UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°2:

Arquitectura Cliente Servidor y Servicios Web

Material Teórico

1° Año – 2° Cuatrimestre

CC BY ND NC

Índice

ARQUITECTURA CLIENTE SERVIDOR Y SERVICIOS WEB

Introducción

En el entorno actual de las empresas, el sistema de información se estructura alrededor de bases de datos que contienen datos críticos, accesibles de forma remota por empleados a través de un modelo cliente-servidor. En este modelo, los datos residen en servidores centralizados, mantenidos por administradores de sistemas, mientras que los empleados utilizan clientes para acceder y manipular estos datos, por ejemplo, integrándolos en hojas de cálculo. Paralelamente, el diseño de software evoluciona hacia sistemas modulares, con aplicaciones compuestas por componentes reutilizables distribuidos en una red de máquinas interconectadas.

Los Servicios Web desempeñan un rol crucial al permitir la distribución de aplicaciones a través de Internet. Estos servicios facilitan la exposición de APIs en la web, permitiendo a aplicaciones cliente interactuar con los datos ofrecidos por cualquier servidor conectado a la red global. Esta capacidad de integración y acceso a datos remotos define el papel esencial de los Servicios Web en la arquitectura moderna de aplicaciones empresariales.

El objetivo de esta unidad es explorar los conceptos fundamentales de los modelos cliente-servidor, con un enfoque específico en el desarrollo de aplicaciones web. Se analizarán las características esenciales de los modelos cliente-servidor, centrándonos en cómo el servidor utiliza una base de datos para generar y actualizar páginas web en respuesta a las solicitudes de los clientes. Además, se estudiarán las tecnologías necesarias para construir una API Web utilizando C#, con el fin de exponer una interfaz de programación de aplicaciones (API) de negocio.

CARACTERÍSTICAS Y COMPONENTES

La **arquitectura cliente servidor** tiene dos partes claramente diferenciadas, por un lado la parte del servidor y por otro la parte de cliente o grupo de clientes donde lo habitual es que un servidor sea una máquina bastante potente con un hardware y software específico que actúa de depósito de datos y funcione como un sistema gestor de base de datos o aplicaciones.

Esta arquitectura se aplica en diferentes modelos informáticos alrededor del mundo donde su propósito es mantener una comunicación de información entre diferentes entidades de una red mediante el uso de protocolos establecidos y el apropiado almacenaje de la misma. El más claro ejemplo de uso de una arquitectura cliente servidor es la red de **Internet** donde existen ordenadores de diferentes personas conectadas alrededor del mundo, las cuales se conectan a través de los servidores de su proveedor de Internet por ISP donde son redirigidos a los servidores de las páginas que desean visualizar y de esta manera la información de los servicios requeridos viajan a través de Internet dando respuesta a la solicitud demandada.

La principal importancia de este modelo es que permite *conectar a varios clientes a los servicios que provee un servidor* y como se sabe hoy en día, la mayoría de las aplicaciones y servicios tienen como gran necesidad que puedan ser consumidos por varios usuarios de forma simultánea.

Algunos ejemplos de la arquitectura cliente servidor son: navegar una web, usar un protocolo FTP, SSH, los juegos en red o un servidor de Correo, entre otros.

Componentes

Para entender este modelo se nombran a continuación algunos conceptos básicos que forma un modelo cliente-servidor.

Red: Una red es un conjunto de clientes, servidores y base de datos unidos de una manera física o no física en el que existen protocolos de transmisión de información establecidos.

Cliente: El concepto de cliente hace referencia a un *demandante de servicios*, este cliente puede ser un ordenador como también una aplicación de informática, la cual requiere información proveniente de la red para funcionar.

Servidor: Un servidor hace referencia *a un proveedor de servicios*, este servidor a su vez puede ser un ordenador o una aplicación informática la cual envía información a los demás agentes de la red.

Protocolo: Un protocolo es un conjunto de normas o reglas y pasos establecidos de manera clara y concreta sobre el flujo de información en una red estructurada.

Servicios: Un servicio es un conjunto de información que busca responder las necesidades de un cliente, donde esta información pueden ser mail, música, mensajes simples entre software, videos, etc.

Base de datos: Son bancos de información ordenada, categorizada y clasificada que forman parte de la red, que son sitios de almacenaje para la utilización de los servidores y también directamente de los clientes.

Clientes y servidores

Como se ha mencionado anteriormente una máquina cliente y servidor se refieren a computadoras que son usadas para diferentes propósitos.

El cliente es un computador pequeño con una estructura al igual a la que se tiene en oficinas u hogares la cual accede a un servidor o a los servicios del mismo a través de Internet o una red interna. Un claro ejemplo a este caso es la forma en que trabaja una empresa modelo con diferentes computadores donde cada uno de ellos se conectan a un servidor para poder obtener archivos de una base de datos o servicios ya sea correos electrónicos o aplicaciones.

El servidor al igual que el cliente, es una computadora pero con diferencia de que tiene una gran capacidad que le permite almacenar gran cantidad de diversos de archivos, o correr varias aplicaciones en simultaneo para así nosotros los clientes poder acceder los servicios. Los mismos pueden contener y ejecutar aplicaciones, sitios web, almacenaje de archivos, diversas bases de datos, entre muchos más.

Es importante mencionar que un cliente también puede tener una función de servidor ya que el mismo puede almacenar datos en su disco duro para luego ser usados en vez de estar conectándose al servidor continuamente por una acción que quizás sea muy sencilla.

Graficamente:

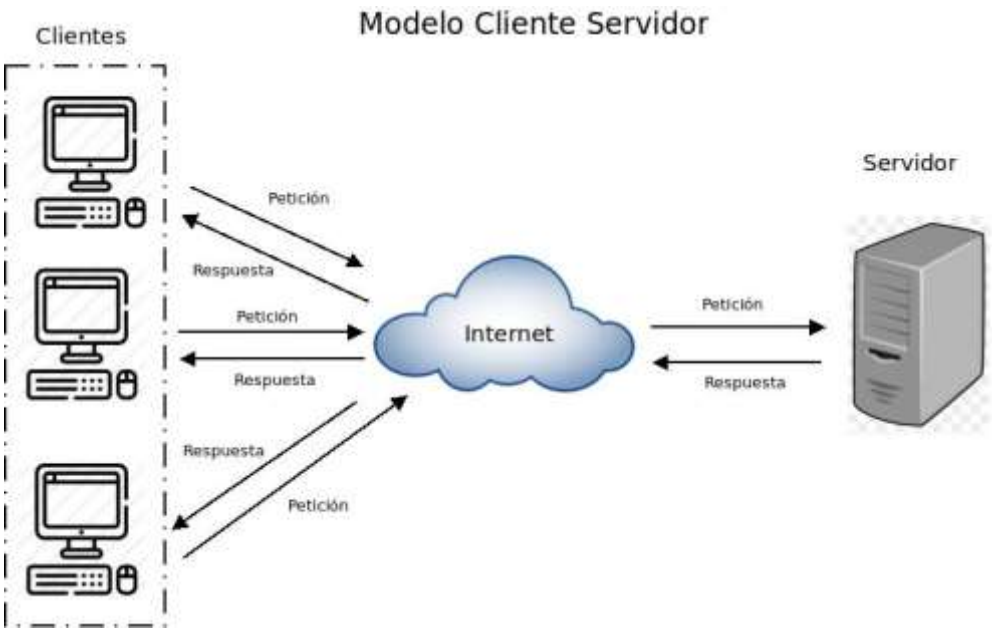

Imagen 1: Elaboración propia

Ventajas y Desventajas

Este modelo cliente servidor tiene varias ventajas y desventajas las cuales son importantes mencionar al momento de decidir qué tipo de arquitectura es más conveniente según la solución a desarrollar:

Ventajas

- Facilita la integración entre diferentes sistemas y comparte información permitiendo por ejemplo que las máquinas ya existentes puedan ser utilizadas mediante una interfaz más amigable para el usuario. Es un modelo flexible y adaptable al servicio que se quiere implementar. Esto permite aumentar el rendimiento así como también, envolver variadas plataformas, bases de datos, redes y sistemas operativos que pueden ser de diferentes distribuidores con arquitecturas totalmente diferentes y funcionando todos al mismo tiempo.
- También es escalable y ante una gran demanda de tráfico se pueden utilizar tecnologías complementarias, por lo que cualquier organización que utilice estos sistemas adquiere ventajas competitivas.
- La estructura modular facilita la integración de nuevas tecnologías y el crecimiento de la infraestructura computacional favoreciendo así la estabilidad de las soluciones.

- Permite el acceso simultáneo de varios clientes al mismo servidor. Esto es de gran utilidad ya que por ejemplo una empresa con varios empleados distribuidos físicamente puede trabajar simultáneamente desde su área o lugar remoto de trabajo.

Desventajas

- Se requiere habilidad para que un servidor sea reparado. Por ejemplo si un problema ocurre en la red, se requiere de alguien con un amplio conocimiento de esta para poder repararla en su totalidad para así dejar que la información y el correcto funcionamiento siga su flujo.
- Otro problema es la seguridad, el hecho que se comparte canales de información entre servidores y clientes requieren que estas pasen por procesos de validación, es decir protocolos de seguridad que pueden tener algún tipo de puerta abierta permitiendo que se generen daños físicos, amenazas o ataques de malware.
- Este modelo representa una limitación importante en cuanto a los costos económicos debido a que estos servidores son computadoras de alto nivel con un hardware y software específicos para poder dar un correcto funcionamiento a nuestras aplicaciones.

SISTEMAS BASADOS EN HTTP

HTTP (Hypertext Transfer Protocol) es un protocolo basado en el principio de cliente-servidor: las peticiones son enviadas por una entidad: **el agente** del usuario (o un proxy a petición de uno). La mayoría de las veces el cliente es un navegador Web, pero podría ser cualquier otro programa, como por ejemplo un programa-robot, que explore la Web, para adquirir datos de su estructura y contenido para uso de un buscador de Internet.

Cada petición individual se envía a un **servidor**, el cuál la gestiona y responde. Entre cada *petición* y *respuesta*, hay varios intermediarios, normalmente denominados **proxies**, los cuales realizan distintas funciones, como: *gateways* o *caches*.

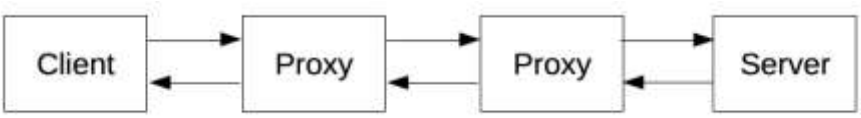

Imagen 2: Elaboración propia

En realidad, hay más elementos intermedios, entre un navegador y el servidor que gestiona su petición: hay otros tipos de dispositivos: como *routers*, *módems*. Es gracias a la arquitectura en capas de la Web, que estos intermediarios, son transparentes al navegador y al servidor, ya que HTTP se apoya en los protocolos de red y transporte. HTTP es un protocolo de aplicación, y por tanto se apoya sobre los anteriores. Aunque para diagnosticar problemas en redes de comunicación, las capas inferiores son irrelevantes para la definición del protocolo HTTP.

HTTP es la base de cualquier intercambio de datos en la Web, y un protocolo de estructura cliente-servidor como se mencionó anteriormente, esto quiere decir que una petición de datos es iniciada por el elemento que recibirá los datos (el cliente), normalmente un navegador Web. Así, una página web completa resulta de la unión de distintos sub-documentos recibidos, como, por ejemplo: un documento que especifique el estilo de maquetación de la página web (CSS), el texto, las imágenes, vídeos, scripts, etc...

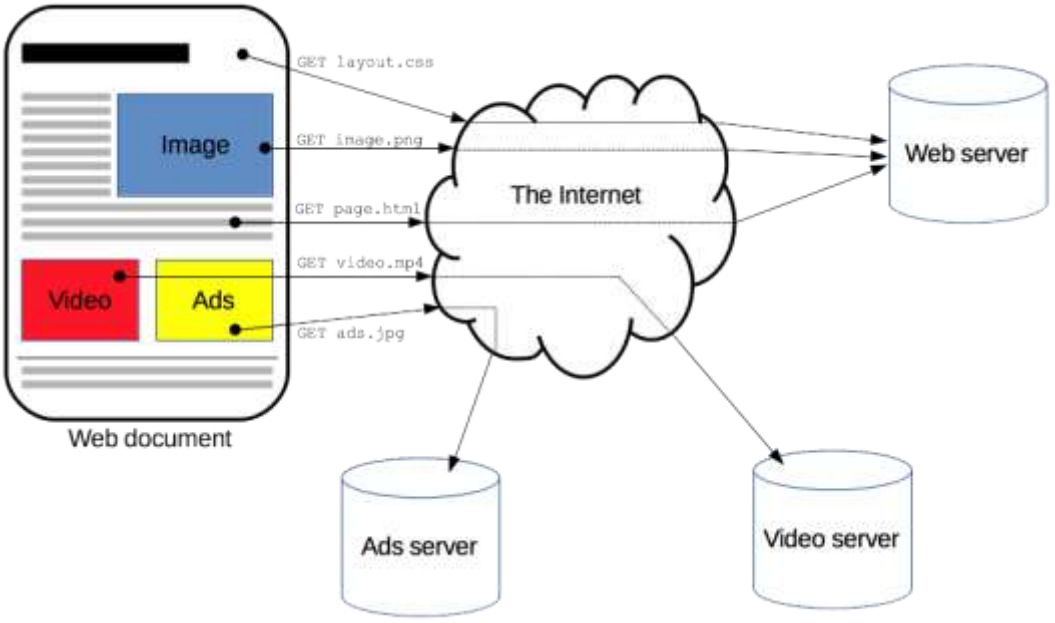

Imagen 3: Elaboración propia

Clientes y servidores se comunican intercambiando mensajes individuales (en contraposición a las comunicaciones que utilizan flujos continuos de datos). Los mensajes que envía el cliente, normalmente un navegador Web, se llaman *peticiones*, y los mensajes enviados por el servidor se llaman *respuestas*.

Cliente: el agente del usuario

El agente del usuario, es cualquier herramienta que actué en representación del usuario. Esta función es realizada en la mayor parte de los casos por un navegador Web. Hay excepciones, como el caso de programas específicamente usados por desarrolladores para desarrollar y depurar sus aplicaciones.

El navegador es **siempre** el que inicia una comunicación (petición), y el servidor nunca la comienza (hay algunos mecanismos que permiten esto, pero no son muy habituales).

Para poder mostrar una página Web, el navegador envía una petición de documento HTML al servidor. Entonces procesa este documento, y envía más peticiones para solicitar scripts, hojas de estilo (CSS), y otros datos que necesite (normalmente vídeos y/o imágenes). El navegador, une todos estos documentos y datos, y compone el resultado final: la página Web. Los scripts, los ejecuta también el navegador, y también pueden generar más peticiones de datos en el tiempo, y el navegador, gestionará y actualizará la página Web en consecuencia.

Una página Web, es un documento de hipertexto (HTTP), luego habrá partes del texto en la página que puedan ser enlaces (links) que pueden ser activados (normalmente al hacer click sobre ellos) para hacer una petición de una nueva página Web, permitiendo así dirigir su agente de usuario y navegar por la Web. El navegador, traduce esas direcciones en peticiones de HTTP, e interpretara y procesará las respuestas HTTP, para presentar al usuario la página Web que desea.

El servidor Web

Al otro lado del canal de comunicación, está el servidor, el cual *"sirve"* los datos que ha pedido el cliente. Un servidor conceptualmente es una unica entidad, aunque puede estar formado por varios elementos, que se reparten la carga de peticiones, (load balancing), u otros programas, que gestionan otros computadores (como cache, bases de datos, servidores de correo electrónico, ...), y que generan parte o todo el documento que ha sido pedido.

Un servidor no tiene que ser necesariamente un único equipo físico, aunque si que varios servidores pueden estar funcionando en un único computador. En el estándar HTTP/1.1 y Host, pueden incluso compartir la misma dirección de IP.

Proxies

Entre el cliente y el servidor, además existen distintos dispositivos que gestionan los mensajes HTTP. Dada la arquitectura en capas de la Web, la mayoría de estos dispositivos solamente gestionan estos mensajes en los niveles de protocolo inferiores: capa de transporte, capa de red o capa física, siendo así transparentes para la capa de comunicaciones de aplicación del HTTP, además esto aumenta el rendimiento de la comunicación. Aquellos dispositivos, que sí operan procesando la capa de aplicación son conocidos como proxies. Estos pueden ser transparentes, o no (modificando las peticiones que pasan por ellos), y realizan varias funciones:

- caching (la caché puede ser pública o privada, como la caché de un navegador)
- filtrado (como un anti-virus, control parental, ...)
- balanceo de carga de peticiones (para permitir a varios servidores responder a la carga total de peticiones que reciben)
- autentificación (para el control al acceso de recursos y datos)
- registro de eventos (para tener un histórico de los eventos que se producen)

Características clave del protocolo HTTP

HTTP es sencillo

Incluso con el incremento de complejidad, que se produjo en el desarrollo de la versión del protocolo HTTP/2, en la que se encapsularon los mensajes, HTTP está pensado y desarrollado para ser leído y fácilmente interpretado por las personas, haciendo de esta manera más fácil la depuración de errores, y reduciendo la curva de aprendizaje para las personan que empieza a trabajar con él.

HTTP es extensible

Presentadas en la versión HTTP/1.0, las cabeceras de HTTP, han hecho que este protocolo sea fácil de ampliar y de experimentar con él. Funcionalidades

nuevas pueden desarrollarse, sin más que un cliente y su servidor, comprendan la misma semántica sobre las cabeceras de HTTP.

HTTP es un protocolo con sesiones, pero sin estados

HTTP es un protocolo sin estado, es decir: no guarda ningún dato entre dos peticiones en la misma sesión. Esto crea problemáticas, en caso de que los usuarios requieran interactuar con determinadas páginas Web de forma ordenada y coherente, por ejemplo, para el uso de "cestas de la compra" en páginas que utilizan en comercio electrónico. Pero, mientras HTTP ciertamente es un protocolo sin estado, el uso de HTTP cookies, si permite guardar datos con respecto a la sesión de comunicación. Usando la capacidad de ampliación del protocolo HTTP, las cookies permiten crear un contexto común para cada sesión de comunicación.

HTTP y conexiones

Una conexión se gestiona al nivel de la capa de trasporte, y por tanto queda fuera del alcance del protocolo HTTP. Aún con este factor, HTTP no necesita que el protocolo que lo sustenta mantenga una conexión continua entre los participantes en la comunicación, solamente necesita que sea un protocolo fiable o que no pierda mensajes (como mínimo, en todo caso, un protocolo que sea capaz de detectar que se ha pedido un mensaje y reporte un error). De los dos protocolos más comunes en Internet, TCP es fiable, mientras que UDP, no lo es. Por lo tanto HTTP, se apoya en el uso del protocolo TCP, que está orientado a conexión, aunque una conexión continua no es necesaria siempre.

FLUJO DE MENSAJES HTTP

Cuando el cliente quiere comunicarse con el servidor, tanto si es directamente con él, o a través de un proxy intermedio, realiza los siguientes pasos:

1. Abre una conexión TCP: la conexión TCP se usará para hacer una petición, o varias, y recibir la respuesta. El cliente pude abrir una conexión nueva, reusar una existente, o abrir varias a la vez hacia el servidor.
2. Hacer una petición HTTP: Los mensajes HTTP (previos a HTTP/2) son legibles en texto plano. A partir de la versión del protocolo HTTP/2, los mensajes se encapsulan en franjas, haciendo que no sean directamente interpretables, aunque el principio de operación es el mismo.

```
GET / HTTP/1.1

Host: developer.mozilla.org

Accept-Language: fr
```

3. Leer la respuesta enviada por el servidor:

```
HTTP/1.1 200 OK

Date: Sat, 09 Oct 2010 14:28:02 GMT

Server: Apache

Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT

ETag: "51142bc1-7449-479b075b2891b"

Accept-Ranges: bytes

Content-Length: 29769

Content-Type: text/html


<!DOCTYPE html... (here comes the 29769 bytes of the requested web
page)
```

4. Cierre o reuso de la conexión para futuras peticiones.

En las versiones del protocolo HTTP/1.1 y anteriores los mensajes eran de formato texto y eran totalmente comprensibles directamente por una persona. En HTTP/2, los mensajes están estructurados en un nuevo formato binario y las tramas permiten la compresión de las cabeceras y su multiplexación. Así pues, incluso si solamente parte del mensaje original en HTTP se envía en este formato, la semántica de cada mensaje es la misma y el cliente puede formar el mensaje original en HTTP/1.1. Luego, es posible interpretar los mensajes HTTP/2 en el formato de HTTP/1.1.

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

Peticiones (request)

Un ejemplo de petición HTTP:

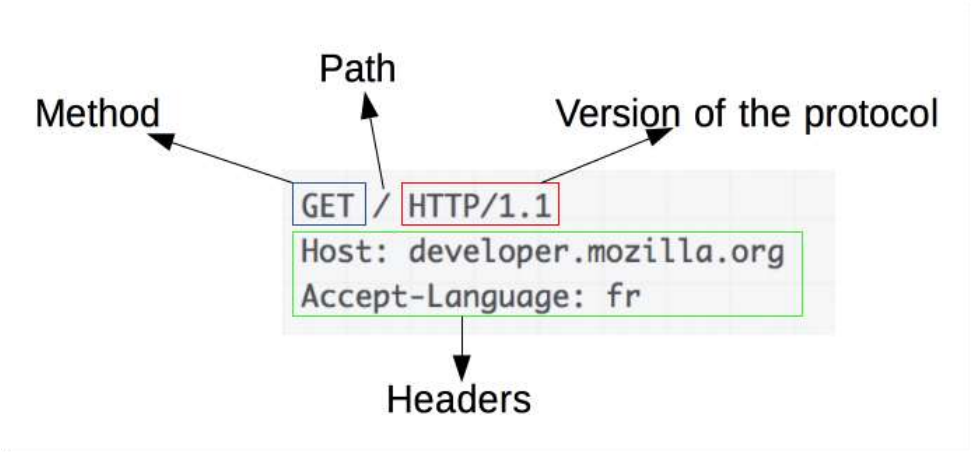

Imagen 4: Elaboración propia

Una petición de HTTP, está formado por los siguientes campos:

- Un **método HTTP**, normalmente pueden ser un verbo,ç como: *GET, POST* o un nombre como: OPTIONS o HEAD, que defina la operación que el cliente quiera realizar. El objetivo de un cliente, suele ser una petición de recursos, usando GET, o presentar un valor de un formulario HTML, usando POST, aunque en otras ocasiones puede hacer otros tipos de peticiones.
- La dirección del recurso pedido; la URL del recurso, sin los elementos obvios por el contexto, como pueden ser: sin el protocolo (http://), el dominio (aquí developer.mozilla.org), o el puerto TCP (aquí el 80).
- La versión del protocolo HTTP.
- Cabeceras HTTP opcionales, que pueden aportar información adicional a los servidores.
- Cuerpo de mensaje, en algún método, como puede ser POST, en el cual envía la información para el servidor.

Respuestas (response)

Un ejemplo de repuesta:

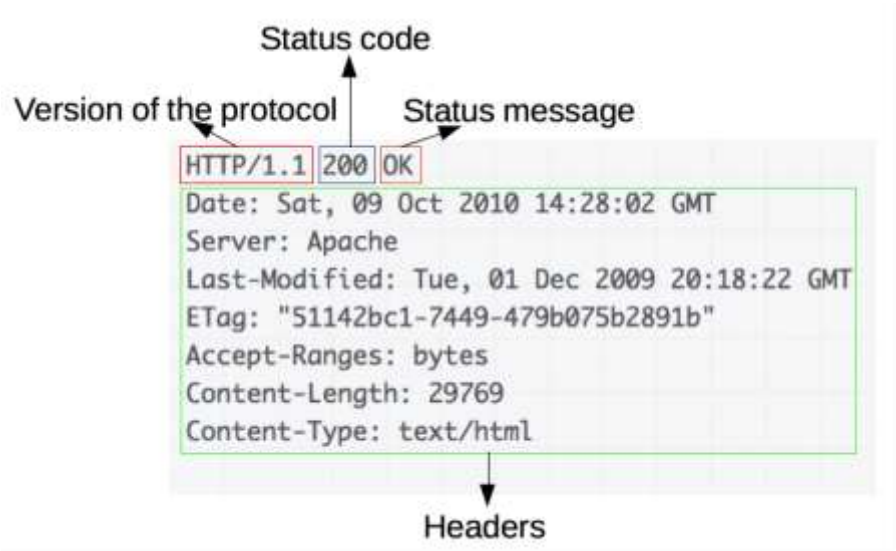

Imagen 5: Elaboración propia

Las respuestas están formadas por los siguientes campos:

- La versión del protocolo HTTP que están usando.
- Un **código de estado**, indicando si la petición ha sido exitosa, o no, y debido a que.
- Un mensaje de estado, una breve descripción del código de estado.
- Cabeceras HTTP, como las de las peticiones.
- Opcionalmente, el recurso que se ha pedido.

Métodos de solicitud

HTTP define un conjunto de métodos de solicitud para indicar la acción que se desea realizar para un recurso determinado. Aunque también pueden ser sustantivos, estos métodos de solicitud a veces se denominan *verbos HTTP*. Aunque comparten algunas características comunes, cada uno de ellos implementa una semántica diferente. A continuación la tabla 1 enumera cada método HTTP con su correspondiente significado.

Método HTTP	Significado
GET	Solicita una representación del recurso especificado. Las solicitudes que utilizan GET solo deben recuperar datos, esto implica que una de las principales características de una petición GET es que no debe causar efectos secundarios en el servidor, no deben producir nuevos registros, ni modificar los ya existentes.
HEAD	Solicita una respuesta idéntica a la de una solicitud *GET*, pero sin el cuerpo de la respuesta.
POST	Se utiliza para enviar una entidad al recurso especificado, lo que a menudo provoca un cambio de estado o efectos secundarios en el servidor.
PUT	Reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la solicitud. Al igual que PATCH ambos se usan para modificar un recurso existente.
DELETE	Elimina un recurso especificado.
CONNECT	Convierte la solicitud en un túnel TCP/IP. Normalmente se usa para crear comunicaciones HTTPS a través de proxys HTTP sin encriptación.
OPTIONS	Se utiliza para describir las opciones de comunicación para el recurso de destino.
TRACE	Realiza una prueba de bucle de mensajes a lo largo de la ruta al recurso de destino.
PATCH	Se utiliza para aplicar modificaciones parciales a un recurso.

Tabla 1: Elaboración propia.

Códigos de Estado

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases:

- Respuestas informativas (`100`–`199`),
- Respuestas satisfactorias (`200`–`299`),
- Redirecciones (`300`–`399`),
- Errores de los clientes (`400`–`499`),

- Errores de los servidores (`500`–`599`).

La tabla 2 enumera los códigos de uso más frecuentes:

Código	Nombre	Tipo	Descripción
100	Continue	Informativa	Esta respuesta provisional indica que todo hasta ahora está bien y que el cliente debe continuar con la solicitud o ignorarla si ya está terminada.
101	Switching Protocol	Informativa	Este código se envía en respuesta a un encabezado de solicitud *Upgrade* por el cliente e indica que el servidor acepta el cambio de protocolo propuesto por el agente de usuario.
200	Ok	Satisfactoria	La solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP
201	Created	Satisfactoria	La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT.
202	Acepted	Satisfactoria	La solicitud se ha recibido, pero aún no se ha actuado. Es una petición "sin compromiso", lo que significa que no hay manera en HTTP que permite enviar una respuesta asíncrona que indique el resultado del procesamiento de la solicitud. Está pensado para los casos en que otro proceso o servidor maneja la solicitud, o para el procesamiento por lotes.
301	Moved Permanently	Redirección	Este código de respuesta significa que la URI del recurso solicitado ha sido cambiado. Probablemente una nueva URI sea devuelta en la respuesta.
308	Permanent Redirect	Redirección	Significa que el recurso ahora se encuentra permanentemente en otra URI, especificada por la respuesta de encabezado HTTP **Location**. Tiene la misma semántica que el código de respuesta HTTP 301

			Moved Permanently, con la excepción de que el agente usuario no debe cambiar el método HTTP usado: si un POST fue usado en la primera petición, otro POST debe ser usado en la segunda petición.
400	Bad request	Error de cliente	Esta respuesta significa que el servidor no pudo interpretar la solicitud dada una sintaxis inválida.
401	Unauthorized	Error de cliente	Es necesario autenticar para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, la autenticación es posible.
403	Forbidden	Error de cliente	El cliente no posee los permisos necesarios para cierto contenido, por lo que el servidor está rechazando otorgar una respuesta apropiada.
404	Not found	Error de cliente	El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.
500	Server Internal Error	Error de servidor	El servidor ha encontrado una situación que no sabe cómo manejarla.
503	Service Unavailable	Error de servidor	El servidor no está listo para manejar la petición. Causas comunes puede ser que el servidor está caído por mantenimiento o está sobrecargado

Tabla 2: Elaboración propia.

SERVICIOS WEB

Concepto de API (Application Programming Interface)

Imagínese que se necesita refactorizar el presupuestador de carpintería metálica pero ahora con posibilidad de que cualquier cliente vía WEB pueda dejar su solicitud de aberturas. Evidentemente se debe pensar en una aplicación cliente-servidor que permita generar un presupuesto por Internet. Suponga que además el dueño solicita la posibilidad de registrar un cobro (en concepto de adelanto) para que el cliente pueda confirmar la ejecución de su presupuesto. ¿Desarrollaría una tecnología para hacer cobros desde cero? La respuesta es simplemente no, directamente se desarrolla un componente que se conecte a una pasarela de pago como Paypal u otras ofrecidas en el mercado y se aprovecha el desarrollo hecho por otras empresas en la aplicación. Ese es el espíritu real de las APIs.

Desde el punto de vista conceptual una API es una **interfaz.** Las interfaces constituyen una capa de abstracción para que dos programas se comuniquen entre si (compartiendo datos) sin necesidad de que ninguno conozca los detalles de implementación del otro. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero. Las API le otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones, y proporcionan oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales).

A veces, las API se consideran como contratos, con documentación que representa un acuerdo entre las partes: si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte.

Debido a que simplifican la forma en que los desarrolladores integran los elementos de las aplicaciones nuevas en una arquitectura actual, las API permiten la colaboración entre el equipo comercial y el de TI. Las necesidades comerciales suelen cambiar rápidamente en respuesta a los mercados digitales en constante cambio, donde la competencia puede modificar un sector entero con una aplicación nueva. Para seguir siendo competitivos, es importante admitir la implementación y el desarrollo rápidos de servicios innovadores.

Graficamente:

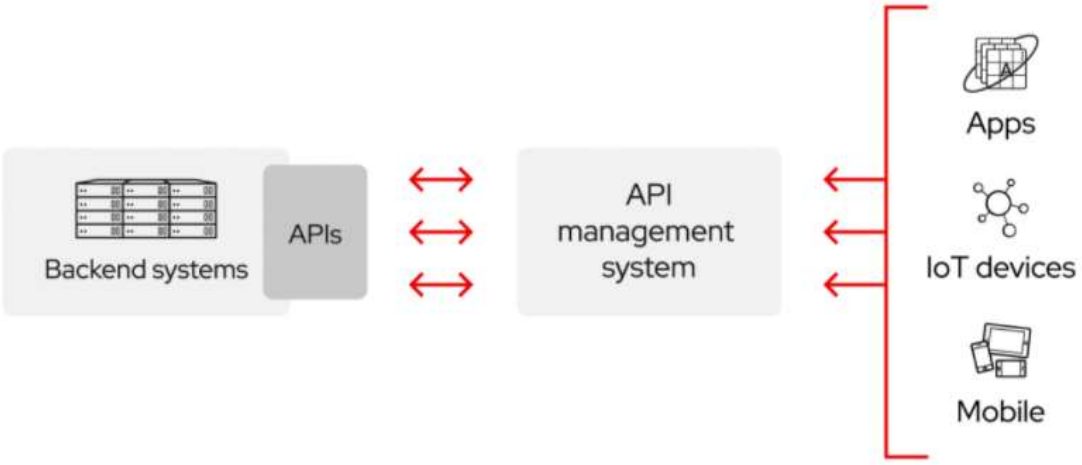

Imagen 1: Recuperado de RedHat

Definición de Servicio Web

Normalmente un Servicio Web es una colección de procedimientos (métodos) a los que podemos llamar desde cualquier lugar de Internet o desde una intranet, siendo este mecanismo de invocación totalmente independiente de la plataforma que utilicemos y del lenguaje de programación en el que se haya implementado internamente el servicio.

Cuando se conecta a un servidor web desde nuestro navegador, el servidor devuelve la página web solicitada, que es un documento que se mostrará en el navegador para que lo visualice el usuario, pero es difícilmente entendible por una máquina. Se puede ver esto como web para humanos. En contraposición, los Servicios Web ofrecen información con un formato estándar que puede ser entendido fácilmente por una aplicación. En este caso se estaría ante una web para máquinas.

Características de los Servicios Web

Los servicios Web son un mecanismo estándar que permite comunicar aplicaciones residentes en equipos conectados a una red. Las características deseables de un servicio Web son:

- Debe poder ser **accesible a través de la Web**. Para ello debe utilizar protocolos de transporte estándares como HTTP, y codificar los mensajes en un lenguaje estándar que pueda conocer cualquier cliente que quiera utilizar el servicio.

- Debe contener una **descripción de sí mismo**. De esta forma, una aplicación podrá saber cuál es la función de un determinado Servicio Web, y cuál es su interfaz, de manera que pueda ser utilizado de forma automática por cualquier aplicación, sin la intervención del usuario.
- Debe poder **ser localizado**. Se debe tener algún mecanismo que permita encontrar un servicio Web que realice una determinada función. De esta forma se tiene la posibilidad de que una aplicación localice el servicio que necesite de forma automática, sin tener que conocerlo previamente el usuario.

Arquitecturas orientadas a servicios

Una arquitectura de software define la forma en la que está diseñado un sistema, cómo se organizan sus componentes, cómo se comunican entre ellos y qué funciones cumplen. En el ámbito de los servicios Web la arquitectura de referencia es SOA (Service-Oriented Architecture).

SOA es una arquitectura que se basa en la integración de aplicaciones mediante servicios, los servicios representan la medida más granular de la arquitectura, sobre la que se construyen otros artefactos. Existen dos enfoques muy comunes que siguen la misma arquitectura orientada a servicios: REST y SOAP.

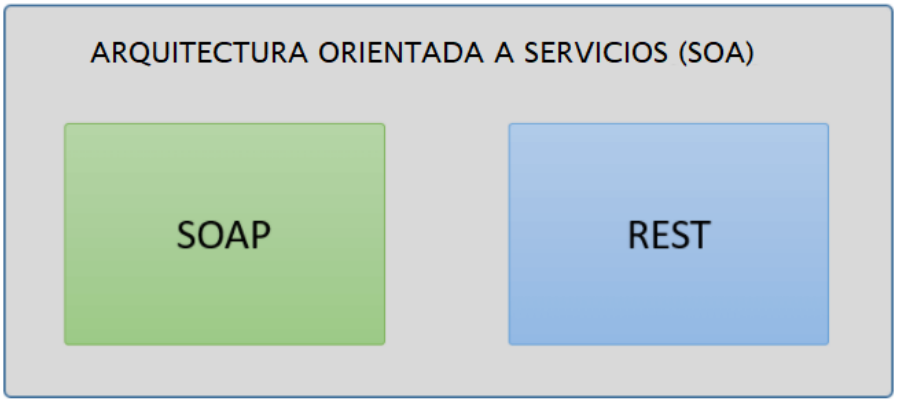

Imagen 2: Elaboración propia

SOAP

El protocolo SOAP (Simple Object Access Protocol) es: “un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML”. Los servicios SOAP funcionan por lo general por el protocolo HTTP que es lo más común cuando invocamos un servicio

Web, sin embargo, SOAP no está limitado a este protocolo, si no que puede ser enviado por FTP, POP3, TCP, Colas de mensajería (JMS, MQ, etc).

Esquemáticamente:

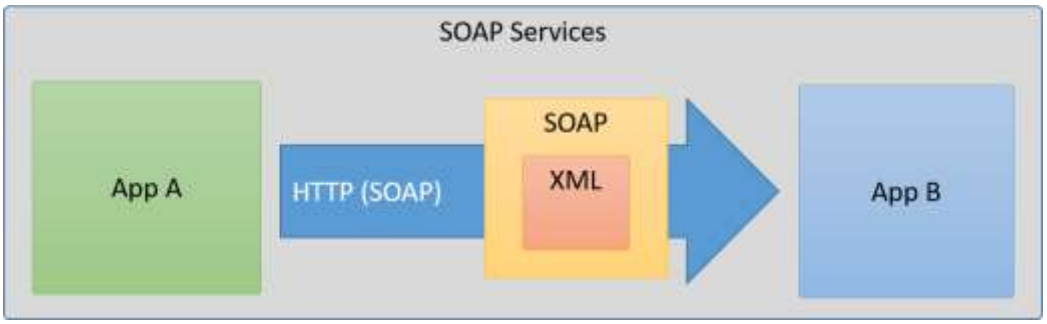

Imagen 3: Elaboración propia

La característica principal de SOAP es que es en sí mismo un protocolo para crear servicios Web. SOAP usa XML (Extensible Markup Language) como lenguaje de intercambio de datos con una estructura compleja que es capaz de albergar todo tipo de datos sobre la solicitud o respuesta generada.

Como es un protocolo, impone reglas integradas que aumentan la complejidad y la sobrecarga, lo cual puede retrasar el tiempo que tardan las páginas en cargarse. Sin embargo, estos estándares también ofrecen normas integradas que pueden ser ideales para el sector empresarial. Con SOAP, es más fácil que las aplicaciones que funcionan en entornos distintos o están escritas en diferentes lenguajes compartan información.

REST

Otra especificación es la Transferencia de Estado Representacional (**REST**).

REST es un conjunto de principios arquitectónicos que se ajusta a las necesidades de los servicios web y las aplicaciones móviles ligeros. Dado que se trata de un conjunto de pautas, la implementación de las recomendaciones depende de los desarrolladores, no es un protocolo como SOAP.

Cuando se envía una solicitud de datos a una API de REST, se suele hacer a través del protocolo HTTP. Una vez que reciben la solicitud, las API diseñadas para REST (conocidas como API o servicios web de RESTful) pueden devolver mensajes en distintos formatos: HTML, XML, texto sin formato y JSON. El formato preferido para los mensajes es la notación de objetos JavaScript (**JSON**), ya que, a pesar de su nombre, puede leerlo cualquier lenguaje de programación, es ligero y lo

comprenden tanto las personas como las máquinas. De esta forma, las API de RESTful son más flexibles y se pueden configurar con mayor facilidad.

Se considera que una aplicación es RESTful si cumple con seis pautas arquitectónicas. Una aplicación de RESTful debe tener lo siguiente:

- Una arquitectura cliente-servidor compuesta por clientes, servidores y recursos.
- Una comunicación cliente-servidor sin estado, lo cual significa que el contenido de los clientes no se almacena en el servidor entre las solicitudes, sino que la información sobre el estado de la sesión queda en el cliente.
- Datos que pueden almacenarse en caché para eliminar la necesidad de algunas interacciones cliente-servidor.
- Una interfaz uniforme entre elementos para que la información se transfiera de forma estandarizada, en lugar de ser específica para las necesidades de cierta aplicación. Roy Fielding, el creador de REST, lo describe como "la característica principal que distingue el estilo arquitectónico de REST de los demás estilos basados en la red".
- Una restricción del sistema en capas, en el que las interacciones cliente-servidor pueden estar mediadas por capas jerárquicas.
- Código según se solicite, lo que permite que los servidores amplíen las funciones de un cliente al transferir el código ejecutable (esto también reduce la visibilidad, así que es una pauta opcional).

La diferencia entre REST y SOAP es básica: SOAP es un protocolo, mientras que REST es un estilo de arquitectura. Esto significa que no hay ningún estándar oficial para las API web de RESTful. Si bien las limitaciones REST pueden parecer demasiadas, son mucho más sencillas que un protocolo definido previamente. Por eso, las API de RESTful son cada vez más frecuentes que las de SOAP.

En los últimos años, la especificación de OpenAPI se ha convertido en un estándar común para definir las API de REST. OpenAPI establece una forma independiente del lenguaje para que los desarrolladores diseñen interfaces API de REST, que permite a los usuarios entenderlas con el mínimo esfuerzo.

Tipos de APIs

Las API pueden ser locales o remotas. Las locales son aquellas se ejecutan dentro del mismo entorno. Por ejemplo, cuando utilizamos una librería .dll para utilizar un servicio de Windows e integrarlo con nuestra aplicación estamos haciendo uso de una API local.

Las APIs remotas están diseñadas para interactuar en una red de comunicaciones. Por "remoto" se refiere a que los recursos que administran las API están, de alguna manera, fuera de la computadora que solicita alguno de dichos recursos. Debido a que la red de comunicaciones más usada es Internet, la mayoría de las API están diseñadas de acuerdo con los estándares web. No todas las API remotas son API web, pero se puede suponer que las API web son remotas.

Las API web normalmente usan HTTP para solicitar mensajes y proporcionar una definición de la estructura de los mensajes de respuesta. Por lo general, estos mensajes de respuesta toman la forma de un archivo XML o JSON, que son los formatos preferidos porque presentan los datos en una manera fácil de manejar para otras aplicaciones.

Esquemáticamente:

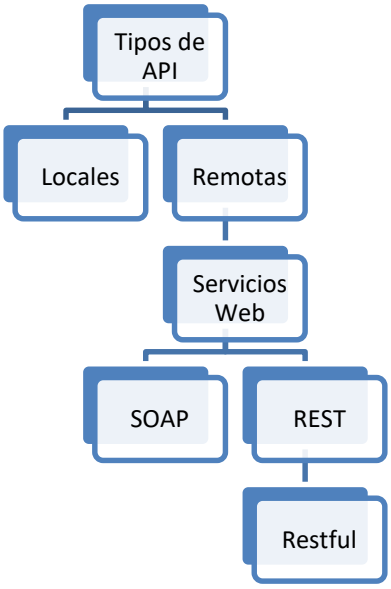

Imagen 4: Elaboración propia

Según el gráfico anterior podemos concluir entonces, que las API Web son APIs expuestas mediante servicios WEB que pueden seguir ciertos principios arquitectónicos como las API REST o bien respetar un protocolo específico como es el caso de los servicios SOAP. A continuación, nos centraremos en construir una Web API con C# siguiendo los principios y recomendaciones del enfoque REST.

CONSTRUIR UNA WEB API CON C#

Desarrollar una API Rest

Cuando se desarrolla una API Rest es necesario comprender ciertos conceptos:

- **Recurso**: Un recurso hace referencia a un concepto importante de nuestro negocio (Facturas, Cursos, Compras etc.). Es lo que habitualmente se denomina un objeto de negocio. Este estilo permite un primer nivel de organización permitiendo acceder a cada uno de los recursos de forma independiente, favoreciendo la reutilización, aumentando la flexibilidad y abordando operaciones de inserción, borrado, búsqueda etc. Cada recurso tiene un identificador único llamado **URI (Identificador de Recurso Uniforme).** Cuando se solicita un recurso a una Web API siempre utilizamos un tipo especial de URI que es la URL (Localizador de Recurso Uniforme). Esta URL contiene la ubicación completa del recurso y generalmente se la llama **endpoint.** Ejemplos de endpoints son:
    - un_dominio/api/products
    - un_dominio/api/products/1
- **Códigos de estado:** cuando solicitamos un recurso el servidor puede contestar con diferentes códigos de estado: 2xx, 3xx, 4xx o 5xx. Con estos códigos podremos saber qué pasó con la petición de nuestro recurso y qué hacer en consecuencia.
- **Verbos:** o métodos HTTP. Nos permite definir cómo interactuar con la API. Dependiendo de cada tipo de operación se utilizará un método diferente de envío.
    - *GET*: Se usará para solicitar consultar a los recursos
    - *POST*: Se usará para insertar nuevos recursos
    - *PUT*: Se usará para actualizar recursos
    - *DELETE*: Se usará para borrar recursos

Imagen 5: Elaboración propia

- **Formatos de respuestas**: las APIs pueden devolver información en diferentes formatos. El formato más común es JSON (Representación de Objetos Javascript). Este no es el único formato válido, es posible devolver archivos XML o incluso texto plano. Lo importante es que sin necesidad de cambiar la API Rest es posible devolver información en diferentes formatos.

.Net Core Web API

Hasta ahora hemos estado trabajando con .NET Framework en su versión 4.x, un framework robusto, altamente probado y eficiente. Sin embargo, al adentrarnos en el desarrollo de aplicaciones web, es el momento propicio para adoptar frameworks más modernos y eficientes. Por lo tanto, en este curso utilizaremos .NET Core como estándar. .NET Core representa la evolución de .NET Framework hacia una arquitectura más contemporánea.

.NET Core es una implementación del estándar .NET que, al igual que otras implementaciones como .NET Framework o Mono, proporciona todo lo necesario para crear y ejecutar aplicaciones, incluyendo compiladores, bibliotecas de clases fundamentales y el entorno de ejecución necesario.

En resumen, con .NET Core puedes desarrollar aplicaciones utilizando lenguajes como C# o VB.NET, utilizando clases básicas estándar como string, int o List<T>, compilarlas a bytecode CIL y ejecutarlas en diversas plataformas. Algunas características distintivas de .NET Core incluyen:

- Es una nueva implementación, escrita prácticamente desde cero.

- Es de código abierto (open source).
- Es multiplataforma.
- Es modular, utilizando paquetes NuGet.
- Las operaciones principales de .NET Core se gestionan desde la línea de comandos.
- Ofrece alto rendimiento, optimizado para manejar un gran número de solicitudes.
- Puede distribuirse de diversas formas.
- Aunque .NET Core no soporta todos los modelos de aplicación ni todos los frameworks, ofrece flexibilidad en el desarrollo de aplicaciones con cualquier editor o IDE.

Web API paso a paso

Para crear una Web API sobre .Net Core vamos a seguir los siguientes pasos:

- **(I) Crear un proyecto WEB:**
  - (a) En el menú *Archivo*, seleccione Nuevo > Proyecto.
  - (b) Seleccione la plantilla ASP.NET Core Web API y haga clic en *Siguiente*

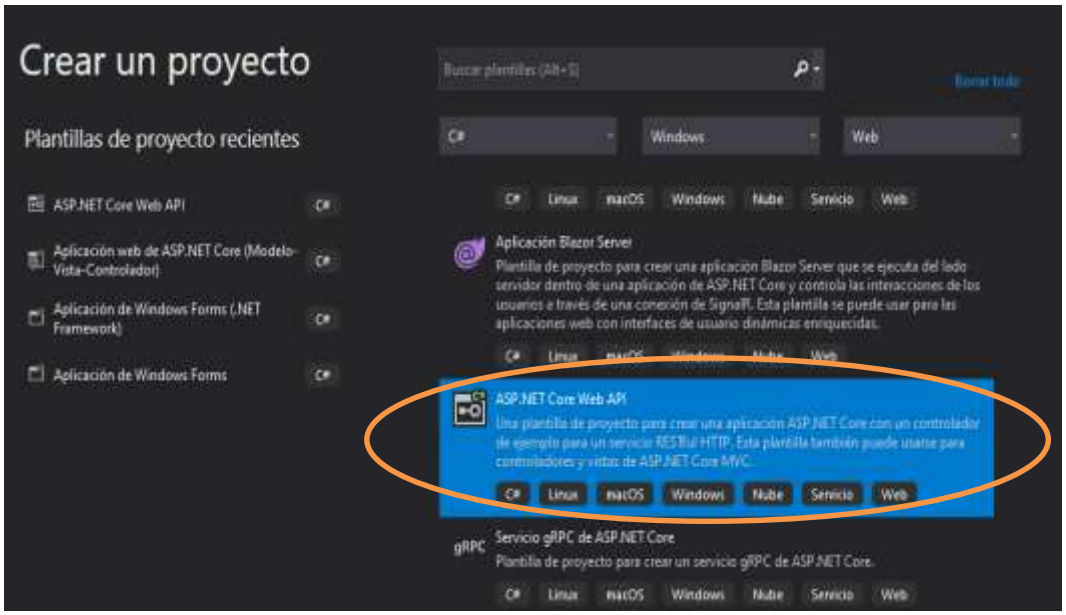

Imagen 6: Elaboración propia

Nombrar el proyecto *FirstWebAPI* y hacer click en Siguiente.

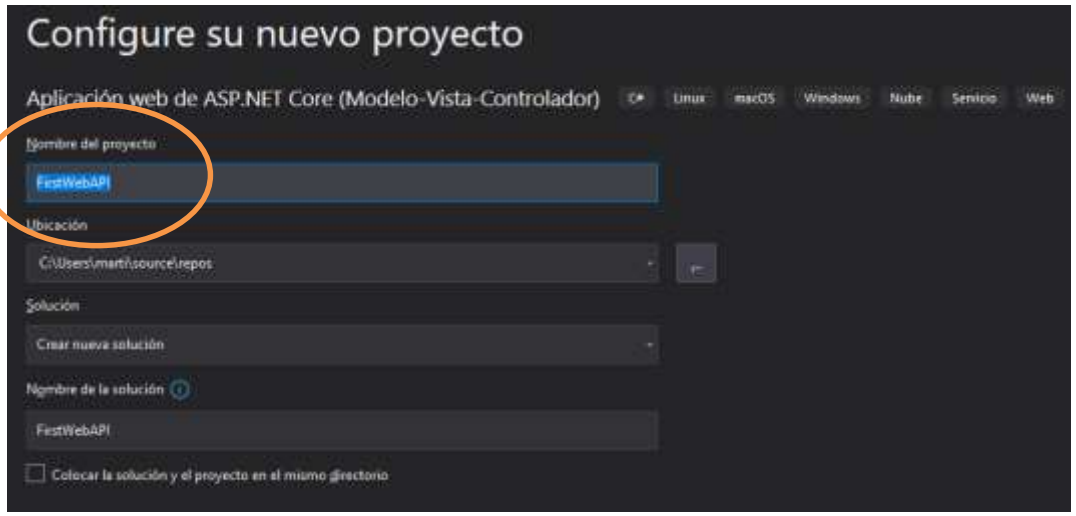

Imagen 7: Elaboración propia

(c) En el cuadro de diálogo Crear una nueva aplicación web ASP.NET Core , confirme que .NET Core y ASP.NET Core 5.0 están seleccionados. Seleccione la plantilla de API y haga clic en *Crear*.

Imagen 8: Elaboración propia

- **(II) Probar el proyecto**

  a) Presione *IIS Express* para ejecutar sin el depurador.

  b) Visual Studio muestra el siguiente cuadro de diálogo cuando un proyecto aún no está configurado para usar SSL:

Imagen 9: Recuperado de microsoft.com

Seleccione Sí si confía en el certificado SSL de IIS Express.

Aparece el siguiente cuadro de diálogo:

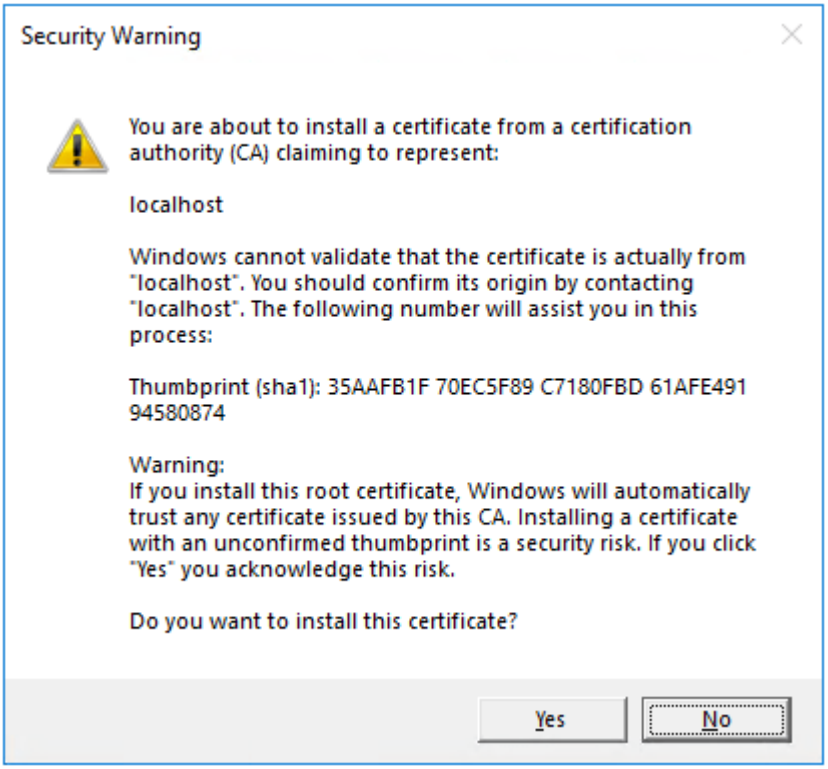

Imagen 10: Recuperado de microsoft.com

Seleccione Sí si acepta confiar en el certificado de desarrollo.

A continuación, se abrirá el Explorador que tengamos configurado por defecto con la siguiente salida:

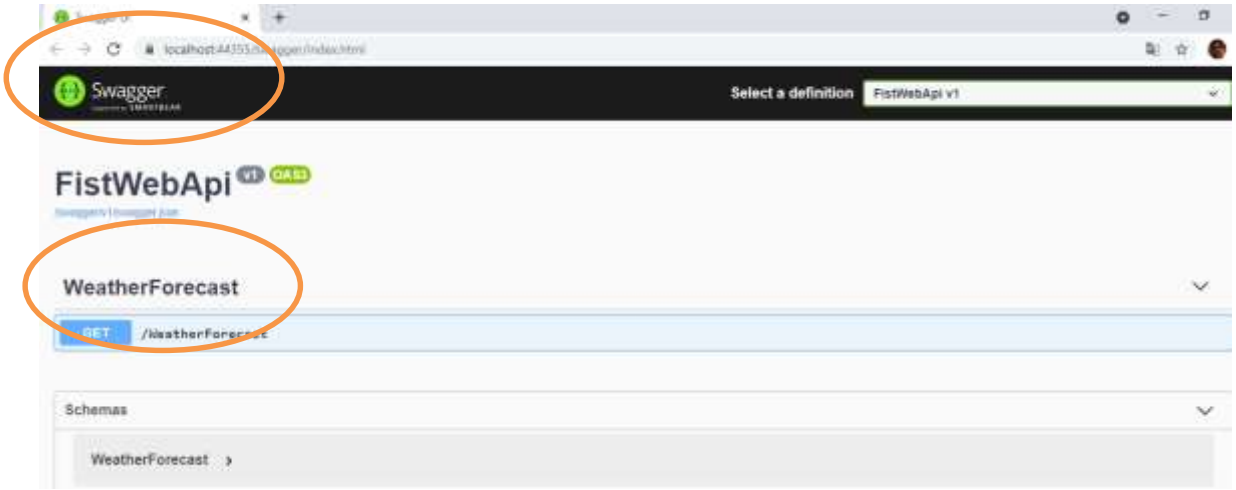

Imagen 11: Elaboración propia

Swagger (OpenAPI) es una especificación independiente del lenguaje para describir las API REST. Permite que tanto las computadoras como los humanos comprendan las capacidades de una API REST sin acceso directo al código fuente. Sus principales objetivos son:

- Minimizar la cantidad de trabajo necesario para conectar servicios desacoplados.
- Reducir la cantidad de tiempo necesario para documentar con precisión un servicio.

Cuando probamos el proyecto sobre IIS (Internet Information Server), servidor por defecto, se ejecuta de manera integrada la herramienta Swagger, que permite probar los servicios de la API, a la vez que ofrece una documentación estandarizada de los mismos. Si se observa la URL escrita en el navegador tiene la forma: https://localhost:44355/swagger/index.html. Desde esta página es posible acceder vía **GET** al **endpoint**: /WeatherForecast y consumir (hacer una petición HTTP) el método Get del controlador **WeatherForecast** (creado automáticamente por el IDE)**.**

Para manejar las peticiones (request) las Web API utilizan **Controllers.** Los controladores son clases que derivan de **ControllerBase.** Toda web API consiste en una o más clases Controllers. La plantilla de proyecto Web API proporciona un controlador de inicio: **WeatherForecastController**.

- **(III) Agregar un modelo**

Un modelo es simplemente una clase que representa datos de una aplicación. Para crear un modelo primero vamos a crear una carpeta llamada *models* y luego agregar una clase llamada Values con las properties Nombre y Valor con se muestra en la siguiente imagen.

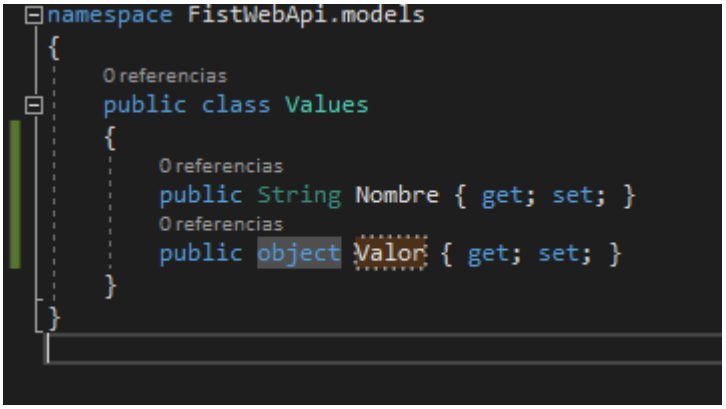

Imagen 12: Elaboración propia

- **(IV) Agregar un controlador propio**

Primero vamos a eliminar desde el Explorador de soluciones el controlador creado por defecto (WeatherForecastController) en la carpeta *Controllers*.

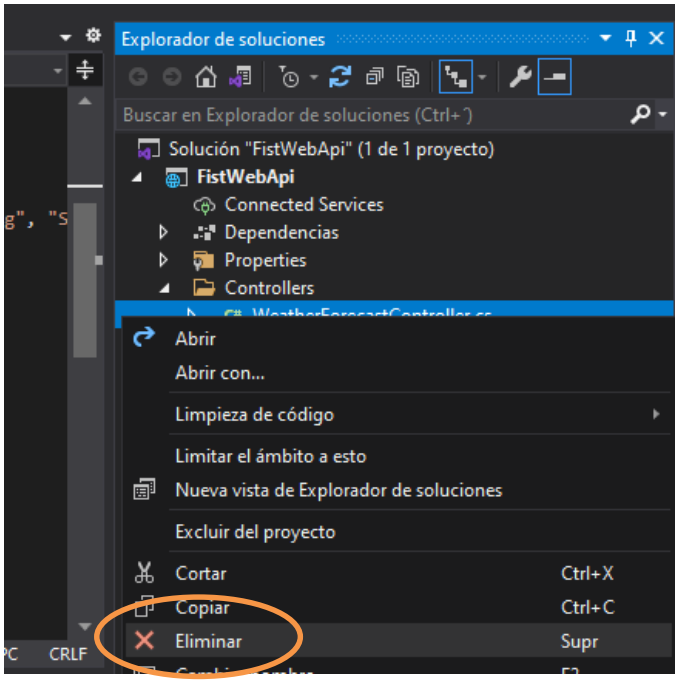

Imagen 13: Elaboración propia

Luego se crea un nuevo controlador haciendo click derecho sobre la carpeta Controllers y seleccionando la opción Agregar>>Controlador…

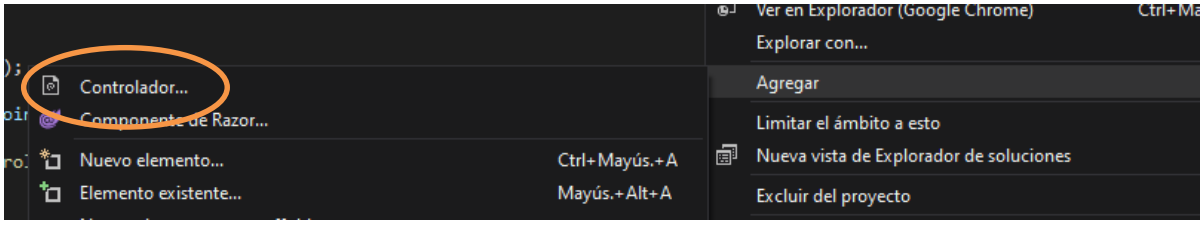

Imagen 14: Elaboración propia

En la pantalla siguiente elegir categoría **API – Controlador de API: en blanco** y seleccionar la opción **Agregar.**

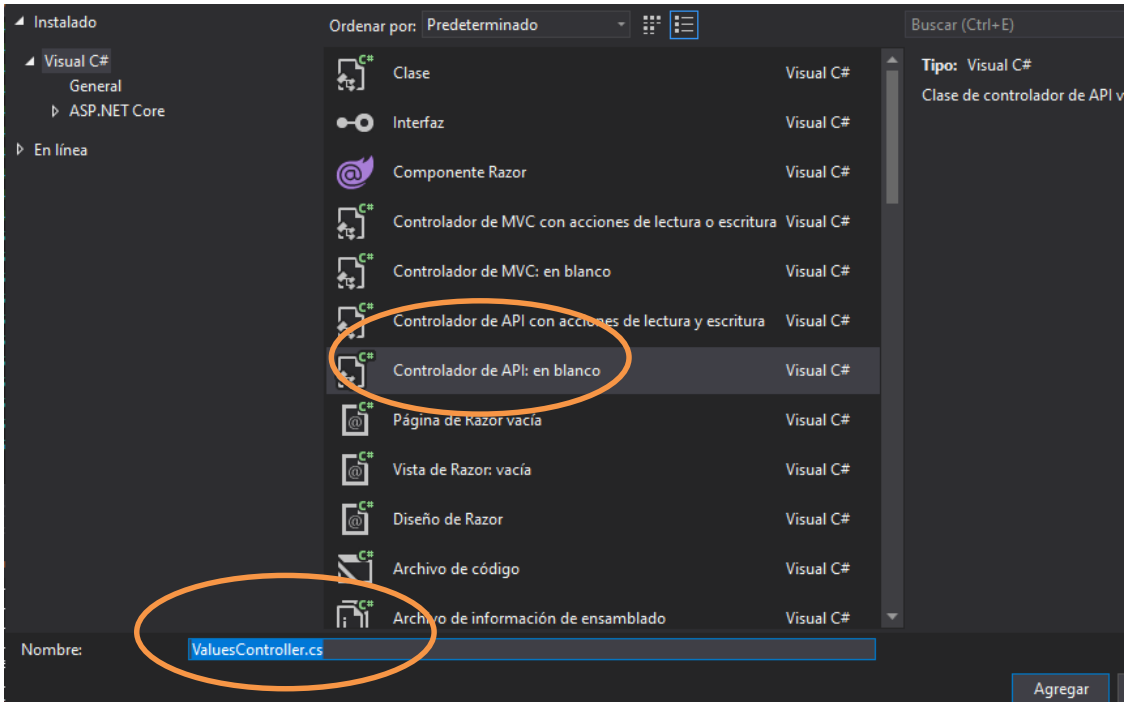

Imagen 15: Elaboración propia

Ingresar un nombre para la clase sufijando siempre con la palabra Controller, por ejemplo: *FirstController*. Mediante este controlador vamos a crear *endpoints* uno para obtener una lista de objetos Values y otro para enviar un nuevo objeto Value desde un cliente. El código completo resulta:

```
[Route("api/[controller]")]
[ApiController]

public class FirstController : ControllerBase
{
    static readonly List<Values> lst = new List<Values>();

    [HttpGet]
    public IActionResult Get()
    {
        return Ok(lst);
    }
```

```
 [HttpPost]
 public IActionResult Save(Values dto)
 {
     if (dto == null)
         return BadRequest();
     lst.Add(dto);
     return Ok("Se añadió existosamente!");
 }

}
```

Notar que:

- La clase hereda de ControllerBase
- En la parte superior se indican dos *decoradores:*
  - ✓ [Route("api/[controller]")]: permite indicar la ruta con la que se accederá al controller. En este caso para consumir cualquiera de los métodos del controlador será necesario escribir la URL: `https://localhost:44355/api/First`
  - ✓ [ApiController]: este atributo permite indicar que la clase será un punto de entrada a nuestra API.
- El controlador tendrá como atributo de clase una lista de objetos Values. Al ser un atributo estático será accesible desde cualquier instancia de FirstController.
- El *primer endpoint* permite obtener la lista de objetos Values. Tiene un decorador [HttpGet] para indicar que el método por el cual se accederá a este punto es mediante **GET.** Con la sentencia:

  ```
  return Ok(lst);
  ```

  Se indica que se va retornar la lista de objetos como un JSON (formato por defecto de respuesta) con un código de estado 200 Ok.
- El *segundo endpoint se utiliza para guardar un objeto Value recibido como parte de la petición.* En este caso el decorador es [HttpPost] para indicar que el método por el cual se accederá es **POST.** Esto implica que la URL es la misma, pero al tener verbos HTTP diferentes el controlador puede responder con métodos distintos. Por último, dentro de este método se valida que el objeto recibido no es nulo. En caso de serlo se genera una respuesta con estado 400. Caso contrario se devuelve una cadena indicando que se agregó exitosamente.

- El tipo de retorno *IActionResult* es apropiado cuando son posibles varios tipos de retorno en una acción. Los tipos ActionResult representan varios códigos de estado HTTP. Cualquier clase no abstracta que se derive de ActionResult califica como un tipo de retorno válido.

¿Qué es POSTMAN?

POSTMAN es una herramienta que permite principalmente crear peticiones sobre APIs de una forma muy sencilla y poder, de esta manera, probar las APIs.

Para instalarte Postman es necesario que se descargue el software desde el área de descargas de https://www.postman.com/downloads/. Están disponibles aplicaciones para *Windows*, *Linux* y *Mac*.

El uso de Postman es gratuito, si bien nos ofrece un par de planes adicionales que serían el *Postman Pro* que nos ofrece más ancho de banda para las pruebas y *Postman Enterprise* que nos permite, entre otras cosas, poder integrar la herramienta en los sistemas de SSO de nuestra empresa.

Algunas características de POSTMAN:

- **Crear Peticiones**, te permite crear y enviar peticiones http a **servicios REST** mediante una interfaz gráfica. Estas peticiones pueden ser guardadas y reproducidas a posteriori.
- **Definir Colecciones**, mediante **Postman** podemos agrupar las APIs en colecciones. En estas colecciones podemos definir el modelo de autentificación de las APIs para que se añada en cada petición. De igual manera podemos ejecutar un conjunto de test, así como definir variables para la colección.
- **Gestionar la Documentación**, genera documentación basada en las API y colecciones que hemos creado en la herramienta. Además, **esta documentación podemos hacerla pública**.
- **Entorno Colaborativo**, permite compartir las API para un equipo entre varias personas. Para ello se apoya en una herramienta de colaborativa en Cloud.
- **Genera código de invocación**, dado un API es capaz de generar el código de invocación para diferentes lenguajes de programación: *C*, *cURL*, *C#*, *Go*, *Java*, *JavaScript*, *NodeJS*, *Objective-C*, *PHP*, *Python*, *Ruby*, *Shell*, *Swift*,…

- **Establecer variables**, con **Postman** podemos crear variables locales y globales que posteriormente utilicemos dentro de nuestras invocaciones o pruebas.
- **Soporta Ciclo Vida API management**, desde **Postman** podemos gestionar el ciclo de vida del API Management, desde la conceptualización del API, la definición del API, el desarrollo del API y la monitorización y mantenimiento del API.
- **Crear mockups**, mediante **Postman** podemos crear un servidor de mockups o sandbox para que se puedan testear nuestras API antes de que estas estén desarrolladas.

Probar la Web API desde POSTMAN

Para probar la API Web desarrollada en el punto anterior vamos a seguir los siguientes pasos:

1) Acceder a la lista de objetos Values mediante GET
   - Seleccionar la opción **APIs** del menú lateral derecho
   - Ingresar la URL https://localhost:44355/api/First en el campo que está junto a la opción **SEND**
   - Seleccionar método **GET** de la lista desplegable
   - Seleccionar opción **SEND**
   - Gráficamente:

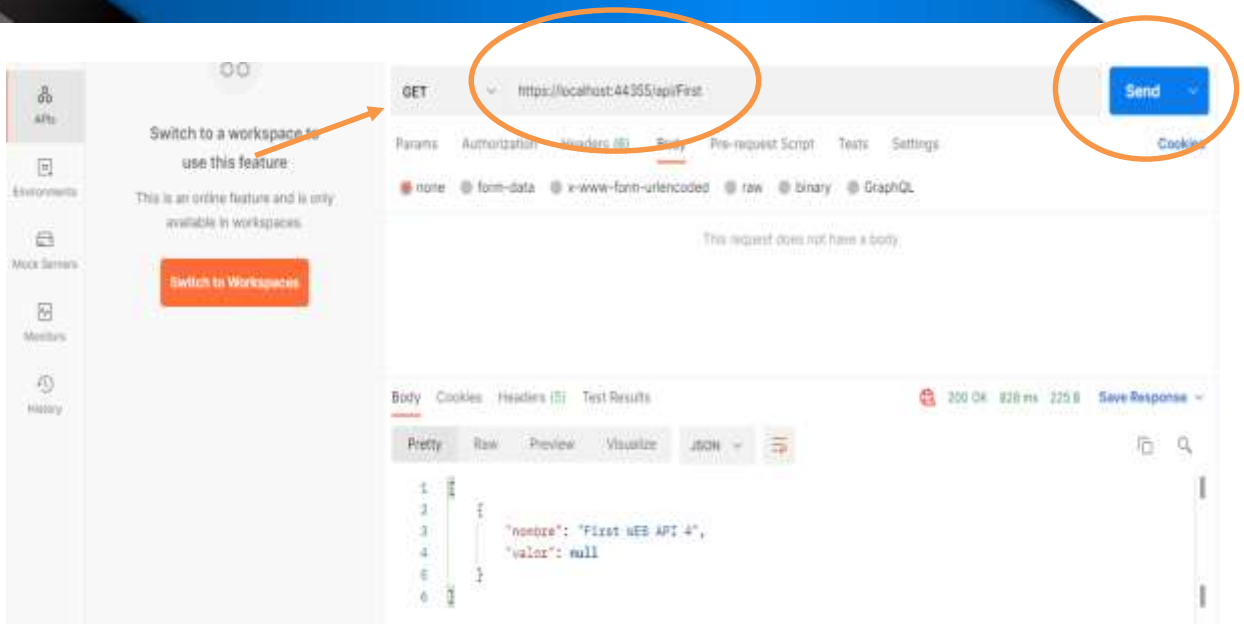

Imagen 16: Elaboración propia

2) Enviar un objeto Values en formato JSON mediante POST

Para enviar un objeto Values necesitamos:

- Con la misma URL cambiar el método HTTP a **POST**
- Seleccionar la pestaña **Body**
- En la parte superior seleccionar la opción **raw**. Esto habilita en un cuadro de texto el ingreso del objeto a enviar en el formato que se desee. Ingresar:

```
{
    "Codigo": 5,
    "Nombre": "Valor 5"
}
```

Las llaves indican comienzo y fin del objeto. Cada property se envía como pareja clave: valor, separada por coma.

- Seleccionar de la lista de formatos: **JSON**
- Seleccionar opción **SEND**
- Gráficamente:

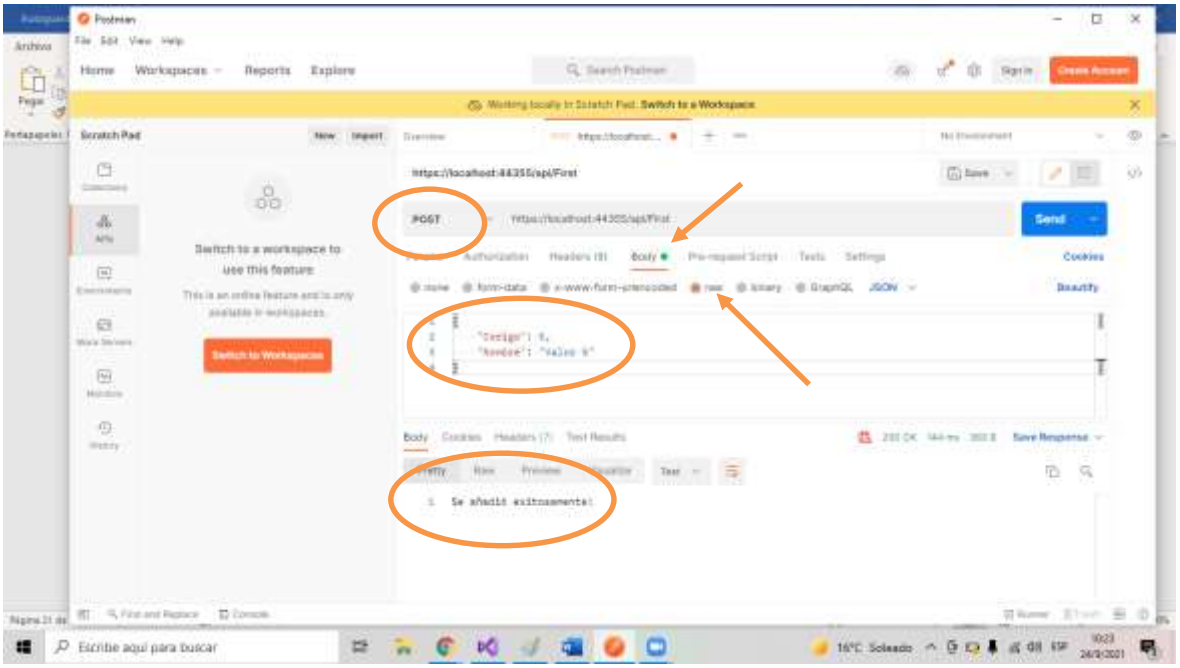

Imagen 17: Elaboración propia

Notar que en **body** de la respuesta se muestra la cadena: “Se añadió exitosamente!”

3) Consumir nuevamente el endpoint de la lista para validar que se agregó el objeto enviado en el punto 2).

Siguiendo los mismos pasos descriptos en 1) consultar la lista de objetos Values nuevamente y validar que la lista tiene un nuevo elemento.

BIBLIOGRAFÍA

Andrew. Tanenbaum, David Wetherall. (2012). Andrew. Tanenbaum, David Wetherall. (2011). Redes de computadoras. Quinta Edición. Pearson Educación, México.

Documentación de .NET. Recuperado de: https://docs.microsoft.com/es-es/dotnet/?view=net-5.0

Generalidades del protocolo HTTP. Recuperado de: https://developer.mozilla.org/es/docs/Web/HTTP

Modelo Cliente-Servidor. Recuperado de: https://blog.infranetworking.com/modelo-cliente-servidor/

Atribución-No Comercial-Sin Derivadas

Se permite descargar esta obra y compartirla, siempre y cuando no sea modificado y/o alterado su contenido, ni se comercialice. Referenciarlo de la siguiente manera: Universidad Tecnológica Nacional Facultad Regional Córdoba (S/D). Material para la Tecnicatura Universitaria en Programación, modalidad virtual, Córdoba, Argentina.