James の使用: 第 1 回 Apache の James エンタープライズ・電子メール・サーバーの紹介

オープンソースプロジェクトについての基本を学ぶ

Claude Duguay
Senior J2EE architect
Capital Stream Inc.

2003年 6月 10日

この記事は、Jamesとして知られているJava Apache Mail Enterprise Serverに関する2回シリーズ中の第1回です。今回はJamesの理解と、サーバーサイドの電子メールアプリケーション開発の基本を学びます。また、この記事ではハイレベルな概要や、Apacheグループの設計目的について簡潔に触れ、開発環境のインストールおよび設定方法について説明します。これにより、Jamesでサポートされている機能を簡単に知ることができます。また、Jamesが備えているmatcherおよびmailetのインプリメンテーションに関する概論や、従来の電子メールサーバーの既存の機能との比較についても理解できることでしょう。

このシリーズの他の記事を見る

Java Apache Mail Enterprise Server(一般的にJamesと呼ばれます)は、Apacheグループによって構築されたポータブルで、セキュアで、100%ピュアなJAVAエンタープライズ・メール・サーバーです。しかしJamesは、組み込み可能なプロトコル・アーキテクチャーやservletがWebサーバーに対して行うことを電子メールに行うmailetインフラストラクチャーを備えているので、それ以上のものになり得る可能性があります。電子メールサーバーには、インターネットの原形となったDARPAの初期の頃からの歴史がありますが、Jamesはインターネットの最初のキラー・アプリケーションと呼ばれる新たな可能性を持っています。

今回はJamesを探求する2回シリーズの第1回です。今回はJamesの可能性を探求するために、Jamesの概要やレベルの高い方向性について説明をします。第2回では、無効メッセージを管理しているmailetアプリケーションを実装することにします。後で分かることですが、Jamesでアプリケーションを作ることは驚くほど簡単なことです。電子メールは毎日世界中の何百万もの人々に使用されています。したがって、その可能性はこのシリーズが提供する基本的な処理をはるかに越えています。しかしながら、ここでの基礎がみなさんの役に立ち、みなさんがJamesの可能性を想像し始める助けとなることが私の希望です。

電子メールはどのように機能しているか

電子メールは原則として単純なものです。mail user agent(MUA)を使用して、1つ以上の受取人アドレスを備えたメッセージを構築します。MUAにはいろいろな形式があり、テキストベース、Webベース、GUIアプリケーションなどがあります。例えば、MicrosoftのOutlookやNetscapeのMessengerは、GUIアプリケーションのカテゴリーに分類されます。各電子メールクライアントは、mail transfer agent(MTA)へメールを送るように設定されています。またユーザーのアドレスへ送られる電子メールメッセージを取得する際にもMTAを使用することができます。このためには、メールサーバー(厳密にはMTA)上にメールアカウントが必要であり、なおかつ標準的なインターネット・プロトコルを使用して、オフライン電子メール(POP3を使用)が機能するかサーバー上(IMAPを使用)に電子メールを残すことができる必要があります。クライアントからMTAへのメール送信、およびMTA間でのメール送信に使用されているプロトコルはSMTP(Simple Mail Transfer Protocol)です。

Jamesアプリケーションの構築

このシリーズの第2回では、実用的なアプリケーションを実装するために基本的なJamesインフラストラクチャーについての知識が必要です。

MTA間で実際に起こることは、よりいっそう興味深いことです。電子メールサーバーは、DNSおよびmail transfer(MX)レコードと呼ばれる電子メール特有のレコードに強く依存しています。MXレコードは、URLを決定するために使用されるDNSレコードとはわずかに異なります。DNSレコードには、より効率的にメールを送信するためのプライオリティ情報が付加されています。ここではそれらの詳細を徹底的に調査しませんが、DNSは電子メールを首尾よく効率的に送信するためのキーであると理解しておくことは重要です。JavaMail APIはMUA用に、JamesはMTA用にフレームワークを提供しています。この記事では、Jamesのインストールに際してテストを行うためにJavaMailを使用します。このシリーズの第2回の記事では、Jamesでのアプリケーション開発方法を示すためにJames Mailet APIを使用します。

Jamesの設計目的

Jamesはある目的を叶えるために設計されました。例えば、ポータビリティを最大限にするために全てJava言語で書かれています。また、セキュリティ目的に書かれおり、サーバー環境を保護し安全なサービスを提供するなどの多くの機能を提供しています。JamesはAvalonフレームワークにおいて利用可能な多くの利点を生かすマルチスレッド・アプリケーションとして機能します。(Avalonは、Phoenix高機能サーバー・インフラストラクチャーを特色とするApache Jakarta projectです。Avalonを理解することは有益なことですが、Jamesアプリケーションの開発には必ずしも必要ではありません。Avalonの詳細については 参考文献 を参照してください。)

Jamesは高性能もしくは安定した電子メールサーバーにおいてのみ通常は利用可能な多くのサービスを備え、一連の包括的なサービスを提供しています。これらのサービスは主にMatcher APIおよびMailet APIを使用して実装されています。Matcher APIとMailet APIは、電子メールを検知して処理するために相互に機能しています。Jamesはプロトコルからのメッセージング・フレームワークを抽象化しておく疎結合プラグイン設計を使用して、標準的な電子メールプロトコル(SMTP、POP3、IMAP)に加えて少数のプロトコルをもサポートしています。これはJamesが今後一般的なメッセージング・サーバーの多くの役割を担うことになるか、もしくはインスタント・メッセージのような既存のものに代わるメッセージング・プロトコルをサポートすることになるかもしれない強力な考え方です。

Jamesの設計グループによって発表された最後の興味深いテーマは、mailetの概念です。mailetは電子メールアプリケーションを開発するためのコンポーネントのライフ・サイクルおよびコンテナ・ソリューションを提供します。確かに、ジョブを実行するためにどんなプログラムも呼びだされ、実行可能なプログラムによってデータが送られれば、sendmailのような他のMTAを使用することはいつでも可能です。しかしJamesは、これらの目的を遂行するために、共通で単純なAPIを提供し、操作可能なオブジェクトのおかげで作業を容易にしています。この記事ではMatcher APIとMailet APIについてより詳細に見ていくこととします。

Jamesのインストールと設定

JamesはApache foundation Web siteでホームページからダウンロードが可能です(リンクに関しては 参考文献 を参照してください)。みなさんは最新版をダウンロードするようにしてください。この記事を書いている時点ではバージョン2.1.2です。Jamesのホームページの左のメニューエリアで Downloads > Binaries を選択するとダウンロードエリアへ遷移できます。そこから、Release Buildsセクションまでスクロールして、James2.1のリンクのうちの1つを選択してください。james-2.1.2.tar.gzまたはjames-2.1.2.zipのどちらか好きなほうを選択してください。

さらに、アプリケーションをテストするためにJavaMail APIを使用します。したがって、こちらもダウンロードする必要があります(参考文献 を参照してください)。この時点で利用可能なバージョンは1.3で、ファイル名はjavamail-1_3.zipです。また、メインのJavaMailページ上において、JavaBeans Activation Framework(JAF)へのリンクに気づくことでしょう。JAFはJavaMail APIが必要とするものです(JAFへのリンクに関しては 参考文献 を参照してください)。JAFの現在のバージョンは1.0.2で、ファイル名はjaf-1_0_2.zipです。これらのファイルをすべてダウンロードすれば、Jamesを機能させるためにシステムをセットアップすることができます。

開発のために必要となるすべての要素を備えたディレクトリー構造を構築します。本番用では、これらの配置はセキュリティーおよび機能性を考慮して全く異なったものになっていなければなりません。例えば、ここでの目的ならばlocalhostで作業することができます。しかし実際の電子メールサーバー配備においては、これは現実的な選択肢ではありません。サーバーを商用に配置する必要がある場合は、主要なMTAとしてのあるいはsendmailと共存する場合のJamesの設定に関する多くのドキュメンテーションおよびメーリング・リスト上のサポートがあります。

Jamesディレクトリーにすべてが解凍されると、ディレクトリー階層はリスト1のようになります。ここではよりコンパクトで分かりやすいように、javadoc、src、JavaMail webapp demoの下のいくつかのサブディレクトリーを削除しています。

リスト1. James、JavaMail、JAFの各ディレクトリー

```
James
+---jaf-1.0.2
|   +---demo
|   \---docs
|       \---javadocs
+---james-2.1.2
|   +---apps
|   +---bin
|   |   \---lib
|   +---conf
|   +---docs
|   |   +---images
```

```
|   |   \---stylesheets
|   +---ext
|   +---lib
|   +---logs
\---javamail-1.3
    +---demo
    |   +---client
    |   +---servlet
    |   \---webapp
    +---docs
    |   \---javadocs
    \---lib
```

ここでは、Jamesファイルに依存しないJavaプラットフォームのバージョン1.4を入手されたと想定しています。James構成メモは、Java 1.3.0の使用でいくつかの問題が発生したことを示しています。したがって、1.3.1かそれ以上を使用するべきです。原則として、Jamesは適切なJava1.4VMをサポートするプラットフォームではうまく機能します。

はじめの第一歩は、Jamesをスタートさせることです。なぜなら、設定ファイルはサーバーが一度起動されるまで解凍されないからです。そうするとjames-2.1.2/binディレクトリーに実行スクリプト(オペレーティング・システムにより、run.batあるいはrun.shのいずれかを使用します)が確認できます。そのスクリプトを実行すると、リスト2のような出力になります(このサンプルはWindowsシステムからの出力です)。

リスト2.James実行時のコンソール出力

```
Using PHOENIX_HOME:   D:\James\james-2.1.2
Using PHOENIX_TMPDIR: D:\James\james-2.1.2\temp
Using JAVA_HOME:      c:\programming\java14
Phoenix 4.0.1
James 2.1.2
Remote Manager Service started plain:4555
POP3 Service started plain:110
SMTP Service started plain:25
NNTP Service started plain:119
Fetch POP Disabled
```

プログラムを終了するためにCtrl+Cを使用することができます。そうすると、Phoenixコンテナーからメッセージを得ることになります。厳密に言えば、Jamesを終了させる適切な方法はリモート管理インターフェースを使用することです。しかし私は開発中Ctrl+Cを使用していましたが、特に影響はありませんでした。とはいえ、配置環境では常にシャット・ダウン・コマンドを使用するべきです。

初めてJamesをシャットダウンすると、james-2.1-2/apps/james/SAR-INFフォルダーにconfig.xmlというファイルができますので、このファイルを見てみてください。通常、最初に変更するのは管理者アカウントです。管理者アカウントはデフォルトでrootに、パスワードもrootに設定されています。開発のためにはそのままにしておきますが、明らかに本番システムでこのままに設定しておくことは愚かなことです。その次に変更するものは通常DNSサーバーアドレスです。この作業は、Jamesを完全な電子メールサーバーとして作動させたいのならば必要です。テストをすべてlocalhostから実行するならば、こちらもそのままにしておいてよいでしょう。しかし、このこともまた知っておくべき重要な設定事項です。設定ファイルを理解することは重要ですが、今回の開発目的では残りの項目はデフォルト設定のままで構わないです。詳細については、james-2.1.2/docsディレクトリーにあるドキュメントから参照が可能です。

ここで、先へ進む前に数人のユーザーを追加してみましょう。そのために、まずコマンド `telnet localhost 4555` で、ポート4555のlocalhostへtelnetしてください。rootユーザー名およびパスワードでログインすることができます。ログインができたら、数人のユーザーを追加します。`adduser` コマンドはユーザー名とパスワードの組み合わせを必要とします。では、このプロジェクトのためにユーザーred、green、blueを追加します。それぞれのパスワードはユーザー名と同じとします(実際のユーザーを追加する場合、これでは不適切なことをみなさんはご存知でしょう。しかし、このようにするとテストケースの設定が簡単になります)。ユーザーを追加したら、エントリーを確認するために `listusers` コマンドを発行します。それから、`quit` コマンドでリモート・マネージャを終了します。全体のセッションはリスト3のようになります。みなさんが入力するテキストは強調表示してあります。

リスト3.リモート・マネージャでのユーザーの追加

```
JAMES Remote Administration Tool 2.1.2
Please enter your login and password
Login id:
root
Password:
root
Welcome root. HELP for a list of commands
adduser red red
User red added
adduser green green
User green added
adduser blue blue
User blue added
listusers
Existing accounts 3
user: blue
user: green
user: red
quit
Bye
```

これで、Jamesサーバーを起動させる全ての設定ができました。お分かりのように、Jamesを配置させたり設定用にリモート・マネージャーを使用することは比較的簡単なことです。メールサーバに安全性を求めるならば、明らかにいくつかの設定パラメーターを変更する必要があります。しかし、それは複雑なプロセスではありません。メールサーバの使用に際してキーとなるのは、マルチユーザーおよびマルチサーバー環境でDNSを正確に設定することです。このことはこの記事の範囲外ではありますが、だからといって決して複雑なプロセスではありません。

JavaMailを使用したJamesのテスト

設定が有効であることを確かめるために、メッセージを送信しinboxの内容をリストする一組のクラスを書いて、標準的な電子メールクライアントの基本的な機能をシュミレートします。ここでは2つのクラスを使用しますが、リスト4に示す `MailClient` クラスは、より複雑な振る舞いをテストするために再利用することができます。このシリーズの第2回の記事でJamesアプリケーションを開発する際に、 `MailClient` クラスを再利用する予定です。

リスト4.MailClient：電子メールクライアントの基本機能のシュミレート

```
import java.io.*;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
public class MailClient
```

```
  extends Authenticator
{
  public static final int SHOW_MESSAGES = 1;
  public static final int CLEAR_MESSAGES = 2;
  public static final int SHOW_AND_CLEAR =
    SHOW_MESSAGES + CLEAR_MESSAGES;
  protected String from;
  protected Session session;
  protected PasswordAuthentication authentication;
  public MailClient(String user, String host)
  {
    this(user, host, false);
  }
  public MailClient(String user, String host, boolean debug)
  {
    from = user + '@' + host;
    authentication = new PasswordAuthentication(user, user);
    Properties props = new Properties();
    props.put("mail.user", user);
    props.put("mail.host", host);
    props.put("mail.debug", debug ? "true" : "false");
    props.put("mail.store.protocol", "pop3");
    props.put("mail.transport.protocol", "smtp");
    session = Session.getInstance(props, this);
  }
  public PasswordAuthentication getPasswordAuthentication()
  {
    return authentication;
  }
  public void sendMessage(
    String to, String subject, String content)
      throws MessagingException
  {
    System.out.println("SENDING message from " + from + " to " + to);
    System.out.println();
    MimeMessage msg = new MimeMessage(session);
    msg.addRecipients(Message.RecipientType.TO, to);
    msg.setSubject(subject);
    msg.setText(content);
    Transport.send(msg);
  }
  public void checkInbox(int mode)
    throws MessagingException, IOException
  {
    if (mode == 0) return;
    boolean show = (mode & SHOW_MESSAGES) > 0;
    boolean clear = (mode & CLEAR_MESSAGES) > 0;
    String action =
      (show ? "Show" : "") +
      (show && clear ? " and " : "") +
      (clear ? "Clear" : "");
    System.out.println(action + " INBOX for " + from);
    Store store = session.getStore();
    store.connect();
    Folder root = store.getDefaultFolder();
    Folder inbox = root.getFolder("inbox");
    inbox.open(Folder.READ_WRITE);
    Message[] msgs = inbox.getMessages();
    if (msgs.length == 0 && show)
    {
      System.out.println("No messages in inbox");
    }
    for (int i = 0; i < msgs.length; i++)
    {
      MimeMessage msg = (MimeMessage)msgs[i];
      if (show)
      {
```

```
        System.out.println(" From: " + msg.getFrom()[0]);
        System.out.println(" Subject: " + msg.getSubject());
        System.out.println(" Content: " + msg.getContent());
      }
      if (clear)
      {
        msg.setFlag(Flags.Flag.DELETED, true);
      }
    }
    inbox.close(true);
    store.close();
    System.out.println();
  }
}
```

`MailClient` クラスは主に、メッセージを送信したり、特定のユーザーのためにサーバー上の利用可能なメッセージのリストを表示したり削除するために設計されています。私はメッセージを表示させたり(`SHOW_MESSAGES`)、削除したり(`CLEAR_MESSAGES`)、表示と削除を両方行う(`SHOW_AND_CLEAR`)ために有用な定数を宣言しました。 `MailClient` クラスは、さらに電子メールを取得する場合にログオン・プロセスの管理を容易にする `Authenticator` インターフェースをインプリメントしています。

私は2つのコンストラクタを作成しました。そのうちの１つは、JavaMailデバッキング・フラグを明示的にセットしています。これは、何が起きているかが分かるように、クライアント/サーバプロトコルのやりとりをコンソールへ出力するものです。短い方のコンストラクタは、このフラグをオフにしています。他の2つの引数は、ユーザー名とホストです。電子メールアドレスをユーザーおよびホストから引き出すことができます。 `Authenticator` インターフェースで指定された `getPasswordAuthentication()` メソッドで返される `PasswordAuthentication` オブジェクトを作成しています。

コンストラクタコードの残りの部分では、指定のユーザーおよびホストを反映するためにJavaMailプロパティを設定し、使用する予定のプロトコルを明示的に指定しています。一度 `Properties` オブジェクトを取り込めば、有効な `Session` 参照のためにスタティックな `Session` メソッドである `getInstance()` を呼ぶことができます。そして、ローカル変数にこの参照を格納します。一旦コンストラクタが指定のユーザーに使用されたならば、指定の電子メールホスト上のそのユーザーのためのメールをいつでも送受信できるようになります。

`sendMessage()` メソッドも同様に分かりやすいものです。指定の受取人、サブジェクト、テキスト内容で `MimeMessage` を構築し、 `Transport` クラスのJavaMailのスタティックな `send()` メソッドを使用してこれを送信しています。何が起きているかを容易に確かめられるように、コンソールへメッセージを出力することもできます。

`checkInbox()` メソッドは、メッセージをリストしたり任意に削除する必要があるので、より多くの作業をこなしています。モード引数で使用するフラグにより、メッセージを見ずに削除することも可能です。実際にメッセージを取得するためには、セッションオブジェクトにより `Store` を参照し、サーバーに接続し、inboxフォルダーをオープンする必要があります。フォルダーを参照した後は、メッセージが存在する限り参照したり削除したりすることができます。

再利用可能な `MailClient` コードができたので、localhost上でJamesサーバーに対する簡単なテストをいつでも行うことができます。リスト5の `JamesConfigTest` クラスは、3つの `MailClient` イ

ンスタンスの作成によりテストを行っています。それぞれのインスタンスは、既出のユーザー(red,green,blue)の中の一人と結び付いています。これらのユーザーが電子メールサーバー上で有効であることを確かめてから、このコードを実行してください。

リスト5.JamesConfigTest:Jamesサーバーの簡単なテスト

```
public class JamesConfigTest
{
  public static void main(String[] args)
    throws Exception
  {
    // CREATE CLIENT INSTANCES
    MailClient redClient = new MailClient("red", "localhost");
    MailClient greenClient = new MailClient("green", "localhost");
    MailClient blueClient = new MailClient("blue", "localhost");
    // CLEAR EVERYBODY'S INBOX
    redClient.checkInbox(MailClient.CLEAR_MESSAGES);
    greenClient.checkInbox(MailClient.CLEAR_MESSAGES);
    blueClient.checkInbox(MailClient.CLEAR_MESSAGES);
    Thread.sleep(500); // Let the server catch up
    // SEND A COUPLE OF MESSAGES TO BLUE (FROM RED AND GREEN)
    redClient.sendMessage(
      "blue@localhost",
      "Testing blue from red",
      "This is a test message");
    greenClient.sendMessage(
      "blue@localhost",
      "Testing blue from green",
      "This is a test message");
    Thread.sleep(500); // Let the server catch up
    // LIST MESSAGES FOR BLUE (EXPECT MESSAGES FROM RED AND GREEN)
    blueClient.checkInbox(MailClient.SHOW_AND_CLEAR);
  }
}
```

3つの `MailClient` インスタンスを作成した後、 `JamesConfigTest` コードは、 `checkInbox()` メソッドを使用して `CLEAR_MESSAGES` モードで各mailboxをクリアし、サーバーが削除処理をしたことを確かめるために500秒待機します。その後、redとgreenからblueへメッセージを送信し、blueアカウントへのメッセージを確認します。 `JamesConfigTest` を実行すると、リスト6のように出力されます。

リスト6.JamesConfigTestの実行結果出力

```
Clear INBOX for red@localhost
Clear INBOX for green@localhost
Clear INBOX for blue@localhost
SENDING message from red@localhost to blue@localhost
SENDING message from green@localhost to blue@localhost
Show and Clear INBOX for blue@localhost
    From: green@localhost
 Subject: Testing blue from green
 Content: This is a test message
    From: red@localhost
 Subject: Testing blue from red
 Content: This is a test message
```

これにより、Jamesの設定が有効であることを証明できます。開発に移る前にこのようにシステムを設定する必要があります。しかしながら、私たちはこのシリーズの第2回まで開発内容の掲載は控えます。この記事の残りでは、Matcher API、MailetのAPI、およびJamesを配置したことで提供

されたmatcherおよびmailetを検討していきます。さらに、Jamesにサポートされたいくつかの追加の機能をさっと見ていくことにします。

Matchers

Jamesは多くの標準的なmatcherを備えています。リスト7で示されるように、各matcherはMatcher APIをインプリメントし、他の便利な継承と同様に既存のMTAに共通の機能を提供しています。インターフェースはかなりシンプルです。インターフェースには、ライフサイクルメソッドの `init()` および `destroy()` や、bookkeepingメソッドの `getMatcherInfo()` および `getMatcherConfig()` や、メインメソッドで `Mail` オブジェクトで機能する `match()` などがあります。 `Mail` 参照はプロセスにコンテナー状態、メール・メッセージ、およびメタデータへのアクセスを供給します。

リスト7.Matcherインターフェース

```
public interface Matcher
{
  void init(MatcherConfig config);
  void destroy();
  String getMatcherInfo();
  MatcherConfig getMatcherConfig();
  Collection match(Mail mail);
}
```

Matcherは受信者のグループを認識し、mailetによって処理される受信者の `String` オブジェクトのコレクションを返します。matcher認識とmailet処理を組み合わせることによって、電子メールメッセージ上で動作する複雑なアプリケーションを開発することができます。

Jamesを配置したことで提供されたmatcherにより、自分でmatcherを実装することなく、いくつか実行できることがあります。matcherの開発をすると決める前に、何が既に存在しているかを知っておくことは重要です。多くの場合、意図している事は既に存在しているかもしれません。テーブル1に、既に存在しているmatcherをリストしておきます。

テーブル1.既に存在しているJamesのMatcher

Matcher	概要
All	処理される全ての電子メールとマッチし、全ての受信者を返します
HasHeader	存在すれば指定のヘッダにマッチします
HasAttachment	メッセージがマルチメッセージ(添付ファイルなど)である場合にマッチします
SubjectStartsWith	メールの題名が指定された文字列で始まっているメールにマッチします
SubjectIs	メールの題名が指定された文字列と一致するメールにマッチします
HostIs	指定されたホスト名に一つでも合うメッセージにマッチします
HostIsLocal	localhostとして設定されているホストからのものにマッチします
UserIs	受信者のアカウントにマッチします
SenderIs	送信者アドレスが指定条件に合う場合にマッチします
SenderInFakeDomain	電子メールアドレスの一部のドメイン名が適切ではないものにマッチします

SizeGreaterThan	メールのサイズが指定されたサイズよりも大きい場合にマッチします
Recipients	受取人が指定されたリストに載っているメールにマッチします
RecipientsLocal	ローカルアカウントであると認識される全てのホストからのメッセージにマッチします
IsSingleRecipient	一つの送信アドレスしか指定されていないものにマッチします
RemoteAddrInNetwork	ネットワークアドレスリストに合致する全てのIPアドレスからのメッセージにマッチします
RemoteAddrNotInNetwork	ネットワークアドレスリストに合致しない全てのIPアドレスからのメッセージにマッチします
RelayLimit	ホップが指定数以上のメールにマッチします
InSpammerBlackList	mail-abuse.orgによって提供されるIPアドレスから受信された全てにマッチします
NESSpamCheck	SPAMメールを特定する一定規則に合致しているものにマッチします
HasHabeasWarrantMark	Habeas Warrantのあるメールにマッチします
FetchedFrom	FetchPOPで使用される `X-fetched-from` ヘッダのメールにマッチします
CommandForListserv	リストサーバーのコマンドにマッチします

このリストからお分かりのように、スパムの探知やリストサーバー・コマンドの処理のようなハイレベルオペレーションだけでなく、ヘッダー、サブジェクト、受信者のマッチングなどのちょっとしたオペレーションまでも既に存在しているので、新たにプログラミングをすることなく多くのタスクを遂行できるようになっています。

Mailets

Jamesの特徴の多くは、リスト8で示されるようにMailet APIを通してインプリメントされています。Mailet APIはServlet APIの使用に慣れている開発者には見覚えがあることでしょう。Matcher APIのように、Mailetインターフェースは、初期化作業(`init()` メソッド)、およびシャット・ダウン(`destroy()` メソッド)の2つのライフ・サイクルメソッドをサポートしています。そして次の2つのメソッドが情報を返しています。1番目のメソッドは、mailetに関連した著者、バージョン、およびcopyrightといった情報を含み `String` オブジェクトを返す `getMailetInfo()` です。そして2番目のメソッドの `getMailetConfig()` は、現在のmailet設定情報を返すのに便利です。init()メソッドは引数に `MailetConfig` オブジェクトを指定します。したがって、変更されているかもしれませんが、 `init()` メソッドは通常 `getMailetConfig()` メソッドによって返されたオブジェクトです。

リスト8.Mailetインターフェース

```
public interface Mailet
{
  void init(MailetConfig config);
  void destroy();
  String getMailetInfo();
  MailetConfig getMailetConfig(); void service(Mail mail);
}
```

メイン処理は、`Mail` オブジェクト引数をとる `services()` メソッドで行われます。このオブジェクトはコンテナー状態、メールのメッセージ、およびメタデータへのさらなるアクセスを行います。

Jamesでサポートされている特徴および既に存在するmailetアプリケーションについて参考にしていただくために、テーブル2で細かい設定を必要とせず利用可能なmailetインプリメンテーションをリストしておきます。

テーブル2.既に存在しているJamesのmailet

Mailet	概論
Null	電子メールメッセージの処理を終了します
AddHeader	メッセージコンテキストにテキストヘッダを追加します
AddFooter	メッセージコンテキストにテキストフッタを追加します
Forward	受信者リストを指定された受信者リストに置き換えます
Redirect	リダイレクトサービスの設定を提供します
ToProcessor	指定したプロセッサーに電子メール処理をリダイレクトします
ToRepository	指定のディレクトリーへメッセージの保存します
NotifySender	元の送信者へ添付ファイルとしてメッセージを転送します
NotifyPostmaster	ポストマスターへ添付ファイルとしてメッセージを転送します
RemoteDelivery	リモートサーバに、メールをリレーします
LocalDelivery	ローカルメールボックスへメッセージをリレーします
JDBCAlias	JDBCデータ・ソースを利用してエイリアストランザクションを行います
JDBCVirtualUserTable	JDBCデータ・ソースを利用してより複雑なエイリアストランザクションを行います
UseHeaderRecipient	メッセージヘッダーよりメール受取人を再作成します
ServerTime	送信者に対しサーバーのタイムスタンプ・メッセージを送信します
PostmasterAlias	個人のアドレスへ `postmaster@<domain>` メッセージをリダイレクトします
AddHabeasWarrantMark	メッセージへHabeas Warrant マークを追加します
AvalonListserv	基本的なサーバー機能リストを提供します
AvalonListservManager	リストサーバー管理コマンドを処理します

このリストからお分かりのように、複雑なリストサーバー・サポート、aliasing、ストレージ機能、ルーティング機能などを含んだMailet APIのおかげで、Jamesには既にたくさんの利用可能な機能が存在しています。

追加機能

Jamesは、このシリーズの範囲外においても他に多くの機能をサポートしています。しかしながら、Jamesが現実にできることについてより理解しやすいように、ここでは簡潔に言及しておくことにします。最初に挙げられるのは、JamesがUsenetサーバーの役割を果たすNNTPサポートです。さらに、Jamesはメールベースのリモート管理機能をサポートするためにFetchPOPプロトコルをインプリメントしています。また、RemoteManagerおよびSpoolManagerは、多数のタイプのストレージおよび管理サポートの存在を可能にするアブストラクションを提供しています。部分的なおよび完全なデータ・ベース中心のソリューションも提供されてはいますが、開発目的ではファイルシステムに基づいたSpoolManagerに依存しても差し支えありません。

Jamesはユーザーを効果的に管理できるインターフェースおよびサービスを提供しています。また、メーリング・リストのサポートも簡単に利用可能です。実際、メーリング・リストは最も使用されているサービスのうちの1つで、管理者がJamesを電子メールソリューションに選ぶ主な理由のうちの1つとなっています。

次回は何？

Jamesインフラストラクチャーは柔軟で簡単なアプリケーション開発のために設計されています。電子メールアプリケーションの可能性はあなたの想像によって制限されているにすぎません。このシリーズの第2回では、vacationメッセージタイプ機能を有効にすることで、ユーザーが特定のJamesのアドレスへ電子メールメッセージを送信できるような簡単なアプリケーションを開発します。ユーザーがこの機能を停止させるために別に指定されたJamesのアドレスへキャンセル・メッセージを送信するまでは、メッセージを送信してくるすべての人に電子メールが送られるように設計することができます。このソリューションは、メールクライアント・ソフトウェアでしばしば実装されているようなメカニズムと似ていますが、通常この機能は一箇所だけに限定されており、メールクライアントをオフにしていると機能しません。しかしメール・サーバーにこの機能をインプリメントしておけば、どこからでも自由にメールをチェックすることができ、たとえ計画が変更しても、「不在」メッセージをより適切な返信へと簡単に変更することができます。

著者について

Claude Duguay

Claude Duguayは氏は20年以上ソフトウェアを開発しています。彼はJavaプラットフォームに関して情熱をもっており可能な限り執筆をしています。