James: The Java Apache Mail Enterprise Server

by Barry Burd and Michael P. Redlich

Introduction

James is an open source, 100% pure Java e-mail server developed by the Apache Software Foundation. James has all the qualities of an industrial-strength e-mail server, and a few additional qualities that make James unique.

James complements the JavaMail 1.3 API in a chain of command that works something like this:

- A user interacts with an interface that's created using JavaMail. This interface is called the *mail user agent* (MUA). Think of the MUA as an e-mail client, like Eudora or Outlook Express.
- The MUA talks to a *mail transfer agent* (MTA). As an MTA, the James server supports the SMTP and POP3 e-mail protocols, sending e-mail messages to their appointed destinations. James also includes support for IMAP, NNTP, LDAP, and other protocols.

This article explains why you should consider using James for your e-mail server needs, shows you how to get started with James, and describes one of James's main features: the *Mailet API*.

Why James?

James is a complete, portable, and standalone, enterprise e-mail server solution. Let's take a peek at what makes James stand out in the crowd:

Portability

James is *portable* because it's a 100% pure Java. With James you can create custom e-mail applications on Windows, on UNIX, on cell phones, on weed whackers, on anything.

Complete and Standalone

James is *complete* because it doesn't require additional support from other servers or applications. This makes James *standalone* because it can handle both mail transport and storage as a single server application. Other e-mail servers may rely on required third party applications for getting the functionality you need. James contains functionality from other Apache projects, but everything is packaged as one application.

Abstraction

James follows a loosely coupled plug-in design. For some products, phrases like "loosely coupled" and "separation of *this* from *that*" are just hype. But James separates things intelligently. With James, each protocol plugs into the messaging framework. So, for example, James supports SMTP. But you can unplug SMTP, plug Jabber in place of SMTP, and have a nice, portable instant messaging application.

In fact, James uses abstraction from top to bottom. Resources such as JDBC (for spooling or user accounts stored in databases) are abstracted just as protocols are. James takes advantage of the defined interfaces for each of these resources to maintain modularity and good design.

Multi-Threaded

James is fully *multi-threaded* for performance and scalability. James is based on Apache Avalon, an open source server framework that provides a common method for developing and implementing server components. The Avalon Framework (AF) supports Inversion of Control (IoC). Best of all, you don't need to write code to use Avalon Framework components. All you do is use a container such as Avalon Phoenix or Excalibur Fortress. The container does the work for you.

(Oops! Apache has closed its Avalon project! But don't worry. All is well with James. The James project committee assures everyone that James will press on without Avalon. Whew!)

Mailet Support

James's Mailet API allows you to customize e-mail processing for your enterprise's needs. James has a number of pre-defined mailet classes. For example, the `AddFooter` class appends specified text to the bottom of message body.

The James documentation gives you the complete list of pre-defined mailets. But, of course, you can write your own mailets. All you have to do is to extend the `GenericMailet` and `GenericMatcher` classes. For more information, see the "Mailets and Matchers" section later in this article.

Reliability and Scalability

James was first released in 1999. Since then, a number of organizations have implemented James in production environments. The James Wiki page lists many of these organizations. One organization designed and implemented a scalability test on James with a result of 2600 e-mail messages per minute. Now that's impressive!

Price

Best of all, James is *free*, and that is certainly good news for organizations and companies with small IT budgets. James isn't an expensive e-mail server such as Microsoft Exchange or Lotus Domino, but James rivals other e-mail server solutions in robustness and scalability.

Getting Started

Now that we've piqued your interest, let's get started working with James.

Download and Installation

The most recent stable version of James (v2.2.0) is available from the James web site. As with most open source projects, you have a choice of downloading the full source code (and compiling it yourself), or downloading the binary code in either ZIP and TAR formats. If you're serious about e-mail, you may want a few additional downloads – namely, the JavaMail 1.3 API, the JavaBeans Application Framework (JAF) 1.0.5, and Apache Ant 1.6.5.

Before starting James for the first time, you should set your `JAVA_HOME` environment variable to the directory where you installed the JDK, e.g.,

```
set JAVA_HOME=C:\jdk1.5.0_04
```

Starting the Server

You can start James by executing one of the command line utilities in James's `bin` subdirectory:

- `run.bat` (Windows)
- `wrapper.exe -c ..\conf\wrapper.conf` (Windows)
- `run.sh` (UNIX)

The wrapper utility includes options for installing/removing the server as an NT service.

When you start James for the very first time, it unpacks the contents of the `james.sar` file and stores the contents in the `james-2.2.0/apps` directory.

Server Administration

You can administer the James server through its Remote Administration Tool. You access the tool by telnetting to port 4555:

```
telnet localhost 4555
```

When it starts, the Remote Administration Tool prompts you for authentication. The default userID is `root` and the default password is `root`.

After logging on you can perform common administrative server tasks such as adding/modifying/deleting user accounts. You can obtain the list of available commands by typing the command, `help`.

Adding Users

You can easily add users with the Remote Administration Tool's `adduser` command:

```
adduser alan 5ibgm8pn
adduser barry f3qaid37
adduser carol z9oxzfva
adduser diane bwwvh002
```

The first parameter is the userID and the second parameter is the password.

Stopping the Server

You can stop the James server by typing `CTRL-C` in the original `run.bat` or `run.sh` window. But when you type `CTRL-C`, you see a nasty-looking "*JVM exiting abnormally*" message. So it's better to stop the server with the Remote Administration Tool's `shutdown` command.

Mailets and Matchers

One of the neat things about James is the use of mailets and matchers.

- A *mailet* is a Java class that does something with a mail message, like adding a footer, forwarding the message, notifying the postmaster about the receipt of the message, or whatever else a mail server may want to do.

- A *matcher* is a Java class that says "Yes, apply this mailet's instructions to that message for these recipients," and "No, don't apply this mailet's instructions to that message for those other recipients."

How Matchers and Mailets Work Together

Let's make this Matcher/Mailet business a bit more precise:

- Each mailet contains a method named `service`. A typical `service` method looks like this:

```
public void service(Mail mail) throws MessagingException {
    MimeMessage mimeMessage = mail.getMessage();

    // Do things with the message

    mimeMessage.saveChanges();
}
```

  The `MimeMessage` class belongs to the `javax.mail.internet` package, and the `Mail` class is part of the `org.apache.mailet` package.

- Each matcher contains a method named `match`. A typical `match` method looks like this:

```
public Collection match(Mail mail) {
    if ( /* something or other is true */ ) {
        return mail.getRecipients();
    } else {
        return null;
    }
}
```

Despite our simplified explanation in the earlier bullets, an invocation of the `match` method doesn't return "yes" or "no." Instead, the `match` method returns a list of recipients for which a mailet should be processed.

Take, for instance, an `AddFooter` mailet that pastes the words "FOR YOUR EYES ONLY" at the bottom a message body. A particular message may have recipients Alan, Barry, Carol, and Diane. So the skeletal `match` method above may return a collection containing Alan, Barry, Carol, and Diane. This collection tells James to add "FOR YOUR EYES ONLY" to the copies sent to all four recipients.

On the other hand, the `match` method above may return `null`. In that case, James adds "FOR YOUR EYES ONLY" to none of the message's recipients. Of course, if you rewrite the `match` method, you may return a collection containing only Barry and Carol, or some other bunch from the message's original list of recipients.

How do you associate a matcher with a mailet? The James `config.xml` file contains `mailet` tags. Here's a nice `mailet` tag:

```
<mailet match="SizeGreaterThan=1k" class="AddFooter">
    <text>Warning: Attachments may contain viruses!</text>
</mailet>
```

The tag uses a matcher named `SizeGreaterThan` and a mailet named `AddFooter`. Both of these classes (the `SizeGreaterThan` matcher and the `AddFooter` mailet) are provided as part of the James distribution. When you write a `mailet` tag, you normally supply parameters to both the mailet and the matcher.

- A parameter to a mailet is an XML element. For example, in the tag above, the `text` element tells the `AddFooter` mailet what text the footer must contain.

- A parameter to a matcher is buried inside the mailet tag's `match` attribute. In the tag above, the `match` attribute specifies the name of the matcher class (`SizeGreaterThan`) and the `1k` (1 kilobyte) parameter value. If a message has size greater than 1 kilobyte, then James adds a `Warning: Attachments may contain viruses!` footer at the bottom of the message.

Matcher and Mailet Classes

We wanted to give James a test run, but we needed an excuse to write a matcher and a mailet. So we dreamt up a fairly frivolous scenario. Here's how it goes:

We're strong advocates of Java technology, and we're dismayed by any mention of .NET. So when we receive e-mail messages containing the characters `.NET` we want to blot out those characters and replace them with some alternate text. What we really need is a `ReplacePlainText` mailet. The mailet looks for all occurrences of a certain string, and replaces such occurrences with other (less upsetting) text.

There's only one exception. We have a mutual friend named Manny, who works with both Java and .NET. (Manny's ongoing effort is to port portions of the Java API to the .NET platform.)

Manny has many e-mail addresses, each of the form Manny*@some-domain-or-other*. Our goal is to allow mail from Manny*@anything-at-all* (and no other mail) to display the word .NET. So our `SenderIsNot` matcher class takes an incoming e-mail message.

- If the sender isn't Manny*@some-domain-or-other*, then the matcher returns all the message's recipients. That is, the matcher tells James to apply the `ReplacePlainText` mailet to every recipient's copy of the message.

- If the sender is Manny*@some-domain-or-other*, then the matcher returns `null`. This tells James to apply the `ReplacePlainText` mailet to none of the recipients' copies.

The matcher and mailet classes are in Listings 1 and 2 respectively. The XML tag to make James use the matcher with the mailet is in Listing 3.

```
/***********************************************************
 * Adapted by Barry Burd from the SubjectStartsWith method         *
 * which is part of James (http://james.apache.org)                *
 * which is Copyright (c) 2000-2004 The Apache Software Foundation.*
 ***********************************************************/
```

```
package org.apache.james.transport.matchers;

import org.apache.mailet.GenericMatcher;
import org.apache.mailet.Mail;

import java.util.Collection;

public class SenderIsNot extends GenericMatcher {
    String senderName = null;

    public void init() throws javax.mail.MessagingException {
        senderName = getCondition();
    }

    public Collection match(Mail mail) {
        if (mail.getSender().getUser().equals(senderName)) {
            return null;
        } else {
            return mail.getRecipients();
        }
    }
}
```

Listing 1: A simple matcher.

```
package org.apache.james.transport.mailets;

import java.io.IOException;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.apache.mailet.GenericMailet;
import org.apache.mailet.Mail;

public class ReplacePlainText extends GenericMailet {
    String oldPhrase;
    String newPhrase;

    public void init() throws MessagingException {
        oldPhrase = getInitParameter("oldPhrase");
        newPhrase = getInitParameter("newPhrase");
    }

    public void service(Mail mail) throws MessagingException {
        MimeMessage mimeMessage = mail.getMessage();
        String oldBody = null, newBody = null;

        try {
            oldBody = mimeMessage.getContent().toString();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (MessagingException e) {
            e.printStackTrace();
        }

        newBody = oldBody.replaceAll(oldPhrase, newPhrase);
        mimeMessage.setText(newBody);
        mimeMessage.saveChanges();
    }
}
```

Listing 2: A simple mailet.

```
<mailet match="SenderIsNot=Manny" class="ReplacePlainText">
    <oldPhrase>\.NET</oldPhrase>
    <newPhrase>!^#%</newPhrase>
</mailet>
```

Listing 3. A mailet tag in the config.xml file.

Both the matcher and the mailet contain `init` methods. In each `init` method, the Java program reads parameters from the `mailet` tag.

- The matcher's init method calls `getCondition`, which reads the word `Manny` from Listing 3.
- The mailet's init method calls `getInitParameter`, which reads the strings `\.NET` and `!^#%` from Listing 3.

Inside the `mailet` tag, the `match` attribute's value takes the form

```
match="matcher-class-name=parameters"
```

The `getCondition` method reads the *`parameters`* string, which in Listing 3 is the word `Manny`. When a mail message arrives, James calls the `match` method in Listing 1. If the message's sender is Manny, the match method says, "Don't apply the mailet to any of the message's recipients." If the message's sender isn't Manny, the match method says "Apply the mailet to all of the message's recipients."

By the way, some matchers have what you'd normally think of as several parameters. For instance, I may want to allow the word `.NET` to appear in messages from three people named Manny, Moe, and Jack. In that case I'd stuff all three names into one delimited string.

```
match="SenderIsNot=Manny,Moe,Jack"
```

Then I'd write business logic inside the `match` method to parse the string and use the three names appropriately.

So assume that *Fred@dotnetlover.com* sends me an e-mail message containing the word `.NET`. Then the `match` method in Listing 1 tells James to apply the mailet's logic to this message. In response, James calls the `service` method of Listing 2, and replaces all occurrences of `.NET` with the string `!^#%`. (In Listing 3, the backslash before `.NET` ensures that the `replaceAll` method doesn't mistake the dot for a regular expression's "any character" symbol.)

Conclusion

James is an efficient, robust e-mail server with an architecture that's ripe for future expansion. Upcoming releases of James may support instant messaging, new e-mail protocols, and other exciting features. James's plug-in design goes hand-in-hand with Java portability, making James a good choice for an enterprise-level e-mail server.

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including "Java 2 For Dummies" and "Eclipse For Dummies," both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He has a Bachelor of Science in Computer Science from Rutgers University. Mike's computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.