

## Aufbau einer Webanwendung mit JavaServer Faces (JSF)

# Vokabeltraining

■ VON ANDY BOSCH

Mit JavaServer Faces steht der Entwicklergemeinde seit knapp zwei Jahren ein neues UI-Standard zur Verfügung. Mit JSF sollen Anwendungsentwickler schnell professionelle Anwendungen im Webumfeld realisieren können. An dieser Stelle werden wir eine Beispiel-Webanwendung schrittweise mit JSF umsetzen. Der Fokus ist dabei auf die Vorgehensweise, die verschiedenen Arbeitsschritte, aber auch auf Besonderheiten und Architekturempfehlungen beim Einsatz von JSF gerichtet.

Die zu erstellende Anwendung stellt einen Vokabeltrainer dar. Es soll eine webbasierte Erfassung von (englischen) Vokabeln sowie eine gezielte Abfrage möglich sein. Zu diesem Zweck wird ein Karteikastensystem umgesetzt. Dieses ist so aufgebaut, dass neue Vokabeln zunächst in Fach eins landen. Wenn diese bei einer Abfrage korrekt übersetzt wurde, landet sie in Fach zwei usw. Maximal gibt es sechs Fächer. Wird eine Vokabel falsch übersetzt, landet sie wieder in Fach eins. Aus Platzgründen sind nicht sämtliche Quelltexte im Artikel abgedruckt. Die komplette Anwendung ist jedoch auf der Heft-CD beigelegt und kann beliebig eingesetzt und erweitert werden.

Als Entwicklungsumgebung wird Exadel Studio verwendet. Es ist in der Standard Edition kostenfrei zu beziehen. Doch selbst die Professional Version (aktuell in der Version 3.0) ist mit 99 US-Dollar durchaus preiswert. Exadel Studio setzt auf der Eclipse-Plattform auf, aktuell wird bereits Eclipse 3.1 unterstützt. Die zu entwickelnde Anwendung benötigt natürlich zur Speicherung der Vokabeln eine Datenbank im Hintergrund. Dafür

wird MySQL eingesetzt, als Persistenz-Framework kommt Apache Torque zum Einsatz. Die Anwendung verwendet keine EJBs, läuft somit vollständig in einem Servlet Container (Tomcat 5.x). In diesem Artikel wird jedoch ausschließlich auf JSF eingegangen; Fragestellungen zu Torque, MySQL etc. werden ausgeblendet.

### Vorgehensweise

Auch bei der Umsetzung einer Anwendung mit JSF empfiehlt es sich, zunächst basierend auf den UseCases einen Prototypen in HTML zu bauen. Gerade wenn die Anwendung mit einem Fachbereich abgestimmt werden muss, ist eine Detailabstimmung basierend auf den HTML-Masken eine sehr gute Möglichkeit, eventuelle Missverständnisse zwischen Fachbereich und Entwicklung zu klären. Da in diesem Beispiel jedoch der Schwerpunkt auf JSF selbst gelegt werden soll, werden der HTML-Prototyp sowie der darin enthaltene Screenflow (der Steuerungsfluss durch die Anwendung) als „abgestimmt und abgenommen“ angesehen. Die weiteren Arbeitsschritte sind sodann:

- Aufbau der JSF-Seiten, basierend auf den HTML-Prototypen
- Einbau der Navigationsregeln, basierend auf dem Screenflow-Diagramm
- Einbau von Konvertern, Validatoren und Festlegung des Fehlerbehandlungskonzepts
- Einbau der Businesslogik
- Umsetzung der benutzerdefinierten Komponente

### Definition der Beans

JSF arbeitet nach dem Konzept der Managed Beans. Dies bedeutet, dass auf deklarativem Wege JSF diejenigen Beans mitgeteilt werden, die in den JSF-Seiten zum Einsatz kommen. JSF sorgt dann per Lazy Loading selbst dafür, dass im entsprechenden Kontext die Beans zur Verfügung stehen. Für den Vokabeltrainer werden Beans (die das Modell gemäß Model View Controller bilden) benötigt, die ein Wort, eine Kategorie sowie die Statistik für einen Bereich abbilden. Managed Beans in JSF unterliegen der JavaBean-Spezifikation, sind somit normale Java-Klassen mit parameterlosem Konstruktor, privaten Member-Variablen und öffentlichen Getter- und Setter-Methoden. Als sehr positiv in JSF ist zu bemerken, dass Managed Beans reine POJOs (Plain Old Java Objects) sein können, also keine Im-

- Definition der Beans, die das Modell der Anwendung darstellen und die Werte aus der Oberfläche speichern



Quellcode auf CD

porte von JSF benötigen. Da diese Beans auch in der Datenbank gespeichert werden müssen, werden in diesem Beispiel die JSF Managed Beans mittels Torque persistiert. Dieser Ansatz hat sicherlich den Nachteil, dass hier keine saubere Schichtentrennung zwischen Datenbank- und Präsentations-Layer vorhanden ist, andererseits wird einem Entwickler mit diesem Ansatz eine Menge Arbeit abgenommen, da eine Speicherung der Beans von der Oberfläche in die Datenbank so gut wie ohne Programmieraufwand vorgenommen werden kann. Einen Auszug aus der Anwendungskonfigurationsdatei *faces-config.xml* mit den hinterlegten Beans zeigt Listing 1.

## Aufbau der JSF-Seiten

Die Umsetzung der Seiten des HTML-Prototyps lässt sich am besten mit einem WYSIWYG-Editor vornehmen, in dem HTML Tags durch entsprechende JSF Tags ersetzt werden. Da in Exadel Studio jedoch kein grafischer Editor vorhanden ist, ist an dieser Stelle ein wenig Handarbeit gefragt. Die entsprechenden Ein- und Ausgabekomponenten werden durch

JSF Tags abgebildet, wobei an dieser Stelle auch gleich das ValueBinding gesetzt wird. Durch ein ValueBinding wird z.B. bei einer Eingabekomponente (einem Eingabefeld) der Wert des Feldes mit dem Wert einer Managed Bean gekoppelt. So wird z.B. über

```
<h:inputText value="#{WordBean.answer}" />
```

ein Eingabefeld an die Variable *answer* im *WordBean* gekoppelt. Damit sorgt JSF selbst dafür, dass beim Aufbau der Seite die Werte des jeweiligen Objektes angezogen und bei einem Verlassen der Seite die Werte des Feldes wieder in das entsprechende Objekt zurückgespeichert werden. Über das ValueBinding ist auch in der Programmierung ein Zugriff auf die Managed Beans möglich. So kann über

```
FacesContext context = FacesContext.getCurrentInstance();
ValueBinding binding = context.getApplication().
    createValueBinding("#{WordList}");
WordList wordList = (WordList)binding.getValue(context);
```

auf die aktuelle Vokabelliste zugegriffen werden, die momentan in JSF hinterlegt

Anzeige

### Listing 1

#### Deklaration der Managed Beans in der *faces-config.xml*

```
<managed-bean>
  <managed-bean-name>WordBean </managed-bean-name>
  <managed-bean-class>com.b2b.wordtrain.bean.Word </managed-bean-class>
  <managed-bean-scope>session </managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>WordList</managed-bean-name>
  <managed-bean-class>com.b2b.wordtrain.bean.WordList</managed-bean-class>
  <managed-bean-scope>session </managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>CategoryList</managed-bean-name>
  <managed-bean-class>com.b2b.wordtrain.bean.CategoryList</managed-bean-class>
  <managed-bean-scope>session </managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>CategoryBean </managed-bean-name>
  <managed-bean-class>com.b2b.wordtrain.bean.Category </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>StatisticBean</managed-bean-name>
  <managed-bean-class>com.b2b.wordtrain.bean.Statistics</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

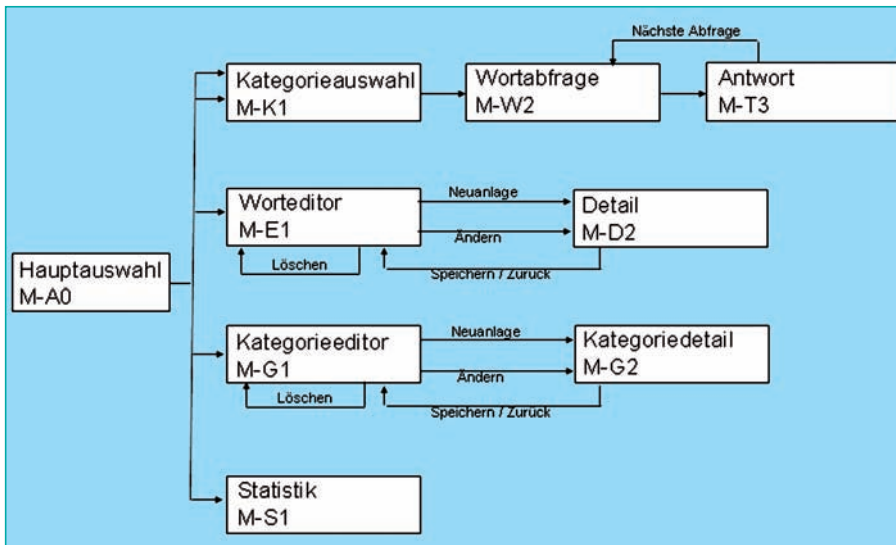


Abb. 1: Screenflow im Vokabeltrainer

ist. Diese Form, per Programmierung auf Managed Beans zuzugreifen, wird üblicherweise in Helper-Klassen ausgelagert. Im konkreten Beispiel wird der Aufruf in eine Bean-Fassade ausgelagert, worauf später detaillierter eingegangen wird.

### Konvertierung und Validierung

In den meisten Fällen sorgt JSF selbst für die richtige Konvertierung der Datentypen. Im Browser liegen üblicherweise in den Eingabefeldern keine Datentypen vor, im Java-Modell dagegen sind verschiedene Datentypen logischerweise bekannt.

Es erfolgt somit eine regelmäßige Wandlung von der Modell- in die Präsentationssicht und umgekehrt. Diesen Mechanismus unterstützt JSF dahingehend, dass es bereits zahlreiche Standardkonverter für die primitiven Datentypen (*integer*, *boolean*, *float* ...) gibt. Im konkreten Beispiel kann eine Vokabel editiert und dabei der Zähler für die richtigen und falschen Übersetzungen beim Training verändert werden. Diese Zahl wird als *int* im Datenmodell abgespeichert. Dennoch ist bei der Umsetzung in JSF keine explizite Angabe eines Converters notwendig. Soll z.B. eine Zahl in eine *int*-Variable gesetzt werden, geschieht dies über

```
<h:inputText value="#{WordBean.trayG}" size="40" />
```

Der Anwendungsentwickler muss sich hier nicht um die korrekte Konvertierung (von der Präsentation als String in das Datenmodell als *int* und umgekehrt) kümmern.

Für die Datenvalidierung existieren ebenfalls einige Standardvalidatoren, wie z.B. eine Prüfung auf eine Pflichteingabe oder die Prüfung einer Mindesteingabelänge oder Eingabe eines Wertes innerhalb eines definierten Wertebereichs. Im Folgenden wird festgelegt, dass bei den Eingabefeldern für eine neue Vokabel für den deutschen und englischen Text eine Eingabe erforderlich ist (Attribut *required="true"*) und dass eine Eingabe mindestens drei Zeichen lang sein muss (*LengthValidator*).

```
<h:inputText required="true" value="#{WordBean.german}" size="40">
  <f:validateLength minimum="3" />
</h:inputText>
```

Damit eventuelle Konvertierungs- und Validierungsfehler angezeigt werden, muss auf der jeweiligen Seite ein entsprechender Bereich dafür festgelegt werden. Dies geschieht über die Einbindung des Tags *<h:messages />*. Liegen keine Fehlermeldungen vor, erzeugt das Tag keine Ausgabe, im Fehlerfall werden an dieser Stelle die Fehler der Seite angezeigt. JSF liefert in der Referenzimplementierung bereits deutsche Fehlertexte mit aus. Es empfiehlt sich jedoch, diese Fehlertexte vor der Auslieferung der Anwendung an Kunden komplett zu überschreiben (oder gefällt Ihnen beispielsweise die Fehlermeldung: „Spezifiziertes liegt nicht im definierten Bereich“?). Um Fehlertexte zu überschreiben, genügt es, ein Resource Bundle in die Anwendung mit einzubinden und die Fehler-ID mit einem neuen Text zu versehen. Im Folgenden wird der Ausschnitt der Datei *faces-config.xml* gezeigt, in der zunächst das Resource Bundle definiert wird. Gemäß Java-Konventionen können damit im Klassenpfad die entsprechenden Properties-Dateien in den jeweiligen Sprachen hinterlegt sein.

```
<application>
  <message-bundle>app-essages</message-bundle>
</application>
```

Anschließend kann der Fehlertext überschrieben werden. Im Folgenden wird ein Ausschnitt einer Properties-Datei gezeigt, in der die Fehlermeldung für die Required-Angabe und für die Längenprüfung überschrieben wurde.

```
javax.faces.validator.LengthValidator.MINIMUM=
  Der eingegebene Text ist zu kurz. Bitte geben sie
  mindestens {0} Zeichen ein.
javax.faces.component.UIInput.REQUIRED=
  Bitte geben Sie einen Wert ein.
```

### Festlegung der Navigationsregeln

Entgegen der Vorgehensweise in reiner JSP-Programmierung, die Navigation direkt in den JSP-Seiten zu hinterlegen, existiert in JSF ein Konzept zur Defi-

### Listing 2

#### Festlegung der Navigationsregeln in der *faces-config.xml*

```
<navigation-rule>
  <from-view-id>/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>selectcategory</from-outcome>
    <to-view-id>/selectkat.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>wordeditor</from-outcome>
    <to-view-id>/vokeditor.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cateditor</from-outcome>
    <to-view-id>/kateditor.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>statistics</from-outcome>
    <to-view-id>/statistik.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

nition der Navigationsregeln in einer separaten Konfigurationsdatei. Dabei können Navigationsregeln und Navigationsfälle in einer XML-Syntax hinterlegt werden. Die meisten IDEs unterstützen dabei bereits den grafischen Aufbau der Navigationsregeln. Eine Navigationsregel bezieht sich dabei meist auf eine Ausgangsseite, auf der verschiedene Aktionen ausgelöst werden können (z.B. Drücken eines Buttons oder Betätigen eines Hyperlinks). Die Folgeaktionen sind in den jeweiligen Navigationsfällen enthalten. So kann eine Navigationsregel mehrere Navigationsfälle enthalten. Im Auszug in Listing 2 ist eine Navigationsregel inklusive der fünf Fälle beschrieben, die sich auf die Startseite des Vokabeltrainers beziehen. Um zunächst festzulegen, welche Regeln benötigt werden, empfiehlt sich der Einsatz von Screenflow-Diagrammen (Abb. 1).

Zum Auslösen einer Aktion kann entweder direkt an einer Komponente ein Bezeichner hinterlegt sein, man spricht dabei von statischer Navigation. Z.B. wird durch `<h:commandLink action="home" />` direkt in der `faces-config.xml` ein Navigationsfall mit dem Wert `home` gesucht. Neben der statischen Navigation ist es möglich, eine dynamische Navigation über Aktionsmethoden abzubilden. So könnte z.B. bei einem Anmeldeformular über den LOGIN-Button `<h:commandButton value="Login" action="#{LoginHandler.checkLogin}" />` in einer Methode ein Ergebnis `success` oder `failed` zurückgeliefert werden, das wiederum über die Navigationsregeln auf die jeweilige Seite verweist.

## Business-Logik

JSF ist ein reines UI-Framework. Dies bedeutet, dass es streng genommen keine Stelle gibt, in der Business-Logik integriert werden kann. In größeren Projekten ist es daher häufig der Fall, dass die Business-Logik beispielsweise in EJBs umgesetzt ist. Die Verbindung zwischen JSF und den EJBs erfolgt dann über die Aktionsmethoden. Hierin kann ein (entfernter) Aufruf zu den Business-Methoden erfolgen. Im konkreten Beispiel des Vokabeltrainers ist die gesamte Business-Logik in den Handler-Klassen enthalten. Dies könnte

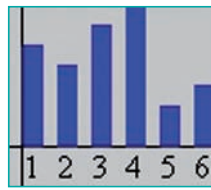


Abb. 2: Benutzerdefinierte Komponente

bei einer komplexeren Logik sicherlich noch ausgelagert werden.

## Benutzerdefinierte Komponenten: die Statistikkomponente

Ein wichtiges Thema bei jedem Vokabeltrainer sind natürlich die Statistiken. Wie viele Vokabeln befinden sich in welchem Fach, wie viele Fragen wurden falsch oder richtig beantwortet? Dafür soll eine eigene Komponente entworfen werden. Der Vorteil an JSF ist, dass die Erweiterung des Frameworks um eigene benutzerspezifische Komponenten recht problemlos zu realisieren ist. Dabei können entweder komplett neue (neuartige) Komponenten entwickelt oder vorhandene Komponenten erweitert werden. Die Ausgabe, die wir für den Vokabeltrainer benötigen, soll zunächst lediglich eine Balkengrafik anzeigen, die die Anzahl der Vokabeln in jedem Fach grafisch darstellt (Abb. 2). Um dies zu realisieren, ist folgende Vorgehensweise notwendig:

1. Entwickeln einer Tag-Handler-Klasse
2. Aufbau eines TLD (Tag Library Descriptor)
3. Entwicklung der Komponentenkasse
4. Programmierung des Renderers
5. Eintragen der neuen Komponente und des Renderers in der `faces-config.xml`

Die Tag-Handler-Klasse ist erster Ansprechpartner bei der Verwendung benutzerspezifischer Tags. Sie steuert, dass die richtige Komponente und der jeweilige Renderer zusammengezogen und die Attribute des Tags korrekt in die Komponente übergeben werden. In Listing 3 ist ein Ausschnitt der Tag-Handler-Klasse abgebildet.

Im der Methode `setProperty` werden die Attribute des Tags in die Komponente gesetzt. Dabei wird der Ausdruck auf ein ValueBinding abgefragt. Dies hat den Hintergrund, dass wir die Statistikkomponente (bzw. das Tag) mit einem Value-

Anzeige

Binding-Ausdruck aufrufen werden. Konkret werden wir das Tag später mittels

```
<cu:statistics entrylist="#{GlobalStatisticBean.traylist}"/>
```

befüllen. Somit muss aus dem ValueBinding zunächst der konkrete Wert (in diesem Falle eine Liste) entnommen werden, der dann wiederum in die Komponente gesetzt wird. Der Tag Library Descriptor enthält die Beschreibung des neuen Tags. Darin ist unter anderem festgelegt, wel-

che Attribute das Tag akzeptiert, ob diese Muss-Attribute sind und eventuell Runtime Expressions erlauben.

Die Komponentenkategorie ist sehr einfach aufgebaut. Die Statistikkomponente erbt dabei von *UIOutput*, da auch die Statistik lediglich eine Ausgabe erzeugt. Es ist somit für die Komponentenkategorie nicht viel Entwicklungsarbeit notwendig. Wichtig sind allerdings die Methoden *saveState* und *restoreState* (Listing 4). Diese sind für das State Saving zu-

ständig. Die Zustandsspeicherung sorgt dafür, dass die Eigenschaften und Werte der Komponente auch über den Request hinaus gespeichert bleiben und nicht verloren gehen.

Die Renderer-Klasse dagegen enthält die gesamte Logik, die für die Darstellung der Komponente notwendig ist. In den Encode-Methoden werden die konkreten HTML-Tags geschrieben, die im Browser das gewünschte Ergebnis erzielen. Zu guter Letzt werden die neue Komponente und der neue Renderer in der *faces-config.xml* eingetragen (Listing 5). Danach kann das neue Tag in einer JSF-Seite eingebunden werden. Das Einbinden des neuen TLD in die Seite darf natürlich nicht vergessen werden.

### Listing 3

#### Ausschnitt aus der Tag-Handler-Klasse

```
...
public class StatisticsTag extends UIComponentTag {
    private static String RENDERER_TYPE
        = "StatisticsRenderer";
    private static String COMPONENTE_TYPE
        = "StatisticsComponent";
    private String entrylist;
    public String getComponentType() {
        return COMPONENTE_TYPE;
    }
    public String getRendererType() {
        return RENDERER_TYPE;
    }
    ...
    protected void setProperties(UIComponent component)
    {
        super.setProperties(component);
        StatisticsCtrl cmp = (StatisticsCtrl)component;
        if ( isValueReference( entrylist ) ) {
            FacesContext context = FacesContext.
                getCurrentInstance();
            ValueBinding vb = context.getApplication().
                createValueBinding( entrylist );
            cmp.setEntrylist( (List)vb.getValue( context ) );
        } //if
    }
}
```

### Architekturüberlegungen

Mit JSF bekommt der Anwendungsentwickler zwar ein Framework an die Hand, in dem bereits eine Menge Konzepte und Architekturüberlegungen enthalten sind, dennoch empfiehlt es sich, aufbauend auf JSF in der eigenen Anwendung weitere Konzepte zu integrieren, etwa die Trennung von Funktionalität und Beans.

In JSF werden Aktionen über Method-Binding-Aufrufe angestoßen. So wird beispielsweise im Vokabeltraining die Abfrage der nächsten Vokabel mit *#{TrainingHandler.answerQuestionOk}* aufgerufen. Der Ausdruck *answerQuestionOk* ist eine sog. Aktionsmethode im Managed Bean *TrainingHandler*. Technisch ist es jederzeit möglich, Aktionsmethoden in beliebigen Beans auszuführen. Es empfiehlt sich jedoch, die Funktionalität (die durch Aktionsmethoden entsteht), in Handler-Klassen auszulagern. Damit bleiben die Bean-Klassen, das Modell der Anwendung, frei von Funktionalität und kapseln lediglich die Eigenschaften einer Entität. Die Handler-Klassen sind zwar auch als Managed Bean deklariert, sind jedoch die zentralen Ansprechpartner für die Funktionalität einer Anwendung.

Ein weiteres Konzept ist die Verwendung einer *BeanFassade*. Speziell in der Handler-Klasse, in der die Funktionalität der Anwendung enthalten ist, findet ein reger Zugriff auf die Managed Beans statt. Dies geschieht wie bereits gezeigt über das ValueBinding. Um den Zugriff über das

### Listing 4

#### Komponentenkategorie

```
public class StatisticsCtrl extends UIOutput {
    ...
    public void setEntrylist(List entrylist) {
        this.entrylist = entrylist;
    }
    public Object saveState(FacesContext context) {
        Object values[] = new Object[2];
        values[0] = super.saveState(context);
        values[1] = getEntrylist();
        return (values);
    }
    public void restoreState(FacesContext context,
        Object state) {
        Object values[] = (Object[]) state;
        super.restoreState(context, values[0]);
        entrylist = (List)values[1];
    }
}
```

### Listing 5

#### Ausschnitt aus der *faces-config.xml*

```
<component>
  <component-type>StatisticsComponent
  </component-type>
  <component-class>
    com.b2b.wordtrain.component.StatisticsCtrl
  </component-class>
</component>
<render-kit>
  <renderer>
    <component-family>StatisticsFamily
    </component-family>
    <renderer-type>StatisticsRenderer</renderer-type>
    <renderer-class>
      com.b2b.wordtrain.component.StatisticsRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

ValueBinding auf die eigentlichen Objekte zu verbergen und die Anwendung übersichtlicher zu gestalten, wurden im Vokabeltrainer alle ValueBinding-Aufrufe hinter einer BeanFassade versteckt. Somit kann in den Aktionsmethoden jederzeit beispielsweise über *Word a Word* = *BeanFassade.getWord()*; auf die aktuelle Vokabel zugegriffen werden.

### Verbesserungspotenzial

Im Vokabeltrainer wurden bereits einige Konzepte angesprochen und umgesetzt, die bei JSF-Anwendungen sinnvoll sind. Das Thema Logging ist im Beispiel noch nicht behandelt worden, hier könnte beispielsweise log4j integriert werden. Im Hinblick auf die Konfigurationsdatei *faces-config.xml* wäre es auch ratsam, statt einer großen (unübersichtlichen) Datei die Inhalte auf mehrere Dateien zu verteilen. JSF bietet die Möglichkeit, dass mehrere Konfigurationsdateien angegeben werden können und somit auch von Seiten der Konfiguration eine Modularisierung möglich ist.

Ein noch offener Punkt im Vokabeltrainer und in JSF generell ist die Verwaltung der Managed Beans im Session Scope. JSF liefert zwar das Konzept, Beans jederzeit nachladen zu können, jedoch existiert kein Mechanismus, der nicht mehr verwendete Beans wieder entfernt. Dies ist auch so nicht möglich, da nur der Anwendungsentwickler weiß, ab wann im Programmfluss er die Beans nicht mehr benötigt. Hier empfiehlt es sich, eine eigene Routine zu schreiben, die Beans per Aufruf wieder aus der Session entfernt. So kann sichergestellt werden, dass speziell bei komplexeren Anwendungen die Session sich nicht zumüllt und dem Application Server den Hauptspeicher raubt.

### Fazit

Der Vokabeltrainer ist ein Beispiel dafür, wie und mit welchem Aufwand eine Webanwendung mithilfe von JSF und Exadel Studio aufgebaut werden kann. Es wird deutlich, dass JSF bereits zahlreiche Konzepte und Vorgehensweisen für eine effektive Anwendungsentwicklung bietet

und anhand von entsprechenden Tools und IDEs auch ein deutlicher Geschwindigkeitsvorteil in der Entwicklung von Webanwendungen zu erzielen ist. Es ist zudem ersichtlich, dass die Möglichkeit der Erweiterung des Frameworks ein großer Vorteil von JSF ist. So können neue Konverter, Validatoren, Renderer und komplette UI-Komponenten ohne großen Aufwand integriert werden. Jedoch wurde auch gezeigt, dass trotz des umfangreichen Frameworks eigene Ergänzungen und Konzepte notwendig bzw. empfehlenswert sind. Um einen guten Architekten kommt man auch bei Einsatz von JSF nicht herum.

---

**Andy Bosch** ist selbstständiger IT-Berater und Projektleiter. Er beschäftigt sich überwiegend mit der Konzeption und Realisierung von Webanwendungen, seit einiger Zeit fast ausschließlich im Umfeld von JSF. Zudem ist er Betreiber der Plattform [www.jsf-forum.de](http://www.jsf-forum.de).

### ■ Links & Literatur

- [1] JSF Studio: [www.exadel.com](http://www.exadel.com)
- [2] [www.eclipse.org](http://www.eclipse.org)
- [3] [www.jsf-forum.de](http://www.jsf-forum.de)