Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Vorlagenmanagement für *Mail-Service*

Praktische
Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Ing. Thomas Herzog

Begutachtet von Titel FH-Prof. DI Dr. Heinz Dobler

Hagenberg, August 2016

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.

Datum Unterschrift

Inhaltsverzeichnis

Kurzfassung

TODO: Add german summary here

Abstract

TODO: Add english summary here

Kapitel 1

Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Konzeption und Implementierung eines Vorlagenmanagements für den, in der theoretischen Bachelorarbeit konzipierten, *Mail-Service* namens *CleverMail*. Das Vorlagenmanagement stellt einen essentiellen Teil von *CleverMail* 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 Anwendungen, die *CleverMail* 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 CleverMail* wurde entschieden, sich vorerst auf das Vorlagenmanagement zu konzentrieren. Das Vorlagenmanagement wird für *CleverMail* 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* haben.

1.1 Das Unternehmen curecomp Software Services GmbH

Das Vorlagenmanagement wird in Zusammenarbeit mit dem Unternehmen *curecomp Software Services GembH* erstellt. Das Unternehmen *curecomp* ist ein Dienstleister im Bereich des *Supplier-Relationship-Managements (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 und Lieferanten.
- *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-Mails* innerhalb von *clevercure*, die durch *CleverMail* abgelöst werden soll.

Das Vorlagenmanagement wird von den Anwendungen innerhalb von *clevercure* verwendet werden, bevor *CleverMail* fertiggestellt wird, da es bereits Softwarekomponenten innerhalb der Anwendungen von *clevercure* gibt, die auf parametrierbare Vorlagen angewiesen sind.

1.2 Das Vorlagenmanagement für *CleverMail*

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

- das neu Kompilieren und Einspielen bei Änderungen der Vorlagen,
- keine Möglichkeit für benutzerdefinierten Vorlagen oder
- keine Möglichkeit der Nutzung von dynamischen Parametern in den Vorlagen eliminiert werden.

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

1.3 Die Rahmenbedingungen

Das Vorlagenmanagement muss in Java in der Version 8 implementiert werden und muss die Plattform *Java-Enterprise-Edition 7 (JEE-7)* verwenden, wobei folgende Spezifikationen Anwendung finden müssen.

- *Java-Persistence-API 2.1 (JPA) (JSR 338)*
  ist die Spezifikation für die Persistenz in Java.
- *Context and Dependency Injection 1.1 (CDI) (JSR 346)*
  ist die Spezifikation für kontextabhängige Injektion innerhalb der Plattform *JEE-7*.
- *Java-Server-Faces 2.2 (JSF) (JSR 344)*
  ist die Spezifikation der *View*-Technologie.

Damit wird das Vorlagenmanagement mit den aktuellsten Standards und Spezifikationen implementiert. Die Funktionalität des Vorlagenmanagement muss weitestgehend ohne die Verwendung externer Bibliotheken implementiert werden. Das Vorlagenmanagement muss folgende Integrationen zur Verfügung stellen.

- Die Integration in *CDI*,
- die Integration in *JSF* und
- die Integration in *Typescript*.

Als Entwicklungsumgebung wird *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 soll so weit wie möglich vermieden werden 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

Das Ziel des Projekts Vorlagenmanagement für *Mail-Service* ist die Entwicklung der Softwarekomponente Vorlagenmanagement für die Verwendung in *CleverMail*, mit dem Vorlagen verwaltet werden können. Das Vorlagenmanagement stellt einen essentiellen Teil von *CleverMail* dar und wird auch von mehreren Anwendungen innerhalb von *clevercure* verwendet werden. Die verschiedenen Anwendungen, die das Vorlagenmanagement verwenden, sind ebenfalls in Java implementiert, werden aber in unterschiedlichen Laufzeitumgebungen betrieben wie z.B:

- Der *IBM-Integration-Bus (IIB)*
  ist ein proprietäres Produkt des Unternehmens *IBM*, das für die *XML*-Konvertierungen und den *XML*-basierten Datenimport und Datenexport verwendet wird.
- Der Anwendungsserver *Wildfly*
  ist ein zertifizierter und frei verfügbarer *JEE-7* Applikationsserver des Unternehmens *Redhat*.

Die verschiedenen Anwendungen von *clevercure* müssen mit möglichst wenig Aufwand in der Lage sein, Vorlagen zu verwenden und *E-Mail*-Nachrichten auf Basis dieser Vorlagen zu erstellen. Dabei müssen die Abhängigkeiten der Anwendungen zum Vorlagenmanagement so gering wie möglich gehalten werden, sowie nur vorgegebene Schnittstellen verwendet werden dürfen. Wird eine *E-Mail*-Nachricht von einer Anwendung auf Basis einer Vorlage erstellt, so müssen die aktuellen Werte der enthaltenen Variablen der Vorlage beim Zeitpunkt des Erstellens der *E-Mail*-Nachricht ermittelt und serialisiert werden, damit die *E-Mail*-Nachricht mit dem selben Inhalt erneut versendet werden kann. Für die Anwendungen darf 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 das Datenbankmodell der Anwendung *CCMail*.

2.1 Die funktionalen Ziele

Für das Vorlagenmanagement wurden die folgende funktionalen Anforderungen definiert, die umgesetzt werden müssen.

2.1.1 Die Variablen für die Vorlagen

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

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

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

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

Die EntwicklerInnen müssen 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 müssen auch global definiert werden und prinzipiell in allen Vorlagen verwendbar 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.

2.1.2 Die Mehrsprachigkeit der Variablen

Die zur Verfügung stehenden Variablen werden durch die EntwicklerInnen statisch definiert und müssen einen Bezeichnung und eine Beschreibung zur Verfügung stellen. Die Bezeichnung und die Beschreibung der Variable müssen mehrsprachig zur Verfügung stehen, wobei als Standardsprache Englisch zu verwenden ist. Die Mehrsprachigkeit soll über *Java-Properties*-Dateien abgebildet werden, wobei als Zeichenkodierung *UTF8* zu verwenden ist, obwohl *Java- Properties*-Dateien laut Spezifikation die Zeichenkodierung *ISO 8859-1* verwenden müssen.

2.1.3 Die automatische Registrierung der Variablen

Innerhalb einer *CDI*-Umgebung sollen die definierten Variablen beim Start der *CDI*-Umgebung automatisch gefunden und registriert werden. Die au-

tomatische Registrierung der Variablen muss mit einer *CDI*-Erweiterung realisiert werden, die beim Start der *CDI*-Umgebung die Variablen findet, registriert und über die Anwendungslebensdauer persistent hält. Mit einer automatischen Registrierung der Variablen wird erreicht, dass neu definierte Variablen automatisch gefunden und registriert werden und somit nicht manuell registriert werden müssen. Ein manuelles Registrieren der Variablen birgt das Risiko in sich, dass Variablen vergessen werden könnten registriert zu werden.

2.1.4 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 für jede angeforderte Sprache immer eine Vorlage zur Verfügung steht. Es ist nicht erforderlich dass die Menge und Position der Variablen in einer Vorlage über alle definierte Sprachen gleich sind. Es dürfen in einer Vorlage, die in mehreren Sprachen definiert wurde, eine unterschiedliche Anzahl von Variablen, unterschiedliche Variablen und unterschiedliche Positionen der Variablen definiert sein.

2.1.5 Die Persistenz der Vorlagen

Die Vorlagen müssen innerhalb einer Datenbank persistent gehalten werden. Da das Vorlagenmanagement vorerst exklusiv für *CleverMail* verwendet wird, muss die Persistenz der Vorlagen innerhalb des *Mail*-DB-Schema von *CleverMail* realisiert werden. Die persistenten Vorlagen müssen versionierbar sein, damit diese von anderen Entitäten referenziert werden können, ohne dass die Gefahr besteht, dass die referenzierte Vorlage verändert wurde. Die Versionierung soll die Konsistenz der Vorlagen sicherstellen, sodass serialisierte Daten für eine Vorlage konsistent mit den enthaltenen Variablen der Vorlage sind. Vorlagen müssen explizit freigegeben werden, bevor diese verwendet dürfen. Nach einer Freigabe darf die Vorlage nicht mehr geändert werden.

2.1.6 Die Verwaltung der Vorlagen über eine Webseite

Die Vorlagen müssen über eine Webseite verwaltet werden können. Die Webseite muss mit der *View*-Technologie *JSF* implementiert werden. Über einen *FacesConverter* soll die Vorlage von ihrer *HTML*-Repräsentation in die Repräsentation der verwendeten *Template-Engine* konvertiert werden und visa versa. Der Quelltext aus Abbildung 2.1 zeigt ein *HTML-Markup* einer Vorlage, wie es in der Webseite bzw. innerhalb des *Editors CKEditor* verwendet

wird. Die Variablen werden als *HTML-Tags* repräsentiert, aus denen die Variable wieder hergestellt werden kann.

Programm 2.1: Beispiel eines *HTML-Markup* einer Vorlage

```
<p>Das ist eine Variable:</p>
<p>
  <span class="variable"
        title="Die Beschreibung der Variable"
        data-variable="VAR_ID">
      Die Bezeichnung der Variable
  </span>
</p>
```

Der Quelltext aus Abbildung 2.2 zeigt das konvertierte *HTML-Markup* aus Abbildung 2.1 als *Freemarker*-Vorlage .

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

```
<p>Das ist eine Variable:</p>
<p>
  ${module.core.VariableHolder["VAR_ID"]!("Nicht verfügbar")}
</p>
```

Auf der Webseite muss der *JavaScript* basierte *Editor CKEditor* verwendet werden, weil für diesen *Editor* von *PrimeFaces-Extensions* eine *JSF*-Integartion, in Form einer *JSF*-Komponente, zur Verfügung gestellt wird. Es muss der *Editor CKEditor* verwendet werden, weil eine Integration in den Lebenszyklus von *JSF* notwendig ist, damit z.B. auf *AJAX-Reqeust* reagiert werden kann, wie es in *JSF* üblich ist.

2.2 Die technischen Ziele

Es wurden die folgenden technischen Zile definiert.

- Die Entwicklung in *Java 8*,
- die Entwicklung mit der Plattform *JEE-7*,
- die Integration in eine *CDI*-Umgebung,
- die Integration in *JSF* und
- die Entwicklung als eigene Softwarekomponente.

Das Vorlagenmanagement muss Schnittstellen definieren, die die Funktionalitäten des Vorlagenmanagements nach außen offenlegen, ohne dass die

Anwendungen in Berüghrung mit den konkreten Implementierungen kommen.

Kapitel 3

Das Lösungskonzept

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

- für die *Template-Engine Freemakrer*,
- für die *Template-Engine Velocity* oder
- für die *Template-Engine Thymeleaf*.

Mit der Möglichkeit verschiedene *Template-Engines* verwenden zu können, soll das Vorlagenmanagement flexibel gehalten werden. Bei einem Wechsel zu einer anderen *Template-Engine*, müssen nur die Ausdrücke in einer Vorlagen in die *Template-Engine* spezifischen Ausdrücke konvertiert werden, was sich einfach realisieren lässt, da die Ausdrücke einer Vorlage immer gefunden werden müssen.

3.1 Die Spezifikation des Vorlagenmanagements

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

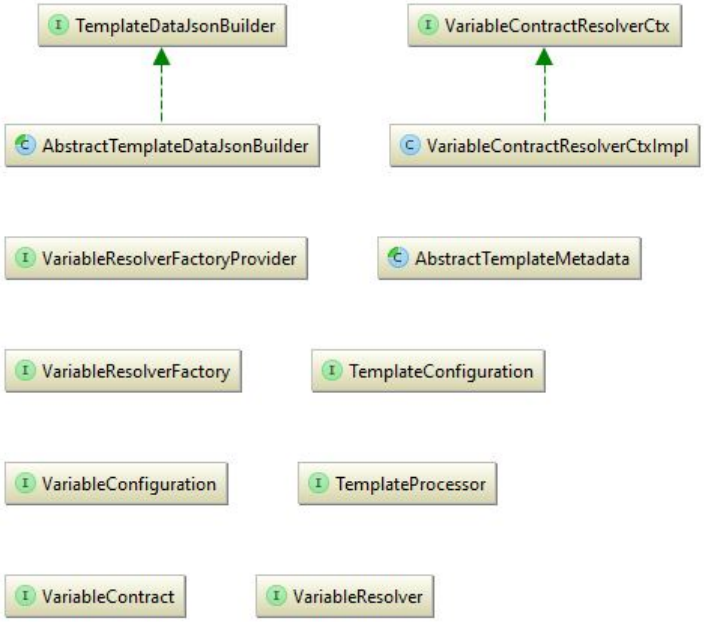


Abbildung 3.1: Klassenhierarchie des Vorlagenmanagements

3.1.1 Die Schnittstellen und abstrakten Klassen

Dieser Abschnitt behandelt die definierten Schnittstellen und abstrakten Klassen des Vorlagenmanagements. Die abstrakten Klassen implementieren die gemeinsame nutzbare Logik, die von alle konkreten Implementierungen des Vorlagenmanagements für jede *Template-Engine* 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*-Datenobjekts, welches die serialisierten Daten der Variablen einer Vorlage, sowie Metadaten der Vorlage enthält.

Die Schnittstelle *VariableContract*

Die Schnittstelle *VariableContract* aus Abbildung 3.1 spezifiziert eine Variable, die in einer Vorlage verwendet werden kann. Objekte dieser Schnittstelle werden beim Anwendungsstart registriert und können grundsätzlich in allen Vorlagen verwendet werden. Eine Variable ist einem Modul zugeordnet, wobei die Variable bezüglich ihres Namens innerhalb des Moduls eindeutig sein muss. Das Modul wird über eine Zeichenkette definiert. Die Mehrsprachigkeit einer Variable wird über Enumerationen realisiert, wobei jede Variable

jeweils einen Schlüssel für die Bezeichnung und die Beschreibung bereit stellen muss.

Da es sich bei einer Variable um statische Daten handelt, also die Variablen sind schon zur Kompilierungszeit bekannt, ist angedacht, dass die Variablen als *Enum* implementiert werden, das die Schnittstelle *VariableContract* implementiert. Durch die Abbildung der Variablen mit einer *Enum* können mehrere Variablen in einer Klasse definiert werden, wobei eine einzelne Enumeration der *Enum* 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 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* global eindeutig identifizierbar, wobei sich die *Id* aus dem Modulnamen und den Variablennamen zusammensetzt (Bsp. *module.core.VAR_1*). Die *Id* sowie der Modulname muss sich dabei an die Namenskonvention eines *Java*-Paketnamen halten. Da der Variablenname immer auf diese Weise zusammengesetzt werden muss, wurde die Methode *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 *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: Die Schnittstelle *VariableContract*

```
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* aus Abbildung 3.2 spezifiziert wie der aktuelle Wert der Variablen ermittelt wird. Beim Erstellen einer *E-Mail*-Nachricht auf Basis einer Vorlage müssen die aktuellen Werte der Variablen der Vorlage ermittelt werden. Da der aktuelle Wert einer Variable kontextabhängig ist, wird beim Ermitteln des aktuellen Werts einer Variable ein Kontextobjekt bereitgestellt, über das kontextabhängige Daten von den EntwicklerInnen bereitgestellt werden können. Durch dieses Kontextobjekt kann eine Variable in mehreren Kontexten verwendet werden und auch der aktuelle Wert einer Variable kontextabhängig ermittelt werden.

Programm 3.2: Die Schnittstelle *VariableResolver*

```
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. Die Verwendung von *Lambda*-Funktionen und Methodenreferenzen 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* aus Abbildung 3.3 spezifiziert wie Objekte der Schnittstelle *VariableResolver* produziert werden. Objekte dieser Schnittstelle können Objekte der Schnittstelle *VariableResolver* für jede Implementierung der Schnittstelle *VariableContract* produzieren. Es wird aber empfohlen, dass es je eine Implementierung der Schnittstelle *VariableResolverFactory* je Modul gibt.

Programm 3.3: Die Schnittstelle *VariableResolverFactory*

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

Die Schnittstelle *VariableResolver* wurde auch als *FunctionalInterface* implementiert, damit Implementierungen über eine *Lambda*-Funktion oder eine Methodenreferenz bereitgestellt werden können.

Die Schnittstelle *VariableResolverFactoryProvider*

Die Schnittstelle *VariableContractFactoryProvider* aus Abbildung 3.4 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. Die Schnittstelle *VaraibleResolverFactoryProvider* wurde spezifiziert, damit in einer *CDI*-Umgebung über ein Objekt dieser Schnittstelle die Objekte der Schnittstelle *VariableResolverFactory* produziert werden können, die von der *CDI*-Umgebung zur Verfügung gestellt werden.

Programm 3.4: Die Schnittstelle *VariableResolverFactoryProvider*

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

Die Schnittstelle *VariableContractResolverCtx*

Die Schnittstelle *VariableContractResolverCtx* aus Abbildung 3.5 spezifiziert den Kontext, der bei der beim Ermitteln des aktuellen Werts einer Variable zur Verfügung gestellt wird. Dieser Kontext stellt alle Daten bereit, die beim Ermitteln des aktuellen Werts einer Variable benötigt werden. Es wird auch ermöglicht, dass Benutzerdaten im Kontext definiert werden können, die bei beim Ermitteln des aktuellen Werts einer Variable verwendet werden können. Es wurde bewusst vermieden, dass beim Ermitteln 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: Die Schnittstelle *VariableContractResolverCtx*

```
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* aus Abbildung 3.7 spezifiziert wie die Variablen in einer Vorlagen behandelt werden. Objekte dieser Schnittstelle können Variablen in einer Vorlage für eine bestimmte *Template-Engine* finden und konvertieren. Ein Objekt der Schnittstelle *TemplateProcessor* muss in der Lage sein ungültige Variablen innerhalb einer Vorlage zu finden, wobei eine ungültige Variable eine Variable ist, die nicht registriert ist. Eine konkrete Implementierung der Schnittstelle *TemplateProcessor* ist eine Implementierung für eine bestimmte *Template-Engine*, da die in der Vorlage verwendeten Variablen in Form von Ausdrücken spezifisch für die verwendete *Template-Engine* sind.

Der Quelltext aus Abbildung 3.6 zeigt die beiden Methoden der Schnittstelle *TemplateProcessor*, die die Variablen in einer Vorlage konvertieren können.

Programm 3.6: Die Methoden für die Konvertierung

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

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

Diese Methoden definieren als Formalparameter für den benötigte Konverter ein *FunctionalInterface* namens *Function*, welches von *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.

Programm 3.7: Die Schnittstelle *TemplateProcessor*

```
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* aus Abbildung 3.9 spezifiziert die Signatur eines *Builders*, der das Datenobjekt erstellt, welches die Daten für das Parsen einer Vorlage enthält. Die *E-Mail*-Nachrichten werden persistent gehalten, wobei nach der Erstellung einer *E-Mail*-Nachricht, dessen Inhalt unveränderbar sein muss. Das Datenobjekt enthält Daten wie

- die Sprache in der die *E-Mail* versendet wird,
- die Zone für die Konvertierung von Datums- und Zeitwerten,
- die Version der Vorlage und
- die Metadaten der Vorlage wie z.B die Anzahl der enthaltenen Variablen.

Dieses Datenobjekt kann als *JSON* in den folgenden Formen vom *Builder* bereitgestellt werden.

- Als *Java*-Objekt,
- als *JSON*-Zeichenkette und
- als Objekt der Klasse *java.util.Map*.

Anstatt der Serialisierung der Daten könnte auch die Vorlage geparst und

persistent gehalten werden, wodurch aber die Menge an persistent gehaltenen Daten stark ansteigen würde. Mit dem Datenobjekt werden nur die benötigten Daten persistent gehalten, wodurch die Menge an persistent gehaltenen Daten so klein wie möglich gehalten wird. Mit diesem Datenobjekt kann die korrespondierende Vorlage zu jedem Zeitpunkt mit demselben Resultat wiederhergestellt 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 Datenobjekt in Form eines *Java*-Objekts zurückliefert.
- *String toJsonString()*
  ist die Methode, die das Datenobjekt als *JSON*-Zeichenkette zurückliefert.
- *Map<String, Object> toJsonMap()*
  ist die Methode, die das Datenobjekt in Form eines Objekts der Klasse *java.util.Map* zurückliefert.

Der Quelltext aus Abbildung 3.8 illustriert, wie der *Builder* verwendet wird.

Programm 3.8: Beispiel der Anwendung des *Builders*

```
1 builder.withStrictMode()
2        .withLocalization(localeObj, zoneIdObj)
3        .withTemplate(templateMetadataObj)
4        .withUserData(userDataMap)
5        .withVariableResolverFactoryProvider(factoryProviderObj)
6        .toJsonModel();
```

Programm 3.9: Die Schnittstelle *TemplateDataJsonBuilder*

```
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 verschiedenen *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 spezifische *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 ein neues Objekt der Metadaten erstellt werden.

TODO: Add source as appendix, because to long ?

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 Hilfsmethoden bereit, die Variablen innerhalb der Vorlage finden, validieren und den aktuellen Wert von Variablen ermitteln können. Das resultierende Datenobjekt des *Builders* ist spezifiziert, jedoch nicht die Abbildung der ermittelten Werte für die enthaltenen Variablen. Diese Daten sind spezifisch für die verwendete *Template-Engine*.

TODO: Add source as appendix, because to long ?

3.2 Die Spezifikation der Vorlagenintegration

Die im Abschnitt 3.1 vorgestellte Spezifikation des Vorlagenmanagements, spezifiziert die Kernfunktionalität des Vorlagenmanagements, das in der Lage ist die Vorlagen sowie deren enthaltene Variablen zu behandeln. Das Vorlagenmanagement benötigt auch Integrationen in verschiedene Umgebungen und Sprachen, um die benötigte Funktionalitäten wie

- die Verwaltung der Variablen in einem *JavaScript*-basierten *CKEditor*,
- 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 Spezifikationen der Integrationen wie in Abschnitt 2.2 vorgegeben.

3.2.1 Das Vorlagenmanagement in *TypeScript*

Wie in Abschnitt 2.1.6 vorgegeben muss der *JavaScript*- basierte *Editor CKEditor* verwendet werden, mit dem *HTML* basierte Vorlagen in einer Webseite bearbeitet werden können. Der *CKEditor* muss angepasst werden, damit die definierten Variablen in einer Vorlage verwendet werden können. Es wird ein *CKEditor-Plugin* in *TypeScript* entwickelt, das es erlaubt, die

definierten Variablen innerhalb des *CKEditors* und dessen enthaltener Vorlage zu verwalten. Es wird die Skriptsprache *TypeScript* verwendet, da es mit dieser Skriptsprache möglich ist typsicher zu entwickeln, was in *JavaScript* nicht möglich ist. Ebenfalls kann *TypeScript* in mehrere *ECMA*-Standards übersetzt werden.

Innerhalb des *CKEditor-Plguins* werden Variablen verwendet, dessen Management in einer eigenen Quelltextdatei implementiert wird, da es unabhängig von *CKEditor-Plguin* ist und daher auch anderweitig verwendet werden kann. Damit ist das Variablenmanagement entkoppelt vom *CKEditor-Plugin*.

3.2.2 Das Vorlagenmanagement in *CDI*

Das Vorlagenmanagement wird in einem *JEE-7*-Anwendungsserver verwendet, der eine *CDI*-Umgebung bereitstellt. Im *CDI*-Standard sind portable Erweiterungen spezifiziert, die es erlauben, dass sich Softwarekomponenten in einer *CDI*-Umgebung integrieren können. Es wird eine *CDI*-Erweiterung implementiert, die beim Start der *CDI*-Erweiterung, die definierten Variablen automatisch registriert und über den Lebenszyklus der Anwendung persistent hält. 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.

Durch die Verwaltung der Objekte von einer *CDI*-Umgebung, können Implementierungen der Schnittstelle *VariableResolver* kontextabhängige Ressourcen sich injizieren lassen. Damit das Variablenmanagement auf diese Objekte zugreifen kann, wurde die Schnittstelle *VariableResolverFactoryProvider* spezifiziert, die die Verbindung des Variablenmanagements zu einer *CDI*-Umgebung herstellt und kontextabhängige Objekte der Schnittstelle *VariableResolverFactory* bereitstellen kann.

3.2.3 Das Vorlagenmanagement in *JSF*

Für die Verwaltung der Vorlagen wird eine *JSF*-Webseite implementiert. Über diese Webseite können Vorlagen erstellt, modifiziert und gelöscht werden können. Für die Verwaltung der Vorlagen wird die von *PrimeFaces-Extension* bereitgestellte *JSF*-Komponente für den *Editor CKEditor* verwendet. Diese Komponente integriert den *Javascript*-basierten *CKEDitor* in den *JSF*-Lebenszyklus. Um die Vorlage in die korrespondierende *Template-Engine* spezifische Repräsentation zu überführen, wird ein *FacesConverter* implementiert, der die Konvertierung der Vorlage von seiner *HTML-*

Repräsentation in die *Template-Engine* spezifische Repräsentation und visa versa übernimmt.

3.2.4 Das Vorlagenmanagement in *Mail*-DB-Schema

Eine Vorlage wird durch eine Zeichenkette repräsentiert, die 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 von *CleverMail* darstellen und daher auch die Vorlagen bzw. deren persistente Repräsentation voll in das *Mail*-DB-Schema integriert werden müssen. Sollten die Vorlagen außerhalb von *CleverMail* verwendet werden so kann dies leicht realisiert werden, da eine Vorlage nur eine Zeichenkette darstellt, die einfach persistent gehalten werden kann.

Kapitel 4

Die Realisierung

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

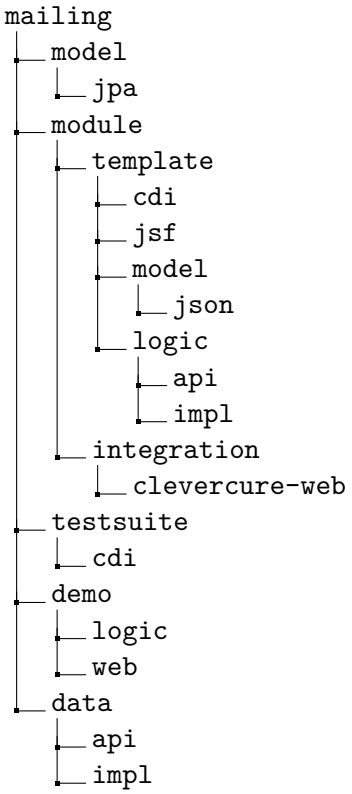


Abbildung 4.1: Verzeichnisstruktur der *Maven*-Projekte

Das *Maven*-Wurzelprojekt *mailing* organisiert die Metadaten wie die EntwicklerInnen, die an diesem Projekt mitwirken, alle benötigten Abhängigkeiten, sowie die auf alle Unterprojekte anwendbare *Build*-Konfigurationen. Die übergeordneten Projekte sind vom Typ *pom*, was bedeutet, dass aus diesen Projekten keine Artefakte erstellt werden und die übergeordneten Projekte die tiefer liegenden Projekte bündeln. Die gesamte Organisation der Abhängigkeiten findet im Wurzelprojekt *mailing* statt. Diese Projektstruktur wurde gewählt, da in diesem Projekt auch die Implementierungen der anderen Softwarekomponenten von *CleverMail* organisiert werden. Die konkreten Artefakte wurden jeweils in ein Artefakt **-api* und **-impl* aufgeteilt, somit sind die Schnittstellen (Spezifikationen) vollständig getrennt von deren Implementierungen. Folgende Auflistung beschreibt alle konkreten Artefakte *(Java-Archive)*, die aus dem Wurzelprojekt *mailing* erstellt werden können:

- ***mailing-model-jpa***
  ist das Artefakt, das die Klassen mit den *JPA*--Entitäten enthält, die die Datenbank in *Java* abbilden.
- ***mailing-module-template-cdi***
  ist das Artefakt, das die Implementierung der *CDI*-Erweiterung für die Integration in eine *CDI*-Umgebung enthält.
- ***mailing-module-template-jsf***
  ist das Artefakt, das die Implementierung für die Integration in *JSF* enthält.
- ***mailing-module-template-model-json***
  ist das Artefakt, das die Implementierung der *JSON*-Datenobjekte in Form von *Java*-Klassen enthält.
- ***mailing-module-template-logic-api***
  ist das Artefakt, das die Spezifikation des Vorlagenmanagements enthält.
- ***mailing-module-template-logic-impl***
  ist das Artefakt, das die Implementierung der Spezifikation des Vorlagenmanagements enthält.
- ***mailing-module-integartion-clevercure-web***
  ist das Artefakt, das die Implementierung der Integration für die Anwendung *CleverWeb* enthält.
- ***mailing-testsuite-cdi***,
  ist das Artefakt, das die Ressourcen aller Tests, die in einer *CDI*-Umgebung lauffähig sein müssen, enthält.
- ***mailing-demo-logic***
  ist das Artefakt, das die Schicht der Geschäftslogik der Beispielanwendung enthält.
- ***mailing-demo-web***

ist das Artefakt, das die *Web*-Anwendung der Beispielanwendung enthält.

- ***mailing-data-api***
  ist das Artefakt, dass die Spezifikation der Geschäftslogik enthält, die die Persistenz der *E-Mail*-Vorlagen behandeln. Es enthält auch die Datenbankzugriffsklassen in Form von *Data-Repository*-Schnittstellen.
- ***mailing-data-impl***
  ist das Artefakt, das die Implementierung der Geschäftslogik enthält.

4.1 Die Implementierung der Spezifikationen

Dieser Abschnitt behandelt die Implementierungen der Spezifikationen, die im Kapitel 3 vorgestellt wurden.

4.1.1 Die Implementierung für *CKEditor*

Wie im Abschnitt 3.2.1 vorgegeben, wurde ein *Plugin* in *TypeScript* implementiert, dass innerhalb des *CKEDitors* die Variablen verwaltet. Die Implementierung des *Plugins* in *TypeScript* war möglich, da für den *Editor CKEditor* vom dem, von *Microsoft* verwalteten, *Open-Source* Projekt *DefinitelyTyped* Typinformationen für *TypeScript* bereitgestellt werden, die die *JavaScript*-Schnittstellen als *TypeScript*-Schnittstellen definieren. Hätten keine Typinformationen zur Verfügung gestanden, hätte man die Typinformationen selber implementieren müssen, was einen erheblichen Mehraufwand bedeutet hätte.

Das *CKEditor-Plugin* in *Typescript*

Das Variablenmanagement ist unabhängig vom verwendeten *Editor* und wurde daher vom *CKEditor-Plugin* logisch und physisch getrennt, wobei das Variablenmanagement im *TypeScript*-Modul *cc.variables* und das *CKEDitor-Plugin* im *TypeScript*-Modul *cc.ckeditor.plugins* implementiert wurden. Die voneinander getrennten *TypeScript*-Quelltextdateien werden beim Kompilieren in eine einzige *JavaScript*-Quelltextdatei zusammengeführt. Mit der Organisation in eigenen *TypeScript*-Modulen wird sichergestellt, dass nur explizit nach außen sichtbar gemachte *(export MyType {...})* Funktionen oder Typen außerhalb des Moduls referenziert werden können. Ein *TypeScript*-Modul wird in ein korrespondierendes *JavaScript*-Modul übersetzt. Die Verwendung von Modulen bring auch den Vorteil, dass am *Window*-Objekt nur das Objekt der Wurzel des Namensraums *cc* gebunden ist, wodurch das *Window*-Objekt nicht mit den eigenen*JavaScript*-Objekten verschmutzt wird. Die Quelltexte aus den Abbildungen 4.1 und 4.2 zeigen ein *TypeSciptt*-Modul und das daraus resultierende *JavaScript*-Modul.

Programm 4.1: Das *TypeScript*-Modul

```
module cc.ckeditor.plugins {
  export module variables {
    export interface VariableMapping{
          id:string
    }
  }
}
```

Programm 4.2: Das *JavaScript*-Modul

```
var cc;
(function (cc) {
    var variables;
    (function (variables_1) {
    // VariableMapping nicht Teil des generierten JavaScripts
    })(variables = cc.variables || (cc.variables = {}));
})(cc || (cc = {}));
```

Die *TypeScript*-Schnittstelle *VariableMapping* aus dem Quelltext aus Abbildung 4.1 ist nicht Teil des generierten *JavaScript*-Moduls, da diese Schnittstelle nur eine Typinformation für *TypeScript* darstellt. Wäre die Schnittstelle *VariableMapping* eine *TypeScript*-Klasse, dann wäre diese Klasse auch Teil des generierten *JavaScript*-Moduls und würde als *JavaScript*-Funktion abgebildet werden.

Das Variablenmanagement in *TypeScript* ist verantwortlich für die *Browser*-seitige Registrierung der Variablen und stellt Hilfsmethoden zur Verfügung, mit denen Variablen in der *HTML*-Vorlage gefunden und konvertiert werden können. Der Quelltext aus Abbildung 4.3 zeigt mehrere Möglichkeiten, wie eine Variable in *TypeScript* konvertiert werden kann.

Programm 4.3: Beispiele für Variablenkonvertierungen in *TypeScript*

```
1  // Hilfsklasse für die Konvertierung der Variablen
2  class VariableUtils {
3      private variables:VariableMapping[] = [];
4
5      // Öffentliche Funktion für die Konvertierung der Variablen
6      public convert(converter:(item:VariableMapping) => any
7                              = (item:VariableMapping)=> item):any[] {
8          var converted:any[] = [];
9          for (var i = 0; i < this.variables.length; i++) {
10             converted[i] = converter(this.variables[i]);
11         }
12         return converted;
13     }
14 }
15
16 // Eigene Klasse für die Konvertierung
17 class MyConverter {
18     // Öffentliche Funktion für die Konvertierung der Variablen
19     public convert(v:VariableMapping): any {
20         return v.displayName;
21     }
22 }
23
24 // Erstellen der Objekte aus den definierten Klassen
25 var util     :VariableUtils = new VariableUtils();
26 var converter:MyConverter   = new MyConverter();
27
28 // Konvertierung mit einer Arrow-Funktion
29 util.convert((v:VariableMapping) => v.displayName);
30
31 // Konvertierung mit einer anonymen Funktion
32 util.convert(function (v:VariableMapping) {
33     return v.displayName;
34 });
35
36 // Konvertierung mit einer Referenz auf eine Funktion
37 util.convert(converter.convert);
```

Die Funktion *convert* der Klasse *VariableUtil* aus dem Quelltext aus Abbildung 4.3 definiert den Formalparameter *converter* als eine *Arrow*-Funktion, die die Signatur der Funktion für die Konvertierung definiert und eine Standardimplementierung definiert, die verwendet wird, wenn bei der Aktivierung der Funktion *convert* für den Formalparameter *converter* kein Aktualparameter bereitgestellt wird. Eine *Arrow*-Funktion ähnelt einer *Lambda*-Funktion in *Java*. Der Typ *any[]* ist vergleichbar mit dem Datentyp *var* aus *.NET* und gibt an, dass jeder Datentyp als Typ des zurückgelieferten *Arrays* erlaubt ist.

Die Variablenrepräsentation in *TypeScript*

Die Variablen werden *Java* seitig als Objekte der Schnittstelle *VariableContract* abgebildet, und müssen für das *Javascript* seitige Variablenmanagement in eine *JSON*-Zeichenkette überführt werden, die als *Javascript*-Objekt innerhalb des *Javascript* seitige Variablenmanagements verwendet werden. Dafür wurde in *Typescript* die Schnittstelle *VariableMapping* aus dem Quelltext 4.4 definiert, die die Struktur einer Variable innerhalb von *Typescript* spezifiziert.

Programm 4.4: *Typescript*-Funktion für die Variablenkonvertierung

```
interface VariableMapping {
  id:string,
  displayName:string,
  info:string,
}
```

Die Schnittstelle *VariableMapping* 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* verwendet werden. Mit der Schnittstelle *VariableMapping* werden Typinformationen für der Variablenpräsentation in *Typescript* bereitgestellt, damit innerhalb von *Typescript* die Typsicherheit der Variablenrepräsentation sichergestellt werden kann.

Die Variablenrepräsentation in Java

Der Quelltext aus 4.5 zeigt die korrespondierende Implementierung der *JSON*-Spezifikation in *Java* mit der Klasse *VariableJson*. Mit der Klasse *VariableJson* wird sichergestellt, das die Variablenrepräsentation in *Java* korrespondierend zur Variablenrepräsentation in *Typescript* ist. 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 kann zu Problemen führen.

Programm 4.5: 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, String displayName, String tooltip) {
12        this.id = id;
13        this.label = displayName;
14        this.info = tooltip;
15    }
16
17    @JsonGetter("id")
18    public String getId() { return id; }
19
20    @JsonSetter("id")
21    public void setId(String id) { this.id = id; }
22
23    @JsonGetter("displayName")
24    public String getLabel() { return label; }
25
26    @JsonSetter("displayName")
27    public void setLabel(String label) { this.label = label; }
28
29    @JsonGetter("info")
30    public String getInfo() { return info; }
31
32    @JsonSetter("info")
33    public void setInfo(String info) { this.info = info; }
34 }
```

Registrierung des *Plugins* im *CKEditor*

Das *Plugin* wird über eine *JavaScript*-Datei im *CKEditor* registriert, wobei folgende Konventionen eingehalten werden müssen.

- *ckeditor/plugins*
  ist das Verzeichnis, in dem das *Plugin* enthalten sein muss.
- *variables*
  ist das Verzeichnis unterhalb des Verzeichnisses *ckeditor/plugins*, in dem die *Plugin*-Ressourcen enthalten sein müssen und das dem Namen des *Plugins* entspricht.
- *plugin.js*

ist die *JavaScript*-Datei, die im Verzeichnis *ckeditor/plugins/variables* liegen muss und das implementierte *Plugin* darstellt.

Der Quelltext aus Abbildung 4.6 zeigt einen Auszug aus der *JavaScript*-Datei mit dem das *Plugin* registriert wird und auch Einstellungen am *CKEditor* vorgenommen werden können. Das *Plguin* wird vom *CKEditor* nach dessen Initialisierung geladen und registriert.

Programm 4.6: Registrierung des *CKEditor-Plugins*

```
1 CKEDITOR.editorConfig = function (config) {
2     config.extraPlugins = "variables";
3 }
```

Integration des *Plugins* im *CKEditor*

Die Abbildung 4.2 zeigt die Funktionsleiste des *CKEditors*, in die der rot markierte *Button* über das *Plugin* eingefügt wurde. Durch einen Klick auf diesen *Button* wird ein Dialog geöffnet, über den die zur Verfügung stehenden Variablen ausgewählt werden können.

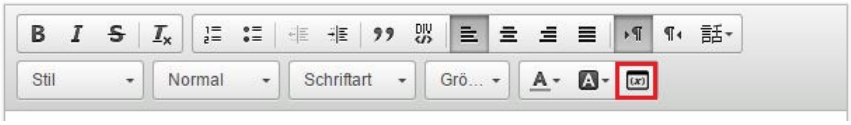


Abbildung 4.2: Die *CKEditor*-Funktionsleiste

Die Abbildung 4.3 zeigt den Dialog, der vom *CKEditor-Plugin* definiert und erstellt wurde. In diesem Dialog stehen alle registrierten Variablen zur Auswahl. Die Bezeichnung der Variable ist der Text in der Auswahlkomponente und die Beschreibung der ausgewählten Variable wird unterhalb der Auswahlkomponente angezeigt. Durch den Klick auf den *Button OK* wird die Variable in die Vorlage eingefügt und der Dialog wird geschlossen.

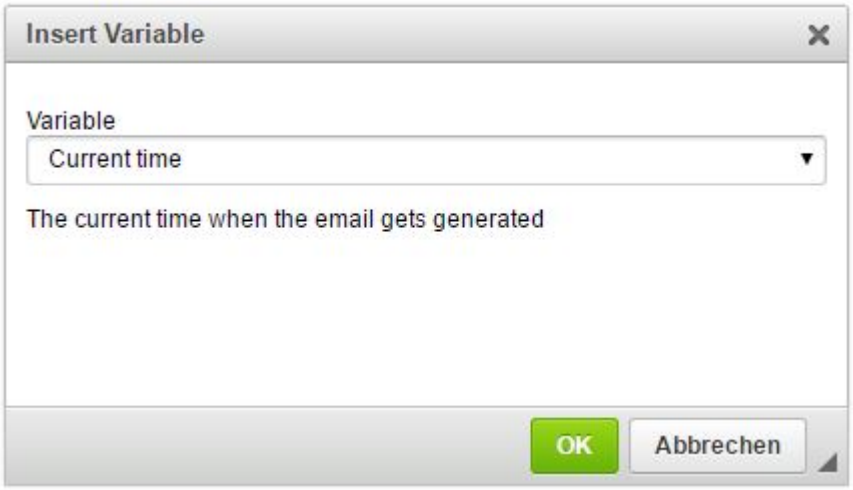


Abbildung 4.3: *CKEditor* Dialog für die Variablenauswahl

Die Abbildung 4.4 zeigt eine Vorlage innerhalb des *CKEditors*, wobei die eingefügten Variablen besonders hervorgehoben werden. Die Bezeichnung der Variable stellt den Namen für den *HTML-Tag* bereit und die Beschreibung dessen Titel. Die eingefügten *HTML-Tags* dürfen nicht verändert werden, daher ist das *Drag* und *Drop* und das Selektieren des eingefügten *HTML-Tags* nicht erlaubt, da dadurch der eingefügte *HTML-Tag* zerstört werden könnte und die Variablen nicht mehr gefunden werden können.

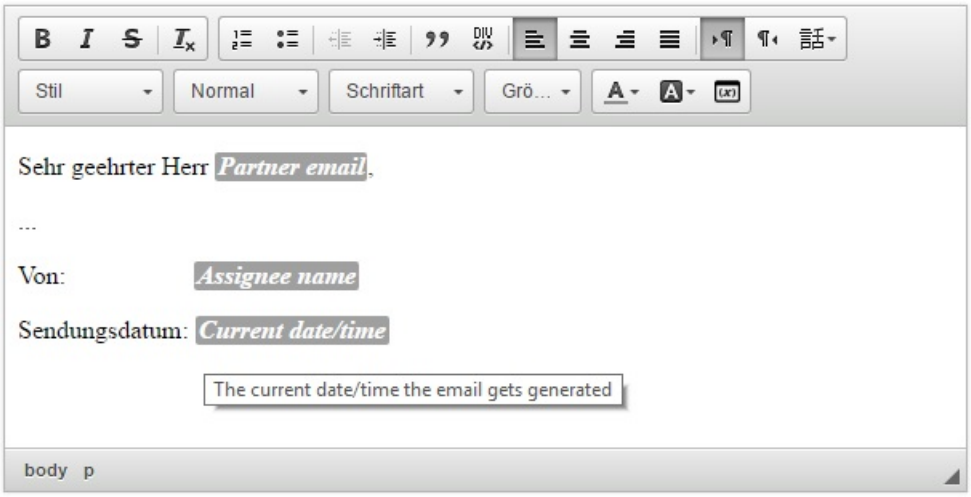


Abbildung 4.4: Beispiel einer Vorlage im *CKEditor*

4.1.2 Die Implementierungen für *CDI*

Folgender Abschnitt behandelt die Implementierungen für die Integration in eine *CDI*-Umgebung. Wie in Abschnitt 3.2.2 beschrieben, sollen die Variablen beim Start der Anwendung, die in einer *CDI*-Umgebung läuft, automatisch registriert werden. Das Vorlagenmanagement stellt Ressourcen die folgend aufgelisteten Ressourcen kontextabhängig über einen implementierten *CDI*-Erzeuger zur Verfügung.

- Objekte der Schnittstelle *VariableConfiguration* sind Objekte, die die registrierten Variablen verwalten.
- Objekte der Schnittstelle *TemplateDataJsonBuilder* sind Objekte, mit denen das Datenobjekt in Form von *JSON* für eine Vorlage und eine spezifische *Template-Engine* erstellt werden kann.
- Objekte der Schnittstelle *TemplateProcessor* sind Objekte, mit denen Variablen in Vorlagen verwaltet werden können.
- Objekte der Klasse *CdiTemplateUtil* sind Objekte mit denen die registrierten Variablen, die Objekte der Schnittstelle *VariableContract* sind, in Objekte der Klasse *VariableJson* konvertieren kann, wobei der Titel und die Beschreibung sprachspezifisch gesetzt werden.

Die Vorlagenmanagement *CDI*-Erweiterung

Um die Variablen beim Start der Anwendung innerhalb einer *CDI*-Umgebung automatisch registrieren zu können, wurde eine *CDI*-Erweiterung *TemplateCdiExtension* implementiert, die das Variablenmanagement in eine *CDI*-Umgebung integriert. Eine Erweiterung für eine *CDI*-Erweiterung muss folgende Voraussetzungen erfüllen.

1. Die Schnittstelle *javax.enterprise.inject.spi.Extension* implementieren und
2. in einer Datei namens *javax.enterprise.inject.spi.Extension*, die im Verzeichnis *META-INF/services* liegen muss, mit ihren vollständigen Namen registriert werden.

Durch die Datei *javax.enterprise.inject.spi.Extension* wird die Erweiterung der *CDI*-Umgebung bekannt gemacht und wird beim Start der *CDI*-Umgebung geladen und kann auf Ereignisse des Lebenszyklus reagieren, in dem Sie Beobachtermethoden implementiert, die für die einzelnen Ereignisse aufgerufen werden. Die Schnittstelle *javax.enterprise.inject.spi.Extension* ist ein Interface, das keine abstrakten Methoden enthält und als Markierung fungiert, um ein Klasse als *CDI*-Erweiterung zu markieren. Die Erweiterung wird über den *Service-Provider-Interface (SPI)* Mechanismus geladen.

Eine *CDI*-Erweiterung ist an sich kein *CDI-Bean*, da das Objekt der *CDI*-Erweiterung bereits beim Start des *CDI-Containers* erstellt wird und somit

schon existiert bevor die *CDI*-Umgebung vollständig gestartet wurde. Trotzdem ist das Objekt der *CDI*-Erweiterung injizierbar und kann in *CDI-Beans* injiziert werden. Das erstellte Objekt der *CDI*-Erweiterung *TemplateCdiExtension* existiert über die Lebensdauer der *CDI*-Umgebung.

Der Quelltext aus Abbildung 4.7 ist ein Auszug aus der implementierten *CDI*-Erweiterung *TemplateCdiExtension* und zeigt die Beobachtermethoden, die Lebenszyklus Ereignisse der *CDI*-Umgebung beobachten. Über die *CDI*-Erweiterung des Vorlagenmanagements werden alle implementierten Typen der Schnittstelle *VariableContract* über die Beobachtermethoden gefunden und im Objekt der Klasse *TemplateConfiguration* registriert, das über die Lebensdauer der *CDI*-Erweiterung existiert. Es werden nur Implementierungen der Schnittstelle *VariableContract* unterstützt, die als *enum* implementiert wurden, obwohl auch Implementierung von Klassen unterstützt werden könnten, in dem die Typen der Schnittstelle *VariableContract* gesammelt und im weiteren Programmverlauf manuell aus der *CDI*-Umgebung geholt werden könnten. Es werden auch alle implementierten Typen der Schnittstelle *VariableResolverFactory* gefunden und in der *CDI*-Erweiterung registriert.

Programm 4.7: Auszug aus der *CDI*-Erweiterung *TemplateCdiExtension*

```
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
9     void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
10        // Init class members
11    }
12
13    <T> void processCdiVariableContracts
14              (@Observes @WithAnnotations({BaseName.class,
15                                           CdiVariableContract.class})
16              ProcessAnnotatedType<T> pat) {
17        // Collect VariableContract types (Enum type only)
18    }
19
20    <T> void processVariableResolverFactoryFactories
21        (@Observes @WithAnnotations(CdiVariableResolverFactory.class)
22        ProcessAnnotatedType<T> pat) {
23        // Collect VariableResolverFactory types
24    }
25 }
```

- *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*-Erzeuger

Es wurde eine *CDI*-Erzeuger Klasse *TemplateResourceProducer* implementiert, mit der kontextabhängig Ressourcen des Vorlagenmanagements produziert werden. Diese Klasse ist die einzige Klasse, die sich die *CDI*-Erweiterung *TemplateCdiExtension* injizieren lässt. Es kann nicht verhindert werden, dass andere *CDI-Beans* sich diese Klasse injizieren lassen, da eine *CDI*-Erweiterung öffentlich sein muss. Es wir aber empfohlen, dass niemand außer die *CDI*-Integration selbst sich das Objekt der *CDI*-Erweiterung injizieren lässt.

Wie im Kapitel 3 vorgegeben sollen mehrere *Template-Engines* unterstützt werden, daher wurde die Annotation *@FreemarkerTemplate* eingeführt, die einen Injektionspunkt für *Freemarker* qualifiziert. In *CDI* wird ein Qualifizierer benötigt, wenn für eine Schnittstelle bzw. eine Typ mehrere Implementierungen zur Verfügung stehen, da die *CDI*-Umgebung in so einem Fall nicht entschieden kann welche Implementierung verwendet werden soll. Es wurde jeweils eine Erzeugermethode für den Qualifizierer *@Default* und *@FreemarkerTemplate* implementiert. Für den Qualifizierer *@Default* wird die Implementierung für die *Template-Engine Freemarker* verwendet, wodurch diese Implementierung als die Standardimplementierungen fungieren. Man setzt sich aber der Gefahr aus, dass die produzierte *@Default* Implementierung nicht die gewollte ist. Der Qualifizierer muss an einem Injektionspunkt angegeben werden, wenn ein anderer Qualifizierer als *@Default* verwendet werden soll. Der Qualifizierer *@Default* wird immer als Standard Qualifizierer herangezogen, wenn keine expliziter Qualifizierer am Injektionspunkt angegeben wurde.

Der Quelltext aus Abbildung 4.8 ist ein Auszug aus der Klasse *TemplateResourceProducer* und zeigt einige der implementierten Erzeugermethoden.

Programm 4.8: 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 }
```

Es wurden zwei Erzeugermethoden implementiert um Objekte der Schnittstelle *TemplateDataJsonBuilder* zu erzeugen.

1. *produceDefaultTemplateBuilder* für *@Default* Qualifizierer und
2. *produceFreeMarkerTemplateBuilder* für *@Freemarker* Qualifizierer.

Diese beiden Methoden produzieren Objekte für den sogenannten Pseudo-*Scope @Dependent*, wobei für jeden Injektionspunkt ein neues Objekt erstellt wird. Der Lebenszyklus von *CDI-Beans* im Pseudo-*Scope @Default* wird nicht von der *CDI*-Umgebung verwaltet. Beim Erzeugen eines solchen Objekts wird lediglich Injektion durchgeführt und die Lebensdauer eines solchen Objekts hängt davon ab, ob das Objekt noch referenziert wird. Als Argument für diesen beiden Methoden wird ein Objekt der Schnittstelle *VariableResolverFactoryProvider* injiziert, das mit dem Qualifzierer *@Default* annotiert 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 *CDI*-Erweiterung bereitgestellt wird. Nachdem diese Schnittstelle nur lesenden Zugriff erlaubt und nur die *CDI*-Erweiterung Variablen registriert, wird dieses Objekt für den Gültigkeitsbereich der Anwendung produziert, also einmalig für die gesamte Anwendungslaufzeit.

Die Vorlagenmanagement *CDI*-Hilfsklasse

Die Klasse *CdiTemplateUtil* wurde implementiert um ein injizierbares *CDI-Bean* zur Verfügung zu stellen, das Hilfsmethoden für die Konvertierung der Variablen von Objekten der Schnittstelle *VariableContract* in Objekte der Klasse *VariableJson* und visa versa zur Verfügung stellt. Diese Implementierung ist statuslos, daher kann dieses *CDI-Bean* in den Anwendungskontext gelegt werden.

Programm 4.9: CdiTemplateUtil.java

```
1 @ApplicationScoped
2 @Typed(CdiTemplateUtil.class)
3 public class CdiTemplateUtil implements Serializable {
4
5     @Inject
6     private VariableConfiguration config;
7
8     public List<VariableJson> convertContractToJsonModel
9                   (final Locale locale) {
10    }
11
12    public List<VariableJson> convertContractToJsonModel
13        (final Collection<VariableContract> contracts,
14             final Locale locale) {
15    }
16
17    public VariableJson convertContractToJsonModel
18           (final VariableContract contract,
19            final Locale locale) {
20    }
21
22    public List<VariableContract> convertJsonModelToContract
23                  (final Collection<VariableJson> jsonModels) {
24    }
25
26    public VariableContract convertJsonModelToContract
27                            (final VariableJson jsonModel) {
28    }
29 }
```

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 der *CKEditor*-Integration, bereitgestellt von *primefaces-extensions*, beschäftigen.

Der Vorlagen *FacesConverter*

Es wurde der Konverter *AbstractTemplateConverter* als abstrakte Klasse implementiert, die die Schnittstelle *javax.faces.Converter* implementiert. Diese abstrakte Klasse wurde implementiert, da die Logik für die Konvertierung0 über alle *Template-Engines* dieselbe ist und sich lediglich die Implementierung der Schnittstelle *TemplateProcessor* unterschiedet. Das Objekt der Schnittstelle *TemplateProcessor* und das Objekt der Klasse *CdiTemplateUtil* werden manuell von der *CDI*-Umgebung geholt, da keine Injektion innerhalb von *JSF*-Artfakten in *JSF 2.2* möglich ist. Die Injektion in *JSF*-Artefakte wird erst ab *JSF 2.3* unterstützt werden. Die Objekte werden über die Klasse *BeanProvider* der Bibliothek *Deltaspike* geholt, die Hilfsmethoden zur Verfügung stellt, mit denen man zur Laufzeit manuell mit der *CDI*-Umgebung interagieren kann. *Deltaspike* ist eine Bibliothek, die eine portable *CDI*-Erweiterung darstellt.

Die konkrete Implementierung *FreemarkerTemplateConverter* für die *Template-Engine Freemarker*, die von der abstrakten Klasse *AbstractTemplateConverter* ableitet, setzt über einen Konstruktor in der Basisklasse den korrespondierenden Qualifizierer für die verwendete *Template-Engine* und das zu verwendende Objekt der Klasse *java.util.Locale*. Mit diesem Qualifizierer wird die korrekte Implementierung der Schnittstelle *TemplateProcessor* aus dem *CDI-Container* geholt. Da der Konverter eine Abhängigkeit auf ein *Locale* Objekt besitzt, muss ein Objekt des Konverters im *Quelltext* erzeugt und über Parameterbindung an eine *JSF*-Komponente gebunden werden. Das Binden des Konverters an eine *JSF*-Komponente über dessen Namen definierbar über die Annotation *@FacesConverter("converterName")* ist nicht möglich.

Programm 4.10: FreemarkerTemplateConverter.java

```
1 public class FreemarkerTemplateConverter
2                           extends AbstractTemplateConverter {
3
4     public FreemarkerTemplateConverter(final Locale locale) {
5         super(new FreemarkerTemplateLiteral(), locale);
6     }
7 }
```

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

```
String tagRegex = "(<span[^,>]*class=\"variable\"[^,>]*>[^,<]*</span>)";
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 mit dem regulären Ausdruck *tagRegex* gefunden wurde.

Die abstrakte Klasse *AbstractTemplateConverter* definiert auch eine Vorlage in Form einer Zeichenkette, mit der die Variablen in ihre *HTML-Tag*-Repräsentation konvertiert werden können, wobei diese Vorlage unabhängig von der verwendeten *Template-Engine* ist und auf alle Variablen gleich angewendet werden kann.

```
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, wobei der Formalparameter *Object...*, der eine variable Argumentliste repräsentiert, über den die dynamischen Werte für die Vorlage bereitgestellt werden können.

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 für den *CKEditor* 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 vorhanden ist, wie von *Primefaces-Extensions* unterstützt wird. Der *CKEditor* ist ein sehr umfangreicher *Editor*, den man sich auch seinen Wünschen entsprechend selbst zusammenstellen kann. Eine solche benutzerdefinierte Zusammenstellung des *CKEditors* kann man heranziehen, um die Standardimplementierung zu ersetzen.

Der Quelltest aus Abbildung 4.11 illustriert die Verwendung des *CKEditors* in Form der zur Verfügung gestellten *JSF*-Komponente.

Programm 4.11: *XHTML-Markup* für *CKEditor*

```
1 <pe:ckEditor id="template_content_editor"
2              wdgetVar="pfEditor"
3              value="#{templateEditModel.content}"
4              converter="#{ckeditorBean.converter}"
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, über Parameterbindung 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 Verwendung in der Geschäftslogik,
- die Verwendung über eine Webseite und
- die Verwendung zum Erstellen einer *E-Mail*

aufzeigen soll. Dazu wurde eine Demowebanwendung implementiert, die die Web seitige Verwaltung der Vorlagen implementiert. Es wurde auch eine Klasse implementiert, die aufzeigen soll, wie eine *E-Mail* basierend auf einer Vorlage, aus einer Geschäftslogik heraus erstellt werden kann.

4.2.1 Die Verwendung in einem *Business*-Service

Der folgende Quelltext aus Abbildung 4.12 zeigt wie eine *E-Mail* über die die implementierte Klasse *EmailServiceImpl* der Schnittstelle *EmailService* erstellt werden kann. Das Objekt der Schnittstelle *EmailService* wird über die *CDI*-Umgebung zur Verfügung gestellt und mittels Injektion in die Geschäftslogik injiziert. Die Schnittstelle *EmailService* und dessen Implementierung *EmailServiceCdiEventImpl* befinden sich im Artefakt *mailing-template-integration-clevercure-web*. Dieses Artefakt stellt die Integration in die Anwendung *CleverWeb* dar. Die *E-Mails* werden in der Implementierung *EmailServiceCdiEventImpl* über *CDI-Events* erstellt, damit ist die Logik für das Erstellen der *E-Mail* vollständig entkoppelt von dieser Implementierung. Im folgenden sind die zur Verfügung gestellten Methoden der Schnittstelle *EmailService* angeführt, die der Geschäftslogik zur Verfügung stehen.

- *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 4.12: 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 }
```

Der Quelltext aus Abbildung 4.13 zeigt das Beispiel der Geschäftslogik, die 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.

Programm 4.13: 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 }
```

Folgende Auflistung erklärt die Attribute, die beim Erstellen eine Objekts der Klasse *EmailDto* angegeben werden müssen.

- *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 über eine *Web*-Oberfläche

Die Abbildung 4.5 zeigt die Weboberfläche, die für die Beispielanwendung implementiert wurde. Über dieses Formular können die Voralgen sprachspezifisch verwaltet werden.

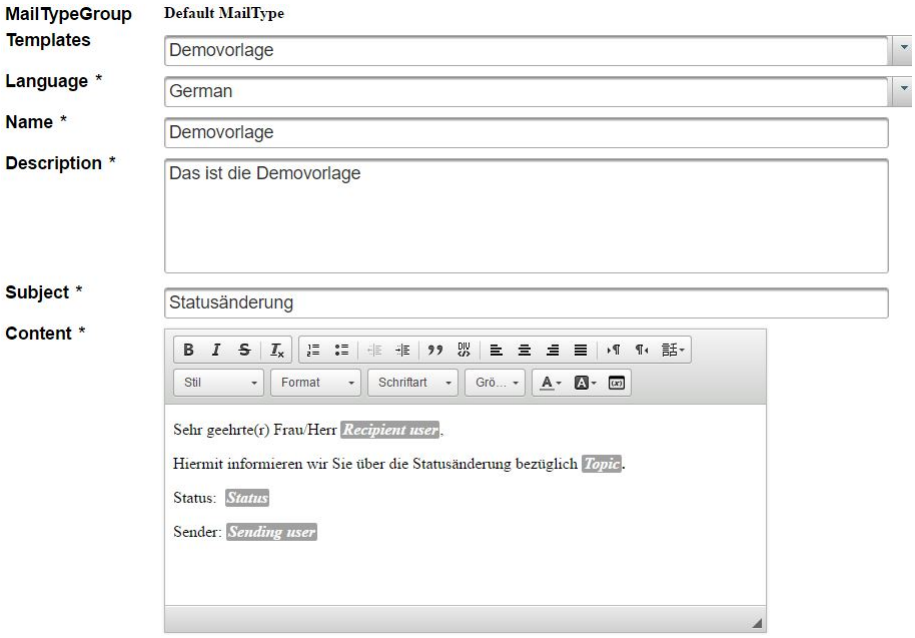


Abbildung 4.5: Formular für die Verwaltung der Vorlagen

Die Abbildung 4.6 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 4.6: Anzeige der aller relevanten Daten einer Vorlage

Die dekorierbare Vorlage

Die Abbildung 4.7 zeigt die dekorierbare *Freemarker*-Vorlage, die alle Vorlagen dekorieren. Sie stellt den *HTML-Body* zur Verfügung, da die Benutzervoralgen nur den Inhalt innerhalb des *HTML-Tags body* bereitstellen.

Abbildung 4.7: Das *Decorator Template*

Die Benutzervorlage

Die Abbildung 4.8 zeigt die *Freemarker*-Vorlage, die von der BenutzerInnen erstellt wird. Die Vorlage enthält zwar *HTML-Markup*, aber nur den Inhalt unterhalb des *HTML-Tags body*. Sie stellt aber kein vollständiges *HTML*-Dokument zur Verfügung.

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 4.8: Die *Benutzervorlage* als *Freemarker Template*

Die serialisierte *JSON*-Zeichenkette

Die Abbildung 4.9 zeigt die serialisierte *JSON*-Zeichenkette, die beim Erstellen einer *E-Mail* erstellt wird und in der Datenbank persistent gehalten wird. Mit diesen Daten kann eine *E-Mail* auf Basis dieser Vorlage jederzeit wiederhergestellt werden.

Template JSON Data

```
{
  "@type" : "template-data-json",
  "unknownProperties" : { },
  "template_metadata" : {
    "@type" : "template-metadata-json",
    "unknownProperties" : { },
    "id" : 1,
    "version" : 1,
    "locale" : "en_US",
    "zoneId" : "Europe/London",
    "variableCount" : 4
  },
  "data" : {
    "cc" : {
      "module" : {
        "di" : {
          "STATUS" : "Inactive",
          "RECIPIENT_USER" : "Hugo Maier",
          "TOPIC" : "User status changed",
          "SENDER_USER" : "Thomas Herzog"
        }
      }
    }
  }
}
```

Abbildung 4.9: Der serialisierte *JSON*-String

Die Vorlagenmetadaten

Die Abbildung 4.10 zeigt die Metadaten der Benutzervorlage. Die Metadaten sind nur für die Entwicklung relevant.

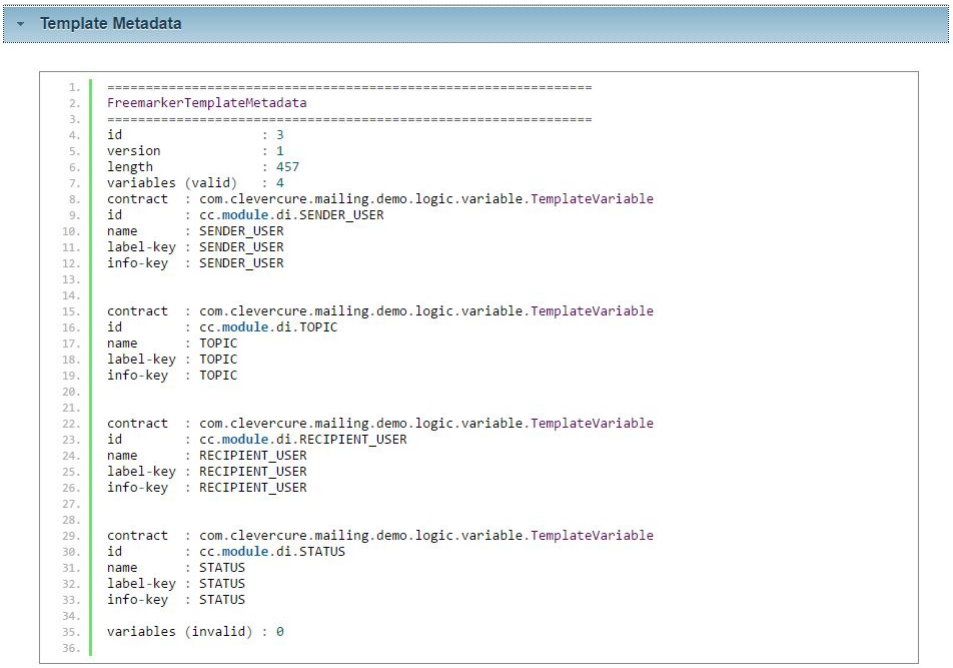


Abbildung 4.10: Die Metadaten der Vorlage

Der erstellte *E-Mail*-Inhalt

Die Vorlage mit den aufgelösten Variablen aus Abbildung 4.11 zeigt die Metadaten der Benutzervorlage.

Abbildung 4.11: Die Metadaten der Vorlage

Kapitel 5

Die Analyse und Tests

Dieses Kapitel beschäftigt sich mit der Analyse der Implementierung und dessen Tests. Es gibt zwei Arten von Tests die implementiert wurden

- die *JUnit*-Tests sind die Tests, die nicht auf eine *CDI*-Umgebung angewiesen sind und
- die *CDI-JUnit*-Tests sind die Tests, die auf eine *CDI*-Umgebung angewiesen sind.

5.1 Die Tests

Dieser Abschnitt beschäftigt sich mit den Implementieren Tests des Vorlagenmanagements und der Implementierten Konfiguration für die Tests. Für die Tests wurden folgende Bibliotheken verwendet.

- *JUnit4* ist ein *Framework*, mit dem wiederholbare Tests implementiert werden können und ist als Standard für Tests in *Java* anzusehen.
- *Deltaspike* ist ein *Open-Source*-Projekt der *Apache Software Foundation (ASF)*, die portable *CDI*-Erweiterungen in Form von Modulen bereitstellt und auch eine Erweiterung für *JUnit*-Tests bereitstellt, mit denen Tests in einer *CDI*-Umgebung lauffähig sind.

Alle implementierten Tests sind nicht auf einen Anwendungsserver angewiesen und sind innerhalb des lokalen Klassenpfades lauffähig und können daher in jeder Entwicklungsumgebung und bei einem Kompilieren über das *Buildtool Maven* ausführbar.

Die Tests wurden wie folgt organisiert.

- *com.clevercure.mailing.test.**
  ist das *Java*-Paket in dem alle implementierten Tests liegen.
- **.[toTestClass]Tests*
  ist das *Java*-Paket, für eine zu testende Klasse, wobei der Paketname

den Namen der zu testenden Klasse mit dem Suffix Tests enthält.

- *[toTestMethod]Test.java*
  ist die implementierte Testklasse für die Tests einer Methode der zu testenden Klasse.
- *test_case*
  ist der Name der einzelnen Testmethoden, der wiedergibt was an einer Methode getestet wird.

Die vorgestellte Konvention der Tests wurde so umgesetzt sofern es möglich war.

5.1.1 Die Tests der *CDI*-Erweiterung

Die Tests aus Abbildung 5.1 testen die Implementierungen des Artefakts *mailing-moule-template-cdi* wie

- die Klasse *TemplateCdiExtension*,
- die Klasse *CdiTemplateUtils* und
- die Klasse *TemplateResourceProducer*.

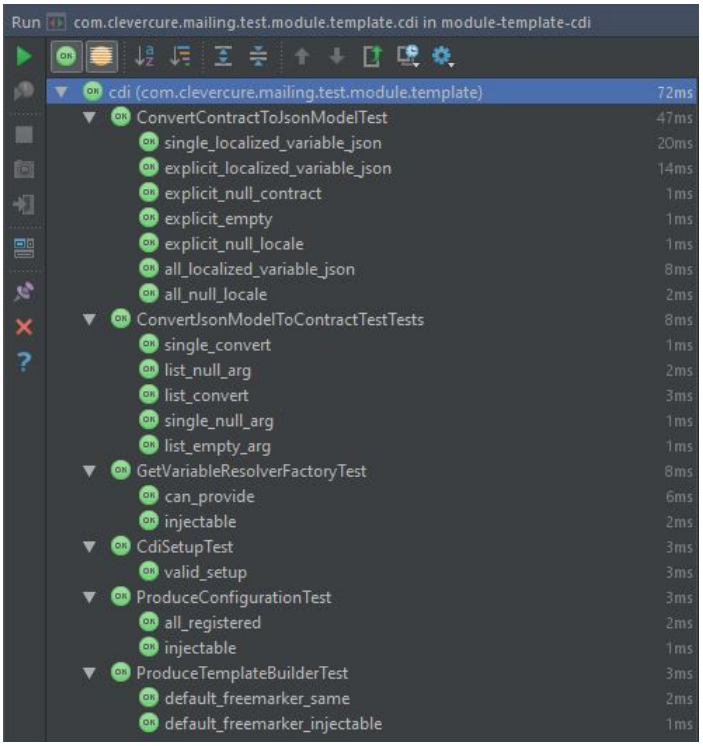


Abbildung 5.1: Testdurchlauf der Tests der *CDI*-Erweiterung

Diese Tests sind nur lauffähig in einer *CDI*-Umgebung, die aber Dank *Deltaspike* auch im Klassenpfad ohne Anwendungsserver gestartet werden kann. Im Klassenpfad der Tests wurde Variablen über eine *Enumeration*, die die Schnittstelle *VariableContract* implementiert, definiert, sowie eine Implementierung der Klasse *VariableResolverFactory*.

5.1.2 Die Tests des implementierten *FacesConverters*

Die Tests der *JSF*-Integration testen den implementierten *FacesConverter*, der die Voralgen von ihrer *HTML*-Repräsentation in die *Freemarker*-Repräsentation konvertiert.

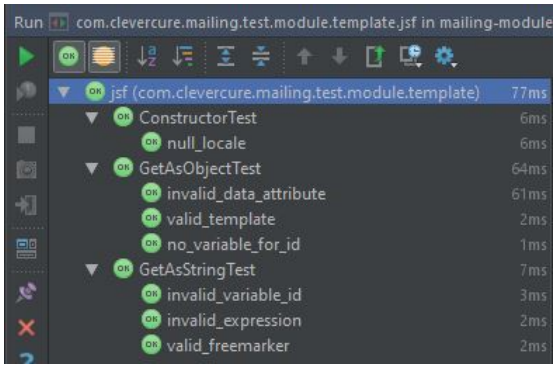


Abbildung 5.2: Testdurchlauf der Tests des *FacesConverters*

Obwohl die Schnittstelle *FacesConverter* aus *JSF* kommt, ist es nicht notwendig eine *JSF*-Umgebung zu starten.

5.1.3 Die Tests des implementierten Vorlagenmanagements

Die Tests aus Abbildung 5.3 testen die Implementierungen der Logik des Vorlagenmanagements. Die Logik ist in den beiden Klassen

- die Klasse *VariableConfigurationImpl* und
- die Klasse *FreemarkerTemplateDataJsonBuilder* implementiert.

Diese Tests sind nicht abhängig von einer *CDI*-Umgebung. Es wird getestet ob Variablen korrekt registriert werden und in einem Objekt der Klasse *VariableConfigurationImpl* korrekt verwaltet werden und ob die Klasse *FreemarkerTemplateDataJsonBuilder* in der Lage ist ein Objekt der Klasse *TemplateRequestJson* zu produzieren, dass die Daten für eine Voralge hält.

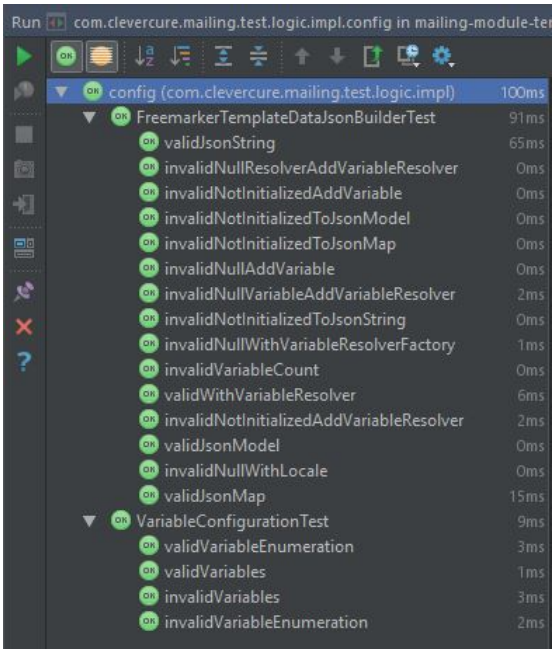


Abbildung 5.3: Testdurchlauf der Tests des Vorlagenmanagement

Diese Tests sind nicht angewiesen auf eine *CDI*-Umgebung und sind ohne zusätzliche Bibliotheken und *Framwork* lauffähig. Sie Testen die implementierten Methoden und vor allem bezüglich derer Fehlerbehandlung.

5.2 Die erreichten Ziele

Dieser Abschnitt beschäftigt sich mit der Betrachtung der erreichten Ziele des implementierten Vorlagenmanagements. Es wurden alle Anforderung aus dem Kapitel 2 erfüllt, somit gilt das Vorlagenmanagement als abgeschlossen. Die Integrationen in die Anwendungen *CleverWeb* und *CleverInterface* wurde noch nicht realisiert, obwohl begonnen wurde eine Integration für die Anwendung in *CleverWeb* zu implementieren. Die Integration in die Anwendung *CleverInterface* wird erst realisiert werden können, wenn diese Anwendung *Java* in Version 8 unterstützt. Zurzeit wird nur *Java* in version 7 unterstützt.

5.2.1 Das Vorlagenmanagement über das *CKEditor-Plugin*

Es wurde erfolgreich ein *Plugin* in *Typescript* für den *CKEditor* implementiert, dass das Variablenmanagement des Vorlagenmanagements erfolgreich *Browser* seitig in den *CKEditor* integriert. Wie in Abschnitt 3.2.1 behandelt, wurde das *Plugin* in *Typescript* umgesetzt und das *CKEditor-Plgun* von

dem Variablenmanagement getrennt und in eigenen Quelltextdateien implementiert. Die implementierten *Typescript*-Quelltexte befinden sich zurzeit noch in der Demowebanwendung, da die Entwicklung in einem eigenen Projekt nicht möglich war, da das *Hot-Code-Deployment* für *Java*-Ressourcen (src/main/resources) nicht unterstützt wird. Diese Quelltextdateien können einfach in ein anderes Projekt verschoben werden. Die Quelltextdateien werden jetzt noch über die Entwicklungsumgebung kompiliert, kann aber in Zukunft über das *Maven-Plugin maven-grunt-plugin* auch automatisiert bei jedem *Build* kompiliert werden.

5.2.2 Das Vorlagenmanagements in *CDI*

Es wurde erfolgreich die Integration des Vorlagenmanagement in eine *CDI*-Umgebung implementiert. Die in Abschnitt 4.1.2 behandelte Integration in eine *CDI*-Umgebung, wurde über eine portierbare *CDI*-Erweiterung realisiert, die beim Start der *CDI*-Umgebung alle implementierten Klassen der Schnittstellen *VariableContract (enum)* und *VariablenResolver* findet, registriert und über die Anwendungsdauer verwaltet. Diese gesammelten Ressourcen werden kontextabhängig zur Verfügung gestellt und können über Injektion in ein Objekt injiziert werden.

5.2.3 Das Vorlagenmanagement in *JSF*

Es wurde erfolgreich eine Integration in *JSF* implementiert, wobei diese Integration über einen *FacesConverter* erreicht wurde, der die Vorlagen von ihrer *HTML*-Repräsentation in die *Freemarker*-Repräsentation konvertieren kann. Wie in Abschnitt 4.1.3 vorgestellt, wurde die gemeinsame Logik in einer abstrakte Klasse *AbstractTemplateConverter* gekapselt, der nur bekanntgegeben werden muss, welche konkrete Implementierung, definiert über eine Qualifizierer Annotation, genutzt werden soll. Über diese abstrakte Klasse wird sichergestellt, dass die *HTML*-Repräsentation der Variablen über alle *Template-Engines* gleich ist.

5.2.4 Das Vorlagenmanagement in *Mail*-DB-Schema

Die Integration der Vorlagen in des *Mail-DB*-Schema war die die einfachste Aufgabe, da hier lediglich eine einfache Datenstruktur definiert werden muss, die in der Lage ist, die Vorlagen mehrsprachig persistent zu halten. Prinzipiell ist eine Vorlage auf einer Datenbank als Zeichenkette präsent, wobei nur auf die Größe der Zeichenkette geachtet werden muss.

Kapitel 6

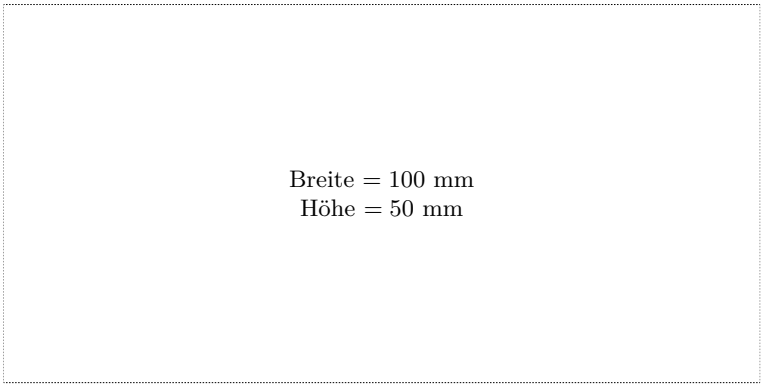
Die Zusammenfassung

TODO: Add sumary about development, experiences and further work

Quellenverzeichnis

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —