JavaMail API Tutorial

JAVAMAIL API TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

JavaMail API tutorial

The JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications. The JavaMail API provides a set of abstract classes defining objects that comprise a mail system. It is an optional package (standard extension) for reading, composing, and sending electronic messages.

Audience

This tutorial has been prepared for the beginners to help them understand basic JavaMail programming. After completing this tutorial you will find yourself at a moderate level of expertise in JavaMail programming from where you can take yourself to next levels.

Prerequisites

JavaMail programming is based on Java programming language so if you have basic understanding on Java programming then it will be a fun to learn using JavaMail in application development.

Table of Content

JavaMail API tutorial 1
Audience 1
Prerequisites 1
JavaMail API – Overview 5
Architecture 6
JavaMail API - Environment Setup 7
SMPT server 7
JavaMail API - Core Classes 8
Session Class 8
Message Class 9
Address Class 10
Authenticator Class 10
Transport Class 11
Store Class 11
Folder Class 11
JavaMail API - Sending Emails 13
JavaMail API - Sending Simple Email 14
Create Java Class 14
Compile and Run 16
Verify Output 16
JavaMail API - Sending Email With Attachment 17
Create Java Class 17
Compile and Run 19
Verify Output 20
JavaMail API - Sending an HTML Email 21
Create Java Class 21
Compile and Run 23
Verify Output 23
JavaMail API - Sending Email With Inline Images 24
Create Java Class 25
Compile and Run 27
Verify Output 27
JavaMail API - Checking Emails 28
Create Java Class 28

Compile and Run 30
Verify Output 30
JavaMail API - Fetching Emails 31
Create Java Class 31
Compile and Run 35
Verify Output 36
JavaMail API – Authentication 38
Create Java Class 38
Compile and Run 40
Verify Output 40
JavaMail API - Replying Emails 41
Create Java Class 41
Compile and Run 44
Verify Output 44
JavaMail API - Forwarding Emails 45
Create Java Class 45
Compile and Run 48
Verify Output 48
JavaMail API - Deleting Emails 50
Create Java Class 50
Compile and Run 52
Verify Output 53
JavaMail API - Gmail SMPT Server 54
Create Java Class 54
Compile and Run 56
Verify Output 56
JavaMail API - Folder Management 57
Opening a Folder 57
Basic Folder Info 58
Managing Folder 58
Managing Messages in Folders 58
Listing the Contents of a Folder 58
Checking for Mail 59
Getting Messages from Folders 59
Searching Folders 59
Flags 60
JavaMail API - Quota Management 61
Create Java Class 61
Compile and Run 63

Verify Output 63
JavaMail API - Bounced Messages 64
Create Java Class 64
Compile and Run 65
Verify Output 66
JavaMail API - SMTP Servers 67
JavaMail API - IMAP Servers 72
JavaMail API - POP3 Servers 76

CHAPTER

1

JavaMail API – Overview

The JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications. The JavaMail API provides a set of abstract classes defining objects that comprise a mail system. It is an optional package (standard extension) for reading, composing, and sending electronic messages.

JavaMail provides elements that are used to construct an interface to a messaging system, including system components and interfaces. While this specification does not define any specific implementation, JavaMail does include several classes that implement RFC822 and MIME Internet messaging standards. These classes are delivered as part of the JavaMail class package.

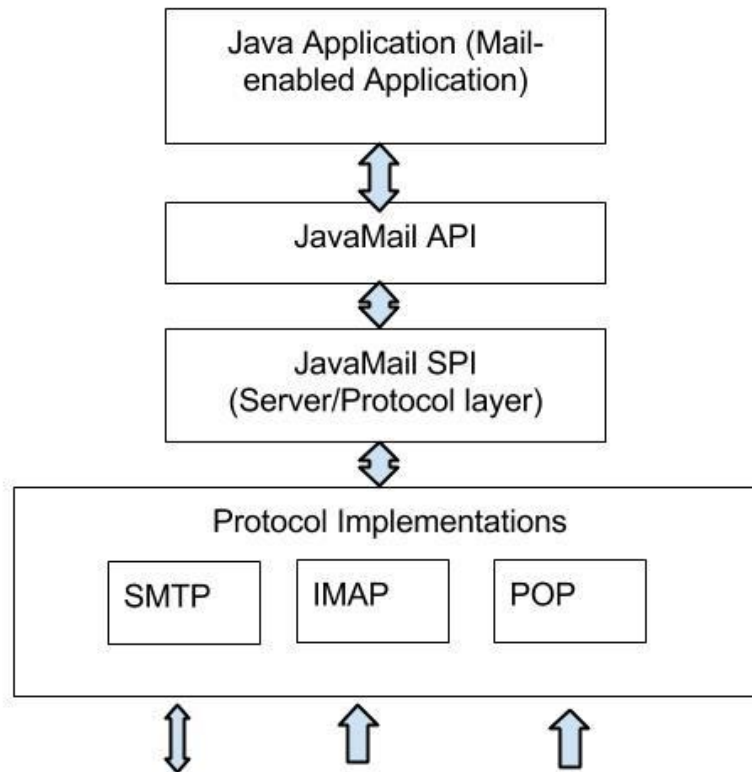
Following are some of the protocols supported in JavaMail API:

- **SMTP**: Acronym for **Simple Mail Transfer Protocol**. It provides a mechanism to deliver email.
- **POP**: Acronym for **Post Office Protocol**. POP is the mechanism most people on the Internet use to get their mail. It defines support for a single mailbox for each user. RFC 1939 defines this protocol.
- **IMAP**: Acronym for **Internet Message Access Protocol**. It is an advanced protocol for receiving messages. It provides support for multiple mailbox for each user, in addition to, mailbox can be shared by multiple users. It is defined in RFC 2060.
- **MIME**: Acronym for **Multipurpose Internet Mail Extensions**. . It is not a mail transfer protocol. Instead, it defines the content of what is transferred: the format of the messages, attachments, and so on. There are many different documents that take effect here: RFC 822, RFC 2045, RFC 2046, and RFC 2047. As a user of the JavaMail API, you usually don't need to worry about these formats. However, these formats do exist and are used by your programs.
- **NNTP and Others**:There are many protocols that are provided by third-party providers. Some of them are Network News Transfer Protocol (NNTP), Secure Multipurpose Internet Mail Extensions (S/MIME) etc.

Details of these will be covered in the subsequent chapters.

Architecture

As said above the java application uses JavaMail API to compose, send and receive emails.The following figure illustrates the architecture of JavaMail:



The abstract mechanism of JavaMail API is similar to other J2EE APIs, such as JDBC, JNDI, and JMS. As seen the architecture diagram above, JavaMail API is divided into two main parts:

- An application-independent part: An application-programming interface (API) is used by the application components to send and receive mail messages, independent of the underlying provider or protocol used.
- A service-dependent part: A service provider interface (SPI) speaks the protocol-specific languages, such as SMTP, POP, IMAP, and Network News Transfer Protocol (NNTP). It is used to plug in a provider of an e-mail service to the J2EE platform.

CHAPTER

2

JavaMail API - Environment Setup

To send an e-mail using your Java Application is simple enough but to start with you should have **JavaMail API** and **Java Activation Framework (JAF)** installed on your machine.

You will need the **JavaBeans Activation Framework (JAF)** extension that provides the *javax.activation* package only when you're not using Java SE 6 or newer.

- You can download latest version of JavaMail (Version 1.5.0) from Java's standard website.
- You can download latest version of JAF (Version 1.1.1) from Java's standard website.

Download and unzip these files, in the newly created top level directories you will find a number of jar files for both the applications. You need to add **mail.jar** and **activation.jar** files in your CLASSPATH.

SMPT server

To send emails, you must have SMTP server that is responsible to send mails. You can use one of the following techniques to get the SMTP server:

- Install and use any SMTP server such as Postfix server (for Ubuntu), Apache James server (Java Apache Mail Enterprise Server)etc. (or)
- Use the SMTP server provided by the host provider for eg: free SMTP provide by JangoSMTP site is *relay.jangosmtp.net* (or)
- Use the SMTP Server provided by companies e.g. gmail, yahoo, etc.

The examples in the subsequent chapters, we've used the free JangoSMTP server to send email. You can create an account by visiting this site and configure your email adress.

CHAPTER

3

JavaMail API - Core Classes

The JavaMail API consists of some interfaces and classes used to send, read, and delete e-mail messages. Though there are many packages in the JavaMail API, will cover the main two packages that are used in Java Mail API frequently: javax.mail and javax.mail.internet package. These packages con tain all the JavaMail core classes. They are:

Class	Description
javax.mail.Session	The key class of the API. A multithreaded object represents the connection factory.
javax.mail.Message	An abstract class that models an e-mail message. Subclasses provide the actual implementations.
javax.mail.Address	An abstract class that models the addresses (from and to addresses) in a message. Subclasses provide the specific implementations.
javax.mail.Authenticator	An abstract class used to protect mail resources on the mail server.
javax.mail.Transport	An abstract class that models a message transport mechanism for sending an e-mail message.
javax.mail.Store	An abstract class that models a message store and its access protocol, for storing and retrieving messages. A Store is divided into Folders.
javax.mail.Folder	An abstract class that represents a folder of mail messages. It can contain subfolders.
javax.mail.internet.**MimeMessage**	Message is an abstract class, hence must work with a subclass; in most cases, you'll use a MimeMessage. A MimeMessage is an e-mail message that understands MIME types and headers.
javax.mail.internet.**InternetAddress**	This class represents an Internet email address using the syntax of RFC822. Typical address syntax is of the form *user@host.domain* or *Personal Name <user@host.domain>*.

Let us study each of these classes in detail and in the subsequent chapters we shall study examples using each of these.

Session Class

The *Session* class is the primary class of the JavaMail API and it is not subclassed. The *Session* object acts as the connection factory for the JavaMail API, which handles both configuration setting and authentication.

Session object can be created in the following ways:

- By looking up the administered object stored in the JNDI service

```
InitialContext ctx = new InitialContext();
Session session = (Session) ctx.lookup("usersMailSession");
```

Users MailSession is the JNDI name object used as the administered object for the Session object.*usersMailSession* can be created and configured with the required parameters as name/value pairs, including information such as the mail server hostname, the user account sending the mail, and the protocols supported by the Session object.

- Another method of creating the Session object is based on the programmatic approach in which you can use a *java.util.Properties* object to override some of the default information, such as the mail server name, username, password, and other information that can be shared across your entire application.

The constructor for *Session* class is *private*. Hence the *Session* class provides two methods (listed below) which get the *Session* object.

- **getDefaultInstance()**: There are two methods to get the session object by using the getDefaultInstance() method. It returns the default session.

```
public static Session getDefaultInstance(Properties props)
public static Session getDefaultInstance(Properties props,Authenticator auth)
```

- **getInstance()**: There are two methods to get the session object by using the getInstance() method. It returns the new session.

```
public static Session getInstance(Properties props)
public static Session getInstance(Properties props,Authenticator auth)
```

Message Class

With Session object created we now move on to creating a message that will be sent. The message type will be *javax.mail.Message*.

- *Message* is an abstract class. Hence its subclass *javax.mail.internet.MimeMessage* class is mostly used.
- To create the message, you need to pass session object in MimeMessage class constructor. For example:

```
MimeMessage message=new MimeMessage(session);
```

- Once the message object is created we need to store information in it. *Message* class implements the javax.mail.Part interface while javax.mail.internet. MimeMessage implements javax.mail.internet.MimePart. You can either use message.setContent() or mimeMessage.setText() to store the content.
- Commonly used methods of MimeMessage class are

Method	Description
public void setFrom(Address address)	used to set the from header field.
public void addRecipients(Message.RecipientType type, String addresses)	used to add the given address to the recipient type.
public void setSubject(String subject)	used to set the subject header field.
public void setText(String textmessage)	used to set the text as the message content using text/plain MIME type.

Address Class

Now that we have a Session and Message (with content stored in it) objects, we need to address the letter by using *Address* object.

- *Address* is an abstract class. Hence its subclass *javax.mail.internet.InternetAddress* class is mostly used.
- *Address* can be created by just passing email address:

```
Address address = new InternetAddress("manisha@gmail.com");
```

- Another way of creating Address is by passing name alogwith the email address:

```
Address address = new InternetAddress("manisha@gmail.com", Manisha);
```

- You can also set the To, From, CC, BCC fields as below
    - message.setFrom(address)
    - message.addRecipient(type, address)
    - Three predefined address types are objects with one of these values:
        - Message.RecipientType.TO
        - Message.RecipientType.CC
        - Message.RecipientType.BCC

Authenticator Class

The class Authenticator represents an object that knows how to obtain authentication for a network connection. Usually, it will do this by prompting the user for information.

- *Authenticator* is an abstract class. You create a subclass *PasswordAuthentication*, passing a username and password to its constructor.
- You must register the Authenticator with the Session when you create session object.

Following is an example of *Authenticator* use:

```
Properties props = new Properties();
//Override props with any customized data
PasswordAuthentication auth = new PasswordAuthentication("manisha", "pswrd")
Session session = Session.getDefaultInstance(props, auth);
```

Transport Class

Transport class is used as a message transport mechanism. This class normally uses the SMTP protocol to send a message.

- It is an abstract class.
- You can use the default version of the class by just calling the static *send()* method:

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

- The other way to send message is by getting a specific instance from the session for your protocol, pass along the username and password (blank if unnecessary), send the message, and close the connection:

```
message.saveChanges(); // implicit with send()
//Get transport for session
Transport transport = session.getTransport("smtp");
//Connect
transport.connect(host, username, password);
//repeat if necessary
transport.sendMessage(message, message.getAllRecipients());
//Done, close the connection
transport.close();
```

Store Class

An abstract class that models a message store and its access protocol, for storing and retrieving messages. Subclasses provide actual implementations. *Store* extends the *Service* class, which provides many common methods for naming stores, connecting to stores, and listening to connection events.

Clients gain access to a Message Store by obtaining a Store object that implements the database access protocol. Most message stores require the user to be authenticated before they allow access. The connect method performs that authentication.

```
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

Folder Class

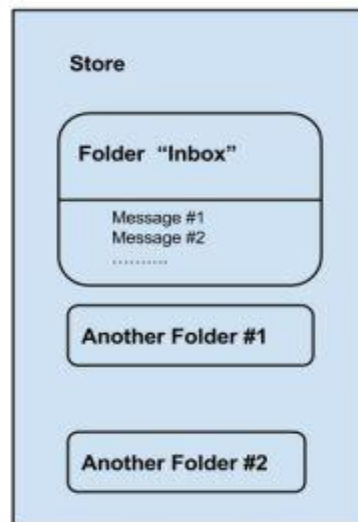
Folder is an abstract class that represents a folder for mail messages. Subclasses implement protocol specific Folders. Folders can contain subfolders as well as messages, thus providing a hierarchical structure.

After connecting to the Store, you can then get a Folder, which must be opened before you can read messages from it.

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
```

The *getFolder(String name)* method for a Folder object returns the named subfolder. Close the both the Store and Folder connection once reading mail is done.

We can see the Store and Folder relation the image below:



As we can see, for each user account, the server has a store which is the storage of user's messages. The store is divided into folders, and the "inbox" folder is the primarily folder which contains e-mail messages. A folder can contain both messages and sub-folders.

CHAPTER

4

JavaMail API - Sending Emails

Now that we have a fair idea about JavaMail API and its core classes, let us now write a simple programs which will send simple email, email with attachments, email with HTML content and email with inline images.

Basic steps followed in all the above scenarios are as below:

- Get the Session object.
- Compose a message.
- Send the message.

In the following sections we have demonstrated simple examples of:

- Send simple email
- Send attachment in email
- Send HTML content in email
- Send inline image in email

CHAPTER

5

JavaMail API - Sending Simple Email

Here is an example to send a simple email. Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

To send a simple email steps followed are:

- Get a Session
- Create a default MimeMessage object and set *From, To, Subject* in the message.
- Set the actual message as:

```
message.setText("your text goes here");
```

- Send the message using the Transport object.

Create Java Class

Create a java class file **SendEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendEmail {
  public static void main(String[] args) {
    // Recipient's email ID needs to be mentioned.
    String to = "destinationemail@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "fromemail@gmail.com";
    final String username = "manishaspatil";//change accordingly
```

```
final String password = "******";//change accordingly

// Assuming you are sending email through relay.jangosmtp.net
String host = "relay.jangosmtp.net";

Properties props = new Properties();
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");
props.put("mail.smtp.host", host);
props.put("mail.smtp.port", "25");

// Get the Session object.
Session session = Session.getInstance(props,
  new javax.mail.Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
      return new PasswordAuthentication(username, password);
    }
  });

try {
    // Create a default MimeMessage object.
    Message message = new MimeMessage(session);

    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));

    // Set To: header field of the header.
    message.setRecipients(Message.RecipientType.TO,
      InternetAddress.parse(to));

    // Set Subject: header field

    message.setSubject("Testing Subject");



    // Now set the actual message

    message.setText("Hello, this is sample for to check send " +

        "email using JavaMailAPI ");



    // Send message

    Transport.send(message);



    System.out.println("Sent message successfully....");
```

```
 } catch (MessagingException e) {
         throw new RuntimeException(e);
      }
   }
}
```

As we are using the SMTP server provided by the host provider JangoSMTP, we need to authenticate the username and password. The *javax.mail.PasswordAuthentication* class is used to authenticate the password.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmail
```

Verify Output

You should see the following message on the command console:

```
Sent message successfully....
```

As I'm sending an email to my gmail address through JangoSMTP, the following mail would be received in my gmail account inbox:

CHAPTER

6

JavaMail API - Sending Email With Attachment

Here is an example to send an email with attachment from your machine. The file on local machine is **file.txt** placed at */home/manisha/*. Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

To send a email with an inline image, the steps followed are:

- Get a Session
- Create a default MimeMessage object and set *From, To, Subject* in the message.
- Set the actual message as below:

```
messageBodyPart.setText("This is message body");
```

- Create a MimeMultipart object. Add the above messageBodyPart with actual message set in it, to this multipart object.
- Next add the attachment by creating a Datahandler as follows:

```
messageBodyPart = new MimeBodyPart();
String filename = "/home/manisha/file.txt";
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);
```

- Next set the multipart in the message as follows:

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

- Send the message using the Transport object.

Create Java Class

Create a java class file **SendAttachmentInEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import javax.mail.BodyPart;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

public class SendAttachmentInEmail {
  public static void main(String[] args) {
    // Recipient's email ID needs to be mentioned.
    String to = "destinationemail@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "fromemail@gmail.com";

    final String username = "manishaspatil";//change accordingly
    final String password = "******";//change accordingly

    // Assuming you are sending email through relay.jangosmtp.net
    String host = "relay.jangosmtp.net";

    Properties props = new Properties();
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.starttls.enable", "true");
    props.put("mail.smtp.host", host);
    props.put("mail.smtp.port", "25");

    // Get the Session object.
    Session session = Session.getInstance(props,
      new javax.mail.Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
          return new PasswordAuthentication(username, password);
        }
      });

    try {
         // Create a default MimeMessage object.
      Message message = new MimeMessage(session);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
```

```
         message.setRecipients(Message.RecipientType.TO,
            InternetAddress.parse(to));

         // Set Subject: header field
         message.setSubject("Testing Subject");

         // Create the message part
         BodyPart messageBodyPart = new MimeBodyPart();

         // Now set the actual message
         messageBodyPart.setText("This is message body");

         // Create a multipar message
         Multipart multipart = new MimeMultipart();

         // Set text message part
         multipart.addBodyPart(messageBodyPart);

         // Part two is attachment
         messageBodyPart = new MimeBodyPart();
         String filename = "/home/manisha/file.txt";
         DataSource source = new FileDataSource(filename);
         messageBodyPart.setDataHandler(new DataHandler(source));
         messageBodyPart.setFileName(filename);
         multipart.addBodyPart(messageBodyPart);

         // Send the complete message parts
         message.setContent(multipart);

         // Send message
         Transport.send(message);

         System.out.println("Sent message successfully....");

      } catch (MessagingException e) {
         throw new RuntimeException(e);
      }
   }
}
```

As we are using the SMTP server provided by the host provider JangoSMTP, we need to authenticate the username and password. The *javax.mail.PasswordAuthentication* class is used to authenticate the password.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendAttachmentInEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendAttachmentInEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendAttachmentInEmail
```

Verify Output

You should see the following message on the command console:

```
Sent message successfully....
```

As I'm sending an email to my gmail address through JangoSMTP, the following mail would be received in my gmail account inbox:

CHAPTER 7

JavaMail API - Sending an HTML Email

Here is an example to send an HTML email from your machine. Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

This example is very similar to sending simple email, except that, here we are using setContent() method to set content whose second argument is "text/html" to specify that the HTML content is included in the message. Using this example, you can send as big as HTML content you like.

To send a email with HTML content, the steps followed are:

- Get a Session
- Create a default MimeMessage object and set *From, To, Subject* in the message.
- Set the actual message using setContent() method as below:

```
message.setContent("<h1>This is actual message embedded in  HTML tags</h1>", "text/html");
```

- Send the message using the Transport object.

Create Java Class

Create a java class file **SendHTMLEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
```

```
public class SendHTMLEmail {
   public static void main(String[] args) {
      // Recipient's email ID needs to be mentioned.

      String to = "destinationemail@gmail.com";

      // Sender's email ID needs to be mentioned
      String from = "fromemail@gmail.com";
      final String username = "manishaspatil";//change accordingly
      final String password = "******";//change accordingly

      // Assuming you are sending email through relay.jangosmtp.net
      String host = "relay.jangosmtp.net";

      Properties props = new Properties();
      props.put("mail.smtp.auth", "true");
      props.put("mail.smtp.starttls.enable", "true");
      props.put("mail.smtp.host", host);
      props.put("mail.smtp.port", "25");

      // Get the Session object.
      Session session = Session.getInstance(props,
         new javax.mail.Authenticator() {
            protected PasswordAuthentication getPasswordAuthentication() {
               return new PasswordAuthentication(username, password);
            }
        });

      try {
            // Create a default MimeMessage object.
            Message message = new MimeMessage(session);

           // Set From: header field of the header.
           message.setFrom(new InternetAddress(from));

           // Set To: header field of the header.
           message.setRecipients(Message.RecipientType.TO,
              InternetAddress.parse(to));

           // Set Subject: header field
           message.setSubject("Testing Subject");

           // Send the actual HTML message, as big as you like
           message.setContent(
              "<h1>This is actual message embedded in HTML tags</h1>",
             "text/html");
```

```
            // Send message
            Transport.send(message);

            System.out.println("Sent message successfully....");

        } catch (MessagingException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
}
```

As we are using the SMTP server provided by the host provider JangoSMTP, we need to authenticate the username and password. The *javax.mail.PasswordAuthentication* class is used to authenticate the password.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendHTMLEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendHTMLEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendHTMLEmail
```

Verify Output

You should see the following message on the command console:

```
Sent message successfully....
```

As I'm sending an email to my gmail address through JangoSMTP, the following mail would be received in my gmail account inbox:

CHAPTER

8

JavaMail API - Sending Email With Inline Images

Here is an example to send an HTML email from your machine with inline image. Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

To send a email with an inline image, the steps followed are:

- Get a Session
- Create a default MimeMessage object and set *From, To, Subject* in the message.
- Create a MimeMultipart object.
- In our example we will have an HTML part and an Image in the email. So first create the HTML content and set it in the multipart object as:

```
// first part (the html)
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hello</H1><img src=\"cid:image\">";
messageBodyPart.setContent(htmlText, "text/html");
// add it
multipart.addBodyPart(messageBodyPart);
```

- Next add the image by creating a Datahandler as follows:

```
// second part (the image)
messageBodyPart = new MimeBodyPart();
DataSource fds = new FileDataSource(
 "/home/manisha/javamail-mini-logo.png");

messageBodyPart.setDataHandler(new DataHandler(fds));
messageBodyPart.setHeader("Content-ID", "<image>");
```

- Next set the multipart in the message as follows:

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

- Send the message using the Transport object.

Create Java Class

- Create a java class file **SendInlineImagesInEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;
import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import javax.mail.BodyPart;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

public class SendInlineImagesInEmail {
  public static void main(String[] args) {
    // Recipient's email ID needs to be mentioned.
    String to = "destinationemail@gmail.com";

    // Sender's email ID needs to be mentioned
    String from = "fromemail@gmail.com";
    final String username = "manishaspatil";//change accordingly
    final String password = "******";//change accordingly

    // Assuming you are sending email through relay.jangosmtp.net
    String host = "relay.jangosmtp.net";

    Properties props = new Properties();
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.starttls.enable", "true");
    props.put("mail.smtp.host", host);
    props.put("mail.smtp.port", "25");

    Session session = Session.getInstance(props,
      new javax.mail.Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
          return new PasswordAuthentication(username, password);
        }
      });
```

```
      try {

         // Create a default MimeMessage object.
         Message message = new MimeMessage(session);

         // Set From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // Set To: header field of the header.
         message.setRecipients(Message.RecipientType.TO,
            InternetAddress.parse(to));

         // Set Subject: header field
         message.setSubject("Testing Subject");

         // This mail has 2 part, the BODY and the embedded image
         MimeMultipart multipart = new MimeMultipart("related");

         // first part (the html)
         BodyPart messageBodyPart = new MimeBodyPart();
         String htmlText = "<H1>Hello</H1><img src=\"cid:image\">";
         messageBodyPart.setContent(htmlText, "text/html");
         // add it
         multipart.addBodyPart(messageBodyPart);

         // second part (the image)
         messageBodyPart = new MimeBodyPart();
         DataSource fds = new FileDataSource(
            "/home/manisha/javamail-mini-logo.png");

         messageBodyPart.setDataHandler(new DataHandler(fds));
         messageBodyPart.setHeader("Content-ID", "<image>");

         // add image to the multipart
         multipart.addBodyPart(messageBodyPart);

         // put everything together
         message.setContent(multipart);
         // Send message
         Transport.send(message);

         System.out.println("Sent message successfully....");

      } catch (MessagingException e) {
         throw new RuntimeException(e);
      }
   }
}
```

As we are using the SMTP server provided by the host provider JangoSMTP, we need to authenticate the username and password. The *javax.mail.PasswordAuthentication* class is used to authenticate the password.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendInlineImagesInEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendInlineImagesInEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendInlineImagesInEmail
```

Verify Output

You should see the following message on the command console:

```
Sent message successfully....
```

As I'm sending an email to my gmail address through JangoSMTP, the following mail would be received in my gmail account inbox:

CHAPTER

9

JavaMail API - Checking Emails

There are two aspects to which needs to understood before proceeding with this chapter. They are **Check** and **Fetch**.

- **Check**ing an email in JavaMail is a process where we open the respective folder in the mailbox and get each message. Here we only check the header of each message i.e the *From, To, subject*. Content is not read.
- **Fetch**ing an email in JavaMail is a process where we open the respective folder in the mailbox and get each message. Alongwith the header we also read the content by recognizing the content-type.

To check or fetch an email using JavaMail API, we would need POP or IMAP servers. To check and fetch the emails, Folder and Store classes are needed. Here we have used GMAIL's POP3 server (pop.gmail.com). In this chapter will learn how to check emails using JavaMail API. Fetching shall be covered in the subsequent chapters. To check emails:

- Get a Session
- Create pop3 Store object and connect with pop server.
- Create folder object. Open the appropriate folder in your mailbox.
- Get your messages.
- Close the Store and Folder objects.

Create Java Class

Create a java class file **CheckingMails**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.Session;
import javax.mail.Store;

public class CheckingMails {

   public static void check(String host, String storeType, String user,
      String password)
   {
```

```
      try {




   //create properties field
      Properties properties = new Properties();

      properties.put("mail.pop3.host", host);
      properties.put("mail.pop3.port", "995");
      properties.put("mail.pop3.starttls.enable", "true");
      Session emailSession = Session.getDefaultInstance(properties);

      //create the POP3 store object and connect with the pop server
      Store store = emailSession.getStore("pop3s");

      store.connect(host, user, password);

      //create the folder object and open it
      Folder emailFolder = store.getFolder("INBOX");
      emailFolder.open(Folder.READ_ONLY);

      // retrieve the messages from the folder in an array and print it
      Message[] messages = emailFolder.getMessages();
      System.out.println("messages.length---" + messages.length);

      for (int i = 0, n = messages.length; i < n; i++) {
         Message message = messages[i];
         System.out.println("---------------------------------");
         System.out.println("Email Number " + (i + 1));
         System.out.println("Subject: " + message.getSubject());
         System.out.println("From: " + message.getFrom()[0]);
         System.out.println("Text: " + message.getContent().toString());
         }

      //close the store and folder objects
      emailFolder.close(false);
      store.close();

      } catch (NoSuchProviderException e) {
         e.printStackTrace();
      } catch (MessagingException e) {
         e.printStackTrace();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
 public static void main(String[] args) {

      String host = "pop.gmail.com";// change accordingly
      String mailStoreType = "pop3";
      String username = "yourmail@gmail.com";// change accordingly
      String password = "*****";// change accordingly
```

```
        check(host, mailStoreType, username, password);
    }
}
```

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class CheckingMails.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: CheckingMails.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: CheckingMails
```

Verify Output

You should see the following message on the command console:

```
messages.length---4
---------------------------------
Email Number 1
Subject: Test Mail--Fetch
From: <abcd@gmail.com>
Text: javax.mail.internet.MimeMultipart@327a5b7f
---------------------------------
Email Number 2
Subject: testing ----checking simple email
From: <abcd@gmail.com>
Text: javax.mail.internet.MimeMultipart@7f0d08bc
---------------------------------
Email Number 3
Subject: Email with attachment
From: <abcd@gmail.com>
Text: javax.mail.internet.MimeMultipart@30b8afce
---------------------------------
Email Number 4
Subject: Email with Inline image
From: <abcd@gmail.com>
Text: javax.mail.internet.MimeMultipart@2d1e165f
```

Here we have printed the number of messages in the INBOX which is 4 in this case. We have also printed Subject, From address and Text for each email message.

CHAPTER

10

JavaMail API - Fetching Emails

In the previous chapter we learnt how to check emails. Now let us see how to fetch each email and read its content.

Let us write a Java class **FetchingEmail** which will read following types of emails:

- Simple email
- Email with attachment
- Email with inline image

Basic steps followed in the code are as below:

- Get the Session object.
- Create POP3 store object and connect to the store.
- Create Folder object and open the appropriate folder in your mailbox.
- Retrieve messages.
- Close the folder and store objects respectively.

Create Java Class

Create a java Create a java class file **FetchingEmail**, contents of which are as below:

```
package com.tutorialspoint;

import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Date;
import java.util.Properties;

import javax.mail.Address;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.NoSuchProviderException;
import javax.mail.Part;
```

```
import javax.mail.Session;
import javax.mail.Store;

public class FetchingEmail {

  public static void fetch(String pop3Host, String storeType, String user,
    String password) {
    try {
      // create properties field
      Properties properties = new Properties();
      properties.put("mail.store.protocol", "pop3");
      properties.put("mail.pop3.host", pop3Host);
      properties.put("mail.pop3.port", "995");
      properties.put("mail.pop3.starttls.enable", "true");
      Session emailSession = Session.getDefaultInstance(properties);
      // emailSession.setDebug(true);

      // create the POP3 store object and connect with the pop server
      Store store = emailSession.getStore("pop3s");

      store.connect(pop3Host, user, password);

      // create the folder object and open it
      Folder emailFolder = store.getFolder("INBOX");
      emailFolder.open(Folder.READ_ONLY);

      BufferedReader reader = new BufferedReader(new InputStreamReader(
          System.in));

      // retrieve the messages from the folder in an array and print it
      Message[] messages = emailFolder.getMessages();
      System.out.println("messages.length---" + messages.length);

      for (int i = 0; i < messages.length; i++) {
        Message message = messages[i];
        System.out.println("---------------------------------");
        writePart(message);
        String line = reader.readLine();
        if ("YES".equals(line)) {
          message.writeTo(System.out);
        } else if ("QUIT".equals(line)) {
          break;
        }
      }

      // close the store and folder objects
      emailFolder.close(false);
      store.close();
```

```
    } catch (NoSuchProviderException e) {
      e.printStackTrace();
    } catch (MessagingException e) {

      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public static void main(String[] args) {

    String host = "pop.gmail.com";// change accordingly
    String mailStoreType = "pop3";
    String username =
      "abc@gmail.com";// change accordingly
    String password = "*****";// change accordingly

    //Call method fetch
    fetch(host, mailStoreType, username, password);

  }

  /*
  * This method checks for content-type
  * based on which, it processes and
  * fetches the content of the message
  */
  public static void writePart(Part p) throws Exception {
    if (p instanceof Message)
      //Call methos writeEnvelope
      writeEnvelope((Message) p);

    System.out.println("----------------------------");
    System.out.println("CONTENT-TYPE: " + p.getContentType());

    //check if the content is plain text
    if (p.isMimeType("text/plain")) {
      System.out.println("This is plain text");
      System.out.println("---------------------------");
      System.out.println((String) p.getContent());
    }
    //check if the content has attachment
    else if (p.isMimeType("multipart/*")) {
      System.out.println("This is a Multipart");
      System.out.println("---------------------------");
      Multipart mp = (Multipart) p.getContent();
      int count = mp.getCount();
      for (int i = 0; i < count; i++)
        writePart(mp.getBodyPart(i));
```

```
}
//check if the content is a nested message
else if (p.isMimeType("message/rfc822")) {
  System.out.println("This is a Nested Message");
  System.out.println("---------------------------");

  writePart((Part) p.getContent());
}
//check if the content is an inline image
else if (p.isMimeType("image/jpeg")) {
  System.out.println("--------> image/jpeg");
  Object o = p.getContent();

  InputStream x = (InputStream) o;
  // Construct the required byte array
  System.out.println("x.length = " + x.available());
  int i = 0;
  byte[] bArray = new byte[x.available()];

  while ((i = (int) ((InputStream) x).available()) > 0) {
    int result = (int) (((InputStream) x).read(bArray));
    if (result == -1)
    break;
  }
  FileOutputStream f2 = new FileOutputStream("/tmp/image.jpg");
  f2.write(bArray);
}
else if (p.getContentType().contains("image/")) {
  System.out.println("content type" + p.getContentType());
  File f = new File("image" + new Date().getTime() + ".jpg");
  DataOutputStream output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f)));
    com.sun.mail.util.BASE64DecoderStream test =
        (com.sun.mail.util.BASE64DecoderStream) p
         .getContent();
  byte[] buffer = new byte[1024];
  int bytesRead;
  while ((bytesRead = test.read(buffer)) != -1) {
    output.write(buffer, 0, bytesRead);
  }
}
else {
  Object o = p.getContent();
  if (o instanceof String) {
    System.out.println("This is a string");
    System.out.println("---------------------------");
    System.out.println((String) o);
  }
  else if (o instanceof InputStream) {
    System.out.println("This is just an input stream");
    System.out.println("---------------------------");
```

```
        InputStream is = (InputStream) o;
        is = (InputStream) o;
        int c;
        while ((c = is.read()) != -1)
          System.out.write(c);
      }
      else {

        System.out.println("This is an unknown type");
        System.out.println("---------------------------");
        System.out.println(o.toString());
      }
    }

  }
  /*
  * This method would print FROM,TO and SUBJECT of the message
  */
  public static void writeEnvelope(Message m) throws Exception {
    System.out.println("This is the message envelope");
    System.out.println("---------------------------");
    Address[] a;

    // FROM
    if ((a = m.getFrom()) != null) {
      for (int j = 0; j < a.length; j++)
      System.out.println("FROM: " + a[j].toString());
    }

    // TO
    if ((a = m.getRecipients(Message.RecipientType.TO)) != null) {
      for (int j = 0; j < a.length; j++)
      System.out.println("TO: " + a[j].toString());
    }

    // SUBJECT
    if (m.getSubject() != null)
      System.out.println("SUBJECT: " + m.getSubject());

  }

}
```

You can set the debug on by uncommenting the statement *emailSession.setDebug(true);*

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class FetchingEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: FetchingEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: FetchingEmail
```

Verify Output

You should see the following message on the command console:

```
messages.length---3
---------------------------------
This is the message envelope
---------------------------
FROM: XYZ <xyz@gmail.com>
TO: ABC <abc@gmail.com>
SUBJECT: Simple Message
----------------------------
CONTENT-TYPE: multipart/alternative; boundary=047d7b343d6ad3e4ea04e8ec6579
This is a Multipart
---------------------------

----------------------------
CONTENT-TYPE: text/plain; charset=ISO-8859-1
This is plain text
---------------------------
Hi am a simple message string....

--
Regards
xyz

This is the message envelope
---------------------------
FROM: XYZ <xyz@gmail.com>
TO: ABC <abc@gmail.com>
SUBJECT: Attachement
----------------------------
CONTENT-TYPE: multipart/mixed; boundary=047d7b343d6a99180904e8ec6751
This is a Multipart
---------------------------

----------------------------
CONTENT-TYPE: text/plain; charset=ISO-8859-1
This is plain text
---------------------------
Hi I've an attachment.Please check

--
Regards
XYZ
```

```
----------------------------
CONTENT-TYPE: application/octet-stream; name=sample_attachement
This is just an input stream
---------------------------
Submit your Tutorials, White Papers and Articles into our Tutorials Directory. This is a tutorials database where
    we are keeping all the tutorials shared by the internet community for the benefit of others.


This is the message envelope
---------------------------
FROM: XYZ <xyz@gmail.com>
TO: ABC <abc@gmail.com>
SUBJECT: Inline Image
----------------------------
CONTENT-TYPE: multipart/related; boundary=f46d04182582be803504e8ece94b
This is a Multipart
---------------------------


----------------------------
CONTENT-TYPE: text/plain; charset=ISO-8859-1
This is plain text
---------------------------
Hi I've an inline image


[image: Inline image 3]

--
Regards
XYZ

----------------------------
CONTENT-TYPE: image/png; name="javamail-mini-logo.png"
content typeimage/png; name="javamail-mini-logo.png"
```

Here you can see there are three emails in our mailbox. First a simple mail with message "Hi am a simple message string....". The second mail has an attachment. The contents of the attachment are also printed as seen above. The third mail has an inline image.

CHAPTER

11

JavaMail API – Authentication

In the previous chapters Checking Emails and Fetching Emails, we passed authorization credentials (user ad password) along with host, when connecting to store of your mailbox. Instead we can configure the *Properties* to have the host, and tell the Session about your custom Authenticator instance. This is shown in the example below:

Create Java Class

We will modify our CheckingMails.java from the chapter Checking Emails. Its contents are as below:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Authenticator;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Store;

public class CheckingMails {

  public static void check(String host, String storeType, String user,
    String password)
  {
    try {

    // create properties field
    Properties properties = new Properties();

    properties.put("mail.pop3s.host", host);
    properties.put("mail.pop3s.port", "995");
    properties.put("mail.pop3s.starttls.enable", "true");

    // Setup authentication, get session
    Session emailSession = Session.getInstance(properties,
      new javax.mail.Authenticator() {
```

```
protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(
               "manishapatil3may@gmail.com", "manisha123");
         }
       });
    // emailSession.setDebug(true);

    // create the POP3 store object and connect with the pop server
    Store store = emailSession.getStore("pop3s");

    store.connect();

    // create the folder object and open it
    Folder emailFolder = store.getFolder("INBOX");
    emailFolder.open(Folder.READ_ONLY);

    // retrieve the messages from the folder in an array and print it
    Message[] messages = emailFolder.getMessages();
    System.out.println("messages.length---" + messages.length);

    for (int i = 0, n = messages.length; i < n; i++) {
       Message message = messages[i];
       System.out.println("---------------------------------");
       System.out.println("Email Number " + (i + 1));
       System.out.println("Subject: " + message.getSubject());
       System.out.println("From: " + message.getFrom()[0]);
       System.out.println("Text: " + message.getContent().toString());
    }
    // close the store and folder objects
    emailFolder.close(false);
    store.close();

    } catch (NoSuchProviderException e) {
       e.printStackTrace();
    } catch (MessagingException e) {
       e.printStackTrace();
    } catch (Exception e) {
       e.printStackTrace();
    }
 }
   public static void main(String[] args) {
   String host = "pop.gmail.com";// change accordingly
    String mailStoreType = "pop3";
    String username = "abc@gmail.com";// change accordingly
    String password = "*****";// change accordingly

    check(host, mailStoreType, username, password);

 }
}
```

You can set the debug on by uncommenting the statement *emailSession.setDebug(true);*

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class CheckingMails.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: CheckingMails.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: CheckingMails
```

Verify Output

You can see a similar message as below on the command console:

```
messages.length---3
---------------------------------
Email Number 1
Subject: Today is a nice day
From: XYZ <xyz@gmail.com>
Text: javax.mail.internet.MimeMultipart@45f676cb
---------------------------------
Email Number 2
Subject: hiiii....
From: XYZ <xyz@gmail.com>
Text: javax.mail.internet.MimeMultipart@37f12d4f
---------------------------------
Email Number 3
Subject: helloo
From: XYZ <xyz@gmail.com>
Text: javax.mail.internet.MimeMultipart@3ad5ba3a
```

CHAPTER

12

JavaMail API - Replying Emails

In this chapter we will see how to reply to an email using JavaMail API. Basic steps followed in the program below are:

- Get the Session object with POP and SMPT server details in the properties. We would need POP details to retrieve messages and SMPT details to send messages.
- Create POP3 store object and connect to the store.
- Create Folder object and open the appropriate folder in your mailbox.
- Retrieve messages.
- Iterate through the messages and type "Y" or "y" if you want to reply.
- Get all information (To,From,Subject, Content) of the message.
- Build the reply message, using Message.reply() method. This method configures a new Message with the proper recipient and subject. The method takes a boolean parameter indicating whether to reply to only the sender (false) or reply to all (true).
- Set From,Text and Reply-to in the message and send it through the instance of Transport object.
- Close the Transport, folder and store objects respectively.

Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

Create Java Class

Create a java class file **ReplyToEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Date;
import java.util.Properties;

import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Store;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
```

```
public class ReplyToEmail {
  public static void main(String args[])
  {
    Date date = null;

    Properties properties = new Properties();
    properties.put("mail.store.protocol", "pop3");
    properties.put("mail.pop3s.host", "pop.gmail.com");
    properties.put("mail.pop3s.port", "995");
    properties.put("mail.pop3.starttls.enable", "true");
    properties.put("mail.smtp.auth", "true");
    properties.put("mail.smtp.starttls.enable", "true");
    properties.put("mail.smtp.host", "relay.jangosmtp.net");
    properties.put("mail.smtp.port", "25");
    Session session = Session.getDefaultInstance(properties);

    // session.setDebug(true);
    try
    {
      // Get a Store object and connect to the current host
      Store store = session.getStore("pop3s");
      store.connect("pop.gmail.com", "xyz@gmail.com",
        "*****");//change the user and password accordingly

      Folder folder = store.getFolder("inbox");
      if (!folder.exists()) {
        System.out.println("inbox not found");
          System.exit(0);
      }
      folder.open(Folder.READ_ONLY);

      BufferedReader reader = new BufferedReader(new InputStreamReader(
        System.in));

      Message[] messages = folder.getMessages();
      if (messages.length != 0) {

        for (int i = 0, n = messages.length; i < n; i++) {
          Message message = messages[i];
          date = message.getSentDate();
          // Get all the information from the message
          String from = InternetAddress.toString(message.getFrom());
          if (from != null) {
            System.out.println("From: " + from);
          }
          String replyTo = InternetAddress.toString(message
            .getReplyTo());
          if (replyTo != null) {
            System.out.println("Reply-to: " + replyTo);
          }
```

```
            String to = InternetAddress.toString(message
              .getRecipients(Message.RecipientType.TO));
            if (to != null) {
              System.out.println("To: " + to);
            }

            String subject = message.getSubject();
            if (subject != null) {
              System.out.println("Subject: " + subject);
            }
            Date sent = message.getSentDate();
            if (sent != null) {
              System.out.println("Sent: " + sent);
            }

            System.out.print("Do you want to reply [y/n] : ");
            String ans = reader.readLine();
            if ("Y".equals(ans) || "y".equals(ans)) {

              Message replyMessage = new MimeMessage(session);
              replyMessage = (MimeMessage) message.reply(false);
              replyMessage.setFrom(new InternetAddress(to));
              replyMessage.setText("Thanks");
              replyMessage.setReplyTo(message.getReplyTo());

              // Send the message by authenticating the SMTP server
              // Create a Transport instance and call the sendMessage
              Transport t = session.getTransport("smtp");
              try {
                //connect to the smpt server using transport instance
                //change the user and password accordingly
                t.connect("abc", "****");
                t.sendMessage(replyMessage,
                  replyMessage.getAllRecipients());
              } finally {
                t.close();
              }
              System.out.println("message replied successfully ....");

              // close the store and folder objects
              folder.close(false);
              store.close();

            } else if ("n".equals(ans)) {
              break;
            }
          }//end of for loop

        } else {
          System.out.println("There is no msg....");
        }
```

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

    }

}
```

You can set the debug on by uncommenting the statement *session.setDebug(true);*

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class ReplyToEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: ReplyToEmail.java
```

Now that the class is compiled, execute the following command to run:

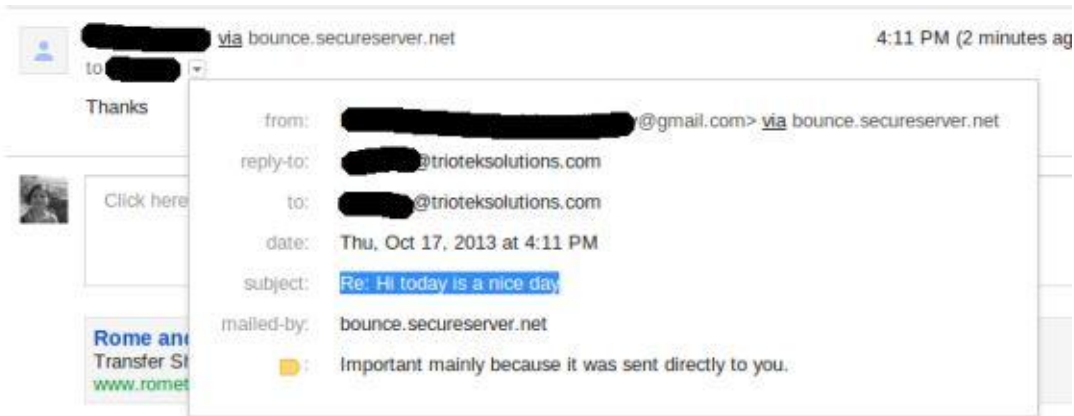
```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: ReplyToEmail
```

Verify Output

You should see the following message on the command console:

```
From: ABC <abc@gmail.com>
Reply-to: abc@trioteksolutions.com
To: XYZ <xyz@gmail.com>
Subject: Hi today is a nice day
Sent: Thu Oct 17 15:58:37 IST 2013
Do you want to reply [y/n] : y
message replied successfully ....
```

Check the inbox to which the mail was sent. In our case the message received looks as below:



CHAPTER

13

JavaMail API - Forwarding Emails

In this chapter we will see how to forward an email using JavaMail API. Basic steps followed in the program below are:

- Get the Session object with POP and SMPT server details in the properties. We would need POP details to retrieve messages and SMPT details to send messages.
- Create POP3 store object and connect to the store.
- Create Folder object and open the appropriate folder in your mailbox.
- Retrieve messages.
- Iterate through the messages and type "Y" or "y" if you want to forward.
- Get all information (To,From,Subject, Content) of the message.
- Build the forward message by working with the parts that make up a message. First part would be the text of the message and a second part would be the message to forward. Combine the two into a multipart. Then you add the multipart to a properly addressed message and send it.
- Close the Transport, folder and store objects respectively.

Here we have used JangoSMPT server via which emails are sent to our destination email address. The setup is explained in the Environment Setup chapter.

Create Java Class

Create a java class file **ForwardEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Date;
import java.util.Properties;

import javax.mail.BodyPart;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.Multipart;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Store;
import javax.mail.Transport;
```

```
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

public class ForwardEmail {

  public static void main(String[] args) {
    Properties properties = new Properties();
    properties.put("mail.store.protocol", "pop3");
    properties.put("mail.pop3s.host", "pop.gmail.com");
    properties.put("mail.pop3s.port", "995");
    properties.put("mail.pop3.starttls.enable", "true");
    properties.put("mail.smtp.auth", "true");
    properties.put("mail.smtp.host", "relay.jangosmtp.net");
    properties.put("mail.smtp.port", "25");
    Session session = Session.getDefaultInstance(properties);
    try {
      // session.setDebug(true);
      // Get a Store object and connect to the current host
      Store store = session.getStore("pop3s");
      store.connect("pop.gmail.com", "xyz@gmail.com",
        "*****");//change the user and password accordingly

      // Create a Folder object and open the folder
      Folder folder = store.getFolder("inbox");
      folder.open(Folder.READ_ONLY);
      BufferedReader reader = new BufferedReader(new InputStreamReader(
        System.in));
      Message[] messages = folder.getMessages();
      if (messages.length != 0) {

      for (int i = 0, n = messages.length; i < n; i++) {
        Message message = messages[i];
        // Get all the information from the message
        String from = InternetAddress.toString(message.getFrom());
        if (from != null) {
          System.out.println("From: " + from);
        }
        String replyTo = InternetAddress.toString(message
          .getReplyTo());
        if (replyTo != null) {
          System.out.println("Reply-to: " + replyTo);
        }
        String to = InternetAddress.toString(message
          .getRecipients(Message.RecipientType.TO));
        if (to != null) {
          System.out.println("To: " + to);
        }
```

```
String subject = message.getSubject();
if (subject != null) {
  System.out.println("Subject: " + subject);
}
Date sent = message.getSentDate();
if (sent != null) {
  System.out.println("Sent: " + sent);
}
System.out.print("Do you want to reply [y/n] : ");
String ans = reader.readLine();
if ("Y".equals(ans) || "y".equals(ans)) {
  Message forward = new MimeMessage(session);
  // Fill in header
  forward.setRecipients(Message.RecipientType.TO,
  InternetAddress.parse(from));
  forward.setSubject("Fwd: " + message.getSubject());
  forward.setFrom(new InternetAddress(to));

  // Create the message part
  MimeBodyPart messageBodyPart = new MimeBodyPart();
  // Create a multipart message
  Multipart multipart = new MimeMultipart();
  // set content
  messageBodyPart.setContent(message, "message/rfc822");
  // Add part to multi part
  multipart.addBodyPart(messageBodyPart);
  // Associate multi-part with message
  forward.setContent(multipart);
  forward.saveChanges();

  // Send the message by authenticating the SMTP server
  // Create a Transport instance and call the sendMessage
  Transport t = session.getTransport("smtp");
  try {
    //connect to the smpt server using transport instance
    //change the user and password accordingly
    t.connect("abc", "*****");
    t.sendMessage(forward, forward.getAllRecipients());
  } finally {
    t.close();
  }
 System.out.println("message forwarded successfully....");

// close the store and folder objects
folder.close(false);
store.close();
}// end if
```

```
        }// end for
    }// end if
    } catch (Exception e) {
      e.printStackTrace();
    }
}

}
```

You can set the debug on by uncommenting the statement *session.setDebug(true);*

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class ForwardEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: ForwardEmail.java
```

Now that the class is compiled, execute the following command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: ForwardEmail
```

Verify Output

You should see the following message on the command console:

```
From: ABC <abc@gmail.com>
Reply-to: abc@trioteksolutions.com
To: XYZ <xyz@gmail.com>
Subject: Hi today is a nice day
Sent: Thu Oct 17 15:58:37 IST 2013
Do you want to reply [y/n] : y
message forwarded successfully....
```

Check the inbox to which the mail was sent. In our case the forwarded message would look as below:

via jangomail.com 4:37 PM (1 minute ago

to me

From:
To: Ma
Cc:
Date:
Subjec

Hello t
--
Regard
XYZ

from: @gmail.com> via jangomail.com

to: @gmail.com>

date: Thu, Oct 17, 2013 at 4:37 PM

subject: Fwd: Hi today is a nice day

mailed-by: jangomail.com

Important mainly because of your interaction with messages in the conversation.

Hi today is a nice day Inbox x

via jangomail.com

to me

---------- Forwarded message ----------
From: @gmail.com>
To: @gmail.com>
Cc:
Date: Thu, 17 Oct 2013 15:58:37 +0530
Subject: Hi today is a nice day

Hello to day is a noce day
--
Regards
XYZ

CHAPTER

14

JavaMail API - Deleting Emails

In this chapter we will see how to delete an email using JavaMail API. Deleting messages involves working with the Flags associated with the messages. There are different flags for different states, some system-defined and some user-defined. The predefined flags are defined in the inner class Flags.Flag and are listed below:

- Flags.Flag.ANSWERED
- Flags.Flag.DELETED
- Flags.Flag.DRAFT
- Flags.Flag.FLAGGED
- Flags.Flag.RECENT
- Flags.Flag.SEEN
- Flags.Flag.USER

POP protocol supports only deleting of the messages.

Basic steps followed in the delete program are:

- Get the Session object with POP and SMPT server details in the properties. We would need POP details to retrieve messages and SMPT details to send messages.
- Create POP3 store object and connect to the store.
- Create Folder object and open the appropriate folder in your mailbox in READ_WRITE mode.
- Retrieves messages from inbox folder.
- Iterate through the messages and type "Y" or "y" if you want to delete the message by invoking the method setFlag(Flags.Flag.DELETED, true) on the Message object.
- The messages marked DELETED are not actually deleted, until we call the expunge() method on the Folder object, or close the folder with expunge set to true.
- Close the store object.

Create Java Class

Create a java class file **ForwardEmail**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Properties;

import javax.mail.Flags;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.Session;
import javax.mail.Store;

public class DeleteEmail {

  public static void delete(String pop3Host, String storeType, String user,
    String password)
  {
    try
    {
      // get the session object
      Properties properties = new Properties();
      properties.put("mail.store.protocol", "pop3");
      properties.put("mail.pop3s.host", pop3Host);
      properties.put("mail.pop3s.port", "995");
      properties.put("mail.pop3.starttls.enable", "true");
      Session emailSession = Session.getDefaultInstance(properties);
      // emailSession.setDebug(true);

      // create the POP3 store object and connect with the pop server
      Store store = emailSession.getStore("pop3s");

      store.connect(pop3Host, user, password);

      // create the folder object and open it
      Folder emailFolder = store.getFolder("INBOX");
      emailFolder.open(Folder.READ_WRITE);

      BufferedReader reader = new BufferedReader(new InputStreamReader(
        System.in));
      // retrieve the messages from the folder in an array and print it
      Message[] messages = emailFolder.getMessages();
      System.out.println("messages.length---" + messages.length);
      for (int i = 0; i < messages.length; i++) {
        Message message = messages[i];
        System.out.println("---------------------------------");
        System.out.println("Email Number " + (i + 1));
        System.out.println("Subject: " + message.getSubject());
        System.out.println("From: " + message.getFrom()[0]);
```

```
            String subject = message.getSubject();
            System.out.print("Do you want to delete this message [y/n] ? ");
            String ans = reader.readLine();
            if ("Y".equals(ans) || "y".equals(ans)) {
               // set the DELETE flag to true
               message.setFlag(Flags.Flag.DELETED, true);
               System.out.println("Marked DELETE for message: " + subject);
            } else if ("n".equals(ans)) {
               break;
            }
         }
         // expunges the folder to remove messages which are marked deleted
         emailFolder.close(true);
         store.close();

      } catch (NoSuchProviderException e) {
         e.printStackTrace();
      } catch (MessagingException e) {
         e.printStackTrace();
      } catch (IOException io) {
         io.printStackTrace();
      }
   }

   public static void main(String[] args) {

      String host = "pop.gmail.com";// change accordingly
      String mailStoreType = "pop3";
      String username = "abc@gmail.com";// change accordingly
      String password = "*****";// change accordingly

      delete(host, mailStoreType, username, password);

   }

}
```

You can set the debug on by uncommenting the statement *emailSession.setDebug(true);*

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class DeleteEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: DeleteEmail.java
```

Now that the class is compiled, execute the following command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: DeleteEmail
```

Verify Output

You should see the following message on the command console:

```
messages.length---1
---------------------------------
Email Number 1
Subject: Testing
From: ABC <abc@gmail.com>
Do you want to delete this message [y/n] ? y
Marked DELETE for message: Testing
```

CHAPTER

15

JavaMail API - Gmail SMPT Server

In all previous chapters we used JangoSMPT server to send emails. In this chapter we will learn about SMPT server provided by Gmail. Gmail (among others) offers use of their public SMTP server free of charge.

Gmail SMTP server details can be found here. As you can see in the details, we can use either TLS or SSL connection to send email via Gmail SMTP server.

The procedure to send email using Gmail SMTP server is similar as explained in chapter Sending Emails, except that we would change the host server. As a pre-requisite the sender email address should be an active gmail account. Let us try an example.

Create Java Class

Create a Java file **SendEmailUsingGMailSMTP**, contents of which are as below:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendEmailUsingGMailSMTP {
  public static void main(String[] args) {
    // Recipient's email ID needs to be mentioned.
    String to = "xyz@gmail.com";//change accordingly

    // Sender's email ID needs to be mentioned
    String from = "abc@gmail.com";//change accordingly
    final String username = "abc";//change accordingly
    final String password = "*****";//change accordingly
```

```
// Assuming you are sending email through relay.jangosmtp.net
String host = "smtp.gmail.com";

Properties props = new Properties();
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");
props.put("mail.smtp.host", host);
props.put("mail.smtp.port", "587");

// Get the Session object.
Session session = Session.getInstance(props,
new javax.mail.Authenticator() {
  protected PasswordAuthentication getPasswordAuthentication() {
    return new PasswordAuthentication(username, password);
  }
});

try {
  // Create a default MimeMessage object.
  Message message = new MimeMessage(session);

  // Set From: header field of the header.
  message.setFrom(new InternetAddress(from));

  // Set To: header field of the header.
  message.setRecipients(Message.RecipientType.TO,
  InternetAddress.parse(to));

  // Set Subject: header field
  message.setSubject("Testing Subject");

  // Now set the actual message
  message.setText("Hello, this is sample for to check send "
    + "email using JavaMailAPI ");

  // Send message
  Transport.send(message);
```

```
        System.out.println("Sent message successfully....");

      } catch (MessagingException e) {
          throw new RuntimeException(e);
      }
   }
}
```

Here the host is set as *smtp.gmail.com* and port is set as *587*. Here we have enabled TLS connection.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendEmailUsingGMailSMTP.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmailUsingGMailSMTP.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmailUsingGMailSMTP
```

Verify Output

You should see the following message on the command console:

```
Sent message successfully....
```

CHAPTER

16

JavaMail API - Folder Management

So far, we've worked in our previous chapters mostly with the INBOX folder. This is the default folder in which most mail resides. Some systems might call it as INBOX and some other might call it by some other name. But,you can always access it from the JavaMail API using the name INBOX.

The JavaMail API represents folders as instances of the abstract Folder class:

```
public abstract class Folder extends Object
```

This class declares methods for requesting named folders from servers, deleting messages from folders, searching for particular messages in folders, listing the messages in a folder, and so forth.

Opening a Folder

We can't create a folder directly as the only constructor in the *Folder* class is *protected*. We can get a *Folder* from:

- a Session
- a Store
- or another Folder

All the above classes have a similar getFolder() method with similar signature:

```
public abstract Folder getFolder(String name) throws MessagingException
```

Some of the methods which help in getting the *Folder* object are:

Method	Description
boolean *exists()*	Checks if the folder really exists. Use this method before getting the Folder object.
abstract void *open*(int mode)	When you get a *Folder*, its closed. Use this method to open it. *mode* can be Folder.READ_ONLY or Folder.READ_WRITE.
abstract boolean *isOpen*()	This method returns *true* if the folder is open, *false* if it's closed
abstract void *close*(boolean expunge)	Closes the folder. If the *expunge* argument is *true*, any deleted messages in the folder are deleted from the actual file on the server. Otherwise, they're simply marked as *deleted*, but the messages can still be undeleted.

Basic Folder Info

Following are some of the methods in Folder class which return basic information about a folder:

Method	Description
abstract String *getName*()	Returns the name of the folder, such as "TutorialsPoint Mail"
abstract String *getFullName*()	Returns the complete hierarchical name from the root such as “books/Manisha/TutorialsPoint Mail”.
URLName *getURLName*()	Return a URLName representing this folder.
abstract Folder *getParent*()	Returns the name of the folder that contains this folder i.e the parent folder. E.g "Manisha" from the previous "TutorialsPoint Mail" example.
abstract int *getType()*	Returns an int indicating whether the folder can contain messages and/or other folders.
int *getMode()*	It returns one of the two named constants Folder.READ_ONLY or Folder.READ_WRITE or -1 when the mode is unknown.
Store *getStore*()	Returns the Store object from which this folder was retrieved.
abstract char *getSeparator*()	Return the delimiter character that separates this Folder's pathname from the names of immediate subfolders.

Managing Folder

Following are some of the methods which help manage the Folder:

Method	Description
abstract boolean *create*(int type)	This creates a new folder in this folder’s Store. Where *type* would be:Folder.HOLDS_MESSAGES or Folder.HOLDS_FOLDERS. Returns *true* if folder is successfully created else returns *false*.
abstract boolean *delete*(boolean recurse)	This deletes the folder only if the folder is closed. Otherwise, it throws an *IllegalStateException*. If *recurse* is *true*, then subfolders are deleted.
abstract boolean *renameTo*(Folder f)	This changes the name of this folder. A folder must be closed to be renamed. Otherwise, an IllegalStateException is thrown.

Managing Messages in Folders

Following are some of the methods that help manage the messages in Folder:

Method	Description
abstract void *appendMessages*(Message[] messages)	As the name implies, the messages in the array are placed at the end of this folder.
void *copyMessages*(Message[] messages, Folder destination)	This copies messages from this folder into a specified folder given as an argument.
abstract Message[] *expunge*()	To delete a message from a folder, set its Flags.Flag.DELETED flag to true. To physically remove deleted messages from a folder, you have to call this method.

Listing the Contents of a Folder

There are four methods to list the folders that a folder contains:

Method	Description
Folder[] *list*()	This returns an array listing the folders that this folder contains.
Folder[] *listSubscribed*()	This returns an array listing all the subscribed folders that this folder contains.
abstract *Folder*[] list(String pattern)	This is similar to the *list()* method except that it allows you to specify a pattern. The pattern is a string giving the name of the folders that match.
Folder[] *listSubscribed*(String pattern)	This is similar to the *listSubscribed()* method except that it allows you to specify a pattern. The pattern is a string giving the name of the folders that match.

Checking for Mail

Method	Description
abstract int *getMessageCount*()	This method can be invoked on an open or closed folder. However, in the case of a closed folder, this method may (or may not) return -1 to indicate that the exact number of messages isn't easily available.
abstract boolean *hasNewMessages*()	This returns *true* if new messages have been added to the folder since it was last opened.
int *getNewMessageCount*()	It returns the new message count by checking messages in the folder whose RECENT flag is set.
int *getUnreadMessageCount*()	This can be invoked on either an open or a closed folder. However, in the case of a closed folder, it may return -1 to indicate that the real answer would be too expensive to obtain.

Getting Messages from Folders

The Folder class provides four methods for retrieving messages from open folders:

Method	Description
abstract Message *getMessage*(int messageNumber)	This returns the nth message in the folder. The first message in the folder is number 1.
Message[] *getMessages*()	This returns an array of *Message* objects representing all the messages in this folder.
Message[] *getMessages*(int start, int end)	This returns an array of *Message* objects from the folder, beginning with start and finishing with end, inclusive.
Message[] *getMessages*(int[] messageNumbers)	This returns an array containing only those messages specifically identified by number in the *messageNumbers* array.
void *fetch*(Message[] messages, FetchProfile fp)	Prefetch the items specified in the FetchProfile for the given Messages. The FetchProfile argument specifies which headers in the messages to prefetch.

Searching Folders

If the server supports searching (as many IMAP servers do and most POP servers don't), it's easy to search a folder for the messages meeting certain criteria. The criteria are encoded in SearchTerm objects. Following are the two search methods:

Method	Description
Message[] *search*(SearchTerm term)	Search this Folder for messages matching the specified search criterion. Returns an array containing the matching messages. Returns an empty array if no matches were found.
Message[] *search*(SearchTerm term, Message[] messages)	Search the given array of messages for those that match the specified search criterion. Returns an array containing the matching messages. Returns an empty array if no matches were found. The the specified Message objects must belong to this folder.

Flags

Flag modification is useful when you need to change flags for the entire set of messages in a Folder. Following are the methods provided in the Folder class:

Method	Description
void *setFlags*(Message[] messages, Flags flag, boolean value)	Sets the specified flags on the messages specified in the array.
void *setFlags*(int start, int end, Flags flag, boolean value)	Sets the specified flags on the messages numbered from start through end, both start and end inclusive.
void *setFlags*(int[] messageNumbers, Flags flag, boolean value)	Sets the specified flags on the messages whose message numbers are in the array.
abstract Flags *getPermanentFlags*()	Returns the flags that this folder supports for all messages.

CHAPTER

17

JavaMail API - Quota Management

A quota in JavaMail is a limited or fixed number or amount of messages in a email store. Each Mail service request counts toward the JavaMail API Calls quota. An email service can apply following quota criterion:

- Maximum size of outgoing mail messages, including attachments.
- Maximum size of incoming mail messages, including attachments.
- Maximum size of message when an administrator is a recipient

For Quota management JavaMail has following classes:

Class	Description
public class Quota	This class represents a set of quotas for a given quota root. Each quota root has a set of resources, represented by the Quota.Resource class. Each resource has a name (for example, "STORAGE"), a current usage, and a usage limit. This has only one method *setResourceLimit(String name, long limit)*.
public static class Quota.Resource	Represents an individual resource in a quota root.
public interface QuotaAwareStore	An interface implemented by Stores that support quotas. The *getQuota* and *setQuota* methods support the quota model defined by the IMAP QUOTA extension. *GmailSSLStore, GmailStore, IMAPSSLStore, IMAPStore* are the known implementing classes of this interface.

Let us see and example in the following sections which checks for mail storage name, limit and its usage.

Create Java Class

Create a java class file **QuotaExample**, the contents of which are as follows:

```
package com.tutorialspoint;

import java.util.Properties;

import javax.mail.Quota;
import javax.mail.Session;
import javax.mail.Store;

import com.sun.mail.imap.IMAPStore;

public class QuotaExample
{
  public static void main(String[] args)
  {
    try
    {
      Properties properties = new Properties();
      properties.put("mail.store.protocol", "imaps");
      properties.put("mail.imaps.port", "993");
      properties.put("mail.imaps.starttls.enable", "true");
      Session emailSession = Session.getDefaultInstance(properties);
      // emailSession.setDebug(true);

      // create the IMAP3 store object and connect with the pop server
      Store store = emailSession.getStore("imaps");

      //change the user and password accordingly
      store.connect("imap.gmail.com", "abc@gmail.com", "*****");
      IMAPStore imapStore = (IMAPStore) store;
      System.out.println("imapStore ---" + imapStore);

      //get quota
      Quota[] quotas = imapStore.getQuota("INBOX");
      //Iterate through the Quotas
      for (Quota quota : quotas) {
        System.out.println(String.format("quotaRoot:'%s'",
          quota.quotaRoot));
        //Iterate through the Quota Resource
        for (Quota.Resource resource : quota.resources) {
          System.out.println(String.format(
            "name:'%s', limit:'%s', usage:'%s'", resource.name,
            resource.limit, resource.usage));
        }
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Here are connection to the gmail service via IMAP (imap.gmail.com) server, as IMAPStore implements the QuotaAwareStore. Once you get the Store object, fetch the Quota array and iterate through it and print the relevant information.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class QuotaExample.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: QuotaExample.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: QuotaExample
```

Verify Output

You should see a similar message on the command console:

```
imapStore ---imaps://abc%40gmail.com@imap.gmail.com

quotaRoot:''

name:'STORAGE', limit:'15728640', usage:'513'
```

CHAPTER

18

JavaMail API - Bounced Messages

A message can be bounced for several reasons. This problem is discussed in depth at rfc1211. Only a server can determine the existence of a particular mailbox or user name. When the server detects an error, it will return a message indicating the reason for the failure to the sender of the original message.

There are many Internet standards covering Delivery Status Notifications but a large number of servers don't support these new standards, instead using ad hoc techniques for returning such failure messages. Hence it get very difficult to correlate the *bounced* message with the original message that caused the problem.

JavaMail includes support for parsing Delivery Status Notifications. There are a number of techniques and heuristics for dealing with this problem. One of the techniques being Variable Envelope Return Paths. You can set the return path in the enveloper as shown in the example below. This is the address where bounce mails are sent to. You may want to set this to a generic address, different than the From: header, so you can process remote bounces. This done by setting *mail.smtp.from* property in JavaMail.

Create Java Class

Create a java class file **SendEmail**, the contents of which are as follows:

```
import java.util.Properties;

import javax.mail.Message;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendEmail {
  public static void main(String[] args) throws Exception {
    String smtpServer = "smtp.gmail.com";
    int port = 587;
    final String userid = "youraddress";//change accordingly
    final String password = "*****";//change accordingly
    String contentType = "text/html";
    String subject = "test: bounce an email to a different address " +
                              "from the sender";
    String from = "youraddress@gmail.com";
    String to = "bouncer@fauxmail.com";//some invalid address
    String bounceAddr = "toaddress@gmail.com";//change accordingly
    String body = "Test: get message to bounce to a separate email address";
```

```
      Properties props = new Properties();

      props.put("mail.smtp.auth", "true");
      props.put("mail.smtp.starttls.enable", "true");
      props.put("mail.smtp.host", smtpServer);
      props.put("mail.smtp.port", "587");
      props.put("mail.transport.protocol", "smtp");
      props.put("mail.smtp.from", bounceAddr);

      Session mailSession = Session.getInstance(props,
        new javax.mail.Authenticator() {
          protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(userid, password);
          }
        });

      MimeMessage message = new MimeMessage(mailSession);
      message.addFrom(InternetAddress.parse(from));
      message.setRecipients(Message.RecipientType.TO, to);
      message.setSubject(subject);
      message.setContent(body, contentType);

      Transport transport = mailSession.getTransport();
      try {
        System.out.println("Sending ....");
        transport.connect(smtpServer, port, userid, password);
        transport.sendMessage(message,
          message.getRecipients(Message.RecipientType.TO));
        System.out.println("Sending done ...");
      } catch (Exception e) {
        System.err.println("Error Sending: ");
        e.printStackTrace();

      }
      transport.close();
    }// end function main()
}
```

Here we can see that the property *mail.smtp.from* is set different from the *from* address.

Compile and Run

Now that our class is ready, let us compile the above class. I've saved the class SendEmail.java to directory : **/home/manisha/JavaMailAPIExercise**. We would need the jars *javax.mail.jar* and *activation.jar* in the classpath. Execute the command below to compile the class (both the jars are placed in /home/manisha/ directory) from command prompt:

```
javac -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmail.java
```

Now that the class is compiled, execute the below command to run:

```
java -cp /home/manisha/activation.jar:/home/manisha/javax.mail.jar: SendEmail
```

Verify Output

You should see the following message on the command console:

```
Sending ....

Sending done ...
```

CHAPTER

19

JavaMail API - SMTP Servers

SMTP is an acronym for **Simple Mail Transfer Protocol**. It is an Internet standard for electronic mail (e-mail) transmission across Internet Protocol (IP) networks. SMTP uses TCP port 25. SMTP connections secured by SSL are known by the shorthand SMTPS, though SMTPS is not a protocol in its own right.

JavaMail API has package ***com.sun.mail.smtp*** which act as SMTP protocol provider to access an SMTP server. Following table lists the classes included in this package:

Class	Description
SMTPMessage	This class is a specialization of the MimeMessage class that allows you to specify various SMTP options and parameters that will be used when this message is sent over SMTP.
SMTPSSLTransport	This class implements the Transport abstract class using SMTP over SSL for message submission and transport.
SMTPTransport	This class implements the Transport abstract class using SMTP for message submission and transport.

The following table lists the exceptions thrown:

Exception	Description
SMTPAddressFailedException	This exception is thrown when the message cannot be sent.
SMTPAddressSucceededException	This exception is chained off a SendFailedException when the *mail.smtp.reportsuccess* property is true.
SMTPSenderFailedException	This exception is thrown when the message cannot be sent.
SMTPSendFailedException	This exception is thrown when the message cannot be sent.The exception includes the sender's address, which the mail server rejected.

The ***com.sun.mail.smtp*** provider use SMTP Authentication optionally. To use SMTP authentication you'll need to set the *mail.smtp.auth* property or provide the SMTP Transport with a username and password when connecting to the SMTP server. You can do this using one of the following approaches:

- Provide an Authenticator object when creating your mail Session and provide the username and password information during the Authenticator callback. *mail.smtp.user* property can be set to provide a default username for the callback, but the password will still need to be supplied explicitly. This approach allows you to use the static Transport send method to send messages. For example:

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

- Call the Transport connect method explicitly with username and password arguments. For example:

```
Transport tr = session.getTransport("smtp");
tr.connect(smtphost, username, password);
msg.saveChanges();
tr.sendMessage(msg, msg.getAllRecipients());
tr.close();
```

The SMTP protocol provider supports the following properties, which may be set in the JavaMail Session object. The properties are always set as strings. For example:

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

Here the **Type** column describes how the string is interpreted.

Name	Type	Description
mail.smtp.user	String	Default user name for SMTP.
mail.smtp.host	String	The SMTP server to connect to.
mail.smtp.port	int	The SMTP server port to connect to, if the connect() method doesn't explicitly specify one. Defaults to 25.
mail.smtp.connectiontimeout	int	Socket connection timeout value in milliseconds. Default is infinite timeout.
mail.smtp.timeout	int	Socket I/O timeout value in milliseconds. Default is infinite timeout.
mail.smtp.from	String	Email address to use for SMTP MAIL command. This sets the envelope return address. Defaults to msg.getFrom() or InternetAddress.getLocalAddress().
mail.smtp.localhost	String	Local host name used in the SMTP HELO or EHLO command. Defaults to InetAddress.getLocalHost().getHostName(). Should not normally need to be set if your JDK and your name service are configured properly.
mail.smtp.localaddress	String	Local address (host name) to bind to when creating the SMTP socket. Defaults to the address picked by the Socket class. Should not normally need to be set.
mail.smtp.localport	int	Local port number to bind to when creating the SMTP socket. Defaults to the port number picked by the Socket class.
mail.smtp.ehlo	boolean	If false, do not attempt to sign on with the EHLO command. Defaults to true.

mail.smtp.auth	boolean	If true, attempt to authenticate the user using the AUTH command. Defaults to false.
mail.smtp.auth.mechanisms	String	If set, lists the authentication mechanisms to consider. Only mechanisms supported by the server and supported by the current implementation will be used. The default is "LOGIN PLAIN DIGEST-MD5 NTLM", which includes all the authentication mechanisms supported by the current implementation.
mail.smtp.auth.login.disable	boolean	If true, prevents use of the AUTH LOGIN command. Default is false.
mail.smtp.auth.plain.disable	boolean	If true, prevents use of the AUTH PLAIN command. Default is false.
mail.smtp.auth.digest-md5.disable	boolean	If true, prevents use of the AUTH DIGEST-MD5 command. Default is false.
mail.smtp.auth.ntlm.disable	boolean	If true, prevents use of the AUTH NTLM command. Default is false.
mail.smtp.auth.ntlm.domain	String	The NTLM authentication domain.
mail.smtp.auth.ntlm.flags	int	NTLM protocol-specific flags.
mail.smtp.submitter	String	The submitter to use in the AUTH tag in the MAIL FROM command. Typically used by a mail relay to pass along information about the original submitter of the message.
mail.smtp.dsn.notify	String	The NOTIFY option to the RCPT command. Either NEVER, or some combination of SUCCESS, FAILURE, and DELAY (separated by commas).
mail.smtp.dsn.ret	String	The RET option to the MAIL command. Either FULL or HDRS.
mail.smtp.sendpartial	boolean	If set to true, and a message has some valid and some invalid addresses, send the message anyway, reporting the partial failure with a SendFailedException. If set to false (the default), the message is not sent to any of the recipients if there is an invalid recipient address.
mail.smtp.sasl.enable	boolean	If set to true, attempt to use the javax.security.sasl package to choose an authentication mechanism for login. Defaults to false.
mail.smtp.sasl.mechanisms	String	A space or comma separated list of SASL mechanism names to try to use.
mail.smtp.sasl.authorizationid	String	The authorization ID to use in the SASL authentication. If not set, the authentication ID (user name) is used.
mail.smtp.sasl.realm	String	The realm to use with DIGEST-MD5 authentication.
mail.smtp.quitwait	boolean	If set to false, the QUIT command is sent and the connection is immediately closed. If set to true (the default), causes the transport to wait for the response to the QUIT command.
mail.smtp.reportsuccess	boolean	If set to true, causes the transport to include an SMTPAddressSucceededException for each address that is successful.
mail.smtp.socketFactory	SocketFactory	If set to a class that implements the javax.net.SocketFactory interface, this class will be used to create SMTP sockets.
mail.smtp.socketFactory.class	String	If set, specifies the name of a class that implements the javax.net.SocketFactory interface. This class will be used to create SMTP sockets.

mail.smtp.socketFactory.fallback	boolean	If set to true, failure to create a socket using the specified socket factory class will cause the socket to be created using the java.net.Socket class. Defaults to true.
mail.smtp.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. If not set, the default port will be used.
mail.smtp.ssl.enable	boolean	If set to true, use SSL to connect and use the SSL port by default. Defaults to false for the "smtp" protocol and true for the "smtps" protocol.
mail.smtp.ssl.checkserveridentity	boolean	If set to true, checks the server identity as specified by RFC 2595. Defaults to false.
mail.smtp.ssl.trust	String	If set, and a socket factory hasn't been specified, enables use of a MailSSLSocketFactory. If set to "*", all hosts are trusted. If set to a whitespace separated list of hosts, those hosts are trusted. Otherwise, trust depends on the certificate the server presents.
mail.smtp.ssl.socketFactory	SSLSocketFactory	If set to a class that extends the javax.net.ssl.SSLSocketFactory class, this class will be used to create SMTP SSL sockets.
mail.smtp.ssl.socketFactory.class	String	If set, specifies the name of a class that extends the javax.net.ssl.SSLSocketFactory class. This class will be used to create SMTP SSL sockets.
mail.smtp.ssl.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. If not set, the default port will be used.
mail.smtp.ssl.protocols	string	Specifies the SSL protocols that will be enabled for SSL connections. The property value is a whitespace separated list of tokens acceptable to the javax.net.ssl.SSLSocket.setEnabledProtocols method.
mail.smtp.starttls.enable	boolean	If true, enables the use of the STARTTLS command (if supported by the server) to switch the connection to a TLS-protected connection before issuing any login commands. Defaults to false.
mail.smtp.starttls.required	boolean	If true, requires the use of the STARTTLS command. If the server doesn't support the STARTTLS command, or the command fails, the connect method will fail. Defaults to false.
mail.smtp.socks.host	string	Specifies the host name of a SOCKS5 proxy server that will be used for connections to the mail server.
mail.smtp.socks.port	string	Specifies the port number for the SOCKS5 proxy server. This should only need to be used if the proxy server is not using the standard port number of 1080.
mail.smtp.mailextension	String	Extension string to append to the MAIL command.
mail.smtp.userset	boolean	If set to true, use the RSET command instead of the NOOP command in the isConnected method. In some cases sendmail will respond slowly after many NOOP commands; use of RSET avoids this sendmail issue. Defaults to false.

In general, applications should not need to use the classes in this package directly. Instead, they should use the APIs defined by javax.mail package (and subpackages). Say for example applications should never construct instances of SMTPTransport directly. Instead, they should use the Session method getTransport to acquire an appropriate Transport object.

Examples to use SMPT server is demonstrated in chapter Sending Emails.

CHAPTER

20

JavaMail API - IMAP Servers

IMAP is Acronym for **Internet Message Access Protocol**. It is an Application Layer Internet protocol that allows an e-mail client to access e-mail on a remote mail server. An IMAP server typically listens on well-known port 143. IMAP over SSL (IMAPS) is assigned to port number 993.

IMAP supports both on-line and off-line modes of operation. E-mail clients using IMAP generally leave messages on the server until the user explicitly deletes them.

Package **com.sun.mail.imap** is an IMAP protocol provider for the JavaMail API that provides access to an IMAP message store. The table below lists the interface and classes of this provider:

Class/Interface	Description
IMAPFolder.ProtocolCommand	This a simple *interface* for user-defined IMAP protocol commands.
ACL	This is a class. An access control list entry for a particular authentication identifier (user or group).
IMAPFolder	This class implements an IMAP folder.
IMAPFolder.FetchProfileItem	This a class for fetching headers.
IMAPMessage	This class implements an ReadableMime object.
IMAPMessage.FetchProfileCondition	This class implements the test to be done on each message in the folder.
IMAPSSLStore	This class provides access to an IMAP message store over SSL.
IMAPStore	This class provides access to an IMAP message store.
Rights	This class represents the set of rights for an authentication identifier (for instance, a user or a group).
Rights.Right	This inner class represents an individual right.
SortTerm	A particular sort criteria, as defined by RFC 5256.

Some points to be noted above this provider:

- This provider supports both the IMAP4 and IMAP4rev1 protocols.
- A connected IMAPStore maintains a pool of IMAP protocol objects for use in communicating with the IMAP server. As folders are opened and new IMAP protocol objects are needed, the IMAPStore will provide them from the connection pool, or create them if none are available. When a folder is closed, its IMAP protocol object is returned to the connection pool if the pool .
- The connected IMAPStore object may or may not maintain a separate IMAP protocol object that provides the store a dedicated connection to the IMAP server.

The IMAP protocol provider supports the following properties, which may be set in the JavaMail Session object. The properties are always set as strings; the **Type** column describes how the string is interpreted.

Name	Type	Description
mail.imap.user	String	Default user name for IMAP.
mail.imap.host	String	The IMAP server to connect to.
mail.imap.port	int	The IMAP server port to connect to, if the connect() method doesn't explicitly specify one. Defaults to 143.
mail.imap.partialfetch	boolean	Controls whether the IMAP partial-fetch capability should be used. Defaults to true.
mail.imap.fetchsize	int	Partial fetch size in bytes. Defaults to 16K.
mail.imap.ignorebodystructuresize	boolean	The IMAP BODYSTRUCTURE response includes the exact size of each body part. Normally, this size is used to determine how much data to fetch for each body part. Defaults to false.
mail.imap.connectiontimeout	int	Socket connection timeout value in milliseconds. Default is infinite timeout.
mail.imap.timeout	int	Socket I/O timeout value in milliseconds. Default is infinite timeout.
mail.imap.statuscachetimeout	int	Timeout value in milliseconds for cache of STATUS command response. Default is 1000 (1 second). Zero disables cache.
mail.imap.appendbuffersize	int	Maximum size of a message to buffer in memory when appending to an IMAP folder.
mail.imap.connectionpoolsize	int	Maximum number of available connections in the connection pool. Default is 1.
mail.imap.connectionpooltimeout	int	Timeout value in milliseconds for connection pool connections. Default is 45000 (45 seconds).
mail.imap.separatestoreconnection	boolean	Flag to indicate whether to use a dedicated store connection for store commands. Default is false.
mail.imap.auth.login.disable	boolean	If true, prevents use of the non-standard AUTHENTICATE LOGIN command, instead using the plain LOGIN command. Default is false.
mail.imap.auth.plain.disable	boolean	If true, prevents use of the AUTHENTICATE PLAIN command. Default is false.
mail.imap.auth.ntlm.disable	boolean	If true, prevents use of the AUTHENTICATE NTLM command. Default is false.
mail.imap.proxyauth.user	String	If the server supports the PROXYAUTH extension, this property specifies the name of the user to act as. Authenticate to the server using the administrator's credentials. After authentication, the IMAP provider will issue the PROXYAUTH command with the user name specified in this property.
mail.imap.localaddress	String	Local address (host name) to bind to when creating the IMAP socket. Defaults to the address picked by the Socket class.
mail.imap.localport	int	Local port number to bind to when creating the IMAP socket. Defaults to the port number picked by the Socket class.
mail.imap.sasl.enable	boolean	If set to true, attempt to use the javax.security.sasl package to choose an authentication mechanism for login. Defaults to false.
mail.imap.sasl.mechanisms	String	A space or comma separated list of SASL mechanism names to try to use.

mail.imap.sasl.authorizationid	String	The authorization ID to use in the SASL authentication. If not set, the authentication ID (user name) is used.
mail.imap.sasl.realm	String	The realm to use with SASL authentication mechanisms that require a realm, such as DIGEST-MD5.
mail.imap.auth.ntlm.domain	String	The NTLM authentication domain.
mail.imap.auth.ntlm.flags	int	NTLM protocol-specific flags.
mail.imap.socketFactory	SocketFactory	If set to a class that implements the javax.net.SocketFactory interface, this class will be used to create IMAP sockets.
mail.imap.socketFactory.class	String	If set, specifies the name of a class that implements the javax.net.SocketFactory interface. This class will be used to create IMAP sockets.
mail.imap.socketFactory.fallback	boolean	If set to true, failure to create a socket using the specified socket factory class will cause the socket to be created using the java.net.Socket class. Defaults to true.
mail.imap.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. Default port is used when not set.
mail.imap.ssl.enable	boolean	If set to true, use SSL to connect and use the SSL port by default. Defaults to false for the "imap" protocol and true for the "imaps" protocol.
mail.imap.ssl.checkserveridentity	boolean	If set to true, check the server identity as specified by RFC 2595. Defaults to false.
mail.imap.ssl.trust	String	If set, and a socket factory hasn't been specified, enables use of a MailSSLSocketFactory. If set to "*", all hosts are trusted. If set to a whitespace separated list of hosts, those hosts are trusted. Otherwise, trust depends on the certificate the server presents.
mail.imap.ssl.socketFactory	SSLSocketFactory	If set to a class that extends the javax.net.ssl.SSLSocketFactory class, this class will be used to create IMAP SSL sockets.
mail.imap.ssl.socketFactory.class	String	If set, specifies the name of a class that extends the javax.net.ssl.SSLSocketFactory class. This class will be used to create IMAP SSL sockets.
mail.imap.ssl.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. If not set, the default port will be used.
mail.imap.ssl.protocols	string	Specifies the SSL protocols that will be enabled for SSL connections. The property value is a whitespace separated list of tokens acceptable to the javax.net.ssl.SSLSocket.setEnabledProtocols method.
mail.imap.starttls.enable	boolean	If true, enables the use of the STARTTLS command (if supported by the server) to switch the connection to a TLS-protected connection before issuing any login commands. Default is false.

mail.imap.starttls.required	boolean	If true, requires the use of the STARTTLS command. If the server doesn't support the STARTTLS command, or the command fails, the connect method will fail. Defaults to false.
mail.imap.socks.host	string	Specifies the host name of a SOCKS5 proxy server that will be used for connections to the mail server.
mail.imap.socks.port	string	Specifies the port number for the SOCKS5 proxy server. This should only need to be used if the proxy server is not using the standard port number of 1080.
mail.imap.minidletime	int	This property sets the delay in milliseconds. If not set, the default is 10 milliseconds.
mail.imap.enableimapevents	boolean	Enable special IMAP-specific events to be delivered to the Store's ConnectionListener. If true, unsolicited responses received during the Store's idle method will be sent as ConnectionEvents with a type of IMAPStore.RESPONSE. The event's message will be the raw IMAP response string. By default, these events are not sent.
mail.imap.folder.class	String	Class name of a subclass of com.sun.mail.imap.IMAPFolder. The subclass can be used to provide support for additional IMAP commands. The subclass must have public constructors of the form public MyIMAPFolder(String fullName, char separator, IMAPStore store, Boolean isNamespace) and public MyIMAPFolder(ListInfo li, IMAPStore store)

In general, applications should not need to use the classes in this package directly. Instead, they should use the APIs defined by javax.mail package (and subpackages). Applications should never construct instances of IMAPStore or IMAPFolder directly. Instead, they should use the Session method getStore to acquire an appropriate Store object, and from that acquire Folder objects.

Examples to use IMAP server is demonstrated in chapter Quota Management.

CHAPTER

21

JavaMail API - POP3 Servers

Post Office Protocol (POP) is an application-layer Internet standard protocol used by local e-mail clients to retrieve e-mail from a remote server over a TCP/IP connection. POP supports simple download-and-delete requirements for access to remote mailboxes. A POP3 server listens on well-known port 110.

Package **com.sun.mail.pop3** is a POP3 protocol provider for the JavaMail API that provides access to a POP3 message store. The table below lists the classes in this package:

Name	Description
POP3Folder	A POP3 Folder (can only be "INBOX").
POP3Message	A POP3 Message.
POP3SSLStore	A POP3 Message Store using SSL.
POP3Store	A POP3 Message Store.

Some points to be noted above this provider:

- POP3 provider supports only a single folder named **INBOX**. Due to the limitations of the POP3 protocol, many of the JavaMail API capabilities like event notification, folder management, flag management, etc. are not allowed.
- The POP3 provider is accessed through the JavaMail APIs by using the protocol name *pop3* or a URL of the form *pop3://user:password@host:port/INBOX"*.
- POP3 supports no permanent flags. For example the *Flags.Flag.RECENT* flag will never be set for POP3 messages. It's up to the application to determine which messages in a POP3 mailbox are *new*.
- POP3 does not support the Folder.expunge() method. To delete and expunge messages, set the Flags.Flag.DELETED flag on the messages and close the folder using the Folder.close(true) method.
- POP3 does not provide a *received date*, so the getReceivedDate method will return null.
- When the headers of a POP3 message are accessed, the POP3 provider uses the TOP command to fetch all headers, which are then cached.
- When the content of a POP3 message is accessed, the POP3 provider uses the RETR command to fetch the entire message.
- The *POP3Message.invalidate* method can be used to invalidate cached data without closing the folder.

The POP3 protocol provider supports the following properties, which may be set in the JavaMail Session object. The properties are always set as strings; the Type column describes how the string is interpreted.

Name	Type	Description
mail.pop3.user	String	Default user name for POP3.
mail.pop3.host	String	The POP3 server to connect to.
mail.pop3.port	int	The POP3 server port to connect to, if the connect() method doesn't explicitly specify one. Defaults to 110.
mail.pop3.connectiontimeout	int	Socket connection timeout value in milliseconds. Default is infinite timeout.
mail.pop3.timeout	int	Socket I/O timeout value in milliseconds. Default is infinite timeout.
mail.pop3.rsetbeforequit	boolean	Send a POP3 RSET command when closing the folder, before sending the QUIT command. Default is false.
mail.pop3.message.class	String	Class name of a subclass of com.sun.mail.pop3.POP3Message. The subclass can be used to handle (for example) non-standard Content-Type headers. The subclass must have a public constructor of the form MyPOP3Message(Folder f, int msgno) throws MessagingException.
mail.pop3.localaddress	String	Local address (host name) to bind to when creating the POP3 socket. Defaults to the address picked by the Socket class.
mail.pop3.localport	int	Local port number to bind to when creating the POP3 socket. Defaults to the port number picked by the Socket class.
mail.pop3.apop.enable	boolean	If set to true, use APOP instead of USER/PASS to login to the POP3 server, if the POP3 server supports APOP. APOP sends a digest of the password rather than the clear text password. Defaults to false.
mail.pop3.socketFactory	SocketFactory	If set to a class that implements the javax.net.SocketFactory interface, this class will be used to create POP3 sockets.
mail.pop3.socketFactory.class	String	If set, specifies the name of a class that implements the javax.net.SocketFactory interface. This class will be used to create POP3 sockets.
mail.pop3.socketFactory.fallback	boolean	If set to true, failure to create a socket using the specified socket factory class will cause the socket to be created using the java.net.Socket class. Defaults to true.
mail.pop3.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. If not set, the default port will be used.
mail.pop3.ssl.enable	boolean	If set to true, use SSL to connect and use the SSL port by default. Defaults to false for the "pop3" protocol and true for the "pop3s" protocol.
mail.pop3.ssl.checkserveridentity	boolean	If set to true, check the server identity as specified by RFC 2595. Defaults to false.
mail.pop3.ssl.trust	String	If set, and a socket factory hasn't been specified, enables use of a MailSSLSocketFactory. If set to "*", all hosts are trusted. If set to a whitespace separated list of hosts, those hosts are trusted. Otherwise, trust depends on the certificate the server presents.

mail.pop3.ssl.socketFactory	SSLSocketFactory	If set to a class that extends the javax.net.ssl.SSLSocketFactory class, this class will be used to create POP3 SSL sockets.
mail.pop3.ssl.socketFactory.class	String	If set, specifies the name of a class that extends the javax.net.ssl.SSLSocketFactory class. This class will be used to create POP3 SSL sockets.
mail.pop3.ssl.socketFactory.port	int	Specifies the port to connect to when using the specified socket factory. If not set, the default port will be used.
mail.pop3.ssl.protocols	string	Specifies the SSL protocols that will be enabled for SSL connections. The property value is a whitespace separated list of tokens acceptable to the javax.net.ssl.SSLSocket.setEnabledProtocols method.
mail.pop3.starttls.enable	boolean	If true, enables the use of the STLS command (if supported by the server) to switch the connection to a TLS-protected connection before issuing any login commands. Defaults to false.
mail.pop3.starttls.required	boolean	If true, requires the use of the STLS command. If the server doesn't support the STLS command, or the command fails, the connect method will fail. Defaults to false.
mail.pop3.socks.host	string	Specifies the host name of a SOCKS5 proxy server that will be used for connections to the mail server.
mail.pop3.socks.port	string	Specifies the port number for the SOCKS5 proxy server.
mail.pop3.disabletop	boolean	If set to true, the POP3 TOP command will not be used to fetch message headers. Defaults to false.
mail.pop3.forgettopheaders	boolean	If set to true, the headers that might have been retrieved using the POP3 TOP command will be forgotten and replaced by headers retrieved as part of the POP3 RETR command. Defaults to false.
mail.pop3.filecache.enable	boolean	If set to true, the POP3 provider will cache message data in a temporary file rather than in memory. Messages are only added to the cache when accessing the message content. Message headers are always cached in memory (on demand). The file cache is removed when the folder is closed or the JVM terminates. Defaults to false.
mail.pop3.filecache.dir	String	If the file cache is enabled, this property can be used to override the default directory used by the JDK for temporary files.
mail.pop3.cachewriteto	boolean	Controls the behavior of the writeTo method on a POP3 message object. If set to true, and the message content hasn't yet been cached, and ignoreList is null, the message is cached before being written. Otherwise, the message is streamed directly to the output stream without being cached. Defaults to false.
mail.pop3.keepmessagecontent	boolean	If this property is set to true, a hard reference to the cached content will be kept, preventing the memory from being reused until the folder is closed or the cached content is explicitly invalidated (using the invalidate method). Defaults to false.

In general, applications should not use the classes in this package directly. Instead, they should use the APIs defined by *javax.mail* package (and subpackages). Applications should never construct instances of *POP3Store* or *POP3Folder* directly. Instead, they should use the Session method getStore to acquire an appropriate Store object, and from that acquire Folder objects.

Examples to use POP3 server is demonstrated in chapter Checking Emails.