Spring doch!

Java EE-Entwicklung mit Spring

Martin Kempa

Mehrschichtige J2EE-Anwendungen sind eine weit verbreitete Architektur für betriebliche Informationssysteme. Zum Testen der fachlichen Funktionalität dieser Anwendungen bedarf es eines korrekt konfigurierten Applikationsservers, was die Komplexität der Entwicklungsumgebung erhöht. Dieser Artikel stellt eine einfache Alternative vor, in der für den Zeitraum der Realisierung auf einen Applikationsserver verzichtet wird. Stattdessen wird mithilfe des Spring-Frameworks ein EJB-Container in die Client-Anwendung eingebettet.

Motivation

Betriebliche Informationssysteme, die nach dem Stand der Kunst, beispielsweise mit der Java Platform, Enterprise Edition (Java EE) [Sun06a], realisiert werden, folgen dem Muster der Drei-Schichtenarchitektur [Fow02]. Dabei besteht die Anwendung aus drei aufeinander aufbauenden Schichten mit unterschiedlichen Aufgaben. Typischerweise handelt es sich dabei um folgende Schichten:

- Die *Persistenzschicht* („persistence tier“) sorgt für die dauerhafte Speicherung der verwendeten Daten. In der Regel kommt hier ein Datenbanksystem zum Einsatz.
- Die *Geschäftslogikschicht* („business tier“) kapselt die Logik, die für Funktionen der Anwendung nötig ist. Dabei darf sie auf die Persistenzschicht zugreifen. Als Laufzeitumgebung ist ein Applikationsserver vorgesehen.
- Die *Präsentationsschicht* („presentation tier“) implementiert die Schnittstelle zum Benutzer. Sie sorgt für die Darstellung der Daten, deren Eingabemöglichkeit sowie der Steuerung des Dialogablaufs. Für diese Aufgaben werden die Dienste der Geschäftslogik aufgerufen.

Abb. 1: Drei-Schichtenarchitektur

Bei der Realisierung zeichnet sich die Architektur durch eine komplexe Entwicklungsumgebung aus, mit der jeder Entwickler umgehen muss. So wird für die Entwicklungsumgebung neben dem Datenbanksystem und einer lauffähigen Client-Anwendung ein Applikationsserver benötigt, um Programmänderungen zu verifizieren. Diese Anordnung führt zu einer erhöhten Wartezeit, der sogenannten Turnaround-Zeit, zwischen einer Änderung im Quellcode und deren Wirksamkeit in der Anwendung. Sie besteht im Wesentlichen aus den Zeiten für die beiden Schritte Compilieren des Quellcodes und Deployen im Applikationsserver.

In den ersten Jahren der Java EE-Applikationsentwicklung konnte die Turnaround-Zeit mehrere Minuten betragen und bestand aus den Wartezeiten für das Compilieren und das Deployen. Der Applikationsserver war nicht in die Entwicklungsumgebung integriert und nutzte so nicht die Vorteile des automatischen Compilers in der Entwicklungsumgebung.

Durch die Integration des Applikationsservers (z. B. [JBo08]) in die Entwicklungsumgebung wird dieser Nachteil abgestellt und die Turnaround-Zeit reduziert sich auf die Wartezeit für das Deployment. Inzwischen gibt es sogar ein kommerzielles Produkt [Zer09], das die Technik der Class-Loader-Instrumentierung nutzt, um eine sofortige Wirksamkeit der meisten Programmänderungen zu erreichen. Diese arbeitet auch mit einem Applikationsserver zusammen.

Entwicklung ohne Applikationsserver

In diesem Artikel wird ein anderer Weg, den der Autor bei der IVU Traffic Technologies AG umgesetzt hat, vorgeschlagen. Die Komplexität in der Entwicklung wird reduziert, indem für die Entwicklungszeit auf den Einsatz eines Applikationsservers verzichtet wird. Dadurch entfallen selbstverständlich sämtliche Dienste, die der Applikationsserver bereitstellt, wie beispielsweise ein Container mit Transaktionsmanagment, Persistenz, ein Namensdienst oder Messaging. Diese Funktionalität muss die Entwicklungsarchitektur ohne Applikationsserver geeignet ersetzen.

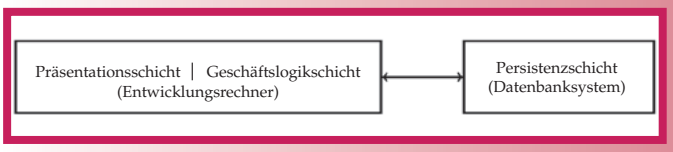

Abb.2: Zweischichtige Entwicklung einer Drei-Schichtenanwendung

Abbildung 2 zeigt diese Entwicklungsarchitektur mit der Integration der Geschäftslogikschicht in die Client-Anwendung. Deren Laufzeitumgebung führt die Geschäftslogikschicht aus und verarbeitet den identischen Programmcode der mehrschichtigen, späteren produktiven Variante mit. Dafür muss die Client-Anwendung um Funktionalität ergänzt werden.

Die folgenden Abschnitte des Artikels zeigen, wie sich die Geschäftslogikimplementierung in Java EE-Technologie in die Client-Anwendung integrieren lässt. Das Spring-Framework liefert dafür den eingebetteten EJB-Container. Die Anbindung zur Persistenz erfolgt mit dem Framework Hibernate und den Namensdienst realisiert eine einfache Spring-Erweiterung. Aus Platzgründen entfällt die Darstellung des Messaging, für das ActiveMQ zum Einsatz kommt.

Spring-Framework als EJB-Container

Die Laufzeitumgebung der Geschäftslogik in der Client-Anwendung stellt das Spring-Framework [Spr08]. Seit der Version 2.5 kann Spring mit EJB-Annotationen umgehen und ist damit für solche Zwecke geeignet. Neben dem Spring-Framework existieren noch mehrere weitere Alternativen für EJB-Container, wie z. B. OpenEJB [Apa09] oder Embedded-JBoss [JBo09], die außerhalb eines Applikationsservers genutzt werden können. Das leistungsstarke und erprobte Framework Spring ist aber sicherlich das mit der größten Verbreitung und Akzeptanz.

Damit Spring als EJB-Container arbeiten kann, müssen sämtliche EJBs mit allen transitiv abhängigen EJBs dem Applikationskontext bekannt sein. Dies geschieht durch die Deklaration der Beans in der Spring-Konfiguration, wofür zwei Varianten zur Verfügung stehen:

- Bei der *expliziten Deklaration* werden alle Session-Beans der Geschäftslogik in der Spring-Konfiguration deklariert:

```
<bean id="EmployeeFacadeBean"
  class="de.ivu.employee.ejb.EmployeeFacadeBean"/>
```

- Bei der *Deklaration per Komponenten-Scan* werden alle Session-Beans der Geschäftslogik einer Komponente durch einen automatischen Scan in der Spring-Konfiguration deklariert:

```
<context:component-scan base-package="de.ivu.employee.ejb"
  annotation-config="false"
  name-generator=
    "de.ivu.util.spring.EJBAnnotationBeanNameGenerator">
  <context:include-filter type="annotation"
    expression="javax.ejb.Stateless"/>
</context:component-scan>
```

Da die Spring-Konfiguration bei einer expliziten Deklaration schnell umfangreich wird, ist eine Modularisierung sinnvoll. So kann beispielsweise jede Komponente der Geschäftslogik eine Konfigurationsdatei, die die expliziten Deklarationen der eigenen Session-Beans enthält, beitragen.

Der Komponenten-Scan hat den Nachteil, dass er beim Applikationsstart etwas mehr Zeit benötigt als die explizite Deklaration. Dafür entfällt die mühselige Pflege der expliziten Deklaration in der Spring-Konfigurationsdatei.

Etwas Sorgfalt ist bei der Wahl der Bean-Bezeichner notwendig, da es unterschiedliche Namenskonventionen gibt. So verwendet Spring standardmäßig den Klassennamen mit kleinem Anfangsbuchstaben als Bezeichner. Demnach bezeichnet der Name **`employeeFacadeBean`** die Bean der Klasse **`de.ivu.employee.ejb.EmployeeFacadeBean`**. Der Applikationsserver JBoss hingegen verwendet als Konvention den Klassennamen mit großem Anfangsbuchstaben, was zur Bezeichnung **`EmployeeFacadeBean`** führt.

Im Programmcode der Geschäftslogikschicht kann es durchaus Stellen geben, die auf die Bean-Bezeichner verweisen, beispielsweise in den Attributen **`mappedName`** oder **`beanName`** der Annotationen **`Resource`** und **`EJB`**. Da der Programmcode nicht verändert werden soll, muss die Namenskonvention des Applikationsservers berücksichtigt werden. Die explizite Deklaration legt den Bezeichner durch simple Angabe im Attribut **`id`** fest. Bei der Deklaration per Komponenten-Scan dagegen ist dies nicht möglich. Hier sorgt die Registrierung des modifizierten Namensgenerators **`EJBAnnotationBeanNameGenerator`** für die Einhaltung der Namenskonvention.

Für die in Spring deklarierten Beans gilt, dass sie keine zirkulären Abhängigkeiten aufweisen dürfen, eine Einschränkung, die für EJBs im Allgemeinen nicht gilt. Ein spezieller Post-Prozessor [Spr07] bietet für diese Restriktion inzwischen auch in Spring eine Lösung an.

Auswertung der EJB-Annotationen

Java EE verfügt über die Möglichkeit, Annotationen für die Konfiguration als Ersatz für Deployment-Deskriptoren im XML-Format zu nutzen. Viele Anwendungen machen von diesen EJB-spezifischen Annotationen regen Gebrauch. Das Spring-Framework nutzt grundsätzlich ebenfalls Annotationen, die sich aber von den EJB-Annotationen vollständig unterscheiden, auch wenn sie eine ähnliche oder überschneidende Bedeutung haben.

Das Pitchfork-Projekt [Pit08] hat diese Unstimmigkeit weitgehend aufgehoben und unterstützt mit einer Ergänzung zum Spring-Framework die folgenden EJB-Annotationen:

- *JSR-250-Injection-Annotationen:* Die Annotationen **`@PostConstruct`**, **`@PreDestroy`** und **`@Resource`** sind im Paket **`javax.annotation`** enthalten und realisieren Dependency Injection für externe Ressourcen.
- *EJB3-Injection-Annotation:* Das Paket **`javax.ejb`** beinhaltet die Annotation **`@EJB`** für die Dependency Injection von EJBs.
- *EJB3-Interception-Annotationen:* Mit den Annotationen **`@AroundInvoke`**, **`@ExcludeClassInterceptors`**, **`@ExcludeDefaultInterceptors`**, **`@Interceptors`** und **`@Invocation`** im Paket **`javax.interceptor`** unterstützt Java EE eine einfache Form der Aspektorientierten Programmierung (AOP).
- *EJB3-Transaction-Annotationen:* Im Paket **`javax.ejb`** finden sich ebenfalls die Annotationen **`@Stateless`**, **`@ApplicationException`** und **`@TransactionAttribute`** zur Steuerung des Transaktionsverhaltens.

Inzwischen hat das Spring-Framework einen Teil dieser Funktionalität integriert, sodass die Version 2.5 die JSR-250-Injection-Annotationen und die EJB3-Injection-Annotation auch ohne die Erweiterung Pitchfork verarbeitet. Nicht verstanden werden weiterhin die EJB3-Interception-Annotationen.

Zwei Konfigurationseinstellungen sind durch diese Gegebenheit möglich:

- Für Anwendungen, die keine EJB-Interception-Annotationen verwenden, reicht die Konfiguration der Post-Prozessoren aus dem Spring-Framework der Version 2.5 aus:

```
<bean class="org.springframework.beans.factory.annotation.\
AutowiredAnnotationBeanPostProcessor"/>
<bean class="org.springframework.context.annotation.\
CommonAnnotationBeanPostProcessor"/>
```

- Anwendungen, die EJB-Interception-Annotationen einsetzen, benötigen statt dessen die Post-Prozessoren aus der Pitchfork-Erweiterung:

```
<bean class=
    "org.springframework.jee.ejb.config.JeeEjbBeanFactoryPostProcessor">
  <property name="serviceEnvironment">
    <bean class=
      "org.springframework.jee.config.SimpleServiceEnvironment">
      <property name="transactionManager"
                ref="transactionManager"/>
    </bean>
  </property>
</bean>
```

Bei der Verwendung des Komponenten-Scans für die Deklaration der EJBs ist zu beachten, dass das Attribut **`annotation-config="false"`** gesetzt ist. Diese Konfiguration schaltet die Standard-Post-Prozessoren für Annotationen von Spring aus, die der Scan sonst automatisch aktiviert.

Transaktionsmanagement

Eine Gewinn durch Java EE ist das deklarative Transaktionsmanagement. Dabei markieren Annotationen an Methoden das Transaktionsverhalten von Diensten. Spring und EJB gehen hier wieder getrennte Wege, da verschiedene Annotationen dafür vorgesehen sind. Die wichtige Annotation **`@TransactionAttribute`** von EJB wird aber auch vom Spring-Framework seit der Version 2.5 unterstützt.

Für den Fall, dass die Pitchfork-Erweiterung die EJB-Annotationen auswertet, ist keine weitere Deklaration notwendig, da der Post-Prozessor von Pitchfork diese Annotationen mit beachtet. Anders sieht es bei der Lösung ohne Pitchfork aus. Dafür bietet das Spring-Framework eine eigene Deklaration an:

```
<!-- configuring declarative transaction management -->
<tx:annotation-driven transaction-manager="transactionManager"
  proxy-target-class="false"/>
```

Der Vorteil der Pitchfork-Erweiterung gegenüber der reinen Spring-Lösung ist, dass bei Pitchfork bereits die richtigen Standardeinstellungen des Transaktionsverhaltens für die Session-Beans gelten.

Beide Varianten benötigen einen Transaktionsmanager. Für Anwendungen, die mit einer Ressource auskommen, reicht der JPA-Transaktionsmanager:

```
<-- declaring a transaction manager -->
<bean id="transactionManager"
  class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="dataSource" ref="ExampleDS"/>
</bean>
```

Werden mehrere Datenbanken oder sogar Message-Oriented-Middleware benutzt, ist ein JTA-Transaktionsmanager für verteilte Transaktionen notwendig. Das Spring-Framework bietet selbst keine eigene JTA-Implementierung an. Die detaillierte Einführung zum Thema JTA in Spring [Mur07] stellt für diese Aufgabe drei Alternativen vor und zeigt deren Konfiguration in Spring.

Kontextinformation für EJBs

Mithilfe der Kontextinformationen, der Schnittstelle **`EJBContext`** und deren Spezialisierungen kann eine Bean Informationen des Containers auslesen. Durch einfache Dependency Injection gelangt die Kontextinformation in eine EJB und ist unter anderem für folgende Aufgaben nutzbar:

- Die Methode **`getCallerPrincipal`** liefert den aktuellen Principal, um beispielsweise Audit-Informationen zu erfassen oder Berechtigungen zu prüfen.
- Den Zustand der aktuellen Transaktion liest die Methode **`getRollbackOnly`** aus. Um die aktuelle Transaktion abzubrechen, steht die Methode **`setRollbackOnly`** bereit.
- Auch kann über den Timer-Service, der mit **`getTimerService`** verfügbar ist, eine zeitbasierte Steuerung erfolgen.

Leider bieten weder das Spring-Framework noch die Erweiterung Pitchfork eine Umsetzung für die Kontextinformation an. Es ist aber nicht besonders schwer, eine eigene Implementierung für den EJB-Container bereitzustellen:

```
<bean id="SessionContextImpl"
      class="de.ivu.util.spring.JpaSessionContextImpl">
  <constructor-arg value="${java.naming.security.principal}"/>
</bean>

<!-- needing for @Resource annotations -->
<alias alias="sessionContext" name="SessionContextImpl"/>
```

Die Klasse **`JpaSessionContextImpl`** ist eine solche geeignete Umsetzung. Sie unterstützt die Methoden **`getCallerPrincipal`**, **`getRollbackOnly`** und **`setRollbackOnly`**. Dafür bekommt sie einen Principal per Konstruktor-Argument übergeben. Auf die aktuelle Transaktion greift sie über den Entity-Manager zu, den die Dependency Injection der Klasse bereitstellt.

Für die Nutzung des Timer-Service ist eine EJB-spezifische Timer-Service-Implementierung notwendig. Das Spring-Framework bietet eine solche nicht, obwohl es zeitgesteuerte Aktivitäten grundsätzlich unterstützt [Spr08]. Es spricht nichts gegen die Realisierbarkeit einer solchen EJB-spezifischen Implementierung des Timer-Service. Allerdings ist fraglich, ob sie für die Entwicklungszeit wirklich nötig ist.

Persistenz

Für Persistenz sorgt in Java EE-Anwendungen die Java-Persistence-API (JPA) [Sun06b], für die eine Implementierung bereitstehen muss. Spring nutzt das Framework Hibernate [BK06], das auch im Applikationsserver JBoss eingesetzt wird. Die Konfiguration benötigt dafür die Deklaration einer Entity-Manager-Factory:

```
<!-- declaring the entity manager factory -->
<bean id="entityManagerFactory"
  class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="ExampleDS"/>
  <property name="persistenceXmlLocation"
            value="META-INF/persistence.xml"/>
  <property name="persistenceUnitName" value="example"/>
  <property name="jpaVendorAdapter">
    <bean class=
"org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="generateDdl" value="false"/>
      <property name="databasePlatform"
        value="org.hibernate.dialect.MySQLDialect"/>
    </bean>
  </property>
</bean>
```

Die Entity-Manager-Factory verweist auf eine Persistence-Unit, die in der Konfigurationsdatei **`persistence.xml`** zu definieren ist. Es ist empfehlenswert, ein eigenes Persistenzarchiv für die Entity-Klassen zu packen, um ein mühsames Aufführen sämtlicher Entity-Klassen in dieser Datei zu vermeiden. Ein Persistenzarchiv enthält neben den Entity-Klassen die **`persistence.xml`**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="example"
    transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>java:ExampleDS</non-jta-data-source>
    <properties>
      <property name="jboss.entity.manager.jndi.name"
                value="java:ExampleEM"/>
    </properties>
  </persistence-unit>
</persistence>
```

Die Persistence-Unit benötigt eine Referenz auf eine DataSource, die üblicherweise über JNDI erreichbar ist. Diese DataSource ist in der Spring-Konfiguration zu deklarieren. In dem Beispiel ist zu sehen, dass mit der Syntax **`${url}`**, **`${username}`** und **`${password}`** auf Eigenschaften aus den System-Propertys, erreichbar über **`System.getProperties`**, zugegriffen wird. Dies hat den Vorteil, dass die Kommandozeile beim Starten der Client-Anwendung die Werte für diese Parameter setzen kann. Aktiviert wird diese Besonderheit in Spring mit dem **`PropertyPlaceholderConfigurer`**. Generell ist dieser Mechanismus ein schönes Verfahren, um notwendige Konfigurationen zwischen Präsentationsschicht und Geschäftslogik zu transportieren:

```
<!-- using system properties in configuration -->
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

<!-- declaring the datasource -->
<bean id="ExampleDS"
      class=
"org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
            value="com.mysql.jdbc.Driver"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
```


```
    <property name="password" value="${password}"/>
</bean>
```

Abschließend benötigt die Konfigurationsdatei für die Persistenz die Deklaration eines weiteren Post-Prozessors, damit das Spring-Framework die JPA-spezifischen Annotationen, wie `@PersistenceContext` und `@PersistenceUnit`, versteht:

```
<!-- processing the JPA annotations -->
<bean class="org.springframework.orm.jpa.support.\
PersistenceAnnotationBeanPostProcessor"/>
```

Spring als JNDI-Provider

Die Client-Anwendung findet die Dienste des Applikationsservers typischerweise über einen Namens- und Verzeichnisdienst, den der Applikationsserver selbst bereitstellt. Bei Java EE-Anwendungen handelt es sich dabei um einen JNDI-Dienst [Sun99]. Durch den fehlenden Applikationsserver entfällt dieser nun und muss innerhalb der Client-Anwendung ersetzt werden.

Die Idee für die Umsetzung des Namensdienstes ist, dass die im Spring-Framework deklarierten Beans unter den der Client-Anwendung bekannten Namen registriert werden und bei Anforderung durch die Präsentationsschicht als Dienst zur Verfügung stehen. Von Hause aus bietet das Framework Spring zunächst keine JNDI-Implementierung. Trotzdem existieren im Umfeld mehrere alternative Lösungsansätze.

Beispielsweise gibt es in den Vorschlägen für neue Spring-Funktionen einen Vorschlag [Spr05], der eine JNDI-Implementierung vorsieht. Er enthält die Realisierung einer `SpringInitialContextFactory`.

Die Datei `jndi.properties` legt die gewählte Context-Factory mit der entsprechenden Spring-Konfigurationsdatei, hier `jndi.xml`, fest:

```
java.naming.factory.initial=\
org.springframework.jndi.spring.SpringInitialContextFactory
java.naming.provider.url=jndi.xml
```

Für jede Bean, die über JNDI erreichbar sein soll, erfolgt ein Eintrag gemäß der EJB/JNDI-Namenskonvention des Applikationsservers in der Spring-Konfigurationsdatei:

```
<!-- defining JNDI names -->
<alias alias="ExampleApp/EmployeeFacadeBean/remote"
       name="EmployeeFacadeBean"/>
<alias alias="ExampleApp/EmployeeFacadeBean/local"
       name="EmployeeFacadeBean"/>
```

Damit kann die Präsentationsschicht ohne Änderungen im Programmcode durch eine herkömmliche JNDI-Abfrage auf die Implementierungen der Geschäftslogik zurückgreifen und wie gewohnt deren Dienste nutzen.

Fazit

Der hier beschriebene Ansatz ist zur Beschleunigung der Entwicklungszeit von mehrschichtigen Anwendungen, wie beispielsweise Java EE-Anwendungen, gedacht:

- Durch den Verzicht auf einen Applikationsserver reduziert das dargestellte Vorgehen unnötige technische Komplexität in der Entwicklungsphase.
- Stattdessen sorgt ein eingebetteter EJB-Container in der Client-Anwendung für die Laufzeitumgebung der Geschäftslogik.
- Der besondere Reiz des Verfahrens liegt darin, dass weder der Quellcode der Client-Anwendung noch der der Geschäftslogik modifiziert werden muss.

Literatur

[Apa09] Apache Software Foundation, OpenEJB 3.1.1, `http://openejb.apache.org`, besucht am 20.8.2009

[BK06] Ch. Bauer, G. King, Java Persistence with Hibernate, Manning Publications, 2006

[Fow02] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman, 2002

[JBo08] JBoss Community Team, JBossTools 2.1.2.GA, `http://www.jboss.org/tools`, besucht am 20.8.2009

[JBo09] JBoss Community Team, Embedded JBoss Beta 3 SP9, `http://www.jboss.org/community/wiki/EmbeddedJBoss`, besucht am 20.8.2009

[Mur07] M. Kosaraju, XA transactions using Spring, JTA/XA transactions without the J2EE container, in: JavaWorld, April, 2007, `http://www.javaworld.com/javaworld/jw-04-2007/jw-04-xa.html`, besucht am 20.8.2009

[Pit08] Pitchfork-Project, Pitchfork 1.0 M5, `http://www.springsource.com/pitchfork`, besucht am 20.8.2009

[SIGS] Quellcode zur Kolumne, `http://www.sigs-datacom.de/sd/publications/js/20010/01/index.htm`

[Spr05] Spring Projects Issue Tracker, Spring JNDI provider, New Feature Request SPR-1422, `http://jira.springframework.org/browse/SPR-1422`, Oktober 2005, besucht am 20.8.2009

[Spr07] Spring Community Forums, Circular dependency solved by BeanPostProcessor, `http://forum.springsource.org/showthread.php?t=35406`, Februar 2007, besucht am 20.8.2009

[Spr08] Spring Community, The Spring Framework - Reference Documentation, 2.5.6, November 2008, `http://static.springsource.org/spring/docs/2.5.x/reference/index.html`

[Sun99] Sun Microsystems, Inc., Java Naming and Directory Interface Application Programming Interface (JNDI API), `http://java.sun.com/products/jndi/reference/api/index.html`, Juli 1999, besucht am 20.8.2009

[Sun06a] Sun Microsystems, Inc., Java Platform, Enterprise Edition (Java EE) Specification, v5, `http://java.sun.com/javaee/technologies/index.jsp`, 2006, besucht am 20.8.2009

[Sun06b] Sun Microsystems, Inc., JSR 220: Enterprise JavaBeans,Version 3.0, Java Persistence API, `http://jcp.org/en/jsr/detail?id=220`, Mai 2006, besucht am 20.8.2009

[Zer09] ZeroTurnaround, JavaRebel 2.0.2b, `http://www.zeroturnaround.com/javarebel`, besucht am 20.8.2009

Dr. Martin Kempa ist seit 2009 Professor im Studiengang Wirtschaftsinformatik an der Hochschule für Technik und Wirtschaft Berlin. Zuvor arbeitete er für die Firmen sd&m und IVU Traffic Technologies in der Beratung und Entwicklung von betrieblichen Informationssystemen. Die Schwerpunkte seiner aktuellen Tätigkeit liegen in den Fachgebieten Datenbanksysteme sowie der Entwicklung von datenbankorientierten Anwendungen.
E-Mail: martin.kempa@htw-berlin.de.