HANSER

Leseprobe

Bernd Müller

JavaServer Faces 2.0

Ein Arbeitsbuch für die Praxis

ISBN: 978-3-446-41992-6

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41992-6

sowie im Buchhandel.

Kapitel 4

JavaServer Faces im Detail

Nachdem wir die Comedian-Anwendung erfolgreich entwickelt haben, ist es an der Zeit, die Grundlagen der JavaServer Faces zu erörtern. Dieses Kapitel beschreibt detailliert die für ein vollständiges Verständnis von JavaServer Faces benötigten Konzepte und Hintergründe, z. B. wie die einzelnen Verarbeitungsschritte einer JSF-Anfrage aussehen, wie existierende Validierer und Konvertierer eingesetzt und eigene entwickelt, wie Events verarbeitet werden und vieles mehr.

Zentral für das Verständnis der inneren Arbeitsweise von JSF ist es, zu verinnerlichen, dass JSF ein server-seitiges Komponenten-Framework ist. Alle JSF-Komponenten, auch wenn sie letztendlich z. B. als Texteingaben oder Schaltflächen auf dem Client dargestellt werden, sind Instanzen bestimmter Java-Klassen auf dem Server. Die komplette Verarbeitung und der Umgang mit diesen Komponenten geschieht auf dem Server, und lediglich am Ende einer Anfrage-Bearbeitung wird die Antwort an den Client geschickt. Diese Antwort ist aber immer ein Spiegelbild der Komponenten auf dem Server und kein selbstständiges Artefakt.

4.1 Bearbeitungsmodell einer JSF-Anfrage

JavaServer Faces werden mit Hilfe des Servlet-APIs realisiert. Servlets wiederum basieren auf dem Request-Response-Modell des zugrunde liegenden HTTP-Protokolls. JavaServer Faces erben somit die Eigenheiten einer HTTP-Anfrage und einer HTTP-Antwort, versuchen aber möglichst viel von diesem Erbe zu verstecken. Insbesondere die Zustandslosigkeit von HTTP, für deren Umgehung bei einer Servlet-Anwendung viel Aufwand investiert werden

muss, wird durch ein vollständiges MVC-Konzept ersetzt, bei dem Zustände eine wichtige Rolle spielen. So kann etwa der Controller mittels Change-Event die Änderung einer Benutzereingabe zwischen zwei HTTP-Anfragen erkennen. Um dies zu ermöglichen, muss ein Abbild des Zustands gespeichert und bei jeder neuen Anfrage mit dem dann neuen, aktuellen Zustand verglichen werden. Validierungen und Konvertierungen müssen durchgeführt, Events verarbeitet, aber eventuell auch neue erzeugt werden. Dieses durchaus komplexe und umfangreiche Verfahren wird in der Spezifikation „*Request Processing Lifecycle*“ genannt; es ist Gegenstand dieses Abschnitts. Wir sprechen im Folgenden von *Bearbeitungsmodell* oder *Lebenszyklus*.

Tabelle 4.1: Möglichkeiten von Anfragen und Antworten

	JSF-Anfrage	andere Anfrage
JSF-Antwort	1	2
andere Antwort	3	4

Prinzipiell kann eine HTTP-Anfrage einer JSF-Seite von einer JSF-Seite oder einer Nicht-JSF-Seite kommen. Genauso kann eine JSF-Seite eine JSF-Antwort oder eine Nicht-JSF-Antwort generieren. Man kann also vier Fälle unterscheiden, die in Tabelle 4.1 dargestellt sind. Unter einer JSF-Anfrage versteht man eine Anfrage, die durch eine zuvor generierte JSF-Antwort, z. B. ein durch JavaServer Faces erzeugtes HTML-Formular, ausgelöst wurde. Man spricht auch von einem *Post-Back*. Eine solche Anfrage enthält immer die Id einer View. Eine JSF-Anfrage kann aber auch eine Anfrage nach Teilen einer Seite sein, etwa einer CSS- oder JavaScript-Datei. Eine andere (Nicht-JSF-)Anfrage ist z. B. ein gewöhnlicher HTML-Verweis.

Eine JSF-Antwort ist eine Antwort, die von der letzten Phase der Anfragebearbeitung, der Render-Phase, erzeugt wurde. Eine andere (Nicht-JSF-)Antwort ist z. B. eine normale HTML-Seite, ein PDF-Dokument oder Teile einer HTML-Seite, z. B. eine CSS- oder JavaScript-Datei. Die Spezifikation spricht in diesem Zusammenhang von „Faces Request“ und „Faces Response“ bzw. von „Non-Faces Request“ und „Non-Faces Response“. Für den Fall von Anfragen oder Antworten von JSF-Teilbereichen spricht die Spezifikation von „Faces Resource Request“ und „Faces Resource Response“.

Es ist offensichtlich, dass die vierte Möglichkeit der Tabelle 4.1 nichts mit JavaServer Faces zu tun hat. Auch die zweite und dritte Möglichkeit sind Sonderfälle. Die folgenden Ausführungen beziehen sich auf die erste Möglichkeit, bei der eine JSF-Anfrage eine JSF-Antwort nach sich zieht. Die seit der Version 2.0 möglichen Ajax-Requests behandeln wir gesondert in Kapitel 7.

Die Bearbeitung einer JSF-Anfrage beginnt, wenn das JSF-Servlet den HTTP-Request erhalten hat. Es gibt insgesamt sechs zu unterscheidende Bearbeitungsphasen. Zwischen diesen sind Event-Verarbeitungsphasen vorgesehen. Abbildung 4.1 veranschaulicht dies grafisch.

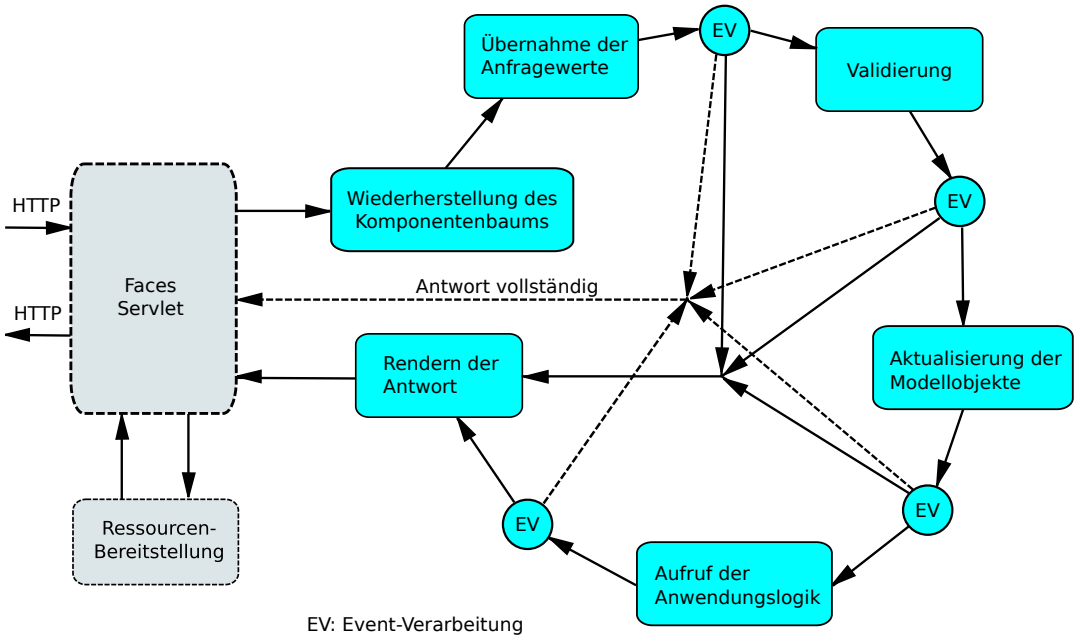

Abbildung 4.1: Bearbeitungsmodell einer JSF-Anfrage

Zunächst beginnen wir mit dem Faces-Servlet. JavaServer Faces sind mit einem Servlet realisiert, das im Deployment-Deskriptor der Servlet-Anwendung definiert werden muss (Abschnitt 4.8.1) bzw. in einer Servlet-Umgebung der Version 3.0 automatisch erkannt wird. Dieses Servlet nimmt HTTP-Anfragen entgegen. Zunächst wird entschieden, ob es sich um eine Ressourcen-Anfrage handelt, beispielsweise um eine JavaScript-Datei. Wenn ja, wird diese Ressource als HTTP-Antwort zurückgeschickt. Falls der Request keine Ressourcen-Anfrage ist, wird das Bearbeitungsmodell angestoßen.

Im Modell werden die sechs Phasen als Rechtecke dargestellt. Man erkennt, dass nach den inneren Phasen (zwei bis fünf) jeweils Events an interessierte Event-Listener übergeben werden können. Bei deren Abarbeitung können JSF-Komponenten verändert oder Anwendungsdaten verarbeitet werden. Man

kann auch komplett auf das Rendern der Antwort verzichten und direkt an das Ende der Bearbeitung springen. Dies ist durch die gestrichelten Linien in Abbildung 4.1 angedeutet. Beispiele hierfür sind binäre Daten, wie etwa eine JPG-Grafik oder ein PDF-Dokument, die als komplette Antwort zurückgeliefert werden und nicht Bestandteil einer HTML-Seite sind. Bei Validierungs- und Konvertierungsfehlern werden die nachfolgenden Phasen übersprungen und nur noch die Antwort generiert, die dann in der Regel Meldungen über den Fehler enthält.

Mit der Einführung von Ajax-Requests in JSF 2.0 ändert sich das dargestellte Verhalten etwas. Die sechste Phase, das Rendern der Anwort-Seite, entfällt, da lediglich eine XML-Struktur und keine komplette Seite zurückgegeben wird. Die ersten fünf Phasen werden auf einen oder mehrere Teilbereiche der Seite beschränkt und arbeiten nicht auf dem gesamten Komponentenbaum. Wir gehen auf die mit Ajax verbundene Komplexität in den nächsten Abschnitten zunächst nicht ein, holen dies aber in Kapitel 7 nach.

4.1.1 Wiederherstellung des Komponentenbaums

JSF-Komponenten besitzen einen Zustand, der bei einer Drop-Down-Liste z.B. die aktuelle Selektion in der Liste enthält. Die View ist eine nicht sichtbare Komponente, die die Wurzel des Baums aller Komponenten dieser Seite darstellt. Da die Zeit zur Beantwortung einer Anfrage wesentlich kürzer als die Zeit zwischen zwei Anfragen derselben Session ist und man nicht alle Komponenten einer Seite zwischen zwei Anfragen benötigt, werden die Komponenten einer Seite nach Beantwortung einer Anfrage gespeichert. Auf diese Weise spart man bei Anwendungen mit Tausenden von Benutzern viel Speicherplatz. Zu Beginn einer Anfragebearbeitung muss daher zunächst der Komponentenbaum wiederhergestellt werden.

Die Speicherung der Komponenten und deren Zustände kann auf dem Client oder dem Server erfolgen. Die Konfiguration der Speicherart wird in Abschnitt 4.8 erläutert. Jede View hat eine View-ID, die das URI der Anfrage darstellt. Bei der Bearbeitung eines Comedians in der Anwendung aus Kapitel 3 (Abbildung 3.2 auf Seite 29) gehört „`/comedians`“ noch zum frei konfigurierbaren Anwendungsnamen, die View-ID ist somit „`/comedian.jsf`“. Die View-ID wird in der Session gespeichert. Bei einer Anfrage kann daher entschieden werden, ob die Anfrage von einer JSF-Seite initiiert wurde (Alternative 1 in Tabelle 4.1) oder ob die Seite zum ersten Mal besucht wird (Alternative 2 in Tabelle 4.1). Bei der ersten Alternative wird der gespeicherte Komponentenbaum im alten Zustand wiederhergestellt, bei der zweiten wird ein neuer Komponentenbaum erstellt. Der wiederhergestellte oder neu erstellte Kompo-

nentenbaum wird dann im aktuellen `FacesContext` gespeichert. Ein Objekt der Klasse `FacesContext` im Package `javax.faces.context` enthält alle Informationen, die im Zusammenhang der Verarbeitung *einer* JSF-Anfrage stehen. Wir werden den `FacesContext` noch häufiger verwenden.

Die Wiederherstellung des Komponentenbaums umfasst nicht nur die Zustände der Komponenten, sondern auch das Wiederherstellen aller mit den Komponenten verbundenen Event-Listener, Validierer, Konvertierer und Managed Beans. In dieser Phase erfolgt ebenfalls die Lokalisierung für die View. Besucht man die Seite zum ersten Mal (Alternative 2), wird dann zur letzten Phase, der Render-Phase, gesprungen, ansonsten erfolgt die Übernahme der Anfragewerte.

4.1.2 Übernahme der Anfragewerte

Einige UI-Komponenten lassen die Eingabe von Werten durch den Benutzer zu, sei es als Text oder als Auswahl von Alternativen. Diese Eingaben werden durch das zugrunde liegende Formular als POST-Parameter des HTTP-Requests codiert. Das JSF-Framework muss sie in dieser Phase *decodieren*, d. h. dem Request entnehmen, und der entsprechenden UI-Komponente zuweisen. Ein Beispiel: In der Comedian-Anwendung wird bei der Detailseite zur Eingabe des Comedian-Vornamens ein `<h:inputText>`-Element verwendet:

```
<h:inputText id="vorname" required="true"
    value="#{comedianHandler.aktuellerComedian.vorname}" />
```

Der durch JavaServer Faces generierte HTML-Code sieht dann so aus :

```
<input id="j_idt3:vorname" type="text" value=""
    name="j_idt3:vorname"/>
```

In diesem Fall ist er von der Referenzimplementierung generiert worden. Eine andere Implementierung erzeugt evtl. anderen Code. Die Id `j_idt3:vorname` ist durch die Spezifikation strukturell definiert, wobei der Präfix generiert wurde, da das Formular selbst keine explizite Id hat. Der folgende Code ist dem POST-Request entnommen:

```
...&j_idt3%3Avorname=Mario&j_idt3%3Anachname=Barth&...
```

Man erkennt im hinteren Teil, dass dem Parameter `j_idt3:vorname` (der Doppelpunkt ist durch „`%3A`“ codiert) der Wert „`Mario`“ zugewiesen wird. Es ist Aufgabe dieser Phase, den gesamten POST-String zu parsen und alle Parameter mit ihren jeweiligen Werten herauszufiltern, zu decodieren. Die Werte werden dann vorläufig den entsprechenden Komponenten zugewiesen. Vorläufig

deshalb, weil in den nachfolgenden Validierungen und Konvertierungen noch Fehler auftreten können.

Alle Eingabe- und Steuerkomponenten besitzen ein boolesches Attribut `immediate`. Ist dieses bei Eingabekomponenten gesetzt, finden Validierung und Konvertierung bereits in dieser Phase (Übernahme der Anfragewerte) und nicht in der nächsten Phase, der Validierung, statt. Ist das Attribut `immediate` bei Steuerkomponenten gesetzt, findet der Aufruf der Action-Methoden bzw. Action-Listener am Ende dieser Phase und nicht in der Phase *Aufruf der Anwendungslogik* statt. Wir gehen auf beide Alternativen an den entsprechenden Abschnitten ausführlich ein.

Am Ende der Phase zur Übernahme der Anfragewerte werden alle existierenden Events an die interessierten Listener weitergereicht. Die JSF-Implementierung kann zur Render-Phase springen oder die Bearbeitung der Anfrage komplett beenden.

4.1.3 Validierung

Zu Beginn der Validierungsphase ist sichergestellt, dass alle aktuellen Anfrageparameterwerte für ihre UI-Komponenten bereitstehen. Die JSF-Implementierung durchläuft nun den Komponentenbaum und stellt sicher, dass alle Werte valide sind. Dazu werden alle registrierten Validierer (einige Komponenten besitzen zusätzlich eigene Validierer) zur Validierung aufgefordert. Eventuell müssen vor der Validierung noch Konvertierungen vorgenommen werden.

In der Comedian-Anwendung wird für jede Eingabe (Vorname, Nachname, Geburtstag) validiert, ob das Eingabefeld einen Wert enthält (`required="true"`). Weitere Validierungen sind möglich. So wäre es z. B. sinnvoll zu prüfen, ob der Vor- und Nachname eine Minimal- und eine Maximallänge hat. Dies ist mit JSF leicht möglich:

```
<h:inputText id="nachname" required="true"
    value="#{comedianHandler.aktuellerComedian.nachname}">
  <f:validateLength minimum="3" maximum="20" />
</h:inputText>
```

Die in der Spezifikation *Process Validation* genannte Phase besteht neben der Validierung in der Regel auch aus einer zuvor vorzunehmenden Konvertierung. In der Comedian-Anwendung wird etwa ein den Geburtstag eines Comedians repräsentierender String in ein Java-Objekt der Klasse `java.util.Date` konvertiert, während Vor- und Nachname ohne Konvertierung übernommen werden können. Nach der Datumskonvertierung könnte man eine weitere Validierung, etwa die Überprüfung, dass das Geburtsdatum in der Vergangenheit

liegt, vornehmen. Hierfür stellt JSF jedoch keinen vordefinierten Validierer bereit. Mit der ab JSF 2.0 verfügbaren Bean-Validierung, die wir in Abschnitt 4.4.9 vorstellen, ist diese Überprüfung allerdings direkt realisierbar.

Nachdem alle Validierungen und Konvertierungen mit Erfolg durchgeführt wurden, wird der Wert nun endgültig der Komponente zugewiesen. Sollte sich der Wert seit der letzten Anfrage geändert haben, wird ein Value-Change-Event geworfen und an registrierte Listener weitergegeben. Diese Listener können nun zur Render-Phase springen, direkt die Antwort erzeugen oder zur Aktualisierung der Modellobjekte übergehen.

4.1.4 Aktualisierung der Modellobjekte

Bis zu diesem Zeitpunkt haben alle Vorgänge in den UI-Komponenten stattgefunden. Die Werte sind valide und vom richtigen Typ und können nun den Modellobjekten zugewiesen werden. In unserem Beispiel

```
<h:inputText id="vorname" required="true"
    value="#{comedianHandler.aktuellerComedian.vorname}" />
```

ist der Ausdruck `"#{comedianHandler.aktuellerComedian.vorname}"` dafür zuständig. Die JSF-Implementierung sucht nach einer Managed Bean mit dem Namen `comedianHandler`. Dann wird das `aktuellerComedian`-Property dieses Objekts ausgewertet. Das Property `vorname` des Ergebnisobjekts bekommt den Wert der UI-Komponente zugewiesen. Wie am Ende jeder Phase werden wieder Events geworfen, Listener informiert, und eventuell wird an das Ende der Bearbeitung gesprungen.

4.1.5 Aufruf der Anwendungslogik

Bis zu diesem Zeitpunkt haben wir noch keinen Applikations-Code verwendet, obwohl bereits relativ viel Aufwand betrieben wurde: Benutzereingaben wurden konvertiert und validiert und die Properties der Managed Beans aktualisiert. Die JSF-Implementierung hat alles automatisch erledigt.

Nun kann die Anwendungslogik aufgerufen werden. Dies geschieht durch Listener, die sich auf Action-Events registriert haben, die durch Schaltflächen oder Hyperlinks ausgelöst werden können. Dabei sind zwei Arten von Listener zu unterscheiden. Die erste Art sind die mit dem Attribut `actionListener` registrierten „richtigen“ Listener. Ein Beispiel sind die neun Schaltflächen des Tic-Tac-Toe-Spiels (siehe Listing 2.2 auf Seite 14):

```
<h:commandButton id="feld-0" image="#{tttHandler.image[0]}"
   actionListener="#{tttHandler.zug}" />
```

Sie registrieren die Listener-Methode `zug`. Eine solche Listener-Methode muss als Parameter ein `ActionEvent` haben (siehe Listing 2.4 auf Seite 17):

```
public void zug(ActionEvent ae) {
  ...
```

Die zweite Art sind die automatisch für Steuerkomponenten registrierten Default-Action-Listener. Dies wird in der Komponente durch ein `action`-Attribut und im Handler durch eine Methode ohne Parameter erfüllt (Beispiel: Speichern eines Comedians):

```
<h:commandButton action="#{comedianHandler.speichern}"
    value="Speichern" />
```

Hier wird die Action-Methode `speichern()` registriert. Eine Action-Methode muss einen String zurückliefern und hat keinen Parameter:

```
public String speichern() {
  ...
```

Seit JSF 1.2 ist der Rückgabewert von Action-Methoden auf `Object` erweitert worden, um Enumerations als mögliche Rückgabetypen zu ermöglichen. Für das Rückgabeobjekt wird die `toString()`-Methode aufgerufen, um den so erhaltenen String zur Navigation verwenden zu können.

Action-Methoden sind nicht auf die Anwendungslogik beschränkt. Sie können z. B. auch Events generieren, Anwendungsmeldungen erzeugen oder gar die Antwort selbst rendern.

4.1.6 Rendern der Antwort

Nach der Abarbeitung der Anwendungslogik bleiben in der letzten Phase der Anfragebearbeitung noch das Rendern der Antwort und das Abspeichern des Komponentenbaums als zentrale Aufgaben.

Die Zielsprache des Renderns der Antwort ist durch die Spezifikation bewusst offen gelassen worden. Es sind mehrere Alternativen denkbar, z. B. XHTML, JSP, WML oder SVG. Bis einschließlich JSF 1.2 war festgelegt, dass jede Implementierung mindestens JSP als Anzeigetechnik unterstützen muss. Mit JSF 2.0 kam die *Page-Description-Language* (*PDL*) auf der Basis von Facelets hinzu, auf die wir ausführlich in Abschnitt 5.6 eingehen. Die Neuerungen der Version 2.0 sind nur in Facelets enthalten, JSP wird nur noch aus Kompatibilitätsgründen weitergeführt.

Während bei der zweiten Phase, der Übernahme der Anfragewerte, die Komponentenwerte decodiert werden mussten, müssen Sie nun codiert werden. Das

Geburtsdatum eines Comedians ist etwa ein `Date`-Objekt, das allerdings in der HTML-Seite als String darzustellen ist.

Da der Komponentenbaum programmatisch in der Phase 5, *Aufruf der Anwendungslogik*, verändert werden kann, ist es möglich, dass mehr, aber auch weniger Komponenten zu rendern sind als im Ursprungs-Code der JSF-Seite vorhanden.

Zuletzt muss das Abspeichern des Komponentenbaums so erfolgen, dass bei einer erneuten Anfrage der Seite der Komponentenbaum in seinem dann ursprünglichen Zustand wiederhergestellt werden kann.

Die Version 2.0 führt das sogenannte *Partial State Saving* ein. Da der Komponentenbaum in der Regel nur selten und in geringem Umfang verändert wird, speichert die JSF-Implementierung nur die Änderungen am Komponentenbaum. Die aktuelle Version kann dann über die textuelle Repräsentation der JSF-Seite und die abgespeicherten Änderungen konstruiert werden. Ob das partielle Speichern des Komponentenbaums tatsächlich stattfindet, wird über eine Konfigurationsoption bestimmt.

4.2 Expression-Language

Die JSTL und JSP führten eine Expression-Language (kurz EL) ein, um Entwicklern von JSP-Seiten eine Möglichkeit für den einfachen Zugriff auf Anwendungsdaten zu ermöglichen, ohne den Weg über Java gehen zu müssen. Bei der Definition von JSF wurden die Möglichkeiten einer Expression-Language ebenfalls als sehr wichtiges Element einer Seitenbeschreibungssprache erkannt, und eine Expression-Language sollte integraler Bestandteil von JSF sein. JSF stellt jedoch andere Anforderungen als JSP an eine Expression-Language, so dass für JSF eine eigene Expression-Language definiert wurde.

Die Verwendung von JSP als Seitenbeschreibungssprache für JSF erlaubt die Verwendung der JSP-EL als auch der JSF-EL innerhalb einer Seite. Dies führte in den JSF-Versionen 1.0 und 1.1 zu Problemen und letztendlich zur Definition der *Unified Expression-Language* [URL-UEL].

Die Unified EL ist als Teildokument der JavaServer-Pages-Spezifikation 2.1 definiert [URL-JSR245] und wird in dieser Form in JSF 1.2 und ein wenig erweitert in JSF 2.0 verwendet. Die Unified EL unterscheidet zwischen sofortiger Auswertung (sogenannte *Immediate Expressions*, beginnend mit einem `$`-Zeichen) zum Zeitpunkt des Renderns der Seite (bei JSPs der Compile-Zeitpunkt) und der verzögerten oder zeitversetzten Auswertung (sogenannte *Deferred Expressions*, beginnend mit einem `#`-Zeichen) zur Laufzeit. Die zeit-

verzögerte Auswertung macht sich insbesondere dadurch bemerkbar, dass EL-Ausdrücke zweimal ausgewertet werden, und zwar während der Übernahme der Anfragewerte (Phase 2) als auch dem Rendern der Antwort (Phase 6). Werteausdrücke werden daher sowohl schreibend (Phase 2) als auch lesend (Phase 6) verwendet.

Durch die Einführung der auf Facelets basierenden PDL in JSF 2.0 ist es in der Regel nicht notwendig, JSP in JSF-Anwendungen zu verwenden. Wir raten dem Leser, auf die Verwendung von JSP vollständig zu verzichten, um nach wie vor vorhandene Probleme beim Mischen beider Technologien zu vermeiden. Im Folgenden verwenden wir ausschließlich Facelets und zeitversetzte Ausdrücke und können daher auf eine Unterscheidung verzichten.

4.2.1 Syntax

Die Expression-Language enthält Konzepte, wie man sie auch in JavaScript und XPath findet. In der Expression-Language navigiert man durch eine Punkt-Notation über Objekt-Properties, so wie man in XPath im Baum des XML-Dokuments navigiert. Die Properties werden bei der Auswertung automatisch mit `get` und `set` erweitert und die Großschreibung den Java-Konventionen angepasst. Wir gehen darauf später noch genauer ein.

Zunächst wollen wir aber die grundlegende Syntax der Expression-Language einführen. Ausdrücke der Expression-Langage schreiben wir als String und schließen sie durch das Rautezeichen (`#`, üblich sind auch die Bezeichnungen „Hash“, Nummernzeichen oder Lattenzaun) und geschweifte Klammern ein:

```
"#{expr}"
```

Innerhalb dieser Klammern steht der eigentliche EL-Ausdruck. Er kann einen Werteausdruck, einen Methodenausdruck, aber auch einen einfachen arithmetischen oder logischen Ausdruck enthalten. Beliebige Kombinationen der genannten Alternativen (ohne Methodenausdrücke) sind ebenfalls möglich. Das Verschachteln von Ausdrücken der Art `#{feld[#{i}]}` ist nicht erlaubt. Anstatt Anführungszeichen ist alternativ die Verwendung einfacher Apostrophe erlaubt. Wir empfehlen die konsequente Verwendung von Anführungszeichen.

4.2.2 Bean-Properties

Über einen *Werteausdruck* (engl. Value Expression) kann der Wert einer UI-Komponente an eine Bean-Property oder die UI-Komponente selbst an eine Bean-Property gebunden werden. Werteausdrücke werden auch verwendet, um Properties einer UI-Komponente zu initialisieren.

Über einen *Methodenausdruck* (engl. Method Expression) lässt sich eine Bean-Methode referenzieren. Dies wird z. B. bei Event-Handlern und Validierungsmethoden verwendet.

Wir konzentrieren uns zunächst auf den einfachsten Fall: den Werteausdruck. Ein Werteausdruck muss zu einer einfachen Bean-Property, einem Element eines Arrays, einer Liste (`java.util.List`) oder einem Eintrag in einer Map (`java.util.Map`) evaluieren. Die Instanz, die der EL-Ausdruck referenziert, kann sowohl gelesen als auch geschrieben werden. Das Schreiben in die Instanz geschieht in der Phase *Aktualisierung der Modellobjekte*, das Lesen in der Phase *Rendern der Antwort*.

Einige kleine Bespiele für verschiedene Property-Zugriffe sind in der JSF-Seite `el.xhtml` enthalten, die wir uns ausschnittweise anschauen:

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">Zugriff auf Bean-Properties</f:facet>
3   <h:outputText value="#{elHandler.name}" />
4   <h:outputText value="#{elHandler['name']}" />
5   <h:outputText value="Dies sind tolle #{elHandler.name}" />
6   <h:outputText value="#{elHandler.array[0]}" />
7   <h:outputText value="#{elHandler.map['zwei']}" />
8   <h:outputText value="#{elHandler.map[elHandler.array[2]]}" />
9 </h:panelGrid>
```

Zunächst müssen wir aber noch den Handler vorstellen. Die Seite verwendet einen einfachen Handler, der in Listing 4.1 in Teilen dargestellt ist. Die Managed Bean `elHandler` der JSF-Seite ist eine Instanz dieser Klasse.

Listing 4.1: Handler für EL-Ausdrücke

```
public class ELHandler {

  private String name = "Übungen mit der Expression-Language";
  private Integer jahr = new Integer(
                          new java.text.SimpleDateFormat("yyyy")
                          .format(new java.util.Date()));
  private String[] array =
          new String[]{ "eins", "zwei", "drei" };
  private Map<String, String> map =
          new HashMap<String, String>();

  public ELHandler() {
    super();
    map.put("eins", "Erster Map-Eintrag");
    map.put("zwei", "Zweiter Map-Eintrag");
    map.put("drei", "Dritter Map-Eintrag");
```

```
}

// ab hier nur einfache Getter und Setter
```

Im Beispiel verwenden wir nur Ausgabekomponenten, die Werteausdrücke werden daher ausschließlich lesend interpretiert. Bei der Verwendung einer Eingabekomponente könnten sie auch zum Setzen eines Properties verwendet werden. Der EL-Ausdruck des ersten Beispiels in Zeile 3 evaluiert zum Wert des Getters `getName()`, in diesem Fall also „`Übungen mit der Expression-Language`“. Dafür wird zunächst der Wert des Teilausdrucks links vom ersten Punkt bestimmt, in diesem Fall `elHandler`. Dies ist der Name einer Managed Bean, einer Instanz der Klasse `ELHandler`. Das Property `name` wird in den schon erwähnten Getter überführt und der Wert durch Aufruf des Getters bestimmt.

Für den Zugriff auf Arrays, Listen oder Maps übernimmt die Expression-Language die Java-Array-Notation der eckigen Klammern. Properties können alternativ auch mit der Klammernotation verwendet werden, so dass die Zeilen 3 und 4 semantisch äquivalent sind. In Zeile 5 wird eine String-Konstante mit einem Werteausdruck verbunden. Eine String-Konstante ohne das `#`-Zeichen wird wie üblich als Literal oder Literalausdruck bezeichnet.

Der Zugriff auf Elemente eines Arrays erfolgt in der Java-üblichen Notation, wie in Zeile 6 dargestellt. Auch Listen werden nach diesem Schema indiziert. Beim Zugriff auf Maps muss der Schlüssel als Konstante in eckigen Klammern geschrieben werden. Die Expression-Language lässt sowohl den Apostroph als auch das Anführungszeichen zur Kennzeichnung von String-Konstanten zu. Da wir als JSF-Implementierung aber Facelets verwenden und diese der XML-Syntax zu gehorchen haben, müssen Tag-Attribute und EL-Ausdrücke verschiedene Quotierungen verwenden. Wie von Ausdrücken nicht anders zu erwarten, können diese beliebig geschachtelt werden. Zeile 8 ist ein Beispiel hierfür.

Die Darstellung der JSF-Seite im Browser zeigt Abbildung 4.2 auf der nächsten Seite. Allerdings haben wir im Augenblick nur die ersten Zeilen der Seite durchgesprochen. Die weiteren folgen später.

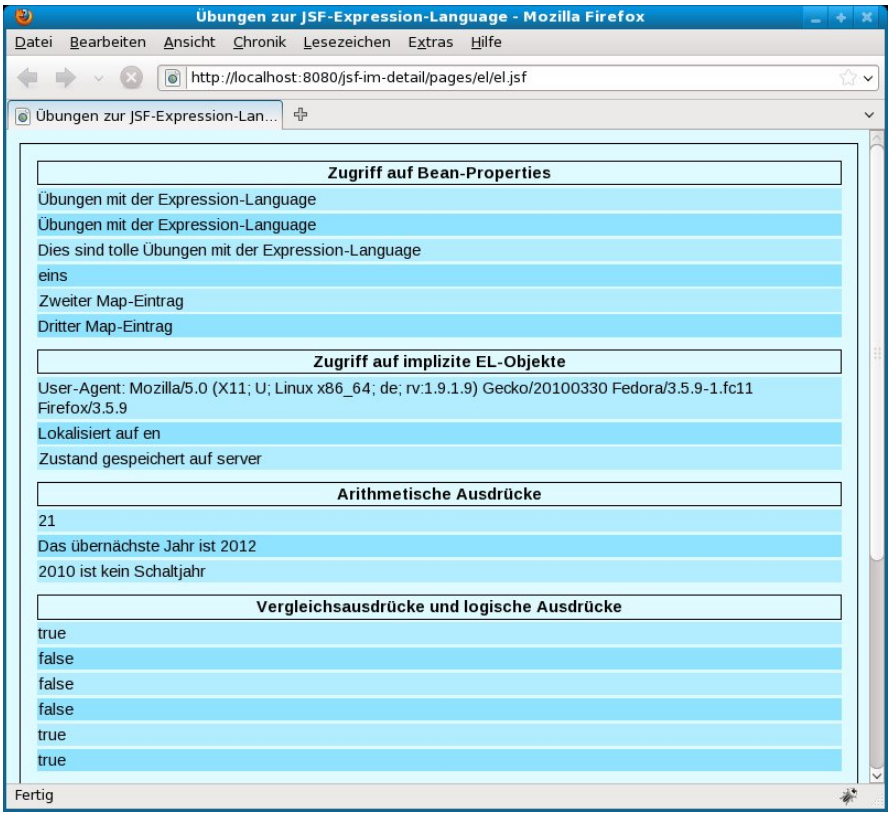

Abbildung 4.2: Beispiele für EL-Ausdrücke

Wir haben in diesem Abschnitt und auch bei der Vorstellung des Bearbeitungsmodells einer JSF-Anfrage immer von Gettern und Settern mit der Syntax `getXxx()` und `setXxx()` gesprochen. Die JavaBean-Spezifikation sieht für boolesche Properties zusätzlich die Möglichkeit eines Getters in der Form `isXxx()` vor. JSF unterstützt diese Form ebenfalls.

4.2.3 Vordefinierte Variablen

JSF definiert einige Objekte implizit, die in der Expression-Language verwendet werden können. Diese vordefinierten Variablen bezeichnen also verschiedene Objekte der zugrunde liegenden Servlet- und JSF-Implementierung. So ist es etwa möglich, auf den HTTP-Request-Header über einen vordefinierten Namen zuzugreifen. Tabelle 4.2 zeigt diese vordefinierten Variablen und erläutert sie kurz.

Tabelle 4.2: Vordefinierte Variablen der Expression-Language

Variablenname	Beschreibung
header	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist *ein* String.
headerValues	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist ein Array von Strings.
cookie	Eine `Map` von Cookies (Klasse `Cookie` im Package `javax.servlet.http`). Schlüssel ist der Cookie-Name.
initParam	Eine `Map` von Initialisierungsparametern der Anwendung (Application-Scope). Diese werden im Deployment-Deskriptor definiert.
param	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist *ein* String.
paramValues	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist ein Array von Strings.
facesContext	Die `FacesContext`-Instanz der aktuellen Anfrage.
component	Die im Augenblick bearbeitete Komponente.
cc	Die im Augenblick bearbeitete zusammengesetzte Komponente.
resource	Eine `Map` von Ressourcen.
flash	Eine `Map` von temporären Objekten für die nächste View.
view	Das View-Objekt (View-Scope).
viewScope	Eine `Map` von Variablen mit View-Scope.
request	Das Request-Objekt (Request-Scope).
requestScope	Eine `Map` von Variablen mit Request-Scope.
session	Das Session-Objekt (Session-Scope).
sessionScope	Eine `Map` von Variablen mit Session-Scope.
application	Das Application-Objekt (Application-Scope).
applicationScope	Eine `Map` von Variablen mit Application-Scope.

Als Beispiel zur Verwendung vordefinierter Variablen dient das folgende Code-Fragment (Fortsetzung der JSF-Seite `el.xhtml`):

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">
3     Zugriff auf implizite EL-Objekte
4   </f:facet>
5   <h:outputText value="User-Agent: #{header['User-Agent']}" />
6   <h:outputText value="Lokalisiert auf #{view.locale}" />
7   <h:outputText value="Zustand gespeichert auf \
```

```
8       #{initParam['javax.faces.STATE_SAVING_METHOD']}" />
9 </h:panelGrid>
```

In Zeile 5 wird der User-Agent des Request-Headers ausgegeben. Jede HTTP-Anfrage enthält den Client, der die Anfrage gestellt hat. In Abbildung 4.2 ist dies ein Mozilla unter Linux. In Zeile 6 wird die Lokalisierung der View erfragt. Im Beispiel ist dies eine deutsche Lokalisierung. In den Zeilen 7/8 wird schließlich auf einen Initialisierungsparameter der Anwendung zugegriffen. Eine Servlet-Anwendung wird durch den Deployment-Deskriptor `web.xml` konfiguriert. Im Beispiel enthält diese Datei die Zeilen

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
```

die die Speicherung des Zustands auf dem Server definieren. Der Parameterwert `server` ist im Beispiel der Wert des EL-Ausdrucks. Auf Fragen der Konfiguration gehen wir in Abschnitt 4.8 ein. Die in Tabelle 4.2 genannten Scopes führen wir in Abschnitt 4.3 ein.

Wir beschließen den Abschnitt mit einem Beispiel für einen weiteren Zugriff auf Konfigurationsinformationen. Der Name einer Servlet-Anwendung wird zum Deployment-Zeitpunkt festgelegt und ist nicht durch JSF definiert. Über die vordefinierte Variable `request` kann jedoch der Kontextpfad ermittelt werden:

```
<head>
  <title>Übungen zur JSF-Expression-Language</title>
  <link rel="stylesheet"  type="text/css"
     href="#{request.contextPath}/css/style.css" />
</head>
```

In diesem Beispiel wird über den Kontextpfad eine Style-Sheet-Datei eingebunden. Dies hätte man alternativ auch kontextrelativ realisieren können, und dient lediglich als Beispiel. Seit Version 2.0 gibt es eine weitere Alternative, Ressource-Dateien einzubinden, auf die wir im Detail in Abschnitt 4.10 eingehen.

4.2.4 Vergleiche, arithmetische und logische Ausdrücke

Die Expression-Language umfasst ein vollständiges Repertoire an Vergleichsausdrücken sowie arithmetischen und logischen Ausdrücken, so dass ein Rückgriff auf Java in der Regel nicht notwendig ist. Tabelle 4.3 führt die Operatoren der Expression-Language auf.

Tabelle 4.3: Operatoren der JSF-Expression-Language

Operator	Alternative	Beschreibung
`.`		Zugriff auf ein Property, eine Methode oder einen Map-Eintrag
`[]`		Zugriff auf ein Array- oder Listen-Element oder einen Map-Eintrag
`()`		Klammerung für Teilausdrücke
`?:`		Bedingter Ausdruck: `<expr> ? <true-value> : <false-value>`
`+`		Addition
`-`		Subtraktion oder negative Zahl
`*`		Multiplikation
`/`	`div`	Division
`%`	`mod`	Modulo
`==`	`eq`	gleich
`!=`	`ne`	ungleich
`<`	`lt`	kleiner
`>`	`gt`	größer
`<=`	`le`	kleiner-gleich
`>=`	`ge`	größer-gleich
`&&`	`and`	logisches UND
`||`	`or`	logisches ODER
`!`	`not`	logische Negation
`empty`		Test auf `null`, einen leeren String, oder Test auf Array, `Map` oder `Collection` ohne Elemente

Beispiele mit Vergleichen sowie arithmetischen und logischen Ausdrücken sind etwa (Fortsetzung der JSF-Seite `el.xhtml`):

```
1  <h:panelGrid columns="1" rowClasses="odd,even">
2    <f:facet name="header">Arithmetische Ausdrücke</f:facet>
3    <h:outputText value="#{17 + 4}" />
4    <h:outputText
5        value="Das übernächste Jahr ist #{elHandler.jahr + 2}" />
6    <h:outputText
7        value="#{elHandler.jahr} ist #{((elHandler.jahr % 4) == 0 \
8               ? 'ein' : 'kein')} Schaltjahr" />
9  </h:panelGrid>
10
11 <h:panelGrid columns="1" rowClasses="odd,even">
```

```
12    <f:facet name="header">
13      Vergleichsausdrücke, arithmetische und logische Ausdrücke
14    </f:facet>
15    <h:outputText value="#{'eins' == elHandler.array[0]}" />
16    <h:outputText value="#{2009 == elHandler.jahr}" />
17    <h:outputText value="#{'2009' == elHandler.jahr}" />
18    <h:outputText value="#{2008 == elHandler.jahr}" />
19    <h:outputText value="#{elHandler.jahr > 2000}" />
20    <h:outputText value="#{elHandler.jahr > 2000 \
21                           and (elHandler.jahr %4 != 0)}" />
22  </h:panelGrid>
```

In Zeile 3 werden zwei Konstanten addiert. In Zeile 5 ist ein Summand ein Werteausdruck. Die Zeilen7/8 sind ein Beispiel für den bedingten Ausdruck. Dieser ternäre Ausdruck hat dieselbe Semantik wie in Java: ergibt die Auswertung des ersten Teilausdrucks `true`, so ist der zweite Teilausdruck der Wert des Gesamtausdrucks, sonst der dritte. Bitte beachten Sie, dass der String-Wert des Attributs `value` in einer Zeile stehen muss. Der Zeilenumbruch am Ende von Zeile 7 ist der Drucktechnik geschuldet und würde zu einem Fehler führen. Auch der Algorithmus zur Schaltjahrbestimmung ist nicht ganz korrekt.

In Zeile 15 beginnen die Beispiele für Vergleiche mit einem String-Vergleich. Der Vergleich in Zeile 16 enthält Integer als Operanden. In Zeile 17 wird ein String mit einem Integer verglichen, was zu keinem Fehler führt. Die EL-Spezifikation definiert, welche impliziten Typkonvertierungen vor der Durchführung des Vergleichs zu erfolgen haben. Da die Konvertierungen relativ intuitiv sind, verzichten wir auf eine Darstellung und verweisen den interessierten Leser auf die EL-Spezifikation. Im Falle der Zeile 17 wird der Integer in einen String konvertiert und dann der Vergleich durchgeführt.

Wir haben in den Beispielen bereits verschiedene Literale wie z. B. Strings und Zahlen verwendet. Strings werden entweder in Anführungszeichen oder Apostrophe eingeschlossen, Zahlen als einfache Zahlen geschrieben, gebrochene Zahlen mit Dezimalpunkt. Die booleschen Literale werden als `true` und `false` geschrieben. Das Schlüsselwort `null` repräsentiert einen „nicht vorhandenen Wert“.

Wie alle Sprachen besitzt auch die Expression-Language reservierte Wörter, die nicht in Ausdrücken verwendet werden dürfen. Neben den in der Tabelle 4.3 in der Spalte *Alternative* genannten Wörtern sind dies die schon erwähnten Wörter `empty`, `true`, `false` und `null`. Weiterhin ist die Verwendung von `instanceof` nicht erlaubt, obwohl es (noch) keine Verwendung in der Expression-Language findet.

4.2.5 Methodenaufrufe und Methodenparameter

Die Unified Expression-Language als Teil von JSP 2.1 erhielt im Rahmen von Java-EE 6 das zweite Maintenance-Review [URL-JSR245II]. In diesem Maintenance-Review wird die Syntax von EL-Ausdrücken um die Möglichkeit von Methodenaufrufen mit Parametern erweitert. Da JSF 2.0 lediglich die Unified EL von JSP 2.1, nicht aber explizit das zweite Maintenance-Review voraussetzt, beschreiben wir diese Erweiterung in einem eigenen Abschnitt. Bitte vergewissern Sie sich, ob Ihre JSF-Implementierung und Ihr Container das zweite Maintenance-Review unterstützen.

Methodenaufrufe in Werteausdrücken waren bisher lediglich in Form von Getter- und Setter-Aufrufen erlaubt. Mit den Erweiterungen im zweiten Maintenance-Review sind Methodenaufrufe und die Verwendung von Methodenparametern erlaubt. Zur Konstruktion eines Beispiels definieren wir drei einfache Methoden in der Bean `elHandler`:

```
public String methodWithOneParam(String param) {
  return param + " " + param;
}

public String methodWithTwoParams(String param1, int param2) {
  return param1 + " " + param2;
}

public List<Integer> getList() { ... }
```

Die beiden ersten Methoden können nun mit Parametern in der üblichen Syntax versehen werden. Die dritte Methode dient zur Demonstration des Aufrufs der `Collection.size()`-Methode, die mit den bisherigen EL-Mitteln nicht aufgerufen werden konnte, da sie nicht dem Getter-Schema (`getSize()`) entspricht. Die Darstellung im Browser zeigt Abbildung 4.3.

```
<h:outputText
    value="#{elHandler.methodWithOneParam('text')}" />
<h:outputText
    value="#{elHandler.methodWithTwoParams('text', 127)}" />
<h:outputText value="#{elHandler.list.size()}" />
```

Analog zu Werteausdrücken können auch Methodenausdrücke parametrisiert werden:

```
<h:commandButton action="#{bean.doAction(arg1, arg2)}" .../>
```

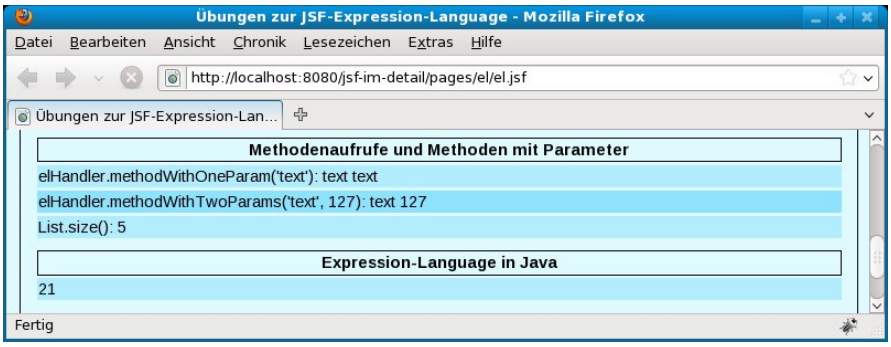

Abbildung 4.3: Weitere Beispiele für EL-Ausdrücke

4.2.6 Verwendung der Expression-Language in Java

Ziel der Expression-Language ist es, komplexe Ausdrücke direkt in JSF-Seiten zu formulieren, ohne auf Java zurückgreifen zu müssen. Damit erscheint die Verwendung der Expression-Language innerhalb von Java wenig sinnvoll. Es finden sich allerdings auch sinnvolle Einsatzgebiete, etwa der Zugriff auf eine Managed Bean, wie wir ihn in Abschnitt 4.3 kennenlernen werden.

Der folgende Ausschnitt der Seite `el.xhtml` zeigt den Quell-Code für die letzte Zeile der Browser-Darstellung in Abbildung 4.3:

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">Expression-Language in Java</f:facet>
3   <h:outputText value="#{elHandler.testAusdruck}" />
4 </h:panelGrid>
```

Der Werteausdruck in Zeile 3 dient als Grundlage zur Realisierung der Auswertung eines EL-Ausdrucks in Java. Bei der Auswertung wird die Methode `getTestAusdruck()` aufgerufen, auf die wir nun eingehen. Um einen Werteausdruck in Java zu erzeugen und auszuwerten, wird auf das `FacesContext`- und das `Application`-Objekt zurückgegriffen. Für das `Application`-Objekt kann mit der Methode `getExpressionFactory()` die `ExpressionFactory`-Instanz der Anwendung erfragt werden. Über diese Fabrik lässt sich bezüglich des aktuellen Kontextes ein Werteausdruck erzeugen und auswerten, wie der folgende Java-Code demonstriert:

```
public Integer getTestAusdruck() {
  FacesContext faces = FacesContext.getCurrentInstance();
  Application app = faces.getApplication();
  ExpressionFactory expressionFactory =
                              app.getExpressionFactory();
  ELContext el = faces.getELContext();
  Integer val = (Integer) expressionFactory
```

```
            .createValueExpression(el, "#{17 + 4}", Integer.class)
            .getValue(el);
    return val;
  }
```

Die Klasse `ExpressionFactory` wurde mit der Unified Expression-Language eingeführt. Das dargestellte Verfahren ist daher seit JSF 1.2 zu verwenden. Die vor der Version 1.2 zu verwendende Methode `createValueBinding()` der `Application`-Klasse zur Erzeugung eines Werteausdrucks ist deprecated und sollte nicht mehr verwendet werden. Weiterhin sind alle Klassen des Package `javax.faces.el`, also Klassen, die die JSF-EL bis Version 1.1 realisiert haben und nun durch die Unified Expression-Language realisiert werden, deprecated.

Im Beispiel wurde aus Gründen der Einfachheit der konstante Ausdruck `17 + 4` verwendet. Alle bisher eingeführten Konstrukte der Expression-Language sowie die impliziten Objekte aus Tabelle 4.2 können selbstverständlich ebenfalls verwendet werden.

Aufgabe 4.1
Laden Sie das Expression-Language-Projekt von der Web-Site des Buches herunter, und installieren Sie es. Erstellen Sie Ausdrücke für vom Client

- akzeptierte Sprachen
- akzeptierte Übertragungs-Codierungen
- akzeptierte Zeichensatz-Codierungen

Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Expression-Language sind im Projekt *jsf-im-detail* enthalten.

4.3 Managed Beans

Managed Beans sind einfache Java-Beans oder POJOs, die Daten von UI-Komponenten sammeln, Event-Listener-Methoden implementieren und ähnliche Unterstützungsaufgaben für UI-Komponenten wahrnehmen können. Des Weiteren ist es möglich, dass sie Referenzen auf UI-Komponenten beinhalten. Wir haben Managed Beans schon mehrfach in EL-Ausdrücken verwendet. Die Integration in die Expression-Language machen Managed Beans zu dem Werkzeug, mit dem Seiten-Entwickler ohne Java-Kenntnisse auf Java zurückgreifen können.