API Management

Contenidos

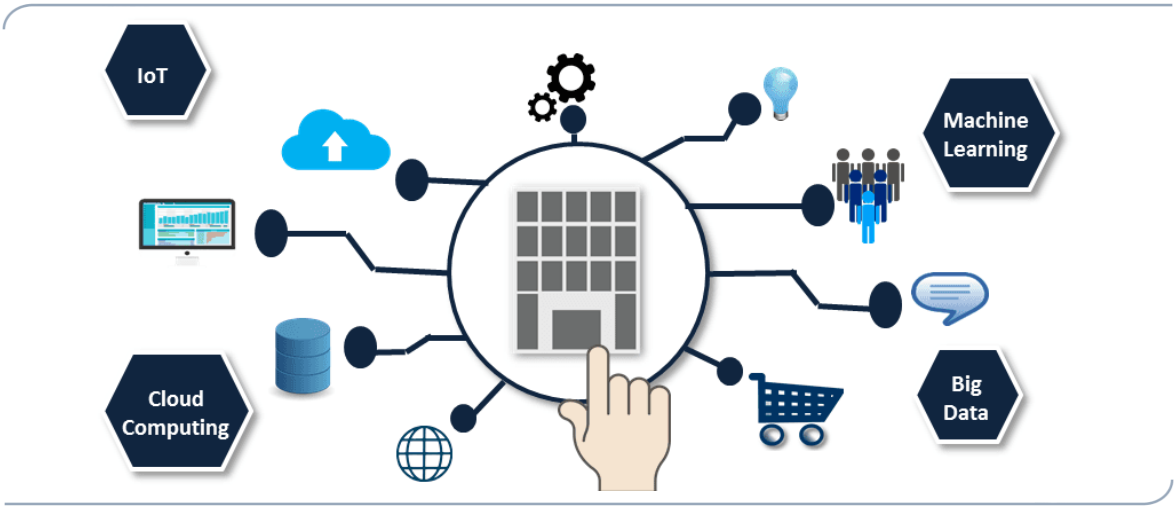
- Introducción
  - Transformación digital
  - Arquitecturas: microservicios, basadas en eventos, …
  - Estilos arquitectónicos: SOAP, REST, AMQP, …
  - Preocupaciones transversales
  - API Economy, API Strategy, API Governance
  - iPaas con Azure Integration Services
- API Governance
  - Definición de recursos
  - Políticas de versionado
  - Seguridad
  - Estándares de definición e implementación
  - Documentación
  - Monitorización
  - Testing
  - Billing
  - Gestión de los entornos
- API First
  - Visión general de servicios REST
  - Diseño de API REST
  - Documentar con Open API - Swagger
  - Definición de una API
  - Visión general de YAML
  - Especificaciones de API con Swagger
  - Enfoque API First Development
  - Ecosistema: Editor de Swagger, Swagger UI, Swagger Codegen, Swagger Inspector, SwaggerHub
- Azure API Management
  - Visión general
  - Portal de administración y
  - Gateway
  - Creación de instancias
  - Importación y publicación de API
  - Políticas y directivas
  - Crear y publicar productos
  - Seguridad
  - Portal del desarrollador
  - Suscripciones
  - Supervisión
  - Ciclo de vida, versionado, revisiones
- Azure Service Bus
  - Conceptos y terminología
  - Creación de Espacios de nombres, colas, temas, retransmisiones y suscripciones
  - Envío/recepción y publicación/suscripción de mensajes
  - Avanzados: filtros, acciones, sesiones, …
  - Supervisión
  - Seguridad

INTRODUCCIÓN

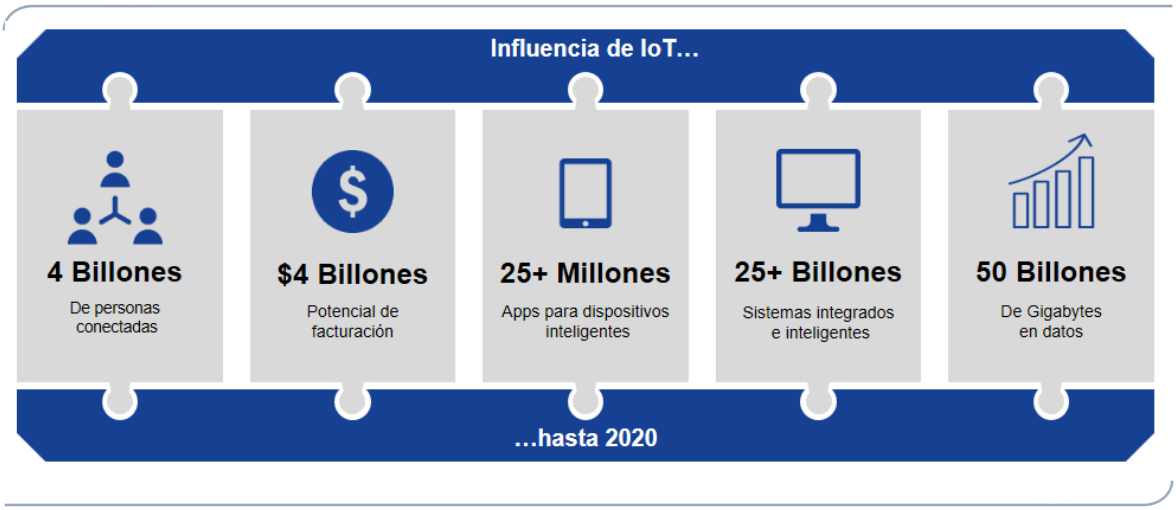
La transformación digital

- 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

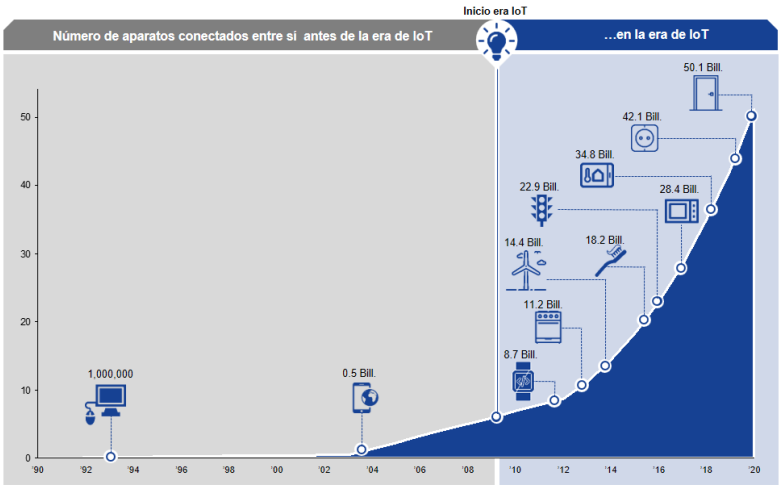


Influencia de IoT



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 asincróno
- 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.

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.

Arquitectura basada en eventos

- Una arquitectura basada en eventos consta de productores de eventos que generan un flujo de eventos, y consumidores de eventos que escuchan los eventos. Los eventos se entregan casi en tiempo real, de modo que los consumidores pueden responder inmediatamente a los eventos cuando se producen. Los productores se desconectan de los consumidores, no saben si los consumidores están escuchando. Los consumidores también se desconectan entre sí, y cada consumidor ve todos los eventos.
- Una arquitectura basada en eventos puede usar un modelo pub/sub o un modelo de flujo de eventos.
  - Pub/sub: la infraestructura de mensajería mantiene el seguimiento de las suscripciones. Cuando se publica un evento, se envía el evento a cada suscriptor. Después de que se consume un evento, no se puede reproducir y los nuevos suscriptores no ven el evento.
  - Streaming de eventos: los eventos se escriben en un registro. Los eventos siguen un orden estricto (dentro de una partición) y son duraderos. Los clientes no se suscriben al flujo, sino que un cliente puede leer desde cualquiera de sus partes y es responsable de avanzar su posición en el flujo. Por lo que un cliente puede unirse en cualquier momento y puede reproducir los eventos.

Arquitectura basada en eventos

- En el lado del consumidor, hay algunas variaciones comunes:
  - Procesamiento sencillo de eventos: Un evento desencadena inmediatamente una acción en el consumidor.
  - Procesamiento de eventos complejos: El consumidor procesa una serie de eventos, en busca de patrones en los datos de eventos.
  - Procesamiento de flujo de eventos: Los procesadores de flujos sirven para procesar o transformar el flujo. Puede haber varios procesadores de flujo para diferentes subsistemas de la aplicación.
- El origen de los eventos puede ser externo con respecto al sistema, como dispositivos físicos en una solución de IoT. En ese caso, el sistema debe ser capaz de ingerir los datos según el volumen y el rendimiento que requiere el origen de datos. En algunos sistemas, como IoT, los eventos se deben ingerir en volúmenes muy altos.
- En la práctica, es habitual tener varias instancias de un consumidor para evitar hacer que el consumidor se convierta en un único punto de error en el sistema. También podrían ser necesarias varias instancias para administrar el volumen y la frecuencia de los eventos.

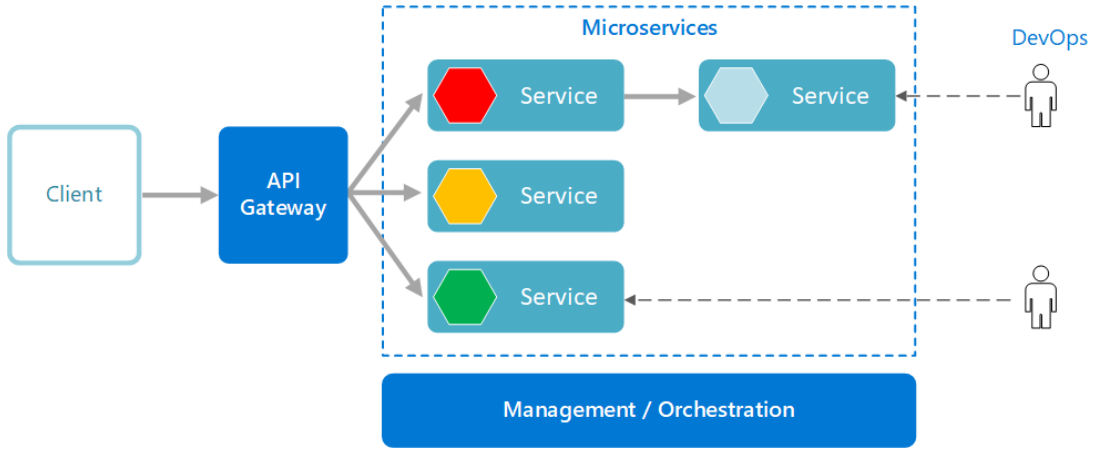
Arquitectura basada en eventos

- Esta arquitectura se debe utilizar para: Procesamiento de los mismos eventos en varios subsistemas , Procesamiento en tiempo real con retardo mínimo, Procesamiento de eventos complejos (como coincidencia de patrones o agregación durante ventanas de tiempo) y Procesamiento de un gran volumen y alta velocidad de datos (como IoT).
- Ventajas
  - Se desvinculan productores y consumidores.
  - No existen integraciones de punto a punto. Es fácil agregar nuevos consumidores al sistema.
  - Los consumidores pueden responder a eventos inmediatamente a medida que llegan.
  - Muy escalable y distribuida.
  - Los subsistemas tienen vistas independientes del flujo de eventos.
- Desafíos
  - Entrega garantizada: En algunos sistemas, especialmente en escenarios de IoT, es fundamental garantizar la entrega de los eventos.
  - Procesamiento de eventos en orden o exactamente solo una vez: Cada tipo de consumidor normalmente se ejecuta en varias instancias, a fin de conseguir resiliencia y escalabilidad. Esto puede suponer un desafío si se deben procesar los eventos en orden (dentro de un tipo de consumidor) o si la lógica de procesamiento no es idempotente.

Arquitectura de microservicios

- Una arquitectura de microservicios consta de una colección de servicios autónomos y pequeños. Los servicios son independientes entre sí y cada uno debe implementar una funcionalidad de negocio individual.
- Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa. Un único equipo reducido de programadores puede escribir y mantener un servicio.
- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
- Los servicios se comunican entre sí mediante APIs bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- No es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos de trabajo.
- El estilo de microservicio surge como alternativa al estilo monolítico.

Arquitectura de microservicios



Arquitectura de microservicios

- Además de los propios servicios, hay otros componentes que aparecen en una arquitectura típica de microservicios:
  - **Administración e implementación**: Este componente es el responsable de la colocación de servicios en los nodos, la identificación de errores, el reequilibrio de servicios entre nodos, etc. Normalmente, este componente es una tecnología estándar, como Kubernetes, en lugar de algo creado de forma personalizada.
  - **Puerta de enlace de API**: Es el punto de entrada para los clientes. En lugar de llamar a los servicios directamente, los clientes llaman a la puerta de enlace de API, que reenvía la llamada a los servicios apropiados en el back-end. Entre las ventajas de usar una puerta de enlace de API se encuentran las siguientes:
    - Desacoplan los clientes de los servicios. Los servicios pueden cambiar de versión o refactorizarse sin necesidad de actualizar todos los clientes.
    - Los servicios pueden utilizar los protocolos de mensajería que no son fáciles de usar para un servicio web, como AMQP.
    - La puerta de enlace de API puede realizar otras funciones transversales como la autenticación, el registro, la terminación SSL y el equilibrio de carga.

Ventajas de los microservicios

- **Agilidad**: Dado que microservicios se implementan de forma independiente, resulta más fácil de administrar las correcciones de errores y las versiones de características. Puede actualizar un servicio sin volver a implementar toda la aplicación y revertir una actualización si algo va mal.
- **Equipos pequeños y centrados**: Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo de características lo pueda compilar, probar e implementar. Los equipos pequeños favorecen la agilidad.
- **Base de código pequeña**: En las aplicaciones monolíticas, con el paso del tiempo se da la tendencia de que las dependencias del código se acaben enredarse, por lo que para agregar una nueva característica, es preciso tocar el código en muchos lugares. Al no compartir el código ni los almacenes de datos, la arquitectura de microservicios minimiza las dependencias y resulta más fácil agregar nuevas características.
- **Mezcla de tecnologías**: Los equipos pueden elegir la tecnología que mejor se adapte al servicio de una combinación de pilas de tecnología, según corresponda.
- **Aislamiento de defectos mejorado**: Si un microservicio individual no está disponible o genera problemas, no interrumpe toda la aplicación, siempre que los microservicios de nivel superior estén diseñados para controlar los errores correctamente (por ejemplo, circuit break).
- **Escalabilidad**: Los servicios se pueden escalar de forma independiente, lo que permite escalar horizontalmente los subsistemas que requieren más recursos, sin tener que escalar horizontalmente toda la aplicación. Mediante un orquestador como Kubernetes o Service Fabric se puede empaquetar una mayor densidad de servicios en un solo host, lo que aumenta la eficacia en el uso de los recursos.
- **Aislamiento de los datos**: Al afectar a un solo microservicio, facilita realizar actualizaciones del esquema.

Inconvenientes de los microservicios

- **Complejidad**: La arquitectura de microservicios es un estilo de programación distribuida. Cada servicio es más sencillo, pero el sistema como un todo es más complejo.
- **Desarrollo y pruebas**: Sobrecarga a los desarrolladores que deben implementar el mecanismo de comunicación entre servicios. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios. La refactorización en los límites del servicio puede resultar difícil. También supone un desafío probar las dependencias de los servicios, especialmente cuando la aplicación está evolucionando rápidamente. La prueba es más difícil, requiere un mayor peso en las pruebas de integración.
- **Falta de gobernanza**: El enfoque descentralizado para la generación de microservicios tiene ventajas, pero también puede causar problemas, se puede acabar con tantos lenguajes y marcos de trabajo diferentes que la aplicación puede ser difícil de mantener.
- **Congestión y latencia de red**: El uso de muchos servicios pequeños y detallados puede dar lugar a más comunicación interservicios. Además, si la cadena de dependencias del servicio se hace demasiado larga, la latencia adicional puede constituir un problema. Tendrá que diseñar las APIs con atención. Hay que evitar que las APIs se comuniquen demasiado, pensar en formatos de serialización y buscar lugares para utilizar patrones de comunicación asincrónica.
- **Integridad de datos**: Cada microservicio es responsable de la conservación de sus propios datos y, como consecuencia, mantener la coherencia de los datos es un problema. Implementar casos de uso que abarcan múltiples servicios sin usar transacciones distribuidas es difícil.
- **Administración**: El mayor consumo de recursos, la complejidad del despliegue y la complejidad operativa de implementar y administrar un sistema que comprende muchos tipos componentes y servicios diferentes requiere una cultura de DevOps consolidada. El registro correlacionado entre servicios puede resultar un desafío.
- **Control de versiones**: Las actualizaciones de un servicio no deben interrumpir servicios que dependen de él. Es posible que varios servicios se actualicen en cualquier momento; por lo tanto, sin un cuidadoso diseño, podrían surgir problemas con la compatibilidad con versiones anteriores o posteriores.
- **Conjunto de habilidades**: Cada microservicio requiere una implementación completa, incluyendo interfaz de usuario, almacenamiento persistente y colaboración externa. En consecuencia, los equipos deben ser interdisciplinares, incluyendo toda la gama de habilidades.

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 dos criterios.
- El primer criterio define si el protocolo 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: Cada solicitud debe ser procesada por un receptor o servicio exactamente (como en el patrón Command).
  - Varios receptores: 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).

Estilos arquitectónicos

- 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-SOAP
1999	Aparición de SOAP 1.0, WSDL, UDDI
2000	Definición del REST
2012	Propuesta de GraphQL por Facebook
2015	Desarrollo de gGRPC por Google

Estilos arquitectónicos

- Basados en Recursos: Los servicios REST / Hypermedia exponen documentos que incluyen tanto identificadores de datos como de acciones (enlaces y formularios). 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 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.

Servicios SOAP

- SOAP es un protocolo de comunicación web altamente estandarizado basado en formatos de tipo texto en XML. Publicado por Microsoft en 1999, un año después de XML-RPC, SOAP heredó mucho de él.
- Basado en operaciones, de tipos texto, en formato XML y comunicaciones síncronas.
- Ventajas:
  - Independiente del lenguaje y la plataforma: La funcionalidad incorporada para crear servicios basados en web permite a SOAP manejar las comunicaciones y hacer que las respuestas sean independientes del lenguaje y la plataforma.
  - Vinculado a una variedad de protocolos de transporte: SOAP es flexible en términos de protocolos de transferencia para adaptarse a múltiples escenarios.
  - Manejo de errores incorporado: La especificación de la API SOAP permite devolver el mensaje Retry XML con el código de error y su explicación.
  - Varias extensiones de seguridad: Integrado con los protocolos WS-Security, SOAP cumple con una calidad de transacción de nivel empresarial. Proporciona privacidad e integridad dentro de las transacciones al tiempo que permite el cifrado a nivel de mensaje.

Servicios SOAP

- Inconvenientes:
  - Solamente acepta formato XML: Los mensajes SOAP contienen una gran cantidad de metadatos y solo admiten estructuras XML detalladas para solicitudes y respuestas.
  - De peso pesado: Debido al gran tamaño de los archivos XML, los servicios SOAP requieren un gran ancho de banda hasta para la información mas nimia.
  - Conocimientos estrictamente especializados: La creación de servidores de API SOAP requiere un conocimiento profundo de todos los protocolos involucrados y sus reglas altamente restringidas o disponer de framework que simplifiquen su utilización que no están disponibles en todas las plataformas y lenguajes.
  - Actualización de mensajes tediosa: Al requerir un esfuerzo adicional para agregar o eliminar las propiedades del mensaje, el esquema SOAP rígido ralentiza la adopción.

Servicios REST

- REpresentational State Transfer es un estilo arquitectónico autoexplicativo basado en el uso del protocolo HTTP e hipermedia y establece un conjunto de restricciones arquitectónicas y destinado a una amplia adopción. Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba. RESTful hace referencia a un servicio web que implementa la arquitectura REST.
- Restringe la forma de usar de las URLs, los métodos (verbos) de HTTP, sus encabezados (Accept, Content-type, …) y sus códigos de estado.
- Basado en recursos, independiente de formato (texto o binario) y comunicaciones síncronas.
- Ventajas:
  - Soporte de múltiples formatos: La capacidad de admitir múltiples formatos (a menudo JSON y XML) para almacenar e intercambiar datos es una de las razones por las que REST es actualmente una opción predominante para crear APIs públicas.
  - Documentación mínima, basados en convenios y el descubrimiento hipermedia.
  - Mínima curva de aprendizaje: usa tecnologías ampliamente conocidas: HTTP, URL, MIME, …

Servicios REST

- Inconvenientes:
  - Uso de HTTP: Aunque es la infraestructura mas ampliamente difundida y utilizada, está restringido a ella.
  - Grandes cargas útiles: REST devuelve una gran cantidad de metadatos enriquecidos para que el cliente pueda comprender todo lo necesario sobre el estado de la aplicación solo a partir de sus respuestas.
  - Sin estructura REST única: No existe una forma exacta y correcta de crear una API REST. Cómo modelar los recursos y qué recursos modelar dependerá de cada escenario. Esto hace que REST sea simple en teoría, pero difícil en la práctica.
  - Problemas de recuperación excesiva o insuficiente: Devolver todo suele conllevar demasiados datos y el convenio no estable mecanismos de filtrado, paginación o proyecciones. El uso de hipermedia para obtener datos relacionados requiere solicitudes adicionales y encadenadas.
  - Convenio genérico: En muchos casos no acorde con las reglas de negocio (create o replace, delete, …) lo que acaba requiriendo documentación.

Servicios GraphQL

- GraphQL es una sintaxis que describe cómo obtener la descripción del modelo de datos y cómo realizar una solicitud de datos precisa, consultas y mutaciones, pensada para los modelos de datos con muchas entidades complejas que hacen referencia entre sí. Fue propuesta por Facebook en 2012, publicada en 2015 y posteriormente pasada a open source en 2018.
- Basado en consultas, en formato JSON y comunicaciones síncronas.
- Ventajas:
  - Un esquema de GraphQL establece una fuente única de información, ofrece una forma de unificar todo en un único servicio.
  - Las llamadas a GraphQL se gestionan mediante HTTP POST en un solo recorrido de ida y vuelta. Los clientes obtienen lo que solicitan sin que se genere una sobrecarga.
  - Los tipos de datos bien definidos reducen los problemas de comunicación entre el cliente y el servidor. Un cliente puede solicitar una lista de los tipos de datos disponibles y esto es ideal para la generación automática de documentación.
  - GraphQL permite que las APIs de las aplicaciones evolucionen sin afectar a las consultas actuales.
  - GraphQL no exige una arquitectura de aplicación específica. Puede incorporarse sobre una API de REST actual y funcionar con las herramientas de gestión de APIs actuales. Hay muchas extensiones open source de GraphQL que ofrecen características que no están disponibles con las APsI de REST.

Servicios GraphQL

- Inconvenientes:
  - GraphQL intercambia complejidad por flexibilidad. Tener demasiados campos anidados en una solicitud puede provocar una sobrecarga del sistema. Además, delega gran parte del trabajo de las consultas de datos en el servidor, lo cual representa una mayor complejidad para los desarrolladores de servidores.
  - GraphQL no indica cómo almacenar los datos ni qué lenguaje de programación utilizar, requiere disponer de framework que simplifiquen su utilización que no están disponibles en todas las plataformas.
  - Como GraphQL no utiliza la semántica de almacenamiento en caché HTTP, requiere un esfuerzo de almacenamiento en caché personalizado.
  - GraphQL presenta una curva de aprendizaje elevada para desarrolladores que tienen experiencia con las APIs de REST.

Servicios gRPC

- gRPC es un marco de llamada a procedimiento remoto (RPC) de alto rendimiento e independiente de lenguaje. Desarrollado por Google en el año 2015, y luego convertido en código abierto. Como GraphQL, es una especificación que se implementa en una variedad de lenguajes.
- Basado en contratos, en el formato binario Protocol Buffers y comunicaciones síncronas/asíncronas.
- Las principales ventajas de gRPC son:
  - Marco de RPC moderno, ligero y de alto rendimiento.
  - La especificación gRPC es preceptiva con respecto al formato que debe seguir un servicio gRPC, formaliza un contrato independiente de lenguaje.
  - Es compatible con el intercambio de datos bidireccional y asíncrono al estar basado en HTTP/2, así como con la compresión, multiplexación y streaming.
  - Uso reducido de red al usar un formato de mensaje binario eficaz que genera cargas de mensajes pequeñas.

Servicios gRPC

- Inconvenientes:
  - Tanto el cliente como el servidor deben admitir la misma especificación de búfer de protocolo, requiere un estricto control de versiones. gRPC es un formato muy particular que proporciona una ejecución ultrarrápida a expensas de la flexibilidad.
  - Basado en HTTP/2 que no tiene soporte universal para interacciones cliente-servidor de cara al público general en Internet y una compatibilidad limitada con exploradores.
  - gRPC es una especificación, por lo que requiere disponer de framework que permitan su utilización tanto en el cliente como en le servidor y que no están disponibles en todas las plataformas y lenguajes.
  - El proceso de serialización/deserialización para su conversión a binario es costoso en términos de CPU.
  - gRPC presenta una curva de aprendizaje mas empinada que la de REST.

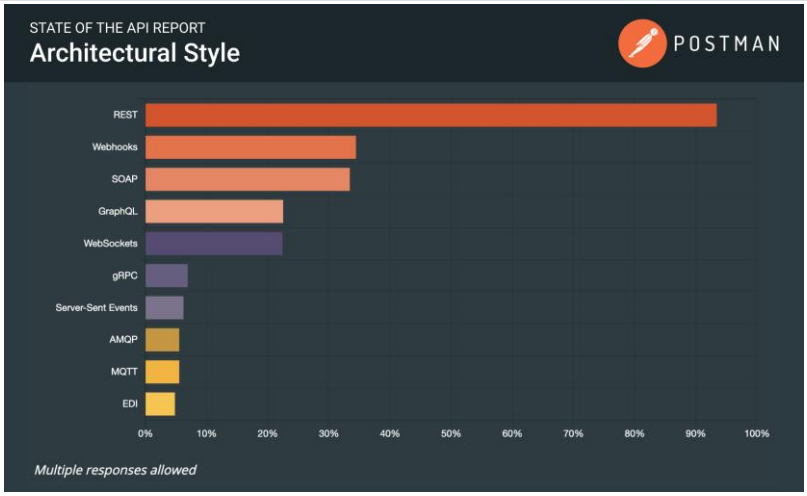
Tecnología de clientes web (Navegadores)

- JavaScript Asíncrono + XML (AJAX) no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: HTML o XHTML, CSS, JavaScript, DOM, XML, XSLT, y lo más importante, el objeto XMLHttpRequest (fetch). Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa.
- WebSockets es una tecnología avanzada que hace posible abrir una sesión de comunicación bidireccional entre el navegador y un servidor: se puede enviar mensajes a un servidor y recibir respuestas controladas por eventos sin tener que consultar al servidor para una respuesta.
- Server Sent Events (SSE) es una tecnología basada en streaming HTTP que pretenden estandarizar la comunicación COMET (long polling) en los navegadores cuyo flujo de comunicación sólo va desde el servidor hacia el cliente (no es bidireccional como los WebSockets). La idea consiste en que el cliente crea una conexión con el servidor una sola vez y este le va enviando información (eventos) al cliente cuando hay nuevos datos.
- ASP.NET Signalr es una biblioteca para desarrolladores de ASP.NET que simplifica el proceso de agregar funcionalidad web en tiempo real: la posibilidad de que el código de servidor inserte el contenido en los clientes conectados al instante a medida que esté disponible sin espera a nuevas solicitudes. Signalr usa WebSocket cuando está disponible o recurre las otras cuando es necesario.

WebHooks

- Los webhooks son eventos que desencadenan acciones. Su nombre se debe a que funcionan como «ganchos» 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.
- Un webhook es una retro llamada HTTP, una solicitud HTTP POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento 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 POST).

Estilos arquitectónicos mas utilizados



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

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.

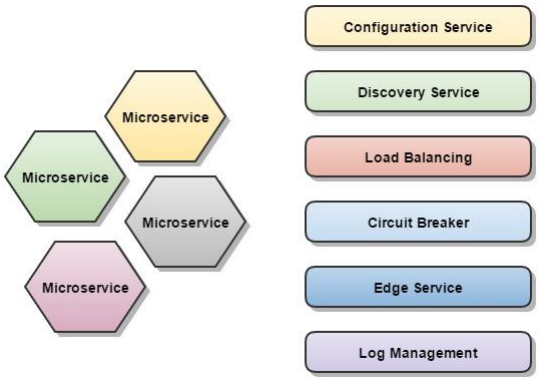
Preocupaciones transversales

- 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

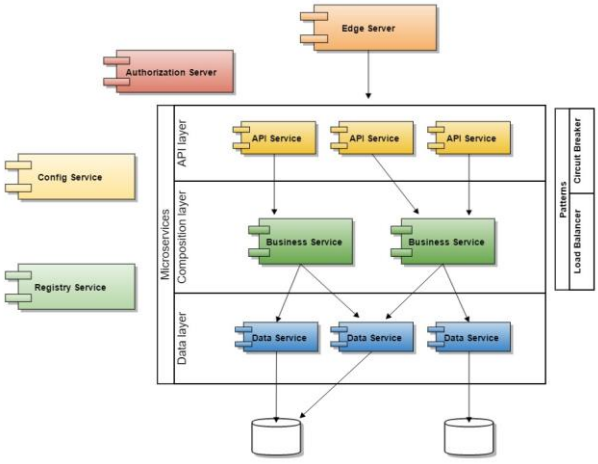
- Servidor perimetral / exposición de servicios (Edge server)
  - Será un gateway en el que se expondrán los servicios a consumir.
- Servicio de registro / descubrimiento
  - Este servicio centralizado será el encargado de proveer los endpoints de los servicios para su consumo. Todo microservicio, en 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

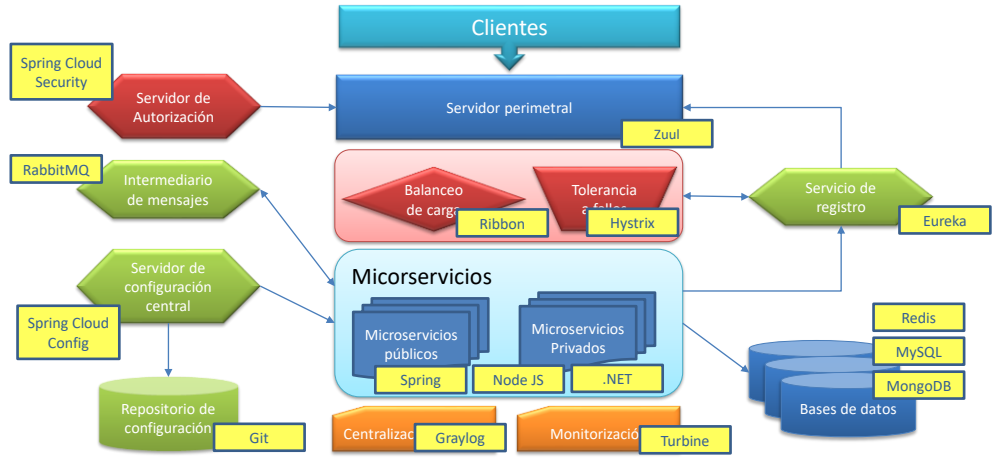
Modelo de referencia



Modelo de implementación (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 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.

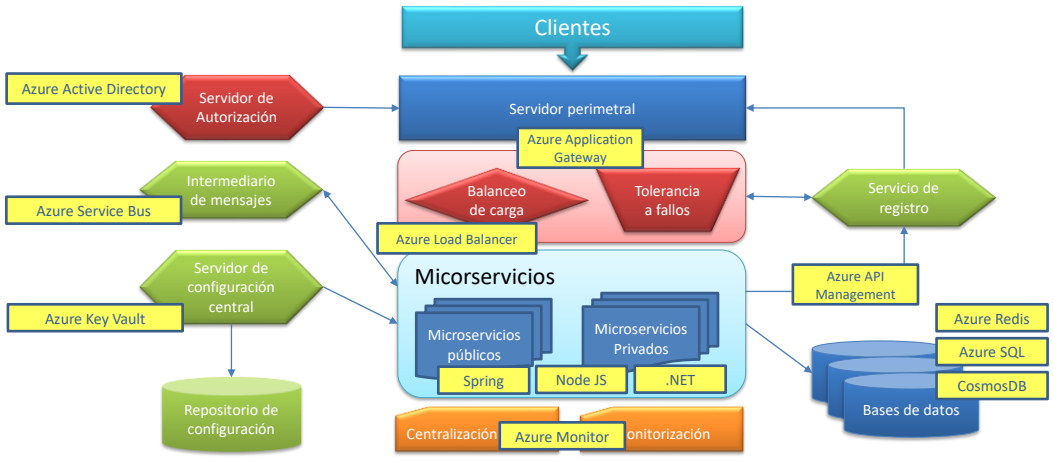
Modelo de implementación (Spring Cloud)



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)



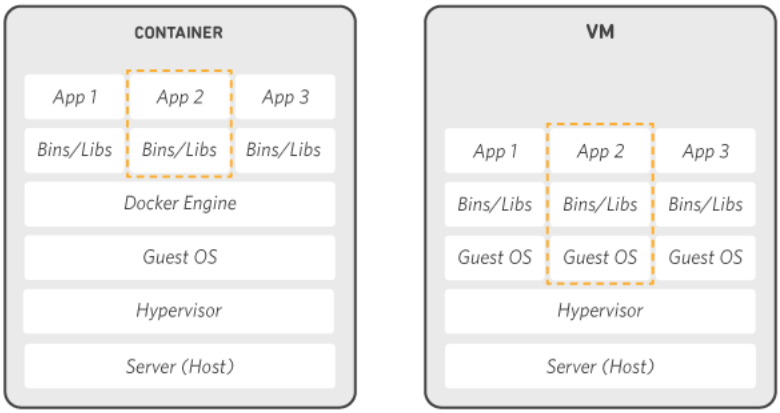
Modelo de despliegue

- 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 dos patrones en este sentido a la hora de encapsular microservicios:
  - Máquinas virtuales.
  - Contenedores.
- 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.

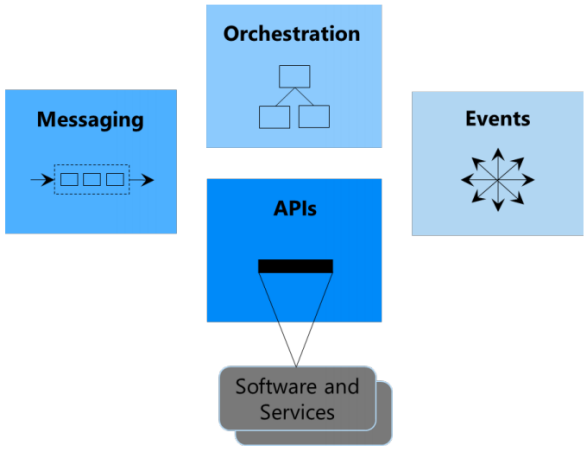
FaaS (Functions-as-a-Service)

- 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).

Arquitectura de Integración

- Ninguna aplicación es una isla. Para aprovechar al máximo el software que se crea o compra, es necesario conectarlo a otro software. Esto significa que la integración de aplicaciones efectiva es esencial para casi todas las organizaciones.
- A veces, todo lo que se necesita hacer es conectar una aplicación directamente a otra. Sin embargo, más a menudo, la integración de aplicaciones significa conectar múltiples sistemas independientes, a menudo de formas complejas. Es por ello que las organizaciones comúnmente confían en plataformas de integración especializadas que brindan los servicios necesarios para hacer esto.
- Como tantas otras cosas hoy en día, esas plataformas se han trasladado a la nube. En lugar de utilizar tecnologías de integración tradicionales como BizTalk Server, cada vez más organizaciones utilizan soluciones de plataforma como servicio de integración (iPaaS), es decir, plataformas de integración basadas en la nube.
- Para satisfacer esta necesidad, Microsoft proporciona servicios de integración de Azure. Esta solución iPaaS es un conjunto de servicios en la nube para la integración empresarial de misión crítica. Para lograr este objetivo, los servicios proporcionan las cuatro tecnologías básicas necesarias para la integración basada en la nube.

Plataforma como servicio de integración (iPaaS)



Plataforma como servicio de integración (iPaaS)

- Los cuatro principales servicios en la nube necesarios para la integración en la actualidad son:
  - Una forma para publicar y administrar interfaces de programación de aplicaciones (APIs): Este servicio de APIs hace que los servicios de software sean accesibles para otro software, se ejecuten en la nube o en local.
  - Una forma sencilla de crear y ejecutar la lógica de integración: Habitualmente es necesario implementar procesos que se basen en varias aplicaciones diferentes, a las que se accede a través de API. Para crear este tipo de flujo de trabajo, una iPaaS debe proporcionar mecanismos de orquestación, generalmente con una herramienta gráfica para definir la lógica del flujo de trabajo.
  - Alguna manera para que las aplicaciones y las tecnologías de integración se comuniquen de forma poco acoplada a través de mensajes: Este servicio proporciona colas que retienen los mensajes hasta que el receptor los puede recoger. Esto permite que las aplicaciones y el software de integración se comuniquen de forma asincróna, incluso a través de diversas plataformas tecnológicas, algo que a menudo lo requieren los escenarios de integración.
  - Una tecnología que apoye la comunicación a través de eventos: En lugar de sondear una cola en un servicio de mensajería, a veces es más fácil y más eficiente conocer los cambios al recibir un evento.

- Estos servicios en la nube, a veces combinados con otras tecnologías en la nube, se pueden utilizar para integrar aplicaciones tanto en la nube como locales.

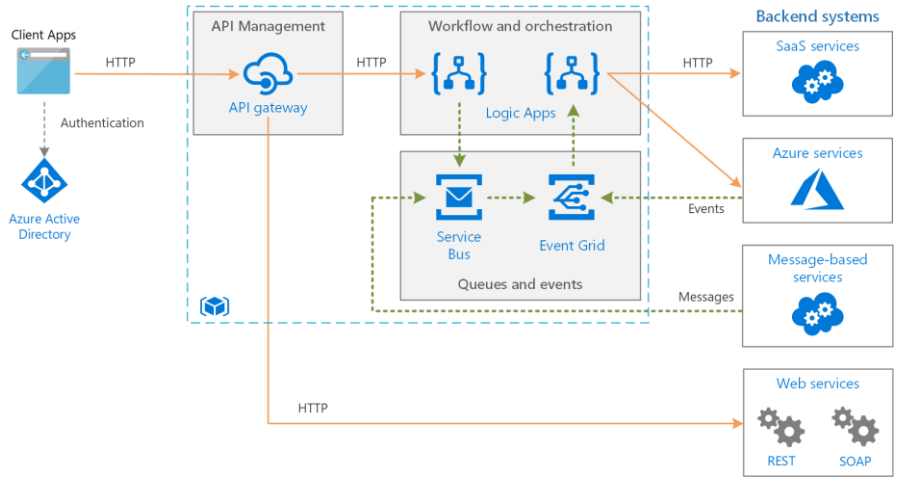
Escenarios comunes

- Conexión de aplicaciones dentro de la organización.
  - Las aplicaciones pueden ejecutarse en su propio centro de datos local, en la nube o en una combinación de ambos. Este tipo de integración de aplicaciones empresariales (EAI) ha sido importante durante décadas; ahora está siendo adaptado para un mundo híbrido.
- Conexión de aplicaciones de la organización con las de un socio comercial.
  - Comúnmente conocida como integración de empresa a empresa (B2B), a menudo se basa en formatos estándar como el intercambio electrónico de datos (EDI).
- Conexión de aplicaciones de la organización a aplicaciones de software como servicio (SaaS).
  - A medida que las aplicaciones comerciales adquiridas van cambiando a SaaS, la integración de aplicaciones requiere conectarse con sus servicios en la nube.
- Integración de aplicaciones con dispositivos de Internet de las cosas (IoT).
  - La creciente popularidad de IoT plantea una variedad de nuevos desafíos de integración. Una solución basada en la nube (iPaaS) es especialmente adecuada para abordar estos problemas, ya que se puede acceder a ella mediante dispositivos que se ejecutan en cualquier lugar y aseguran la escalabilidad necesaria.

Azure Integration Services

- Azure Integration Services es la plataforma iPaas de Microsoft. Los cuatro componentes de Azure Integration Services en la actualidad son:
  - API Management, que proporciona un servicio para publicar y administrar APIs.
  - Logic Apps, que admiten la orquestación de procesos de negocio, flujos de trabajo y demás.
  - Service Bus, que proporciona mensajería empresarial confiable.
  - Event Grid , que permite generar y entregar eventos.
- Estos servicios son individuales e independientes, se puede usar uno o varios según sea necesario. Estos servicios son parte de la oferta más grande de Azure, que brinda acceso inmediato e integrado a cualquier otro servicio en la nube que se necesite.
- La premisa de la nube es: usas lo que necesitas y pagas por lo que usas.

Azure Integration Services



Azure API Management

- Las aplicaciones modernas suelen exponer al menos parte de su funcionalidad a través de APIs. Esto permite que la lógica de integración acceda a la funcionalidad de una aplicación a través de sus APIs. Por ejemplo, una orquestación de flujo de trabajo puede implementar un proceso empresarial completo invocando diferentes APIs de diferentes aplicaciones, cada una de las cuales lleva a cabo una parte de ese proceso.
- Sin embargo, hacer que el conjunto creciente de las APIs estén disponibles para que otro software las llame no es tarea fácil y acaba requiriendo administración para tratar aspectos de:
  - ¿Cómo controlar cómo se autentican las llamadas a las API de la aplicación? La seguridad siempre es importante, especialmente para las aplicaciones accesibles a través de la Internet pública.
  - ¿Cómo se limita la cantidad de llamadas que la aplicación puede recibir de los clientes? La mayoría de las aplicaciones que exponen APIs no pueden manejar un número ilimitado de solicitudes, por lo que se necesita una forma de controlar esto
  - ¿Cómo se asegura de que las API sean lo suficientemente rápidas? Es posible que se desee almacenar en caché las respuestas a solicitudes frecuentes, por ejemplo, para acelerar las respuestas.
  - ¿Cómo supervisar y analizar cómo se utilizan las APIs? Los patrones en el uso de una API pueden indicar una tendencia que impacta significativamente en el negocio.
  - ¿Cómo facilitar a los desarrolladores el uso de las APIs? Se necesita una forma de proporcionarles documentación, ejemplos de código y demás.

Azure API Management

- Azure API Management aborda todos estos aspectos.
- API Management puede exponer las API REST y SOAP de todo tipo de software de backend, incluidas las aplicaciones que se ejecutan en la nube y en local. La pila de tecnología utilizada para construir ese software (.NET, Java o cualquier otra cosa) es irrelevante; API Management funciona con cualquier cosa. Incluso otros servicios de Azure pueden exponerse a través de API Management, lo que permite colocar una API administrada encima de Service Bus, Logic Apps y otros servicios si así se desea.
- Para hacer esto, API Management crea fachadas (proxies) de las API reales del backend. Los clientes, incluidas las aplicaciones lógicas, las aplicaciones en la nube y locales, y otras, pueden llamar a dichas fachadas. Todo lo que se requiere es que el cliente pueda crear y recibir solicitudes HTTP. Cada llamada se enruta finalmente a la aplicación subyacente para que genere una respuesta, pero API Management puede manipular la petición antes de enrutarla y posteriormente el resultado antes de devolverlo.
- API Management suministra un portal administrativo para publicar, supervisar y especificar los detalles necesarios de las APIs, un API Gateway para enrutar las peticiones y un portal para los desarrolladores que consumen las APIs.

Azure Service Bus

- La esencia de la integración de aplicaciones es que el software se comunica con otro software. El estilo de comunicación síncrono permite llamadas directas a través de API Management. En otros casos, si ambas aplicaciones no están disponibles al mismo tiempo, se requiere un enfoque asíncrono.
- Service Bus permite que las aplicaciones intercambien mensajes a través de colas y las interacciones sin bloqueo entre diferentes partes de software. Service Bus proporciona mensajería empresarial confiable entre muchos tipos de software, incluidas aplicaciones en la nube, aplicaciones locales y servicios de Azure.
- Dispone de:
  - Semántica de la cola, incluida la persistencia de mensajes y el orden estricto de los mensajes de primero en entrar, primero en salir. El servicio también detecta y elimina mensajes duplicados.
  - Transacciones atómicas, permitiendo que una cola de lectura o escritura sea parte de una operación más grande que tiene éxito o falla como una sola unidad.
  - Manejo de mensajes envenenados, por lo que un mensaje que causa problemas no pondrá al receptor en un bucle infinito.
  - Alta disponibilidad, incluida la replicación geográfica, junto con la recuperación ante desastres incorporada.
- En resumen, Service Bus proporciona todas las funciones necesarias para la mensajería empresarial.

Azure Event Grid

- La comunicación a través de mensajes exige que el software receptor compruebe periódicamente si ha llegado un nuevo mensaje, comúnmente conocido como sondeo. En determinados escenarios, esto puede ser un desperdicio.
- Event Grid permite que el receptor sea notificado a través de un evento: el receptor registra un controlador de eventos para la fuente de eventos que le interesa y Event Grid invoca ese controlador cuando ocurre el evento especificado.
- Para recibir un evento de un servicio de Azure, el receptor se suscribe a un tema estándar proporcionado para ese servicio. Las aplicaciones en la nube y locales también pueden crear temas personalizados, lo que permite que los servicios de Azure y otro software reciban eventos personalizados al suscribirse a estos temas.
- Event Grid se puede ver como una simplificación del Service Bus, que proporciona solo una forma simple y rápida de enviar eventos, para que sea significativamente más escalable que Service Bus (admite hasta 10,000,000 de eventos por segundo en una sola región de Azure), aunque para lograr esto tenga que entregar eventos fuera de orden. Aunque Service Bus es rápido, Event Grid proporciona un rendimiento casi en tiempo real, con el 99% de los eventos entregados en menos de un segundo.

Mensajes vs Eventos

- Aunque están estrechamente relacionados, ambos son datagramas (paquetes de datos que se envían de un componente a otro) y a menudo utilizan la misma arquitectura, existen diferencias fundamentales entre los dos.
- Un mensaje es un elemento de datos que se envía a un destino específico. Un evento es una señal emitida por un componente al cambiar a un estado determinado.
- En un sistema dirigido por mensajes, los destinatarios direccionables esperan la llegada de los mensajes y reaccionan a ellos, mientras tanto permanecen inactivos. En un sistema de notificación controlado por eventos, los oyentes se suscriben a las fuentes de los eventos de manera que se invocan cuando se emite el evento.
- Esto significa que un sistema impulsado por eventos se enfoca en fuentes de eventos direccionables mientras que un sistema impulsado por mensajes se concentra en destinatarios direccionables. Un mensaje puede contener un evento codificado como su carga útil.
- Si el productor sabe quién es el destinatario previsto, debe confirmar que se entrega la información o el comando y es probable que desee que se produzca algún tipo de respuesta o acción, entonces es un mensaje.
- Un evento es algo que sucede y el servicio donde ocurre lo publica en un flujo de eventos, independientemente de las acciones que ocurran después de eso (si las hubiera). Otros servicios que estén interesados en ese tipo de eventos pueden suscribirse para recibirlos. Puede haber cualquier número de suscriptores que recibirán cada evento, incluido cero.
- En otras palabras, en un sistema impulsado por mensajes, el emisor conoce a los destinatarios previstos, mientras que en un sistema impulsado por eventos el receptor decide a qué fuentes de eventos desea suscribirse.

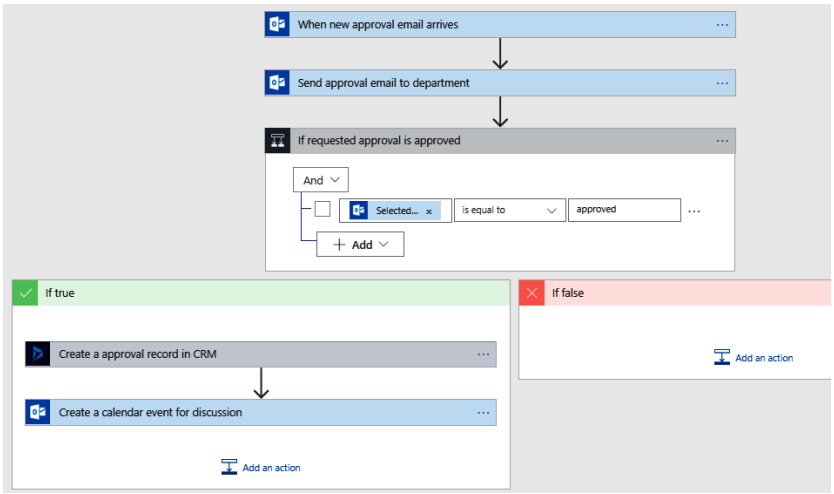
Azure Logic Apps

- La integración de aplicaciones normalmente requiere implementar todo o parte de un proceso de negocio: un proceso puede acceder a una aplicación en la nube como Salesforce CRM, actualizar los datos locales almacenados en las bases de datos de SQL Server y Oracle e invocar operaciones en una aplicación local. Hacer esto significa construir una lógica personalizada de orquestación que lleve a cabo los pasos de este proceso.
- La opción de construirlo de la forma tradicional, usando C#, JavaScript, Java o algún otro lenguaje de programación, puede ser costosa sobre todo si cuenta con partes asíncronas.
- Logic Apps permite crear la lógica utilizando una tecnología de flujo de trabajo, con acciones predefinidas, que permiten implementar los procesos sin requerir programación.
- Una aplicación lógica consta de una serie de acciones, cada una de las cuales es un paso lógico en el proceso que implementa el flujo de trabajo. Estas acciones pueden expresar condicionales, bucles y demás controles de flujo. También se pueden utilizar acciones para llamar a software y servicios externos, Logic Apps proporciona más de 200 conectores para facilitar el trabajo con aplicaciones comunes, fuentes de datos y otros servicios, cada conector proporciona una forma sencilla para interactuar con un servicio externo específico. Una “aplicación lógica” puede invocar las APIs expuestas a través de API Management, incluso puede exponerse en sí misma como una API a través del mismo.

Azure Logic Apps

- La creación de una “aplicación lógica” normalmente no requiere escribir ningún código: los desarrolladores utilizan Logic Apps Designer, una herramienta visual, para crear los nuevos procesos.
- El punto clave es la falta de necesidad de escribir código, lo que permite a los desarrolladores implementar procesos de negocio de forma rápida y sencilla, libre de errores de codificación.
- En realidad, una “aplicación lógicas” es un archivo de texto en JavaScript Object Notation (JSON), esto ayuda a que las aplicaciones lógicas encajen bien en los entornos DevOps e incluso es posible crear un nuevo flujo de trabajo directamente en JSON.
- Logic Apps es una tecnología sin servidor. Si bien cada “aplicación lógica” se ejecuta en última instancia en alguna máquina virtual, todo esto está oculto a los desarrolladores. No necesitan pensar en la infraestructura o la escalabilidad; ya está resuelto. Esto los libera para centrarse únicamente en crear una lógica de integración.
- Azure Logic Apps es el núcleo de los procesos de integración, siendo el resto de los servicios facilitadores de dicho proceso.

Azure Logic Apps



Azure Data Factory

- Azure Integration Services esta enfocado a la integración de procesos y aplicaciones.
- Si bien puede trabajar con datos e implementar un extraer/transformar/cargar (ETL) en la nube con Azure SQL Data Warehouse o extraer, cargar y procesar (ELT) big data no estructurado con Azure Data Lake Storage, no es necesariamente la mejor opción. Microsoft proporciona Azure Data Factory, un servicio en la nube para la integración de datos.
- Azure Data Factory es el servicio ETL/ELT en la nube de Azure para la integración y transformación de datos sin servidor de escalabilidad horizontal. Ofrece una interfaz de usuario sin código que favorece la creación intuitiva, así como la supervisión y administración desde un único panel.
- También se puede realizar la migración mediante lift-and-shift de los paquetes de SSIS existentes a Azure y ejecutarlos con plena compatibilidad en ADF. SSIS Integration Runtime ofrece un servicio totalmente administrado, por lo que no hay que preocuparse de la administración de la infraestructura.

- Los diferentes tipos de integración requieren diferentes tecnologías de integración.

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.

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 exito.
- 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.

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.

Organización de equipos

- Cuando se busca dividir un sistema grande en partes, a menudo la administración de equipos de trabajo se centra en la capa tecnológica, lo que lleva a tener equipos de interfaz de usuario, equipos de lógica de servidor y equipos de bases de datos. Cuando los equipos están separados de esta amanera, incluso los cambios mas simples pueden conducir a proyectos cruzados entre equipos que suponen tiempo y coste.
- La estrategia API es diferente, dividiendo los equipos por servicios organizados alrededor de la capacidad empresarial.
- Cada servicio requiere una implementación completa de software, incluyendo interfaz de usuario, almacenamiento persistente y colaboración externa.
- En consecuencia, los equipos son interdisciplinares, incluyendo toda la gama de habilidades necesarias para el desarrollo: experiencia de usuario, base de datos y gestión de proyectos.

Productos y Gobernanza

- **Productos no Proyectos**
  - La mayoría de los esfuerzos de desarrollo de aplicaciones que vemos utilizan un modelo de proyecto: donde el objetivo es entregar algún software que se considera completado.
  - Al terminar el software se entrega a una organización de mantenimiento y el equipo de proyecto que lo construyó se disuelve.
  - La estrategia API tienden a evitar este modelo, prefiriendo en cambio la noción de que un equipo debe poseer un producto durante toda su vida útil.
- **Gobernanza descentralizada**
  - Una de las consecuencias de la gobernanza centralizada es la tendencia a estandarizar las plataformas con tecnología única. La experiencia demuestra que este enfoque es limitante.
  - La estrategia API debe permitir usar la herramienta adecuada para cada caso pero manteniendo unas directrices comunes y establecer una cultura anti-burocracia: libertad con responsabilidad.

Puntos finales inteligentes y conexiones tontas

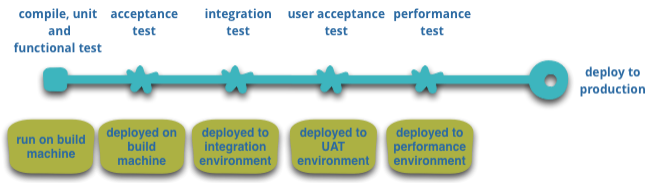
- Al construir estructuras de comunicación entre diferentes procesos, hemos visto muchos productos y enfoques que ponen énfasis en el mecanismo de comunicación.
- Un buen ejemplo de esto es el Enterprise Service Bus (ESB), donde los productos de ESB a menudo incluyen sofisticadas instalaciones para enrutamiento de mensajes, coordinación, transformación y aplicación de reglas de negocio.
- La estrategia API favorece un enfoque alternativo: **puntos finales inteligentes y conexiones tontas**.
- Las aplicaciones construidas a partir de APIs tienen como objetivo estar tan desacopladas y cohesivas como sea posible (poseen su propia lógica de dominio y actúan más como filtros) recibiendo una petición, aplicando la lógica según corresponda y produciendo una respuesta.
- Estos son coordinados utilizando simples protocolos REST (HTTP/HTTPS, WebSockets o AMQP) en lugar de complejos protocolos como WS o BPEL u orquestación de una herramienta central.

Gestión descentralizada de datos

- La descentralización de la gestión de datos se presenta de diferentes maneras.
- En el nivel más abstracto, significa que el modelo conceptual diferirá entre sistemas. Este es un problema común cuando se necesita hacer integración en una gran empresa.
- Una forma útil de pensar sobre esto es la noción de Contexto delimitado del Diseño Dirigido por Dominio (DDD). DDD divide un dominio complejo en múltiples contextos acotados y mapea las relaciones entre ellos.
- Antiguamente se recomendaba construir un modelo unificado de toda la empresa, pero hemos aprendido que “la unificación total del modelo de dominio para un sistema grande no será factible ni rentable“.
- Dicha unificación a conllevado la preponderancia del modelo relacional frente a modelos no-sql mas apropiados en determinados escenarios.
- Este proceso es útil tanto para arquitecturas monolíticas como para una estrategia API.
- La estrategia API prefieren dejar que cada servicio administre su propia fuente de datos, ya sea diferentes instancias de la misma tecnología de base de datos, o sistemas de base de datos completamente diferentes, un enfoque llamado Polyglot Persistence.

Automatización de Infraestructura

- Las técnicas de automatización de la infraestructura han evolucionado enormemente en los últimos años: la evolución de la nube y de AWS o Azure en particular ha reducido la complejidad operativa de la creación, implementación y operación de las APIs.
- Muchos de los productos o sistemas que se están construyendo con APIs están siendo construidos por equipos con amplia experiencia en Entrega Continua y su precursor, la Integración Continua. Los equipos que crean software de esta manera hacen un uso extensivo de las técnicas de automatización de infraestructura.



- Para realizar el proceso con garantías:
  - Requiere exhaustivas pruebas automatizadas.
  - La promoción del software en funcionamiento "hacia arriba" implica automatizar la implementación en cada nuevo entorno.

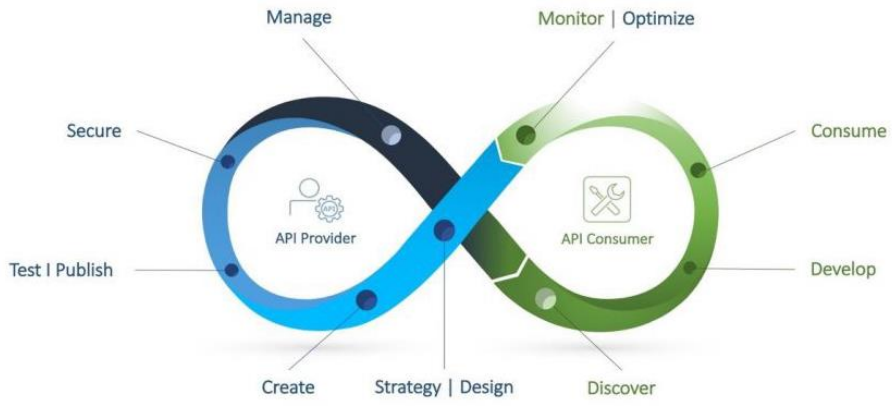
Diseño tolerante a fallos

- Una consecuencia del uso de APIs como componentes distribuidos, es que las aplicaciones deben diseñarse de manera que puedan tolerar el fallo de los servicios. Cualquier llamada a un servicio podría fallar debido a la falta de disponibilidad del proveedor.
- Esto es una desventaja en comparación con un diseño monolítico ya que introduce complejidad adicional para manejarlo.
- Dado que los servicios pueden fallar en cualquier momento, es importante poder detectar los fallos rápidamente y restaurar automáticamente el servicio si es posible.
- La estrategia API debe poner mucho énfasis en el monitoreo en tiempo real de la aplicación, comprobando los elementos arquitectónicos (cuántas solicitudes por segundo tiene la base de datos) y métricas relevantes para el negocio (por ejemplo, cuántas órdenes por minuto se reciben).
- El monitoreo semántico puede proporcionar un sistema de alerta temprana de algo que va mal, lo que indicará a los equipos de desarrollo que deben investigarlo.

Diseño Evolutivo

- La implementación de componentes en APIs añade una oportunidad para una planificación de entrega más granular.
- Con un monolito, cualquier cambio requiere una compilación completa y despliegue de toda la aplicación.
- Con la estrategia API, sin embargo, sólo es necesario volver a implementar el servicio que se modificó. La propiedad clave de un componente es la noción de reemplazo y la capacidad de actualización independientes, lo que implica que buscamos puntos en los que podamos imaginar reescribir un componente sin afectar a sus colaboradores.
- Esto puede simplificar y acelerar el proceso de entrega.
- En una estrategia API, solo se necesita volver a implementar los servicios que se modificaron. Esto puede simplificar y acelerar el proceso de lanzamiento aunque el inconveniente es que hay que preocuparse por si los cambios en un servicio rompan a sus consumidores.
- El enfoque de integración tradicional es tratar de abordar este problema utilizando el control de versiones, pero la preferencia en las estrategias API es utilizar sólo el versionado como último recurso: deberemos diseñar los servicios para que sean lo más tolerantes posible a los cambios en sus proveedores.

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

API GOVERNANCE

Introducción

- La proliferación de APIs con sus correspondientes equipos de desarrollo, así como su uso externo que afecta a la imagen de la organización, impone la necesidad de una gobernanza.
- El gobierno de APIs es como cualquier otra área de TI en la cual hay una necesidad de entender el uso de recursos, gestionar reglas y definir o validar políticas de seguridad. Debe identificar y gestionar las buenas prácticas de diseño, documentación, seguridad, pruebas de calidad y control de todos los aspectos de su ciclo de vida.
- La naturaleza propia de las APIs requiere una gobernanza descentralizada o adaptativa, mas enfocada a la coordinación que a la dirección. Existen pequeños equipos que tienen autonomía para evaluar y decidir sobre ciertos aspectos arquitectónicos, pero que siguen unas directrices comunes. Se deben aplicar metodologías agiles y TDD, así como establecer una cultura anti-burocracia: libertad con responsabilidad.

Introducción

- A nivel global hay que establecer objetivos comunes, marcando plazos y usando métricas para controlar su cumplimiento.
- Hay que establecer un marco común donde:
  - Definir procesos y roles, que hagan que todo el mundo tenga claro cuales son sus responsabilidades y sus obligaciones.
  - Definir y validar las políticas de seguridad.
  - Definir y mejorar las metodologías de trabajo de manera constante.
  - Establecer estándares y buenas prácticas dentro de la organización y como formar a todos los involucrados en estos estándares.
  - Establecer los criterios generales de calidad, supervisión y control.
- Con la llegada de los API Management, la gobernanza se puede centralizar en una única herramienta donde establecer las diferentes directivas, que ponga en contacto los diferentes equipos tanto internos como externos, permita supervisar los diferentes criterios, …

Introducción

- Los principales aspectos que debe establecer un marco común son:
  - Definición de recursos
  - Políticas de versionado
  - Seguridad
  - Estándares de definición e implementación
  - Documentación
  - Monitorización
  - Testing
  - Monetización
  - Gestión de los entornos

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).

Documentación

- Las definiciones del recurso se deben formalizar mediante uno de los mecanismos de definición estándar de API: OpenApi, WADL, RAML, …
- La documentación actúa como contrato, obliga a ambas partes.
- Reduce el riesgo al posibilitar la retroalimentación con el consumidor, reduciendo los fallos al facilitar las pruebas para garantizar que las APIs sean confiables, consistentes y fáciles de usar para los desarrolladores.
- 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.
- La inversión en la documentación formal se preserva, dado que gran parte del proceso de creación de la API se puede automatizar mediante herramientas que permiten importar archivos de definición de API y generar el esqueleto del backend y del cliente frontend, así como un mocking server para las pruebas.
- Hay que establecer claramente las responsabilidades y los mecanismos de publicación de la documentación, su versionado, el acceso, …

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 sistema SEMVER (Semantic Versioning), ampliamente adoptado, es un conjunto de reglas para proporcionar un significado claro y definido a las versiones del software.
- La versión SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z, donde:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo, cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.
- Las nuevas versiones del código de las APIs no tienen por que implicar nuevas versiones de las APIs, solo los cambios rupturistas (major) deberían ser reflejados.

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.

Seguridad

- Para reducir riesgos, desde el punto de vista de la gobernanza, hay que priorizar la seguridad al invertir los recursos disponibles, tanto económicos como humanos, así como el tiempo necesario. Las restricciones en esos recursos también afectan a la implementación de la seguridad en toda la organización.
- Hay que establecer directivas organizativas para las operaciones, tecnologías y configuraciones basadas en factores tanto internos (requisitos empresariales, riesgos y evaluación de recursos) como externos (pruebas comparativas, estándares normativos, entorno de amenazas).
- Las directivas deben cubrir estrategias de segmentación, requisitos normativos, administración de identidades y acceso, protección de datos, supervisión y auditorías.
- Hay que identificar los servicios que tienen un alto impacto potencial o una alta exposición potencial al riesgo y clasificarlos: con datos críticos, con datos regulados, de disponibilidad crítica, con altos privilegios, con alta exposición a ataques …
- La protección de un servicio o aplicación requiere garantías de seguridad para tres aspectos: código de aplicación, servicios utilizados y plataforma de hospedaje.

Seguridad

- Si bien la seguridad es critica y cubre múltiples aspectos, nos centraremos en lo referente específicamente a una API concreto.
- La segmentación se entiende como el aislamiento de los recursos de otras partes de la organización o el exterior. Una estrategia de segmentación efectiva guiará a todos los equipos técnicos (TI, seguridad, aplicaciones) a aislar de forma coherente el acceso mediante redes, aplicaciones, identidades y otros controles de acceso. Por ejemplo, la pertenencia o no a un segmento perimetral con acceso externo determinara todas las consideraciones posteriores.
- La autenticación es el proceso de comprobación de la identidad, siempre debe ser externo a la API y acogido a estándares.
- La autorización es el proceso que concede o deniega el acceso a un sistema al comprobar si la identidad tiene los permisos necesarios para realizar la acción solicitada. La API puede delegar la autorización o asumirla, en cuyo caso es necesario establecer los mecanismos de comunicación de la identidad. Las identidades externas se pueden recibir mediante token por cookies (desaconsejado) o en la cabecera Authorization, el token debe estar encriptado con JWT o mecanismos similares.
- La identidad se puede establecer para aplicaciones o usuarios individuales. La autorización se puede establecer para identidades individuales o grupos (roles), desde el nivel de recursos completos hasta bajar a nivel de operaciones o ámbitos de datos.

Codificación segura

- Usar el modelado de amenazas durante el diseño permite identificar posibles amenazas de seguridad para asegurarse de establecer las mitigaciones adecuadas.
- El modelo STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service y Elevation of Privilege) identifica:
  - Suplantación de identidad: Requerir conexiones HTTPS.
  - Alteración de datos: Validar certificados SSL/TLS.
  - Rechazo: Habilitar opciones de supervisión y diagnóstico.
  - Divulgación de información: Cifrar datos confidenciales en reposo y en tránsito.
  - Denegación de servicio: Supervisar las métricas de rendimiento de las posibles condiciones de denegación de servicio. Implementar filtros de conexión.
  - Elevación de privilegios: Establecer una política de privilegios mínimos y re-autenticación en operaciones criticas.

Codificación segura

- Hay que reducir la superficie expuesta a ataques y diseñar una defensa en profundidad: la validación y saneamiento de todas las entradas, siempre hay que tratarlas como no confiables y son el vehículo para los ataques por inyección.
- Hay que evitar las vulnerabilidades por exceso de información: eliminando de encabezados innecesarios, que revelan información sobre el servidor y las tecnologías subyacentes, y capturando las excepciones para devuelvan una respuesta significativa a los clientes.
- No se debe permitir que las excepciones no detectadas se propaguen al marco. Las excepciones deben capturase para formatear una respuesta con la información significativa para el cliente exclusivamente y permitir un control coherente. También se debería incluir algún tipo de registro de errores debidamente segurizado que capturase los detalles completos de cada excepción.
- La devolución de la excepción también debe incluir el código de estado HTTP adecuado, en lugar de abusar del código 500 para todas las situaciones. El protocolo HTTP distingue entre los errores que se producen debido a la aplicación de cliente (HTTP 4xx) y los errores causados por un problema en el servidor (HTTP 5xx). Hay que respetar esta convención en los mensajes de error de respuesta.

Estándares de definición e implementación

- La estandarización del diseño de API garantiza que todas las APIs creadas por una organización sigan siendo coherentes. Una forma de estandarizar el diseño de API es adoptar un estándar como OData, otra es estableciendo y aplicando pautas de estilo para todas las API:
  - Directrices de definición de URL
  - Política de uso de métodos HTTP
  - Directrices de devolución de códigos de estado HTTP
  - Política de uso de encabezados: comunes, para solicitudes, para respuestas
  - Directrices de uso parámetros de consulta y estandarización de los parámetros comunes
  - Recomendaciones de uso de formatos (tipos de medios) de entrada y salida
  - Recomendaciones de uso de hipermedia
- La estandarización del diseño de API no solo garantiza que las APIs sean coherentes en toda la organización, sino que también que contienen componentes reutilizables.
- Así mismo hay que fijar una serie de directrices que fijen cuando y como utilizar las diferentes plataformas de desarrollo, bases de datos, documentación, …

Estándares de definición e implementación

- Plataformas
  - JEE:
    - Spring
    - JAX-RS, JAX-WS
  - .NET:
    - Framework
    - Core
  - NodeJS:
    - Express
    - Loopback
  - Otras: PHP, Django, …
- Documentación
  - OpenAPI
  - HATEOAS
  - GraphQL
- Bases de datos
  - Relacionales: Oracle, Azure SQL Database (SQL Server), …
  - NoSQL: MongoDB, CosmosDB, Redis, …

Estándares de definición e implementación

- La creación de una aplicación cliente que invoque solicitudes REST para tener acceso a una API web requiere que se escriba una cantidad significativa de código para construir cada solicitud y se le dé el formato adecuado, que se envié la solicitud al servidor que hospeda el servicio web y se analice la respuesta para determinar si la solicitud se realizó correctamente o no y que se extraigan los datos devueltos. Para aislar la aplicación cliente de estas cuestiones, se puede proporcionar un SDK que encapsule la interfaz REST, por lo menos en las de consumo externo, y abstraiga estos detalles de bajo nivel dentro de un conjunto de métodos más funcional. Estas facilitan y simplifican el consumo de la API ocultando los detalles de la implementación, evitando que se cometan determinados tipos de errores y aplicando las mejores practicas.
- El problema con las SDK es que deben ser específicas para una plataforma o un lenguaje, y deberían cubrir las mas utilizados por los posibles clientes: .NET, Java, Java para Android, Objective-C (IOS), JavaScript, … La problemática se amplia no solo a la creación del SDK sino al mantenimiento de los mismos en paralelo al versionado de las APIs y al versionado de las plataformas o lenguajes.
- Para facilitar esta tarea, actualmente hay productos como Amazon API Gateway que permite generar un SDK para una API, que tenga alojada, en Java, JavaScript, Java para Android y Objective-C o Swift para iOS y Ruby.
- Swagger Codegen (open source) es un generador de código fuente para crear stubs de servidor y SDK de cliente directamente desde un especificación OpenApi o Swagger.

Pruebas

- Una API web debe probarse tan exhaustivamente como cualquier otro producto de software. Requiere pruebas unitarias para validar la funcionalidad, pero la naturaleza especifica de las APIs web incorpora sus propios requisitos adicionales para comprobar que funciona correctamente. Se debe prestar especial atención a los siguientes aspectos:
  - Ejercitar todas las rutas y parámetros para comprobar que invocan las operaciones correctas y devuelven los códigos de estado apropiados.
  - 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.
- Se deben automatizar con herramientas como postman, soapui, jmeter, …

Monitorización

- La supervisión es una parte fundamental en el mantenimiento de los objetivos de calidad de servicio. La monitorización es la recopilación de datos de supervisión en las siguientes situaciones:
  - Garantizar que el sistema esté en buen estado.
  - Realizar un seguimiento de la disponibilidad del sistema y sus elementos componentes.
  - Mantener el rendimiento para garantizar que el procesamiento del sistema no se degrada inesperadamente a medida que el volumen de trabajo aumenta.
  - Garantizar que el sistema satisface los contratos de nivel de servicio (SLA) establecidos con los clientes.
  - Proteger la privacidad y la seguridad del sistema, los usuarios y sus datos.
  - Realizar un seguimiento de las operaciones que se llevan a cabo como auditoría para fines normativos.
  - Supervisar el uso diario del sistema e identificar las tendencias que podrían conducir a problemas a fin de actuar con anticipación.
  - Crear estadísticas de uso: number of requests per app, user, resource, developer, …
  - Realizar un seguimiento de los problemas que se producen.
  - Realizar un seguimiento de las operaciones y depurar las versiones de software.
- Hay que establecer los criterios de supervisión del estado, la disponibilidad, el rendimiento, la seguridad, el uso, el seguimiento de problemas y auditoría, que permitan el análisis y la toma de decisiones, así como fijar alertas para la detección temprana de problemas.

Monetización

- La introducción y expansión de las APIs son una nueva oportunidad de negocio. A través de las APIs las compañías pueden ofrecer servicios a clientes o proveedores, de forma que, cobrando en diferentes modalidades por su uso, pueda obtener ingresos de las mismas.
- Los métodos utilizados de forma más habituales son:
  - **Free**: Utilización gratis de las APIs, solamente es necesario autorizar al consumidor.
  - **Freemium**: Utilización gratis hasta cierto nivel de consumo. Una vez se alcanza este nivel gratuito, el consumidor debe pagar por el uso.
  - **Suscripción**: Los clientes se suscriben a un nivel predefinido de uso mensual de las APIs, con alertas y recargos si el límite de suscripción se excede.
  - **Pay per use**: Precio basado directamente en el uso, por número de llamadas o MB de datos consumidos.
  - **Revenue share**: El desarrollador recibe un valor fijo o porcentual de los beneficios obtenidos por el usos de la API.
  - **Versiones gratis y de pago**: Se publican diferentes versiones de la misma API, la básica es gratis y la versión más completa es de pago. Incluso la API podría ser la misma y en función del consumidor puede tener o no acceso a las funcionalidades avanzadas.

Gestión de los entornos

- Es necesario establecer y definir los diferentes entornos teniendo en cuenta los costes y la seguridad:
  - Desarrollo
  - Pruebas
  - Preproducción
  - Producción
- El entorno de Preproducción debe ser una idéntico del entorno de Producción. Para minimizar los costos se puede implementar en la nube una réplica del entorno de producción, ejecutar las pruebas y, a continuación, liberar los recurso.
- Según las recomendaciones de seguridad, los entornos de preproducción no deberían utilizar datos reales e implementar la generación realista de datos de pruebas.
- Así mismos, los secretos y configuraciones deben almacenarse de forma separada para los diferentes entornos.
- En algunos casos se requerirán sandbox para determinadas pruebas especificas.

API FIRST

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

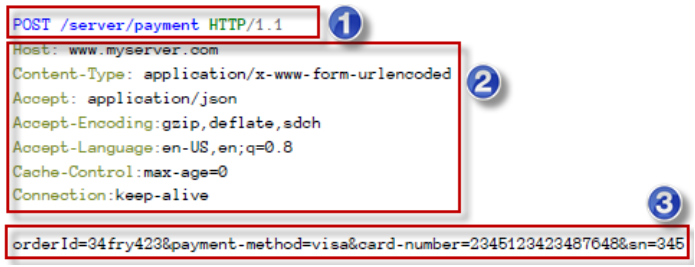
- Asegura buenas experiencias de desarrollador
  - Las APIs bien diseñadas, bien documentadas y consistentes brindan experiencias positivas para los desarrolladores porque es más fácil reutilizar el código y los desarrollos integrados, reduciendo la curva de aprendizaje.
- Reduce el riesgo de fallos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

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.

```
HTTP/1.1 201 Created
Content-Type: application/json;charset=utf-8
Location: https://www.myserver.com/services/payment/3432
Cache-Control:max-age=21600
Connection:close
Date:Mon, 23 Jul 2012 14:20:19 GMT
ETag:"2cc8-3e3073913b100"
Expires:Mon, 23 Jul 2012 20:20:19 GMT

{
  "id":"https://www.myserver.com/services/payment/3432",
  "status": "pending"
}
```


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

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 de una API web RESTful
- Documentación Open API

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.

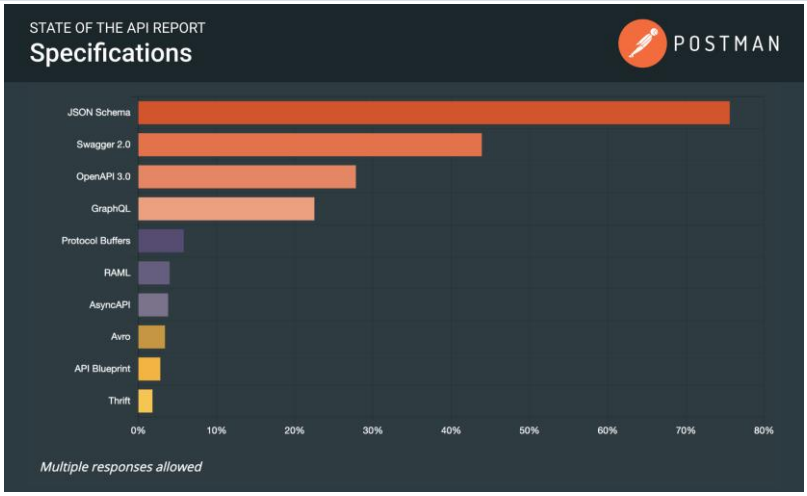
Documentar servicios Rest

- 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 de describir API prácticamente 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



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.

Swagger

https://swagger.io/

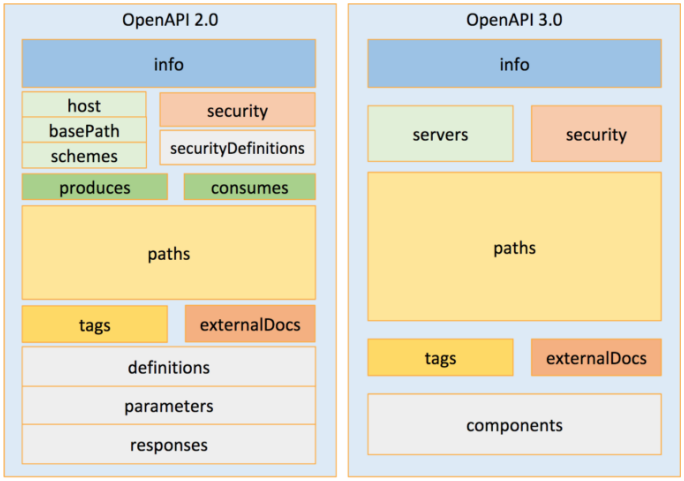
- Swagger (OpenAPI Specification) es una especificación abierta y su correspondiente implementación para probar y documentar servicios REST. Uno de los objetivos de Swagger es que podamos actualizar la documentación en el mismo instante en que realizamos los cambios en el servidor.
- Un documento Swagger 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 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

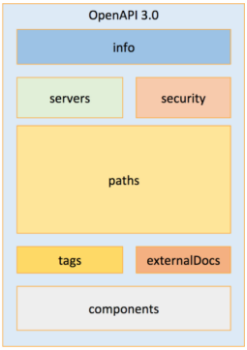
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.
- Es sensible a mayúsculas y minúsculas, todas las propiedades (palabras reservadas) de la especificación deben ir en minúsculas y terminar en dos puntos (:).
- Las propiedades requieren líneas independiente, 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/

Spring Boot: Instalación (v3.0)

- Se debe añadir la dependencia Maven del starter de springfox-swagger.

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-boot-starter</artifactId>
   <version>3.0.0</version>
</dependency>
```

- Establecer el contexto en el fichero application.properties

```
server.servlet.contextPath=/
```

- Para activar la documentación se anota la aplicación :

```
@EnableOpenApi
@SpringBootApplication
```

- Para acceder a la documentación:
  - http://localhost:8080/swagger-ui/index.html (versión HTML)
  - http://localhost:8080/v3/api-docs (versión JSON)
  - https://springfox.github.io/springfox/docs/current/#introduction (framework)

Spring Boot: Soporte adicional (v3.0)

- En la versión 2.3.2, se agregó soporte para las anotaciones de validación de bean JSR-303, específicamente para @NotNull, @Min, @Max y @Size. Es necesario incluir la dependencia:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-bean-validators</artifactId>
  <version>3.0.0</version>
</dependency>
```

- En la versión 2.6.0, se agregó soporte para Spring Data Rest. Es necesario incluir la dependencia:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-data-rest</artifactId>
  <version>3.0.0</version>
</dependency>
```


Spring Boot: Anotar el modelo

- **@ApiModel**: documenta la entidad, con una descripción corta (value) y una descripción más larga (description).

```
@ApiModel(value = "Entidad Personas", description = "Información completa de la personas")
public class Persona {
```

- **@ApiModelProperty**: documenta las propiedades, con una descripción (value) y si es obligatoria (required). También contiene otros parámetros de posibles valores (allowableValues), no mostrarlo con Swagger (hidden) y otros.

```
@ApiModelProperty(value = "Identificador de la persona", required = true)
private Long id;
```

Spring Boot: Anotar el servicio

- **@Api**: documenta el servicio REST en si. Va a ser la descripción que salga en el listado, entre otras cosas.

```
@RestController
@Api(value = "Microservice Personas", description = "API que permite el mantenimiento de personas")
public class PersonasResource {
```

- **@ApiOperation**: documenta cada método del servicio.
- **@ApiParam**: documenta los parámetros de cada método del servicio.

```
@GetMapping(path = "/{id}")
@ApiOperation(value = "Buscar una persona", notes = "Devuelve una persona por su identificador" )
public Persona getOne(@ApiParam(value = "id" ,required=true) @PathVariable int id) {
```

- **@ApiResponses**, **@ApiResponse**: documenta las posibles respuestas del método, con un mensaje explicativo.

```
@ApiResponses({
        @ApiResponse(code = 200, message = "Persona encontrada"),
        @ApiResponse(code = 404, message = "Persona no encontrada")
})
```


Spring Boot: Configuración

```
@Configuration
public class SwaggerConfiguration {
        @Bean
        public Docket api() {
          return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("curso.controllers"))
            .paths(PathSelectors.ant("/**"))
            .build()
            .apiInfo(new ApiInfoBuilder()
                .title("Mis microservicios")
                .version("1.0")
                .license("Apache License Version 2.0")
                .contact(new Contact("Yo Mismo", "http://www.example.com", "myeaddress@example.com"))
                .build());
        }
}
```

Spring Boot: Archivo de propiedades

- Hay que crear en resources un archivo de propiedades, por ejemplo, swagger.properties
- Insertar los mensajes deseados como pares clave-valor donde la clave se usará como marcador de posición:

  person.id.value = Identificador único de la persona
- En lugar del texto en la anotación, se inserta un marcador de posición:

  @ApiModelProperty(value = "${person.id.value}", required = true)
- Hay que registrar el archivo de propiedades de la configuración a nivel de clase:

  @PropertySource ("classpath: swagger.properties")

ASP.NET Core: Swashbuckle

- Instalación:
  - Package Manager : Install-Package Swashbuckle.AspNetCore -Version 6.0.7
  - CLI : dotnet add package --version 6.0.7 Swashbuckle.AspNetCore
- La “Comentarios de documentación XML” proporcionan las descripciones y detalles de la documentación, es necesario exportarlos para que Swashbuckle los presente.
  - <GenerateDocumentationFile>true</GenerateDocumentationFile>
- En el método Startup.Configure, se habilita el middleware para servir el documento JSON generado y la interfaz de usuario de Swagger:

```
app.UseSwagger();
app.UseSwaggerUI(c => {
   c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ejemplo de API V1");
   c.RoutePrefix = string.Empty; // en la raíz del sitio
});
```

- Agregar el generador de Swagger a la colección de servicios en el método Startup.ConfigureServices:

```
services.AddSwaggerGen(c => {
   c.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
      $"{Assembly.GetExecutingAssembly().GetName().Name}.xml"));
});
```

http://localhost:<port>/swagger

ASP.NET Core: Personalizar

- La acción de configuración que se pasa al método AddSwaggerGen permite agregar información al interfaz de usuario, como el autor, la licencia y la descripción:

```
using Microsoft.OpenApi.Models;
services.AddSwaggerGen(c => {
   c.SwaggerDoc("v1", new OpenApiInfo {
      Version = "v1",
      Title = "Demos del curso",
      Description = "Un ejemplo simple ASP.NET Core Web API",
      TermsOfService = new Uri("https://example.com/terms"),
      Contact = new OpenApiContact {
         Name = "Yo Mismo",
         Email = "myeaddress@example.com",
         Url = new Uri("https://example.com/people/yo-mismo"),
      },
      License = new OpenApiLicense {
         Name = "Apache License Version 2.0", Url = new Uri("https://example.com/license"),
      }
   });
});
```


ASP.NET Core: Documentar el modelo

- Es necesario documentar tanto los ApiController y sus operaciones así como cualquier modelo utilizado.
- Para documentar los modelos hay documentar tanto la clase como las propiedades. En caso de estar anotados para fijar las restricciones con los atributos que se encuentran en el espacio de nombres System.ComponentModel.DataAnnotations, estas se traspasaran a la documentación de la API.

```
/// <summary>
/// Pronostico del tiempo
/// </summary>
public class WeatherForecast {
   /// <summary>
   /// Identificador de pronostico
   /// </summary>
   [Required]
   public int Id { get; set; }
   /// <summary>
   /// Temperatura en grados centígrados
   /// </summary>
   [Range(minimum: -50, maximum: 67)]
   public int TemperatureC { get; set; }
```

ASP.NET Core: Documentar el servicio

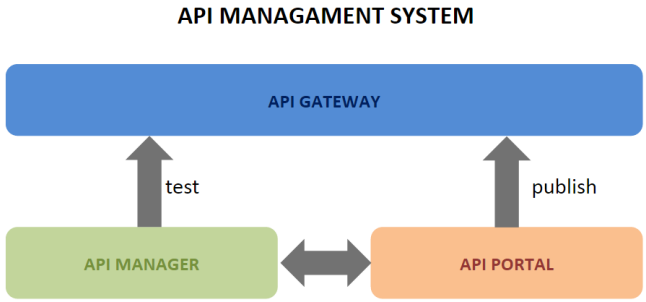
- La interfaz de usuario de Swagger muestra el texto interno de los elementos <summary>, <remarks> (description), <param> y <returns> según corresponda.
- Lo que más preocupa a los desarrolladores que consumen una API es lo que se devuelve; sobre todo, los tipos de respuesta y los códigos de error (si no son los habituales). Los tipos de respuesta y los códigos de error se indican en las anotaciones y los comentarios XML <response>.

```
/// <summary>
/// Obtener una temperatura concreta
/// </summary>
/// <remarks>
/// Descripción detallada con ejemplos
/// </remarks>
/// <param name="id">Identificador de captura</param>
/// <returns>Datos de la previsión</returns>
/// <response code="200">Success</response>
/// <response code="404">Not found</response>
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound, Type = typeof(ErrorMessage))]
public WeatherForecast Get(int id) {
```


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…

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.

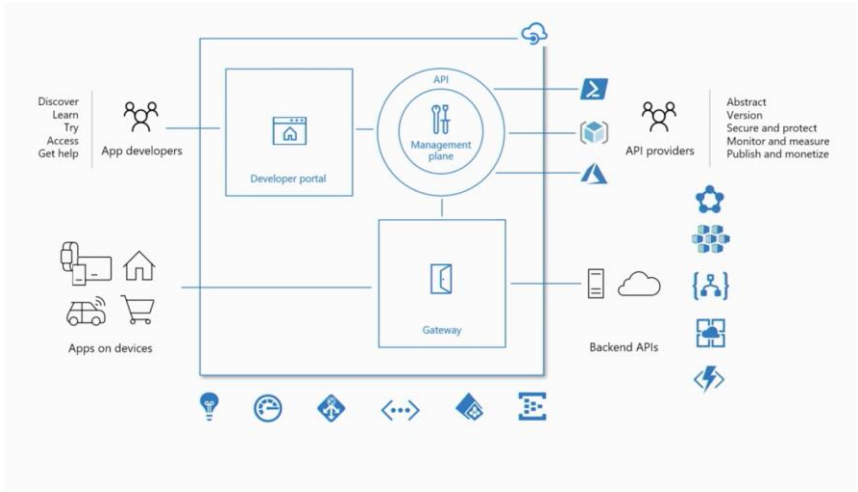
Azure Portal

- 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



API y operaciones

- Las APIs son el fundamento de una instancia del servicio API Management.
- Cada API representa un conjunto de operaciones disponibles para los desarrolladores.
- Cada API contiene una referencia a un servicio back-end que implementa la API y sus operaciones se asignan a las operaciones implementadas por dicho servicio.
- Las operaciones de API Management son altamente configurables, con control sobre asignación de direcciones URL, parámetros de consulta y ruta de acceso, contenidos de solicitudes y respuestas y almacenamiento en caché de respuestas de operaciones. A nivel de API completa o en el ámbito de cada operación individual, también se pueden implementar directivas de límite de tasa, cuotas y restricción de direcciones IP.

Directivas

- Las directivas son la funcionalidad de API Management que permite a Azure Portal cambiar el comportamiento de la API a través de la configuración. Las directivas son una colección de declaraciones que se ejecutan secuencialmente en la solicitud o respuesta de una API.
- Entre las declaraciones más usadas se encuentran la conversión de formato de XML a JSON, JSON a XML, modificar o suprimir cabeceras o parametros y la limitación de tasa de llamadas para restringir el número de llamadas entrantes de un desarrollador, pero también hay muchas otras directivas disponibles.
- Las expresiones de directiva pueden utilizarse como valores de atributos o valores de texto en cualquiera de las directivas de API Management, a menos que la directiva especifique lo contrario.
- Algunas directivas como Flujo de control y Establecer variable se basan en expresiones de directiva.

Productos

- Los productos son la forma de presentar un conjunto coherente de APIs a los desarrolladores para su consumo, simplifican la administración y consumo de las APIs.
- Los productos en API Management tienen una o varias APIs y se configuran con un título, una descripción y términos de uso.
- Los productos pueden ser de tipo Abierto o Protegido. Para poder usar los productos protegidos es necesario suscribirse antes a ellos, mientras que los productos abiertos pueden usarse sin suscripción.
- Cuando un producto está preparado para que lo usen los desarrolladores, se puede publicar. Una vez publicado, los desarrolladores pueden verlo (y, en el caso de los productos protegidos, suscribirse a él).
- La aprobación de la suscripción se configura en el ámbito de producto y puede requerir la aprobación del administrador o aprobarse automáticamente.
- Los grupos se usan para administrar la visibilidad de productos a los desarrolladores. Los productos conceden visibilidad a los grupos y los desarrolladores pueden ver los productos visibles a los grupos a los que pertenecen y suscribirse a ellos.

Grupos

- Los grupos se usan para administrar la visibilidad de productos a los desarrolladores. API Management tiene los siguientes grupos invariables del sistema:
  - Administradores: los administradores de la suscripción de Azure son miembros de este grupo. Los administradores controlan las instancias del servicio API Management y crean las APIs, las operaciones y los productos que usan los desarrolladores.
  - Desarrolladores: los usuarios del portal para desarrolladores autenticados se incluyen en este grupo. Los desarrolladores son los clientes que compilan aplicaciones con las APIs. Los desarrolladores, después de que se les concede acceso al portal para desarrolladores, crean aplicaciones que llaman a las operaciones de una API.
  - Invitados: a este grupo pertenecen los usuarios del portal para desarrolladores no autenticados como, por ejemplo, clientes potenciales que visitan el portal para desarrolladores de una instancia de API Management. Se les concede determinado acceso de solo lectura, como por ejemplo la posibilidad de ver APIs pero no llamarlas.
- Además de estos grupos del sistema, los administradores pueden crear grupos personalizados o aprovechar los grupos externos en inquilinos de Azure Active Directory asociados. Los grupos personalizados y externos pueden usarse junto con grupos del sistema en la concesión a los desarrolladores de visibilidad y acceso a productos de la API.

Desarrolladores

- Los desarrolladores representan las cuentas de usuario de una instancia del servicio API Management.
- Los desarrolladores pueden ser creados por administradores o invitados por estos y también pueden suscribirse desde el Portal para desarrolladores.
- Cada desarrollador es miembro de uno o varios grupos y se puede suscribir a los productos que conceden visibilidad a esos grupos.
- Cuando los desarrolladores se suscriben a un producto, se les concede la clave principal (usuario) y secundaria (contraseña) para dicho producto. Esta clave se usa cuando se realizan llamadas en las APIs del producto.

Herramientas

- Azure Portal:
  - Portal web de administración de Azure a través de navegador.
- Azure CLI:
  - Los comandos del grupo de comandos az apim del interfaz de línea de comandos.
- Azure PowerShell:
  - Mediante comandos o script con cmdlets de PowerShell.
- Visual Studio Code:
  - Mediante la extensión de Azure API Management para Visual Studio Code.
- REST:
  - Azure API Management ofrece una API REST para realizar operaciones en entidades seleccionadas, como usuarios, grupos, productos y suscripciones.
- Plantilla de Resource Manager:
  - Es un archivo de notación de objetos JavaScript (JSON) que define la infraestructura y la configuración del proyecto.

Instancia de API Management

- Creación de un nuevo servicio
  - Crear un recurso → Integración > API Management.
  - Definir:
    - Nombre
    - Suscripción
    - Grupos de recursos
    - Ubicación
    - Nombre de la organización.
    - Correo electrónico del administrador
    - Plan de tarifa.
- Eliminar servicio
  - Grupos de recursos → Eliminar grupo de recursos.

Publicación de una API

- Se puede partir de una API en blanco y especificar manualmente todos los parámetros necesarios.
- Existen varios marcos y estándares de API para los que API Management proporciona opciones para la importación de las API:
  - OpenAPI: OpenAPI es una especificación que documenta todos los puntos de conexión y operaciones de API de RESTful, y todos los parámetros de entrada y salida. OpenAPI se llamó originalmente Swagger.
  - WADL: Web Application Description Language es una descripción XML de servicios web basados en HTTP. Es un formato más sencillo y más ligero que WSDL.
  - WSDL: Web Service Description Language es una descripción XML de cualquier servicio de red, no solo HTTP. Permite la importación de Servicios Web XML (SOAP).
  - Logic App: Las aplicaciones lógicas se usan para organizar y automatizar los flujos de trabajo y las integraciones con varios orígenes de datos.
  - Aplicación de API: Una API que se hospeda dentro de un servicio de aplicación de API en Azure.
  - Function App: Código sin servidor que se puede llamar mediante desencadenadores.

Importación de una API

- En Azure Portal → OpenAPI → Crear a partir de la especificación OpenAPI → Completa.
  - Especificación OpenAPI: URL con la especificación OpenApi que describe la API. API Management reenvía las solicitudes a esta dirección.
  - Nombre para mostrar: El nombre se muestra en el portal para desarrolladores.
  - Nombre: Nombre único para la API.
  - Descripción: Descripción opcional de la API.
  - Esquema URL: Protocolos con los que se puede acceder a la API: HTTP o HTTPS o ambos.
  - Sufijo de dirección URL de API: Sufijo que se anexa a la dirección URL base del servicio API Management, debe ser único para cada API de un publicador determinado y cumplir con el URL Encoding.
  - Etiquetas: Etiquetas para buscar, agrupar o filtrar las APIs.
  - Productos: Asociación de una o más APIs.
  - Puertas de enlace: Puertas de enlace de API que exponen la API (solo para niveles Developer y Premium). Administrada indica la puerta de enlace integrada en el servicio API Management y hospedada por Microsoft en Azure.
  - ¿Definir versión de esta API?: Para cuando se publiquen de varias versiones de la API.

Edición de una API

- La definición de la API esta compuesta por:
  - Propiedades de configuración en el API Management
  - Operaciones
  - Estructuras de datos
  - Directivas
- Se pueden modificar mayoría de las propiedades de configuración con las que se creo o importo la API a través de la solapa ***Settings***.
- Las operaciones y los datos se pueden modificar y completar con el interfaz gráfico del portal en la pestaña de ***Design*** o directamente con el editor swagger, que se encarga de la validación. En la misma solapa también se pueden modificar las directivas usando el interfaz gráfico, los asistentes o directamente con el editor xml que también realiza la validación.

Entorno de pruebas

- En el portal, la solapa de ***Test*** del mantenimiento de APIs da acceso al entorno de pruebas.
- Una vez seleccionada la API y la operación, en la página se muestran los parámetros de consulta, los encabezados y cuerpo para configurar la petición. Se puede establecer un ámbito de producto especifico para que se apliquen sus credenciales y directivas. El interfaz muestra la URL y la petición HTTP que se va a enviar.
- Una vez enviada se puede consultar la respuesta en ***Message*** y realizar el seguimiento de la llamada en ***Trace***.
- La información de seguimiento permite la depuración de la API y de las directivas que se le aplican:
  - Entrada: muestra la solicitud original API Management recibida del autor de llamada y las directivas aplicadas a la solicitud.
  - Back-end: muestra las solicitudes que API Management envió al back-end de la API y la respuesta que recibió.
  - Salida: muestra todas las directivas que se aplican a la respuesta antes de enviarla de vuelta al autor de llamada.

Depuración

- Para realizar la depuración es necesario tener habilitado el seguimiento.
  - Se puede habilitar el valor “Permitir seguimiento” para la suscripción que se usa para enviar la prueba. Si se usa la suscripción integrada de acceso a todo, este valor está habilitado de forma predeterminada.
  - Para habilitarlo a nivel de prueba concreta, se puede incluir el encabezado HTTP de la solicitud: Ocp-Apim-Trace: True
- En la pestaña ***Message***, el encabezado ocp-apim-trace-location muestra la ubicación de los datos de seguimiento almacenados en Azure Blob Storage. Si es necesario, se puede recuperar el seguimiento desde esta ubicación.

Directivas

- En Azure API Management (APIM), las directivas constituyen el mecanismo fundamental del sistema que permite al publicador cambiar el comportamiento de la API mediante la configuración. Las directivas son una colección de declaraciones que se ejecutan secuencialmente en la solicitud o respuesta de una API.
- Las directivas ayudan a abordar cuestiones transversales, como la autenticación, la autorización, la limitación, el almacenamiento en caché y la transformación.
- Las directivas se aplican en la puerta de enlace que se encuentra entre el consumidor de la API y la API administrada. La puerta de enlace recibe todas las solicitudes y normalmente las reenvía sin modificar a la API subyacente. Sin embargo, una directiva puede aplicar cambios a la solicitud de entrada y en la respuesta de salida.
- La definición de la directiva es un simple documento XML que describe una secuencia de declaraciones de entrada y de salida. El XML se puede editar directamente en la ventana de definición. Se ofrece una lista de instrucciones a la derecha y las instrucciones aplicables al ámbito actual están habilitadas y resaltadas.

Ámbitos de la directiva

- Las directivas se pueden configurar a diferentes niveles, desde lo mas general a lo mas concreto. Para configurar una directiva, se debe seleccionar en primer lugar el ámbito en el que se debe aplicar.
- Los ámbitos de la directiva se evalúan en el orden siguiente:
  - Ámbito global (All APIs)
  - Ámbito del producto
  - Ámbito de la API (All Operations)
  - Ámbito de la operación
- Las instrucciones dentro de las directivas se evalúan dentro de su ámbito. La instrucción de directiva <base /> ejecuta las instrucciones de ámbitos superiores. Su orden permite establecer si las directivas locales se ejecutan antes o después de las heredadas. Si <base /> no aparece en la directiva local se omite la ejecución de las heredadas.

Secciones de la directiva

- La configuración se divide en:
  - inbound: se aplica sobre la petición recibida del consumidor.
  - backend: se aplica antes de reenviar la petición a la API subyacente.
  - outbound: se aplica sobre la respuesta obtenida y antes de devolverla al consumidor.
  - on-error: solo se aplica si produce un error.
- La orden de las instrucciones de directiva determinan orden en se ejecutan para una solicitud y una respuesta.

```
<policies>
  <inbound></inbound>
  <backend></backend>
  <outbound></outbound>
  <on-error></on-error>
</policies>
```

Expresiones de directiva

- Las instrucciones de directivas se pueden expresar como simples elementos y atributos XML, o mediante sofisticadas expresiones de directiva en C# 7.
- Cada expresión tiene acceso a la variable de contexto proporcionada de forma implícita y a un subconjunto permitido de tipos de .NET Framework.
- Las expresiones de declaración única se incluyen en @(expression) y las expresiones de múltiples declaraciones se incluyen en @{expression} pero todas las rutas de código de las expresiones de múltiples deben terminar con una declaración return. Las expresiones pueden utilizarse como valores de atributos o valores de texto en cualquier directiva a menos que la referencia de la directiva especifique lo contrario.
- Al utilizar expresiones de directiva, solo hay una verificación limitada de estas cuando se define la directiva. La puerta de enlace ejecuta las expresiones por lo que las excepciones generadas por las expresiones generaran un error en tiempo de ejecución.

Expresiones de directiva

- DotLiquid es un sistema de plantillas adaptado al framework .net desde Ruby's Liquid Markup. Por medio de la sintaxis DotLiquid se pueden personalizar los cuerpos de las respuestas utilizando un lenguaje de definición de plantillas:
  - Los marcadores permiten mostrar el valor de una expresión y se indican con llaves dobles: {{ y }}. Los filtros cambian la salida del marcador y van separados por un | de la expresión que transforman.
  - Las instrucciones del flujo de control (if / elseif / else / endif, unless / endunless , case / when / else / endcase, for / break / continue / forend) de las plantillas se indican mediante llaves y signos de porcentaje: {% y %}. Para suprimir el salto de línea que normalmente seguiría al final de una instrucción se puede cerrar con un guión ("-"): -%}.

```
<GetDetails>
{% if body.id -%}
        <Id>{{body.id}}</Id>
{% else -%}
        <Id xsi:nil="true" />
{% endif -%}
</GetDetails>
```

Control de errores en las directivas

- Si se produce un error durante el procesamiento de una solicitud, los pasos restantes se omiten y la ejecución salta a las instrucciones de la sección on-error.
- Azure API Management proporciona un objeto ProxyError que permite a los publicadores responder a las condiciones de error que se pueden producir durante el procesamiento de solicitudes. Para acceder al objeto ProxyError se usa la propiedad context.LastError. Este objeto se puede usar en las directivas de la sección de directivas on-error.
- Mediante la colocación de instrucciones de directiva en la sección on-error se puede revisar el error con la propiedad context.LastError para inspeccionar y personalizar la respuesta de error con la directiva set-body o la y configurar lo que ocurre si se produce un error.
- Existen códigos de error predefinidos para pasos integrados y códigos de errores predefinidos en directivas que pueden producirse durante el procesamiento de las instrucciones de directiva.

Personalización de errores

- Devolución de error en la cabecera:

```
<on-error>
   <set-header name="X-Error-Source" exists-action="override"><value>@(context.LastError.Source)</value></set-header>
   <set-header name="X-Error-Reason" exists-action="override"><value>@(context.LastError.Reason)</value></set-header>
   <set-header name="X-Error-Message" exists-action="override"><value>@(context.LastError.Message)</value></set-header>
   <set-header name="X-Error-Scope" exists-action="override"><value>@(context.LastError.Scope)</value></set-header>
   <set-header name="X-Error-Section" exists-action="override"><value>@(context.LastError.Section)</value></set-header>
   <set-header name="X-Error-Path" exists-action="override"><value>@(context.LastError.Path)</value></set-header>
   <set-header name="X-Error-PolicyId" exists-action="override"><value>@(context.LastError.PolicyId)</value></set-header>
   <set-header name="X-Error-StatusCode" exists-action="override"><value>@(context.Response.StatusCode.ToString())</value></set-header>
   <base />
</on-error>
```

- Devolución de error en el body:

```
<on-error>
   <return-response>
     <set-status code="400" reason= "Bad Request" />
      <set-body>@{
            return new JObject(new JProperty("message", context.LastError.Message), new JProperty("reason", context.LastError.Reason)).ToString();
         }</set-body>
   </return-response>
   <base />
</on-error>
```

Variable de contexto

- Una variable denominada context está disponible implícitamente en todas las expresiones de directiva. Sus miembros proporcionan información relativa a petición. Todos los miembros de context son de solo lectura:
  - Api, Operation, Product, Subscription, Deployment, User, Variables, RequestId, Request, Response, Elapsed, Timestamp, LastError, Tracing, void Trace(message: string)
- Las propiedades son a su vez objetos que contienen propiedades, no todas de solo lectura, con la información relativa a sus ámbitos.

```
@(Regex.Match(context.Response.Headers.GetValueOrDefault("Cache-Control",""),
        @"max-age=(?<maxAge>\d+)").Groups["maxAge"]?.Value)
```

- La colección Variables permite incorporar información al contexto para su recuperación y tratamiento posterior, dado que las variable locales desaparecen al salir del bloque.

```
@(context.Variables.ContainsKey("max") ? int.Parse((string)context.Variables["max"]) : 100)
```

- El método Trace permite incorporar información a la traza.

Directivas de transformación

- Enmascarar URL en el contenido: reescribe (enmascara) vínculos en el cuerpo de respuesta para que apunten al vínculo equivalente a través de la puerta de enlace (HATEOAS).
- Convertir JSON a XML: convierte el cuerpo de solicitud o respuesta de JSON a XML.
- Convertir XML a JSON: convierte el cuerpo de solicitud o respuesta de XML a JSON.
- Buscar y reemplazar la cadena en el cuerpo: encuentra una subcadena de solicitud o de respuesta y la reemplaza por una subcadena diferente.
- Establecer cuerpo: establece el cuerpo del mensaje para las solicitudes entrantes y salientes.
- Establecer el servicio back-end: cambia el servicio back-end para una solicitud entrante.
- Establecer encabezado HTTP: asigna un valor a un encabezado de respuesta o de solicitud existente o agrega un nuevo encabezado de este tipo.
- Establecer el parámetro de cadena de consulta: agrega, reemplaza el valor o elimina el parámetro de la cadena de consulta de la solicitud.
- URL de reescritura: convierte una URL de solicitud de su forma pública a la forma esperada por el servicio web.
- Transformar XML mediante una XSLT: aplica una transformación de XSL al XML del cuerpo de la solicitud o respuesta.

Directivas avanzadas

- Establecer variable: conserva un valor en una variable de contexto con nombre para el acceso posterior.
- Flujo de control (condicional múltiple): aplica condicionalmente instrucciones de directiva basadas en la evaluación de expresiones booleanas.
- Reintentar (bucle): reintenta ejecutar las instrucciones de directiva adjuntas, si y hasta que se cumple la condición. La ejecución se repite en los intervalos de tiempo especificados y hasta el número de reintentos indicado.
- Limitar la simultaneidad: evita que las directivas delimitadas las ejecute simultáneamente un número de solicitudes mayor que el especificado.
- Trace: agrega seguimientos personalizados a la salida de la inspección de la API, a la telemetría de Application Insights y a los registros de recursos.
- Wait (Esperar): espera a que se completen las directivas adjuntas Send request (Enviar solicitud), Get value from cache (Obtener el valor de caché) o Control flow (Flujo de control) antes de continuar.

Directivas avanzadas

- Enviar solicitud unidireccional: envía una solicitud a la dirección URL especificada sin esperar una respuesta.
- Enviar solicitud: envía una solicitud a la dirección URL especificada.
- Reenviar solicitud: reenvía la solicitud al servicio back-end.
- Registro en el centro de eventos: envía mensajes en el formato especificado a un destino de mensaje definido por una entidad del registrador.
- Establecer el proxy HTTP: permite enrutar las solicitudes reenviadas a través de un proxy HTTP.
- Establecer método de solicitud: le permite cambiar el método HTTP de una solicitud.
- Establecer código de estado: cambia el código de estado HTTP al valor especificado.
- Devolver respuesta: anula la ejecución de la canalización y devuelve la respuesta especificada directamente al llamador.
- Mock response (Simular respuesta): anula la ejecución de la canalización y devuelve la respuesta ficticia directamente al llamador.

Directivas de autenticación

- Autenticar con opción básica: autenticar con un servicio de back-end mediante la autenticación básica.
- Autenticar con certificado de cliente: autenticar con un servicio de back-end mediante certificados de cliente.
- Autenticar con identidad administrada: autenticar con un servicio de back-end mediante una identidad administrada.

Directivas de restricción de acceso

- Activar encabezado HTTP: aplica la existencia o el valor de un encabezado HTTP.
- Validar JWT: aplica la existencia y la validez de un JWT extraído de un encabezado HTTP especificado o un parámetro de consulta especificado.
- Establecer cuota de uso por suscripción: permite aplicar un volumen de llamadas renovables o permanentes o una cuota de ancho de banda por suscripción.
- Establecer cuota de uso por clave: permite aplicar un volumen de llamadas renovables o permanentes o una cuota de ancho de banda por clave.
- Limitar la frecuencia de llamadas por suscripción: evita los picos de uso de la API limitando la frecuencia de llamadas, por suscripción.
- Limitar la frecuencia de llamadas por clave: evita los picos de uso de la API limitando la frecuencia de llamadas, por clave.
- Restringir IP de autor de llamada: filtra (permite/deniega) las llamadas de direcciones IP específicas o de intervalos de direcciones.

Directivas de almacenamiento en caché

- Almacenar en caché: almacena en caché la respuesta de acuerdo con la configuración de control de caché especificada.
- Obtener de caché: realiza una búsqueda en caché y devuelve una respuesta en caché válida cuando esté disponible.
- Almacenar valor en caché: almacena un elemento en la memoria caché por clave.
- Obtener valor de caché: recupera un elemento almacenado en caché por clave.
- Quitar valor de caché: quita un elemento de la memoria caché por clave.

Directivas entre dominios

- CORS: agrega soporte de uso compartido de recursos entre orígenes (CORS) a una operación o a una API para permitir llamadas entre dominios desde clientes basados en explorador.
- Permitir llamadas entre dominios: permite que la API sea accesible desde los clientes basados en explorador de Adobe Flash y Microsoft Silverlight.
- JSONP: agrega JSON con soporte de relleno (JSONP) a una operación o a una API para permitir llamadas entre dominios desde clientes basados en explorador de JavaScript.

Directivas de integración de Dapr

Las Directivas de integración de Distributed Application Runtime (Dapr) solo funcionan en la versión autohospedada de la puerta de enlace de API Management con la compatibilidad con Dapr habilitada y se encuentran en versión preliminar pública.

- Enviar una solicitud a un servicio: utiliza el tiempo de ejecución de Dapr para localizar y comunicarse de forma confiable con un microservicio de Dapr.
- Envío de un mensaje a un tema de publicación/suscripción: utiliza el entorno de ejecución de Dapr para publicar un mensaje en un tema de publicación/suscripción.
- Desencadenar el enlace de salida: utiliza el tiempo de ejecución de Dapr para invocar un sistema externo a través del enlace de salida.

Simulaciones

- Para aplicar el enfoque API First, se puede partir de una API en blanco.
- A través de editor OpenApi o los asistentes de formulario se definen las operaciones y las definiciones.
- Cada definición puede tener un ejemplo de esquema.
- Cada posible respuesta de una operación puede tener un ejemplo de contenido.
- La directiva de entrada mock-response, como el nombre indica, se utiliza para simular las API y las operaciones. Se anula la ejecución de la canalización normal y devuelve una respuesta simulada al llamador.
- La directiva siempre trata de devolver las respuestas de mayor fidelidad. Prefiere ejemplos de contenido de respuesta, siempre que estén disponibles. Genera las respuestas de ejemplo a partir de esquemas, cuando se proporcionan esquemas y no ejemplos de contenido. Si no se encuentran ni ejemplos ni esquemas, se devuelven las respuestas sin contenido.

Importación de API de SOAP y conversión en REST

- API Management permite la importación de API de SOAP y su conversión en REST.
- Al añadir un nuevo API WSDL hay que marcar De SOAP a REST.
- Con esta opción, APIM trata de realizar una transformación automática entre SOAP y JSON. En este caso, los consumidores deben llamar a la API como API RESTful, que devuelve JSON. APIM convierte cada solicitud en una llamada SOAP.
- Cada acción SOAP se mapea en una operacción REST.
- A través de directivas en la entrada (rewrite-uri, set-header, set-body) se convierte la solicitud REST en una invocación SOAP.
- De igual forma, se intercepta el resultado y a través de directivas en la salida (choose, set-header, set-body) se transforma el resultado SOAP en JSON.
- El mapeo de datos es directo por lo que refleja la estructura SOAP subyacente, por lo que requiere intervención manual.

Creación y publicación de un producto

- En Azure API Management, un producto contiene una o varias APIs, así como una cuota de uso y los términos de uso. Una API puede estar incluida en varios productos. Una vez publicado un producto, los desarrolladores pueden suscribirse al producto y comenzar a usar las APIs del producto.
- En Azure Portal → Productos → Agregar.
  - Nombre para mostrar: El nombre que desea que aparezca en el portal para desarrolladores.
  - Descripción: Proporciona información sobre el producto, como su finalidad, las APIs a las que proporciona acceso y otros detalles.
  - State: Publicado si se quiere da acceso el producto. De forma predeterminada, los nuevos productos no se publican y solo son visibles para el grupo Administradores.
  - Requiere suscripción: Cuando un usuario debe suscribirse para usar el producto.
  - Requiere aprobación: Cuando se quiere que un administrador revise y acepte o rechace los intentos de suscripción a este producto. Por defecto, los intentos de suscripción se aprueban automáticamente.
  - Límite de recuento de suscripciones: Opción de limitar el número de varias suscripciones simultáneas.
  - Términos legales: Para incluir los términos de uso del producto que deben aceptar los suscriptores para usarlo.
  - APIs existentes: Selección de una o varias APIs, se pueden agregar las APIs después de crear el producto.
- A los productos se le deben asignar grupos para dar visibilidad a los usuarios una vez publicados.

Administrar grupos

- Por defecto hay tres grupos creados:
  - Administrators, Developers y Guests.
- Los administradores pueden crear grupos personalizados o aprovechar los grupos externos de asociados inquilinos de Azure Active Directory. Para la creación de un grupo personalizado:
  - En Azure Portal → Groups → Agregar.
  - Hay que especificar un nombre único y un nombre visible para el grupo, opcionalmente se puede dar una descripción.
- Para la asociación de un grupo a un producto:
  - En Azure Portal → Products → Access control → Add group.
  - Seleccionar el grupo que desea agregar.
- Para la asociación de grupos a desarrolladores:
  - En Azure Portal → Groups → Members → Add.

Desarrolladores

- En Azure API Management, los desarrolladores son los usuarios de las APIs que se exponen mediante API Management.
- Para la creación de un desarrollador:
  - En Azure Portal → Users → Add.
  - Hay que especificar nombre y apellidos, email, nombre de usuario y contraseña.
- De forma predeterminada, las cuentas de desarrollador recién creadas se encuentran en estado Activo y se asocian al grupo *Developers*. Las cuentas de desarrollador que se encuentran en estado activo se pueden usar para obtener acceso a todas las APIs para las que tienen suscripciones.
- Para invitar a un desarrollador, Azure Portal → Users → Invite, se mostrará un mensaje de confirmación, pero el desarrollador recién invitado no aparecerá en la lista hasta que acepte la invitación. Cuando se invita a un desarrollador, se le envía un correo electrónico. Este correo electrónico se genera con una plantilla que se puede personalizar (Deployment + infrastructure → Plantillas de notificaciones). Una vez aceptada la invitación, la cuenta se activa.
- La cuenta se puede bloquear para posteriormente volver a activar. Una cuenta bloqueada no puede obtener acceso al portal para desarrolladores ni llamar a ninguna API. Una cuenta de usuario se puede eliminar definitivamente en cualquier momento.

Suscripciones

- Se puede optar por publicar de forma gratuita las APIs y la información que contienen, pero por lo general se querrá restringir el acceso a las organizaciones con las que mantenga una relación o a los usuarios que han pagado. Se puede controlar el acceso a las APIs mediante una suscripción. Las suscripciones permiten segmentar los niveles de acceso a una API.
- En determinadas situaciones, es posible que los publicadores de una API deseen publicarla para el público sin necesidad de suscripciones. Pueden anular la obligación de usar una suscripción mediante la opción Require subscription (Requerir suscripción) en la página Settings (Configuración) de la API en Azure Portal. Como resultado, se puede tener acceso a todas las APIs del producto sin una clave de API.
- Al crear la suscripción, aparte del nombre único, el nombre publico y si permite el seguimiento, se puede establecer el ámbito:
  - Todas las APIs: Se aplica a todas las APIs accesibles desde la puerta de enlace.
  - Producto: Un producto es una colección de una o varias APIs que se configuran en API Management. Se pueden asignar una misma API a más de un producto, de tal forma que pueden tener diferentes reglas de acceso, cuotas de uso y condiciones de uso.
  - Una sola API: Este ámbito se aplica a una sola API y a todos sus puntos de conexión.

Suscripciones y claves

- Al publicar las APIs mediante API Management, es sencillo y frecuente proteger el acceso a ellas mediante claves de suscripción. API Management también admite otros mecanismos para proteger el acceso a las APIs, por ejemplo: OAuth2.0, Certificados de cliente, Restringir IP de autor de llamada.
- Las claves de suscripción forman la autorización para habilitar el acceso a estas suscripciones. Cada vez que una aplicación cliente realiza una solicitud a una API protegida, debe incluir una clave de suscripción válida en la solicitud HTTP o la llamada será rechazada.
- Una suscripción es básicamente un contenedor con nombre para un par de claves de suscripción. Una clave de suscripción es una hash, cadena única generada de forma automática, que se puede pasar en los encabezados de la solicitud de cliente o como un parámetro de cadena de consulta. La clave está directamente relacionada con una suscripción, que puede tener diferentes áreas como ámbito. Las suscripciones proporcionan un control detallado sobre los permisos y las directivas.
- Estas claves de suscripción se pueden volver a generar en cualquier momento, por ejemplo, si se sospecha que una clave se ha compartido con usuarios no autorizados.

Suscripciones y claves

- Las aplicaciones que llaman a una API protegida deben incluir la clave en todas las solicitudes. El nombre de encabezado predeterminado es Ocp-Apim-Subscription-Key y la cadena de consulta predeterminada es subscription-key.
  - curl -X GET https://curso.azure-api.net/prediccion/api/hoy/53/1 -H 'Ocp-Apim-Subscription-Key: 4f97317392c448009956d04a6307de81'
  - curl -X GET https://curso.azure-api.net/prediccion/api/hoy/53/1?subscription-key= 4f97317392c448009956d04a6307de81
- Cada suscripción tiene dos claves, una principal y una secundaria. Tener dos claves facilita la regeneración de una clave. Por ejemplo, si se quiere cambiar la clave principal y se puede evitar el tiempo de inactividad usando la clave secundaria en las aplicaciones.
- En el caso de los productos con suscripciones habilitadas, las aplicaciones clientes deben proporcionar una clave cuando realicen llamadas a una API en ese producto. Los desarrolladores pueden obtener una clave mediante el envío de una solicitud de suscripción. Si se aprueba la solicitud, hay que enviarles la clave de suscripción de forma segura, por ejemplo, en un mensaje cifrado. Este paso es una parte fundamental del flujo de trabajo de API Management.

Protección mediante la autenticación de certificados de cliente

- API Management proporciona la capacidad de proteger el acceso a las APIs (es decir, de cliente a API Management) mediante certificados de cliente. Se puede validar el certificado entrante y comprobar que las propiedades del certificado tienen los valores deseados mediante expresiones de directiva.
- Los certificados se pueden usar para proporcionar autenticación mutua de TLS entre el cliente y la puerta de enlace de API. Se puede configurar la puerta de enlace de API Management para permitir solamente las solicitudes con certificados que contengan una huella digital específica. La autorización en el nivel de puerta de enlace se controla mediante directivas de entrada.
- Con la autenticación de clientes TLS, la puerta de enlace de API Management puede inspeccionar el certificado incluido en la solicitud de cliente y comprobar propiedades como las siguientes:
  - Entidad de certificación (CA): Solo se permiten certificados firmados por una CA concreta.
  - Huella digital: Se permiten certificados que contengan una huella digital especificada.
  - Asunto: Solo se permiten certificados con un asunto especificado.
  - Fecha de expiración: Solo se permiten los certificados que no hayan expirado.
- Se pueden combinar varias propiedades entre sí para formar requisitos de directiva propios. Por ejemplo, puede especificar que el certificado que se pasa en la solicitud esté firmado por una entidad de certificación determinada y que no haya expirado.

Certificados de cliente

- Los certificados de cliente se firman para garantizar que no se hayan modificado. Cuando una aplicación cliente envía un certificado, se debe comprobar que proviene de él y no de un impostor. Hay dos formas habituales de comprobar un certificado:
  - Comprobar quién ha emitido el certificado: Si el emisor ha sido una entidad de certificación en la que se confía, se puede usar el certificado. Para automatizar este proceso se puede configurar las entidades de certificación de confianza en Azure Portal.
  - Si el certificado lo ha emitido el asociado, compruebe que procede de él. Por ejemplo, si entrega el certificado en persona, puede estar seguro de su autenticidad. Estos se conocen como certificados autofirmados.
- Para recibir y comprobar los certificados de cliente a través de HTTP/2 en los niveles Desarrollador, Básico, Estándar o Premium, se debe activar la opción "Negociación del certificado de cliente" en la hoja "Dominios personalizados". En el nivel Consumo, se debe activar la opción "Solicitar certificado de cliente" en la hoja "Dominios personalizados".

Certificados de cliente

- Comprobación del emisor y el asunto:

```
<choose>
  <when condition="@(context.Request.Certificate == null ||
            !context.Request.Certificate.Verify() ||
            context.Request.Certificate.Issuer != "trusted-issuer" ||
            context.Request.Certificate.SubjectName.Name != "expected-subject-name")" >
    <return-response><set-status code="403" reason="Invalid client certificate" /></return-response>
  </when>
</choose>
```

- Comprobación de la huella digital:

```
<choose>
  <when condition="@(context.Request.Certificate == null || !context.Request.Certificate.Verify() ||
            context.Request.Certificate.Thumbprint != "DESIRED-THUMBPRINT-IN-UPPER-CASE")" >
    <return-response><set-status code="403" reason="Invalid client certificate" /></return-response>
  </when>
</choose>
```

Portal del desarrollador

- El portal para desarrolladores es un sitio web totalmente personalizable que se genera automáticamente con la documentación de las APIs. Aquí, los consumidores de APIs pueden descubrir las APIs, solicitar acceso (suscribirse), aprender a usarlas y probarlas.
- Es necesario publicar el portal para que este disponible para los desarrolladores y volverá a ser necesario para pone los cambios y las personalizaciones a disposición de los visitantes.
- El uso compartido de recursos entre orígenes (CORS) es un mecanismo que permite solicitar recursos de una página web desde otro dominio, fuera del dominio desde el que se proporcionó el primer recurso. Se necesita CORS para permitir que los visitantes del portal usen la consola interactiva en las páginas de referencia de la API y se debe habilitar para los dominios, incluidos los personalizados.
- Por defecto el inicio de sesión en el portal se realiza con las credenciales de usuario (desarrolladores) creadas en API Management. Se puede habilitar el inicio de sesión de usuario con Azure Active Directory, pero requiere aprovisionar una aplicación de Azure Active Directory y configurar el proveedor de identidades de Azure Active Directory en el servicio de API Management. Se pueden configurar otros proveedores de identidades.

Portal del desarrollador

- Desde el portal de desarrollo, con las opciones *APIs* y *Products*, el usuario tiene acceso a las APIs y productos publicados de los grupos a los que pertenezca así como a las que no requieran suscripción.
- Una vez localizada la API, tiene acceso a la documentación y al entorno de pruebas. Podrá realizar las pruebas si cuenta con las claves necesarias.
- Desde puede *Products* hacer las peticiones de suscripción.
- En *Profile* puede modificar los datos personales y tiene acceso a las suscripciones, donde puede obtener las claves o regenerarlas y cancelar una suscripción.
- En *Reports* se le muestra la información de los consumos realizados a través de sus suscripciones, se muestra:
  - Para: Last hour, Today, Last 7 days, Last 30 days, Last 90 days
  - Por: API calls, Data transfer, API response times, Product, Subscriptions, APIs, Operations
  - Indicando: Successful calls, Blocked calls, Failed calls, Other calls, Total calls, Average response time, Bandwidth
- Todas estas opciones se pueden personalizar y reubicar.

Seguridad

- Azure API Management utiliza el control de acceso basado en rol (RBAC) de Azure para permitir una administración de acceso pormenorizada de servicios y entidades de API Management (por ejemplo, API y directivas).
- Actualmente, API Management cuenta con tres roles integrados y próximamente agregará otros dos (*). Estos roles pueden asignarse en distintos ámbitos, como la suscripción, el grupo de recursos y la instancia concreta de API Management.
  - **API Management Service Contributor**: Superusuario, tiene acceso completo a los servicios y a las entidades de API Management. Tiene acceso al portal del editor heredado.
  - **API Management Service Reader**: Tiene acceso de solo lectura a los servicios y a las entidades de API Management.
  - **API Management Service Operator**: Puede administrar el servicios de API Management, pero no sus entidades (APIs, direcctivas, …).
  - **API Management Service Editor***: Puede administrar las entidades de API Management, pero no el servicios.
  - **API Management Content Manager***: Puede administrar el portal del desarrollador. Acceso de solo lectura a servicios y entidades.
- Si ninguno de los roles integrados satisface las necesidades específicas, se pueden crear roles personalizados para proporcionar un acceso más pormenorizada a entidades de API Management.

Protección con directivas

- Eliminar encabezados de respuesta

  <set-header name="X-Powered-By" exists-action="delete" />
- Reemplazo de URL originales en el cuerpo de la respuesta de API con las URL de puerta de enlace de APIM

  <redirect-content-urls />
- Protección de una API agregando límites de frecuencia

  <rate-limit-by-key calls="3" renewal-period="15" counter-key="@(context.Subscription.Id)" />

  <rate-limit-by-key calls="10" renewal-period="60" counter-key="@(context.Request.IpAddress)" />

  <quota-by-key calls="1000000" bandwidth="10000" renewal-period="2629800" counter-key="@(context.Request.IpAddress)" />

Versiones y revisiones

- Es útil explicar primero lo que significa estos términos, ya que son dos características distintas, pero relacionadas:
  - Las versiones permiten presentar grupos de APIs relacionadas a los desarrolladores. Las versiones se diferencian por un número de versión (que es una cadena de cualquier valor que se elija) y un esquema de versiones (ruta, cadena de consulta o encabezado).
  - Las revisiones permiten realizar cambios en las APIs de forma controlada y segura. Cuando se desee realizar cambios, se crea una nueva revisión. Luego se puede editar y probar la API sin molestar a los consumidores de APIs. Cuando esté lista, se puede actualizar la revisión; al mismo tiempo, se puede publicar una entrada en el nuevo registro de cambios, para mantener a los consumidores de APIs actualizados con los cambios.

Versiones y revisiones

- Con las versiones se puede:
  - Publicar varias versiones de la API al mismo tiempo
  - Usar la ruta / cadena de consulta o encabezado para diferenciar entre versiones.
  - Utilizar cualquier valor de cadena que se desee para identificar la versión (un número, una fecha, un nombre).
  - Mostrar las versiones de APIs agrupadas en el portal para desarrolladores.
- Con las revisiones se puede:
  - Realizar cambios de forma segura en API Management de las definiciones y políticas de la API, sin perturbar la API de producción.
  - Probar los cambios antes de publicarlos.
  - Documentar los cambios que se realicen, para que los desarrolladores puedan comprender las novedades.
  - Revertir si se encuentran problemas.

Revisiones

- Las revisiones se crean desde la solapa ***Revisions*** del editor de APIs o los … asociados a la API. Todas las revisiones se numeran automáticamente y se puede añadir un comentario al Change log al crearlas.
- En el editor de APIs, el selector de revisiones (que está inmediatamente encima de la pestaña de diseño) permite indicar que revisión se esta editando, los cambios que se realicen solo afectaran a la revisión seleccionada.
- Solo una revisión estará marcada como CURRENT y conserva la URL por defecto de la API, al restos se le añade el sufijo ";rev*<numRevisión>*" automáticamente. Las revisiones marcadas como ONLINE aceptaran peticiones, con Take offline se podrá cambiar el estado.
- Una vez terminadas las pruebas y confirmaciones, una revisión se puede pasar a actual con Make current, toma la URL por defecto y la obsoleta debe empezar a usar el sufijo en su URL. Se pueden revertir los cambios rápidamente volviendo a hacer actual a la revisión anterior.
- Una revisión, si cuenta con cambios rupturistas, permite crear una nueva versión.
- Las revisiones obsoletas se pueden eliminar, pero no tiene vuelta atras.

Versiones

- Las nuevas versiones se crean con "Add version" desde los … asociados a la API. Al crear la nueva versión hay que dar un nuevo nombre único, seleccionar un esquema de control de versiones: ruta, encabezado o cadena de consulta, asignar un identificador de versión, y se puede registrar la nueva versión en diversos productos. Si el esquema de control es Encabezado o Cadena de consulta hay que proporcionar el nombre del parámetro de encabezado o de la cadena de consulta.
- Después de crear la versión, en la lista de APISs, se desdobla la API en Original (si no se creo con control de versiones) y la nueva versión o sucesivas versiones.
- A partir de este momento se gestionan como si fueran APIs diferentes que comparten el prefijo de la URL.
- En la nueva versión, con la solapa ***Settings*** habrá que cambiar la configuración de backend para que apunte a la implementación de la nueva versión.

Supervisión

- Visualización de las métricas de las API
- Registros de actividad
- Registros del recurso
- Visualización de datos de diagnóstico en Azure Monitor
- Configuración de una regla de alerta

Visualización de las métricas de las APIs

- API Management emite métricas cada minuto, lo que le ofrece visibilidad casi en tiempo real sobre el estado y el mantenimiento de las APIs. Las siguientes son las dos métricas que se usan con más frecuencia.
  - Capacity (Capacidad): ayuda a tomar decisiones acerca de cómo actualizar o degradar los servicios de APIM. La métrica se emite por minuto y refleja la capacidad de la puerta de enlace en el momento del informe. La métrica oscila entre 0 y 100, y se calcula en función de los recursos de la puerta de enlace, como el uso de la CPU y de la memoria.
  - Requests (Solicitudes): ayuda a analizar el tráfico de API que pasa por los servicios API Management. La métrica se emite por minuto y notifica el número de solicitudes de puerta de enlace con dimensiones que incluyen los códigos de respuesta, la ubicación, el nombre de host y los errores.

Registros de actividad y del recurso

- Los registros de actividad proporcionan información sobre las operaciones llevadas a cabo en los servicios API Management. Con los registros de actividades, puede determinar los interrogantes “qué, quién y cuándo” de las operaciones de escritura (PUT, POST, DELETE) llevadas a cabo en los servicios API Management. Los registros de actividad no incluyen las operaciones de lectura (GET) ni las realizadas en Azure Portal o mediante las APIs de administración originales.
- Los registros de recurso proporcionan información valiosa acerca de las operaciones y los errores que son importantes para la auditoría, así como para solucionar problemas. Los registros de recurso son diferentes de los registros de actividad. El registro de actividad proporciona información detallada sobre las operaciones llevadas a cabo en los recursos de Azure. Los registros de recurso proporcionan información detallada sobre las operaciones que realiza el recurso.

Azure Monitor

- Azure Monitor ayuda a maximizar la disponibilidad y el rendimiento de las aplicaciones y los servicios. Ofrece una solución completa para recopilar, analizar y actuar en la telemetría desde los entornos local y en la nube. Esta información le ayudará a conocer el rendimiento de las aplicaciones y a identificar de manera proactiva los problemas que les afectan y los recursos de los que dependen.
- Si se habilita la recopilación de registros de GatewayLogs o métricas en un área de trabajo de Log Analytics, los datos pueden tardar unos minutos en aparecer en Azure Monitor.
- Se pueden recibir alertas basadas en métricas y registros de actividad. Azure Monitor permite configurar una alerta que haga lo siguiente cuando se desencadena:
  - Enviar una notificación por correo electrónico
  - Llamar a un webhook
  - Invocar una aplicación lógica de Azure
- Para configurar una regla de alerta de ejemplo basada en una métrica de solicitud:
  - En Azure Portal → Alerts → New alert rule.

Observabilidad

- La observabilidad es la capacidad de entender el estado interno de un sistema a partir de los datos que genera y la capacidad de explorar esos datos para responder a las preguntas sobre qué sucedió y por qué.
- Azure API Management ayuda a las organizaciones a centralizar la administración de todas las APIs. Dado que actúa como un único punto de entrada de todo el tráfico de las APIs, es un lugar ideal para observar estas.
- En siguiente lista es un resumen todas las herramientas que administra API Management para observar las APIs, cada una de ellas es útil en uno o varios escenarios:
  - Built-in Analytics (Análisis integrado)
  - API Inspector
  - Azure Monitor Metrics
  - Azure Monitor Logs
  - Azure Application Insights
  - Registro a través de Azure Event Hub

Cache

- Las APIs y operaciones de API Management pueden configurarse con almacenamiento en caché de respuesta. El almacenamiento en caché de respuesta puede reducir considerablemente la latencia de los llamadores de APIs y la carga de back-end de los proveedores de APIs.
- La caché integrada es limitada, volátil y se comparte entre todas las unidades de la misma región del mismo servicio de API Management.
- La cache se gestiona mediante la directivas de almacenamiento en caché.
- Además de utilizar la memoria caché integrada, Azure API Management permite almacenar en caché las respuestas en una memoria caché compatible con Redis como Azure Cache for Redis.
- El uso de una caché externa permite superar algunas limitaciones de la caché integrada:
  - Evitar que la memoria caché se borre periódicamente durante las actualizaciones de API Management
  - Tener más control sobre la configuración de la memoria caché
  - Almacenar en memoria caché más datos de los que permite el plan de API Management
  - Usar el almacenamiento en caché con el nivel de consumo de API Management
  - Habilitar el almacenamiento en caché tal como se explica para las puertas de enlace autohospedadas de API Management

Aprovisionamiento y escalado

- Azure API Management no aplicará ninguna limitación de nivel de servicio para evitar una sobrecarga física de las instancias. Cuando una instancia alcanza su capacidad física, se comporta de forma similar a cualquier servidor web sobrecargado que no puede procesar las solicitudes entrantes: la latencia aumentará, se eliminarán las conexiones, se producirán errores de tiempo de espera, etc. Es decir, los clientes de APIs deben estar preparados para hacer frente a esta posibilidad de manera similar a como ocurre con cualquier otro servicio externo (por ejemplo, mediante la aplicación de directivas de reintento).
- La capacidad es un indicador de la carga de una instancia de API Management. Refleja el uso de los recursos (CPU y memoria) y las longitudes de cola de la red.
- Para escalar una instancia de Azure API Management, los clientes pueden agregar o quitar unidades. Una unidad se compone de recursos de Azure dedicados y tiene cierta capacidad de carga, que se expresa mediante el número de llamadas API que se realizan cada mes. Dicho número no representa un límite de llamadas, sino un valor de rendimiento máximo que permite el planeamiento de la capacidad aproximada. El rendimiento y la latencia reales varían considerablemente en función de factores como el número y la tasa de conexiones concurrentes, el tipo y número de directivas configuradas, los tamaños de las solicitudes y respuestas y la latencia del back-end.
- Después de comenzar a usar los recursos de API Management, se utilizan las características de Cost Management para establecer presupuestos y supervisar los costos. También se puede revisar los costos previstos e identificar las tendencias de gasto para identificar las áreas en las que podría querer actuar. Los costos de API Management son solo una parte de los costos mensuales de la factura de Azure.

Nombre de dominio personalizado

- Cuando se crea una instancia del servicio Azure API Management, Azure le asigna un subdominio de azure-api.net (por ejemplo, apim-service-name.azure-api.net). Sin embargo, los puntos de conexión de APIM se pueden exponer con un nombre de dominio personalizado, como mi.dominio.com, como un servicio adicional.
- API Management solo acepta solicitudes con valores de encabezado de host que coinciden con el nombre de dominio predeterminado o con cualquiera de los nombres de dominio personalizados configurados.
- Una de las reglas básicas es proteger todas las comunicaciones con SSL/TLS (https), lo cual requiere certificados vinculados al dominio del servidor.
- Los clientes que quieran usar la asignación de certificados para mejorar la seguridad de sus aplicaciones deben usar un nombre de dominio personalizado y un certificado propio que administren, no el predeterminado. Si utilizar el certificado predeterminado, en lugar de uno propio, tendrán una gran dependencia de las propiedades de un certificado que no controlan, lo que no es un procedimiento recomendado.

Puerta de enlace autohospedada

- La característica de puerta de enlace autohospedada amplía la compatibilidad de API Management con entornos híbridos y de varias nubes y permite a las organizaciones administrar de forma eficaz y segura las APIs hospedadas localmente y en diferentes nubes desde un único servicio de API Management en Azure.
- Con la puerta de enlace autohospedada, los clientes tienen la flexibilidad de implementar una versión en contenedor del componente de puerta de enlace de API Management en los mismos entornos donde hospedan sus APIs. Todas las puertas de enlace autohospedadas se administran desde el servicio de API Management con el que están federadas, lo que proporciona a los clientes visibilidad y una experiencia de administración unificada en todas las APIs internas y externas. La colocación de las puertas de enlace cerca de las APIs permite a los clientes optimizar los flujos de tráfico de la API y abordar los requisitos de seguridad y cumplimiento.
- Cada servicio de API Management consta de los siguientes componentes clave:
  - Plano de administración, expuesto como una API, que se usa para configurar el servicio mediante Azure Portal, PowerShell y otros mecanismos compatibles.
  - La puerta de enlace (o el plano de datos) es responsable de redirigir mediante proxy las solicitudes de API y también de la aplicación de directivas y la recopilación de datos de telemetría.
  - Portal para desarrolladores que les permite descubrir, conocer y usar las APIs.

Back-end

- Un back-end (o back-end de API) de API Management es un servicio HTTP que implementa la API de front-end y sus operaciones.
- Al importar ciertas APIs, API Management configura el back-end de la API automáticamente. Por ejemplo, API Management configura el back-end al importar una especificación de OpenAPI, API de SOAP o recursos de Azure como una aplicación de funciones de Azure o una aplicación lógica desencadenada mediante HTTP.
- API Management también admite el uso de otros recursos de Azure, como un clúster de Service Fabric o un servicio personalizado, como back-end de API. El uso de estos back-ends personalizados requiere una configuración adicional, por ejemplo, para autorizar las credenciales de las solicitudes al servicio back-end y para definir las operaciones de la API. Estos back-ends se configuran y administran en Azure Portal o mediante las APIs o las herramientas de Azure.
- Después de crear un back-end, se puede hacer referencia a la dirección URL del back-end en las APIs. Se utiliza la directiva set-backend-service para redirigir una solicitud de API entrante al back-end personalizado en lugar de al back-end predeterminado para esa API.
- Un back-end personalizado tiene varias ventajas, entre las que se incluyen:
  - Ofrece un resumen de la información sobre el servicio back-end y promueve la reutilización entre API y una gobernanza mejorada.
  - Fácil de usar mediante la configuración de una directiva de transformación en una API existente.
  - Aprovecha la funcionalidad de API Management para mantener secretos en Azure Key Vault si los valores con nombre están configurados para la autenticación de parámetros de consulta o encabezado.

Tutoriales

- https://docs.microsoft.com/es-es/learn/browse/?expanded=others&products=azure-api-management
- Apis
  - git clone https://github.com/MicrosoftDocs/mslearn-control-authentication-with-apim.git
  - cd mslearn-control-authentication-with-apim
  - bash setup.sh
  - git clone https://github.com/MicrosoftDocs/mslearn-publish-manage-apis-with-azure-api-management.git
  - cd mslearn-publish-manage-apis-with-azure-api-management
  - bash setup.sh
  - git clone https://github.com/MicrosoftDocs/mslearn-protect-apis-on-api-management.git
  - cd mslearn-protect-apis-on-api-management
  - bash setup.sh
- Funciones
  - git clone https://github.com/MicrosoftDocs/mslearn-apim-and-functions.git ~/OnlineStoreFuncs
  - cd ~/OnlineStoreFuncs
  - bash setup.sh

AZURE SERVICE BUS

Introducción

- Microsoft Azure Service Bus es un agente de mensajes empresarial totalmente administrado que incluye colas de mensajes y temas que se pueden publicar y a los que es posible suscribirse.
- Service Bus se usa para desacoplar aplicaciones de servicios, lo que proporciona las siguientes ventajas:
  - Equilibrio de carga del trabajo entre trabajadores paralelos
  - Enrutamiento y transferencia de datos de forma segura y control entre los límites de aplicaciones y servicios
  - Coordinación del trabajo transaccional que requiere un alto grado de confiabilidad

Bibliotecas de clientes

- Azure Service Bus se puede gestionar desde el Portal de Azure, Azure PowerShell, Azure CLI y plantillas ARM.
- Todas bibliotecas cliente de Service Bus están disponibles a través de Azure SDK.
  - Azure Service Bus para .NET (Azure.Messaging.ServiceBus, Microsoft.Azure.ServiceBus)
  - Bibliotecas de Azure Service Bus para Java y Proveedor de Azure Service Bus para Java JMS 2.0
  - Módulos de Azure Service Bus para JavaScript y TypeScript
  - Bibliotecas de Azure Service Bus para Python
- El protocolo principal de Azure Service Bus es AMQP 1.0 y se puede usar desde cualquier cliente compatible con este protocolo.
  - Java — Apache Qpid Proton-J
  - C/C++ — Azure uAMQP C, Apache Qpid Proton-C
  - Python — Azure uAMQP for Python, Apache Qpid Proton Python
  - PHP — Azure uAMQP for PHP
  - Ruby — Apache Qpid Proton Ruby
  - Go — Azure Go AMQP, Apache Qpid Proton Go
  - JavaScript/Node — Rhea

Comunicación asíncrona

- En determinadas ocasiones, los servicios deben integrarse con otros actores, componentes o sistemas internos y externos, siendo necesario aportar o recibir información de ellos. En la mayoría de los casos, estas comunicaciones tienen que estar permanentemente disponibles, ser rápidas, seguras, asíncronas y fiables entre otros requisitos.
- Las colas de mensajes (MQ) solucionan estas necesidades, actuando de intermediario entre emisores y destinatarios, o en un contexto más definido, productores y consumidores de mensajes. Las colas de Service Bus son compatibles con el modelo de comunicación de mensajería asíncrona.
- Cuando se usan colas, los componentes de una aplicación distribuida no se comunican directamente entre sí, sino que intercambian mensajes a través de una cola, que actúa como un intermediario (agente). El productor del mensaje (remitente) manda un mensaje a la cola y, a continuación sigue con su procesamiento. De forma asíncrona, el destinatario del mensaje (receptor) extrae el mensaje de la cola y lo procesa.
- El productor no tiene que esperar una respuesta del destinatario para continuar el proceso y el envío de más mensajes. Las colas ofrecen una entrega de mensajes según el modelo El primero en entrar es el primero en salir (FIFO) a uno o más destinatarios de la competencia. Es decir, normalmente los receptores reciben y procesan los mensajes en el orden en el que se agregaron a la cola y solo un destinatario del mensaje recibe y procesa cada uno de los mensajes.

Comunicación asíncrona

- Se pueden usar para reducir las cargas y los tiempos de entrega por parte de los servicios, ya que las tareas, que normalmente tardarían bastante tiempo en procesarse, se pueden delegar a un tercero cuyo único trabajo es realizarlas.
- El uso de colas de mensajes también es bueno cuando se desea distribuir un mensaje a múltiples destinatarios. Además aportan otros beneficios como:
  - Garantía de entrega y orden: Los mensajes se consumen, en el mismo orden que se llegaron a la cola, y son consumidos una única vez
  - Redundancia: Las colas persisten los mensajes hasta que son procesados por completo
  - Desacoplamiento: Siendo capas intermedias de comunicación entre procesos, aportan la flexibilidad en la definición de arquitectura de cada uno de ellos de manera separada, siempre que se mantenga una interfaz común
  - Escalabilidad: Con más unidades de procesamiento, las colas balancean su respectiva carga
- Los datos se transfieren entre distintas aplicaciones y servicios mediante mensajes. Un mensaje es un contenedor decorado con metadatos que contiene datos. Los datos pueden ser cualquier tipo de información, como datos estructurados codificados con los formatos comunes, como los siguientes: JSON, XML, Apache Avro, texto sin formato.

Espacios de nombres

- Un espacio de nombres es un contenedor para todos los componentes de mensajería. Varias colas y temas pueden estar en un solo espacio de nombres, y los espacios de nombres suelen servir como contenedores de aplicación.
- Un espacio de nombres de Service Bus es un segmento de capacidad de un clúster grande compuesto por docenas de máquinas virtuales activas. Opcionalmente, puede abarcar tres zonas de disponibilidad de Azure. Por lo tanto, obtiene todas las ventajas de disponibilidad y solidez de la ejecución del agente de mensajes a gran escala.
- No es necesario preocuparse por las complejidades subyacentes, Service Bus es la mensajería "sin servidor".

Creación de un espacio de nombres

- Para crear un espacio de nombres en el Portal de Azure, en el panel de navegación izquierdo del portal, seleccione sucesivamente + Crear un recurso, Integración y Service Bus + Crear espacio de nombres.
  - Nombre para el espacio de nombres
  - Plan de tarifa (Básico, Estándar o Premium)
    - unidades de mensajería, redundancia de zona
  - Suscripción
  - Grupo de recursos
  - Ubicación

Cadena de conexión

- La creación un nuevo espacio de nombres genera automáticamente RootManageSharedAccessKey una regla de firma de acceso compartido (SAS) inicial con un par asociado de claves principal y secundaria en el que ambas conceden control total sobre todos los aspectos del espacio de nombres.
- En Directivas de acceso compartido se pueden crear nuevas en Agregar directiva SAS
  - Nombre de directiva
  - Permiso para Administrar o Enviar o Escuchar
- Cada directiva tiene asociada una Cadena de conexión principal y otra secundaria que permiten a los emisores y receptores acceder a las colas y temas.

```
// Create a ServiceBusClient that will authenticate through Active Directory
ServiceBusClient client = new ServiceBusClient("yournamespace.servicebus.windows.net", new
        DefaultAzureCredential());
// Create a ServiceBusClient that will authenticate using a connection string
ServiceBusClient srv = new ServiceBusClient("<connection_string>");
```


Colas

- Los mensajes se envían y se reciben desde colas. Las colas almacenan mensajes hasta que la aplicación receptora está disponible para recibirlos y procesarlos.
- Los mensajes de las colas se ordenan y se les asigna una marca de tiempo a su llegada. Una vez aceptados por el agente, el mensaje siempre se mantiene de forma duradera en un almacenamiento. Service Bus nunca deja mensajes en memoria o almacenamiento volátil una vez que se han notificado al emisor como aceptados.
- Los mensajes se entregan en modo de extracción y solo se entregan mensajes cuando se solicitan. Una vez extraído de la cola, el mensaje se elimina de la cola y no puede volver a ser extraído por lo que solo es procesado por un único receptor.
- Uno o varios emisores pueden dejar mensajes en la cola. Uno o varios receptores pueden sondear la cola, pero solo uno podrá extraer y procesar el mensaje, esto permite que la carga de trabajo del tratamiento de los mensajes se pueda repartir entre varios receptores.
- La operación de extracción puede ser de larga duración (long polling) y completarse solamente una vez que un mensaje esté disponible.

Creación de una cola

- Nombre
- Número máximo de entregas
- Período de vida del mensaje: Días, Horas, Minutos, Segundos
- Duración del bloqueo: Días, Horas, Minutos, Segundos
- Habilitar eliminación automática en la cola inactiva
- Habilitar detección de duplicados
- Habilitar cola de mensajes fallidos al expirar el mensaje
- Habilitar la creación de particiones
- Habilitar sesiones
- Reenviar mensajes a la cola/tema

Creación de una cola

- El período de vida del mensaje determina cuánto tiempo permanecerá el mensaje en la cola hasta que expire y se quite o se ponga en una cola de mensajes fallidos. Al enviar mensajes, se puede especificar un período de vida distinto para un mensaje concreto. El valor predeterminado se usará para todos los mensajes de la cola para los que no se especifique ningún período de vida.
- La duración del bloqueo establece la cantidad de tiempo que un mensaje permanecerá bloqueado para otros receptores mientras el extractor confirma el tratamiento. Una vez expirado el bloqueo, un mensaje extraído por un receptor aparece disponible para que otros receptores lo puedan extraer.
- La característica de reenvío automático encadena una cola o suscripción a otra cola o tema dentro del mismo espacio de nombres. Al usar esta característica, Service Bus mueve automáticamente los mensajes de una cola o suscripción a una cola o tema de destino. Todos estos movimientos se realizan de manera transaccional.

Creación de una cola

- Habilitar la detección de duplicados configura la cola para que conserve un historial de todos los mensajes que se envían a la cola durante un periodo de tiempo configurable. Durante ese intervalo, la cola no aceptará mensajes duplicados. Habilitar esta propiedad garantiza una entrega única en un intervalo de tiempo definido por el usuario.
- Las sesiones de Service Bus permiten el control ordenado de secuencias ilimitadas de mensajes relacionados. Con sesiones habilitadas, una cola puede garantizar la entrega de mensajes primero en entrar, primero en salir.
- Crea particiones en una cola en varios agentes y almacenes de mensajes. Desconecta el rendimiento general de una entidad con particiones de cualquier agente o almacén de mensajes. Esta propiedad no se puede modificar una vez creada una cola.

Mensajes

- Los mensajes llevan una carga, así como metadatos, en forma de propiedades de par clave-valor, que describen la carga y ofrecen instrucciones de control a Service Bus y a las aplicaciones. En ocasiones, los metadatos por si solos son suficientes para transportar la información que el remitente desea comunicar a los receptores, y solo la carga permanece vacía.
- Un mensaje de Service Bus consta de una sección de carga binaria (body) y dos conjuntos de propiedades. Las propiedades predefinidas controlan la funcionalidad en el nivel de mensaje en el agente o se asignan a los elementos de metadatos comunes y estandarizados. Las propiedades de usuario son una colección de pares clave-valor que la aplicación puede definir y establecer.
- Al crear el mensaje se pueden asignar valores a las propiedades de sistema y se pueden añadir nuevas propiedades con sus correspondientes valores. Al recibir el mensaje se puede acceder al cuerpo del mensaje o a cualquiera de sus propiedades, de sistema o personalizadas.
- Las propiedades se pueden utilizar en el proceso de filtrado de las suscripciones:

  sys.To = 'Alumnos' AND categoria IN ('Avanzado', 'Medio')
- Cuando está en tránsito o almacenada en Service Bus, la carga siempre es un bloque opaco y binario. La propiedad ContentType permite que las aplicaciones utilicen el formato mas apropiado para enviar la carga e indiquen al receptor el formato utilizado, el formato sugerido para los valores de la propiedad son los descriptores de tipo de medio MIME.

Propiedades de enrutamiento y correlación

- **MessageId** (message-id): El identificador del mensaje es un valor definido por la aplicación que identifica de forma única el mensaje y su carga. El identificador es una cadena de forma libre y puede reflejar un GUID o un identificador que se deriva del contexto de la aplicación. Si está habilitada, la característica de detección de duplicados identifica y quita el segundo y los siguientes envíos de mensajes con la misma clase MessageId.
- **CorrelationId** (correlation-id): Permite que una aplicación especifique un contexto del mensaje con fines de correlación, por ejemplo, que refleje el elemento MessageId de un mensaje que ha respondido.
- **SessionId** (group-id): Para entidades que tienen en cuenta la sesión, este valor definido por la aplicación especifica la afiliación de sesión del mensaje. Los mensajes con el mismo identificador de sesión están sujetas al bloqueo de resumen y habilitan el procesamiento en orden exacto y la desmultiplexación. Para las entidades sin reconocimiento de la sesión, este valor se ignora.
- **To** (to): Esta propiedad está reservada para un uso futuro en escenarios de enrutamiento y actualmente la ignora el propio agente. Las aplicaciones pueden utilizar este valor en escenarios de encadenamiento de reenvío automático controlados por reglas para indicar el destino lógico previsto del mensaje.
- **ReplyTo** (reply-to): Este valor opcional y definido por la aplicación es un método estándar de expresar una ruta de acceso de respuesta al receptor del mensaje. Cuando un remitente espera una respuesta, establece el valor en la ruta de acceso absoluta o relativa de la cola o tema al que espera que se envíe la respuesta.
- **ReplyToSessionId** (reply-to-group-id): Este valor aumenta la información de ReplyTo y especifica qué propiedad SessionId debe establecerse para la respuesta cuando se envió a la entidad de respuesta.

Propiedades de configuración

- **Label** (subject): Esta propiedad permite a la aplicación indicar el propósito del mensaje al receptor de modo estandarizado, similar a una línea de asunto de correo electrónico.
- **ContentType** (content-type): Opcional, formato MIME de la carga del mensaje: text/plain, application/json, …
- **ScheduledEnqueueTimeUtc**: Para los mensajes que solo están disponibles para su recuperación después de un retraso, esta propiedad define el instante en hora UTC en el que el mensaje se va a poner lógicamente en cola, se secuenciará y, por tanto, estará disponible para la recuperación.
- **TimeToLive**: Este valor es la duración relativa tras la cual expira el mensaje, desde el instante en que el mensaje se ha aceptado y lo ha almacenado el agente, tal y como se captura en EnqueueTimeUtc. Por defecto usa DefaultTimeToLive y, si TimeToLive es mayor, seguirá usando el valor por defecto.
- **ForcePersistence**: Para las colas o temas con la marca EnableExpress establecida, esta propiedad se puede establecer para indicar que se debe conservar el mensaje en el disco antes de que se confirme. Este es el comportamiento estándar para todas las entidades no rápidas.
- **PartitionKey**: En el caso de entidades con particiones, la configuración de este valor permite asignar mensajes relacionados a la misma partición interna, por lo que el orden de la secuencia de envío se registra correctamente. La partición la elige una función hash sobre este valor y no se puede seleccionar directamente. En el caso de entidades que tienen en cuenta la sesión, la propiedad SessionId reemplaza este valor.
- **ViaPartitionKey**: Si se envía un mensaje a través de una cola de transferencia en el ámbito de una transacción, este valor selecciona la partición de la cola de transferencia.

Propiedades de solo lectura

- **SequenceNumber**: El número de secuencia es un entero de 64 bits único asignado a un mensaje cuando el agente lo acepta y lo almacena, y funciona como su verdadero identificador. Los números de secuencia aumentan de forma continuada.
- **Size**: Refleja el tamaño almacenado del mensaje en el registro de agente como un recuento de bytes, tal y como tiene en cuenta para la cuota de almacenamiento.
- **State**: Indica el estado del mensaje en el registro. Esta propiedad solo es pertinente durante la exploración de mensajes ("inspección"), para determinar si un mensaje está "activo" y disponible para su recuperación cuando llega a la parte superior de la cola, con independencia de si está aplazada o de si está esperando su programación.
- **DeliveryCount**: Número de entregas que se han intentado para este mensaje. El recuento se incrementa cuando expira un bloqueo del mensaje o si el mensaje lo abandona explícitamente el receptor.
- **DeadLetterSource**: Solo se establece en aquellos mensajes fallidos y posteriormente reenviados automáticamente de la cola de mensajes fallidos a otra entidad. Indica la entidad donde se encontraba el mensaje fallido.
- **EnqueuedSequenceNumber**: Para los mensajes que se reenvían de manera automática, esta propiedad refleja el número de secuencia que primero se había asignado al mensaje en su momento del envío original.
- **EnqueuedTimeUtc**: El instante en hora UTC en el que el mensaje se ha aceptado y almacenado en la entidad. Este valor puede usarse como un indicador de hora de llegada autoritativo y neutro cuando el receptor no desea confiar en el reloj del remitente.
- **ExpiresAtUtc** (absolute-expiry-time): Instante en hora UTC en el que el mensaje se marca para su eliminación y ya no está disponible para la recuperación de la entidad debido a su expiración.
- **LockedUntilUtc**: Para los mensajes que se recuperan en un bloqueo (modo de recepción de bloque de inspección, no liquidado previamente), esta propiedad refleja el instante en hora UTC hasta que el mensaje se mantenga bloqueado en la cola o suscripción. Cuando expira el bloqueo, la propiedad DeliveryCount se incrementa y el mensaje está disponible para la recuperación.
- **LockToken**: El token de bloqueo es una referencia al bloqueo utilizado por el agente en el modo de recepción de bloque de inspección. El token puede usarse para anclar el bloqueo de forma permanente y sacar el mensaje del flujo de estado de entrega normal.

Creación de un emisor Microsoft.Azure.ServiceBus

```
string[] sessionIds = { Guid.NewGuid().ToString(), Guid.NewGuid().ToString(),
Guid.NewGuid().ToString() };
IQueueClient queueClient = new QueueClient(ServiceBusConnectionString, QueueOrTopicName);

for (int i = 0; i < NumeroDeEnvios; i++)
    try {
        string messageBody = $"Mensaje {DateTime.Now:MM:ss:ff}: valor {rnd.Next(1, 100)}.";
        var message = new Message(Encoding.UTF8.GetBytes(messageBody));
        message.MessageId = $"{DateTime.Now.Minute}-{i}";
        message.SessionId = sessionIds[i % 3];
        if (TimeToLive > 0)
            message.TimeToLive = TimeSpan.FromSeconds(TimeToLive);
        await queueClient.SendAsync(message);
        Console.WriteLine($"Sending message: {message.MessageId} - {messageBody}");
    } catch (Exception exception) {
        Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
    }

await queueClient.CloseAsync();
Console.WriteLine("Message was sent successfully.");
```

Creación de un receptor Microsoft.Azure.ServiceBus

```
IQueueClient queueClient = new QueueClient(ServiceBusConnectionString, QueueOrTopicName);

Console.WriteLine("======================================================");
Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("======================================================");

queueClient.RegisterMessageHandler(async (message, token) => {
    Console.WriteLine($"Received message: SequenceNumber: {message.SystemProperties.SequenceNumber}
        Body: {Encoding.UTF8.GetString(message.Body)}");
    if (AbandonarMultiplosDe > 0 && message.SystemProperties.SequenceNumber % AbandonarMultiplosDe ==
        0) {
        await queueClient.AbandonAsync(message.SystemProperties.LockToken);
    } else
        await queueClient.CompleteAsync(message.SystemProperties.LockToken);
  }, new MessageHandlerOptions(ExceptionReceivedHandler) {
      MaxConcurrentCalls = 1,
      AutoComplete = false
  });
Console.Read();
await queueClient.CloseAsync();
```


Creación de un receptor Microsoft.Azure.ServiceBus

```
Task ExceptionReceivedHandler(ExceptionReceivedEventArgs ex) {
    Console.WriteLine($"Message handler encountered an exception
         {ex.Exception}.");
    var context = ex.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
```

Creación de un emisor Azure.Messaging.ServiceBus

```
ServiceBusSender sender = srv.CreateSender(QueueOrTopicName);
try {
    string messageBody = $"Mensaje {DateTime.Now:MM:ss:ff}: valor {rnd.Next(1, 100)}.";
    var item = new Item() { Id = 1, Mensaje = messageBody };
    var message = item.AsMessage();
    message.To = item.Categoria;
    message.Subject = item.Nivel;
    message.ApplicationProperties.Add("nivel", item.Nivel);
    message.MessageId = $"{DateTime.Now.Minute}-1";
    message.SessionId = Guid.NewGuid().ToString();
    if (TimeToLive > 0)
        message.TimeToLive = TimeSpan.FromSeconds(TimeToLive);
    await sender.SendMessageAsync(message);
    Console.WriteLine($"Sending message: {message.MessageId} - {item}");
} catch (Exception exception) {
    Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
}

await sender.CloseAsync();
Console.WriteLine("Messages was sent successfully.");
```


Creación de un receptor Azure.Messaging.ServiceBus

```
Console.WriteLine("============================================================");
Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("============================================================");
var options = new ServiceBusProcessorOptions {
    AutoCompleteMessages = false
};
await using ServiceBusProcessor processor = string.IsNullOrWhiteSpace(SubscriptionName) ?
    srv.CreateProcessor(QueueOrTopicName, options) :
    srv.CreateProcessor(QueueOrTopicName, SubscriptionName, options);
processor.ProcessMessageAsync += async (arg) => {
    Console.WriteLine($"Received message: SequenceNumber: {arg.Message.SequenceNumber} Body:
        {Encoding.UTF8.GetString(arg.Message.Body)}");
    if (AbandonarMultiplosDe > 0 && arg.Message.SequenceNumber % AbandonarMultiplosDe == 0) {
        await arg.AbandonMessageAsync(arg.Message);
    } else
        await arg.CompleteMessageAsync(arg.Message);
};
processor.ProcessErrorAsync += ExceptionReceivedHandler;
await processor.StartProcessingAsync();
Console.Read();
Console.WriteLine("Exit processor...");
```

Creación de un receptor Azure.Messaging.ServiceBus

```
Task ExceptionReceivedHandler(ProcessErrorEventArgs ex) {
    Console.WriteLine($"Message handler encountered an
        exception {ex.Exception}.");
    Console.WriteLine("Exception context for
        troubleshooting:");
    Console.WriteLine($"- ErrorSource:
        {ex.ErrorSource}");
    Console.WriteLine($"- Entity Path: {ex.EntityPath}");
    Console.WriteLine($"- Fully Qualified Namespace:
        {ex.FullyQualifiedNamespace}");
    return Task.CompletedTask;
}
```


Limpieza automática

- La limpieza automática resulta útil en escenarios de desarrollo y pruebas en los que las entidades se crean dinámicamente y no se limpian tras su uso debido a alguna interrupción de la ejecución de la depuración o prueba. También es útil cuando una aplicación crea entidades dinámicas, como una cola de respuestas, para volver a recibir respuestas en un proceso de servidor web, o en otro objeto de duración relativamente corta donde resulta difícil limpiar esas entidades de forma confiable cuando la instancia del objeto desaparece.
- La eliminación automática en modo inactivo le permite especificar un intervalo de inactividad después del cual se elimina automáticamente la suscripción a una cola o tema. La duración mínima es de 5 minutos.
- Se considera inactividad:
  - Colas: No envía, No recibe, No hay actualizaciones a la cola, No hay mensajes programados, No hay navegación o inspección
  - Temas: No envía, No hay actualizaciones al tema, No hay mensajes programados
  - Suscripciones: No recibe, No hay actualizaciones a la suscripción, No se han agregado nuevas reglas a la suscripción, No hay navegación o inspección

Temas

- Los temas permiten enviar y recibir mensajes, pero el mismo mensaje podrá ser tratado por varios receptores. Mientras que una cola se utiliza a menudo para la comunicación punto a punto, los temas son útiles en escenarios de publicación y suscripción.
- Las suscripciones y los temas de Service Bus son compatibles con el modelo de comunicación de mensajería de publicación/suscripción. Cuando se usan temas y suscripciones, los componentes de una aplicación distribuida no se comunican directamente entre sí, sino que intercambian mensajes a través de un tema, que actúa como un intermediario.
- Los temas pueden tener varias suscripciones independientes asociadas; por lo demás, funcionan exactamente igual que las colas del lado receptor. Un suscriptor a un tema puede recibir una copia de cada mensaje enviado a ese tema. Las suscripciones son entidades con nombre. De forma predeterminada, las suscripciones son duraderas, pero se pueden configurar para que expiren y se eliminen luego automáticamente.
- De cara a los emisores el tema se comporta como si fuera una única cola. De cara a los receptores, el tema se comporta como si fueran múltiples colas, una por suscripción.

Suscripciones

- A diferencia de las colas de Service Bus, en las que un solo destinatario procesa cada mensaje, los temas y las suscripciones proporcionan una forma de comunicación de uno a varios mediante un patrón de publicación/suscripción.
- Es posible registrar varias suscripciones en un tema. Cuando un mensaje se envía a un tema, pasa a estar disponible para cada suscripción para la administración o el procesamiento de manera independiente.
- Una suscripción a un tema se asemeja a una cola virtual que recibe copias de los mensajes que se enviaron al tema. Opcionalmente, puede registrar reglas de filtros para un tema por suscripción, lo que le permite filtrar o restringir qué mensajes para un tema reciben las suscripciones a un tema.
- Las suscripciones y temas de Service Bus permiten escalar para realizar el procesamiento de un número elevado de mensajes en una serie amplia de usuarios y aplicaciones.

Filtros

- Puede definir reglas en una suscripción. Una regla de suscripción tiene un filtro para definir una condición del mensaje que se va a copiar en la suscripción, y una acción opcional que puede modificar los metadatos del mensaje. Esta característica es útil en los escenarios siguientes:
  - No quiere que una suscripción reciba todos los mensajes enviados a un tema.
  - Quiere marcar los mensajes con metadatos adicionales cuando pasan a través de una suscripción.
- Los filtros pueden ser de tres tipos:
  - Filtros booleanos: TrueFilter garantiza que todos los mensajes enviados al tema se entreguen a la suscripción actual, FalseFilter garantiza que ninguno se entregue (Esto bloquea o desactiva la suscripción de forma eficaz).
  - Filtros de SQL: Un filtro de SQL especifica una condición con la misma sintaxis que una cláusula WHERE en una consulta SQL. Solo los mensajes que devuelven True, cuando se evalúen con respecto a esta suscripción, se entregarán a los suscriptores.
  - Filtros de correlación: Un filtro de correlación contiene un conjunto de condiciones que se comparan con las propiedades de cada mensaje: si tienen el mismo valor, se entrega.

Crear tema

- Nombre
- Período de vida del mensaje: Días, Horas, Minutos, Segundos
- Habilitar eliminación automática en el tema inactivo
- Habilitar detección de duplicados
- Habilitar la creación de particiones

Crear suscripción

- Nombre
- Número máximo de entregas
- Eliminar automáticamente después de un tiempo de inactividad de:
  - Días, Horas, Minutos, Segundos
  - No eliminar automáticamente nunca
  - Reenviar mensajes a la cola/tema
- Habilitar sesiones
- Período de vida de los mensajes y mensajes fallidos
  - Tiempo del mensaje para activarse (predeterminado): Días, Horas, Minutos, Segundos
  - Habilitar la cola de mensajes fallidos cuando expire un mensaje
  - Mover mensajes que producen excepciones de evaluación de filtros a la subcola de mensajes fallidos
- Duración del bloqueo: Días, Horas, Minutos, Segundos

Creación de un publicador Microsoft.Azure.ServiceBus

```
ITopicClient topicClient = new TopicClient(ServiceBusConnectionString, QueueOrTopicName);

for (int i = 0; i < NumeroDeEnvios; i++)
    try {
        string messageBody = $"Mensaje {DateTime.Now:MM:ss:ff}: valor {rnd.Next(1, 100)}.";
        var item = new Item() { Id = i, Mensaje = messageBody };
        var message = item.AsMessage();
        message.To = item.Categoria;
        message.Label = item.Nivel;
        message.UserProperties.Add("nivel", item.Nivel);
        await topicClient.SendAsync(message);
        Console.WriteLine($"Sending message: {item}");
    } catch (Exception exception) {
        Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
    }
await topicClient.CloseAsync();
Console.WriteLine("Message was sent successfully.");
```

Creación de un suscriptor Microsoft.Azure.ServiceBus

```
ISubscriptionClient subscriptionClient = new SubscriptionClient(ServiceBusConnectionString,
    QueueOrTopicName, SubscriptionName);

Console.WriteLine("======================================================");
Console.WriteLine($"{SubscriptionName} Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("======================================================");

subscriptionClient.RegisterMessageHandler(async (message, token) => {
    Console.WriteLine($"Received {SubscriptionName}:
          SequenceNumber:{message.SystemProperties.SequenceNumber} Body:
          {Encoding.UTF8.GetString(message.Body)} {message.Label}");
    await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);
}, new MessageHandlerOptions(ExceptionReceivedHandler) {
    MaxConcurrentCalls = 1,
    AutoComplete = false
});
Console.Read();
await subscriptionClient.CloseAsync();
```


Creación de un suscriptor Azure.Messaging.ServiceBus

```
Console.WriteLine("======================================================");
Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("======================================================");
var options = new ServiceBusProcessorOptions {
    AutoCompleteMessages = false
};
await using ServiceBusProcessor processor = string.IsNullOrWhiteSpace(SubscriptionName) ?
    srv.CreateProcessor(QueueOrTopicName, options) :
    srv.CreateProcessor(QueueOrTopicName, SubscriptionName, options);
processor.ProcessMessageAsync += async (arg) => {
    Console.WriteLine($"Received message: SequenceNumber: {arg.Message.SequenceNumber} Body:
        {Encoding.UTF8.GetString(arg.Message.Body)}");
    if (AbandonarMultiplosDe > 0 && arg.Message.SequenceNumber % AbandonarMultiplosDe == 0) {
        await arg.AbandonMessageAsync(arg.Message);
    } else
        await arg.CompleteMessageAsync(arg.Message);
};
processor.ProcessErrorAsync += ExceptionReceivedHandler;
await processor.StartProcessingAsync();
Console.Read();
Console.WriteLine("Exit processor...");
```

Sesiones de mensajes

- Para crear una garantía primero en entrar/primero en salir (FIFO) en Service Bus, se deben usar sesiones. Las sesiones de mensajes permiten la administración ordenada y exclusiva de secuencias de mensajes relacionados no enlazadas. Para permitir la administración de sesiones en sistemas de alta disponibilidad y a gran escala, la característica de sesión también permite almacenar el estado de la sesión, lo que hace que las sesiones se muevan de forma segura entre controladores. Las colas y las suscripciones deben habilitar sesiones.
- Service Bus no prescribe con respecto a la naturaleza de la relación entre los mensajes, y tampoco define un modelo en particular para determinar dónde comienza o termina una secuencia de mensajes.
- Cualquier remitente puede crear una sesión al enviar mensajes en un tema o una cola si se establece la propiedad SessionId en algún identificador definido por la aplicación que sea único para la sesión. Se puede establecer la propiedad Label del primer mensaje en 'start', de los mensajes intermedios en 'content' y del último mensaje en 'end', la posición relativa de los mensajes se puede calcular como el SequenceNumber de los mensajes.
- Cuando están habilitadas las sesiones es obligatorio asignar el SessionId en los envíos de mensajes.

Sesiones de mensajes

- El cliente que acepta la sesión crea un receptor MessageSession. Cuando el objeto MessageSession se acepta y mientras está contenido en el cliente, ese cliente mantiene un bloqueo exclusivo sobre todos los mensajes con ese valor de SessionId de la sesión que existe en la cola o la suscripción y también sobre todos los mensajes con ese valor de SessionId que sigan llegando mientras tiene lugar la sesión.
- El bloqueo se libera cuando se llama a Close o CloseAsync, o cuando el bloqueo expira en caso de que la aplicación sea incapaz de llevar a cabo la operación de cierre. La aplicación debería cerrar el MessageSession en cuanto ya no lo necesite o no espere ningún otro mensaje.
- Cuando varios receptores simultáneos se extraen de la cola, los mensajes que pertenecen a una sesión determinada se envían al receptor específico que actualmente mantiene el bloqueo en esa sesión. Con esa operación, una secuencia de mensajes intercalados en una cola o suscripción se demultiplexa limpiamente en receptores diferentes, y esos receptores también pueden encontrarse en distintas máquinas cliente, puesto que la administración del bloqueo tiene lugar en el servidor, dentro de Service Bus.

Receptor con sesiones Microsoft.Azure.ServiceBus

```
IQueueClient queueClient = new QueueClient(ServiceBusConnectionString, QueueOrTopicName,
ReceiveMode.PeekLock);
Console.WriteLine("======================================================");
Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("======================================================");
queueClient.RegisterSessionHandler(async (session, message, token) => {
    Console.WriteLine($"Received message: {session.SessionId}
        SequenceNumber:{message.SystemProperties.SequenceNumber} Body: {message.MessageId} -
        {Encoding.UTF8.GetString(message.Body)}");
    if (AbandonarMultiplosDe > 0 && message.SystemProperties.SequenceNumber % AbandonarMultiplosDe ==
        0) {
        await session.AbandonAsync(message.SystemProperties.LockToken);
    } else
        await session.CompleteAsync(message.SystemProperties.LockToken);
}, new SessionHandlerOptions(ExceptionReceivedHandler) {
    MessageWaitTimeout = TimeSpan.FromSeconds(1),
    MaxConcurrentSessions = 1,
    AutoComplete = false
});
Console.Read();
await queueClient.CloseAsync();
```


Receptor con sesiones Azure.Messaging.ServiceBus

```
Console.WriteLine("======================================================");
Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
Console.WriteLine("======================================================");
var options = new ServiceBusSessionProcessorOptions {
    AutoCompleteMessages = false, MaxConcurrentSessions = 5, MaxConcurrentCallsPerSession = 2
};
await using ServiceBusSessionProcessor processor = string.IsNullOrWhiteSpace(SubscriptionName) ?
    srv.CreateSessionProcessor(QueueOrTopicName, options) :
    srv.CreateSessionProcessor(QueueOrTopicName, SubscriptionName, options);
processor.ProcessMessageAsync += async (arg) => {
    Console.WriteLine($"Received message: {arg.SessionId} SequenceNumber: {arg.Message.SequenceNumber} Body:
{Encoding.UTF8.GetString(arg.Message.Body)}");
    if (AbandonarMultiplosDe > 0 && arg.Message.SequenceNumber % AbandonarMultiplosDe == 0) {
        await arg.AbandonMessageAsync(arg.Message);
    } else
        await arg.CompleteMessageAsync(arg.Message);
};
processor.ProcessErrorAsync += ExceptionReceivedHandler;
await processor.StartProcessingAsync();
Console.Read();
await processor.CloseAsync();
```

Envío de mensajes

- Cuando un cliente envía un mensaje, normalmente quiere saber si el mensaje se ha transferido correctamente y lo ha aceptado el agente o si se ha producido algún tipo de error. Esta confirmación positiva o negativa establece la comprensión del cliente y del agente sobre el estado de la transferencia del mensaje; por tanto, esto se conoce como liquidación.
- Si Service Bus rechaza el mensaje, el rechazo contiene un indicador de error y un texto con una propiedad "tracking-id" en el interior. El rechazo también incluye información sobre si se puede volver a intentar la operación con alguna expectativa de éxito. En el cliente, esta información se convierte en una excepción y se envía al autor de llamada de la operación de envío. Si se acepta el mensaje, la operación se completa en modo silencioso.
- Las aplicaciones nunca deben iniciar una operación de envío asincrónica de forma "enviar y olvidarse" sin recuperar el resultado de la operación. Si la aplicación genera ráfagas de mensajes, debería realizar envíos por lotes.

Creación de un emisor por lotes Azure.Messaging.ServiceBus

```
ServiceBusSender sender = srv.CreateSender(QueueOrTopicName);
string[] sessionIds = { Guid.NewGuid().ToString(), Guid.NewGuid().ToString(), Guid.NewGuid().ToString() };
ServiceBusMessageBatch messageBatch = await sender.CreateMessageBatchAsync();
try {
    for (int i = 0; i < NumeroDeEnvios; i++) {
        string messageBody = $"Mensaje {DateTime.Now:MM:ss:ff}: valor {rnd.Next(1, 100)}.";
        var item = new Item() { Id = i, Mensaje = messageBody };
        var message = item.AsMessage();
        message.To = item.Categoria;
        message.Subject = item.Nivel;
        message.ApplicationProperties.Add("nivel", item.Nivel);
        message.MessageId = $"{DateTime.Now.Minute}-{i}";
        message.SessionId = sessionIds[i % 3];
        if (TimeToLive > 0) message.TimeToLive = TimeSpan.FromSeconds(TimeToLive);
        Console.WriteLine($"Sending message: {message.MessageId} - {item}");
        if (!messageBatch.TryAddMessage(message)) {
            await sender.SendMessagesAsync(messageBatch);
            Console.WriteLine($"Sending Batch: {messageBatch.Count} - {messageBatch.SizeInBytes}");
            messageBatch.Dispose();
            messageBatch = await sender.CreateMessageBatchAsync();
            messageBatch.TryAddMessage(message);
        }
    }
    Console.WriteLine($"Sending Batch: {messageBatch.Count} - {messageBatch.SizeInBytes}");
    await sender.SendMessagesAsync(messageBatch);
} catch (Exception exception) {
    Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
}
await sender.CloseAsync();
Console.WriteLine("Messages was sent successfully.");
```

Entrega programada

- Se puede enviar mensajes a una cola o tema para su procesamiento retrasado; por ejemplo, para programar un trabajo de forma que esté disponible para que lo procese el sistema a una hora determinada.
- Los mensajes programados no se materializan en la cola hasta la hora de puesta en cola definida. Antes de esa hora, pueden cancelarse. La cancelación elimina el mensaje.
- Para programar mensajes, establezca la propiedad ScheduledEnqueueTimeUtc al enviar un mensaje mediante la ruta de envío normal, o explícitamente con la API ScheduleMessageAsync. Lo segundo devuelve inmediatamente el valor de SequenceNumber del mensaje programado, que luego puede usarse para cancelar dicho mensaje si es necesario.
- Los mensajes programados y sus números de secuencia también se pueden detectar mediante la exploración de mensajes.

Recepción de mensajes

- El modo Receive-and-Delete indica al agente que considere todos los mensajes que envía al cliente receptor como liquidados cuando se envían. Esto significa que el mensaje se considera consumido tan pronto como el agente lo pone en circulación. Si se produce un error en la transferencia del mensaje, el mensaje se pierde. La ventaja de este modo es que el receptor no necesita realizar ninguna otra acción en el mensaje y además no se ralentizará al esperar el resultado de la liquidación. Si los datos contenidos en los mensajes individuales tienen un valor bajo o solo son significativos durante muy poco tiempo, este modo es una opción razonable.
- El modo PeekLock indica al agente que el cliente receptor desea liquidar explícitamente los mensajes recibidos. El mensaje ahora está disponible para que el receptor lo procese, mientras se mantiene en un bloqueo exclusivo en el servicio para que los demás receptores competidores no puedan verlo. La duración del bloqueo se define inicialmente en el nivel de cola o suscripción y la puede ampliar el cliente que posee el bloqueo, a través de la operación RenewLock.
- Cuando se bloquea un mensaje, los otros clientes pueden recuperar los siguientes mensajes disponibles que no se encuentran bloqueados. Cuando se libera explícitamente el bloqueo en un mensaje o cuando expira el bloqueo, el mensaje vuelve a emerger en o cerca de la parte delantera del orden de recuperación para poder entregarlo de nuevo.

Recepción de mensajes

- Cuando los receptores liberan repetidamente el mensaje o permiten el bloqueo transcurra durante un número definido de veces (maxDeliveryCount), el mensaje se quita de la cola o suscripción automáticamente y se coloca en cola de mensajes fallidos asociada.
- El receptor inicia la liquidación de un mensaje recibido con una confirmación positiva cuando llama al método Complete en el nivel de API. Esto indica al agente que el mensaje se ha procesado correctamente y este se quita de la cola o suscripción. El agente responde al intento de liquidación del receptor con una respuesta que indica si se pudo realizar la liquidación.
- Cuando el cliente receptor no puede procesar un mensaje pero quiere que el mensaje se vuelva a entregar, puede pedir explícitamente que el mensaje se libere y se desbloquee al instante mediante una llamada al método Abandon o bien puede no hacer nada y dejar que el bloqueo transcurra.
- Si un cliente receptor no puede procesar un mensaje y sabe que volver a enviar el mensaje e reintentar la operación no va a ser la solución, puede rechazar el mensaje, que lo traslada a la cola de mensajes fallidos mediante una llamada al método DeadLetter. Este método también permite establecer una propiedad personalizada con un código de motivo que se puede recuperar con el mensaje procedente de la cola de mensajes fallidos.

Aplazamiento de mensajes

- El cliente de una cola o suscripción puede aplazar la recuperación de un mensaje recibido para otro momento. Es posible que el mensaje se haya publicado fuera del orden esperado y que el cliente desee esperar hasta recibir otro mensaje.
- Cuando un cliente de cola o suscripción recibe un mensaje que se desea procesar, pero cuyo procesamiento no es posible en ese momento debido a circunstancias especiales dentro de la aplicación, tiene la opción de "aplazar" la recuperación del mensaje para un momento posterior. Los mensajes aplazados permanecen en la cola o suscripción, pero se mantienen separados, y se deben reactivar explícitamente mediante el número de secuencia asignado por el servicio.
- El aplazamiento es una característica creada específicamente para escenarios de procesamiento de flujo de trabajo. Los marcos de flujo de trabajo pueden requerir que se procesen ciertas operaciones en un orden concreto y puede que tenga que posponer el procesamiento de algunos mensajes recibidos hasta que se haya completado el trabajo anterior prescrito notificado por otros mensajes.
- Las APIs que permiten manejar los aplazamientos son BrokeredMessage, MessageReceiver.DeferAsync, IMessageReceiver.defer o IMessageReceiver.deferAsync.

Exploración de mensajes

- La exploración de mensajes, o la inspección, permite a un cliente de Service Bus enumerar todos los mensajes de una cola o una suscripción con fines de diagnóstico o depuración: devuelve todos los mensajes de la cola, no solo los disponibles para la adquisición inmediata con Receive() o el bucle de OnMessage(). La propiedad State de cada mensaje indica si el mensaje está activo (disponible para su recepción), se ha aplazado o está programado. La operación de inspección en una suscripción no devuelve los mensajes programados en el registro de mensajes de la suscripción.
- Como los mensajes consumidos y expirados se limpian mediante una ejecución de "recolección de elementos no utilizados" asincrónica, es posible que Peek pueda devolver mensajes que ya han expirado. Estos mensajes se quitarán o se pondrán en la cola de mensajes fallidos cuando se invoque una operación de recepción en la cola o la suscripción la próxima vez. Estos mensajes se devuelven por diseño dado que Peek es una herramienta de diagnóstico que refleja el estado actual del registro.
- Peek también devuelve los mensajes que estaban bloqueados y que actualmente están procesando otros receptores. Sin embargo, dado que Peek devuelve una instantánea desconectada, el estado de bloqueo de un mensaje no se puede observar en los mensajes inspeccionados. Las propiedades LockedUntilUtc y LockToken producen una excepción InvalidOperationException cuando la aplicación intenta leerlas.
- Los métodos Peek/PeekAsync y PeekBatch/PeekBatchAsync permiten la exploración de mensajes.

Cola de mensajes fallidos

- Todas las colas y suscripciones a temas de Service Bus llevan asociada una cola de mensajes con problemas de entrega (DLQ). Esta cola contiene mensajes que cumplen los criterios siguientes:
  - No se pueden entregar correctamente a ningún receptor.
  - Se les ha agotado el tiempo de espera.
  - La aplicación receptora los deja a un lado explícitamente.
- Los mensajes de la cola de mensajes con problemas de entrega se anotan con el motivo de que se colocaran allí. La cola de mensajes con problemas de entregas tiene un punto de conexión especial, pero, por lo demás, funciona como cualquier cola normal. Una aplicación o herramienta puede examinar una cola de mensajes con problemas de entrega o quitarlos de esta. También se puede reenviar automáticamente una cola de mensajes con problemas de entrega.
- Se puede obtener la ruta de acceso para la cola de mensajes fallidos mediante el método SubscriptionClient.FormatDeadLetterPath().

Receptor de mensajes fallidos Microsoft.Azure.ServiceBus

```
var deadletterReceiver = new MessageReceiver(ServiceBusConnectionString,
    EntityNameHelper.FormatDeadLetterPath(QueueOrTopicName), ReceiveMode.PeekLock);

while (true) {
    var message = await deadletterReceiver.ReceiveAsync(TimeSpan.FromSeconds(1));
    if (message == null)
        break;
    Console.WriteLine($"Deadletter message: {message.SessionId}
        SequenceNumber:{message.SystemProperties.SequenceNumber} Body:
        {message.MessageId} - {Encoding.UTF8.GetString(message.Body)}");
    foreach (var prop in message.UserProperties) {
        Console.WriteLine("\t{0}={1}", prop.Key, prop.Value);
    }
    await deadletterReceiver.CompleteAsync(message.SystemProperties.LockToken);
}
```


Receptor de mensajes fallidos Azure.Messaging.ServiceBus

```
var deadletterReceiver = srv.CreateReceiver(QueueOrTopicName, new ServiceBusReceiverOptions() {
    ReceiveMode = ServiceBusReceiveMode.PeekLock,
    SubQueue = SubQueue.DeadLetter
});

while (true) {
    var message = await deadletterReceiver.ReceiveMessageAsync(TimeSpan.FromSeconds(1));
    if (message == null)
        break;
    Console.WriteLine($"Deadletter message: {message.SessionId}
        SequenceNumber:{message.SequenceNumber} Body: {message.MessageId} -
        {Encoding.UTF8.GetString(message.Body)}");
    foreach (var prop in message.ApplicationProperties) {
        Console.WriteLine("\t{0}={1}", prop.Key, prop.Value);
    }
    await deadletterReceiver.CompleteMessageAsync(message);
}
await deadletterReceiver.CloseAsync();
```

Transacciones en Service Bus

- Una transacción agrupa dos o más operaciones en un ámbito de ejecución. Por naturaleza, una transacción de este tipo debe garantizar que todas las operaciones que pertenecen a un grupo determinado de operaciones tendrán éxito o darán error de forma conjunta. En este sentido, las transacciones actúan como una unidad, lo que a menudo se conoce como atomicidad.
- Service Bus es un agente de mensajes transaccional, y garantiza la integridad transaccional de todas las operaciones internas en sus almacenes de mensajes. Todas las transferencias de mensajes dentro de Service Bus, como mover mensajes a una cola de mensajes fallidos o reenviar automáticamente mensajes entre entidades, son transaccionales. Por lo tanto, si Service Bus acepta un mensaje, es que ya se ha almacenado y etiquetado con un número de secuencia. Desde ese momento, todas las transferencias de mensajes dentro de Service Bus son operaciones coordinadas entre entidades, y ninguna de ellas dará lugar a la pérdida (el origen tiene éxito y el destino da error) o a la desduplicación (el origen da error y el destino tiene éxito) del mensaje.
- Service Bus admite operaciones de agrupación en una sola entidad de mensajería (cola, tema, suscripción) dentro del ámbito de una transacción. Por ejemplo, puede enviar varios mensajes a una cola desde dentro de un ámbito de transacción y los mensajes solo se confirmarán en el registro de la cola cuando la transacción se complete correctamente.
- Las operaciones que pueden realizarse dentro de un ámbito de transacción son las siguientes:
  - QueueClient, MessageSender, TopicClient : Send, SendAsync, SendBatch, SendBatchAsync
  - BrokeredMessage : Complete, CompleteAsync, Abandon, AbandonAsync, Deadletter, DeadletterAsync, Defer, DeferAsync, RenewLock, RenewLockAsync

Transacciones en Service Bus

- Para permitir la entrega transaccional de datos de una cola o un tema a un procesador y luego a otra cola u otro tema, Service Bus admite transferencias. En las operaciones de transferencia, en primer lugar un remitente envía un mensaje a una cola o un tema de transferencia, que lo mueve inmediatamente a la cola o el tema de destino deseado usando la misma implementación de transferencias sólida en la que se basa la funcionalidad de reenvío automático. El mensaje nunca se confirma en el registro del tema o la cola de transferencia de forma que se vuelve visible para los consumidores de dicho tema o cola.
- La eficacia de esta funcionalidad transaccional se hace evidente cuando el propio tema o la cola de transferencia es el origen de los mensajes de entrada del remitente. En otras palabras, Service Bus puede transferir el mensaje a la cola o el tema de destino "mediante" la cola o el tema de transferencia y al mismo tiempo realizar una operación completa (o de aplazamiento o correo devuelto) en el mensaje de entrada, todo en una operación atómica.

```
using (var ts = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled)) {
    try  {
        // complete the transaction
        ts.Complete();
    } catch (Exception ex) {
        // This rolls back send and complete in case an exception happens
        ts.Dispose();
    }
}
```

Transacciones Microsoft.Azure.ServiceBus

```
IQueueClient queueClient = new QueueClient(ServiceBusConnectionString,
    QueueOrTopicName);

using (var ts = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled)) {
    await queueClient.SendAsync(new Message(Encoding.UTF8.GetBytes("Paso1")));
    await queueClient.SendAsync(new Message(Encoding.UTF8.GetBytes("Paso2")));
    await queueClient.SendAsync(new Message(Encoding.UTF8.GetBytes("Paso3")));
    if (commit) {
        ts.Complete();
        Console.WriteLine("Messages was sent successfully.");
    } else {
        Console.WriteLine("Messages was cancel.");
        ts.Dispose();
    }
}
await queueClient.CloseAsync();
```


Transacciones Azure.Messaging.ServiceBus

```
ServiceBusSender sender = srv.CreateSender(QueueOrTopicName);

using (var ts = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled)) {
    await sender.SendMessageAsync(new ServiceBusMessage("Paso1"));
    await sender.SendMessageAsync(new ServiceBusMessage("Paso2"));
    await sender.SendMessageAsync(new ServiceBusMessage("Paso3"));
    if (commit) {
        ts.Complete();
        Console.WriteLine("Messages was sent successfully.");
    } else {
        Console.WriteLine("Messages was cancel.");
        ts.Dispose();
    }
}

await sender.CloseAsync();
```

Seguridad

- Service Bus admite los protocolos estándar AMQP 1,0 y HTTP o REST y sus respectivas utilidades de seguridad, incluida la seguridad de nivel de transporte (TLS). Los clientes pueden tener acceso autorizado mediante Firma de acceso compartido o la seguridad basada en roles de Azure Active Directory.
- Cumplimiento normativo de Azure Policy proporciona definiciones de iniciativas creadas y administradas por Microsoft, conocidas como integraciones, para los dominios de cumplimiento y los controles de seguridad relativos a distintos estándares de cumplimiento.
- Azure Service Bus permite establecer las siguientes características de seguridad para proteger contra el tráfico no deseado:
  - Etiquetas de servicio (grupo de prefijos de direcciones IP)
  - Reglas de firewall de IP
  - Puntos de conexión de servicio de red
  - Puntos de conexión privados

Autenticación y autorización de Service Bus

- La integración de Azure AD para recursos de Service Bus proporciona control de acceso basado en rol de Azure (RBAC de Azure) para el control específico del acceso de los clientes a los recursos. Puede usar Azure RBAC para conceder permisos a una entidad de seguridad, que puede ser un usuario, un grupo o una entidad de servicio de aplicación. Azure AD autentica la entidad de seguridad para devolver un token de OAuth 2.0. El token se puede usar para autorizar una solicitud de acceso a un recurso de Service Bus (cola, tema, etc.).
- La autenticación de SAS permite conceder acceso a una aplicación a los recursos de Service Bus con derechos específicos e implica la configuración de una clave criptográfica con derechos asociados en un recurso de Service Bus. Los clientes pueden obtener acceso a ese recurso presentando un token SAS, que consta del URI del recurso al que se tiene acceso y una fecha de expiración firmada con la clave configurada.
- Se puede configurar claves SAS en un espacio de nombres de Service Bus, se aplica a todas las entidades de mensajes de ese espacio de nombres, o para colas y temas. Para usar SAS, hay que configurar un objeto SharedAccessAuthorizationRule en el espacio de nombres, cola o tema (un máximo de 12 reglas). Esta regla está formada por los siguientes elementos:
  - KeyName: identifica la regla.
  - PrimaryKey: es una clave criptográfica usada para firmar o validar tokens SAS.
  - SecondaryKey: es una clave criptográfica usada para firmar o validar tokens SAS.
  - Rights: representa la recopilación de derechos de escucha, envío o administración concedidos.
- La autorización de usuarios o aplicaciones mediante un token de OAuth 2.0 devuelto por Azure AD proporciona una seguridad superior y facilidad de uso sobre las firmas de acceso compartido (SAS).

Supervisión

- Los registros operativos capturan todas las operaciones de administración que se realizan en el espacio de nombres de Azure Service Bus. Las operaciones de datos no se capturan debido al elevado volumen de este tipo de operaciones que se realizan en Azure Service Bus.
  - En Azure Portal, Azure Service Bus → Supervisión →Configuración de diagnóstico.
- Azure Monitor proporciona interfaces de usuario unificadas para la supervisión de distintos servicios de Azure. Las métricas de Service Bus permiten conocer el estado de los recursos de la suscripción de Azure. Con un amplio conjunto de datos de métricas, se puede evaluar el estado general de los recursos de Service Bus, no solo en el nivel de espacio de nombres, sino también en el nivel de entidad. Estas estadísticas pueden ser importantes, ya que ayudan a supervisar el estado de Service Bus. Las métricas también pueden ayudar a solucionar problemas de causa principal sin necesidad de ponerse en contacto con el soporte técnico de Azure.

ANEXOS

Enlaces

- https://microservices.io/
- https://docs.microsoft.com/es-es/azure/architecture/patterns/
- https://swagger.io/docs/specification/about/
- http://spec.openapis.org/oas/v3.0.3
- https://github.com/Azure/azure-sdk-for-net/tree/master/sdk/servicebus/Azure.Messaging.ServiceBus/samples
- https://github.com/Azure/azure-service-bus/tree/master/samples/DotNet

- Extension for Visual Studio Code
  - OpenAPI (Swagger) Editor
  - REST Client
  - Azure API Management

GenerateData

- GenerateData es una herramienta para la generación automatizada de juegos de datos.
- Ofrece ya una serie de datos precargados en BBDD y un conjunto de tipos de datos bastante amplio, así como la posibilidad de generar tipos genéricos.
- Podemos elegir en que formato se desea la salida de entre los siguientes:
  - CSV
  - Excel
  - HTML
  - JSON
  - LDIF
  - Lenguajes de programación (JavaScript, Perl, PHP, Ruby, C#)
  - SQL (MySQL, Postgres, SQLite, Oracle, SQL Server)
  - XML
- Online: http://www.generatedata.com/?lang=es
- Instalación (PHP): http://benkeen.github.io/generatedata/

Mockaroo

- https://www.mockaroo.com/
- Mockaroo permite descargar rápida y fácilmente grandes cantidades de datos realistas de prueba generados aleatoriamente en función de sus propias especificaciones que luego puede cargar directamente en su entorno de prueba utilizando formatos CSV, JSON, XML, Excel, SQL, … No se requiere programación y es gratuita (para generar datos de 1.000 en 1.000).
- Los datos realistas son variados y contendrán caracteres que pueden no funcionar bien con nuestro código, como apóstrofes o caracteres unicode de otros idiomas. Las pruebas con datos realistas harán que la aplicación sea más robusta porque detectará errores en escenarios de producción.
- El proceso es sencillo ya que solo hay que ir añadiendo nombres de campos y escoger su tipo. Por defecto nos ofrece mas de 140 tipos de datos diferentes que van desde nombre y apellidos (pudiendo escoger estilo, género, etc...) hasta ISBNs, ubicaciones geográficas o datos encriptados simulados.
- Además es posible hacer que los datos sigan una distribución Normal o de Poisson, secuencias, que cumplan una expresión regular o incluso que fuercen cadenas complicadas con caracteres extraños y cosas así. Tenemos la posibilidad de crear fórmulas propias para generarlos, teniendo en cuenta otros campos, condicionales, etc... Es altamente flexible.