# YOUR NEXT JAVA WEB APP:

LESS XML, NO LONG RESTARTS, FEWER HASSLES

a ZT Labs tutorial

ZEROTURNAROUND

# My new web **application...**

Did you realize that Hibernate has been around for more than 10 years? And Spring will begin its second decade next year? There was a time when Spring+Hibernate was widely considered an unofficial industry standard, but today is portrayed as an ancient beast whose XML appetite kills little rainbow scroll wheels. That assessment is not true, however. Both technologies have seen continuous development and are still competitive today.

Doesn't it make more sense to compare apples to apples? For example, don't put JEE6 with CDI up against Spring 1.0. Spring and Hibernate don't require three miles of XML anymore. In fact, it's possible to set them both up with zero lines of XML.

With servlet 3.0, even web.xml can go the way of the dodo.

When all you're trying to accomplish is a simple helloWorld guestbook, then Java, Spring and Hibernate will require more effort to get there, but can you name any app which is that trivial in real life? Proper applications are expected to satisfy far greater needs, both functional and non-functional, which is the area where Java, supported by wisely-chosen tools & frameworks, really shines.

Why don't we name some qualities of a modern web application? Let's think about what we would like to see as the result, and work back from there. The app should:

- Be simple and easy to set up, rivalling pure JEE6, Play! Framework, or ***-on-rails
- Remain simple and easy when complexity gets beyond what is usually shown in tutorials
- Be easy to run and test on a developer machine as well as on remote servers
- Maximize developer productivity
- Be flexible and robust enough to support a wide array of enterprise requirements (which might be completely different after your boss gets back from the lunch break)
- Have a huge community and know-how behind it
- Keep away from magic and be easy to debug

Java apps are never just about pure Java. It takes a combination of Java, frameworks and tools to get the most out of the platform.

## The **lineup**

For this task, my arsenal includes the latest versions of Spring Framework 3.1 and Hibernate 4.1. I'm also using a tiny bit of BeanValidation to make the code sparkling. In addition to the frameworks, I have chosen some tools for developing the beast. One of the biggest benefits to productivity is having a Maven project which can be launched from within your IDE and updated on the fly with JRebel (Disclaimer: I joined ZeroTurnaround to work on the JRebel team in 2010).

### MAVEN 3

I'm creating a Maven project which can be built and deployed with one command on the remote server. On the developer side, it removes all the hassle of managing dependencies and trouble-shooting errors when some library is missing.
http://maven.apache.org/download.html

### TOMCAT 7

IMHO, the app server you choose is probably the least important decision. Stick to what you feel most comfortable with. However, it should be as lightweight as possible. In the past it meant using only a servlet container instead of a full blown JEE server, but today there are several JEE Web Profile offerings which also start up pretty quickly.
I prefer Tomcat, but this can freely be substituted with alternatives (such as Jetty 8, JBoss 7 or Glass-fish 3.1, which are all free and start up reasonably fast). Use any of those if you dislike Tomcat.
http://tomcat.apache.org

### ECLIPSE 3.7

I am personally used to Eclipse and will be using it in the article. Feel free to use IntelliJ IDEA or NetBeans instead. The important part is having the proper plugins that play well with Maven and JRebel. For Eclipse, these plugins are:

- Maven integration (m2e)
- Maven integration for WTP
- JRebel for Eclipse

Maven Integration for WTP is a separate plugin from m2e and is essential for launching a Maven project in a container from within Eclipse. You can install all of them from the Eclipse Marketplace.
http://www.eclipse.org/downloads/

### JREBEL 5.0 (WITH REMOTING)

JRebel allows you to see all sorts of changes to Java code in your app instantly, and it includes changes to Spring bean definitions and Hibernate mappings. It removes what is perhaps the biggest weakness of Java when compared to Grails, PHP or the like - down-time during the build/compile/restart phases. Plus you get to keep all what's great about Java.

JRebel Remoting, new to JRebel 5.0, allows you to upload changes to a remote machine and have the changes instantly reflected there too. It works with Jelastic, Rackspace and Amazon EC2 out of the box.
http://zeroturnaround.com/jrebel/current/
http://zeroturnaround.com/software/jrebel/remoting

## Creating the **Project**

I like to start by creating a blank **Dynamic Web Project** in Eclipse. I select "None" as a target runtime, "3.0" as the module version and "minimal" under "Configuration". That should result in a minimum possible setup for a servlet 3.0 application. Next I add Maven support to it. To do that, right-click on the newly created project and select **Configure->Convert to Maven Project**. Sometimes that messes up the Eclipse build and source directories. Make sure the src dirs of the project are src/main/java and src/main/resources, creating the folders when needed, and output dir should be target/classes.

Next you need to add a bunch of dependencies, like Spring and Hibernate, as well as Servlet API and a few more. Configuring the POM is the most boilerplate'ish part of the whole process, but it can actually be accomplished via GUI in Eclipse. Adding dependencies is easy. The biggest problem of Maven is that it's nontrivial to make adjustments to the build process (such as changing the default Java version). In the end, I end up with the following POM. I have annotated the file with comments as to why one or the other bit of configuration is necessary.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <!--replace with your namespace and version -->
 <groupId>com.zt</groupId>
 <artifactId>helloWeb</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <!--should not be changed -->
 <packaging>war</packaging>

 <!--some custom options are necessary. Perhaps they make their way to a convention in the future -->
 <build>
   <plugins>

     <!--source level should be 1.6 (which is not Maven default) for java EE 6 projects, so let's change it -->
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-compiler-plugin</artifactId>
       <configuration>
         <source>1.6</source>
         <target>1.6</target>
       </configuration>
     </plugin>

     <!-- When using xml-less approach, you need to disable Maven's warning about missing web.xml -->
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <configuration>
         <failOnMissingWebXml>false</failOnMissingWebXml>
       </configuration>
     </plugin>

   </plugins>
 </build>

 <dependencies>

   <!--We need servlet API for compiling the classes. Not needed in runtime -->
   <dependency>
```

```
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.0.1</version>
      <scope>provided</scope>
    </dependency>

    <!--adding spring mvc is easy -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>3.1.1.RELEASE</version>
    </dependency>

    <!-- Required for xml-less configuration of Spring via @Configuration
annotations -->
    <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib-nodep</artifactId>
      <version>2.2.2</version>
    </dependency>

    <!-- required for getting spring working with Hibernate -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>3.1.1.RELEASE</version>
    </dependency>

    <!-- Adding Hibernate -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>4.1.2</version>
    </dependency>

    <!-- Only needed when using JPA instead of "pure Hibernate" -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>4.1.2</version>
    </dependency>

    <!-- DB connection pooling for production applications -->
    <dependency>
      <groupId>commons-dbcp</groupId>
      <artifactId>commons-dbcp</artifactId>
      <version>1.4</version>
    </dependency>

    <!-- replace with concrete JDBC driver depending on your DB -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.19</version>
    </dependency>

    <!-- Add Taglib support -->
    <dependency>
      <groupId>jstl</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
      <!-- Use Provided when using JBoss or Glassfish, since they already
bundle it. I'm using Tomcat so it has to go in -->
      <!-- <scope>provided</scope> -->
    </dependency>

  </dependencies>

</project>
```

All that's left to do is changing the src directory to src/main/java and output folder to target/classes in Eclipse. You could also check whether paths are mapped correctly under Deployment Assembly. Note that you may have to refresh the project configuration after modifying the pom.xml. To do that, right click on the project and **Maven->Update Dependencies**.

## Hello World, **goodbye XML!**

Servlet 3.0 introduces a ServletContainerInitializer interface whose implementing classes are notified by the container about webapp startup events. Spring 3.1 uses this in its WebApplicationInitializer interface, which is the hook through which you can set up ApplicationContext without using the web.xml. Combine that with Spring's Java-based configuration, and you can successfully ditch XML files entirely.

First, you need to create a class that implements that Spring's WebApplicationInitializer interface. Then create the application context programmatically and finally add the servlet to the ServletContext (again, programmatically). The resulting initializer class looks like this:

```
public class Initializer implements WebApplicationInitializer {

 // gets invoked automatically when application starts up
 public void onStartup(ServletContext servletContext) throws Servle-
tException {

   // Create ApplicationContext. I'm using the
   // AnnotationConfigWebApplicationContext to avoid using beans xml
files.
   AnnotationConfigWebApplicationContext ctx =
       new AnnotationConfigWebApplicationContext();
   ctx.register(WebappConfig.class);

   // Add the servlet mapping manually and make it initialize auto-
matically
   Dynamic servlet =
       servletContext.addServlet("dispatcher", new
DispatcherServlet(ctx));
   servlet.addMapping("/");
   servlet.setLoadOnStartup(1);
 }

}
```

The trick that helps us get rid of applicationContext.xml files is achieved by using AnnotationConfigWebApplicationContext instead of XmlWebApplicationContext. I put the configuration in the WebappConfig class, which is annotated with @Configuration. All that is available in XML files can be done here programmatically. For example, adding @ComponentScan("com.zt.helloWeb") turns on Spring's scanning for annotated beans, akin to the previous <context:component-scan> directive. WebMVC is enabled via @EnableWebMvc and PropertyPlaceholderConfigurer is replaced with @PropertySource("classpath:filename.properties"). Beans are declared via adding @Bean annotation to a method that produces the object. The name of the method is not important. I'm using this to configure a ViewResolver (see below). The whole simple configuration class looks like this:

```
@Configuration
@ComponentScan("com.zt.helloWeb")
@EnableWebMvc
public class WebappConfig {

 @Bean
 public InternalResourceViewResolver setupViewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceVie-
wResolver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");

   return resolver;
 }

}
All that's left to do is adding a simple controller class, which maps
to /hello and renders hello.jsp ...
@Controller
public class FooController {
```

```
 @RequestMapping("/hello")
 public String helloWorld(Model model) {
   //let's pass some variables to the view script
   model.addAttribute("wisdom", "Goodbye XML");

   return "hello"; // renders /WEB-INF/views/hello.jsp
 }

}
..and even smaller jsp file:
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"%>
<html>
   <head>
       <title>This goes to top</title>
   </head>
   <body>
       <h1>Hello World, ${wisdom}!</h1>
   </body>
</html>
The final application source contains just 4 files + pom.xml:
-- pom.xml
-- src
  -- main
     |-- java
     |   -- com
     |      -- zt
     |         -- helloWeb
     |            -- controllers
     |                -- FooController.java
     |             -- init
     |                -- Initializer.java
     |                -- WebappConfig.java
     |-- webapp
         -- WEB-INF
            -- views
               -- hello.jsp
```

This app can be easily launched from Eclipse and also built on a remote server using Maven. Try it, it's a fully functional HelloWorld webapp! Even though it is possible to set up the webapp without any XML files, I leave it up to you decide how big of an improvement this actually is. It's certainly not a silver bullet. Also, app containers' support for the new options is still buggy. For example, the webxml-less. It is buggy on the 7.1.1 version and fails completely on 7.0.2. You might also not be always able to use servlet 3.0 capable servers in production.

For me, the most offensive part of XML is having to include a boatload of namespaces in Spring's applicationContext.xml. For example, I never create that file from scratch and copy-paste it instead:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:mvc="http://www.springframework.org/schema/mvc"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context http://www.spring-
framework.org/schema/context/spring-context.xsd
   http://www.springframework.org/schema/mvc http://www.springframe-
work.org/schema/mvc/spring-mvc-3.0.xsd" >
```

Web.xml is far less verbose however and works even on JBoss. In the above setup, the class Initializer can be replaced by the following web.xml (we'll still be using Spring's javaconfig, and for compatibility's sake I reduced the servlet version to 2.5):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd">

 <servlet>
   <servlet-name>mvc</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
   <!-- we'll use AnnotationConfigWebApplicationContext instead of the
default XmlWebApplicationContext... -->
   <init-param>
     <param-name>contextClass</param-name>
     <param-value>org.springframework.web.context.support.Annotation-
ConfigWebApplicationContext</param-value>
   </init-param>

   <!-- ... and tell it which class contains the configuration -->
   <init-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>com.zt.helloWeb.init.WebappConfig</param-value>
   </init-param>

   <load-on-startup>1</load-on-startup>

 </servlet>

 <servlet-mapping>
   <servlet-name>mvc</servlet-name>
   <url-pattern>/</url-pattern>
 </servlet-mapping>

 <welcome-file-list>
   <welcome-file>/</welcome-file>
 </welcome-file-list>

</web-app>
```

## Default**ServletHandler**

The caveat of mapping DispatcherServlet to "/" is that by default it breaks the ability to serve static resources like images and CSS files. To remedy this, I need to configure Spring MVC to enable defaultServletHandling.
To do that, my WebappConfig needs to extend WebMvcConfigurerAdapter and override the following method:

```
@Override

 public void configureDefaultServletHandling(DefaultServletHandle
rConfigurer configurer) {
   configurer.enable();
 }
```

## Introducing **Hibernate**

Adding Hibernate to the mix is rather easy. All I need to do is add a few beans to the WebappConfig class. Which beans to add depends whether you wish to stick to the classic Hibernate API that uses SessionFactory or switch to a modern JPA, which uses EntityManager instead. Note that JPA is a specification and we're still using Hibernate as an implementation. The basic concepts and programming model are the same for classic Hibernate and JPA, but in my opinion JPA is easier to set up and more beautiful, not to mention it being an official JEE specification. By the way, even when using plain old Hibernate, the class annotations such as @ Entity still belong to JPA.

There are a couple of things to do in order to introduce Hibernate. First, I don't want to hardcode database connection parameters and would prefer them to be in the db.properties file. This can be achieved by annotating the class with @ PropertySource("classpath:db.properties") and introducing a field:

```
@Autowired Environment env;
Secondly, I need to create a dataSource bean:
@Bean(destroyMethod = "close")
public DataSource getDataSource() {
DriverManagerDataSource ds = new DriverManagerDataSource(env.
getProperty("url"));
ds.setDriverClassName(env.getProperty("driver"));
ds.setUsername(env.getProperty("user"));
ds.setPassword(env.getProperty("pass"));

 return ds;
}
```

For development time, using DriverManagerDataSource is just fine, but for production I would recommend a proper connection pool such as commons-dbcp. In that case, the method body would look like this:

```
@Bean
public DataSource getDataSource() {
BasicDataSource ds = new BasicDataSource();
ds.setUrl(env.getProperty("url"));
ds.setDriverClassName(env.getProperty("driver"));
ds.setUsername(env.getProperty("user"));
ds.setPassword(env.getProperty("pass"));
ds.setValidationQueryTimeout(5);
ds.setValidationQuery("select 1 from DUAL");

return ds;
}
I'm using org.apache.commons.dbcp.BasicDataSource here and also
added a keepalive query to prevent closing the connection.
How to add commons-dbcp to the pom:
<!-- DB connection pooling for production applications -->
<dependency>
<groupId>commons-dbcp</groupId>
<artifactId>commons-dbcp</artifactId>
<version>1.4</version>
</dependency>
The properties are stored in src/main/resources/db.properties:
driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:res:snoutdb
hibernate.dialect=org.hibernate.dialect.HSQLDialect
user=SA
pass=
```

***Note that I'm using HSQLDB as a database here. It's great for testing and it's easy to include the necessary test data to the application. I'm using snoutdb as a database name (more on that later) and it's accompanied by two additional files in src/main/resources: snoutdb.properties which is HSQL's configuration and snoutdb.script that contains my application's schema and sample data.***

There are three more things to add: EntityManager factory, TransactionManager and an interceptor that allows session/entityManager to be used in a view layer. However, they depend on whether I'm using JPA or pure Hibernate.

## JPA

For JPA , adding the EntityManager factory is easy:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManager-
FactoryBean();
emf.setDataSource(getDataSource());
emf.setPackagesToScan("com.zt.helloWeb.model");

//let Hibernate know which database we're using.
//note that this is vendor specific, not JPA
Map<String, Object> opts = emf.getJpaPropertyMap();
opts.put("hibernate.dialect", env.getProperty("hibernate.dialect"));

HibernateJpaVendorAdapter va = new HibernateJpaVendorAdapter();
emf.setJpaVendorAdapter(va);

return emf;
}
```

Adding TransactionManager is important because it makes it possible to avoid starting and flushing transactions/sessions manually. It works in concert with @EnableTransactionManagement on the config class and @Transactional on the DAO class.

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
OpenEntityManagerInViewInterceptor interceptor = new OpenEntityManagerIn-
ViewInterceptor();
interceptor.setEntityManagerFactory(entityManagerFactory().getObject());
registry.addWebRequestInterceptor(interceptor);
}
```

## Pure **Hibernate**

For pure Hibernate, the steps are the same, but details differ.
Hibernate uses SessionFactory instead of EntityManagerFactory:

```
@Bean
public SessionFactory buildSessionFactory() {
LocalSessionFactoryBuilder builder = new LocalSessionFactoryBuilder(getDataSo
urce());
return builder.scanPackages("com.zt.helloWeb.model").buildSessionFactory();
}
```

TransactionManager is also different:

```
@Bean
public HibernateTransactionManager transactionManager() {
return new HibernateTransactionManager(sessionFactory());
}
```

And the interceptor class differs too. Note that there are separate versions of OpenSessionInViewInterceptor classes depending on whether Hibernate 3 or 4 is used.

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
OpenSessionInViewInterceptor interceptor = new OpenSessionInViewIntercep-
tor();
interceptor.setSessionFactory(buildSessionFactory());
registry.addWebRequestInterceptor(interceptor);
}
```

Hibernate also needs to be added to pom.xml. Again, this is different whether or not JPA is used:

```
<!-- When using JPA -->
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-entitymanager</artifactId>
 <version>4.1.2</version>
</dependency>
<!-- When using pure Hibernate -->
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-core</artifactId>
 <version>4.1.2</version>
</dependency>
```

There is one more difference between setting up JPA and Hibernate. For JPA, the EntityManager is injected (not EntityManagerFactory), and it's done via @ PersistenceContext annotation:

```
@PersistenceContext
private EntityManager entityManager;
```

Hibernate's SessionFactory is injected the standard way:

```
@Autowired
private SessionFactory sessionFactory;
```

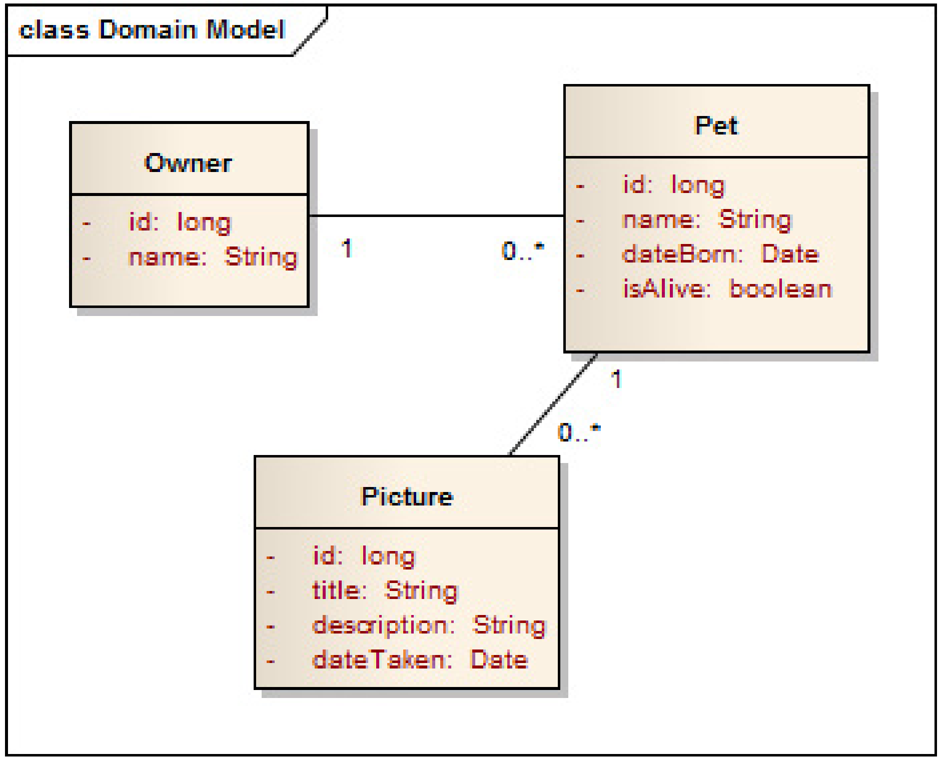
The complete WebappConfig class I end up with is this:

```
@Configuration
@ComponentScan("com.zt.helloWeb")
@EnableWebMvc
@PropertySource("classpath:db.properties")
@EnableTransactionManagement
public class WebappConfigJpa extends WebMvcConfigurerAdapter {

 @Autowired
 Environment env;

 //Tell SpingMVC where to find view scripts
 @Bean
 public InternalResourceViewResolver setupViewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewRe-
solver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");

   return resolver;
 }

 //Enable serving static resources even when DispatcherServlet is mapped
to /
 @Override
 public void configureDefaultServletHandling(DefaultServletHandlerConfigur
er configurer) {
   configurer.enable();
 }

 //Enable accessing entityManager from view scripts. Required when using
lazy loading
 @Override
 public void addInterceptors(InterceptorRegistry registry) {
   OpenEntityManagerInViewInterceptor interceptor = new OpenEntityMan-
agerInViewInterceptor();
   interceptor.setEntityManagerFactory(entityManagerFactory().getOb-
ject());
   registry.addWebRequestInterceptor(interceptor);
 }

 //Set up dataSource to be used by Hibernate. Also make sure the connection
doesn't go down
 @Bean
 public DataSource getDataSource() {
   BasicDataSource ds = new BasicDataSource();
   ds.setUrl(env.getProperty("url"));
   ds.setDriverClassName(env.getProperty("driver"));
   ds.setUsername(env.getProperty("user"));
   ds.setPassword(env.getProperty("pass"));
   ds.setValidationQuery("select 1 from DUAL");
   return ds;
 }

 //Set up JPA and transactionManager
 @Bean
 public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
   LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityMan-
agerFactoryBean();
   emf.setDataSource(getDataSource());
   emf.setPackagesToScan("com.zt.helloWeb.model");

   //let Hibernate know which database we're using.
   //note that this is vendor specific, not JPA
   Map opts = emf.getJpaPropertyMap();
   opts.put("hibernate.dialect", env.getProperty("hibernate.dialect"));

   HibernateJpaVendorAdapter va = new HibernateJpaVendorAdapter();
   emf.setJpaVendorAdapter(va);

   return emf;

 }

@Bean
 public PlatformTransactionManager transactionManager() {
   return new JpaTransactionManager(entityManagerFactory().getObject());
 }
}
```

## Enough with the setup, **let’s write some code!**

Once upon a time, Sun started with the PetStore sample application and Spring continued it with the PetClinic. Now that the cute little goblins have been bought and cared for, let's publish their pictures for the whole world to see - introducing Snoutbook!

I didn’t want to use something trivial, but didn’t want too much complexity either. I finally settled for a three-class model: owners, pets and pictures. While simple, it still allows for demonstrating one-to-many and many-to-one relationships.



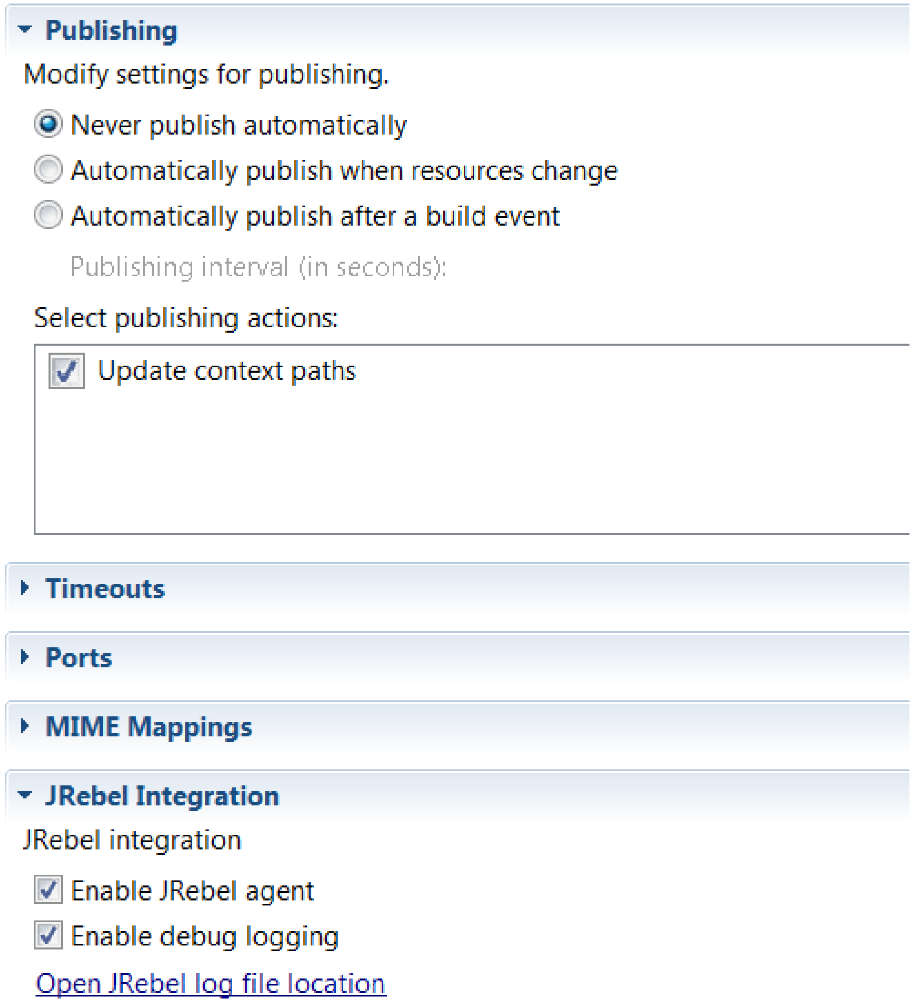
The application itself also is simple at first. It consists of only three screens: Owners list, addPet form and Pet’s profile.

## Developing **with JRebel**

**Disclaimer:** I work on the JRebel team at ZeroTurnaround, but before I worked at ZT I used JRebel in every project I could.

Using JRebel puts app development on the fast track. I can add new model classes, decorate them with JPA annotations and see the results in the browser immediately after pressing F5 to refresh. The same goes for adding things like new Spring beans and form controllers. The only time throughout this entire process that I needed to start the app server again was when a Windows update forcibly restarted my computer.

**Note:** When starting the app server from within Eclipse, you need to disable automatic publishing and enable the JRebel agent on the server page. When starting the container from a command line, I need to make sure the JVM argument -javaagent:path/to/jrebel.jar gets passed to the JVM that is running the server. It's usually the startup script such as catalina.bat, or the MAVEN_OPTS system property when using mvn jetty:run for example.


Publishing

Modify settings for publishing.

- ● Never publish automatically
- ○ Automatically publish when resources change
- ○ Automatically publish after a build event

Publishing interval (in seconds):

Select publishing actions:

- ✓ Update context paths

Timeouts

Ports

MIME Mappings

JRebel Integration

JRebel integration

- ✓ Enable JRebel agent
- ✓ Enable debug logging

Open JRebel log file location

So in fact you do need a little XML here: rebel.xml, a configuration file that makes JRebel monitor the folder where Eclipse is compiling the classes to and reloading them as necessary. Just so you know, rebel.xml is best generated in Eclipse by right-clicking on the project and selecting JRebel->Generate rebel.xml...

JRebel also monitors the web resources directory, which means that I don't need to run the build scripts for any change I'm making. As long as the "build automatically" option is selected in Eclipse, all I need to do is save the file (whether it's Java, JSP or CSS file), ALT-TAB to the browser and hit refresh.

The generated rebel.xml is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.zeroturnaround.com"
 xsi:schemaLocation="http://www.zeroturnaround.com http://www.zeroturnaround.
com/alderaan/rebel-2_0.xsd">

 <classpath>
   <dir name="${rebel.workspace.path}/HelloWeb/target/classes">
   </dir>
   <dir name="${rebel.workspace.path}/HelloWeb/src/main/resources">
   </dir>
 </classpath>

 <web>
   <link target="/">
     <dir name="${rebel.workspace.path}/HelloWeb/src/main/webapp">
     </dir>
   </link>
 </web>

</application>
```

The interesting part here is ${rebel.workspace.path}, which gets substituted with the real path when starting the server. This means that you can commit the file to a version control system and not worry about different team members having different workspace locations.

## Model **classes**

I'm starting with Owner and Pet classes for now and add the Picture class later (again, with JRebel this is all done without restarting the application).

```
@Entity
public class Owner {

 @Id
 @GeneratedValue
 private Long id;

 private String name;

 @OneToMany (cascade=CascadeType.ALL, mappedBy="owner")
 private List pets;

//getters and setters omitted

}

@Entity
public class Pet {

 @Id
 @GeneratedValue
 private Long id;

 private String name;
 private Date dateBorn;
 private boolean alive;

 @ManyToOne
 @JoinColumn(name="owner")
 private Owner owner;

 //getters and setters omitted

}
```

I don't need to add JPA annotations to every field, as JPA uses convention over configuration and assumes the field name maps to column "name". The relationship between the Owner and a Pet is one-to-many and many-to-one respectively. That's it. In case your tables and column names correspond closely to class names and their fields, you only need to explicitly annotate the classes and identify fields & relationships.

## Adding a **controller**

Spring finds and instantiates the controller bean automatically since the class is annotated with @Controller and our WebappConfig class was annotated with @ComponentScan("com.zt.helloWeb"). The dependency to the yet-to-be-written SnoutDao object is provided automatically, I don't even need to add a setter function.

```
@Controller
public class SnoutContoller {

 @Autowired
 private SnoutDao dao;

 @RequestMapping("/owners")
 public String listOwners (Model model) {
   List owners = dao.getOwners();
   model.addAttribute("owners", owners);

   return "owners";
 }

}
```

Spring finds and instantiates the controller bean automatically since the class is annotated with @Controller and our WebappConfig class was annotated with @ComponentScan("com.zt.helloWeb"). The dependency to yet-to-be-written SnoutDao object is provided automatically, I don't even need to add a setter function.
The annotation-based approach in SpringMVC departs significantly from the older Controller based approach where you had to implement ModelAndView#handleRequest(request, response).
Now the whole job is done via annotations and that gives tremendous flexibility when defining handler functions. You can add Model object in your argument list, but you don't have to. You can also bind http request parameters to method arguments, which makes the code even cleaner. You can return a ModelAndView object if you like, but are also free to return a String object, which will denote the view script to be rendered.
You can return the output directly to the browser by adding a @ResponseBody annotation to the method. The flexibility here is almost unlimited, but it does not become vague or ambiguous. In a nutshell, my method listOwners is mapped to /owners URL, uses a Model argument for passing data to the view script easily and renders owners.jsp as a result.

## The **DAO**

The SnoutDao's humble beginnings:

```
@Repository
@Transactional
public class SnoutDao {

 @PersistenceContext
 private EntityManager entityManager;

 public List getOwners() {
   return entityManager.createQuery("select o from com.zt.helloWeb.model.Owner o", Owner.class).getResultList();
 }

}
```

Dao classes should have @Repository and @Transactional annotations. The @Repository annotation denotes a bean with a specialized role and @Transactional makes Spring manage transactions for you. Otherwise you would need to open/close sessions and transactions yourself manually.
Returning a list of objects from a database is a simple one-liner in this CRUD application. JPA provides a ton of options for retrieving your data from a database, but for now this will suffice.

## JSP files, **TLD's and the like**

The last piece of the puzzle is the view script

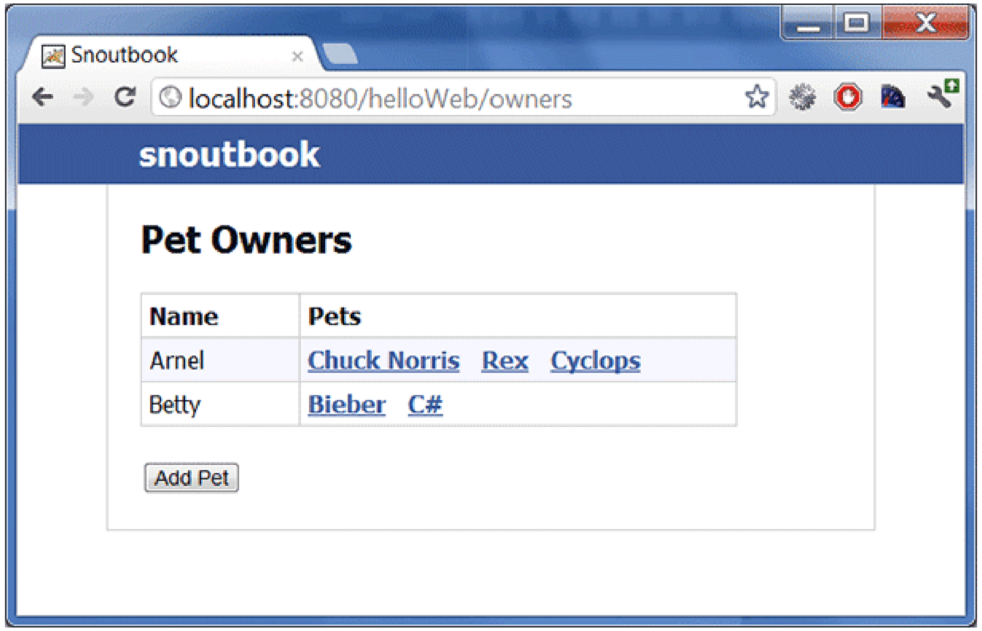
```
<%@ include file="/WEB-INF/layout/header.jsp" %>

<div id="content">
 <h1>Pet Owners</h1>
 <table id="petowners">
   <tr>
     <th>Name</th>
     <th>Pets</th>
   </tr>
   <c:forEach items="${owners}" var="owner">
     <tr>
       <td>${owner.name}</td>
       <td>
         <c:forEach items="${owner.pets}"
var="pet">
           <a href="pet/${pet.id}">${pet.name}</a>
         </c:forEach>
       </td>
     </tr>
   </c:forEach>
 </table>
 <button onclick="window.location='addPet'">Add
Pet</button>
</div>

<%@include file="/WEB-INF/layout/footer.jsp" %>
```

The trick here is to make good use of various Tag Libraries. Think of them as frameworks for building HTML markup in view scripts. For more complex applications, it's often a good idea to create your own as well, but in my case I'm sticking to the standard and Spring Framework ones. This makes it easy to iterate over a collection. I'm already generating links to other pages in my application (namely /pet/{id}), which I will add next.

I'm including header and footer.jsp files that contain the rest of the HTML markup that makes up a page. Header.jsp also contains the import statements for all the JSTL libraries:

```
<%@page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Snoutbook</title>
<link rel="stylesheet" href="/helloWeb/snout.css" />
</head>
<body>
<div id="blueBar">&nbsp;</div>
<div id="wrapper">
<div id="container">
<div id="top"><a href="/helloWeb/owners">snoutbook</a></div>
```

All that's left is starting the server from within Eclipse and checking the result:



Pet Owners

| Name | Pets |
|---|---|
| Arnel | Chuck Norris Rex Cyclops |
| Betty | Bieber C# |

## Moving **along**

Now it's time to add the third entity "Picture"

```
@Entity
@Table(name="picture")
public class PetPicture {

 @Id
 @GeneratedValue
 private Long id;
 private String title;
 private String description;

 private Date dateTaken;

 @ManyToOne
 @JoinColumn (name="pet")
 private Pet pet;
 //getters and setters omitted
}
```

Note that this time the table name and the class name do not match, which is why I need the @Table annotation.

I also need to add the relationship to the Pet class:

@OneToMany (mappedBy="pet")
private List<PetPicture> pics;

Next I'm adding the method to my SnoutController:

```
@RequestMapping("/pet/{petId}")
public String pet(@PathVariable Long petId, Model model) {
Pet pet = dao.getPet(petId);
model.addAttribute("pet", pet);

return "viewPet";
}
```

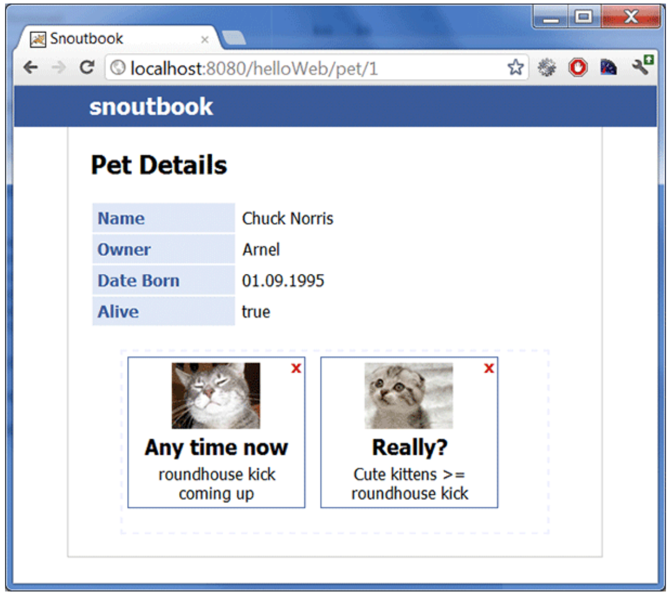
What just happened? I mapped part of the request URI as an input variable! This is achieved by using {varname} in request mapping and using a parameter with the same name in the method (together with @ PathVariable annotation).
The addition to the SnoutDao is simple:

```
public Pet getPet(Long petId) {
 return entityManager.find(Pet.class, petId);
}
```

Finally, since I return viewPet from the method, I need to create a view-Pet.jsp in the WEB-INF/views folder.

```
<%@ include file="/WEB-INF/layout/header.jsp" %>

<div id="content">
 <h1>Pet Details</h1>
 <table class="petDetails">
   <tr>
     <th scope="row">Name</th>
     <td>${pet.name}</td>
   </tr>
   <tr>
     <th scope="row">Owner</th>
     <td>${pet.owner.name}</td>
   </tr>
   <tr>
     <th scope="row">Date Born</th>
     <td>
       <fmt:formatDate value="${pet.dateBorn}" pattern="dd.
MM.yyyy" />
     </td>
   </tr>
   <tr>
     <th scope="row">Alive</th>
     <td>${pet.alive}</td>
   </tr>
 </table>

 <div class="photos">
   <c:forEach items="${pet.pics}" var="pic">
     <div class="pic">
       <div class="del">
         <a href="/helloWeb/deletePic/${pic.id}">x</a>
       </div>
       <img class="img" src="/helloWeb/img/${pic.id}.jpg"
alt="${pic.title}" />
       <h2>${pic.title}</h2>
       <p>${pic.description}</p>
     </div>

   </c:forEach>
   <div style="clear:both">
   </div>
 </div>

</div>

<%@include file="/WEB-INF/layout/footer.jsp" %>
```

The result is here:



Snoutbook

localhost:8080/helloWeb/pet/1

snoutbook

**Pet Details**

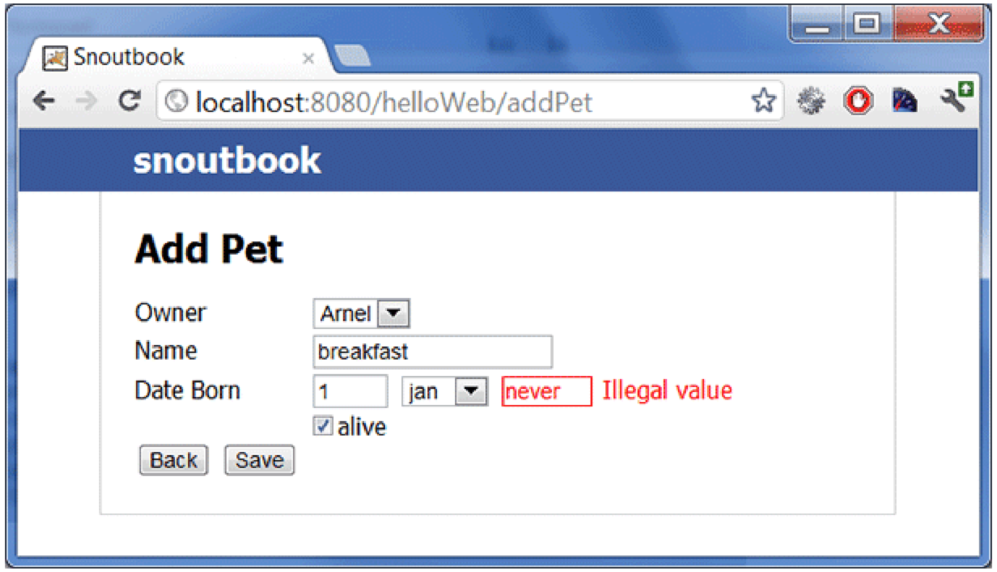
| | |
|---|---|
| Name | Chuck Norris |
| Owner | Arnel |
| Date Born | 01.09.1995 |
| Alive | true |

**Any time now**
roundhouse kick coming up

**Really?**
Cute kittens >= roundhouse kick

## Creating **Forms**

The last step in building the purring network I'm willing to share is adding a new Pet.
I'm going skip ahead and show the final result first this time, and then detail the steps necessary to get there.



Snoutbook

localhost:8080/helloWeb/addPet

snoutbook

**Add Pet**

| | |
|---|---|
| Owner | Arnel |
| Name | breakfast |
| Date Born | 1 jan never Illegal value |
| | ✓ alive |

Yes, I know about fancy date pickers and date string parsing, but for education's sake let's use three distinct values for inserting a date. There are several ways to accomplish this. Often form controls are directly bound to the underlying model class, but usually this creates problems and complexity (or it may not, YMMV). I like to create a separate form class that corresponds to the HTML and the user model. It is similar, but differnent from the program model (yes, I'm referencing Joel Splosky here - http://www.joelonsoftware.com/uibook/chapters/fog0000000058.html and http://www.joelonsoftware.com/uibook/chapters/fog0000000060.html).

The Form class in question looks like this:

```
public class PetForm {

 @NotEmpty
 @Length(min=1, max=32)
 private String name="breakfast";

 @Range(min=1, max=31)
 private int dayBorn=1;

 @Range(min=0, max=11)
 private int monthBorn=0;

 @Range(min=1900, max=2038)
 private int yearBorn=1990;

 private Date dateBorn;

 private boolean alive=true;

 private long owner;
```

```
 public void validate (Errors errors) {
   Calendar cal = new GregorianCalendar(yearBorn, month-
Born, dayBorn);
   dateBorn = cal.getTime();
   if (dateBorn.after(new Date())) {
     errors.rejectValue("dateBorn", null, "Date is in fu-
ture");
   }
 }
```

Often Spring tutorials advocate using separate Validator classes and sometimes it is a great idea. However, since I'm using a separate form Object, it makes sense to couple that with validation. I'm combining Spring MVC with bean validation here. This allows for declaring mundane and simple constraints declaratively, and then adding some additional complex validation where needed. There's no point in validating String length manually when the annotation-based approach fits so neatly here. All that's left for me to do is building the dateBorn value from its components and validating the result.

## Form **controller**

The controller class is quite large, but it has to do some heavy lifting as well. The controller has to first set up the form and later handle the submitted input.

I'm using a combination of @RequestMappings here. First I'm annotating the class itself with the URI path:

```
@RequestMapping("/addPet")
public class AddPetFormController
```

Next, I'm creating a method "setupForm" and make it handle GET requests only.

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(Model model) {
PetForm form = new PetForm();
model.addAttribute("petForm", form);
return "addPet";
}
```

It is not required to pass all the data to the view script in a single method. By annotating a method with @ModelAttribute("myAttributeName"), it is invoked and the result made available to the script. I'm using this approach to populate the Owner and Month selection boxes.

```
@ModelAttribute("owners")
 public Collection getOwners(){
   return dao.getOwners();
 }

 @ModelAttribute("monthNames")
 public Collection<Month> getMonthNames() {
   String[] names = new String[]{"jan", "feb",
"mar", "apr", "may", "jun", "jul", "aug", "sep",
"oct", "nov", "dec"};
   List answer = new ArrayList();
   for (int i=0; i<names.length; i++) {
     answer.add(new Month(i, names[i]));
   }

   return answer;
 }

class Month {
   private int id;
   private String label;

   public Month(int id, String label) {
     this.id=id;
     this.label = label;
   }

}
```

***Note: It is not required to pass all the data to the view script in a single method. By annotating a method with @ModelAttribute("myAttributeName"), it is invoked and the result is made available to the script. I'm using this approach to populate the Owner and Month selection boxes.***

## Handling the **submit**

```
@RequestMapping(method = RequestMethod.POST)
 public String handleSubmit(@Valid PetForm form, BindingResult result, SessionStatus sta-
tus) {

   form.validate(result);

   if (result.hasErrors()) {
     return "addPet";
   }
   else {
     Pet pet = new Pet();
     pet.setName(form.getName());
     pet.setAlive(form.isAlive());

     Calendar cal = new GregorianCalendar(form.getYearBorn(), form.getMonthBorn(), form.
getDayBorn());
     pet.setDateBorn(cal.getTime());

     Owner o = dao.findOwner(form.getOwner());
     pet.setOwner(o);
     o.getPets().add(pet);
     dao.saveOwner(o);

     status.setComplete();
     return "redirect:owners";
   }

 }
```

Keep in mind that the @Valid annotation here validates fields marked with bean validation annotations automatically. The validation process itself is simple - validators are invoked, passing the Errors object along, and the controller later checks whether something raised an error or not. If validation passes, the controller builds a model object from the PetForm and persists it.

Another good approach is using GET-POST-redirect approach, where the browser is redirected away after making a successful POST request. That prevents double entries when refresh is pressed in the browser.

## Finally, the **jsp file**

```
<%@ include file="/WEB-INF/layout/header.jsp" %>

<div id="content">
 <h1>Add Pet</h1>
 <form:form action="addPet" method="POST" modelAttribute="petForm">
   <div>
     <label>Owner</label>
     <form:select path="owner" items="${owners}" itemValue="id"
itemLabel="name" />
   </div>
   <div>
     <label>Name</label>
     <form:input path="name" cssErrorClass="error" />
   </div>
   <div>
     <label>Date Born</label>
     <form:input path="dayBorn" size="2" cssErrorClass="error" />
     <form:select path="monthBorn" items="${monthNames}" itemValue="id"
itemLabel="label" />
     <form:input path="yearBorn" size="4" cssErrorClass="error" />
     <form:errors path="yearBorn">
       <span class="error">Illegal value</span>
     </form:errors>
     <form:errors path="dateBorn" cssClass="error" />
   </div>
   <div>
     <label>
     </label>
     <form:checkbox path="alive" />
     <label>alive</label>
   </div>

   <button type="button" onclick="window.location='owners'">Back</button>
   <form:button value="Save" name="add">Save</form:button>
 </form:form>
</div>

<%@include file="/WEB-INF/layout/footer.jsp" %>
```

Do you see the ease of mapping collections to a <select> element together with ids and labels? I'm using Spring's form tag library here for ease of use, where the association is done via path attribute. Errors are signalled using form:errors tag or cssErrorClass attribute on the form tags itself.

I prefer Spring's approach due to the lack of magic involved. There is no fragile component tree and it is clear how state is stored. Even though I'm using Spring's form tags, they resolve to simple HTML elements and to bind a variable to "name" field in the PetForm class, I could also use <input type="text" name="name" value="" />.

## The last six **lines of code**

needed in Snvoid outDao:

```
public saveOwner(Owner o) {
   entityManager.persist(o);
 }
 public Owner findOwner(Long id) {
   return entityManager.find(Owner.class, id);
 }
```

And it works!
And I still haven't redeployed or restarted anything. At all.
In fact, writing the content that you are reading now took more time than developing the app itself.
Thank you JRebel!

## What’s coming next? **Improvements!**

The little cute application is far for being complete. There are still some snags I’m not happy with and will discuss them at some point later on. I particularly dislike the fact that the mvc and persistence config are both in the single AppConfig class. I’m going to change that. I also wish for a better way to address templating issues than <%@ include’ing (I’m not swearing, this is the actual syntax) header and footer files in view scripts. And I’m also going to address scaling up, when, for example, I wish to separate webapp from the core logic completely a provide the rest of the world with some kind of an API to fetch all those cute pictures programmatically.

## Try it **yourself!**

I uploaded the sample app to BitBucket and you can explore the files in the Bitbucket repository
To clone the repo directly and start it up on your machine, install Mercurial and
hg clone https://bitbucket.org/arnelism/snoutbook

## About **the author**

Arnel Pällo is a Java engineer at ZeroTurnaround. He works on adding support for web-service frameworks into JRebel and is a major fanboy of pretty code. He plays Bioware games and hacks his Android phone on weekends. He also likes salsa dancing and can make a great Mojito. You can connect with Arnel on Facebook.

ZEROTURNAROUND

Thanks for reading

Contact Us

Twitter: @RebelLabs

Web: http://zeroturnaround.com/rebellabs

Email: labs@zeroturnaround.com

**Estonia**
Ülikooli 2, 5th floor
Tartu, Estonia, 51003
Phone: +372 740 4533

**USA**
545 Boylston St., 4th flr.
Boston, MA, USA, 02116
Phone: 1(857)277-1199