Workshop Java EE 7

Ein praktischer Einstieg in die Java Enterprise Edition mit dem Web Profile

von
Marcus Schießer, Martin Schmollinger

2., akt. u. erw. Aufl.

dpunkt.verlag 2014

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 195 9

Zu Inhaltsverzeichnis

Enge und lose Kopplung von Komponenten

Anwendungskomponenten bestehen aus einer Menge von Klassen. Eine Klasse einer Komponente A kann eine Klasse einer anderen Komponente B direkt importieren. Dadurch entsteht eine starke Abhängigkeit von Komponente A zu B, da sie nur verwendet werden kann, wenn Komponente B im Klassenpfad enthalten ist. Man spricht von einer engen Kopplung mit der Folge, dass Änderungen an Komponente B oft Änderungen an Komponente A bedingen.

Führt Komponente B eine Schnittstelle ein, über die Komponente A ausschließlich auf die Funktionalität von B zugreift, so ist die Abhängigkeit der Komponenten geringer. Bei einer Änderung der Komponente B muss diese lediglich dafür sorgen, dass sich die Schnittstelle nicht ändert. Komponente A muss dann nicht geändert werden. Man spricht von einer losen Kopplung der Komponenten.

Die Abhängigkeit von Komponenten kann noch weiter verringert werden, indem diese lediglich wohldefinierte Nachrichten untereinander senden und empfangen. Wird eine Komponente aktualisiert, so muss das Format der Nachricht abwärtskompatibel sein, sodass keine Änderungen an der empfangenden Komponente notwendig sind.

Mehr über lose und enge Kopplung finden Sie in (Siedersleben, 2004).

Nach unserer ersten JSF-Iteration sind die Abhängigkeiten der Klassen noch nicht so groß wie in einer gewöhnlichen Java-Anwendung, da wir bereits (ohne es zu wissen) CDI verwenden. Ein schönes Beispiel für eine ungewünschte Kopplung ist allerdings, dass der `EditCampaignController` in der Methode `doSave` direkt auf die Liste der `Campaign`-Objekte des `CampaignListProducer` zugreift.

Besser wäre es hier, eine lose Kopplung beider Komponenten einzuführen. Im konkreten Fall könnte die Methode `doSave` die Nachricht an den `CampaignListProducer` senden, dass ein `Campaign`-Objekt hinzugefügt werden soll. Der `CampaignListProducer` könnte dann selbstständig entscheiden, wie er auf diese Nachricht reagieren möchte. Wenn später eine andere, noch zu entwickelnde Komponente auf diese Nachricht reagieren möchte, so ist dies einfach möglich, ohne die bestehenden sendenden Komponenten anzupassen. Dies sichert die zukünftige Erweiterbarkeit und Wartbarkeit der Anwendung.

In der Praxis finden sich bei großen Anwendungen Hunderte solcher Beispiele, die sich einfacher und eleganter mit CDI lösen lassen.

Sie sind immer noch nicht überzeugt? Schauen Sie sich einfach die nachfolgenden Abschnitte an und stellen Sie sich am besten dabei vor, dass unsere Anwendung aus Hunderten von Anwendungsfällen bestehen würde.

6.2 Der Laufzeit-Container

Bei einer normalen Java-Anwendung muss der Programmierer selbstständig dafür sorgen, dass Klassen instanziiert werden. Dies ist etwas anders mit CDI – dessen Hauptbestandteil ein sogenannter Container ist, der selbstständig Instanzen von Klassen erzeugt und diese auch wieder aus dem Speicher entfernt. Das

Objektdiagramm in Abbildung 6–1 veranschaulicht die Beziehung zwischen einem Container und den von ihm verwalteten Instanzen. In diesem Fall wird ein Objektdiagramm anstatt eines Klassendiagramms verwendet, da Instanzen anstelle von Klassen betrachtet werden.

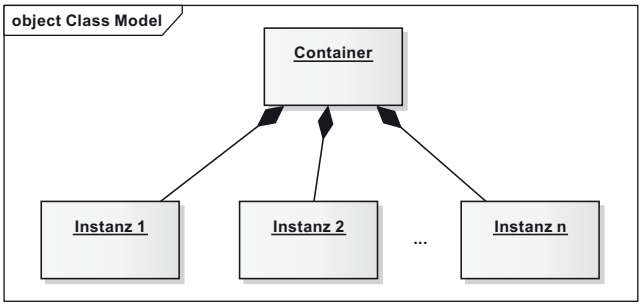

Abb. 6–1 *Objektdiagramm eines Containers und der von ihm verwalteten Instanzen*

Der Container von CDI kann, bis auf wenige Ausnahmen, Instanzen jeder konkreten Klasse verwalten. Dazu gehören insbesondere JavaBeans und auch EJB Session Beans (siehe Kap. 8). Die von CDI verwalteten Klassen heißen allgemein Beans. Damit CDI diese als Beans verwaltet, muss die Bean-Klasse lediglich mit einer Annotation versehen werden[1].

Instanzen der Beans besitzen übrigens keine Referenz auf den Container, jedoch umgekehrt der Container eine auf sie. In der Literatur ist dieses Konzept auch als Inversion of Control (siehe Abschnitt 1.2.2) bekannt.

Bei einem Container stellen sich insbesondere die Fragen, wann die Instanzen erstellt und gelöscht werden und wie man die Instanzen referenziert. Diese und weitere Fragen werden in den folgenden Abschnitten behandelt.

6.2.1 Der Sichtbarkeitsbereich (Scope) einer Bean

Der Zeitpunkt, wann die Beans erstellt und gelöscht werden, hängt vom Sichtbarkeitsbereich (engl. *scope*) der Beans ab. CDI ermöglicht es auch, eigene Scopes zu erzeugen; für uns von Bedeutung sind allerdings zunächst nur der *RequestScope*, der *SessionScope,* der *ViewScope* und der *DependentScope*.

6.2.1.1 RequestScope

Eine Bean mit *RequestScope* wird für jede eingehende Anfrage (engl. *request*) neu erzeugt. Wird die Anfrage beendet, wird die Bean wieder gelöscht. Genau genommen wird sie zunächst vom Container nur zum Löschen markiert, das eigentliche Löschen übernimmt dann zu einem späteren Zeitpunkt der Garbage Collector.

1. Vor CDI 1.1 musste zur Aktivierung in dem Anwendungsarchiv noch zusätzlich eine leere Datei mit dem Namen `WEB-INF\beans.xml` enthalten sein.

Gehen also mehrere Anfragen gleichzeitig an den Server ein, so wird für jede Anfrage eine eigene Bean erzeugt. Daten, die in der Bean gespeichert sind, gehen nach dem Ende der Anfrage verloren. Da eine Anfrage üblicherweise eine sehr kleine Lebenszeit hat (kleiner als eine Sekunde), sind Beans mit *RequestScope* ebenfalls sehr kurzlebig. Sie eignen sich daher nicht, um Daten zu speichern, die für eine längere Zeit benötigt werden, wie beispielsweise Benutzerdaten. Dafür binden diese Beans nicht dauerhaft Ressourcen und gehen daher sparsam mit diesen um.

Natürlich instanziiert der Container eine Bean mit *RequestScope* in einer Anfrage nur dann, wenn sie auch in der Anfrage referenziert wird, sonst würden unnötig Ressourcen verbraucht werden.

Um zu definieren, dass eine Bean den Sichtbarkeitsbereich *RequestScope* hat, muss man der Bean-Klasse die Annotation `@RequestScoped` aus dem Paket `javax.enterprise.context` hinzufügen.

6.2.1.2 SessionScope

Eine Bean mit *SessionScope* wird für jede Benutzersitzung neu erzeugt. Erst wenn die Benutzersitzung beendet wird oder wenn sie nach einer längeren Inaktivität in einen Timeout läuft, wird die Bean wieder gelöscht.

Dies bedeutet, dass jeder Benutzer seine eigene Instanz von dieser Bean besitzt. Beans mit *SessionScope* sind daher ideal dafür geeignet, Benutzerdaten zu speichern. Ein gutes Beispiel in unserem Fall ist der `CampaignListProducer`, der die Liste aller `Campaign`-Objekte des Benutzers enthält.

Da eine Benutzersitzung sehr lange dauern kann (mehrere Stunden), ist es möglich, dass der Server eine inaktive Benutzersitzung aus Ressourcengründen zwischenspeichern muss. Daher müssen Beans mit *SessionScope* serialisierbar sein (das heißt, sie müssen die Schnittstelle `Serializable` aus dem Paket `java.io` implementieren). Der Server kann dann bei Bedarf eine solche Bean serialisieren, auf Festplatte speichern und bei Bedarf wieder deserialisieren. Beans mit *SessionScope* benötigen daher mehr Ressourcen als Beans mit *RequestScope*. Nur wenn es nötig ist, sollte eine Bean deshalb einen *SessionScope* bekommen.

Um zu definieren, dass eine Bean den *SessionScope* erhält, muss man der Bean-Klasse die Annotation `@SessionScoped` aus dem Paket `javax.enterprise.context` hinzufügen.

6.2.1.3 ViewScope

Für eine bessere Ressourcennutzung wurde mit Java EE 7 ein weiterer Scope für CDI-Beans, der sogenannte *ViewScope*[2], eingeführt. Dieser ist ausschließlich für Anwendungen von Interesse, die JavaServer Faces nutzen[3]. Beachten Sie, dass die-

2. Der ViewScope existierte in JSF eigentlich schon vor Version 2.2, jedoch nur für die JSF-eigenen Managed Beans, nicht aber für CDI-Beans.

ser Scope daher, obwohl er ein CDI-Scope ist, mit JSF 2.2 und nicht mit CDI 1.1 ausgeliefert wird.

Die Lebensdauer der Beans hängt bei diesem Scope von den aktuellen View-Komponenten ab. Wird in einer JSF-View eine Bean neu referenziert, dann wird eine neue Instanz der Bean angelegt. Wenn die View-Komponenten gelöscht werden, dann entfernt im Gegenzug der CDI-Container auch die Beans, die innerhalb der View initialisiert wurden und den *ViewScope* besitzen. Eine View existiert normalerweise über mehrere Anfragen hinweg. Daher hat eine Bean mit dem *ViewScope* eine höhere Lebensdauer als der *RequestScope*.

Interessant ist dies insbesondere für Beans, die zu validierende Daten speichern, da diese länger als eine Anfrage überleben müssen. Für solche Beans wäre es aber eine Verschwendung von Ressourcen, diese in der Benutzersitzung über den *SessionScope* zu speichern.

Um zu definieren, dass eine Bean den *ViewScope* erhält, muss man der Bean-Klasse die Annotation `@ViewScoped` aus dem Paket `javax.faces.view` hinzufügen. An dem abweichenden Paketnamen wird weiterhin deutlich, dass dieser Scope nicht zu CDI gehört.

6.2.1.4 DependentScope

Dieser Scope bedeutet schlicht, dass die Lebensdauer der Bean abhängig von einer anderen Bean ist und daher nicht unabhängig von einer anderen Bean existieren kann. Eine Bean mit *DependentScope* wird erzeugt, wenn die Bean, von der sie abhängig ist, erzeugt wird, und sie wird gelöscht, wenn diese gelöscht wird.

Der *DependentScope* wird über die Annotation `@Dependent` aus dem Paket `javax.enterprise.context` festgelegt.

6.2.1.5 Setzen der Scopes für unsere Anwendung

Zum besseren Verständnis hatten wir in unserer JSF-Iteration den Scope für alle Beans auf *SessionScope* gesetzt. Dadurch ist unsere Anwendung wie gewünscht lauffähig, die Ressourcennutzung ist jedoch nicht optimal.

Um die Ressourcennutzung zu verbessern, profitieren wir von der klaren Struktur, die wir in den vorherigen Kapiteln aufgebaut haben:

Daten, die länger als die Lebensdauer einer JSF-View überleben müssen, haben wir in die Klassen `CampaignListProducer` und `CampaignProducer` ausgelagert. Diese müssen daher weiterhin im *SessionScope* bleiben.

In den Controller-Beans, deren eigentliche Aufgabe die Steuerung des Kontrollflusses ist, speichern wir in einigen Fällen noch Daten, die über die Lebensdauer einer View erhalten bleiben müssen. Beispiele hierfür sind das zu löschende

---

3. *SessionScope* und *RequestScope* benötigen hingegen kein JSF – sie funktionieren mit jeder Webanwendung, die lediglich auf Servlets aufbaut.

`Campaign`-Objekt in der Klasse `ListCampaignsController` und die Spende, die im `DonateMoneyController` neu angelegt wird.

Da diese Daten über mehrere Anfragen erhalten bleiben müssen, ist der *RequestScope* für diese Beans nicht ausreichend. Der *ViewScope* hingegen ist ideal, da es sich lediglich um Daten handelt, die lokal in der aktuellen View benötigt werden.

Dies ist eine bewusste Designentscheidung: Die Controller-Beans dürfen lediglich lokale Daten zwischenspeichern.

Der aufmerksame Leser wird beobachten, dass nicht jede Controller-Bean einen *ViewScope* benötigt. So kommt der `ListDonationsController` lediglich mit einem *RequestScope* aus, da er überhaupt keine Daten speichert.

Da es sich jedoch um eine Designentscheidung handelt, dass Daten gespeichert werden dürfen, kann sich dieser Zustand in der Zukunft für jede Bean ändern. Konsequenterweise setzen wir daher nun den Scope aller Controller-Beans auf den *ViewScope*. Zwar hat dies den Nachteil, dass die Ressourcennutzung nicht optimal ist, der Scope aller Controller-Beans ist dadurch jedoch einheitlich.

Aus der Sicht der Autoren führt dies zu einem klareren Design und vermeidet für die zukünftige Lebensdauer der Anwendung verwirrende Fragen und Diskussionen der Art, warum das eine Controller-Bean den *ViewScope* besitzt und ein anderes lediglich den *RequestScope*.

6.2.2 Beans referenzieren über Dependency Injection

Bisher haben wir geklärt, wie der Container eine Bean instanziiert und diese wieder löscht. Ungeklärt ist bis jetzt, wie eine Bean referenziert wird. Dies geschieht über die Annotation `@Inject` aus dem Paket `javax.inject`. Betrachten Sie hierzu den folgenden Ausschnitt aus dem `EditCampaignController`:

```
@Inject
private CampaignListProducer campaignListProducer;
```

Das Attribut `campaignListProducer` soll also eine Instanz der Bean `CampaignListProducer` speichern. Ohne die Annotation `@Inject` hätte das Attribut einen undefinierten Wert. Diese Annotation ist jedoch für den Container der Hinweis, dass nach der Instanziierung der Bean (in diesem Fall `EditCampaignController`) in dem annotierten Attribut eine Instanz des Typs (in diesem Fall `CampaignListProducer`) referenziert werden soll.

Durch diese Annotation wird daher die Referenz einer Bean in einer anderen Bean zur Verfügung gestellt. Dieser Vorgang wird im Englischen Dependency Injection genannt. Die Beziehungen der beteiligten Klassen unseres Beispiels sind in dem Klassendiagramm in Abbildung 6–2 dargestellt.

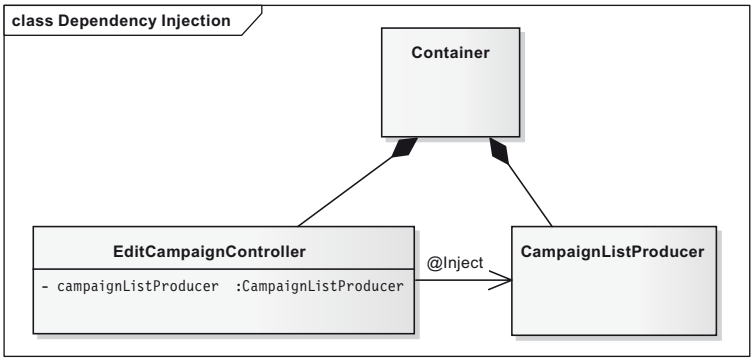

Abb. 6–2 *Beziehungen zwischen dem Container und den Beans EditCampaignController und CampaignListProducer*

Unklar ist noch, welche Instanz von `CampaignListProducer` zur Laufzeit genommen wird. Dies ist jedoch eindeutig, da die Bean `EditCampaignController` durch den *ViewScope* an die aktuelle JSF-View gebunden ist. Diese wiederum ist genau einer Sitzung zugeordnet – der des aktuellen Benutzers. Da sich weiterhin die Bean `CampaignListProducer` im *SessionScope* befindet, existiert pro Benutzersitzung exakt eine Instanz dieser Bean. Eben diese wird in unserem Beispiel referenziert.

Was ist jedoch, wenn der Typ der Instanzvariablen eine Superklasse oder eine Schnittstelle ist, die von mehreren Beans implementiert wird? Mit den aktuellen Kenntnissen kann der Container die Bean nicht eindeutig auflösen. Dieses Problem lässt sich mit sogenannten Qualifiern beheben und wird in Abschnitt 6.4.2 behandelt.

Die Annotation `@Inject` kann übrigens nicht nur bei Attributen, sondern auch bei Konstruktoren oder Methoden verwendet werden. Dies wird zwar in unserer Beispielanwendung nicht benötigt, der Vollständigkeit halber wird nachfolgend aber kurz dargestellt, wie dies aussehen würde.

Wird der Konstruktor mit `@Inject` annotiert, dann können andere Beans als Parameter übergeben werden. Der Container übergibt dann bei Instanziierung der Bean die anderen Beans. Bei unserem Beispiel sähe dies folgendermaßen aus:

```
private CampaignListProducer campaignListProducer;

@Inject
public EditCampaignController(CampaignListProducer campaignListProducer) {
   this.campaignListProducer = campaignListProducer;
}
```

Ebenso ist es möglich, Methoden mit `@Inject` zu annotieren. Der Container ruft nach der Instanziierung der Bean diese Methoden auf und übergibt die Beans, die in den Methodenparametern angegeben sind. Hier der Code für unser Beispiel:

```
private CampaignListProducer campaignListProducer;

@Inject
public void setCampaignListProducer(CampaignListProducer
campaignListProducer) {
   this.campaignListProducer = campaignListProducer;
}
```

6.2.3 Der Lebenszyklus

Wie bereits erwähnt werden Beans vom Container selbstständig instanziiert und auch wieder gelöscht. Der Programmierer kann Methoden definieren, die vom Container zu bestimmten Zeitpunkten dieses Lebenszyklus aufgerufen werden. Im Folgenden stellen wir die beiden Lebenszyklusmethoden von CDI vor, die man über die Annotationen `@PostConstruct` und `@PreDestroy` markiert.

6.2.3.1 Lebenszyklusmethode PostConstruct

Annotiert der Programmierer eine beliebige Methode der Bean mit der Annotation `@PostConstruct` aus dem Paket `javax.annotation`, dann wird diese Methode direkt nach der Instanziierung der Bean von dem Container aufgerufen.

Im Gegensatz zum Konstruktor der Bean sind zum Zeitpunkt des Aufrufs der `PostConstruct`-Methode bereits alle Abhängigkeiten zu anderen Beans aufgelöst, das heißt, die mit `@Inject` annotierten Attribute enthalten schon die Referenzen auf die Beans, und die mit `@Inject` annotierten Methoden wurden ebenfalls bereits aufgerufen.

Beispielhaft ersetzen wir nun in den Beans `DonateMoneyController` und `CampaignListProducer` den Konstruktor durch eine Initialisierungsmethode `init`, die mit `@PostConstruct` annotiert ist.

Für den `DonateMoneyController` bedeutet dies, dass

```
public DonateMoneyController() {
   this.donation = new Donation();
}
```

durch

```
@PostConstruct
public void init() {
   this.donation = new Donation();
}
```

ersetzt wird. Dies hat konkret den Vorteil, dass die Methode `init` im Gegensatz zum Konstruktor erneut aufgerufen werden kann, wodurch die Bean reinitialisiert wird. Daher kann in der Methode `doDonation` die Zeile

```
this.donation = new Donation();
```

durch

```
init();
```

ausgetauscht werden.

Für den `CampaignListProducer` wird nun ebenfalls der Konstruktor ersetzt. Hierbei ist

```
public CampaignListProducer() {
   campaigns = createMockCampaigns();
}
```

durch

```
@PostConstruct
public void init() {
   campaigns = createMockCampaigns();
}
```

zu ersetzen.

Aktuell ist dies noch ohne Vorteil, in einer späteren Iteration werden wir jedoch in der Methode `init` eine weitere Bean aufrufen, die wir über `@Inject` einbinden und die daher im Konstruktor nicht zur Verfügung stehen würde. Wir schauen an dieser Stelle also ein wenig in die Zukunft.

6.2.3.2 Lebenszyklusmethode PreDestroy

Methoden einer Bean, die man mit der Annotation `@PreDestroy` aus dem Paket `javax.annotation` versieht, werden vom Container aufgerufen, bevor die Bean aus dem Speicher gelöscht wird. Hierdurch kann der Entwickler noch Ressourcen freigeben, die die Bean angelegt hat, beispielsweise temporär angelegte Dateien löschen.

In unserer Beispielanwendung wird diese Funktionalität nicht benötigt, daher sind keine Änderungen an dieser Stelle vorzunehmen.

6.2.4 Beliebige Klassen als Beans mit Producer-Methoden

Bisher konnten wir nur Klassen als Beans verwenden, die entweder einen parameterlosen oder einen mit `@Inject` annotierten Konstruktor besitzen. Durch diese Konvention kann der Container selbstständig Instanzen dieser Beans erstellen. Für Klassen mit beliebigem Konstruktor ist dies nach dem aktuellen Kenntnisstand jedoch nicht möglich.

Um beliebige Klassen verwenden zu können, stellt CDI spezielle Methoden zur Verfügung, die mit der Annotation `@Produces` aus dem Paket `javax.enterprise.inject` versehen werden. Solche Producer-Methoden erstellen Instanzen beliebiger Klassen, die dadurch als Bean innerhalb des Containers zur Verfügung stehen und über die Annotation `@Inject` wie in Abschnitt 6.2.2 besprochen verknüpft werden können.

Diese Funktionalität verwenden wir nun in der Klasse `Resources`, die eine Reihe von Producer-Methoden besitzt, die der Anwendung Ressourcen als Beans zur Verfügung stellen. Listing 6–1 enthält den Quelltext der Klasse, die im Paket `de.dpunkt.myaktion.util` gespeichert werden muss.

```
package de.dpunkt.myaktion.util;

import javax.enterprise.context.Dependent;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Produces;
import javax.enterprise.inject.spi.InjectionPoint;
import javax.faces.context.FacesContext;
import java.util.logging.Logger;

@Dependent
public class Resources {
   @Produces
   public Logger produceLog() {
      return Logger.getLogger("MyLogger", "messages");
   }

   @Produces
   @RequestScoped
   public FacesContext produceFacesContext() {
      return FacesContext.getCurrentInstance();
   }

}
```

Listing 6–1 *Klasse Resources*

Die Klasse `Resources` stellt einen Logger mit dem Namen »MyLogger" und den aktuellen `FacesContext` zur Verfügung. Beide Ressourcen kann man über die Annotation `@Inject` in anderen Beans verwenden.

Bei einem Logger handelt es sich um eine Hilfsklasse, die Ausgaben in eine Log-Datei protokolliert. Der Logger verwendet für die Internationalisierung der Log-Ausgaben das in Kapitel 4 definierte Message-Bundle `messages`. Dies bedeutet, dass Nachrichten, die über den Logger ausgegeben werden, in den Dateien `messages_de.properties` und `messages_en.properties` definiert werden müssen. Dazu in Kürze mehr.

Die Bean `DonateMoneyController` profitiert direkt von dem Logger und dem `FacesContext`. Daher werden diese zunächst in dieser Bean als weitere Attribute eingefügt:

```
@Inject
private FacesContext facesContext;

@Inject
private Logger logger;
```

In der Methode doDonation der Bean können diese Ressourcen dann verwendet werden. Zunächst kann man nun die Zeile

```
final FacesContext facesContext = FacesContext.getCurrentInstance();
```

löschen, wodurch in der Methode stattdessen die injizierte Instanzvariable facesContext referenziert wird.

Weiterhin ist es an dieser Stelle sinnvoll, eine Log-Meldung über die erfolgreiche Spende auszugeben. Hierbei kommt der Logger zum Einsatz. Die Methode doDonation sieht durch diese Änderungen nun insgesamt so aus:

```
public String doDonation() {
   logger.log(Level.INFO, "log.donateMoney.thank_you",
      new Object[]{getDonation().getDonorName(),
      getDonation().getAmount()});
   final ResourceBundle resourceBundle =
facesContext.getApplication().getResourceBundle(facesContext, "msg");
   final String msg = resourceBundle.getString("donateMoney.thank_you");
   facesContext.addMessage(
         null,
         new FacesMessage(FacesMessage.SEVERITY_INFO, msg, null));
   init();
   return Pages.DONATE_MONEY;
}
```

Die auszugebende Log-Meldung wird über den Schlüssel log.donateMoney.thank_you des Message-Bundles messages aufgelöst. Dieses müssen wir nun um den Ausgabetext der beiden Zielsprachen Deutsch und Englisch erweitern.

Für Deutsch fügen Sie hierzu der Datei messages_de.properties am Ende folgende Zeile hinzu:

```
log.donateMoney.thank_you={0} hat {1} Euro gespendet.
```

Für Englisch erweitern Sie die Datei messages_en.properties um diesen Eintrag:

```
log.donateMoney.thank_you={0} has donated {1} Euro.
```

Unschön ist, dass der Logger den Namen »MyLogger« hat anstatt, wie üblich, den Namen der Klasse, die den Logger aufruft.

Um dem Logger den Namen der aufrufenden Klasse zu geben, kann man der Producer-Methode für die Ressource Logger Laufzeitinformationen über die verknüpfte Bean-Klasse mitteilen. Hierzu wird der Methode produceLog der Klasse Resources als Parameter der Typ InjectionPoint aus dem Paket javax.enterprise.inject.spi übergeben:

```
@Produces
public Logger produceLog(InjectionPoint injectionPoint) {
   return Logger.getLogger(injectionPoint.getMember().getDeclaringClass()
         .getName(), "messages");
}
```

Der Ausdruck `injectionPoint.getMember().getDeclaringClass()` gibt dabei das `Class`-Objekt der aufrufenden Klasse zurück, die den Logger über die Annotation `@Inject` einbindet. Wird der Logger nun in der Bean `DonateMoneyController` verwendet, so hat er den Namen `DonateMoneyController` oder generell den Namen der aufrufenden Bean.

Da es sich bei den von den Producer-Methoden erzeugten Instanzen um Beans handelt, haben diese auch einen Scope. Standardmäßig ist dies der Scope der Klasse, in unserem Fall durch die Annotation `@Dependent` der *DependentScope* (siehe Abschnitt 6.2.1.4). Auf Wunsch kann dieser jedoch geändert werden, indem die Producer-Methode mit dem gewünschten Scope annotiert wird.

In unserem Fall bleibt der Logger auf *DependentScope*, sodass für jede aufrufende Bean eine eigene Instanz des Loggers erstellt wird. Die Producer-Methode für den `FacesContext` wird jedoch auf *RequestScope* gesetzt, da sich dieses Objekt innerhalb einer Anfrage nicht ändert.

Java EE 7 stellt weiterhin intern eine Producer-Methode für den `HttpServletRequest` zur Verfügung. Diese möchten wir an dieser Stelle nutzen, indem wir die Methode `getAppUrl` in dem `EditDonationFormController` durch folgenden Codeabschnitt ersetzen:

```
@Inject
private HttpServletRequest req;

private String getAppUrl() {
   String scheme = req.getScheme();
   String serverName = req.getServerName();
   int serverPort = req.getServerPort();
   String contextPath = req.getContextPath();

   return scheme+"://"+serverName+":"+serverPort+contextPath;
}
```

Durch diese Änderung wird der `HttpServletRequest` der aktuellen Anfrage in den Controller injiziert und direkt in der Methode `getAppUrl` genutzt. Zuvor war es bei jedem Aufruf der Methode notwendig, diesen über den `FacesContext` abzufragen.

Es gibt noch zwei weitere Stellen in unserer Anwendung, die von Producer-Methoden profitieren können: Es handelt sich um die Beans `CampaignProducer` und `CampaignListProducer`.

Die Aufgabe des `CampaignListProducer` besteht darin, eine Liste von `Campaign`-Objekten zur Verfügung zu stellen. Eine Klasse, die diese Liste verwenden möchte, benötigt eigentlich keine Abhängigkeit zu dem `CampaignListProducer`, sondern lediglich zu dieser Liste. Damit dies möglich ist, annotieren wir nun die Methode `getCampaigns` des `CampaignListProducer` mit `@Produces`. Da anwendungsweit keine andere Bean eine Liste von `Campaign`-Objekten zur Verfügung stellt, kann diese Liste nun folgendermaßen in andere Beans injiziert werden:

```
@Inject
private List<Campaign> campaigns;
```

Auf diesbezügliche Änderungen wollen wir aber verzichten, da wir im folgenden Abschnitt noch eine weitere Methode kennenlernen, um Abhängigkeiten zum `CampaignListProducer` zu vermeiden: anwendungsweite Nachrichten.

Wir wollen die neue Producer-Methode `getCampaigns` aber dazu nutzen, um sie in Facelets zu referenzieren. Hierzu annotieren wir die Methode zusätzlich noch mit `@Named` und entfernen dieselbe Annotation auf Klassenebene.

Dadurch kann die DataTable-Komponente der Datei `listCampaigns.xhtml` direkt auf die Liste der `Campaign`-Objekte über den Namen `campaigns` zugreifen und benötigt nicht mehr den Umweg über den `CampaignListProducer`. Um dies umzusetzen, ändern Sie folgende Zeile der Datei `listCampaigns.xhtml`:

```
<p:dataTable value="#{campaignListProducer.campaigns}" var="campaign">
```

in nachstehenden Ausdruck um:

```
<p:dataTable value="#{campaigns}" var="campaign">
```

Als Folge enthält die Klasse `CampaignListProducer` nun eine Producer-Methode, die eine Liste von `Campaign`-Objekten zur Verfügung stellt und damit ihrem Namen alle Ehre macht.

Analog hierzu möchten wir nun ähnliche Producer-Methoden auch für die Bean `CampaignProducer` einführen. Hierzu entfernen wir zunächst die `@Named`-Annotation der Bean auf Klassenebene und fügen den Getter-Methoden `getSelectedCampaign` und `isAddMode` die Annotation `@Named` und `@Produces` hinzu.

Für Letztere ergeben sich dadurch die folgenden Definitionen:

```
@Produces
@Named
public Campaign getSelectedCampaign() {
   return campaign;
}
```

und

```
@Produces
@Named
public boolean isAddMode() {
   return mode == Mode.ADD;
}
```

Nach dieser Änderung können Sie die Producer-Methoden direkt in den Facelets verwenden. Hierzu müssen Sie alle Referenzen der Getter in den Dateien `editCampaign.xhtml` und `listDonations.xhtml` ersetzen. Betrachten Sie dabei lediglich die Ausdrücke der EL (*Expression Language*), also alles innerhalb von `#{` und `}`.

Konkret müssen Sie in den Facelets den Ausdruck `campaignProducer.selectedCampaign` durch `selectedCampaign` und `campaignProducer.addMode` durch `addMode` ersetzen. Da es sich um einige Referenzen handelt, verwenden Sie dabei am besten die »*Suchen und Ersetzen*«-Funktion Ihres Editors. Seien Sie dabei aber vor-

sichtig und testen Sie anschließend Ihre Änderungen, um spätere Probleme zu vermeiden.

6.3 Anwendungsweite Nachrichten

Bisher haben wir hauptsächlich CDI-Funktionen erklärt, die in der vorherigen Iteration eingeführt wurden (abgesehen von den Producer-Methoden und der Annotation `@PostConstruct`).

CDI kann jedoch nicht nur Beans, die von einem Container verwaltet werden, miteinander verknüpfen. Das Prinzip der losen Kopplung von Komponenten bei CDI geht noch einen Schritt weiter. Komponenten können sich während der Laufzeit beliebige Nachrichten senden und diese empfangen. Die Komponenten haben dabei keine Abhängigkeit mehr untereinander, sondern nur noch jeweils eine Abhängigkeit zu der Klasse, die die Nachricht an sich enthält. Durch diese Art der losen Kopplung können neue Komponenten, die Nachrichten empfangen, hinzugefügt werden, ohne dass die sendende Komponente angepasst werden muss. Die Erweiterbarkeit und Wartbarkeit der Anwendung wird dadurch weiter verbessert.

6.3.1 Events senden und empfangen

Wie in Abschnitt 6.1 erläutert, besteht eine störende Abhängigkeit der Bean `EditCampaignController` zur Bean `CampaignListProducer`. Diese kann mit CDI einfach entfernt werden, indem der `EditCampaignController` eine Nachricht sendet, wenn eine Aktion hinzugefügt wird. Der `CampaignListProducer` kann dann auf diese Nachricht reagieren und, da er die Liste aller `Campaign`-Objekte verwaltet, das neue `Campaign`-Objekt zu dieser Liste hinzufügen.

Verändern wir zunächst den `EditCampaignController` so, dass dieser beim Hinzufügen einer Aktion eine entsprechende Nachricht sendet. Zunächst ist entscheidend, welche Nachricht gesendet werden soll: Da ein `Campaign`-Objekt hinzugefügt werden soll, ist es vorerst ausreichend, nur dieses als Nachricht zu senden. Der zu sendende Nachrichtentyp ist daher vom Typ `Campaign`. Da CDI Nachrichten über die generische Klasse `Event` aus dem Paket `javax.enterprise.event` sendet, muss `Campaign` als Typparameter für diese Klasse festgelegt werden. Wir müssen daher der Bean `EditCampaignController` ein Attribut des Typs `Event<Campaign>` injizieren, indem wir die folgenden Zeilen einfügen:

```
@Inject
private Event<Campaign> campaignAddEvent;
```

In der Methode `doSave` muss nun die entsprechende Nachricht gesendet werden, anstatt direkt auf die Bean `CampaignListProducer` zuzugreifen. Hierzu muss die folgende Zeile: