Vorlagenmanagement für *Mail*-Service

Ing. Thomas Herzog

BACHELORARBEIT

Nr. S1310307011-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Juli 2015

Diese Arbeit entstand im Rahmen des Gegenstands

Gegenstand??

im

Semester??

Betreuer:

FH-Prof. DI Dr. Dobler

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 14. Juli 2015

Ing. Thomas Herzog

Inhaltsverzeichnis

Kurzfassung

TODO: Add german summary here

Abstract

TODO: Add english summary here

Kapitel 1

Einleitung

Die vorliegende Sachlage beschäftigt sich mit der Konzeption und Implementierung eines Vorlagenmanagement für den, in der theoretischen Bachelorarbeit konzipierten, *Mail*-Service. Das Vorlagenmanagement stellt einen essentiellen Teil des *Mail*-Service dar, mit dem sich parametrisierte *E-Mail*-Vorlagen erstellen lassen. Das Vorlagenmanagement soll es den BenutzerInnen ermöglichen einfach eigene parametrisierte *E-Mail*-Vorlagen zu erstellen, die in einer Anwendung, die den *Mail*-Service nutzen, verwendet werden können, um benutzerdefinierte *E-Mail*-Nachrichten zu versenden. Mit dem Vorlagenmanagement ist es nicht mehr erforderlich die *E-Mail*-Vorlagen statisch zu definieren und die *E-Mail*-Vorlagen können von den BenutzerInnen nach ihren Wünschen angepasst werden.

Aufgrund des Umfangs des konzipierten *Mail*-Service wurde entschieden sich vorerst auf das Vorlagenmanagement zu konzentrieren. Das Vorlagenmanagement wird für den *Mail*-Service entwickelt, könnte jedoch ohne weiteres auch in anderen Anwendungen verwendet werden, sofern diese Anwendungen die technischen Voraussetzungen erfüllen. Das Vorlagenmanagement wird als eigene Softwarekomponente entwickelt und wird keine Abhängigkeiten auf Ressourcen des *Mail*-Service aufweisen.

1.1 Das Unternehmen curecomp Software Service GmbH

Das Vorlagenmanagement wird in Zusammenarbeit mit dem Unternehmen *curecomp Software Service GembH* erstellt. Das Unternehmen *curecomp* ist ein ein Dienstleister im *Supplier-Relationship-Management (SRM)* und betreibt eine eigene Softwarelösung namens *clevercure*. Die Softwarelösung *clevercure* besteht aus den folgenden Anwendungen:

- *CleverWeb* ist eine *Web*-Anwendung für den webbasierten Zugriff auf

clevercure.

- *CleverInterface* ist eine Schnittstellenanwendung für den XML-basierten Datenimport und Datenexport zwischen *clevercure* und den *ERP*-Systemen der Kunden.
- *CleverSupport* ist eine unternehmensinterne *Web*-Anwendung zur Unterstützung für die Abwicklung von *Support*-Prozessen.
- *CleverDocument* ist ein Dokumentenmanagementsystem für die Verwaltung aller anfallender Dokumente innerhalb von *clevercure*.
- *CCMail* ist die bestehende *Mail*-Anwendung für den Versand der *E-Mail*-Nachrichten innerhalb von *clevercure*, die durch *CleverMail* abgelöst werden soll.

Das Vorlagenmanagement wird von den Anwendungen innerhalb von *clevercure* verwendet werden bevor der *Mail*-Service fertiggestellt wird, da es bereits Softwarekomponenten innerhalb der Anwendungen von *clevercure* gibt, die darauf angewiesen sind.

1.2 Das Vorlagenmanagement für den *Mail*-Service

Mit dem Vorlagenmanagement können *E-Mail*-Vorlagen einerseits von den EntwicklerInnen und BenutzerInnen benutzerdefiniert und parametrisiert erstellt werden. Damit können *E-Mail*-Vorlagen dynamisch auch zur Laufzeit erstellt, modifiziert und gelöscht werden. Es sind keine statischen *E-Mail*-Vorlagen mehr nötig und alle damit verbunden Nachteile wie z.B.

- das neu Kompilieren und Einspielen bei Änderungen der *E-Mail*-Vorlagen,
- keine Möglichkeit für benutzerdefinierten Vorlagen oder
- keine Möglichkeit der Nutzung von dynamischen Parametern in den *E-Mail*-Vorlagen

eliminiert werden. Das Vorlagenmanagement kann auch in einem anderen Kontext verwendet werden, wobei sich die vorliegende Sachlage ausschließlich mit der Verwendung des Vorlagenmanagement innerhalb des *Mail*-Service beschäftigen wird. Obwohl das Vorlagenmanagement als eigene Softwarekomponente implementiert wird, wird die vorliegende Sachlage aufzeigen, wie sich das Vorlagenmanagement in Anwendungen im Kontext von *E-Mail*-Vorlagen verwendet lässt.

1.3 Die Rahmenbedingungen

Das Vorlagenmanagement wird in Java in der Version 8 implementiert und wird sich an der *Java-Enterprise-Edition 7 (JEE-7)* Spezifikation orientieren, wobei folgende Teilspezifikationen Anwendung finden werden:

- *Java-Persistence-API 2.1 (JPA)* ist die Spezifikation für die Persistenz.
- *Context and dependency Injection 1.1 (CDI)* ist die Spezifikation für kontextabhängige Injektion innerhalb einer *JEE7*-Umgebung.
- *Java-Server-Faces 2.2 (JSF)* ist die Spezifikation der *View*-Technologie.

Damit wird das Vorlagenmanagement mit den aktuellsten Standards implementiert und wird daher für die Zukunft gut gewappnet sein. Die Funktionalität des Vorlagenmanagement wird weitestgehend ohne die Verwendung spezieller Bibliotheken implementiert. Es werden Integrationen des Vorlagenmanagement in die folgende Technologien implementiert:

- Integration in *CDI*:
  Innerhalb einer *CDI*-Umgebung werden Ressourcen des Vorlagenmanagements kontextabhängig zur Verfügung gestellt.
- Integration in *JSF*:
  Mit der *View*-Technologie *JSF* wird eine Webseite erstellt, über welche die Vorlagen verwaltet werden.
- Integration in *Typescript*:
  Mit *Typescript* wird ein *Plugin* für den *Rich-Editor CKEditor* implementiert, welches die Variablen für eine *E-Mail*-Vorlage innerhalb des *CKEditors* verwaltet.

Als Entwicklungsumgebung wird die *IDE Intellij* verwendet, die eine bekannte Entwicklungsumgebung im *Java*-Umfeld ist und ein Produkt des Unternehmens *Jetbrains* mit Sitz in Tschechien ist. Als Applikationsserver wird *Wildfly 10.0.0*, vormals *JbossAS* genannt, des Unternehmens *Redhat* verwendet, der ein zertifizierter *JEE-7*-Server ist und somit alle benötigten Spezifikationen unterstützt. Es wird so weit wie möglich vermieden Bibliotheken von Drittanbietern zu verwenden, außer sie sind für die Funktionalitäten des Vorlagenmanagements unerlässlich oder bieten einen essentiellen Vorteil.

Kapitel 2

Das Ziel des Projekts

Ziel ist es die Softwarekomponente Vorlagenmanagement für den *Mail*-Service zu implementieren, mit dem *E-Mail*-Vorlagen erstellt und verwaltet werden können. Das Vorlagenmanagement stellt einen essentiellen Teil des *Mail*-Service dar und wird auch von mehreren Anwendungen verwendet werden. Die verschiedenen Anwendungen, die das Vorlagenmanagement verwenden, sind ebenfalls in Java implementiert, werden aber in unterschiedlichen Laufzeitumgebungen betrieben wie z.B.:

- *IBM-Integration-Bus (IIB)*
  ist ein proprietäres Produkt des Unternehmens *IBM*, für *XML*-Konvertierungen und den *XML*-basierten Datenimport und Datenexport.
- *Wildfly*
  ist ein zertifizierter *JEE-7* Applikationsserver des Unternehmens *Redhat*.

Die verschiedenen Anwendungen von *clevercure* sollen mit geringsten Aufwand in der Lage sein *E-Mail*-Vorlagen zu verwenden und *E-Mail*-Nachrichten auf Basis dieser *E-Mail*-Vorlagen zu erstellen. Dabei sollen die Abhängigkeiten der Anwendungen zu dem Vorlagenmanagement so gering wie möglich gehalten werden, sowie nur vorgegebene Schnittstellen verwendet werden. Wird eine *E-Mail*-Nachricht von einer Anwendung auf Basis einer *E-Mail*-Vorlage erstellt, so müssen dessen enthaltene Variablen beim Zeitpunkt des Erstellens der *E-Mail*-Nachricht aufgelöst und serialisiert werden, damit die *E-Mail*-Nachricht mit denselben Daten zu jedem Zeitpunkt erneut versendet werden kann. Für die Anwendungen soll nicht erkennbar sein wie die *E-Mail*-Nachrichten nach ihrer Erstellung weiter verwendet werden. Zurzeit interagieren die Anwendungen direkt mit der Datenbank anstatt von ihr abstrahiert zu sein und sind daher stark an die bestehende Anwendung *CCMail* gekoppelt bzw. an dessen Datenbankschema.

2.1 Die funktionalen Ziele

Für das Vorlagenmanagement wurden die folgende funktionalen Anforderungen definiert.

2.1.1 Die Persistenz der Vorlagen

Die Vorlagen müssen innerhalb einer Datenbank persistent gehalten werden. Da das Vorlagenmanagement vorerst exklusiv für den *Mail*-Service verwendet wird, soll die Persistenz der Vorlagen innerhalb des *Mail*-DB-Schema realisiert werden. Die persistenten Vorlagen müssen versioniert werden, damit diese von anderen Entitäten referenziert werden können, ohne dass die Gefahr besteht, dass sich die referenzierte Vorlage geändert hat, wodurch die Konsistenz verloren gehen würde. Persistente Vorlagen müssen explizit freigegeben werden bevor diese verwendet dürfen. Nach einer Freigabe darf die Vorlage nicht mehr geändert werden.

2.1.2 Die Mehrsprachigkeit der Vorlagen

Die Vorlagen müssen in mehreren Sprachen erstellt und verwaltet werden können, wobei eine Sprache als Standardsprache zu definieren ist und es für diese Sprache immer einen Eintrag geben muss. Auf die Standardsprache wird zurückgegriffen, wenn es für eine angeforderte Sprache keinen Eintrag gibt. Somit ist gewährleistet, dass immer eine Vorlage für jede angeforderte Sprache zur Verfügung steht. Es ist nicht erforderlich dass dieselben Variablen über alle definierte Sprachen gleich sind. Es dürfen in einer Vorlage, die in mehreren Sprachen definiert wurde, eine unterschiedliche Anzahl oder unterschiedliche Variablen definiert sein.

2.1.3 Die Variablen für die Vorlagen

Die Vorlagen werden für einen bestimmten *Mail*-Typ definiert, der einen bestimmten Kontext darstellt wie z.B.

- ein Benutzer wurde erstellt,
- eine Bestellung wurde erstellt oder
- ein Dokument wurde hochgeladen.

Für die Vorlagen, die für einen bestimmten *Mail*-Typ erstellt werden können, sollen Variablen zur Verfügung gestellt werden können wie z.B.:

- *CURRENT_USER* ist der Benutzer, der die *E-Mail*-Nachricht erstellt halt.
- *ORDER_NUMBER* ist die Nummer der erstellten Bestellung.

Die EntwicklerInnen sollen für einen bestimmten *Mail*-Typ in der Lage sein einfach Variablen zu definieren, die von den BenutzerInnen beim Erstellen einer Vorlage für den korrespondierenden *Mail-Typ* frei verwendet werden können. Die Variablen sollen auch global definiert werden können und in allen Vorlagen anwendbar sein. Die EntwicklerInnen müssen in der Lage sein die Menge der zur Verfügung stehenden Variablen zur Laufzeit aufgrund von bestimmten Zuständen verändern zu können. Die Menge der Variablen könnte z.B von Berechtigungen der BenutzerInnen abhängig sein.

2.1.4 Die Mehrsprachigkeit der Variablen

Die zur Verfügung stehenden Variablen werden durch die EntwicklerInnen statisch definiert und müssen einen Titel und eine Beschreibung einer Variable zur Verfügung stellen. Der Titel und die Beschreibung der Variable müssen mehrsprachig zur Verfügung stehen, wobei als Standardsprache Englisch zu verwenden ist.

2.1.5 Die automatische Registrierung der Variablen

Innerhalb einer *CDI*-Umgebung sollen die definierten Variablen beim Start des *CDI-Containers* automatisch gefunden und registriert werden. Die automatische Registrierung der Variablen soll mit einer *CDI-Extension (javax.inject.spi.Extension)* realisiert werden, die beim Start des *CDI-Containers*, die Variablen findet und registriert. Mit einer automatischen Registrierung der variablen wird erreicht das neu definierte Variablen automatisch gefunden und registriert werden und somit nicht manuell registriert werden müssen, was ein gewisses Risiko in sich birgt, wenn Variablen vergessen werden.

2.1.6 Die Verwaltung der Vorlagen über eine Webseite

Die Vorlagen sollen über eine Webseite verwaltet werden können. Die Webseite soll mit der *View*-Technologie *JSF* implementiert werden. Über einen *FacesConverter* soll die Vorlage von der *View*-Repräsentation in die Repräsentation der verwendeten *Template-Engine* konvertiert werden.
Das folgende *HTML-Markup* enthält die Variablen in ihrer *HTML*-Repräsentation wie sie in dem *Rich-Editor* verwendet wird.
Der folgende Text stellt das konvertierte *HTML-Markup* aus 2.2 als *Freemarker-Template* dar.
Als *Rich-Editor* soll *CKEditor* verwendet werden, da es für diesen *Rich-Editor* von *primefaces-extensions* eine *JSF*-Integartion in Form einer *JSF*-Komponente zur Verfügung gestellt wird. Dadurch entfällt die Integration eines reinen *Javascript Rich-Editors* der keine Integartion in den Lebenszyklus von *JSF* hat und daher auch keine *AJAX*-Events unterstützt, die von *JSF* verarbeitet werden können.

Programm 2.1: *HTML-Markup* einer Vorlage

```
<p>Das ist eine Variable:</p>
<p>
  <span class="variable" title="Beschreibung" data-variable="VAR_1">
    Variable 1
  </span>
</p>
```

Programm 2.2: Konvertiertes *HTML-Markup* als *Freemarker-Template*

```
<p>Das ist eine Variable:</p>
<p>
  ${module.core.VariableHolder["VAR_1"]!("Variable nicht gefunden")}
</p>
```

2.2 Die technischen Ziele

Als technische Ziele wurde die Implementierung in *Java 8*, die Integration in eine *CDI*-Umgebung und die komponentenorientierte Entwicklung- des Vorlagenmanagements definiert. Das Templatemanagement soll als eine eingeständige Softwarekomponenete implementiert werden, die ohne großen Aufwand in anderen Anwendungen verwendet werden kann, sofern die technischen Vorraussetzunge wie die Version von *Java* und die Unterstützung der verwendeten Bibliotheken, erfüllt sind. Das Vorlagenmanagement soll Schnittstellen definieren, die Funktionalitäten des Vorlagenmanagements nach außen offenlegen, ohne dass die Anwendungen in Berüghrung mit konkreten Implementierungen kommen.

Kapitel 3

Das Lösungskonzept

In diesem Kapitel wird der Lösungsansatz und die Spezifikation des Vorlagenmanagements erörtert. Bei der Spezifikation handelt es sich um Schnittstellen und abstrakte Klassen, die die Struktur des Vorlagenmanagements definieren. Diese Schnittstellen und abstrakten erlauben es Implementierungen für verschiedene *Template-Engines* wie z.B

- *Freemakrer*,
- *Velocity* oder
- *Thymeleaf*

zur Verfügung zu stellen, wobei die abstrakten Klassen die gemeinsam nutzbare Logik implementieren, die über die verschiedenen *Template-Engines* verwendet werden kann.

Mit der Möglichkeit die verschiedensten *Template-Engines* verwenden zu können, wird erreicht dass das Vorlagenmanagement sehr flexibel ist. Bei dem Wechsel zu einer anderen *Template-Engine* müssen nur die *Expressions* einer Vorlagen in die *Template-Engine* spezifischen *Expressions* umgewandelt werden.

3.1 Die Spezifikation des Vorlagenmanagements

Dieses Kapitel behandelt die erstellten Spezifikationen des Vorlagenmanagements. Auf Basis dieser Spezifikationen wird das Vorlagenmanagement und die Integrationen in die verschiedenen Umgebungen und Technologien implementiert. Die erstellte Spezifikationen sind frei von Abhängigkeiten auf konkrete Implementierungen jeglicher Art. Sie haben nur Abhängigkeiten auf andere Spezifikationen wie die *JEE-7* Spezifikation.

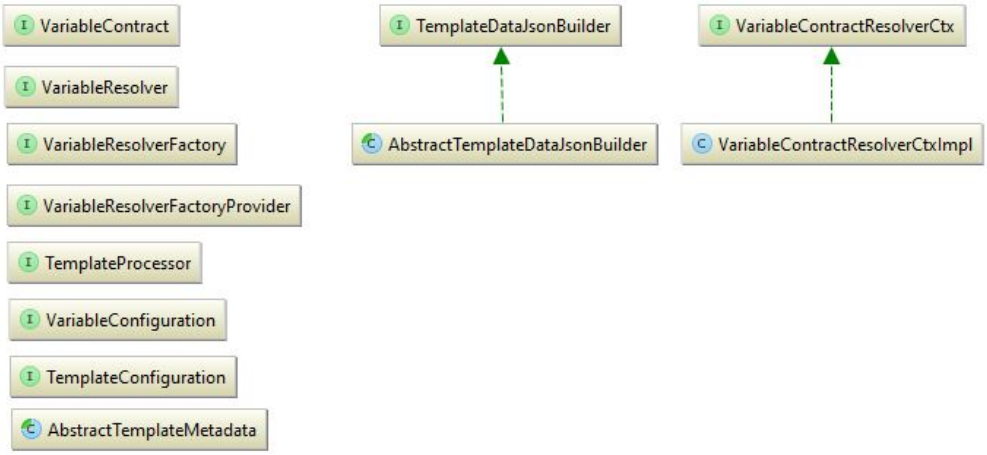


Abbildung 3.1: Klassenhierarchie der Vorlagen-*API*

3.1.1 Die Schnittstellen und abstrakten Klassen

Dieser Abschnitt behandelt die implementierten Schnittstellen und abstrakten Klassen des Vorlagenmanagements. Die abstrakten Klassen implementieren die gemeinsam nutzbaren Funktionalitäten, welche von alle konkreten Implementierungen des Vorlagenmanagements genutzt werden können. Diese Spezifikationen spezifizieren Aspekte des Vorlagenmanagements wie

1. das Variablenmanagement innerhalb des Vorlagenmanagement,
2. die Handhabung von Variablen in einer Vorlage
3. die Abbildung der Metadaten einer Voralge und
4. das Erstellen des *JSON*-Objekts, welches die Daten für die Vorlage beinhaltet.

Die Schnittstelle *VariableContract*

Die Schnittstelle *VariableContract* spezifiziert eine Variable, die in einer Vorlage verwendet werden kann. Objekte dieser Schnittstelle werden beim Anwendungsstart registriert und können grundsätzlich in allen Vorlagen ver-

wendet werden. Eine Variable ist einem Modul zugeordnet, in dem die Variable bezüglich ihres Namen eindeutig sein muss. Das Modul wird über einen *String* definiert. Die Mehrsprachigkeit einer Variable wird über Enumerationen realisiert, wobei jede Variable jeweils einen Schlüssel für den Titel und die Beschreibung bereit stellt.

Da es sich bei einer Variable um statische Daten handelt, also die Variablen sind schon zur Kompilierungszeit bekannt, ist angedacht, dass die Variablen mit dem *Java*-Typ *enum* implementiert werden, dass die Schnittstelle *VariableContract* implementiert. Durch die Abbildung der Variablen mit dem *Java*-Typ *enum* können mehrere Variablen in einer Klasse definiert werden, wobei eine einzelne Enumeration ein Objekt der Schnittstelle *VariableContract* darstellt. Alle Variablen die mit einer *enum* abgebildet werden, sollten demselben Modul zugeordnet sein, obwohl dies nicht zwingend erforderlich ist. Die Zuordnung der Variablen zum selben Modul erleichtert die Wartung und Strukturierung der Variablen. Die Variablen, die mit einer *enum* definiert wurden, werden innerhalb des Vorlagenmanagements trotzdem als einzelne Objekte der Schnittstelle *VariableContract* betrachtet. Die Tatsache dass die Variablen mit einer *enum* abgebildet wurden, ist für das Vorlagenmanagement nur beim Registrieren der Variablen von belang und nicht bei deren weiterer Verwendung.

Eine Variable ist über seine *Id* eindeutig referenzierbar, wobei sich die *Id* aus dem Modulnamen und den Variablennamen zusammensetzt (Bsp. module.core.VAR_1). Die *Id* hält sich dabei an die Namenskonvention eines *Java*-Paketnamen. Da der Variablenname immer auf diese Weise zusammengesetzt werden sollte, wurde die Methode *String getId();* als *default Methode* implementiert, was seit *Java 8* möglich ist. Ein EntwicklerIn muss diese Methode nicht mehr implementieren, obwohl es immer noch möglich ist diese Methode zu überschreiben. Auch die Methode *String toInfoString()* wurde als *default Methode* implementiert, da auch diese Methode nicht von den EntwicklerInnen implementiert werden sollte, da ihre Funktionalität sich nicht ändern sollte.

Programm 3.1: VariableContract.java

```
1 public interface VariableContract extends Serializable {
2
3     String getName();
4
5     String getModule();
6
7     Enum<?> getInfoKey();
8
9     Enum<?> getLabelKey();
10
11    default String getId() {
12        return getModule() + "." + getName();
13    }
14
15    default String toInfoString() {
16        final String ls = System.lineSeparator();
17        final StringBuilder sb = new StringBuilder();
18        sb.append("contract  : ").append(this.getClass().getName())
19          .append(ls)
20          .append("id        : ").append(getId())
21          .append(ls)
22          .append("name      : ").append(getName())
23          .append(ls)
24          .append("label-key : ").append((getLabelKey() != null)
25                                         ? getLabelKey().name()
26                                         : "not available")
27          .append(ls)
28          .append("info-key  : ").append((getInfoKey() != null)
29                                         ? getInfoKey().name()
30                                         : "not available")
31          .append(ls)
32          .toString();
33    }
34 }
```

Die Schnittstelle *VariableResolver*

Die Schnittstelle *VariableResolver* spezifiziert wie der aktuelle Wert der Variablen aufgelöst wird. Eine Variable wird in einer Vorlage verwendet und beim Erstellen eines Datenbankeintrags, der diese Vorlage verwendet, müssen die aktuellen Werte der beinhalteten Variablen aufgelöst werden. Da der aktuelle Wert der Variable kontextabhängig ist, wird beim Auflösen des Werts der Variable ein Kontext bereitgestellt, über den kontextabhängige Daten vom EntwicklerIn bereitgestellt werden können, die in einer Implementierung der Schnittstelle *VariableResolver* angewendet werden können. Dadurch kann die Variable in mehreren Kontexten verwendet werden und

auch kontextabhängig aufgelöst werden.

Programm 3.2: VariableResolver.java

```
1 @FunctionalInterface
2 public interface VariableResolver {
3
4     String resolve(VariableContract variable,
5                    VariableContractResolverContext ctx);
6 }
```

Die Schnittstelle *VariableResolver* wurde als *FunctionalInterface* implementiert. Ein *FunctionalInterface* ist eine Schnittstelle, die nur eine abstrakte Methode definiert, die implementiert werden muss. Eine Implementierung eines *FunctionalInterface* kann über eine *Lambda*-Funktion oder Methodenreferenz bereitgestellt werden, wodurch die Notwendigkeit einer anonymen Implementierung oder der Implementierung einer Klasse für diese Schnittstelle entfällt. Dieser Ansatz macht den Quelltext lesbarer, obwohl angemerkt sei, dass dieser Ansatz sich negativ auf das Laufzeitverhalten auswirkt, was in der Art und Weise der Ausführung einer *Lambda*-Funktion oder Methodenreferenz begründet ist. Die negativen Auswirkungen auf das Laufzeitverhalten können, im Bezug auf das Vorlagenmanagement, vernachlässigt werden.

Die Schnittstelle *VariableResolverFactory*

Die Schnittstelle *VariableResolverFactory* spezifiziert wie Objekte der Schnittstelle *VariableResolver* produziert werden. Objekte dieser Schnittstelle können Objekte der Schnittstelle *VariableResolver* für jede Implementierung der Schnittstelle *VariableContract* produzieren, obwohl es zu empfohlen ist, dass es eine Implementierung der Schnittstelle *VariableResolverFactory* je Modul zur Verfügung gestellt wird.

Programm 3.3: VariableResolverFactory.java

```
1 @FunctionalInterface
2 public interface VariableResolverFactory extends Serializable {
3
4   VariableResolver getVariableResolver(VariableContract contract,
5                                        VariableContractResolverCtx ctx);
6 }
```

Die Schnittstelle *VariableResolver* wurde ebenfalls als *FunctionalInterface*

implementiert, damit Implementierungen über eine *Lambda*-Funktion oder eine Methodenreferenz bereitgestellt werden kann.

Die Schnittstelle *VariableResolverFactoryProvider*

Die Schnittstelle *VariableContractFactoryProvider* spezifiziert wie Objekte der Schnittstelle *VariableResolverFacotry* produziert werden. Ein Objekt der Schnittstelle *VariableResolverFactoryProvider* kann Objekte der Schnittstelle *VariableResolverFactory* für die Schnittstelle *VariableContract*, einer Ableitung von dieser Schnittstelle oder einer konkreten Implementierung dieser Schnittstelle zur Verfügung stellen.

Programm 3.4: VariableResolverFactoryProvider.java

```
1 @FunctionalInterface
2 public interface VariableResolverFactoryProvider extends Serializable {
3
4     VariableResolverFactory getVariableResolverFactory
5             (Class<? extends VariableContract> contractType);
6 }
```

Die Schnittstelle *VariableResolverFactoryProvider* wurde auch als *FunctionalInterface* implementiert um Implementierungen über eine *Lambda*-Funktion oder Methodenreferenz zur Verfügung stellen zu können.

Die Schnittstelle *VariableContractResolverCtx*

Die Schnittstelle *VariableContractResolverCtx* spezifiziert den Kontext, der bei der beim Auflösen des aktuellen Werts einer Variablen zur Verfügung gestellt wird. Dieser Kontext stellt alle Daten bereit, die beim Auflösen des aktuellen Werts einer Variable benötigt werden. Es wird auch ermöglicht, dass Benutzerdaten im Kontext definiert werden können, die bei beim Auflösen des aktuellen Werts einer Variable verwendet werden können. Es wurde bewusst vermieden, dass beim Auflösen eines aktuellen Werts einer Variable bekannt ist, in welcher Vorlage die Variable verwendet wird. Dadurch bleibt die Handhabung der Variablen einer Vorlage entkoppelt von der Vorlage selbst. Dadurch wäre es möglich die Variablen außerhalb vom Vorlagenmanagements zu verwenden.

Programm 3.5: VariableContractResolverCtx.java

```
1 public interface VariableContractResolverCtx {
2
3     Locale getLocale();
4
5     ZoneId getZoneId();
6
7     TimeZone getTimeZone();
8
9     <T> T getUserData(Object key,
10                      Class<T> clazz);
11 }
```

Die Schnittstelle *TemplateProcessor*

Die Schnittstelle *TemplateProcessor* spezifiziert wie die Vorlagen behandelt werden. Objekte dieser Schnittstelle können Variablen in einer Vorlage, einer bestimmten *Template-Engine* finden und konvertieren. Ein *TemplateProcessor* muss ebenfalls in der Lage sein ungültige Variablen innerhalb einer Vorlage zu finden, wobei eine ungültige Variable eine Variable ist, die nicht registriert ist und somit auch der aktuelle Wert der Variable nicht aufgelöst werden kann.

Eine konkrete Implementierung dieser Schnittstelle ist eine Implementierung für eine bestimmte *Template-Engine*, da die in der Vorlage verwendeten *Expressions* spezifisch für die verwendete *Template-Engine* sind.

Besonders sind beiden folgenden Methoden hervorzuheben.

```
String replaceExpressions(String template,
                          Function<VariableContract, String> converter);

String replaceCustom(String template,
                     Pattern itemPattern,
                     Function<String, String> converter);
```

Diese Methoden verwenden als Formalparameter für den benötigte Konverter ein *FunctionalInterface* namens *Function*, welches von der Sprache *Java 8* bereitgestellt wird. Dadurch ist das Spezifizieren einer eigenen Schnittstelle für die Konvertierung nicht mehr nötig. Der Konverter kann über eine *Lambda*-Funktion oder Methodenreferenz bereitgestellt werden. Dadurch ist die Konvertierung der Variablen einer Vorlage abstrahiert von der Implementierung der Schnittstelle *TemplateProcessor*, wodurch die Variablen durch eine beliebige Repräsentation ersetzt werden können und visa versa.

Programm 3.6: TemplateProcessor.java

```
1 public interface TemplateProcessor {
2
3     String replaceExpressions(String template,
4                               Function<VariableContract, String>
      converter);
5
6     String replaceCustom(String template,
7                          Pattern itemPattern,
8                          Function<String, String> converter);
9
10    Set<VariableContract> resolveExpressions(String template);
11
12    Set<String> resolveInvalidExpressions(String template);
13
14    String variableToExpression(VariableContract contract);
15
16    VariableContract expressionToVariable(String expression);
17 }
```

Die Schnittstelle *TemplateDataJsonBuilder*

Die Schnittstelle *TemplateDataJsonBuilder* spezifiziert die Signatur eines *Builders*, der das *JSON*-Objekt erstellt, welches die Daten für das Parsen einer Vorlage enthält. Eine Anforderung ist es, die *E-Mail*-Nachrichten persistent zu halten, wobei nach der Erstellung einer *E-Mail*-Nachricht, dessen Inhalt unveränderbar sein soll. Es werden die Metadaten wie die Sprache sowie die aufgelösten Werte der in der Vorlage enthaltenen Variablen mit einem *JSON*-Objekt persistent gehalten. Es könnte auch die Vorlage geparst werden und die gesamte geparste Vorlage persistent gehalten werden, wodurch aber die Menge an persistent gehaltenen Daten stark ansteigen würde. Da nur die Metadaten und die Werte der Variablen persistent gehalten werden, wird die Menge an persistent gehaltenen Daten so klein wie möglich gehalten, da nur die variablen Teile einer Vorlage für eine *E-Mail*-Nachricht persistent gehalten werden. Mit diesem *JSON*-Objekt kann die korrespondierende Vorlage zu jedem Zeitpunkt mit demselben Resultat erneut geparst werden.

Es wurde hier das *Builder*-Muster angewendet, da sich die Konfiguration des *Builders* mit einer *Fluent-API*, wie bei einem *Builder* üblich, sehr gut abbilden lässt. Die Schnittstelle *TemplateDataJsonBuilder* spezifiziert folgende Terminalmethoden.

- *TemplateRequestJson toJsonModel()* ist die Methode, die das *JSON*-Objekt in Form eines *Java*-Objekts zurückliefert.

- *String toJsonString()* ist die Methode, die das *JSON*-Objekt als String zurückliefert.
- *Map<String, Object> toJsonMap()* ist die Methode, die das *JSON*-Objekt in Form einer *java.util.Map* zurückliefert.

Folgender Quelltext illustriert, wie der *Builder* verwendet wird.

```
builder.withStrictMode()
       .withLocalization(localeObj, zoneIdObj)
       .withTemplate(templateMetadataObj)
       .withUserData(userDataMap)
       .withVariableResolverFactoryProvider(factoryProviderObj)
       .toJsonModel();
```

Programm 3.7: TemplateDataJsonBuilder.java

```
1  public interface TemplateDataJsonBuilder<I,
2      M extends AbstractTemplateMetadata<I>,
3      B extends TemplateDataJsonBuilder> extends Serializable {
4
5      B withWeakMode();
6
7      B withLocalization(Locale locale,
8                         ZoneId zoneId);
9
10     B withUserData(Map<Object, Object> userData);
11
12     B withStrictMode();
13
14     B withVariableResolverFactoryProvider
15                           (VariableResolverFactoryProvider factory);
16
17     B withVariableResolverFactory(VariableResolverFactory factory);
18
19     B withTemplate(M metadata);
20
21     void end();
22
23     B addVariable(VariableContract contract,
24                   Object value);
25
26     B addVariableResolver(VariableContract contract,
27                           VariableResolver resolver);
28
29     TemplateRequestJson toJsonModel();
30
31     String toJsonString();
32
33     Map<String, Object> toJsonMap();
34 }
```

Die abstrakte Klasse *AbstractTemplateMetadata*

Die abstrakte Klasse *AbstractTemplateMetadata* implementiert die Logik, die von allen konkreten Implementierungen dieser abstrakten Klasse für die verschiedene *Template-Engines* genutzt werden kann. Die Metadaten wie

- die Anzahl der gültigen Variablen in der Vorlage,
- die Anzahl der ungültigen Variablen in der Vorlage,
- die Zeichenlänge der Vorlage,
- die eindeutige *Id* der Vorlage,
- die Version der Vorlage und
- die Vorlage selbst

werden in dieser Klasse abgebildet. Diese Metadaten sind unabhängig der verwendeten *Template-Engine* und eine konkrete Implementierung für eine *Template-Engine* kann zusätzliche Metadaten definieren. Die Metadaten werden einmalig ermittelt und sind über die Lebenszeit des Objekts unveränderbar. Wird die Vorlage geändert so muss auch eine neues Objekt der Metadaten erstellt werden.

TODO: Add reference to appendix for this source

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder*

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder* implementiert die gemeinsam nutzbare Logik, die von allen konkreten Implementierungen für die verschiedenen *Template-Engines* verwendet werden kann. Sie stellt ebenso Hilfsmethoden bereit, die Variablen innerhalb der Vorlage finden, validieren und deren aktuellen Wert auflösen können. Das resultierende *JSON*-Objekt des *Builders* ist spezifiziert, jedoch nicht die Abbildung der aufgelösten Werte für die beinhalteten Variablen. Diese Daten sind spezifisch für die verwendete *Template-Engine*. Es könnten auch noch andere Daten für das Verarbeiten einer Vorlage von Nöten sein, die in der *JSON*-Spezifikation nicht vorhanden sind.

TODO: Add reference to appendix for this source

3.2 Die Spezifikation der Vorlagenintegration

Die vorgestellte Spezifikation des Vorlagenmanagements spezifiziert die Kernfunktionalität des Vorlagenmanagements, dass in der Lage ist die Vorlagen sowie deren enthaltene Variablen zu behandeln. Das Vorlagenmanagement benötigt jedoch Integrationen in andere Technologien, Umgebungen und Sprachen, um die benötigte Funktionalitäten wie

- die Verwaltung der Variablen in einem *Javascript* basierten *Rich-Editor*,

- die automatische Registrierung der Variablen in einer *CDI*-Umgebung,
- die Verwaltung der Vorlagen in einer Webseite und
- die Persistenz der Vorlagen

realisieren zu können. Folgender Abschnitt behandelt die Integrationen in die Technologien, Umgebungen und Sprachen und diese Funktionalitäten realisieren zu können.

3.2.1 Das Vorlagen-*Management* in Typescript und Javascript

Es wird ein *Rich-Editor* benötigt mit dem *HTML* basierte Vorlagen in einer Webseite verwaltet werden können. Dieser *Rich-Editor* muss angepasst werden, damit die definierten Variablen in einer Vorlage verwendet werden können. Es soll der *Rich-Editor CKEDitor* verwendet werden, für den es bereits eine Integration in *JSF* in Form einer *JSF*-Komponente gibt, die von *primefaces-extensions* bereitgestellt wird. Es soll ein *Plugin* in *Typescript* entwickelt werden, das es erlaubt die definierten Variablen innerhalb des *Rich-Editors* zu verwalten. Es soll die Skriptsprache *Typescript* verwendet werden, da es mit dieser Skriptsprache möglich ist typsicher zu entwickeln, was in *Javascript* nicht möglich ist. Ebenfalls kann *Typescript* in mehrere *ECMA*-Standars übersetzt werden.

3.2.2 Das Vorlagen-*Management* in CDI

Das Vorlagenmanagement wird in einem *JEE-7*-Anwendungsserver verwendet, der *CDI* bereitstellt. Im *CDI*-Standard sind portable *Extensions* spezifiziert, die es erlauben, dass sich Softwarekomponenten in den *CDI-Container* integrieren. Es soll eine *CDI-Extension* implementiert werden, die beim Start des *CDI-Containers*, die definierten Variablen automatisch registriert und über den Lebenszyklus des *CDI-Containers* persistent. Es sollen Ressourcen des Vorlagenmanagements wie z.B

- Objekte der Schnittstelle *VariableResolver*
- Objekte der Schnittstelle *VariableResolverFactory* oder
- Objekte der Schnittstelle *TemplateDataJsonBuilder*

kontextabhängig zur Verfügung gestellt werden. Dadurch können die Implementierungen der Schnittstelle *VariableResolver* kontextabhängige Ressourcen nutzen. Damit das Variablenmanagement auf diese Objekte zugreifen kann wurde die Schnittstelle *VariableResolverFactoryProvider* spezifiziert, die die Verbindung des Variablenmanagements zu *CDI* herstellt und kontextabhängige Objekte der Schnittstelle *VariableResolverFactory* bereitstellen kann.

3.2.3 Das Vorlagen-*Management* in JSF

Für die Verwaltung der Vorlagen soll eine *JSF*-Webseite implementiert werden. Über diese Webseite sollen variablen erstellt, modifiziert, gelöscht und freigegeben werden können. Für die Verwaltung der Vorlagen soll die von *primefaces-extension* bereitgestellte *JSF*-Komponente für den *Rich-Editor CKEDitor* verwendet werden. Diese Komponente integriert den *Javascript* basierten *CKEDitor* in den *JSF*-Lebenszyklus. Um die Vorlage in die korrespondierende *Template-Engine* spezifische Repräsentation zu überführen, soll ein *FacesConverter* implementiert werden, der die Konvertierung der Vorlage von seiner Repräsentation für die Webseite in die Repräsentation der *Template-Engine* überführt und visa versa.

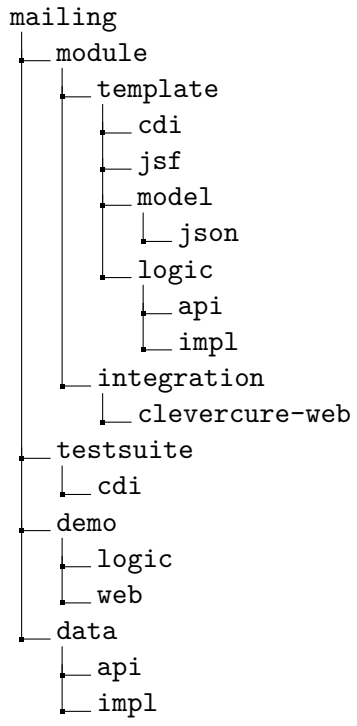
3.2.4 Das Vorlagen-*Management* in *Mail*-DB-Schema

Eine Vorlage wird durch einen *String* repräsentiert, der innerhalb des *Mail*-DB-Schema sprachspezifisch persistent gehalten wird. Es ist ist nicht erforderlich eine eigene Tabellenstruktur für die Vorlagen zu definieren um es von den *Mail*-Tabellen zu abstrahieren, da die Vorlagen einen essentiellen Teil des *Mail*-Service darstellen und daher auch die Vorlagen bzw. deren persistente Repräsentation voll in das *Mail*-DB-Schema integriert werden sollen.

Kapitel 4

Die Realisierung

Folgendes Kapitel befasst sich mit der Implementierung, der im Kapitel 3 vorgestellten Spezifikation des Vorlagenmanagements. Die Implementierung wurde in *Java 8* mit dem *Buildtool Maven* realisiert, wobei die Implementierungen in der folgenden Projektstruktur organisiert wurden.



Das Wurzelprojekt *mailing* organisiert alle Abhängigkeiten für die Unterprojekte sowie die gemeinsame *Build*-Konfiguration, die auf alle konkreten Projekte angewendet werden kann. Ebenfalls enthält es die Metadaten wie die EntwicklerInnen, die an diesem Projekt mitwirken. Die Unterprojekte,

die ebenfalls *Paren*-Projekte bündeln ihre Unterprojekte und organisieren keine Abhängigkeiten und definieren keine Metadaten. Die gesamte Organisation findet im *Paren*-Projekt *mailing* statt. Diese Projektstruktur wurde gewählt, da in diesem Projekt in weitere Folge auch die Implementierungen der anderen benötigten Softwarekomponenten des *Mail-Service* beinhalten wird. Die konkreten Artefakte wurden jeweils in ein Artefakt **-api* und **-impl* aufgeteilt, somit sind die Schnittstellen vollständig getrennt von der Implementierung.

Die folgenden Artefakte resultieren aus dieser Projektstruktur, wobei nur die konkreten Artefakte und nicht die *Parent*-Artefakte angeführt sind.

- ***mailing-module-template-logic-api*** ist das Artefakt, das die Spezifikation des Vorlagenmanagement enthält.
- ***mailing-module-template-logic-impl*** ist das Artefakt, das die Implementierung der Spezifikation des Vorlagenmanagements enthält.
- ***mailing-module-template-cdi*** ist das Artefakt, das die Implementierung für die Integration in einen *CDI-Container* enthält.
- ***mailing-module-template-jsf*** ist das Artefakt, das die Implementierung für die Integartion in *JSF* enthält.
- ***mailing-module-template-model-json*** ist das Artefakt, das die Implementierung der *JSON*-Spezifikation enthält.
- ***mailing-module-integartion-clevercure-web*** ist das Artefakt, das die Implementierung der Integration für die Anwendung *CleverWeb* enthält.
- ***mailing-data-api*** ist das Artefakt, dass die Spezifikation der *Services* enthält.
- ***mailing-data-impl*** ist das Artefakt, das die Implementierung der *Service*-Spezifikation enthält.
- ***mailing-testsuite-cdi*** ist das Artefakt, das die Basis aller Tests, die in einem *CDI-Container* lauffähig sein sollen darstellt.
- ***mailing-demo-web*** ist das Artefakt, das die Demowebanwendung darstellt.

4.1 Die Implementierung der Spezifikationen

Der folgende Abschnitt behandelt die Implementierungen der im Kapitel 3 vorgestellten Spezifikation.

4.1.1 Die Implementierung für *CKEditor*

CKEditor ist ein *Javascript* basierter mit dem die Vorlagen bearbeitet werden können. Wie in 3.2.1 vorgegeben wird ein *Plugin* benötigt, dass innerhalb des *CKEDitor* die Variablen verwalten kann. Diese *Plugin* wurde in *Typescript* implementiert, da hier Typsicherheit vorhanden ist im Gegenzug zu Javascript das nicht typsicher ist. Die Implementierung des *Plugins* in *Typescript* war möglich, da für den *CKEditor* Typinformationen für *Typescript* vom dem *Open-Source* Projekt *DefinitelyTyped* bereitgestellt wird. *Typescript* benötigt Typinformationen für *Javascript* Quelltext, damit die Typsicherheit in *Typescript* gewährleistet werden kann. Würden keine Typinformationen zur Verfügung stehen, hätte man sie selber implementieren müssen.

Der Quelltext befindet sich zurzeit im Projekt *mailing-demo-web*, da die Verwendung eines *Web-Fragment* das Problem mit sich bringt, dass während der Entwicklung die Ressourcen nicht automatisch nachgeladen werden können, was die Entwicklung sehr erschwert. Da es sich aber nur um zwei Quelltexte handelt, die einfach verschoben werden können, stellt das kein Problem dar.

Das *CKEDitor-Plugin* in Typescript

Da das Variablenmanagement unabhängig vom verwendeten *CKEditor* ist, wurde die Verwaltung der Variablen in einem eigenen *Javascript* Modul *cc.variables* zusammengefasst. Das *CKEditor Plugin* wurde im *Javascript* Modul *cc.ckeditor.plugins* zusammengefasst. Das Variablenmanagement in *Typescript* ist verantwortlich für die *Browser* seitige Registrierung der Variablen und stellt Hilfsmethoden zur Verfügung zur Verfügung, mit denen Variablen in der *Registry* gefunden und konvertiert werden können. Folgendes Beispiel soll illustrieren wie eine Variable konvertiert werden kann.

```
// Signature of the converter function
public convertVariables(converter:(item:VariableMapping) => any
                        = (item:VariableMapping)=> item):any[]

// Convert to the variable's set displayName
variablesHandler.convertVariables(
  function (variable) {
    return variable.displayName;
  }
)
```

Die Funktion *convertVariables* definiert den Formalparameter *converter* als eine sogenannte *Arrow*-Funktion, die einer *Lambda*-Funktion in *Java* ähnelt. Mit der *Arrow*-Funktion wird die Signatur der Funktion für die Konvertierung vorgegeben. Ebenfalls wird für den Formalparameter *converter* eine Standardimplementierung definiert, die verwendet wird, sollte der Formal-

parameter *converter* bei Aktivierung der Funktion *convertVariables* nicht gesetzt sein. Der Typ *any[]* ist vergleichbar mit dem Datentyp *var* aus *.NET* und gibt an das jeder Datentyp als Typ des zurückgelieferten *Arrays* erlaubt ist.

Das *CKEditor Plugin* ist für die Integration der Variablen in den *Editor* verantwortlich, wobei die zur Verfügung stehenden Variablen über einen Dialog ausgewählt werden können. Ausgewählte Variablen werden an die aktuelle Position des Cursors im *Editor* in Form eines *HTML Tags* platziert. Die *HTML*-Repräsentation der Variable ist gekoppelt an den *FacesConverter*, da der Konverter die Variable von dessen *HTML*-Repräsentation in die *Template-Engine* spezifische Repräsentation überführen können muss. Die verwendete *Template-Engine* ist für diese *Plugin* irrelevant, da die Variablen immer in dieselbe *JSON*-Repräsentation überführt werden.

Abbildung 4.1: *CKEditor Toolbar Button* zum Öffnen des Dialogs

Der in Abbildung 4.1 rot markierte *Toolbar Button* wird über das *Plugin* registriert und in die *Toolbar* eingefügt. Über diesen *Button* kann der Dialog für die Variablenauswahl geöffnet werden.

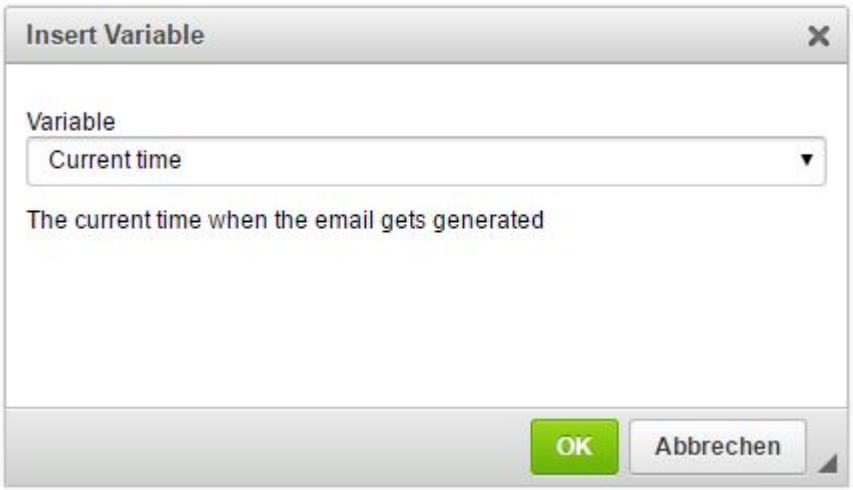


Abbildung 4.2: *CKEditor* Dialog für die Variablenauswahl

Über den Dialog aus Abbildung 4.2 stehen alle im *CKEditor Plugin* registrierten Variablen zur Auswahl. Der Titel der Variable ist der Text in der

Auswahlkomponente und die Beschreibung der ausgewählten Variable wird unterhalb der Auswahlkomponente angezeigt. Durch klick auf den *Button OK* wird die Variable in die Vorlage eingefügt und der Dialog wird geschlossen.

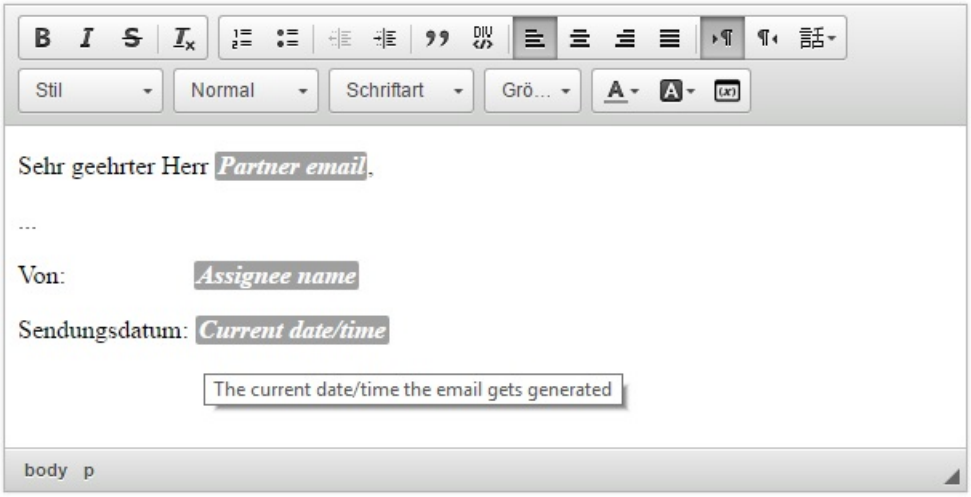


Abbildung 4.3: Beispiel einer Vorlage im *CKEditor*

Die Abbildung 4.3 illustriert eine Vorlage innerhalb des *CKEditors*, wobei die eingefügten Variablen besonders hervorgehoben werden. Der Titel der Variable stellt den Name für den *HTML Tag* bereit und die Beschreibung dessen Titel. Die eingefügten *HTML Tags* dürfen nicht verändert werden, daher ist das *Drag and Drop* und das Selektieren dieses eingefügten Texts nicht erlaubt, da dadurch die *HTML*-Repräsentation der Variablen zerstört werden könnte und die Variablen nicht mehr vom *FacesConverter* gefunden werden können.

Die Variablenrepräsentation in JSON

Die Variablen werden als Objekte der Schnittstelle *VariableContract* definiert, und müssen für den *CKEditor* in eine *JSON*-Repräsentation überführt werden, die als *Javascript*-Objekte innerhalb des *CKEDitor Plugins* verwendet werden können. Dafür wurde in *Typescript* eine Schnittstelle definiert, die die Struktur einer Variable innerhalb von *Typescript* und *Javascript* spezifiziert.

```
module cc.variables {

    export interface VariableMapping {

        id:string,

        displayName:string,

        info:string,
    }

    ...
}
```

Diese Schnittstelle ist Teil des Moduls *cc.variables* und wird mit dem Schlüsselwort *export* nach außen offengelegt und kann über den vollständigen Pfad *cc.variables.VariableMapping* innerhalb von *Typescript* und *Javascript* verwendet werden. Mit dieser Schnittstelle werden Typinformationen für *Typescript* bereitgestellt, die innerhalb von *Typescript* die Typsicherheit sicherstellen. Solange die Variablen, die registriert werden, dieser Spezifikation folgen, können innerhalb des *Typescipt* Quelltext keine Fehler auftreten.

Der Quelltext aus 4.1 zeigt die Implementierung der *JSON*-Spezifikation in *Java*, die dazu verwendet wird, die Variablen in den spezifizierten *JSON-String* zu überführen. Damit wird sichergestellt, das die Variablen in *Typescript* korrekt registriert werden. Als *JSON Provider* wird die Bibliothek *fasterxml-jackson-json* vormals *jackson-json* verwendet, die es erlaubt mit Annotationen deklarativ Attribute und/oder Methoden einer Klasse auf *JSON*-Attribute abzubilden. Durch diesen deklarativen Ansatz sind die Attribute und/oder die Methoden einer Klasse entkoppelt von der *JSON*-Spezifikation und können daher abgeändert werden. Nur ein Ändern des Datentyps eines Attributes könnte zu Problemen führen.

Programm 4.1: VariableJson.java

```
1 @JsonTypeName(value = "variable-json")
2 public class VariableJson extends AbstractJsonModel {
3
4     private String id;
5     private String label;
6     private String info;
7
8     public VariableJson() {
9     }
10
11    public VariableJson(String id,
12                        String displayName,
13                        String tooltip) {
14        this.id = id;
15        this.label = displayName;
16        this.info = tooltip;
17    }
18
19    @JsonGetter("id")
20    public String getId() {
21        return id;
22    }
23
24    @JsonSetter("id")
25    public void setId(String id) {
26        this.id = id;
27    }
28
29    @JsonGetter("displayName")
30    public String getLabel() {
31        return label;
32    }
33
34    @JsonSetter("displayName")
35    public void setLabel(String label) {
36        this.label = label;
37    }
38
39    @JsonGetter("info")
40    public String getInfo() {
41        return info;
42    }
43
44    @JsonSetter("info")
45    public void setInfo(String info) {
46        this.info = info;
47    }
48
49 }
```

4.1.2 Die Implementierungen für CDI

Folgender Abschnitt behandelt die Implementierungen für die Integration in einen *CDI Container*. Wie in Abschnitt 3.2.2 beschrieben sollen die Variablen automatisch beim start des *CDI Containers* registriert werden, sowie Vorlagenmanagementressourcen kontextabhängig über eine *CDI Producer* zur Verfügung gestellt werden können.

Die Vorlagenmanagement *CDI-Extension*

Um die Variablen beim Start des *CDI Containers* automatisch registrieren zu können, wurde die Klasse *TemplateCdiExtension* implementiert, die in der Lage ist auf Lebenszyklus *Events* des *CDI -Containers* zu reagieren und die Schnittstelle *javax.enterprise.inject.spi.Extension* implementiert. Die Schnittstelle *javax.enterprise.inject.spi.Extension* enthält keine abstrakten Methoden und markiert ein Implementierung als eine *CDI-Extension*. Die *CDI-Extension* wird als *Service Provider* über die Schnittstelle *Service-Provider-Interface* (SPI) registriert, in dem man eine folgende Datei erstellt *META-INF/services/javax.enterprise.inject.spi.Extension*, die eine normale Textdatei ist und den vollständigen Namen der *Service Provider* Implementierung enthält.

Eine *CDI-Extension* ist eigentlich kein *CDI-Bean*, da ein Objekt der *CDI-Extension* bereits beim Start des *CDI-Containers* erstellt wird und somit existiert bevor der *CDI-Container* vollständig gestartet ist. Trotzdem ist eine Extension injizierbar und kann in *CDI-Beans* injiziert werden. Das erstellte Objekt der *CDI-Extension* existiert über die Lebensdauer des *CDI-Containers*.

Das folgende Programm 4.2 ist ein Auszug aus der implementierten *CDI-Extension* und illustriert die *CDI-Container* Lebenszyklus *Events*, auf die reagiert wird. Über diese *CDI-Extension* werden alle implementierten Typen der Schnittstelle *VariableContract* gefunden und im Objekt der Klasse *TemplateConfiguration* registriert. Vorerst werden nur Implementierungen der Schnittstelle *VariableContract* unterstützt, die mit dem *Java*-Typ *enum* implementiert wurden. Ebenfalls werden alle implementierten Typen der Schnittstelle *VariableResolverFactory* gefunden und in der *Extension* registriert. Die *Extension* wird in der später vorgestellten *CDI-Producer* Implementierung verwendet um Objekte der Schnittstelle *VariableResolverFactory* zu produzieren.

Programm 4.2: TemplateCdiExtension.java

```
1 public class TemplateCdiExtension implements Extension,
2         Serializable {
3
4     private TemplateConfiguration templateConfig;
5     private Map<Class<? extends VariableContract>,
6                 Class<VariableResolverFactory>>
7             variableResolverFactoryMap;
8     private Logger log;
9
10    void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
11        // Init class members
12    }
13
14    <T> void processCdiVariableContracts
15             (@Observes @WithAnnotations({BaseName.class,
16                                          CdiVariableContract.class})
17             ProcessAnnotatedType<T> pat) {
18        // Collect VariableContract implementations (Enum type only)
19    }
20
21    <T> void processVariableResolverFactoryFactories
22        (@Observes @WithAnnotations(CdiVariableResolverFactory.class)
23        ProcessAnnotatedType<T> pat) {
24        // Collect VariableResolverFactory implementations
25    }
26
27    // Getter for class members templateConfig
28    // and variableResolverFactoryMap
29 }
```

- *void beforeBeanDiscovery(...)*
  ist die *Observer*-Methode, die einmalig aufgerufen wird bevor mit dem Auffinden der *CDI-Beans* begonnen wird. In dieser Methode wird die Extension initialisiert.
- *<T> void processCdiVariableContracts(...)*
  ist die *Observer*-Methode, die für jeden annotierten Typ aufgerufen wird, der mit den Annotationen *@BaseName* und *@CdiVariableContract* annotiert ist.
- *<T> void processVariableResolverFactoryFactories(...)*
  ist die *Observer*-Methode, die für jeden annotierten Typ aufgerufen wird, der mit den Annotationen *@CdiVariableResolverFactory* annotiert ist.

Der Vorlagenmanagement *CDI-Producer*

Es wurde ein *CDI-Producer TemplateResourceProducer* implementiert, mit dem kontextabhängig Ressourcen des Vorlagenmanagements produziert werden können. Diese Klasse ist die einzige Klasse, die sich die *Extension TemplateCdiExtension* injizieren lässt. Es kann nicht verhindert werden, dass andere *CDI-Beans* sich diese Klasse injizieren, da eine *CDI-Extension*-Klasse öffentlich sein muss. Da mehrere *Template-Engines* unterstützt werden sollen, wurde die Annotation *@FreemarkerTemplate* eingeführt, die einen Injektionspunkt für *Freemarker* qualifiziert. Ebenso wurde jeweils eine *Producer*-Methode für den Qualifizierer *@Default* implementiert, über die Standardimplementierung bereitgestellt wird, die in der derzeitigen Implementierung Implementierungen die den Qualifizierer *@FreemarkerTemplate* produziert. Damit ist sichergestellt, dass keine *UnsatisfiedResolutionException* auftreten kann, obwohl man sich der Gefahr aussetzt, dass die produzierte *@Default* Implementierung nicht die gewollte ist.

Programm 4.3: TemplateResourceProducer.java

```
1 @ApplicationScoped
2 public class TemplateResourceProducer implements Serializable {
3     @Produces
4     @ApplicationScoped
5     @Default
6     public VariableConfiguration produceConfiguration() {
7         return extension.getVariableConfiguration();
8     }
9
10    @Produces
11    @Dependent
12    @Default
13    public TemplateDataJsonBuilder produceDefaultTemplateBuilder
14          (final @Default VariableResolverFactoryProvider factory) {
15        return produceFreeMarkerTemplateBuilder(factory);
16    }
17
18    @Produces
19    @Dependent
20    @FreemarkerTemplate
21    public TemplateDataJsonBuilder produceFreeMarkerTemplateBuilder
22           (final @Default VariableResolverFactoryProvider factory) {
23        return new FreemarkerTemplateDataJsonBuilder()
24                .withWeakMode()
25                .withVariableResolverFactoryProvider(factory);
26    }
27 }
```

Der Quelltext aus 4.3 ist ein Auszug aus der Klasse *TemplateResourcePro-*

ducer und dient als Beispiel für die implementierten *Producer*-Methoden. Die beiden implementierten *Producer*-Methoden *produceDefaultTemplateBuilder* und *produceFreeMarkerTemplateBuilder* produzieren Objekte der Schnittstelle *TemplateDataJsonBuilder* für verschiedene Qualifizierer. Die Methode *produceDefaultTemplateBuilder* produziert Objekte der Schnittstelle *TemplateDataJsonBuilder* qualifiziert mit *@Default*, wobei ein Injektionspunkt diesen Qualifizierer nicht explizit angeben muss, da dieser als Standard verwendet wird. Die Methode *produceFreeMarkerTemplateBuilder* produziert Objekte der Schnittstelle *TemplateDataJsonBuilder* qualifiziert mit *@FreemarkerTemplate*, wobei ein Injektionspunkt diesen Qualifizierer explizit angeben muss. Beide Methoden produzieren Objekte für den sogenannten Pseudo-*Scope @Dependent*, wobei für jeden Injektionspunkt ein neues Objekt erstellt wird. Als Argument für diesen beiden Methoden wird ein Objekt der Schnittstelle *VariableResolverFactoryProvider* injiziert, der *@Default* qualifiziert ist. Dieses Objekt wird kontextabhängig injiziert, wobei der Geltungsbereich dieses Objekts für die Methoden nicht bekannt ist.

Die Methode *produceConfiguration* produziert ein Objekt der Schnittstelle *VariableConfiguration*, die die registrierten Variablen enthält und von der *Extension* bereitgestellt wird. Nachdem diese Schnittstelle nur lesenden Zugriff erlaubt und nur die *Extension* Variablen registriert, wird dieses Objekt für den Gültigkeitsbereich der Anwendung produziert, als einmalig für die gesamte Anwendungslaufzeit.

Die Vorlagenmanagement *CDI-Utility*

Die Klasse *CdiTemplateUtil* wurde implementiert um ein kontextabhängiges *CDI-Bean* zur Verfügung zu stellen, das Hilfsmethoden für die Konvertierung und Lokalisierung der Variablen bereitstellt. Für die Lokalisierung wurde die Schnittstelle *VariableLocaleProvider* spezifiziert, dessen Implementierung kontextabhängig als *CDI-Bean* zur Verfügung gestellt wird, um ein Objekt der Klasse *java.util.Locale* zu bekommen, dass für die Lokalisierung benötigt wird. Eine Implementierung der Schnittstelle *VariableLocaleProvider* kann dabei das Objekt der Klasse *java.util.Locale* abhängig von einem eingeloggten BenutzerIn bereitstellen, in einem beliebigen Gültigkeitsbereich.

Es wird die Implementierung *DefaultVariableLocaleProvider* der Schnittstelle *VariableLocaleProvider* bereitgestellt, die mit *@ApplicationScoped* qualifiziert ist und die das Objekt der Klasse *java.util.Locale* über *Locale.getDefault();* bereitstellt. Diese Implementierung stellt die Standardimplementierung dar. Um eine eigene Implementierung zur Verfügung zu stellen, stehen folgende Möglichkeiten zur Verfügung.

- *Spezialisierung* ist eine Möglichkeit ein *Bean* zu spezialisieren, wobei

die eigene Implementierung von der spezialisierten Implementierung ableiten und mit *@Specializes* annotiert werden muss. Oder es gibt eine *Producer*-Methode, die mit *@Specializes* annotiert ist. Mit diesem Ansatz werden alle gesetzten Qualifizierer, der Name und der Gültigkeitsbereich geerbt. Die spezialisierte Implementierung wird mit der Spezialisierung deaktiviert und steht nicht mehr zur Verfügung.

- *Alternative Implementierung* ist die Möglichkeit für den Typ, in diesem Fall *VariableLocaleProvider*, eine alternative Implementierung zur Verfügung zu stellen, wobei die alternative Implementierung in der *beans.xml* registriert und/oder mit *@Alternative* annotiert werden muss (Abhängig von der konkreten *CDI*-Implementierung). Mit diesem Ansatz kann ein unterschiedlicher Gültigkeitsbereich, Name und Qualifizierer verwendet werden. Mit einer alternativen Implementierung kann die andere Implementierung noch immer zur Verfügung stehen, abhängig ob die Qualifizierer der beiden Implementierungen gleich sind.

4.1.3 Die Implementierungen für JSF

Folgender Abschnitt behandelt die Implementierung des Variablenmanagements für die *View*-Technologie *JSF*. In diesem Abschnitt wird sich nur dem implementierten *FacesConverter* und die *CKEditor*-Integration, bereitgestellt von *primefaces-extensions*, beschäftigen.

Der Vorlagen *FacesConverter*

Der *FacesConverter* wurde als abstrakte Klasse *AbstractTemplateConverter* implementiert, die die Schnittstelle *javax.faces.Converter* implementiert. Diese abstrakte Klasse wurde implementiert, da die Konvertierungslogik über alle *Template-Engines* dieselbe ist und lediglich sich die Implementierung der Schnittstelle *TemplateProcessor* unterschiedet. Das Objekt der Schnittstelle *TemplateProcessor* und das Objekt der Klasse *CdiTemplateUtil* werden vom *CDI-Container* bereitgestellt, wobei diese Objekte manuell geholt werden müssen, da keine Injektion innerhalb eines *FacesCovnerters* möglich ist. Die Objekte werden über die Klasse *BeanProvider* der Bibliothek *Deltaspike* geholt. *Deltaspike* ist eine Bibliothek, die portable *CDI-Exntensions* bereitstellt. Die konkrete Implementierung *FreemarkerTemplateConverter*, die von der abstrakten Klasse *AbstractTemplateConverter* ableitet, setzt über einen Konstruktor in der Basisklasse den korrespondierenden Qualifizierer für die verwendete *Template-Engine*, um mit diesem Qualifizierer die korrekte Implementierung der Schnittstelle *TemplateProcessor* aus dem *CDI-Container* zu holden.

Programm 4.4: FreemarkerTemplateConverter.java

```
1 @FacesConverter("template.converter.freemarker")
2 public class FreemarkerTemplateConverter extends
      AbstractTemplateConverter {
3
4     public FreemarkerTemplateConverter() {
5         super(new FreemarkerTemplateLiteral());
6     }
7 }
```

Der Quelltext in 4.1.3 ist die implementierte Klasse für die *Template-Engine Freemarker*. Die Annotation *@FacesConverter("template.converter.freemarker")* definiert einen eindeutigen Namen für diesen Konverter, der in *xhtml* Dateien als Konverter definiert werden kann. Das *JSF-Framework* erstellt dann ein Objekt dieser Klasse für die *JSF*-Komponente, die diesen Konverter verwendet.

Die abstrakte Klasse *AbstractTemplateConverter* definiert reguläre Ausdrücke, um die Variablen einer Vorlage in Form von *HTML-Tags* zu finden und zu konvertieren.

```
1 String tagRegex = "(<span[^,>]*class=\"variable\"[^,>]*>[^,<]*</span>)";
2 String idRegex = "data-variable-id=\"(\\S+)\"";
```

- *tagRegex* ist der reguläre Ausdruck, um die Variablen in ihrer *HTML*-Repräsentation in einer Vorlage zu finden.
- *idRegex* ist der reguläre Ausdruck, um die *Id* einer Variable, aus deren *HTML*-Repräsentation zu bekommen und wird auf den gefundenen *HTML-Tag* einer Variable angewendet.

Die abstrakte Klasse *AbstractTemplateConverter* definiert auch eine *String*-Vorlage, um die Variablen in ihre *HTML*-Repräsentation zu konvertieren, wobei diese Vorlage unabhängig von der verwendeten *Template-Engine* ist und auf alle Variablen gleich angewendet wird.

```
1 String template = "<span class=\"variable\" contentEditable=\"false\" "
2                 + "data-variable-id=\"{0}\" title=\"{1}\">{2}</span>";
```

Die Vorlage *template* wird mit *java.text.MessageFormat(String, Object...)* verarbeitet.

Die *Primefaces-Extension* für den *CKEditor*

Der *Rich-Editor CKEditor* ist eine *Javascript* basierte Anwendung, die nur am *Browser* der BenutzerInnen läuft. Es wird aber eine *JSF*-Integration

benötigt, damit man

- auf *AJAX-Events* reagieren kann,
- *FacesConverter* verwenden kann und
- Parameterbindungen definieren kann.

Da es nicht trivial ist eine vollwertige *JSF*-Komponente zu implementieren, und das Implementieren einer solchen Komponente auch viel Zeit in Anspruch nimmt, wurde auf die Implementierung von *primefaces-extensions* zurückgegriffen, die bereits eine vollwertige *JSF*-Integration bereitstellt. *Primefaces-extensions* ist eine quelloffene Bibliothek, die die quelloffene Bibliothek *Primefaces* erweitert. *Primefaces* ist zurzeit eine der bekanntesten *JSF*-Komponenten Bibliothek im *Java*-Umfeld.

Die Ressourcen für den *CKEDitor* bewegen sich in der Größenordnung von 1,5 Megabyte, daher werden die Ressourcen in einem separaten Artefakt zur Verfügung gestellt. Man kann auch eine eigene Implementierung zur verfügung stellen, sofern diese Implementierung in derselben Version ist. Der *CKEditor* ist ein sehr umfangreicher *Rich-Editor*, den man sich auch seinen Wünschen gemäß selbst zusammenstellen kann. Diese Zusammenstellung des *CKEditors* kann man heranziehen, um die Standardimplementierung zu ersetzen.

Folgender Quelltest illustriert die Verwendung des *CKEditors* in Form der zur Verfügung gestellten *JSF*-Komponente.

```
1 <pe:ckEditor id="template_content_editor"
2              wdgetVar="pfEditor"
3              value="#{templateEditModel.content}"
4              converter="template.converter.freemarker"
5              contentsCss="resources/css/myStyle.css"
6              customConfig="./ckeditor-config.js">
7 </pe:ckEditor>
```

- *id* ist das Attribute, um die eindeutige *Id* innerhalb des Namensraums der Komponente zu definieren.
- *widgetVar* ist das Attribut, um einen eindeutigen Name des *Javascript*-Objekts, das den Zugriff auf den *CKEditor* in *Javascript* erlaubt, zu definieren.
- *value* ist das Attribut, um die Parameterbindung der Vorlage zu einem *Java*-Objekt zu definieren.
- *converter* ist das Attribut, um den verwendeten Konverter, der die Vorlagen konvertiert, zu setzen.
- *contentCss* ist das Attribut, um eine eigene *CSS*-Datei für den Inhalt der Vorlage zu definieren. Die Vorlage wird innerhalb des *Editors* als eigenständige *HMTL*-Datei behandelt, das in einer *Iframe*-Komponente

gehalten wird.
- *customConfig* ist das Attribut, um die eigene Konfiguration des *Editors* in Form von einer eigenen *Javascript*-Datei zu definieren.

4.2 Die Vorlagenmanagement Beispielanwendung

Der folgende Abschnitt beschäftigt sich mit der implementierten Beispielanwendung, für das Vorlagenmanagement, die die Verwendung des Vorlagenmanagement im Bezug auf

- die *Business-Logik*,
- die Webseite und
- die Persistenz einer *E-Mail*

illustrieren soll. Dazu wurde eine Demowebanwendung implementiert, die die webseitige Verwaltung der Vorlagen implementiert. Es wurde auch *Business-Service* implementiert, der illustrieren soll, wie eine *E-Mail* basierend auf einer Vorlage, aus einem *Business-Service* heraus erstellt werden kann.

4.2.1 Die Verwendung in einem *Business*-Service

Folgender Quelltext 4.5 illustriert die indirekte Verwendung einer Vorlage über die implementierte Klasse *EmailServiceImpl* der Schnittstelle *Email-Service*, die im Projekt *mailing-template-integration-clevercure-web* liegt und für die Verwendung für der Anwendung *CleverWeb* implementiert wurde. Diese Implementierung der Schnittstelle *EmailService* benutzt *CDI-Events* um die *E-Mails* anzulegen. Die Persistenz wird innerhalb der *Mail-Service*-Projekte behandelt. Es werden über diese Implementierung mehrere Möglichkeiten angeboten, wie *E-Mails* aus einem *Business-Service* heraus erstellt werden können.

- *public void create(EmailDTO dto)*
  ist die Methode, mit der eine *E-Mail* sofort erstellt werden können.
- *public void create(List<EmailDTO> dtos)*
  ist die Methode, mit der mehrere *E-Mails* sofort erstellt werden können.
- *public void createAfterSuccess(EmailDTO dto)*
  ist die Methode, mit der eine *E-Mail* nach dem erfolgreichem Beenden einer Transaktion erstellt werden kann.
- *public void createAfterSuccess(List<EmailDTO> dto)*
  ist die Methode, mit der mehrere *E-Mails* nach dem erfolgreichem Beenden einer Transaktion erstellt. werden kann.

Programm 4.5: EmailServiceCdiEventImpl.java

```
1 @RequestScoped
2 @Transactional(Transactional.TxType.SUPPORTS)
3 public class EmailServiceCdiEventImpl implements EmailService {
4     @Inject
5     private Event<CreateEmailsEvent<CreateEmailsEvent.CreateImmediate>>
      createImmediateEvent;
6     @Inject
7     private Event<CreateEmailsEvent<CreateEmailsEvent.CreateAfterSuccess
      >> createAfterSuccessEvent;
8     @Inject
9     private Event<CreateEmailsEvent<CreateEmailsEvent.CreateAfter>>
      createAfterEvent;
10
11    @Override
12    @Transactional(Transactional.TxType.REQUIRED)
13    public void create(EmailDTO dto) {
14        createImmediateEvent.fire(new CreateEmailsEvent<>(dto));
15    }
16
17    @Override
18    @Transactional(Transactional.TxType.REQUIRED)
19    public void create(List<EmailDTO> dtos) {
20        createImmediateEvent.fire(new CreateEmailsEvent<>(dtos));
21    }
22
23    @Override
24    public void createAfterSuccess(EmailDTO dto) {
25        createAfterSuccessEvent.fire(new CreateEmailsEvent<>(dto));
26    }
27
28    @Override
29    public void createAfterSuccess(List<EmailDTO> dtos) {
30        createAfterSuccessEvent.fire(new CreateEmailsEvent<>(dtos));
31    }
32 }
```

Programm 4.6: BusinessServiceImpl.java

```
1 @RequestScoped
2 @Transactional(Transactional.TxType.REQUIRED)
3 public class BusinessServiceImpl implements BusinesService {
4
5     @Inject
6     private EmailService emailService;
7
8     @Override
9     public void doBusinessEmailImmediate() {
10        emailService.create(createEmailDto());
11    }
12
13    @Override
14    public void doBusinessEmailAfterSuccess() {
15        emailService.createAfterSuccess(createEmailDto());
16    }
17
18    private EmailDTO createEmailDto() {
19        final String email = "herzog.thomas8@gmail.com";
20        final Long mailUserId = 1L;
21        final List<Long> mailTypeIds = Collections.singletonList(1L);
22        final Locale locale = Locale.US;
23        final ZoneId zone = ZoneId.systemDefault();
24        final Map<Object, Object> userData =
25          new HashMap<Object, Object>() {{
26                put(TemplateVariable.SENDER_USER, "Thomas Herzog");
27                put(TemplateVariable.RECIPIENT_USER, "Hugo Maier");
28                put(TemplateVariable.TOPIC, "User status changed");
29                put(TemplateVariable.STATUS, "Inactive");
30            }};
31        return new EmailDTO(email,
32                  locale,
33                  zone,
34                  mailUserId,
35                  userData,
36                  mailTypeIds);
37    }
38 }
```

Der Quelltext aus 4.6 illustriert einen *Business-Service*, der über die Schnittstelle *EmailService E-Mails* erstellt. Die zu erstellende *E-Mail* wird durch ein Objekt der Klasse *EmailDTO* repräsentiert, das alle benötigten Informationen für das Erstellen einer *E-Mail* enthält.

- *email* ist die Zeichenkette, die die *E-Mail*-Adresse definiert.
- *mailUserId* ist die *Id* des virtuellen Benutzers, der die *E-Mail* auf der Datenbank erstellt.

- *mailTypeIds* ist die Menge von *Ids*, die die *Mail*-Typen repräsentieren. Jedem *Mail*-Typ ist eine Voralge zugeordnet.
- *locale* ist das Objekt der Klasse *java.util.Locale* , das die Sprache definiert.
- *zone* ist das Objekt der Klasse *java.time.ZoneId*, das die Zone für die Datumsformatierung definiert.
- *userData* ist der assoziative Behälter, der die Benutzerdaten enthält, die bei der Evaluierung verwendet werden.

4.2.2 Die Verwendung in der *Web*-Oberfläche

Die Abbildung 4.4 zeigt die Weboberfläche, die für die Beispielanwendung implementiert wurde. Über dieses Formular können die Voralgen sprachspezifisch verwaltet werden.

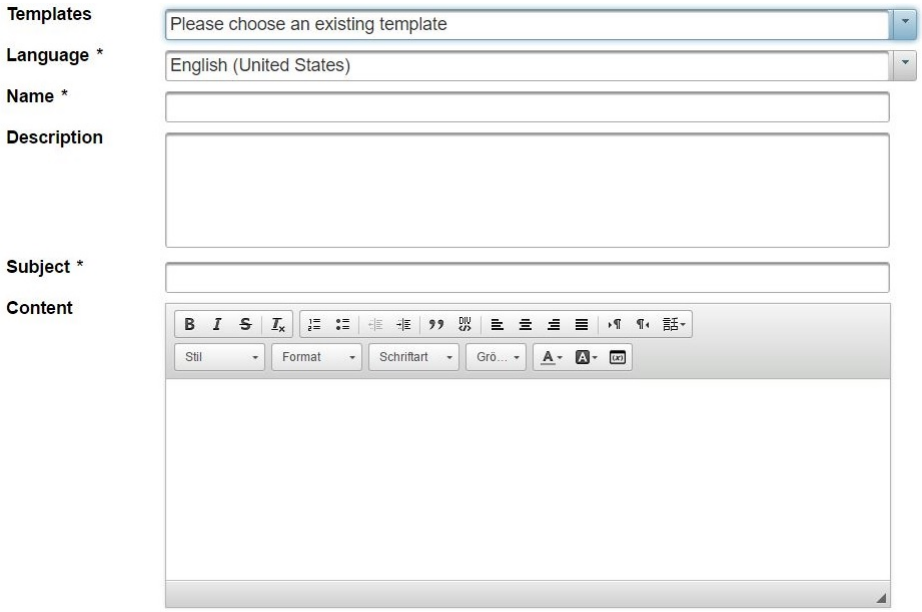


Abbildung 4.4: Formular für die Verwaltung der Vorlagen

Die Abbildung 4.5 zeigt, den Teil der Webseite, der die relevanten Daten einer Vorlage anzeigt.

- *Decorator Template* ist die *Freemarker*-Voralge, die von jeder Vorlage dekoriert wird.
- *User Template* ist die Vorlage, die von einem BenutzerIn erstellt wurde.
- *Template JSON Data* ist die *JSON*-Zeichenkette, die erstellt wird, wenn die Daten für eine Vorlage serialisiert werden.

- *Parsed Template* ist die Vorlage, in der die Variablen durch die serialisierten Werte ersetzt wurden.
- *Template Metadata* sind die Metadaten der Vorlage, wie z.B. die Anzahl der enthalten Variablen.

Abbildung 4.5: Anzeige der aller relevanten Daten einer Vorlage

Die folgende Abbildung 4 zeigt eine Vorlage im Formular. Sie verwendet, die zur Verfügung gestellten Variablen und ist zwei Sprachen verfügbar. Eine Vorlage muss immer eine Standardsprache definieren, für die es immer einen Eintrag geben muss.
TODO: Add image with template

Die Dekoriervorlage

Die dekorierbare Vorlage aus Abbildung 4.6 ist die *Freemarker*-Vorlage, die alle Vorlagen dekorieren. Sie stellt den *HTML-Body* zur Verfügung.

Abbildung 4.6: Das *Decorator Template*

Die Benutzervorlage

Die Benutzervorlage aus Abbildung 4.7 ist die *Freemarker*-Vorlage, die von der BenutzerIn erstellt wurde. Die Vorlage enthält zwar *HTML*, aber nur den Inhalt der Vorlage in Form von *HTML*, stellt aber kein vollständiges *HTML*-Dokument dar.

User Template

```
<p>Sehr geehrte(r) Frau/Herr ${(cc.module.di["RECIPIENT_USER"])!("variable: 'RECIPIENT_USER' not
found")},</p>

<p>Hiermit informaierne wir Sie &uuml;ber die Status&auml;nderung bez&uuml;glich
&#39;${(cc.module.di["TOPIC"])!("variable: 'TOPIC' not found")}&#39;.</p>

<p>Sttus:    ${(cc.module.di["STATUS"])!("variable: 'STATUS' not found")}<br />
Sender: ${(cc.module.di["SENDER_USER"])!("variable: 'SENDER_USER' not found")}</p>
```

Abbildung 4.7: Die *Benutzervorlage* als *Freemarker Template*

Die Vorlagenmetadaten

Die Vorlagenmetadaten aus Abbildung 4.8 zeigt die Metadaten der Benutzervorlage. Die Metadaten sind nur für die Entwicklung relevant.

Template Metadata

```
================================================================
FreemarkerTemplateMetadata
================================================================
id                  : 3
version             : 1
length              : 457
variables (valid)   : 4
contract  : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id        : cc.module.di.SENDER_USER
name      : SENDER_USER
label-key : SENDER_USER
info-key  : SENDER_USER


contract  : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id        : cc.module.di.TOPIC
name      : TOPIC
label-key : TOPIC
info-key  : TOPIC


contract  : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id        : cc.module.di.RECIPIENT_USER
name      : RECIPIENT_USER
label-key : RECIPIENT_USER
info-key  : RECIPIENT_USER


contract  : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id        : cc.module.di.STATUS
name      : STATUS
label-key : STATUS
info-key  : STATUS

variables (invalid) : 0

```

Abbildung 4.8: Die Metadaten der Vorlage

Kapitel 5

Die Analyse und Tests

5.1 Die Tests

5.1.1 Die Tests der *Services*

5.1.2 Die Tests der *CDI*-Integration

5.1.3 Die Tests der *Web*-Oberfläche

5.2 Die erreichten Ziele

5.2.1 Das Vorlagen-*Management über* CKEditor

5.2.2 Das Vorlagen-*Management* in einer *CDI*-Umgebung

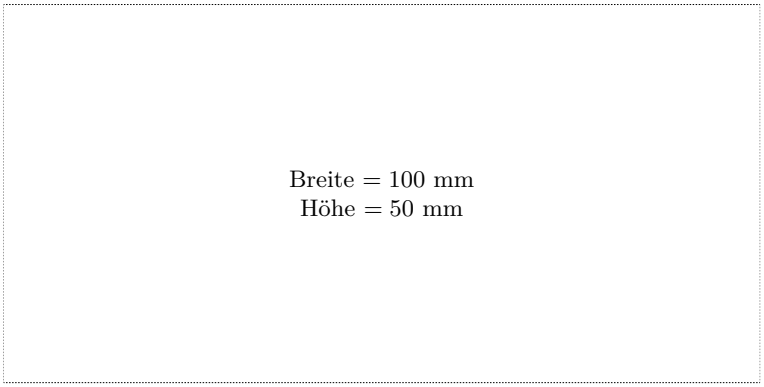
5.2.3 Das Vorlagen-*Management* in JSF

5.2.4 Das Vorlagen-*Management* in *Mail*-DB-Schema

Quellenverzeichnis

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —