UD4 – GENERACIÓN DE SERVICIOS EN RED

Contenido

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.

Algunos métodos de la clase FTPClient (derivada de FTP) que utilizaremos para logearnos, subir, bajar y eliminar ficheros, movernos entre directorios, etc. son:

MÉTODO	UTILIDAD
void connect (String host)	Abrir conexión con el servidor FTP indicado en *host*
int getReplyCode()	Devuelve el valor entero del código de respuesta de la última respuesta FTP
String getReplyCode()	Devuelve el texto completo de la respuesta del servidor FTP
void disconnect()	Cierra la conexión con el servidor FTP y restaura los parámetros de conexión a los valores predeterminados
boolean login (String user, String code)	Inicia sesión en el servidor FTP usando el nombre de usuario y contraseña. Devuelve true si se inicia con éxito, false en caso contrario.
boolean logout()	Sale del servidor FTP
void enterLocalActiveMode()	Se establece el modo de conexión de datos actual en modo activo
void enterLocalPasiveMode()	Se establece el modo de conexión de datos actual en modo pasivo
String printWorkingDirectory()	Devuelve la ruta del directorio de trabajo actual
FTPFile [] listFiles()	Obtiene una lista de ficheros del directorio actual como un array de objetos FTPFile
FTPFile [] listFiles(String path)	Obtiene una lista de ficheros del directorio indicado en el path
String [] listNames()	Obtiene una lista de ficheros del directorio actual como un array de cadenas
FTPFile [] listDirectories()	Obtiene la lista de directorios que se encuentran en el directorio actual
FTPFile [] listDirectories(String parent)	Obtiene una lista de directorios que se encuentran en el directorio indicado en el *parent*)
boolean changeWorkingDirectory(String pathname)	Cambia el directorio de trabajo actual de la sesión FTP al indicado en el *pathname*
boolean changeToParentDirectory()	Cambia al directorio padre del actual
boolean setFileType(int fileType)	Establece el tipo de fichero a transferir: ASCII_FILE_TYPE, BINARY_FILE_TYPE, etc
boolean storeFile(String nombre, InputStream local)	Almacena un fichero en el servidor con el nombre indicado tomando como entrada el InputStream, si el fichero existe lo sobreescribe
boolean retrieveFile(String nombre, OutputStream local)	Recupera un fichero del servidor y lo escribe en el OutputStream dado
boolean deleteFile(String pathname)	Elimina un fichero en el servidor FTP
boolean rename(String antiguo, String nuevo)	Cambia el nombre de un fichero del servidor FTP
boolean removeDirectory (String pathname)	Elimina un directorio en el servidor FTP (si está vacío)
boolean makeDirectory (String pathname)	Crea un nuevo subdirectorio en el directorio actual

La clase FTPFile se utiliza para representar la información acerca de los ficheros almacenados en un servidor FTP. Algunos métodos importantes son:

MÉTODO	UTILIDAD
String getName()	Devuelve el nombre del fichero
long getSize()	Devuelve el tamaño del fichero en bytes
int getType()	Devuelve el tipo de fichero, 0 si es un fichero (FILE_TYPE), 1 un directorio (DIRECTORY_TYPE) y 2 un enlace simbólico (SYMBOLIC_LINK_TYPE)
String getUser()	Devuelve el nombre del usuario propietario del fichero
boolean isDirectory()	Devuelve true si el fichero es un directorio
boolean isFile()	Devuelve true si es un fichero
boolean isSymbolicLink()	Devuelve true si es un enlace simbólico

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*

Otros de los elementos de HTTP es el referente a los códigos de respuesta. Cuando se realiza una petición HTTP a un servidor este genera una respuesta , identifica como un código de tres dígitos, siendo el primero el que determina a nivel global el tipo de respuesta:

Grupo de códigos	Descripción
100-199	Respuesta informativa
200-299	Éxito

300-399	Redirección
400-499	Error del cliente
500-599	Error del servidor

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).

Entre las clases disponibles de manera nativa en Java encontramos ***java.net.HttpURLConnection***, que nos proporciona los mecanismos para gestionar una conexión HTTP. Al heredar de ***URLConnection*** dispone de sus mismos métodos.

Método	Descripción
disconnect	Desconecta la conexión
getResponseCode	Proporciona el código de retorno HTTP enviado por el servidor
getRequestMethod	Proporciona método de petición
getByName	Método estático que proporciona la IP de un host a partir de su nombre
getLocalHost	Método estático que proporciona la IP del host local
getHostAddress	Proporciona la IP del host
getHostName	Proporciona el alias del host
getCanonicalHostName	Proporciona el nombre del host
getInputStream	Proporciona un stream de lectura
getOutputStream	Proporciona un stream de escritura
setRequestProperty	Asigna un valor a una propiedad

Contiene también las **constantes** que representan los códigos de estado del protocolo HTTP, como por ejemplo ***HttpURLConnection.HTTP_OK*** que tiene el valor entero 200 y que significa que la petición se ha realizado correctamente.

La clase **HttpClient** nos permite realizar peticiones HTTP y obtener sus respuestas. Las instancias deben ser creadas a través de un builder u objeto de instanciación.

Algunos métodos son:

Método	Descripción
newBuilder	Crea un objeto builder (objeto de la interfaz HttpClient.Builder)
send	Envia la petición HTTP y devuelve una instancia de HttpResponse. Recibe como parámetro a demás de la petición, un objeto de la clase HttpResponse.
build	Proporciona el objeto HttpClient con la configuración proporcionada.
followsRedirect	Proporciona mecanismos para determinar como debe comportarase la petición frente a redirecciones del servidor.
versión	Especifica la versión del protocolo HTTP.

La clase **HttpRequest** es una clase abstracta que representa una petición HTTP. Las instancias se configuran y crean a través de un **builder**. Este constructor se obtiene a través del método **newBuilder** de la propia clase al que se le indicarán el método HTTP a utilizar, los parámetros de la petición o el tiempo limite de espera.

Los métodos más relevantes de la interfaz **HttpRequest.Builder** son:

Método	Descripción
build	Proporciona el objeto HttpClient con la configuración proporcionada.
DELETE	Asigna el método DELETE al builder.
GET	Asigna el método GET al builder.
POST	Asigna el método POST al builder.
PUT	Asigna el método PUT al builder.
header	Permite añadir un par parámetro-valor a la petición.
headers	Lo mismo que header pero varios pares.
setHeader	Permite asignar un par clave-valor a la petición.
timeout	Permite determinar un tiempo limite.
uri	Asigna la URI a la petición.
versión	Permite especificar la versión del protocolo HTTP.

La clase **HttpResponse** es una interfaz que representa la respuesta de una petición HTTP. Las instancias de esta interfaz no se crean directamente, sino que se crean con

el método ***send*** de la clase HttpClient. El método más importante es statusCode, que nos proporciona ese código de estado de la petición.

Esta clase nos permite obtener instancias de la interfaz **HttpResponse.BodyHandler**. Estas instancias se emplean como parámetro de la llamada al método send y determinan la manera en la que va a procesarse el cuerpo de la respuesta de la petición.

Los métodos más relevantes de **HttpResponse.BodyHandlers**:

Método	Descricpión
ofByteArray	Método estático devuelve un objeto de tipo BodyHandler<byte[]>
ofFile	Método estático devuelve un objeto de tipo BodyHandler<Path>
ofInputStream	Método estático devuelve un objeto de tipo BodyHandler<InputStream>
ofString	Método estático devuelve un objeto de tipo BodyHandler<String>

La interfaz **BodyHandler** solo dispone de un único método, **apply**, que se invoca de manera automática cuando se utiliza. En función del tipo de clase utilizado este método realizará una u otra acción.

Por ejemplo, tras la ejecución de este código, una llamada a response.body() proporciona un **array de bytes**:

```
HttpResponse<byte[]> response = httpClient.send (request,
HttpResponse.BodyHandlers.ofByteArray());
```

En cambio la misma llamada al método body() sobre el objeto response obtenido en la siguiente ejecución, proporciona un **String**:

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

Por último, la ejecución de la siguiente sentencia almacena el fichero indicado en la variable path el contenido del cuerpo de la respuesta.

```
HttpResponse<Path> response = httpClient.send (request,
HttpResponse.BodyHandlers.ofFile(Path.of(path)));
```

Para realizar una petición HTTP se deben realizar los siguientes pasos:

1. Crear el objeto HttpClient, indicando la versión del protocolo, asícomo los datos opcionales, como el comportamiento en caso de que existan redirecciones.

2. Crear el objeto HttpRequest, indicando la URI y los parámetros de la cabecera de la petición.
3. Realizar la petición a través del método send del HttpClient y asignar la respuesta de la petición a un objeto HttpResponse.
4. Procesar la respuesta.

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.

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.

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.

La clase **SMTPClient** (extiende SMTP) encapsula toda la funcionalidad necesaria para enviar ficheros a través de un servidor SMTP. Al igual que todas las clases derivadas de **SocketClient**, antes de hacer cualquier operación es necesario conectarse al servido y una vez finalizada la interacción es necesario desconectarse.

La clase **SMTPReply** (similar a **FTPReply**) almacena un conjunto de constantes para códigos de respuesta SMTP. Para interpretar estos códigos se puede consultar RFC

2821. El método **isPositiveCompletion(int respuesta)** devuelve true si se ha terminado correctamente. Los métodos **getReplyString()** y **getReplyCode()** son métodos de SMTP también.

Métodos	Descripción
SMTPClient()	Constructor por defecto
boolean addRecipient(String address)	Añade la dirección de correo de un destinatario usando el comando RCPT
boolean completePendingCommand()	Este método se utiliza para finalizar la transacción y verificar el éxito o el fracaso de la respuesta del servidor.
boolean login()	Inicia sesión en el servidor SMTP enviando el comando HELO
boolean login (String hostname)	Igual que la anterior, pero envía el nombre del host como argumento.
boolean logout()	Finaliza sesión con el servidor enviando el comando QUIT
Writer sendMessageData()	Envía el comando DATA para después enviar el mensaje de correo. La clase Writer se usará para escribir secuencias de caracteres, como la cabecera y el cuerpo del mensaje.
boolean sendShortMessageData (String message)	Método útil para envío de mensajes cortos
boolean sendSimpleMessage (String remitente, String[] destinatarios, String message)	Método útil para el envío de un correo electrónico corto. Se especifica remitente, destinatarios y el mensaje.
boolean sendSimpleMessage (String remitente, String destinatario, String message)	Igual que el anterior pero un solo destinatario
boolean setSender (String address)	Se especifica la dirección del remitente usando el comando MAIL.
boolan verify(String username)	Comprueba que el nombre de usuario o dirección de correo es válido (envía el comando VRFY por lo que tiene que estar soportado por el servidor)

La clase **SimpleSMTPHeader** se utiliza para la construcción de una cabecera mínima aceptable para el envío de mensajes. El constructor es el siguiente:

```
SimpleSMTPHeader (String from, String to, String subject);
```

Cuyos parámetros son ***from***, para indicar la dirección de correo origen, ***to*** para la dirección de correo destino y ***subject*** para el asunto del mensaje. Además proporciona los siguientes métodos:

Métodos	Descripción
void addCC (String address)	Agrega una dirección de correo electrónico a la lista CC.
void addHeaderField (String headerField, String value)	Agrega un campo de encabezado arbitrario indicado en headerField, con el valor dado
String toString()	Convierte el SimpleSMTPHeader a una cadena que contiene el encabezado con el formato correcto, incluyendo la línea en blanco para separar la cabecera del cuerpo del correo.

La autenticación SMTP se configura con el fin de elevar los niveles de seguridad y eficacia del servicio de correo electrónico y con el objetivo de prevenir que nuestra dirección de correo sea utilizada sin autorización.

Apache Commons Net nos proporciona la clase AuthenticatingSMTPClient con soporte de autenticación al conectarnos al servidor SMTP. La clase SMTPSClient proporciona soporte SMTP sobre el protocolo SSL (Secure Socket Layer). TLS (Transport Layer Security) es el protocolo sucesor de SSL.

STARTTLS es una extensión que nos permite cifrar las comunicaciones entre el cliente y el servidor de tal forma que aunque se use el protocolo SMTP que por defecto no usa cifrado, todos los mensajes enviados y recibidos usandolo están cifrados.

La clase SMTPSClient proporciona varios constructores:

- **SMTPSClient ():** establece el modo seguridad en explicito, en false.
- **SMTPSClient (booleano implícito):** estable el modo de seguridad en modo implícito si la variable es true, y en modo explicito si la variable es false.

En modo implícito la negociación SSL/TLS comienza justo al establecer la conexión. En modo explícito, la negociación SSL/TLS se da cuando el usuario llama al método execTLS() y el servidor acepta el comando.

Ejemplo del modo implícito:

```
SMTPSClient c = new SMTPSClient();
c.connect (“servidorsmtp”, 25);
//resto de comandos
```

Ejemplo del uso explícito:

```
SMTPSClient c = new SMTPSClient();
c. connect(“smtp.gmail.com”, 587);
if (c.execTLS())
//resto de comandos
```

Algunos métodos que usaremos para realizar la autenticación SMTP son:

Métodos	Descripción
void setKeyManager(KeyManager newKeyManager)	Para obtener el certificado para la autenticación se usa la interfaz KeyManager. Con este método se establece la clave para llevar a cabo la autenticación. Las KeyManager se pueden crear usando un KeyManagerFactory.
boolean execTLS()	Ejecuta el comando STARTTLS. La palabra clave STARTTLS se usa para indicarle al cliente SMTP que el servidor SMTP esta en disposición de negociar el uso de TLS. Devuelve true si el comando y la negociación han tenido éxito.

En cuanto a POP3, es el servidor del correo electrónico entrante y se le asigna el puerto 110. Al igual que SMTP funciona con comandos de texto, algunos de ellos son los siguientes:

Comando	Misión
USER login	A la derecha se escribe el login de la cuenta de correo
PASS contraseña	A la derecha si escribe la contraseña
STAT	Muestra el número de mensajes de la cuenta
LIST	Listado de mensajes (numero – tamaño total del mensaje)
RETR número-mensaje	Obtiene el mensaje del número colocado a la derecha
DELE número-mensaje	Borra el mensaje indicado
QUIT	Cierra conexión
TOP número-mensaje n	Muestra las ‘n’ primeras líneas del mensaje indicado

La clase **POP3Client** implementa el lado cliente del protocolo POP3 de Internet definido en la RFC 1939, **POP3SClient** es POP3Client pero con soporte SSL y **POP3MessageInfo** se utiliza para devolver información acerca de los mensajes almacenados en el servidor POP3.

Los métodos que se suelen emplear son los siguientes:

Métodos	Descripción
boolean deleteMessage(int messageID)	Elimina el mensaje con número messageID del servidor POP3. Devuelve true si la operación se realiza correctamente.
POP3MessageInfo listMessage(int messageID)	Lista el mensaje indicado en el parámetro
POP3MessageInfo[] listMessages() **POP3MessageInfo listUniqueIdentifier (int messageID)**	Obtiene un array con información de todos los mensajes
boolean login(String username, String password)	Obtiene la lista de un único mensaje

boolean logout()	Inicia session en el servidor POP3 enviando el nombre de usuario y la clave. Devuelve true si la operación se ha realizado correctamente.
Reader retrieveMessage(int messageID)	Recupera el mensaje con número messageID del servidor POP3.
Reader retrieveMessageTop (int messageID, int numLines)	Igual que el anterior pero solo el número de líneas especificadas en el paramentro numLines. Para recuperar la cabecera, numLines debe ser 0.

La clase **POP3SClient** presenta varios constructores, pero usaremos los básicos:

- El constructor sin parámetros, modo explícito
- El constructor con el parámetro booleano POP3SClient (boolean implicit)

La negociación ocurre **igual** que en la clase **SMTPSClient()**, en el segundo constructor comienza justo después de establecer la conexión, mientras que en el primero es necesario que el usuario llame a **execTLS()** y el servidor acepte el comando.

La clase **POP3MessageInfo** se utiliza para devolver información acerca de los mensajes almacenados en el servidor POP3. Sus campos (identifier, number y size) se utilizan para referirse a cosas ligeramente diferentes dependiendo de la información que se devuelve:

- En respuesta a un **comando de estado**, *number* contiene el número de mensajes en el buzón de correo, *size* contiene el tamaño del buzón de correo en bytes, y el campo *identifier* es nulo.
- En respuesta a **una lista de mensajes**, *number* contiene el número de mensaje, *size* contiene el tamaño del mensaje en bytes, e *identifier* es nulo.
- En respuesta a **una lista de un único mensaje**, *number* contiene el número de mensaje, *size* no está definido, e *identifier* contiene el identificador único del mensaje.

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.