

# ploinMailFactory

Robert Reiz

- Problem
- ploidMailFactory
- PostOffice-Pattern
- Templating
- Map
- mail.properties
- Internationalisierung
- Sonderzeichen
- Sonderfaelle

- Eine Applikation soll E-Mails verschicken.
- SUN bietet hierfür die JavaMail API. Funktioniert sehr gut, API ist aber sehr unschön.
- Die commons-email.jar von apache ist ein wrapper für die mail API von SUN und bietet eine sehr schöne API.

- E-Mails sollen abhängig vom Mandanten verschickt werden und Internationalisiert.
- Hierfür gab es bis vor kurzem keine Standardlösung!

In den meisten Fällen wurde eine Textvorlage für jeden Mandanten angelegt mit markierten Strings die vor dem absenden mit sinnvollen Werten ersetzt wurden.

```
:ansprache:  
  
wir habe ihr Passwort zurueckgesetzt. Ihr neues Passwort lautet: :neuesPasswort:  
  
Mit freundlichen Gruessen,  
Ihr Dienstleister FooBar
```

Im Quellcode wurde dann abgefangen zu welchem Mandanten der User gehört und in abhängigkeit davon wurde die eine oder andere E-Mail verschickt.

```
2    if (user.getMandant().equals("mandant1")) {  
3        sendPasswordHtml();  
4    } else {  
5        sendPasswordPlain();  
6    }
```

- Der Quellcode der die Templates einliest und die Ersetzungen macht war in allen Applikation sehr ähnlich
- Auch die Abfragen nach den Mandanten musste immer wieder gecodet werden.
- Quellcode der die Templates einliest und die Ersetzungen macht wurde von App. zu App. kopiert.

Aus der eben gezeigten Problematik heraus ist die ploidMailFactory entstanden. Hier sind die wichtigsten Features.

- Templating-System
- Mandantenfähigkeit
- Internationalisierung
- Einheitliches Ablagesystem für E-Mail Templates
- Loadbalancing
- Retry-Funktion
- FallBack-Server-Definition

## Das Framework ist:

- OpenSource
- Apache 2.0 Lizenz
- Veröffentlicht auf SourceForge
- Download: <https://sourceforge.net/projects/ploinmailfactor/>
- Doku:
  - <http://www.ploinMailFactory.org>
  - <http://ploin.de/robertReiz/blogShow.action?blogId=69>



## Konfiguration

Die ploinMailFactory erwartet im Root-Class-Loader eine Konfigurationsdatei "mail.properties".

```
propFileName=mail.properties  
mailDirectory=service/businesslogic/mail  
htmlExtension=.html  
plainExtension=.txt  
loadbalanceNodes=0  
singleThread=false
```

## **propFileName=mail.properties**

Diese Property legt fest wie die weiteren Konfigurationsdateien in den Templates heißen sollen.

## **mailDirectory=service/businesslogic/mail**

Diese Property gibt den Pfad zum Verzeichnis an, in dem die Templates liegen.

## **htmlExtension=.html**

Das Standard-Präfix für HTML-Templates

## **plainExtension=.txt**

Das Standard-Präfix für Text-Templates

## **loadbalanceNodes=0**

Die Anzahl der Nodes für Loadbalancing

## **singleThread=true**

Startet den Versand der E-Mails im gleichen Thread. Bei “false” wird für den MailVersand ein separater Thread aufgemacht.

Wenn der Quellcode der die E-Mails verschickt ueberall in der Applikation verstreut wird, dann fuehrt das zu grosser Unuebersichtlichkeit und zu Fehlern. Vielmehr sollte der Quellcode der die E-Mails verschickt an einer zentralen stelle konzentriert sein. Ähnlich dem DAO-Pattern

Es empfiehlt sich in der BusinessLogic eine Klasse (PostOffice) zu haben, welche alle E-Mails verschickt. Die Klasse sollte, ausser private Getter und Setter und Hilfsmethoden, zu jeder E-Mail eine public Methode haben.

Nehmen wir an wir haben eine Anwendung die zwei E-Mails verschickt.

- registMail
- newPasswordMail

Dann würde die PostOffice Klasse dazu so aussehen.

```
01
02 public class PostOffice Serializable {
03
04     public String sendRegistMail(String mandant, Locale locale, .. more params ..){
05         doSomething();
06         return "success";
07     }
08
09
10     public String sendNewPasswordMail(String mandant, Locale locale, .. more params ..){
11         doSomething();
12         return "success";
13     }
14 }
```

Die wichtigste Klasse im Framework ist die `org.ploin.pmf.impl.MailFactory`. Von dieser Klasse sollte eine Instanz in der Klasse `PostOffice` vorhanden sein. Die Klasse `PostOffice` sollte am besten den Scope Singleton haben. Mit einem IoC Container wie Spring lässt sich das sehr leicht einrichten.

```
<bean id="mailFactory" class="org.ploin.pmf.impl.MailFactory" />  
  
<bean id="postOffice" class="de.company.project.java.service.businesslogic.impl.PostOffice" >  
  <property name="mailFactory" ref="mailFactory" />  
</bean>
```

Im Mail-Verzeichnis sollte zu jedem Mandanten ein Unterverzeichnis angelegt werden mit dem Mandantennamen. Mit den zwei Mandanten mandant1 und mandant2 haetten wir also folgende Verzeichnisstruktur:

- de/company/project/resources/mail
- de/company/project/resources/mail/mandant1
- de/company/project/resources/mail/mandant2

Templates werden zuerst immer im Mandantenverzeichnis gesucht. Wenn hier keins gefunden wird, dann sucht das Framework im mail-directory (de/company/project/resources/mail). Wenn hier auch nichts gefunden wurde dann wird zu guter letzt im Root-Classloader gesucht. Das sind die drei moeglichen Orte an denen Templates platziert werden koennen.

Die MailFactory Klasse enthaelt die Methode

```
public String sendMail(String mailname, Map<String, Object> map);
```

mit der alle Templates verschickt werden koennen. Ein gueltiger Aufruf ist der nachfolgende Code.

```
mailFactory.sendMail("RegisterMail", map);
```



Die zentrale Datenkomponente im Framework ist eine Map, die als zweiter Parameter in der sendMail Methode uebergeben wird. Die keys in der Map sind entweder E-Mail-Propertys aus der email API oder die Strings die in den Templates ersetzt werden sollen. Aber hier nun ein Beispiel.

```
02 public String sendRegisterMail(final String toEmail, final String toName,  
03     boolean sex, final String mandantenString, final String mandant,  
04     final String credential){  
05     String link = getUri(mandantenString);  
06     String reglink = link + "&credential=" + credential;  
07     Map<String, Object> map = new HashMap<String, Object>();  
08     putEmbeds(map, mandant);  
09     map.put("toEmail", toEmail);  
10     map.put("toName", toName);  
11     map.put("subject", "Registrierung");  
12     map.put("mandant", mandant);  
13     map.put(":ansprache:", getAnsprache(toName, sex));  
14     map.put(":link:", link);  
15     map.put(":reglink:", reglink);  
16     String ret = null;  
17     try {  
18         ret = mailFactory.sendMail("RegisterMail", map);  
19     } catch (Exception e) {  
20         log.error("ERROR by sending RegisterMail " + e);  
21     }  
22     if (ret == null){  
23         log.error("ERROR in sendRegisterMail");  
24     }  
25     return ret;  
26 }
```



Jetzt stellt sich natuerlich die Frage, woher weis das Framework ueber welchen E-Mailserver es die E-Mails verschicken soll. Vielleicht besteht jeder Mandant auf seinen eigenen E-Mailserver.

Zugangsdaten zu den E-Mailservern werden in der Datei mail.properties konfiguriert. Die Datei kann sich in den drei Verzeichnissen befinden in dem sich auch die Templates befinden.

```
smtp.0.host=100.168.50.123
smtp.0.fromEmail=hola@comp.de
smtp.0.fromName=portalserver foobar
smtp.0.authUser=
smtp.0.authPassword=
smtp.0.replyTo=dieWaldFee@comp.de

mail.contactForm.toEmail=fee@comp.de
mail.contactForm.toName=Info Mitarbeiter

mail.reglink=http://www.server.de/reg/
```

Aber das ist noch nicht alles. Auf die Werte aus der mail.properties, kann in JSTL Schreibweise zugegriffen werden. Wenn wir ein Kontaktformular haben dessen Inhalt fuer jeden Mandanten an die gleiche E-Mailadresse geschickt werden soll, dann macht es Sinn diese E-Mailadresse ebenfalls in der mail.properties abzulegen. In der Map kann mit folgender Zeile auf diesen Wert zugegriffen werden.

```
map.put("toEmail", "${mail.contactForm.toEmail}");
```

Das gleiche Prinzip kann auch auf die Ersetzungen im Template angewandt werden. Wenn ein Registrierungslink immer gleich ist dann wuerde dieser auch in der mail.properties abgelegt werden.

```
map.put(":reglink:", "${mail.reglink}");
```

Fuer Internationalisierung bietet Java das Konzept der ResourceBundles. Genau in der gleichen Manier werden auch in diesem Framework die Templates abgelegt, mit dem Aufbau "<name>\_<locale>.<endung>". Hier ein Beispiel.

- RegisterMail\_de.html
- RegisterMail\_en.html

```
map.put("locale", "en");
```

oder

```
java.util.Locale locale = java.util.Locale.UK;  
map.put("locale", locale);
```

Zu Sonderzeichen in E-Mails habe ich bereits einen Blogeintrag geschrieben.

<http://ploin.de/robertReiz/blogShow.action?blogId=64>

Die JavaMailFactory setzt das Charset immer explizit auf ISO\_8859\_1, sofern nichts anderes angegeben ist. Damit werden alle Sonderzeichen korrekt verschickt.

# Noch Fragen?