

# Procedimientos recomendados para el diseño de LA API web RESTful

03/04/2025

Una implementación de la API web RESTful es una API web que emplea principios de arquitectura de Transmisión de Estado Representacional (REST) para lograr una interfaz sin estado, débilmente acoplada, entre un cliente y un servicio. Una API web que es RESTful admite el protocolo HTTP estándar para realizar operaciones en recursos y devolver representaciones de recursos que contienen vínculos de hipermedia y códigos de estado de operación HTTP.

Una API web RESTful debe alinearse con los siguientes principios:

- **Independencia de la plataforma**, lo que significa que los clientes pueden llamar a la API web independientemente de la implementación interna. Para lograr la independencia de la plataforma, la API web usa HTTP como protocolo estándar, proporciona documentación clara y admite un formato conocido de intercambio de datos, como JSON o XML.
- **Acoplamiento flexible**, lo que significa que el cliente y el servicio web pueden evolucionar de forma independiente. El cliente no necesita conocer la implementación interna del servicio web y el servicio web no necesita conocer la implementación interna del cliente. Para lograr un acoplamiento flexible en una API web RESTful, use solo protocolos estándar e implemente un mecanismo que permita al cliente y al servicio web aceptar el formato de los datos que se van a intercambiar.

En este artículo se describen los procedimientos recomendados para diseñar API web RESTful. También se tratan los patrones de diseño y las consideraciones comunes para crear API web fáciles de entender, flexibles y fáciles de mantener.

## Conceptos de diseño de la API web RESTful

Para implementar una API web RESTful, debe comprender los conceptos siguientes.

- **Identificador uniforme de recursos (URI)**: Las API REST están diseñadas en torno a los recursos, que son cualquier tipo de objeto, datos o servicio al que puede acceder el cliente. Cada recurso se representa mediante un URI que identifica de forma única ese recurso. Por ejemplo, el URI de un pedido de cliente en particular podría ser:

HTTP

<https://api.contoso.com/orders/1>

- La **representación de recursos** define cómo se codifica y transporta un recurso identificado por su URI a través del protocolo HTTP en un formato específico, como XML o JSON. Los clientes que quieran recuperar un recurso específico deben usar el URI del recurso en la solicitud a la API. La API devuelve una representación de recursos de los datos que indica el URI. Por ejemplo, un cliente puede realizar una solicitud GET al identificador <https://api.contoso.com/orders/1> URI para recibir el siguiente cuerpo JSON:

JSON

```
{"orderId":1,"orderValue":99.9,"productId":1,"quantity":1}
```

- La **interfaz uniforme** es cómo las API RESTful logran un acoplamiento flexible entre las implementaciones de cliente y servicio. En el caso de las API REST que se basan en HTTP, la interfaz uniforme incluye el uso de verbos HTTP estándar para realizar operaciones como GET, POST, PUT, PATCH y DELETE en recursos.
- **Modelo de solicitud sin estado:** Las API DE RESTful usan un modelo de solicitud sin estado, lo que significa que las solicitudes HTTP son independientes y pueden producirse en cualquier orden. Por este motivo, mantener la información de estado transitorio entre las solicitudes no es factible. El único lugar donde se almacena la información es en los propios recursos y cada solicitud debe ser una operación atómica. Un modelo de solicitud sin estado admite una alta escalabilidad, ya que no es necesario conservar ninguna afinidad entre clientes y servidores específicos. Sin embargo, el modelo sin estado también puede limitar la escalabilidad debido a los desafíos de la escalabilidad del almacenamiento back-end del servicio web. Para obtener más información sobre las estrategias para escalar horizontalmente un almacén de datos, consulte [Creación de particiones de datos](#).
- **Vínculos de Hipermédia:** Las API REST se pueden controlar mediante vínculos de hipermédia contenidos en cada representación de recursos. Por ejemplo, el siguiente bloque de código muestra una representación JSON de un pedido. Contiene vínculos para obtener o actualizar el cliente asociado con el pedido.

JSON

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
```

```
"orderValue":16.60,  
"links": [  
    {"rel":"product","href":"https://api.contoso.com/customers/3"},  
    {"action":"GET" },  
    {"rel":"product","href":"https://api.contoso.com/customers/3"},  
    {"action":"PUT" }  
]  
}
```

## Definición de URI de recursos de API web RESTful

Una API web RESTful se organiza en torno a los recursos. Para organizar el diseño de la API en torno a los recursos, defina los URI de recursos que se asignan a las entidades empresariales. Cuando sea posible, los URI de recursos deben basarse en sustantivos (el recurso) y no en verbos (las operaciones del recurso).

Por ejemplo, en un sistema de comercio electrónico, las entidades empresariales principales pueden ser clientes y pedidos. Para crear un pedido, un cliente envía la información del pedido en una solicitud HTTP POST al URI del recurso. La respuesta HTTP a la solicitud indica si la creación del pedido es correcta.

El URI para crear el recurso de pedido podría ser algo parecido a:

HTTP

<https://api.contoso.com/orders> // Good

Evite usar verbos en URI para representar operaciones. Por ejemplo, no se recomienda el siguiente URI:

HTTP

<https://api.contoso.com/create-order> // Avoid

A menudo, las entidades se agrupan en colecciones como clientes o pedidos. Una colección es un recurso independiente de los elementos de la colección, por lo que debe tener su propio URI. Por ejemplo, el siguiente URI podría representar la colección de pedidos:

```
https://api.contoso.com/orders
```

Una vez que el cliente recupera la colección, puede realizar una solicitud GET al URI de cada elemento. Por ejemplo, para recibir información sobre un pedido específico, el cliente realiza una solicitud HTTP GET en el URI `https://api.contoso.com/orders/1` para recibir el siguiente cuerpo JSON como una representación de recursos de los datos de pedidos internos:

```
JSON
```

```
{"orderId":1,"orderValue":99.9,"productId":1,"quantity":1}
```

## Convenciones de nomenclatura de URI de recursos

Al diseñar una API web RESTful, es importante usar las convenciones de nomenclatura y relación correctas para los recursos:

- **Utiliza sustantivos para los nombres de recursos.** Use nombres para representar recursos. Por ejemplo, usa `/orders` en lugar de `/create-order`. Los métodos HTTP GET, POST, PUT, PATCH y DELETE ya implican la acción verbal.
- **Usa nombres plurales para las URIs de colecciones.** En general, resulta útil usar nombres plurales para los URI que hagan referencia a colecciones. Es recomendable organizar los URI de colecciones y elementos en una jerarquía. Por ejemplo, `/customers` es la ruta de acceso a la colección del cliente y `/customers/5` es la ruta de acceso al cliente con un identificador que es igual a 5. Este enfoque ayuda a mantener la API web intuitiva. Además, muchos marcos de API web pueden enrutar solicitudes basadas en URIs con parámetros, por lo que puede definir una para el camino `/customers/{id}`.
- **Tenga en cuenta las relaciones entre diferentes tipos de recursos y cómo podría exponer estas asociaciones.** Por ejemplo, el `/customers/5/orders` podría representar todos los pedidos para el cliente 5. También puede abordar la relación en la otra dirección representando la asociación desde un pedido hacia un cliente. En este escenario, el URI podría ser `/orders/99/customer`. Sin embargo, este modelo llevado demasiado lejos puede ser difícil de implementar. Un mejor enfoque es incluir vínculos en el cuerpo del mensaje de respuesta HTTP para que los clientes puedan acceder fácilmente a los recursos relacionados. [Use Hypertext como motor de estado de aplicación \(HATEOAS\) para habilitar la navegación a recursos relacionados](#) describe este mecanismo con más detalle.

- **Mantenga las relaciones sencillas y flexibles.** En sistemas más complejos, puede estar inclinado a proporcionar URI que permitan al cliente navegar por varios niveles de relaciones, como `/customers/1/orders/99/products`. Sin embargo, este nivel de complejidad puede ser difícil de mantener y es inflexible si las relaciones entre los recursos cambian en el futuro. En su lugar, intente que los identificadores URI sean relativamente sencillos. Después de que una aplicación tenga una referencia a un recurso, debería poder usar esta referencia para buscar elementos relacionados con ese recurso. Puede reemplazar la consulta anterior por el URI `/customers/1/orders` para buscar todos los pedidos del cliente 1 y, a continuación, usar `/orders/99/products` para buscar los productos en este pedido.

 **Sugerencia**

Evite requerir URLs de recursos que sean más complejos que *colección/elemento/colección*.

- **Evite demasiados recursos pequeños.** Todas las solicitudes web imponen una carga en el servidor web. Cuantas más solicitudes, más grande la carga. Las API web que exponen un gran número de recursos pequeños se conocen como *API web chatty*. Intente evitar estas API porque requieren una aplicación cliente para enviar varias solicitudes para encontrar todos los datos que requiere. En su lugar, considere la posibilidad de desnormalizar los datos y combinar información relacionada en recursos más grandes que se pueden recuperar a través de una sola solicitud. Sin embargo, sigue siendo necesario equilibrar este enfoque con la sobrecarga de capturar datos que el cliente no necesita. La recuperación de objetos de gran tamaño puede aumentar la latencia de una solicitud y incurrir en más costos de ancho de banda. Para más información sobre estos antipatrones de rendimiento, consulte [Antipatrón Chatty I/O](#) y [Extraneous Fetching](#).
- **Evite crear API que reflejen la estructura interna de una base de datos.** El propósito de REST es modelar entidades empresariales y las operaciones que una aplicación puede realizar en esas entidades. Un cliente no debe exponerse a la implementación interna. Por ejemplo, si los datos están almacenados en una base de datos relacional, la API web no necesita exponer cada una de las tablas como una colección de recursos. Este enfoque aumenta la superficie expuesta a ataques y podría dar lugar a pérdidas de datos. En su lugar, considere la API web como una abstracción de la base de datos. Si es necesario, introduzca una capa de asignación entre la base de datos y la API web. Esta capa garantiza que las aplicaciones cliente estén aisladas de los cambios en el esquema de base de datos subyacente.

### Sugerencia

Es posible que no sea posible asignar todas las operaciones implementadas por una API web a un recurso específico. Puede controlar estos escenarios que *no son de recursos* a través de solicitudes HTTP que invocan una función y devuelven los resultados como un mensaje de respuesta HTTP.

Por ejemplo, una API web que implementa operaciones de calculadora sencillas, como agregar y restar, puede proporcionar URI que expongan estas operaciones como pseudoreCURSOS y usen la cadena de consulta para especificar los parámetros necesarios. Una solicitud GET al URI `/add?operando1=99&operando2=1` devuelve un mensaje de respuesta con el cuerpo que contiene el valor 100.

Sin embargo, debe usar estas formas de URI con moderación.

## Definición de métodos de API web RESTful

Los métodos de API web RESTful se alinean con los métodos de solicitud y los tipos de medios definidos por el protocolo HTTP. En esta sección se describen los métodos de solicitud más comunes y los tipos de medios usados en las API web RESTful.

### Métodos de solicitud HTTP

El protocolo HTTP define muchos métodos de solicitud que indican la acción que desea realizar en un recurso. Los métodos más comunes que se usan en las API web RESTful son [GET](#), [POST](#), [PUT](#), [PATCH](#) y [DELETE](#). Cada método corresponde a una operación específica. Al diseñar una API web RESTful, use estos métodos de forma coherente con la definición del protocolo, el recurso al que se accede y la acción que se está realizando.

Es importante recordar que el efecto de un método de solicitud específico debe depender de si el recurso es una colección o un elemento individual. En la tabla siguiente se incluyen algunas convenciones que usan la mayoría de las implementaciones de RESTful.

### Importante

En la tabla siguiente se usa una entidad de comercio `customer` electrónico de ejemplo. Una API web no necesita implementar todos los métodos de solicitud. Los métodos que implementa dependen del escenario específico.

 Expandir tabla

Recurso	PUBLICACIÓN	OBTENER	PONER	ELIMINAR
/clientes	Crear un nuevo cliente	Recuperar todos los clientes	Actualización masiva de clientes	Eliminar todos los clientes
/customers/1	Error	Recuperar los detalles del cliente 1	Actualizar los detalles del cliente 1 si existe	Eliminar al cliente 1
/clientes/1/pedidos	Crear un nuevo pedido para el cliente 1	Recuperar todos los pedidos del cliente 1	Actualización masiva de pedidos del cliente 1	Eliminar todos los pedidos del cliente 1

## Solicitudes de tipo GET

Una solicitud GET recupera una representación del recurso en el URI especificado. El cuerpo del mensaje de respuesta contiene los detalles del recurso solicitado.

Una solicitud GET debe devolver uno de los siguientes códigos de estado HTTP:

 Expandir tabla

Código de estado	Motivo
<b>HTTP</b>	
200(OK)	El método ha devuelto correctamente el recurso.
204 (Sin Contenido)	El cuerpo de la respuesta no contiene ningún contenido, como cuando una solicitud de búsqueda no devuelve ninguna coincidencia en la respuesta HTTP.
404 (No encontrado)	No se encuentra el recurso solicitado.

## Solicitudes de POST

Una solicitud POST debe crear un recurso. El servidor asigna un URI para el nuevo recurso y devuelve ese URI al cliente.

### Importante

En el caso de las solicitudes POST, un cliente no debe intentar crear su propio URI. El cliente debe enviar la solicitud al URI de la colección y el servidor debe asignar un URI al nuevo recurso. Si un cliente intenta crear su propio URI y emite una solicitud POST a un URI específico, el servidor devuelve el código de estado HTTP 400 (SOLICITUD INCORRECTA) para indicar que el método no se admite.

En un modelo RESTful, las solicitudes POST se usan para agregar un nuevo recurso a la colección que identifica el URI. Sin embargo, una solicitud POST también se puede usar para enviar datos para su procesamiento a un recurso existente, sin la creación de ningún recurso nuevo.

Una solicitud POST debe devolver uno de los siguientes códigos de estado HTTP:

 Expandir tabla

Código de estado HTTP	Motivo
200(OK)	El método ha realizado algún procesamiento, pero no crea un nuevo recurso. El resultado de la operación puede incluirse en el cuerpo de la respuesta.
201 (Creado)	El recurso se creó con éxito. El URI del nuevo recurso se incluye en el encabezado Location de la respuesta. El contenido de la respuesta incluye una representación del recurso.
204 (Sin Contenido)	El cuerpo de la respuesta no contiene contenido.
400 (Solicitud incorrecta)	El cliente ha colocado datos no válidos en la solicitud. El cuerpo de la respuesta puede contener más información sobre el error o un vínculo a un URI que proporciona más detalles.
405 (método no permitido)	El cliente ha intentado realizar una solicitud POST en un URI que no admite solicitudes POST.

## Solicitud PUT

Una solicitud PUT debe actualizar un recurso existente si existe o, en algunos casos, crear un nuevo recurso si no existe. Para realizar una solicitud PUT:

1. El cliente especifica el URI del recurso e incluye un cuerpo de solicitud que contiene una representación completa del recurso.
2. El cliente realiza la solicitud.
3. Si ya existe un recurso que tiene este URI, se reemplaza. De lo contrario, se crea un nuevo recurso si la ruta la admite.

Los métodos PUT se aplican a recursos que son elementos individuales, como un cliente específico, en lugar de colecciones. Un servidor puede admitir actualizaciones, pero no la creación mediante PUT. Si se admite la creación a través de PUT depende de si el cliente puede asignar de forma significativa y confiable un URI a un recurso antes de que exista. Si no es posible, use POST para crear recursos y hacer que el servidor asigne el URI. A continuación, use PUT o PATCH para actualizar el URI.

### Importante

Las solicitudes PUT deben ser *idempotentes*, lo que significa que enviar la misma solicitud varias veces siempre da como resultado que el mismo recurso se modifique con los mismos valores. Si un cliente vuelve a enviar una solicitud PUT, los resultados deben permanecer sin cambios. Por el contrario, no se garantiza que las solicitudes POST y PATCH sean idempotentes.

Una solicitud PUT debe devolver uno de los siguientes códigos de estado HTTP:

 Expandir tabla

Código de estado	Motivo
<b>HTTP</b>	
200(OK)	El recurso se actualizó correctamente.
201 (Creado)	El recurso se creó con éxito. El cuerpo de la respuesta puede incluir una representación del recurso.
204 (Sin Contenido)	El recurso se actualizó correctamente, pero el cuerpo de la respuesta no contiene ningún contenido.
409 (Conflicto)	No se pudo completar la solicitud debido a un conflicto con el estado actual del recurso.

## Sugerencia

Considere 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. El cuerpo de la solicitud debe especificar los detalles de los recursos que se van a modificar. Este enfoque puede ayudar a reducir la charlatanería y mejorar el rendimiento.

## Solicitudes PATCH

Una solicitud PATCH realiza una actualización parcial a un recurso existente. El cliente especifica el URI del recurso. El cuerpo de la solicitud especifica un conjunto de cambios que se aplicarán al recurso. Este método puede ser más eficaz que usar solicitudes PUT porque el cliente solo envía los cambios y no toda la representación del recurso. PATCH también puede crear un nuevo recurso especificando un conjunto de actualizaciones en un recurso vacío o *nulo* si el servidor admite esta acción.

Con una solicitud PATCH, el cliente envía un conjunto de actualizaciones a un recurso existente en forma de documento de revisión. El servidor procesa el documento de revisión para realizar la actualización. El documento de revisión especifica solo un conjunto de cambios que se aplicarán en lugar de describir todo el recurso. La especificación del método PATCH, [RFC 5789](#), no define un formato específico para los documentos de revisión. El formato debe deducirse del tipo de medio de la solicitud.

JSON es uno de los formatos de datos más comunes para las API web. Los dos formatos principales de parches basados en JSON son el parche JSON y el parche de combinación JSON.

El parche de combinación de JSON es más sencillo que el parche JSON. El documento de revisión tiene la misma estructura que el recurso JSON original, pero solo incluye el subconjunto de campos que se deben cambiar o agregar. Además, se puede eliminar un campo especificando `null` como valor del campo en el documento de revisión. Esta especificación significa que el parche de fusión no es adecuado si el recurso original puede tener valores nulos explícitos.

Por ejemplo, suponga que el recurso original tiene la representación JSON siguiente:

JSON

```
{  
  "name": "gizmo",
```

```
    "category": "widgets",
    "color": "blue",
    "price": 10
}
```

Este es un posible parche de fusión JSON para este recurso:

JSON

```
{
  "price": 12,
  "color": null,
  "size": "small"
}
```

Esta revisión de mezcla indica al servidor que actualice `price`, elimine `color` y agregue `size`. Los valores de `name` y `category` no se modifican. Para obtener más información sobre la revisión de combinación JSON, consulte [RFC 7396](#) . El tipo de medio para la revisión de combinación JSON es `application/merge-patch+json`.

El parche de fusión no es adecuado si el recurso original puede contener valores nulos explícitos debido al significado especial de `null` en el documento de parche. El documento de revisión tampoco especifica el orden en que el servidor debe aplicar las actualizaciones. Si este orden es importante depende de los datos y del dominio. La revisión JSON, definida en [RFC 6902](#) , es más flexible porque especifica los cambios como una secuencia de operaciones que se van a aplicar, como agregar, quitar, reemplazar, copiar y probar para validar los valores. El tipo de medio para la revisión de JSON es `application/json-patch+json` .

Una solicitud PATCH debe devolver uno de los siguientes códigos de estado HTTP:

 Expandir tabla

Código de estado HTTP	Motivo
200(OK)	El recurso se actualizó correctamente.
400 (Solicitud incorrecta)	El documento de revisión tiene un formato incorrecto.
409 (Conflicto)	El documento de revisión es válido, pero los cambios no se pueden aplicar al recurso en su estado actual.
415 (Tipo de medio no compatible)	No se admite el formato de documento de revisión.

## Solicitudes DELETE

Una solicitud DELETE quita el recurso en el URI especificado. Una solicitud DELETE debe devolver uno de los siguientes códigos de estado HTTP:

 Expandir tabla

Código de estado	Motivo
<b>HTTP</b>	
204 (SIN CONTENIDO)	El recurso fue eliminado con éxito. El proceso se ha controlado correctamente y el cuerpo de la respuesta no contiene más información.
404 (NO ENCONTRADO)	El recurso no existe.

## Tipos MIME de recursos

La representación de recursos es cómo se codifica y transporta un recurso identificado por URI a través del protocolo HTTP en un formato específico, como XML o JSON. Los clientes que quieran recuperar un recurso específico deben usar el URI de la solicitud a la API. La API responde devolviendo una representación de recursos de los datos indicados por el URI.

En el protocolo HTTP, los formatos de representación de recursos se especifican mediante tipos multimedia, también denominados tipos MIME. Para los datos nbinarios, la mayoría de las API web admiten JSON (tipo de medio = `application/json`) y posiblemente XML (tipo de medio = `application/xml`).

El encabezado `Content-Type` de una solicitud o respuesta especifica el formato de representación de recursos. En el ejemplo siguiente se muestra una solicitud POST que incluye datos JSON:

HTTP
<pre>POST https://api.contoso.com/orders Content-Type: application/json; charset=utf-8 Content-Length: 57  {"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}</pre>

Si el servidor no admite el tipo de medio, debe devolver el código de estado HTTP 415 (tipo de medio no compatible).

Una solicitud de cliente puede incluir un encabezado Accept que contiene una lista de tipos multimedia que el cliente acepta del servidor en el mensaje de respuesta. Por ejemplo:

HTTP

```
GET https://api.contoso.com/orders/2
Accept: application/json, application/xml
```

Si el servidor no puede coincidir con ninguno de los tipos de medios enumerados, debe devolver el código de estado HTTP 406 (no aceptable).

## Implementación de métodos asincrónicos

A veces, un método POST, PUT, PATCH o DELETE puede requerir el procesamiento que tarda tiempo en completarse. Si espera la finalización antes de enviar una respuesta al cliente, puede provocar una latencia inaceptable. En este escenario, considere la posibilidad de realizar el método asincrónico. Un método asincrónico debe devolver el código de estado HTTP 202 (aceptado) para indicar que la solicitud se aceptó para su procesamiento, pero está incompleta.

Exponga un punto de conexión que devuelva el estado de una solicitud asincrónica para que el cliente pueda supervisar el estado sondeando el punto de conexión de estado. Incluya el URI del punto de conexión de estado en el encabezado Location de la respuesta 202. Por ejemplo:

HTTP

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

Si el cliente envía una solicitud GET a este punto de conexión, la respuesta debe contener el estado actual de la solicitud. Opcionalmente, puede incluir un tiempo estimado para finalizar o un vínculo para cancelar la operación.

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status": "In progress",
    "link": { "rel": "cancel", "method": "delete", "href": "/api/status/12345" }
}
```

Si la operación asincrónica crea un nuevo recurso, el punto de conexión de estado debe devolver el código de estado 303 (ver Otros) una vez completada la operación. En la respuesta 303, se incluye un encabezado Location que proporciona el URI del nuevo recurso:

HTTP

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

Para obtener más información, consulte [Proporcionar compatibilidad asincrónica con solicitudes de ejecución prolongada y patrón de Request-Reply asincrónico](#).

## Implementación de la paginación y el filtrado de datos

Para optimizar la recuperación de datos y reducir el tamaño de la carga, implemente la paginación de datos y el filtrado basado en consultas en el diseño de la API. Estas técnicas permiten a los clientes solicitar solo el subconjunto de datos que necesitan, lo que puede mejorar el rendimiento y reducir el uso del ancho de banda.

- La **paginación** divide grandes conjuntos de datos en fragmentos más pequeños y administrables. Use parámetros de consulta como `limit` para especificar el número de elementos que se van a devolver y `offset` para especificar el punto inicial. Asegúrese de proporcionar también valores predeterminados significativos para `limit` y `offset`, como `limit=25` y `offset=0`. Por ejemplo:

HTTP

```
GET /orders?limit=25&offset=50
```

- `limit`: especifica el número máximo de elementos que se van a devolver.

### Sugerencia

Para ayudar a evitar ataques por denegación de servicio, considere la posibilidad de imponer un límite superior en el número de elementos devueltos. Por ejemplo, si el servicio establece `max-limit=25` y un cliente solicita `limit=1000`, el servicio puede

devolver 25 elementos o un error de BAD-REQUEST HTTP, en función de la documentación de la API.

- **offset**: especifica el índice inicial de los datos.
- El **filtrado** permite a los clientes refinar el conjunto de datos aplicando condiciones. La API puede permitir que el cliente pase el filtro en la cadena de consulta del URI:

HTTP

```
GET /orders?minCost=100&status=shipped
```

- **minCost**: filtra los pedidos que tienen un costo mínimo de 100.
- **status**: filtra los pedidos que tienen un estado específico.

Puede usar los siguientes procedimientos recomendados:

- La **ordenación** permite a los clientes ordenar los datos mediante un `sort` parámetro como `sort=price`.

#### Importante

El enfoque de ordenación puede tener un efecto negativo en el almacenamiento en caché porque los parámetros de cadena de consulta forman parte del identificador de recursos que muchas implementaciones de caché usan como clave para almacenar en caché los datos.

- La **selección de campos para proyecciones definidas por el cliente** permite a los clientes especificar solo los campos que necesitan mediante un `fields` parámetro como `fields=id,name`. Por ejemplo, puede usar un parámetro de cadena de consulta que acepte una lista delimitada por comas de campos, como `/orders?fields=ProductID,Quantity`.

La API debe validar los campos solicitados para asegurarse de que el cliente tiene permiso para acceder a ellos y no expondrá los campos que normalmente no están disponibles a través de la API.

## Soporte para respuestas parciales

Algunos recursos contienen campos binarios grandes, como archivos o imágenes. Para superar los problemas causados por conexiones poco confiables e intermitentes y mejorar los tiempos de respuesta, considere la posibilidad de admitir la recuperación parcial de recursos binarios grandes.

Para admitir respuestas parciales, la API web debe admitir el encabezado Accept-Ranges para las solicitudes GET para recursos grandes. Este encabezado indica que la operación GET admite solicitudes parciales. La aplicación cliente puede enviar solicitudes GET que devuelven un subconjunto de un recurso, especificado como un intervalo de bytes.

Además, considere la posibilidad de implementar solicitudes HTTP HEAD para estos recursos. Una solicitud HEAD es similar a una solicitud GET, excepto que solo devuelve los encabezados HTTP que describen el recurso, con un cuerpo de mensaje vacío. Una aplicación cliente puede emitir una solicitud HEAD para determinar si se debe capturar un recurso mediante solicitudes GET parciales. Por ejemplo:

```
HTTP
```

```
HEAD https://api.contoso.com/products/10?fields=productImage
```

Este es un mensaje de respuesta de ejemplo:

```
HTTP
```

```
HTTP/1.1 200 OK
```

```
Accept-Ranges: bytes  
Content-Type: image/jpeg  
Content-Length: 4580
```

El encabezado Content-Length proporciona el tamaño total del recurso, y el encabezado Accept-Ranges indica que la operación GET correspondiente admite resultados parciales. La aplicación cliente puede usar esta información para recuperar la imagen en fragmentos más pequeños. La primera solicitud captura los primeros 2500 bytes mediante el encabezado Range:

```
HTTP
```

```
GET https://api.contoso.com/products/10?fields=productImage  
Range: bytes=0-2499
```

El mensaje de respuesta indica que esta respuesta es parcial devolviendo el código de estado HTTP 206. El encabezado Content-Length especifica el número real de bytes devueltos en el cuerpo del mensaje y no el tamaño del recurso. El encabezado Content-Range indica qué parte del recurso se devuelve (bytes de 0 a 2499 de 4580):

```
HTTP
```

```
HTTP/1.1 206 Partial Content
```

```
Accept-Ranges: bytes  
Content-Type: image/jpeg  
Content-Length: 2500  
Content-Range: bytes 0-2499/4580
```

```
[...]
```

Una solicitud posterior desde la aplicación cliente puede recuperar el resto del recurso.

## Implementación de HATEOAS

Una de las principales razones para usar REST es la capacidad de navegar por todo el conjunto de recursos sin tener conocimiento previo del esquema de URI. Cada solicitud HTTP GET debe devolver la información necesaria para encontrar los recursos relacionados directamente con el objeto solicitado a través de hipervínculos incluidos en la respuesta. La solicitud también debe incluir 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 eficazmente una máquina de estado finito y la respuesta a cada solicitud contiene la información necesaria para pasar de un estado a otro. No debe ser necesaria ninguna otra información.

### ⚠ Nota

No hay estándares de uso general que definan cómo modelar el principio HATEOAS. Los ejemplos de esta sección muestran una posible solución propietaria.

Por ejemplo, para controlar la relación entre un pedido y un cliente, la representación de un pedido podría incluir vínculos que identifiquen las operaciones disponibles para el cliente del pedido. El siguiente bloque de código es una posible representación:

```
JSON
```

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links": [
    {
      "rel": "customer",
      "href": "https://api.contoso.com/customers/3",
      "action": "GET",
      "types": ["text/xml", "application/json"]
    },
    {
      "rel": "customer",
      "href": "https://api.contoso.com/customers/3",
      "action": "PUT",
      "types": ["application/x-www-form-urlencoded"]
    },
    {
      "rel": "customer",
      "href": "https://api.contoso.com/customers/3",
      "action": "DELETE",
      "types": []
    },
    {
      "rel": "self",
      "href": "https://api.contoso.com/orders/3",
      "action": "GET",
      "types": ["text/xml", "application/json"]
    },
    {
      "rel": "self",
      "href": "https://api.contoso.com/orders/3",
      "action": "PUT",
      "types": ["application/x-www-form-urlencoded"]
    },
    {
      "rel": "self",
      "href": "https://api.contoso.com/orders/3",
      "action": "DELETE",
      "types": []
    }
  ]
}
```

En este ejemplo, la matriz `links` tiene un conjunto de vínculos. Cada vínculo representa una operación en una entidad relacionada. Los datos de cada vínculo incluyen la relación ("customer"), el URI (<https://api.contoso.com/customers/3>), el método HTTP y los tipos MIME admitidos. La aplicación cliente necesita esta información para invocar la operación.

La `links` matriz también incluye información de referencia automática sobre el recurso recuperado. Estos vínculos tienen la relación `self`.

El conjunto de vínculos que se devuelven puede cambiar en función del estado del recurso. La idea de que hipertexto es el *motor del estado de la aplicación* describe este escenario.

## Implementación del control de versiones

Una API web no permanece estática. A medida que cambian los requisitos empresariales, se agregan nuevas colecciones de recursos. A medida que se agregan nuevos recursos, es posible que cambien las relaciones entre los recursos y que se modifique la estructura de los datos de los recursos. La actualización de una API web para controlar los requisitos nuevos o diferentes es un proceso sencillo, pero debe tener en cuenta los efectos que estos cambios tienen en las aplicaciones cliente que consumen la API web. El desarrollador que diseña e implementa una API web tiene control total sobre esa API, pero no tienen el mismo grado de control sobre las aplicaciones cliente creadas por las organizaciones asociadas. Es importante seguir admitiendo las aplicaciones cliente existentes al tiempo que permite que las nuevas aplicaciones cliente usen nuevas características y recursos.

Una API web que implementa control de versiones puede indicar las características y los recursos que expone, y una aplicación cliente puede enviar solicitudes dirigidas a una versión específica de una característica o recurso. En las secciones siguientes se describen varios enfoques diferentes, cada uno de los cuales tiene sus propias ventajas y desventajas.

### Sin control de versiones

Este enfoque es el más sencillo y puede funcionar para algunas API internas. Los cambios significativos se pueden representar como nuevos recursos o vínculos nuevos. Es posible que agregar contenido a los recursos existentes no presente un cambio importante porque las aplicaciones cliente que no esperan ver este contenido lo omiten.

Por ejemplo, una solicitud al URI `https://api.contoso.com/customers/3` debe devolver los detalles de un solo cliente que contiene los campos `id`, `name`, y `address` que la aplicación cliente espera.

#### HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
{"id":3,"name":"Fabrikam, Inc.","address":"1 Microsoft Way Redmond WA 98053"}
```

### ⓘ Nota

Por motivos de simplicidad, las respuestas de ejemplo que se muestran en esta sección no incluyen vínculos HATEOAS.

Si el `DateCreated` campo se agrega al esquema del recurso de cliente, la respuesta será similar a la siguiente:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","dateCreated":"2025-03-
22T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA 98053"}
```

Las aplicaciones cliente existentes pueden seguir funcionando correctamente si pueden omitir campos no reconocidos. Mientras tanto, las nuevas aplicaciones cliente se pueden diseñar para controlar este nuevo campo. Sin embargo, podrían producirse modificaciones más drásticas en el esquema de recursos, incluidas las eliminaciones de campos o el cambio de nombre. O bien, las relaciones entre los recursos pueden cambiar. Estas actualizaciones pueden constituir cambios importantes que impiden que las aplicaciones cliente existentes funcionen correctamente. En estos escenarios, considere uno de los enfoques siguientes:

- [Control de versiones de URI](#)
- [Control de versiones de cadenas de consulta](#)
- [Versionado de encabezados](#)
- [Control de versiones del tipo de medio](#)

## Control de versiones de 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 existentes anteriormente deben seguir funcionando normalmente devolviendo recursos que se ajustan a su esquema original.

Por ejemplo, el `address` campo del ejemplo anterior se reestructura en subcampos que contienen cada parte constituyente de la dirección, como `streetAddress`, `city`, `state` y `zipCode`. Esta versión del recurso se puede exponer a través de un URI que contiene un número de versión, como <https://api.contoso.com/v2/customers/3>:

HTTP

HTTP/1.1 200 OK

`Content-Type`: application/json; charset=utf-8

```
{"id":3,"name":"Fabrikam, Inc.", "dateCreated":"2025-03-22T12:11:38.0376089Z", "address":{"streetAddress":"1 Microsoft Way", "city":"Redmond", "state":"WA", "zipCode":98053}}
```

Este mecanismo de control de versiones es sencillo, pero depende del servidor para enrutar la solicitud al punto de conexión adecuado. Sin embargo, puede volverse inconfundible a medida que la API web madura a través de varias iteraciones y el servidor tiene que admitir muchas versiones diferentes. Desde el punto de vista de un purista, en todos los casos, las aplicaciones cliente capturan los mismos datos (cliente 3), por lo que el URI no debe diferir según la versión. Este esquema también complica la implementación de HATEOAS porque todos los vínculos deben incluir el número de versión en sus URI.

## Control de versiones de cadena de consulta

En lugar de proporcionar varios URI, puede especificar la versión del recurso mediante un parámetro dentro de la cadena de consulta anexada a la solicitud HTTP, como

<https://api.contoso.com/customers/3?version=2>. El parámetro `version` debe tener como valor predeterminado un valor significativo, como 1, si las aplicaciones cliente anteriores lo omiten.

Este enfoque tiene la ventaja semántica de que el mismo recurso siempre se recupera del mismo URI. Sin embargo, este método depende del código que controla la solicitud para analizar la cadena de consulta y devolver la respuesta HTTP adecuada. Este enfoque también complica la implementación de HATEOAS de la misma manera que el mecanismo de control de versiones de URI.

### Nota

Algunos exploradores web antiguos y servidores proxy web no almacenan en caché las respuestas de las solicitudes que incluyen una cadena de consulta en el URI. Las respuestas

sin almacenar en caché pueden degradar el rendimiento de las aplicaciones web que usan una API web y se ejecutan desde un explorador web anterior.

## Control de versiones de encabezado

En lugar de anexar el número de versión como parámetro de cadena de consulta, puede implementar un encabezado personalizado que indique la versión del recurso. Este enfoque requiere que la aplicación cliente agregue el encabezado adecuado a las solicitudes. Sin embargo, el código que controla la solicitud de cliente puede usar un valor predeterminado, como la versión 1, si se omite el encabezado de versión.

En los ejemplos siguientes se usa un encabezado personalizado denominado *Custom-Header*. El valor de este encabezado indica la versión de la API web.

Versión 1:

HTTP

```
GET https://api.contoso.com/customers/3
Custom-Header: api-version=1
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.", "address": "1 Microsoft Way Redmond WA 98053"}
```

Versión 2:

HTTP

```
GET https://api.contoso.com/customers/3
Custom-Header: api-version=2
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.", "dateCreated": "2025-03-22T12:11:38.0376089Z", "ad-
```

```
dress":{"streetAddress":"1 Microsoft  
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

De forma similar al [versionado de URI](#) y al [versionado de cadenas de consulta](#), debe incluir el encabezado personalizado adecuado en cualquier enlace para implementar HATEOAS.

## Control de versiones del tipo de medio

Cuando una aplicación cliente envía una solicitud HTTP GET a un servidor web, debe usar un encabezado Accept para especificar el formato del contenido que puede controlar. Normalmente, el propósito del encabezado Accept es permitir que la aplicación cliente especifique si el cuerpo de la respuesta debe ser XML, JSON o algún otro formato común que el cliente pueda analizar. Sin embargo, es posible definir tipos multimedia personalizados que incluyan información que permita a la aplicación cliente indicar qué versión de un recurso espera.

En el ejemplo siguiente se muestra una solicitud que especifica un encabezado Accept con el valor `application/vnd.contoso.v1+json`. El `vnd.contoso.v1` elemento indica al servidor web que debe devolver la versión 1 del recurso. El `json` elemento especifica que el formato del cuerpo de la respuesta debe ser JSON:

HTTP

```
GET https://api.contoso.com/customers/3  
Accept: application/vnd.contoso.v1+json
```

El código que controla la solicitud es responsable de procesar el encabezado Accept y respetarlo tanto como sea posible. La aplicación cliente puede especificar varios formatos en el encabezado Accept, lo que permite al servidor web elegir el formato más adecuado para el cuerpo de la respuesta. El servidor web confirma el formato de los datos en el cuerpo de respuesta mediante el encabezado Content-Type:

HTTP

```
HTTP/1.1 200 OK  
Content-Type: application/vnd.contoso.v1+json; charset=utf-8  
  
{"id":3,"name":"Fabrikam, Inc.", "address":"1 Microsoft Way Redmond WA 98053"}
```

Si el encabezado Accept no especifica ningún tipo de medio conocido, el servidor web puede generar un mensaje de respuesta HTTP 406 (no aceptable) o devolver un mensaje con un tipo de

medio predeterminado.

Este mecanismo de control de versiones es sencillo y adecuado para HATEOAS, que puede incluir el tipo MIME de datos relacionados en vínculos de recursos.

#### ! Nota

Al seleccionar una estrategia de control de versiones, implicaciones, especialmente en relación con el almacenamiento en caché del servidor web. Los esquemas de versionado de URI y de cadenas de consulta son amigables con el almacenamiento en caché porque la misma combinación de URI o cadena de consulta se refiere a los mismos datos cada vez.

Normalmente, los mecanismos de control de versiones de tipo multimedia y control de versiones de encabezado requieren más lógica para examinar los valores en el encabezado personalizado o en el encabezado Accept. En un entorno a gran escala, muchos clientes que usan versiones diferentes de una API web pueden producir una cantidad significativa de datos duplicados en una caché del servidor. Este problema puede ser agudo si una aplicación cliente se comunica con un servidor web a través de un proxy que implementa el almacenamiento en caché y solo reenvía una solicitud al servidor web si actualmente no contiene una copia de los datos solicitados en su caché.

## APIs web multiinquilino

Una solución de API web *multicliente* es compartida por varios inquilinos, como organizaciones distintas que tienen sus propios grupos de usuarios.

La multitenencia afecta significativamente al diseño de la API web porque dicta cómo se accede y se descubren los recursos en varios inquilinos dentro de una única API web. Diseñe una API teniendo en cuenta la multitenencia para evitar la necesidad de refactorización futura y así poder implementar aislamiento, escalabilidad o personalizaciones específicas del inquilino.

Una API bien diseñada debe definir claramente cómo se identifican los inquilinos en las solicitudes, ya sea a través de subdominios, rutas de acceso, encabezados o tokens. Esta estructura garantiza una experiencia coherente y flexible para todos los usuarios del sistema. Para más información, consulte [Asignación de solicitudes a inquilinos en una solución multiinquilino](#).

El multiinquilino afecta a la estructura del punto de conexión, el control de solicitudes, la autenticación y la autorización. Este enfoque también influye en cómo las puertas de enlace de

API, los equilibradores de carga y los servicios back-end enrutan y procesan las solicitudes. Las estrategias siguientes son formas comunes de lograr el multiarrendamiento en una API de la web.

## Usar aislamiento basado en subdominio o dominio (inquilino de nivel DNS)

Este enfoque enruta las solicitudes mediante [dominios específicos del inquilino](#). Los dominios comodín usan subdominios para ofrecer flexibilidad y simplicidad. Los dominios personalizados, que permiten a los inquilinos usar sus propios dominios, proporcionan un mayor control y se pueden adaptar para satisfacer necesidades específicas. Ambos métodos se basan en la configuración de DNS adecuada, incluidos A y CNAME registros, para dirigir el tráfico a la infraestructura adecuada. Los dominios comodín simplifican la configuración, pero los dominios personalizados proporcionan una experiencia más personalizada.

[Preserve el nombre de host](#) entre el proxy inverso y los servicios de back-end para ayudar a evitar problemas como el redireccionamiento de URLs y prevenir la exposición de URLs internas. Este método garantiza el enrutamiento correcto del tráfico específico del inquilino y ayuda a proteger la infraestructura interna. La resolución de DNS es fundamental para lograr la residencia de datos y garantizar el cumplimiento normativo.

```
HTTP
```

```
GET https://adventureworks.api.contoso.com/orders/3
```

## Pasar encabezados HTTP específicos del inquilino

La información del arrendatario se puede pasar a través de encabezados HTTP personalizados como X-Tenant-ID o X-Organization-ID, a través de encabezados de host como Host o x-Forwarded-Host, o se puede extraer de declaraciones de JSON Web Token (JWT). La elección depende de las funcionalidades de enrutamiento de la puerta de enlace de API o del proxy inverso, con soluciones basadas en encabezado que requieren una puerta de enlace de nivel 7 (L7) para inspeccionar cada solicitud. Este requisito agrega sobrecarga de procesamiento, lo que aumenta los costos de proceso cuando se escala el tráfico. Sin embargo, el aislamiento basado en encabezados proporciona ventajas clave. Permite la autenticación centralizada, lo que simplifica la administración de seguridad en las API multiinquilino. Mediante el uso de SDK o clientes de API, el contexto de inquilino se administra dinámicamente en tiempo de ejecución, lo que reduce la complejidad de la configuración del lado cliente. Además, mantener el contexto de inquilino en

los encabezados da como resultado un diseño de API RESTful más limpio al evitar datos específicos del inquilino en el URI.

Una consideración importante para el enrutamiento basado en encabezados es que complica el almacenamiento en caché, especialmente cuando las capas de caché dependen únicamente de claves basadas en URI y no tienen en cuenta los encabezados. Dado que la mayoría de los mecanismos de almacenamiento en caché optimizan las búsquedas de URI, confiar en encabezados puede provocar entradas de caché fragmentadas. Las entradas fragmentadas reducen los aciertos de caché y aumentan la carga de back-end. De forma más crítica, si una capa de almacenamiento en caché no diferencia las respuestas por encabezados, puede servir datos almacenados en caché destinados a un inquilino a otro y crear un riesgo de pérdida de datos.

HTTP

```
GET https://api.contoso.com/orders/3  
X-Tenant-ID: adventureworks
```

o

HTTP

```
GET https://api.contoso.com/orders/3  
Host: adventureworks
```

o

HTTP

```
GET https://api.contoso.com/orders/3  
Authorization: Bearer <JWT-token including a tenant-id: adventureworks claim>
```

## Pasar información específica del inquilino a través de la ruta del URI

Este enfoque anexa identificadores de arrendatario dentro de la jerarquía de recursos y se basa en la puerta de enlace API o en el proxy inverso para determinar el arrendatario adecuado en función del segmento de ruta. El aislamiento basado en rutas de acceso es eficaz, pero pone en peligro el diseño RESTful de la API web e introduce lógica de enrutamiento más compleja. A menudo, se

requiere la coincidencia de patrones o expresiones regulares para analizar y canonicalizar la ruta de acceso del URI.

En cambio, el aislamiento basado en encabezados transmite información de inquilino a través de encabezados HTTP como pares clave-valor. Ambos enfoques permiten un uso compartido de infraestructura eficaz para reducir los costos operativos y mejorar el rendimiento en api web multiinquilino a gran escala.

```
HTTP
```

```
GET https://api.contoso.com/tenants/adventureworks/orders/3
```

## Habilitación del seguimiento distribuido y el contexto de seguimiento en las API

A medida que los sistemas distribuidos y las arquitecturas de microservicios se convierten en el estándar, [aumenta la complejidad de las arquitecturas modernas](#). El uso de encabezados, como Correlation-ID, X-Request-ID O X-Trace-ID, para propagar el contexto de seguimiento en las solicitudes de API es un procedimiento recomendado para lograr visibilidad de un extremo a otro. Este enfoque permite realizar un seguimiento sin problemas de las solicitudes a medida que fluyen desde el cliente a los servicios back-end. Facilita la identificación rápida de errores, supervisa la latencia y asigna dependencias de API entre servicios.

Las API que admiten la inclusión de información de seguimiento y contexto mejoran sus capacidades de observabilidad y depuración. Al habilitar el seguimiento distribuido, estas API permiten una comprensión más detallada del comportamiento del sistema y facilitan el seguimiento, el diagnóstico y la resolución de problemas en entornos complejos de varios servicios.

```
HTTP
```

```
GET https://api.contoso.com/orders/3
```

```
Correlation-ID: aaaa0000-bb11-2222-33cc-444444dddddd
```

```
HTTP
```

```
HTTP/1.1 200 OK
```

```
...
```

```
Correlation-ID: aaaa0000-bb11-2222-33cc-444444dddddd
```

# Modelo de madurez de API web

En 2008, Leonard Richardson propuso lo que ahora se conoce como modelo de madurez de Machine Learning (RMM) para las API web. El RMM define cuatro niveles de madurez para las API web y se basa en los principios de REST como un enfoque arquitectónico para diseñar servicios web. En RMM, a medida que aumenta el nivel de madurez, la API se vuelve más RESTful y sigue con más detalle los principios de REST.

Los niveles son:

- **Nivel 0:** Defina un URI y todas las operaciones son solicitudes POST a este URI.  
Normalmente, los servicios web de Protocolo simple de acceso a objetos se encuentran en este nivel.
- **Nivel 1:** Cree URI independientes para recursos individuales. Este nivel aún no es RESTful, pero comienza a alinearse con el diseño RESTful.
- **Nivel 2:** Use métodos HTTP para definir operaciones en recursos. En la práctica, muchas API web publicadas se alinean aproximadamente con este nivel.
- **Nivel 3:** Utilice hipermedia ([HATEOAS](#)). Este nivel es realmente una API RESTful, según la definición de Fielding.

# OpenAPI Initiative

[OpenAPI Initiative](#) fue creado por un consorcio del sector para estandarizar las descripciones de la API REST entre proveedores. La especificación de estandarización se llamó a Swagger antes de que se llevara bajo la iniciativa OpenAPI y se cambiara el nombre a la especificación openAPI (OAS).

Es posible que quiera adoptar OpenAPI para las API web RESTful. Considere los siguientes puntos:

- La OEA viene con un conjunto de directrices opinativas para diseño de API REST. Las directrices son ventajosas para la interoperabilidad, pero requieren que el diseño se ajuste a las especificaciones.
- OpenAPI promueve un enfoque de contrato primero en lugar de un enfoque primero de implementación. El contrato primero significa que diseña primero el contrato de API (la interfaz) y, a continuación, escribe código que implementa el contrato.

- Las herramientas como Swagger (OpenAPI) pueden generar bibliotecas cliente o documentación a partir de contratos de API. Para obtener un ejemplo, consulte la documentación de la API web de ASP.NET Core con Swagger/OpenAPI.

## Pasos siguientes

- Consulte [recomendaciones detalladas para diseñar API REST en Azure](#).
- Consulte una [lista de comprobación](#) de los elementos que se deben tener en cuenta al diseñar e implementar una API web.
- Crea [software como servicio \(SaaS\)](#) y arquitecturas de soluciones multicliente en Azure.