UD3 – GENERACIÓN DE SERVICIOS EN RED

Contenido

1 Servicios y Protocolos Estándares 2
1.1 Protocolo FTP 2
1.2 Protocolo HTTP 6
1.3 Protocolos de Correo Electrónico 13
2 Servicios 18
2.1 Servicios Web (WS) 18
2.2 SOA (Service Oriented Architecture) 19
2.3 SOA y los servicios Web 21
2.4 SOAP 21
2.5 REST 21
2.6 API Web 22
2.7 Microservicios 23
3 Uso de Web APIs 23
4 Framework gRPC 32
4.1 Desarrollo de un servicio gRPC con C# 33
4.2 Desarrollo de un cliente gRPC con C# 36
4.3 Autenticación y autorización con gRPC para .NET 39
4.4 Tipos llamadas en gRPC 41
4.5 Ventajas e inconvenientes de gRPC 42

1 Servicios y Protocolos Estándares

Los **servicios** son programa auxiliares utilizados en un sistema informático para gestionar un conjunto de recursos y prestar su funcionalidad a los usuarios y aplicaciones.

Por ejemplo, cuando enviamos un documento a una impresora conectada en red estamos usando un servicio de impresión, este servicio permite gestionar y compartir dicha impresora.

Dentro de la familia de protocolos TCP/IP encontramos diversos protocolos de servicios en la capa de aplicación, por ejemplo: Telnet, FTP, SMTP, POP, IMAP o HTTP entre otros. Estos protocolos son utilizados para proveer ***servicios de red*** y se basan en el modelo cliente-servidor.

Diremos que un **servicio de red** es aquel proceso que ejecuta una tarea predeterminada y bien definida dirigida a responder las peticiones concretas realizadas desde dispositivos remotos que actúan a modo de clientes.

Algunos servicios como Telnet que ofrecía una emulación del terminal de manera remota han quedado obsoletos y en desuso debido, por un lado, a su falta de seguridad al no ofrecer encriptación de la comunicación (el protocolo SSH ha sustituido a Telnet al ofrecer comunicaciones seguras), y, por otro lado, a la existencia de protocolos que permiten una interfaz visual con tiempos de respuesta adecuados que facilitan la interacción del usuario.

Hoy encontramos muchos servicios basados en aplicaciones web, que aseguran el acceso de cualquier usuario que disponga de un navegador. Las tecnologías web son muy versátiles y permiten también la implementación de servicios que utilizan HTTP como protocolo de transferencia y con formatos XML o JSON para dar estructura a la información transmitida.

1.1 Protocolo FTP

FTP, File Transfer Protocol, es un servicio confiable orientado a conexión que se utiliza para transferir ficheros de una máquina a otra a través de la red. Los sitios FTP son lugares desde los que podemos descargar o enviar ficheros.

Para poder transferir ficheros usando FTP entre dos ordenadores es necesario que uno de ellos tome el papel de servidor y otor el de cliente. El servidor tendrá tener instalado y configurado un software que le permita responder a las peticiones del cliente. El cliente podrá enviar comandos del protocolo FTP al servidor para ejecutar las acciones correspondientes (subir ficheros, bajar ficheros, borrar ficheros, etc.)

FTP dispone de dos tipos fundamentales de acceso; acceso anónimo y acceso autorizado.

FTP utiliza dos conexiones TCP distintas; una de control (por el puerto 21 del servidor) y una de datos (puerto 20 del servidor). La primera se encarga de iniciar y mantener la

comunicación entre cliente y servidor y la segunda exclusivamente para la transferencia de datos.

Existen dos modos de comunicación: activo y pasivo. En el modo activo es el servidor el que establece las conexiones de datos en su puerto 20 haciendo una petición al cliente en un puerto mayor que 1024. Esta configuración no es posible en muchos casos debido a las configuraciones de los firewalls, ya que los clientes no suelen aceptar conexiones de ningún tipo. Para resolver este problema disponemos del modo pasivo. En este caso el servidor indica al cliente el puerto que va a abrir para la conexión y es el cliente el que inicia la comunicación con dicho puerto evitando así las restricciones de configuración de los cortafuegos.

El protocolo FTP permite dos tipos de transferencia: binario y texto que podremos usar dependiendo del tipo de fichero que se vaya a transmitir.

El protocolo FTP no provee ningún tipo de cifrado de comunicación por lo que se deberá usar junto con algún otro protocolo que proporcione seguridad a la comunicación. Así, encontramos SFTP que utiliza FTP sobre un túnel SSH y FTPS, en este caso FTP se ejecuta sobre TLS de forma que la comunicación FTP discurre cifrada.

La siguiente tabla contiene los comandos básicos de FTP.

Comando	**Descripción**
ftp	Accede al intérprete de comandos ftp.
ftp sistema remoto	Establece una conexión ftp a un sistema remoto. Para obtener instrucciones, consulte Cómo abrir una conexión ftp a un sistema remoto.
open	Inicia sesión en el sistema remoto desde el intérprete de comandos.
close	Cierra la sesión del sistema remoto y vuelve al intérprete de comandos.
bye	Sale del intérprete de comandos ftp.
help	Muestra todos los comandos ftp o, si se proporciona un nombre de comando, se describe brevemente lo que hace el comando.
reset	Vuelve a sincronizar la secuenciación de respuesta de comando con el servidor ftp remoto.
ls	Muestra los contenidos del directorio de trabajo remoto.
pwd	Muestra el nombre del directorio de trabajo remoto.
cd	Cambia el directorio de trabajo remoto.
lcd	Cambia el directorio de trabajo local.
mkdir	Crea un directorio en el sistema remoto.
rmdir	Elimina un directorio en el sistema remoto.
get, mget	Copia un archivo (o varios archivos) del directorio de trabajo remoto al directorio de trabajo local.
put, mput	Copia un archivo (o varios archivos) del directorio de trabajo local al directorio de trabajo remoto.

Originalmente, el framework de .NET incluía las clases FtpWebRequest y FtpWebResponse para implementar un cliente del protocolo FTP con .NET. En el siguiente enlace vemos un ejemplo de uso de un cliente FTP desde programa con estas clases: Ejemplo tecnlogía FTP Client

Actualmente, Microsoft no recomienda su uso para nuevos desarrollos por lo que veremos un ejemplo con librería ***FluentFTP*** (enlaces a la librería en GitHub y NuGet).

Para incluir FluentFTP como NuGet en nuestro proyecto debemos ir a *Herramientas* → *Administrador de paquetes NuGet* → *Consola Administrador de Paquetes* y ejecutar la orden que encontramos en FluentFTP NuGet.

Veamos un ejemplo de acceso y descarga de un fichero desde el FTP del instituto:

```
using FluentFTP;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Authentication;
using System.Text;
using System.Threading.Tasks;

class Program
{
	static void Main(string[] args)
	{
		//creamos el objeto cliente del FTP, pasamos el host y las credenciales
		FtpClient client = new FtpClient("servidorifc.iesch.org", "user", "psw");
		//establecemos el modo de encriptación explícito TLS
		//como requiere el ftp del instituto
		client.EncryptionMode = FtpEncryptionMode.Explicit;
		client.SslProtocols = SslProtocols.Tls;
		//establecemos un evento delegado para la validación del certificado
		client.ValidateCertificate += new FtpSslValidation(OnValidateCertificate);
		//conectamos y listamos el directorio raiz del ftp
		client.Connect();
		ListDirectory(client);
		//ofrecemos mover a un subdirectorio
		Console.WriteLine("A qué directorio quieres moverte?");
		string path = Console.ReadLine().Trim();
		if (path.Length > 0)
		{
			//si introducimos el nombre del directorio
			//establecemos dicho subdirectorio como directorio de trabajo
			client.SetWorkingDirectory(path);
			//listamos el contenido del subdirectorio
			ListDirectory(client);
		}
		//ofrecemos descargar un fichero
		Console.WriteLine("Qué fichero quieres descargar?");
		string file = Console.ReadLine().Trim();
		if (file.Length > 0)
		{
		//si introduce el nombre de un fichero lo bajamos a la carpeta de descargas
			client.DownloadFile(@"F:\Users\username\Downloads\" + file, file);
			Console.WriteLine("Hecho");
		}

		Console.ReadKey();
	}
	//mostramos el contenido del directorio de trabajo
	private static void ListDirectory(FtpClient client)
	{
		//obtenemos el listado del contenido
		var list = client.GetListing();
		//recorremos la lista y mostramos por consola
		foreach (FtpListItem it in list)
		{
			Console.WriteLine(it.FullName);
		}
	}
	//evento para validar el certificado del servidor
	static void OnValidateCertificate(FtpClient control, FtpSslValidationEventArgs e)
	{
		//validamos la información del certificado
		//en el caso del ftp del instituto tiene un certificado autofirmado
		//por lo que la validación no podemos hacerla a través de una CA
		//atención: esta validación no sería aceptable en una situación real
		e.Accept = (e.Certificate.Subject == "CN=ftpifc.iesch.org");
	}
}
```

1.2 Protocolo HTTP

El protocolo **HTTP** (HyperText Transfer Protocol) es el estándar usado para establecer la comunicación entre clientes y servidores de un servicio web.

Se trata de un protocolo que puede funcionar tanto sobre UDP como sobre TCP, aunque la mayoría de implementaciones se encuentran realizadas sobre TCP porque ofrece una mayor calidad.

En su momento se diseñó para que los navegadores (clientes web) se conectasen a servidores y descargasen archivos HTML (HiperText Makup Language). Sin embargo, su gran versatilidad ha hecho se haya popularizado su uso para la ejecución de aplicaciones remotas (aplicaciones web).

HTTP es un protocolo basado en texto. El protocolo define que la comunicación se estructura siempre en dos fases, la petición y la respuesta. Los clientes realizan las peticiones y los servidores los vuelven la respuesta. De acuerdo con el protocolo, las peticiones de los clientes deben ser independientes entre sí, por lo que no debería ser necesario que el servidor tenga que recordar las peticiones realizadas con anterioridad para poder responder una. ***El protocolo HTTP no tiene estado***.

Si el servidor al que el cliente envía la petición mantiene levantado el servicio WWW, generará siempre una respuesta con los datos solicitados o con el motivo por el que no se ha podido satisfacer la demanda.

El puerto usado por el protocolo HTTP es el puerto 80, para el protocolo HTTPS (HTTP sobre TLS) se utiliza el puerto 443.

La siguiente tabla contiene los comandos básicos de HTTP.

Comando	**Descripción**
GET	El método GET solicita una representación del recurso especificado. Las solicitudes que usan GET solo deben recuperar datos y no deben tener ningún otro efecto.
HEAD	Pide una respuesta idéntica a la que correspondería a una petición GET, pero en la respuesta no se devuelve el cuerpo. Esto es útil para poder recuperar los metadatos de los encabezados de respuesta, sin tener que transportar todo el contenido.
POST	Envía los datos para que sean procesados por el recurso identificado. Los datos se incluirán en el cuerpo de la petición. Esto puede resultar en la creación de un nuevo recurso o de las actualizaciones de los recursos existentes o ambas cosas.
PUT	Sube, carga o realiza un *upload* de un recurso especificado (archivo o fichero) y es un camino más eficiente ya que POST utiliza un **mensaje multiparte** y el mensaje es decodificado por el servidor. En contraste, el método PUT permite escribir un archivo en una conexión *socket* establecida con el servidor. La desventaja del método PUT es que los servidores de alojamiento compartido no lo tienen habilitado.
DELETE	Borra el recurso especificado.

TRACE	Este método solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición. Se utiliza con fines de depuración y diagnóstico ya que el cliente puede ver lo que llega al servidor y de esta forma ver todo lo que añaden al mensaje los servidores intermedios
OPTIONS	Devuelve los métodos HTTP que el servidor soporta para un URL específico. Esto puede ser utilizado para comprobar la funcionalidad de un servidor web mediante petición en lugar de un recurso específico.
CONNECT	Se utiliza para saber si se tiene acceso a un host, no necesariamente la petición llega al servidor, este método se utiliza principalmente para saber si un proxy nos da acceso a un host bajo condiciones especiales, como por ejemplo flujos de datos bidireccionales encriptadas (como lo requiere TLS/SSL).
PATCH	Su función es la misma que PUT, el cual sobrescribe completamente un recurso. Se utiliza para actualizar, de manera parcial una o varias partes. Está orientado también para el uso con *proxy*

El método **GET** indica al servidor que el cliente pide la información del recurso referenciada por la URL que se encuentra a continuación. Es el método con el que el navegador hará la petición cuando escribimos directamente la dirección en la barra de herramientas. El método GET no usa cuerpo de mensaje. Podemos probar esta sintaxis abriendo una aplicación cliente de tipo Telnet, realizando la conexión a un servicio WWW de algún servidor existente y seguidamente, una vez conectado, hacer algunas peticiones para observar la respuesta del servidor. Recuerda que una vez introducida la última línea de la cabecera hay que confirmar con un nuevo salto de línea para que el servidor sepa que no se enviaremos ningún más dato y que debe procesar la petición recibida.

Veamos un ejemplo de petición y respuesta (copiado de Wikipedia)

```
GET /index.html HTTP/1.1
Host: www.example.com
Referer: www.google.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101
Firefox/45.0
Connection: keep-alive
[Línea en blanco]
```

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html lang="eo">
<head>
<meta charset="utf-8">
<title>Título del sitio</title>
</head>
<body>
<h1>Página principal de tuHost</h1>
(Contenido)
  .
  .
  .
</body>
</html>
```

El método **POST** en cambio, sirve para enviar datos específicos al servidor en el cuerpo del mensaje. Este suele ser el método escogido para enviar los datos de un formulario, para subir un archivo o para enviar cualquier otro dato desde el cliente al servidor. El método admite también los campos opcionales de la cabecera y su uso indica al servidor que tendrá que esperar datos al cuerpo del mensaje. Hay que informar del tamaño en bytes de los datos del cuerpo utilizando el campo de la cabecera llamado Content-Length y del tipo de contenido usando el campo Content-Type con uno de los tipos especificados por la norma MIME. Esto permitirá al servidor saber cuántos bytes tiene que esperar en el cuerpo, una vez pasado el salto de línea de separación posterior a la cabecera. Si los datos del cuerpo provienen de un formulario, irán aparejadas de dos en dos. El primer elemento de la pareja representará el nombre del parámetro y el segundo su valor. Ambos elementos irán separados por un símbolo = y entre parámetro y parámetro (pareja de nombre valor) se intercalará un salto de línea (CR-LF). El tipo de contenido que se deberá especificar es *application/x-www-form-urlencoded* y se debe tener en cuenta las limitaciones de los caracteres especiales como el espacio que habrá que cambiar por su valor numérico (expresado en hexadecimal) precedido del carácter *%*.

Desde mayo de 2015, HTTP va por su versión 2.0 que incluye mejoras en la latencia de los tiempos de respuesta y mejoras en el empaquetado de los datos.

En .NET disponemos de la clase HttpClient para consumir recursos utilizando el protocolo HTTP.

Aunque también existen las clases HttpWebRequest, WebClient para este mismo propósito su uso está obsoleto y no se recomiendan para nuevos desarrollos.

La clase HttpClient proporciona una clase base para enviar solicitudes y recibir respuestes HTTP de un recurso identificado por un URI.

La instancia de la clase *HttpClient* actúa como una sesión para enviar solicitudes HTTP. Cada instancia utiliza su propio grupo de conexiones, aislando sus solicitudes de otras ejecutadas por otras instancias.

HttpClient está diseñada para que se cree una instancia una vez en una aplicación y se vuelva a usar dicha instancia a lo largo de la vida de la aplicación. Podemos hacer esto, por ejemplo, declarando el objeto como estático. La creación de instancias de la clase para cada solicitud agotaría el número de sockets disponibles en cargas pesadas.

La *HttpClient* soporta la palabra clave *await* que proporciona las ventajas del manejo de hilos en segundo plano. Los siguientes métodos de la clase son seguros para subprocesos:

CancelPendingRequests	GetByteArrayAsync	PostAsync
DeleteAsync	GetStreamAsync	PutAsync
GetAsync	GetStringAsync	SendAsync

HttpClient es una API de alto nivel que incluye la funcionalidad de nivel inferior disponible en cada plataforma en la que se vaya a ejecutar.

En el siguiente ejemplo vemos como implementar la misma funcionalidad de los ejemplos anteriores con HttpClient:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

/// <summary>
/// Ejemplo de uso de la clase HttpClient
/// </summary>
public static class SampleHttpClient
{
	/// <summary>
	/// Objeto HttpClient general, se aplica la validación de certificado
	/// </summary>
	private static readonly HttpClient client = new HttpClient();
	/// <summary>
	/// Objeto HttpClient para sitios de confianza en los que nos saltaremos
	/// la validación de certificado
	/// </summary>
	private static readonly HttpClient trustedClient;
	/// <summary>
	/// Manejador que permitirá evitar las validaciones de certificado
	/// </summary>
	private static readonly HttpClientHandler trustedHandler;
	/// <summary>
	/// Listado de direcciones de sitios de confianza
	/// Se suprimirán las validaciones del certificado del servidor
	/// para las direcciones que estén en esta lista
	/// </summary>
	public static List<string> TrustedSites { get; }
	/// <summary>
	/// Constructor estático para inicializar los objetos estáticos
	/// </summary>
	static SampleHttpClient()
	{
		//listado de sitios de confianza
		TrustedSites = new List<string>();
		//manejador de httpclient
		trustedHandler = new System.Net.Http.HttpClientHandler();
		//damos por válido cualquier certificado
		trustedHandler.ServerCertificateCustomValidationCallback
			= delegate { return true; };
		//creamos un cliente que no valida certificados
		trustedClient = new HttpClient(trustedHandler);

	}
```

```
        /// <summary>
        /// Metodo que obtiene el hipertexto de una dirección
        /// </summary>
        /// <returns>string que representa el recurso descargado</returns>
        public static string get(string address)
        {
                //validamos la dirección, si está en la lista de sitios de confianza
                //usamos el cliente que no aplica validaíón de certificado
                HttpClient cli;
                if (TrustedSites.Contains(address))
                {
                        cli = trustedClient;
                }
                else
                {
                        cli = client;
                }
                //obtenemos la respuesta al ser un método asíncrono obtenemos
                // un objeto Task que contiene dentro el valor de la respuesta
                Task<string> task = cli.GetStringAsync(address);
                return task.Result; //devolvemos la repuesta obtenida
        }
}
```

El siguiente código permite probar esta clase:

```
private static void TestHttpClient()
{
        SampleHttpClient.TrustedSites.Add("https://iesch.org");

        while (true)
        {
                Console.WriteLine("Enter the address of the web resource");
                string address = Console.ReadLine();
                Console.WriteLine(SampleHttpClient.get(address));
                Console.ReadKey();
                Console.Clear();
        }
}
```

Veamos un ejemplo de uso del método POST:

```
/// <summary>
/// Método que recoge la respuesta a un método Post
/// </summary>
/// <param name="address">Dirección del servidor</param>
/// <param name="requestURI">Uri del recurso que al que se envia la petición</param>
/// <param name="fields">Campos del formulario a enviar</param>
/// <returns>Devuelve el contenido de la respuesta al método POST</returns>
public static async Task<string> post(string address, string requestURI,
KeyValuePair<string, string>[] fields)
{
	//validamos la dirección, si está en la lista de sitios de confianza
	//usamos el cliente que no aplica validaión de certificado
	HttpClient cli;
	if (TrustedSites.Contains(address))
	{
		cli = trustedClient;
	}
	else
	{
		cli = client;
	}
	//esteblecemos la dirección del servicio
	cli.BaseAddress =  new Uri(address);
	//creamos el contenido del formulario a partir de un
	//array de campos y valores
	FormUrlEncodedContent content = new FormUrlEncodedContent(fields);
	//realizamos la petición post y recogemos la respuesta
	HttpResponseMessage result = await cli.PostAsync(requestURI, content);
	//recogemos el contenido de la respuesta
	Task<string> res = result.Content.ReadAsStringAsync();
	return res.Result;
}
```

Probamos el método anterior con este código:

```
private static void TestHttpClientPost()
{
	KeyValuePair<string,string>[] par = new KeyValuePair<string, string>[6];
	par[0] = new KeyValuePair<string, string>("custname", "Joe");
	par[1] = new KeyValuePair<string, string>("custemail", "j@j.com");
	par[2] = new KeyValuePair<string, string>("custtel", "978978978");
	par[3] = new KeyValuePair<string, string>("topping", "mozzarela");
	par[4] = new KeyValuePair<string, string>("delivery", "21:30");
	par[5] = new KeyValuePair<string, string>("comments", "No llamar al timbre");

	Task<string> task = SampleHttpClient.post("https://httpbin.org", "/post", par);

	Console.WriteLine(task.Result);
	Console.ReadKey();
}
```

1.3 Protocolos de Correo Electrónico

Los principales protocolos de correo electrónico son: SMTP, POP e IMAP.

SMTP, Simple Mail Transfer Protocol, protocolo estándar para el envío de correo electrónico (correo saliente). Funciona con comandos de texto que se envían al servidor SMTP. A cada comando le sigue una respuesta del servidor compuesta por un número y un mensaje descriptivo. A continuación, vemos los principales comandos del protocolo SMTP:

Comando	Parámetros	Descripción
HELO	nombre dominio	Hay que usarla siempre al inicio de la conexión e informa al servidor del dominio desde el que el cliente está conectando
MAIL FROM	Dirección del remitente	Informa al servidor de la dirección del que envía el mensaje
DATA	Mensaje de correo electrónico (cabeceras y cuerpo) a enviar	Para finalizar el mensaje deberán escribir la secuencia formada por un salto de línea un punto y otro salto de línea
QUIT	-	Indica al servidor que el emisor quiere cerrar la conexión

Las especificaciones de este protocolo se definen en la RFC2821.

Los protocolos POP e IMAP corresponden con el correo entrante y se utilizan para acceder al buzón de correo.

POP, Post Office Protocol, proporciona acceso a los mensajes de los servidores SMTP. Actualmente se utiliza la versión 3 (POP3). Las especificaciones de este protocolo se definen en la RFC1939.

IMAP, Internet Message Access Protocol, permite acceder a los mensajes de correo almacenados en los servidores de correo. Permite acceder al usuario desde cualquier equipo que tenga conexión. Las ventajas de IMAP sobre POP son que con IMAP los mensajes permanecen en el servidor y no se borran de éste al descargarlos y además permite la organización de correos en carpetas. En contra, POP consume menos recursos. Las especificaciones de este protocolo se definen en la RFC3501.

Como hemos visto el servidor SMTP funciona enviando mensajes de texto. En cambio, estamos acostumbrados a recibir correos con formato HTML o XML y con archivos adjuntos. Para conseguir esto se utilizan las especificaciones **MIME**, Multipurpose Internet Mail Extensions, estas extensiones van dirigidas al intercambio a través de internet de todo tipo de archivos (texto, imagen, audio, video, etc.) de forma transparente para el usuario, pero utilizando para la comunicación texto en ASCII de 7 bits. Las extensiones MIME van encaminadas a soportar:

- Texto en conjuntos de caracteres distintos a ASCII de 7 bits.
- Archivos adjuntos que no sean de tipo texto.
- Cuerpos de mensajes con múltiples partes (multi-part).
- Información en encabezados con conjuntos de caracteres distintos de ASCII.

Los clientes y servidores de correo convierten automáticamente desde y hacia formato MIME cuando envían o reciben mensajes de correo electrónico.

El espacio de nombres System.Net.Mail contiene clases que se utilizan para enviar correo electrónico a un servidor SMTP (Simple Mail Transfer Protocol, Protocolo simple de transferencia de correo) para la entrega. Dentro de este namespace encontramos la clase SmtpClient, que permite que las aplicaciones envíen correo electrónico usando el protocolo SMTP. Al igual que hemos visto con las clases que existían en .NET para protocolos FTP y HTTP, SmtpClient ahora está obsoleta y Microsoft recomienda no utilizarlas para nuevos desarrollos.

En su lugar propone el uso de la librería MailKit. La forma más fácil de utilizarla en nuestros proyectos es vía NuGet.

Veamos el siguiente ejemplo de envío de un mensaje de correo electrónico: el objeto SmtpClient se utiliza para conectar con el servidor SMTP. Para la definición del mensaje de correo electrónico utilizamos el objeto MimeMessage.

```
using MailKit.Net.Smtp;
using MailKit.Security;
using MimeKit;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class Program
{
	static void Main(string[] args)
	{
		//datos de la cuenta de correo origen
		string address = "address@domain";
		string pwd = "password";
		//creamos un objeto MailboxAddress para la conexión
		MailboxAddress from = new MailboxAddress("User name", address);

		//creamos un objeto mensaje para enviar
		MimeMessage msg = new MimeMessage();
		//cuenta origen
		msg.From.Add(from);
		//cuenta destino
		msg.To.Add(new MailboxAddress("username", "destination address"));
		//asunto
		msg.Subject = "Mail from code";
		//cuerpo del mensaje
		msg.Body = new TextPart("plain") {
			Text = "My first mail from c# code"
		};
		//creamos un cliente para la conexión
		using (SmtpClient cli = new SmtpClient()) {
			//conectamos con el servidor de correo
			//Puerto SMTP: habitualmente 465 (SslOnConnect) o 587 (StartTls)
			cli.Connect("smtp host", 587, SecureSocketOptions.StartTls);
			// Sólo cuando el servidor smtp requiera autenticación
			cli.Authenticate(address, pwd);
			//envío del mensaje
			cli.Send(msg);
			//desconexión
			cli.Disconnect(true);
		}
	 }
}
```

En la siguiente tabla encontramos la información de los servidores SMTP más populares:

Mail Server	SMTP Server (Host)	Port
Gmail	smtp.gmail.com	587
Outlook	smtp.live.com	587
Yahoo Mail	smtp.mail.yahoo.com	465
Yahoo Mail	smtp.mail.yahoo.com	587
Yahoo Mail Plus	plus.smtp.mail.yahoo.com	465
Hotmail	smtp.live.com	465
Office365.com	smtp.office365.com	587
zoho Mail	smtp.zoho.com	465

Habitualmente el puerto 465 se utiliza con una conexión implícita SSL/TLS, mientras que el puerto 587 requiere que el cliente utilice STARTLS para elevar la conexión al protocolo TLS y requiere también que el cliente introduzca usuario y contraseña para autenticarse.

Si utilizas una cuenta de Google para enviar el correo deberás ir “Gestionar tu cuenta de Google” → “Seguridad” y Activar el acceso de aplicaciones poco seguras. Mejor si utilizas una cuenta para pruebas ya que la cuenta que usemos podría ser marcada como generadora de spam (este consejo también para los demás servicios de correo).

Otra opción es utilizar OAuth2 para autenticar tu cuenta de Google, esta opción requiere ciertas condiciones por parte de Google para activarla por lo que no la veremos ahora.

El siguiente ejemplo muestra como incluir un archivo adjunto en un mensaje de correo electrónico: lo primero que tendremos que hacer es crear un contenedor *multipart/mixed* al que luego agregaremos primero el cuerpo del mensaje. Una vez que hayamos agregado el cuerpo, podemos agregar partes *MIME* que contengan el contenido de los archivos que deseamos adjuntar; nos aseguremos que establecemos el valor del encabezado *Content-Disposition* en el archivo adjunto. Probablemente también queramos establecer el parámetro de nombre de archivo en el encabezado *Content-Disposition*, así como el parámetro de nombre en el encabezado *Content-Type*. La forma más conveniente de hacer esto es usar la propiedad *MimePart.FileName* que establecerá ambos parámetros además del valor del encabezado *Content-Disposition* en el archivo adjunto si aún no se ha establecido con otro valor.

```
var message = new MimeMessage ();
message.From.Add (new MailboxAddress ("Joey", "joey@friends.com"));
message.To.Add (new MailboxAddress ("Alice", "alice@wonderland.com"));
message.Subject = "How you doin?";

// create our message text, just like before (except don't set it as the message.Body)
var body = new TextPart ("plain") {
    Text = @"Hey Alice,

What are you up to this weekend? Monica is throwing one of her parties on
Saturday and I was hoping you could make it.

Will you be my +1?

-- Joey
"
};

// create an image attachment for the file located at path
var attachment = new MimePart ("image", "gif") {
    Content = new MimeContent (File.OpenRead (path), ContentEncoding.Default),
    ContentDisposition = new ContentDisposition (ContentDisposition.Attachment),
    ContentTransferEncoding = ContentEncoding.Base64,
    FileName = Path.GetFileName(path)
};

// now create the multipart/mixed container to hold the message text and the
// image attachment
var multipart = new Multipart ("mixed");
multipart.Add (body);
multipart.Add (attachment);

// now set the multipart/mixed as the message body
message.Body = multipart;
```

La librería *MailKit* permite también recibir mensajes de correo mediante los protocolos POP e IMAP y tratar dichos mensajes. No veremos estas funcionalidades ya que es mucho más común que nuestras aplicaciones tengan que enviar correos electrónicos a los usuarios y no tanto que deban automatizar la recepción y tratamiento de correos electrónicos.

2 Servicios

Vamos con “un poco” de teoría y diversos conceptos sobre servicios.

Los modelos de desarrollo han ido evolucionando con el paso de los años. En los años 80 aparecieron los modelos orientados a objetos, en los 90 aparecieron los modelos basados en componentes y en la actualidad han aparecido los modelos orientados a servicios.

Aunque la arquitectura orientada a servicios no es un concepto nuevo (si bien fue descrita por primera vez por Gartner hasta en 1996), sí se ha visto incrementada su presencia en la actualidad, en gran medida debido al aumento de uso de servicios web. Con la llegada de éstos, se ha estandarizado la arquitectura SOA que ha hecho que el desarrollo de software orientado a servicios sea factible. Aunque los servicios web usan con frecuencia SOA, SOA es neutral e independiente de la tecnología utilizada y por tanto no depende de los servicios web.

Un servicio es una representación lógica de una actividad de negocio que tiene un resultado de negocio específico (ejemplo: comprobar el crédito de un cliente, obtener datos de clima, consolidar reportes de perforación).

2.1 Servicios Web (WS)

Un **servicio web** (web service WS) es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet. La interoperabilidad se consigue mediante la adopción de estándares abiertos. Las organizaciones OASIS y W3C son los comités responsables de la arquitectura y reglamentación de los servicios Web.

W3C define un servicio web como:

Un servicio web es un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en WSDL, Web Services Description Language), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes SOAP, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web.

La principal razón para usar servicios Web es que se pueden utilizar con HTTP sobre TCP en el puerto de red 80. Dado que las organizaciones protegen sus redes mediante firewalls (que filtran y bloquean gran parte del tráfico de Internet), cierran casi todos los puertos TCP salvo el 80. Los servicios Web utilizan este puerto, por la simple razón de que no resultan bloqueados. Es importante señalar que los servicios web se pueden utilizar sobre cualquier protocolo, sin embargo, TCP es el más común.

Otra razón por la que los servicios Web son muy prácticos es que pueden aportar gran independencia entre la aplicación que usa el servicio Web y el propio servicio. De esta forma, los cambios a lo largo del tiempo en uno no deben afectar al otro. Esta flexibilidad será cada vez más importante, dado que la tendencia a construir grandes aplicaciones a partir de componentes distribuidos más pequeños es cada día más utilizada.

Veamos a continuación algunos de los estándares empleados por los servicios Web:

- Web Services Protocol Stack: conjunto de servicios y protocolos de los servicios web.
- XML: formato estándar para los datos que se vayan a intercambiar.
- SOAP (Simple Object Access Protocol) o XML-RPC (XML Remote Procedure Call): protocolos sobre los que se establece el intercambio.
- Los datos en XML también pueden enviarse de una aplicación a otra mediante protocolos normales como HTTP, FTP o SMTP.
- WSDL (Web Services Description Language): es el lenguaje de la interfaz pública para los servicios web. Es una descripción basada en XML de los requisitos funcionales necesarios para establecer una comunicación con los servicios web.
- REST (Representational State Transfer): arquitectura que, haciendo uso del protocolo HTTP, proporciona una API que utiliza cada uno de sus métodos (GET, POST, PUT, DELETE, etc.) para poder realizar diferentes operaciones entre la aplicación que ofrece el servicio web y el cliente.

2.2 SOA (Service Oriented Architecture)

La **Arquitectura Orientada a Servicios** (**SOA**, *Service Oriented Architecture*) es un estilo de arquitectura de TI que se apoya en la orientación a servicios.

El estilo de arquitectura SOA se caracteriza por:

- Estar basado en el diseño de servicios que reflejan las actividades del negocio en el mundo real, estas actividades hacen parte de los procesos de negocio de la compañía.
- Representar los servicios utilizando descripciones de negocio para asignarles un contexto de negocio.
- Tener requerimientos de infraestructura específicos y únicos para este tipo de arquitectura, en general se recomienda el uso de estándares abiertos para la interoperabilidad y transparencia en la ubicación de servicios.
- Estar implementada de acuerdo con las condiciones específicas de la arquitectura de TI en cada compañía.
- Requerir un gobierno fuerte sobre las representación e implementación de servicios.
- Requerir un conjunto de pruebas que determinen que es un buen servicio.

La aplicación de la orientación a servicios se divide en 2 grandes etapas:

1. Análisis orientado a servicios.
2. Diseño orientado a servicios. Se siguen 8 principios de diseño que se aplican sobre cada uno de los servicios modelados, estos principios de diseño son:
    - Contrato de servicio estandarizado: Los contratos de servicio cumplen con los mismos estándares de diseño.
    - Bajo acoplamiento: los servicios evitan acoplarse a la tecnología que los implementa y a su vez reducen el acoplamiento impuesto a los consumidores.
    - Abstracción: los contratos presentan la información mínima requerida y la información de los servicios se limita a los expuesto en el contrato.
    - Reusabilidad: los servicios expresan y contienen lógica de negocio independiente del consumidor y su entorno, por lo tanto, se convierten en activos de la empresa.
    - Autonomía: los servicios deben tener un gran control de los recursos tecnológicos sobre los cuales están implementados.
    - Sin estado: el servicio reduce el consumo de servicios al delegar el manejo de estados (sesiones) cuando se requiera.
    - Garantizar su descubrimiento: los servicios cuentan con metadatos que permiten descubrirlos e interpretarlos en términos de negocio.
    - Preparado para ser usado en composiciones: los servicios pueden hacer parte de una composición sin importar el tamaño y complejidad de la misma.

Terminología:

Término	**Definición**
Servicio	Una función sin estado, auto-contenida, que acepta una(s) llamada(s) y devuelve una(s) respuesta(s) mediante una interfaz bien definida. Los servicios pueden también ejecutar unidades discretas de trabajo como serían editar y procesar una transacción. Los servicios no dependen del estado de otras funciones o procesos. La tecnología concreta utilizada para prestar el servicio no es parte de esta definición. Existen servicios asíncronos en los que una solicitud a un servicio crea, por ejemplo, un archivo, y en una segunda solicitud se obtiene ese archivo.
Orquestación	Secuenciar los servicios y proveer la lógica adicional para procesar datos. No incluye la presentación de los datos. Coordinación.
Sin estado	No mantiene ni depende de condición pre-existente alguna. En una SOA los servicios no son dependientes de la condición de ningún otro servicio. Reciben en la llamada toda la información que necesitan para dar una respuesta. Debido a que los servicios son "sin estado", pueden ser secuenciados (orquestados) en numerosas secuencias (algunas veces llamadas tuberías o pipelines) para realizar la lógica del negocio.
Proveedor	La función que brinda un servicio en respuesta a una llamada o petición desde un consumidor.
Consumidor	La función que consume el resultado del servicio provisto por un proveedor

2.3 SOA y los servicios Web

Hay que tener cuidado cuando se manejan estos términos y no confundirlos.

Web Services (WS) engloba varias tecnologías, incluyendo XML, SOAP, WSD entre otros, los cuales permiten construir soluciones de programación para mensajes específicos y para problemas de integración de aplicaciones.

En cambio, SOA es una arquitectura de aplicación en la cual todas las funciones están definidas como servicios independientes con interfaces invocables que pueden ser llamados en secuencias bien definidas para formar los procesos de negocio. SOA incide en que los servicios deben cumplir con una serie de principios de diseño como hemos visto.

2.4 SOAP

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. Este protocolo deriva de un protocolo creado por Dave Winer en 1998, llamado XML-RPC. SOAP fue creado por Microsoft, IBM entre otros. Está actualmente bajo el auspicio de la W3C. Es uno de los protocolos utilizados en los servicios Web.

SOAP es un paradigma de mensajería de una dirección sin estado, que puede ser utilizado para formar protocolos más completos y complejos según las necesidades de las aplicaciones que lo implementan. Puede formar y construir la capa base de una "pila de protocolos de web service", ofreciendo un framework de mensajería básica para la construcción de WS. Este protocolo está basado en XML y se conforma de tres partes:

- Sobre (envelope): define qué hay en el mensaje y cómo procesarlo.
- Conjunto de reglas de codificación para expresar instancias de tipos de datos.
- La convención para representar llamadas a procedimientos y respuestas.

El protocolo SOAP tiene tres características principales:

- Extensibilidad.
- Neutralidad (bajo protocolo de transporte TCP puede ser utilizado sobre cualquier protocolo de aplicación como HTTP, SMTP o JMS).
- Independencia (permite cualquier modelo de programación).

2.5 REST

REST, REpresentational State Transfer, es un estilo de arquitectura de software para la comunicación entre hipermedia distribuidos.

Si bien el término REST se refería originalmente a un conjunto de principios de arquitectura, en la actualidad se usa en el sentido más amplio para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato (XML, JSON, etc.) sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes, como por ejemplo SOAP.

Existen una serie de puntos clave en el diseño que hacen que REST proporcione escalabilidad:

- Un protocolo cliente/servidor sin estado: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST)
- Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE.
- Una sintaxis universal para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.
- El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Un concepto importante en REST es la existencia de recursos (elementos de información), que pueden ser accedidos utilizando un identificador global (URI). Para manipular estos recursos, los componentes de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian representaciones de estos recursos (los ficheros que se descargan y se envían).

2.6 API Web

Una API es una interfaz de programación de aplicaciones (Application Programming Interface). Es un conjunto de rutinas que provee acceso a funciones de un determinado software.

Se publican por las compañías desarrolladoras de software para permitir acceso a características de bajo nivel o propietarias, detallando solamente la forma en que cada rutina debe ser llevada a cabo y la funcionalidad que brinda, sin otorgar información acerca de cómo se lleva a cabo la tarea. Las podemos utilizar para construir nuestras propias aplicaciones sin necesidad de volver a programar funciones ya hechas por otros, reutilizando código que se sabe que está probado y que funciona correctamente.

En la web, las API's son publicadas por los sitios web para brindar la posibilidad de realizar alguna acción o acceder a alguna característica o contenido que el sitio provee. Algunas de las más conocidas son: Google Search, Google Maps, Flickr o Amazon.

La mayoría de las Web API están desarrolladas con SOAP o REST o incluso muchas tanto con SOAP como con REST.

2.7 Microservicios

Una arquitectura de microservicios es un enfoque para desarrollar una aplicación software como una serie de pequeños servicios, cada uno ejecutándose de forma autónoma y comunicándose entre sí, habitualmente, a través de peticiones HTTP a sus API.

Normalmente hay un número mínimo de servicios que gestionan cosas comunes para los demás como el acceso a base de datos, pero cada microservicio es pequeño y corresponde a un área de negocio de la aplicación.

Cada microservicio es independiente al resto y su código debe poder ser desplegado sin afectar a los demás. Incluso cada uno de ellos puede escribirse en un lenguaje de programación diferente, ya que solo exponen la API al resto de microservicios.

Más adelante en esta unidad didáctica veremos cómo programar microservicios con gRPC.

3 Uso de Web APIs

Veamos cómo podemos acceder a una Web API programáticamente. Vamos a utilizar para el ejemplo la API AEMET OpenData

Lo primero que necesitamos es conseguir una clave, para ello vamos a “Obtención de API Key” → “Solicitar”.

Una vez tenemos nuestra API Key podemos ir “Acceso General” y probar algunas de las consultas, por ejemplo, Valores Climatotógicos → Climatologías diarias, Predicción Nacional Hoy. Último elaborado, Predicción Comunidad Autonoma Hoy o Información Nivológica.

Vemos que devuelve 4 campos:

- descripción
- estado
- datos
- metadatos

El “estado” 200 corresponde con una respuesta exitosa, para ver el resultado de la consulta vamos al enlace del campo “datos” y para interpretar estos datos vamos al enlace “metadatos”.

Ahora que ya hemos visto cómo funcionan las peticiones a la API, vamos a “Acceso Desarrolladores” → Entrar. Una vez dentro vamos a “Documentación AEMET OpenData. HATEOAS”. Aquí podemos encontrar los detalles de las llamadas a la REST API. Por ejemplo:

```
GET /api/valores/climatologicos/inventarioestaciones/todasestaciones
GET /api/valores/climatologicos/diarios/datos/fechaini/{fechaIniStr}/fechafin/{fechaFinStr}/estacion/{idema}
```

Encontramos detalles de los parámetros de entrada y de los valores devueltos. Cómo ya hemos visto, las respuestas son del tipo

```
200 {
   descripcion (string),
   estado (integer),
   datos (string),
   metadatos (string)
}
```

Ejemplo de respuesta:

```
{
  "descripcion": "Éxito",
  "estado": 200,
  "datos": "string",
  "metadatos": "string"
}
```

En todos los casos deberemos incluir como parámetro nuestra clave para la API añadiéndola al uri del recurso:

```
/?api_key=mi_clave
```

Con esta información ya podemos ir preparando nuestro código de cliente, para ello podemos ver el ejemplo que nos propone la web AEMET Open Data para el lenguaje cliente que vamos a utilizar en el apartado “Ejemplos de programas cliente”. En nuestro caso, para C#, encontramos:

```
var client = new
RestClient("https://opendata.aemet.es/opendata/api/valores/climatologicos/inventarioesta
ciones/todasestaciones/?api_key=miclave");
var request = new RestRequest(Method.GET);
request.AddHeader("cache-control", "no-cache");
IRestResponse response = client.Execute(request);
```

Este código hace uso de la librería RestSharp, la cual puede ser utilizada en nuestro proyecto a través de los paquetes NuGet.

El enlace de metadatos de las respuestas de la API nos da la información sobre los datos que nos permitirá interpretar correctamente los datos.

Los datos que encontramos en los enlaces de datos de las respuestas suelen venir en dos formatos, como texto plano (por ejemplo, en algunas informaciones de las predicciones meteorológicas) o con estructuras JSON. En este último caso, podemos hacer uso de asistentes como la web https://www.jsonutils.com/ para convertir los datos JSON en objetos que nuestro programa podrá manipular fácilmente.

Vamos a construir una librería que nos permitirá usar los datos de la AEMET en nuestros programas.

La clase *Respuesta* representa una respuesta de la API:

```
//clase que representa la respuesta de la API de AEMET
public class Respuesta
{
	//respuesta exitosa
	public const int OK = 200;

	//descripción de la respuesta
	public string descripcion { get; set; }
	//estado de la respuesta
	//200 = OK
	public int estado { get; set; }
	//enlace a los datos generados, url
	public string datos { get; set; }
	//enlaces a los metadatos de los datos generados, url
	public string metadatos { get; set; }
}
```

La clase *ClienteDatos* permite obtener los datos y metadatos mediante el protocolo http desde las url obtenidas en las respuestas (*datos* y *metadatos*).

```
//clase cliente HTTP
public class ClienteDatos
{
	//objeto estático HttpClient
	//tendremos un sólo cliente para la aplicación
	//para ahorrar recursos
	private static readonly HttpClient cli = new HttpClient();

	//devolvemos los datos obtenidos en las peticiones a la API REST
	//como una cadena de texto
	public static string GetDatos(string uri)
	{
		//petición http para obtener los datos
		Task<string> res = cli.GetStringAsync(uri);
		//devolvemos la cadena de texto
		return res.Result;
	}
}
```

Generamos la clase *Estacion* utilizando la ayuda de https://www.jsonutils.com/ a partir de los datos obtenidos de la función “Inventario Estaciones” de la API.

```
public class Estacion
{
	public string latitud { get; set; }
	public string provincia { get; set; }
	public string altitud { get; set; }
	public string indicativo { get; set; }
	public string nombre { get; set; }
	public string indsinop { get; set; }
	public string longitud { get; set; }
}
```

Para los datos diarios utilizamos la ayuda de Quicktype para generar la clase *ValoresDiarios*:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;

public partial class ValoresDiarios
{
        [JsonProperty("fecha")]
        public DateTimeOffset Fecha { get; set; }

        [JsonProperty("indicativo")]
        public string Indicativo { get; set; }

        [JsonProperty("nombre")]
        public string Nombre { get; set; }

        [JsonProperty("provincia")]
        public string Provincia { get; set; }

        [JsonProperty("altitud")]
        [JsonConverter(typeof(ParseStringConverter))]
        public long Altitud { get; set; }

        [JsonProperty("tmed")]
        public string TemperaturaMedia { get; set; }

        [JsonProperty("prec")]
        public string Precipitacion { get; set; }

        [JsonProperty("tmin")]
        public string TemperaturaMin { get; set; }

        [JsonProperty("horatmin")]
        public string HoraTemperaturaMin { get; set; }

        [JsonProperty("tmax")]
        public string TemperaturaMax { get; set; }

        [JsonProperty("horatmax")]
        public string HoraTemperaturaMax { get; set; }

        [JsonProperty("dir")]
        public string Dir { get; set; }

        [JsonProperty("velmedia")]
        public string VelocidadMedia { get; set; }

        [JsonProperty("racha")]
        public string Racha { get; set; }

        [JsonProperty("horaracha")]
        public string HoraRacha { get; set; }

        [JsonProperty("sol")]
        public string Sol { get; set; }
```

```
        [JsonProperty("presMax")]
        public string PresMax { get; set; }

        [JsonProperty("horaPresMax")]
        public string HoraPresMax { get; set; }

        [JsonProperty("presMin")]
        public string PresMin { get; set; }

        [JsonProperty("horaPresMin")]
        public string HoraPresMin { get; set; }

        public override string ToString()
        {
                return Nombre + ", " + Altitud + "m (" + Provincia + ")\r\n"
                        + Fecha.ToString("dd / MM / yyyy") + "\r\n"
                        + "T Max: " + TemperaturaMax + "\r\n"
                        + "T Min: " + TemperaturaMin + "\r\n"
                        + "T Media: " + TemperaturaMedia + "\r\n"
                        + "Precipitacion: " + Precipitacion;
        }
}

public partial class ValoresDiarios
{
        public static ValoresDiarios[] FromJson(string json) =>
JsonConvert.DeserializeObject<ValoresDiarios[]>(json,
AemetCli.Converter.Settings);
}
```

Esta misma utilidad también genera las siguientes clases utilizadas de forma auxiliar:

```
public static class Serialize
{
	public static string ToJson(this ValoresDiarios[] self) =>
JsonConvert.SerializeObject(self, AemetCli.Converter.Settings);
}

internal static class Converter
{
	public static readonly JsonSerializerSettings Settings = new
JsonSerializerSettings
	{
		MetadataPropertyHandling = MetadataPropertyHandling.Ignore,
		DateParseHandling = DateParseHandling.None,
		Converters =
		{
			new IsoDateTimeConverter { DateTimeStyles =
DateTimeStyles.AssumeUniversal }
		},
	};
}

internal class ParseStringConverter : JsonConverter
{
	public override bool CanConvert(Type t) => t == typeof(long) || t ==
typeof(long?);

	public override object ReadJson(JsonReader reader, Type t, object
existingValue, JsonSerializer serializer)
	{
		if (reader.TokenType == JsonToken.Null) return null;
		var value = serializer.Deserialize<string>(reader);
		long l;
		if (Int64.TryParse(value, out l))
		{
			return l;
		}
		throw new Exception("Cannot unmarshal type long");
	}

	public override void WriteJson(JsonWriter writer, object
untypedValue, JsonSerializer serializer)
	{
		if (untypedValue == null)
		{
			serializer.Serialize(writer, null);
			return;
		}
		var value = (long)untypedValue;
		serializer.Serialize(writer, value.ToString());
		return;
	}

	public static readonly ParseStringConverter Singleton = new
ParseStringConverter();
}
```

La clase principal de la librería es *ClienteAemet* desde la que hacemos las peticiones a la REST API

```
//Cliente que hará las peticiones a la REST API
public class ClienteAemet
{
//dirección de la API
private const string DIR = "https://opendata.aemet.es/opendata";
//cabecera del parámetro API KEY
private const string API_KEY = @"/?api_key=";
//propiedad con la clave obtenida de la API
public static string ApiKey { get; set; }
//propiedad de lectura que genera el parámetro que se incluirá
// en las llamadas a la API
private static string ApiKeyParam { get { return API_KEY + ApiKey; } }
//método que obtiene el listado completo de las estaciones meteorológicas de AEMET
public static List<Estacion> InventarioEstacionesTodas()
{
	//dirección del recurso
	const string INVENTARIO
		= "/api/valores/climatologicos/inventarioestaciones/todasestaciones";
	//uri del recurso
	string uri = DIR + INVENTARIO + ApiKeyParam;
	//lista que se devolverá en la llamada
	List<Estacion> estaciones = new List<Estacion>();
	//establecemos la URL de la llamada
	var client = new RestClient(uri);
	//establecemos el tipo de petición: GET
	var request = new RestRequest(Method.GET);
	//añadimos en el encabezado
	request.AddHeader("cache-control", "no-cache");
	//ejecutamos la petición y obtenemos la respuesta
	IRestResponse response = client.Execute(request);
	//verificamos el estado de la repuesta REST
	if (response.ResponseStatus == ResponseStatus.Completed)
	{
		//si se ha completado obtenemos el contenido de la respuesta
		Respuesta respuesta
	= (Respuesta)SimpleJson.DeserializeObject(response.Content, typeof(Respuesta));
		//verificamos el estado de la respuesta de la API
		if (respuesta.estado == Respuesta.OK)
		{
			//mediante una petición HTTP al enlace recibido
			//obtenemos los datos generados en nuestra petición a la API
			string datos = ClienteDatos.GetDatos(respuesta.datos);
			//convertimos en objetos el array JSON y retornamos la colección
			return JsonConvert.DeserializeObject<List<Estacion>>(datos);
		}
		else
		{
			//lanzamos una excepción en caso de respuesta no existosa de la API
			throw new ApplicationException("Error: " + respuesta.estado.ToString() +
". " + respuesta.descripcion);
		}
	}
	else
	{
		//lanzamos una excepción en caso de respuesta no existosa del servicio REST
		throw new ApplicationException(response.StatusCode.ToString() + " " +
response.StatusDescription);
	}
}
```

```
        //devolvemos un array con los valores diarios obtenidos
        public static ValoresDiarios[] ValoresClimaDiarios(string fechaIni, string fechaFin,
string idema)
        {
                //constantes para la construcción del uri del end-point
                const string FECHA_INI = "/fechaini/";
                const string FECHA_FIN = "/fechafin/";
                const string ESTACION = "/estacion/";
                //dirección del recurso
                const string VALORES_CLIMATOLOGICOS_DIARIOS
                        = "/api/valores/climatologicos/diarios/datos";
                //construcción del uri del recurso
                string uri = DIR + VALORES_CLIMATOLOGICOS_DIARIOS + FECHA_INI + fechaIni
                        + FECHA_FIN + fechaFin + ESTACION + idema + ApiKeyParam;
                //establecemos la URL de la llamada
                var client = new RestClient(uri);
                //establecemos el tipo de petición: GET
                var request = new RestRequest(Method.GET);
                //añadimos en el encabezado
                request.AddHeader("cache-control", "no-cache");
                //ejecutamos la petición y obtenemos la respuesta
                IRestResponse response = client.Execute(request);
                //verificamos el estado de la repuesta REST
                if (response.ResponseStatus == ResponseStatus.Completed)
                {
                        //si se ha completado obtenemos el contenido de la respuesta
                        Respuesta respuesta
        = (Respuesta)SimpleJson.DeserializeObject(response.Content, typeof(Respuesta));
                        //verificamos el estado de la respuesta de la API
                        if (respuesta.estado == Respuesta.OK)
                        {
                                //mediante una petición HTTP al enlace recibido
                                //obtenemos los datos generados en nuestra petición a la API
                                string datos = ClienteDatos.GetDatos(respuesta.datos);
                                //convertimos en objetos el array JSON obtenido
                                //y retornamos el array de objetos
                                return ValoresDiarios.FromJson(datos);
                        }
                        else
                        {
                                //lanzamos una excepción en caso de respuesta no existosa
                throw new ApplicationException("Error: " + respuesta.estado.ToString() + ". " +
respuesta.descripcion);
                        }
                }
                else
                {
                        //lanzamos una excepción en caso de respuesta no existosa del servicio REST
                        throw new ApplicationException(response.StatusCode.ToString() + " " +
response.StatusDescription);
                }
        }
}
```

Por último, la API provee un generador de código de manera que podemos obtener ya una librería que accede a la API de la AEMET sin necesidad de programarla como hemos hecho nosotros (opción “AEMET Codegen”). Este generador de código permite generar clientes para una variedad de lenguajes como Java, Python, PHP, Ruby, HTML, C#, JavaScript o Perl entre otros.

4 Framework gRPC

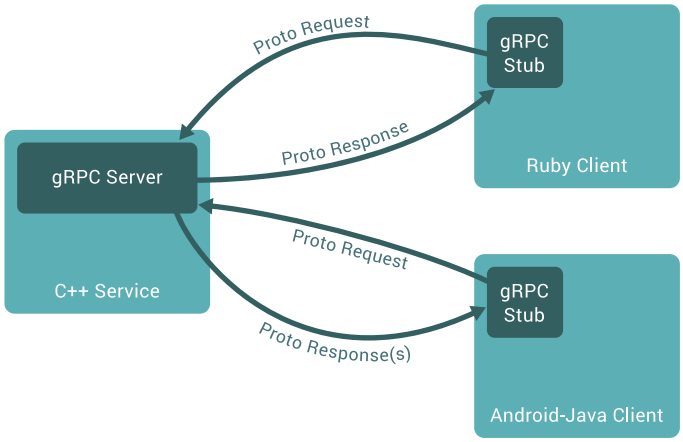
gRPC es un framework de llamada a procedimiento remoto (RPC) de código abierto. Permite que las aplicaciones cliente y servidor se comuniquen de forma transparente y facilita la creación de sistemas conectados. El nombre deriva del nombre del protocol (Remote Procedure Calls) y la g de Google, sponsor del framework.

Los principales escenarios de uso:

- Sistemas distribuidos de baja latencia, altamente escalables.
- Desarrollo de clientes móviles que se comunican con un servidor en la nube.
- Diseño de una aplicación con una interfaz precisa, eficiente e independiente del lenguaje.
- Diseño en capas para permitir la extensión, por ejemplo. autenticación, equilibrio de carga, registro y supervisión, etc

Se puede usar gRPC con los lenguajes de programación Go, C/C++, C#, Java, Python, Dart, Kotlin, Node.js, Objective-C, PHP y Ruby.

Con gRPC, una aplicación cliente puede llamar directamente a un método en una aplicación de servidor en una máquina diferente como si fuera un objeto local, lo que le facilita la creación de aplicaciones y servicios distribuidos. gRPC se basa en la idea de definir un servicio, especificando los métodos que se pueden llamar de forma remota con sus parámetros y tipos de retorno. En el lado del servidor, el servidor implementa esta interfaz y ejecuta un servidor gRPC para manejar las llamadas de los clientes. En el lado del cliente, el cliente tiene un código auxiliar (Stub o Cliente) que proporciona los mismos métodos que el servidor.

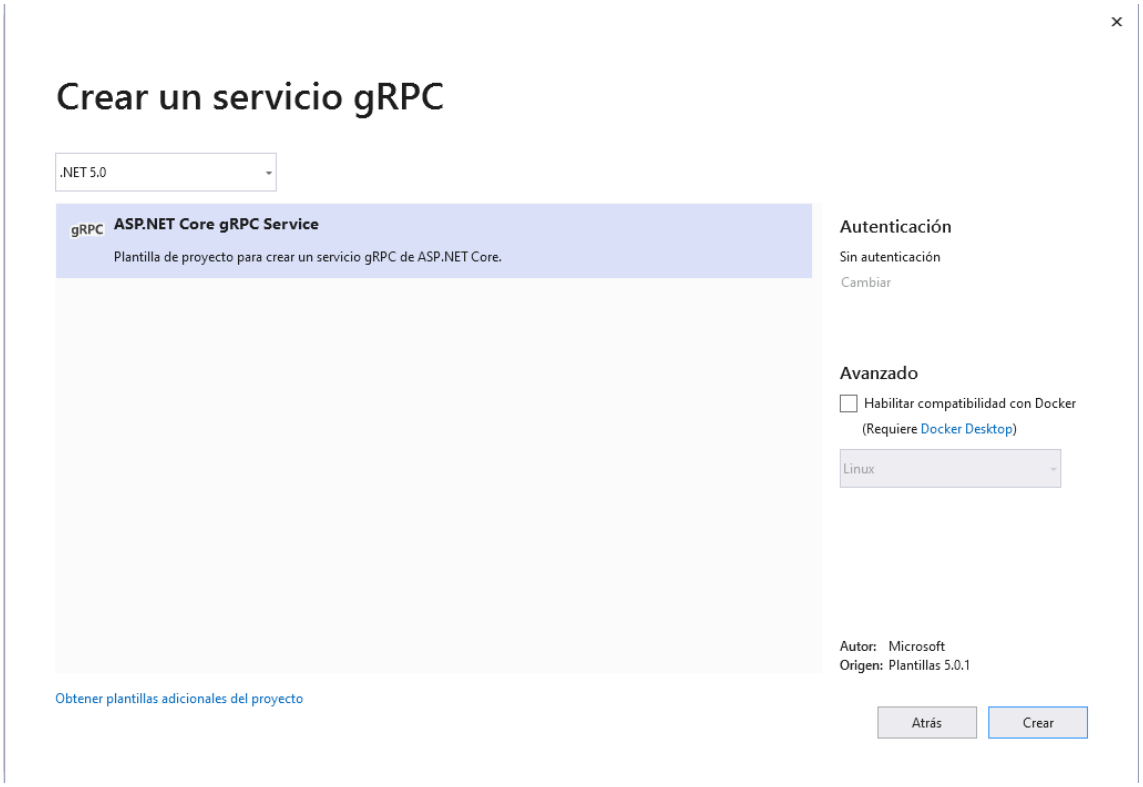


La definición de las llamadas se realiza mediante protobuf, la interfaz de las llamadas se define de acuerdo a dicho protocolo mediante un archivo .proto y los mensajes se serializan con un formato binario mejorando de esa forma el rendimiento frente a los formatos serializados en texto con json o xml.

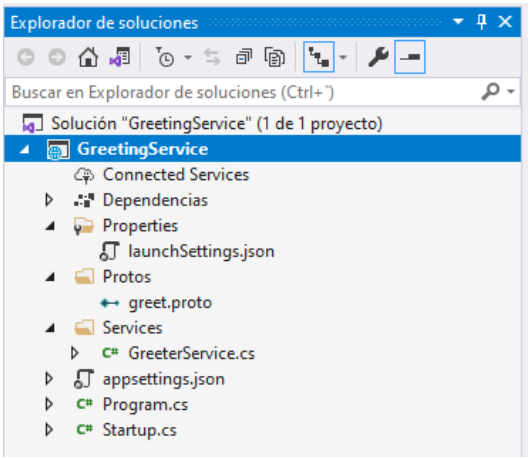
gRPC utiliza HTTP/2 como protocolo de comunicación.

4.1 Desarrollo de un servicio gRPC con C#

Con Visual Studio, creamos un nuevo proyecto, elegimos la plantilla “Servicio gRPC”, elegimos .NET 5.0 y pulsamos *Crear*.



El proyecto creado tiene la siguiente estructura:



El archivo greet.proto define la interfaz del servicio

```
syntax = "proto3";

option csharp_namespace = "GreetingService";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

Puedes encontrar en el siguiente enlace la especificación de la versión 3 de Protocol Buffers.

En este enlace puedes encontrar la equivalencia entre los tipos de protobuf y los tipos de C#.

La clase *GreeterService* implementa el servicio gRPC

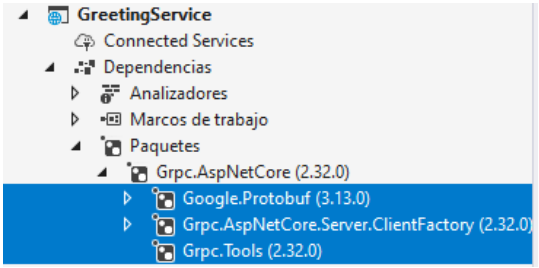
```
using Grpc.Core;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace GreetingService
{
    public class GreeterService : Greeter.GreeterBase
    {
        private readonly ILogger<GreeterService> _logger;
        public GreeterService(ILogger<GreeterService> logger)
        {
            _logger = logger;
        }

        public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
        {
            return Task.FromResult(new HelloReply
            {
                Message = "Hello " + request.Name
            });
        }
    }
}
```

En concreto el método *SayHello* corresponde con la implementación de la llamada definida en el archivo *greet.proto*.

Para usar gRPC en el proyecto se necesitan las referencias a los siguientes paquetes nuget (estas dependencias se crean al crear proyectos con esta plantilla):



El paquete Grpc.Tools se encarga de crear las clases base del servicio (Greeter.GreeterBase). Si modificamos el archivo .proto (o creamos nuevos archivos) se modificarán (o crearán) las clases base de manera que nosotros sólo nos tenemos que encargar de la implementación de las llamadas en el servidor.

En el archivo launchSettings.json encontramos la configuración del servicio, en concreto vemos la dirección y puerto dónde se publicará el servicio (endpoints)

```
{
  "profiles": {
    "GreetingService": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5000;https://localhost:5001",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

La clase Startup se encarga se levantar el servicio:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit
https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGrpcService<GreeterService>();

            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Communication with gRPC endpoints must be made
through a gRPC client. To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909");
            });
        });
    }
```

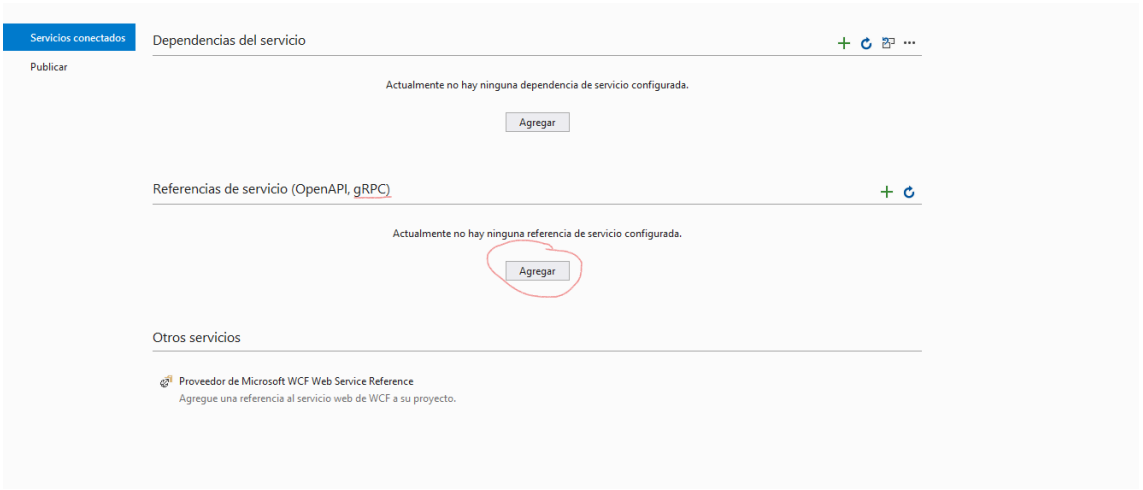
4.2 Desarrollo de un cliente gRPC con C#

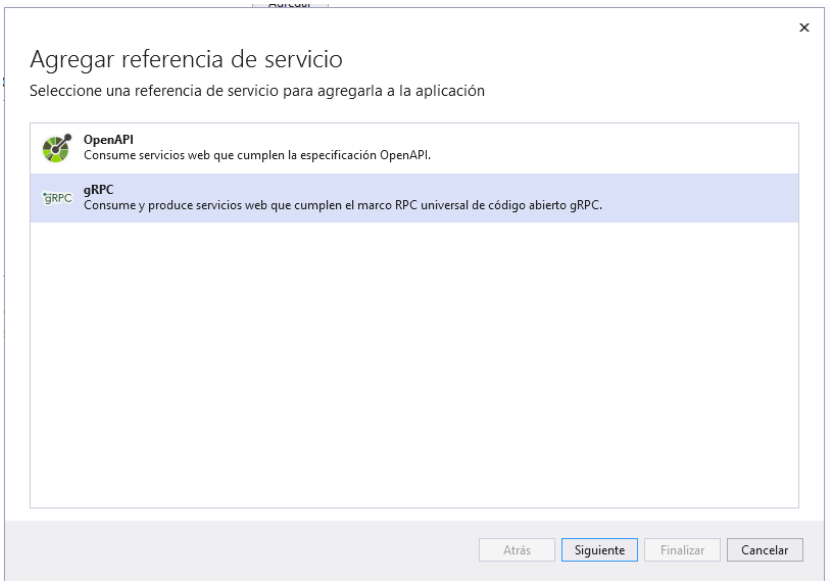
Con Visual Studio, creamos un nuevo proyecto, elegimos la plantilla “Aplicación de consola (.NET Core)”.

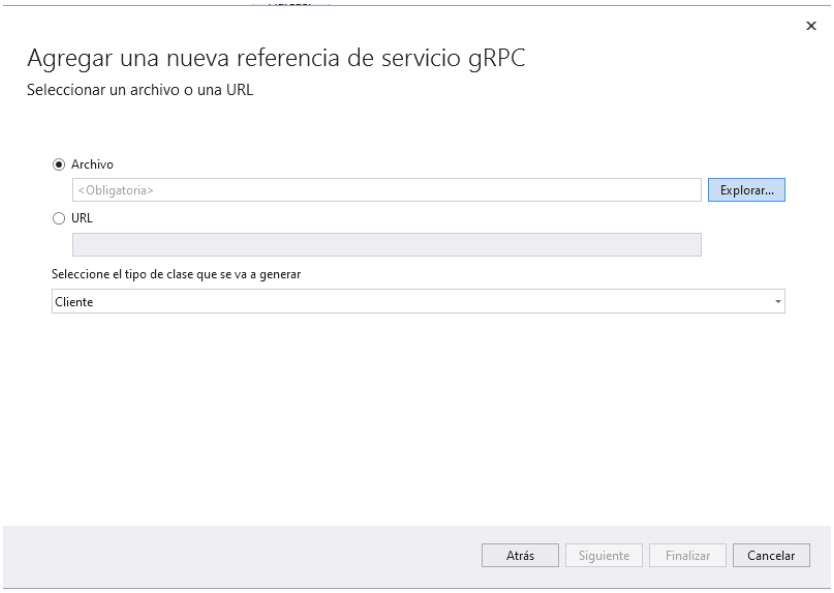
Debemos añadir los siguientes paquetes nuget al proyecto: Grpc, Google.Protobuf y Grpc.Tools. Para ello vamos a “Herramientas” → “Administrador de paquetes NuGet” → “Consola del Administrador de paquetes” y ejecutamos los siguientes comandos:

```
Install-Package Grpc
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

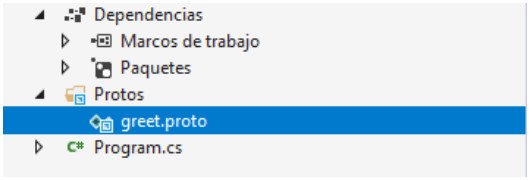
Ahora vamos a agregar una dependencia al servicio gRPC. Como los servicios gRPC vienen definidos por los ficheros .proto, basta con incluir en nuestro proyecto el archivo .proto del servicio al que queremos acceder. Para ello, pinchamos en las Dependencias del proyecto, botón derecho, “Agregar servicio conectado”







Buscamos el archivo proto del servicio y pulsamos finalizar. Una vez añadido, vemos que el archivo .proto aparece en nuestro proyecto de cliente:



Veamos cómo realizar llamadas al servicio. Las herramientas de grpc han creado una clase cliente, GreeterClient, que nos permitirán realizar las llamadas al servicio. Para poder utilizar el este cliente primero debemos abrir un canal grpc al endpoint del servicio. Una vez abierto el canal creamos un objeto cliente (GreeterClient) y a partir de ahí ya podemos crear el objeto que está esperando en la llamada y recibir las repuestas del servidor:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Escribe tu nombre");
        string name = Console.ReadLine();

        // abrir el canal al endpoint del servicio
        using var channel = GrpcChannel.ForAddress("https://localhost:5001");
        // crear el objeto cliente del servicio
        var client = new Greeter.GreeterClient(channel);
        // crear el mensaje de petición a la llamada del servicio
        var req = new HelloRequest { Name = name };
        // llamar al servicio y obtener la respuesta
        var resp = client.SayHello(req);
        // mostrar el mensaje recibido
        Console.WriteLine(resp.Message);
        Console.ReadKey();
    }
}
```

4.3 Autenticación y autorización con gRPC para .NET

Podemos usar la autenticación de ASP.NET Core con gRPC.

Veamos cómo habilitar la **autenticación y autorización por token de portador JWT**.

Para ello deberemos llamar a los métodos *UseAuthentication* y *UseAuthorization* del objeto *IApplicationBuilder* en el método Configure de la clase Startup del servidor tras la llamada al método *UseRouting*.

```
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
```

Una vez que se ha configurado la autenticación, se puede acceder a la información de usuario en los métodos de un servicio gRPC mediante *ServerCallContext*. Para requerir autorización utilizaremos el atributo *Authorize*.

```
[Authorize]
public override Task<SimpleReply> SendVerificationCode(SendVerificationCodeRequest request,
    ServerCallContext context)
{
    var user = context.GetHttpContext().User;
    ...
}
```

En el atributo *Authorize* podríamos indicar también la política de autorización a aplicar (policy), se indicaría de la siguiente forma:

```
[Authorize("MyAuthorizationPolicy")]
```

El cliente puede proporcionar un token de acceso para la autenticación. El servidor valida el token y lo usa para identificar al usuario.

En el servidor, la autenticación de token de portador se configura mediante el middleware de portador JWT. Para definir las opciones de autenticación por jwt y las opciones de autorización de métodos podemos incluir dichas opciones en el método *ConfigureServices* de la clase *Startup*, veamos un ejemplo:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();

    // Opciones de autorización, en este el jwt debe tener el token “id”
    services.AddAuthorization(options =>
    {
        options.AddPolicy(JwtBearerDefaults.AuthenticationScheme, policy =>
        {
            policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
            policy.RequireClaim("id");
        });
    });
    // Opciones de autenticación
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters = new TokenValidationParameters
            {
                ValidateAudience = false,
                ValidateIssuer = false,
                ValidateActor = false,
                ValidateLifetime = true,
                IssuerSigningKey = SigningKey // clave que permite validar el jwt
            };
        });
}
```

El cliente gRPC de .NET deberá enviar el token JWT en el encabezado de las llamadas como usando la colección Metadata:

```
public static (bool ok, string msg) SendVerificationCode()
{
    var headers = new Metadata();
    headers.Add("Authorization", $"Bearer {Token}");
    var request = new SendVerificationCodeRequest();
    var reply = _client.SendVerificationCode(request, headers);
    return (reply.Ok, reply.Message);
}
```

Además de jwt existen otros mecanismos de autenticación admitidos por ASP.NET Core que funcionan con gRPC como: Azure Active Directory, Certificado de cliente, IdentityServidor, OAuth 2.0, OpenID Connect o WS-Federation.

4.4 Tipos llamadas en gRPC

gRPC tiene distintos tipos de métodos:

- **Unario**. El cliente envía una solicitud, el servidor la recibe y envía una respuesta.
- **Streaming de servidor**. El cliente envía una petición y el servidor inicia un stream en el que va enviando una secuencia de respuestas. El cliente irá leyendo los mensajes hasta que no haya más mensajes por leer en el stream. gRPC garantiza el orden de los mensajes de una llamada.
- **Streaming de cliente**. El cliente escribe una secuencia de mensajes y los envía al servidor. Una vez que ha concluido espera a que el servidor lea dichos mensajes y envíe una respuesta. Se garantiza el orden de los mensajes.
- **Streaming bidireccional**. Ambos lados envían una secuencia de mensajes usando un stream. Los dos flujos funcionan de forma independiente, por lo que los clientes y los servidores pueden leer y escribir en el orden que deseen: por ejemplo, el servidor podría esperar a recibir todos los mensajes del cliente antes de escribir sus respuestas, o alternativamente podría leer un mensaje y luego escribir un mensaje o alguna otra combinación de lecturas y escrituras. Se conserva el orden de los mensajes en cada flujo.

4.5 Ventajas e inconvenientes de gRPC

El framwork gRPC ofrece una serie de ventajas sobre otras soluciones de servicios:

- **Rendimiento**. Al usar serialización binaria (Protobuf) y el protocolo HTTP/2 para la comunicación consiguiendo una comunicación más rápida con menos volumen de tráfico.
- **Interoperabilidad**. Hay herramientas y bibliotecas de gRPC para todos los lenguajes de programación y plataformas principales, como .NET, Java, Python, Go, C++, Node.js, Swift, Dart, Ruby y PHP.
- **Facilidad de uso y productividad**. gRPC es una solución completa de RPC. Funciona de forma coherente en varios lenguajes y plataformas. También ofrece herramientas que generan código automáticamente.
- **Streaming**. gRPC tiene un streaming bidireccional completo.
- **Fechas límite y timeout**. gRPC permite cancelar operaciones si se superan tiempos de espera evitando bloquear clientes y / o servidores.
- **Seguridad**. gRPC puede usar TLS y HTTP/2 para proporcionar una conexión cifrada de un extremo a otro entre el cliente y el servidor. La compatibilidad con la autenticación de certificados de cliente aumenta la seguridad en la comunicación cliente - servidor

Este framework también tiene algunos inconvenientes frente a otras soluciones:

- Compatibilidad limitada con los exploradores.
- Los mensajes no son legibles por el usuario al estar serializados en binario.
- Retransmisión de la comunicación en tiempo real. gRPC admite la comunicación en tiempo real a través de streaming, pero no existe el concepto de retransmisión de un mensaje a las conexiones registradas. No se podría enviar una comunicación desde el servidor a todos los clientes activos en un momento dado. En .NET el framework SignalR sería útil para este escenario.