heise Developer

Werkzeuge 06.07.2012 - 11:41

Jan Petzold

Mit Java ins Web: Eine Spring-MVC-Anwendung im Detail, Teil 1

Großstadtgespräch

Mit Spring MVC lassen sich in Kombination mit JavaScript dynamische und REST-konforme Webapplikationen mit Java umsetzen. Ein zweiteiliges Tutorial bietet einen Leitfaden zur Entwicklung einer Webanwendung mit Spring 3, JavaScript und einer Datenbank.

Spring ist in den letzten Jahren zu einem wichtigen Java-Framework geworden. Entstanden aus einer gewissen Frustration über bestehende Java-EE-Techniken (Java Enterprise Edition) und dem Wunsch nach einer leichteren Entkopplung von Softwarekomponenten bietet es heute viele vereinheitlichte Module an, die in vielen Softwareprojekten nützlich sein können. Für Webanwendungen sind neben dem Kern-Framework vor allem die Module Spring MVC und Spring Web Flow relevant.

Mit dem hier behandelten Spring MVC lassen sich in Kombination mit JavaScript dynamische und REST-konforme Webapplikationen mit Java umsetzen. Es bietet eine übersichtliche Model-View-Controller-Struktur und kann auf etablierte Techniken wie JPA (Java Persistence API) und JSP (Java Server Pages) zurückgreifen. Im Tutorial entsteht unter Eclipse eine Webanwendung mit Datenbankanbindung, die auf Spring 3 beruht und neben den üblichen CRUD-Methoden (Create, Read, Update, Delete) Hilfsfunktionen wie eine Suche mit automatischer Vervollständigung sowie eine Fehlerkorrektur einbindet.

Der Autor zeigt den Quelltext in den Artikeln jedoch nur auszugsweise. Sicht- und benutzbar ist die Anwendung unter **citiesoftheworld.cloudfoundry.com[1]**. Der komplette Quelltext der Anwendung lässt sich als Eclipse-Projekt **herunterladen[2]**.

Kernkomponenten von Spring

Zwei wesentliche Funktionen von Spring (Abstraktion, Dependency Injection, aspektorientierte Programmierung) sind in der Fachliteratur umfänglich beschrieben. Erwähnenswert bleibt, dass sich Dependency Injection seit Version 2.5 auch im Java-Code über Annotationen setzen lässt. Dazu ein Kurz-Beispiel:

```
// Klasse CityService
@Service
public class CityService {
    // Hier folgen Felder und Methoden
}

// andere Klasse, bekommt Objekt der Klasse CityService injiziert
public class myClass {
    private CityService cityservice;

    @Autowired
    public void setCityService(cityService) {
         this.cityService = cityService;
    }
}
```

Im Wesentlichen entkoppelt Dependency Injection die Klassen einer Anwendung, sie lassen sich leichter wiederverwenden und testen. Generell gibt es unter den einzelnen Spring-Modulen wenig Abhängigkeiten, der Entwickler kann sich die Teile herausgreifen, die für seine Anwendung von Belang sind, und bekommt darüber hinaus nichts "aufgezwungen". Somit lässt sich Spring gut mit anderen Techniken aus dem Java-Umfeld kombinieren.

Spring 3 MVC im Detail

MVC besagt, dass in einer Anwendung jederzeit das Datenmodell von der Präsentationsschicht und der Ablaufsteuerung getrennt sein muss. Übersetzt auf typische Java-Webapplikationen bedeutet das, dass auf der (JSP-)Seite keinerlei Anwendungslogik stehen darf und die Daten zwischen Oberfläche (= View) und Datenbank (= Model) über einen sogenannten Controller übermittelt werden. Der Ansatz trennt die Schichten sauber, erleichtert Tests und vermeidet schwer zu pflegenden "Spaghetti-Code". Außerdem lässt sich die Arbeit im Team leichter aufteilen – der Datenbankspezialist kann sich mit der Datenbank und dem zugehörigen "Model" beschäftigen, während sich ein Frontend-Entwickler weitgehend auf seine "View" konzentriert. Spring führt zur Implementierung das zentrale DispatcherServlet ein, das in der Serverkonfiguration (*web.xml*) einzurichten ist:

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.
           DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/app/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

Das Servlet hat den Namen *appServlet* und reagiert auf alle eingehenden Requests unter */spring/*. Dadurch lässt sich der MVC-Teil vom Rest der Applikation trennen. Darüber hinaus ist in der Konfiguration noch ein Package zu definieren, in dem Spring nach Controllern sucht:

```
<context:component-scan base-package="com.example.spring3cities" />
```

In der Anwendung findet der Leser den Controller im Package *com.example.spring3cities*. Sofern in einer Klasse die Annotation *@Controller* gesetzt ist, erkennt das Programm diese automatisch als Controller. Hier ein Beispiel:

```
@Controller
Public class CityController {
    @RequestMapping(value="/city", method=RequestMethod.GET)
    public String homeHandler(Model model) {
        String message = "Hello world!"
        model.addAttribute("myMessage", message);
        return "home";
    }
}
```

Die Annotation *@RequestMapping* weist der URL eine Methode zu: Im Beispiel würde die Funktion *homeHandler* alle Aufrufe unterhalb des Verzeichnisses */city* verarbeiten. Als HTTP-Requestmethode ist hier *GET* angegeben. Lässt der Entwickler die Angabe weg, gilt die Funktion für alle HTTP-Request-Methoden.

homeHandler liest die Variable *model* aus und liefert den String *home* zurück, was unter Spring automatisch dazu führt, dass das Programm die View *home.jsp* aufruft. *model* übergibt sozusagen die initial aufgerufene View und erhält zusätzlich das Attribut *message*. In der JSP wird zur Ausgabe der gelieferten Informationen einfach ein Platzhalter eingefügt:

```
<h1>${myMessage}</h1>
```

In der Überschrift steht anstelle des Platzhalters nun wie erwartet "Hello world!". Im Wesentlichen ist das die Umsetzung des MVC-Prinzips in Spring. Der Ansatz hat mehrere Vorteile: Zum einen lässt sich anhand der URL eindeutig nach Funktionen unterscheiden. Im Ergebnis entstehen automatisch REST-konforme Webanwendungen mit sauberen (suchmaschinenfreundlichen) URLs ohne weitere Konfiguration. Nicht definierte URLs werden ignoriert.

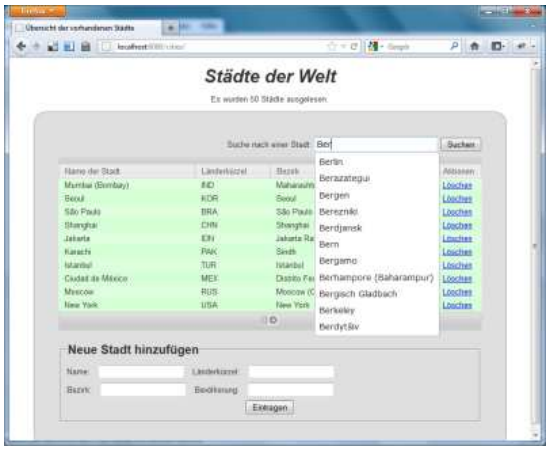
Das *RequestMapping* kann natürlich auch GET- und POST-Parameter aus der JSP-Seite enthalten. Der Controller setzt diese in eine sogenannte *PathVariable*:

```
@RequestMapping(value="/city/details/{cityId}", method=RequestMethod.POST)
public String getCityDetails(@PathVariable("cityid") int id) {
     // Variable    cityid enthält die ID aus der URL
```

Mit **Spring Roo[3]** existiert darüber hinaus ein sich besonders für MVC-Anwendungen passender Generator zum Aufsetzen einer Spring-Anwendung. Roo eignet sich bisher vor allem für das Prototyping, setzt aber ein gutes Spring-Verständnis voraus, da der zu großen Teilen automatisch generierte Sourcecode nie unangepasst bleiben dürfte.

Ein Beispielprojekt mit Spring 3 MVC

Zur Veranschaulichung von Spring 3 MVC entsteht nun die Beispielapplikation "Cities", die einige der größten Städte der Welt in einer Tabelle anzeigt und sich außerdem pflegen beziehungsweise aktualisieren lässt. Für jede Stadt erfasst sie das zugehörige Land, den Bezirk und die Einwohnerzahl. Die Anwendung gibt die Städte geordnet nach Größe aus und erlaubt es, neue Städte hinzuzufügen oder bestehende zu entfernen. Der erste Artikel erstellt dafür das "Grundgerüst". Der zweite ergänzt die Suchfunktion um eine Auto-Vervollständigung bei der Eingabe des Suchbegriffs. Weiterhin kommt eine Fehlerkorrektur dazu, die bei Schreibfehlern Vorschläge für verwandte Städtenamen ausgibt.



Beispiel für die Suche nach einer Stadt (Abb. 1)

Sämtliche Daten hält eine MySQL-Datenbank vor. Die Ausgabe der Daten erfolgt in einer optisch ansprechenden Tabelle, die das **DataTables[4]**-Plug-in für jQuery umsetzt. Die Anwendung soll dem Prinzip des "Progressive Enhancement" folgen. Das heißt, Nutzer mit alten Browsern oder deaktiviertem JavaScript und/oder CSS sollen das Angebot zumindest noch rudimentär nutzen können. jQuery und jQuery UI folgen dem Prinzip, indem sie das DOM gegebenenfalls ergänzen, aber nicht verstümmeln.

Die Grundlagen der Umgebung

Die Ausgangsbasis für das Projekt ist ein installiertes Eclipse for Java Developers (Indigo, Service Release 2; das neue Eclipse Juno dürfte einem ersten Test zufolge ebenfalls funktionieren). Für die Auflösung der Abhängigkeiten kommt Maven zum Einsatz. Jedes Maven-Projekt benötigt einen passenden Archetype (= Vorlage für ein Softwareprojekt), der die benötigten Komponenten einrichtet. Ein guter Ausgangspunkt für das Projekt ist der Archetype *spring-mvc-jpa-archetype* (zu finden unter der Group-ID *org.fluttercode.knappsack* im Katalog *NexusIndexer*). Er setzt noch Spring 3.0.5 voraus, was für das Szenario aber genügt. Möglicherweise ist er nicht direkt verfügbar, da Eclipse beim Erstaufruf die Liste der Maven-Repositories im Hintergrund aktualisiert. Der Fortschritt lässt sich gut in der Progress View beobachten, eventuell kann man in der View "Maven-Repositories" mit einem Klick auf REBUILD INDEX nachhelfen.

Die weiteren Angaben sind frei wählbar, als Group-ID bietet sich eine Domain an, etwa *com.example.spring3mvc*. Als Artifakt-ID setzt man den Projektnamen, z. B. "cities". Das "package" wird freigelassen. Eclipse erstellt das Projekt und lädt die Bibliotheken herunter.

Im Projekt sind bereits eine Menge XML-Konfigurationsdateien für den Servlet-Container und Spring vorhanden, was zunächst verwirrend ist. Es empfiehlt sich aber, die Unterteilung weitgehend beizubehalten, damit die Konfigurationsbereiche auch in größeren Projekten überschaubar bleiben. Hier eine kurze Erklärung:

- *src/main/webapp/web.xml*: die Basis-Konfigurationsdatei für alle Webangebote (Servlets), in der unter anderem die gewünschten Spring-Module gesetzt werden.
- *src/main/webapp/spring/root-context.xml*: konfiguriert den globalen Application Context, der für alle auf dem Server laufenden Spring-Anwendungen gilt.
- *src/main/webapp/spring/db.xml*: übergreifende Konfiguration der Datenbank und des zu verwendenden ORM (z. B. Hibernate).
- *src/main/resources/META-INF/persistence.xml*: sammelt ORM-spezifische Einstellungen.
- *src/main/webapp/spring/app/servlet-context.xml*: die wesentliche Spring-Konfigurationsdatei für eine spezifische Anwendung, hier für das DispatcherServlet als Grundlage von Spring MVC.
- *src/main/webapp/spring/app/controllers.xml*: listet alle Controller der Anwendung beziehungsweise setzt das Basis-Package für die automatische Suche nach Controllern.

Für die lokale Entwicklung benötigt der Entwickler einen Server, etwa **Tomcat 7.0[5]**. Dieser lässt sich nach der Installation komplett über Eclipse steuern. In der IDE sind noch einige Plug-ins zu installieren. Alle in der Folge gelisteten Module sind unter dem Knoten "Web, XML, Java EE and OSGi Enterprise Development" der **Indigo-Release-URL[6]** verfügbar:

- Eclipse Java EE Developer Tools
- Eclipse Java Web Developer Tools
- Eclipse Web Developer Tools
- JavaScript Development Tools
- JST Server Adapters
- JST Server Adapters Extensions

Die ersten drei Module sind für die Web- und JavaScript-Entwicklung hilfreich, die JST-Server-Adapter benötigt der Entwickler für die Verbindung zwischen dem lokal installierten Tomcat und Eclipse. Der Server lässt sich nach dem Neustart von Eclipse in der View "Servers" anlegen und starten.

Die Anwendung liest die Städtedaten aus einer MySQL-Datenbank, obgleich sich auch andere Datenbanken mit JDBC-Connector nutzen ließen. Nach der Installation des **MySQL Community Server[7]** ist auszuwählen, ob MySQL als Dienst (also bei jedem Systemstart im Hintergrund) oder manuell gestartet werden soll. Für den lokalen Test ist Letzteres besser, zum manuellen Start im Installationsverzeichnis muss der Entwickler die *\bin\mysqld.exe*-Datei ausführen

Für die Anwendung benötigt er nun noch eine Datenbank mit Städtedaten. Eine brauchbare Grundlage findet man **hier[8]**. Die Datenbank enthält die gewünschten Angaben. Die Daten sind nicht auf dem aktuellen Stand, genügen aber für das Beispiel. Für Fortgeschrittene wäre die freie Geodatenbank von **Maxmind[9]** eine Alternative, da sie umfangreicher und aktueller ist. Für das Einstiegsbeispiel hier ist sie aber etwas überdimensioniert.

Das Einrichtungsskript *world.sql* nutzt leider die Zeichenkodierung *latin1*. Vor der Einrichtung sollte der Anwender das auf UTF-8 ändern. Dazu muss er das Skript in einem UTF-8-fähigen Texteditor (z. B. **Notepad++[10]**) öffnen, die Zeichenkodierung auf UTF-8 umstellen und am Anfang des Skripts folgende Zeilen einfügen:

```
SET NAMES utf8;
SET character_set_client = utf8;
```

Dann hat er per Suchen und Ersetzen alle *latin1*-Vorkommen in *utf8* zu ändern und zu speichern. Nun lässt sich die Datenbank sauber in UTF-8 importieren.

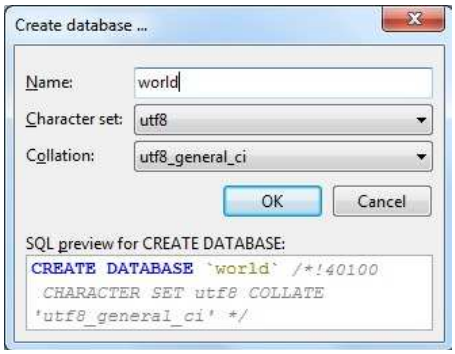
Unter Windows gibt es viele Tools zur MySQL-Verwaltung, etwa **HeidiSQL[11]**. Um eine neue Datenbank anzulegen, muss der Entwickler sich zunächst mit den Benutzerdaten (Nutzername root, kein Passwort) auf dem MySQL-Server einloggen. Danach hat er eine neue Datenbank "world" mit den Parametern wie im Screenshot anzulegen sowie das eben bearbeitete Skript *world.sql* zu importieren und auszuführen. Nach kurzer Wartezeit sollte die Tabelle "city" 4079 Einträge haben.

Mit HeidiSQL lassen sich komfortabel neue Nutzer anlegen und Passwörter ändern, was für den Nutzer "root" im Produktivbetrieb dringend zu empfehlen ist.

Damit der Entwickler im Eclipse-Projekt den installierten Tomcat-Server zuweisen kann, aktiviert er unter dem Punkt "Project Facets" in den Projekteigenschaften "Dynamic Web Project". Standardmäßig sieht Eclipse als Ausgabeverzeichnis den Ordner "WebContent" vor, der im Projekt aber nicht existiert – stattdessen wird das Verzeichnis "webapp" genutzt. Man muss also in Eclipse in den Projekteigenschaften unter "Deployment Assembly" den Ordner "src/main/webapp" hinzufügen. Den Ordner "WebContent" kann der Entwickler hier und auf der Festplatte löschen. Außerdem sind die von Maven bereitgestellten Bibliotheken einzubinden, das geht im selben Fenster über ADD | JAVA

BUILD PATH ENTRIES | MAVEN DEPENDENCIES.

Nun sollte den Leser eine funktionierende Anwendung begrüßen – dazu muss er das Projekt "cities" dem Tomcat in der View "Servers" hinzufügen und den Server starten. Im Browser erscheint nun unter *http://localhost:8080/cities* ein "Hello World!".



Einstellungen der MySQL-Tabelle (Abb. 2)

Klassenaufteilung

Um die Anwendung sauber gemäß MVC aufzuteilen, sind zunächst Packages und Klassen für den jeweiligen Zweck erforderlich. Für die Anwendung wäre folgende Gliederung sinnvoll:

- Package *CityModel* mit Klasse *City* enthält das Datenmodell als POJO (Plain Old Java Object).
- Package *CityRepository* umfasst die grundsätzlichen Methoden zum Lesen, Schreiben, Suchen und Löschen.
- Package *CityController* sammelt alle Requests und delegiert diese weiter.

In der Praxis käme noch mindestens ein weiteres Package *CityService* dazu, dessen Klassen dem Controller sozusagen "zuarbeiten". Würde der Entwickler die gesamte Logik im Controller unterbringen, wäre dieser schnell unübersichtlich, da fast jede Anwendung einige URLs bereitstellt. Insofern ist es ratsam, den Controller schlank zu halten und möglichst alle verarbeitenden Methoden in anderen Klassen zu sammeln. Darüber hinaus bringt Spring noch **zahlreiche Validatoren[12]** mit, die vor allem dazu dienen, Nutzereingaben auf Gültigkeit zu prüfen. Sie könnte man im Package *CityValidator* sammeln.

Die Views sind in JSP-Dateien in einem eigenen Verzeichnis untergebracht, etwa unter *WEB-INF/views*.

Controller und View

Die zentrale Ablaufsteuerung der Anwendung findet im Controller statt. Hier legt der Entwickler alle URLs sowie die weitere Logik und Verarbeitung der (eingegebenen) Informationen fest. Er benötigt im Wesentlichen vier Mappings:

- Standard-GET-Handler für die Startansicht (Liste der Städte)
- POST-Handler für das Anlegen einer neuen Stadt (z. B. */city/add*)
- GET-Handler für das Löschen einer Stadt aus der Datenbank (z. B. */city/delete/ID*)
- GET-Handler für die Suchfunktion (z. B. */city/search?term=CITY*)

Hinweis: Eine simple GET-Methode zum Löschen eines Datensatzes darf natürlich nicht in der Praxis verwendet werden – jeder Nutzer, der die ID eines Datensatzes kennt, könnte diesen dann löschen. Löschzugriffe sollten nur angemeldeten Nutzern erlaubt sein. Über **Spring Security[13]** lassen sich darüber hinaus auch Regeln für bestimmte URLs und Pfade festlegen.

Im nächsten Schritt ist festzulegen, was die einzelnen Mappings als Ergebnis liefern. Für die ersten drei Mappings ist das einfach, letztlich soll einfach die (aktualisierte) Liste der Städte in einer Tabelle erscheinen. Bei einer Suche muss das Suchergebnis in derselben Tabelle erscheinen wie die Gesamtliste, damit die Anwendung einheitlich wirkt.

Die Ergebnisse der jeweiligen Operation werden in einer Java-Collection gesammelt (List-Interface). Um diese zwischen Model und View zu übertragen, definiert Spring den Datentyp *ModelAndView*. Im Prinzip ist das ein Sammelobjekt, das den Namen der zugehörigen View enthält und dem sich die Unterobjekte zuordnen lassen. Die Objekte sind in der View per Expression abrufbar – ein Beispiel:

```
ModelAndView mv = new ModelAndView("city");
mv.addObject("message", "Hallo Welt");
```

In der View *city* wird das Objekt *message* wie folgt aufgerufen:

```
<p>Die Botschaft lautet ${message}</p>
```

Für die Mappings sieht das Quelltext-Gerüst der Klasse *CityController* so aus:

```
@Controller
public class CityController {
    @RequestMapping(method=RequestMethod.GET, value="/")
    public ModelAndView showCities() {
        ModelAndView cities = new ModelAndView("city");
        // TODO: lese alle Städte aus der Datenbank aus
        return cities;
    }

    @RequestMapping(method=RequestMethod.POST, value="/city/add")
    public String addCity(@ModelAttribute("city") City city,
           BindingResult result) {
        // TODO: lege neue Stadt anhand der Parameter des POST-Requests an
        return "redirect:/";
```

```
    }

    @RequestMapping(method=RequestMethod.GET, value="/city/delete/{id}")
    public String deleteCity(@PathVariable("id") int id) {
        // TODO: lösche einen Datensatz anhand der übergebenen ID
        return "redirect:/";
    }

    @RequestMapping(method=RequestMethod.GET, value="/city/search")
    public ModelAndView getSearchResult(@RequestParam("term") String term) {
        ModelAndView cities = new ModelAndView("city");
        // TODO: verarbeite die Suche nach dem Begriff "term"
        // und ermittle das Ergebnis
        return cities;
    }
}
```

Die Methoden zum Hinzufügen und Löschen von Städten führen also nach der Verarbeitung einen simplen Redirect auf die Startseite (= Ansicht aller Städte) aus.

Erwähnenswert ist noch die Unterscheidung zwischen *@PathVariable* und *@RequestParam*: Ersteres steht für einen dynamischen Teil der URL wie die ID des zu löschenden Datensatzes. Letzteres ist ein benannter Parameter, hier der Suchbegriff mit dem Bezeichner *term*. Eine Beispiel-URL für die Suche wäre etwa: *http://localhost:8080/cities/city/search/?term=Berlin*.

Zur Darstellung benötigt der Programmierer nur eine einzige View, die er unter *src/main/webapp/WEB-INF/views/city.jsp* ablegt. Die übergebenen Variablen und Objekte werden wie beschrieben über Expressions ausgelesen. Interessant ist noch die Ausgabe der Tabelle mit den Städtedaten, was sich leicht über eine *forEach*-Schleife aus der JSTL-Bibliothek (JavaServer Pages Standard Tag Library) erledigen lässt:

```
<table>
    <tbody>
        <c:forEach items="${cities}" var="currentcity">
            <tr>
                <td>${currentcity.name}</td>
                <td>${currentcity.country}</td>
                <td>${currentcity.district}</td>
                <td>${currentcity.population}</td>
                <td><a href="/cities/city/delete/${currentcity.id}"
                     class="deleteLink">Löschen</a></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

So lässt sich in JSP recht einfach über eine Collection iterieren. Zur sauberen Unterstützung von JSTL ist die Maven-Konfiguration (im Stammverzeichnis unter *pom.xml*) um eine Abhängigkeit zu ergänzen

```
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

Anbindung der Datenbank über JPA

Für den Zugriff auf die Datenbank bietet Spring mit dem JDBC-Modul eine Basis-Implementierung an, zeitgemäßer ist aber ein automatisches Mapping auf das Basisobjekt *City*. Für das OR-Mapping lässt sich beispielsweise Hibernate einsetzen. Um auf die MySQL-Datenbank zuzugreifen, muss der Leser den Connector in der Maven-Konfiguration ergänzen:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.18</version>
</dependency>
```

Folgende Basis-Einstellungen sind in *src/main/resources/db.properties* einzurichten:

```
db.username=root
db.password=
db.driver=com.mysql.jdbc.Driver
db.url=jdbc\:mysql\://localhost\:3306/world
db.dialect=org.hibernate.dialect.MySQLDialect
```

Die Werte für Nutzername / Passwort entsprechen den Default-Einstellungen nach der Installation von MySQL. In einer Produktivumgebung sind sie natürlich unbedingt zu ändern.

Die Eigenschaften lassen sich nun in *src/main/webapp/WEB-INF/spring/db.xml* nutzen:

```
<bean id="placeholderConfig"
    class="org.springframework.beans.factory.config.
          PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:db.properties" />
</bean>
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.
           LocalContainerEntityManagerFactoryBean">
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.
              HibernateJpaVendorAdapter">
            <property name="showSql" value="true" />
            <property name="generateDdl" value="true" />
            <property name="databasePlatform" value="${db.dialect}" />
        </bean>
    </property>
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${db.driver}" />
    <property name="url" value="${db.url}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>

<bean id="transactionManager"
      class="org.springframework.orm.jpa.
          JpaTransactionManager"></bean>
```

Erwähnt sei außerdem die Bean *entityManagerFactory*, die ein zentrales Objekt erstellt, das in der Anwendung alle Persistenz-Aktionen (Speichern, Löschen ...) durchführt.

Für das Mapping zwischen Datenbank-Tabelle und der Model-Klasse *City* sind deren Felder um einige Annotationen zu erweitern:

```
@Entity
@Table(name = "city")
public class City {
    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Integer id;

    @Column(name = "Name")
    private String name;

    @Column(name = "CountryCode")
    private String country;

    @Column(name = "District")
    private String district;

    @Column(name = "Population")
    private String population;
    // es folgen Getter und Setter ...
}
```

Für die eigentlichen Queries empfiehlt sich der Einsatz von **Spring Data[14]**, womit sich die grundsätzlichen CRUD-Operationen leicht über JPA definieren lassen. Spring Data sieht dafür ein zentrales JPA-Repository vor, das bereits einige Basis-Methoden enthält (z. B. die Methode *findAll()*). Komplexere Queries können in JPA-Query-Syntax (also ähnlich wie SQL) als Annotationen über Methodenaufrufen in einem definierten Interface gesetzt werden – hier ein Beispiel für eine LIKE-Suche mit Wildcard:

```
public interface CityRepository extends JpaRepository<City, Long> {
     @Query("from City where Name LIKE CONCAT(:currentName, '%')
        order by ABS(Population) desc")
    List<City> searchCity(@Param("currentName") String name);
}
```

Für das Speichern eines Datensatzes genügt ein simpler Aufruf der (direkt in der Library enthaltenen) Methode *cityRepository.save()*, der automatisch ein übergebenes Objekt in der Datenbank persistiert.

In der bisherigen Konfiguration ist Spring Data nicht enthalten, darum muss es in der Maven-Konfiguration ergänzt werden:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.0.0.RELEASE</version>
 </dependency>
```

Zwischenfazit

Damit haben Autor und geneigter Leser eine funktionstüchtige, aber noch nicht besonders ansprechende beziehungsweise benutzerfreundliche Anwendung geschrieben. Sie kann Städte aus der Datenbank anzeigen sowie nach Städten suchen, Städte entfernen und neue Städte hinzufügen. Die wesentliche Struktur einer MVC-Anwendung und die Implementierung in Spring 3 sollte damit erklärt sein.

Der zweite Artikel des Tutorials wird die rudimentäre Anwendung um zeitgemäße und sinnvolle Funktionen (Auto-Vervollständigung, Fehlerkorrektur) und eine verbesserte Darstellung ergänzen. (ane)

Jan Petzold[15]
arbeitet als Senior-Softwareingenieur für Capgemini in Berlin.

Onlinequellen

- Eberhard Wolff; Spring 3 – Framework für die Java-Entwicklung; dpunkt.verlag 2010
- Eberhard Wolff; Spring 3 steht vor der Tür – ein erster Ausblick; **Artikel[16]** auf *heise developer*
- Kai Wähner; Das gewisse Extra; Einsatz und Grenzen von Spring Roo; **Artikel[17]** auf *heise Developer*
- Lars Vogel; **Dependency Injection with the Spring Framework – Tutorial[18]**
- Martin Fowler; **Inversion of Control Containers and the Dependency Injection pattern[19]**
- **Spring-Dokumentation[20]**

URL dieses Artikels:
http://www.heise.de/developer/artikel/Mit-Java-ins-Web-Eine-Spring-MVC-Anwendung-im-Detail-Teil-1-1623020.html

Links in diesem Artikel:
[1] http://citiesoftheworld.cloudfoundry.com
[2] ftp://ftp.heise.de/pub/ix/developer/petzold_springmvc.zip
[3] http://www.springsource.org/spring-roo
[4] http://datatables.net/
[5] http://tomcat.apache.org/download-70.cgi
[6] http://download.eclipse.org/releases/indigo
[7] http://dev.mysql.com/downloads/mysql/
[8] http://downloads.mysql.com/docs/world.sql.zip
[9] http://www.maxmind.com/app/worldcities
[10] http://notepad-plus-plus.org/
[11] http://www.heidisql.com/
[12] http://static.springsource.org/spring/docs/current/spring-framework-reference/html/validation.html
[13] http://static.springsource.org/spring-security/site/
[14] http://www.springsource.org/spring-data/
[15] https://www.xing.com/profile/Jan_Petzold6
[16] http://www.heise.de/developer/artikel/Spring-3-steht-vor-der-Tuer-ein-erster-Ausblick-227174.html
[17] http://www.heise.de/developer/artikel/Einsatz-und-Grenzen-von-Spring-Roo-1337918.html
[18] http://www.vogella.de/articles/SpringDependencyInjection/article.html
[19] http://martinfowler.com/articles/injection.html
[20] http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/