CLASE 1

Fundamentos de Web API y HTTP

Introducción

Bienvenidos al primer módulo del curso de web APIs con .Net Core.

Comenzaremos respondiendo a la pregunta ¿Qué es un API?.

Más adelante hablaremos de REST y qué características debe tener web API para considerarse RESTful.

Luego daremos una introducción de HTTP.

Es importante conocer los fundamentos de HTTP ya que como programadores web vamos a necesitar conocer el protocolo que permite a nuestros clientes comunicarse con nuestro web API.

Veremos la anatomía de una petición HTTP, los métodos HTTP y los códigos de estado HTTP. Ya que éste es un curso de Web API comenzaremos respondiendo la pregunta Qué es un API.

¿Qué es una API?

Estamos rodeados de software y aplicaciones. Nuestra vida día a día se relaciona de algún modo con un conjunto de programas de computadora. Estos programas tienen la capacidad de comunicarse entre sí. ¿Cómo pueden hacerlo? Pues porque ponen un API o Interfaz de Programación de Aplicaciones.

Antes de hablar de APIs en software, veamos un ejemplo que hoy existe en tu hogar un enchufe o tomacorriente. Te has fijado que en el mismo enchufe donde puedes conectar una laptop puedes también conectar un microondas o una tostadora, una heladera, un secador de pelo y hasta el cargador de tu celular.

La razón de esto es que el enchufe actúa como una **interfaz** que permite pasar corriente a cualquier aparato que se conecte en éste. Lo poderoso de esto es que cualquier aparato eléctrico puede utilizar el enchufe, por tanto un conjunto de aparatos puede utilizar el mismo recurso para funcionar. En nuestra analogía el enchufe viene siendo el API y los aparatos son los clientes del API.

Regresando al mundo del desarrollo web un API o Web API es un conjunto de interfaces definidas las cuales permiten a un conjunto de aplicaciones externas consumir el software de la aplicación web. En términos más prácticos, Web API expone un conjunto de funciones de una aplicación web los cuales podrán ser consumidos por aplicaciones desktop, aplicaciones móviles e incluso otras

aplicaciones web, aunque Web API puede ser más que un simple conjunto de funciones expuestas.

Una API permite que dos aplicaciones completamente distintas puedan comunicarse entre sí. Una aplicación de Android está en lenguaje Kotlin puede comunicarse con una aplicación web hecha con C#, a través de un Web Api.

Otra importancia de los APIs a nivel general, es que permite realizar abstracciones. La idea de una abstracción es que nos permite facilitar el uso de un software sin necesitar conocer cómo funciona internamente, basta con utilizar las funciones que el Api expone.

Regresando al ejemplo del enchufe no necesitas ser ingeniero eléctrico ni saber cómo funciona la física eléctrica para poder conectar una tostadora al enchufe, basta con que utilices la API para poder consumirlo.

En resumen una Web API la podemos ver como un conjunto de funciones de nuestra aplicación web, las cuales pueden ser consumidas por otras aplicaciones a distancia.

Hay distintas maneras de desarrollar y organizar web APIs; en este curso nos enfocaremos en el estilo Rest del cual hablaremos a continuación.

¿Qué es Rest?

Ya hablamos acerca de las Apis, ahora vamos a hablar acerca de un estilo de cómo construir web APIs.

REST (Representation State Transfer), es un estilo de construir servicios web que se adhieren a un conjunto de principios establecidos, es decir que hay un conjunto de condiciones que un API debe de tener para poder decir que implementa Rest; cuando un Web API respeta esas condiciones se lo llama RESTful.

Normalmente cuando consumimos un API es porque queremos acceder a sus recursos en nuestro contexto. Un recurso hace referencia a cosas o entidades que se pueden consumir de un web API.

Por ejemplo si tenemos un web Api que nos permite trabajar con el sistema de una biblioteca, un recurso que el Web API podría exponer sería el listado de libros de la biblioteca. Otros recursos que la web API podría exponer sería el listado de empleados de la biblioteca.

Una Api Rest, usa los métodos del protocolo HTTP sobre una URL (Uniform Resource Locator) para ejecutar distintas funciones del Web API.

Por ejemplo supongamos que tenemos URL:

http://biblioteca.com/api/User

Y sobre esta hacemos un HTTP GET, entonces obtendremos un listado de los usuarios de la biblioteca, y si hacemos un HTTP POST a esta misma URL, donde

enviamos la información de un usuario, entonces vamos a ejecutar la funcionalidad de crear un usuario en nuestro servidor.

A esto le llamamos HTTP CRUD (Create, Read, Update, Delete), porque podemos crear, leer, actualizar y eliminar información consumiendo el Web API utilizando el protocolo HTTP.

Sin embargo esto no es suficiente para decir que un Web API es RESTful. Más adelante hablaremos acerca de los métodos HTTP, por ahora concentrémonos en las condiciones que hacen que un Web API sea RESTful.

La ventaja de respetar estas condiciones es que en general tenemos beneficios agregados, como un software que puede desarrollarse y responder a cambios de requerimientos del negocio de una manera eficiente. Es importante destacar, que no todos los papeles que hagas tienen que ser RESTful, REST es sólo una guía para desarrollar web APIs.

Como ya dijimos para que un Web Api sea RESTful, debes respetar las 6 condiciones del REST:

- **Arquitectura Cliente-Servidor**: Esta arquitectura nos habla de la separación entre un cliente y un proveedor o servidor. En el caso de los web Apis el servidor es un servidor web, el cliente puede ser cualquier software capaz de comunicarse utilizando el protocolo HTTP con el servidor web, por ejemplo una página web en un navegador o una aplicación en un celular e incluso una aplicación de escritorio.
  Con este principio aseguramos la separación de responsabilidades entre nuestros servicios Web API y los clientes que consumen dicho servicio, así el web API puede evolucionar en el servidor y eso no necesariamente debe de afectar a los clientes de nuestro servicio.
- **Interfaz uniforme**: La idea del interfaz uniforme es tener una forma estandarizada de transmisión de la información. Con esta condición tenemos una manera universal de utilizar web APIs comunes, es decir, si sabes consumir un Web API, se pueden consumir otros APIs sin mucha dificultad.
  Para cumplir con la condición de interfaz uniforme, hay que cumplir con cuatro subcondiciones:
  a. La primera subcondición es la *identificación de recurso*, usamos URLs para identificar recursos, por ejemplo, con esta URL, http://biblioteca.com/api/libro, podríamos acceder al listado de libros del web API suponiendo que el web API expone dicho recurso.
  b. La segunda subcondición es manipulación de recursos usando *representaciones*, por ejemplo, si el cliente tiene una manera para acceder a ese recurso, normalmente una URL, entonces ya con esto puede modificar el recurso, es decir usar métodos HTTP para esto.
  c. La tercera subcondición es tener *mensajes autodescriptivos*, todos los mensajes son completos en el sentido que indican toda la información

necesaria para que sean procesados por el servidor en forma satisfactoria. Cuando hablamos de mensajes en nuestro caso nos referimos a las peticiones HTTP que hacemos al servidor. Un mensaje puede indicar el formato en el que quiere recuperar la información del servidor, utilizando *media types*. Los media types son identificadores de formato que utilizamos para indicar el formato de la información que retornará el web Api, por ejemplo solicitar que la información del web api sea en formato JSON, XML, PDF, CSV, etc. Aclaramos que aunque el cliente tiene el poder de pedir la información en el formato que desea, es responsabilidad del servidor poder satisfacer la petición del cliente, y si la petición del cliente no es razonable, el servidor no tiene por qué satisfacerla. Por ejemplo si el cliente quiere un listado de libros en formato JPG, que es un formato de imagen, no tiene sentido por lo que el web Api no tiene por qué satisfacer dicha demanda. Sin embargo sí es razonable que el cliente nos pueda pedir esa información en formato XML o JSON.

d. La cuarta subcondición es *HATEOAS*. El término HATEOAS, introducido por Fielding en su definición de REST, es el acrónimo de **H**ypermedia **a**s **t**he engine **o**f **a**pplicacion **s**tate (en español, *hipermedia como el motor del estado de la aplicación*) y describe una de las propiedades más significativas de REST: como este enfoque de diseño de aplicaciones ha de ofrecer una **interfaz universal**, lo que postula HATEOAS es que el cliente pueda moverse por la aplicación web únicamente siguiendo a los identificadores únicos URI en formato hipermedia (URI = **Uniform Resource Identifier**), es decir que la información que nos da el web API cuando hacemos una petición, debe incluir links para poder seguir explorando los demás recursos del API. Por ejemplo si pedimos un listado de libros, sería ideal que cada uno de esos libros tenga un link para acceder al detalle de cada uno de los libros.

- **Protocolos sin estado**: Cada una de las peticiones realizadas al web API tienen toda la información necesaria para que la petición sea resuelta de manera satisfactoria. Sí el web API requiere que el cliente esté debidamente identificado para acceder y manipular un recurso entonces el cliente debe de enviarnos algún tipo de información que identifique al cliente que está haciendo la petición cada vez que se haga una petición HTTP al servidor

- **Caché**: Las respuestas del Web API deben indicar cuándo se deben guardar en caché, es decir nos referimos a que el cliente puede guardar el recurso dado por la URL de manera local en su dispositivo, para que en subsiguientes peticiones HTTP dichos recursos no tengan que ser pedidos nuevamente al Web API, sino que se pueda consumir la versión local (tener en cuenta que según el tipo de información de negocio que maneja la aplicación, no siempre es posible o conveniente, por ejemplo, no guardar información sensible como nros de cuenta bancaria, tarjetas de crédito, claves, etc., o si la información no es sensible pero se actualiza constantemente, en caché quedará información

desactualizada). Esto disminuye el tiempo de respuesta que deben esperar los clientes de nuestra aplicación.

- **Sistema de capas**. El servicio del servidor debe tener una arquitectura de capas donde su evolución sea completamente transparente para el cliente. Así por ejemplo si el servicio web va a utilizar *Load Balance* (balanceo de carga: se refiere a la distribución del tráfico de red entrante a través de un grupo de servidores backend, también conocido como *Server Farm* (conjunto de servidor) o *Server Pool* (conjunto de servidores)), los clientes no tienen por qué tener presente ese detalle pues esto debe ser algo completamente transparente para ellos.
- **Código en demanda (opcional):** El servicio web tiene la opción de enviar código fuente el cual se va a ejecutar en el cliente, típicamente este código es JavaScript.

Anatomía de una petición HTTP

Cuando trabajamos con Web Api la comunicación entre los clientes y el Web API, se realiza utilizando peticiones HTTP. Una petición HTTP es un mensaje que una computadora envía a otra utilizando el protocolo HTTP. La petición HTTP las hacen los clientes de la API hacia la API; cuando la API recibe esta petición, la procesa y luego retorna una respuesta llamada respuesta HTTP.

Los clientes y el Web API se van a comunicar utilizando **peticiones** y **respuestas** HTTP (en inglés **Request** y **Response**, y en español otra forma de llamarlas en **Solicitud** y **Respuesta**).

Estos mensajes tienen una estructura, mencionamos a continuación cada una de las 3 partes que forman la petición HTTP:

1. Una línea de petición
2. Un conjunto de campos cabecera
3. Un cuerpo (opcional)

Detallamos a continuación el contenido de cada parte:

- **Línea de petición**: Contiene el *método* HTTP a usar, la *URI* de la petición y el *protocolo* HTTP a usar. Los métodos HTTP básicamente indican qué tipo de acción quiere realizar el cliente, por ejemplo, leer un recurso o si quiere enviar información hacia el API, etc., luego profundizaremos en los métodos HTTP (o verbos HTTP).
  La URI se refiere a la dirección donde se encuentra el recurso y el protocolo HTTP se refiere a qué protocolo HTTP se va a usar, porque existen varias versiones del protocolo HTTP, por ejemplo, HTTP 1.1

(https://www.w3.org/Protocols/) y existen otras revisiones como la revisión HTTP 2.0 (https://http2.github.io/).

Vemos un ejemplo de una línea de petición:

GET /api/autores HTTP/1.1
El método HTTP es GET y significa que queremos leer un recurso o que el cliente del Web Api quiere leer un recurso, *¿qué recurso?*, el que se indica **api/autores**, y finalmente se indica el protocolo HTTP a usar.

Veamos otro ejemplo:

POST /test.html HTTP/1.1

Aquí el método es POST y significa que el usuario quiere enviarnos algo, *¿qué es lo que desea enviar?*, ese algo estará especificado en el cuerpo de la petición. Vemos que la URI y es **test.html** y el protocolo es HTTP 1.1.

- **Cabecera de la petición**: La cabecera de la petición es donde se encuentran las *cabeceras de la petición*, que son metadatos que se envían en la petición para brindar información sobre dicha petición. Cada cabecera se especifica con un nombre, luego dos puntos y seguido por el valor de dicha cabecera.

  Veamos un ejemplo de una cabecera:

  Host: en.wikipedia.org

  En este caso el nombre de la cabecera es **Host** y su valor es **en.wikipedia.org**. La cabecera **Host** indica el dominio del servidor.

  Otro ejemplo de cabecera:

  Cache-Control:no-cache

  El nombre de esta cabecera es **Cache-Control** y su valor es **no-cache**.

  En la cabecera de la petición pueden haber varias cabeceras, por ejemplo estas tres:

```
GET /api/autores HTTP/1.1
Host: en.wikipedia.org
Cache-Control:no-cache
```

En el ejemplo anterior vemos que la primera línea es la línea de petición:

GET /api/autores HTTP/1.1

Debajo se encuentra la cabecera de la petición que se compone de varias cabeceras individuales:

Host: en.wikipedia.org
Cache-Control:no-cache

Estas dos cabeceras individuales son cabeceras estándar que ya tienen un propósito bien definido, sin embargo podemos usar nuestras propias cabeceras personalizadas; cuando necesitamos metadata propia de la petición HTTP, podemos usar cabeceras personalizadas, lo único que se debe hacer es mandarlas en la cabecera de la petición HTTP.

- **Cuerpo de la petición (opcional)**: El cuerpo de la petición es donde colocamos información adicional que vamos a enviar al servidor, y podemos colocar virtualmente lo que queramos, desde el nombre de un usuario y password de un usuario que intenta loguearse a nuestra aplicación, hasta las respuestas de un complejo formulario de una encuesta, o de registro.

  El cuerpo es bastante importante pues representa en muchos casos el contenido que se quiere transmitir. Y destacamos que las peticiones GET no utilizan un cuerpo porque no se tiende a enviar muchos datos al momento de leer información.

  En el caso del método POST sí se suele utilizar el cuerpo de la petición para colocar lo que queremos enviar.

  Veamos un ejemplo de un cuerpo de la petición:

  Hola

  El cuerpo de la petición es para enviar lo que queramos desde un simple saludo hasta información más estructurada como la siguiente:

  {

  “Nombre”: “Juan Perez”,

  “PublicadoEn”: “2019”
  }

  Esto es información en formato **JSON** (**J**ava**S**cript **O**bject **N**otation) que representa el cuerpo de la petición, con los datos nombre y año de

publicación.

Vemos ahora una petición HTTP con sus tres partes bien definidas:

```
POST /api/autores HTTP/1.1
Host: miWebApi.com
Content-Type: application/json
Cache-Control:no-cache

{
        "Nombre": "Juan Perez",
        "PublicadoEn": "2019"
}
```

- **Línea de petición**
- **Cabecera de la petición (compuesta por 3 cabeceras)**
- **Línea en blanco**
- **Cuerpo de la petición**

En la primer línea, **la petición con el método HTTP POST, la dirección y la versión del protocolo HTTP**, a partir de la segunda línea está **la cabecera compuesta de tres cabeceras: Host, Content-Type y Caché-Control**, y finalmente **el cuerpo de la petición**. Hay también **una línea en blanco que es la separación entre la cabecera HTTP y el cuerpo**.

Cuando el cliente nos envía una petición HTTP nuestro servidor debe responder con una respuesta HTTP. La respuesta HTTP también tiene su propia estructura que es bastante similar a la estructura de la petición, estas partes son:

1. Línea de estado
2. Cabecera
3. Cuerpo (opcional).

En la línea de estado se indica el "estado" de la petición, es decir si fue exitosa o si hubo un error o si se requiere que tomemos algún tipo de acción. En esta línea también se coloca un código de estado HTTP que detallaremos luego.

La segunda parte, la cabecera, es un conjunto de cabeceras también igual que la cabecera de la petición. El servidor puede enviarnos tantas cabeceras como requiera y tiene la opción de enviar un cuerpo con los datos que desea transmitirnos.

Aunque el cuerpo también es opcional es bastante fundamental a la hora de utilizar páginas web, ya que es a través del cuerpo que recibimos el HTML de una página web que queremos visualizar en nuestro navegador.

Veamos un ejemplo de una respuesta HTTP:

HTTP/1.1 200 OK **Date: Thu, 03 Jan 2020 12:23:03 GMT** **Server: Saturno** **Accept-Ranges: bytes** **Content-Length: 688** **Content-Type: text/html; charset=UTF-8** **<!doctype html><html …**	**Línea de estado** **Cabecera de la respuesta (compuesta por varias cabeceras)** **Línea en blanco** **Cuerpo de la respuesta**

En la primera línea tenemos **la línea de estado, con el protocolo HTTP utilizado HTTP 1.1 y en luego en este caso un 200 OK** que es el código de estado HTTP. Luego tenemos **un conjunto de cabeceras como fecha, servidor, Content-Length, Content-Type, etc.** Y finalmente tenemos **el cuerpo de la respuesta** también **separado por una línea en blanco de la cabecera**, vemos que inicia con un doctype es decir que es un documento HTML

Métodos HTTP

Ya hablamos acerca de las condiciones que hace que un web sea RESTful. Una de ellas hablaba acerca del manejo/procesamiento que podemos hacer a los recursos una vez que tengamos su URL. Este manejo se indican utilizando métodos HTTP.

Los métodos HTTP también conocidos informalmente como "**verbos HTTP**", son un mecanismo del protocolo HTTP que permite expresar la acción que queremos hacer sobre un recurso.

Por ejemplo, si tenemos un recurso localizado en la URL https://miAPI.com/usuarios, que se relaciona con usuarios de nuestra aplicación, y quisiéramos obtener un listado de usuarios, podemos utilizar un HTTP GET a esa URL, y si quisiéramos agregar un nuevo usuario a través del web API, podemos hacer un HTTP POST a la misma URL.

En síntesis, los métodos HTTP nos permiten "*expresar la acción que queremos realizar sobre un recurso*". Y nosotros como desarrolladores diseñaremos y programaremos la funcionalidad del webapi que queremos exponer hacia los clientes que lo consumirán, a través de los métodos HTTP.

Cuando un cliente utiliza un método HTTP sobre algún recurso, además podemos definir si un determinado verbo HTTP no es permitido para acceder a ese recurso.

Existen varios métodos HTTP entre los cuales podemos destacar:

- **HTTP GET**: se utiliza para solicitar datos del servidor. Cuando colocamos una web en nuestro navegador y presionamos ENTER lo que el navegador hace es

enviar una petición HTTP GET para indicar el servidor que queremos obtener la información de dicho recurso. La representación de este recurso puede ser en un documento HTML, JSON, XML, etc., que literalmente

- **HTTP HEAD**: hace lo mismo que el método GET, pero no recupera el cuerpo de la respuesta, sino solamente la cabecera. Como mencionamos anteriormente, una respuesta HTTP tiene una cabecera y opcionalmente un cuerpo. La cabecera tiene información como, el tipo de contenido, su fecha de última modificación, ubicación, estado entre otros. Es decir que *la cabecera da una información acerca del mensaje*, pero no el mensaje en sí mismo, mientras que el cuerpo de la respuesta es típicamente la información que el usuario solicita, por ejemplo, el documento HTML de la página web que el usuario desea visualizar.
- **HTTP POST**: sirve para indicar que queremos enviar información al servidor y esta información la enviamos a través del cuerpo de la petición HTTP. Con la información que enviamos al servidor, el servidor puede realizar algún tipo de operación y dar una respuesta al cliente. Un uso común que se le da a POST es para agregar información enviada al webapi, en una base de datos.
- **HTTP PUT**: normalmente lo utilizamos casi de la misma forma que POST, con la diferencia de que PUT se usa para actualizar un recurso ya existente en la base de datos, aunque de acuerdo con la especificación HTTP 1.1. si el recurso al cual se le hace un HTTP PUT no existe, el servidor puede crearlo en vez de actualizarlo, aunque en la práctica típicamente se reserva el POST para crear y el PUT para actualizar.
- **HTTP DELETE**: sirve para indicar que queremos eliminar un recurso determinado.

Los métodos HTTP mencionados hasta ahora, los podemos asociar con las operaciones de un CRUD, es decir lectura, inserción, actualización y eliminación de información. A los web APIs que usan estos métodos para realizar esas operaciones sobre una base de datos, informalmente se los llama HTTP CRUD, y formalmente en el ***modelo de madurez de Richardson***, es un web API Nivel 2.

Nota: dejamos aquí un link de buenas fuentes sobre el ***modelo de madurez de Richarson***: https://martinfowler.com/articles/richardsonMaturityModel.html

https://translate.google.com/translate?hl=es-419&sl=en&u=https://martinfowler.com/articles/richardsonMaturityModel.html&prev=search

Finalmente veremos un último método:

- **HTTP PATCH**: se utiliza para realizar actualizaciones parciales a un recurso. Las ventajas de esta operación es que es relativamente rápida comparada con

el método PUT. Sin embargo su implementación requiere un poco más de trabajo pero su implementación es bastante directa y bastante conveniente.

Aclaramos que principalmente en este curso usaremos los métodos GET, POST, PUT y DELETE. En los ejercicios prácticos, generalmente tendremos una clase para procesar las peticiones a un recurso, y esta clase va a tener funciones (métodos en la terminología de **POO** □ **P**rogramación **O**rientada a **O**bjetos), y según el método HTTP utilizado se ejecutará una de las funciones de la clase, es decir que habrá un "mapeo" entre la función de la clase y el método HTTP.

Nota 1: dejamos aquí un link de buenas fuentes con más detalles de métodos HTTP https://developer.mozilla.org/es/docs/Web/HTTP/Methods

Nota 2: cuando hablamos de "métodos HTTP" nos referimos a los informalmente llamados verbos del protocolo HTTP, y cuando hablamos de "métodos de una clase", nos referimos a las funciones de una clase escritas en un lenguaje de programación orientado a objetos como C#, C++, Java, etc.

Códigos de estado

Cuando se hace una petición HTTP a un servidor web recibiremos una respuesta HTTP, y dentro de la información contenida en la respuesta, se encuentra el código de estado de la respuesta.

El código de estado es un número que indica el resultado de la operación. Este código es un número de tres dígitos más un nombre. El primero de los tres dígitos indica la categoría del estado. Existen cinco categorías, las describimos:

Código	Categoría
1xx	Informacional
2xx	Exitoso
3xx	Redirección
4xx	Error del cliente
5xx	Error del servidor

Además del número de tres dígitos los códigos de estado tienen una descripción, por ejemplo, la descripción del famoso código de estado 404 es "*Not Found*" que significa no encontrado.

Vamos a profundizar sobre estas categorías y veremos ejemplos de códigos de estado de cada una:

- **1xx Respuestas informativas**: cuando enviamos una petición HTTP al servidor, el servidor verifica si va a procesar la solicitud. De ser así, le devuelve una respuesta al cliente que la petición va a ser procesada y que debe esperar la respuesta final más adelante, quizás dicha respuesta llegará en unos segundos. Esta es una respuesta transitoria y no final. Algunos ejemplos de códigos de estado de esta categoría son:
    - ***100 Continue***: esta respuesta quiere decir que la cabecera de la petición ha sido recibida y que el cliente debería ver el cuerpo de la petición. Esta es una forma eficiente de intercambio de "paquetes de información" porque da la oportunidad al servidor de denegar una petición HTTP sin tener que recibir y procesar todo el cuerpo de la petición HTTP. Además desde el punto de vista de la seguridad de la información del usuario, el servidor puede detectar si la conexión va a ser insegura es decir utilizando HTTP y no HTTPS, y poder denegar dicha operación sin que el cliente exponga sus datos enviando el cuerpo de la petición sobre una conexión insegura.
    - ***101 Switching Protocols***: esta respuesta se da cuando el cliente pide un upgrade o actualización de protocolo y el servidor está de acuerdo con la demanda. Un ejemplo de esto es cuando el cliente pide al servidor actualizar la conexión para utilizar Web Socket.

- **2xx Códigos exitosos**: esta categoría es para indicar que la petición ha sido exitosa. Algunos ejemplos de códigos de estado de esta categoría son:
    - ***200 OK***: probablemente el más común y significa que la operación ha sido exitosa. Si por ejemplo hacemos una petición GET sobre un recurso, el 200 OK indicaría que el recurso ha sido devuelto en la respuesta. Si intentamos hacer una actualización sobre un recurso utilizando un PUT, entonces un 200 OK indicaría que el recurso fue actualizado exitosamente.
    - ***201 Created***: la petición se ha completado resultando en la creación de un nuevo recurso. Esta es la respuesta exitosa específica que debemos retornar cuando la petición del usuario resulta en crear un nuevo recurso.
    - ***202 Accepted***: la petición ha sido aceptada para ser procesada pero el procesamiento aún no ha terminado. Esto se utiliza principalmente para largos procesos en los cuales el usuario no tiene que esperar un resultado inmediato, sino que en el futuro se le va a indicar el resultado final.
    - ***204 No Content***: la petición ha sido realizada con éxito y no se va a retornar un cuerpo en la respuesta, es decir en este caso el servidor sólo debe de enviar la línea de respuesta y la cabecera de la respuesta.

- **3xx Códigos de redirección**: esta categoría de código de estado indica que se necesita que el cliente tome acciones adicionales para completar la petición. En ocasiones esta acción adicional se cubre de forma automática sin necesidad de la interacción del usuario. Algunos ejemplos de códigos de estado de esta categoría son:
    - ***300 Multiple Choices***: se utiliza para indicar que el recurso solicitado tiene muchas representaciones y el usuario puede elegir la representación preferida y ser redirigido a esa dirección según dicha elección. Un ejemplo de esto es cuando tenemos un vídeo con múltiples formatos y le indicamos al usuario que puede indicar el formato deseado.
    - ***301 Moved Permanently***: se utiliza para indicar una versión permanente de una URL. Esto quiere decir que cualquier link que apunte a esta URL debe ser actualizado por la nueva URL correcta. Dicha URL correcta la devuelve el servidor en la cabecera *Location*.
    - ***302 Found***: indica que el recurso solicitado se encuentra en una URI diferente y en forma temporal, dado que la nueva ubicación es temporal, se necesita que el cliente siga haciendo la petición a la URI original en el futuro.
    - ***303 See Other***: indica que la respuesta del recurso puede ser encontrada en una URI diferente y debe ser obtenida haciendo un GET a dicho recurso. Lamentablemente varios agentes de usuario como los navegadores (User Agent es una aplicación informática que funciona como cliente en un protocolo de red; el nombre se aplica generalmente para referirse a aquellas aplicaciones que acceden a la World Wide Web), implementaron en forma incorrecta la funcionalidad de respuesta del ***302 Found*** y lo implementaron como un ***303 See Other***. Para mayor claridad se introdujo el código 307 que hace lo mismo que el 302, sin embargo se espera la correcta implementación del mismo por los agentes de usuario.
    - ***307 Temporary Redirect***: el recurso solicitado reside temporalmente en una URI. Esto es lo mismo que el 302 Found.
    - ***308 Permanent Redirect***: indica que el recurso destino ha sido asignado a una nueva URI permanente y que cualquier referencia al recurso debe hacerse a la nueva URI indicada. Esto es similar al estado 301 Moved Permanently, sin embargo el estado 308 no permite cambiar un POST por un GET.

- ***4xx Códigos de error del cliente***: esta categoría de códigos de estado indica que el cliente ha cometido un error al realizar la petición. La naturaleza de los errores es determinada por el código de estado. Algunos ejemplos de códigos de estado de esta categoría son:

- ***400 Bad Request***: el servidor no puede completar la petición por un error del cliente. El 400 Bad Request un error genérico. Lo normal y aconsejado es que el cuerpo de la respuesta coloquemos una explicación de qué fue lo que salió mal para que el cliente pueda corregirlo e intentar la petición nuevamente.
- ***401 Unauthorized***: esto quiere decir que el usuario necesita autenticarse o loguearse en la aplicación para poder realizar la acción que intenta realizar.
- ***403 Forbidden***: este código de estado se envía cuando a pesar que el usuario está autenticado en la aplicación, no tiene permisos suficientes para realizar la acción que ha intentado realizar.
- ***404 Not Found***: indica que el recurso solicitado no ha sido encontrado.
- ***405 Method Not Allowed***: el método utilizado no está disponible para el recurso solicitado. Recordamos que los métodos HTTP incluyen GET, POST, PUT, DELETE, etc., entonces si a un recurso que no permitimos que sea borrado se le intenta hacer un DELETE, el servidor retornará este valor de estado.
- ***408 Request Timeout***: el servidor tuvo un "timeout" procesando la petición. Esto ocurre cuando una operación se toma más tiempo del permitido.

- ***5xx Códigos de error del servidor***: esta categoría indica que el servidor ha fallado en poder procesar la petición. Estos errores son diferentes a los errores de la categoría de errores del cliente, pues en la categoría de errores de cliente se entiende que el error es del usuario que interactúa con la aplicación web o del cliente web (el navegador), pero en los errores del servidor el problema radica en el servidor web. Algunos ejemplos de códigos de estado de esta categoría son:
  - ***500 Internal Server Error***: este es un error genérico cuya causa puede ser casi cualquier cosa. Por ejemplo si el código de la api da error porque no se pudo conectar a una base de datos, entonces es probable que un error 500 sea lo que podamos enviar al cliente. Dado que no es su culpa que la api haya perdido comunicación con la base de datos.
  - ***501 Not Implemented***: se da cuando el servidor no reconoce ni soporta el método solicitado por el cliente. Esto es diferente a un método reconocido pero no soportado. Por ejemplo un DELETE es un método reconocido aún cuando no permitamos la operación de borrado sobre un recurso, pero CHUNK no es un método reconocido por el protocolo HTTP, por lo tanto no está implementado.
  - ***503 Servicio Unavailable***: el servidor no está disponible en el momento que se quiere acceder a él, típicamente es algo temporal.