Network Services

Network services consist of applications serving protocols such as **SSH**, **SMTP**, **DNS**, **Telnet**, **FTP**, etc. These services are found throughout networks and provide remote interactive access, file transfer, and critical administration functions for system operation. Network services are sometimes exposed to the Internet to support business functions. In addition, network equipment and server configurations leave network services enabled by default.

SMTP

A **Simple Mail Transfer Protocol** (**SMTP) server** refers to a **mail server** that **forwards emails** from a **sender** to **one** or **more recipients** in accordance with network protocol regulations across the Internet. One important function of the SMTP mail server is to prevent spam using authentication mechanisms that only allow authorised users to deliver emails. To enable this, most modern mail servers support the protocol extension ESMTP with SMTP-Auth.

Like all servers, an SMTP server is an application that provides a service to other applications within a network, called clients. Specifically, an SMTP server handles the sending, receiving, and relaying (transmission of email messages from one mail server to another) of email.

Once an SMTP server is established, email clients can connect to and communicate with it. When the user clicks send on an email message, the email client opens an SMTP connection to the server so it can send.

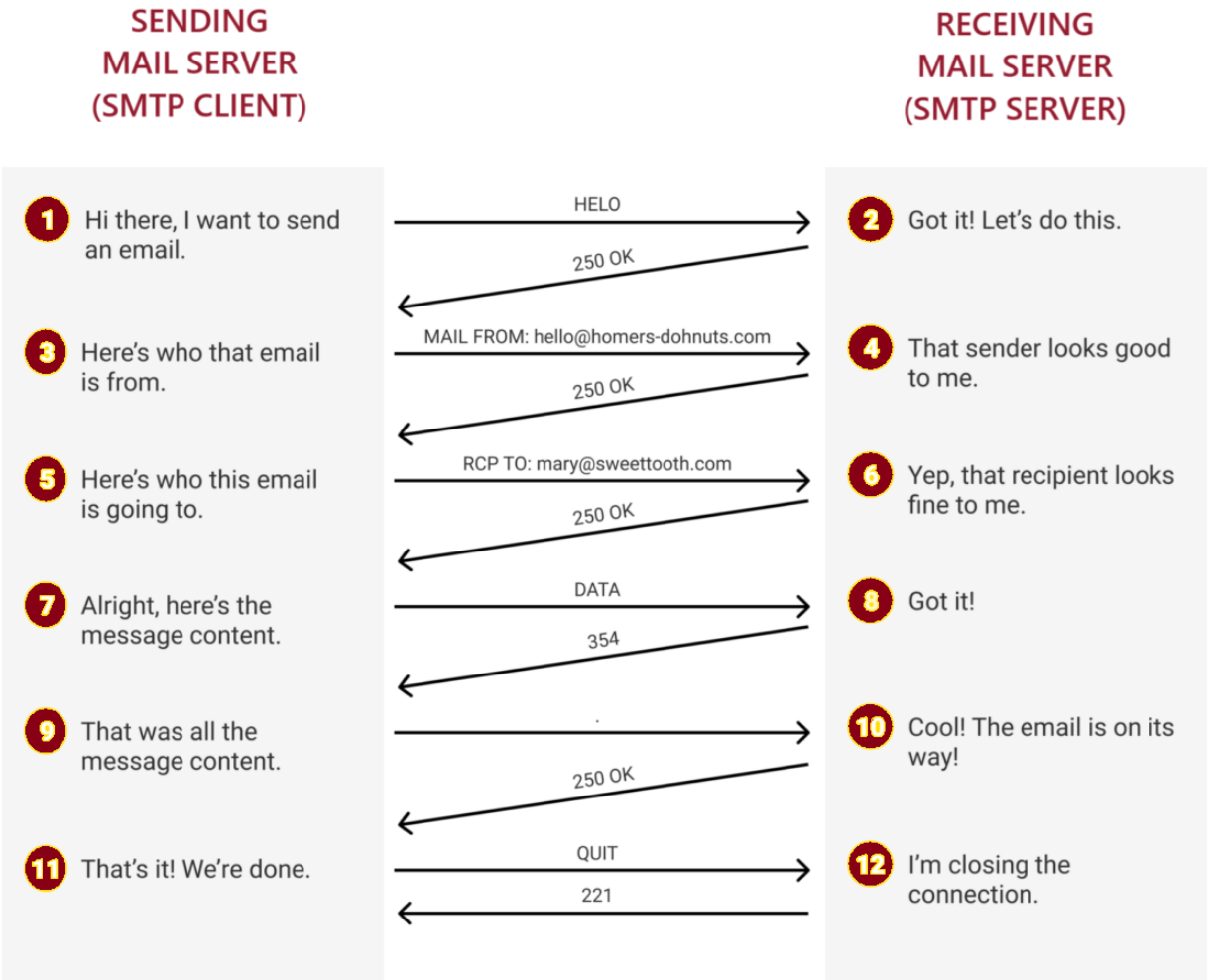
The SMTP client uses commands to tell the server what to do and transfer data, like the sender's email address, the recipient's email address, and the email's content.

Basic SMTP commands

SMTP commands are a set of codes that power the transmission of email messages between servers. Here are the basic SMTP commands:

- **HELO or EHLO**. This is a crucial command for beginning the entire email sending process. The email client is identifying itself to the SMTP server. It is the beginning of a conversation and usually involves the server sending a HELO command back complete with its domain name/IP address.
- **MAIL FROM**. Following the identification command, the sender will share code that specifies who the mail is from. This outlines the email address and tells the SMTP server that a new transaction is about to start. From here, the server resets everything and is ready to accept the email address. Once accepted, it will reply with a 250 OK reply code.

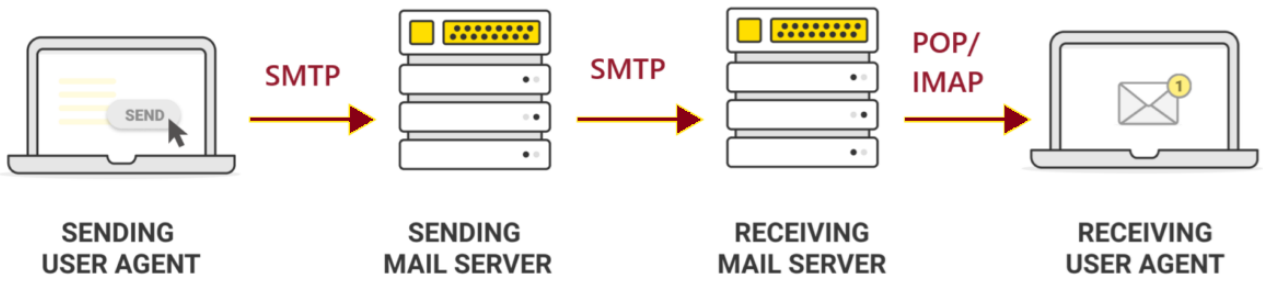
- **RCPT TO**. The next command follows the 250 OK reply code identifying who the email is being sent to. Again, the SMTP server responds with the same code, at which point another RCPT TO command can be sent with a different recipient's email address. This can go back and forth as meany times as required depending on how many people will receive the email.
- **DATA**. This triggers the transfer of data between the client and the server. All of the message contents will be moved to the SMTP server, which will respond with a 345 reply code. The contents of the messages are transferred to the server, where a single dot is sent in a line by itself to signal the end of the message. If accepted and ready for delivery, the server sends another 250 OK code. At this point, the message is on its way to the recipients.
- **QUIT**. When the email has been sent, the client sends the QUIT command to the server. If it has been successfully closed, the server will reply with a 221 code.
- **RSET**. This command is sent to the server when the mail transaction needs to be aborted. It doesn't close the connection, but it does reset everything and remove all previous data about the email and the parties involved. You will commonly use this when there has been an error, like inputting the wrong recipient information, and the process needs to be restarted.



There are other SMTP commands that handle authentication and enhance security, such as AUTH and STARTTLS.

POP and IMAP are the other two most common email protocols in use.

The crucial difference between these protocols is that SMTP is the only protocol for sending email from one unknown mail server to another. POP and IMAP are protocols for receiving mail for the recipient from their own mail server. So, POP and IMAP limit the transfer of mail to verified mail servers only. They can't be used for communication outside of your own networks.



Default email ports

Email ports are communication endpoints that define how a message should be transmitted. That includes whether a message should be encrypted and exchanged securely.

To establish a connection between an email client and an email server, the IP address of the latter and a port number are required. These attributes are assigned by **IANA** (Internet Assigned Numbers Authority).

Each protocol has its own port numbers to connect through, and each port supports a different type of encryption.

The available SMTP ports are four and each of them underlies a different type of encryption for mail sending.

- **25**. This port serves to send messages in plain text, although if the mail server supports it, it can be encrypted with TLS. Therefore, many internet service providers block it, as it represents a security risk.
- **2525**. This port is an alternative to the SMTP port 25 and can be encrypted over TLS.
- **587.** This is the port IANA registered as the secure SMTP port, and it requires an explicit TLS connection. However, if the email server does not support TLS, the message will be sent in plain text.
- **465**. This port works over an implicit SSL connection and if the server does not support it, the operation will be aborted.

Java Mail API

JavaMail is the former name of Jakarta Mail, so the two represent the same software.

Jakarta Mail is an **API** for sending and receiving emails via **SMTP**, **POP3**, as well as **IMAP** and is the most popular option that also supports both TLS and SSL authentication.

Angus Mail is a program created by Eclipse that follows the rules set out by the **Jakarta Mail API** specification.

You can find the **angus-mail.jar** file in the Maven repository and add it to your environment with Maven dependencies.

```
<dependencies>
    <dependency>
        <groupId>org.eclipse.angus</groupId>
        <artifactId>angus-mail</artifactId>
        <version>2.0.3</version>
    </dependency>
</dependencies>
```

Classes and methods

Let's go over some Angus Mail basics you should be familiar with to build and send messages.

The **Session** class, which is not subclassed, is a multi-threaded object that acts as the connection factory for the Jakarta Mail API. Apart from collecting the API's properties and defaults, it is responsible for configuration settings and authentication.

To get an instance of the Session class, you can call either of the following two methods:

- **getDefaultInstance()**, which returns the default session.
- **getInstance()**, which returns a new session.

The **Message** class is abstract, meaning that in order to create a message object, you need to use one of its subclasses capable of implementing the message.

For **Internet** email messages that conform to the **RFC 822** (Standard for the format of Arpa Internet text messages) and **MIME** (Multiporpose Internet mails extensions) standards, that subclass is **MimeMessage**.

Some of the commonly used **methods** of the **MimeMessage** class are:

- **setFrom(Address addresses)**. It is used to set the "From" header field.
- **setRecipients(Message.RecipientType type, String addresses)**. It is used to set the stated recipient type to the provided addresses. The possible defined recipient types are "TO" (Message.RecipientType.TO), "CC" (Message.RecipientType.CC) and "BCC" (Message.RecipientType.BCC).
- **setSubject(String subject)**. It is used to set the email's subject header field.

- **setText(String text)**. It is used to set the provided String as the email's content, using MIME type of "text/plain".
- **setContent(Object message, String contentType)**. It is used to set the email's content using MIME type of "text/html".

The **Address** class is an abstract class that models the addresses (To and From addresses) in an email message. Its subclasses support the actual implementations. Usually, it's Jakarta.mail.internet.InternetAddress subclass, which denotes an Internet email address, is commonly used.

The **Authenticator** class is an abstract class that is used to get authentication to access the mail server resources, often by requiring the user's information. Its **PasswordAuthentication** subclass is commonly used.

The **Transport** class is an abstract class that uses the SMTP protocol for submitting and transporting email messages.

There are four steps to send email using **Angus Mail**. They are as follows:

- **Configure the SMTP server** details using the Java **Properties** object.
- **Create a Session object** by calling the getInstance() method.
- **Compose the message**. To do this, start by passing the created session object in the **MimeMessage** class constructor.

  Set the **From**, **To** and **Subject** fields for the email message.

  Use the **setText()** method to set the content of the email message.
- **Send the message.** Use the **Transport** object to send the email message.

Properties

Most Java applications need to use **properties** at some point, generally to store simple parameters as key-value pairs, outside of compiled code.

A Java **properties file** can be used to store project configuration data or settings.

The file extension used to store the configurable parameters of an application is **.properties**. They can also be used for storing strings for internationalisation and localisation.

Each line in a **.properties** file normally stores a single property.

You can create the properties directly in the code:

```
Properties prop = new Properties();
prop.put("mail.smtp.host", "smtp.gmail.com");
prop.put("mail.smtp.port", "587");
prop.put("mail.smtp.auth", "true");
prop.put("mail.smtp.starttls.enable", "true"); //TLS
```

But it is recommended to use a file instead.

Some of the **properties** to use with **SMTP servers** are the following:

- **mail.smtp.host**. Host of the SMTP server
- **mail.smtp.port**. Port
- **mail.smtp.auth**. Indicates that authentication is required
- **mail.smtp.startttl.enable**. TLS enabled or not.

The path to a properties file in a **Maven project** is: **src/main/resources**.

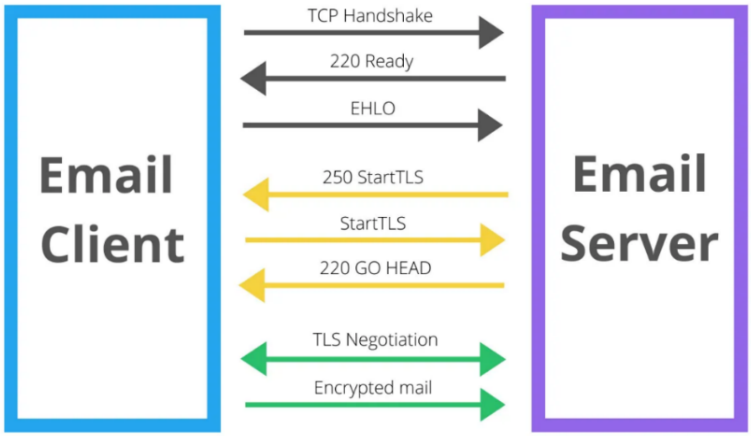
A **properties file** can have the following content:

```
mail.smtp.host=smtp.gmail.com
mail.smtp.port=587
mail.smtp.auth=true
mail.smtp.starttls.enable=true
```

To access the properties file using Java in a Maven project, we can do:

```
try (InputStream input = new FileInputStream("src/main/resources/smtp.properties")) {
    Properties prop = new Properties();
    // load a properties file
    prop.load(input);
    // get the property value and print it out
    System.out.println(prop.getProperty("mail.smtp.host"));
    System.out.println(prop.getProperty("mail.smtp.port"));
    System.out.println(prop.getProperty("mail.smtp.auth"));
    System.out.println(prop.getProperty("mail.smtp.starttls.enalbe"));
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Most of the SMTP servers use some sort of **authentication,** such as **TLS** or **SSL** authentication.



SSL (Secure Socket Layer) is an old protocol deprecated in favour of **TLS (Transport Layer Security)**.

TLS is a protocol for the secure transmission of data based on SSLv3. It offers **confidentiality, integrity, and authentication**. This means:

- **Confidentiality**: hides the content of the messages
- **Integrity**: detects when the messages have been tampered with
- **Authentication**: ensures that whoever is sending them is who he says he is

Additionally, it detects missing and duplicated messages.

The **PasswordAuthentication** class can be used to store the **user credentials**.

```
//create Authenticator object to pass in Session.getInstance argument
Authenticator auth = new Authenticator() {
    //override the getPasswordAuthentication method
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(email, password);
    }
};
Session session = Session.getInstance(props, auth);
```

Mailslurper SMTP server

MailSlurper is a 100% free and **open-source mail server**. It's a simple and easy setup free SMTP server for testing during software development. MailSlurper is useful for individual developers or small teams writing mail-enabled applications that wish to test email functionality without the risk or hassle of installing and configuring a full-blown email server.

Example

```
import jakarta.mail.*;
import jakarta.mail.internet.InternetAddress;
import jakarta.mail.internet.MimeMessage;
import java.io.UnsupportedEncodingException;
import java.util.Properties;
public class SendEmailMailslurper {
    public static void main(String[] args) {
        final String username = "user";
        final String password = "user";
        String sender = "user.testing@yopmail.com";
        String receiver="someone@gmail.com";
        Properties props = new Properties();
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.starttls.enable", "true");
        props.put("mail.smtp.host", "localhost");
        props.put("mail.smtp.port", "2500");
        Session session = Session.getInstance(props,
                new jakarta.mail.Authenticator() {
                  protected PasswordAuthentication getPasswordAuthentication() {
                        return new PasswordAuthentication(username, password);
                  }});
        try {
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress(sender,"Name1 Surnames1"));
            message.setRecipient(Message.RecipientType.TO, new
InternetAddress(receiver,"Name2 Surnames2"));
            message.setSubject("Hello from Java");
            message.setText("Email sent from Java app and captured by
Mailslurper.");
            Transport.send(message);
            System.out.println("Email sent.");
        } catch (MessagingException e) {
            e.printStackTrace();
            System.out.println("Error sending mail.");
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Gmail SMTP server

SMTP server address: smtp.gmail.com

Use authentication: yes

Secure connection: TLS/SSL based on your mail client/website SMTP plugin

SMTP username: your Gmail account (xxxx@gmail.com)

SMTP password: your Gmail password

Gmail SMTP port: 465 (SSL) or 587 (TLS)

To use Gmail SMTP, you need a Gmail account.

To send emails via Gmail SMTP server, you either need to allow "Less secure apps" in your Gmail account or generate an "App Password" if you have 2-Step Verification enabled.

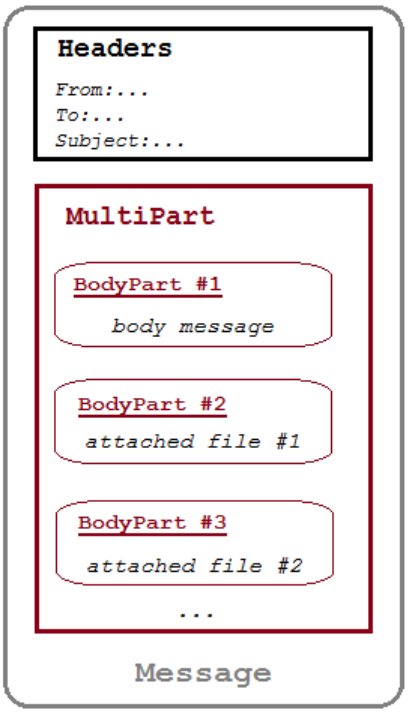
Starting in Fall of 2024, less secure apps, third-party apps, or devices that have you sign in with only your username and password will no longer be supported for Google Workspace accounts.

Sending email with attachments

Adding attachments to emails is often necessary and in **Angus Mail** it is achieved using the **MimeMultipart** object.

The API adheres to the **Multipurpose Internet Mail Extensions** (MIME) standard, allowing developers to create emails with body parts of different media types — text, HTML, images, and video — and to handle attachments effortlessly.

To understand how attachments are stored inside an email message, let's look at the following picture:



As we can see, a **message** consists of a **header** and a **body**. The body must be of type **Multipart** for containing attachments. The Multipart object holds multiple parts, in which each part is represented as a type of **BodyPart** whose subclass, **MimeBodyPart**, can take a file as its content.

We can create a MimeBodyPart object with the email body.

```
MimeBodyPart bodyPart = new MimeBodyPart();
String htmlMessage="…";
bodyPart.setContent(htmlMessage, "text/html");
```

We need to create another MimeBodyPart to add an attachment in our mail.

```
MimeBodyPart attachmentPart = new MimeBodyPart();
attachmentPart.attachFile(new File("/path/to/file"));
```

We have two **MimeBodyPart** objects for one mail Session. So, we need to create one **MimeMultipart** object and then add both the MimeBodyPart objects into it.

```
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(bodyPart);
multipart.addBodyPart(attachmentPart);
```

Finally, the MimeMultiPart is added to the MimeMessage object as our mail content and then, we can send the message.

```
message.setContent(multipart);
Transport.send(message);
```

To summarize, the **Message** contains **MimeMultiPart** which further contains multiple **MimeBodyPart**(s). That's how we assemble the complete email.

Moreover, to send multiple attachments, we can simply add another **MimeBodyPart**.

Inserting images into email for sending using content id

The embedded images are called inline attachments, which users see the images right inside the email's content. That's different from regular attachments, which the users must download and open them manually.

The message must be in HTML format to have inline images.

To tell the mail client to display the attached file within the message body, we refer to the image that is in the message itself. The URL of the image is not on a server, it's in the message.

The MIME specification allows us to identify message parts using the header Content-ID. HTML in one part of the message can refer to another part of the message with a URL that uses the scheme **cid:**. Content ID should be unique across all messages, so you may want to use a generator that uses the host name of the computer sending the message, the current time, and a sequence number to avoid possible collision. Practically, however, many programmers don't bother following the specs and just use an id they find meaningful. Email clients seem to handle it without trouble.

When you set the ContentID on the MimeBodyPart, you must include angle brackets (< and >) around the content id.

The HTML part is built using:

```
String cid = UUID.randomUUID().toString();
MimeBodyPart bodyPart = new MimeBodyPart();

String htmlMessage="<html><head>"
        + "<title>Email protocolos</title>"
        + "</head>"
        + "<body><h1>Hi!</h1>"
        + "<p>The protocol used to send an email over the internet is the Simple Mail
Transfer (SMTP) protocol.</p>"
        + "<p>It defines how the email gets from the sender's email server to the
recipient's email server.</p>"
        + "<img src=\"cid:" + cid+ "\"/>"
        +"<p>I hope this information is useful</p>"
        + "</body></html>";
bodyPart.setContent(htmlMessage,"text/html");
```

And the image part is built with:

```
MimeBodyPart imagePart = new MimeBodyPart();
imagePart.attachFile(new File("/path/to/file"));
imagePart.setContentID("<" + cid + ">");
imagePart.setDisposition(MimeBodyPart.INLINE);
```

Notice how we tell mail clients that the image is to be displayed inline (not as an attachment) with **getDisposition()**.

Lastly, we also tell the mail client that the text part and the image part are related and should be shown as a single item, not as separate pieces of the message. We do so by changing how we create the message content:

```
Multipart multipart = new MimeMultipart("related");
```

Inserting images into email for sending using base64 encoding

HTML inline embedding is much simpler, mostly because you don't have to completely roll your emails and dig around in Multipurpose Internet Mail Extensions or MIME to use it.

Embedding an image in an email first requires that you have a version of the said image as a Base64 encoded string. Once you encode your image, embed it using a standard HTML image tag.

Base 64 is an encoding scheme that converts binary data into text format so that encoded textual data can be easily transported over network uncorrupted and without any data loss.

The basic encoding means no line feeds are added to the output, and the output is mapped to a set of characters in A-Zaz0-9+ character set and the decoder rejects any character outside of this set.

In Java, the Base64 class, available in the java.util package, provides methods for encoding and decoding binary data as Base64 strings. The class contains convenience methods for encoding and

decoding bytes, streams, and strings using the Base64 encoding scheme. By utilising the methods of the Base64 class, we can easily convert images to Base64 strings and vice versa.

To convert an image to a Base64 string, we need to first read the image file into a byte array.

```
File file=new File("/path/to/file");
FileInputStream imageInFile=new FileInputStream(file);
byte imageData[]=new byte[(int)file.length()];
imageInFile.read(imageData);
String imageB64=Base64.getEncoder().encodeToString(imageData);
```

The image should be inserted in the HTML as follows:

```
<img src=" imageB64" />
```

Gmail and other services do not support base64 images in HTML emails. They block display of all base64 encoded strings for security and spamming reasons.

POP3/IMAP

The **POP3** abbreviation stands for **Post Office Protocol version 3**, which provides access to an inbox stored on an email server. It executes the download and deletes operations for messages. Thus, when a POP3 client connects to the mail server, it retrieves all messages from the mailbox. Then it stores them on your local computer and deletes them from the remote server.

Thanks to this protocol, you are able to access the messages locally in offline mode as well.

Modern POP3 clients allow you to keep a copy of your messages on the server if you explicitly select this option.

POP3 ports are the following:

- **110**. This is the default POP3 port, and it is not encrypted.
- **995**. This is the encrypted port for POP3 and works over TLS/SSL.

IMAP

The **Internet Message Access Protocol** (IMAP) allows you to access and manage your email messages on the email server. This protocol permits you to manipulate folders, permanently delete and efficiently search through messages. It also gives you the option to set or remove email flags or fetch email attributes selectively. By default, all messages remain on the server until the user specifically deletes them.

IMAP supports the connection of multiple users to a single mail server.

By default, **IMAP** works on two **ports** like POP3:

- **143**. This is the default port, which does not provide any encryption.
- **993**. This is the secure port for IMAP and it works over TLS/SSL encryption.

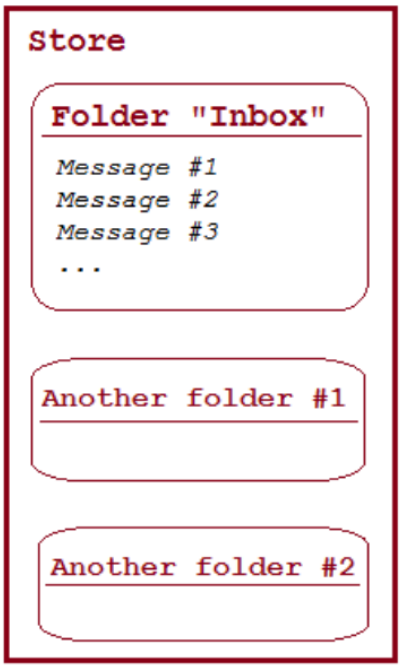
Receiving email from the server using Java

To user Java POP3 to read emails, you first need to add the library Jakarta mail to your project.

Once you have added the dependency, you can start using the Java POP3 library.

The first step is to create a session object that contains the connection properties for the mail server.

The following picture depicts how messages are logically stored on the server:



We can see, for each user account, the server has a **store** which is the storage of the user's messages. The **store** is divided into **folders**, and the "**inbox**" folder is the primary folder, which contains email messages. A folder can contain both messages and sub-folders.

Angus Mail provides three classes: **Store**, **Folder** and **Message**.

A **Store** object can be obtained from the current session by invoking the method **getStore(String protocol)** of **Session** class. Connection to the **Store** can be realised by calling its method **connect(String user, String pass)**, disconnecting by calling its **close()** method.

A **Folder** object can be obtained from the store by invoking the **getFolder(String folderName)** method. For a regular mailbox, the folder name must be "inbox" (case-insensitive). The most important methods of the Folder class are:

- **open(int mode)**. Opens the folder either in **READ_ONLY** mode or **READ_WRITE** mode.
- **getMessages()**. Retrieves an array of Message objects which are marked un-read in the folder. A Message object may be a lightweight reference, and its detailed content will be filled up on demand.
- **close(Boolean expunge)**. Closes the folder, and permanently removes all messages which are marked delete, if expunge is true.

A Message object represents an email message. To get detailed attributes of a message, the following methods can be called on the Message object:

- **Address[] getFrom()**. Returns a list of senders in From attribute of the message.
- **Address[] getRecipients(Message.RecipientType type)**. Gets recipient addresses of the message. Type can be either TO or CC.
- **String getSubject()**. Gets subject of the message.
- **Date getSentDate()**. Gets date and time when the message was sent.
- **Object getContent()**. Gets content of the message.

Typically, the steps to connect to a server and download new email messages are as follows:

- Prepare a **Properties** object which holds server settings such as host, port, protocol…
- Create a **session** to initiate a working session with the server.
- Obtain a **store** from the session by a specific protocol (IMAP or POP3). IMAP is recommended.
- **Connect** to the store using a credential (username and password).
- Get **inbox folder** from the store.
- **Open** the inbox folder.
- **Retrieve messages** from the folder.
- **Fetch details** for each message.
- **Close** the **folder**.
- **Close** the store.

Here is an example of how to create a session object:

```
Properties prop = new Properties();
prop.put("mail.store.protocol", "pop3");
prop.put("mail.pop3.host", "outlook.office365.com");
prop.put("mail.pop3.port", "995");
prop.put("mail.pop3.ssl.enable", "true");
//prop.setProperty("mail.debug", "true");
Session session=Session.getDefaultInstance(prop);
```

Once we have the session object, we can create a store object that represents the connection to the mail server. Here is an example of how to create a store object:

```
Store store=session.getStore("pop3");
store.connect("xxx@outlook.es","xyz_123");
```

Once we have connected to the mail server, we can create a folder object that represents the folder containing the emails that we want to read. Here is an example of how to create a folder object:

```
Folder folder=store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
```

Here is an example of how to retrieve the emails in the folder:

```
Message[] messages=folder.getMessages();
for (int i = 0; i < messages.length; i++) {
    Message message=messages[i];
    System.out.println("Subject:"+ message.getSubject());
    System.out.println("Text:"+message.getContent().toString());
}
```

Search email messages

The Jakarta Mail API allows developers writing code for searching email messages within user's inbox. This can be accomplished by calling the search() function, which is provided by the Folder class: **Message[] search(SearchTerm term)**.

The method returns an array of Message objects which match a search criterion specified by a subclass of **SearchTerm** class. We need to create a class that extends from **SearchTerm** class and overrides its method **match()**.

For example:

```
SearchTerm term = new SearchTerm() {
    public boolean match(Message message) {
        try {
            if (message.getSubject().contains("image")) {
                return true;
            }
        } catch (MessagingException ex) {
            ex.printStackTrace();
        }
        return false;
    }
};
```

As we can see, the match() method above returns true if there is a message whose subject contains the word "image". In other words, the search term above will match all messages having the word "image" in its **Subject** field. And pass this new **SearchTerm** object to the search method as follows:

```
Message[] foundMessages = folder.search(term);
```

The **search()** method returns an **empty array** if no matches were found. Subclasses of Folder class implement this search functionality for corresponding protocol, i.e. **IMAPFolde**r class and **POP3Folder** class.

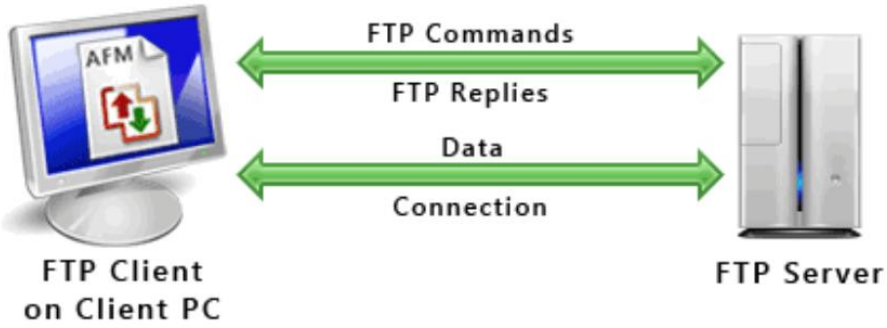
FTP

FTP (File Transfer Protocol) is a standard network protocol used to transfer files between a client computer and a server over a network, such as the Internet.

FTP enables users to **upload**, **download**, and **manage files** on a remote server, making it a widely used method for sharing and distributing files across different platforms and locations.

The protocol operates on a **client-server** model, where the client initiates a connection to the server to perform file transfers.

FTP is commonly used by web developers, content creators and IT professionals to update websites, transfer large files, and perform various file management tasks.



FTP exchanges **data** using **two** separate **channels**:

- **Command Channel**: The command channel is typically used for transmitting (send and receive) commands (e.g. USER, PASS commands) over port 21 (on the server side) between the FTP client and server. This channel will remain open until the client sends out the QUIT command, or if the server forcibly disconnects due to inactivity.
- **Data Channel**: The data channel is used for transmitting data. For an active mode FTP, the data channel will normally be in port 20 (on the server side). And for passive mode, a random port will be selected and used. In this channel, data in the form of directory listings (e.g. LIST, STOR and RETR commands) and file transfers (e.g. normal uploading and downloading of a file). Unlike the command channel, the data channel will close the connection on the port once the data transfer is complete.

FTP is an **unencrypted** protocol and is susceptible to interception and attacks. The requirement of ports to remain open also poses a security risk.

FTP made handling data across the Internet much easier and intuitive. Without FTP and its later iterations, we could not easily stream video content, use video calls, play online games, share files, or enjoy cloud storage.

Today, FTP operates behind the scenes as a backbone for data transfer from servers around the world to millions of clients every second of every day.

Types of FTP

There are different ways through which a server and a client do a file transfer using FTP. Some of them are mentioned below:

- **Anonymous FTP:** Anonymous FTP is enabled on some sites whose files are available for public access. A user can access these files without having any username or password.

Instead, the username is set to **anonymous**, and by convention, users supply an **email** address when prompted for a **password**. Here, user access is very limited. For example, the user can be allowed to copy the files but not to navigate through directories.

- **Password Protected FTP:** This type of FTP is similar to the previous one, but the change in it is the use of username and password.
- **FTP Secure (FTPS):** It is also called as FTP Secure Sockets Layer (FTP SSL). It is a more secure version of FTP data transfer. Whenever FTP connection is established, Transport Layer Security (TLS) is enabled.
- **FTP over Explicit SSL/TLS (FTPES):** FTPES helps by upgrading FTP Connection from port 21 to an encrypted connection.
- **Secure FTP (SFTP):** SFTP is not a FTP Protocol, but it is a subset of Secure Shell Protocol, as it works in port 22.

Java FTP

Although it is possible to use Java networking API (by using the interfaces and classes available in the packages java.net and javax.net) to write code that communicates with a FTP server, that approach is discouraged because you will have to spend a lot of time on understanding the underlying FTP protocol, implementing the protocol handlers, testing, fixing bugs… and finally you re-invent the wheel! Instead, it's advisable to look around and pick up some ready-made libraries, and that definitely saves your time! The **Apache Commons Net** library is an ideal choice for developing FTP based applications.

Apache Commons Net

Apache Commons Net library implements the client side of many basic Internet protocols. The purpose of the library is to provide fundamental protocol access, not high-level abstractions. Therefore, some of the design violates object-oriented design principles. Their philosophy is to make the global functionality of a protocol accessible (e.g., TFTP send file and receive file) when possible, but also provide access to the fundamental protocols where applicable so that the programmer may construct his own custom implementations (e.g, the TFTP packet classes and the TFTP packet send and receive methods are exposed).

Supported protocols include FTP/FTPS, FTP over HTTP (experimental), NNTP, SMTP(S), POP3(S), IMAP(S), Telnet, TFTP, Finger, Whois, rexec/rcmd/rlogin, Time (rdate) and Daytime, Echo, Discard and NTP/SNTP.

The dependency to be added to use the Apache Commons Net library is:

```
<!-- https://mvnrepository.com/artifact/commons-net/commons-net -->
```

```
<dependency>
    <groupId>commons-net</groupId>
    <artifactId>commons-net</artifactId>
    <version>3.11.1</version>
</dependency>
```

The class **FTPClient** provides the necessary APIs to work with a server via **FTP** protocol. To connect to a server, use this method:

void connect(String server, int port)

Where server can be either host name or IP address, and port is a number (FTP protocol is using port number 21).

After connected, use this method to login:

boolean login(String username, String password)

The login() method returns true if login successfully, false if not.

It's advisable to check server's reply code after each call of a void method, such as the connect() method above, for example:

```
int replyCode = ftpClient.getReplyCode();
if (!FTPReply.isPositiveCompletion(replyCode)) {
    // The operation was not successful
    // for some reasons, the server refuses or
    // rejects requested operation
    return;
}
```

After each method call, the server may return some messages. The following method displays server's messages in the standard output:

```
private static void showServerReply(FTPClient ftpClient) {
    String[] replies = ftpClient.getReplyStrings();
    if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
        }
    }
}
```

The following code is an example of a program:

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

import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class Main {
    private static void showServerReply(FTPClient ftpClient) {
        String[] replies = ftpClient.getReplyStrings();
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        String server = "localhost";
        int port = 21;
        String user = "noemi";
        String pass = "noemi";
        FTPClient ftpClient = new FTPClient();
        try {
            ftpClient.connect(server, port);
            showServerReply(ftpClient);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " +
replyCode);
                return;
            }
            boolean success = ftpClient.login(user, pass);
            showServerReply(ftpClient);
            if (!success) {
                System.out.println("Could not login to the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch (IOException ex) {
            System.out.println("Something wrong happened");
            ex.printStackTrace();
        }
        ftpClient.logout();
        ftpClient.disconnect();
    }
}
```

Listing files and directories in an FTP server

The **FTPClient** class provides the following methods for listing **content** of a directory on the FTP server:

- FTPFile[] listDirectories()
- FTPFile[] listDirectories(String parent)
- FTPFile[] listFiles()
- FTPFile[] listFiles(String pathname)

All the above methods return an array of **FTPFile objects** which represent files and directories. The **FTPFile** class provides various methods for **querying detailed information** of a file or directory, to name some useful ones:

- String getName():returns name of the file or directory.
- long getSize():return size in bytes.
- boolean isDirectory():determines if the entry is a directory.
- boolean isFile():determines if the entry is a file.
- Calendar getTimestamp():returns last modified time.

The **FTPClient** class also has two simple methods for listing **files** and **directories**:

- String[] listNames()
- String[] listNames(String pathname)

Unlike the listFiles() and listDirectories() methods, the **listNames()**methods simply return an **array** of **String** represents file/directory names; and they list both files and directories.

The following code snippet demonstrates listing files and directories under the current working directory.

```
FTPFile[] files = ftpClient.listFiles();
// iterates over the files and prints details for each
DateFormat dateFormater = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
for (FTPFile file : files) {
   String details = file.getName();
   if (file.isDirectory()) {
      details = "[" + details + "]";
   }
   details += "\t\t" + file.getSize();
   details += "\t\t" + dateFormater.format(file.getTimestamp().getTime());
   System.out.println(details);
}
```

Other interesting variations are:

```
FTPFile[] files = ftpClient.listFiles("/document");
FTPFile[] files = ftpClient.listDirectories("/images");
FTPFile[] files = ftpClient.listDirectories();
```

The following code snippet demonstrates the use of the **listNames** method:

```
String[] files = ftpClient.listNames();
if (files != null && files.length > 0) {
    for (String aFile: files) {
        System.out.println(aFile);
    }
}
```

Java FTP file download

The **FTPClient** class of **Apache Commons Net API** provides **two methods** for downloading files from an FTP server:

- **boolean retrieveFile(String remote, OutputStream local)**: This method retrieves a remote file whose path is specified by the parameter remote, and writes it to the OutputStream specified by the parameter local. The method returns true if the operation completed successfully, or false otherwise. This method is suitable in case we don't care how the file is written to the disk, just let the system use the given OutputStream to write the file. We should close OutputStream after the method returns.
- **InputStream retrieveFileStream(String remote)**: This method does not use an OutputStream, instead it returns an InputStream which we can use to read bytes from the remote file. This method gives us more control on how to read and write the data. But there are two important points when using this method:
    - The method **completePendingCommand()** must be called afterward to finalise file transfer and check its return value to verify if the download is actually done successfully.
    - We must **close** the **InputStream** explicitly.

Which method is used suitable for you? Here are a few tips:

- The **first** method provides the **simplest way** for downloading a remote file, as just passing an **OutputStream** of the file will be written on the disk.
- The second method requires more code to be written, as we must create a new **OutputStream** for writing file's content while reading its byte arrays from the returned **InputStream**. This method is useful when we want to measure the progress of the download, i.e. how many percentages of the file have been transferred. In addition, we have to call the **completePendingCommand()** to finalise the download.
- Both the methods throw an **IOException** exception (or one of its descendants, FTPConnectionClosedException and CopyStreamException). Therefore, make sure to handle these exceptions when calling the methods.

In addition, the following two methods must be invoked before calling the **retrieveFile()** and **retrieveFileStream()** methods:

- **void enterLocalPassiveMode()**: this method switches data connection mode from server-to-client (default mode) to client-to-server, which can pass through firewall. There might be some connection issues if this method is not invoked.
- **boolean setFileType(int fileType)**: this method sets the file type to be transferred, either as ASCII text file or binary file. It is recommended to set file type to FTP.BINARY_FILE_TYPE, rather than FTP.ASCII_FILE_TYPE.

The **steps** to **download** a **file** are the following:

- **Connect** and **login** to the **server**.
- Enter local **passive mode** for data connection.
- Set **file type** to be transferred to **binary**.
- Construct **path** of the **remote file** to be downloaded.
- Create a new **OutputStream** for writing the file to the disk.
- If using the first method (**retrieveFile**):
    - **Pass** the remote file path and the **OutputStream** as arguments of the method **retrieveFile()**.
    - **Close** the **OutputStream**.
    - **Check** the return value of **retrieveFile()** to verify success.
- If using the second method (**retrieveFileStream**):
    - Retrieve an **InputStream** returned by the method **retrieveFileStream()**.
    - Repeatedly a byte array from the **InputStream** and write these bytes into the **OutputStream**, until the **InputStream** is empty.
    - Call **completePendingCommand()** method to complete transaction.
    - Close the opened **OutputStream** the **InputStream**.
    - Check the return value of **completePendingCommand()** to verify success.
- **Logout** and **disconnect** from the **server**.

The following sample program uses both methods for transferring a file from the FTP server to the local computer.

```
import org.apache.commons.net.ftp.FTP;
import org.apache.commons.net.ftp.FTPClient;
import java.io.*;

public class FtpFileDownload {
    public static void main(String[] args) {
```

```
        String server = "192.168.56.1";
        int port = 21;
        String user = "xxxxx";
        String pass = "xxxxx";

        FTPClient ftpClient = new FTPClient();
        try {
            ftpClient.connect(server, port);
            ftpClient.login(user, pass);
            ftpClient.enterLocalPassiveMode();
            ftpClient.setFileType(FTP.BINARY_FILE_TYPE);

            // APPROACH #1: using retrieveFile(String, OutputStream)
            String remoteFile1 = "/user/image.jpg";
            File downloadFile1 = new File("image.jpg");
            OutputStream outputStream1 = new BufferedOutputStream(new
FileOutputStream(downloadFile1));
            boolean success = ftpClient.retrieveFile(remoteFile1,
outputStream1);
            outputStream1.close();
            if (success) {
                System.out.println("File #1 has been downloaded
successfully.");
            }
            // APPROACH #2: using InputStream retrieveFileStream(String)
            String remoteFile2 = "/images.png";
            File downloadFile2 = new File("images.png");
            OutputStream outputStream2 = new BufferedOutputStream(new
FileOutputStream(downloadFile2));
            InputStream inputStream =
ftpClient.retrieveFileStream(remoteFile2);
            byte[] bytesArray = new byte[4096];
            int bytesRead = -1;
            while ((bytesRead = inputStream.read(bytesArray)) != -1) {
                outputStream2.write(bytesArray, 0, bytesRead);
            }
            success = ftpClient.completePendingCommand();
            if (success) {
               System.out.println("File #2 has been downloaded successfully.");
            }
            outputStream2.close();
            inputStream.close();
        } catch (IOException ex) {
            System.out.println("Error: " + ex.getMessage());
            ex.printStackTrace();
        } finally {
            try {
                if (ftpClient.isConnected()) {
                    ftpClient.logout();
                    ftpClient.disconnect();}
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Java FTP file upload

Uploading a single file to an FTP server is a single task.

The following code uses a method called **uploadSingleFile** to upload a file to an FTP server.

```
import org.apache.commons.net.ftp.FTP;
import org.apache.commons.net.ftp.FTPClient;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class FtpFileUpload {
    /**
     * Upload a single file to the FTP server.
     *
     * @param ftpClient
     *            an instance of org.apache.commons.net.ftp.FTPClient class.
     * @param localFilePath
     *            Path of the file on local computer
     * @param remoteFilePath
     *            Path of the file on remote the server
     * @return true if the file was uploaded successfully, false otherwise
     * @throws IOException
     *             if any network or IO error occurred.
     */
    public static boolean uploadSingleFile(FTPClient ftpClient,
                                           String localFilePath, String
remoteFilePath) throws IOException {
        File localFile = new File(localFilePath);
        InputStream inputStream = new FileInputStream(localFile);
        try {
            ftpClient.setFileType(FTP.BINARY_FILE_TYPE);
            return ftpClient.storeFile(remoteFilePath, inputStream);
        } finally {
            inputStream.close();
        }
    }
    public static void main(String[] args) {
        String server = "192.168.56.1";
        int port = 21;
        String user = "xxxxx";
        String pass = "xxxxx";

        FTPClient ftpClient = new FTPClient();


        try {
            // connect and login to the server
            ftpClient.connect(server, port);
            ftpClient.login(user, pass);

            // use local passive mode to pass firewall
            ftpClient.enterLocalPassiveMode();
```

```
            System.out.println("Connected");

            String remoteFilePath = "./file.xml";
            String localFilePath = "C:\\Users\\Name\\Downloads\\file.xml";

            Boolean res=
uploadSingleFile(ftpClient,localFilePath,remoteFilePath);
            if (res){
                System.out.println("Upload successfully completed");
            }

            // log out and disconnect from the server
            ftpClient.logout();
            ftpClient.disconnect();

            System.out.println("Disconnected");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

SFTP

Secure Shell (SSH) is an essential protocol for secure communication over unsecured networks. Developed in the 1990s and designed as a replacement for insecure remote shell protocols like **telnet** and **rsh**, providing encrypted channel communication and strong authentication. With two iterations, **SSH-1** and **SSH-2**, SSH-2 is the most widely used version today because of its enhanced security features and extensive protocol standardisation.

Beyond its primary use for remote **command-line login** and **command execution**, SSH offers a suite of functionalities. It supports secure file transfer via **SFTP** (SSH File Transfer Protocol) and **SCP** (Secure Copy Protocol). It allows for **secure tunnelling** of application protocols for other **TCP-based services** through port forwarding. Additionally, it supports remote shell access, enabling users to execute commands on a remote machine as if they were physically present.

Java has seen a variety of libraries developed to handle **SSH connections**. These libraries have evolved substantially since the early 2000s, both in the open-source and commercial sectors.

JSch

The **JSch** (Java Secure Channel) library is a widely used Java library that provides functionality for **secure SSH communication**. JSch is designed to allow Java applications to connect to remote servers using the SSH protocol, ensuring **data** is **encrypted** and **securely transmitted** over the network. This library is particularly useful for tasks such as executing remote commands, **transferring files**, managing tunnels, and interacting with remote systems without manual intervention.

Add the **JSch dependency** to your **pom.xml** to include the library in your project:

```
<dependency>
    <groupId>com.github.mwiede</groupId>
    <artifactId>jsch</artifactId>
    <version>0.2.23</version>
</dependency>
```

JSch enables us to use either **Password Authentication** or **Public Key Authentication** to access a remote server.

Below is an example of how to connect to an SFTP server using JSch and download a file:

```
public class SFTPExample {
    public static void main(String[] args) {
        String host = "192.168.1.130";
        String username = "xxxx";
        String password = "yyyy";
        String remoteFile = "file_xxx_remote.txt";
        String localFile = "file_xxx.txt";

        JSch jsch = new JSch();
        Session session = null;
        try {
            session = jsch.getSession(username, host, 22);
            session.setPassword(password);
            session.setConfig("StrictHostKeyChecking", "no");
            session.connect();

            Channel channel = session.openChannel("sftp");
            channel.connect();
            ChannelSftp sftpChannel = (ChannelSftp) channel;

            sftpChannel.put(localFile, remoteFile);

            sftpChannel.disconnect();
            session.disconnect();
        } catch (SftpException | JSchException e) {
            e.printStackTrace();
        }
    }
}
```

Once we've established an **SFTP connection**, we can perform various operations, such as **uploading files**, **downloading files**, **listing directory contents**, **creating directories**, etc. Here are a few examples:

Upload a file:

```
sftpChannel.put("path/to/local/file.txt", "path/to/remote/file.txt");
```

Download a file:

```
sftpChannel.get("path/to/local/file.txt", "path/to/remote/file.txt");
```

List directory content:

```
Vector<ChannelSftp.LsEntry> files = sftpChannel.ls(".");
for (ChannelSftp.LsEntry entry : files) {
    System.out.println(entry.getLongname());}
```

In the code above, we invoke the **ls()** method on the **ChannelSftp** object which returns a Vector of **ChannelSftp.LsEntry** object, each representing a file or directory. Then, we loop over the list of files and directories and log the long name of each file or directory. The **getLongname()** method includes additional details like permissions, owner, group, and size. If we are interested only in the filename, we can invoke the **getFilename()** method on **ChannelSftp.LsEntry** object.

Public key authentication with Java over SSH

Public key authentication enables users to establish an SSH connection without providing (i.e. typing in) explicit password. The immediate benefit is that the password is not transferred over the network, thus preventing the possibility of the password being compromised.

The private key should be stored in the ssh keychain and protected with the encryption passphrase. **PuTTYgen** is a key generator tool for creating **SSH keys**. Its basic function is to create **public** and **private key** pairs. **PuTTY** stores keys in its own format in .**ppk** files. However, the tool can also convert keys to and from other formats.

To create a new key pair, select the type of key to generate from the bottom of the screen (using SSH-2 RSA with 2048-bit key size is good for most people). The click Generate and start moving the mouse within the Window. The exact way you are going to move your mouse cannot be predicted by an external attacker. Putty uses mouse movement to collect randomness. The exact way you are going to move your mouse cannot be predicted by an external attacker. You may need to move the mouse for some time, depending on the size of your key. As you move it, the green progress bar should advance. Once the progress bar becomes full, the actual key generation computation takes place. This may take from several seconds to several minutes. When complete, the public key should appear in the window. You can now specify a **passphrase** for the key.

You should save at least the private key by clicking Save private key. It may be advisable to also save the public key, though it can be later regenerated by loading the private key (by clicking Load).

It is recommended to use a passphrase for private key files intended for interactive use. If keys are needed for automation, then they may be left without a passphrase.

The **public key** must be copied on the **remote server**.

PuTTYgen supports exporting a **private key** to an **OpenSSH** compatible format if needed.

Here is the sample code for configuring public key authentication:

```
public class PrivateKeySFTP {
    public static void main(String[] args) {
        String host = "192.168.1.130";
        String username = "zzzzz";
        int port=22;
        JSch jsch = new JSch();
        Session session = null;
```

```
        String privateKeyPath = "src/main/resources/private_rsa_ssh2_v2.ppk";
        try {
            jsch.addIdentity(privateKeyPath);
            session = jsch.getSession(username, host, port);
            session.setConfig("PreferredAuthentications", "publickey,keyboard-
interactive,password");
            java.util.Properties config = new java.util.Properties();
            config.put("StrictHostKeyChecking", "no");
            session.setConfig(config);
            session.connect();
            System.out.println("Connected");
        } catch (JSchException e) {
            throw new RuntimeException("Failed to create Jsch Session object.", e);
        }
    }
}
```