Universidad Nacional de La Matanza

Desarrollo de Aplicaciones Web

Departamento de Ingeniería e Investigaciones Tecnológicas

API Rest en Net Core, Swagger y Microservicios

Trabajo práctico Programación Web III

Integrantes

Aragón Yamila

Cortes Valdes Scarlet

Galeano Carla

Machicado Cuba José

Profesores

Ing. Matias Paz Wasiuchnik

Ing. Pablo Nicolás Sanchez

Ing. Mariano Juiz

API REST EN NET CORE , SWAGGER Y MICROSERVICIOS

La arquitectura de software es una estructura de alto nivel para el desarrollo e implementación de sistemas de software. A medida que el software se vuelve cada vez más frecuente y generalizado, los diferentes estilos arquitectónicos están destinados a evolucionar por esta razón la arquitectura de microservicios ha ganado mucha relevancia.

MICROSERVICIOS

La arquitectura de microservicios es un método de desarrollo de aplicaciones software que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, proporcionando una funcionalidad de negocio completa. En ella, cada microservicio es un código que puede estar en un lenguaje de programación diferente, y que desempeña una función específica. Los microservicios se comunican entre sí a través de APIs, y cuentan con sistemas de almacenamiento propios, lo que evita la sobrecarga y caída de la aplicación.

Los microservicios han creado infraestructuras IT más adaptables y flexibles. Porque si se quiere modificar solamente un servicio, no es necesario alterar el resto de la infraestructura. Cada uno de los servicios se puede desplegar y modificar sin que ello afecte a otros servicios o aspectos funcionales de la aplicación.

Cada servicio se ejecuta en su propio proceso y se comunica con otros procesos mediante protocolos como HTTP/HTTPS, WebSockets o AMQP. Cada microservicio implementa un dominio de un extremo a otro específico o una capacidad empresarial dentro de un determinado límite de contexto, y cada uno se debe desarrollar de forma autónoma e implementar de forma independiente. Por último, cada microservicio debe poseer su modelo de datos de dominio relacionado y su lógica del dominio (soberanía y administración de datos descentralizada), y podría basarse en otras tecnologías de almacenamiento de datos (SQL, NoSQL) y lenguajes de programación.

¿Por qué microservicios?

Para comprender la popularidad de los microservicios, es importante comprender las aplicaciones monolíticas. El desarrollo de software estuvo marcado hasta el tiempo reciente por desarrollar aplicaciones monolíticas. Cualquier aplicación consistirá en una interfaz del lado del cliente, una base de datos para almacenar y manejar datos y lógica del lado del servidor. Una aplicación monolítica se construye como una sola unidad. En una aplicación monolítica, la lógica del lado del servidor se agrupa en forma de monolito, con toda la lógica para manejar las solicitudes ejecutándose en un solo proceso. Todas las actualizaciones de este sistema implicarán la implementación de una nueva versión de la aplicación del lado del servidor. Si

bien hay, y ha habido miles de aplicaciones monolíticas exitosas en el mercado, los equipos de desarrollo están empezando a sentir el peso de tales operaciones de implementación a gran escala. Por ejemplo, en una aplicación monolítica, cualquier pequeña modificación está ligada a un ciclo de cambio completo, y estos ciclos están invariablemente vinculados entre sí. El escalado granular de funciones y módulos específicos tampoco es posible, ya que los equipos tienen que escalar una aplicación completa.

Entonces, las aplicaciones monolíticas se enfrentan a los siguientes inconvenientes:

- Dificultades para la ejecución de pruebas a los componentes.
- Equipo de desarrollo difícil de gestionar.
- La incorporación de nuevas características puede impactar en todo el sistema debido al alto acoplamiento.
- Limitación para aprovechar la diversidad tecnológica y las ventajas particulares que ofrece cada tecnología para solucionar problemas específicos que suelen tener lugar.
- Al estar todo unido el rendimiento puede verse comprometido, a la vez que la escalabilidad es más costosa toda vez que deben desplegarse instancias completas del sistema para aumentar sus capacidades de respuesta antes volúmenes de uso creciente.

Con el fin de superar la creciente complejidad de la gestión, actualización e implementación de aplicaciones, particularmente en la nube, la adopción de la arquitectura de microservicios despegó. Cada microservicio define una función empresarial específica y solo define las operaciones que pertenecen a esta función, lo que las hace muy ligeras y portátiles.

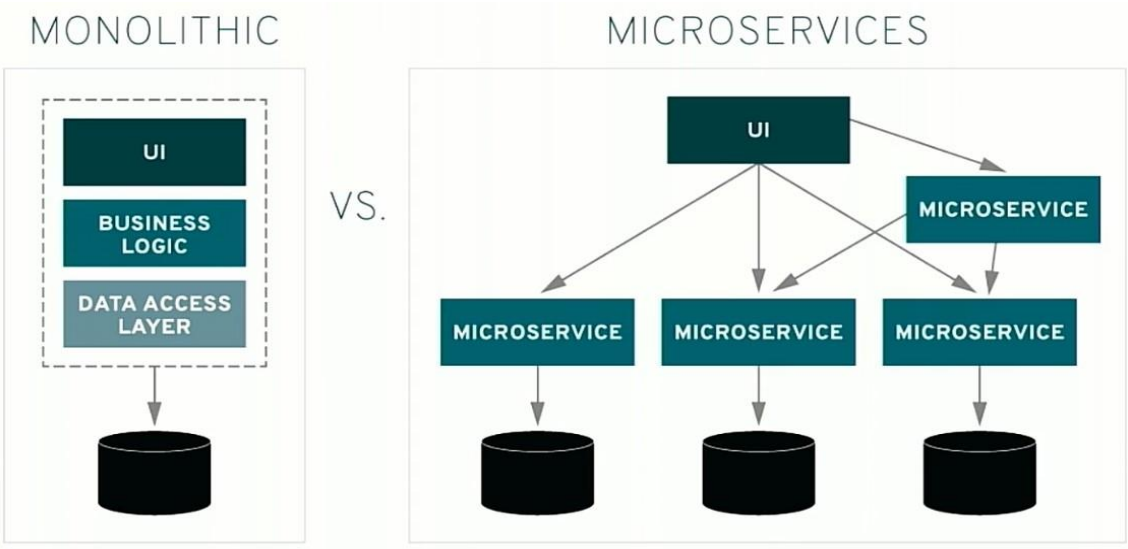


Figura 1. Implementación monolítica frente al enfoque de los microservicios

Diseñar la arquitectura de aplicaciones específicas basadas en microservicios habilita una integración continua y prácticas de entrega continua. También acelera la entrega de nuevas funciones en la aplicación. La composición específica de las aplicaciones también le permite ejecutar y probar los microservicios de manera aislada y hacerlos evolucionar de forma autónoma a la vez que mantiene contratos claros entre ellos. Siempre y cuando no cambie las interfaces o los contratos, puede cambiar la implementación interna de cualquier microservicio o agregar nuevas funciones sin que ello interrumpa otros microservicios.

Así, mientras que en una arquitectura monolítica el software se desarrolla como una única unidad, sin separación entre módulos, algo que puede ser lo indicado para algunas aplicaciones, pero totalmente rígido para muchas otras. Una arquitectura de microservicios funciona con un conjunto de pequeños servicios que se ejecutan de manera autónoma e independiente.

Actualmente muchos desarrolladores están dejando de utilizar arquitecturas monolíticas para pasarse a los microservicios, y muchas empresas líderes ya se han dado cuenta de que la arquitectura de microservicios es la mejor opción para sus organizaciones.

Ventajas

Modularidad: al tratarse de servicios autónomos, se pueden desarrollar y desplegar de forma independiente. Además un error en un servicio no debería afectar la capacidad de otros servicios para seguir trabajando según lo previsto.

Escalabilidad: Como ventaja adicional, los microservicios se pueden escalar horizontalmente de forma independiente. En lugar de disponer de una sola aplicación monolítica que debe escalar horizontalmente como una unidad, puede escalar horizontalmente microservicios concretos. De esa forma, puede escalar solo el área funcional que necesita más potencia de procesamiento o ancho de banda para admitir la demanda, en lugar de escalar horizontalmente otras partes de la aplicación que no hace falta escalar.

Versatilidad: se pueden usar diferentes tecnologías y lenguajes de programación. Lo que permite adaptar cada funcionalidad a la tecnología más adecuada y rentable.

Rapidez de actuación: el reducido tamaño de los microservicios permite un desarrollo menos costoso, así como el uso de “contenedores de software” permite que el despliegue de la aplicación se pueda llevar a cabo rápidamente.

Mantenimiento simple y barato: al poder hacerse mejoras de un solo módulo y no tener que intervenir en toda la estructura, el mantenimiento es más sencillo y barato que en otras arquitecturas.

Agilidad: se pueden utilizar funcionalidades típicas (autenticación, trazabilidad, etc.) que ya han sido desarrolladas por terceros, no hace falta que el desarrollador las cree de nuevo.

Desventajas

Alto consumo de memoria: al tener cada microservicio sus propios recursos y bases de datos, consumen más memoria y CPU.

Inversión de tiempo inicial: al crear la arquitectura, se necesita más tiempo para poder fragmentar los distintos microservicios e implementar la comunicación entre ellos.

Complejidad en la gestión: si contamos con un gran número de microservicios, será más complicado controlar la gestión e integración de los mismos. Es necesario disponer de una centralización de trazas y herramientas avanzadas de procesamiento de información que permitan tener una visión general de todos los microservicios y orquesten el sistema.

Perfil de desarrollador: los microservicios requieren desarrolladores experimentados con un nivel muy alto de experiencia y un control exhaustivo de las versiones. Además de conocimiento sobre solución de problemas como latencia en la red o balanceo de cargas.

No uniformidad: aunque disponer de un equipo tecnológico diferente para cada uno de los servicios tiene sus ventajas, si no se gestiona correctamente, conducirá a un diseño y arquitectura de aplicación poco uniforme.

Dificultad en la realización de pruebas: debido a que los componentes de la aplicación están distribuidos, las pruebas y test globales son más complicados de realizar.

Coste de implantación alto: una arquitectura de microservicios puede suponer un alto coste de implantación debido a costes de infraestructura y pruebas distribuidas.

Teniendo en cuenta las ventajas e inconvenientes de la implantación de una arquitectura de microservicios, lo más importante antes de decidir qué arquitectura elegir, será determinar qué solución es la que mejor se adapta a las necesidades del proyecto o la organización y qué ayudará a conseguir los objetivos propuestos.

Está claro que para empresas líderes como Google, Amazon, Netflix o Uber, con sistemas grandes y con mucha carga computacional, los microservicios son la mejor opción. Pero antes de tomar la decisión de implantar una arquitectura de microservicios, se deben tener en cuenta varias cuestiones. Como por ejemplo: el número de usuarios finales, el volumen de peticiones, los picos de demanda, el tamaño de la empresa y el equipo del que se dispone, si en la

actualidad se cuenta con una arquitectura monolítica, si hay alguien en el equipo con la experiencia necesaria o se deben contratar servicios, etc.

Una arquitectura de microservicios tendrá más sentido cuanto más diversa y grande sea la audiencia, y cuanto mayor sea el tamaño de la organización, el equipo y los servicios ofrecidos.

¿Qué tamaño debe tener un microservicio?

Al desarrollar un microservicio, el tamaño no debe ser lo más importante. En su lugar, el punto importante debe ser crear libremente servicios acoplados para que tenga autonomía de desarrollo, implementación y escala, para cada servicio. Por supuesto, al identificar y diseñar microservicios, debe intentar que sean lo más pequeños posible, siempre y cuando no tenga demasiadas dependencias directas con otros microservicios. Más importante que el tamaño del microservicio es la cohesión interna que debe tener y su independencia respecto a otros servicios.

¿Por qué se debe tener una arquitectura de microservicios?

En resumen, proporciona agilidad a largo plazo. Con los microservicios puede crear aplicaciones basadas en muchos servicios que se pueden implementar de forma independiente y que tienen ciclos de vida granulares y autónomos, lo que permite un mejor mantenimiento en sistemas complejos, grandes y altamente escalables.

El rol de las API

Las ventajas que ofrecen los microservicios vienen acompañadas de ciertas concesiones como la dispersión del servicio, una mayor complejidad y el riesgo de trabajo redundante. Las organizaciones necesitan aunar los microservicios y las API para implementar la arquitectura de forma eficaz.

El reto es aprender a combinar la arquitectura de microservicios con los muchos otros patrones de arquitectura que ya se han desplegado en la organización. El uso de APIs es una forma de gestionar la velocidad y flexibilidad que proporcionan los microservicios, y además mantiene a raya el nivel de complejidad. Una estrategia de API facilita la gestión de los microservicios y les permite coexistir con los sistemas heredados existentes, evitando su aislamiento de los sistemas esenciales. Combinar una arquitectura de microservicios con una estrategia de API holística es una manera contrastada de obtener las ventajas de los microservicios al tiempo que se limitan sus desventajas.

¿QUÉ ES UNA API?

Una interfaz de programa de aplicación (API) define las reglas que se deben seguir para comunicarse con otros sistemas de software, es un intermediario de software entre dos aplicaciones que interactúan entre sí. Conecta dos ordenadores o programas informáticos a través de una interfaz y no es para que la utilice directamente el usuario final, salvo el programador que quiera integrarla en una solución de software.

Las API simplifican la programación y, de hecho, pueden ocultar los detalles internos de un sistema, como su funcionamiento, y exponer partes útiles para un programador, manteniendo la coherencia de las partes a pesar de los cambios internos. Hoy en día se puede encontrar una gran variedad de API para diversos fines, como sistemas operativos, bibliotecas de software, lenguajes de programación, hardware informático, etc.

¿Qué es una API REST?

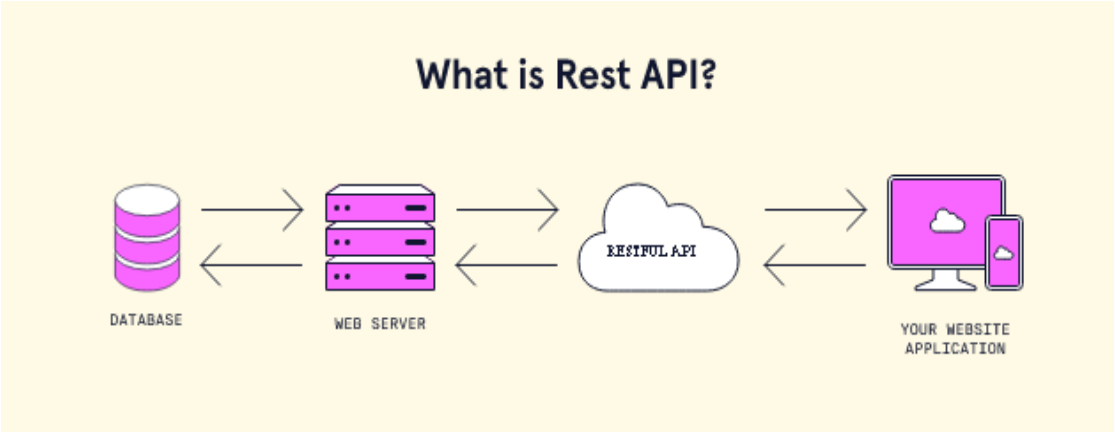


Figura 2. Esquema representativo de API Rest.

Una API REST es una API que se ajusta a los principios de diseño del REST. REST, o Representational State Transfer, es un estilo arquitectónico para proporcionar estándares entre sistemas informáticos en la web, lo que facilita que los sistemas se comuniquen entre sí. Los sistemas compatibles con REST, a menudo llamados sistemas RESTful, se caracterizan por la forma en que no tienen estado y separan las preocupaciones del cliente y el servidor.

Definido por primera vez en 2000 por el científico informático Dr. Roy Fielding en su tesis doctoral, REST proporciona un nivel relativamente alto de flexibilidad y libertad para los desarrolladores. Esta flexibilidad es solo una de las razones por las que las API REST se han convertido en un método común para conectar componentes y aplicaciones en una arquitectura de microservicios.

Principios de diseño REST

En el nivel más básico, una API es un mecanismo que permite a una aplicación o servicio acceder a un recurso dentro de otra aplicación o servicio. La aplicación o servicio que realiza el acceso se denomina cliente, y la aplicación o servicio que contiene el recurso se denomina servidor.

Algunas API, como SOAP o XML-RPC, imponen un marco estricto a los desarrolladores. Pero las API REST se pueden desarrollar utilizando prácticamente cualquier lenguaje de programación y admiten una variedad de formatos de datos.

Una implementación concreta de un servicio web REST sigue cuatro principios de diseño fundamentales:

1. **Utiliza los métodos HTTP de manera explícita:**

Una de las características clave de un servicio web RESTful es el uso explícito de métodos HTTP de una manera que sigue el protocolo definido por RFC 2616. HTTP GET, por ejemplo, se define como un método de producción de datos destinado a ser utilizado por una aplicación cliente para recuperar un recurso, para obtener datos de un servidor web o para ejecutar una consulta con la expectativa de que el servidor web buscará y responderá con un conjunto de recursos coincidentes.

REST pide a los desarrolladores que usen métodos HTTP explícitamente y de una manera que sea coherente con la definición del protocolo. Este principio básico de diseño REST establece una asignación uno a uno entre las operaciones de creación, lectura, actualización y eliminación (CRUD) y los métodos HTTP. Según este mapeo:

- Para crear un recurso en el servidor, utilice POST.
- Para recuperar un recurso, utilice GET.
- Para cambiar el estado de un recurso o actualizarlo, utilice PUT.
- Para quitar o eliminar un recurso, utilice DELETE

  .

2. **Expone URL´s con forma de directorios:**

Desde el punto de vista de las aplicaciones cliente que abordan los recursos, los URI determinan cuán intuitivo será el servicio web REST y si el servicio se usará de manera que los diseñadores puedan anticiparse. Una tercera característica del servicio web RESTful tiene que ver con los URI.

Los URI del servicio web REST deben ser intuitivos hasta el punto en que sean fáciles de adivinar. Piense en un URI como un tipo de interfaz autodocumentada que requiere poca o

ninguna explicación o referencia para que un desarrollador entienda a qué apunta y derive recursos relacionados. Con este fin, la estructura de un URI debe ser sencilla, predecible y fácil de entender.

Una forma de lograr este nivel de usabilidad es definir URI similares a la estructura de directorios. Este tipo de URI es jerárquico, tiene su raíz en una sola ruta y se ramifican a partir de ella subrutas que exponen las áreas principales del servicio. De acuerdo con esta definición, un URI no es simplemente una cadena delimitada por barras, sino más bien un árbol con ramas subordinadas y superiores conectadas en nodos.

3. **Stateless**

Una aplicación de servicio web REST (o cliente) incluye dentro de los encabezados HTTP y el cuerpo de una solicitud todos los parámetros, el contexto y los datos que necesita el componente del lado servidor para generar una respuesta. . En otras palabras, las API de REST no requieren ninguna sesión del lado del servidor, no tienen estado. Si el acceso a un recurso requiere autenticación, el cliente debe autenticarse con cada solicitud. No tener estado en este sentido mejora el rendimiento del servicio web y simplifica el diseño y la implementación de los componentes del lado del servidor porque la ausencia de estado en el servidor elimina la necesidad de sincronizar los datos de sesión con una aplicación externa.

4. **Transferir XML, JSON o ambos**

Una representación de recursos suele reflejar el estado actual de un recurso y sus atributos en el momento en que una aplicación cliente lo solicita. Las representaciones de recursos en este sentido son meras instantáneas en el tiempo. Esto podría ser algo tan simple como una representación de un registro en una base de datos que consiste en una asignación entre nombres de columna y etiquetas XML, donde los valores de elemento en el XML contienen los valores de fila. O bien, si el sistema tiene un modelo de datos, de acuerdo con esta definición, una representación de recursos es una instantánea de los atributos de una de las cosas del modelo de datos del sistema.

Los objetos del modelo de datos suelen estar relacionados de alguna manera, y las relaciones entre los objetos del modelo de datos (recursos) deben reflejarse en la forma en que se representan para transferirlos a una aplicación cliente.

Y para dar a las aplicaciones cliente la capacidad de solicitar un tipo de contenido específico que sea más adecuado para ellas, se debe construir su servicio para que haga uso del encabezado HTTP Accept integrado, donde el valor del encabezado es un tipo MIME. Algunos tipos MIME comunes utilizados por los servicios RESTful son:

Tipo MIME	Tipo de contenido
JSON	aplicación/json
.XML	Aplicación/XML
.XHTML	application/xhtml xml

Figura 3. Tipos MIME

Esto permite que el servicio sea utilizado por una variedad de clientes escritos en diferentes idiomas que se ejecutan en diferentes plataformas y dispositivos. El uso de tipos MIME y el encabezado HTTP Accept es un mecanismo conocido como negociación de contenido, que permite a los clientes elegir qué formato de datos es el adecuado para ellos y minimiza el acoplamiento de datos entre el servicio y las aplicaciones que lo utilizan.

Otras restricciones arquitectónicas también incluyen:

Desacoplamiento cliente-servidor. En el estilo arquitectónico REST, la implementación del cliente y la implementación del servidor se pueden hacer de forma independiente sin que cada uno sepa del otro. Esto significa que el código en el lado del cliente se puede cambiar en cualquier momento sin afectar el funcionamiento del servidor, y el código en el lado del servidor se puede cambiar sin afectar el funcionamiento del cliente.

Mientras cada lado sepa qué formato de mensajes enviar al otro, se pueden mantener modulares y separados. Al separar las preocupaciones de la interfaz de usuario de las preocupaciones de almacenamiento de datos, mejoramos la flexibilidad de la interfaz en todas las plataformas y mejoramos la escalabilidad simplificando los componentes del servidor. Además, la separación permite a cada componente la capacidad de evolucionar de forma independiente.

Interfaz uniforme. Para desacoplar un cliente del servidor, debe tener una interfaz unificada que permita el desarrollo autónomo de la aplicación sin acoplar estrechamente sus servicios, modelos y acciones a la capa API.

Este principio de diseño agiliza toda la arquitectura del sistema y mejora la visibilidad de las comunicaciones. Para lograr una interfaz uniforme, se requieren varios controles arquitectónicos para guiar el desempeño de los elementos dentro de la arquitectura de la API REST.

Los principios REST están definidos por cuatro controles de interfaz, que incluyen la identificación de recursos, la gestión de recursos a través de representaciones, comunicaciones autodescriptivas e hipermedia como motor del estado de la aplicación.

Capacidad de caché. Cuando sea posible, los recursos deben poder almacenarse en caché en el lado del cliente o del servidor. Las respuestas del servidor también deben contener información sobre si se permite el almacenamiento en caché para el recurso entregado. El objetivo es mejorar el rendimiento en el lado del cliente, al tiempo que aumenta la escalabilidad en el lado del servidor.

Arquitectura de sistema en capas. La arquitectura de la API REST incluye varias capas que operan juntas para construir una jerarquía que ayuda a generar una aplicación más escalable y flexible. Debido a su sistema de capas, una aplicación tiene mejor seguridad ya que los componentes de cada capa no pueden interactuar fuera de la capa siguiente. Además, equilibra las cargas y ofrece cachés compartidos para estimular escalabilidad.

Un sistema de arquitectura de API REST en capas tiene una mayor estabilidad porque restringe el rendimiento de los componentes. de modo que cada componente no pueda "ver" más allá de la capa inmediata con la que se entremezcla.

Código bajo demanda. Este principio REST permite que la codificación o los applets se comuniquen a través de la API utilizada dentro de la aplicación.

Una definición de API REST permite ampliar la funcionalidad del cliente descargando e implementando codificación en forma de applets o scripts. Esto agiliza a los clientes al disminuir la cantidad de características esenciales para ser implementadas previamente.

La mayoría de las veces, un servidor devuelve la representación de recursos estáticos en formato XML o JSON. Pero cuando es necesario, los servidores pueden entregar código ejecutable al cliente.

Cómo funcionan las API de REST

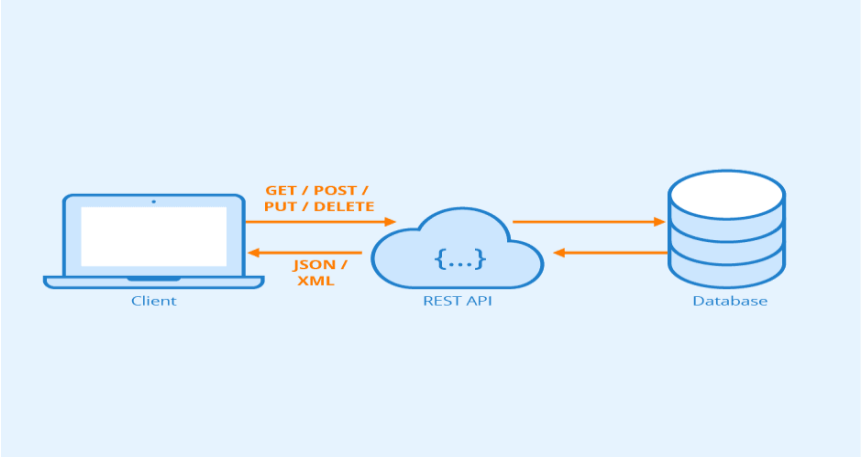


Figura 4. Representación del funcionamiento API Rest.

Las API REST se comunican a través de solicitudes HTTP para realizar funciones de base de datos estándar como crear, leer, actualizar y eliminar registros (también conocidos como CRUD) dentro de un recurso. Por ejemplo, una API REST usaría una solicitud GET para recuperar un registro, una solicitud POST para crear uno, una solicitud PUT para actualizar un registro y una solicitud DELETE para eliminar uno. Todos los métodos HTTP se pueden usar en llamadas a la API. Una API REST bien diseñada es similar a un sitio web que se ejecuta en un navegador web con funcionalidad HTTP integrada.

El estado de un recurso en cualquier instante determinado, o marca de tiempo, se conoce como la representación del recurso. Esta información se puede entregar a un cliente en prácticamente cualquier formato, incluida la notación de objetos JavaScript (JSON), HTML, XLT, Python, PHP o texto sin formato. JSON es popular porque es legible tanto por humanos como por máquinas, y es independiente del lenguaje de programación.

Los encabezados y parámetros de solicitud también son importantes en las llamadas a la API de REST porque incluyen información de identificador importante, como metadatos, autorizaciones, identificadores uniformes de recursos (URI), almacenamiento en caché, cookies y más. Los encabezados de solicitud y los encabezados de respuesta, junto con los códigos de estado HTTP convencionales, se utilizan dentro de API REST bien diseñadas.

En los casos en que el servidor envía una carga útil de datos al cliente, el servidor debe incluir un encabezado de la respuesta. Este campo de encabezado alerta al cliente sobre el tipo de datos que está enviando en el cuerpo de la respuesta. Estos tipos de contenido son tipos MIME, al igual que en el campo del encabezado de solicitud. El que el servidor envía de vuelta en la respuesta debe ser una de las opciones que el cliente especificó en el campo de la solicitud.

Las respuestas del servidor contienen códigos de estado para alertar al cliente sobre la información sobre el éxito de la operación. Los más comunes y cómo se utilizan:

Código de estado	Significado
200 (OK)	Esta es la respuesta estándar para las solicitudes HTTP correctas.
201 (CREADO)	Esta es la respuesta estándar para una solicitud HTTP que dio como resultado la creación correcta de un elemento.
204 (SIN CONTENIDO)	Esta es la respuesta estándar para las solicitudes HTTP correctas, donde no se devuelve nada en el cuerpo de la respuesta.
400 (MALA SOLICITUD)	La solicitud no se puede procesar debido a una sintaxis de solicitud incorrecta, tamaño excesivo u otro error del cliente.
403 (PROHIBIDO)	El cliente no tiene permiso para acceder a este recurso.
404 (NO ENCONTRADO)	No se pudo encontrar el recurso en este momento. Es posible que se haya eliminado o que aún no exista.
500 (ERROR INTERNO DEL SERVIDOR)	La respuesta genérica para un error inesperado si no hay información más específica disponible.

Figura 5. Tabla de códigos de estado

Para cada verbo HTTP, hay códigos de estado esperados que un servidor debe devolver si se realiza correctamente:

GET — devolver 200 (OK)

POST — return 201 (CREADO)

PUT — devolver 200 (OK)

DELETE — return 204 (SIN CONTENIDO) Si se produce un error en la operación, devuelva el código de estado más específico posible correspondiente al problema encontrado.

Ventajas y desventajas de la API REST

Ventajas de la API REST

- La API REST es fácil de entender y aprender, debido a su simplicidad, API conocida.
- Con la API REST, poder organizar aplicaciones complicadas y facilita el uso de recursos.
- La alta carga se puede administrar con la ayuda del servidor proxy HTTP y la memoria caché.
- La API de REST es fácil de explorar y descubrir.
- Facilita que los nuevos clientes trabajen en otras aplicaciones, ya sea que esté diseñado específicamente para un propósito o no.
- Utilice llamadas de procedimiento HTTP estándar para recuperar datos y solicitudes.

- La API REST depende de códigos, puede usarla para sincronizar datos con un sitio web sin ninguna complicación.
- Los usuarios pueden aprovechar el acceso a los mismos objetos estándar y modelo de datos en comparación con los servicios web basados en SOAP.
- Aporta formatos flexibles mediante la serialización de datos en formato XML o JSON.
- Permite la protección basada en estándar con el uso de protocolos O Auth para verificar sus solicitudes REST.

Desventajas o desafíos en REST:

- Falta de estado: la mayoría de las aplicaciones web requieren mecanismos con estado. Supongamos que compra un sitio web que tiene un mecanismo para tener un carrito de compras. Se requiere saber el número de artículos en el carrito de compras antes de realizar la compra real. Esta carga de mantener el estado recae en el cliente, lo que hace que la aplicación cliente sea pesada y difícil de mantener.
- Último de seguridad: REST no impone seguridad como SOAP. Esa es la razón por la que REST es apropiado para las URL públicas, pero no es bueno para el paso de datos confidenciales entre el cliente y el servidor.

¿Por qué elegir REST?

- **Escalabilidad**. Este protocolo destaca por su escalabilidad. Gracias a la separación entre cliente y servidor, un producto puede ser escalado por un equipo de desarrollo sin mucha dificultad.

- **Flexibilidad y portabilidad.** Con el requisito indispensable de que los datos de una de las solicitudes se envíen correctamente, es posible realizar una migración de un servidor a otro o realizar cambios en la base de datos en cualquier momento. Por lo tanto, la parte delantera y trasera se pueden alojar en diferentes servidores, lo que es una ventaja de administración significativa.

- **Independencia.** Con la separación entre cliente y servidor, el protocolo facilita que los desarrollos en un proyecto se lleven a cabo de forma independiente. Además, la API REST se adapta en todo momento a la sintaxis y plataforma de trabajo. Esto ofrece la oportunidad de utilizar múltiples entornos durante el desarrollo.

¿Cuándo usar REST?

- Ancho de banda y recursos limitados: REST debe usarse donde el ancho de banda de red es la restricción.

- Facilidad de codificación: Si se requiere una solución rápida para los servicios web, entonces el servicio web REST es la mejor manera.
- Caché: si hay una necesidad de caché y muchas solicitudes, entonces REST es el perfecto. Esto aumenta el número de solicitudes que se envían al servidor. Al implementar una memoria caché, los resultados de las consultas más frecuentes se almacenan en una solución intermedia. Cada vez que los clientes solicitan un recurso, comprueba la primera memoria caché. Si existen recursos, entonces no procederá al servidor. El almacenamiento en caché puede ayudar a minimizar el número de viajes.

Microservicios VS API

Una API es un canal de comunicación entre servicios que permite a los clientes utilizar los servicios subyacentes de una aplicación. Mientras que un microservicio es un diseño arquitectónico que divide una aplicación en pequeños servicios de contenido reducido, lo que facilita la creación y el mantenimiento del software.

Aunque ambos son diferentes entre sí, generalmente están emparejados, ya que los servicios de la arquitectura de microservicios usan API para comunicarse entre sí. Dentro de una aplicación basada en microservicios, cada servicio posee una API separada que determina qué solicitudes recibe y la forma en que responde.

Una cosa importante a tener en cuenta aquí es que ningún microservicio es igual y cada uno de ellos usa API de manera diferente. Algunos pueden asignar una API para acceder a diferentes servicios o varias API a un servicio.

Por último, las API no se limitan a los microservicios, tienen usos mucho más allá. Pueden usarse para compartir datos entre sistemas en la aplicación web o usarse internamente sin la implementación de microservicios.

Otros estilos de arquitectura para API

Figura 6. Línea de tiempo de arquitecturas API

RCP (Remote Procedure Call): es una especificación que permite la ejecución remota de una función en un contexto diferente. RPC amplía la noción de llamada a procedimientos locales, pero la pone en el contexto de una API HTTP.

XML-RPC inicial fue problemático porque garantizar los tipos de datos de las cargas útiles XML es difícil. Entonces, más tarde, API RPC comenzó a usar una especificación JSON-RPC más concreta que se considera una alternativa más simple a SOAP. gRPCes la última versión de RPC desarrollada por Google en 2015. Con soporte conectable para equilibrio de carga, seguimiento, comprobación de estado y autenticación, gRPC es adecuado para conectar microservicios.

gRPC (Remote Procedure Call) es una tecnología de intercambio de datos de código abiertodesarrollada por Google utilizando el protocolo HTTP / 2.

Utiliza el formato binario Protocol Buffers (Protobuf) para el intercambio de datos. Además, este estilo arquitectónico aplica reglas que un desarrollador debe seguir para desarrollar o consumir API web.

Diferencias con Rest:

REST es un conjunto de directrices para diseñar API web sin hacer cumplir nada. Por otro lado, gRPC aplica reglas definiendo un archivo .proto que debe ser respetado tanto por el cliente como por el servidor para el intercambio de datos.

REST proporciona un modelo de comunicación de solicitud-respuesta basado en el protocolo HTTP 1.1. Por lo tanto, cuando varias solicitudes llegan al servidor, está obligado a manejar

cada una de ellas, una a la vez. Sin embargo, gRPC sigue un modelo de comunicación cliente-respuesta paradiseñar API web que se basan en HTTP/2. Por lo tanto, gRPC permite la transmisión de comunicaciones y atiende múltiples solicitudes simultáneamente. Además de eso, gRPC también admite comunicación unaria similar a REST.

REST normalmente utiliza formatos JSON y XML para la transferencia de datos. Sin embargo, gRPC se basa en Protobuf para un intercambio de datos a través del protocolo HTTP / 2.

REST, en la mayoría de los casos, utiliza JSON oXML que requiere serialización y conversión al lenguaje de programación de destino tanto para el cliente como para el servidor, lo que aumenta el tiempo de respuesta y la posibilidad de errores al analizar la solicitud / respuesta.

Sin embargo, gRPC proporciona mensajes fuertemente tipados convertidos automáticamente utilizando el formato de intercambio Protobuf al lenguaje de programación elegido.

REST utilizando HTTP 1.1 requiere un protocolo de enlace TCP para cada solicitud. Por lo tanto, las API REST con HTTP 1.1 pueden sufrir problemas de latencia.

Por otro lado, gRPC se basa en el protocolo HTTP / 2, que utiliza flujos multiplexados. Por lo tanto, varios clientes pueden enviar varias solicitudes simultáneamente sin establecer una nueva conexión TCP para cada una. Además, el servidor puede enviar notificaciones push a los clientes a través de la conexión establecida.

Las API de REST en HTTP 1.1 tienen compatibilidad universal con exploradores. Sin embargo, gRPC tiene soporte de navegador limitado porque numerosos navegadores (generalmente las versiones anteriores) no tienen soporte maduro para HTTP / 2. Por lo tanto, puede requerir gRPC-web y una capa proxy para realizar conversiones entre HTTP 1.1 y HTTP / 2. Por lo tanto, en este momento, gRPC se utiliza principalmente para servicios internos.

REST no proporciona características integradas de generación de código. Sin embargo, podemos usar herramientas de terceros como Swagger o Postman para producir código para solicitudes de API.

Por otro lado, gRPC, utilizando su compilador protoc, viene con características nativas de generación de código, compatibles con varios lenguajes de programación.

REST es útil para integrar microservicios y aplicaciones de terceros con los sistemas centrales. gRPC puede encontrar su aplicación en varios sistemas, como sistemas IoT que requieren transmisión de mensajes liviana, aplicaciones móviles sin soporte de navegador y aplicaciones que necesitan flujos multiplexados.

SOAP(Simple Object Access Protocol): El formato de datos XML arrastra detrás de mucha formalidad. Junto con la estructura de mensajes masivos, hace que SOAP sea el estilo de API más detallado.

Un mensaje SOAP se compone de:

- una etiqueta de sobre que comienza y termina cada mensaje,
- un organismo que contiene la solicitud o respuesta
- un encabezado si un mensaje debe determinar cualquier detalle o requisito adicional, y
- Un fallo que informa de cualquier error que pueda ocurrir a lo largo del procesamiento de la solicitud.

```
<?xml version='1.0' Encoding='UTF-8' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
                 env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2007-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
                 env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
   <n:name>Fred Bloggs</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2007-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2007-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference></p:seatPreference>
   </p:return>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

Figura 7. Un ejemplo del mensaje SOAP.

La lógica de la API SOAP está escrita en lenguaje de descripción de servicios web (WSDL). Este lenguaje de descripción de API define los puntos finales y describe todos los procesos que se pueden realizar. Esto permite que diferentes lenguajes de programación e IDE configuren rápidamente la comunicación.

SOAP admite mensajes con y sin estado. En un escenario con estado, el servidor almacena la información recibida que puede ser realmente pesada. Pero está justificado para operaciones que involucran a múltiples partes y transacciones complejas.

	REST	SOAP
Características	Las operaciones se definen en los mensajes. Una dirección única para cada instancia del proceso. Cada objeto soporta las operaciones estándares definidas. Componentes débilmente acoplados.	Las operaciones son definidas como puertos WSDL. Dirección única para todas las operaciones. Múltiple instancias del proceso comparten la misma operación. Componentes fuertemente acoplados.
Ventajas declaradas	Bajo consumo de recursos. Las instancias del proceso son creadas explícitamente. El cliente no necesita información de enrutamiento a partir de la URI inicial. Los clientes pueden tener una interfaz "listener" (escuchadora) genérica para las notificaciones. Generalmente fácil de construir y adoptar.	Fácil (generalmente) de utilizar. La depuración es posible. Las operaciones complejas pueden ser escondidas detrás de una fachada. Envolver APIs existentes es sencillo Incrementa la privacidad. Herramientas de desarrollo.
Posibles desventajas	Gran número de objetos. Manejar el espacio de nombres (URIs) puede ser engorroso. La descripción sintáctica/semántica muy informal (orientada al usuario). Pocas herramientas de desarrollo.	Los clientes necesitan saber las operaciones y su semántica antes del uso. Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones. Las instancias del proceso son creadas implícitamente.

Figura 8. Diferencias entre REST y SOAP

GraphQL: Es una sintaxis que describe cómo realizar una solicitud de datos precisa. La implementación de GraphQL vale la pena para el modelo de datos de una aplicación con muchas entidades complejas que se referencian entre sí.

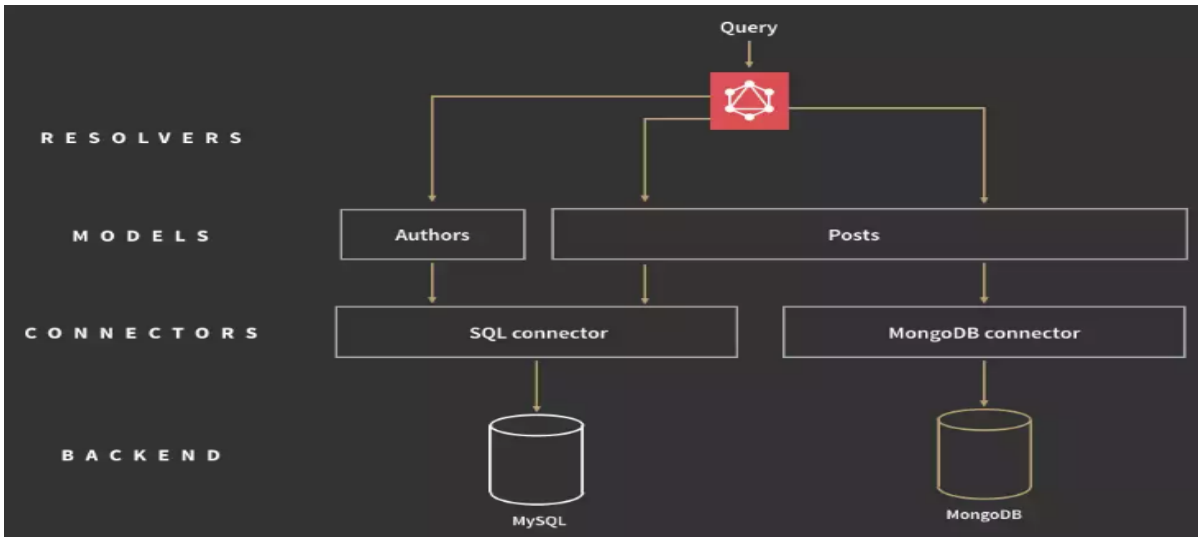


Figura 9. Ejecución de consultas en GraphQL

GraphQL comienza con la creación de unesquema, que es una descripción de todas las consultas que puede realizar en una API de GraphQL y todos lostiposque devuelven. La creación de esquemas es difícil, ya que requiere una escritura sólida en el lenguaje de definición de esquemas (SDL).

Al tener el esquema antes de realizar la consulta, un cliente puede validar su consulta para asegurarse de que el servidor podrá responder a ella. Al llegar a la aplicación backend, una operación GraphQL se interpreta contra todo el esquema y se resuelve con datos para la aplicación frontend. Al enviar una consulta masiva al servidor, la API devuelve una respuesta JSON con exactamente la forma de los datos que solicitamos.

Prácticas recomendadas de la API de REST

Aunque la flexibilidad es una gran ventaja del diseño de la API REST, esa misma flexibilidad facilita el diseño de una API que está rota o funciona mal. Por este motivo, los desarrolladores profesionales comparten las prácticas recomendadas en las especificaciones de la API de REST.

La especificación OpenAPI (OAS) establece una interfaz para describir una API de una manera que permita a cualquier desarrollador o aplicación descubrirla y comprender completamente sus parámetros y capacidades: puntos finales disponibles, operaciones permitidas en cada punto final, parámetros de operación, métodos de autenticación y otra información. La última versión, OAS3, incluye herramientas prácticas, como OpenAPI Generator, para generar clientes API y stubs de servidor en diferentes lenguajes de programación.

Asegurar una API REST también comienza con las mejores prácticas de la industria, como el uso de algoritmos hash para la seguridad de contraseñas y HTTPS para la transmisión segura de datos. Un marco de autorización como OAuth 2.0 puede ayudar a limitar los privilegios de las aplicaciones de terceros. Usando una marca de tiempo en el encabezado HTTP, una API también puede rechazar cualquier solicitud que llegue después de un cierto período de tiempo. La validación de parámetros y los tokens web JSON son otras formas de garantizar que solo los clientes autorizados puedan acceder a la API

OPENAPI

OpenAPI es un estándar para la descripción de las interfaces de programación, o application programming interfaces (API). La especificación OpenAPI define un formato de descripción abierto e independiente de los fabricantes para los servicios de API. En particular, OpenAPI puede utilizarse para describir, desarrollar, probar y documentar las API compatibles con REST.

La actual especificación OpenAPI surgió del proyecto predecesor Swagger. La empresa de desarrollo SmartBear sometió la especificación existente de Swagger a una licencia abierta y dejó el mantenimiento y desarrollo posterior en manos de la iniciativa OpenAPI. Además de SmartBear, entre los miembros de la iniciativa OpenAPI se encuentran gigantes de la industria como Google, IBM y Microsoft. La Fundación Linux también apoya este proyecto.

Con OpenAPI, una API puede describirse de manera uniforme. Esto se conoce como "definición API" y se genera en un formato legible por máquina. En particular, se utilizan dos lenguajes: YAML y JSON.

Técnicamente, YAML y JSON difieren solo ligeramente, por lo que es posible convertir automáticamente una definición API existente de un lenguaje a otro. Sin embargo, YAML tiene una estructura más clara y es más fácil de leer para las personas. A continuación, un ejemplo del mismo objeto del servidor OpenAPI, presentado en YAML y JSON:

La especificación OpenAPI define una serie de propiedades que pueden utilizarse para desarrollar una API propia. Estas propiedades se agrupan en los llamados **objetos** (en inglés, *objects*). En la actual versión 3.0.3, OpenAPI define la estructura de los siguientes objetos, entre otros:

- **Info Object:** versión, nombre, etc. de la API.
- **Contact Object:** datos de contacto del proveedor de la API.
- **License Object:** licencia bajo la cual la API proporciona sus datos.
- **Server Object:** nombres del host, estructura del URL y puertos del servidor a través del cual se dirige la API.
- **Components Object:** componentes encapsulados que pueden utilizarse varias veces dentro de una definición de API.
- **Paths Object:** rutas relativas a los puntos finales de la API que se utilizan junto con el servidor del objeto.
- **Path Item Object**: operaciones permitidas para una ruta específica como GET, PUT, POST, DELETE.
- **Operation Object:** especifica, entre otras cosas, los parámetros y las respuestas del servidor que se esperan de una operación.

¿Cuáles son las áreas de aplicación de OpenAPI?

En general, OpenAPI se utiliza para describir API REST de manera uniforme. Como esta descripción, es decir, la definición API, está disponible en un formato legible por máquina, se pueden generar automáticamente diversos artefactos virtuales a partir de ella. En concreto, estos incluyen:

- Creación de documentación API: La documentación basada en HTML se genera automáticamente a partir de la definición API legible por máquina. Esta sirve como material de consulta para los desarrolladores que acceden a los servicios API. Si la definición API cambia, la documentación se vuelve a generar para que ambas concuerden.
- Creación de conexiones en diferentes lenguajes de programación: Con las herramientas apropiadas, se puede crear una biblioteca de software adecuada del lado del cliente a partir de la definición API en un lenguaje de programación compatible. Esto permite a los programadores de todo tipo acceder a la API. La biblioteca de software se incorpora de manera convencional. Por lo tanto, el acceso a los servicios de API tiene lugar, por ejemplo, mediante el acceso a las funciones dentro del mismo entorno de programación.
- Elaboración de casos de prueba: Cada componente de un software debe someterse a diversas pruebas para asegurar su funcionalidad. En concreto, es preciso volver a probar un componente de software cada vez que se cambia el código subyacente. A partir de la definición API, se pueden generar estos casos de prueba automáticamente para poder comprobar la funcionalidad de los componentes del software en todo momento.

En última instancia, cabe señalar que no todas las API pueden representarse utilizando OpenAPI. Sin embargo, las API REST son compatibles sin duda.

¿Qué ventajas tiene OpenAPI?

En general, la ventaja de OpenAPI es que la puesta en marcha, documentación y prueba de una API son coherentes y constantes durante el desarrollo y el mantenimiento. Además, el uso de la especificación OpenAPI permite una mejor coordinación del desarrollo de la API entre los equipos de backend y frontend. En ambos equipos, los componentes del código pueden generarse a partir de la definición API para que tanto en backend como en frontend puedan desarrollarlos y probarlos sin tener que esperar al otro.

Además, el uso de OpenAPI ofrece las siguientes ventajas:

- Definir las API HTTP independientemente de un lenguaje de programación específico.
- Generar código de servidor para una API definida en OpenAPI.
- Generar bibliotecas del lado del cliente para una API compatible con OpenAPI en más de 40 lenguajes de programación.
- Procesar una definición OpenAPI con las herramientas apropiadas.

- Crear documentación interactiva de API.
- Permitir que las personas y las máquinas descubran y entiendan las capacidades de un servicio sin tener que mirar el código fuente o la documentación adicional.
- Acceder a los servicios de API con un gasto mínimo de puesta en marcha.

¿Qué versiones de OpenAPI están disponibles y en qué se diferencian?

En el momento de la redacción de este artículo, la versión 3.0.3 de OpenAPI es la más actualizada. A continuación, presentamos un breve resumen de las versiones anteriores:

Versión	Nombre	Estado
1.0, agosto de 2011	Especificación Swagger	No está en uso
2.0, septiembre de 2014	Especificación Swagger > especificación OpenAPI	Todavía en uso
3.0, julio de 2017	Especificación OpenAPI	Todavía en uso

Figura 10. Versiones de OpenAPI

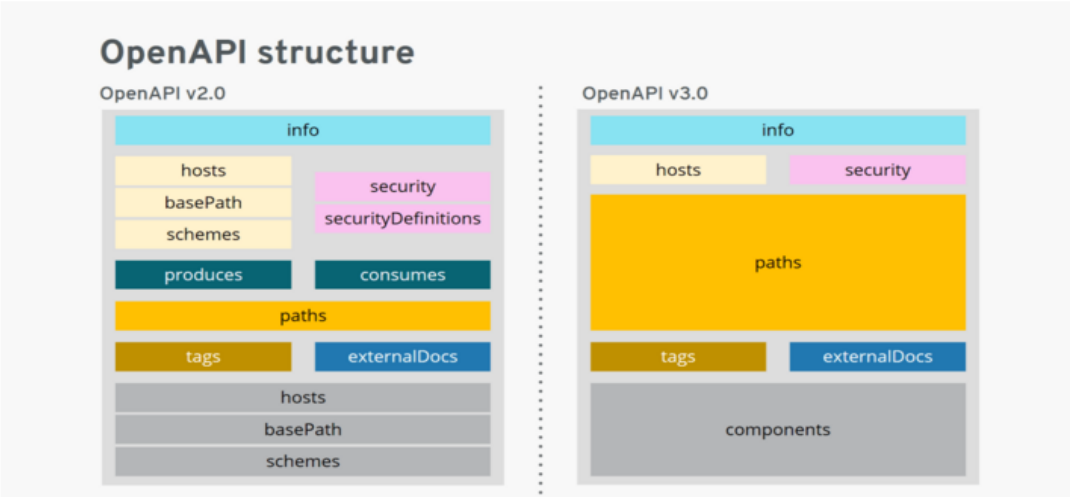


Figura 11. En general, la estructura de la especificación OpenAPI se ha simplificado.

SWAGGER

Swagger API es un conjunto de herramientas de código abierto creadas para ayudar a los programadores a desarrollar, diseñar, documentar y usar API REST. La herramienta se basa en la especificación OpenAPI y contiene tres componentes: Swagger Editor, Swagger UI y Swagger Codegen.

La especificación Swagger se conocía anteriormente como la especificación OpenAPI. La diferencia ahora es que OpenAPI es la instrucción o "blueprint" y Swagger es la

implementación de esas instrucciones. Por lo tanto, Swagger proporciona las herramientas para implementar la especificación OpenAPI. Las especificaciones de Swagger y OpenAPI describen la estructura de la API REST para que las máquinas puedan leerlas y simularlas. Gracias a las automatizaciones realizadas por OpenAPI y Swagger, el proceso de documentación de la API es mucho más fácil de generar y mantener para los desarrolladores.

SwaggerHub es una herramienta de documentación de API en línea diseñada para simplificar y acelerar la documentación de API. Swaggerhub se centra en un lenguaje de descripción de API: Swagger. Repasemos las muchas características de cada componente de Swagger: Swagger Editor, Swagger UI y Swagger Codegen.

Swagger Editor

Swagger Editor es un editor basado en navegador donde puede escribir y editar documentación de API y especificaciones de OpenAPI. Puede usar Swagger Editor a través del navegador, descargar para ejecutar localmente o usar una versión de host como SwaggerHub.

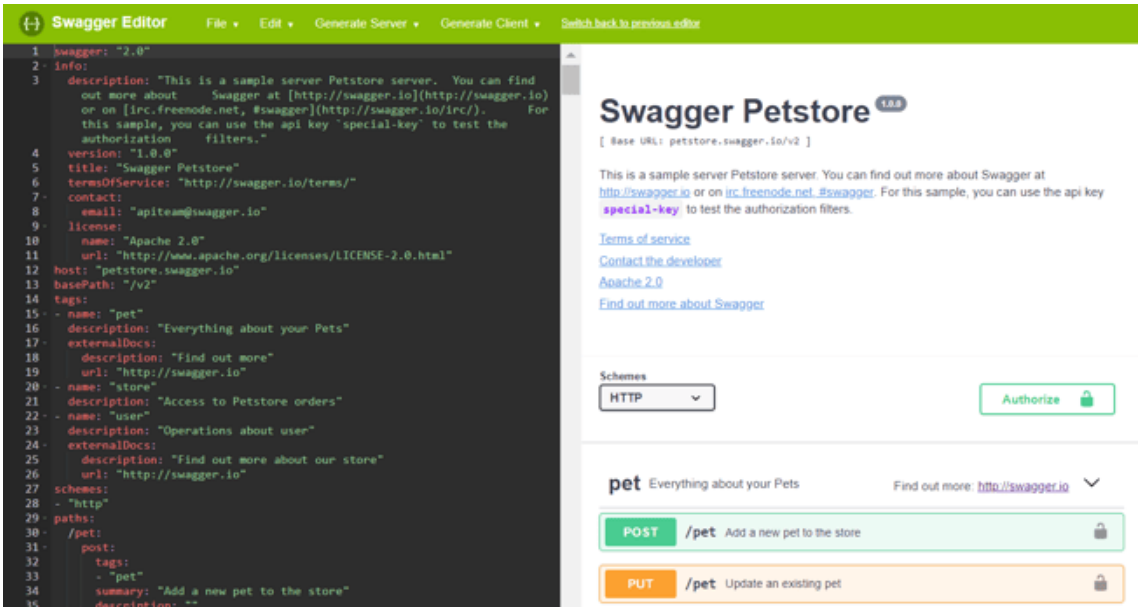


Figura 12. Vista de ejemplo del Editor de Swagger.

Swagger UI

Ahora que tenemos nuestra documentación creada con nuestro editor, necesitamos una forma de compartirla con nuestros usuarios. La interfaz de usuario de Swagger muestra las especificaciones de OpenAPI como documentación interactiva de API. Toma el archivo YAML y lo convierte en una documentación orientada al usuario que permite a los usuarios probar las llamadas a la API directamente en el navegador.

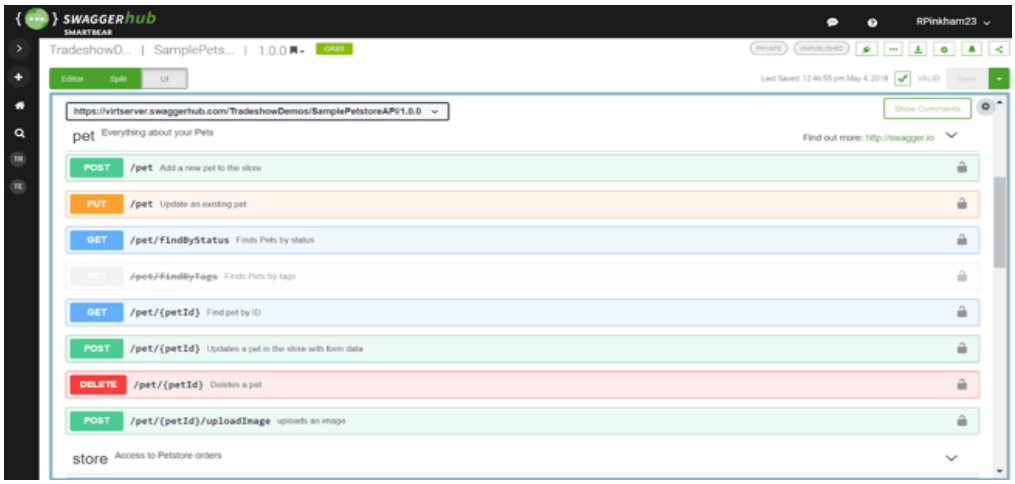


Figura 13. Visualización de la interfaz de usuario de Swagger

Swagger Codegen

.Swagger Codegen genera stubs de servidor, SDK de cliente y bibliotecas de cliente a partir de una especificación OpenAPI.

Puede crear rápida y fácilmente SDK de cliente para API en lenguajes comoJavaScript, Java, C #, Swift, etc. Un SDK de cliente contiene clases contenedoras que puede usar para llamar a la API desde la aplicación sin tener que lidiar con solicitudes y respuestas HTTP.

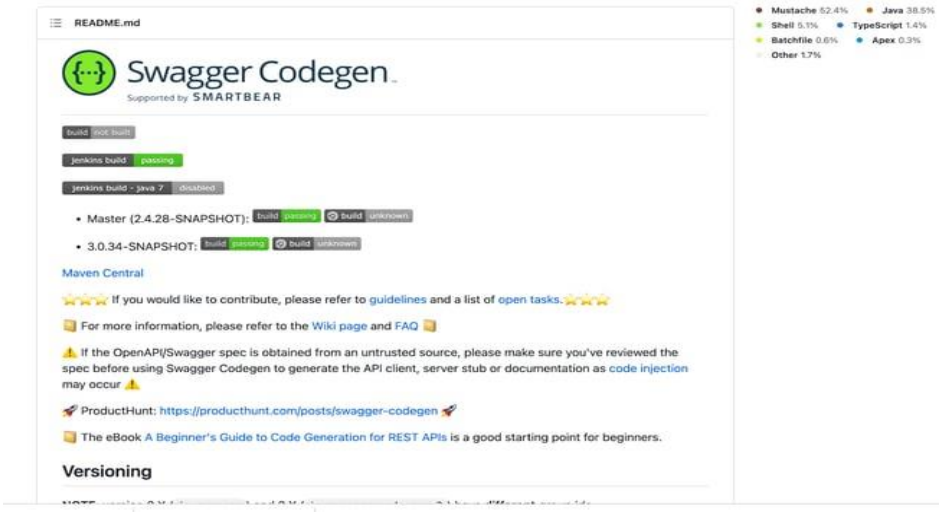
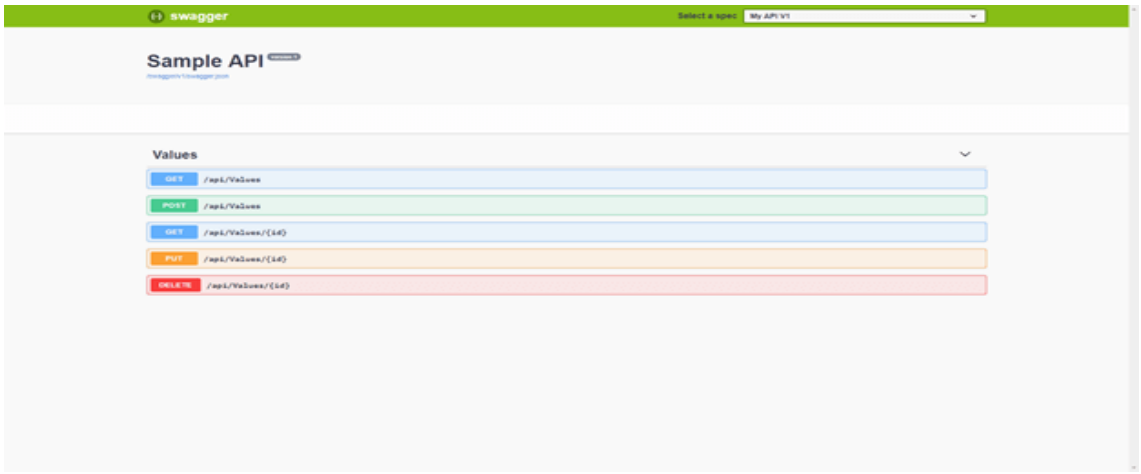


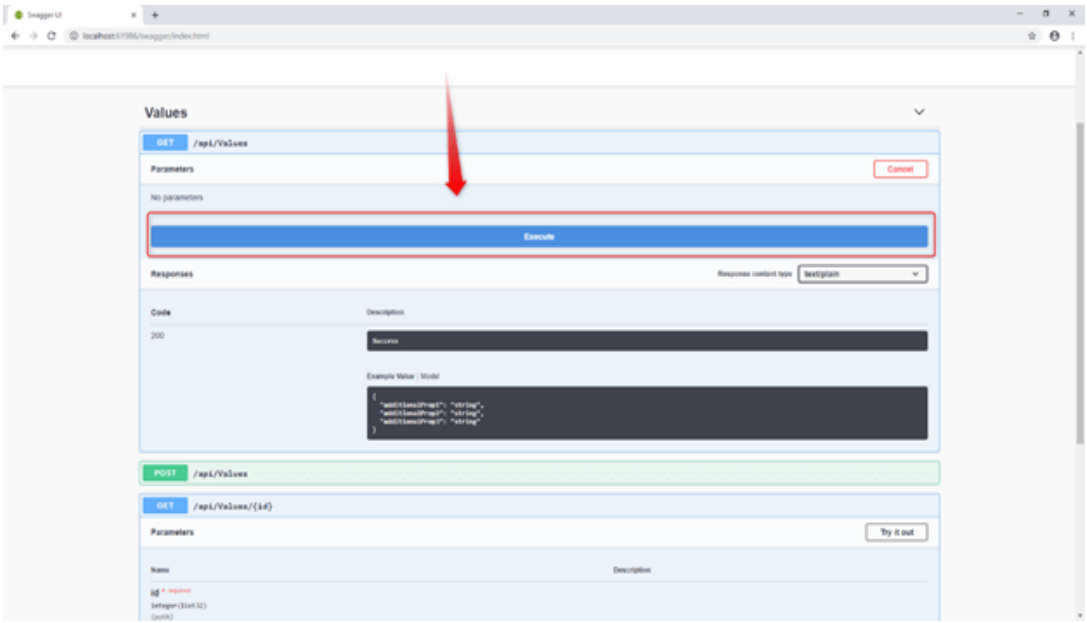
Figura 14. Archivo Readme de Github de Swagger Codegen

Cómo utilizar Swagger

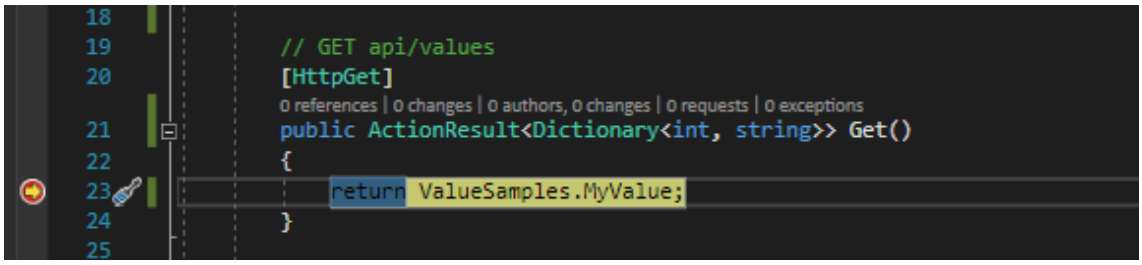
Una vez levantado el proyecto, con la URL de esta manera: http://localhost:61986/swagger , debe lucir así:



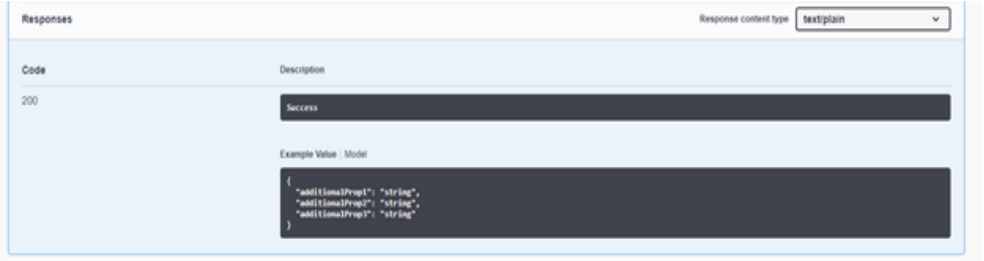
Para testear un método GET, llamamos al método desde Swagger



Se valida el método llamado desde el controlador.



Devuelve una respuesta.



OpenApi frente a Swagger

El proyecto Swagger fue donado a la Iniciativa OpenAPI en 2015 y desde entonces se ha denominado OpenAPI. Ambos nombres se usan indistintamente. Sin embargo, "OpenAPI" se refiere a la especificación. "Swagger" se refiere a la familia de productos comerciales y de código abierto de SmartBear que funcionan con la especificación OpenAPI. Los productos de código abierto posteriores, comoOpenAPIGenerator, también caen bajo el nombre de la familia Swagger, a pesar de no haber sido lanzados por SmartBear.

Especificación OpenAPI

La especificación OpenAPI es un documento que describe las capacidades de su API. El documento se basa en el XML y las anotaciones de atributos dentro de los controladores y modelos. Es la parte central del flujo de OpenAPI y se utiliza para impulsar herramientas como SwaggerUI.

Interfaz de usuario de Swagger

La interfaz de usuario de Swagger ofrece una interfaz de usuario basada en web que proporciona información sobre el servicio, utilizando la especificación OpenAPI generada. Tanto Swashbuckle como NSwag incluyen una versión integrada de la interfaz de usuario de Swagger, para que se pueda hospedar en su aplicación ASP.NET Core mediante una llamada de registro de middleware.

En resumen: OpenAPI es una especificación. Swagger es una herramienta que utiliza la especificación OpenAPI. Por ejemplo, OpenAPIGenerator y SwaggerUI.

Fuentes:

https://www.yogihosting.com/aspnet-core-consume-api/

https://medium.com/net-core/build-a-restful-web-api-with-asp-net-core-6-30747197e229

https://ninenines.eu/docs/en/cowboy/2.6/guide/rest_principles/

https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio

https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-6.0

https://www.codecademy.com/article/what-is-rest

https://developer.ibm.com/articles/ws-restful/

https://ichi.pro/es/microservicios-vs-api-en-que-se-diferencian-241946750622058

https://wlip.es/diferencia-entre-api-y-microservicios/

https://ladiferenciaentre.net/diferencia-entre-api-y-microservicios/

https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them

https://stoplight.io/openapi

https://www.baeldung.com/rest-vs-grpc

https://blog.hubspot.com/website/what-is-swagger

https://learn.microsoft.com/es-es/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture

https://decidesoluciones.es/arquitectura-de-microservicios/

https://www.arsys.es/blog/arquitectura-microservicios

https://supertokens.com/blog/what-is-jwt

https://aldeahost.com.mx/todo-lo-que-necesitas-saber-sobre-el-web-service

https://openwebinars.net/blog/arquitectura-de-software/

https://codigoestudianteblog.blogspot.com/2022/04/como-usar-jwt-en-api-rest-aspnet-core.html