

UD3 – GENERACIÓN DE SERVICIOS EN RED

Contenido

1	Servicios y Protocolos Estándares	1
1.1	Protocolo FTP.....	2
1.2	Protocolo HTTP	5
1.3	Protocolos de Correo Electrónico.....	14
2	Servicios.....	21
2.1	Servicios Web (WS).....	21
2.2	SOA (Service Oriented Architecture).....	22
2.3	SOA y los servicios Web.....	24
2.4	SOAP	24
2.5	REST	24
2.6	API Web	25
2.7	Microservicios.....	26
3	Uso de Web APIs.....	26

1 Servicios y Protocolos Estándares

Los **servicios** son programas 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 otro 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 tecnología 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:

Dependencias necesaria en Maven

```
<dependency>
  <groupId>commons-net</groupId>
  <artifactId>commons-net</artifactId>
  <version>3.9.0</version>
</dependency>
```

```
import org.apache.commons.net.ftp.FTPFile;
import org.apache.commons.net.ftp.FTPReply;
import org.apache.commons.net.ftp.FTPSClient;

import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.cert.X509Certificate;
import java.util.Scanner;

public class Program {

    public static void main(String[] args) {
        FTPSClient ftp = new FTPSClient("TLS", false);

        try {
            // Aceptamos certificado autofirmado (NO seguro en producción)
            ftp.setTrustManager(new X509TrustManager() {
                public void checkClientTrusted(X509Certificate[] xcs,
String string) {}
                public void checkServerTrusted(X509Certificate[] xcs,
String string) {}
                public X509Certificate[] getAcceptedIssuers() { return
null; }
            });

            // Conectamos
            ftp.connect("servidorifc.iesch.org");

            if (!FTPReply.isPositiveCompletion(ftp.getReplyCode())) {
                ftp.disconnect();
                System.out.println("No se pudo conectar al servidor");
                return;
            }

            // Login
            ftp.login("user", "psw");
            ftp.execPBSZ(0);
            ftp.execPROT("P"); // protección de datos
            ftp.enterLocalPassiveMode();

            // Listar directorio raíz
            listDirectory(ftp);

            Scanner sc = new Scanner(System.in);

            // Cambiar de directorio
            System.out.println("¿A qué directorio quieres moverte?");
            String path = sc.nextLine().trim();
            if (!path.isEmpty()) {
                ftp.changeWorkingDirectory(path);
                listDirectory(ftp);
            }

            // Descargar fichero
            System.out.println("¿Qué fichero quieres descargar?");
            String file = sc.nextLine().trim();
            if (!file.isEmpty()) {
                try (FileOutputStream fos =
                    new
FileOutputStream("F:/Users/username/Downloads/" + file)) {
                    ftp.retrieveFile(file, fos);
                    System.out.println("Hecho");
                }
            }
        }
    }
}
```

```

ftp.logout();
    ftp.disconnect();

    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Mostrar contenido del directorio
private static void listDirectory(FTPClient ftp) throws IOException {
    FTPFile[] files = ftp.listFiles();
    for (FTPFile file : files) {
        System.out.println(file.getName());
    }
}
}

```

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 (HyperText Markup Language). Sin embargo, su gran versatilidad ha hecho que 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 les devuelven 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 <i>upload</i> 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 <i>socket</i> 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 <i>proxy</i>

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 respuestas 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*:


```
import javax.net.ssl.*;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.security.SecureRandom;
import java.security.cert.X509Certificate;
import java.util.ArrayList;
import java.util.List;

/**
 * Ejemplo de uso de la clase HttpClient
 */
public class SampleHttpClient {

    /**
     * HttpClient general (valida certificados)
     */
    private static final HttpClient client = HttpClient.newHttpClient();

    /**
     * HttpClient para sitios de confianza (no valida certificados)
     */
    private static final HttpClient trustedClient;

    /**
     * Listado de sitios de confianza
     */
    public static final List<String> TrustedSites;

    /**
     * Inicialización estática
     */
    static {
        TrustedSites = new ArrayList<>();
        trustedClient = createTrustedHttpClient();
    }

    /**
     * Crea un HttpClient que acepta cualquier certificado SSL
     */
    private static HttpClient createTrustedHttpClient() {
        try {
            TrustManager[] trustAllCerts = new TrustManager[]{
                new X509TrustManager() {
                    public void checkClientTrusted(X509Certificate[] certs, String authType) {}
                    public void checkServerTrusted(X509Certificate[] certs, String authType) {}
                    public X509Certificate[] getAcceptedIssuers() { return new X509Certificate[0]; }
                }
            };

            SSLContext sslContext = SSLContext.getInstance("TLS");
            sslContext.init(null, trustAllCerts, new SecureRandom());

            return HttpClient.newBuilder()
                .sslContext(sslContext)
                .build();

        } catch (Exception e) {
            throw new RuntimeException("Error creando HttpClient de confianza", e);
        }
    }
}
```

```
/**
 * Obtiene el contenido de una dirección HTTP
 */
public static String get(String address) {
    try {
        HttpClient cli = TrustedSites.contains(address)
            ? trustedClient
            : client;

        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(address))
            .GET()
            .build();

        HttpResponse<String> response =
            cli.send(request, HttpResponse.BodyHandlers.ofString());

        return response.body();
    } catch (Exception e) {
        throw new RuntimeException("Error realizando la petición HTTP", e);
    }
}
```

El siguiente código permite probar esta clase:

```
import java.util.Scanner;

public class Main {

    private static void testHttpClient() {

        // Añadimos el sitio de confianza

        SampleHttpClient.TrustedSites.add("https://iesch.org");

        Scanner scanner = new Scanner(System.in);

        while (true) {

            System.out.println("Enter the address of the web resource");

            String address = scanner.nextLine();

            // Llamada al HttpClient

            System.out.println(SampleHttpClient.get(address));

            System.out.println("Press ENTER to continue...");

            scanner.nextLine();

            // Limpiar consola (no es estándar en Java, se simula)

            clearConsole();

        }

    }

    // Método auxiliar para limpiar consola

    private static void clearConsole() {

        try {

            if (System.getProperty("os.name").toLowerCase().contains("win")) {

                new ProcessBuilder("cmd", "/c", "cls")

                    .inheritIO()

                    .start()

                    .waitFor();

            } else {

                System.out.print("\033[H\033[2J");

                System.out.flush();

            }

        } catch (Exception e) {

            // Ignorar errores

        }

    }

    public static void main(String[] args) {

        testHttpClient();

    }

}
```

Veamos un ejemplo de uso del método POST:

```
import java.net.URI;

import java.net.URLEncoder;

import java.net.http.HttpClient;

import java.net.http.HttpRequest;

import java.net.http.HttpResponse;

import java.nio.charset.StandardCharsets;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

public class SampleHttpClient {

    // ... aquí va la definición de client, trustedClient y TrustedSites ...

    /**

     * Método que recoge la respuesta a un método POST

     * @param address Dirección del servidor (https://...)

     * @param requestURI URI del recurso al que se envía la petición

     * @param fields Campos del formulario a enviar

     * @return contenido de la respuesta al POST

     */

    public static String post(String address, String requestURI, Map<String, String> fields) {

        try {

            HttpClient cli = TrustedSites.contains(address)

                ? trustedClient

                : client;

            // Convertimos los campos del formulario a application/x-www-form-urlencoded

            String form = fields.entrySet().stream()

                .map(entry -> URLEncoder.encode(entry.getKey(), StandardCharsets.UTF_8) + "=" +

                    URLEncoder.encode(entry.getValue(), StandardCharsets.UTF_8))

                .collect(Collectors.joining("&"));

            HttpRequest request = HttpRequest.newBuilder()

                .uri(URI.create(address + requestURI))

                .header("Content-Type", "application/x-www-form-urlencoded")

                .POST(HttpRequest.BodyPublishers.ofString(form))

                .build();

            HttpResponse<String> response = cli.send(request, HttpResponse.BodyHandlers.ofString());

            return response.body();

        } catch (Exception e) {

            throw new RuntimeException("Error realizando POST a " + address + requestURI, e);

        }

    }

}
```

Probamos el método anterior con este código:

```
import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class Main {

    private static void testHttpClientPost() {

        // Creamos los campos del formulario

        Map<String, String> params = new HashMap<>();

        params.put("custname", "Joe");

        params.put("custemail", "j@j.com");

        params.put("custtel", "978978978");

        params.put("topping", "mozzarella");

        params.put("delivery", "21:30");

        params.put("comments", "No llamar al timbre");


        // Realizamos la petición POST

        String response = SampleHttpClient.post("https://httpbin.org", "/post", params);

        // Mostramos la respuesta

        System.out.println(response);

        System.out.println("Press ENTER to continue...");

        new Scanner(System.in).nextLine();

    }


    public static void main(String[] args) {

        // Prueba del POST

        testHttpClientPost();

    }

}
```

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, Smtplib 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 Smtplib se utiliza para conectar con el servidor SMTP. Para la definición del mensaje de correo electrónico utilizamos el objeto MimeMessage.

Dependencia en Maven

```
<dependency>
  <groupId>com.sun.mail</groupId>
  <artifactId>jakarta.mail</artifactId>
  <version>2.1.1</version>
</dependency>
```



```
import jakarta.mail.*;

import jakarta.mail.internet.*;

import java.util.Properties;

public class SendMailExample {

    public static void main(String[] args) {

        // Datos de la cuenta origen

        String fromEmail = "address@domain";

        String password = "password";

        // Destinatario

        String toEmail = "destination@domain";

        // Propiedades SMTP

        Properties props = new Properties();

        props.put("mail.smtp.host", "smtp host"); // ej: smtp.gmail.com

        props.put("mail.smtp.port", "587");    // STARTTLS

        props.put("mail.smtp.auth", "true");

        props.put("mail.smtp.starttls.enable", "true");

        // Creamos la sesión con autenticación

        Session session = Session.getInstance(props, new Authenticator() {

            @Override

            protected PasswordAuthentication getPasswordAuthentication() {

                return new PasswordAuthentication(fromEmail, password);

            }

        });

        try {

            // Creamos el mensaje

            Message message = new MimeMessage(session);

            message.setFrom(new InternetAddress(fromEmail, "User name"));

            message.setRecipients(Message.RecipientType.TO,

                InternetAddress.parse(toEmail));

            message.setSubject("Mail from code");

            message.setText("My first mail from Java code");

            // Envío del mensaje

            Transport.send(message);

            System.out.println("Correo enviado correctamente.");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

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 aseguramos 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.

```
import jakarta.activation.DataHandler;

import jakarta.activation.DataSource;

import jakarta.activation.FileDataSource;

import jakarta.mail.*;

import jakarta.mail.internet.*;

import java.io.File;

import java.util.Properties;

public class SendMailWithAttachment {

    public static void main(String[] args) {

        String fromEmail = "joey@friends.com";

        String toEmail = "alice@wonderland.com";

        String smtpHost = "smtp host"; // ej: smtp.gmail.com

        String smtpPort = "587";

        String password = "your_password";

        // Ruta del archivo adjunto

        String path = "C:/path/to/image.gif";

        // Configuración SMTP

        Properties props = new Properties();

        props.put("mail.smtp.host", smtpHost);

        props.put("mail.smtp.port", smtpPort);

        props.put("mail.smtp.auth", "true");

        props.put("mail.smtp.starttls.enable", "true");

        Session session = Session.getInstance(props, new Authenticator() {

            protected PasswordAuthentication getPasswordAuthentication() {

                return new PasswordAuthentication(fromEmail, password);

            }

        });

        try {

            // Crear mensaje

            Message message = new MimeMessage(session);

            message.setFrom(new InternetAddress(fromEmail, "Joey"));

            message.setRecipients(Message.RecipientType.TO,

                InternetAddress.parse(toEmail));

            message.setSubject("How you doin?");

            // Cuerpo del mensaje (texto plano)

            MimeBodyPart textPart = new MimeBodyPart();
```

```
String bodyText = "Hey Alice,\n\n" +  
    "What are you up to this weekend? Monica is throwing one of her parties on " +  
    "Saturday and I was hoping you could make it.\n\n" +  
    "Will you be my +1?\n\n-- Joey";  
  
textPart.setText(bodyText);  
  
// Adjuntar imagen  
  
MimeBodyPart attachmentPart = new MimeBodyPart();  
  
DataSource source = new FileDataSource(new File(path));  
  
attachmentPart.setDataHandler(new DataHandler(source));  
  
attachmentPart.setFileName(new File(path).getName());  
  
attachmentPart.setDisposition(Part.ATTACHMENT);  
  
// Crear multipart/mixed  
  
Multipart multipart = new MimeMultipart("mixed");  
  
multipart.addBodyPart(textPart);  
  
multipart.addBodyPart(attachmentPart);  
  
// Asignar multipart como cuerpo del mensaje  
  
message.setContent(multipart);  
  
  
// Enviar mensaje  
  
Transport.send(message);  
  
System.out.println("Correo enviado con adjunto correctamente.");  
  
} catch (Exception e) {  
    e.printStackTrace();  
}  
  
}  
  
}
```

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 lo 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 Climatológicos → Climatologías diarias, Predicción Nacional Hoy. Último elaborado, Predicción Comunidad Autónoma 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 Java encontramos:

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class AemetClient {

    private static final String API_KEY = "miclave";

    private static final String URL =
"https://opendata.aemet.es/opendata/api/valores/climatologicos/inventarioestaciones/toda
s estaciones/?api_key=" + API_KEY;

    public static void main(String[] args) {

        try {

            HttpClient client = HttpClient.newHttpClient();

            HttpRequest request = HttpRequest.newBuilder()

                .uri(URI.create(URL))

                .header("cache-control", "no-cache")

                .GET()

            HttpResponse response = client.send(request, HttpResponse.BodyHandlers.ofString());

            System.out.println(response.body());

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

```
        .build();

        HttpResponseMessage<String> response = client.send(request,
        HttpResponseMessage.BodyHandlers.ofString());

        System.out.println("Código HTTP: " + response.statusCode());
        System.out.println("Cuerpo de la respuesta:");
        System.out.println(response.body());

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

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:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class Respuesta {

    // Respuesta exitosa

    public static final int OK = 200;

    // Descripción de la respuesta
    @JsonProperty("descripcion")
    private String descripcion;

    // Estado de la respuesta (200 = OK)
    @JsonProperty("estado")
    private int estado;

    // Enlace a los datos generados (URL)
    @JsonProperty("datos")
    private String datos;

    // Enlace a los metadatos de los datos generados (URL)
    @JsonProperty("metadatos")
    private String metadatos;

    // Getters y setters

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public int getEstado() {
        return estado;
    }

    public void setEstado(int estado) {
        this.estado = estado;
    }

    public String getDatos() {
        return datos;
    }

    public void setDatos(String datos) {
        this.datos = datos;
    }

    public String getMetadatos() {
        return metadatos;
    }
}
```

```
public void setMetadatos(String metadatos) {  
    this.metadatos = metadatos;  
}  
  
@Override  
public String toString() {  
    return "Respuesta{" +  
        "descripcion=" + descripcion + "\" +  
        ", estado=" + estado +  
        ", datos=" + datos + "\" +  
        ", metadatos=" + metadatos + "\" +  
        '}'  
    }  
}
```

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

```
import java.net.URI;

import java.net.http.HttpClient;

import java.net.http.HttpRequest;

import java.net.http.HttpResponse;


public class ClienteDatos {

    // Objeto HttpClient estático para toda la aplicación
    private static final HttpClient cli = HttpClient.newHttpClient();


    /**
     * Devuelve los datos obtenidos de la API REST como String
     *
     * @param uri URL completa del recurso
     * @return contenido de la respuesta
     */
    public static String getDatos(String uri) {
        try {
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(uri))
                .GET()
                .build();

            HttpResponse<String> response = cli.send(request, HttpResponse.BodyHandlers.ofString());

            return response.body();

        } catch (Exception e) {
            throw new RuntimeException("Error obteniendo datos de " + uri, e);
        }
    }
}
```

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.

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class Estacion {

    @JsonProperty("latitud")
    private String latitud;

    @JsonProperty("provincia")
    private String provincia;

    @JsonProperty("altitud")
    private String altitud;

    @JsonProperty("indicativo")
    private String indicativo;

    @JsonProperty("nombre")
    private String nombre;

    @JsonProperty("indsinop")
    private String indsinop;

    @JsonProperty("longitud")
    private String longitud;

    // Getters y setters

    public String getLatitud() {

        return latitud;

    }

    public void setLatitud(String latitud) {

        this.latitud = latitud;

    }

    public String getProvincia() {

        return provincia;

    }

    public void setProvincia(String provincia) {

        this.provincia = provincia;

    }

    public String getAltitud() {

        return altitud;

    }

    public void setAltitud(String altitud) {

        this.altitud = altitud;

    }

    public String getIndicativo() {

        return indicativo;

    }

}
```



```
public void setIndicativo(String indicativo) {  
    this.indicativo = indicativo;  
}  
  
public String getNombre() {  
    return nombre;  
}  
  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
  
public String getIndsinop() {  
    return indsinop;  
}  
  
public void setIndsinop(String indsinop) {  
    this.indsinop = indsinop;  
}  
  
public String getLongitud() {  
    return longitud;  
}  
  
public void setLongitud(String longitud) {  
    this.longitud = longitud;  
}  
  
@Override  
public String toString() {  
    return "Estacion{" +  
        "latitud=" + latitud + "\" +  
        ", provincia=" + provincia + "\" +  
        ", altitud=" + altitud + "\" +  
        ", indicativo=" + indicativo + "\" +  
        ", nombre=" + nombre + "\" +  
        ", indsinop=" + indsinop + "\" +  
        ", longitud=" + longitud + "\" +  
        '}'  
    }  
}
```

Para los datos diarios utilizamos la ayuda de [Quicktype](#) para generar la clase *ValoresDiarios*:

```
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.time.OffsetDateTime;
import java.time.format.DateTimeFormatter;

public class ValoresDiarios {

    @JsonProperty("fecha")
    private OffsetDateTime fecha;

    @JsonProperty("indicativo")
    private String indicativo;

    @JsonProperty("nombre")
    private String nombre;

    @JsonProperty("provincia")
    private String provincia;

    @JsonProperty("altitud")
    private long altitud;

    @JsonProperty("tmed")
    private String temperaturaMedia;

    @JsonProperty("prec")
    private String precipitacion;

    @JsonProperty("tmin")
    private String temperaturaMin;

    @JsonProperty("horatmin")
    private String horaTemperaturaMin;

    @JsonProperty("tmax")
    private String temperaturaMax;

    @JsonProperty("horatmax")
    private String horaTemperaturaMax;

    @JsonProperty("dir")
    private String dir;

    @JsonProperty("velmedia")
    private String velocidadMedia;

    @JsonProperty("racha")
    private String racha;

    @JsonProperty("horaracha")
    private String horaRacha;

    @JsonProperty("sol")
    private String sol;

    @JsonProperty("presMax")
    private String presMax;

    @JsonProperty("horaPresMax")
    private String horaPresMax;
```

```

@JsonProperty("presMin")
private String presMin;

@JsonProperty("horaPresMin")
private String horaPresMin;

// Getters y setters omitidos por brevedad (puedes generarlos con tu IDE)

@Override
public String toString() {
    String fechaStr = fecha != null ? fecha.format(DateTimeFormatter.ofPattern("dd / MM / yyyy")) : "N/A";
    return nombre + ", " + altitud + "m (" + provincia + ")\n"
        + fechaStr + "\n"
        + "T Max: " + temperaturaMax + "\n"
        + "T Min: " + temperaturaMin + "\n"
        + "T Media: " + temperaturaMedia + "\n"
        + "Precipitacion: " + precipitacion;
}

// Método para deserializar desde JSON a array de ValoresDiarios
public static ValoresDiarios[] fromJson(String json) {
    try {
        ObjectMapper mapper = new ObjectMapper();
        return mapper.readValue(json, ValoresDiarios[].class);
    } catch (Exception e) {
        throw new RuntimeException("Error parseando JSON", e);
    }
}
}

```

Esta misma utilidad también genera las siguientes clases utilizadas de forma auxiliar. Con este ejemplo vamos a ver la serialización de valoresDiarios[] a JSON:

```

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;

public class Serialize {

    private static final ObjectMapper mapper = new ObjectMapper();

    static {
        // Opciones equivalentes a MetadataPropertyHandling.Ignore y DateParseHandling.None
        mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
    }

    // Método equivalente a C# toJson()
    public static String toJson(ValoresDiarios[] valores) {
        try {
            return mapper.writeValueAsString(valores);
        } catch (Exception e) {
            throw new RuntimeException("Error serializando JSON", e);
        }
    }
}

```

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

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.type.TypeReference;

public class ClienteAemet {

    private static final String DIR = "https://opendata.aemet.es/opendata";
    private static final String API_KEY_PARAM_PREFIX = "?api_key=";

    private static final HttpClient httpClient = HttpClient.newHttpClient();
    private static final ObjectMapper mapper = new ObjectMapper();

    // Clave API que se debe asignar antes de hacer peticiones
    public static String apiKey;

    private static String getApiKeyParam() {
        if (apiKey == null || apiKey.isEmpty()) {
            throw new IllegalStateException("API key no asignada");
        }
        return API_KEY_PARAM_PREFIX + apiKey;
    }

    // Obtiene todas las estaciones
    public static List<Estacion> inventarioEstacionesTodas() {
        try {
            String uri = DIR + "/api/valores/climatologicos/inventarioestaciones/todasestaciones" + getApiKeyParam();

            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(uri))
                .header("cache-control", "no-cache")
                .GET()
                .build();

            HttpResponse<String> response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());

            if (response.statusCode() != 200) {
                throw new RuntimeException("Error HTTP: " + response.statusCode());
            }

            // Parseamos la respuesta de la API
            Respuesta resp = mapper.readValue(response.body(), Respuesta.class);

            if (resp.getEstado() != Respuesta.OK) {
                throw new RuntimeException("Error API: " + resp.getEstado() + " - " + resp.getDescripcion());
            }

            // Obtenemos los datos reales desde la URL que devuelve la API
            String datosJson = ClienteDatos.getDatos(resp.getDatos());

            // Convertimos JSON a lista de Estacion
            return mapper.readValue(datosJson, new TypeReference<List<Estacion>>() {});

        } catch (Exception e) {
            throw new RuntimeException("Error obteniendo inventario de estaciones", e);
        }
    }

    // Obtiene los valores diarios de una estación
    public static ValoresDiarios[] valoresClimaDiarios(String fechaIni, String fechaFin, String idema) {

        try {

            String uri = DIR + "/api/valores/climatologicos/diarios/datos"

                + "/fechaini/" + fechaIni

                + "/fechafin/" + fechaFin

                + "/estacion/" + idema

                + getApiKeyParam();

        }

    }

}
```

```
        HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(uri))
        .header("cache-control", "no-cache")
        .GET()
        .build();

    HttpResponse<String> response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());

    if (response.statusCode() != 200) {
        throw new RuntimeException("Error HTTP: " + response.statusCode());
    }

    // Parseamos la respuesta de la API
    Respuesta resp = mapper.readValue(response.body(), Respuesta.class);

    if (resp.getEstado() != Respuesta.OK) {
        throw new RuntimeException("Error API: " + resp.getEstado() + " - " + resp.getDescripcion());
    }

    // Obtenemos los datos reales desde la URL que devuelve la API
    String datosJson = ClienteDatos.getDatos(resp.getDatos());

    // Convertimos JSON a array de ValoresDiarios
    return ValoresDiarios.fromJson(datosJson);

} catch (Exception e) {
    throw new RuntimeException("Error obteniendo valores diarios", e);
}
}
```

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.