Seam Mail

Seam Mail Introduction

Seam mail is an portable CDI extension designed to make working with Java Mail easier via standard methods or `plugable` templating engines.

1.1. Getting Started

No better way to start off then with a simple example to show what we are talking about.

```
@Inject
private Instance<MailMessage> mailMessage;

public void sendMail() {

   MailMessage m = mailMessage.get();
   m.from("John Doe<john@test.com>")
     .to("Jane Doe<jane@test.com>")
     .subject(subject)
     .bodyHtml(htmlBody)
     .importance(MessagePriority.HIGH)
     .send();
}
```

Very little is required to enable this level of functionality in your application. Let's start off with a little requenableTlsjndiSessionNamejava.net.InetAddress.getLocalHost()ired configuration.

Configuration

By default the configuration parameters for Seam Mail are handled via configuration read from your application's seam-beans.xml. This file is then parsed by Seam Solder to configure the MailConfig class. You can override this and provide your own configuration outside of Seam Mail but we will get into that later.

2.1. Minimal Configuration

First lets add the relevant maven configuration to your pom.xml

```
 <dependency>
  <groupId>org.jboss.seam.mail</groupId>
  <artifactId>seam-mail-impl</artifactId>
  <version>${seam.mail.version}</version>
</dependency>
```

Now now that is out of the way lets provide `JavaMail` with the details of your `SMTP` server so that it can connect and send your mail on it's way.

This configuration is handled via Seam Solder which reads in the configuration from your application's `seam-beans.xml` and configures the `MailConfig` class prior to injection.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:s="urn:java:ee"
       xmlns:mail="urn:java:org.jboss.seam.mail.core"
       xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://docs.jboss.org/cdi/beans_1_0.xsd">

    <mail:MailConfig
          serverHost="my-server.test.com"
          serverPort="25">
       <s:modifies/>
    </mail:MailConfig>

</beans>
```

That is all the configuration necessary to send a simple email message. Next we will take a look at how to configure and use the supported templating engines.

Important

JBoss AS 7.0.x does not correctly load all the modules to support sending mail *AS7-1375* [https://issues.jboss.org/browse/AS7-1375]. This is easily fixed By replacing the module definition at $JBOSS_HOME/modules/javax/activation/api/main/module.xml with the following

```
<module xmlns="urn:jboss:module:1.0" name="javax.activation.api">
   <dependencies>
      <module name="javax.api" />
      <module name="javax.mail.api" >
         <imports><include path="META-INF"/></imports>
      </module>
   </dependencies>

    <resources>
      <resource-root path="activation-1.1.1.jar"/>

      <!-- Insert resources here -->
   </resources>
</module>
```

This will be fixed in AS 7.1.x

Core Usage

3.1. Intro

While Seam Mail does provide methods to produce templated email, there is a core set of functionality that is shared whether you use a templating engine or not.

3.2. Contacts

At it's base an email consists of various destinations and content. Seam Mail provides a wide varerity of methods of ways to configure the following address fields

- From
- To
- CC
- BCC
- REPLY-TO

3.2.1. String Based

Seam Mail leverages the JavaMail InternetAddress object internally for parsing and storage and provides a varargs method for each of the contact types. Thus you can provide either a String, multiple Strings or a String []. Addresses are parsed as RFC 822 addresses and can be a valid Email Address or a Name + Email Address.

```
MailMessage m = mailMessage.get();
m.from("John Doe<john@test.com>")
.to("jane@test.com")
.cc("Dan<dan@test.com", "bill@test.com")
```

3.2.2. InternetAddress

Since we leverage standard InternetAddress object we might as well provide a method to use it.

```
MailMessage m = mailMessage.get();
m.from(new InternetAddress("John Doe<john@test.com>"))
```

3.2.3. EmailContact

Since applications frequently have their own object to represent a user who will have an email set to them we provide a simple interface which your object can implement.

```
public interface EmailContact {
  public String getName();

  public String getAddress();
}
```

Let's define this interface on an example user entity

```
@Entity
public class User implements EmailContact {

  private String username; //"john@test.com"
  private String firstName; //"John"
  private String lastName; //"Doe"

  public String getName() {
    return firstName + " " + lastName;
  }

  public String getAddress() {
    return username;
  }
}
```

Now we can use our User object directly in an of the contact methods

```
User user;

MailMessage m = mailMessage.get();
m.from("John Doe<john@test.com>");
m.to(user);
```

3.2.4. Content

Adding content to a message is simply a matter of calling the appropriate method

For plain text body

```
MailMessage m = mailMessage.get();
m.bodyText("This is the body of my message");
```

For HTML body

```
MailMessage m = mailMessage.get();
m.bodyHtml("This is the <b>body</b> of my message");
```

For HTML + plain text alternative body

```
MailMessage m = mailMessage.get();
m.bodyHtmlTextAlt("This is the <b>body</b> of my message", "This is a plain text alternative");
```

3.2.5. Attachments

Attachments can be added using a few different implementations of the EmailAttachment.java interface

- BaseAttachment.java: byte[]
- InputStreamAttachment.java: java.io.InputStream
- FileAttachment.java: java.io.File
- URLAttachment.java: Loads attachment from URL

Attachments are added by providing the relevant information per implementations as well as a ContentDisposition which specifics if the attachment is simply "ATTACHMENT" or "INLINE" which allows the attachment to be referenced in the body when using templating

```
MailMessage m = mailMessage.get();
m.addAttachment(new FileAttachment(ContentDispostion.ATTACHMENT, new File("myFile"));
```

Here is how to reference an attachment inline. Inline attachments are referenced by their filename.

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <body>
  <p><b>Dear <a href="mailto:$person.email">$person.name</a>,</b></p>
  <p>This is an example <i>HTML</i> email sent by $version and Velocity.</p>
  <p><img src="$mailContext.insert("seamLogo.png")" /></p>
  <p>It has an alternative text body for mail readers that don't support html.</p>
 </body>
</html>
```

Templating

4.1. Basics

Instead of creating your message body as a String and setting it via the bodyText, bodyHtml and bodyHtmlTextAlt methods you can use integrated templating to substitute variables for values. Seam Mail supports templating in any of the body parts as well as the subject.

The template can either be loaded as from a java.lang.String, java.io.File or java.io.InputStream

Values to be used in place of your template variables are loaded via the put method, as a Map or optionally resolved via EL in Velocity

4.2. Velocity

Example Usage. See Velocity documentation for syntax.

```
MailMessage m = mailMessage.get();
m.subject(new VelocityTempalte("Weather for $city, $state")
m.bodyText(new VelocityTemplate(new File("myFile.template"));
m.put("city", "Boston");
m.put("state", "Massachusetts");
m.put("now", new Date());
```

A Velocity Template can be optionally created with an CDI injected CDIVelocityContext. This adds the ability to resolve your variables via the built in velocity context and a fall through to resolve CDI beans which are @Named.

```
@Inject
private CDIVelocityContext cdiContext;

MailMessage m = mailMessage.get();
m.bodyText(new VelocityTemplate(new File("myFile.template"), cdiContext);
```

4.3. FreeMarker

FreeMarker usage is essentially the same as Velocity except there is no EL functionality. Consult FreeMarker documentation for Syntax

```
MailMessage m = mailMessage.get();
m.subject(new FreeMarkerTemplate("ATTN: ${name}")
m.bodyText(new FreeMarkerTemplate("Hello ${name} you have won 1M USD!"));
m.put("name", "Ed Mcmahon");
```

Advanced Features

5.1. MailTransporter

MailTransporter is an interface that allows you to control how the message is handled when the send() method is called. The default implementation simply sends the message with a javax.mail.Session. The main drawback of this is that the thread is blocked until your configured email server accepts your messages. This can take milliseconds or minutes depending on the size of your message load on your SMTP server. While this may be fine for most simple cases, larger applications may wish to employ a queuing function so that messages appear to send instantly and do not block application execution.

Overiding the default MailTransporter is simple

```
MailTransporter myTransporter = new MailQueueTransporter();
MailMessage msg = mailMessage.get();
msg.setMailTransporter(myTransporter);
```

A simple implementation might convert the message into a MimeMessage for sending and then fire as a CDI event so that a CDI @Observer can queue and send the message at a later time.

The MailTransporter might look something like this.

```
public class MailQueueTransporter implements Serializable, MailTransporter {

  private static final long serialVersionUID = 1L;

  @Inject
  @QueuedEmail
  private Event<MimeMessage> queueEvent;

  @Inject
  @ExtensionManaged
  private Instance<Session> session;

  @Override
  public EmailMessage send(EmailMessage emailMessage) {

     MimeMessage msg = MailUtility.createMimeMessage(emailMessage, session.get());
```

```
        queueEvent.fire(msg);

        emailMessage.setMessageId(null);

        try {
            emailMessage.setLastMessageId(msg.getMessageID());
        }
        catch (MessagingException e) {
            throw new SendFailedException("Send Failed ", e);
        }
        return emailMessage;
    }
}
```

5.2. MailConfig

MailConfig supports the following options

- serverHost - SMTP server to connect to
- serverPort - Port to connect to SMTP server
- domainName - Used to build the Message-ID header. Format is UUID@domainName (uses `java.net.InetAddress.getLocalHost().getHostName()` if unset)
- username - Used when your SMTP server requires authentication
- password - Used when your SMTP server requires authentication
- enableTls - Allow TLS to be used on this connection
- requireTls - Require TLS to be used on this connection
- enableSsl - Allow SSL to be used on this connection
- auth - Used when your SMTP server requires authentication
- jndiSessionName - Load the javax.mail.Session via JNDI rather than creating a new one

5.3. EmailMessage

The MimeMessage as defined by javax.mail is a flexible and thus complicated object to work with in it's simplest configuration. Once multiple content types and attachments are added it can be downright confusing. To make working messages easier to work with, Seam Mail provides the

EmailMessage.java class as part of it's core implementation. An instance of EmailMessage is returned from all of the send methods as well after manually calling template merge methods.

While Seam Mail does not provide a mechanism to receive messages it does provide a way to convert a `javax.mail.MimeMessage` or `javax.mail.Message` received via POP or IMAP back into a `EmailMessage.java` for easy of reading and manipulation via the MessageConverter.

```
Session session = Session.getInstance(pop3Props, null);
Store store = new POP3SSLStore(session, url);
store.connect();

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

Message[] messages = inbox.getMessages();
List<EmailMessage> emailMessages = new LinkedList<EmailMessage>();
for (Message m : messages) {
   EmailMessage e = MessageConverter.convert(m);
   emailMessages.add(e);
}
inbox.close(false);
store.close();
```