Article Series: Migrating Spring Applications to Java EE 6

Goal
This series consists of a multi-part article that covers migrating Spring Applications to Java EE 6 technology. It also consists of sample code and a working project. In the series we will discuss the rationale for migrating your applications from Spring to Java EE 6 and we will show you real examples of upgrading the web UI, replacing the data access layer, migrating AOP to CDI interceptors, migrating JMX, how to deal with JDBC templates, and as an added bonus we will demonstrate how to integration test your Java EE 6 applications using Arquillian.

About the authors

Bert Ertman (@BertErtman) is a Fellow at Luminis in the Netherlands and a Sun/Oracle recognized Java Champion. Besides his day-job he is a JUG Leader for the Netherlands Java User Group (3500 members).

Paul Bakker (@pbakker) is a senior developer at Luminis Technologies in the Netherlands and a contributor on the JBoss Seam, Arquillian, and Forge projects.

Both authors have extensive experience in building enterprise applications using a variety of technologies ranging from the pre-J2EE, J2EE, Spring, and modern Java EE technologies. They have been discussing the use of new enterprise technologies regularly and had endless discussions on Spring vs. Java EE. Currently, they believe that both new, green field, enterprise applications, and large-scale maintenance migration on legacy apps can be done best using Java EE 6 technology. The authors have been evangelizing Java EE (6) technology at various conferences around the world, including J-Fall, Jfokus, Devoxx, and JavaOne. This series of articles is based upon their well-received JavaOne 2011 presentation titled “Best Practices for Migrating Spring to Java EE 6”.

PART 1 - Introduction

Over the past couple of years there has been a growing interest in all things Java EE. Generally speaking Java EE 5 to some extent, but Java EE 6 in particular marked the re-birth of Java EE's credibility with developers. While Java EE is considered to be hot again, people wonder whether to jump on this train. Back in the days around 2003/2004, J2EE was at the height of its unpopularity. Developers hated its ivory tower specification process and its blindness to solving real-world problems. By that time a book came out that radically changed the way that enterprise Java development would be done for the next five years. This book of course was the "J2EE Design and Development" book, later followed by the "J2EE without EJB" book, both from Rod Johnson et all.

Developers loved these books and their way of presenting better solutions to the real-world problems they were facing at that time. In fact people still love those books without realizing that the world has changed dramatically ever since. The reality check here is to wonder whether the rhetorics set forth by Rod Johnson in his 2003/2004 books are still actual today. In our opinion all of the problems presented in the book can now be easily solved using standard, boiler-plate free, configuration-by-exception, easy to comprehend, lightweight POJO-based Java EE technologies. So if you still care about those books, the best way to show your appreciation is probably to use them as your monitor stand.

There are two ways to approaching the Spring vs. Java EE discussion nowadays. You are either in the situation that you have to build a green field enterprise application and you ponder what gear to stuff into your toolbox. Or you are facing a serious legacy enterprise application upgrade issue. Upgrading six or seven year old Java technology, being it Spring or J2EE inflicts a lot of pain anyway. The discussion whether or not to use Spring vs. Java EE for new enterprise Java applications is a no-brainer in our opinion. It took a while but Java EE has finally made the leap to be a standard, lightweight, fitting solutions to the vast majority of real-world development challenges in the mainstream enterprise applications space. You should have no single reason beyond choosing the standard. The question whether Spring or Java EE is the better solution is not the main focus of this series of articles. We want to present you an approach on how to tackle the problem of upgrading your legacy Spring applications from the past. It is important to understand that there are means to an end. In other words we don't encourage you to take the migration all the way or else it wouldn't be complete. Our intention is to present you with a migration path that you can head onto for just as many steps as you are willing to take. There can be all sorts of valid reasons not to take the migration all the way. These can be based on available time, money, and expertise. We are sure though that if you find yourself stuck with a legacy Spring application and wonder how to move forward from this point on towards a modernized enterprise application that is considered fit enough to last at least another five years, this series of articles will present you with hands-on, ready-to-use examples.

Why migrate?

Before we start talking about what a migration path for legacy Spring applications would look like, it is valid to ask yourself the imminent question "why should I migrate my application in the first place?". This is definitely a valid question and there are multiple answers to this one. First off, Spring was born out of frustration with mainstream technology, has been a hugely successful - do no evil - Open Source project, and has since then fallen a prey to the hungry minds of Venture Capitalists and finally into the hands of a virtualization company called VMware. Within the Java world VMware is probably best known for being able to run Windows on your Mac, and we're not even sure if that is a good thing to be remembered of. Despite various efforts Spring nor any of its sub-projects has ever become a true Java standard. While the different companies and individuals behind the Spring framework have been doing some work in the JCP their voting behavior on important JSRs is peculiar to say the least. From how we see it, neither the Spring framework, nor the companies behind SpringSource have any benefit of truly standardizing it ever. This is of course a political motive and most developers don't care about that at all, so let's move on to more technical arguments of why a migration is a good solution for your problem. The technical reason is that upgrading your old-school Spring applications, even to modern Spring-based solutions, requires a lot of work anyway. Some technologies that might be used in the original application might even be considered end-of-life soon, or worse, are already part of a dark past.

When we mention "old-school Spring apps" we picture lots of complex XML configuration, an outdated ORM solution like JDBC templates, Kodo, or TopLink, and an old fashioned approach towards the web using a deprecated extension based Web MVC SimpleFormController. Just to give you an example.

Can you still find developers that master the techniques and frameworks used in the application? And if so, can you still find them say in another five years? From our own experience we have also seen applications that show 'tells' of each and every year that maintenance took place. We could tell from the use of different web an/or ORM frameworks and techniques that were all used within the same applications as a representative of the current most popular framework of that time. Ploughing through this framework erosion is tedious, boring, and above all complicated work leaving much room for mistakes causing even more damage to the application. And so this calls for an extreme makeover approach. Parts of the application have to be radically renewed, sometimes from the ground up. While you are confronted with that, why not take a little extra step and take it to the standard?

Busting myths
If you still have some concerns whether Java EE is suited for the job we encourage you to read on and get some misunderstandings out of the way first. We already discussed some of the J2EE problems of the past. However, some developers completely stopped looking at new developments in the Java EE space and might have lost track of the current state of technology. It is not the aim of this article to introduce you to Java EE 6, but as a quick refresher, let's take a look at some key technologies of both Spring and Java EE 6 and see how they compare and moreover to see how Java EE caught up on Spring.

Lightweight
Some of the complaints of the past include the statement that Java EE has become fat and bloated. Some people even rebranded Java EE to Java Evil Edition. Is this still true today? You can of course take our word for it, but it is probably best to take a look at modern application server startup times. Firing up the latest JBoss AS 7 Application Server from scratch and deploying a full blown Java EE 6 application into the server takes somewhere between two and five seconds on a standard machine. This is in the same league as a Tomcat / Spring combo. There is a dramatic difference however if you consider the size of the deployment archive as well. A fairly standard Java EE 6 application will take up about 100 kilobytes while a comparable Spring application weighs in at a whopping 30 Megabytes! The Java EE 6 application is of course benefiting heavily from the fact that all Java EE 6 APIs are provided by the Application Server runtime. One of our

favorite quotes is from JBoss' own Andrew Lee Rubinger stating that "nowhere in the Java EE spec does it say that Java EE servers should be heavyweight and slow..". Exactly true!

Dependency injection
From a technical standpoint it is interesting to take a look at some of the technology capabilities that Java EE lacked in the past and that have put Spring in the driver's seat. Let's start with one of the most important reasons for developers to resort to Spring in the past: Inversion of Control (IoC) or whats more popularly known as Dependency Injection (DI). According to Wikipedia Inversion of Control is a style of software construction where reusable generic code controls the execution of problem-specific code. It carries the strong connotation that the reusable code and the problem-specific code are developed independently, which often results in a single integrated application. Inversion of Control is sometimes referred to as the "Hollywood Principle: Don't call us, we'll call you", because implementations typically rely on callbacks.

Although IoC is commonly thought off to be invented by the Spring framework authors, it is in fact an old design paradigm which was mentioned in official publications in the late eighties of the previous century. What the Spring framework did however is putting the IoC pattern to use in Java to heavily reduce coupling in enterprise Java implementations leading to much better testable code. It took quite a while for Java EE to catch up but finally as part of Java EE 6, the Context and Dependency Injection (CDI) specification was introduced to the Java platform, which has a very powerful contextual DI model adding extensibility of injectable enterprise services along the way. If you don't know about CDI yet, you are really missing out!

Aspect Oriented Programming
Another technique that was popularized by the Spring framework is Aspect Oriented Programming (AOP). In AOP the object oriented nature of the programming language is extended by a technique that weaves some cross-cutting concerns into the application at compile time or at runtime. Typical examples of cross-cutting concerns are logging and security, but it can be used for anything. A common pitfall when taking AOP too far is that your code might end up all asymmetric and unreadable. This is due to the fact that the aspect and its implementation are not in the same place. Determining what a piece of code will do at runtime at a glance will be really hard. This does not make AOP useless. If you use it responsibly and in the right amount it might prove to be a very powerful way to achieve separation of concerns and reusability. We often refer to that as "AOP Light" and this is exactly what Java EE Interceptors do for example. The implementation of interceptors on EJBs for example is somewhat limited making it harder to burn your fingers on too much AOP. In a way this is very much comparable to lightweight AOP in Spring AOP as well. The intercepter model was introduced in Java EE 5 and combining it with CDI (which was introduced in Java EE 6) creates a very powerful interceptor model that will give you a clean separation of concerns and greatly reduces the amount of code duplication in your application.

Testing
We already touched upon the subject of testing when we discussed IoC. Testing in J2EE was hard if not impossible. Mainly this was due to the fact that most J2EE components required lots of runtime services to be available in order to function properly. This is the death of a testable design. As Java EE returned to a POJO-based approach to enterprise components, testability increased a lot. Still also Java EE components require runtime services. In order to truly be able to test a Java EE component in its natural surrounding - the application server - we need a mocking framework. Or maybe not? Not that long ago a

new revelation on the testing horizon has surfaced. Its name is Arquillian. With Arquillian we can get rid of mocking frameworks and test Java EE components in their natural environment. Before running a unit test Arquillian creates a so-called micro deployment containing the Java EE component under test and its direct dependencies and creates a small deployable archive, deploys it into a running container or temporarily starts one, runs the tests and undeploys the archive from the container. Arquillian does not force an alternative way of unit testing upon you, but integrates non-intrusively with your favorite testing framework, e.g. JUnit. Arquillian is the ultimate companion to Java EE when it comes to testing. There is some good documentation with lots of examples available for Arquillian but as part of this article series we will show you how to write some basic integration tests of your own.

Tooling
Finally, we all have nightmares from firing up overweight tooling providing UML-savvy, crappy wizards that only worked once - when you were lucky - in order to do some decent productive development. Your computer felt like a car being trampled upon by a monster truck. Thankfully, this is now truly something from the past. All three major Java IDEs offer decent support for developing Java EE components without violating your computer's resources. Developing Java EE applications is no longer tight to single vendor's tooling solutions but you can use your IDE of choice, a little bit of Maven, and the application server of your liking. Setting up a Java EE project based on Maven from scratch is a breeze with tools like JBoss Forge which is pretty much compatible with all respected frameworks and application servers out there.

Comparing Spring and Java EE capabilities
For your convenience we listed a capabilities comparison matrix below to map Spring's technology to that of Java EE.

Capability	**Spring**	**JavaEE**
Dependency Injection	Spring Container	CDI
Transactions	AOP / annotations	EJB
Web framework	Spring Web MVC	JSF
AOP	AspectJ (limited to Spring beans)	Interceptors
Messaging	JMS	JMS / CDI
Data Access	JDBC templates / other ORM / JPA	JPA
RESTful Web Services	Spring Web MVC (3.0)	JAX-RS
Integration testing	Spring Test framework	Arquillian *

** Not part of the Java EE specification*

As a summary to what you have been reading so far it is safe to say that apparently it can all be done using plain vanilla lightweight Java EE. This leaves us with the migration issue at hand.

Migration approach
What would be the best way to move forward and start the migration? Some might be tempted to completely throw away the legacy application and start all over again. While this might appeal as an attractive solution and if time nor money constraints are an obstacle this might work. However for most realistic situations this is clearly not an option. What we do need is a gradual way of renovating the application without it collapsing on us on the way. Therefore we need a recipe that moves us forward step by step. The recipe should also allow for your own creativity and it should be possible to end the migration at any step that you would like. We came up with the following steps:

1. Upgrade Spring to the latest version
2. Replace old or out-dated frameworks (e.g. ORM, web frameworks) within Spring
3. Run Spring and Java EE container side-by-side
4. Replace Spring entirely
5. Remove Spring container

In the next part of the series we will elaborate on the migration approach and then take you through the different steps by showing you real-world examples of the migration in progress. There will be code examples and a link to a special GitHub project where you can take the project for a migration spin of your own, or subsequently apply all the change sets that we've prepared in order to demo your way through parts of the migration process.

PART 2 - Migration approach

In the first part of the series we introduced you to the idea of migrating your legacy Spring applications to Java EE 6. The design and framework choices that you made five or six years ago are increasingly adding up to the technical debt and maintenance burden of your current application. Since fixing these issues is a lot of work anyway, why not take it to the standard? In this article we will explain the migration steps that we came up with and show you how to apply them to a real world example application. This article contains a number of code examples and a reference to a a project in GitHub that you can use to as a reference.

Step 1 - Upgrade Spring to the latest version
The first thing you should do is simply upgrade the Spring version; update your Maven dependencies or replace the jar files. We will not change any code or configuration yet, so the upgrade to the latest Spring version doesn't really improve the situation. It does give the flexibility to use some newer technologies such as JPA 2 within Spring however, and that's what we need next. Upgrading Spring should be as easy as dropping in the new dependencies. Backwards compatibility in Spring is generally very good.

Step 2 - Replace old or out-dated frameworks within Spring
Most Spring applications are not just using Spring. Instead they use a whole bunch of frameworks and integrate them using Spring. A typical Spring application could look like figure 1.

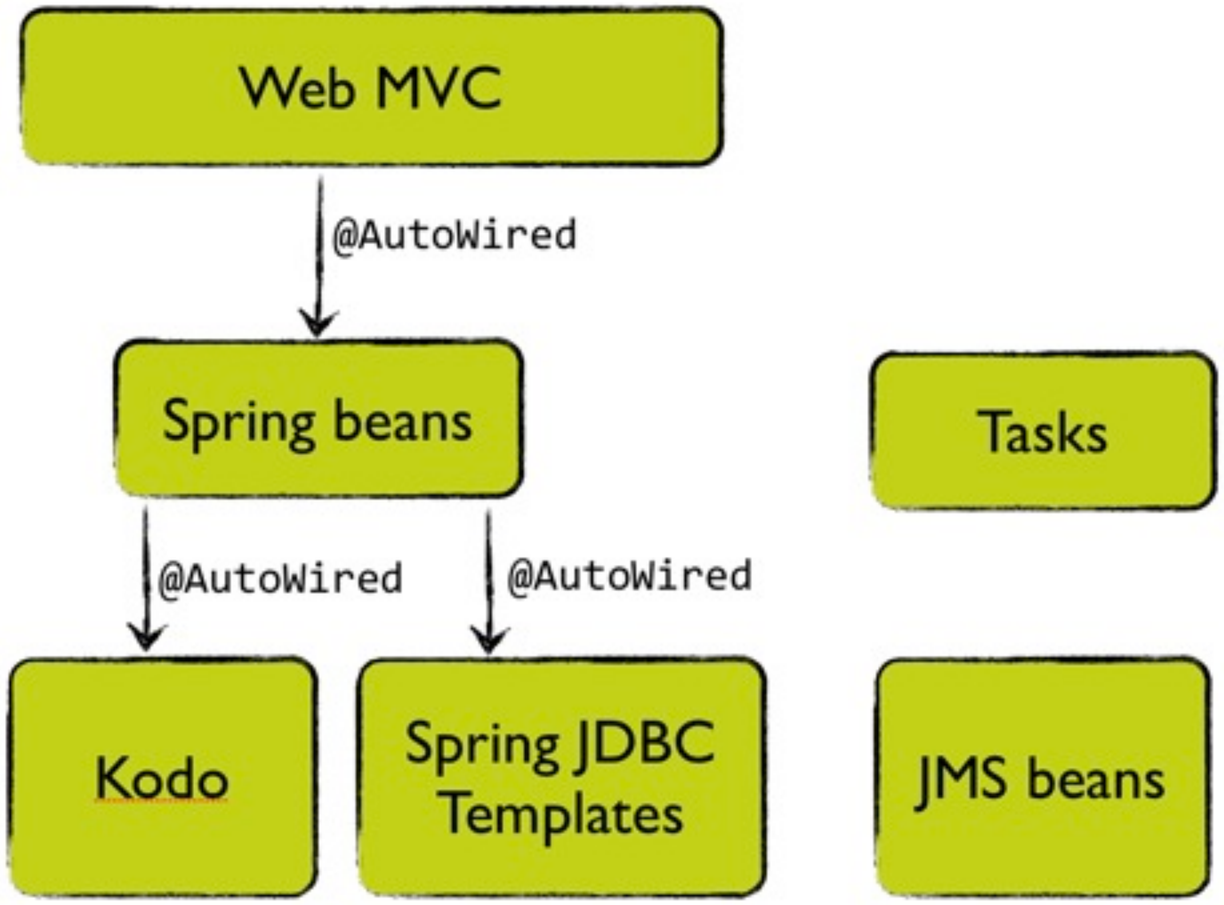


Figure 1: A typical Spring application

In this case there are two obvious parts of the application that needs to be updated:
1) The DAO framework (Kodo)
2) Spring Web MVC

Kodo, and many other old ORM frameworks, are not maintained any more and are now obsolete due to the introduction of JPA. It's also hard to find any developers for frameworks like this. Modern Spring applications also focus on using JPA and have good support for this. Spring Web MVC is replaced by an annotation based version of the framework. The old model is deprecated. Unfortunately it is not very trivial to migrate from the old model to the new model.
Because we need to take small steps to migrate a large code base to modern technology we start by replacing one of the outdated technologies within Spring. For example, you could replace Kodo by JPA 2 within Spring. Notice that we are using some Java EE technology now, but we don't have to start using a Java EE application server yet.

At some point you will run into places where you either want to start developing new functionality or start using Java EE technology that is not supported by Spring.

Step 3 - Run Spring and Java EE side-by-side
When new functionality is developed it should use the new technology stack, while still integrating with existing code. For example, we start writing JSF/CDI code while still using our Spring JDBC templates for data access. If we look at the deployment model of both Spring and Java EE there is a clear difference. If you compare figures 2 and 3, you see that Spring packages framework functionality within the application. Because of this you only need a Servlet container, but end up with huge WAR files. Java EE has all the framework functionality in the application server, so you don't need to package it in your WARs. There is really no good reason to use a Servlet container instead of a real application server now days. Startup times are the same, so why bother with the complexity of managing a large amount of dependencies in your project and packaging them with each build? There are 14 Java EE application servers at the moment of writing, both for free and commercially supported.

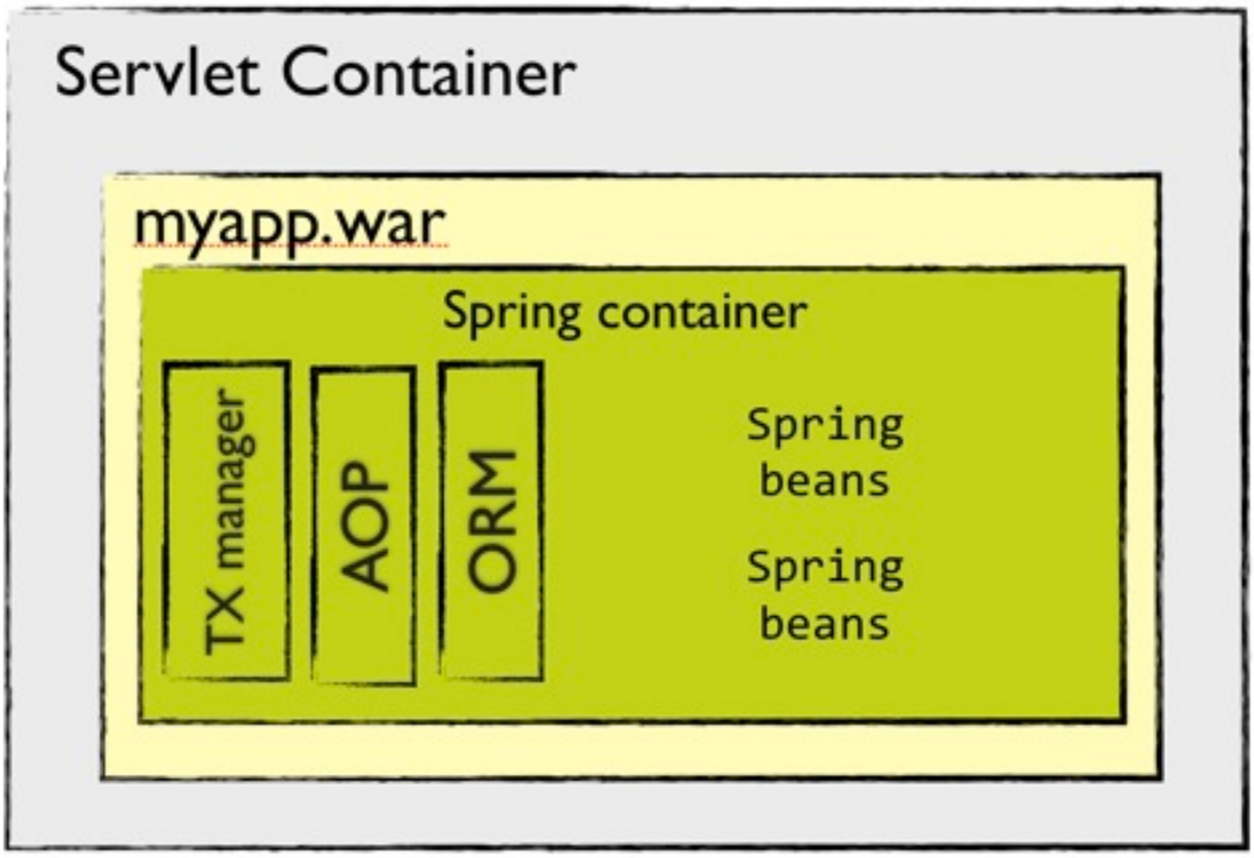


Figure 2: packaging of a Spring application.

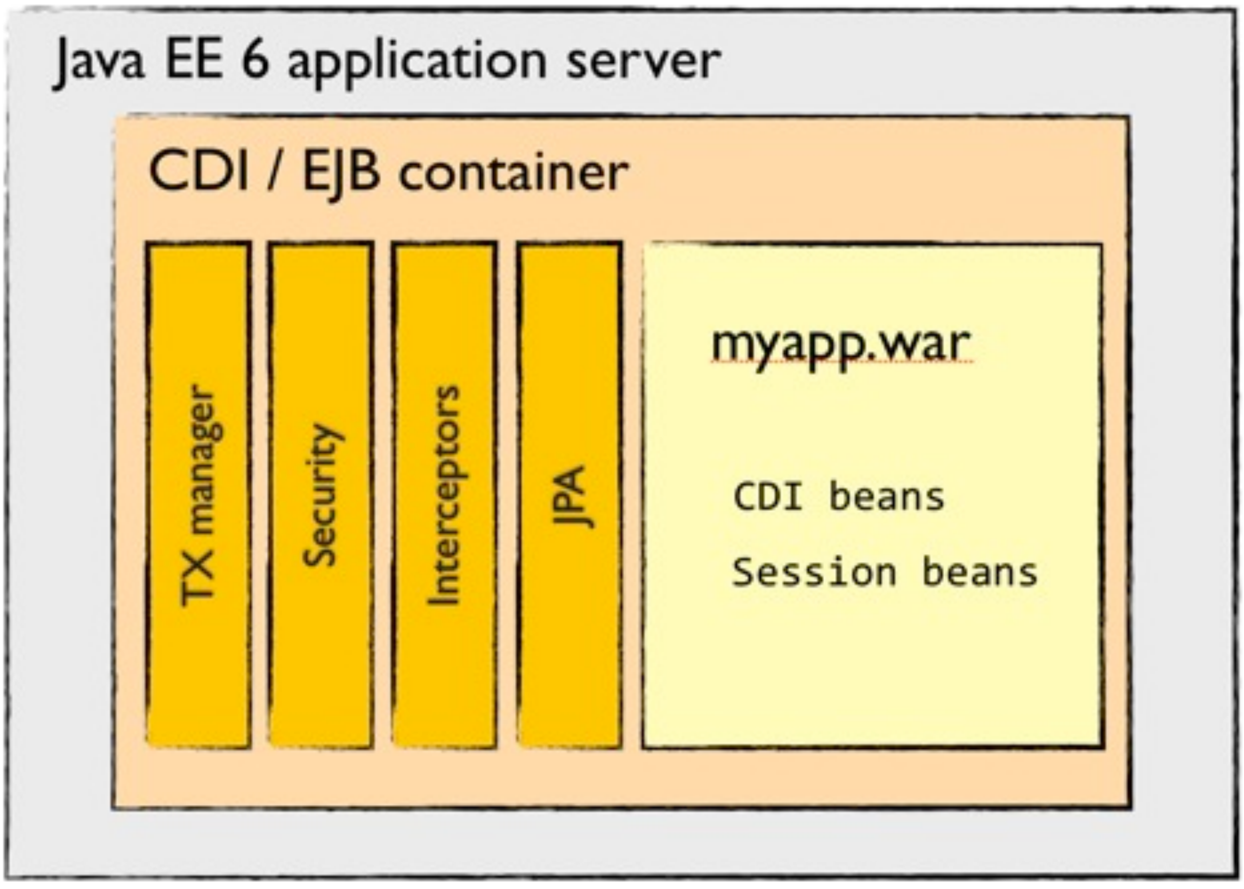
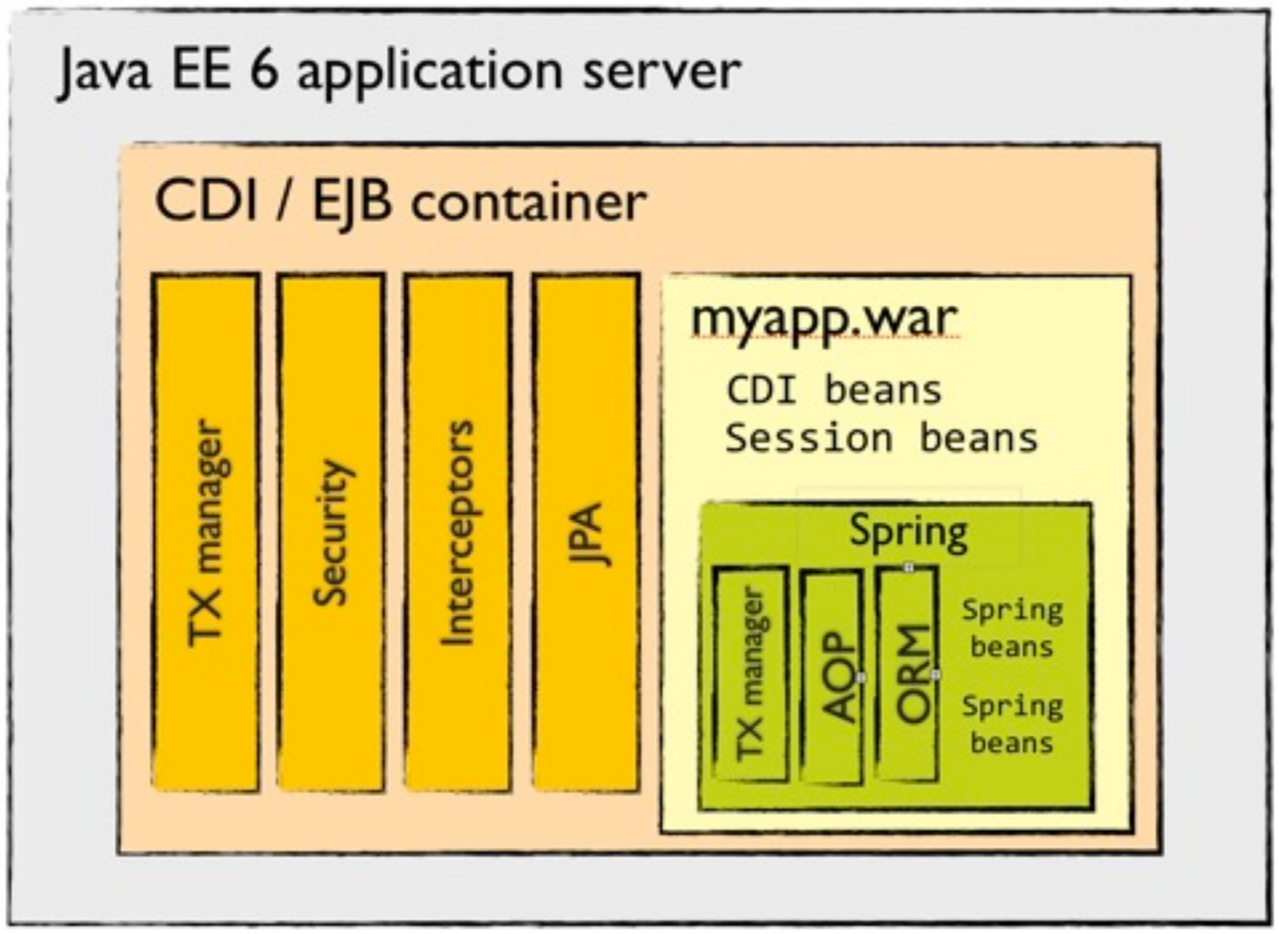


Figure 3: packaging of a Java EE application.

What we do to start using Java EE without loosing old code, is running the two container models side-by-side. To be more precise, the Spring container will run within the Java EE container as displayed in figure 4. We will use CDI to integrate both containers. The Spring container starts up as usual, but CDI will then make all Spring beans available to Java EE code using CDI injects. In the new code we can't even see if injected beans are created by Java EE or Spring, so further migration will not force us to change any new code. How CDI and Spring are integrated is discussed on a more technical level further in the article.



Step 4 - Replace Spring entirely
We have been replacing Spring code for a while now, and there is not much left. Most work was in the migration of outdated frameworks. It will be a relatively minor task to remove the remaining Spring code too, because the programming models are now quite similar. There might still be challenges such as the use of Spring JDBC Templates. We will discuss this in depth in the following articles in this series.

Step 5 - Remove Spring
Now that there is no Spring code any more, we can also remove all it's dependencies. For Java EE we don't need to package any libraries with our application, so we can go from a whole list of dependencies to just one: The Java EE API.

Example application

The example application is a modified version of the Spring Pet Clinic. The original pet clinic has multiple implementations of the data access layer to explain the integration with different ORM solutions. Because this is not relevant for this article series we stripped out the Hibernate, Toplink and JDBC implementations which leaves us with JPA. The application is already implemented using the latest Spring version. The user interface is built using moderns Spring MVC. Because the application already uses the latest Spring version and is already using JPA you could say this application is halfway of migration step two.

We will migrate this example application step-by-step to a full Java EE 6 application using the following approach:

1. Upgrade the web UI to JSF
2. Replace the Data Access Layer to EJB / JPA
3. Migrate AOP to CDI interceptors
4. Migrate JMX
5. Migrate JDBC Templates
6. Remove Spring
7. Integration testing

This article is the first of a series and only discusses the general approach and migrate the web layer. In the next articles we will migrate the rest of the code. This article focusses on step two and three in the migration approach discussed above. In the final article in the series we will have replaced all Spring components and can finish by removing Spring entirely.

When migrating your own applications you don't have to stick to this approach exactly; this strongly depends on the application. It is also not required to focus on a specific technology (e.g. data access) at any point, and integration testing should be something you work on constantly.

The example application is available at github: https://github.com/paulbakker/petclinic. Each step can be traced back as a change set. We highly recommend cloning the repository and keeping the code next to the article.

Migrating the web UI

The goal of this step is to replace Spring Web MVC by JSF and CDI while keeping the functionality more or less the same. This might be a realistic scenario in some projects if there is a lot of unmaintainable legacy code. When projects grow and teams are too large this often happens, no matter what technology is used. The other scenario might be that new functionality must be added to an existing application, but the current technology stack is outdated. Unfortunately this is already the situation if the old-style Spring Web MVC (pre Spring 2.5) is used.

JSF is probably the part of Java EE that receives most critics. It's fair to say that JSF is not perfect for each situation, but to our opinion; no web framework is. It depends a lot if you are building an intranet application for 50 users, or a high profile news site that has a hundred thousand hits each second. JSF makes it easy to build the first type of application, but impossible to do the second. Whatever application you are building, choose the web technology fit for the situation. In the final article of the series we will discuss an approach where the UI is entirely client-side. For this article we will focus on JSF however, because it works well for this type of application.

To show some different interesting capabilities of Java EE 6 we changed the UI a little at some places. After this step we should have removed the following:

- The web and validation packages
- All configuration in web.xml
- petclinic-servlet.xml
- All JSP files

Step 1 - Install JSF
First we need to add JSF and CDI to our project. To work with Java EE we need only one new POM dependency that contains all the Java EE 6 APIs. An application server includes the implementations of these APIs so the scope should be “provided” (not included in the WAR file).

```
<dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-6.0</artifactId>
      <version>1.0.0.Final</version>
      <type>pom</type>
      <scope>provided</scope>
</dependency>

<repository>
     <id>JBOSS_NEXUS</id>
     <url>http://repository.jboss.org/nexus/content/groups/public</url>
   </repository>
</repositories>
```

We also need an empty *faces-config.xml* file to turn on JSF in the WEB-INF folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config 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-facesconfig_2_0.xsd"
   version="2.0">
</faces-config>
```

To turn on CDI we need an empty file *beans.xml* in the WEB-INF folder. As a last step we also need to upgrade the *web.xml* to the latest version so that Servlet 3.0 becomes active and JSF can register itself automatically. Just change the versions in the namespaces to do so:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" version="3.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

These steps can be simplified by using JBoss Forge. This doesn’t create any lock-in on Forge; it is designed to work on standard Maven projects.

Start Forge and cd into your project and simply type:

```
$ setup faces

***SUCCESS*** Installed [forge.spec.servlet] successfully.
***SUCCESS*** Installed [forge.spec.jsf] successfully.
 ? Do you also want to install CDI? [Y/n] Y

***SUCCESS*** Installed [forge.spec.cdi] successfully.
 ? Do you also want to install the Faces servlet and mapping? [y/N] n

***SUCCESS*** JavaServer Faces is installed.
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/webapp
Wrote /Users/paul/tmp/forge-projects/jsftest/pom.xml
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/webapp/WEB-INF/
web.xml
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/webapp/WEB-INF/
faces-config.xml
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/webapp/WEB-INF/
beans.xml
```

Forge will add the Maven dependency and create the required files.

Step 2 - JSF page template
In our new pages we want to keep the same look & feel like before. While you need includes for this in JSP we have a much cleaner template mechanism in JSF. A template is just a JSF page that contains defines page specific blocks using “ui:define”.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head>
    <title>Simple JSF Facelets page</title>
    <h:outputStylesheet library="styles" name="petclinic.css"/>
</h:head>

<h:body>
    <div id="main">
        <ui:insert name="content"/>

        <table class="footer">
            <tr>
                <td><a href="/">Home</a>
                </td>
                <td align="right">
                    <h:graphicImage library="images" name="springsource-
logo.png"/>
                </td>
            </tr>
        </table>
    </div>
</h:body>
</html>
```

Something to notice is the use of a resource library in the outputStylesheet and graphicImage. It has always been cumbersome to correctly reference images and stylesheets without getting problems with the relative location of files. When using a resource library you simply put all files into a directory under the “resources” directory. The name of the directory will be the name of the library. JSF will create the correct urls when rendering pages. You can use the same mechanism in CSS files:

```
background-image: url(#{resource['images:banner-graphic.png']});
```

To use this template you define a composition in a xhtml page.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://
java.sun.com/jsf/core"
                xmlns:ui="http://java.sun.com/jsf/facelets"
template="template.xhtml">
    <ui:define name="content">
       Content within a template.
    </ui:define>
</ui:composition>
```

Step 3 - Integrate Spring and CDI component models
We are now ready to start implementing use-cases. Let’s start with the simple vets overview page. To implement this page we need a new xhtml file and a new class that feeds data into the page. To retrieve the list of vets we want to re-use our existing DAO implementation, which of course is a Spring bean.

What we want up with is the following:

```
@Named
@RequestScoped
public class VetsController {

    @Inject
    private Clinic clinic;

    public Collection<Vet> getVets() {
       return clinic.getVets();
    }
}
```

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://
java.sun.com/jsf/core"
                xmlns:ui="http://java.sun.com/jsf/facelets"
template="template.xhtml">
    <ui:define name="content">
        <h:dataTable value="#{vetsController.vets}" var="vet">
            <h:column>
                <f:facet name="header">Name</f:facet>
                #{vet.firstName} #{vet.lastName}
            </h:column>

            <h:column>
                <f:facet name="header">Specialities</f:facet>
                <ui:repeat value="#{vet.specialties}" var="speciality">
                    #{speciality.name}
                </ui:repeat>
                <h:outputText rendered="#{vet.nrOfSpecialties == 0}"
value="none"/>
            </h:column>
        </h:dataTable>
    </ui:define>
</ui:composition>
```

The xhtml page uses the VetsController from Expression Language and the VetsController injects the Clinic DAO. There we have a problem however; if CDI creates an instance of Clinic like it would normally it will not work correctly. The Clinic implementation (EntityManagerClinic) requires all kind of Spring configuration such as a data source and entity manager. Of course we could migrate this code to Java EE 6, but it would be unrealistic in a large application to do so through all layers. For now we want the Clinic to just work as it used to do, and use this code from our new CDI bean. Technically this requires us to integrate the Spring and CDI component models.

For this to work we need a CDI extension that retrieves bean instances from the Spring application context and publishes those instance to CDI. You could easily write such an extension yourself, the presentation contains an example of this: https://github.com/paulbakker/migrating-spring-to-javaee/blob/master/src/main/java/demo/framework/StartupListener.java

You don't have to implement this yourself however, the Seam Spring module already does this for you. Seam 3 is a set of CDI extensions that adds functionality to Java EE 6 in a standard way. Seam 3 is modular, you only depend on the module that you actually use, so the lock-in you create by using such a module is very low. In the example application we use Seam Spring to create the integration between CDI and Spring.

There is one additional difficulty in the way the PetClinic works. Because the PetClinic is a web application it creates a Spring WebApplicationContext at application startup time. To avoid creating bean instances twice we want to use this WebApplicationContext to retrieve bean instances to publish to CDI. The CDI bean discovery events are fired before the WebApplicationContext is created however which makes it impossible to automatically publish all Spring beans as CDI beans at application startup. There are different approaches to deal with this, but the most straightforward approach is to use CDI producers to publish Spring beans. You do need to write a producer for each Spring bean that you want to use in CDI, but this also keeps the process very explicit.

When using Seam Spring you can write producers as follows:

```
public class WebApplicationContextProducer {
    @Produces
    @SpringContext
    @Configuration(locations = "classpath:applicationContext-jpa.xml")
    ApplicationContext context;

    @Produces
    @SpringBean
    Clinic clinic;

    @Produces
    @SpringBean
    ClinicReporting clinicReporting;
}
```

Of course we also need to add Seam Spring to our POM file.

```
<dependency>
      <groupId>org.jboss.seam.spring</groupId>
      <artifactId>seam-spring-core</artifactId>
      <version>3.1.0-SNAPSHOT</version>
</dependency>

<dependency>
      <groupId>org.jboss.seam.solder</groupId>
      <artifactId>seam-solder</artifactId>
      <version>3.0.0.Final</version>
</dependency>
```

To prevent CDI from creating instances of the Clinic and ClinicReporting itself we need to “veto” them. This can be done using Seam Solder by creating a *page-info.java* file in the package that contains the classes.

```
@Veto
package org.springframework.samples.petclinic.dao.jpa;
import org.jboss.seam.solder.core.Veto;
```

That’s all we need to do to integrate CDI and Spring! The @Inject Clinic in our CDI bean now works.

Step 4 - Migrating search owners
The search owners use case is very easy to migrate, this can be done using a plain JSF data table and a request scoped managed bean. To make things a bit more interesting we

improved the searching process with some AJAX; whenever a user starts typing a name the list should be filtered.

```
 <h:inputText value="#{ownerController.lastname}" id="lastnameInput">
     <f:ajax event="keyup" render="owners" execute="@this"/>
</h:inputText>

<h:dataTable value="#{ownerController.owners}" var="owner" id="owners">
....
</h:dataTalbe>
```

Step 5 - Migrating owner details
Owner details is a more interesting use case. After selecting an owner the owner details are shown, including the owner's pet. From there, a user can edit the owner details, and add/edit pets. In the example application this is implemented in a stateless way by passing around the owner id with each request. There is nothing wrong with this approach, but can be implemented simpler when using a stateful programming model which is possible with Java EE 6. A stateful approach is not always a good choice because you loose bookmarkability of pages because the pages need state set by other pages. Bookmarkability in this example is not an issue however so we can have the benefit of a simpler programming model.

When working with state in web applications people would generally use the http session to store state. Java EE 6 has something better though; Conversations. A conversation can be described as a sub-session; isolated data within a session identified by an id with their own time-out. Conversations prevents problems when using multiple browser tabs because of the explicit id, and reduce memory leaks because there is a clear start and end point for each conversation.

The basic structure will be that a conversation is started whenever an owner is selected. The conversations ends when the user returns to the find owner screen. Within the conversation we can simply keep variables in memory without passing around ids or re-loading data from the database.

In the code below you see how a single class is used to handle all owner related functionality.

```
@Named
@ConversationScoped
public class OwnerDetailsController implements Serializable {
    @Inject
    transient Clinic clinic;

    @Inject
    Conversation conversation;

    private boolean edit = false;
    private Owner owner;
    private Pet pet;
    public String showOwner(Owner owner) {
        conversation.begin();
        this.owner = owner;

        return "showowner.xhtml";
    }

    public String newPet() {
        pet = new Pet();
        owner.addPet(pet);
        return "editpet.xhtml";
    }

    public String editPet(Pet pet) {
        this.pet = pet;
        return "editpet.xhtml";
    }

    public Collection<PetType> getPetTypes() {
        return clinic.getPetTypes();
    }

    public String savePet() {
        clinic.storePet(pet);
        return "showowner.xhtml?faces-redirect=true";
    }

    public String toOverview() {
        conversation.end();
        return "findowner.xhtml?faces-redirect=true";
    }

    public Owner getOwner() {
        return owner;
    }

   //Getters and setters
}
```

The owner field contains the currently selected owner and the pet field contains the currently selected pet (or a newly created pet). In the xhtml pages we can now simple reference those fields:

```
<h:inputText id="nameInput" value="#{ownerDetailsController.pet.name}"/>
```

The conversation is started by clicking on an owner in the find owner screen:

```
<h:commandLink action="#{ownerDetailsController.showOwner(owner)}"
value="#{owner.firstName} #{owner.lastName}"/>
```

In the edit pet screen there is a drop down field to select the type of pet which is set as a PetType on the Pet entity. A select tag in HTML sends a single value back to the server. The problem here is that PetType is a complex type, and JSF can't know how to identify a PetType as a single string value. Spring obviously has the same problem and uses Editors to convert from string to complex type and vice versa. JSF uses Converters which do basically the same.

A Converter implements two methods; one to convert a complex type to a String and one for conversion the other way around.

```
@FacesConverter("petTypeConverter")
public class PetTypeConverter implements Converter {
    @Inject
    Clinic clinic;

    public Object getAsObject(FacesContext facesContext, UIComponent
uiComponent, String s) {
        for (PetType type : clinic.getPetTypes()) {
            if(type.getName().equals(s)) {
                return type;
            }
        }
       throw new ConverterException("Couldn't find pet type: " + s);
    }

    public String getAsString(FacesContext facesContext, UIComponent
uiComponent, Object o) {
        PetType petType = (PetType) o;
        return petType.getName();
    }
}
```

The converter can then be used in the select box.

```
<h:selectOneMenu value="#{ownerDetailsController.pet.type}"
id="typeInput">
    <f:converter converterId="petTypeConverter"/>
    <f:selectItems value="#{ownerDetailsController.petTypes}"/>
</h:selectOneMenu>
```

Although this code looks very straight forward there is a problem; it doesn't work. Due to a bug in the JSF specification, dependency injection doesn't work in Converters. We do need the Clinic to do the conversion however. Of course there are all sorts of work arounds possible, but dependency injection makes most sense. To fix this issue we will use the Seam Faces CDI extension. By simply having Seam Faces on the classpath we can use injection in Converters again. Besides fixing this issue there are a lot of other goodies in Seam Faces which are not discussed here, but it's worth to have a better look at.

```
 <dependency>
      <groupId>org.jboss.seam.faces</groupId>
      <artifactId>seam-faces</artifactId>
      <version>3.0.2.Final</version>
</dependency>
```

Step 6 - Validation
Although Spring 3.0 has support for the Bean Validation specification the example application does not make use of this. Validation is implemented using implementations of Spring's Validator interface. Bean Validation is a more DRY mechanism because both JPA and JSF can use the same validation rules. Migrating the old validation rules to Bean Validation means adding annotations on our entity classes. The Bean Validation specification already describes a whole bunch of annotations and makes it easy to define new annotations yourself. However, you can get some more out-of-the-box rules by using Hibernate Validator. This library includes annotations such as @Email, @NotEmpty etc. Add Hibernate Validator to the Maven dependencies.

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.2.0.Final</version>
      <scope>provided</scope>
</dependency>
```

Alternatively you can use JBoss Forge to do so:

```
$ setup validation
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
***SUCCESS*** Installed [forge.spec.validation] successfully.
Which version of hibernate-validator would you like to use?

  1 - [org.hibernate:hibernate-validator:::4.1.0.Final]
  2 - [org.hibernate:hibernate-validator:::4.2.0-SNAPSHOT]
  3 - [org.hibernate:hibernate-validator:::4.2.0.Beta1]
  4 - [org.hibernate:hibernate-validator:::4.2.0.Beta2]
  5 - [org.hibernate:hibernate-validator:::4.2.0.CR1]
  6 - [org.hibernate:hibernate-validator:::4.2.0.Final]
  7 - [org.hibernate:hibernate-validator:::4.3.0-SNAPSHOT]*

 ? Choose an option by typing the number of the selection [*-default]
[0] 6
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/resources/META-INF/validation.xml
Wrote /Users/paul/tmp/forge-projects/jsftest/pom.xml
```

Now we can add constraints to domain classes by annotating it's fields.

```
public class Owner extends Person {
   @NotEmpty
   private String address;

   @NotEmpty
   private String city;

   @NotEmpty @Pattern(regexp = "^[0-9]+$", message = "Only digits are 
allowed")
   private String telephone;
```

JSF automatically triggers up these constraints when submitting data. You do need to display error messages however using the “h:message” tag.

```
Last Name: <h:message for="lastnameInput"/><br/>
<h:inputText value="#{newOwnerController.owner.lastName}"
id="lastnameInput"/>
```

Step 7 - RESTful web services
The list of vets can also be retrieved as XML or JSON. While RESTful web services are implemented in your web controllers when using Spring we have a separate specification for this in Java EE 6 (JAX-RS). Besides the more clear separation between web controllers and web services in Java EE 6 the programming model is very similar to Spring. First of all we need to enable JAX-RS by extending the Application class. We also annotate this class to define the root url for our web services. The class doesn’t have to be registered anywhere.

```
@ApplicationPath("rest")
public class RestApplication extends Application {
}
```

Now we have to implement our web service method.

```
@Path("vets")
@Stateless
public class VetsResource {
    @Inject
    Clinic clinic;

    @Produces({"application/xml", "application/json"})
    @GET
    public Collection<Vet> listVets() {
        return clinic.getVets();
    }
}
```

Similar to Spring we can return our own data types. We annotate our classes with JAXB annotations to specify the mapping between Java and XML/JSON.

```
@XmlRootElement
public class Vet extends Person {
  ....
}
```

Step 8 - URL rewriting
One of the really nice things about Spring Web MVC is that application URLs are by default “pretty”. “/vets” is obviously nicer then “/faces/vets.xhtml”. JSF does not support this out of the box, but with the help of PrettyFaces we can easily add this. First of all we need the PrettyFaces dependency in Maven.

```
<dependency>
    <groupId>com.ocpsoft</groupId>
    <artifactId>prettyfaces-jsf2</artifactId>
    <version>3.3.2</version>
</dependency>
```

PrettyFaces doesn’t have to be registered, the only thing left to do is configure our application URLs. Create a file “WEB-INF/pretty-config.xml”. In this file you configure rules for rewriting pretty URLs to JSF URLs. Alternatively you can use JBoss Forge to create the dependency and configuration file.

```
$ setup prettyfaces

Install PrettyFaces for which technology?

  1 - [JSF 1.1 and Servlet <= 2.3]
  2 - [JSF 1.2 and Servlet >= 2.4]
  3 - [JSF 2.0 and Servlet >= 2.5]
  4 - [Java EE 6 and Servlet >= 3.0]*

 ? Choose an option by typing the number of the selection [*-default]
[0] 4
Install which version?

  ....
  14 - [com.ocpsoft:prettyfaces-jsf2:::3.3.2]
  15 - [com.ocpsoft:prettyfaces-jsf2:::3.3.3-SNAPSHOT]
  16 - [com.ocpsoft:prettyfaces-jsf2:::4.0.0-SNAPSHOT]*

 ? Choose an option by typing the number of the selection [*-default]
[0] 15
Warning:  The encoding 'UTF-8' is not supported by the Java runtime.
Wrote /Users/paul/tmp/forge-projects/jsftest/src/main/webapp/WEB-INF/
pretty-config.xml
```

For the example application the pretty-config file looks as follows.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<pretty-config xmlns="http://ocpsoft.com/prettyfaces/3.3.3-SNAPSHOT"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://ocpsoft.com/prettyfaces/3.3.3-
SNAPSHOT http://ocpsoft.com/xml/ns/prettyfaces/ocpsoft-pretty-
faces-3.3.3-SNAPSHOT.xsd">
    <url-mapping id="home">
        <pattern value="/"/>
        <view-id value="/faces/welcome.xhtml"/>
    </url-mapping>

    <url-mapping id="vets">
        <pattern value="/vets"/>
        <view-id value="/faces/vets.xhtml"/>
    </url-mapping>

    <url-mapping id="searchowners">
        <pattern value="/owners"/>
        <view-id value="/faces/findowner.xhtml"/>
    </url-mapping>

    <url-mapping id="reports">
        <pattern value="/reports"/>
        <view-id value="/faces/reports.xhtml"/>
    </url-mapping>
</pretty-config>
```

Step 9 - Remove Spring Web MVC
Now that we successfully replaced all the web related code with Java EE 6 code we can remove a large part of the Spring code and configuration.

- Remove the web and validation packages
- Remove all JSP files
- Remove petclinic-servlet.xml
- Remove everything from web.xml

The code that we have thrown aways is of course replaced by new, to our opinion better, code. All the configuration is replaced by nothing but void however. One of the big advantages of Java EE 6 is that almost no configuration is required. Spring might be more flexible in the sense that it integrates with every possible framework, but this comes at the cost of a lot of required configuration. It's not so much the writing of the configuration, this is easy, but understanding what an application does exactly is much more difficult.

What's next?
So far we have only focussed on the web layer. In the next article of this series we will work on migrating the data access layer away from Spring. We will also introduce CDI as a migration solution for Spring Interceptors and AOP. Finally, we will show how to deal with JMX.