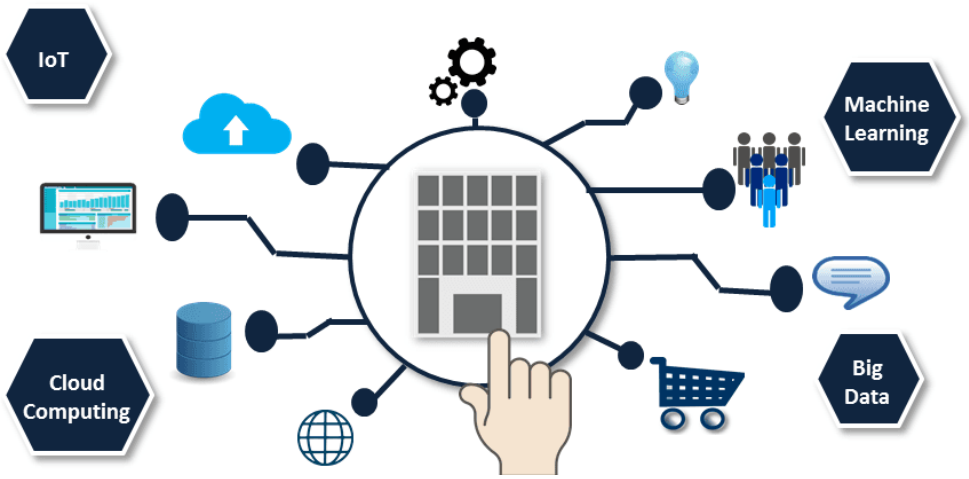
Application Programing Interfaces (APIs)

INTRODUCCIÓN

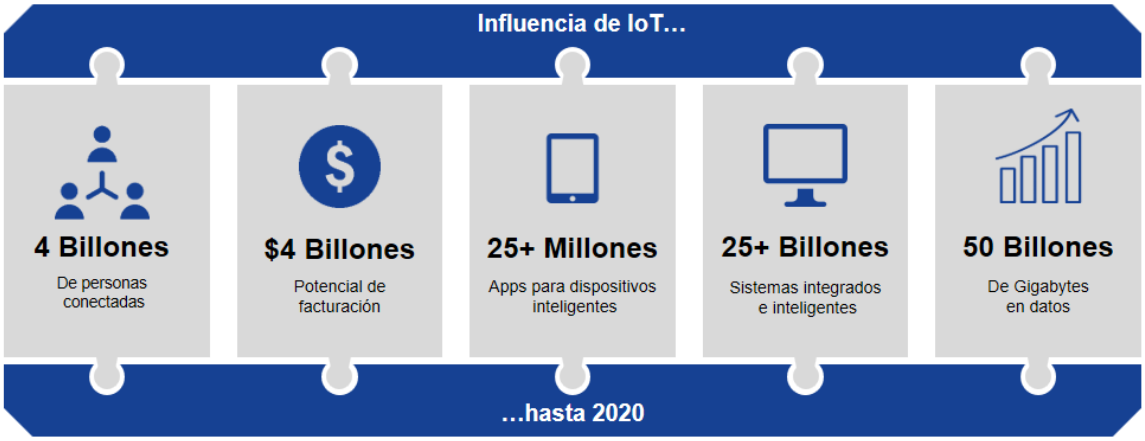
La transformación digital

- Los clientes buscan establecer relaciones con las empresas a través de diferentes dispositivos y canales. La transformación digital permite a las empresas responder con agilidad a las demandas de interrelación con clientes y otras partes interesadas. Las APIs (Application Programming Interface) son el mecanismo en el que se basa esta capacidad de respuesta. Mediante una API el desarrollador es capaz de construir una aplicación que dé respuesta a las necesidades de un grupo de clientes o que permita la comunicación directa entre organizaciones.
- Las APIs actuales tienen su origen en los primeros intentos de integración de sistemas donde las organizaciones comunicaban diferentes sistemas en aras de automatizar procesos manuales o de obtener una información global de sus sistemas. Estos sistemas actualmente han logrado contar con interfaces sencillos, con mecanismos de comunicación estándar y manejo de información de forma simple y desacoplada. Todo lo cual facilita su utilización de forma ágil por aplicaciones móviles, IoT u otras aplicaciones dentro o fuera de una organización.
- Igualmente, al basar su comunicación en el protocolo HTTP, las APIs facilitan la comunicación con o desde sistemas en la nube. Abriendo de esta forma el abanico de servicios con el que una organización puede dar servicio a sus clientes o los canales a través de los cuales dota dichos servicios. El paradigma de comunicaciones y consumo de servicios digitales actual conlleva que para garantizar el éxito en la adopción de un producto las organizaciones no solo deban proporcionar el mejor producto si no la mejor forma de integrarse digitalmente con el servicio de las APIs.
- La transformación digital se puede definir como la integración de las nuevas tecnologías en todas las áreas de una organización u otros ámbitos para cambiar su forma de funcionar.

Transformación digital

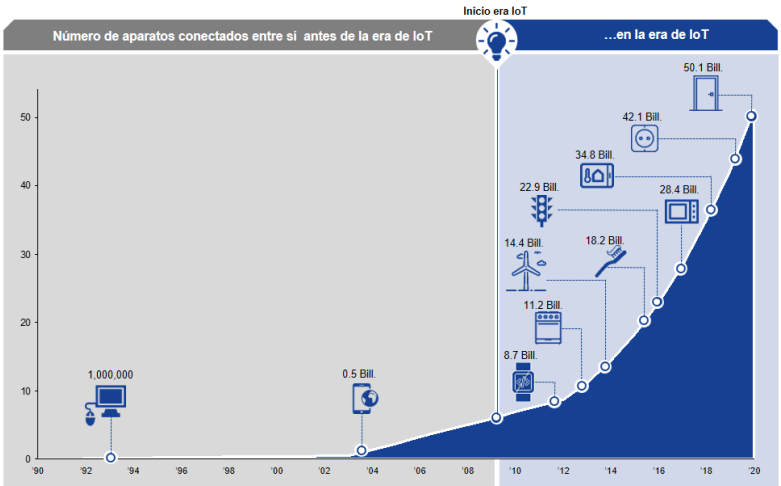


Influencia de IoT



https://www.fostec.com/wp-content/uploads/Internet-de-las-cosas-1-ES.png

Desarrollo explosivo de IoT



https://www.fostec.com/wp-content/uploads/Internet-de-las-cosas-2-ES.png

La nube

- La nube está cambiando la forma en que se diseñan y protegen las aplicaciones. En lugar de ser monolitos, las aplicaciones se descomponen en servicios menores y descentralizados. Estos servicios se comunican a través de APIs, mediante el uso de eventos o de mensajería asincrónica. Las aplicaciones se escalan horizontalmente, agregando nuevas instancias, tal y como exigen las necesidades.
- Estas tendencias agregan nuevos desafíos:
  - El estado de las aplicaciones se distribuye.
  - Las operaciones se realizan en paralelo y de forma asincrónica.
  - Las aplicaciones deben ser resistentes cuando se produzcan errores.
  - Las aplicaciones son continuamente atacadas por actores malintencionados.
  - Las implementaciones deben estar automatizadas y ser predecibles.
  - La supervisión y la telemetría son fundamentales para obtener una visión general del sistema.

Cambio de paradigma

Local tradicional

- Monolítica
- Diseñada para una escalabilidad predecible
- Base de datos relacional
- Procesamiento síncrono
- Diseño para evitar errores (MTBF)
- Actualizaciones grandes, ocasionales
- Administración manual
- Servidores en copo de nieve

Nube moderna

- Descompuesto
- Diseñado para un escalado elástico
- Persistencia poliglota (combinación de tecnologías de almacenamiento)
- Procesamiento asíncrono
- Diseño resiliente a errores (MTTR)
- Pequeñas actualizaciones, frecuentes
- Administración automatizada
- Infraestructura inmutable

¿Qué es una API?

- API es el acrónimo de Application Programming Interface, que es un intermediario de software que permite que dos aplicaciones se comuniquen entre sí.
- A lo largo de los años, lo que es una API a menudo se ha descrito como cualquier tipo de interfaz de conectividad genérica para una aplicación. Más recientemente, sin embargo, una API moderna ha adquirido algunas características que las hacen extraordinariamente valiosas y útiles:
  - Las API modernas se adhieren a los estándares (generalmente HTTP y REST), que son amigables para los desarrolladores, de fácil acceso y comprensibles ampliamente
  - Se tratan más como productos que como código. Están diseñados para el consumo de audiencias específicas, están documentados y están versionadas de manera que los usuarios puedan tener ciertas expectativas sobre su mantenimiento y ciclo de vida.
  - Debido a que están mucho más estandarizados, tienen una disciplina mucho más sólida para la seguridad y la gobernanza, además de monitorear y administrar el rendimiento y la escalabilidad.
  - Como cualquier otra pieza de software producido, una API moderna tiene su propio ciclo de vida de desarrollo de software (SDLC) de diseño, prueba, construcción, administración y control de versiones.

API Strategy

- Una empresa debe desarrollar una estrategia de API que consista en APIs tanto públicas como privadas. Cuando una empresa lanza APIs públicas que potencian las aplicaciones orientadas al consumidor, habilita nuevas formas de interactuar y conectarse con sus clientes a través de aplicaciones web, móviles y sociales. Al desarrollar APIs privadas, las empresas pueden ofrecer a sus empleados y socios nuevas herramientas que les ayuden a agilizar las operaciones y servir a los clientes aún mejor. En este entorno dinámico, a medida que más y más empresas crean e incorporan APIs, es cada vez más crítico que las empresas innovadoras desarrollen y ejecuten estrategias API de éxito.
- Como ejemplo de los beneficios que una API Strategy puede aportar a una organización, alrededor de 2002, Jeffrey Preston Bezos, director ejecutivo de Amazon, envió un correo a sus empleados con los siguientes puntos:
  - Todos los equipos expondrán sus datos y funcionalidad a través de interfaces de servicios.
  - Los equipos deben comunicarse entre sí a través de estas interfaces.
  - No se permitirá otra forma de comunicación: ni vinculación directa, ni acceso directo a bases de datos de otros equipos, ni memoria compartida ni utilización de ningún tipo de puerta trasera. Sólo se permitirán comunicaciones a través de llamadas que utilicen interfaces de red.
  - La tecnología empleada por cada equipo no debe ser un problema.
  - Todas las interfaces de los servicios, sin excepción, deben ser diseñadas con el objetivo de ser externalizables. Esto es, el equipo debe planear y diseñar sus interfaces para los desarrolladores del resto del mundo. Sin excepciones.
- El correo finalizaba de la siguiente manera: “Todo aquel que no siga las directrices será despido. Gracias, ¡pasad un buen día!”. Desde hace ya varios años Amazon es el primer proveedor IaaS mundial distanciado significativamente de sus competidores.

API Economy

- El ecosistema de APIs especifica de qué manera el uso de estas micro aplicaciones por terceros puede beneficiar económicamente a una organización, bien por reducción de costes o bien por alquiler o venta de sus propios desarrollos:
  - API as a Service: Obtención de beneficios mediante la exposición de APIs de servicios que son valiosos para terceros y están dispuestos a pagar por su uso.
  - API Products: Desarrollo de herramientas encargadas de facilitar la exposición e integración de aplicaciones a través de sus APIs.
- Una API Economy es, en definitiva, un servicio basado en API que demuestra algún tipo de rentabilidad al negocio, ya sea económica o estratégicamente. Es fundamental pensar en la API como un producto. La economía está cambiando gracias a que las APIs abren nuevos canales, tanto de ingresos como de innovación.
- En general existen muchos servicios APIs de terceros que permiten a un negocio escalar rápidamente para crear productos finales con una inversión y riesgo mínimo.
- Una empresa puede cambiar su estrategia de ventas a la comercialización como proveedor de servicios APIs a terceros: los recursos aquí son sus datos y servicios. La API Economy es un facilitador para convertir una empresa u organización en una plataforma.

Tipos de API por propósito

- Es raro que una organización decida que necesita una API de la nada; la mayoría de las veces, las organizaciones comienzan con una idea, aplicación, innovación o caso de uso que requiere conectividad a otros sistemas o conjuntos de datos. Las APIs entran en escena como un medio para permitir la conectividad entre los sistemas y los conjuntos de datos que deben integrarse.
- Las organizaciones pueden implementar APIs para muchos propósitos: desde exponer internamente la funcionalidad de un sistema central hasta habilitar una aplicación móvil orientada al cliente. El marco de conectividad incluye:
  - APIs del sistema: las APIs del sistema desbloquean datos de los sistemas centrales de registro dentro de una organización. Los ejemplos de sistemas críticos de los que las API podrían desbloquear datos incluyen ERP, sistemas de facturación, CRM y bases de datos.
  - APIs de proceso: las APIs de proceso interactúan y dan forma a los datos dentro de un solo sistema o entre sistemas, rompiendo los silos de datos. Las APIs de proceso proporcionan un medio para combinar datos y organizar varias APIs del sistema para un propósito comercial específico. Algunos ejemplos de esto incluyen la creación de una vista de 360 grados del cliente, el cumplimiento del pedido y el estado del envío.
  - APIs de experiencia: las APIs de experiencia proporcionan un contexto empresarial para los datos y procesos que se desbloquearon y establecieron con las APIs de proceso y sistema. Las APIs de experiencia exponen los datos para que los consuma su público objetivo; esto funciona en un amplio conjunto de canales en una variedad de formas. Algunos ejemplos son las aplicaciones móviles, los portales internos para los datos del cliente o un sistema de cara al cliente que rastrea las entregas.

Tipos de API por estrategias de gestión

- Una vez que se haya determinado el caso de uso de las APIs en la organización, es hora de determinar quién accederá a estas APIs. La mayoría de las veces, el caso de uso y el usuario previsto van de la mano; por ejemplo, es posible que desee mostrar los datos del cliente para sus agentes de ventas y servicios internos; el usuario final previsto, en este caso, son los empleados internos.
- Los tres tipos de APIs según cómo se administran y quién accede a ellas son:
  - APIs externas: Los terceros, que son externos a la organización, pueden acceder a las APIs externas . A menudo, hacen que los datos y servicios de una organización sean fácilmente accesibles en autoservicio por desarrolladores de todo el mundo que buscan crear aplicaciones e integraciones innovadoras.
  - APIs internas: Las APIs internas son lo opuesto a las APIs abiertas, ya que no son accesibles para los consumidores externos y solo están disponibles para los desarrolladores internos de una organización. Las APIs internas pueden permitir iniciativas en toda la empresa, desde la adopción de DevOps y arquitecturas de microservicios hasta la modernización heredada y la transformación digital. El uso y la reutilización de estas APIs pueden mejorar la productividad, la eficiencia y la agilidad de una organización.
  - APIs de socios: Las APIs de socios se encuentran en algún lugar entre las APIs internas y externas. Son APIs a las que acceden otras personas ajenas a la organización con permisos exclusivos. Por lo general, este acceso especial se otorga a terceros específicos para facilitar una asociación comercial estratégica. Un caso de uso común de una API de socio es cuando dos organizaciones desean compartir datos entre sí, se configuraría una API de socio para que cada organización tenga acceso a los datos necesarios con el conjunto correcto de credenciales y permisos.

API First

- El enfoque basado en API First significa que, para cualquier proyecto de desarrollo dado, las APIs se tratan como "ciudadanos de primera clase": que todo sobre un proyecto gira en torno a la idea de que el producto final es un conjunto de APIs consumido por las aplicaciones del cliente.
- El enfoque de API First implica que los desarrollos de APIs sean consistentes y reutilizables, lo que se puede lograr mediante el uso de un lenguaje formal de descripción de APIs para establecer un contrato sobre cómo se supone que se comportará la API. Establecer un contrato implica pasar más tiempo pensando en el diseño de una API.
- A menudo también implica una planificación y colaboración adicionales con las partes interesadas, proporcionando retroalimentación de los consumidores sobre el diseño de una API antes de escribir cualquier código evitando costosos errores.
- Entre sus ventajas se encuentran:
  - Los equipos de desarrollo pueden trabajar en paralelo.
  - Reduce el coste de desarrollar aplicaciones
  - Aumenta la velocidad de desarrollo.
  - Asegura buenas experiencias de desarrollador
  - Reduce el riesgo de fallos
  - Proporcionar un sólido perímetro de seguridad

Ciclo de vida de las API

https://www.postman.com/api-first/

Estilos de arquitectura

- **N Niveles**: es una arquitectura tradicional para aplicaciones empresariales.
- **Web-queue-worker**: solución puramente PaaS, la aplicación tiene un front-end web que controla las solicitudes HTTP y un trabajador back-end que realiza tareas de uso intensivo de la CPU u operaciones de larga duración. El front-end se comunica con el trabajador a través de una cola de mensajes asincrónicos.
- **Orientada a servicios (SOA)**: término sobre utilizado pero, como denominador común, significa que se estructura descomponiéndola en varios servicios que se pueden clasificar en tipos diferentes, como subsistemas o niveles.
- **Microservicios**: en un sistema que requiere alta escalabilidad y alto rendimiento, la arquitectura de microservicios se descompone en muchos servicios pequeños e independientes.
- **Basadas en eventos**: usa un modelo de publicación-suscripción (pub-sub), en el que los productores publican eventos y los consumidores de suscriben a ellos. Los productores son independientes de los consumidores y estos, a su vez, son independientes entre sí.
- **Big Data**: permite dividir un conjunto de datos muy grande en fragmentos, realizando un procesamiento paralelo en todo el conjunto, con fines de análisis y creación de informes.
- **Big compute**: también denominada informática de alto rendimiento (HPC), realiza cálculos en paralelo en un gran número (miles) de núcleos.

Componentización a través de APIs

- Un componente es una unidad de software que es reemplazable y actualizable de manera independientemente.
- Definimos las librerías como componentes que están vinculados a un programa y se llaman mediante llamadas a función en memoria, mientras que los APIs son componentes fuera de proceso que se comunican con un mecanismo como una solicitud de servicio web o una llamada a procedimiento remoto.
- Las arquitecturas de APIs usarán librerías, pero su manera primaria de componentización y reutilización es dividir en servicios.
- La razón principal para usar servicios como componentes (en lugar de bibliotecas) es que los servicios son desplegables de forma independiente.
- Otra consecuencia es una interfaz de componentes más explícita.

Estilos de comunicación

- El cliente y los servicios, o los servicios entre si, pueden comunicarse a través de muchos tipos diferentes de comunicación, cada uno destinado a un escenario y unos objetivos distintos. Inicialmente, estos tipos de comunicaciones se pueden clasificar por diferentes criterios.
- El primer criterio define si el proceso es sincrónico o asincrónico:
  - Protocolo síncrono: HTTP es el protocolo síncrono mas utilizado. El cliente envía una solicitud y espera una respuesta del servicio, solo puede continuar su tarea cuando recibe la respuesta del servidor. Es independiente de la ejecución de código de cliente, que puede ser síncrono (el subproceso está bloqueado) o asíncrono (el subproceso no está bloqueado y la respuesta dispara una devolución de llamada).
  - Protocolo asíncrono: Otros protocolos como AMQP (un protocolo compatible con muchos sistemas operativos y entornos de nube) usan mensajes asíncronos. Normalmente el código de cliente o el remitente del mensaje no espera ninguna respuesta. Simplemente se envía el mensaje a una cola de un agente de mensajes, que son escuchadas por los consumidores.
- El segundo criterio define si la comunicación tiene un único receptor o varios:
  - Receptor único 1:1 (comando, punto a punto, P2P): Cada solicitud debe ser procesada por un receptor o servicio exactamente (como en el patrón Command).
  - Varios receptores 1:N (eventos, publicación/subscripción, Pub/Sub): Cada solicitud puede ser procesada por entre cero y varios receptores. Este tipo de comunicación debe ser asíncrona (basada en un bus de eventos o un agente de mensajes).

Criterio según su base

- Basados en Recursos: Los servicios exponen información, documentos que incluyen tanto identificadores de datos como de acciones (enlaces y formularios), las operaciones CRUD están predefinidas. REST es un estilo arquitectónico que separa las preocupaciones del consumidor y del proveedor de la API al depender de comandos que están integrados en el protocolo de red subyacente. REST (Representational State Transfer) es extremadamente flexible en el formato de sus cargas útiles de datos, lo que permite una variedad de formatos de datos populares como JSON y XML, entre otros.
- Basados en Procedimientos: Las llamadas a procedimiento remoto, o RPC, generalmente requieren que los desarrolladores ejecuten bloques específicos de código en otro sistema: operaciones. RPC es independiente del protocolo, lo que significa que tiene el potencial de ser compatible con muchos protocolos, pero también pierde los beneficios de usar capacidades de protocolo nativo (por ejemplo, almacenamiento en caché). La utilización de diferentes estándares da como resultado un acoplamiento más estrecho entre los consumidores y los proveedores de API y las tecnologías implicadas, lo que a su vez sobrecarga a los desarrolladores involucrados en todos los aspectos de un ecosistema de APIs impulsado por RPC. Los patrones de arquitectura de RPC se pueden observar en tecnologías API populares como SOAP, GraphQL y gRPC.
- Basados en Eventos/Streaming: a veces denominadas arquitecturas de eventos, en tiempo real, de transmisión, asíncronas o push, las APIs impulsadas por eventos no esperan a que un consumidor de la API las llame antes de entregar una respuesta. En cambio, una respuesta se desencadena por la ocurrencia de un evento. Estos servicios exponen eventos, con una información mínima, a los que los clientes pueden suscribirse para recibir actualizaciones cuando cambian los valores del servicio. Hay un puñado de variaciones para este estilo que incluyen (entre otras) reactivo, publicador/suscriptor, notificación de eventos y CQRS.

Evolución histórica de los servicios

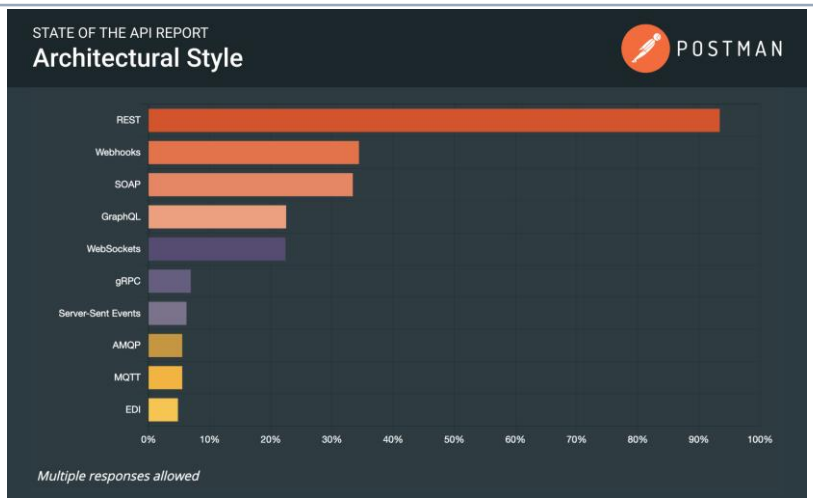
- Precursores:
  - RPC: Llamadas a Procedimientos Remotos
  - Binarios: CORBA, Java RMI, .NET Remoting
  - XML-RPC: Precursor del SOAP
- Actuales:
  - Servicios Web XML o Servicios SOAP
  - Servicios Web REST o API REST
    - WebHooks
  - Servicios GraphQL
  - Servicios gRPC

AÑO	Descripción
1976	Aparición de RPC (Remote Procedure Call) en Sistema Unix
1990	Aparición de DCE (Distributed Computing Environment) que es un sistema de software para computación distribuida, basado en RPC.
1991	Aparición de Microsoft RPC basado en DCE para sistemas Windows.
1992	Aparición de DCOM (Microsoft) y CORBA (ORB) para la creación de componentes software distribuidos.
1997	Aparición de Java RMI en JDK 1.1
1998	Aparición de XML-RPC
1999	Aparición de SOAP 1.0, WSDL, UDDI
2000	Definición del REST
2012	Propuesta de GraphQL por Facebook
2015	Desarrollo de gRPC por Google

Protocolos y Estándares

- Síncronos:
  - SOAP: Basado en operaciones, de tipos texto, en formato XML y comunicaciones síncronas.
  - REST: Basado en recursos, independiente de formato (texto/binario) y comunicaciones síncronas.
  - GraphQL: Basado en consultas, en formato JSON y comunicaciones síncronas.
  - gRPC: Basado en contratos, en el formato binario Protocol Buffers y comunicaciones síncronas/asíncronas.
- Asíncronos:
  - WebSockets: protocolo estándar abierto, elemental, texto y binario.
  - STOMP (Simple/Streaming Text Oriented Messaging Protocol): protocolo estándar abierto, basado en texto, soluciones simples y ligero.
  - AMQP (Advanced Message Queuing Protocol): protocolo estándar abierto, binario, encolamiento, P2P y Pub/Sub, exactitud y seguridad
  - MQTT (Message Queue Telemetry Transport): protocolo estándar abierto, binario, P2P, liviano, soluciones simples y seguridad
  - JMS (Java Message Service): API, binario Java, P2P y Pub/Sub

Estilos arquitectónicos mas utilizados



https://www.postman.com/state-of-api/api-technologies/#api-technologies

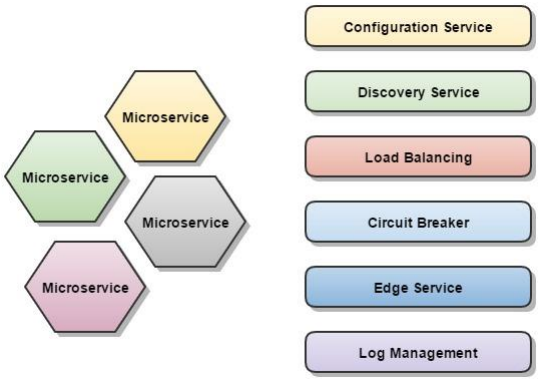
Preocupaciones transversales

- En referencia a las APIs, servicios y microservicios, la tendencia natural es a crecer, tanto por nuevas funcionalidades del sistema como por escalado horizontal.
- Todo ello provoca una serie de preocupaciones adicionales:
  - Localización de los servicios.
  - Balanceo de carga.
  - Tolerancia a fallos.
  - Gestión de la configuración.
  - Gestión de logs.
  - Gestión de los despliegues.
  - y otras …

Implantación

- Para la implantación de una arquitectura basada en APIs hemos tener en cuenta 3 aspectos principalmente:
  - Un modelo de referencia en el que definir las necesidades de una arquitectura de las APIs.
  - Un modelo de implementación en el que decidir y concretar la implementación de los componentes vistos en el modelo de referencia.
  - Un modelo de despliegue donde definir cómo se van a desplegar los distintos componentes de la arquitectura en los diferentes entornos.

Modelo de referencia



Modelo de referencia

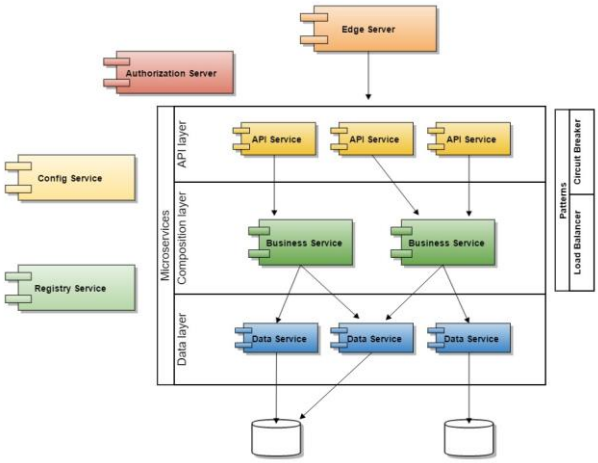
- Servidor perimetral / exposición de servicios (Edge server)
  - Será un gateway en el que se expondrán los servicios a consumir.
- Servicio de registro / descubrimiento
  - Este servicio centralizado será el encargado de proveer los endpoints de los servicios para su consumo. Todo microservicio, en su proceso de arranque, se registrará automáticamente en él.
- Balanceo de carga (Load balancer)
  - Este patrón de implementación permite el balanceo entre distintas instancias de forma transparente a la hora de consumir un servicio.
- Tolerancia a fallos (Circuit breaker)
  - Mediante este patrón conseguiremos que cuando se produzca un fallo, este no se propague en cascada por todo el pipe de llamadas, y poder gestionar el error de forma controlada a nivel local del servicio donde se produjo.
- Mensajería:
  - Las invocaciones siempre serán síncronas (REST, SOAP, …) o también llamadas asíncronas (AMQP).

Modelo de referencia

- Servidor de configuración central
  - Este componente se encargará de centralizar y proveer remotamente la configuración a cada API. Esta configuración se mantiene convencionalmente en un repositorio Git, lo que nos permitirá gestionar su propio ciclo de vida y versionado.
- Servidor de Autorización
  - Para implementar la capa de seguridad (recomendable en la capa de servicios API)
- Centralización de logs
  - Se hace necesario un mecanismo para centralizar la gestión de logs. Pues sería inviable la consulta de cada log individual de cada uno de los microservicios.
- Monitorización
  - Para poder disponer de mecanismos y dashboard para monitorizar aspectos de los nodos como, salud, carga de trabajo...

Modelo de referencia

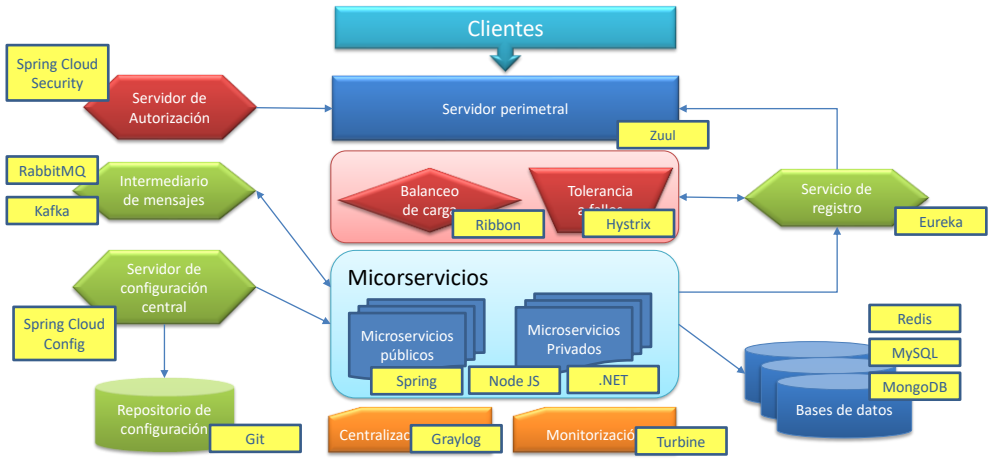
Modelo de referencia



Modelo de implementación (Netflix OSS)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Spring Cloud y Netflix OSS:
  - Microservicios propiamente dichos: Serán aplicaciones Spring Boot con controladores Spring MVC. Se puede utilizar Swagger para documentar y definir nuestra API.
  - Config Server: microservicio basado en Spring Cloud Config y se utilizará Git como repositorio de configuración.
  - Registry / Discovery Service: microservicio basado en Eureka de Netflix OSS.
  - Load Balancer: se puede utilizar Ribbon de Netflix OSS que ya viene integrado en REST-template de Spring.
  - Circuit breaker: se puede utilizar Hystrix de Netflix OSS.
  - Gestión de Logs: se puede utilizar Graylog
  - Servidor perimetral: se puede utilizar Zuul de Netflix OSS.
  - Servidor de autorización: se puede utilizar el servicio con Spring Cloud Security.
  - Agregador de métricas: se puede utilizar el servicio Turbine.
  - Intermediario de mensajes: se puede utilizar AMQP con RabbitMQ o Kafka.

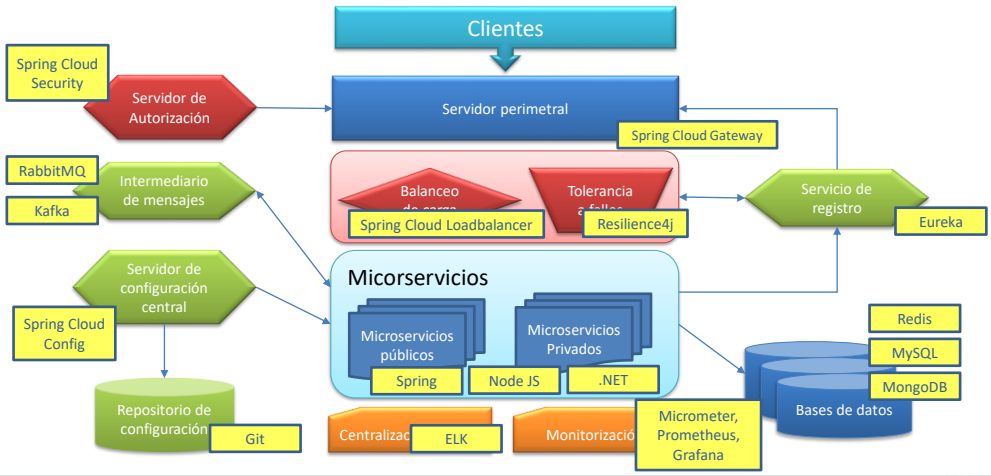
Modelo de implementación (Netflix OSS)



Modelo de implementación (Spring Cloud)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Spring Cloud y Netflix OSS:
  - Microservicios propiamente dichos: Serán aplicaciones Spring Boot con controladores Spring MVC. Se puede utilizar Swagger para documentar y definir nuestra API.
  - Config Server: microservicio basado en Spring Cloud Config y se utilizará Git como repositorio de configuración.
  - Registry / Discovery Service: microservicio basado en Eureka de Netflix OSS.
  - Load Balancer: se puede utilizar Spring Cloud Loadbalancer que ya viene integrado en REST-template de Spring.
  - Circuit breaker: se puede utilizar Spring Cloud Circuit Breaker con Resilience4j.
  - Gestión de Logs: se puede utilizar Elasticsearch, Logstash y Kibana
  - Servidor perimetral: se puede utilizar Spring Cloud Gateway.
  - Servidor de autorización: se puede utilizar el servicio con Spring Cloud Security.
  - Agregador de métricas: se puede utilizar el servicio Turbine.
  - Intermediario de mensajes: se puede utilizar AMQP con RabbitMQ o Kafka.

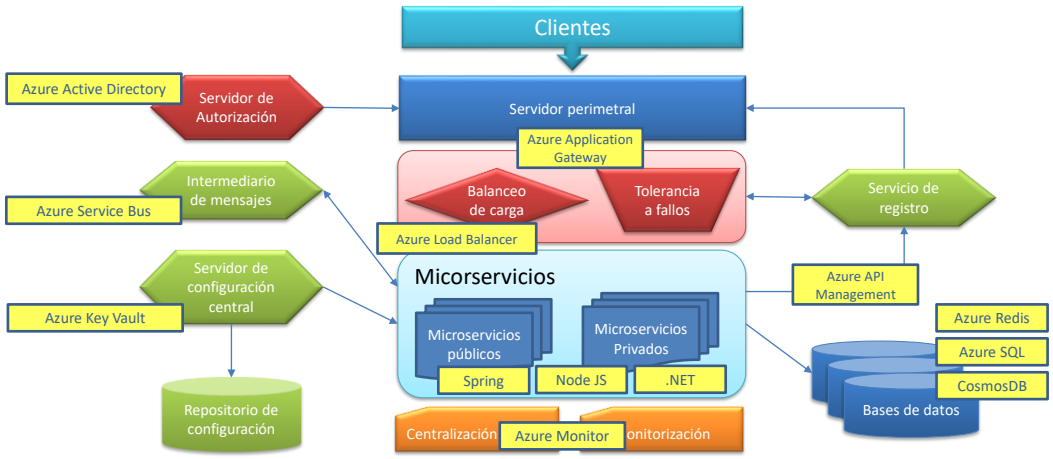
Modelo de implementación (Spring Cloud)



Modelo de implementación (Azure)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de suministrado por Azure:
  - Microservicios propiamente dichos: Serán aplicaciones ASP.NET Core con WebApi. Se puede utilizar OpenAPI para documentar y definir nuestra API.
  - Azure Key Vault: Se puede utilizar para almacenar de forma segura y controlar de manera estricta el acceso a los tokens, contraseñas, certificados, claves de API y otros secretos.
  - Azure API Management: es una solución completa para publicar API para clientes externos e internos.
    - Servidor perimetral, Registry / Discovery Service, Load Balancer (con Azure Application Gateway), Circuit breaker.
  - Servidor de autorización: Azure Active Directory (Azure AD) es un servicio de administración de identidades y acceso basado en la nube de Microsoft.
  - Azure Monitor: ayuda a maximizar la disponibilidad y el rendimiento de las aplicaciones y los servicios.
    - Agregador de métricas: Detección y diagnóstico de problemas en aplicaciones y dependencias con Application Insights.
    - Gestión de Logs: Profundización en sus datos de supervisión con Log Analytics para la solución de problemas y diagnósticos profundos.
  - Intermediario de mensajes: se puede utilizar AMQP con Azure Service Bus.

Modelo de implementación (Azure)

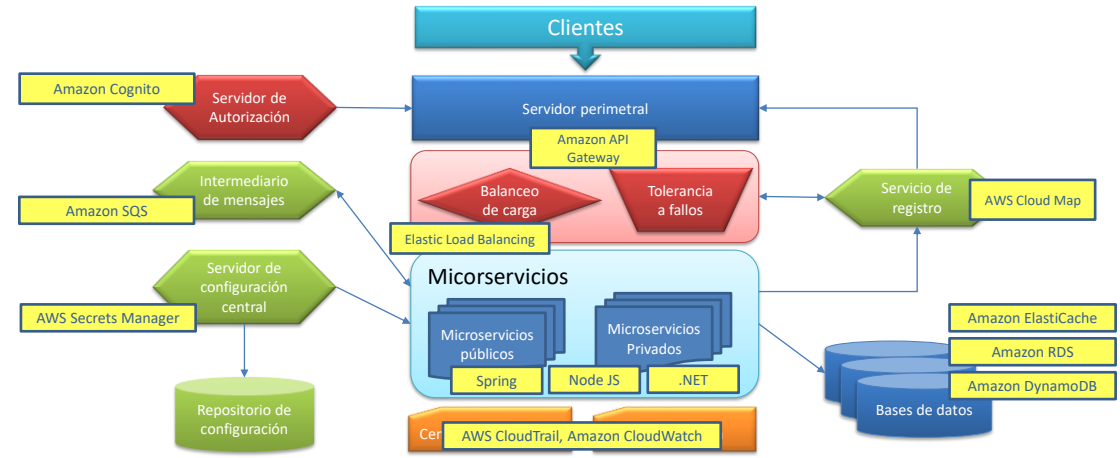


Modelo de implementación (AWS)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Amazon Web Services:
  - Amazon API Gateway: Proxy de la API
  - Elastic Load Balancing: Balanceador de carga de aplicaciones
  - AWS Cloud Map: Detección de servicios
  - Amazon RDS: Bases de datos relacionales
  - Amazon DynamoDB: Bases de datos NoSQL
  - Amazon ElastiCache: Almacenamiento en caché
  - Amazon Simple Queue Service (Amazon SQS): Colas de mensajes
  - AWS CloudTrail, Amazon CloudWatch: Monitorización de API
  - Amazon Cognito, AWS IAM: Registro, inicio de sesión y control de acceso de usuarios
  - AWS Secrets Manager, AWS KMS: Datos confidenciales de configuración

https://aws.amazon.com/es/microservices/

Modelo de implementación (AWS)



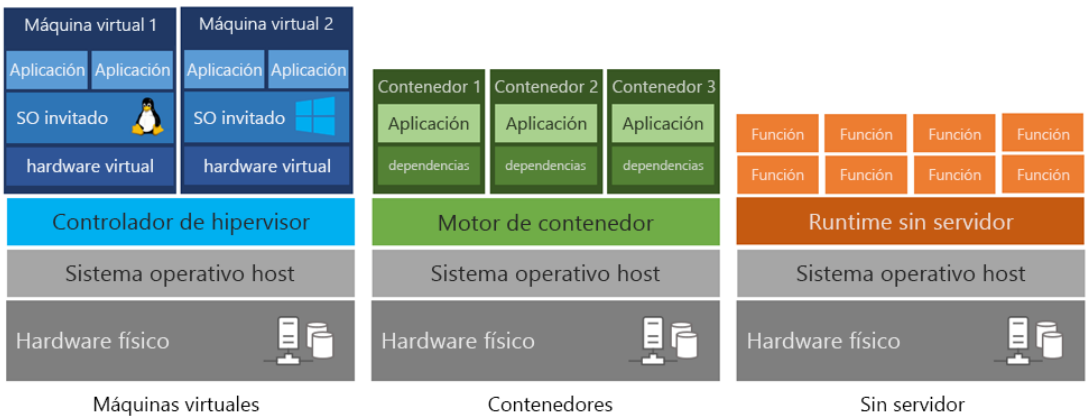
Modelo de despliegue

- El modelo de despliegue hace referencia al modo en que vamos a organizar y gestionar los despliegues de los microservicios, así como a las tecnologías que podemos usar para tal fin.
- El despliegue de los microservicios es una parte primordial de esta arquitectura. Muchas de las ventajas que aportan, como la escalabilidad, son posibles gracias al sistema de despliegue.
- Existen convencionalmente varios patrones en este sentido a la hora de encapsular microservicios:
  - Máquinas virtuales.
  - Contenedores.
  - Sin servidor: FaaS (Functions-as-a-Service)
- Los microservicios están íntimamente ligados al concepto de contenedores (una especie de máquinas virtuales ligeras que corren de forma independiente, pero utilizando directamente los recursos del host en lugar de un SO completo). Hablar de contenedores es hablar de Docker. Con este software se pueden crear las imágenes de los contenedores para después crear instancias a demanda.

Modelo de despliegue

- Las imágenes Docker son como plantillas. Constan de un conjunto de capas y cada una aporta un conjunto de software a lo anterior, hasta construir una imagen completa.
- Por ejemplo, podríamos tener una imagen con una capa Ubuntu y otra capa con un servidor LAMP. De esta forma tendríamos una imagen para ejecutar como servidor PHP.
- Las capas suelen ser bastante ligeras. La capa de Ubuntu, por ejemplo, contiene algunos los ficheros del SO y otros, como el Kernel, los toma del host.
- Los contenedores toman una imagen y la ejecutan, añadiendo una capa de lectura/escritura, ya que las imágenes son de sólo lectura.
- Dada su naturaleza volátil (el contenedor puede parar en cualquier momento y volver a arrancarse otra instancia), para el almacenamiento se usan volúmenes, que están fuera de los contenedores.

Contenedores



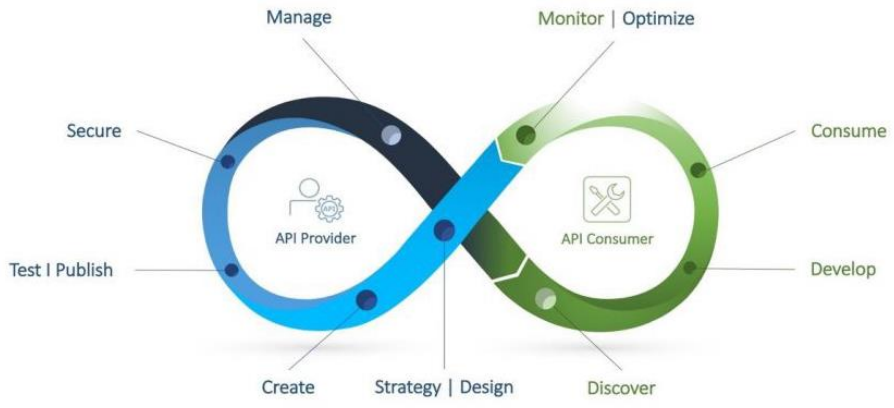
Modelo de despliegue

- Sin embargo, esto no es suficiente para dotar a nuestro sistema de una buena escalabilidad. El siguiente paso será pensar en la automatización y orquestación de los despliegues siguiendo el paradigma cloud. Se necesita una plataforma que gestione los contenedores, y para ello existen soluciones como Kubernetes.
- Kubernetes permite gestionar grandes cantidades de contenedores, agrupándolos en pods. También se encarga de gestionar servicios que estos necesitan, como conexiones de red y almacenamiento, entre otros. Además, proporciona también esta parte de despliegue automático, que puede utilizarse con sus componentes o con componentes de otras tecnologías como Spring Cloud+Netflix OSS.
- Todavía se puede dar una vuelta de tuerca más, incluyendo otra capa por encima de Docker y Kubernetes: Openshift. En este caso estamos hablando de un PaaS que, utilizando Docker y Kubernetes, realiza una gestión más completa y amigable de nuestro sistema de microservicios. Por ejemplo, nos evita interactuar con la interfaz CLI de Kubernetes y simplifica algunos procesos. Además, nos provee de más herramientas para una gestión más completa del ciclo de vida, como construcción, test y creación de imágenes. Incluye los despliegues automáticos como parte de sus servicios y, en sus últimas versiones, el escalado automático.
- Openshift también proporciona sus propios componentes, que de nuevo pueden mezclarse con los de otras tecnologías.

FaaS (Functions-as-a-Service)

- El auge de la informática sin servidor es una de las innovaciones más importantes de la actualidad. Las tecnologías sin servidor, como Azure Functions, AWS Lambda o Google Cloud Functions, permiten a los desarrolladores centrarse por completo en escribir código. Toda la infraestructura informática de la que dependen (máquinas virtuales (VM), compatibilidad con la escalabilidad y demás) se administra por ellos. Debido a esto, la creación de aplicaciones se vuelve más rápida y sencilla. Ejecutar dichas aplicaciones a menudo resulta más barato, porque solo se le cobra por los recursos informáticos que realmente usa el código.
- La arquitectura serverless habilita la ejecución de una aplicación mediante contenedores efímeros y sin estado; estos son creados en el momento en el que se produce un evento que dispare dicha aplicación. Contrariamente a lo que nos sugiere el término, serverless no significa «sin servidor», sino que éstos se usan como un elemento anónimo más de la infraestructura, apoyándose en las ventajas del cloud computing.
- La tecnología sin servidor apareció por primera vez en lo que se conoce como tecnologías de plataforma de aplicaciones como servicio (aPaaS), actualmente como FaaS (Functions-as-a-Service).

Ciclo de vida



Ciclo de vida

- Estrategia: que camino se va a seguir y como se planifica
- Creación: una vez se tenga una estrategia y un plan sólidos, es hora de crear las APIs.
- Pruebas: antes de publicar, es importante completar las pruebas de API para garantizar que cumplan con las expectativas de rendimiento, funcionalidad y seguridad.
- Publicación: una vez probado, es hora de publicar la API para que estén disponibles para los desarrolladores.
- Protección: los riesgos y las preocupaciones de seguridad son un problema común en la actualidad.
- Administración: una vez publicadas, los creadores deben administrar y mantener las APIs para asegurarse de que estén actualizadas y que la integridad de sus APIs no se vea comprometida.
- Integración: cuando se ofrece las APIs para consumo público o privado, la documentación es un componente importante para que los desarrolladores comprendan las capacidades clave.
- Monitorización: una vez las APIs están activas, es necesario supervisarlas y analizar los datos para detectar anomalías o detectar nuevas necesidades.
- Promoción: hay varias formas de comercializar las APIs, incluida su inclusión en un mercado de APIs.
- Monetización: se puede optar por ofrecer las APIs de forma gratuita o, cuando existe la oportunidad, se puede monetizar las APIs y generar ingresos adicionales para el negocio.
- Retirada: Retirar las APIs es la última etapa del ciclo de vida de una API y ocurre por una variedad de razones, incluidos cambios tecnológicos y preocupaciones de seguridad.

Ciclo de vida

HERRAMIENTAS

Instalaciones

- curl - https://curl.se/
- Postaman - https://www.postman.com/downloads/
- Extensiones Chrome
  - JSON Formatter
  - Yet Another REST Client
- Visual Studio Code - http://code.visualstudio.com/
  - Extensiones:
    - OpenAPI extension for Visual Studio Code
    - REST Client

Estilos de comunicación

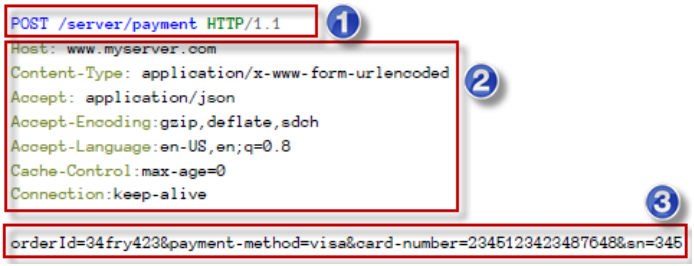
SERVICIOS REST

REST (REpresentational State Transfer)

- En 2000, Roy Fielding propuso la transferencia de estado representacional (REST) como enfoque de arquitectura para el diseño de servicios web. REST **es un estilo de arquitectura** para la creación de sistemas distribuidos basados en hipermedia. REST es independiente de cualquier protocolo subyacente y no está necesariamente unido a HTTP. Sin embargo, en las implementaciones más comunes de REST se usa HTTP como protocolo de aplicación, y esta guía se centra en el diseño de API de REST para HTTP.
- Originalmente se basaba en lo que ya estaba disponible en HTTP:
  - URL como identificadores de recursos
  - HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT + PATCH (HTTP1.1)
  - HTTP permite transmitir en el encabezado la información de comportamiento: Content-type, Accept, Authorization, Cache-control, …
  - HTTP utiliza códigos de estado en la respuesta para indicar como se ha completado una solicitud HTTP específica: respuestas informativas (1xx), respuestas satisfactorias (2xx), redirecciones (3xx), Errores en la petición (4xx) y errores de los servidores (5xx).

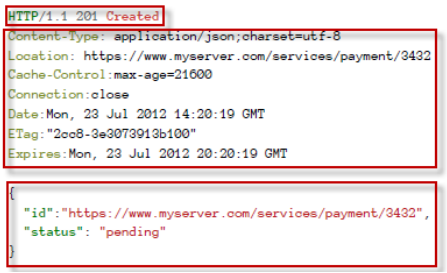
Petición HTTP

- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después el cuerpo del mensaje, que contiene un documento, que puede estar en cualquier formato (XML, HTML, JSON → Content-type)



Respuesta HTTP

- Los mensajes HTTP de respuesta siguen el mismo formato que los de envío.
- Sólo difieren en la primera línea
  - Donde se indica un código de respuesta junto a una explicación textual de dicha respuesta.
  - El código de respuesta indica si la petición tuvo éxito o no.



Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único, cualquier tipo de objeto, dato o servicio que sea direccionable a través de internet.
- Un recurso es un objeto que es lo suficientemente importante como para ser referenciado por sí mismo. Un recurso tiene datos, relaciones con otros recursos y métodos que operan contra él para permitir el acceso y la manipulación de la información asociada. Un grupo de recursos se llama colección. El contenido de las colecciones y los recursos depende de los requisitos de la organización y de los consumidores.
- En REST todos los recursos comparten una interfaz única y constante, la URI. (https://...)
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE
- Normalmente estos recursos son accesibles en una red o sistema.
- Las URI son el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Para que un URI sea correcto, debe cumplir los requisitos de formato, REST no indica de forma específica un formato obligatorio.
  - <esquema>://<host>:puerto/<ruta><querystring><fragmento>
- Los URI asociados a los recursos pueden cambiar si modificamos el recurso (nombre, ubicación, características, etc)

Tipos MIME

- Otro aspecto muy importante es la posibilidad de negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa). La representación de los recursos es el formato de lo que se envía un lado a otro entre clientes y servidores. Como REST utiliza HTTP, podemos transferir múltiples tipos de información.
- Los datos se transmiten a través de TCP/IP, el navegador sabe cómo interpretar las secuencias binarias (CONTENT-TYPE) por el protocolo HTTP.
- La representación de un recurso depende del tipo de llamada que se ha generado (Texto, HTML, PDF, etc).
- En HTTP cada uno de estos formatos dispone de su propio tipos MIME, en el formato <tipo>/<subtipo>.
  - application/json application/xml text/html text/plain image/jpeg
- Para negociar el formato:
  - El cliente, en la cabecera ACCEPT, envía una lista priorizada de tipos MIME que entiende.
  - Tanto cliente como servidor indican en la cabecera CONTENT-TYPE el formato MIME en que está codificado el body.
- Si el servidor no entiende ninguno de los tipos MIME propuestos (ACCEPT) devuelve un mensaje con código 406 (incapaz de aceptar petición).

Métodos HTTP

HTTP	REST	Descripción
GET	RETRIEVE	Sin identificador: Recuperar el estado completo de un recurso (HEAD + BODY) Con identificador: Recuperar el estado individual de un recurso (HEAD + BODY)
HEAD		Recuperar la cabecera del estado de un recurso (HEAD)
POST	CREATE or REPLACE	Crea o modifica un recurso (sin identificador)
PUT	CREATE or REPLACE	Crea o modifica un recurso (con identificador)
DELETE	DELETE	Sin identificador: Elimina todo el recurso Con identificador: Elimina un elemento concreto del recurso
CONNECT		Comprueba el acceso al host
TRACE		Solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición
OPTIONS		Devuelve los métodos HTTP que el servidor soporta para un URL específico
PATCH	REPLACE	HTTP 1.1 Reemplaza parcialmente un elemento del recurso

Códigos HTTP (status)

status	statusText	Descripción
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera Upgrade indica el nuevo protocolo
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código 200
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Descripción
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio
304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error en la petición

status	statusText	Descripción
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

Códigos de error en la petición

status	statusText	Descripción
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET
415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera Expect no se pueden cumplir

Códigos de error del servidor

status	statusText	Descripción
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

Problem Details for HTTP APIs (RFC 7807)

- Un requisito común para los servicios REST es incluir detalles en el cuerpo de las respuestas de error. La especificación RFC 7807 para "Detalles del problema para las API HTTP" es la mas ampliamente aceptada.
- La respuesta con los detalles del problema puede contener los siguientes miembros:
  - "type" (cadena): URI que identifica el tipo de problema y proporciona documentación legible por humanos para el tipo de problema. El valor "about:blank" (predeterminado) indica que el problema no tiene semántica adicional a la del código de estado HTTP.
  - "title" (cadena): Breve resumen legible por humanos del problema escribe. NO DEBE cambiar de una ocurrencia a otra del mismo problema, excepto para fines de localización. Con "type": "about:blank", DEBE coincidir con la versión textual del status.
  - "detail" (cadena): Explicación legible por humanos específica de la ocurrencia concreta del problema.
  - "status" (número): Código de estado HTTP (por conveniencia, opcional, debe coincidir).
  - "instance" (cadena): URI de referencia que identifica el origen de la ocurrencia del problema.
- Las definiciones de tipo de problema PUEDEN extender la respuesta con miembros adicionales.

Estilo de arquitectura

- Las APIs de REST se diseñan en torno a recursos, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente.
- Un recurso tiene un identificador, que es un URI que identifica de forma única ese recurso.
- Los clientes interactúan con un servicio mediante el intercambio de representaciones de recursos.
- Las APIs de REST usan una interfaz uniforme, que ayuda a desacoplar las implementaciones de clientes y servicios. En las APIs REST basadas en HTTP, la interfaz uniforme incluye el uso de verbos HTTP estándar para realizar operaciones en los recursos. Las operaciones más comunes son GET, POST, PUT, PATCH y DELETE. El código de estado de la respuesta indica el éxito o error de la petición.
- Las APIs de REST usan un modelo de solicitud sin estado. Las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden, por lo que no es factible conservar la información de estado transitoria entre solicitudes. El único lugar donde se almacena la información es en los propios recursos y cada solicitud debe ser una operación atómica.
- Las APIs de REST se controlan mediante vínculos de hipermedia.

Estilo de arquitectura

- Métodos GET
  - Normalmente, un método GET correcto devuelve el código de estado HTTP 200 (Correcto). Si no se encuentra el recurso, el método debe devolver HTTP 404 (No encontrado).
- Métodos POST
  - Si un método POST crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado). El URI del nuevo recurso se incluye en el encabezado Location de la respuesta. El cuerpo de respuesta contiene una representación del recurso.
  - Si el método realiza algún procesamiento pero no crea un nuevo recurso, puede devolver el código de estado HTTP 200 e incluir el resultado de la operación en el cuerpo de respuesta. O bien, si no hay ningún resultado para devolver, el método puede devolver el código de estado HTTP 204 (Sin contenido) sin cuerpo de respuesta.
  - Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver el código de estado HTTP 400 (Solicitud incorrecta). El cuerpo de respuesta puede contener información adicional sobre el error o un vínculo a un URI que proporciona más detalles.
- Métodos DELETE
  - El servidor web debe responder con un 204 (Sin contenido), que indica que la operación de eliminación es correcta, pero que el cuerpo de respuesta no contiene información adicional. Si el recurso no existe, el servidor web puede devolver un 404 (No encontrado).

Estilo de arquitectura

- Métodos PUT
  - Si un método PUT crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), al igual que con un método POST. Si el método actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o el recurso no existe, puede devolver un 404 (No encontrado).
  - Hay que considerar la posibilidad de implementar operaciones HTTP PUT masivas que pueden procesar por lotes las actualizaciones de varios recursos de una colección. La solicitud PUT debe especificar el URI de la colección y el cuerpo de solicitud debe especificar los detalles de los recursos que se van a modificar. Este enfoque puede ayudar a reducir el intercambio de mensajes y mejorar el rendimiento.
- Métodos PATCH
  - Con una solicitud PATCH, el cliente envía un conjunto de actualizaciones a un recurso existente, en forma de un documento de revisión. El servidor procesa el documento de revisión para realizar la actualización.
  - Si el método actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud o el documento de revisión tiene un formato incorrecto, el servidor debe devolver un 400 (Solicitud incorrecta), si no se admite el formato de documento de revisión devolverá un 415 (Tipo de medio no compatible), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o el recurso no existe, puede devolver un 404 (No encontrado).

Encabezado HTTP Cache-Control

- El encabezado HTTP Cache-Control especifica directivas (instrucciones) para almacenar temporalmente (caching) tanto en peticiones como en respuestas. Una directiva dada en una petición no significa que la misma directiva estar en la respuesta.
- Los valores estándar que pueden ser usados por el servidor en una respuesta HTTP son:
  - public: La respuesta puede estar almacenada en cualquier memoria cache.
  - private: La respuesta puede estar almacenada sólo por el cache de un navegador.
  - no-cache: La respuesta puede estar almacenada en cualquier memoria cache pero DEBE pasar siempre por una validación con el servidor de origen antes de utilizarse.
  - no-store: La respuesta puede no ser almacenada en cualquier cache.
  - max-age=<seconds>: La cantidad máxima de tiempo un recurso es considerado reciente.
  - s-maxage=<seconds>: Anula el encabezado max-age o el Expires, pero solo para caches compartidos (e.g., proxies).
  - must-revalidate: Indica que una vez un recurso se vuelve obsoleto, el cache no debe usar su copia obsoleta sin validar correctamente en el servidor de origen.
  - proxy-revalidate: Similar a must-revalidate, pero solo para caches compartidos (es decir, proxies). Ignorado por caches privados.
  - no-transform: No deberían hacerse transformaciones o conversiones al recurso.

Encabezados HTTP ETag, If-Match y If-None-Match

- El encabezado de respuesta de HTTP ETag es un identificador (resumen hash) para una versión específica de un recurso y los encabezados If-Match e If-None-Match de la solicitud HTTP hace que la solicitud sea condicional.
- Para los métodos GET y HEAD con If-None-Match: si el ETag no coincide con los dados, el servidor devolverá el recurso solicitado con un estado 200, si coincide el servidor debe devolver el código de estado HTTP 304 (No modificado) y DEBE generar cualquiera de los siguientes campos de encabezado que se habrían enviado en una respuesta 200 (OK) a la misma solicitud: Cache-Control, Content-Location, Date, ETag, Expires y Vary.
- Para los métodos PUT y DELETE con If-Match: si el ETag coincide con los dados, se realiza la actualización o borrado y se devuelve un estado HTTP 204 (sin contenido) incluyendo el Cache-Control y el ETag de la versión actualizada del recurso en el PUT. Si no coinciden, se ha producido un error de concurrencia, la versión del servidor ha sido modificada desde que la recibió el cliente, debe devolver una respuesta HTTP con un cuerpo de mensaje vacío y un código de estado 412 (Precondición fallida).
- Si los datos solicitados ya no existen, el servido debe devolver una respuesta HTTP con el código de estado 404 (no encontrado).

Estilo de arquitectura

- Request: Método /uri?parámetros
  - GET: Recupera el recurso (200)
    - Todos: Sin identificador
    - Uno: Con identificador
  - POST: Crea o reemplaza un nuevo recurso (201)
  - PUT: Crea o reemplaza el recurso identificado (200, 204)
  - DELETE: Elimina el recurso (204)
    - Todos: Sin identificador
    - Uno: Con identificador
- Cabeceras:
  - Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
  - Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

Peticiones

- Request: GET /users
  - accept:application/json
- Response: 200
  - content-type:application/json
  - BODY

- Request: GET /users/11
  - accept:application/json
- Response: 200
  - content-type:application/json
  - BODY

- Request: POST /users
  - accept:application/json
  - content-type:application/json
  - BODY
- Response: 201
  - content-type:application/json
  - BODY

- Request: PUT /users/11
  - accept:application/json
  - content-type:application/json
  - BODY
- Response: 200
  - content-type:application/json
  - BODY

- Request: DELETE /users/11
- Response: 204 no content

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario definir una serie de cosas:
  - Analizar el/los recurso/s a implementar
  - Diseñar la REPRESENTACION del recurso.
    - Deberemos definir el formato de trabajo del recurso: XML, JSON, HTML, imagen, RSS, etc
  - Definir el URI de acceso.
    - Deberemos indicar el/los URI de acceso para el recurso
  - Establecer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - Fijar qué códigos de estado pueden ser devueltos
    - Los códigos de estado HTTP típicos que podrían ser devueltos
- Todo lo anterior dependerá del servicio a implementar.

Definir operaciones

- Sumario y descripción de la operación.
- Dirección: URL
  - Sin identificador
  - Con identificador
  - Con parámetros de consulta
- Método: GET | POST | PUT | DELETE | PATCH
- Solicitud:
  - Cabeceras:
    - ACCEPT: formatos aceptables si espera recibir datos
    - CONTENT-TYPE: formato de envío de los datos en la solicitud
    - Otras cabeceras: Authorization, Cache-control, X-XSRF-TOKEN, …
  - Cuerpo: en caso de envío, estructura de datos formateados según el CONTENT-TYPE.
- Respuesta:
  - Cabeceras:
    - Códigos de estado HTTP: posibles y sus causas.
    - CONTENT-TYPE: formato de envío de los datos en la respuesta
    - Otras cabeceras
  - Cuerpo: en caso de respuesta, estructura de datos según código de estado y formateados según el CONTENT-TYPE.

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- Nivel 0: Definir un URI y todas las operaciones son solicitudes POST a este URI.
- Nivel 1 (Pobre): Crear distintos URI para recursos individuales pero utilizan solo un método.
  - Se debe identificar un recurso
    /entities/?invoices =2 → entities/invoices/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- Nivel 2 (Medio): Usar métodos HTTP para definir operaciones en los recursos.
- Nivel 3 (Óptimo): Usar hipermedia (HATEOAS, se describe a continuación).

Hypermedia

- Uno de los principales propósitos que se esconden detrás de REST es que debe ser posible navegar por todo el conjunto de recursos sin necesidad de conocer el esquema de URI. Cada solicitud HTTP GET debe devolver la información necesaria para encontrar los recursos relacionados directamente con el objeto solicitado mediante los hipervínculos que se incluyen en la respuesta, y también se le debe proporcionar información que describa las operaciones disponibles en cada uno de estos recursos.
- Este principio se conoce como HATEOAS, del inglés Hypertext as the Engine of Application State (Hipertexto como motor del estado de la aplicación). El sistema es realmente una máquina de estado finito, y la respuesta a cada solicitud contiene la información necesaria para pasar de un estado a otro; ninguna otra información debería ser necesaria.
- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- HATEOAS: Content-Type: application/hal+json

```
{
   "_links": {
     "self": {"href": "/orders/523" },
     "warehouse": {"href": "/warehouse/56" },
     "invoice": {"href": "/invoices/873"}
   },
   , "currency": "USD"
   , "status": "shipped"
   , "total": 10.20
}
```

Características de una API bien diseñada

- **Fácil de leer y trabajar**: con una API bien diseñada será fácil trabajar, y sus recursos y operaciones asociadas pueden ser memorizados rápidamente por los desarrolladores que trabajan con ella constantemente.
- **Difícil de usar mal**: la implementación e integración con una API con un buen diseño será un proceso sencillo, y escribir código incorrecto será un resultado menos probable porque tiene comentarios informativos y no aplica pautas estrictas al consumidor final de la API.
- **Completa y concisa**: Finalmente, una API completa hará posible que los desarrolladores creen aplicaciones completas con los datos que expone. Por lo general, la completitud ocurre con el tiempo, y la mayoría de los diseñadores y desarrolladores de API construyen gradualmente sobre las APIs existentes. Es un ideal por el que todo ingeniero o empresa con una API debe esforzarse.

Guía de diseño

- Organización de la API en torno a los recursos
- Definición de operaciones en términos de métodos HTTP
- Conformidad con la semántica HTTP
- Filtrado y paginación de los datos
- Compatibilidad con respuestas parciales en recursos binarios de gran tamaño
- Uso de HATEOAS para permitir la navegación a los recursos relacionados
- Control de versiones en la API RESTful
- Documentación Open API

Definición de recursos

- La organización de la API en torno a los recursos se centran en las entidades de dominio que debe exponer la API. Por ejemplo, en un sistema de comercio electrónico, las entidades principales podrían ser clientes y pedidos. La creación de un pedido se puede lograr mediante el envío de una solicitud HTTP POST que contiene la información del pedido. La respuesta HTTP indica si el pedido se realizó correctamente o no.
- Un recurso no tiene que estar basado en un solo elemento de datos físico o tablas de una base de datos relacional. La finalidad de REST es modelar entidades y las operaciones que un consumidor externo puede realizar sobre esas entidades, no debe exponerse a la implementación interna.
- Es necesario adoptar una convención de nomenclatura coherente para los URI. Los URI de recursos deben basarse en nombres (de recurso), nunca en verbos (las operaciones en el recurso) y, en general, resulta útil usar nombres plurales que hagan referencia a colecciones. Debe seguir una estructura jerárquica que refleje las relaciones entre los diferentes tipos de recursos.
- Hay que considerar el uso del enfoque HATEOAS para permitir el descubrimiento y la navegación a los recursos relacionados o el enfoque del patrón agregado.

Definición de recursos

- Exponer una colección de recursos con un único URI puede dar lugar a que las aplicaciones capturen grandes cantidades de datos cuando solo se requiere un subconjunto de la información. A través de la definición de parámetros de cadena de consulta se pueden realizar particiones horizontales con filtrado, ordenación y paginación, o particiones verticales con la proyección de las propiedades a recuperar:
  - https://host/users?page=1&rows=20&projection=userId,name,lastAccess
- La definición de operaciones con el recurso se realiza en términos de métodos HTTP, estableciendo cuales serán soportadas. Las operaciones no soportadas por métodos HTTP deben sustantivarse al crearles una URI especifica y utilizar el método HTTP semánticamente mas próximo.
  - DELETE https://host/users/bloqueo (desbloquear)
  - POST https://host/pedido/171/factura (facturar)
- Hay que establecer el tipo o tipos de formatos mas adecuados para las representaciones de recursos. Los formatos se especifican mediante el uso de tipos de medios, también denominados tipos MIME. En el caso de datos no binarios, la mayoría de las APIs web admiten JSON (application/json) y, posiblemente, XML (application/xml).

Políticas de versionado

- Es muy poco probable que una API permanezca estática. Conforme los requisitos empresariales cambian, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos de los recursos debe adecuarse.
- Los cambios rupturistas no son compatibles con la versión anterior, el consumidor tendrá que adaptar su código para pasar su aplicación existente a la nueva versión y evitar que se rompa.
- Hay dos razones principales por las que las APIs HTTP se comportan de manera diferente al resto de las APIs:
  - El código del cliente dicta lo que lo romperá: Un proveedor de API no tiene control sobre las herramientas que un consumidor puede usar para interpretar una respuesta de la API y la tolerancia al cambio que tienen esas herramientas varían ampliamente, si es rupturista o no.
  - El proveedor de API elige si los cambios son opcionales o transparentes: Los proveedores de API pueden actualizar su API y los cambios en las respuestas afectarán inmediatamente a los clientes. Los clientes no pueden decidir si adoptar o no la nueva versión, lo que puede generar fallos en cascada en los cambios rupturistas.

Políticas de versionado

- El control de versiones permite que una API indique la versión expuesta y que una aplicación cliente pueda enviar solicitudes que se dirijan a una versión específica con una característica o un recurso.
  - Sin control de versiones: Este es el enfoque más sencillo y puede ser aceptable para algunas APIs internas. Los grandes cambios podrían representarse como nuevos recursos o nuevos vínculos.
  - Control de versiones en URI: Cada vez que modifica la API web o cambia el esquema de recursos, agrega un número de versión al URI para cada recurso. Los URI ya existentes deben seguir funcionando como antes y devolver los recursos conforme a su esquema original.

    http://host/v2/users
  - Control de versiones en cadena de consulta: En lugar de proporcionar varios URI, se puede especificar la versión del recurso mediante un parámetro dentro de la cadena de consulta anexada a la solicitud HTTP:

    http://host/users?versión=2.0

Políticas de versionado

- Control de versiones en encabezado: En lugar de anexar el número de versión como un parámetro de cadena de consulta, se podría implementar un encabezado personalizado que indica la versión del recurso. Este enfoque requiere que la aplicación cliente agregue el encabezado adecuado a las solicitudes, aunque el código que controla la solicitud de cliente puede usar un valor predeterminado (versión actual) si se omite el encabezado de versión.

  ```
  GET https://host/users HTTP/1.1
  Custom-Header: api-version=1
  ```

- Control de versiones por MIME (tipo de medio): Cuando una aplicación cliente envía una solicitud HTTP GET a un servidor web, debe prever el formato del contenido que puede controlar mediante el uso de un encabezado Accept.

  ```
  GET https:// host/users/3 HTTP/1.1
  Accept: application/vnd.mi-api.v1+json
  ```

- Si la versión no está soportada, el servicio podría generar un mensaje de respuesta HTTP 406 (no aceptable) o devolver un mensaje con un tipo de medio predeterminado.
- Los esquemas de control de versiones de URI y de cadena de consulta son compatibles con la caché HTTP puesto que la misma combinación de URI y cadena de consulta hace referencia siempre a los mismos datos.

Políticas de versionado

- Dentro de la política de versionado es conveniente planificar la obsolescencia y la política de desaprobación.
- La obsolescencia programada establece el periodo máximo, como una franja temporal o un número de versiones, en que se va a dar soporte a cada versión, evitando los sobrecostes derivados de mantener versiones obsoletas indefinidamente.
- Dentro de la política de desaprobación, para ayudar a garantizar que los consumidores tengan tiempo suficiente y una ruta clara de actualización, se debe establecer el número de versiones en que se mantendrá una característica marcada como obsoleta antes de su desaparición definitiva.
- La obsolescencia programada y la política de desaprobación beneficia a los consumidores de la API porque proporcionan estabilidad y sabrán qué esperar a medida que las APIs evolucionen.
- Para mejorar la calidad y avanzar las novedades, se podrán realizar lanzamientos de versiones Beta y Release Candidatos (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

Guía de implementación

- Procesamiento de solicitudes
  - Las acciones GET, PUT, DELETE, HEAD y PATCH deben ser idempotentes.
  - Las acciones POST que crean nuevos recursos no deben tener efectos secundarios no relacionados.
  - Evitar implementar operaciones POST, PUT y DELETE que generen mucha conversación.
  - Seguir la especificación HTTP al enviar una respuesta.
  - Admitir la negociación de contenido.
  - Proporcionar vínculos que permitan la navegación y la detección de recursos de estilo HATEOAS.

Guía de implementación

- Administración de respuestas y solicitudes de gran tamaño
  - Optimizar las solicitudes y respuestas que impliquen objetos grandes.
  - Admitir la paginación de las solicitudes que pueden devolver grandes cantidades de objetos.
  - Implementar respuestas parciales para los clientes que no admitan operaciones asincrónicas.
  - Evitar enviar mensajes de estado 100-Continuar innecesarios en las aplicaciones cliente.
- Mantenimiento de la capacidad de respuesta, la escalabilidad y la disponibilidad
  - Ofrecer compatibilidad asincrónica para las solicitudes de ejecución prolongada.
  - Comprobar que ninguna de las solicitudes tenga estado.
  - Realizar un seguimiento de los clientes e implementar limitaciones para reducir las posibilidades de ataques de denegación de servicio.
  - Administrar con cuidado las conexiones HTTP persistentes.

Guía de implementación

- Control de excepciones
  - Capturar todas las excepciones y devolver una respuesta significativa a los clientes.
  - Distinguir entre los errores del lado cliente y del lado servidor.
  - Evitar las vulnerabilidades por exceso de información.
  - Controlar las excepciones de una forma coherente y registrar la información sobre los errores.
- Optimización del acceso a los datos en el lado cliente
  - Admitir el almacenamiento en caché del lado cliente.
  - Proporcionar ETags para optimizar el procesamiento de las consultas.
  - Usar ETags para admitir la simultaneidad optimista.

Guía de implementación

- Publicación y administración de una API web
  - Todas las solicitudes deben autenticarse y autorizarse, y debe aplicarse el nivel de control de acceso adecuado.
  - Una API web comercial puede estar sujeta a diversas garantías de calidad relativas a los tiempos de respuesta. Es importante asegurarse de que ese entorno de host es escalable si la carga puede variar considerablemente con el tiempo.
  - Puede ser necesario realizar mediciones de las solicitudes para fines de monetización.
  - Es posible que sea necesario regular el flujo de tráfico a la API web e implementar la limitación para clientes concretos que hayan agotado sus cuotas.
  - Los requisitos normativos podrían requerir un registro y una auditoría de todas las solicitudes y respuestas.
  - Para garantizar la disponibilidad, puede ser necesario supervisar el estado del servidor que hospeda la API web y reiniciarlo si hiciera falta.

Guía de implementación

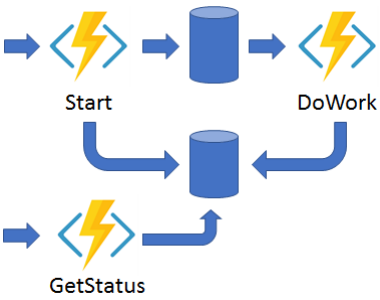
- Pruebas de la API
  - Ejercitar todas las rutas y parámetros para comprobar que invocan las operaciones correctas.
  - Verificar que cada operación devuelve los códigos de estado HTTP correctos para diferentes combinaciones de entradas.
  - Comprobar que todas las rutas estén protegidas correctamente y que estén sujetas a las comprobaciones de autenticación y autorización apropiadas.
  - Verificar el control de excepciones que realiza cada operación y que se devuelve una respuesta HTTP adecuada y significativa de vuelta a la aplicación cliente.
  - Comprobar que los mensajes de solicitud y respuesta están formados correctamente.
  - Comprobar que todos los vínculos dentro de los mensajes de respuesta no están rotos.

WebHooks

- Los webhooks son eventos HTTP que desencadenan acciones. Su nombre se debe a que funcionan como «enganches» de los programas en Internet y casi siempre se utilizan para la comunicación entre sistemas. Son la manera más sencilla de obtener un aviso cuando algo ocurre en otro sistema y para el intercambio de datos entre aplicaciones web, permitiendo las llamada asíncronas en HTTP.
- Un webhook es una retro llamada HTTP, una solicitud HTTP GET/POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP GET/POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento como parámetros o en cuerpo en JSON o XML) a una determinada dirección http:// o https://, que puede:
  - almacenar los datos del evento en JSON o XML
  - generar una respuesta que permita actualizarse al sistema donde se produce el evento
  - ejecutar un proceso en el sistema receptor del evento (Ej: enviar un correo electrónico)
- Los webhook están pensados para su utilización desde páginas web y sus diferentes consumidores: navegadores, correo electrónico, webapps, …
- Un ejemplo típico es su utilización en correos electrónico de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario a abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo GET/POST).

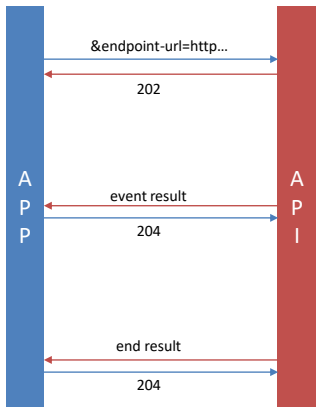
WebHooks

- El patrón Async HTTP APIs soluciona el problema de coordinar el estado de las operaciones de larga duración con los clientes externos. Una forma habitual de implementar este patrón es que un punto de conexión HTTP desencadene la acción de larga duración. A continuación, el cliente se redirige a un punto de conexión de estado al que sondea para saber el estado actual del proceso y cuando finaliza la operación.
- Un endpoint HTTP desencadena el proceso con la acción de larga duración y devuelve inmediatamente un 202 si la petición es correcta y, opcionalmente, como carga útil las URLs de los endpoints de estado y control. Las peticiones incorrectas recibirán el 4XX apropiado.
- El cliente sondea periódicamente el endpoint de estado y obtiene:
  - 202 con el estado actual del proceso como carga útil mientras el proceso este en marcha.
  - 200 con el estado final (correcto o fallido) del proceso como carga útil cuando haya finalizado la operación.
- El endpoint HTTP desencadenador puede suministrar endpoints adicionales para pausar, reanudar, cancelar/terminar, reiniciar o enviar eventos al proceso en marcha.



WebHooks

- El patrón Reverse API soluciona el problema del sondeo periódico, las API inversas invierten esta situación, para que sea la API invocada la que notifique automáticamente cuando se ha producido un determinado evento. El cliente debe crear su propia API para recibir las notificaciones.
- Al realizar la petición al endpoint HTTP desencadenador, se suministra un endpoint propio para que el proceso desencadenado pueda notificar cuándo ocurre algo de interés. Le da la vuelta a la comunicación, el cliente pasa a ser servidor y el servidor a cliente. El desencadenador devuelve inmediatamente un 202 si la petición es correcta o el 4XX apropiado se es incorrecta.
- Este esquema de funcionamiento tiene muchas ventajas en ambos extremos:
  - **Ahorro de recursos y tiempo**: con el sondeo se harán muchas llamadas "para nada", que no devolverán información relevante. Los dos extremos gastan recursos para hacer y responder a muchas llamadas que no tienen utilidad alguna (no nos llame, ya les llamaremos).
  - **Eliminación de los retrasos**: la aplicación usuaria recibirá una llamada en el momento exacto en el que se produce y no tendrá retrasos al próximo sondeo.
  - **Velocidad de las llamadas**: generalmente la llamada que se hace a un webhook es muy rápida porque solo se envía una pequeña información sobre el evento y se suele procesar asíncronamente. Muchas veces ni siquiera se espera por el resultado: se hace una llamada del tipo "fire and forget" (o sea, dispara y olvídate), pues se trata de notificar el evento y listo.



SEGURIDAD

Autorización de solicitudes

- Las API utilizan la autorización para garantizar que las solicitudes de los clientes accedan a los datos de forma segura. Esto puede implicar la autenticación del remitente de una solicitud y la confirmación de que tiene permiso para acceder o manipular los datos relevantes.
- Si se está creando una API, se puede elegir entre una variedad de modelos de autenticación.
  - API Key en encabezados, cadena de consulta o cookies
  - Esquemas de autenticación HTTP (encabezado Authorization):
    - Basic, Digest, Bearer y otros esquemas HTTP según lo definido por RFC 7235 y HTTP Authentication Scheme Registry
  - OAuth 2
  - Descubrimiento de OpenID Connect
- Si se está integrando una API de terceros, el proveedor de la API especificará la autorización requerida.

Patrones de diseño

- Patrón: Valet Key
  - Usa un token (api key) que proporciona a los clientes acceso directo restringido a un recurso específico, con el fin de descargar la transferencia de datos desde la aplicación. Esto es especialmente útil en aplicaciones que usan sistemas o colas de almacenamiento hospedado en la nube, ya que puede minimizar los costes y maximizar la escalabilidad y el rendimiento.
- Patrón: Token de acceso
  - La aplicación consta de numerosos servicios. La puerta de enlace API es el único punto de entrada para las solicitudes de los clientes. Autentica las solicitudes y las reenvía a otros servicios, que a su vez podrían invocar otros servicios. ¿Cómo comunicar la identidad del solicitante a los servicios que tramitan la solicitud? El API Gateway autentica la solicitud y pasa un token de acceso (por ejemplo, JSON Web Token) que identifica de forma segura al solicitante en cada solicitud a los servicios. Un servicio puede incluir el token de acceso en las solicitudes que realiza a otros servicios.
- Patrón: Federated Identity
  - La autenticación se delega a un proveedor de identidad externo. Esto puede simplificar el desarrollo, minimizar los requisitos de administración de usuarios y mejorar la experiencia del usuario de la aplicación.

API Key

- Algunas API utilizan claves de API para la autorización. Una clave API es un token que suministra el proveedor y que el cliente debe proporcionar como parámetro, encabezado o cookie cuando realiza las llamadas a la API para tener acceso a las mismas. El nombre del parámetro, encabezado o cookie lo define el proveedor.
- La clave se puede enviar en la cadena de consulta:

  GET /something?api_key=abcdef12345
- o como encabezado de solicitud:

  GET /something HTTP/1.1

  X-API-Key: abcdef12345
- o como cookie:

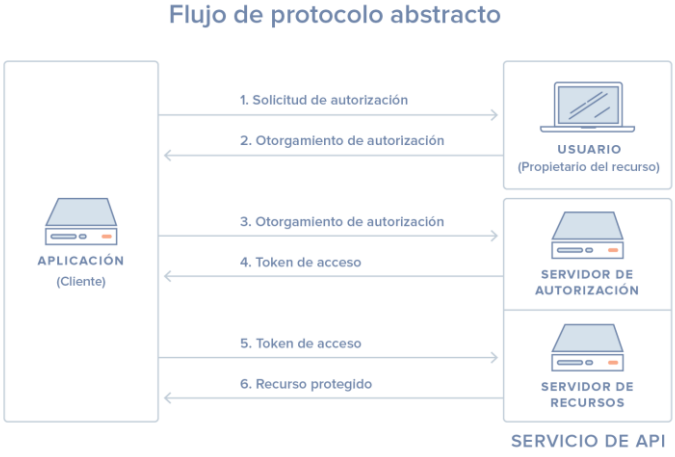
  GET /something HTTP/1.1

  Cookie: X-API-KEY=abcdef12345
- Se supone que las claves API son un secreto que solo el cliente y el servidor conocen. Al igual que en la autenticación básica, la autenticación basada en claves API solo se considera segura si se usa junto con otros mecanismos de seguridad, como HTTPS/SSL.

OAuth 2

- OAuth 2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a los recursos de usuario en un servicio HTTP, como Facebook, GitHub y Google. Delega la autenticación del usuario al servicio que aloja la cuenta del mismo y autoriza a las aplicaciones de terceros el acceso a dicha cuenta de usuario.
- OAuth define cuatro roles:
  - Propietario del recurso: Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se le conoce como usuario final.
  - Servidor de recursos: El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso.
  - Cliente: Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular.
  - Servidor de autorizaciones: El servidor que emite tokens de acceso al cliente después de haber realizado correctamente autenticar al propietario del recurso y obtener autorización.

OAuth 2



Autenticación de portador (Bearer)

- La autenticación de portador (también llamada autenticación de token) es un esquema de autenticación HTTP que involucra tokens de seguridad llamados tokens de portador. El nombre "Autenticación de portador" puede entenderse como "dar acceso al portador de este token". El token portador es una cadena encriptada, generalmente generada por el servidor en respuesta a una solicitud de inicio de sesión. El cliente debe enviar este token en el encabezado Authorization al realizar solicitudes a recursos protegidos:

  Authorization: Bearer <token>
- El esquema de autenticación Bearer se creó originalmente como parte de OAuth 2.0 en RFC 6750, pero a veces también se usa solo. De manera similar a la autenticación básica, la autenticación de portador solo debe usarse a través de HTTPS (SSL).

JWT: JSON Web Tokens

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier datos entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo si tenemos un servicio de envío de email otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

Tokens

- Los tokens son una serie de caracteres cifrados y firmados con una clave compartida entre servidor OAuth y el servidor de recurso o para mayor seguridad mediante clave privada en el servidor OAuth y su clave pública asociada en el servidor de recursos, con la firma el servidor de recursos el capaz de comprobar la autenticidad del token sin necesidad de comunicarse con él.
- Se componen de tres partes separadas por un punto, una cabecera con el algoritmo hash utilizado y tipo de token, un documento JSON con datos y una firma de verificación.
- El hecho de que los tokens JWT no sea necesario persistirlos en base de datos elimina la necesidad de tener su infraestructura, como desventaja es que no es tan fácil de revocar el acceso a un token JWT y por ello se les concede un tiempo de expiración corto.
- La infraestructura requiere varios elementos configurables de diferentes formas son:
  - El servidor OAuth que realiza la autenticación y proporciona los tokens.
  - El servicio al que se le envía el token, es el que decodifica el token y decide conceder o no acceso al recurso.
  - En el caso de múltiples servicios con múltiples recursos es conveniente un gateway para que sea el punto de entrada de todos los servicios, de esta forma se puede centralizar las autorizaciones liberando a los servicios individuales.

DOCUMENTACIÓN

Enfoque API First

- El enfoque basado en API First significa que, para cualquier proyecto de desarrollo dado, las APIs se tratan como "ciudadanos de primera clase": que todo sobre un proyecto gira en torno a la idea de que el producto final es un conjunto de APIs consumido por las aplicaciones del cliente.
- El enfoque de API First implica que los desarrollos de APIs sean consistentes y reutilizables, lo que se puede lograr mediante el uso de un lenguaje formal de descripción de APIs para establecer un contrato sobre cómo se supone que se comportará la API. Establecer un contrato implica pasar más tiempo pensando en el diseño de una API.
- A menudo también implica una planificación y colaboración adicionales con las partes interesadas, proporcionando retroalimentación de los consumidores sobre el diseño de una API antes de escribir cualquier código evitando costosos errores.

Beneficios de API First

- Los equipos de desarrollo pueden trabajar en paralelo.
  - Los equipos pueden simular APIs y probar sus dependencias en función de la definición de la API establecida.
- Reduce el coste de desarrollar aplicaciones
  - Las APIs y el código se pueden reutilizar en muchos proyectos diferentes.
- Aumenta la velocidad de desarrollo.
  - Gran parte del proceso de creación de API se puede automatizar mediante herramientas que permiten importar archivos de definición de API y generar el esqueleto del backend y el cliente frontend, así como un mocking server para las pruebas.

Beneficios de API First

- Asegura buenas experiencias de desarrollador
  - Las APIs bien diseñadas, bien documentadas y consistentes brindan experiencias positivas para los desarrolladores porque es más fácil reutilizar el código y los desarrollos integrados, reduciendo la curva de aprendizaje.
- Reduce el riesgo de fallos
  - Reduce el riesgo de fallos al facilitar las pruebas para garantizar que las APIs sean confiables, consistentes y fáciles de usar para los desarrolladores.

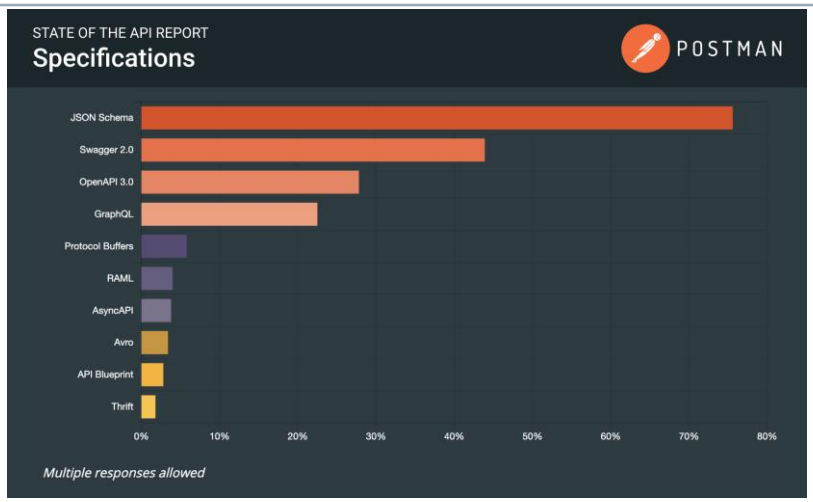
Documentar servicios Rest

- Dado que las API están diseñadas para ser consumidas, es importante asegurarse de que el cliente o consumidor pueda implementar rápidamente una API y comprender qué está sucediendo con ella. Desafortunadamente, muchas API hacen que la implementación sea extremadamente difícil, frustrando su propósito.
- La documentación es uno de los factores más importantes para determinar el éxito de una API, ya que la documentación sólida y fácil de entender hace que la implementación de la API sea muy sencilla, mientras que la documentación confusa, desincronizada, incompleta o intrincada hace que sea una aventura desagradable, una que generalmente conduce a desarrolladores frustrados a utilizar las soluciones de la competencia.
- Una buena documentación debe actuar como referencia y como formación, permitiendo a los desarrolladores obtener rápidamente la información que buscan de un vistazo, mientras también leen la documentación para obtener una comprensión de cómo integrar el recurso / método que están viendo.
- Con la expansión de especificaciones abiertas como OpenApi, RAML, … y las comunidades que las rodean, la documentación se ha vuelto mucho más fácil, aun así requiere invertir tiempo y recursos, todo ello con una cuidadosa planificación.

Documentar servicios Rest

- Web Application Description Language (WADL) (https://www.w3.org/Submission/wadl/)
  - Especificación de W3C, que la descripción XML legible por máquina de aplicaciones web basadas en HTTP (normalmente servicios web REST). Modela los recursos proporcionados por un servicio y las relaciones entre ellos. Está diseñado para simplificar la reutilización de servicios web basados en la arquitectura HTTP existente de la web. Es independiente de la plataforma y del lenguaje, tiene como objetivo promover la reutilización de aplicaciones más allá del uso básico en un navegador web.
- Spring REST Docs (https://spring.io/projects/spring-restdocs)
  - Documentación a través de los test (casos de uso), evita enterrar el código entre anotaciones.
- RAML (https://raml.org/)
  - RESTful API Modeling Language es una forma práctica de describir un API RESTful de una manera que sea muy legible tanto para humanos como para maquinas.
- Open API (anteriormente Swagger)
  - Especificación para describir, producir, consumir y visualizar servicios web RESTful. Es el más ampliamente difundido y cuenta con un ecosistema propio.
- JSON Schema (https://json-schema.org/)
  - JSON Schema es una especificación para definir, anotar y validar las estructuras de datos JSON.

Especificaciones mas utilizadas



Swagger

https://swagger.io/

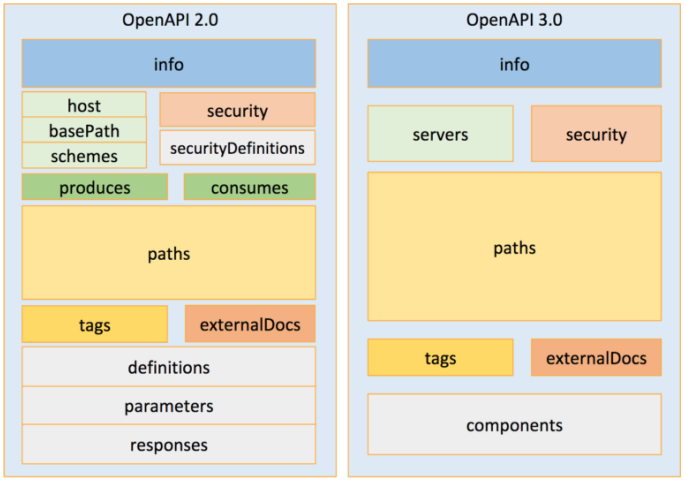
- Swagger (OpenAPI Specification) es una especificación abierta y su correspondiente implementación para probar y documentar servicios REST. Uno de los objetivos de Swagger es que podamos actualizar la documentación en el mismo instante en que realizamos los cambios en el servidor.
- Un documento Swagger es el equivalente de API REST de un documento WSDL para un servicio web basado en SOAP.
- El documento Swagger especifica la lista de recursos disponibles en la API REST y las operaciones a las que se puede llamar en estos recursos.
- El documento Swagger especifica también la lista de parámetros de una operación, que incluye el nombre y tipo de los parámetros, si los parámetros son necesarios u opcionales, e información sobre los valores aceptables para estos parámetros.

OpenAPI

https://www.openapis.org/

- OpenAPI es un estándar para definir contratos de API. Los cuales describen la interfaz de una serie de servicios que vamos a poder consumir por medio de una signatura. Conocido previamente como Swagger, ha sido adoptado por la Linux Foundation y obtuvo el apoyo de compañías como Google, Microsoft, IBM, Paypal, etc. para convertirse en un estándar para las APIs REST.
- Las definiciones de OpenAPI se pueden escribir en JSON o YAML. La versión actual de la especificación es la 3.0.3 y orientada a YAML y la versión previa la 2.0, que es idéntica a la especificación 2.0 de Swagger antes de ser renombrada a “Open API Specification”.
- Actualmente nos encontramos en periodo de transición de la versión 2 a la 3, sin soporte en muchas herramientas.

Cambio de versión



Sintaxis

- Un documento de OpenAPI que se ajusta a la especificación de OpenAPI es en sí mismo un objeto JSON con propiedades, que puede representarse en formato JSON o YAML.
- YAML es un lenguaje de serialización de datos similar a XML pero que utiliza el sangrado para indicar el anidamiento, estableciendo la estructura jerárquica, y evitar la necesidad de tener que cerra los elementos.
- Para preservar la capacidad de ida y vuelta entre los formatos YAML y JSON, se RECOMIENDA la versión 1.2 de YAML junto con algunas restricciones adicionales:
  - Las etiquetas DEBEN limitarse a las permitidas por el conjunto de reglas del esquema JSON .
  - Las claves utilizadas en los mapas YAML DEBEN estar limitadas a una cadena escalar, según lo definido por el conjunto de reglas del esquema YAML Failsafe.
- Todos los nombres de propiedades o campos de la especificación distinguen entre mayúsculas y minúsculas. Esto incluye todas los propiedades que se utilizan como claves asociativas, excepto donde se indique explícitamente que las claves no distinguen entre mayúsculas y minúsculas .
- El esquema expone dos tipos de propiedades:
  - propiedades fijas: tienen el nombre establecido en el estándar
  - propiedades con patrón: sus nombres son de creación libre pero deben cumplir una expresión regular (patrón) definida en el estándar y deben ser únicos dentro del objeto contenedor.

Sintaxis

- El sangrado utiliza espacios en blanco, no se permite el uso de caracteres de tabulación.
- Los miembros de las listas van entre corchetes ([ ]) y separados por coma seguida de espacio (,), o uno por línea con un guion (-) inicial.
- Los vectores asociativos se representan usando los dos puntos seguidos por un espacio, "clave: valor", bien uno por línea o entre llaves ({ }) y separados por coma seguida de espacio (,).
- Un valor de un vector asociativo viene precedido por un signo de interrogación (?), lo que permite que se construyan claves complejas sin ambigüedad.
- Los valores sencillos (o escalares) por lo general aparecen sin entrecomillar, pero pueden incluirse entre comillas dobles ("), o apostrofes (').

Sintaxis

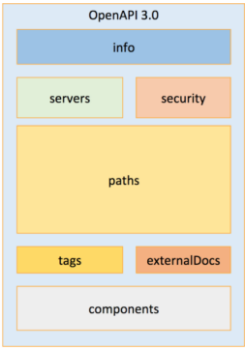
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.
- Es sensible a mayúsculas y minúsculas, todas las propiedades (palabras reservadas) de la especificación deben ir en minúsculas y terminar en dos puntos (:).
- Las propiedades requieren líneas independiente, su valor puede ir a continuación en la misma línea (precedido por un espacio) o en múltiples líneas (con sangrado)
- Las descripciones textuales pueden ser multilínea y admiten el dialecto CommonMark de Markdown para una representación de texto enriquecido. El HTML es compatible en la medida en que lo proporciona CommonMark (Bloques HTML en la Especificación 0.27 de CommonMark).
- $ref permite sustituir, reutilizar y enlazar una definición local con una externa.

CommonMark de Markdown

- Requiere doble y triple salto de línea para saltos de párrafos y cierre de bloques. Dos espacios al final de la línea lo convierte en salto de línea.
- Regla horizontal (separador): ---
- Énfasis: *cursiva* **negrita** ***cursiva y negrita***
- Enlaces: <http://www.example.com> [texto](http://www. example.com)
- Imágenes: ![Image](http://www.example.com/logo.png "icon")
- Citas: > Texto de la cita con sangría
- Bloques de códigos: `Encerrados entre tildes graves`
- Listas: Dos espacios en blanco por nivel de sangrado.
  + Listas desordenadas 1. Listas ordenadas
- Encabezado: dos líneas debajo del texto, añadir cualquier número de caracteres = para el nivel de título 1, <h1> … <h6> (el # es interpretado como comentario).

Estructura básica

- Un documento de OpenAPI puede estar compuesto por un solo documento o dividirse en múltiples partes conectadas a discreción del usuario. En el último caso, los campos $ref deben utilizarse en la especificación para hacer referencia a esas partes.
- Se recomienda que el documento raíz de OpenAPI se llame: openapi.json u openapi.yaml.
- La especificación de la API se puede dividir en 3 secciones principales:
  - Meta información
  - Elementos de ruta (puntos finales):
    - Parámetros de las solicitud
    - Cuerpo de las solicitud
    - Respuestas
  - Componentes reutilizables:
    - Esquemas (modelos de datos)
    - Parámetros
    - Respuestas
    - Otros componentes



Estructura básica

```
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in ...
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```


Estructura básica (cont)

```
components:
  schemas:
    User:
      properties:
        id:
          type: integer
        name:
          type: string
      # Both properties are required
      required:
        - id
        - name
  securitySchemes:
      BasicAuth:
        type: http
        scheme: basic
security:
  - BasicAuth: []
```

Prologo

- Cada definición de API debe incluir la versión de la especificación OpenAPI en la que se basa el documento en la propiedad openapi.
- La propiedad info contiene información de la API:
  - title es el nombre de API.
  - description es información extendida sobre la API.
  - version es una cadena arbitraria que especifica la versión de la API (no confundir con la revisión del archivo o la versión del openapi).
  - también admite otras palabras clave para información de contacto (nombre, url, email), licencia (nombre, url), términos de servicio (url) y otros detalles.
- La propiedad servers especifica el servidor API y la URL base. Se pueden definir uno o varios servidores (elementos precedidos por -).
- Con la propiedad externalDocs se puede referenciar la documentación externa adicional.

Rutas

- La sección paths define los puntos finales individuales (rutas) en la API y los métodos (operaciones) HTTP admitidos por estos puntos finales.
- Las ruta es relativa a la ruta del objeto Server.
- Los parámetros de la ruta se pueden usar para aislar un componente específico de los datos con los que el cliente está trabajando. Los parámetros de ruta son parte de la ruta y se expresan entre llaves (/users/{userId}), participan en la jerarquía de la URL y, por lo tanto, se apilan secuencialmente. Los parámetros de ruta deben describirse obligatoriamente en parameters (común para todas las operaciones) o a nivel de operación individual.
- No puede haber dos rutas iguales o ambiguas, que solo se diferencian por el parámetro de ruta.
- La definición de la ruta pude tener con un resumen (summary) y una descripción (description).
- Una ruta debe contar con un conjunto de operaciones, al menos una.
- Opcionalmente, servers permite dar una matriz alternativa de server que den servicio a todas las operaciones en esta ruta.

Rutas

```
'/users/{id}/roles':
   get:
     summary: Returns a list of users's roles.
     operationId: getDirecciones
     parameters:
       - in: path
         name: id
         description: User ID
         required: true
         schema:
           type: number
       - in: query
         name: size
         schema:
           type: string
           enum: [long, medium, short]
         required: true
       - in: query
         name: page
         schema:
           type: integer
           minimum: 0
           default: 0
```

```
responses:
  '200':
    description: List of roles
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Roles'
  '400':
    description: Bad request. User ID must be an
      integer and larger than 0.
  '401':
    description: Authorization information is missing
      or invalid.
  '404':
    description: A user with the specified ID was not
      found.
  '5XX':
    description: Unexpected error.
  default:
    description: Default error sample response
```


Operaciones

- Describe una única operación de API en una ruta y se identifica con el nombre del método HTTP: get, put, post, delete, options, head, patch, trace.
- Una definición de operación puede incluir un breve resumen de lo que hace (summary), una explicación detallada del comportamiento (description), una referencia a documentación externa adicional (externalDocs), un identifidor único para su uso en herramientas y bibliotecas (operationId) y si está obsoleta y debería dejar de usarse (deprecated).
- Las operaciones pueden tener parámetros pasados a través de la ruta URL (/users/{userId}), cadena de consulta (/users?role=admin), encabezados (X-CustomHeader: Value) o cookies (Cookie: debug=0).
- Si la petición (POST, PUT, PATCH) envía un cuerpo en la solicitud (body), la propiedad requestBody permite describir el contenido del cuerpo y el tipo de medio.
- Para cada las respuestas de la operación, se pueden definir los posibles códigos de estado y el schema del cuerpo de respuesta. Los esquemas pueden definirse en línea o referenciarse mediante $ref. También se pueden proporcionar ejemplos para los diferentes tipos de respuestas.

Parámetros

- Un parámetro único se define mediante una combinación de nombre (name) y ubicación (in: "query", "header", "path" o "cookie") en la propiedad parameters.
- Opcionalmente puede ir acompañado por una breve descripción del parámetro (description), si es obligatorio (required), si permite valores vacíos (allowemptyvalue) y si está obsoleto y debería dejar de usarse (deprecated).
- Las reglas para la serialización del parámetro se especifican dos formas:
  - Para los escenarios más simples, con schema y style se puede describir la estructura y la sintaxis del parámetro.
  - Para escenarios más complejos, la propiedad content puede definir el tipo de medio y el esquema del parámetro.
- Un parámetro debe contener la propiedad schema o content, pero no ambas.
- Se puede proporcionar un example o examples pero debe seguir la estrategia de serialización prescrita para el parámetro.

Parámetros

```
paths:
  /users:
    get:
      description: Returns a list of users
      parameters:
        - name: rows
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: page
          in: query
          description: Specifies the page number of the users to be displayed
          schema:
            type: integer
```

Cuerpo de la solicitud

- En versiones anteriores, el cuerpo de la solicitud era un parámetro mas in: body.
- Actualmente se utiliza la propiedad requestBody con una breve descripción (description) y si es obligatorio para la solicitud (required), ambas opcionales.
- La descripción del contenido (content) es obligatoria y se estructura según los tipos de medios que actúan como identificadores. Para las solicitudes que coinciden con varias claves, solo se aplica la clave más específica (text/plain → text/* → */*).
- Por cada tipo de medio se puede definir el esquema del contenido de la solicitud (schema), uno (example) o varios (examples) ejemplos y la codificación (encoding).
- El requestBody sólo se admite en métodos HTTP donde la especificación HTTP 1.1 RFC7231 haya definido explícitamente semántica para cuerpos de solicitud.

Cuerpo de la solicitud

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - username
              properties:
                username:
                  type: string
                password:
                  type: string
                  format: password
                name:
                  type: string
```

Respuestas

- Es obligaría la propiedad responses con la lista de posibles respuestas que se devuelven al ejecutar esta operación.
- No se espera necesariamente que la documentación cubra todos los códigos de respuesta HTTP posibles porque es posible que ni se conozcan de antemano. Sin embargo, se espera que cubra la respuesta de la operación cuando tiene éxito y cualquier error previsto.
- La posibles respuestas se identifican con el código de respuesta HTTP. Con default se puede definir las respuesta por defecto para todos los códigos HTTP que no están cubiertos por la especificación individual.
- La respuesta cuenta con una breve descripción de la respuesta (description) y, opcionalmente, el contenido estructurado según los tipos de medios (content), los encabezados (headers) y los enlaces de operaciones que se pueden seguir desde la respuesta (links).

Respuestas

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody: # …
      responses:
        '201':
          description: Successfully created a new user
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  code:
                    type: integer
                  message:
                    type: string
```

Etiquetas

- Las etiquetas son metadatos adicionales que permiten organizar la documentación de la especificación de la API y controlar su presentación. Las etiquetas se pueden utilizar para la agrupación lógica de operaciones por recursos o cualquier otro calificador. El orden de las etiquetas se puede utilizar para reflejar un orden en las herramientas de análisis.
- Cada nombre de etiqueta en la lista debe ser único (name) y puede ir acompañado por una explicación detallada (description) y una referencia a documentación externa adicional (externalDocs).
- Las etiquetas se pueden declarar en la propiedad tags del documento:

```
tags:
  - name: security-resource
    description: Gestión de la seguridad
```


Etiquetas

- Las etiquetas se aplican en la propiedad tags de las operaciones:

```
paths:
  /users:
    get:
      tags:
        - security-resource
  /roles:
    get:
      tags:
        - security-resource
        - read-only-resource
```

- No es necesario declarar todas las etiquetas, se pueden usar directamente pero no se podrá dar información adicional y se mostraran ordenadas al azar o según la lógica de las herramientas.

Componentes

- La propiedad global components permite definir las estructuras de datos comunes utilizadas en la especificación de la API: Contiene un conjunto de objetos reutilizables para diferentes aspectos de la especificación.
- Todos los objetos definidos dentro del objeto de componentes no tendrán ningún efecto en la API a menos que se haga referencia explícitamente a ellos desde propiedades fuera del objeto de componentes.
- La sección components dispones de propiedades para schemas, responses, parameters, examples, requestBodies, headers, securitySchemes, links y callbacks.
- Se puede hacer referencia a ellos con $ref cuando sea necesario. $ref acepta referencias internas con # o externas con el nombre de un fichero. La referencia debe incluir la trayectoria para encontrar el elemento referenciado:

  ```
  $ref: '#/components/schemas/Rol'
  $ref: responses.yaml#/404Error
  ```
- El uso de referencias permite la reutilización de elementos ya definidos, facilitando la mantenibilidad y disminuyendo sensiblemente la longitud de la especificación, por lo que se deben utilizar extensivamente. Las referencias no interfieren con la presentación en el UI.

Esquemas de datos

- Los schemas definen los modelos de datos consumidos y devueltos por la API.
- Los tipos de datos OpenAPI se basan en un subconjunto extendido del JSON Schema Specification Wright Draft 00 (también conocido como Draft 5).
- Los tipos base son string, number, integer, boolean, array y object.
- Con la propiedad format se pueden especificar otros tipos especiales partiendo de los tipos base: long, float, double, byte, binary, date, dateTime, password.
- Los tipos array se definen como una colección de ítems y en dicha propiedad se define el tipo y la estructura de los elementos que lo componen. Los objetos son un conjunto de propiedades, cada una definida dentro de properties.
- Cada tipo y propiedad se identifica por un nombre que no debe estar repetido en su ámbito.
- Cada propiedad puede definir description, default, minimum, maximum, maxLength, minLength, pattern, required, readOnly, …
- Para una propiedad se pueden definir varios tipos (tipos mixtos o unión).
- Los tipos pueden hacer referencia a otros tipos.

Tipos de datos

type	format	Comentarios
boolean		Booleanos: true y false
integer	int32	Enteros con signo de 32 bits
integer	int64	Enteros con signo de 64 bits (también conocidos como largos)
number	float	Reales cortos
number	double	Reales largos
string		Cadenas de caracteres
string	password	Una pista a las IU para ocultar la entrada.
string	date	Según lo definido por full-date RFC3339 (2018-11-13)
string	date-time	Según lo definido por date-time- RFC3339 (2018-11-13T20:20:39+00:00)
string	byte	Binario codificados en base64
string	binary	Binario en cualquier secuencia de octetos
array		Colección de items
object		Colección de properties

Propiedades de los objetos de esquema

- type: integer, number, boolean, string, array, object
- format: long, float, double, byte, binary, date, dateTime, password
- title: Nombre a mostrar en el UI
- description: Descripción de su uso
- maximum: Valor máximo
- exclusiveMaximum: Valor menor que
- minimum: Valor mínimo
- exclusiveMinimum: Valor mayor que
- multipleOf: Valor múltiplo de
- maxLength: Longitud máxima
- minLength: Longitud mínima
- pattern: Expresión regular del patrón
- deprecated: Si está obsoleto y debería dejar de usarse
- nullable: Si acepta nulos
- default: Valor por defecto
- enum: Lista de valores con nombre
- example: Ejemplo de uso
- externalDocs: referencia a documentación externa adicional
- items: Definición de los elementos del array
- maxItems: Número máximo de elementos
- minItems: Número mínimo de elementos
- uniqueItems: Elementos únicos
- properties: Definición de las propiedades del objeto,
- maxProperties: Número máximo de propiedades
- minProperties: Número mínimo de propiedades
- readOnly: propiedad de solo lectura
- writeOnly: propiedad de solo escritura
- additionalProperties: permite referenciar propiedades adicionales
- required: Lista de propiedades obligatorias

Modelos de entrada y salida

```
components:
  schemas:
    Roles:
      type: array
      items:
        $ref: '#/components/schemas/Rol'
    Rol:
      type: object
      description: Roles de usuario
      properties:
        rolId:
          type: integer
          format: int32
          minimum: 0
          maximum: 255
        name:
          type: string
          maxLength: 20
        description:
          type: string
        last_updated:
          type: string
          format: dateTime
          readOnly: true
        level:
            type: string
            description: Nivel de permisos
            enum:
            - high
            - normal
            - low
            default: normal
    required:
    - rolId
    - name
```


Autenticación

- La propiedad securitySchemes de components y la propiedad security del documento se utilizan para describir y establecer los métodos de autenticación utilizados en la API.
- securitySchemes define los esquemas de seguridad que pueden utilizar las operaciones. Los esquemas admitidos son la autenticación HTTP, una clave API (ya sea como encabezado, parámetro de cookie o parámetro de consulta), los flujos comunes de OAuth2 (implícito, contraseña, credenciales de cliente y código de autorización) tal y como se define en RFC6749 y OpenID Connect Discovery. Cada esquema cuenta con un identificador, un tipo (type: "apiKey", "http", "oauth2", "openIdConnect") y opcionalmente puede ir acompañado por una breve descripción (description).
- Según el tipo seleccionado será obligatorio:
  - apiKey: ubicación (in: "query", "header" o "cookie") y su nombre (name) de parámetro, encabezado o cookie.
  - http: esquema de autorización HTTP que se utilizará en el encabezado Authorization (scheme): Basic, Bearer, Digest, OAuth, … y, si es Bearer, prefijo del token de portador (bearerFormat).
  - openIdConnect: URL de OpenId Connect para descubrir los valores de configuración de OAuth2 (openIdConnectUrl).
  - oauth2: objeto que contiene información de configuración para los tipos de flujo admitidos (flows).
- La propiedad security enumera los esquemas de seguridad que se pueden utilizar en la API.

Autenticación

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    JWTAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    ApiKeyAuth:
      type: apiKey
      name: x-api-key
      in: header
    ApiKeyQuery:
      type: apiKey
      name: api-key
      in: query
security:
  - ApiKeyAuth: []
  - ApiKeyQuery: []
```


Ejemplos

- Los ejemplos son fundamentales para la correcta comprensión de la documentación. La especificación permite proporcionar uno (example) o varios (examples) ejemplos asociados a las estructuras de datos.
- Por cada uno se puede dar un resumen del ejemplo (summary), una descripción larga (description), el juego de valores de las propiedades de la estructura (value) o una URL que apunta al ejemplo literal para ejemplos que no se pueden incluir fácilmente en documentos JSON o YAML (externalValue). value y externalValue son mutuamente excluyentes. Cuando son varios ejemplos deber estar identificados por un nombre único.

```
examples:
  first-page:
    summary: Primera página
    value: 0
  second-page:
    summary: Segunda página
    value: 1
```

- Los ejemplos pueden ser utilizados automáticamente por las herramientas de UI y de generación de pruebas.

Ecosistema Swagger

- Swagger Open Source Tools
  - Swagger Editor: Diseñar APIs en un potente editor de OpenAPI que visualiza la definición y proporciona comentarios de errores en tiempo real.
  - Swagger Codegen: Crear y habilitar el consumo de su API generando la fontanería del servidor y el cliente.
  - Swagger UI: Generar automáticamente la documentación desde la definición de OpenAPI para la interacción visual y un consumo más fácil.
- Swagger Pro Tools
  - SwaggerHub: La plataforma de diseño y documentación para equipos e individuos que trabajan con la especificación OpenAPI.
  - Swagger Inspector: La plataforma de pruebas y generación de documentación de las APIs
- https://openapi.tools/

Que debe incluir

- Una explicación clara de lo que hace el método / recurso.
- Una lista de los parámetros utilizados en este recurso / método,
- Posibles respuestas, que comparten información importante con los desarrolladores, incluidas advertencias y errores
- Descripción de los tipos, formatos especial, reglas y restricciones.
- Una invocación y una respuesta de ejemplo, incluido los cuerpos con los media-type correspondientes.
- Ejemplos de código para varios lenguajes, incluido todo el código necesario (por ejemplo, Curl con PHP, así como ejemplos para Java, .Net, Ruby, etc.)
- Ejemplos de SDK (si se proporcionan SDK) que muestren cómo acceder al recurso / método utilizando el SDK para los lenguajes en que se suministra.
- Experiencias interactivas para probar las llamadas API.
- Preguntas frecuentes / escenarios con ejemplos de código
- Enlaces a recursos adicionales (otros ejemplos, blogs, etc.)
- Una sección de comentarios donde los usuarios pueden compartir / discutir el código.

CONSULTAS ENTRE SERVICIOS

Patrón: Registro de servicios

- Motivación:
  - Los clientes de un servicio utilizan el descubrimiento del lado del cliente o el descubrimiento del lado del servidor para determinar la ubicación de una instancia de servicio a la que enviar las solicitudes.
- Intención:
  - ¿Cómo saben los clientes de un servicio (en el caso del descubrimiento del lado del cliente) y / o enrutadores (en el caso del descubrimiento del lado del servidor) acerca de las instancias disponibles de un servicio?
- Requisitos:
  - Cada instancia de un servicio expone una API remota como HTTP/REST o Thrift, etc. en una ubicación particular (host y puerto)
  - El número de instancias de servicios y sus ubicaciones cambia dinámicamente. A las máquinas virtuales y los contenedores se les suele asignar una dirección IP dinámica. Un grupo de autoescalado de EC2, por ejemplo, ajusta el número de instancias en función de la carga.
- Solución:
  - Implementar un registro de servicios, que es una base de datos de servicios, sus instancias y sus ubicaciones. Las instancias de servicio se registran con el registro de servicios en el inicio y se cancelan en el cierre. El cliente del servicio y / o los enrutadores consultan el registro de servicios para encontrar las instancias disponibles de un servicio. Un registro de servicios puede invocar la API de comprobación de estado de una instancia de servicio para verificar que es capaz de manejar solicitudes

Patrón: Registro de servicios

- Implementación:
  - Crear un servidor Eureka con Spring Boot y Spring Cloud
- Consecuencias:
  - Los beneficios del patrón de registro de servicios incluyen:
    - El cliente del servicio y / o los enrutadores pueden descubrir la ubicación de las instancias de servicio.
  - También hay algunos inconvenientes:
    - A menos que el registro de servicios esté integrado en la infraestructura, es otro componente de la infraestructura que debe que debe instalarse, configurarse y mantenerse.
    - Además, el registro de servicios es un componente crítico del sistema, debe estar altamente disponible.
    - Si bien los clientes deben almacenar en caché los datos proporcionados por el registro de servicios, si el registro de servicios falla, dichos datos eventualmente quedarán desactualizados.

Patrón: Registro de servicios

- Consecuencias:
  - Se debe decidir cómo se registran las instancias de servicio con el Registro de servicios. Hay dos opciones:
    - Patrón de auto registro: las instancias de servicio se registran a sí mismas.
    - Patrón de registro de terceros : un tercero registra las instancias de servicio con el Registro de servicios.
  - Los clientes del registro de servicios necesitan conocer las ubicaciones de las instancias del registro de servicios. Las instancias de registro de servicios deben implementarse en direcciones IP fijas y conocidas. Los clientes se configuran con esas direcciones IP.
  - Por ejemplo, las instancias de servicio de Netflix Eureka se implementan normalmente utilizando direcciones IP elásticas. El grupo disponible de direcciones IP elásticas se configura mediante un archivo de propiedades o mediante DNS. Cuando se inicia una instancia de Eureka, consulta la configuración para determinar qué dirección IP elástica disponible usar. Un cliente Eureka también está configurado con el conjunto de direcciones IP elásticas.
- Patrones relacionados:
  - Descubrimiento del lado del cliente y el descubrimiento del lado del servidor crean la necesidad de un registro de servicios
  - El registro automático y el registro de terceros son dos formas diferentes en que las instancias de servicio pueden registrarse con el registro de servicio
  - Health Check API : el registro de servicios invoca la API de control de salud de una instancia de servicio para verificar que puede manejar solicitudes

Patrón: Auto registro

- Motivación:
  - Ha aplicado el patrón de descubrimiento de servicios del lado del cliente o el patrón de descubrimiento de servicios del lado del servidor. Las instancias de servicio se deben registrar en el registro de servicios al inicio para que puedan descubrirse y anularse el registro en el cierre.
- Intención:
  - ¿Cómo se registran y anulan del registro de servicios las instancias de servicio?
- Requisitos:
  - Las instancias de servicio se deben registrar con el registro de servicios en el inicio y anular el registro en el cierre
  - Las instancias de servicio que se cuelgan se deben anular del registro de servicios
  - Las instancias de servicio que se ejecutan pero que no pueden manejar las solicitudes deben ser anuladas del registro de servicios
- Solución:
  - Una instancia de servicio es responsable de registrarse en el registro de servicios. Al iniciarse, la instancia de servicio se registra a sí misma (host y dirección IP) con el registro de servicios y queda disponible para su descubrimiento. El cliente normalmente debe renovar periódicamente su registro para que el servidor de registro sepa que aún está vivo. Al cerrarse, la instancia de servicio se anula el registro del servicio.

Patrón: Auto registro

- Implementación:
  - Incluir y configurar el cliente de Eureka en los microservicios
- Consecuencias:
  - Los beneficios del patrón de auto registro incluyen los siguientes:
    - Una instancia de servicio conoce su propio estado, por lo que puede implementar un modelo de estado más complejo que ARRIBA / ABAJO, por ejemplo, INICIAR, DISPONIBLE, ...
  - También hay algunos inconvenientes:
    - Acopla el servicio al registro de servicios.
    - Debe implementar la lógica de registro de servicios en cada tecnología que se utilice para escribir sus servicios, por ejemplo, NodeJS / JavaScript, Java, .NET, etc.
    - Una instancia de servicio que se está ejecutando pero que no puede manejar las solicitudes a menudo carecerá de la autoconciencia para darse de baja del registro de servicios.
- Patrones relacionados:
  - Registro de servicios: una parte esencial del descubrimiento de servicios
  - Descubrimiento del lado del cliente: una forma en que se descubre una instancia de servicio
  - Descubrimiento del lado del servidor: otra forma en que se descubre una instancia de servicio
  - Chasis de microservicio: el auto registro es responsabilidad del marco del chasis de microservicio
  - El registro de terceros es una solución alternativa

Patrón: Registro de terceros

- Motivación:
  - Ha aplicado el patrón de descubrimiento de servicios del lado del cliente o el patrón de descubrimiento de servicios del lado del servidor. Las instancias de servicio se deben registrar en el registro de servicios al inicio para que puedan descubrirse y anularse el registro en el cierre.
- Intención:
  - ¿Cómo se registran y anulan del registro de servicios las instancias de servicio?
- Requisitos:
  - Las instancias de servicio se deben registrar con el registro de servicios en el inicio y anular el registro en el cierre
  - Las instancias de servicio que se cuelgan se deben anular del registro de servicios
  - Las instancias de servicio que se ejecutan pero que no pueden manejar las solicitudes deben ser anuladas del registro de servicios
- Solución:
  - Un registrador externo es responsable de registrar y anular el registro de una instancia de servicio en el registro de servicios. Cuando se inicia la instancia de servicio, el registrador registra la instancia de servicio con el registro de servicios. Cuando la instancia de servicio se apaga, el registrador anula el registro de la instancia de servicio del registro de servicios.

Patrón: Registro de terceros

- Implementación:
  - Netflix Prana: una aplicación de "side car" que se ejecuta junto con una aplicación que no es JVM y registra la aplicación con Eureka.
  - AWS Autoscaling Groups: (des)registran automáticamente las instancias EC2 con Elastic Load Balancer
  - Joyent's Container o Registrator: (des)registran el registro de contenedores Docker
  - Los marcos de clústeres como Kubernetes y Marathon (des)registran instancias de servicio con el registro integrado / implícito
- Consecuencias:
  - Los beneficios del patrón de registro de terceros incluyen:
    - No requiere código en el servicio como cuando se usa el patrón de registro automático, ya que no es responsable de registrarse.
    - El registrador puede realizar comprobaciones de estado en una instancia de servicio y registrar / anular el registro de la instancia basándose en la comprobación de estado
  - También hay algunos inconvenientes:
    - El registrador externo solo puede tener un conocimiento superficial del estado de la instancia de servicio, por ejemplo, EN EJECUCIÓN o NO EN EJECUCIÓN y, por lo tanto, puede no saber si puede manejar solicitudes. Sin embargo, algunos registradores como Netflix Prana realizan una verificación de estado para determinar la disponibilidad de la instancia de servicio.
    - A menos que el registrador sea parte de la infraestructura, es otro componente que debe instalarse, configurarse y mantenerse. Además, como es un componente crítico del sistema, debe estar altamente disponible.
- Patrones relacionados:
  - Registro de servicios
  - Descubrimiento del lado del cliente
  - Descubrimiento del lado del servidor
  - Auto registro es una solución alternativa

Patrón: Descubrimiento del lado del cliente

- Motivación:
  - Los servicios normalmente necesitan llamarse unos a otros. En una aplicación monolítica, los servicios se invocan unos a otros a través de un método de nivel de lenguaje o llamadas a procedimientos. En una implementación de sistema distribuido tradicional, los servicios se ejecutan en ubicaciones fijas y bien conocidas (hosts y puertos) y, por lo tanto, pueden llamarse fácilmente utilizando HTTP / REST o algún mecanismo RPC. Sin embargo, una aplicación moderna basada en microservicios generalmente se ejecuta en entornos virtualizados o en contenedores, donde la cantidad de instancias de un servicio y sus ubicaciones cambian dinámicamente.
- Intención:
  - ¿Cómo descubre el cliente de un servicio, la puerta de enlaces API u otro servicio, la ubicación de una instancia de servicio?
- Requisitos:
  - Cada instancia de un servicio expone una API remota como HTTP / REST o Thrift, etc. en una ubicación particular (host y puerto)
  - El número de instancias de servicios y sus ubicaciones cambia dinámicamente.
  - Las máquinas virtuales y los contenedores suelen tener asignadas direcciones IP dinámicas.
  - El número de instancias de servicios puede variar dinámicamente. Por ejemplo, un grupo de autoescalado de EC2 ajusta el número de instancias en función de la carga.

Patrón: Descubrimiento del lado del cliente

- Solución:
  - Al realizar una solicitud a un servicio, el cliente obtiene la ubicación de una instancia de servicio consultando un registro de servicios, que conoce las ubicaciones de todas las instancias de servicio.
- Implementación:
  - Crear un cliente con Spring Boot y Spring Cloud: Cliente Eureka, balanceo de carga (Ribbon), cliente Rest (RestTemplate o Feign)
- Consecuencias:
  - El descubrimiento del lado del cliente tiene los siguientes beneficios:
    - Menos partes móviles y saltos de red en comparación con el descubrimiento del lado del servidor
  - El descubrimiento del lado del cliente también tiene los siguientes inconvenientes:
    - Este patrón une al cliente con el registro de servicios.
    - Debe implementar la lógica de descubrimiento en cada tecnología que se utilice para escribir los servicios, por ejemplo, NodeJS / JavaScript, Java, .NET, etc. Por ejemplo, Netflix Prana proporciona un enfoque basado en proxy HTTP para el descubrimiento de servicios para clientes que no son JVM.
- Patrones relacionados:
  - Registro de servicios: una parte esencial del descubrimiento de servicios
  - Chasis de microservicio: el descubrimiento del servicio del lado del cliente es responsabilidad del marco del chasis de microservicio
  - Descubrimiento del lado del servidor es una solución alternativa

Patrón: Descubrimiento del lado del servidor

- Motivación:
  - Los servicios normalmente necesitan llamarse unos a otros. En una aplicación monolítica, los servicios se invocan unos a otros a través de un método de nivel de idioma o llamadas a procedimientos. En una implementación de sistema distribuido tradicional, los servicios se ejecutan en ubicaciones fijas y bien conocidas (hosts y puertos) y, por lo tanto, pueden llamarse fácilmente utilizando HTTP / REST o algún mecanismo RPC. Sin embargo, una aplicación moderna basada en microservicios generalmente se ejecuta en entornos virtualizados o en contenedores, donde la cantidad de instancias de un servicio y sus ubicaciones cambian dinámicamente.
- Intención:
  - ¿Cómo descubre el cliente de un servicio, la puerta de enlaces API u otro servicio, la ubicación de una instancia de servicio?
- Requisitos:
  - Cada instancia de un servicio expone una API remota como HTTP / REST o Thrift, etc. en una ubicación particular (host y puerto)
  - El número de instancias de servicios y sus ubicaciones cambia dinámicamente.
  - Las máquinas virtuales y los contenedores suelen tener asignadas direcciones IP dinámicas.
  - El número de instancias de servicios puede variar dinámicamente. Por ejemplo, un grupo de autoescalado de EC2 ajusta el número de instancias en función de la carga.

Patrón: Descubrimiento del lado del servidor

- Solución:
  - Al realizar una solicitud a un servicio, el cliente realiza una solicitud a través de un enrutador (también conocido como equilibrador de carga) que se ejecuta en una ubicación bien conocida. El enrutador consulta un registro de servicios, que podría estar integrado en el enrutador, y reenvía la solicitud a una instancia de servicio disponible.
- Implementación:
  - AWS Elastic Load Balancer (ELB) o algunas solución de agrupación en clústeres, como Kubernetes, Marathon o Azure Service Fabric, que ejecutan un proxy en cada host que funciona como un enrutador de descubrimiento del lado del servidor.
  - Crear un cliente con Spring Boot y Spring Cloud: cliente Rest (RestTemplate o Feign)
- Consecuencias:
  - El descubrimiento del lado del servidor tiene los siguientes beneficios:
    - En comparación con el descubrimiento del lado del cliente, el código del cliente es más simple, ya que no tiene que lidiar con el descubrimiento. En su lugar, un cliente simplemente hace una solicitud al enrutador
    - Algunos entornos de nube proporcionan esta funcionalidad, por ejemplo, AWS Elastic Load Balancer.
  - El descubrimiento del lado del servidor también tiene los siguientes inconvenientes:
    - A menos que sea parte del entorno de la nube, el enrutador debe ser otro componente del sistema que debe instalarse y configurarse. También deberá ser replicado para disponibilidad y capacidad.
    - El enrutador debe admitir los protocolos de comunicación necesarios (por ejemplo, HTTP, gRPC, Thrift, etc.) a menos que sea un enrutador basado en TCP
    - Se requieren más saltos de red que al usar el descubrimiento del lado del cliente y se tiene menos control.
- Patrones relacionados:
  - Registro de servicios: una parte esencial del descubrimiento de servicios
  - Descubrimiento del lado del cliente es una solución alternativa

Patrón: API Gateway

- Motivación:
  - Ha aplicado el patrón de arquitectura de Microservicios, necesita desarrollar varias versiones de la interfaz de usuario:
    - UI basada en JavaScript / HTML5 para navegadores de escritorio y móviles (SPA): el navegador del usuario interactúa con el servidor a través de peticiones AJAX a las API REST
    - Clientes nativos de Android y iPhone: estos clientes interactúan con el servidor a través de las API REST
  - Dado que se utiliza el patrón de arquitectura de Microservicios, los datos a mostrar y mantener se distribuyen en múltiples servicios.
- Intención:
  - ¿Cómo acceden los clientes de una aplicación basada en microservicios a los servicios individuales?
- Requisitos:
  - La granularidad de las API proporcionadas por los microservicios (APIs detalladas) a menudo es diferente de lo que necesita un cliente, lo que implica que los clientes necesitan interactuar con múltiples servicios.
  - Diferentes clientes necesitan diferentes datos. Por ejemplo, la versión del navegador de escritorio de una página suele ser más elaborada que la versión móvil.
  - El rendimiento de la red es diferente para los diferentes tipos de clientes: una red móvil suele ser mucho más lenta y tiene una latencia mucho mayor que una red no móvil, cualquier WAN es mucho más lenta que una LAN. Esto significa que un cliente móvil nativo usa una red que tiene características de rendimiento muy diferentes a las de una LAN utilizada por una aplicación web del lado del servidor. La aplicación web del lado del servidor puede realizar múltiples solicitudes a los servicios de backend sin afectar la experiencia del usuario, donde un cliente móvil solo puede realizar algunas.
  - La cantidad de instancias de servicio y sus ubicaciones (host + puerto) cambian dinámicamente
  - La partición en servicios puede cambiar con el tiempo y debe ocultarse a los clientes
  - Los servicios pueden usar un conjunto diverso de protocolos, algunos de los cuales pueden no ser compatibles con la web

Patrón: API Gateway

- Solución:
  - Implementar una puerta de enlaces API que sea el único punto de entrada para todos los clientes.
  - La puerta de enlaces API maneja las solicitudes de una de dos maneras.
    - Algunas solicitudes simplemente se envían por proxy / enrutan al servicio apropiado.
    - Otras solicitudes se maneja mediante la distribución a múltiples servicios.
  - La puerta de enlaces API puede exponer una API diferente para cada tipo de cliente, en lugar de proporcionar una API de estilo único para todos: un adaptador que proporciona a cada tipo de cliente la API que mejor se adapta a sus requisitos.
  - Backends para frontends es una variación de este patrón que define un API Gateway separado para cada tipo de cliente (web, móviles, terceros, …).
  - La puerta de enlaces API también puede implementar balanceo de carga y seguridad, por ejemplo, para verificar que el cliente esté autorizado para realizar la solicitud
- Implementación:
  - Crear un servidor Zuul o Spring Cloud Gateway con Spring Boot y Spring Cloud

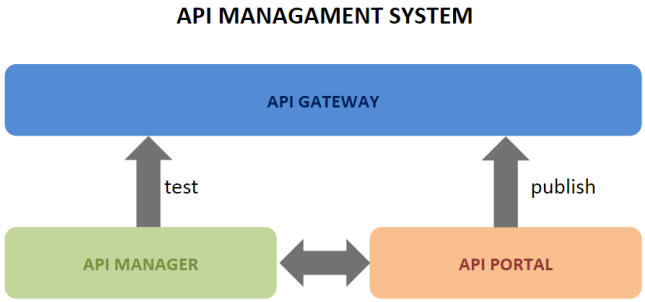
Patrón: API Gateway

- Consecuencias:
  - La puerta de enlaces API tiene los siguientes beneficios:
    - Aísla a los clientes de cómo se particiona la aplicación en microservicios
    - Aísla a los clientes del problema de determinar las ubicaciones de las instancias de servicio
    - Proporciona la API óptima para cada cliente
    - Simplifica el cliente moviendo la lógica para llamar múltiples servicios desde el cliente a la puerta de enlaces API, lo que reduce el número de peticiones: un solo viaje de ida y vuelta para recuperar datos de múltiples servicios. Menos solicitudes implica menos gastos generales y mejora la experiencia del usuario. Una puerta de enlaces API es esencial para las aplicaciones móviles.
    - Puede traducir de un protocolo API público estándar y fácil de usar a cualquier protocolo que se use internamente.
  - La puerta de enlaces API también tiene los siguientes inconvenientes:
    - Mayor complejidad: la puerta de enlaces API es otro componente que debe desarrollarse, implementarse y administrarse
    - Aumento del tiempo de respuesta debido al salto de red adicional a través de la puerta de enlaces API. Sin embargo, para la mayoría de las aplicaciones, el costo de un viaje de ida y vuelta adicional es insignificante.
- Patrones relacionados:
  - La puerta de enlaces API debe usar uno de descubrimiento del servicio
  - Circuit Breaker para invocar servicios
  - Access token, si implementa seguridad.
  - API Composition, si unifica las llamada a múltiples servicios en una.

API MANAGEMENT

API Managament System

- Podemos definir el sistema de gestión de APIs como el proceso de publicar, promocionar y supervisar APIs en un entorno seguro y escalable. Asimismo, incluye todos aquellos recursos enfocados a la creación, documentación y socialización de las APIs.



Usuarios y roles

- **Creador**: Un creador es una persona en un rol técnico que comprende los aspectos técnicos de la API (interfaces, documentación, versiones, cómo está expuesta por Gateway, etc.) y utiliza el editor de API para aprovisionar APIs en el API Store. El creador utiliza la API Store para consultar las calificaciones y comentarios proporcionados por los usuarios de API. Los creadores pueden agregar APIs al store pero no pueden administrar su ciclo de vida (por ejemplo, hacerlos visibles para el mundo exterior).
- **Publicador**: un publicador es una persona en un rol de gestión que administra un conjunto de APIs de la empresa, o unidad de negocio, y controla el ciclo de vida de la API así como los aspectos de SLA y monetización.
- **Consumidor**: un consumidor es una persona en un rol de desarrollador que utiliza al API Store para descubrir APIs, ver la documentación y los foros, y calificar/comentar las APIs. Los consumidores se suscriben a las APIs para obtener claves de la API.

API Gateway

- Routing: Enrutamiento de mensajes a diferentes destinos dependiendo del contexto o del contenido del mensaje.
- Soporte muti-protocolo: Protocolos soportados tanto para la publicación de APIs en el componente Gateway como para el enrutamiento a los servicios internos.
- Soporte multi-formato: Componentes destinados a transformar los datos de un formato a otro, o de su enmascaramiento.
- Monitoring: Monitorización del tráfico de entrada y salida.
- Políticas de seguridad: Otorga a las API características de autenticación, autorización y cifrado utilizando estándares o tecnologías conocidas como el cifrado de transporte mediante HTTPS, la suite de seguridad WS-Security para SOAP o el estándar de autorización OAuth para interfaces REST. Compatibilidad con sistemas de gestion de identidades: Active Directory, LDAP, JDBC, etc.
- Políticas de uso: Capacita a las APIs para gestionar políticas de consumo, rendimiento, fallos, etc. para asegurar SLAs y sistemas de pago por uso.

API Manager

- Publicación: Publica las APIs en el componente API Gateway definiendo sus endpoint.
- Edición: Herramienta para el diseño de la interfaz de la API.
- Gestor del ciclo de vida: Permite gestionar los diferentes estados por lo que pasa una API, así como su versión o descatalogación.
- Gestor de políticas de uso: Herramienta para la configuración de reglas de uso tales como pay per use, SLAs, QA, etc.
- Consumo: Monitorización del uso de las APIs y sistema de configuración de alertas según los parámetros de consumo.
- Gestor de políticas de seguridad: Gestiona todas la configuración de seguridad de una API.

API Portal

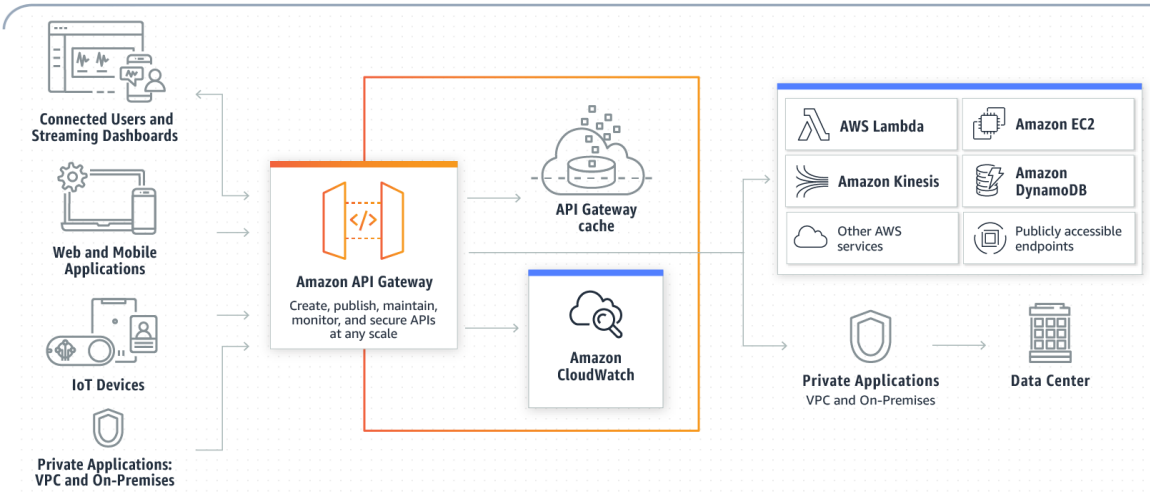
- Tienda: «APIs' Store», donde se localizan las API publicadas, accesos directos a las comunidades de consumidores, herramientas de testing, monitorización, recomendaciones de usuarios, etc.
  - Navegador interno: Buscador de API registradas en el sistema, con varios filtros de consulta como estado, versión, mejor valoración, etc.
  - Comunidad de desarrollo: Publicaciones de noticias y comentarios referentes al uso, configuración, errores y soluciones de las APIs publicadas.
  - Documentación: Repositorio de documentación referente a las APIs publicadas.
  - Probador: Sistema integrado de testeo de cada API.
  - Estadísticas de uso: Sistemas de monitorización y análisis desde la perspectiva del consumidor: timing, status…

AMAZON API GATEWAY

Introducción

- Amazon API Gateway es un servicio completamente administrado que facilita a los desarrolladores la creación, la publicación, el mantenimiento, el monitoreo y la protección de API a cualquier escala. Las API actúan como la "puerta de entrada" para que las aplicaciones accedan a los datos, la lógica empresarial o la funcionalidad de sus servicios de backend.
- Con API Gateway, se pueden crear API RESTful y API WebSocket que permiten aplicaciones de comunicación bidireccional en tiempo real. API Gateway admite cargas de trabajo en contenedores y sin servidor, así como aplicaciones web.
- API Gateway gestiona todas las tareas implicadas en la aceptación y el procesamiento de hasta cientos de miles de llamadas a API simultáneas, entre ellas, la administración del tráfico, compatibilidad con CORS, el control de autorizaciones y acceso, la limitación controlada, el monitoreo y la administración de versiones de API. API Gateway no requiere pagos mínimos ni costos iniciales. Se paga por las llamadas a las API que se reciben y por la cantidad de datos salientes transferidos; además, con el modelo de precios por niveles de API Gateway, puede reducir sus costos a medida que cambie la escala de uso de las API.

Funcionamiento de API Gateway



AZURE API MANAGEMENT

Azure API Management

- Azure API Management (APIM) es una manera de crear puertas de enlace de API coherentes y modernas para servicios de back-end existentes. Optimiza el trabajo en entornos híbridos y multi nube con un único lugar para administrar todas las APIs.
- Para usar API Management, los administradores referencian APIs. Cada API consta de una o varias operaciones y se puede agregar a uno o varios productos. Para usar una API, los desarrolladores se suscriben a un producto que contiene esa API y después pueden llamar a la operación de la API cumpliendo cualquier directiva de uso que pueda estar en vigor.
- El sistema consta de los siguientes componentes:
  - La puerta de enlace de las API
  - Azure Portal
  - Portal para desarrolladores

Azure Application Gateway

- Azure Application Gateway es un equilibrador de carga de tráfico web que permite administrar el tráfico a las aplicaciones web. Opera en la capa de transporte (OSI capa 4: TCP y UDP) y enrutan el tráfico en función de la dirección IP y puerto de origen a una dirección IP y puerto de destino. También puede tomar decisiones de enrutamiento basadas en atributos adicionales de una solicitud HTTP, por ejemplo los encabezados de host o la ruta de acceso del URI.
- La puerta de enlace de la API es el extremo que:
  - Acepta llamadas de API y las enruta a los back-end.
  - Comprueba las claves de API, los tokens JWT, los certificados y otras credenciales.
  - Aplica cuotas de uso y límites de frecuencia.
  - Transforma la API sobre la marcha sin modificaciones de código.
  - Almacena en caché las respuestas de back-end donde se instalaron.
  - Registra los metadatos de llamada para fines de análisis.

Azure Portal

- Azure Portal es la interfaz administrativa donde se configura el programa de API.
- Sirve para:
  - definir o importar el esquema de API
  - empaquetar las APIs en productos
  - establecer directivas, como cuotas o transformaciones, en las APIs
  - obtener información del análisis
  - administrar usuarios

Portal para desarrolladores

- El portal para desarrolladores actúa como la presencia web principal para desarrolladores consumidores de las APIs, donde estos pueden:
  - leer documentación de la API
  - probar una API a través de la consola interactiva
  - crear una cuenta y suscribirse para obtener claves de API
  - obtener acceso a análisis sobre su propio uso
- La dirección URL del portal para desarrolladores se encuentra en Azure Portal para la instancia del servicio API Management.
- Se puede personalizar el aspecto y apariencia del portal para desarrolladores agregando contenido personalizado, personalizando estilos e incorporando su toque diferenciador.

Arquitectura