Nicht nur oberflächlich

Einsatz und Grenzen von JavaServer Faces 2.0 – Teil 1: Grundlagen

Kai Wähner

Ein Web-Framework ist darauf ausgelegt, schnell und einfach lauffähige Webanwendungen zu erstellen. Dabei hat es das Ziel, sich wiederholende Tätigkeiten zu vereinfachen, rein technische Aufgabenstellungen abzunehmen und die Wiederverwendung von Code zu fördern. Java Server Faces (JSF) ist ein Web-Framework im Java-Umfeld, das Teil der Standardspezifikationen von JEE 6 ist. Die aktuelle Version 2.0 zeichnet sich durch viele Verbesserungen im Vergleich zur Vorgängerversion 1.2 aus, sodass die Entwicklung deutlich vereinfacht wird. Durch die Standardisierung und die damit verbundene hohe Verbreitung sind eine große Community und dadurch auch sehr viele Erweiterungen zu JSF entstanden. Diese Artikelreihe stellt die großen Potenziale, aber auch die Grenzen von JSF vor.

Überblick über die Artikelreihe

Es sind bereits zahlreiche gute Fachartikel [JSF_JavaMagazin], Blogs und Bücher [GeaHor10] verfügbar, die JSF ausführlich und mit komplexen Code-Beispielen vorstellen. Hier werden daher stattdessen die vielen verschiedenen Einsatzmöglichkeiten und auch Grenzen dieses Web-Frameworks beschrieben. Die Vorstellung ist in drei Artikel aufgeteilt:

1. Grundlagen von JSF und konzeptionelle Neuigkeiten von Version 2.0
2. Vorstellung wichtiger Erweiterungen für JSF, beispielsweise Komponenten-Bibliotheken, Portal-Unterstützung und Frameworks zum Testen von JSF-Anwendungen
3. Aufzeigen der Grenzen von JSF, inklusive der Erläuterung, wann und warum manchmal ein anderes Web-Framework besser geeignet ist

Dieser erste Artikel stellt zunächst die grundlegenden Konzepte von JSF vor. Danach wird gezeigt, wie sich JSF an die Ziele „Leichtgewichtigkeit" und „Convention over Configuration" der aktuellen JEE angepasst hat. Abschließend wird erläutert, wieso sich JSF 2.0 deutlich reifer als der Vorgänger JSF 1.2 präsentiert und deshalb dem Entwickler in Zukunft deutlich weniger Kopfzerbrechen bereiten wird.

Vorteile des Standards

JSF ist ein Standard. Auf IT-Konferenzen und in Foren bzw. Blogs wird immer wieder diskutiert, ob das wirklich ein Vorteil oder nur als Marketing-Gag verwendbar ist. Aus Sicht des Autors ist dies ein nicht zu unterschätzender Vorteil. Die Standardisierung führt zu einer sehr hohen Verbreitung und diese bringt viele Vorteile mit sich. Die Anzahl an Ressourcen (Bücher, Blogs, Tutorien) ist vielfach größer als bei konkurrierenden Web-Frameworks im Java-Umfeld. Auch auf der Suche nach neuen Mitarbeitern mit Vorkenntnissen und Erfahrungen hat man deutlich bessere Chancen. Darüber hinaus existieren viele Erweiterungen (Erläuterungen hierzu im zweiten Artikel), welche durch die große Community bereitgestellt werden.

Architektur

Die Architektur von JSF ist in Abbildung 1 dargestellt und enthält die folgenden Bestandteile:

- Das *Faces-Servlet* ist der Einstiegspunkt für die JSF-Implementierung, welches jede Anfrage des Clients abfängt.
- Optional sind zusätzliche Konfigurationsmöglichkeiten in der Datei „faces-config.xml" verfügbar.
- Jede Seite (*Page*) besteht aus mehreren Komponenten (*Component*).
- Der *Renderer* ist für das Anzeigen einer Komponente verantwortlich. Dabei kann aus mehreren View Declaration Language (VDL), wie beispielsweise HTML, XUL oder WML, gewählt werden.
- Der *Converter* wandelt die Werte der Komponenten (z. B. Date oder Integer) von und zu Markup-Werten (in der Regel String) um.
- Ein *Validator* überprüft, ob der vom Nutzer eingetragene Wert erlaubt ist. Dabei ist auch der JEE 6-Standard „Bean Validation" nutzbar (siehe weiter unten).
- Die *Managed Bean* enthält die Anwendungslogik oder Referenzen auf diese (z. B. durch EJBs oder Webservices) und optional die Logik für Navigation zu anderen Seiten.

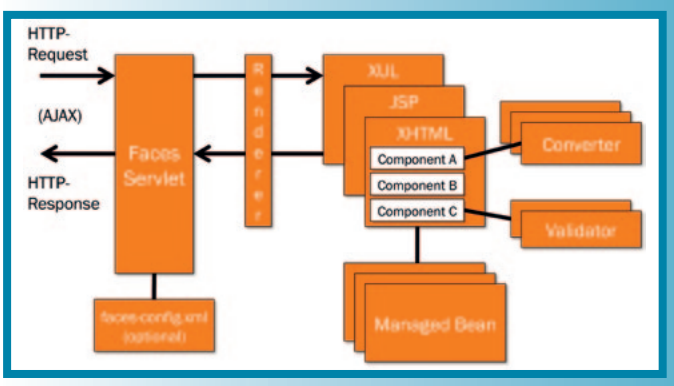

Abb. 1: Architektur von JSF

Architekturmuster: Model View Controller

Model View Controller (MVC) ist ein Architekturmuster zur Strukturierung der Softwareentwicklung in die drei Einheiten

- Datenmodell (*Model*),
- Präsentation (*View*) und
- Programmsteuerung (*Controller*).

Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Eine ausführliche Erläuterung anhand eines Java-Beispiels ist unter [MVC] verfügbar. JSF basiert auf diesem Muster.

Zusammenhang zwischen der JSF-Architektur und MVC

Das MVC-Muster ist direkt in der Architektur von JSF ersichtlich:

- Das Model wird durch die Managed Bean realisiert.
- Die View wird durch die Pages und deren zugehörige Komponenten und Renderer meistens in HTML dargestellt.
- Die JSF-Implementierung agiert durch das Faces-Servlet und die optionale Konfigurationsdatei „faces-config.xml" als Controller und verbindet die View mit dem Model. Auch in der Managed Bean können direkt Controller-Funktionen umgesetzt werden, beispielsweise die direkte Navigation zu einer anderen Seite.

Lebenszyklus von JSF

Bei jedem neuen Request, beispielsweise durch das Klicken eines Buttons, wird der Lebenszyklus von JSF neu gestartet. Diesem Umstand widmen Einsteiger oft erst einmal keine Beachtung. Solange alles gut verläuft, ist dies auch nicht notwendig. Aber spätestens bei den ersten Problemen und Fehlermeldungen ist der Entwickler froh, wenn er versteht, was die JSF-Implementierung vollbringt. Der Lebenszyklus ist in Abbildung 2 dargestellt und besitzt sechs Phasen:

- *Restore View*: Erzeugt den initialen Komponentenbaum oder stellt diesen wieder her, falls die angeforderte Seite schon einmal angezeigt wurde. Wenn keine Request Values vorhanden sind, d. h. Parameter, die vom Client übertragen werden, wird direkt zur Phase Render Response gesprungen.
- *Apply Request Values*: Ordnet jeder Komponente die zugehörigen Request Values zu.

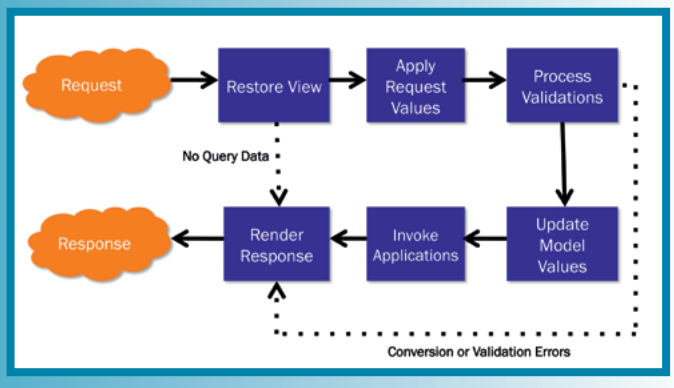

Abb. 2: Lebenszyklus von JSF

- *Process Validations*: Konvertiert und validiert die Request Values. Scheitert diese Phase aufgrund eines Fehlers, wird direkt die Phase Render Response aufgerufen.
- *Update Model Values*: Aktualisiert die Beans mit den nun garantiert validen Request Values, welche an die Komponenten gebunden sind.
- *Invoke Applications*: Führt die action-Methode des Aufrufers (z. B. eines Buttons) aus. Dabei kann beliebige Anwendungslogik ausgeführt werden. Die Rückgabe ist ein String, welcher an den Navigation Handler übergeben wird, um die Antwortseite anzuzeigen.
- *Render Response*: Erzeugt die Antwort und sendet sie an den Webbrowser.

Design-Konzepte von JSF

JSF ist *komponentenorientiert* und abstrahiert dadurch vom Anfrage-/Antwort-Konzept von Servlets, auf dem JSF aufbaut. Die Webanwendung wird als eine Sammlung von Komponenten behandelt, die jeweils eine eigene Darstellung und eigene Aktionen besitzen. Hierdurch wird insbesondere die Erstellung von eigenen Komponenten und deren Wiederverwendbarkeit deutlich erleichtert. Konkret sichtbar wird dieses Konzept bei JSF schon in der Architektur und während der Entwicklung bei der Erstellung von Kompositkomponenten (engl. Composite Components).

Mit JSF können sehr einfach Webanwendungen mit *Multi-Page-Struktur* erstellt werden. Diese besitzen mehrere Seiten. Der Nutzer navigiert mit Hilfe von Hyperlinks. Dadurch sind typische „Browser-Features" wie Lesezeichen oder Vor- und Zurücknavigation ohne großen Zusatzaufwand realisierbar. Ein gutes Beispiel dafür ist Amazon. Gerade mit JSF ist dieses Konzept sehr einfach durch die Navigation Handler und Rückgabewerte in Managed Beans realisierbar. Durch „View Parameter" und „GET Request Links" stellen auch Lesezeichen im Gegensatz zu früheren Versionen kein Problem mehr dar.

Die Standard-Implementierung von JSF ist ein *Server-zentrisches* Web-Framework. Die Darstellung der Oberfläche im Webbrowser wird dabei auf dem Server erstellt, an den Client übertragen und dort dargestellt. Dies bedeutet, dass die Darstellung bei jeder Anfrage erneut vollständig vom Server auf den Client übertragen werden muss. Dieser Ansatz ist sinnvoll, wenn aus Sicherheitsgründen oder wegen eines leistungsschwachen Clients keine Steuerungs- und Darstellungslogik an den Client ausgelagert werden kann. Bei JSF wird bei jeder neuen Anfrage des Clients das darzustellende HTML vom Server generiert und als Antwort an den Client geschickt, wo es dann im Webbrowser nur noch dargestellt wird.

Die grundsätzlichen Design-Konzepte von JSF 2.0 haben sich gegenüber JSF 1.2 nicht verändert. Dennoch sind einige konzeptionelle Neuigkeiten erwähnenswert, die die Entwicklung mit JSF deutlich vereinfachen.

Leichtgewichtigkeit

Bei Version 6 der JEE stand nicht das Hinzufügen neuer Funktionen an erster Stelle, sondern die Leichtgewichtigkeit und weitere Vereinfachung bei der Entwicklung, die schon bei JEE 5 begonnen hat. Folgende Merkmale zeigen die Leichtgewichtigkeit von JSF im Vergleich zu seinen Vorgängern:

- Auch JSF unterstützt nun wie die anderen JEE-Spezifikationen den Einsatz von *Annotationen* (insbesondere für Ma-

naged Beans, Validatoren, Converter und Navigation) und setzt eine XML-Konfiguration nicht mehr zwingend voraus.

- *Dependency Injection* ist ein weiteres wichtiges Feature seit JEE 5 und kann mit einem Satz beschrieben werden: *„Dependency injection means giving an object its instance variables“* [DI]. Mit `@ManagedProperty(value="#{userService}"` können bei JSF 2.0 andere Managed Beans injiziert werden. Dadurch wird der Quellcode besser lesbar sowie einfacher testbar. Zudem werden die Abhängigkeiten reduziert. Stattdessen kann aber auch direkt der deutlich mächtigere JEE 6-Standard CDI genutzt werden (siehe weiter unten).
- *Project Stages* ermöglichen das Festlegen des Status der eingesetzten Webanwendung. Dadurch kann die JSF-Implementierung ihr Verhalten an den jeweiligen Status anpassen. Zur Auswahl stehen „Development“, „UnitTest“, „SystemTest“ und „Production“. Beispielsweise werden in der Rolle „Development“ automatisch mit Hilfe von JavaScript Warnhinweise in der Oberfläche der Webanwendung angezeigt. Dies hat zwar auch Performance-Einbußen zur Folge, aber dafür fällt die Fehlersuche deutlich einfacher.

Convention over Configuration

Convention over Configuration ist ein weiterer wichtiger Ansatz, der in JSF 2.0 Berücksichtigung findet. Fast jede Konfiguration verwendet eine Default-Einstellung, falls nichts explizit angegeben wird. Einige Beispiele:

- Der Name der Managed Bean wird implizit aus dem Klassenname hergeleitet. Die Klasse `EinstellungenBean.java` erhält daher den Namen `einstellungenBean`, sofern nicht explizit ein anderer Name angegeben wird.
- Die früher notwendige Konfiguration der Navigation Rules in der Datei „faces-config.xml“ ist nun optional. Die Navigation kann auch implizit in der Managed Bean im Java-Code implementiert werden. Um beispielsweise auf die Seite „Einstellungen.xhtml“ zu wechseln, ist die Anweisung `return Einstellungen` (ohne „.xhtml“) ausreichend.
- Ressourcen-Management für Builder, CSS, JavaScript und weitere Ressourcen erfolgt standardisiert durch die Einführung der ResourceHandler-API. Bisher musste das bei jeder eigenen oder externen Komponenten-Bibliothek proprietär gelöst werden. Für den Pfad wird implizit ein Default-Wert für Ressource-Bundles verwendet. Nun können Ressourcen einfach in das JAR-Archiv der Anwendung integriert werden, ohne dass diese Artefakte mit zusätzlichen Servlets, Servlet-Filtern oder Phase-Listenern zu laden sind.

Kombination/Integration mit anderen JEE 6 APIs

In JEE 6 ist die Integration der einzelnen Spezifikationen zueinander deutlich verbessert worden. Für JSF sind insbesondere die *Context and Dependency Injection (CDI)* des JSR 299 und *Bean Validation* des JSR 303 von Bedeutung.

CDI ermöglicht den Einsatz von Dependency Injection inklusive Typsicherheit und zustandsbehafteter Kommunikation durch den Conversation-Scope über alle Schichten hinweg. Dadurch können Enterprise-JavaBeans oder auch andere Java-Ressourcen mit einer Annotation injiziert werden. Durch das Service Provider Interface (SPI) von CDI können weitere Funktionalitäten wie beispielsweise zusätzliche Scopes hinzufügt werden. Der Einarbeitungsaufwand für CDI ist aber nicht zu unterschätzen. Zu beachten sind Überschneidungen bei den Scopes. `@Named` macht die `@ManagedBean`-Annotation von JSF sogar überflüssig.

Deutlich einfacher ist die Integration des Standards „Bean Validation“. JSF besitzt zwar eigene Validatoren, beispielsweise für die Überprüfung der Länge eines Feldes oder durch den Einsatz von regulären Ausdrücken. Ein JSF-Validator ist allerdings konkret an eine Seite gebunden und erfordert weitere Validierungen in anderen Schichten.

Bean-Validation ermöglicht hingegen eine standardisierte und Schichten übergreifende Validierung. Durch Validierungsgruppen ist außerdem eine situationsbedingte Prüfung möglich. Beispielsweise wäre beim Registrieren für einen Online-Shop nur E-Mail-Adresse und Passwort notwendig, vor dem Kauf müssen aber auch Adresse und Zahlungsinformationen gespeichert werden. Bean-Validation besitzt mit den sogenannten Constraints eine solide Grundlage an Überprüfungsmöglichkeiten. Weitere eigene Validatoren können selbst entwickelt werden.

Höherer Reifegrad

Facelets werden nun als View-Technologie empfohlen, JSPs sind nur noch wegen der Rückwärtskompatibilität enthalten. Facelets sind XHTML-konform und bieten gegenüber JSPs einige Vorteile:

- Der Lebenszyklus passt zu dem von JSF mit seinen sechs Phasen. JSPs nutzen hingegen den Request-/Response-Ansatz von Servlets, da aus der JSP vor dem ersten Aufruf durch den Benutzer vom JSP-Compiler ein Servlet generiert wird, auf das dann zugegriffen wird.
- Die Erstellung eigener Komponenten und deren Wiederverwendbarkeit werden durch XHTML stark vereinfacht.
- Vorlagen sind durch den Template-Mechanismus einfach realisierbar.
- Eigene Komponenten können durch Kompositkomponenten einfach erstellt werden. Die Wiederverwendung (auch in anderen Projekten) ist simpel, da sie in einem JAR-Archiv zusammengefasst werden können. JSF ist zwar schon immer komponentenbasiert, aber die View-Technologie JSP wurde nie dafür konzipiert.
- Durch detaillierte und präzise Fehlermeldungen sind deutlich bessere Debugging-Möglichkeiten als bei JSPs verfügbar.
- Facelets bieten eine bessere Performance als JSPs, da kein Java-Bytecode generiert und kompiliert werden muss.

Scopes dienen der sauberen Abgrenzung des Lebensraums von Managed Beans. Die Erweiterung der bisherigen Scopes um den zusätzlichen View-Scope, die Möglichkeit zur Erstellung von Custom-Scopes sowie insbesondere die Integration weiterer Scopes durch CDI ermöglichen eine deutlich einfachere Lösung von im Alltag auftretenden Problemen bezüglich der Zustandsverwaltung von Managed Beans ohne störende Workarounds.

Die *standardisierte Integration von Ajax* vereinfacht dessen einheitliche Benutzung. Bisher musste jede Erweiterung eigene Lösungen dafür realisieren – diese waren dann jedoch zueinander inkompatibel.

Fazit

JSF hat sich durch den Standard im Java-Umfeld etabliert. Mit dem Versionssprung auf Version 2.0 hat sich die Entwicklung dank Leichtgewichtigkeit, Konventionen, höherem Reifegrad und einfacher Integration in den restlichen JEE 6-Standard deutlich vereinfacht. Nun macht es wirklich Spaß, mit JSF Webanwendungen zu erstellen. Auch Kritikern, die JSF 1.2 (berechtigterweise) verworfen haben und sich für ein anderes Web-

Framework wie Wicket, Tapestry oder Spring MVC entschieden haben, sei deshalb empfohlen, JSF 2.0 eine Chance zu geben. Der Reifegrad und der Komfort bei der Entwicklung einer Webanwendung sind in keiner Weise mehr mit der Vorgängerversion vergleichbar. Der Maintenance-Release JSF 2.1 steht ebenfalls schon in den Startlöchern, um der Spezifikation noch weiteren „Feinschliff" zu verpassen [JSF21].

Ausblick

JSF an sich ist nur der grundlegende „Baustein", auf dem viele zusätzliche Erweiterungen aufbauen, die den Entwicklungsaufwand noch weiter minimieren. Diese Erweiterungen werden ausführlich im Artikel der nächsten Ausgabe vorgestellt. Dadurch wird das enorme Potenzial von JSF aufgezeigt. Aber auch die Grenzen von JSF sollen nicht verschwiegen werden. Diese beleuchtet der dritte Teil dieser Artikelreihe.

Literatur und Links

[DI] J. Shore, Dependency Injection Demystified, 22.3.2006, `http://jamesshore.com/Blog/Dependency-Injection-Demystified.html`

[GeaHor10] D. Geary, C. S. Horstmann, Core JavaServer Faces, 3. Auflage, Sun Microsystems Press book, 2010

[JSF21] A. Bosch, Was ist neu in JSF 2.1, 29.11.2010, `http://it-republik.de/jaxenter/news/Was-ist-neu-in-JSF-2.1-057653.html`

[JSFJavaMagazin] JSF 2.0 – Next Generation Enterprise Web Development, JavaMagazin Spezial (JEE 6), Dezember 2010

[JSR314] Java Specification Request 314: JavaServer Faces, 2.0, `http://jcp.org/en/jsr/detail?id=314`

[MVC] R. Eckstein, Java SE Application Design With MVC, 2007, `http://www.oracle.com/technetwork/articles/javase/mvc-136693.html`

Kai Wähner ist als IT-Consultant bei *MaibornWolff et al* tätig. Seine Schwerpunkte liegen in den Bereichen JEE, EAI und SOA. Außerdem berichtet er in seinem Blog über seine Erfahrungen mit neuen Technologien, IT-Konferenzen und Zertifizierungen.
E-Mail: kai.waehner@mwea.de