Spring by Example

David Winterfeldt

Version 1.2.2

Table of Contents

Preface

Susan Kerschbaumer
2009

> Winter slumbering in the open air,
> Wears on his smiling face a dream of Spring!
>
> —Samuel Taylor Coleridge

The Spring framework is aptly named – in many ways Spring ushers in a new season for Java programming, and brings a sense of renewal to Java's roots in portability and object oriented concepts. Spring evolved from the needs of the J2EE community at a particular moment in time. In the process it has become a much broader tool in the drive toward more modular, portable, and now, aspect oriented, programming. If you are a Java programmer and have access to a JVM, you can leverage any part of Spring to begin to build applications that are easier to code, easier to test, and easier to manage. This book will show you how.

1 Spring: Evolution Over Intelligent Design

One of the reasons that Spring has become so popular and so usable is the simple fact that it is evolving. Yes, there was some intelligent design involved at its inception (thanks to Spring's creator Rod Johnson). But what started out as a good idea to help address some very specific problems for J2EE developers became a conversation, a dialog with the developer community. This conversation has helped to shape the direction of this thing we call Spring.

Programmers are linguists. We make up words, assemble grammars, and use them to express concepts both abstract and concrete. The evolution of programming languages and technologies, Spring included, has been inspired and conducted by the people that use them. And just as in every language, usage dictates meaning.

From procedural, to object oriented, and to the newer linguistics of aspect oriented programming, the way we think about coding is eternally evolving alongside the languages and technologies we use. Spring evolved from within a context of needed change in Java programming environments. Not only has Spring popularized some core best practices in Java development, it is now influencing the evolution of Java itself.

2 A Little History

The Spring framework began as one programmer's need to solve a set of problems that Sun had tried to solve through edict in the Enterprise JavaBeans (EJB) Specification. At the time, companies were putting a lot of money and effort into developing web applications. The lack of established industry standards made management nervous, and development, haphazard. The J2EE specification promised scalability, security, high availability. Enterprise JavaBeans (EJB), as part of the J2EE suite of specifications from Sun, were intended to be as reusable and portable as their non-enterprise counterparts, plain JavaBeans.

When the final draft of the first EJB specification was released in 1998, Java web programmers looked to it as the golden child of enterprise web development.

But what they found was that the many interfaces and configuration files required to create an EJB were awkward, tedious, and prone to error. The EJBs' marriage to the container made unit testing close to impossible. And applications became heavy in the bulk of extra container features.

Since then, certain IDEs and EJB 3.0 have addressed some of these issues, but to many application developers, it's too little, too late.

The reality is that while Java and your business needs are here to stay, everything in between is likely to change, and change often. This takes us back to Java's early notion of portability (“compile once, run everywhere”) - portability is important because in today's programming environment, hardware, operating systems, IDEs, and containers, rapidly change.

Spring provides a POJO (Plain Old Java Object) based configuration environment, a container to manage the instantiation and lifecycle of your POJO components, and a framework to help you put into place some established best practices for your applications. The idea behind Spring is that your code should be well-factored, and components, kept pristine. Your components should run with or without a container, and be testable with minimal to no intrusion from outside classes. In essence, your components should have a life outside of the framework.

And as complete entities unto the themselves, these truly modular components should have a loose affiliation with other components but should not be bogged down in these dependencies. Component factoring in this way has become central to object oriented programming.

While helping you make your code more truly object oriented, Spring introduces you to two new grammars in the world of application development – the Inversion of Control (IoC) container and Aspect Oriented Programming (AOP). Both of these concepts arose naturally from the needs of developers to improve maintainability, productivity, portability, and design.

Spring also provides practical and modular solutions to JDBC connectivity and transaction management. These solutions offer a superset of the traditional JDBC and JTA offered in J2EE containers. The modular design of these aspects of Spring also allow you to leverage these pieces outside of the context of Spring's container.

And this is the beauty of Spring. Spring provides all of these features and tools, and more, in a non-intrusive framework. Spring is a container, but it is also a set of utilities which can be used outside of the container. And in keeping with the idea of portability, the authors of Spring have taken into account where your applications might be today (tangled up in EJB interfaces, or RMI services, for example), and have provided means to port your existing code as simply as possible.

3 Goals of This Book

Spring was not developed in a boardroom and as a result it has a vibrant and sometimes confusing context. This book will first introduce you to the core concepts behind Spring and how Spring will make your life easier (and more interesting!). Spring is built around a few key patterns and some relatively new concepts in Java programming. In this book after the introduction, each article will be based on a working example available from the Spring by Example Subversion repository [http://svn.springbyexample.org/].

While Spring evolved out of the J2EE realm, in keeping with the idea that Java components should not be tied to a particular architecture, we only require that you have some knowledge of Java and XML in order to understand and make use of most of this book. Although more advanced examples may require some additional knowledge to understand the example fully, but there are references at the end of each example to read more about the subject covered.

4 A Note about Format

New terms appear in ***bold italics*** the first time they are defined or described. ***Buzzwords***, or keywords that have already been defined but that are important to a concept being explained may then appear in *italics*. Direct references to coding examples in the text of the book (methods, classnames, etc), and the coding examples themselves, are displayed in `courier font`, otherwise known as `That Typewriter Typeface` (TTT). Incidentally, there are a lot of acronyms in this book and some of them may be new to you. Generally if the acronym is the most common way to refer to a particular term, we will use the term, followed by the acronym, the first time it is mentioned in a section. The acronym will be used for the remainder of the section.

Part I. Spring Introduction

Basic introduction to Inversion of Control (IoC), Dependency Injection (DI), and the Spring Framework.

Spring In Context: Core Concepts

Susan Kerschbaumer

David Winterfeldt

2009

> ...The architecture that actually predominates in practice has yet to be discussed: the BIG BALL OF MUD...A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailingwire, spaghetti code jungle...Why is this architecture so popular?...Can we avoid this? Should we? How can we make such systems better?
>
> — **Big Ball of Mud** , by Brian Foote and Joseph Yoder

Everyone knows the Big Ball of Mud. You've either written it (accidentally of course), you've had to read it, or you've had to refactor it. The Big Ball of Mud is disorganized, unwieldy code -- it blossomed in the late 1990's when the dotcom boom was in full force and there was great demand for fast application development, technology personnel was constantly in flux, and core technologies were changing regularly in order to accommodate issues of scalability.

Today the Big Ball of Mud behaves more like *The Blob*[1], growing larger as it consumes more application functionality, and threatening to swallow up good coding practices everywhere. Disorganized and poorly factored applications can be disastrous -- wasteful, expensive, and miserable to work with. Sometimes The Big Ball of Mud is so unwieldy it can't be worked with at all.

Fortunately, as web applications began to standardize on J2EE platforms (now known, and from here on referred to, as Java EE), design patterns for Java EE began to take shape (http://java.sun.com/blueprints/patterns). Frameworks such as Struts and Avalon gave more formal structure to web applications, and best practices began to emerge from, and inform, these frameworks. Spring's framework expanded on the concepts introduced in Struts and Avalon, adding its own twist in the form of its extremely lightweight Inversion of Control (IoC) container. Spring is different from other frameworks in that it can be used with virtually any Java component, including POJOs (Plain Old Java Objects), and can be leveraged in small pieces.

This chapter is about the **Big Picture**. We will explore the core concepts behind Spring and what it brings to Java EE Object Oriented methodologies. We'll also briefly summarize some of the other features of Spring that exist outside of the IoC container and illustrate how these features fit together, and tease apart.

What makes a framework, work?

A framework is essentially a skeleton, a structure around which the fleshing out of your application occurs. Frameworks tend to be built around a design pattern and consist of frozen spots – structural components which are unmodified by the developer -- and hot spots, the pieces that an application developer contributes [2]. In Spring, the

[1] A 1950's horror film featuring the big screen debut of Steve McQueen. The film stars an ever-growing, all-consuming amoeba-like monster that terrorizes a small Pennsylvania town.

hot spots are developer-contributed POJOs which are configured to run within the framework.

In this chapter we'll explore the context of Spring's most important component – the IoC container – and how Spring can lead you to better Object Oriented programming and eventually to new possibilities with Aspect Oriented Programming. The Spring framework can't make bad code good, but it can enforce good coding practices that make it easier for you and your fellow developers to cooperate in writing well-factored, reusable, readable, and manageable application components.

1. Spring and Inversion of Control

There is a lot of confusion about the definition of the IoC container – some equate it with a design pattern called **Dependency Injection** – but in reality IoC is much larger than dependency injection. The Spring IoC container enforces dependency injection in its various forms and employs a number of established design patterns to achieve this.

The main idea behind Inversion of Control as a concept is that component dependencies, lifecycle events, and configuration reside outside of the components themselves, and in the case of Spring, in the framework. This may sound a bit like giving up too much control, but in fact it can make your code more manageable, more testable, and more portable.

Before we discuss the Spring IoC container in detail, it's important to understand on the most basic level what the dependency injection pattern is and how it emerged in object oriented programming methodology. Spring's Inversion of Control framework is based on some best practice patterns – aspects of Spring IoC resemble and include the *Factory* and *Observer* patterns, and its most prominent feature is its implementation of a framework which enforces use of the dependency injection pattern.

Dependency Inversion: Precursor to Dependency Injection

The first reference to what would eventually become **Dependency Injection** appeared in 1994 in a paper by Robert C. Martin called “The Dependency Inversion Principle”.

In “The Dependency Inversion Principle” (or **DIP**), the author states the three defining factors of “bad code”:

1. It is hard to change because every change affects too many other parts of the system (Rigidity)
2. When you make a change, unexpected parts of the system break (Fragility)
3. It is hard to reuse in another application because it cannot be disentangled from the current application (Immobility) [3]

According to Martin, interdependency causes these coding problems (we'll call them **RFI** for Rigidity, Fragility, and Immobility). To fix RFI issues in your OO code, DIP has two basic rules:

1. High level modules should not depend upon low level modules, both should depend upon abstractions.

In other words, high level modules – which contain your business logic and all of the important meat of your

[2] Pree, W. (1994) *Meta Patterns* – a means for capturing the essentials of reusable object-oriented design. Springer-Verglag, proceedings of the ECOOP, Bologna, Italy: 150-162

[3] *The Dependency Inversion Principle*, Martin, Robert C., .c 1994

application – should not depend on lower level components. The reason for this is if these lower level components were to change, the changes might affect the higher level components as well. This is the defining concept behind dependency inversion, that the prevailing wisdom of having higher-level modules dependent on lower-level modules is in fact a bad idea.

2. Abstractions should not depend upon details, details should depend upon abstractions.

This is another way to say that before you begin coding to the abstraction – the interface or abstract class -- you should find the common behaviors in the code and work backwards. Your interface /abstraction should cater to the intersection between the needs of your business logic and the common behaviors of the lower level modules. You should also leave the details of how these behaviors are implemented to the implementation classes.

This simple example of a voting booth program shows a non-DIP compliant program.

```
package org.springbyexample.vote;

public class VotingBooth {

    VoteRecorder voteRecorder = new VoteRecorder();

    public void vote(Candidate candidate) {
        voteRecorder.record(candidate);
    }

    class VoteRecorder {
        Map hVotes = new HashMap();

        public void record(Candidate candidate) {
            int count = 0;

            if (!hVotes.containsKey(candidate)){
                hVotes.put(candidate, count);
            } else {
                count = hVotes.get(candidate);
            }

            count++; ❶

            hVotes.put(candidate, count);  ❷

        }
    }

}
```

❶ This increments the *count* variable.
❷ Add candidate and votes to *Map*.

In this example, the *VotingBooth* class is directly dependent on *VoteRecorder*, which has no abstractions and is the implementing class.

A dependency “inverted” version of this code might look a little different. First, we would define our `VoteRecorder` interface.

```
package org.springbyexample.vote;

public interface VoteRecorder {

    public void record(Candidate candidate) ;

}
```

And our implementing classes.

The `LocalVoteRecorder`, which implements the `VoteRecorder` interface:

```
package org.springbyexample.vote;

public class LocalVoteRecorder implements VoteRecorder {

    Map hVotes = new HashMap();

    public void record(Candidate candidate) {
        int count = 0;

        if (!hVotes.containsKey(candidate)){
            hVotes.put(candidate, count);
        } else {
            count = hVotes.get(candidate);
        }

        count++;

        hVotes.put(candidate, count);
    }

}
```

And the `VotingBooth` class:

```
package org.springbyexample.vote;

public class VotingBooth {

    VoteRecorder recorder = null;

    public void setVoteRecorder(VoteRecorder recorder) {
        this.recorder = recorder;
    }

    public void vote(Candidate candidate) {
        recorder.record(candidate);
    }

}
```

Now the `LocalVoteRecorder` class – the implementing class of the `VoteRecorder` interface -- is completely decoupled from the `VotingBooth` class. We have removed all hard-coded references to lower level classes. According to the rules of DIP, this is all we need to do in order to rid our code of **RFI**.

However, there is one problem with this implementation. We don't have a main method. We definitely need one in order to run our application, and somewhere in this main method we will need to instantiate the `LocalVoteRecorder`.

By instantiating the `LocalVoteRecorder` in our `main` method, we would break Rule #1 of Dependency Inversion. We have coded to the abstraction, we have integrated our changes, but our application would still have a dependency on a lower level class.

2. Dependency Injection To The Rescue

Dependency Injection takes the level of decoupling that began with the Dependency Inversion Principle one step further. Dependency injection has the concept of an **assembler** [4] – or what in Java is commonly referred to as a **Factory** -- that instantiates the objects required by an application and "injects" them into their dependent objects.

In the case of a dependency injection-informed framework such as Spring, components are coded to interfaces, just as in the DIP example above. But now the IoC container manages the instantiation, management, and class casting of the implemented objects so that the application doesn't have to. This removes any true dependencies on low-level implemented classes.

There are three types of Dependency Injection employed by IoC container providers.

Table 1. Dependency Injection Types

DI Type	**Description**
Constructor Injection	The constructor arguments are injected during instance instantiation.
Setter Injection	This is the most favored method of dependency injection in Spring. Dependencies are "set" in the objects through setter methods defined in a Spring configuration file.
Interface Injection	This is not implemented in Spring currently, but by Avalon. It's a different type of DI that involves mapping items to inject to specific interfaces.

Spring uses the concept of a `BeanFactory` as its assembler, and it is the `BeanFactory` that manages the JavaBeans you have configured to run within it. In the next section we will discuss Spring's IoC container and how it makes use of dependency injection patterns to make your code, well, RFI-free, and just better.

[4] Martin Fowler, *Inversion of Control Containers and The Dependency Injection Pattern*

3. Bean management through IoC

Through its factory, Spring's IoC container manages the objects that it is configured to instantiate. Spring's management of the container objects adds flexibility and control to your application, and provides a central place of configuration management for your Plain Old Java Objects.

For example, through Spring IoC you can configure the number of instances of the component – whether the component is a singleton or not – and at what point the component is created and destroyed in memory. In Spring, the initialization of a bean by the framework is exactly equivalent to using the `new` keyword to instantiate an object in Java code. Once the framework has instantiated the object, it manages the scope of the bean, based on its configuration.

Because the IoC container is managing the beans, JNDI lookups that are typical in Java EE containers are no longer required, leaving your code container-agnostic and easier to unit test both inside and outside of the framework. And while you are coding to interfaces as part of good OO practice, Spring allows you to manage what implementations are used by leveraging dependency injection, resulting in cleaner, decoupled components.

The IoC container can also be configured to receive instantiation and destruction callback events for a particular bean. Certain components such as a database connection pool obviously need to be initialized and destroyed when the application is shutdown. Instead of using your custom code, Spring can manage these lifecycle events.

4. Our Example In Spring IoC

So how would we configure our simple voting counter example for the Spring IoC container?

We can use our code exactly as is. All we need to do is inform Spring through an XML configuration file that the *recorder* bean is implemented by the `LocalVoteRecorder` class. We do this with the following line:

```
<bean id="recorder"
      class="com.springindepth.LocalVoteRecorder" />
```

Then we simply map the *recorder* bean to the `VotingBooth` bean by setter injection in **that** beans definition.

```
<bean id="votingBooth"
      class="com.springindepth.VotingBooth">
    <property name="voteRecorder" ref="recorder"/>
</bean>
```

Spring works its magic to handle class instantiation for you, so your application code never becomes aware of the implementing classes. Now with this configuration and the Spring framework, and through dependency injection, we have finally removed the low-level component dependencies and have achieved true dependency inversion!

A Practical Introduction to Inversion of Control

Susan Kerschbaumer

David Winterfeldt

2009

As we may have mentioned, the core of the Spring Framework is its **Inversion of Control** (Ioc) container. The IoC container manages java objects – from instantiation to destruction – through its `BeanFactory`. Java components that are instantiated by the IoC container are called beans, and the IoC container manages a bean's scope, lifecycle events, and any AOP features for which it has been configured and coded.

The IoC container enforces the *dependency injection* pattern for your components, leaving them loosely coupled and allowing you to code to abstractions. This chapter is a tutorial – in it we will go through the basic steps of creating a bean, configuring it for deployment in Spring, and then unit testing it.

1. Basic Bean Creation

A Spring bean in the IoC container can typically be any POJO (plain old java object). POJOs in this context are defined simply as reusable modular components – they are complete entities unto themselves and the IoC container will resolve any dependencies they may need. Creating a Spring bean is as simple as coding your POJO and adding a bean configuration element to the Spring XML configuration file or annotating the POJO, although XML based configuration will be covered first.

To start our tutorial, we'll use a simple POJO, a class called `Message` which does not have an explicit constructor, just a `getMessage()` and `setMessage(String message)` method. `Message` has a zero argument constructor and a default message value.

```
public class DefaultMessage {

    private String message = "Spring is fun.";

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Example 1 `DefaultMessage`

The *bean element* below indicates a bean of type `Message` – defined by the class attribute – with an id of 'message'. The instance of this bean will be registered in the container with this id.

An end tag for the beans element closes the document (we said it was simple!).

DefaultMessageTest-context.xml

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

    <bean id="message"
          class="org.springbyexample.di.xml.DefaultMessage" />

</beans>
```

When the container instantiates the *message* bean, it is equivalent to initializing an object in your code with '`new DefaultMessage()`'.

2. Basic Constructor Injection

Now that we have our POJO and a basic configuration for the message bean, we can introduce our first dependency injection example. Through the Spring beans XML file you can configure your bean to initialize with an argument for the constructor, and then assign the arguments. Spring essentially “injects” the argument into your bean. This is referred to as **constructor injection**.

The following example passes in the String message using a constructor. The class is the same as the one in Basic Bean Creation except the default message on the `message` variable has been cleared and is now set to `null`. A single parameter constructor has been added to set a message.

```
public class ConstructorMessage {

    private String message = null;

    /**
     * Constructor
     */
    public ConstructorMessage(String message) {
        this.message = message;
    }

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
```

```
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Example 2 `ConstructorMessage`

The configuration for this bean is exactly the same as in the previous example, but now we have a new element, the *constructor-arg*. The *constructor-arg* element injects a message into the bean using the *constructor-arg* element's value attribute.

ConstructorMessageTest-context.xml

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

    <bean id="message"
          class="org.springbyexample.di.xml.ConstructorMessage">
        <constructor-arg value="Spring is fun." />
    </bean>

</beans>
```

3. Basic Setter Injection

The Spring IoC container also supports **setter injection**, which is the preferred method of dependency injection in Spring. Setter injection uses the `set*` methods in a class file to garner property names that are configurable in the spring XML config.

From a configuration standpoint, setter injection is easier to read because the property name being set is assigned as an attribute to the bean, along with the value being injected.

To determine the property names, Spring follows the JavaBeans Specification (http://java.sun.com/products/javabeans/docs/spec.html).

In most cases Spring will lowercase the first letter after “set” in the method name and use the rest of the method name as-is for deducing the property name. So for example if there is a `setMessage()` method in your class, the property name you would use when setting that property on a bean in the XML config is 'message'. If there is a `setFirstName()` method in your class, the property name you would use when setting the value is 'firstName'.

In cases where the letters after “set” are all uppercase, Spring will leave the property name as uppercase. So if you have `setXML()` on a class, the property name would be 'XML'.

Because Spring uses the set method names to infer the property name, the naming of your set methods should follow the JavaBeans Spec, or at least be consistent within the confines of your application. See the example below for basic setter injection on our Message class.

```
public class SetterMessage {

    private String message = null;

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Example 3 `SetterMessage`

The *property* element is used to define the setter injection:

SetterMessageTest-context.xml

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

    <bean id="message"
          class="org.springbyexample.di.xml.SetterMessage">
        <property name="message" value="Spring is fun." />
    </bean>

</beans>
```

4. Reference Injection

So far we have only injected constructor and property values with static values, which is useful if you want to eliminate configuration files. Values can also be injected *by reference* -- one bean definition can be injected into another. To do this, you use the *constructor-arg* or *property*'s *ref* attribute instead of the *value* attribute. The *ref* attribute then refers to another bean definition's id.

In the following example, the first bean definition is a `java.lang.String` with the id *springMessage*. It is

injected into the second bean definition by reference using the *property* element's *ref* attribute.

ReferenceSetterMessageTest-context.xml

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

    <bean id="springMessage"
          class="java.lang.String">
        <constructor-arg value="Spring is fun." />
    </bean>

    <bean id="message"
          class="org.springbyexample.di.xml.SetterMessage">
        <property name="message" ref="springMessage" />
    </bean>

</beans>
```

5. Creating a Spring Application

Spring can be used in standard applications, web applications, full Java EE applications, and other containers, the only requirement is that you run a standard JVM. Spring's resource abstraction allows you to load configuration files from wherever you'd like -- the classpath, the file system, FTP, and HTTP locations. You can also use Spring's resource abstraction for loading other files required for your application.

Once the IoC container is initialized, you can retrieve your Spring beans. By delegating as much bean creation as possible to Spring, there should only be a few key points where the application code needs to directly access the IoC container, and this is true even for legacy applications. [1] If you're developing a web application, you may not need to directly access the IoC container at all since it will automatically handle instantiation of your controller and any beans it requires.

The lowest level implementation of the IoC container is the `BeanFactory`, but it is recommended to use an `ApplicationContext` for your application. The `ApplicationContext` is a subclass of the `BeanFactory` interface so it has all the functionality a `BeanFactory` has and more. Unless you are writing an application that needs an extremely small memory footprint, `BeanFactory` shouldn't be used directly.

There are a few different `ApplicationContext` implementations that can be used, which can be learned about by reading the Spring Framework's documentation and source code. For the purposes of this example, we'll use a very popular one –`ClassPathXmlApplicationContext`, which defaults to reading resources from the classpath. If you need to use a different location for your classes, you can append prefixes before the configuration file's path such as 'file', 'http', etc. This will force the `ApplicationContext` to read from somewhere other than the default location.

The following class is a standard Java application with a `main` method. The first line of the main method creates a

[1] Instantiating Spring in your application does not violate rule #1 of dependency inversion (see Spring In Context: Core Concepts) because Spring would be a higher level dependency.

`ClassPathXmlApplicationContext` passing in '/application-context.xml' to its constructor which is assigned to the `ApplicationContext` interface. In this case the configuration file is in the root of the classpath. The `ApplicationContext`'s `getBean(String beanName)` method is used on the next line to retrieve the message bean from the IoC container.

```
public class MessageRunner {

    final static Logger logger = LoggerFactory.getLogger(MessageRunner.class);

    /**
     * Main method.
     */
    public static void main(String[] args) {
        logger.info("Initializing Spring context.");

        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("/application-context.xml");

        logger.info("Spring context initialized.");

        Message message = (Message) applicationContext.getBean("message");

        logger.debug("message='" + message.getMessage() + "'");
    }

}
```

Example 4 `MessageRunner`

application-context.xml

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

    <bean id="message"
          class="org.springbyexample.di.app.Message">
        <property name="message" value="Spring is fun." />
    </bean>

</beans>
```

6. Unit Test Beans from Application Context

There are two types of testing that developers typically perform before they release code into production. Functional testing is the testing you do when your application is mostly built and you want to make sure that everything works according to the *functional* requirements you were given.

Unit testing is the process by which you test each component of your application *in isolation*. Unit testing can be done far in advance of functional testing and code release. EJBs are difficult to unit test because it is cumbersome to

tease them apart from the container for testing. Spring makes unit testing easy because each component is intended to be an entity unto itself – callable from anywhere.

Unit testing is an important part of the development process, so important in fact that some methodologies, including Agile [2] , require that the unit test be written before the class that it will be testing. Certainly writing the unit test first will give you a very clear idea of the use case for your class file. But the real value in unit testing is in giving you the ability to isolate where the problems may lie in your code. This will shorten the amount of time you would need to debug your code and ultimately lead to faster time to production.

As you create your unit tests, you may organize them into test suites. There are several third party and open source products that can help you set up and conduct your unit tests. JUnit (http://www.junit.org) is the most popular unit testing framework and can be integrated with Ant, Maven, the Eclipse IDE, and others. Maven (http://maven.apache.org) is a build framework used for compiling, assembling jars, and running unit tests. All of the examples that accompany this book can be built and tested using Maven, and Eclipse plugins for Junit allow you to run your unit tests from within the IDE.

Spring also has unit testing support and it can reduce the size of your test classes if you can use them. Most unit tests are annotated with `@RunWith(SpringJUnit4ClassRunner.class)` and `@ContextConfiguration`. The `@ContextConfiguration` default to loading an XML configuration file from the package and name of the test plus '-context.xml'. For `org.springbyexample.di.xml.SetterMessageTest`, the default XML file is *org/springbyexample/di/xml/SetterMessageTest-context.xml*. Specific XML configuration files can be set on the annotation if the default isn't acceptable.

The `SetterMessageTest` will use the Spring XML configuration file *org/springbyexample/di/xml/SetterMessageTest-context.xml*. The `testMessage()` method uses the `SetterMessage` injected by the test framework into the `message` field. The bean from the XML configuration will be injected into this field because of the `@Autowired` annotation which indicates the field should be autowired by type. Finally, the `testMessage()` method uses JUnit's `assertNotNull` and `assertEquals` methods to check if the `SetterMessageTest` instance isn't `null`, the message isn't `null`, and the message is the expected value. For `assertEquals` the first parameter is the error message that will be shown if the test fails. The second parameter is the expected value and third parameter is the value from the *message* bean.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class SetterMessageTest {

    final Logger logger = LoggerFactory.getLogger(SetterMessageTest.class);

    @Autowired
    private SetterMessage message = null;

    /**
     * Tests message.
     */
    @Test
    public void testMessage() {
        assertNotNull("Constructor message instance is null.", message);

        String msg = message.getMessage();

        assertNotNull("Message is null.", msg);

        String expectedMessage = "Spring is fun.";
```

[2] A conceptual framework that promotes development iterations throughout the lifecycle of a project.

```
        assertEquals("Message should be '" + expectedMessage + "'.", expectedMessage, msg);

        logger.info("message='{}'", msg);
    }

}
```

Example 5 `SetterMessageTest`

SetterMessageTest-context.xml

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

    <bean id="message"
          class="org.springbyexample.di.xml.SetterMessage">
        <property name="message" value="Spring is fun." />
    </bean>

</beans>
```

7. Getting Started

Now that we've walked you through the creation and testing of your first Spring application, it's time to dig a little deeper into Spring. Articles and examples will cover different areas of the Spring Framework and other Spring projects. Before we do, use the following instructions to set up your Spring environment, and check out these basic DI examples.

Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd core/basic-dependency-injection
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

8. Reference

Related Links

- Spring 3.1.x 'The IoC Container' Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-introduction]

Project Information

- Spring Framework 3.1.x

Part II. Core

Core Spring examples.

AspectJ Load-time Weaving in Spring

David Winterfeldt

2008

This example shows using Spring with AspectJ's Load-time Weaving (LTW) to gather statistics. Spring 2.5 added support to make it easier to use AspectJ's LTW. It's also finer grained because you can activate it in specific classloaders using the new *context* namespace by adding this, *<context:load-time-weaver/>*, to your XML configuration. Besides adding a JVM argument and the custom namespace, the rest is standard AspectJ LTW and how to configure the AspectJ configuration file can be found on AspectJ's site.

Note

This example must be run with Java 5 or higher since support for the *-javaagent* JVM argument was added in Java 5. Also, Maven and Eclipse are currently configured to look for the spring-agent.jar at *c:/spring-framework-2.5/dist/weaving/spring-agent.jar* so you will need to edit this to match where it is actually located. This can be changed at the bottom of the Maven pom.xml or in the Eclipse Run Dialog's argument tab.

1. JVM Argument

This JVM argument needs to be set in order for Spring's *context:load-time-weaver* to work, although with certain classloaders this isn't necessary.

```
-javaagent:/path/to/jar/spring-agent.jar
```

2. Spring Configuration

The *context:load-time-weaver* registers AspectJ's Load-time Weaver to the current classloader. So, not only Spring beans will be targeted, but any class loaded in the classloader that match the defined pointcuts.

applicationContext.xml

```
<?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"
    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.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver />
```

```
    <bean id="processor" class="org.springbyexample.aspectjLoadTimeWeaving.Processor" />

</beans>
```

3. AspectJ Configuration

For AspectJ LTW a configuration file will automatically be loaded from META-INF/aop.xml. Multiple includes or excludes can be used to target weaving, and if you wanted to include all subpackages and not just a specific package the package name could be specified as `org.springbyexample.aspectjLoadTimeWeaving..*` (notice the double periods before the asterisk). In this example, the aspect specified contains the pointcut and advice being applied.

/META-INF/aop.xml

```
<!DOCTYPE aspectj PUBLIC
        "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
    <weaver>
        <!-- only weave classes in this package -->
        <include within="org.springbyexample.aspectjLoadTimeWeaving.*" />
    </weaver>
    <aspects>
        <!-- use only this aspect for weaving -->
        <aspect name="org.springbyexample.aspectjLoadTimeWeaving.PerformanceAdvice" />
    </aspects>
</aspectj>
```

4. Code Example

```
@Aspect
public class PerformanceAdvice {

    @Pointcut("execution(public * org.springbyexample.aspectjLoadTimeWeaving..*.*(..))")
    public void aspectjLoadTimeWeavingExamples() {
    }

    @Around("aspectjLoadTimeWeavingExamples()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        final Logger logger = LoggerFactory.getLogger(pjp.getSignature().getDeclaringType());

        StopWatch sw = new StopWatch(getClass().getSimpleName());

        try {
            sw.start(pjp.getSignature().getName());

            return pjp.proceed();
        } finally {
            sw.stop();
```

```
            logger.debug(sw.prettyPrint());
        }
    }

}
```

Example 1 `PerformanceAdvice`

5. Reference

Related Links

- AspectJ Load-time Weaving Spring Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/aop.html#aop-aj-ltw]
- AspectJ Load-time Weaving [http://www.eclipse.org/aspectj/doc/next/devguide/ltw.html]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd core/aspectj-load-time-weaving
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- AspectJ 1.6.x

Part III. Persistence

Persistence examples.

Simple Spring JDBC Template

David Winterfeldt

2008

A simple example using `SimpleJdbcTemplate`.

1. Spring Configuration

The Spring JDBC Template just needs a `DataSource`. The *jdbc:embedded-database* element automatically initializes an HSQL DB `DataSource` with the schema.sql script.

```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:/schema.sql" />
</jdbc:embedded-database>
```

2. Code Example

Inject the `DataSource` into a bean and in the setter for the `DataSource` create a `SimpleJdbcTemplate`. Then the template has many methods for doing updates, queries, and deletes. The example below shows getting a `List` of `Maps` from the template.

```
protected SimpleJdbcTemplate simpleJdbcTemplate = null;

@Autowired
public void setDataSource(final DataSource dataSource) {
    this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
}


List<Map<String, Object>> lPersonMaps = simpleJdbcTemplate.queryForList("SELECT * FROM PERSON");

Map<String, Object> hPerson = lPersonMaps.get(0);

Integer id = (Integer)hPerson.get("ID");
String firstName = (String)hPerson.get("FIRST_NAME");
String lastName = (String)hPerson.get("LAST_NAME");
```

3. Reference

Related Links

- Spring 3.1.x `SimpleJdbcTemplate` Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-SimpleJdbcTemplate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]

Simple Hibernate XML Configuration

David Winterfeldt

2008

A simple example using Hibernate with an XML configuration to find, save, and update a record.

1. Spring Configuration

The *jdbc:embedded-database* element is used to initialize the test HSQLDB database and the `LocalSessionFactoryBean` is using to configure Hibernate. The *mappingLocations* property is used to set a list of Hibernate XML mapping files. The Person DAO is configured using Hibernate's session factory.

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" />
    </jdbc:embedded-database>

    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
          p:dataSource-ref="dataSource">
        <property name="mappingLocations">
            <list>
                <value>classpath:org/springbyexample/orm/hibernate3/bean/Address.hbm.xml</value>
                <value>classpath:org/springbyexample/orm/hibernate3/bean/Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.format_sql=true
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

</beans>
```

2. Hibernate Configuration

A very simple Hibernate configuration mapping the PERSON table to the `Person` class.

```
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.springbyexample.orm.hibernate3.bean" default-access="field">

    <class name="Person" table="PERSON">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>

        <property name="firstName" column="FIRST_NAME" />
        <property name="lastName" column="LAST_NAME" />
    </class>

</hibernate-mapping>
```

3. SQL Script

```
CREATE TABLE PERSON (
    ID INTEGER generated by default as identity (start with 1) not null,
    FIRST_NAME varchar(50) not null,
    LAST_NAME varchar(50) not null
);
```

4. Code Example

The Hibernate `SessionFactory` is used to create a `HibernateTemplate` as it's set. The template is then used for any Hibernate database operations. Spring's `HibernateTemplate` converts all exceptions to runtime exceptions so it isn't necessary to handle any exceptions.

```
@Repository
public class PersonDaoImpl implements PersonDao {

    protected HibernateTemplate template = null;

    /**
     * Sets Hibernate session factory and creates a
     * <code>HibernateTemplate</code> from it.
     */
    public void setSessionFactory(SessionFactory sessionFactory) {
        template = new HibernateTemplate(sessionFactory);
    }

    /**
     * Find all persons.
     */
```

```
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() throws DataAccessException {
        return template.find("from Person");
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) throws DataAccessException {
        return template.find("from Person p where p.lastName = ?", lastName);
    }

    /**
     * Saves person.
     */
    public void save(Person person) {
        template.saveOrUpdate(person);
    }

}
```

5. Reference

Related Links

- Spring 3.1.x Hibernate Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-hibernate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]

One to Many Hibernate XML Configuration

David Winterfeldt

2008

A simple example using a one-to-many relationship in Hibernate with an XML configuration to find, save, and update a record. A `Person` has a one-to-many relationship with `Address`.

1. Spring Configuration

The *jdbc:embedded-database* element is used to initialize the test HSQLDB database and the `LocalSessionFactoryBean` is used to configure Hibernate. The mappingLocations property is used to set a list of Hibernate XML mapping files. The Person DAO is configured using Hibernate's session factory.

shared-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" />
    </jdbc:embedded-database>

    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
          p:dataSource-ref="dataSource">
        <property name="mappingLocations">
            <list>
                    <value>classpath:org/springbyexample/orm/hibernate3/bean/Address.hbm.xml</value>
                <value>classpath:org/springbyexample/orm/hibernate3/bean/Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.format_sql=true
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

</beans>
```

PersonDaoTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <import resource="classpath:org/springbyexample/orm/hibernate3/shared-context.xml"/>

    <bean id="personDao"
          class="org.springbyexample.orm.hibernate3.dao.PersonDaoImpl"
          p:sessionFactory-ref="sessionFactory" />

</beans>
```

2. Hibernate Configuration

A very simple Hibernate configuration mapping the PERSON table to the `Person` class and the ADDRESS table to the `Address` class. There is a one-to-many relationship between the PERSON and ADDRESS table. The *set* element in the `Person` mapping creates the relationship from the PERSON to the ADDRESS table and stores the `Address` instances in a `java.util.Set`. The *key* element inside the *set* element indicates ADDRESS.PERSON_ID is the column to match against PERSON.ID to retrieve addresses associated with a person.

Person.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.springbyexample.orm.hibernate3.bean" default-access="field">

    <class name="Person" table="PERSON">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>

        <property name="firstName" column="FIRST_NAME" />
        <property name="lastName" column="LAST_NAME" />
        <set name="addresses" lazy="false" inverse="false">
            <key column="PERSON_ID"/>
            <one-to-many class="Address"/>
        </set>
        <property name="created" column="CREATED" />
    </class>

</hibernate-mapping>
```

Address.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.springbyexample.orm.hibernate3.bean" default-access="field">

    <class name="Address" table="ADDRESS">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>

        <property name="address" column="ADDRESS" />
        <property name="city" column="CITY" />
        <property name="state" column="STATE" />
        <property name="zipPostal" column="ZIP_POSTAL" />
        <property name="created" column="CREATED" />
    </class>

</hibernate-mapping>
```

3. SQL Script

Excerpt from schema.sql

```
CREATE TABLE PERSON (
    ID integer identity primary key,
    FIRST_NAME varchar(50) not null,
    LAST_NAME varchar(50) not null,
    CREATED timestamp,
    CONSTRAINT IDX_PERSON_ID PRIMARY KEY (ID)
);

CREATE TABLE ADDRESS (
    ID integer identity primary key,
    PERSON_ID integer,
    ADDRESS varchar(255),
    CITY varchar(50) not null,
    STATE varchar(50) not null,
    ZIP_POSTAL varchar(30) not null,
    CREATED timestamp,
    CONSTRAINT IDX_ADDRESS_ID PRIMARY KEY (ID),
    CONSTRAINT FK_ADDRESS_PERSON_ID FOREIGN KEY (PERSON_ID) REFERENCES PERSON(ID) on delete cascade
);
```

4. Code Example

The Hibernate `SessionFactory` is used to create a `HibernateTemplate` as it's set. The template is then used for any Hibernate database operations. Spring's `HibernateTemplate` converts all exceptions to runtime exceptions so it isn't necessary to handle any exceptions.

The `@Transactional` annotation isn't used in this example because there isn't anything configuring transactions in the Spring configuration.

```
@Repository
@Transactional(readOnly = true)
public class PersonDaoImpl implements PersonDao {

    protected HibernateTemplate template = null;

    /**
     * Sets Hibernate session factory.
     */
    public void setSessionFactory(SessionFactory sessionFactory) {
        template = new HibernateTemplate(sessionFactory);
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() throws DataAccessException {
        return template.find("from Person");
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) throws DataAccessException {
        return template.find("from Person p where p.lastName = ?", lastName);
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void save(Person person) {
        template.saveOrUpdate(person);
    }

}
```

Example 1 `PersonDaoImpl`

5. Reference

Related Links

- Spring 3.1.x Hibernate Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-hibernate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]

- One to Many Hibernate Annotation Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-hibernate
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Hibernate 3.6.10.Final

One to Many Hibernate Annotation Configuration

David Winterfeldt

2008

A simple example using a one-to-many relationship in Hibernate with an Annotation configuration to find, save, and update a record. A `Person` has a one-to-many relationship with `Address`.

1. Spring Configuration

The *jdbc:embedded-database* element is used to initialize the test HSQLDB database and the `LocalSessionFactoryBean` is used to configure Hibernate.

shared-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" />
    </jdbc:embedded-database>

    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
          p:dataSource-ref="dataSource">
        <property name="mappingLocations">
            <list>
                    <value>classpath:org/springbyexample/orm/hibernate3/bean/Address.hbm.xml</value>
                <value>classpath:org/springbyexample/orm/hibernate3/bean/Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.format_sql=true
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

</beans>
```

The *annotatedClasses* property is used to set a list of Hibernate annotated classes. The Person DAO is configured using Hibernate's session factory.

PersonAnnotationDaoTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <import resource="classpath:org/springbyexample/orm/hibernate3/shared-context.xml"/>

    <!-- Override xml session factory imported from shared-context.xml -->
    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
          p:dataSource-ref="dataSource">
        <property name="annotatedClasses">
                <list>
                        <value>org.springbyexample.orm.hibernate3.annotation.bean.Person</value>
                        <value>org.springbyexample.orm.hibernate3.annotation.bean.Address</value>
                </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.format_sql=true
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

    <bean id="personDao"
          class="org.springbyexample.orm.hibernate3.annotation.dao.PersonDaoImpl"
          p:sessionFactory-ref="sessionFactory" />

</beans>
```

2. Hibernate Configuration

The `@Entity` annotation indicates that the JavaBean is a persistent entity. By default the class name will be used for the Hibernate entity name as this class is registered with Hibernate. A different entity name could be specified by using the annotations *name* attribute (ex: `@Entity(name="Employee")`). An `@Table` annotation can explicitly configure which table the entity is mapped to, although in this case it isn't necessary since it would default to the name of the class which matches the table name.

The one-to-many relationship from `Person` to `Address` is configured on `public Set<Address> getAddresses()`. The `@OneToMany(fetch=FetchType.EAGER)` annotation indicates a one-to-many relationship and that any associated addresses should be eagerly fetched. The `@JoinColumn(name="PERSON_ID", nullable=false)` annotation indicates ADDRESS.PERSON_ID is the column to match against PERSON.ID to retrieve addresses associated with a person.

```
@Entity
public class Person {

    private Integer id = null;
    private String firstName = null;
    private String lastName = null;
```

```
    private Set<Address> addresses = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets first name.
     */
    @Column(name="FIRST_NAME")
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Gets last name.
     */
    @Column(name="LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets list of <code>Address</code>es.
     */
    @OneToMany(fetch=FetchType.EAGER)
    @JoinColumn(name="PERSON_ID", nullable=false)
    public Set<Address> getAddresses() {
        return addresses;
    }

    /**
     * Sets list of <code>Address</code>es.
     */
    public void setAddresses(Set<Address> addresses) {
        this.addresses = addresses;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }
```

```
    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

...


}
```

Example 1 Excerpt from `Person`

```
@Entity
public class Address {

    private Integer id = null;
    private String address = null;
    private String city = null;
    private String state = null;
    private String zipPostal = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets address.
     */
    public String getAddress() {
        return address;
    }

    /**
     * Sets address.
     */
    public void setAddress(String address) {
        this.address = address;
    }

    /**
     * Gets city.
     */
    public String getCity() {
        return city;
    }

    /**
```

```
     * Sets city.
     */
    public void setCity(String city) {
        this.city = city;
    }

    /**
     * Gets state.
     */
    public String getState() {
        return state;
    }

    /**
     * Sets state.
     */
    public void setState(String state) {
        this.state = state;
    }

    /**
     * Gets zip or postal code.
     */
    @Column(name="ZIP_POSTAL")
    public String getZipPostal() {
        return zipPostal;
    }

    /**
     * Sets zip or postal code.
     */
    public void setZipPostal(String zipPostal) {
        this.zipPostal = zipPostal;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

...

}
```

Example 2 Excerpt from `Address`

3. SQL Script

Excerpt from schema.sql

```
CREATE TABLE PERSON (
    ID integer identity primary key,
    FIRST_NAME varchar(50) not null,
    LAST_NAME varchar(50) not null,
```

```
    CREATED timestamp,
    CONSTRAINT IDX_PERSON_ID PRIMARY KEY (ID)
);

CREATE TABLE ADDRESS (
    ID integer identity primary key,
    PERSON_ID integer,
    ADDRESS varchar(255),
    CITY varchar(50) not null,
    STATE varchar(50) not null,
    ZIP_POSTAL varchar(30) not null,
    CREATED timestamp,
    CONSTRAINT IDX_ADDRESS_ID PRIMARY KEY (ID),
    CONSTRAINT FK_ADDRESS_PERSON_ID FOREIGN KEY (PERSON_ID) REFERENCES PERSON(ID) on delete cascade
);
```

4. Code Example

The Hibernate `SessionFactory` is used to create a `HibernateTemplate` as it's set. The template is then used for any Hibernate database operations. Spring's `HibernateTemplate` converts all exceptions to runtime exceptions so it isn't necessary to handle any exceptions.

The `@Transactional` annotation isn't used in this example because there isn't anything configuring transactions in the Spring configuration.

```


@Repository
@Transactional(readOnly = true)
public class PersonDaoImpl implements PersonDao {

    protected HibernateTemplate template = null;

    /**
     * Sets Hibernate session factory.
     */
    public void setSessionFactory(SessionFactory sessionFactory) {
        template = new HibernateTemplate(sessionFactory);
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() throws DataAccessException {
        return template.find("from Person");
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) throws DataAccessException {
        return template.find("from Person p where p.lastName = ?", lastName);
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void save(Person person) {
        template.saveOrUpdate(person);
```

```
    }
}
```

Example 3 `PersonDaoImpl`

5. Reference

Related Links

- Spring 3.1.x Hibernate Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-hibernate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- One to Many Hibernate XML Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-hibernate
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Hibernate 3.6.10.Final

One to Many `JpaTemplate` Hibernate Configuration

David Winterfeldt

2009

This is a one-to-many `JpaTemplate` example using Hibernate that can to find, save, and update a record. There is a `Person` that has a one-to-many relationship to `Address`. This example is very similar to One to Many Hibernate Annotation Configuration example since that example was already using `javax.persistence` annotations to configure the `Person` and `Address` beans. Using `JpaTemplate` can make an easier upgrade path to JPA from an application already using Spring's `HibernateTemplate`.

Note

This example has the DAO class extend `JpaDaoSupport` and uses the `JpaTemplate`, but Spring recommends the native JPA style of coding.

> `JpaTemplate` mainly exists as a sibling of JdoTemplate and HibernateTemplate, offering the same style for people used to it..
>
> — Spring 3.1.x Classic ORM JPA Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/classic-spring.html#orm-jpa-template]

1. Spring Configuration

The Person DAO is configured using *context:component-scan*. Then *tx:annotation-driven* will configure transactions on any beans with `@Transactional`, and just after it the JPA transaction manager is setup. The *jdbc:embedded-database* element is used to initialize the test HSQLDB database. The `LocalContainerEntityManagerFactoryBean` is configured with the `DataSource` and for use with Hibernate as the JPA adapter. It also sets the *persistenceUnitName* attribute to specify which persistence unit to use for this entity manager (multiple examples use this *persistence.xml*).

PersonTemplateDaoTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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.springframework.org/schema/context/spring-context.xsd
                        http://www.springframework.org/schema/jdbc
                        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                        http://www.springframework.org/schema/tx
                        http://www.springframework.org/schema/tx/spring-tx.xsd">
```

```
    <context:component-scan base-package="org.springbyexample.orm.jpa.template.dao" />

    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager"
          p:entityManagerFactory-ref="entityManagerFactory"/>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" encoding="UTF-8" />
    </jdbc:embedded-database>

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              p:dataSource-ref="dataSource"
          p:persistenceUnitName="simple-jpa-template">
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
                  p:showSql="true"
                  p:generateDdl="true"
                  p:database="HSQL" />
        </property>
    </bean>

</beans>
```

2. JPA Entity Configuration

The `@Entity` annotation indicates that the JavaBean is a persistent entity. JPA would automatically pick up this class except the JPA configuration explicitly registers the classes each persistence unit should handle. An `@Table` annotation can explicitly configure which table the entity is mapped to, although in this case it isn't necessary since it would default to the name of the class which matches the table name.

Note

Using the `ImprovedNamingStrategy` in your JPA *persistence.xml* can give better translation from camel case classes and field names to standard underscore delimited database names. An example of this is that instead of having to explicitly put `@Column(name="FIRST_NAME")` on the first name field, it automatically converts the camel case of the field name to use underscores.

```
<property name="hibernate.ejb.naming_strategy" value="org.hibernate.cfg.ImprovedNamingStrategy"/>
```

The one-to-many relationship from `Person` to `Address` is configured on `public Set<Address> getAddresses()`. The `@OneToMany(fetch=FetchType.EAGER)` annotation indicates a one-to-many relationship and that any associated addresses should be eagerly fetched. The `@JoinColumn(name="PERSON_ID", nullable=false)` annotation indicates `ADDRESS.PERSON_ID` is the column to match against `PERSON.ID` to retrieve addresses associated with a person.

```
```

```
@Entity
public class Person implements Serializable {

    private static final long serialVersionUID = -8712872385957386182L;

    private Integer id = null;
    private String firstName = null;
    private String lastName = null;
    private Set<Address> addresses = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets first name.
     */
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Gets last name.
     */
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets list of <code>Address</code>es.
     */
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "PERSON_ID", nullable = false)
    public Set<Address> getAddresses() {
        return addresses;
    }

    /**
     * Sets list of <code>Address</code>es.
     */
    public void setAddresses(Set<Address> addresses) {
        this.addresses = addresses;
    }
```

```
    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

    ...

}
```

Example 1 Excerpt from `Person`

```
@Entity
public class Address implements Serializable {

    private static final long serialVersionUID = 7851794269407495684L;

    private Integer id = null;
    private String address = null;
    private String city = null;
    private String state = null;
    private String zipPostal = null;
    private String country = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets address.
     */
    public String getAddress() {
        return address;
    }

    /**
     * Sets address.
     */
    public void setAddress(String address) {
        this.address = address;
    }

    /**
     * Gets city.
```

```
     */
    public String getCity() {
        return city;
    }

    /**
     * Sets city.
     */
    public void setCity(String city) {
        this.city = city;
    }

    /**
     * Gets state.
     */
    public String getState() {
        return state;
    }

    /**
     * Sets state.
     */
    public void setState(String state) {
        this.state = state;
    }

    /**
     * Gets zip or postal code.
     */
    public String getZipPostal() {
        return zipPostal;
    }

    /**
     * Sets zip or postal code.
     */
    public void setZipPostal(String zipPostal) {
        this.zipPostal = zipPostal;
    }

    /**
     * Gets country.
     */
    public String getCountry() {
        return country;
    }

    /**
     * Sets country.
     */
    public void setCountry(String country) {
        this.country = country;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

    ...

}
```

```
```

Example 2 Excerpt from `Address`

3. JPA Configuration

Excerpt from *META-INF/persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="simple-jpa-template" transaction-type="RESOURCE_LOCAL">
        <class>org.springbyexample.orm.jpa.bean.Person</class>
        <class>org.springbyexample.orm.jpa.bean.Address</class>

        <exclude-unlisted-classes/>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
        </properties>
    </persistence-unit>

    ...

</persistence>
```

4. Code Example

```
@Repository
@Transactional(readOnly = true)
public class PersonTemplateDaoImpl extends JpaDaoSupport implements PersonTemplateDao {

    /**
     * Constructor
     */
    @Autowired
    public PersonTemplateDaoImpl(EntityManagerFactory entityManagerFactory) {
        super.setEntityManagerFactory(entityManagerFactory);
    }

    /**
     * Find persons.
     */
```

```
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() {
        return getJpaTemplate().find("from Person");
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) {
        return getJpaTemplate().find("from Person p where p.lastName = ?", lastName);
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void save(Person person) {
        getJpaTemplate().merge(person);
    }

}
```

Example 3 `PersonTemplateDaoImpl`

5. Reference

Related Links

- Spring 3.1.x JPA Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-jpa]
- Hibernate [http://www.hibernate.org/]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- One to Many JPA Hibernate Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-jpa
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Java Persistence API 2.0

One to Many JPA Hibernate Configuration

David Winterfeldt

2009

This is a one-to-many JPA configuration that can to find, save, and update a record. Hibernate is used as the JPA provider. There is a `Person` that has a one-to-many relationship to `Address`. This example is very similar to One to Many `JpaTemplate` Hibernate Configuration and even uses the same `javax.persistence` annotated beans, but uses the native JPA style for accessing data objects.

1. Spring Configuration

The Person DAO is configured using *context:component-scan*. Then *tx:annotation-driven* will configure transactions on any beans with `@Transactional`, and just after it the JPA transaction manager is setup. The *jdbc:embedded-database* element is used to initialize the test HSQLDB database. The `LocalContainerEntityManagerFactoryBean` is configured with the `DataSource` and for use with Hibernate as the JPA adapter. It also sets the *persistenceUnitName* attribute to specify which persistence unit to use for this entity manager (multiple examples use the *persistence.xml*).

PersonDaoTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

    <context:component-scan base-package="org.springbyexample.orm.jpa.dao" />

    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager"
          p:entityManagerFactory-ref="entityManagerFactory"/>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" encoding="UTF-8" />
    </jdbc:embedded-database>

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              p:dataSource-ref="dataSource"
          p:persistenceUnitName="simple-jpa">
```

```
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
    </bean>

</beans>
```

2. JPA Entity Configuration

The `@Entity` annotation indicates that the JavaBean is a persistent entity. JPA would automatically pick up this class except the JPA configuration explicitly registers the classes each persistence unit should handle. An `@Table` annotation can explicitly configure which table the entity is mapped to, although in this case it isn't necessary since it would default to the name of the class which matches the table name.

The one-to-many relationship from `Person` to `Address` is configured on `public Set<Address> getAddresses()`. The `@OneToMany(fetch=FetchType.EAGER)` annotation indicates a one-to-many relationship and that any associated addresses should be eagerly fetched. The `@JoinColumn(name="PERSON_ID", nullable=false)` annotation indicates `ADDRESS.PERSON_ID` is the column to match against `PERSON.ID` to retrieve addresses associated with a person.

Note

Using the `ImprovedNamingStrategy` in your JPA *persistence.xml* can give better translation from camel case classes and field names to standard underscore delimited database names. An example of this is that instead of having to explicitly put `@Column(name="FIRST_NAME")` on the first name field, it automatically converts the camel case of the field name to use underscores.

```
<property name="hibernate.ejb.naming_strategy" value="org.hibernate.cfg.ImprovedNamingStrategy"/>
```

```
@Entity
public class Person implements Serializable {

    private static final long serialVersionUID = -8712872385957386182L;

    private Integer id = null;
    private String firstName = null;
    private String lastName = null;
    private Set<Address> addresses = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Integer getId() {
        return id;
    }
```

```
    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets first name.
     */
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Gets last name.
     */
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets list of <code>Address</code>es.
     */
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "PERSON_ID", nullable = false)
    public Set<Address> getAddresses() {
        return addresses;
    }

    /**
     * Sets list of <code>Address</code>es.
     */
    public void setAddresses(Set<Address> addresses) {
        this.addresses = addresses;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

    ...

}
```

Example 1 Excerpt from `Person`

```
@Entity
public class Address implements Serializable {

    private static final long serialVersionUID = 7851794269407495684L;

    private Integer id = null;
    private String address = null;
    private String city = null;
    private String state = null;
    private String zipPostal = null;
    private String country = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets address.
     */
    public String getAddress() {
        return address;
    }

    /**
     * Sets address.
     */
    public void setAddress(String address) {
        this.address = address;
    }

    /**
     * Gets city.
     */
    public String getCity() {
        return city;
    }

    /**
     * Sets city.
     */
    public void setCity(String city) {
        this.city = city;
    }

    /**
     * Gets state.
     */
    public String getState() {
        return state;
    }

    /**
     * Sets state.
     */
```

```
    public void setState(String state) {
        this.state = state;
    }

    /**
     * Gets zip or postal code.
     */
    public String getZipPostal() {
        return zipPostal;
    }

    /**
     * Sets zip or postal code.
     */
    public void setZipPostal(String zipPostal) {
        this.zipPostal = zipPostal;
    }

    /**
     * Gets country.
     */
    public String getCountry() {
        return country;
    }

    /**
     * Sets country.
     */
    public void setCountry(String country) {
        this.country = country;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

    ...

}
```

Example 2 Excerpt from `Address`

3. JPA Configuration

Hibernate is setup as the JPA provider. Another JPA provider, like EclipseLink [http://www.eclipse.org/eclipselink/], could be specified and if all your code just uses JPA nothing else would need to be changed.

The JPA implementation is specified using the *provider* element. The `Person` & `Address` classes are explicitly configured, and JPA's scanning for entity beans is turned off by specifying the *exclude-unlisted-classes* element. This is mainly because there are multiple JPA entities in one project for these examples, otherwise having automatic

scanning would normally be fine.

The *properties* element can have a list of properties to pass into JPA provider for configuring provider specific details. In this case the database being used, whether or not to show generated SQL, whether or not to to generate the schema or validate an existing one, the naming strategy, and the cache provider.

Note

In production it would be better to use Ehcache [http://ehcache.org/] for the cache provider.

```
<property name="hibernate.cache.provider_class" value="org.hibernate.cache.EhCacheProvider" />
```

Excerpt from *META-INF/persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    ...

    <persistence-unit name="simple-jpa">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>org.springbyexample.orm.jpa.bean.Person</class>
        <class>org.springbyexample.orm.jpa.bean.Address</class>

        <exclude-unlisted-classes/>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
        </properties>
    </persistence-unit>

    ...

</persistence>
```

4. Code Example

```
@Repository
```

```
@Transactional(readOnly = true)
public class PersonDaoImpl implements PersonDao {

    private EntityManager em = null;

    /**
     * Sets the entity manager.
     */
    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    /**
     * Find persons.
     */
    public Person findPersonById(Integer id) {
        return em.find(Person.class, id);
    }

    /**
     * Find persons using a start index and max number of results.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons(final int startIndex, final int maxResults) {
        return em.createQuery("select p from Person p order by p.lastName, p.firstName")
            .setFirstResult(startIndex).setMaxResults(maxResults).getResultList();
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() {
        return em.createQuery("select p from Person p order by p.lastName,
p.firstName").getResultList();
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) {
        return em.createQuery("select p from Person p where p.lastName = :lastName order by
p.lastName, p.firstName")
            .setParameter("lastName", lastName).getResultList();
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person save(Person person) {
        return em.merge(person);
    }

    /**
     * Deletes person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void delete(Person person) {
        em.remove(em.merge(person));
    }

    /**
     * Saves address to person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person saveAddress(Integer id, Address address) {
        Person person = findPersonById(id);
```

```
        if (person.getAddresses().contains(address)) {
            person.getAddresses().remove(address);
        }

        person.getAddresses().add(address);

        return save(person);
    }

    /**
     * Deletes address from person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person deleteAddress(Integer id, Integer addressId) {
        Person person = findPersonById(id);

        Address address = new Address();
        address.setId(addressId);

        if (person.getAddresses().contains(address)) {
            for (Address a : person.getAddresses()) {
                if (a.getId().equals(addressId)) {
                    em.remove(a);
                    person.getAddresses().remove(address);

                    break;
                }
            }
        }

        return person;
    }

}
```

Example 3 `PersonDaoImpl`

5. Reference

Related Links

- Spring 3.1.x JPA Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-jpa]
- Hibernate [http://www.hibernate.org/]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- One to Many `JpaTemplate` Hibernate Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-jpa
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Java Persistence API 2.0

JPA Joined Inheritance

David Winterfeldt

2009

This shows a JPA configuration for inheritance. There are three different types, which are single table, joined, and table per class. This example uses joined inheritance. Joined inheritance involves a base table for shared fields and a table for each subclass. There is a `Person` class that is the parent for `Student` and `Professional`. This example is based on One to Many JPA Hibernate Configuration, but has been modified enough to add inheritance.

1. Spring Configuration

The Person DAO is configured using *context:component-scan*. Then *tx:annotation-driven* will configure transactions on any beans with `@Transactional`, and just after it the JPA transaction manager is setup. The *jdbc:embedded-database* element is used to initialize the test HSQLDB database. The `LocalContainerEntityManagerFactoryBean` is configured with the `DataSource` and for use with Hibernate as the JPA adapter. The *persistenceUnitName* attribute is set to specify which persistence unit to use (multiple examples use the *persistence.xml*).

PersonInheritanceDaoTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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.springframework.org/schema/context/spring-context.xsd
                        http://www.springframework.org/schema/jdbc
                        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                        http://www.springframework.org/schema/tx
                        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <context:component-scan base-package="org.springbyexample.orm.jpa.inheritance.dao" />

    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager"
          p:entityManagerFactory-ref="entityManagerFactory"/>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema_inheritance.sql" encoding="UTF-8" />
    </jdbc:embedded-database>

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              p:dataSource-ref="dataSource"
          p:persistenceUnitName="inheritance-jpa">
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
    </bean>
```

```
</beans>
```

2. JPA Entity Configuration

The `javax.persistence` annotation `@Entity` indicates this is a persistent entity bean, and JPA would pick it up during classpath scanning. Although in this configuration, JPA scanning for entity beans is turned off. An `@Table` annotation can explicitly configure which table the entity is mapped to, although in this case it isn't necessary since it would default to the name of the class which matches the table name.

Note

Using the `ImprovedNamingStrategy` in your JPA *persistence.xml* can give better translation from camel case classes and field names to standard underscore delimited database names. An example of this is that instead of having to explicitly put `@Column(name="FIRST_NAME")` on the first name field, it automatically converts the camel case of the field name to use underscores.

```
<property name="hibernate.ejb.naming_strategy" value="org.hibernate.cfg.ImprovedNamingStrategy"/>
```

Inheritance is configured using the `@Inheritance` and `@DiscriminatorColumn` annotations. The `@Inheritance` configures the inheritance type to joined. The `@DiscriminatorColumn` annotation sets the field used to determine the subclass to the *TYPE* field and the field type to integer.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.INTEGER)
public class Person implements Serializable {

    private static final long serialVersionUID = -2175150694352541150L;

    private Integer id = null;
    private String firstName = null;
    private String lastName = null;
    private Set<Address> addresses = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
     * Sets id (primary key).
     */
    public void setId(Integer id) {
```

```
        this.id = id;
    }

    /**
     * Gets first name.
     */
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Gets last name.
     */
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets list of <code>Address</code>es.
     */
    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    @JoinColumn(name="PERSON_ID", nullable=false)
    public Set<Address> getAddresses() {
        return addresses;
    }

    /**
     * Sets list of <code>Address</code>es.
     */
    public void setAddresses(Set<Address> addresses) {
        this.addresses = addresses;
    }

    /**
     * Gets date created.
     */
    public Date getCreated() {
        return created;
    }

    /**
     * Sets date created.
     */
    public void setCreated(Date created) {
        this.created = created;
    }

    public Address findAddressById(Integer id) {
        Address result = null;

        if (addresses != null) {
            for (Address address : addresses) {
                if (address.getId().equals(id)) {
                    result = address;

                    break;
                }
            }
```

```
        }

        return result;
    }

    ...

}
```

Example 1 Excerpt from `Person`

The `Student` class extends `Person`. The `@DiscriminatorValue` indicates the value to be used for the discriminator column in the parent class when storing and retrieving this subclass. The *PERSON_STUDENT* table stores all the extra values unique to this subclass, which in this case is just the school name of the student.

```
@Entity
@Table(name="PERSON_STUDENT")
@DiscriminatorValue("1")
public class Student extends Person {

    private static final long serialVersionUID = -8933409594928827120L;

    private String schoolName = null;

    /**
     * Gets school name.
     */
    public String getSchoolName() {
        return schoolName;
    }

    /**
     * Sets school name.
     */
    public void setSchoolName(String schoolName) {
        this.schoolName = schoolName;
    }

}
```

Example 2 `Student`

The `Professional` class is very similar to the `Student` class except it has a different table, discriminator value, and it's unique field is the professional's company name.

```
@Entity
@Table(name="PERSON_PROFESSIONAL")
@DiscriminatorValue("2")
public class Professional extends Person {

    private static final long serialVersionUID = 8199967229715812072L;

    private String companyName = null;

    /**
     * Gets company name.
```

```
     */
    public String getCompanyName() {
        return companyName;
    }

    /**
     * Sets company name.
     */
    public void setCompanyName(String companyName) {
        this.companyName = companyName;
    }

}
```

Example 3 `Professional`

3. JPA Configuration

Hibernate is setup as the JPA provider. Another JPA provider, like EclipseLink [http://www.eclipse.org/eclipselink/], could be specified and if all your code just uses JPA nothing else would need to be changed. The `Person` & `Address` classes are explicitly configured, and JPA's scanning for entity beans is turned off by specifying the *exclude-unlisted-classes* element. Some Hibernate specific configuration items are set within the *properties* element.

Excerpt from *META-INF/persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    ...

    <persistence-unit name="inheritance-jpa">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>org.springbyexample.orm.jpa.inheritance.bean.Student</class>
        <class>org.springbyexample.orm.jpa.inheritance.bean.Professional</class>
        <class>org.springbyexample.orm.jpa.inheritance.bean.Address</class>

        <exclude-unlisted-classes/>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
        </properties>
    </persistence-unit>

</persistence>
```

4. Code Example

```
@Repository
@Transactional(readOnly = true)
public class PersonInheritanceImpl implements PersonInheritanceDao {

    private EntityManager em = null;

    /**
     * Sets the entity manager.
     */
    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    /**
     * Find persons.
     */
    public Person findPersonById(Integer id) {
        return em.find(Person.class, id);
    }

    /**
     * Find persons using a start index and max number of results.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons(final int startIndex, final int maxResults) {
        return em.createQuery("select p from Person p order by p.lastName, p.firstName")
            .setFirstResult(startIndex).setMaxResults(maxResults).getResultList();
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() {
        return em.createQuery("select p from Person p order by p.lastName,
p.firstName").getResultList();
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) {
        return em.createQuery("select p from Person p where p.lastName = :lastName order by
p.lastName, p.firstName")
            .setParameter("lastName", lastName).getResultList();
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person save(Person person) {
        return em.merge(person);
    }

    /**
     * Deletes person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void delete(Person person) {
        em.remove(em.merge(person));
    }

    /**
```

```
     * Saves address to person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person saveAddress(Integer id, Address address) {
        Person person = findPersonById(id);

        if (person.getAddresses().contains(address)) {
            person.getAddresses().remove(address);
        }

        person.getAddresses().add(address);

        return save(person);
    }

    /**
     * Deletes address from person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person deleteAddress(Integer id, Integer addressId) {
        Person person = findPersonById(id);

        Address address = new Address();
        address.setId(addressId);

        if (person.getAddresses().contains(address)) {
            for (Address a : person.getAddresses()) {
                if (a.getId().equals(addressId)) {
                    em.remove(a);
                    person.getAddresses().remove(address);

                    break;
                }
            }
        }

        return person;
    }

}
```

Example 4 `PersonInheritanceImpl`

5. SQL Script

Excerpt from *schema_inheritance.sql*

```
...

CREATE TABLE PERSON_TYPE (
    ID integer identity primary key,
    NAME varchar(50) not null,
    CREATED timestamp,
    CONSTRAINT IDX_PERSON_TYPE_ID PRIMARY KEY (ID)
);

CREATE TABLE PERSON (
    ID integer identity primary key,
    FIRST_NAME varchar(50) not null,
    LAST_NAME varchar(50) not null,
    TYPE integer,
```

```
    CREATED timestamp,
    CONSTRAINT IDX_PERSON_ID PRIMARY KEY (ID),
    CONSTRAINT FK_PERSON_TYPE FOREIGN KEY (TYPE) REFERENCES PERSON_TYPE(ID)
);

CREATE TABLE PERSON_STUDENT (
    ID integer not null,
    SCHOOL_NAME varchar(50) not null,
    CREATED timestamp,
    CONSTRAINT FK_PERSON_STUDENT_ID FOREIGN KEY (ID) REFERENCES PERSON(ID) on delete cascade
);

CREATE TABLE PERSON_PROFESSIONAL (
    ID integer not null,
    COMPANY_NAME varchar(50) not null,
    CREATED timestamp,
    CONSTRAINT FK_PERSON_PROFESSIONAL_ID FOREIGN KEY (ID) REFERENCES PERSON(ID) on delete cascade
);

CREATE TABLE ADDRESS (
    ID integer identity primary key,
    PERSON_ID integer,
    ADDRESS varchar(255),
    CITY varchar(50) not null,
    STATE varchar(50) null,
    ZIP_POSTAL varchar(30) not null,
    COUNTRY varchar(50) not null,
    CREATED timestamp,
    CONSTRAINT IDX_ADDRESS_ID PRIMARY KEY (ID),
    CONSTRAINT FK_ADDRESS_PERSON_ID FOREIGN KEY (PERSON_ID) REFERENCES PERSON(ID) on delete cascade
);

...
```

6. Reference

Related Links

- Spring 3.1.x JPA Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-jpa]
- Hibernate [http://www.hibernate.org/]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- One to Many JPA Hibernate Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-jpa
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Java Persistence API 2.0

Spring Data JPA

David Winterfeldt

2012

This shows basic usage of Spring Data JPA [http://www.springsource.org/spring-data/jpa] for some simple queries, and create/update/delete. It has a lot of customization points, including items like query auto-generation by convention, query hints, and auditing. This example builds on the JPA Joined Inheritance example and has the same data structure.

1. Spring Configuration

Instead of using *context:component-scan*, Spring Data JPA [http://www.springsource.org/spring-data/jpa]'s *jpa* namespace is used. The *jpa:repositories* scans for all interfaces that extend `JpaRepository` and creates implementations for use at runtime.

Excerpt from *PersonRepositoryTest-context.xml*

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jpa:repositories base-package="org.springbyexample.orm.repository" />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager"
          p:entityManagerFactory-ref="entityManagerFactory"/>

    <jdbc:embedded-database id="dataSource" type="HSQL">
        <jdbc:script location="classpath:/schema.sql" encoding="UTF-8" />
    </jdbc:embedded-database>

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              p:dataSource-ref="dataSource"
          p:persistenceUnitName="hsql">
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
    </bean>

    ...

</beans>
```

2. JPA Configuration

The *persistence-unit* is named 'hsql'. When supporting tests and a production database, controlling which persistence unit is loaded can be done with Spring Profiles.

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="hsql">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
            <property name="jadira.usertype.autoRegisterUserTypes" value="true" /> ❶
            <property name="jadira.usertype.databaseZone" value="jvm" />
        </properties>
    </persistence-unit>

</persistence>
```

❶ Registering Joda Time [http://joda-time.sourceforge.net/] custom JPA user types.

3. Repository

The repository extends `JpaRepository` and passes the JPA entity and it's primary key being managed. Basic methods for finding a single record, all records, paginated records, create/update, and delete are automatically provided. It's also very easy to overload any custom query to add pagination and sorting.

The `findByFirstNameLike` method let's Spring Data JPA [http://www.springsource.org/spring-data/jpa] automatically generate a like query for the first name column, and `findByLastName` for an equals query for the last name column. The third method, `findByAddress`, creates a custom query using `@Query` and a standard JQL query. `@Param` is used before the method param to create a named parameter. Otherwise it would have created a position based param (ex: '*a.address = ?0*').

Note

Refer to Spring Data JPA [http://www.springsource.org/spring-data/jpa]'s query creation section to see all the different ways the method name can be overloaded to automatically generate queries (And/Or/Between/LessThan/GreaterThan/etc.).

```
public interface PersonRepository extends JpaRepository<Person, Integer> {

    public final static String FIND_BY_ADDRESS_QUERY = "SELECT p " +
                                                       "FROM Person p LEFT JOIN p.addresses a " +
                                                       "WHERE a.address = :address";

    /**
     * Find persons like first name.
     */
    public List<Person> findByFirstNameLike(String firstName);

    /**
     * Find persons by last name.
     */
    public List<Person> findByLastName(String lastName);

    /**
     * Find persons by address.
     */
    @Query(FIND_BY_ADDRESS_QUERY)
    public List<Person> findByAddress(@Param("address") String address);

    ...

}
```

Example 1 `PersonRepository`

The second `findByAddress` query performs the same search as the first one, but a `Pageable` parameter has been added to the method. The page, and number of records for a page can be passed in.

Note

Besides being able to add `Pageable`, there is also a `Sort` parameter that can be added. A `PageRequest`, which implements `Pageable`, can also take sort information. So only a `Pageable` or `Sort` needs to be added to a repository method.

```
public interface PersonRepository extends JpaRepository<Person, Integer> {

    ...

    /**
     * Find paged persons by address.
     */
    @Query(FIND_BY_ADDRESS_QUERY)
    public Page<Person> findByAddress(@Param("address") String address, Pageable page);

    ...

}
```

Example 2 `PersonRepository` *Page Query*

The final query, `findByName`, searches for first and last name using a custom query. But instead of embedding the query in Java it is named query in the *orm.xml*. This may make it easier to manage larger queries.

Note

The query is automatically matched to '*${domainClassName}.${methodName}*', but this can be overridden using the `@Query` *name* attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_2_0.xsd"
                 version="2.0">

    <named-query name="Person.findByName">
        <query>select p from Person p where p.firstName = :firstName AND p.lastName =
:lastName</query>
    </named-query>

</entity-mappings>
```

```
public interface PersonRepository extends JpaRepository<Person, Integer> {

    ...

    /**
     * Find persons by first and last name.
     */
    public List<Person> findByName(@Param("firstName") String firstName, @Param("lastName") String
lastName);

}
```

Example 3 `PersonRepository` Named Query

4. Code Example

```
Person person = personRepository.findOne(FIRST_ID);
```

Example 4 Excerpt from `PersonRepositoryTest` Find by PK

```
Collection<Person> persons = personRepository.findAll();
```

Example 5 Excerpt from `PersonRepositoryTest` Find All

```
List<Person> persons = personRepository.findByFirstNameLike("J%");
```

Example 6 Excerpt from `PersonRepositoryTest` Find by First Name Like

```
List<Person> persons = personRepository.findByLastName(LAST_NAME);
```

Example 7 Excerpt from `PersonRepositoryTest` Find by Last Name

```
List<Person> persons = personRepository.findByAddress(ADDR);
```

Example 8 Excerpt from `PersonRepositoryTest` Find by Address

```
Page<Person> pageResult = personRepository.findByAddress(ADDR, new PageRequest(page, size));
List<Person> persons = pageResult.getContent();
```

Example 9 Excerpt from `PersonRepositoryTest` Paginated Find by Address

```
List<Person> persons = personRepository.findByName(FIRST_NAME, LAST_NAME);
```

Example 10 Excerpt from `PersonRepositoryTest` Find by First & Last Name

Note

In Spring Data JPA [http://www.springsource.org/spring-data/jpa], save and update are both handled by `save` or `saveAndFlush`.

```
Professional person = new Professional();
Set<Address> addresses = new HashSet<Address>();
Address address = new Address();
addresses.add(address);
```

```
address.setAddress(addr);
address.setCity(CITY);
address.setState(STATE);
address.setZipPostal(ZIP_POSTAL);
address.setCountry(COUNTRY);

person.setFirstName(firstName);
person.setLastName(lastName);
person.setCompanyName(companyName);
person.setCreated(new Date());
person.setAddresses(addresses);

Person result = personRepository.saveAndFlush(person);
```

Example 11 Excerpt from `PersonRepositoryTest` Save

```


Person person = personRepository.findOne(FIRST_ID);
testPersonOne(person);

String lastName = "Jones";
person.setLastName(lastName);

personRepository.saveAndFlush(person);
```

Example 12 Excerpt from `PersonRepositoryTest` Update

```


personRepository.delete(FIRST_ID);
```

Example 13 Excerpt from `PersonRepositoryTest` Delete

5. Reference

Related Links

- Spring Data JPA [http://www.springsource.org/spring-data/jpa]
- Spring Data JPA 1.1.x Documentation [http://static.springsource.org/spring-data/data-jpa/docs/1.1.x/reference/html/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/spring-data-jpa
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Data JPA 1.1.x
- Java Persistence API 2.0

Spring Data JPA Auditing

David Winterfeldt

2012

This example shows how to use Spring Data JPA [http://www.springsource.org/spring-data/jpa] auditing. It sets up AOP based auditing for a create and last updated user & date.

1. Spring Configuration

The *jpa:auditing* element activates auditing and needs an instance of `AuditorAware`.

Excerpt from *PersonRepositoryTest-context.xml*

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    ...

    <!-- auditing -->
    <jpa:auditing auditor-aware-ref="auditorAware" />

    <bean id="auditorAware" class="org.springbyexample.orm.entity.AuditorAwareImpl" />

</beans>
```

2. JPA Configuration

The `Auditable` interface uses the Joda Time [http://joda-time.sourceforge.net/] `DateTime` type. Joda Time [http://joda-time.sourceforge.net/]'s default configuration is set using the two properties below.

Excerpt from *META-INF/persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
```

```
             version="2.0">

    <persistence-unit name="hsql">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            ...

            <property name="jadira.usertype.autoRegisterUserTypes" value="true" />
            <property name="jadira.usertype.databaseZone" value="jvm" />
        </properties>
    </persistence-unit>

</persistence>
```

3. Code Example

This is a very simple example that just returns a static value.

Note

If Spring Security [http://www.springsource.org/spring-security/] was setup, the method could return the current user.

```
return SecurityContextHolder.getContext().getAuthentication().getName();
```

```
public class AuditorAwareImpl implements AuditorAware<String> {

    @Override
    public String getCurrentAuditor() {
        return "SYSTEM";
    }


}
```

Example 1 `AuditorAwareImpl`

This auditing class extends Spring Data JPA [http://www.springsource.org/spring-data/jpa]'s `AbstractPersistable`, which has an auto-increment primary key field in it and some utility methods. The `Auditable` interface uses generics to take the user and it's primary key type.

Note

If the reference of `AuditorAware` was setup to be a user entity, then Spring Data JPA

http://www.springsource.org/spring-data/jpa]'s `AbstractAuditable` could be used as the entity base.

Notice the `@MappedSuperclass` annotation. It indicates it is designated to have it's field mappings used by subclasses.

```
@MappedSuperclass
@SuppressWarnings("serial")
public class AbstractAuditableEntity extends AbstractPersistable<Integer> implements
Auditable<String, Integer> {

    private DateTime lastUpdated;
    private String lastUpdateUser;
    private DateTime created;
    private String createUser;

    /**
     * Gets created by audit user.
     */
    @Override
    public String getCreatedBy() {
        return createUser;
    }

    /**
     * Sets created by audit user.
     */
    @Override
    public void setCreatedBy(String createdBy) {
        this.createUser = createdBy;
    }

    /**
     * Gets create audit date.
     */
    @Override
    public DateTime getCreatedDate() {
        return created;
    }

    /**
     * Sets create audit date.
     */
    @Override
    public void setCreatedDate(DateTime creationDate) {
        this.created = creationDate;
    }

    /**
     * Gets last modified by audit user.
     */
    @Override
    public String getLastModifiedBy() {
        return lastUpdateUser;
    }

    /**
     * Sets last modified by audit user.
     */
    @Override
    public void setLastModifiedBy(String lastModifiedBy) {
        this.lastUpdateUser = lastModifiedBy;
    }

    /**
     * Gets last modified audit date.
```

```
     */
    @Override
    public DateTime getLastModifiedDate() {
        return lastUpdated;
    }

    /**
     * Sets last modified audit date.
     */
    @Override
    public void setLastModifiedDate(DateTime lastModifiedDate) {
        this.lastUpdated = lastModifiedDate;
    }

}
```

Example 2 `AbstractAuditableEntity`

The `Person` class extends `AbstractAuditableEntity`. By doing that it gets a primary key and audit fields, along with getters/setters for the inherited fields, `equals`, and `hashCode`. Only the fields that are specific to this type need to be defined.

```


@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.INTEGER)
public class Person extends AbstractAuditableEntity {

    private static final long serialVersionUID = -2175150694352541150L;

    private String firstName = null;
    private String lastName = null;

    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    @JoinColumn(name="PERSON_ID", nullable=false)
    private Set<Address> addresses = null;

    ...

}
```

Example 3 Excerpt from `Person`

4. SQL Script

Excerpt from *schema.sql*

```


...

CREATE TABLE PERSON (
    ID INTEGER generated by default as identity (start with 1) not null,
    FIRST_NAME varchar(50) not null,
    LAST_NAME varchar(50) not null,
    TYPE integer,
        LAST_UPDATED TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
```

```
        LAST_UPDATE_USER VARCHAR(255) DEFAULT 'SYSTEM' NOT NULL,
        CREATED TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
        CREATE_USER VARCHAR(255) DEFAULT 'SYSTEM' NOT NULL,
    CONSTRAINT IDX_PERSON_ID PRIMARY KEY (ID),
    CONSTRAINT FK_PERSON_TYPE FOREIGN KEY (TYPE) REFERENCES PERSON_TYPE(ID)
);

...

CREATE TABLE ADDRESS (
    ID INTEGER generated by default as identity (start with 1) not null,
    PERSON_ID integer,
    ADDRESS varchar(255),
    CITY varchar(50) not null,
    STATE varchar(50) null,
    ZIP_POSTAL varchar(30) not null,
    COUNTRY varchar(50) not null,
    LAST_UPDATED TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    LAST_UPDATE_USER VARCHAR(255) DEFAULT 'SYSTEM' NOT NULL,
    CREATED TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    CREATE_USER VARCHAR(255) DEFAULT 'SYSTEM' NOT NULL,
    CONSTRAINT IDX_ADDRESS_ID PRIMARY KEY (ID),
    CONSTRAINT FK_ADDRESS_PERSON_ID FOREIGN KEY (PERSON_ID) REFERENCES PERSON(ID) on delete cascade
);

...
```

5. Reference

Related Links

- Spring Data JPA Example
- Spring Data JPA [http://www.springsource.org/spring-data/jpa]
- Spring Data JPA 1.1.x Documentation [http://static.springsource.org/spring-data/data-jpa/docs/1.1.x/reference/html/]
- Joda Time [http://joda-time.sourceforge.net/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/spring-data-jpa
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Data JPA 1.1.x
- Java Persistence API 2.0

Hibernate Transaction Annotation Configuration

David Winterfeldt

2008

Creating basic transaction using annotations with Spring for Hibernate.

1. Spring Configuration

To process annotation-based transaction configuration a *transactionManager* bean needs to be created and this will be used by *<tx:annotation-driven/>* for managing transactions.

PersonDaoTransactionTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

    <import resource="classpath:org/springbyexample/orm/hibernate3/shared-context.xml"/>

    <bean id="transactionManager"
          class="org.springframework.orm.hibernate3.HibernateTransactionManager"
          p:sessionFactory-ref="sessionFactory" />

    <tx:annotation-driven/>

    <bean id="personDao"
          class="org.springbyexample.orm.hibernate3.dao.PersonDaoImpl"
          p:sessionFactory-ref="sessionFactory" />

</beans>
```

2. Code Example

The Hibernate `SessionFactory` is used to create a `HibernateTemplate` as it's set. The template is then used for any Hibernate database operations. Spring's `HibernateTemplate` converts all exceptions to runtime exceptions so it isn't necessary to handle any exceptions.

The `@Transactional` annotation configures the class and all it's methods for read only access, but the save method overrides this by specifying it's own annotation of `@Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)`.

```
@Repository
@Transactional(readOnly = true)
public class PersonDaoImpl implements PersonDao {

    protected HibernateTemplate template = null;

    /**
     * Sets Hibernate session factory.
     */
    public void setSessionFactory(SessionFactory sessionFactory) {
        template = new HibernateTemplate(sessionFactory);
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() throws DataAccessException {
        return template.find("from Person");
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) throws DataAccessException {
        return template.find("from Person p where p.lastName = ?", lastName);
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void save(Person person) {
        template.saveOrUpdate(person);
    }

}
```

Example 1 `PersonDaoImpl`

3. Reference

Related Links

- Spring 3.1.x Hibernate Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-hibernate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- One to Many Hibernate XML Configuration

- One to Many Hibernate Annotation Configuration

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-hibernate
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Hibernate 3.6.10.Final

Simple Spring Transactional JUnit 4 Test

David Winterfeldt

2008

Simple Spring Transactional JUnit 4 Test of a Hibernate transaction.

1. Spring Configuration

A *transactionManager* bean is setup for the transactional annotations to use and the `DataSource` from the shared context (not shown) is used by `AbstractTransactionalJUnit4SpringContextTests` to make a `SimpleJdbcTemplate` available.

PersonDaoTransactionUnitTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

    <import resource="classpath:org/springbyexample/orm/hibernate3/shared-context.xml"/>

    <bean id="transactionManager"
          class="org.springframework.orm.hibernate3.HibernateTransactionManager"
          p:sessionFactory-ref="sessionFactory" />

    <tx:annotation-driven/>

    <bean id="personDao"
          class="org.springbyexample.orm.hibernate3.dao.PersonDaoImpl"
          p:sessionFactory-ref="sessionFactory" />

</beans>
```

2. Code Example

The `@RunWith` annotation is part of JUnit 4 and is set to use `SpringJUnit4ClassRunner` to run the unit test. The other annotations, except for `@Test`, are Spring annotations. `@ContextConfiguration` initializes the Spring context and by default looks for a Spring XML file in the same package as the unit test with the file name the same as the class with '-context.xml' as a suffix (ex: PersonDaoTransactionUnitTest-context.xml.). `@TransactionConfiguration` and `@Transactional` configure transactions for the tests.

The method with `@Test` is the main test method which saves a person in a transaction. The method with `@BeforeTransaction` is run before the transaction starts and the method with `@AfterTransaction` is run after the transaction ends. They both use `SimpleJdbcTemplate` to directly check the database and avoid any caching Hibernate might be performing.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TransactionConfiguration
@Transactional
public class PersonDaoTransactionUnitTest extends AbstractTransactionalJUnit4SpringContextTests {

    final Logger logger = LoggerFactory.getLogger(PersonDaoTransactionUnitTest.class);

    protected static int SIZE = 2;
    protected static Integer ID = new Integer(1);
    protected static String FIRST_NAME = "Joe";
    protected static String LAST_NAME = "Smith";
    protected static String CHANGED_LAST_NAME = "Jackson";

    @Autowired
    protected PersonDao personDao = null;

    /**
     * Tests that the size and first record match what is expected
     * before the transaction.
     */
    @BeforeTransaction
    public void beforeTransaction() {
        testPerson(true, LAST_NAME);
    }

    /**
     * Tests person table and changes the first records last name.
     */
    @Test
    public void testHibernateTemplate() throws SQLException {
        assertNotNull("Person DAO is null.", personDao);

        Collection<Person> lPersons = personDao.findPersons();

        assertNotNull("Person list is null.", lPersons);
        assertEquals("Number of persons should be " + SIZE + ".", SIZE, lPersons.size());

        for (Person person : lPersons) {
            assertNotNull("Person is null.", person);

            if (ID.equals(person.getId())) {
                assertEquals("Person first name should be " + FIRST_NAME + ".", FIRST_NAME,
person.getFirstName());
                assertEquals("Person last name should be " + LAST_NAME + ".", LAST_NAME,
person.getLastName());

                person.setLastName(CHANGED_LAST_NAME);

                personDao.save(person);
            }
        }
    }

    /**
     * Tests that the size and first record match what is expected
     * after the transaction.
     */
    @AfterTransaction
    public void afterTransaction() {
        testPerson(false, LAST_NAME);
    }
```

```
    /**
     * Tests person table.
     */
    protected void testPerson(boolean beforeTransaction, String matchLastName) {
        List<Map<String, Object>> lPersonMaps = simpleJdbcTemplate.queryForList("SELECT * FROM
PERSON");

        assertNotNull("Person list is null.", lPersonMaps);
        assertEquals("Number of persons should be " + SIZE + ".", SIZE, lPersonMaps.size());

        Map<String, Object> hPerson = lPersonMaps.get(0);

        logger.debug((beforeTransaction ? "Before" : "After") + " transaction.  " +
hPerson.toString());

        Integer id = (Integer)hPerson.get("ID");
        String firstName = (String)hPerson.get("FIRST_NAME");
        String lastName = (String)hPerson.get("LAST_NAME");

        if (ID.equals(id)) {
            assertEquals("Person first name should be " + FIRST_NAME + ".", FIRST_NAME, firstName);
            assertEquals("Person last name should be " + matchLastName + ".", matchLastName,
lastName);
        }
    }

}
```

Example 1 `PersonDaoTransactionUnitTest`

3. Reference

Related Links

- JUnit [http://www.junit.org/]
- Spring 3.1.x Hibernate Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-hibernate]
- Spring 3.1.x JDBC Custom Namespace [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd persistence/simple-hibernate
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Hibernate 3.6.10.Final

Part IV. Web

Web application examples.

Simple Tiles 2 Spring MVC Webapp

David Winterfeldt

2008

This is a simple example showing how to setup Spring MVC to use Tiles 2. Any request coming in mapped for Tiles processing will attempt to find a Tiles definition that matches the request and then render it.

1. Spring Configuration

The *tilesConfigurer* bean initializes tiles with all the tiles configuration files (more than one can be specified). The *tilesViewResolver* bean defines using Spring's `TilesView` which uses the url to lookup the Tiles definition and render it.

/WEB-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       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/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:view-controller path="/index.html" />
    <mvc:view-controller path="/info/about.html" />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
          p:definitions="/WEB-INF/tiles-defs/templates.xml" />

    <bean id="tilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:viewClass="org.springframework.web.servlet.view.tiles2.TilesView" />

</beans>
```

2. Tiles XML Configuration

The Tiles '.mainTemplate' sets up the default layout and the other two definitions extend the main template and just set their body.

/WEB-INF/tiles-defs/templates.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
```

```
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

    <!-- Default Main Template -->
    <definition name=".mainTemplate" template="/WEB-INF/templates/main.jsp">
        <put-attribute name="title" value="Simple Tiles 2 Example" type="string" />
        <put-attribute name="header" value="/WEB-INF/templates/header.jsp" />
        <put-attribute name="footer" value="/WEB-INF/templates/footer.jsp" />
        <put-attribute name="menu" value="/WEB-INF/templates/menu.jsp" />
        <put-attribute name="body" value="/WEB-INF/templates/blank.jsp" />
    </definition>

    <definition name="index" extends=".mainTemplate">
        <put-attribute name="body" value="/WEB-INF/jsp/index.jsp" />
    </definition>

    <definition name="info/about" extends=".mainTemplate">
        <put-attribute name="body" value="/WEB-INF/jsp/info/about.jsp" />
    </definition>

</tiles-definitions>
```

3. JSP Example

This JSP has the main layout for where the header, footer, menu, and body are located. They are inserted using Tiles custom JSP tags.

/WEB-INF/templates/menu.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>

<html>
<head>
    <title><tiles:getAsString name="title" /></title>
    <link rel="stylesheet" type="text/css" href="<c:url value="/css/main.css"/>" />
</head>
<body>
    <div id="header">
       <div id="headerTitle"><tiles:insertAttribute name="header" /></div>
    </div>
    <div id="menu">
       <tiles:insertAttribute name="menu" />
    </div>
    <div id="content">
       <td><tiles:insertAttribute name="body" />
    </div>
    <div id="footer">
       <tiles:insertAttribute name="footer" />
    </div>
</body>
</html>
```

4. Reference

Related Links

- Spring 3.1.x Tiles Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/view.html#view-tiles]
- Tiles 2 [http://tiles.apache.org/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-tiles2-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Basic Webapp Internationalization

David Winterfeldt

2008

Creating a basic webapp that will handle UTF-8 characters from form input and also have internationalized messages.

1. Web Configuration

The character encoding type for the request can be set using the CharacterEncodingFilter [http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/filter/CharacterEncodingFilter.html]. By setting this, when a form values are retrieved from the request the encoding type will be UTF-8.

/WEB-INF/web.xml

```
<filter>
    <filter-name>encoding-filter</filter-name>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>encoding-filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2. Spring Configuration

This configures the web application for internationalized messages that can then be displayed in a JSP page using the JSTL message format tag. The *basenames* property of `ResourceBundleMessageSource` is set to 'messages' which will then look for the default message resource of messages.properties. Based on different locales, other property files can be defined like messages_es.properties for Spanish.

The `LocaleChangeInterceptor` is configured to look for the parameter name 'locale' to indicate a change of the user's locale, and is registered as an interceptor using the Spring MVC Namespace. The Spring MVC Namespace is new in Spring 3.0. For example, adding 'locale=es' to a URL would change the locale to Spanish. The `SessionLocaleResolver` keeps the user's currently configured locale in session scope.

Excerpt from /WEB-INF/spring/webmvc-context.xml

```
```

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
      p:basenames="messages" />

<!-- Declare the Interceptor -->
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
          p:paramName="locale" />
</mvc:interceptors>

<!-- Declare the Resolver -->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

3. JSP Example

To make sure pages display UTF-8 characters properly, it's important to set the page encoding and content type at the very top of the JSP page. Since this example webapp is using Tiles, it's only necessary to set it at the top of the main template.

Excerpt from /WEB-INF/templates/main.jsp

```
<%@ page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
```

The footer of the Tiles template shows how to display something from the message resource files and also how to create a link to change locales. The *fmt:message* retrieves 'button.locale' from the property file that matches the current locale, or if that key isn't in the current locale's file it will use the default locale's message.

The URL to change the locale uses the *c:url* JSTL tag and redirects to the home page. It sets the parameter to match what was configured on the `LocaleChangeInterceptor` bean's *paramName* property. Which in this case is 'locale'. Passing in no locale switches to the default locale and for Spanish the parameter 'es' is used.

/WEB-INF/templates/footer.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<div align="right">
    <div>
        <fmt:message key="button.locale"/>:
            <c:url var="englishLocaleUrl" value="/index.html">
                <c:param name="locale" value="" />
            </c:url>
            <c:url var="spanishLocaleUrl" value="/index.html">
                <c:param name="locale" value="es" />
            </c:url>

            <a href='<c:out value="${englishLocaleUrl}"/>'><fmt:message key="locale.english"/></a>
            <a href='<c:out value="${spanishLocaleUrl}"/>'><fmt:message key="locale.spanish"/></a>
    </div>

    <div> </div>
```

```
    <div><fmt:message key="site.footer"/></div>
</div>
```

4. Message Resource Property Files

Excerpt from messages.properties

```
button.cancel=Cancel
button.create=Create
button.edit=Edit
button.delete=Delete
button.reset=Reset
button.save=Save
button.search=Search

button.locale=Language
locale.english=English
locale.spanish=Español

site.title=Simple Spring MVC Form Annotation-based Configuration
```

Excerpt from messages_es.properties

```
button.cancel=Cancelar
button.create=Crear
button.edit=Corregir
button.delete=Borrar
button.reset=Restaurar
button.save=Guardar
button.search=Buscar

button.locale=Lenguaje

site.title=Simple Configuración Spring MVC usando Anotaciones
```

5. Reference

Related Links

- Spring 3.1.x Internationalization using `MessageSource` Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#context-functionality-messagesource]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-form-annotation-config-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Simple Spring MVC Form Annotation Configuration Webapp

David Winterfeldt

2008

Simple Spring MVC form using annotation-based configuration. The webapp has basic create, update, delete, and search functionality for a person form. The form basically just has a hidden id (primary key), first name, and last name fields. Tiles is implemented with Spring by Example's Dynamic Tiles Spring MVC Module, Hibernate, and internationalized messages are configured, but this example will focus on explaining the MVC configuration.

1. Web Configuration

Below is the basic web.xml configuration. All Spring contexts in */WEB-INF/spring* and end in **-context.xml* will be loaded into one context. The default context file specific to the *simple-form* servlet is overridden to not load anything (*/WEB-INF/simple-form-servlet.xml* would have been loaded otherwise, the name of of the `DispatcherServlet` plus '-servlet.xml'). The servlet-mapping for 'simple-form' is configured to handle all requests ending in '.html'. The 'encoding-filter' sets all requests to the encoding type of UTF-8.

/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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_3_0.xsd"
         version="3.0" metadata-complete="true">

    <display-name>simple-form</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/*-context.xml
        </param-value>
    </context-param>

    <filter>
        <filter-name>encoding-filter</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>

    <filter-mapping>
```

```
        <filter-name>encoding-filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>simple-form</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>simple-form</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

2. Spring Configuration

This standard Spring MVC configuration file creates handlers, configures Tiles, and also internationalization. The *context:component-scan* registers the `PersonController`, which is defined in the `org.springbyexample.web.servlet.mvc` package. The *mvc:annotation-driven* element registers a `DefaultAnnotationHandlerMapping` and `AnnotationMethodHandlerAdapter`. It also sets up type converters and a Bean Validation (JSR-303) validator (if JSR-303 library is present on the classpath). The *mvc:view-controller* element sets an explicit mapping to the static index page.

The *tilesConfigurer* bean configures tiles and *dynamicTilesViewResolver* takes the url of a request and uses it for the body in a Tiles template. The last three beans configure locale and message resource handling. The `LocaleChangeInterceptor` is registered as an interceptor with the default handler by the surrounding *mvc:interceptors* element.

/WEB-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="org.springbyexample.web.servlet.mvc" />

    <mvc:annotation-driven />
```

```
    <mvc:view-controller path="/index.html" />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
          p:definitions="/WEB-INF/tiles-defs/templates.xml" />

    <bean id="tilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:viewClass="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
          p:basenames="messages" />

    <!-- Declare the Interceptor -->
    <mvc:interceptors>
        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
              p:paramName="locale" />
    </mvc:interceptors>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

3. JSP Example

The *form:form* custom JSP tag is configured to post to the URL '/person/form.html', which based on the `ControllerClassNameHandlerMapping` would map to the `form` method on the `PersonController` configured to receive a post request. The form is bound to the 'person' model, which is the default for the `PersonController`. The *form:hidden* custom JSP tag binds to the `Person`'s id and *form:input* is used to bind 'firstName' and 'lastName'.

/WEB-INF/jsp/person/form.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1><fmt:message key="person.form.title"/></h1>

<c:if test="${not empty statusMessageKey}">
    <p><fmt:message key="${statusMessageKey}"/></p>
</c:if>

<c:url var="url" value="/person/form.html" />
<form:form action="${url}" commandName="person">
    <form:hidden path="id" />

    <fieldset>
        <div class="form-row">
            <label for="firstName"><fmt:message key="person.form.firstName"/>:</label>
            <span class="input"><form:input path="firstName" /></span>
        </div>
        <div class="form-row">
            <label for="lastName"><fmt:message key="person.form.lastName"/>:</label>
            <span class="input"><form:input path="lastName" /></span>
        </div>
```

```
        <div class="form-buttons">
            <div class="button"><input name="submit" type="submit" value="<fmt:message
key="button.save"/>" /></div>
        </div>
    </fieldset>
</form:form>
```

The search displays all records in the person table and generates links to edit and delete a record. The *fmt:message* tag retrieves the local sensitive message based on the message resource key passed in.

/WEB-INF/jsp/person/search.jsp

```


<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1><fmt:message key="person.search.title"/></h1>

<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
    </tr>
<c:forEach var="person" items="${persons}" varStatus="status">
    <tr>
        <c:set var="personFormId" value="person${status.index}"/>

        <c:url var="editUrl" value="/person/form.html">
            <c:param name="id" value="${person.id}" />
        </c:url>
        <c:url var="deleteUrl" value="/person/delete.html"/>
        <form id="${personFormId}" action="${deleteUrl}" method="POST">
            <input id="id" name="id" type="hidden" value="${person.id}"/>
        </form>

            <td>${person.firstName}</td>
        <td>${person.lastName}</td>
            <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <a href="javascript:document.forms['${personFormId}'].submit();"><fmt:message
key="button.delete"/></a>
        </td>
    </tr>
</c:forEach>
</table>


```

4. Code Example

The `@Controller` indicates the class is a Spring MVC controller stereotype which is automatically registered by *context:component-scan* in the *web-application-context.xml*. The `@RequestMapping` annotation on the methods use the *value* attribute to map the method to a path. The *method* attribute is used to indicate the HTTP request type (ex: GET, POST, DELETE). More sophisticated targeting based on available parameters can also be done, but is not needed in this example.

The first method, `newRequest`, is annotated with `@ModelAttribute`. This indicates that the method will be called before every request. In this case it takes the request parameter 'id', but doesn't require the parameter. By default specifying `@RequestParameter` would cause an error if it wasn't available and no other method was available for processing the request. The `newRequest` method looks up the person from the database and returns it or if the 'id' param is null it returns a new instance for the form to bind to. Without specifying a specific name for the model to be bound to, it will be bound to the class name. So `Person` will be bound to 'person'.

The first `form` method handles a create and edit for an HTTP GET request since it's annotated with `@RequestMapping(method=RequestMethod.GET)`. It's just a place holder since the `newRequest` method has already created or retrieved the appropriate bean from the db. By default, Spring will continue forwarding the request where it was headed before it was intercepted by the controller. This could be changed by returning a `String` with the new path or by returning a view instance (like `ModelAndView`.

The second `form` method handles a save from the person form. It will only accept a request that is an HTTP POST and it has the method signature `form(Person person, Model model)`. By specifying the `person` variable, Spring will automatically retrieve or create (depending on it's scope) an instance of `Person` and bind any available request parameters to it. Since it is also the default model object, any values set on it will be available on the page the request forwards to. The `Model` is made available just by specifying it. This can also be done for the `HttpServletRequest` and `HttpServletResponse`. The method sets a create date if one isn't already set, saves the person bean, then returns the saved person instance which replaces the existing model after a success message is set on the model for display on form.

The last two method are `delete` and `search`. The `delete` method is very straight forward. It just deletes person, and then redirects to the search page. The `search` method retrieves all persons and returns them in a `Collection`. It doesn't explicitly set the return value to be bound to the scope 'persons' using the `@ModelAttribute(SEARCH_MODEL_KEY)`.

```
@Controller
public class PersonController {

    private static final String SEARCH_VIEW_KEY = "redirect:search.html";
    private static final String SEARCH_MODEL_KEY = "persons";

    private final PersonRepository repository;

    @Autowired
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    /**
     * For every request for this controller, this will
     * create a person instance for the form.
     */
    @ModelAttribute
    public Person newRequest(@RequestParam(required=false) Integer id) {
        return (id != null ? repository.findOne(id) : new Person());
    }

    /**
     * <p>Person form request.</p>
     *
     * <p>Expected HTTP GET and request '/person/form'.</p>
     */
    @RequestMapping(value="/person/form", method=RequestMethod.GET)
    public void form() {}

    /**
```

```
     * <p>Saves a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/form'.</p>
     */
    @RequestMapping(value="/person/form", method=RequestMethod.POST)
    public Person form(Person person, Model model) {
        if (person.getCreated() == null) {
            person.setCreated(new Date());
        }

        Person result = repository.saveAndFlush(person);

        model.addAttribute("statusMessageKey", "person.form.msg.success");

        return result;
    }

    /**
     * <p>Deletes a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/delete'.</p>
     */
    @RequestMapping(value="/person/delete", method=RequestMethod.POST)
    public String delete(Person person) {
        repository.delete(person);

        return SEARCH_VIEW_KEY;
    }

    /**
     * <p>Searches for all persons and returns them in a
     * <code>Collection</code>.</p>
     *
     * <p>Expected HTTP GET and request '/person/search'.</p>
     */
    @RequestMapping(value="/person/search", method=RequestMethod.GET)
    public @ModelAttribute(SEARCH_MODEL_KEY) Collection<Person> search() {
        return repository.findAll();
    }

}
```

Example 1 `PersonController`

5. Reference

Related Links

- Spring 3.1.x Defining a controller with `@Controller` Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html#mvc-ann-controller]
- Juergen Hoeller's Blog 'Annotated Web MVC Controllers in Spring 2.5' [http://blog.springsource.com/main/2007/11/14/annotated-web-mvc-controllers-in-spring-25/]
- Spring by Example's Dynamic Tiles 2 Spring MVC Module

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-form-annotation-config-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Simple Spring Security Webapp

David Winterfeldt

2008

Simple Spring Security 3.1 example securing a webapp based on the Simple Spring MVC Form Annotation Configuration Webapp. All URLs are restricted to valid users except the login, logoff, and style sheet. Only admins have the ability to delete a record. A non-admin doesn't see the link on the search page to delete a record and also calling the delete method on the service is restricted to admins.

1. Web Configuration

The 'springSecurityFilterChain' filter needs to be configured to intercept all URLs so Spring Security can control access to them. The filter must be named this to match the default bean it retrieves from the Spring context.

/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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_3_0.xsd"
         version="3.0" metadata-complete="true">

    <display-name>simple-security</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/*-context.xml
        </param-value>
    </context-param>

    <!-- Enables Spring Security -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>

    <filter>
        <filter-name>encoding-filter</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>

    <filter-mapping>
```

```
        <filter-name>encoding-filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>simple-form</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>simple-form</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

2. Spring Configuration

The *security:global-method-security* element configures annotation based security so `@Secured` can be used to restrict access to methods.

The *security:http* is set to auto-configure basic HTTP security. Inside the the login, logout, and main style sheet are set to the anonymous role (unrestricted access). The rest of the site is restricted to an authenticated user in the user role. The default login and logout configuration is also customized to use custom pages to maintain the sites look & feel.

The authentication is set to use jdbc based user authentication. Only the `DataSource` needs to be set on the *security:jdbc-user-service* element if the default tables are used. Although other tables can be used by setting custom queries on the element.

/WEB-INF/spring/security-context.xml

```
<?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:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <security:global-method-security secured-annotations="enabled" />

    <security:http auto-config="true">
        <!-- Restrict URLs based on role -->
```

```
        <security:intercept-url pattern="/login*" access="IS_AUTHENTICATED_ANONYMOUSLY" />
        <security:intercept-url pattern="/logoutSuccess*" access="IS_AUTHENTICATED_ANONYMOUSLY" />

        <security:intercept-url pattern="/css/main.css" access="IS_AUTHENTICATED_ANONYMOUSLY" />
        <security:intercept-url pattern="/resources/**" access="IS_AUTHENTICATED_ANONYMOUSLY" />

        <security:intercept-url pattern="/**" access="ROLE_USER" />

        <!-- Override default login and logout pages -->
        <security:form-login login-page="/login.html"
                             login-processing-url="/loginProcess"
                             default-target-url="/index.jsp"
                             authentication-failure-url="/login.html?login_error=1" />
        <security:logout logout-url="/logout" logout-success-url="/logoutSuccess.html" />
    </security:http>

    <security:authentication-manager>
        <security:authentication-provider >
            <security:jdbc-user-service data-source-ref="dataSource" />
        </security:authentication-provider>
    </security:authentication-manager>

</beans>
```

3. JSP Example

The security tag is defined at the top of the page with a prefix of 'sec'. Then around delete link the *sec:authorize* tag is configured to only show the link if the user is in the role 'ROLE_ADMIN'. Now, this doesn't actually stop someone from executing a delete query if they know the URL. Below, in the `PersonService`, the `@Secured` tag is configured to enforce the rule that only an admin can delete a record.

/WEB-INF/jsp/person/search.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<h1><fmt:message key="person.search.title"/></h1>

<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
    </tr>
<c:forEach var="person" items="${persons}" varStatus="status">
    <tr>
        <c:set var="personFormId" value="person${status.index}"/>

        <c:url var="editUrl" value="/person/form.html">
            <c:param name="id" value="${person.id}" />
        </c:url>
        <sec:authorize ifAllGranted="ROLE_ADMIN">
            <c:url var="deleteUrl" value="/person/delete.html"/>
            <form id="${personFormId}" action="${deleteUrl}" method="POST">
                <input id="id" name="id" type="hidden" value="${person.id}"/>
            </form>
        </sec:authorize>

        <td>${person.firstName}</td>
```

```
        <td>${person.lastName}</td>
        <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <sec:authorize ifAllGranted="ROLE_ADMIN">
                <a href="javascript:document.forms['${personFormId}'].submit();"><fmt:message
key="button.delete"/></a>
            </sec:authorize>
        </td>
    </tr>
</c:forEach>
</table>
```

4. Code Example

The delete method has access restricted to users in the admin role by putting the `@Secured` annotation above it and setting the allowed roles. Which in this case is only the 'ROLE_ADMIN' role. By securing the service interface, even if a non-admin user tries to execute the delete URL they will not be able to delete a record.

```
public interface PersonService {

    /**
     * Find person by id.
     */
    public Person findById(Integer id);

    /**
     * Find persons.
     */
    public Collection<Person> find();

    /**
     * Saves person.
     */
    public Person save(Person person);

    /**
     * Deletes person.
     */
    @Secured ({"ROLE_ADMIN"})
    public void delete(Person person);

}
```

Example 1 `PersonService`

You may wonder why the '/delete/person*' URL wasn't restricted. For the current application, this would have been sufficient. But we don't really want to restrict the URL, we want to restrict the actual delete action. Spring Security makes it very easy to restrict access to actual methods. If at some point in the future the another URL is made to also delete a record, our rule will still be enforced. Also, if at some point someone creates a method that calls delete that is accessed by a completely different URL, only admins will be able to execute this new part of the application successfully. A new implementation of the interface will also have the same rule applied to since the security annotation was placed on the interface and not the implementation itself.

5. SQL Script

security_schema.sql

```
SET IGNORECASE TRUE;

CREATE TABLE users (
   username VARCHAR(50) NOT NULL PRIMARY KEY,
   password VARCHAR(50) NOT NULL,
   enabled BIT NOT NULL
);

CREATE TABLE authorities (
   username VARCHAR(50) NOT NULL,
   authority VARCHAR(50) NOT NULL
);
CREATE UNIQUE INDEX ix_auth_username ON authorities (username, authority);

ALTER TABLE authorities ADD CONSTRAINT fk_authorities_users foreign key (username) REFERENCES
users(username);

INSERT INTO users VALUES ('david', 'newyork', true);
INSERT INTO users VALUES ('alex', 'newjersey', true);
INSERT INTO users VALUES ('tim', 'illinois', true);

INSERT INTO authorities VALUES ('david', 'ROLE_USER');
INSERT INTO authorities VALUES ('david', 'ROLE_ADMIN');
INSERT INTO authorities VALUES ('alex', 'ROLE_USER');
INSERT INTO authorities VALUES ('tim', 'ROLE_USER');
```

6. Reference

Related Links

- Spring Security Site [http://static.springsource.org/spring-security/site/]
- Craig Walls' Blog 'Method-Level Security in Spring Security 2.0' [http://www.jroller.com/habuma/entry/method_level_security_in_spring]
- Simple Spring MVC Form Annotation Configuration Webapp

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-spring-security-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Security 3.1.x

Simple Spring Web Flow Webapp

David Winterfeldt

2008

A very simple Spring Web Flow 2.3 example using a flow to create and edit a Person. A Spring MVC annotation-based controller still handles search and deleting records. The example is built on Simple Spring MVC Form Annotation Configuration Webapp and Simple Spring Security Webapp which can be referred to for better explanations of Spring MVC Annotations and Spring Security.

1. Web Configuration

This would be optional, but to use any of the Spring JavaScript the `ResourceServlet` needs to be configured. You can see some basic resources loaded in the excerpt from the master Tiles template.

Excerpt from /WEB-INF/web.xml

```
<!-- Serves static resource content from .jar files such as spring-faces.jar -->
<servlet>
    <servlet-name>resources</servlet-name>
    <servlet-class>org.springframework.js.resource.ResourceServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>

<servlet>
    <servlet-name>simple-form</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value></param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- Map all /resources requests to the Resource Servlet for handling -->
<servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>simple-form</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

2. Spring Configuration

Basic Spring Web Flow configuration with Tiles as the view resolver and the security flow execution listener. The *webflow:flow-registry* element registers the person flow. The person flow XML file is stored with the person form and search page. A flow specific message resources file (messages.properties) could also be put in this location.

/WEB-INF/spring/webflow-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:webflow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd">

    <!-- Enables FlowHandlers -->
    <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter"
          p:flowExecutor-ref="flowExecutor" />

    <!-- Executes flows: the entry point into the Spring Web Flow system -->
    <webflow:flow-executor id="flowExecutor">
        <webflow:flow-execution-listeners>
            <webflow:listener ref="securityFlowExecutionListener" />
        </webflow:flow-execution-listeners>
    </webflow:flow-executor>

    <!-- The registry of executable flow definitions -->
    <webflow:flow-registry id="flowRegistry"
                           flow-builder-services="flowBuilderServices">
        <webflow:flow-location path="/WEB-INF/jsp/person/person.xml" />
    </webflow:flow-registry>

    <!-- Plugs in a custom creator for Web Flow views -->
    <webflow:flow-builder-services id="flowBuilderServices"
view-factory-creator="mvcViewFactoryCreator" />

    <!-- Configures Web Flow to use Tiles to create views for rendering; Tiles allows for applying
consistent layouts to your views -->
    <bean id="mvcViewFactoryCreator"
          class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator"
          p:viewResolvers-ref="tilesViewResolver" />

    <!-- Installs a listener to apply Spring Security authorities -->
    <bean id="securityFlowExecutionListener"
          class="org.springframework.webflow.security.SecurityFlowExecutionListener" />

    <!-- Used in 'create' action-state of Person Flow -->
    <bean id="personBean"
          class="org.springbyexample.web.orm.entity.Person"
          scope="prototype" />

</beans>
```

The handlers are configured so flows and annotation-based controllers can be used together. The url '/person.html' is mapped to the person flow in the *flowMappings* bean and assigned a custom flow handler, which redirects to the search page at the end of the flow and if an exception not handled by the flow occurs.

The *mvc:annotation-driven* configures annotation-based handlers for the controllers. The *mvc:view-controller* element sets explicit mappings to the index page, login page, and logout page. None of which needs to go through a controller for rendering.

The *tilesViewResolver* in the Spring Web Flow example is the `AjaxUrlBasedViewResolver`, which is able to handle rendering fragments of a Tiles context. It's *viewClass* property set to use `FlowAjaxDynamicTilesView`.

This example uses AJAX to populate just the body of the page on a form submit. Also, Spring by Example's Dynamic Tiles Spring MVC Module is used to reduce the Tiles configuration.

/WEB-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="org.springbyexample.web.servlet.mvc" />

    <!-- URL to flow mapping rules -->
    <bean id="flowMappings"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
          p:order="0">
        <property name="mappings">
            <value>/person.html=personFlowHandler</value>
        </property>
    </bean>

    <mvc:annotation-driven />

    <mvc:view-controller path="/index.html" />
    <mvc:view-controller path="/login.html" />
    <mvc:view-controller path="/logoutSuccess.html" />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
          p:definitions="/WEB-INF/tiles-defs/templates.xml" />

    <bean id="tilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:viewClass="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
          p:basenames="messages" />

    <!-- Declare the Interceptor -->
    <mvc:interceptors>
        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
              p:paramName="locale" />
    </mvc:interceptors>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

Custom flow for person handling create and edit. The *decision-state* checks if the id is null and if it is it goes to the 'create' *action-state*, otherwise it goes to the 'editPerson' *action-state*. In 'create' the *personBean* bean, which is scoped as a prototype bean (new instance for each call), is called and the value is put into 'flowScope' under 'person.

The evaluation is performed using the Spring Expression Language (Spring EL), which has been used by Spring Web Flow [http://www.springsource.org/spring-web-flow] since version 2.1. The 'edit' *action-state* uses the Spring Data JPA [http://www.springsource.org/spring-data/jpa] person repository to look the person record based on the id in the edit URL.

Both create and edit forward to the 'personForm' view where the user has a save and cancel button. Both of these buttons are handled using the transition element. The 'save' *transition* saves the person using the person repository. Then both save and cancel populate the latest search results and forward to *end-state* elements that have their view set to the person search page.

The flow is secured to the Spring Security role of 'ROLE_USER'. Which in this case is redundant since the entire webapp is secured to this role, but finer grained rules can make use of this and also it's good to secure the flow since they are reusable components (as subflows).

Person Flow (/WEB-INF/jsp/person/person.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <secured attributes="ROLE_USER" />

    <input name="id" />

    <decision-state id="createOrEdit">
        <if test="id == null" then="create" else="edit" />
    </decision-state>

    <action-state id="create">
        <evaluate expression="personBean" result="flowScope.person" />
        <transition to="personForm" />
    </action-state>

    <action-state id="edit">
        <evaluate expression="personService.findById(id)"
            result="flowScope.person" />
        <transition to="personForm" />
    </action-state>

    <view-state id="personForm" model="person" view="/person/form">
        <transition on="save" to="save">
            <evaluate expression="personService.save(person)" />

            <evaluate expression="personService.find()"
                      result="flowScope.persons" />
        </transition>
        <transition on="cancel" to="cancel" bind="false">
            <evaluate expression="personService.find()"
                      result="flowScope.persons" />
        </transition>
    </view-state>

    <end-state id="save"/>

    <end-state id="cancel"/>

</flow>
```

3. JSP Example

All '/resources' URLs are resolved by Spring JavaScript's the `ResourceServlet` configured in the web.xml.

Excerpt from /WEB-INF/templates/main.jsp

```
<link type="text/css" rel="stylesheet" href="<c:url
value="/resources/dijit/themes/tundra/tundra.css" />" />
<script type="text/javascript" src="<c:url value="/resources/dojo/dojo.js" />"></script>
<script type="text/javascript" src="<c:url value="/resources/spring/Spring.js" />"></script>
<script type="text/javascript" src="<c:url value="/resources/spring/Spring-Dojo.js" />"></script>
```

The create link goes to '/person.html' which is mapped to the person flow. The search link still goes through the `PersonController`. Both the create and search links are secured so when the login and logout pages are rendered, these links aren't displayed until the user logs in.

/WEB-INF/templates/menu.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<div id="side-bar">
    <a href="<c:url value="/"/>">Home</a>

    <sec:authorize ifAllGranted="ROLE_USER">
        <p><fmt:message key="person.form.title"/></p>
            <a href="<c:url value="/person.html"/>"><fmt:message key="button.create"/></a>
            <a href="<c:url value="/person/search.html"/>"><fmt:message key="button.search"/></a>
    </sec:authorize>
</div>
```

The edit link goes to the person flow (ex: '/person.html?id=1'). The flow will look up the person record based on the id request parameter.

/WEB-INF/jsp/person/search.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<h1><fmt:message key="person.search.title"/></h1>

<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
    </tr>
<c:forEach var="person" items="${persons}" varStatus="status">
```

```
    <tr>
        <c:set var="personFormId" value="person${status.index}"/>

        <c:url var="editUrl" value="/person.html">
            <c:param name="id" value="${person.id}" />
        </c:url>

        <sec:authorize ifAllGranted="ROLE_ADMIN">
            <c:url var="deleteUrl" value="/person/delete.html"/>
            <form id="${personFormId}" action="${deleteUrl}" method="POST">
                <input id="id" name="id" type="hidden" value="${person.id}"/>
            </form>
        </sec:authorize>

        <td>${person.firstName}</td>
        <td>${person.lastName}</td>
        <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <sec:authorize ifAllGranted="ROLE_ADMIN">
                <a href="javascript:document.forms['${personFormId}'].submit();"><fmt:message
key="button.delete"/></a>
            </sec:authorize>
        </td>
    </tr>
</c:forEach>
</table>
```

The *form:form* element just needs to have it's *modelAttribute* set to correspond to where the flow put the `Person` instance. The save and cancel buttons specify which event id they are associated with for this step in the flow. This value should match the flow's *transition* element's *on* attribute.

Messages are displayed at the top if there are any status or error messages available. Basic validation is performed by the `PersonValidator`. Which just checks that something has been filled in for the first and last name.

/WEB-INF/jsp/person/form.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<h1><fmt:message key="person.form.title"/></h1>

<div id="messages">
    <c:if test="${not empty statusMessageKey}">
       <p><fmt:message key="${statusMessageKey}"/></p>
    </c:if>

    <spring:hasBindErrors name="person">
        <h2>Errors</h2>
        <div class="formerror">
            <ul>
            <c:forEach var="error" items="${errors.allErrors}">
                <li>${error.defaultMessage}</li>
            </c:forEach>
            </ul>
        </div>
    </spring:hasBindErrors>
</div>

<form:form modelAttribute="person">
```

```
    <form:hidden path="id" />

    <fieldset>
        <div class="form-row">
            <label for="firstName"><fmt:message key="person.form.firstName"/>:</label>
            <span class="input"><form:input path="firstName" /></span>
        </div>
        <div class="form-row">
            <label for="lastName"><fmt:message key="person.form.lastName"/>:</label>
            <span class="input"><form:input path="lastName" /></span>
        </div>
        <div class="form-buttons">
            <div class="button">
                <input type="submit" id="save" name="_eventId_save" value="<fmt:message
key="button.save"/>"/> 
                <input type="submit" name="_eventId_cancel" value="<fmt:message
key="button.cancel"/>"/> 
            </div>
        </div>
    </fieldset>
</form:form>
```

4. Code Example

The person controller still handles delete and search.

```


@Controller
public class PersonController {

    final Logger logger = LoggerFactory.getLogger(getClass());

    static final String SEARCH_VIEW_PATH_KEY = "/person/search";

    private static final String DELETE_PATH_KEY = "/person/delete";

    private static final String SEARCH_VIEW_KEY = "redirect:search.html";
    private static final String SEARCH_MODEL_KEY = "persons";

    private final PersonService service;

    @Autowired
    public PersonController(PersonService service) {
        this.service = service;
    }

    /**
     * <p>Deletes a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/delete'.</p>
     */
    @RequestMapping(value=DELETE_PATH_KEY, method=RequestMethod.POST)
    public String delete(@RequestParam("id") Integer id) {
        logger.info("'{}'  id={}", DELETE_PATH_KEY, id);

        service.delete(id);

        return SEARCH_VIEW_KEY;
    }

    /**
     * <p>Searches for all persons and returns them in a
     * <code>Collection</code>.</p>
```

```
     *
     * <p>Expected HTTP GET and request '/person/search'.</p>
     */
    @RequestMapping(value=SEARCH_VIEW_PATH_KEY, method=RequestMethod.GET)
    public @ModelAttribute(SEARCH_MODEL_KEY) Collection<Person> search() {
        return service.find();
    }

}
```

Example 1 `PersonController`

At the end of the flow and when exception occurs that the flow doesn't handle, the `PersonFlowHandler` redirects to the search page.

```
@Component
public class PersonFlowHandler extends AbstractFlowHandler {

    /**
     * Where the flow should go when it ends.
     */
    @Override
    public String handleExecutionOutcome(FlowExecutionOutcome outcome,
                                         HttpServletRequest request, HttpServletResponse response) {
        return getContextRelativeUrl(PersonController.SEARCH_VIEW_PATH_KEY);
    }

    /**
     * Where to redirect if there is an exception not handled by the flow.
     */
    @Override
    public String handleException(FlowException e,
                                  HttpServletRequest request, HttpServletResponse response) {
        if (e instanceof NoSuchFlowExecutionException) {
            return getContextRelativeUrl(PersonController.SEARCH_VIEW_PATH_KEY);
        } else {
            throw e;
        }
    }

    /**
     * Gets context relative url with an '.html' extension.
     */
    private String getContextRelativeUrl(String view) {
        return "contextRelative:" + view + ".html";
    }

}
```

Example 2 `PersonFlowHandler`

Person Validator that is automatically called by Spring Web Flow based on bean name (${model} + 'Validator') and the method based binding in a *view-state*.

```
@Component
public class PersonValidator {
```

```
    /**
     * Spring Web Flow activated validation (validate + ${state}).
     * Validates 'personForm' view state after binding to person.
     */
    public void validatePersonForm(Person person, MessageContext context) {
        if (!StringUtils.hasText(person.getFirstName())) {
            context.addMessage(new
MessageBuilder().error().source("firstName").code("person.form.firstName.error").build());
        }

        if (!StringUtils.hasText(person.getLastName())) {
            context.addMessage(new
MessageBuilder().error().source("lastName").code("person.form.lastName.error").build());
        }
    }

}
```

Example 3 `PersonValidator`

5. Reference

Related Links

- Spring Web Flow Site [http://www.springsource.org/spring-web-flow]
- Simple Spring MVC Form Annotation Configuration Webapp
- Simple Spring Security Webapp

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-spring-webflow-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Security 3.1.x
- Spring Web Flow 2.3.x

Spring Web Flow Subflow Webapp

David Winterfeldt

2008

A Spring Web Flow 2.0 example using a flow to create and edit a Person and a subflow to create and edit a Person's Addresses. A Spring MVC annotation-based controller still handles search and deleting Person records. The example is built on Simple Spring Web Flow Webapp which can be referred to if necessary.

1. Spring Configuration

This is a basic Spring Web Flow configuration with Tiles as the view resolver and with a Spring Security flow execution listener configured. The *webflow:flow-registry* element registers the person flow and address subflow. The person flow XML file is stored with the person form and search page, and the address flow is stored with the address form page. A flow specific message resources file (messages.properties) could also be put in these locations.

/WEB-INF/spring/webflow-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:webflow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd">

    <!-- Enables FlowHandlers -->
    <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter"
          p:flowExecutor-ref="flowExecutor" />

    <!-- Executes flows: the entry point into the Spring Web Flow system -->
    <webflow:flow-executor id="flowExecutor">
        <webflow:flow-execution-listeners>
            <webflow:listener ref="securityFlowExecutionListener" />
        </webflow:flow-execution-listeners>
    </webflow:flow-executor>

    <!-- The registry of executable flow definitions -->
    <webflow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
        <webflow:flow-location path="/WEB-INF/jsp/person/person.xml" />
        <webflow:flow-location path="/WEB-INF/jsp/address/address.xml" />
    </webflow:flow-registry>

    <!-- Plugs in a custom creator for Web Flow views -->
    <webflow:flow-builder-services id="flowBuilderServices"
view-factory-creator="mvcViewFactoryCreator" />

    <!-- Configures Web Flow to use Tiles to create views for rendering; Tiles allows for applying
consistent layouts to your views -->
    <bean id="mvcViewFactoryCreator"
          class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator"
          p:viewResolvers-ref="tilesViewResolver" />

    <!-- Installs a listener to apply Spring Security authorities -->
    <bean id="securityFlowExecutionListener"
          class="org.springframework.webflow.security.SecurityFlowExecutionListener" />
```

```
    <!-- Used in 'create' action-state of Person Flow -->
    <bean id="personBean"
          class="org.springbyexample.web.orm.entity.Person"
          scope="prototype" />

    <!-- Used in 'create' action-state of Address Flow -->
    <bean id="addressBean"
          class="org.springbyexample.web.orm.entity.Address"
          scope="prototype" />

</beans>
```

The handlers are configured so flows and annotation-based controllers can be used together. The url '/person.html' is mapped to the person flow in the *flowMappings* bean and assigned a custom flow handler, which redirects to the search page at the end of the flow and if an exception not handled by the flow occurs.

The *tilesViewResolver* in the Spring Web Flow example is the `AjaxUrlBasedViewResolver`, which is able to handle rendering fragments of a Tiles context. It's *viewClass* property is set to use `FlowAjaxDynamicTilesView`. This example uses AJAX to populate just the body of the page on a form submit. Also, Spring by Example's Dynamic Tiles Spring MVC Module is used to reduce the Tiles configuration.

/WEB-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="org.springbyexample.web.servlet.mvc" />

    <!-- URL to flow mapping rules -->
    <bean id="flowMappings"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
          p:order="0">
        <property name="mappings">
            <value>/person.html=personFlowHandler</value>
        </property>
    </bean>

    <mvc:annotation-driven />

    <mvc:view-controller path="/index.html" />
    <mvc:view-controller path="/login.html" />
    <mvc:view-controller path="/logoutSuccess.html" />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
          p:definitions="/WEB-INF/tiles-defs/templates.xml" />

    <bean id="tilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:viewClass="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView"
```

```
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
          p:basenames="messages" />

    <!-- Declare the Interceptor -->
    <mvc:interceptors>
        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
              p:paramName="locale" />
    </mvc:interceptors>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

Custom flow for person handling create and edit. The *decision-state* checks if the id is null and if it is it goes to the 'createPerson' *action-state*, otherwise it goes to the 'editPerson' *action-state*. In 'createPerson' the prototype scoped *personBean* bean is put into 'flowScope' under 'person'. The evaluation is performed using the Spring Expression Language (Spring EL), which has been used by Spring Web Flow [http://www.springsource.org/spring-web-flow] since version 2.1. The 'editPerson' *action-state* uses the Spring Data JPA [http://www.springsource.org/spring-data/jpa] person repository to look the person record based on the id in the edit URL.

Both create and edit forward to the 'personForm' view where the user has a save and cancel button. Both of these buttons are handled using the *transition* element. The 'save' transition saves the person using the person repository, and puts the result of the save into 'flowScope.person' along with the person id (in case of a create). The success message key is put into 'flashScope' (scope available until next view render), and then goes back to the person form. The cancel populates the latest search results and forward to end-state elements that have their view set to the person search page.

The 'personForm' view also has transitions for handling adding, editing, and deleting addresses. Adding and editing are passed to the 'address' subflow-state. The delete is handled by an evaluate element calling person repository. Both the address id and the person instance are passed into the address subflow. The person instance is used by an edit to retrieve the address to edit instead of looking it up from that database since it's already in scope. At the end of the address flow the address instance is output from the subflow and saved by the person flow's 'address' *subflow-state*.

Both flows are secured to the Spring Security role of 'ROLE_USER'. Which in this case is redundant since the entire webapp is secured to this role, but finer grained rules can make use of this and also it's good to secure the flow since they are reusable components (as subflows). The subflow could have only allowed only admins to access the address subflow.

Person Flow (/WEB-INF/jsp/person/person.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <secured attributes="ROLE_USER" />
```

```
    <input name="id" />

    <decision-state id="createOrEdit">
        <if test="id == null" then="create" else="edit" />
    </decision-state>

    <action-state id="create">
        <evaluate expression="personBean" result="flowScope.person" />
        <transition to="personForm" />
    </action-state>

    <action-state id="edit">
        <evaluate expression="personService.findById(id)" result="flowScope.person" />
        <transition to="personForm" />
    </action-state>

    <view-state id="personForm" model="person" view="/person/form">
        <transition on="addAddress" to="address" bind="false">
            <set name="flashScope.addressId" value="''" />
        </transition>
        <transition on="editAddress" to="address">
            <set name="flashScope.addressId" value="requestParameters.addressId" />
        </transition>
        <transition on="deleteAddress" to="personForm">
            <evaluate expression="personService.deleteAddress(id, requestParameters.addressId)"
result="flowScope.person" />
        </transition>

        <transition on="save" to="personForm">
            <evaluate expression="personService.save(person)" result="flowScope.person" />

            <set name="flowScope.id" value="person.id" />

            <set name="flashScope.statusMessageKey" value="'person.form.msg.success'" />

            <render fragments="content" />
        </transition>
        <transition on="cancel" to="cancel" bind="false">
            <evaluate expression="personService.find()" result="flowScope.persons" />
        </transition>
    </view-state>

    <subflow-state id="address" subflow="address">
        <input name="id" value="addressId"/>
        <input name="person" value="person"/>

        <output name="address" />

        <transition on="saveAddress" to="personForm">
            <evaluate expression="personService.saveAddress(id, address)" result="flowScope.person"
/>

            <set name="flashScope.statusMessageKey" value="'address.form.msg.success'" />
        </transition>
        <transition on="cancelAddress" to="personForm" />
    </subflow-state>

    <end-state id="personConfirmed" />

    <end-state id="cancel" />

</flow>
```

The flow is very similar to the person flow. The *decision-state* handles a create or an edit based on whether or not an id is passed into the flow. Then the *action-state* for the create put a new `Address` instance into scope and the edit

gets it from the person instance. The save outputs the address instance and let's the parent flow handle saves.

Address Flow (/WEB-INF/jsp/address/address.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <secured attributes="ROLE_USER" />

    <input name="id" />
    <input name="person" />

    <decision-state id="createOrEdit">
        <if test="id == ''" then="createAddress" else="editAddress" />
    </decision-state>

    <action-state id="createAddress">
        <evaluate expression="addressBean" result="flowScope.address" />
        <transition to="addressForm" />
    </action-state>

    <action-state id="editAddress">
        <evaluate expression="person.findAddressById(id)" result="flowScope.address" />
        <transition to="addressForm" />
    </action-state>

    <view-state id="addressForm" model="address" view="/address/form">
        <transition on="save" to="saveAddress" />
        <transition on="cancel" to="cancelAddress" bind="false" />
    </view-state>

    <end-state id="saveAddress">
        <output name="address" value="address"/>
    </end-state>

    <end-state id="cancelAddress" />

</flow>
```

2. JSP Example

The create link goes to '/person.html' which is mapped to the person flow. The search link still goes through the `PersonController`.

/WEB-INF/templates/menu.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<div id="side-bar">
    <a href="<c:url value="/"/>">Home</a>

    <p><fmt:message key="person.form.title"/></p>
        <a href="<c:url value="/person.html"/>"><fmt:message key="button.create"/></a>
        <a href="<c:url value="/person/search.html"/>"><fmt:message key="button.search"/></a>
</div>
```

The edit link goes to the person flow (ex: '/person.html?id=1'). The flow will look up the person record based on the id request parameter.

/WEB-INF/jsp/person/search.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<h1><fmt:message key="person.search.title"/></h1>

<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
    </tr>
<c:forEach var="person" items="${persons}">
    <tr>
        <c:url var="editUrl" value="/person.html">
            <c:param name="id" value="${person.id}" />
        </c:url>
        <c:url var="deleteUrl" value="/person/delete.html">
            <c:param name="id" value="${person.id}" />
        </c:url>

       <td>${person.firstName}</td>
        <td>${person.lastName}</td>
       <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <sec:authorize ifAllGranted="ROLE_ADMIN">
                <a href='<c:out value="${deleteUrl}"/>'><fmt:message key="button.delete"/></a>
            </sec:authorize>
        </td>
    </tr>
</c:forEach>
</table>
```

The *form:form* element just needs to have it's *modelAttribute* set to correspond to where the flow put the `Person` instance. The save and cancel buttons specify which event id they are associated with for this step in the flow. This value should match the flow's transition element's on attribute.

The link to add an address goes to the address subflow. If there are any addresses, they are displayed in a table with an edit next to each record. There is also a delete link if the user is an admin.

The top of the page has a 'messages' div which can display a status message key or any Spring bind errors. This is where Spring Web Flow will store any errors for automatic validation performed based on a method on the model instance or a bean. In this case, there is a *personValidator* bean that validates if the first and last name are not blank.

/WEB-INF/jsp/person/form.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<h1><fmt:message key="person.form.title"/></h1>

<div id="messages">
    <c:if test="${not empty statusMessageKey}">
       <p><fmt:message key="${statusMessageKey}"/></p>
    </c:if>

    <spring:hasBindErrors name="person">
        <h2>Errors</h2>
        <div class="formerror">
            <ul>
            <c:forEach var="error" items="${errors.allErrors}">
                <li>${error.defaultMessage}</li>
            </c:forEach>
            </ul>
        </div>
    </spring:hasBindErrors>
</div>

<form:form modelAttribute="person">
    <form:hidden path="id" />

    <fieldset>
        <div class="form-row">
            <label for="firstName"><fmt:message key="person.form.firstName"/>:</label>
            <span class="input"><form:input path="firstName" /></span>
        </div>
        <div class="form-row">
            <label for="lastName"><fmt:message key="person.form.lastName"/>:</label>
            <span class="input"><form:input path="lastName" /></span>
        </div>
        <div class="form-buttons">
            <div class="button">
                <input type="submit" id="save" name="_eventId_save" value="<fmt:message
key="button.save"/>"/> 
                <input type="submit" name="_eventId_cancel" value="Cancel"/> 
            </div>
        </div>
    </fieldset>
</form:form>

<c:if test="${not empty person.id}">
<div style="clear: both;float:left;">
<div>
<a href="${flowExecutionUrl}&_eventId=addAddress" ><fmt:message key="address.form.button.add"/></a>
</div>
</c:if>

<c:if test="${empty person.addresses}">
    <p> </p>
</c:if>

<c:if test="${not empty person.addresses}">
<table class="search">
    <tr>
        <th><fmt:message key="address.form.address"/></th>
        <th><fmt:message key="address.form.city"/></th>
        <th><fmt:message key="address.form.state"/></th>
        <th><fmt:message key="address.form.zipPostal"/></th>
        <th><fmt:message key="address.form.country"/></th>
    </tr>
<c:forEach var="address" items="${person.addresses}">
    <tr>
            <td>${address.address}</td>
            <td>${address.city}</td>
            <td>${address.state}</td>
```

```
            <td>${address.zipPostal}</td>
            <td>${address.country}</td>
            <td>
                <a href="${flowExecutionUrl}&_eventId=editAddress&addressId=${address.id}"
><fmt:message key="button.edit"/></a>
                <sec:authorize ifAllGranted="ROLE_ADMIN">
                    <a href="${flowExecutionUrl}&_eventId=deleteAddress&addressId=${address.id}"
><fmt:message key="button.delete"/></a>
                </sec:authorize>
            </td>
    </tr>
</c:forEach>
</table>
</c:if>

</div>
```

The 'messages' div at the top of the page can display a status message or any Spring bind errors. The validation for address is in a method in the `Address` class. The method in this case is `validateAddressForm(MessageContext context)`. It is for the 'addressForm' *view-state* so the method name should be 'validate' + ${viewStateId} and take a `MessageContext` as a parameter.

/WEB-INF/jsp/address/form.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1><fmt:message key="address.form.title"/></h1>

<div id="messages">
    <c:if test="${not empty statusMessageKey}">
       <p><fmt:message key="${statusMessageKey}"/></p>
    </c:if>

    <spring:hasBindErrors name="address">
        <h2>Errors</h2>
        <div class="formerror">
            <ul>
            <c:forEach var="error" items="${errors.allErrors}">
                <li>${error.defaultMessage}</li>
            </c:forEach>
            </ul>
        </div>
    </spring:hasBindErrors>
</div>

<form:form modelAttribute="address">
    <form:hidden path="id" />

    <fieldset>
        <div class="form-row">
            <label for="address"><fmt:message key="address.form.address"/>:</label>
            <span class="input"><form:input path="address" /></span>
        </div>
        <div class="form-row">
            <label for="city"><fmt:message key="address.form.city"/>:</label>
            <span class="input"><form:input path="city" /></span>
        </div>
        <div class="form-row">
            <label for="state"><fmt:message key="address.form.state"/>:</label>
```

```
            <span class="input"><form:input path="state" /></span>
        </div>
        <div class="form-row">
            <label for="zipPostal"><fmt:message key="address.form.zipPostal"/>:</label>
            <span class="input"><form:input path="zipPostal" /></span>
        </div>
        <div class="form-row">
            <label for="country"><fmt:message key="address.form.country"/>:</label>
            <span class="input"><form:input path="country" /></span>
        </div>
        <div class="form-buttons">
            <div class="button">
                <input type="submit" id="save" name="_eventId_save" value="<fmt:message
key="button.save"/>"/> 
                <input type="submit" name="_eventId_cancel" value="Cancel"/> 
            </div>
        </div>
    </fieldset>
</form:form>
```

3. Code Example

The person controller still handles delete and search.

```


@Controller
public class PersonController {

    final Logger logger = LoggerFactory.getLogger(getClass());

    static final String SEARCH_VIEW_PATH_KEY = "/person/search";

    private static final String DELETE_PATH_KEY = "/person/delete";

    private static final String SEARCH_VIEW_KEY = "redirect:search.html";
    private static final String SEARCH_MODEL_KEY = "persons";

    private final PersonService service;

    @Autowired
    public PersonController(PersonService service) {
        this.service = service;
    }

    /**
     * <p>Deletes a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/delete'.</p>
     */
    @RequestMapping(value=DELETE_PATH_KEY, method=RequestMethod.POST)
    public String delete(@RequestParam("id") Integer id) {
        logger.info("'{}'  id={}", DELETE_PATH_KEY, id);

        service.delete(id);

        return SEARCH_VIEW_KEY;
    }

    /**
     * <p>Searches for all persons and returns them in a
     * <code>Collection</code>.</p>
     *
     * <p>Expected HTTP GET and request '/person/search'.</p>
```

```
     */
    @RequestMapping(value=SEARCH_VIEW_PATH_KEY, method=RequestMethod.GET)
    public @ModelAttribute(SEARCH_MODEL_KEY) Collection<Person> search() {
        return service.find();
    }

}
```

Example 1 `PersonController`

At the end of the flow and when exception occurs that the flow doesn't handle, the `PersonFlowHandler` redirects to the search page.

```


@Component
public class PersonFlowHandler extends AbstractFlowHandler {

    /**
     * Where the flow should go when it ends.
     */
    @Override
    public String handleExecutionOutcome(FlowExecutionOutcome outcome,
                                         HttpServletRequest request, HttpServletResponse response) {
        return getContextRelativeUrl(PersonController.SEARCH_VIEW_PATH_KEY);
    }

    /**
     * Where to redirect if there is an exception not handled by the flow.
     */
    @Override
    public String handleException(FlowException e,
                                  HttpServletRequest request, HttpServletResponse response) {
        if (e instanceof NoSuchFlowExecutionException) {
            return getContextRelativeUrl(PersonController.SEARCH_VIEW_PATH_KEY);
        } else {
            throw e;
        }
    }

    /**
     * Gets context relative url with an '.html' extension.
     */
    private String getContextRelativeUrl(String view) {
        return "contextRelative:" + view + ".html";
    }

}
```

Example 2 `PersonFlowHandler`

Person Validator that is automatically called by Spring Web Flow based on bean name (${model} + 'Validator') and the method based binding in a *view-state*.

```


@Component
public class PersonValidator {

    /**
```

```
     * Spring Web Flow activated validation (validate + ${state}).
     * Validates 'personForm' view state after binding to person.
     */
    public void validatePersonForm(Person person, MessageContext context) {
        if (!StringUtils.hasText(person.getFirstName())) {
            context.addMessage(new
MessageBuilder().error().source("firstName").code("person.form.firstName.error").build());
        }

        if (!StringUtils.hasText(person.getLastName())) {
            context.addMessage(new
MessageBuilder().error().source("lastName").code("person.form.lastName.error").build());
        }
    }

}
```

Example 3 `PersonValidator`

```


/**
 * Validates 'addressForm' view state after binding to address.
 * Spring Web Flow activated validation ('validate' + ${state}).
 */
public void validateAddressForm(MessageContext context) {
    if (!StringUtils.hasText(address)) {
        context.addMessage(new
MessageBuilder().error().source("address").code("address.form.address.error").build());
    }

    if (!StringUtils.hasText(city)) {
        context.addMessage(new
MessageBuilder().error().source("city").code("address.form.city.error").build());
    }

    if (!StringUtils.hasText(state)) {
        context.addMessage(new
MessageBuilder().error().source("state").code("address.form.state.error").build());
    }

    if (!StringUtils.hasText(zipPostal)) {
        context.addMessage(new
MessageBuilder().error().source("zipPostal").code("address.form.zipPostal.error").build());
    }

    if (!StringUtils.hasText(country)) {
        context.addMessage(new
MessageBuilder().error().source("country").code("address.form.country.error").build());
    }
}
```

Example 4 Excerpt from `Address`

4. Reference

Related Links

- Spring Web Flow Site [http://www.springsource.org/spring-web-flow]
- Simple Spring Web Flow Webapp

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/spring-webflow-subflow-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Security 3.1.x
- Spring Web Flow 2.3.x

Simple Grails Webapp

David Winterfeldt

2009

This is a simple Grails [http://www.grails.org/] web application with a basic form to create and edit a person and addresses.

Note

If Grails [http://www.grails.org/] isn't installed, download it from the Grails Download [http://www.grails.org/Download] page and follow the Installation Instructions [http://www.grails.org/Installation].

1. Create Application

Create the Grails application, domain classes, and controller classes using Grails commands.

Create the application by running the **grails create-app** command.

```
$ grails create-app simple-grails-webapp

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: ...
Running script /usr/local/grails/scripts/CreateApp_.groovy
Environment set to development
    [mkdir] Created dir: /simple-grails-webapp/src
    [mkdir] Created dir: /simple-grails-webapp/src/java
    [mkdir] Created dir: /simple-grails-webapp/src/groovy
    [mkdir] Created dir: /simple-grails-webapp/grails-app
    [mkdir] Created dir: /simple-grails-webapp/grails-app/controllers
    [mkdir] Created dir: /simple-grails-webapp/grails-app/services
    [mkdir] Created dir: /simple-grails-webapp/grails-app/domain
    [mkdir] Created dir: /simple-grails-webapp/grails-app/taglib
    [mkdir] Created dir: /simple-grails-webapp/grails-app/utils
    [mkdir] Created dir: /simple-grails-webapp/grails-app/views
    [mkdir] Created dir: /simple-grails-webapp/grails-app/views/layouts
    [mkdir] Created dir: /simple-grails-webapp/grails-app/i18n
    [mkdir] Created dir: /simple-grails-webapp/grails-app/conf
    [mkdir] Created dir: /simple-grails-webapp/test
    [mkdir] Created dir: /simple-grails-webapp/test/unit
    [mkdir] Created dir: /simple-grails-webapp/test/integration
    [mkdir] Created dir: /simple-grails-webapp/scripts
    [mkdir] Created dir: /simple-grails-webapp/web-app
    [mkdir] Created dir: /simple-grails-webapp/web-app/js
    [mkdir] Created dir: /simple-grails-webapp/web-app/css
    [mkdir] Created dir: /simple-grails-webapp/web-app/images
    [mkdir] Created dir: /simple-grails-webapp/web-app/META-INF
    [mkdir] Created dir: /simple-grails-webapp/lib
    [mkdir] Created dir: /simple-grails-webapp/grails-app/conf/spring
    [mkdir] Created dir: /simple-grails-webapp/grails-app/conf/hibernate
[propertyfile] Creating new property file: /simple-grails-webapp/application.properties
     [copy] Copying 1 resource to /simple-grails-webapp
    [unjar] Expanding: /simple-grails-webapp/grails-shared-files.jar into /simple-grails-webapp
```

```
   [delete] Deleting: /simple-grails-webapp/grails-shared-files.jar
     [copy] Copying 1 resource to /simple-grails-webapp
    [unjar] Expanding: /simple-grails-webapp/grails-app-files.jar into /simple-grails-webapp
   [delete] Deleting: /simple-grails-webapp/grails-app-files.jar
     [move] Moving 1 file to /simple-grails-webapp
     [move] Moving 1 file to /simple-grails-webapp
     [move] Moving 1 file to /simple-grails-webapp
     [copy] Copying 1 file to ~/.grails/1.1.2/plugins
Installing plug-in hibernate-1.1.2
    [mkdir] Created dir: ~/.grails/1.1.2/projects/simple-grails-webapp/plugins/hibernate-1.1.2
    [unzip] Expanding: ~/.grails/1.1.2/plugins/grails-hibernate-1.1.2.zip into
~/.grails/1.1.2/projects/simple-grails-webapp/plugins/hibernate-1.1.2
Executing hibernate-1.1.2 plugin post-install script ...
Plugin hibernate-1.1.2 installed
Created Grails Application at /simple-grails-webapp
```

Move to the directory of the newly created project and create the `Person` domain class using **grails create-domain-class**.

```
$ cd simple-grails-webapp
```

```
$ grails create-domain-class Person

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: /simple-grails-webapp
Running script /usr/local/grails/scripts/CreateDomainClass.groovy
Environment set to development
Created DomainClass for Person
Created Tests for Person
```

Create the `Address` domain class.

```
$ grails create-domain-class Address

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: /simple-grails-webapp
Running script /usr/local/grails/scripts/CreateDomainClass.groovy
Environment set to development
Created DomainClass for Address
Created Tests for Address
```

Create the `PersonController` using the **grails create-controller** command.

```
$ grails create-controller Person

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: /simple-grails-webapp
```

```
Running script /usr/local/grails/scripts/CreateController.groovy
Environment set to development
Created Controller for Person
    [mkdir] Created dir: /simple-grails-webapp/grails-app/views/person
Created Tests for Person
```

Create the `AddressController`.

```
$ grails create-controller Address

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: /simple-grails-webapp
Running script /usr/local/grails/scripts/CreateController.groovy
Environment set to development
Created Controller for Address
    [mkdir] Created dir: /simple-grails-webapp/grails-app/views/address
Created Tests for Address
```

2. Modify Code

Edit the generated Grails domain and controller classes. This will add fields and validation rules to the domain classes, and activate scaffolding in the controller classes. Grails' scaffolding automatically generates a controller and JSP pages that work with the associatd domain class.

First add a first name, last name, created, and list of addresses to `Person`. By assigning the `hasMany` field to `[addresses:Address]`, this creates a relationship to `Address` with an `addresses` list variable. The `constraints` section is where validation rules can be set. All the fields are set to not allow `null` values.

Edit *grails-app/domain/Person.groovy* to match the code below.

```


class Person {

        String firstName
        String lastName
        List<Address> addresses
        Date created

        static hasMany = [addresses:Address]

    static constraints = {
        firstName(nullable:false)
        lastName(nullable:false)
        created(nullable:false)
    }

}


```

Example 1 `Person`

Next, add the basic fields to `Address` and define that it is the many part of a one-to-many relationship to `Person` by assigning the `belongsTo` field to `Person`.

Edit *grails-app/domain/Address.groovy* to match the code below.

```
class Address {

    String address
    String city
    String state
    String zipPostal
    String country
    Date created

    static belongsTo = Person

    static constraints = {
        address(nullable:false)
        city(nullable:false)
        state(nullable:false)
        zipPostal(nullable:false)
        country(nullable:false)
        created(nullable:false)
    }

}
```

Example 2 `Address`

Now that the domain objects are setup, the controllers will be set to use Grails' scaffolding. This will autogenerate the controller interface and basic JSP pages. The `scaffold` variable just needs to be set to `true` and Grails will handle everything else. The `PersonController` is automatically associated with the `Person` domain class based on default naming conventions.

Edit *grails-app/controllers/PersonController.groovy* to match the code below.

```
class PersonController {

   def scaffold = true

}
```

Example 3 `PersonController`

Activate scaffolding for the `AddressController`.

Edit *grails-app/controllers/AddressController.groovy* to match the code below.

```
class AddressController {

   def scaffold = true
```

```
}
```

Example 4 `AddressController`

3. Run Application

Start the application by running the **grails run-app** command.

```
$ grails run-app

Welcome to Grails 1.1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /usr/local/grails

Base Directory: /simple-grails-webapp
Running script /usr/local/grails/scripts/RunApp.groovy
Environment set to development
    [mkdir] Created dir: ~/.grails/1.1.2/projects/simple-grails-webapp/classes
  [groovyc] Compiling 10 source files to ~/.grails/1.1.2/projects/simple-grails-webapp/classes
    [mkdir] Created dir: ~/.grails/1.1.2/projects/simple-grails-webapp/resources/grails-app/i18n
[native2ascii] Converting 11 files from /simple-grails-webapp/grails-app/i18n to
~/.grails/1.1.2/projects/simple-grails-webapp/resources/grails-app/i18n
     [copy] Copying 1 file to ~/.grails/1.1.2/projects/simple-grails-webapp/classes
     [copy] Copied 2 empty directories to 2 empty directories under
~/.grails/1.1.2/projects/simple-grails-webapp/resources
Running Grails application..
Server running. Browse to http://localhost:8080/simple-grails-webapp
```

Navigating to the application's home page, there will be a link to the person and address controllers. After creating and address, from the person page the address can be associated with the person. The application is very basic, but it's functional and was made very quickly.

4. Reference

Related Links

- Grails [http://www.grails.org/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd web/simple-grails-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Grails 1.1.x

Simple Flex Webapp

David Winterfeldt

2009

The simple Flex [http://www.adobe.com/products/flex/] webapp is based on the Simple Spring MVC Form Annotation Configuration Webapp. It's the same except the HTML results on the search page have been replaced with a Flex [http://www.adobe.com/products/flex/] application and services for Flex [http://www.adobe.com/products/flex/] were configured on the server using Spring BlazeDS Integration [http://www.springsource.org/spring-flex] and Adobe BlazeDS.

Adobe Flex [http://www.adobe.com/products/flex/] can run in a Flash plugin available for most browsers and operating systems. Applications can also be deployed as desktop application using Adobe AIR [http://www.adobe.com/products/air/]. Adobe BlazeDS is an open source project for integrating server side Java with a Flex [http://www.adobe.com/products/flex/] client for remoting and messaging. Spring BlazeDS Integration [http://www.springsource.org/spring-flex] provides reduced configuration and ease of use on top of Adobe BlazeDS.

Note

The project is configured with Flex Builder (Eclipse plugin) and also can be run from the command line.

```
$ mvn jetty:run
```

1. Web Configuration

This is a standard Spring web application configuration except a `DispatcherServlet` specifically for handling Flex [http://www.adobe.com/products/flex/] remoting requests has been added and the Spring config specified will configure Spring BlazeDS Integration [http://www.springsource.org/spring-flex].

/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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_3_0.xsd"
         version="3.0" metadata-complete="true">

    <display-name>simple-form</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/*-context.xml
```

```
        </param-value>
    </context-param>

    <filter>
        <filter-name>encoding-filter</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>encoding-filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>simple-form</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value></param-value>
                </init-param>
    </servlet>

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

    <servlet-mapping>
        <servlet-name>simple-form</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>spring-flex</servlet-name>
        <url-pattern>/spring/*</url-pattern> ❷
    </servlet-mapping>


    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

❶ Spring BlazeDS Integration servlet.
❷ Mapping Spring BlazeDS Integration servlet to handle all requests to '/spring/*'.

2. Spring Configuration

Setting up the Spring BlazeDS Integration [http://www.springsource.org/spring-flex] configuration is very simple. The *flex:message-broker* initializes BlazeDS and it's configuration files, which by default are expected in

'/WEB-INF/flex' and for the main configuration to be called 'services-config.xml'. Two services are loaded, 'personRepository' and 'personService'. It's not best practice to expose the DAO directly as a service, but for this example it was done to illustrate exposing a remoting service using annotations and the custom namespace. One from the *context:component-scan* and the other is exposed using the Spring BlazeDS Integration [http://www.springsource.org/spring-flex] custom namespace. The `PersonService` exposed through scanning will be shown later in the Code Example section, and the JPA Person DAO is exposed using *flex:remoting-destination*. It's created just by referencing the Person DAO. To have a remoting destination name other than the bean name the *destination-id* could specify something different.

/WEB-INF/spring/flex/flex-servlet-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:flex="http://www.springframework.org/schema/flex"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/flex
                           http://www.springframework.org/schema/flex/spring-flex-1.5.xsd">

    <context:component-scan base-package="org.springbyexample.web.service" />

    <flex:message-broker/>  ❶

    <flex:remoting-destination ref="personRepository" /> ❷

</beans>
```

❶ Spring BlazeDS Integration configuration of the BlazeDS message broker, which handles remoting and messaging requests.

❷ Exposes the *personRepository* bean as a BlazeDS remoting destination.

3. Adobe BlazeDS Configuration

The BlazeDS configuration first imports the 'remoting-config.xml' and sets up a default channel called 'person-amf'. Then in the channels section, it's URL and the class that will handle requests to the URL is configured. The parameters in the URL 'server.name' and 'server.port' are supplied by the Flex runtime. The 'context.root' parameter needs to be supplied during compilation using the 'context-root' compiler option.

/WEB-INF/flex/services-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>

    <services>
        <service-include file-path="remoting-config.xml" />

        <default-channels>
           <channel ref="person-amf"/>
```

```
        </default-channels>
    </services>

    <channels>
        <channel-definition id="person-amf" class="mx.messaging.channels.AMFChannel">
            <endpoint
url="http://{server.name}:{server.port}/{context.root}/spring/messagebroker/amf"
                      class="flex.messaging.endpoints.AMFEndpoint"/>
        </channel-definition>
    </channels>

    <logging>
        <target class="flex.messaging.log.ConsoleTarget" level="Error">
            <properties>
                <prefix>[BlazeDS] </prefix>
                <includeDate>false</includeDate>
                <includeTime>false</includeTime>
                <includeLevel>false</includeLevel>
                <includeCategory>false</includeCategory>
            </properties>
            <filters>
                <pattern>Endpoint.*</pattern>
                <pattern>Service.*</pattern>
                <pattern>Configuration</pattern>
            </filters>
        </target>
    </logging>

    <system>
        <redeploy>
            <enabled>false</enabled>
        </redeploy>
    </system>

</services-config>
```

A `JavaAdapter` is configured to handle remoting requests. The 'person-amf' channel configured in the main config is set as the default channel. If Spring weren't used, remote services would be configured here.

/WEB-INF/flex/remoting-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="remoting-service"
         class="flex.messaging.services.RemotingService">

    <adapters>
        <adapter-definition id="java-object"
                            class="flex.messaging.services.remoting.adapters.JavaAdapter"
                            default="true"/>
    </adapters>

    <default-channels>
        <channel ref="person-amf"/>
    </default-channels>

</service>
```

4. Code Example

This is the other remoting service previously mentioned. It's picked up by the *context:component-scan* because of the `Service` annotation and exposed as a Flex [http://www.adobe.com/products/flex/] remoting service because of the `RemotingDestination` annotation. The service exposed is 'personService' based on the class name, but if a value could be passed into the `Service` annotation to expose it under a different name (ex: `@Service("otherService")`). To explicitly expose or hide methods the annotations `RemotingInclude` and `RemotingExclude` can be used.

Note

Trying to pass the `Person` instance returned from a remote request directly into the Person DAO's delete would caused an error since it's missing information woven into the class by the JPA implementation for the one to many relationship with `Address`.

```
@Service
@RemotingDestination
public class PersonService {

    private final PersonRepository repository;

    @Autowired
    public PersonService(PersonRepository repository) {
        this.repository = repository;
    }

    /**
     * <p>Deletes person.</p>
     *
     * <p><strong>Note</strong>: Passing back the person
     * from the Flex client causes a problem with Hibernate.</p>
     */
    public void remove(int id) {
        repository.delete(id);
    }

}
```

Example 1 `PersonService`

5. Flex Code Example

This section will go over the code for the Flex [http://www.adobe.com/products/flex/] search page. It will cover remoting, the UI, internationlization (i18n), and logging. Parts of the application use the Flex [http://www.adobe.com/products/flex/] MVC framework from Adobe called Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/]. The model, view, and controller are all in the ActionScript code and helps separate business logic from the UI components.

Note

```
-locale=en_US,es_ES -source-path=../locales/{locale}
```

```
                -context-root simple-flex
                -compiler.services ${user.dir}/../webapp/WEB-INF/flex/services-config.xml
```

These additional compiler arguments are necessary when building the Flex [http://www.adobe.com/products/flex/] part of the example. The 'locale' option specifies that both the 'en_US' and 'es_ES' locales should be compiled into the binary. The 'source-path' option indicates where the different locales properties files should be found. To use locales other than english (en_US) a command must be run for the Flex SDK to copy the default locale of 'en_US' to create the new locale.

```
$ /Applications/Adobe\ Flex\ Builder\ 3\ Plug-in/sdks/3.2.0/bin/copylocale en_US es_ES
```

The 'context-root' is the web applications context path and is used as a variable in the 'services-config.xml' when defining the remoting channel's URL. The 'compiler.services' option points to the location of the BlazeDS configuration. The different channels defined are compiled into the binary, so when a `RemoteObject` is defined in the code below it isn't necessary to specify it's endpoint.

The 'search.mxml' is the entry point for the Flex [http://www.adobe.com/products/flex/] application. So it's main enclosing element is *mx:Application*. In Flex [http://www.adobe.com/products/flex/] components can either be made in mxml files or ActionScript files. It configures the *mx* namespace (Flex [http://www.adobe.com/products/flex/] components) and *controller* namespace (applicaton specific classes). The layout is set to 'horizontal', but isn't important since there is just one component displayed. There are multiple events during the components initialization that can have callbacks registered with them. The 'initialize' and 'addedToStage' events are used here.

The *mx:Metadata* element loads the 'messages' resource bundles. The *controller* namespace is used to instantiate the application's two Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/] controllers.

The *mx:Script* element contains ActionScript. At the beginning of it are imports just like in Java. Below that fields for the logging are defined. Underneath that are the two methods for handling initialization events and a key down handler for the logging window. During initialization logging is setup with the application's log target and the logging window is initialized. The Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/] events for initializing the locale from the server and the initial data for search are dispatched as well as setting the model for the search results to the `DataGrid`. The added to stage event sets up a key down handler to hide and show the logging window when 'ctrl + shift + up' is pressed.

The *mx:DataGrid* is the display component for search. If columns weren't explicitly defined, the data would still be shown but the property name would be used for the header. By using the *mx:DataGridColumn* element the columns being shown and the i18n column name is used. Also the third column is a custom renderer that creates an edit and delete button for each row. The edit button redirects to the edit page using the `navigateToURL` method, and delete sends a request to the person service.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:controller="org.springbyexample.web.flex.controller.*"
                layout="horizontal"
                initialize="initializeHandler()"
                addedToStage="addedToStageHandler()">
```

```
<!--
    To use other locales besides en_US, the en_US locale must be copied in the sdk.

        ex: /Applications/Adobe\ Flex\ Builder\ 3\ Plug-in/sdks/3.2.0/bin/copylocale en_US es_ES
-->
<mx:Metadata>
    [ResourceBundle("messages")] ❶
</mx:Metadata>

<controller:ResourceController/>
<controller:PersonController/>

<mx:Script>
    <![CDATA[
        import org.springbyexample.web.flex.log.LogWindow;
        import mx.logging.Log;
        import mx.logging.ILogger;

        import org.springbyexample.web.flex.event.LocaleChangeEvent;
        import org.springbyexample.web.flex.event.PersonSearchEvent;
        import org.springbyexample.web.flex.log.StringBufferTarget;
        import org.springbyexample.web.flex.model.PersonSearchModelLocator;

        private const logger:ILogger = Log.getLogger("search.mxml");  ❷
        private var logTarget:StringBufferTarget = new StringBufferTarget();
        private var logWindow:LogWindow;

        /**
         * Initialize component.
         */
        private function initializeHandler():void { ❸
            Log.addTarget(logTarget);

            logWindow = new LogWindow()
            logWindow.logTarget = logTarget;

            var lce:LocaleChangeEvent = new LocaleChangeEvent(resourceManager);
            lce.dispatch();

            var psml:PersonSearchModelLocator = PersonSearchModelLocator.getInstance();
            searchDataGrid.dataProvider = psml.personData;

            var pse:PersonSearchEvent = new PersonSearchEvent();
            pse.dispatch();
        }

        /**
         * Handles the 'addedToStage' event.
         */
        private function addedToStageHandler():void {
            stage.addEventListener(KeyboardEvent.KEY_DOWN, keydownHandler);
        }

        /**
         * Handles key down event.  Toggles showing the log text area
         * if 'ctrl + shift + up' is pressed.
         */
        private function keydownHandler(event:KeyboardEvent):void {
            if (event.ctrlKey && event.shiftKey && event.keyCode == Keyboard.UP) {
                if (logWindow.active) {
                    logWindow.hide();
                } else {
                    logWindow.open(this);
                }
            }
        }
    ]]>
</mx:Script>

<mx:DataGrid id="searchDataGrid"> ❹
```

```
        <mx:columns>
            <mx:DataGridColumn headerText="{resourceManager.getString('messages',
'person.form.firstName')}" dataField="firstName"/>  ❺
            <mx:DataGridColumn headerText="{resourceManager.getString('messages',
'person.form.lastName')}" dataField="lastName"/> ❻
            <mx:DataGridColumn width="150" editable="false">
                <mx:itemRenderer> ❼
                    <mx:Component>
                        <mx:HBox>
                            <mx:Script>
                                <![CDATA[
                                    import org.springbyexample.web.flex.event.PersonDeleteEvent;
                                    import org.springbyexample.web.jpa.bean.Person;
                                ]]>
                            </mx:Script>
                            <mx:Button label="{resourceManager.getString('messages',
'button.edit')}"
                                       click="navigateToURL(new URLRequest('../person/form.html?id='
+ data.id), '_self');"/>
                            <mx:Button label="{resourceManager.getString('messages',
'button.delete')}"
                                       click="new PersonDeleteEvent((data as
Person).id).dispatch();"/>
                        </mx:HBox>
                    </mx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

Example 2 `search.mxml`

❶ Loads all locale resource bundles that start with 'messages' (all 'messages.properties' resource bundles).

❷ Configures a logger.

❸ Initialization callback handler that configures logging with the application's log target, initializes the log window, dispatch the `LocaleChangeEvent` to get the locale from the server and set the matching one in the client, sets the Person model to the search `DataGrid`, and dispatch the `PersonSearchEvent` to get the search data from the server and initialize the search `DataGrid`

❹ The search `DataGrid` that displays the search results.

❺ Defines the column for the 'firstName' field and retrieves the header from the resource manager's 'messages' bundle using the 'person.form.firstName' key.

❻ Defines the column for the 'lastName' field and retrieves the header from the resource manager's 'messages' bundle using the 'person.form.lastName' key.

❼ Creates a custom renderer for the third column that has an edit and delete button for the current row. The edit button redirects to the HTML edit page. The delete button sends a request to the server and removes the row from the UI.

The `Person` ActionScript class is very similar to a Java class. A package is defined, imports, and a class that can contain variables and functions. The class has *RemoteClass* metadata set on it indicating that if the `org.springbyexample.web.jpa.bean.Person` Java class is serialized by either a remoting or a messaging request, Flex [http://www.adobe.com/products/flex/] will bind the incoming data to the matching ActionScript class. More can be read about the mapping between ActionScript and Java at Explicitly mapping ActionScript and Java objects [http://livedocs.adobe.com/blazeds/1/blazeds_devguide/help.html?content=serialize_data_3.html].

```
package org.springbyexample.web.jpa.bean {

import mx.collections.ArrayCollection;


/**
 * <p>Person information which binds to the Java
 * remote class <code>org.springbyexample.web.jpa.bean.Person</code>.</p>
 *
 * @author David Winterfeldt
 */
[RemoteClass(alias="org.springbyexample.web.jpa.bean.Person")]
public class Person {

    public var id:int;
    public var firstName:String;
    public var lastName:String;
    public var addresses:ArrayCollection;
    public var created:Date;

}

}
```

Example 3 `Person`

This is the Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/] model. The `PersonSearchModelLocator` has the metadata value *[Bindable]* set on it. This indicates that any changes to values in this class will fire events to anything a value is bound to. In this case the search `DataGrid` is bound to the `personData ArrayCollection` so when the controller updates the data, the `DataGrid` is automatically updated.

The `PersonSearchModelLocator` is using the standard singleton pattern, but ideally a Dependency Injection (DI) framework would be used instead to inject the model where it's needed. This wasn't done to keep this example simpler, and this is actually the suggested way to create a model in Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/]. There are multiple DI frameworks are available. Two are Parsley [http://www.spicefactory.org/parsley/] and Spring ActionScript [http://www.springactionscript.org/]. Spring ActionScript [http://www.springactionscript.org/] is a SpringSource sponsored project.

```
[Bindable]
public class PersonSearchModelLocator implements ModelLocator {

    public var personData:ArrayCollection = new ArrayCollection();

    private static var _instance:PersonSearchModelLocator = null;

    /**
     * Implementation of <code>ModelLocator</code>.
     */
    public static function getInstance():PersonSearchModelLocator {
            if (_instance == null)  {
                    _instance = new PersonSearchModelLocator();
        }

        return _instance;
    }
```

```
}
```

Example 4 `PersonSearchModelLocator`

This is the custom Cairngorm [http://opensource.adobe.com/wiki/display/cairngorm/] event for retrieving person search data. As was seen in 'search.mxml' an instance of the event can be created and `dispatch` can then be called on it.

```
public class PersonSearchEvent extends CairngormEvent {

    public static var EVENT_ID:String = "org.springbyexample.web.flex.event.PersonSearchEvent";

    /**
     * Constructor
     */
    public function PersonSearchEvent() {
        super(EVENT_ID);
    }

}
```

Example 5 `PersonSearchEvent`

The `PersonController` is a front controller and allows mapping of custom events to command implementations for the events.

```
public class PersonController extends FrontController {

    /**
     * Constructor
     */
    public function PersonController() {
        super();

        addCommand(PersonSearchEvent.EVENT_ID, PersonSearchCommand);
        addCommand(PersonDeleteEvent.EVENT_ID, PersonDeleteCommand);
    }

}
```

Example 6 `PersonController`

The `PersonSearchCommand` was associated with the `PersonSearchEvent` in the `PersonController`. In `execute(event:CairngormEvent)`, which is the implementation of `ICommand`, a remoting request to the 'personRepository' service is made. A `RemoteObject` is created passing in the name of the service, and the method matching the Java class on the server is called. It's very simple and straightforward. An event listener is attached to the `RemoteObject` to listen for a result. An event listener could also be registered to listen for a failure. Flex [http://www.adobe.com/products/flex/] has an excellent event model that is easy to leverage for custom

events.

```
public class PersonSearchCommand implements ICommand {

    /**
     * Implementation of <code>ICommand</code>.
     */
    public function execute(event:CairngormEvent):void {
        var ro:RemoteObject = new RemoteObject("personRepository");
        ro.findAll();

        ro.addEventListener(ResultEvent.RESULT, updateSearch);
    }

    /**
     * Updates search.
     */
    private function updateSearch(event:ResultEvent):void {
        var psml:PersonSearchModelLocator = PersonSearchModelLocator.getInstance();

        psml.personData.source = (event.result as ArrayCollection).source;
    }

}
```

Example 7 `PersonSearchCommand`

The `PersonDeleteCommand` was associated with the `PersonDeleteEvent` in the `PersonController`. It retrieves the person id from the event and makes a delete request to the server. Upon success a `PersonSearchEvent` is fired to display the latest data in the search results. It would be more efficient to just remove the row from the `ArrayCollection`, but this was done to illustrate how easy it is to perform different tasks in the application once everything is cleanly decoupled using MVC.

```
public class PersonDeleteCommand implements ICommand {

    private const logger:ILogger =
Log.getLogger("org.springbyexample.web.flex.controller.command.PersonDeleteCommand");

    /**
     * Implementation of <code>ICommand</code>.
     */
    public function execute(event:CairngormEvent):void {
        var pde:PersonDeleteEvent = event as PersonDeleteEvent;
        var id:int = pde.id;

        logger.info("Delete person.  id=" + id);

        var ro:RemoteObject = new RemoteObject("personService");
        ro.remove(id);

        ro.addEventListener(ResultEvent.RESULT, updateSearch);
    }

    /**
     * Updates search.
     */
    private function updateSearch(event:ResultEvent):void {
        var pse:PersonSearchEvent = new PersonSearchEvent();
        pse.dispatch();
    }
```

```
}
```

Example 8 `PersonDeleteCommand`

6. Reference

Related Links

- Adobe Flex [http://www.adobe.com/products/flex/]
- Spring BlazeDS Integration [http://www.springsource.org/spring-flex]
- Adobe BlazeDS [http://sourceforge.net/adobe/blazeds/wiki/Home/]
- Explicitly mapping ActionScript and Java objects [http://livedocs.adobe.com/blazeds/1/blazeds_devguide/help.html?content=serialize_data_3.html]
- Adobe Cairngorm [http://sourceforge.net/adobe/cairngorm/home/Home/]
- Maven Book: Developing with Flexmojos [http://www.sonatype.com/books/mvnref-book/reference/flex-dev.html]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd flex/simple-flex-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring BlazeDS Integration 1.5.x

Part V. Enterprise

Enterprise examples.

Simple Spring JMS

David Winterfeldt

2009

This is a very simple example using a Spring JMS Template to send messages and also having a JMS listener process the messages sent. An embedded ActiveMQ instance is used as the broker.

1. Producer Configuration

Spring Configuration

The Spring configuration shows a *context:component-scan* that picks up the JMS producer and listener. Following this the Spring custom namespace for Apache's ActiveMQ is used to create an embedded JMS broker. A queue is configured for 'org.springbyexample.jms.test'. Then a JMS connection factory is made for the `JmsTemplate` to use. The template will be used by the producer to send messages.

Excerpt from *JmsMessageListenerTest-context.xml*

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:amq="http://activemq.apache.org/schema/core"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/jms
                           http://www.springframework.org/schema/jms/spring-jms.xsd
                           http://activemq.apache.org/schema/core
                           http://activemq.apache.org/schema/core/activemq-core.xsd">

    <context:component-scan base-package="org.springbyexample.jms" />

    <!--  Embedded ActiveMQ Broker -->
    <amq:broker id="broker" useJmx="false" persistent="false">
        <amq:transportConnectors>
            <amq:transportConnector uri="tcp://localhost:0" />
        </amq:transportConnectors>
    </amq:broker>

    <!--  ActiveMQ Destination  -->
    <amq:queue id="destination" physicalName="org.springbyexample.jms.test" />

    <!-- JMS ConnectionFactory to use, configuring the embedded broker using XML -->
    <amq:connectionFactory id="jmsFactory" brokerURL="vm://localhost" />

    <!-- JMS Producer Configuration -->
    <bean id="jmsProducerConnectionFactory"
          class="org.springframework.jms.connection.SingleConnectionFactory"
          depends-on="broker"
          p:targetConnectionFactory-ref="jmsFactory" />
```

```
    <bean id="jmsProducerTemplate" class="org.springframework.jms.core.JmsTemplate"
          p:connectionFactory-ref="jmsProducerConnectionFactory"
          p:defaultDestination-ref="destination" />

    ...

</beans>
```

Code Example

The producer uses `@PostConstruct` to indicate that `generateMessages()` is an initialization method. It uses the `JmsTemplate` to send text messages and also sets an `int` property for the message count.

src/main/java/org/springbyexample/jms/JmsMessageProducer.java

```


@Component
public class JmsMessageProducer {

    private static final Logger logger = LoggerFactory.getLogger(JmsMessageProducer.class);

    protected static final String MESSAGE_COUNT = "messageCount";

    @Autowired
    private JmsTemplate template = null;
    private int messageCount = 100;

    /**
     * Generates JMS messages
     */
    @PostConstruct
    public void generateMessages() throws JMSException {
        for (int i = 0; i < messageCount; i++) {
            final int index = i;
            final String text = "Message number is " + i + ".";

            template.send(new MessageCreator() {
                public Message createMessage(Session session) throws JMSException {
                    TextMessage message = session.createTextMessage(text);
                    message.setIntProperty(MESSAGE_COUNT, index);

                    logger.info("Sending message: " + text);

                    return message;
                }
            });
        }
    }

}
```

Example 1 `JmsMessageProducer`

2. Client Configuration

Spring Configuration

This shows configuring the JMS listener using Springs *jms* custom namespace. The *jmsMessageListener* bean was loaded by the *context:component-scan* and implements `MessageListener`. If it didn't the *jms:listener* element could specify which method should process a message from the queue. The *jms:listener* specifies the *destination* attribute to be 'org.springbyexample.jms.test', which matches the queue defined by the *amq:queue* element in the embedded ActiveMQ configuration.

The `AtomicInteger` is used by the listener to increment how many messages it processes, and is also used by the unit test to confirm is received all the messages from the producer.

Excerpt from *JmsMessageListenerTest-context.xml*

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:amq="http://activemq.apache.org/schema/core"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/jms
                           http://www.springframework.org/schema/jms/spring-jms.xsd
                           http://activemq.apache.org/schema/core
                           http://activemq.apache.org/schema/core/activemq-core.xsd">

    <context:component-scan base-package="org.springbyexample.jms" />

    ...

    <!-- JMS Consumer Configuration -->
    <bean id="jmsConsumerConnectionFactory"
          class="org.springframework.jms.connection.SingleConnectionFactory"
          depends-on="broker"
          p:targetConnectionFactory-ref="jmsFactory" />

    <jms:listener-container container-type="default"
                            connection-factory="jmsConsumerConnectionFactory"
                            acknowledge="auto">
        <jms:listener destination="org.springbyexample.jms.test" ref="jmsMessageListener" />
    </jms:listener-container>

    <!-- Counter for consumer to increment and test to verify count -->
    <bean id="counter" class="java.util.concurrent.atomic.AtomicInteger" />

</beans>
```

Code Example

The `JmsMessageListener` implements the JMS interface `MessageListener`. The `int` property for the message count can be retrieved before casting the message to `TextMessage`. Then the message and message count are both logged.

src/main/java/org/springbyexample/jms/JmsMessageListener.java

```
@Component
public class JmsMessageListener implements MessageListener {

    private static final Logger logger = LoggerFactory.getLogger(JmsMessageListener.class);

    @Autowired
    private AtomicInteger counter = null;

    /**
     * Implementation of <code>MessageListener</code>.
     */
    public void onMessage(Message message) {
        try {
            int messageCount = message.getIntProperty(JmsMessageProducer.MESSAGE_COUNT);

            if (message instanceof TextMessage) {
                TextMessage tm = (TextMessage)message;
                String msg = tm.getText();


                logger.info("Processed message '{}'.  value={}", msg, messageCount);

                counter.incrementAndGet();
            }
        } catch (JMSException e) {
            logger.error(e.getMessage(), e);
        }
    }

}
```

Example 2 `JmsMessageListener`

3. Reference

Related Links

- Spring 3.1.x JMS Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jms.html]
- Java Message Service (JMS) [http://www.oracle.com/technetwork/java/jms/index.html]
- Apache ActiveMQ [http://activemq.apache.org/]
- Apache ActiveMQ Spring Support Documentation [http://activemq.apache.org/spring-support.html]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/simple-spring-jms
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Simple Spring Web Services using JAXB for Marshalling

David Winterfeldt

2008

A very simple example of using Spring Web Services 2.1.x with JAXB for marshalling and unmarshalling requests. A JAXB plugin for Maven is used to generate the JAXB beans from and XSD and the XSD is reused to generate a WSDL. The response from the server sends a person list, but could easily be modified to retrieve person based on an ID.

1. Server Configuration

Web Configuration

The `MessageDispatcherServlet` needs to be defined and the URL patterns it will handle. The *contextConfigLocation* is specified instead of allowing the default (*/WEB-INF/spring-ws-servlet.xml*) because this location makes the configuration easier to share with the unit test.

/WEB-INF/web.xml

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

    <servlet>
        <servlet-name>spring-ws</servlet-name>
<servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>classpath:/spring-ws-context.xml</param-value>
                </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Spring Configuration

The `PersonEndpoint` is defined as a bean and will automatically be registered with Spring Web Services since the class is identified as an endpoint by the `@Endpoint` annotation. This configuration uses the person.xsd that was used to generated the JAXB beans to generate the WSDL. The locationUri matches the URL pattern specified in the web.xml.

The JAXB marshaller/unmarshaller is configured using Spring OXM and also set on the `MarshallingMethodEndpointAdapter` bean.

/src/main/resources/spring-ws-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.ws.service" />

    <bean id="person" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition"
          p:portTypeName="Person"
          p:locationUri="/personService/"
          p:requestSuffix="-request"
          p:responseSuffix="-response">
        <property name="schema">
            <bean class="org.springframework.xml.xsd.SimpleXsdSchema"
                  p:xsd="classpath:/person.xsd" />
            </bean>
        </property>
    </bean>

    <bean
class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodEndpointMapping">
        <description>An endpoint mapping strategy that looks for @Endpoint and @PayloadRoot
annotations.</description>
    </bean>

    <bean class="org.springframework.ws.server.endpoint.adapter.MarshallingMethodEndpointAdapter">
        <description>Enables the MessageDispatchServlet to invoke methods requiring OXM
marshalling.</description>
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
          p:contextPath="org.springbyexample.person.schema.beans" />

</beans>
```

XML Schema Descriptor

A very simple XSD defining an element to indicate an incoming request to get all persons (name element isn't used) and a person response element that contains a list of person elements.

person.xsd

```
<xsd:schema xmlns="http://www.springbyexample.org/person/schema/beans"
            targetNamespace="http://www.springbyexample.org/person/schema/beans"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="get-persons-request">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="name" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="person-response">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="person" type="person"
                             minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="person">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:int" />
            <xsd:element name="first-name" type="xsd:string" />
            <xsd:element name="last-name" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Code Example

Interface for getting persons using the JAXB generated beans ('get-persons-request' element --> GetPersonsRequst? , 'person-response' element --> PersonResponse?). It also has constants for the namespace (matches XSD) and a request constant.

```
public interface MarshallingPersonService {

    public final static String NAMESPACE = "http://www.springbyexample.org/person/schema/beans";
    public final static String GET_PERSONS_REQUEST = "get-persons-request";

    /**
     * Gets person list.
     */
    public PersonResponse getPersons(GetPersonsRequest request);

}
```

Example 1 `MarshallingPersonService`

It is indicated as an endpoint by the `@Endpoint` annotation and implements `MarshallingPersonService`. The `getPersons` method is indicated to handle a specific namespace and incoming request element.

The endpoint just prepares a static response, but this could very easily have a DAO injected into it and information retrieved from a database that is then mapped into the JAXB beans. The `PersonResponse` is created using the JAXB Fluent API which is less verbose than the standard JAXB API.

```
@Endpoint
public class PersonEndpoint implements MarshallingPersonService {

    /**
     * Gets person list.
     */
    @PayloadRoot(localPart=GET_PERSONS_REQUEST, namespace=NAMESPACE)
    public PersonResponse getPersons(GetPersonsRequest request) {
        return new PersonResponse().withPerson(
                    new Person().withId(1).withFirstName("Joe").withLastName("Smith"),
                    new Person().withId(2).withFirstName("John").withLastName("Jackson"));
    }

}
```

Example 2 `PersonEndpoint`

2. Client Configuration

Spring Configuration

The `org.springbyexample.ws.service` package is scanned for beans and will find the `PersonServiceClient` and inject the `WebServiceTemplate` into it. The JAXB marshaller/umarshaller is defined and set on the template.

The import of the *jetty-context.xml* isn't relevant to creating a client, but it creates an embedded jetty instance that loads the *spring-ws-context.xml* and it's services. Then client in the unit test is able to run in isolation.

PersonServiceClientTest.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="jetty-context.xml"/>

    <context:component-scan base-package="org.springbyexample.ws.client" />

    <context:property-placeholder location="org/springbyexample/ws/client/ws.properties"/>

    <bean id="personWsTemplate" class="org.springframework.ws.client.core.WebServiceTemplate"
              p:defaultUri="http://${ws.host}:${ws.port}/${ws.context.path}/personService/"
              p:marshaller-ref="marshaller"
              p:unmarshaller-ref="marshaller" />
```

```
    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
          p:contextPath="org.springbyexample.person.schema.beans" />

</beans>
```

Code Example

At this point Spring Web Services handle almost everything. The template just needs to be called and it will return the `PersonResponse` from the service endpoint. The client can be used like this: `PersonResponse response = client.getPersons(new GetPersonsRequest());`

```
public class PersonServiceClient implements MarshallingPersonService {

    @Autowired
    private WebServiceTemplate wsTemplate;

    /**
     * Gets person list.
     */
    public PersonResponse getPersons(GetPersonsRequest request) {
        PersonResponse response =
            (PersonResponse) wsTemplate.marshalSendAndReceive(request);

        return response;

    }

}
```

Example 3 `PersonServiceClient`

3. Unit Test

Spring Configuration

The unit test's main XML configuration was shown in the Client Spring Configuration section, but this is the configuration that it imported. It creates an embedded Jetty instance and registers the Spring Web Service `MessageDispatcherServlet`. The *spring-ws-context.xml* configuration is passed into the servlet.

jetty-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="org/springbyexample/ws/client/ws.properties"/>

    <bean id="jettyServer"
          class="org.mortbay.jetty.Server"
          init-method="start" destroy-method="stop">
        <property name="threadPool">
            <bean id="ThreadPool"
                  class="org.mortbay.thread.concurrent.ThreadPool">
                <constructor-arg value="0" />
            </bean>
        </property>
        <property name="connectors">
            <list>
                <bean id="Connector"
                      class="org.mortbay.jetty.nio.SelectChannelConnector"
                      p:port="${ws.port}"
                      p:maxIdleTime="30000"
                      p:acceptors="2"
                      p:confidentialPort="8443" />
            </list>
        </property>
        <property name="handlers">
            <list>
                <bean class="org.mortbay.jetty.servlet.Context"
                      p:contextPath="/${ws.context.path}">
                    <property name="sessionHandler">
                        <bean class="org.mortbay.jetty.servlet.SessionHandler" />
                    </property>
                    <property name="servletHandler">
                        <bean class="org.mortbay.jetty.servlet.ServletHandler">
                            <property name="servlets">
                                <list>
                                    <bean class="org.mortbay.jetty.servlet.ServletHolder"
                                          p:name="spring-ws">
                                        <property name="servlet">
                                            <bean
class="org.springframework.ws.transport.http.MessageDispatcherServlet" />
                                        </property>
                                        <property name="initParameters">
                                            <map>
                                                <entry key="contextConfigLocation"
value="classpath:/spring-ws-context.xml" />
                                            </map>
                                        </property>
                                    </bean>
                                </list>
                            </property>
                            <property name="servletMappings">
                                <list>
                                    <bean class="org.mortbay.jetty.servlet.ServletMapping"
                                          p:servletName="spring-ws"
                                          p:pathSpec="/*" />
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>
                <bean class="org.mortbay.jetty.handler.DefaultHandler" />
                <bean class="org.mortbay.jetty.handler.RequestLogHandler" />
            </list>
        </property>
    </bean>

</beans>
```

4. Reference

Related Links

- Spring Web Services Site [http://www.springframework.org/spring-ws]
- JAXB Reference Implementation Project [http://jaxb.java.net/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/simple-spring-web-services
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Web Services 2.1.x

Embedded Spring Web Services

David Winterfeldt

2009

When running a standalone application, there might be a need to have it controlled by web services. This example shows how to configure an embedded Jetty instance to run your Spring Web Services and still allow the embedded Spring Web Services context reference the main application context as a parent.

This example will primarily focus on embedding the Web Services, but it's based on Simple Spring Web Services. Anything not covered here should be explained in the other example.

1. Spring Configuration

The Jetty configuration configures what would be considered the server context even though there isn't anything there except a `Person` bean to be shared with the web application context. The Spring Web Services web application is created by adding a `Context` to Jetty that has the `MessageDispatcherServlet` and the Spring configuration file *spring-ws-embedded-context.xml* in it.

Note

There isn't a need for a *web.xml*. The configuration is completely handled by configuring Jetty in Spring.

embedded-jetty-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <!--
        This bean will be available for injection in the child context
        that the Web Service is in.
    -->
    <bean class="org.springbyexample.person.schema.beans.Person"
          p:id="1" p:firstName="Joe" p:lastName="Smith" />  ❶

    <context:property-placeholder location="org/springbyexample/ws/embedded/ws.properties"/>

    <!-- Manually start server after setting parent context. (init-method="start") -->
    <bean id="jettyServer"
          class="org.mortbay.jetty.Server"
          destroy-method="stop">  ❷
        <property name="threadPool">
            <bean id="ThreadPool"
                  class="org.mortbay.thread.concurrent.ThreadPool">
                <constructor-arg value="0" />
            </bean>
```

```
        </property>
        <property name="connectors">
            <list>
                <bean id="Connector"
                      class="org.mortbay.jetty.nio.SelectChannelConnector"
                      p:port="${ws.port}"
                      p:maxIdleTime="30000"
                      p:acceptors="2"
                      p:confidentialPort="8443" />
            </list>
        </property>
        <property name="handlers">
            <list>
                <bean class="org.mortbay.jetty.servlet.Context"
                      p:contextPath="/${ws.context.path}">  ❸
                    <property name="sessionHandler">
                        <bean class="org.mortbay.jetty.servlet.SessionHandler" />
                    </property>
                    <property name="servletHandler">
                        <bean class="org.mortbay.jetty.servlet.ServletHandler">
                            <property name="servlets">
                                <list>
                                    <bean class="org.mortbay.jetty.servlet.ServletHolder"
                                          p:name="spring-ws">
                                        <property name="servlet">
                                            <bean
class="org.springframework.ws.transport.http.MessageDispatcherServlet" />  ❹
                                        </property>
                                        <property name="initParameters">
                                            <map>
                                                <entry key="contextConfigLocation"
value="classpath:/spring-ws-embedded-context.xml" />  ❺
                                            </map>
                                        </property>
                                    </bean>
                                </list>
                            </property>
                            <property name="servletMappings">
                                <list>
                                    <bean class="org.mortbay.jetty.servlet.ServletMapping"
                                          p:servletName="spring-ws"
                                          p:pathSpec="/*" />  ❻
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>
                <bean class="org.mortbay.jetty.handler.DefaultHandler" />
                <bean class="org.mortbay.jetty.handler.RequestLogHandler" />
            </list>
        </property>
    </bean>

</beans>
```

❶ Bean from the main application context that will be wired into the embedded Spring Web Services context.

❷ Jetty Server bean. Note that it's init method is not configured since this will be done programatically so the parent context is available when the Web Services application starts.

❸ This bean removes the need for having a *web.xml* by creating a web context for the Spring Web Services. The context path for the webapp is set on this bean.

❹ The `MessageDispatcherServlet` is configured as a servlet for the webapp. This is the main servlet that handles Spring Web Services.

❺ The Spring configuration file, *spring-ws-embedded-context.xml*, is configured for the

`MessageDispatcherServlet` as a servlet parameter. This overrides the servlet's default configuration file, which would be */WEB-INF/spring-ws-servlet.xml* (*/WEB-INF/${servlet.name}-servlet.xml*).

❻ This configures the `MessageDispatcherServlet` to handle any incoming request.

This configuration file is the same as any that would be used to configure a Spring Web Services web application. If this file wasn't explicitly specified in the `MessageDispatcherServlet`, the servlet would look for the default file */WEB-INF/spring-ws-servlet.xml* (*/WEB-INF/${servlet.name}-servlet.xml*).

/src/main/resources/spring-ws-embedded-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.ws.embedded.service" />

    <bean id="person" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition"
          p:portTypeName="Person"
          p:locationUri="/personService/"
          p:requestSuffix="-request"
          p:responseSuffix="-response">
        <property name="schema">
            <bean class="org.springframework.xml.xsd.SimpleXsdSchema"
                  p:xsd="classpath:/person.xsd" />
        </property>
    </bean>

    <bean
class="org.springframework.ws.server.endpoint.mapping.PayloadRootAnnotationMethodEndpointMapping">
        <description>An endpoint mapping strategy that looks for @Endpoint and @PayloadRoot
annotations.</description>
    </bean>

    <bean class="org.springframework.ws.server.endpoint.adapter.MarshallingMethodEndpointAdapter">
        <description>Enables the MessageDispatchServlet to invoke methods requiring OXM
marshalling.</description>
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
          p:contextPath="org.springbyexample.person.schema.beans" />

</beans>
```

2. Code Example

The code excerpt below is from `EmbeddedPersonServiceClientTest`, and is run before any of the tests are run to initialize Jetty. Even though this is a unit test, this would be the same code that would be used to start a standalone Spring application. It could contain an executor pool, db connection pool, timer processes, etc.

The main application context is initialized, and then the Jetty Server bean is retrieved from it. Then the `ServletContext` from the Spring Web Services webapp is retrieved. An intermediary web application context is

created between the main application context and the Spring Web Services context. The intermediary web context is then set as an attribute on the `ServletContext` under the constant `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE`, which indicates it should be used as the parent context for the web application. Then the Jetty server is ready to be started.

```
AbstractApplicationContext ctx =
    new
ClassPathXmlApplicationContext("/org/springbyexample/ws/embedded/embedded-jetty-context.xml");  ❶
ctx.registerShutdownHook();

Server server = (Server) ctx.getBean("jettyServer");  ❷

ServletContext servletContext = null;

for (Handler handler : server.getHandlers()) {
    if (handler instanceof Context) {
        Context context = (Context) handler;

        servletContext = context.getServletContext();❸
    }
}

XmlWebApplicationContext wctx = new XmlWebApplicationContext();  ❹
wctx.setParent(ctx);
wctx.setConfigLocation("");
wctx.setServletContext(servletContext);
wctx.refresh();

servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, wctx);  ❺

server.start();  ❻
```

Example 1 `EmbeddedPersonServiceClientTest`

❶ Create the main application context which has Jetty configured in it.
❷ Retrieve the Jetty server bean.
❸ Retrieve the `ServletContext` from the Spring Web Services webapp. The context path of the `Context` doesn't need to be checked because it's the only `Context` configured.
❹ Create a web application context as an intermediary between the main application context and the webapp's context.
❺ Set created web application context as the parent context of the web application by setting it on the `ServletContext` under the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` constant.
❻ Starts the Jetty server.

3. Reference

Related Links

- Spring Web Services Site [http://www.springframework.org/spring-ws]
- JAXB Reference Implementation Project [http://jaxb.java.net/]
- Jetty [http://www.mortbay.org/jetty/]

- Simple Spring Web Services

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/simple-spring-web-services
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Web Services 2.1.x

Simple Spring Integration

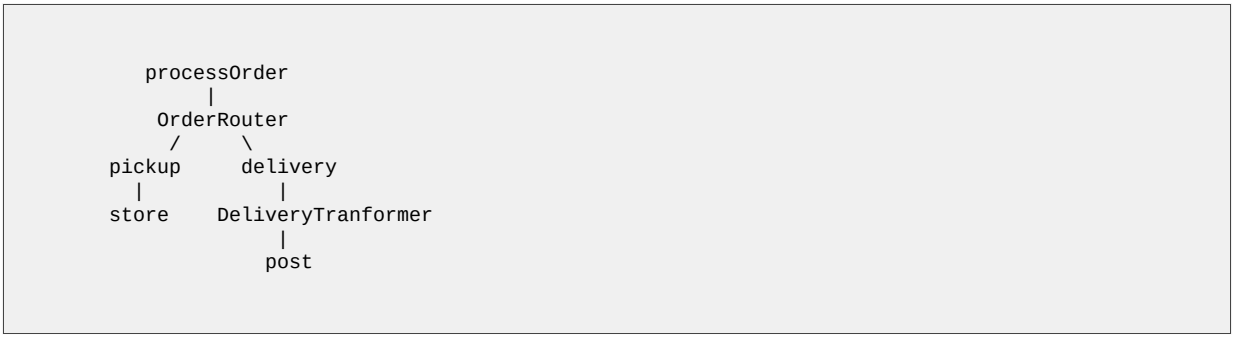
David Winterfeldt

2009

This example uses Spring Integration [http://www.springframework.org/spring-integration] to process a book order and appropriately route the message depending on if it's a pickup from the store or it should be delivered. Spring Integration [http://www.springframework.org/spring-integration] is an implementation of Enterprise Integration Patterns [http://eaipatterns.com/].

From the `Order` gateway, the `BookOrder` is sent to the 'processOrder' channel. An `OrderRouter` routes the order either to the 'pickup' or 'delivery' channels based on the order's `OrderType` annotation. If the order is a pickup, the 'pickup' channel is bridged to the 'store' channel which is configured to have the `StoreEndpoint` do the final processing for this part of the flow. If the order is a delivery, the `DeliveryTransformer` converts the `BookOrder` into an `OnlineBookOrder` that contains the delivery address. The address is just hard coded in the example, but could have looked up an address in a real application. The online order is sent to the 'post' channel, which is configured to have the `PostEndpoint` handle the end of this part of the flow.

Order Message Flow



1. Spring Configuration

This configuration uses a combination of XML and classes with annotations to configure the message flow. The *annotation-config* element enables annotation-based configuration for Spring Integration [http://www.springframework.org/spring-integration]. The *context:component-scan* loads the annotated part of the configuration. The *gateway* element creates the `Order` gateway, which is the beginning of the flow. Different channels are created, a bridge between two channels using the *bridge* element, and the *outbound-channel-adapter* is used to configure an endpoint for the 'store' and 'post' channels.

OrderTest-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xmlns:context="http://www.springframework.org/schema/context"
```

```
            xmlns:stream="http://www.springframework.org/schema/integration/stream"
            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.springframework.org/schema/context/spring-context.xsd
                                http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration-2.1.xsd
                                http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-integration-stream-2.1.xsd">

    <!--
           processOrder
                |
            OrderRouter
             /     \
        pickup     delivery
          |           |
        store    DeliveryTranformer
                      |
                    post
    -->

    <annotation-config/>

    <context:component-scan base-package="org.springbyexample.integration.book.annotation"/>

    <gateway id="order" service-interface="org.springbyexample.integration.book.Order"/>

    <channel id="processOrder"/>

    <channel id="delivery"/>
    <channel id="pickup"/>

    <bridge input-channel="pickup" output-channel="store" />

    <channel id="store"/>
    <channel id="post"/>

    <outbound-channel-adapter channel="store" ref="storeEndpoint" method="processMessage" />
    <outbound-channel-adapter channel="post" ref="postEndpoint" method="processMessage" />

</beans:beans>
```

2. Code Example

The gateway, or main entry point to the order flow, is the `Order` interface. It was defined in the Spring configuration and the `@Gateway` annotation indicates what channel any items passed to `process(BookOrder order)` will be sent to.

```


public interface Order {

    /**
     * Process a book order.
     */
    @Gateway(requestChannel="processOrder")
    public void process(BookOrder order);

}
```

Example 1 `Order`

The `OrderRouter` indicates it is a message endpoint by using the `@MessageEndpoint` on the class. The `@Router` annotation on `processOrder(BookOrder order)` configures the incoming channel by setting the *inputChannel* attribute. The method should return a `String` that matches one of the configured channels. In this case the incoming channel is set to 'processOrder' and depending on the order type either 'delivery' or 'pickup' will be returned.

```
@MessageEndpoint
public class OrderRouter {

    final Logger logger = LoggerFactory.getLogger(OrderRouter.class);

    /**
     * Process order.  Routes based on whether or
     * not the order is a delivery or pickup.
     */
    @Router(inputChannel="processOrder")
    public String processOrder(BookOrder order) {
        String result = null;

        logger.debug("In OrderRouter.  title='{}'  quantity={}  orderType={}",
                     new Object[] { order.getTitle(),
                                    order.getQuantity(),
                                    order.getOrderType() });

        switch (order.getOrderType()) {
            case DELIVERY:
                result = "delivery";
                break;
            case PICKUP:
                result = "pickup";
                break;
        }

        return result;
    }

}
```

Example 2 `OrderRouter`

The `StoreEndpoint` is the end of the store pickup flow. It just retrieves the message payload and logs it's output. In a real application the order could be electronically sent or printed at the appropriate store so the order could be prepared for pickup.

```
@MessageEndpoint
public class StoreEndpoint {

    final Logger logger = LoggerFactory.getLogger(StoreEndpoint.class);

    /**
     * Process an order for a pickup from the store.
     */
    public void processMessage(Message<BookOrder> message) {
        BookOrder order =  message.getPayload();

        logger.debug("In StoreEndpoint.  title='{}'  quantity={}  orderType={}",
```

```
                new Object[] { order.getTitle(),
                               order.getQuantity(),
                               order.getOrderType() });
    }

}
```

Example 3 `StoreEndpoint`

The `DeliveryTransformer` indicates `processMessage(BookOrder order)` is a transformer by using the `@Transformer` annotation. It converts a `BookOrder` from the 'delivery' channel to an `OnlineBookOrder` which will be sent to the 'post' channel. The address for the online order is hard coded, but in a real application the transformer could look up an address.

```


@MessageEndpoint
public class DeliveryTransformer {

    final Logger logger = LoggerFactory.getLogger(DeliveryTransformer.class);

    /**
     * Transforms a <code>BookOrder</code> that is a delivery
     * into a <code>OnlineBookOrder</code>.
     */
    @Transformer(inputChannel="delivery", outputChannel="post")
    public OnlineBookOrder processMessage(BookOrder order) {
        logger.debug("In DeliveryTransformer.  title='{}'  quantity={}  orderType={}",
                new Object[] { order.getTitle(),
                               order.getQuantity(),
                               order.getOrderType() });

        return new OnlineBookOrder(order.getTitle(), order.getQuantity(),
                                   order.getOrderType(),
                                   "1060 West Addison Chicago, IL 60613");
    }

}
```

Example 4 `DeliveryTransformer`

The `PostEndpoint` is the end of the delivery flow. It just retrieves the message payload for the online order and logs it's output. In a real application the order could be electronically sent or printed at the appropriate warehouse so the order could be prepared for shipping to the address added by the `DeliveryTransformer`.

Note

The `Message` was passed in as an example, but since only the message payload is being used it could have been passed in directly (ex: `processMessage(OnlineBookOrder order)`). Also, `@Header` could have been used to annotate a method parameter to extract a value from the message header.

```

```

```
@MessageEndpoint
public class PostEndpoint {

    final Logger logger = LoggerFactory.getLogger(PostEndpoint.class);

    /**
     * Process a delivery order for sending by mail.
     */
    public void processMessage(Message<OnlineBookOrder> message) {
        OnlineBookOrder order =  message.getPayload();

        logger.debug("In PostEndpoint.  title='{}'  quantity={}  orderType={}  address='{}'",
                new Object[] { order.getTitle(),
                               order.getQuantity(),
                               order.getOrderType(),
                               order.getAddress() });
    }

}
```

Example 5 `PostEndpoint`

3. Reference

Related Links

- Spring Integration [http://www.springframework.org/spring-integration]
- Enterprise Integration Patterns [http://eaipatterns.com/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/spring-integration
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring Integration 2.1.x

Spring JMX

David Winterfeldt

2009

JMX is good way to expose parts of your application for monitoring and management. Spring also provides support for exposing JMX for remote management (JSR-160 [http://jcp.org/en/jsr/detail?id=160]) and creating a client to manage it. This example registers a Spring bean as a JMX MBean, exposes the JMX server over JMX Messaging Protocol (JMXMP), and creates a client to access it.

To connect to the example using a JMX Management tool, JConsole can be used and comes with the JDK installation. In the IDE a break point could be set in the unit test or `Thread.sleep` could be added to pause the application from shutting down. To activate local JMX access, the Java argument *-Dcom.sun.management.jmxremote* must be used when starting the test.

```
$ jconsole
```

1. Spring Configuration

The *context:component-scan* creates a Spring bean from the `ServerManagerImpl`. The *context:mbean-export* element will register any annotation beans as JMX MBeans.

The *serverConnector* bean will expose JMX for remote management over JMXMP.

The *clientConnector* creates a remote reference as an `MBeanServerConnection`. From this `MBeanInfo` can be retrieved for different JMX MBeans. But even more convenient is to create a proxy of the MBean using the `MBeanProxyFactoryBean`. This creates a remote proxy to the MBean specified by the *p:objectName* attribute, and uses the *p:proxyInterface* attribute to set what interface should be used for the proxy.

JmxTest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.jmx" />

    <context:mbean-export/>

    <!-- Expose JMX over JMXMP -->
    <bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean" />

    <!-- Client connector to JMX over JMXMP -->
    <bean id="clientConnector"
class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean"
          p:serviceUrl="service:jmx:jmxmp://localhost:9875" />
```

```
    <!-- Client ServerManager proxy to JMX over JMXMP -->
    <bean id="serverManagerProxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean"
          p:objectName="org.springbyexample.jmx:name=ServerManager"
          p:proxyInterface="org.springbyexample.jmx.ServerManager"
          p:server-ref="clientConnector" />

</beans>
```

2. Code Example

The `ServerManagerImpl` is registered as a Spring bean, and then exposed as a JMX MBean.

The `@Component` annotation indicates that this class is eligible to be a Spring bean when *context:component-scan* runs. The `@ManagedResource` annotation indicates that the bean can be exported as a JMX MBean. It's *objectName* attribute is optional, but can be used to specify a specific name for the MBean. In this case it will be exposed under the 'org.springbyexample.jmx' directory with the name 'ServerManager'. By default it would have been exposed under the directory 'org.springbyexample.jmx' and subdirectory 'serverManagerImpl' with the name 'ServerManagerImpl'.

The `@ManagedAttribute` can expose a getter or setter by annotating the method. A method that takes multiple values can be exposed using `@ManagedOperation` along with `@ManagedOperationParameters` and `@ManagedOperationParameter` to parameters the operation can take.

```


@Component
@ManagedResource(objectName="org.springbyexample.jmx:name=ServerManager",
                 description="Server manager.")
public class ServerManagerImpl implements ServerManager {

    final Logger logger = LoggerFactory.getLogger(ServerManagerImpl.class);

    private String serverName = "springServer";
    private boolean serverRunning = true;
    private int minPoolSize = 5;
    private int maxPoolSize = 10;

    /**
     * Gets server name.
     */
    @ManagedAttribute(description="The server name.")
    public String getServerName() {
        return serverName;
    }

    /**
     * Whether or not the server is running.
     */
    @ManagedAttribute(description="Server's running status.")
    public boolean isServerRunning() {
        return serverRunning;
    }

    /**
     * Sets whether or not the server is running.
     */
    @ManagedAttribute(description="Whether or not the server is running.",
                      currencyTimeLimit=20,
                      persistPolicy="OnUpdate")
```

```
    public void setServerRunning(boolean serverRunning) {
        this.serverRunning = serverRunning;
    }

    /**
     * Change db connection pool size.
     *
     * @param   min     Minimum pool size.
     * @param   max     Maximum pool size.
     *
     * @return  int     Current pool size.
     */
    @ManagedOperation(description="Change db connection pool size.")
    @ManagedOperationParameters({
        @ManagedOperationParameter(name="min", description= "Minimum pool size."),
        @ManagedOperationParameter(name="max", description= "Maximum pool size.")})
    public int changeConnectionPoolSize(int minPoolSize, int maxPoolSize) {
        Assert.isTrue((minPoolSize > 0),
                      "Minimum connection pool size must be larger than zero.  min=" + minPoolSize);
        Assert.isTrue((minPoolSize < maxPoolSize),
                      "Minimum connection pool size must be smaller than the maximum." +
                      "  min=" + minPoolSize + ", max=" + maxPoolSize);

        this.minPoolSize = minPoolSize;
        this.maxPoolSize = maxPoolSize;

        int diff = (maxPoolSize - minPoolSize);

        // randomly generate current pool size between new min and max
        Random rnd = new Random();
        int currentSize = (minPoolSize + rnd.nextInt(diff));

        logger.info("Changed connection pool size. min={}, max={}, current={}",
                    new Object[] { this.minPoolSize, this.maxPoolSize, currentSize});

        return currentSize;

    }

}
```

Example 1 `ServerManagerImpl`

```


@Autowired
private MBeanServerConnection clientConnector = null;

@Autowired
@Qualifier("serverManagerProxy")
private ServerManager serverManager = null;

public void testMBeanServerConnection() {
    MBeanInfo beanInfo = null;

    try {
        beanInfo = clientConnector.getMBeanInfo(new
ObjectName("org.springbyexample.jmx:name=ServerManager"));
    } ...
}

public void testServerManagerRemoteProxy() {
    ...

    logger.info("serverName={}, serverRunning={}",
                serverManager.getServerName(), serverManager.isServerRunning());
```

```
    int min = 10;
    int max = 20;

    int result = serverManager.changeConnectionPoolSize(min, max);

    ...
}
```

Example 2 Excerpt from `JmxTest`

3. Reference

Related Links

- Spring 3.1.x JMX Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jmx.html]
- Java Management Extenstions (JMX) [http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html]
- Monitoring and Management Using JMX [http://docs.oracle.com/javase/1.5.0/docs/guide/management/agent.html]
- JSR-160 - Remote Management & Notifications for JMX [http://jcp.org/en/jsr/detail?id=160]
- Using JConsole to Monitor Applications [http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/spring-jmx
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Spring Modules JCR Node Creation & Retrieval

David Winterfeldt

2008

This example shows how to create and retrieve nodes and files (content nodes) using Spring Modules JCR (Java Content Repository (JSR-170)) module. This example uses Apache Jackrabbit for the Java Content Repository which is the reference implementation for JSR-170.

1. Spring Configuration

The first bean definition defines defines the Jackrabbit repository by specifying the configuration file to use and the location of the repository. If the repository doesn't already exist, it will be created on startup. The next bean creates a session factory based on the respository and the next one creates a JcrTemplate using the session factory.

JcrNodeCreationIT-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <!-- Register Annotation-based Post Processing Beans -->
    <context:annotation-config />

    <!-- normal repository -->
    <bean id="repository"
          class="org.springmodules.jcr.jackrabbit.RepositoryFactoryBean"
          p:configuration="classpath:/jackrabbit-repository.xml"
          p:homeDir="file:./target/repo" />

    <bean id="sessionFactory"
          class="org.springmodules.jcr.jackrabbit.JackrabbitSessionFactory"
          p:repository-ref="repository" />

    <bean id="jcrTemplate"
          class="org.springmodules.jcr.JcrTemplate"
          p:sessionFactory-ref="sessionFactory"
          p:allowCreate="true" />

</beans>
```

2. Code Example

This code using `JcrTemplate` creates a node if one doesn't already exist in the repository.

```
private String nodeName = "fileFolder";

...

public void testAddNodeIfDoesntExist() {
    assertNotNull("JCR Template is null.", template);

    template.execute(new JcrCallback() {
        public Object doInJcr(Session session) throws RepositoryException {
            Node root = session.getRootNode();

            logger.debug("Starting from root node.  node={}", root);

            Node node = null;

            if (root.hasNode(nodeName)) {
                node = root.getNode(nodeName);

                logger.debug("Node exists.  node={}", node);
            } else {
                node = root.addNode(nodeName);

                session.save();

                logger.info("Saved node.  node={}", node);
            }

            assertNotNull("Node is null.", node);

            return node;
        }
    });
}
```

Example 1 Node Creation (Excerpt from `JcrNodeCreationIT`)

This code using `JcrTemplate` creates a file node by attaching one if one doesn't already exist in the repository.

If the file node doesn't exist, it is created from the log4j.xml file in the classpath by passing in an `InputStream` of the file. After this you will see the part of the unit test that retrieves the file node and checks if the first and last line are correct.

The first thing that is done is to retrieve the root node from the JCR Session by calling `session.getRootNode()`. Then the expected folder node that should contain the file is located just under the root node, so `root.getNode(nodeName)` is called to retrieve the folder node. Then if the file node doesn't exist under the folder node, it's created. First a node named based on the file name is created under the folder node, then a JCR content node is created under the file node and a data property is where the actual file (as an `InputStream`) is set.

```
private String nodeName = "fileFolder";

...

public void testAddFileIfDoesntExist() {
    Node node = (Node) template.execute(new JcrCallback() {
```

```
        @SuppressWarnings("unchecked")
        public Object doInJcr(Session session) throws RepositoryException,
                IOException {
            Node resultNode = null;

            Node root = session.getRootNode();
            logger.info("starting from root node.  node={}", root);

            // should have been created in previous test
            Node folderNode = root.getNode(nodeName);

            String fileName = "log4j.xml";

            if (folderNode.hasNode(fileName)) {
                logger.debug("File already exists.  file={}", fileName);
            } else {
                InputStream in = this.getClass().getResourceAsStream("/" + fileName);

                Node fileNode = folderNode.addNode(fileName, JcrConstants.NT_FILE);

                // create the mandatory child node - jcr:content
                resultNode = fileNode.addNode(JcrConstants.JCR_CONTENT, JcrConstants.NT_RESOURCE);

                resultNode.setProperty(JcrConstants.JCR_MIMETYPE, "text/xml");
                resultNode.setProperty(JcrConstants.JCR_ENCODING, "UTF-8");
                resultNode.setProperty(JcrConstants.JCR_DATA, in);
                Calendar lastModified = Calendar.getInstance();
                lastModified.setTimeInMillis(System.currentTimeMillis());
                resultNode.setProperty(JcrConstants.JCR_LASTMODIFIED, lastModified);

                session.save();

                IOUtils.closeQuietly(in);

                logger.debug("Created '{}' file in folder.", fileName);
            }

            ...
        }
    });
}
```

Example 2 File Node Creation (Excerpt from `JcrNodeCreationIT`)

This code using `JcrTemplate` to retrieve the content of a file node from the repository.

```


private String nodeName = "fileFolder";

...

public void testAddFileIfDoesntExist() {
    Node node = (Node) template.execute(new JcrCallback() {
        public Object doInJcr(Session session) throws RepositoryException,
                IOException {
            Node resultNode = null;

            JcrConstants jcrConstants = new JcrConstants(session);

            Node root = session.getRootNode();
            logger.info("starting from root node.  node={}", root);

            // should have been created in previous test
            Node folderNode = root.getNode(nodeName);

            String fileName = "log4j.xml";
```

```
...
            Node contentNode = folderNode.getNode(fileName).getNode(JcrConstants.JCR_CONTENT);
            Property dataProperty = contentNode.getProperty(JcrConstants.JCR_DATA);
            List<String> list = (List<String>) IOUtils.readLines(dataProperty.getStream());
...
        }
    });
}
```

Example 3 File Node Retrieval (Excerpt from `JcrNodeCreationIT`)

3. Reference

Related Links

- Spring by Example JCR Site [http://springbyexample.org/maven/site/sbe-jcr/1.0.2/sbe-jcr/]
- Spring Modules [http://www.springsource.org/spring-modules]
- Apache Jackrabbit [http://jackrabbit.apache.org/]
- Integrating Java Content Repository and Spring [http://www.infoq.com/articles/spring-modules-jcr]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/spring-modules-jcr
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Velocity E-mail Template

David Winterfeldt

2009

A Velocity template is used to generate the body of an e-mail and then is sent using Spring mail sender.

Note

When running the example, the *mailSender* bean in the Spring configuration needs to be edited to a valid host, and the 'mail.properties' file also needs to have valid user and password for this host entered along with setting the test e-mail recipient.

1. Spring Configuration

The *context:component-scan* loads the `VelocityEmailSender` bean, and the *context:property-placeholder* loads the mail properties which have the mail user, password, and e-mail recipient.

The *velocityEngine* bean is created by Spring's `VelocityEngineFactoryBean`. It sets the *resourceLoaderPath* property to 'classpath:/org/springbyexample/email', so the Velocity engine will use this as the root of any template references. By setting *preferFileSystemAccess* property to 'false', a Spring resource loader will be used for loading template resources instead of the default Velocity file resource loader.

The *mailSender* bean is configured to send to Gmail. It can be customized to match another host and many will probably not need any extra Java Mail properties defined. The *templateMessage* bean at the end of the configuration is just used to pass in the from address, recipient address, and the subject to use when constructing `MimeMessageHelper` in the `VelocityEmailSender` (which is below in the Code Example).

VelocityEmailSenderIT-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.email" />

    <!-- Edit 'mail.properties' to set a valid user and password for the mail sender. -->
    <context:property-placeholder location="classpath:/mail.properties" />

    <bean id="velocityEngine" class="org.springframework.ui.velocity.VelocityEngineFactoryBean"
          p:resourceLoaderPath="classpath:/org/springbyexample/email"
          p:preferFileSystemAccess="false"/>

    <!-- Mail sender configured for using Gmail -->
    <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl"
          p:host="smtp.gmail.com"
```

```
          p:username="${mail.username}"
          p:password="${mail.password}">
        <property name="javaMailProperties">
            <props>
                <prop key="mail.smtp.auth">true</prop>
                <prop key="mail.smtp.starttls.enable">true</prop>
            </props>
        </property>
    </bean>

    <bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage"
          p:from="dwinterfeldt@springbyexample.org"
          p:to="${mail.recipient}"
          p:subject="Greetings from Spring by Example" />

</beans>
```

2. Code Example

The `send(SimpleMailMessage, Map<Object, Object>)` method uses a `MimeMessagePreparator` to create an e-mail with an HTML body. The recipient address, from address, and subject are set on the message. Then Spring's `VelocityEngineUtils` is used to generate the message body. It uses the Velocity engine, the location of the template (which is in the '/org/springbyexample/email' package), and the variables for Velocity to use during template generation. After the mime message is created is is send using the `JavaMailSender`.

If a plain text body was being used the `MailSender` interface could be used to send a `SimpleMailMessage`.

```


@Component
public class VelocityEmailSender implements Sender {

    private static final Logger logger = LoggerFactory.getLogger(VelocityEmailSender.class);

    private final VelocityEngine velocityEngine;
    private final JavaMailSender mailSender;

    /**
     * Constructor
     */
    @Autowired
    public VelocityEmailSender(VelocityEngine velocityEngine,
                               JavaMailSender mailSender) {
        this.velocityEngine = velocityEngine;
        this.mailSender = mailSender;
    }

    /**
     * Sends e-mail using Velocity template for the body and
     * the properties passed in as Velocity variables.
     *
     * @param   msg                 The e-mail message to be sent, except for the body.
     * @param   hTemplateVariables  Variables to use when processing the template.
     */
    public void send(final SimpleMailMessage msg,
                     final Map<Object, Object> hTemplateVariables) {
        MimeMessagePreparator preparator = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage) throws Exception {
               MimeMessageHelper message = new MimeMessageHelper(mimeMessage);
               message.setTo(msg.getTo());
```

```
                message.setFrom(msg.getFrom());
                message.setSubject(msg.getSubject());

                String body = VelocityEngineUtils.mergeTemplateIntoString(velocityEngine,
"/emailBody.vm", hTemplateVariables);

                logger.info("body={}", body);

                message.setText(body, true);
             }
          };

          mailSender.send(preparator);

         logger.info("Sent e-mail to '{}'.", msg.getTo());
    }

}
```

Example 1 `VelocityEmailSender`

3. Reference

Related Links

- Spring 3.1.x E-mail Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mail.html]
- Spring 3.1.x E-mail Template Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mail.html#mail-templates-example]
- Velocity Engine [http://velocity.apache.org/engine/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/velocity-email-template
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Test Setup

After updating the *mailSender* bean in the Spring configuration to have a valid host, and the 'mail.properties', run the command below to run the test from the command line. Because of manual configuration changes the test is

excluded from the other integration tests.

```
$ mvn -Pit -Dit.test=VelocityEmailSenderIT verify
```

Project Information

- Spring Framework 3.1.x

Solr Client

David Winterfeldt

2009

Spring by Example Utils's `HttpClientTemplate`, `HttpClientOxmTemplate`, and `SolrOxmClient` are used for making different Apache Solr [http://lucene.apache.org/solr/] client requests. Solr [http://lucene.apache.org/solr/] provides an XML based API over HTTP to the Apache Lucene [http://lucene.apache.org/] search engine.

Note

The Solr [http://lucene.apache.org/solr/] server needs to be started before running the unit tests. After downloading Solr [http://lucene.apache.org/solr/] and changing to it's directory, the example server can be run and in another console window the sample data can be loaded into the server using the commands below.

```
$ cd example
$ java -jar start.jar

$ cd example/exampledocs
$ java -jar post.jar *.xml
```

1. Connecting to Solr using `SolrOxmClient`

Spring Configuration

The *context:component-scan* loads the `CatalogItemMarshaller` which marshalls an update request and unmarshalls a search request. The *context:property-placeholder* loads values for the Solr [http://lucene.apache.org/solr/] host and port. The *selectUrl* bean sets up the URL for a select, which is used by an `HttpClientTemplate`, but just for debugging the XML of the search result. The *solrOxmClient* bean just needs the base url for Solr [http://lucene.apache.org/solr/] and a marhsaller and unmarshaller.

SolrOxmClientIT-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <!-- Loads CatalogItemMarshaller -->
    <context:component-scan base-package="org.springbyexample.enterprise.solr" />
```

```
    <context:property-placeholder location="org/springbyexample/enterprise/solr/solr.properties"/>

    <!-- Just used for debugging -->
    <bean id="selectUrl" class="java.lang.String">
        <constructor-arg value="http://${solr.host}:${solr.port}/solr/select" />
    </bean>

    <bean id="solrOxmClient" class="org.springbyexample.httpclient.solr.SolrOxmClient"
          p:baseUrl="http://${solr.host}:${solr.port}/solr"
          p:marshaller-ref="catalogItemMarshaller"
          p:unmarshaller-ref="catalogItemMarshaller" />

</beans>
```

Code Example

The `SolrOxmClient` supports marshalling searches and unmarshalling updates. Updates and deletes autocommit, but their methods are overridden so a boolean can be passed in to control commits. It also allows for direct calls to commit, rollback, and optimize. Commit and optimize both support creating requests with max segments, wait flush, and wait searcher by using `SolrRequestAttributes`.

Note

The statistics for searches, updates, commits, etc. can be checked using Solr Stats [http://localhost:8983/solr/admin/stats.jsp].

Search

Simple search passing in a query. It could be any valid Solr [http://lucene.apache.org/solr/] query.

```
List<CatalogItem> lCatalogItems = client.search(SEARCH_QUERY_PARAM);
```

Example 1 Excerpt from `SolrOxmClientIT.testSearch()`

Paginated Search

The 'start' & 'rows' indicate what range of the results to return in the `Map`. The search query is 'electronics' and is passed into the search along with the `Map`.

The 'indent' value isn't used by the unmarshaller, and shouldn't be set since it will introduce whitespace in the XML results. In this case it is set so a debug request request that logs the unprocessed XML results is easier to read.

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put("start", "5");
hParams.put("rows", "5");
hParams.put("indent", "on");
```

```
...

List<CatalogItem> lCatalogItems = client.search("electronics", hParams);
```

Example 2 Excerpt from `SolrOxmClientIT.testPaginatedSearch()`

Update

Adds or updates any records in the list based on their id. A commit request is sent immediately after the update.

```
List<CatalogItem> lCatalogItems = new ArrayList<CatalogItem>();

CatalogItem item = new CatalogItem();
item.setId(CATALOG_ITEM_ID);
item.setManufacturer(CATALOG_ITEM_MANUFACTURER);
item.setName(CATALOG_ITEM_NAME);
item.setPrice(CATALOG_ITEM_PRICE);
item.setInStock(CATALOG_ITEM_IN_STOCK);
item.setPopularity(expectedPopularity);

lCatalogItems.add(item);

client.update(lCatalogItems);
```

Example 3 Excerpt from `SolrOxmClientIT.testUpdate()`

Rollback

Update is called passing in the list and a boolean value of `false` indicating not to commit the update. Then commit or rollback can be called manually. In this example rollback is called.

```
List<CatalogItem> lCatalogItems = new ArrayList<CatalogItem>();

CatalogItem item = new CatalogItem();
item.setId(CATALOG_ITEM_ID);
item.setManufacturer(CATALOG_ITEM_MANUFACTURER);
item.setName(CATALOG_ITEM_NAME);
item.setPrice(CATALOG_ITEM_PRICE);
item.setInStock(CATALOG_ITEM_IN_STOCK);
item.setPopularity(popularity);

lCatalogItems.add(item);

// update without commit
client.update(lCatalogItems, false);

...

client.rollback();
```

Example 4 Excerpt from `SolrOxmClientIT.testRollback()`

Delete

This deletes by using a query matching all 'manu' fields with 'Belkin'. A commit is immediately sent after the delete request. There is also a `deleteById(String)` for deleting specific records based on their id.

```
client.deleteByQuery("manu:Belkin");
```

Example 5 Excerpt from `SolrOxmClientIT.testDelete()`

Optimize

Sends a request to optimize the search engine.

```
client.optimize();
```

Example 6 Excerpt from `SolrOxmClientIT.testOptimize()`

Creating a marshaller/unmarshaller is the most work setting up the `SolrOxmClient` since it handles the custom marshalling and unmarshalling between the custom JavaBean and the search fields configured in Solr [http://lucene.apache.org/solr/].

Implementation of Spring OXM `Marshaller` and `Unmarshaller` using dom4j [http://www.dom4j.org/].

```
@Component
public class CatalogItemMarshaller implements Marshaller, Unmarshaller {

    final Logger logger = LoggerFactory.getLogger(CatalogItemMarshaller.class);

    private static final String ADD_ELEMENT_NAME = "add";
    private static final String DOC_ELEMENT_NAME = "doc";
    private static final String FIELD_ELEMENT_NAME = "field";
    private static final String FIELD_ELEMENT_NAME_ATTRIBUTE = "name";

    /**
     * Implementation of <code>Marshaller</code>.
     */
    @SuppressWarnings("unchecked")
    public void marshal(Object bean, Result result) throws XmlMappingException, IOException {
        List<CatalogItem> lCatalogItems = (List<CatalogItem>) bean;

        OutputStream out = null;
        XMLWriter writer = null;

        if (result instanceof StreamResult) {
            try {
                out = ((StreamResult) result).getOutputStream();
```

```
                Document document = DocumentHelper.createDocument();
                Element root = document.addElement(ADD_ELEMENT_NAME);

                for (CatalogItem item : lCatalogItems) {
                    Element doc = root.addElement(DOC_ELEMENT_NAME);

                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
"id")
                        .addText(item.getId());
                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
"manu")
                        .addText(item.getManufacturer());
                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
FIELD_ELEMENT_NAME_ATTRIBUTE)
                        .addText(item.getName());
                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
"price")
                        .addText(new Float(item.getPrice()).toString());
                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
"inStock")
                        .addText(BooleanUtils.toStringTrueFalse(item.isInStock()));
                    doc.addElement(FIELD_ELEMENT_NAME).addAttribute(FIELD_ELEMENT_NAME_ATTRIBUTE,
"popularity")
                        .addText(new Integer(item.getPopularity()).toString());
                }

                writer = new XMLWriter(out);

                writer.write(document);
            } finally {
                try { writer.close(); } catch (Exception e) {}
                IOUtils.closeQuietly(out);
            }

        }

        logger.debug("Marshalled bean of size {}.", lCatalogItems.size());
    }


    /**
     * Implementation of <code>Unmarshaller</code>
     */
    @SuppressWarnings("unchecked")
    public Object unmarshal(Source source) throws XmlMappingException, IOException {
        List<CatalogItem> lResults = new ArrayList<CatalogItem>();

        if (source instanceof StreamSource) {
            InputStream in = null;

            try {
                in = ((StreamSource) source).getInputStream();

                SAXReader reader = new SAXReader();
                Document document = reader.read(in);

                List<Node> lNodes =
document.selectNodes("//response/result[@name='response']/doc/*");

                CatalogItem item = null;

                // loop over all matching nodes in order, so can create a new bean
                // instance on the first match and add it to the results on the last
                for (Node node : lNodes) {
                    if (BooleanUtils.toBoolean(node.valueOf("./@name='id'"))) {
                        item = new CatalogItem();

                        item.setId(node.getText());
                    } else if (BooleanUtils.toBoolean(node.valueOf("./@name='inStock'"))) {
                        item.setInStock(BooleanUtils.toBoolean(node.getText()));
                    } else if (BooleanUtils.toBoolean(node.valueOf("./@name='manu'"))) {
                        item.setManufacturer(node.getText());
```

```
                    } else if (BooleanUtils.toBoolean(node.valueOf("./@name='name'"))) {
                        item.setName(node.getText());
                    } else if (BooleanUtils.toBoolean(node.valueOf("./@name='popularity'"))) {
                        item.setPopularity(Integer.parseInt(node.getText()));
                    } else if (BooleanUtils.toBoolean(node.valueOf("./@name='price'"))) {
                        item.setPrice(Float.parseFloat(node.getText()));

                        lResults.add(item);
                    }
                }
            } catch (DocumentException e) {
                throw new UnmarshallingFailureException(e.getMessage(), e);
            } finally {
                IOUtils.closeQuietly(in);
            }

            logger.debug("Unmarshalled bean of size {}.", lResults.size());
        }

        return lResults;
    }

    /**
     * Implementation of <code>Marshaller</code>.
     */
    @SuppressWarnings("unchecked")
    public boolean supports(Class clazz) {
        return (clazz.isAssignableFrom(List.class));
    }

}
```

Example 7 `CatalogItemMarshaller`

2. Connecting to Solr using `HttpClientTemplate` & `HttpClientOxmTemplate`

Spring Configuration

An `HttpClientTemplate` and `HttpClientOxmTemplate` are configured. The former is used for non-marshalling requests and the latter is for marhsallin requests.

SolrHttpClientTemplateIT-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <!-- Loads CatalogItemMarshaller -->
```

```
    <context:component-scan base-package="org.springbyexample.enterprise.solr" />

    <context:property-placeholder location="org/springbyexample/enterprise/solr/solr.properties"/>

    <bean id="selectUrl" class="java.lang.String">
        <constructor-arg value="http://${solr.host}:${solr.port}/solr/select" />
    </bean>
    <bean id="updateUrl" class="java.lang.String">
        <constructor-arg value="http://${solr.host}:${solr.port}/solr/update" />
    </bean>

    <bean id="selectTemplate" class="org.springbyexample.httpclient.HttpClientTemplate"
          p:defaultUri-ref="selectUrl" />

    <bean id="updateTemplate" class="org.springbyexample.httpclient.HttpClientTemplate"
          p:defaultUri-ref="updateUrl" />

    <bean id="marshallingSelectTemplate"
class="org.springbyexample.httpclient.HttpClientOxmTemplate"
          p:defaultUri-ref="selectUrl"
          p:marshaller-ref="catalogItemMarshaller"
          p:unmarshaller-ref="catalogItemMarshaller" />

    <bean id="marshallingUpdateTemplate" parent="marshallingSelectTemplate"
          p:defaultUri-ref="updateUrl" />

</beans>
```

Code Example

A basic search, paginated search, and update will be shown using marshalling and unmarshalling with `HttpClientOxmTemplate`. Also `HttpClientTemplate` is used to make search requests and log the results for debugging.

Search

A simple search is made by passing in the search query as the key of 'q'. The `selectTemplate` is the `HttpClientTemplate` and performs a search based `Map` and logs the results. The same `Map` is then used by the `marshallingSelectTemplate` to make a request that is unmarshalled into a list of `CatalogItem`s.

```


public void testSearch() {
    assertNotNull("HttpClientOxmTemplate is null.", marshallingSelectTemplate);

    Map<String, String> hParams = new HashMap<String, String>();
    hParams.put("q", SEARCH_QUERY_PARAM);
    hParams.put("indent", "on");

    // just for debugging
    selectTemplate.executeGetMethod(hParams, new ResponseStringCallback() {
        public void doWithResponse(String response) throws IOException {
            logger.debug(response);
        }
    });

    marshallingSelectTemplate.executeGetMethod(hParams,
        new ResponseCallback<List<CatalogItem>>() {
            public void doWithResponse(List<CatalogItem> lCatalogItems) throws IOException {
                assertNotNull("Catalog item list is null.", lCatalogItems);
```

```
                int expectedCount = 2;
                assertEquals("Catalog item list should be '" + expectedCount + "'.", expectedCount,
lCatalogItems.size());

                CatalogItem item = lCatalogItems.get(0);

                logger.debug("id={}  manufacturer={}  name={} price={} inStock={} popularity={}",
                        new Object[] { item.getId(), item.getManufacturer(), item.getName(),
                                       item.getPrice(), item.isInStock(), item.getPopularity() });

...
            }
    });
}
```

Example 8 Excerpt from `SolrHttpClientTemplateIT.testSearch()`

Paginated Search

This search is paginated since it defines what range of results to return, which are defined by the 'start' key and the 'rows' key. The search parameter is defined by the 'q' key, and searches for any record that has 'electronics' in it. The 'indent' key is useful for having formatted results logged by `selectTemplate` for debugging.

```
public void testPaginatedSearch() {
    assertNotNull("HttpClientOxmTemplate is null.", marshallingSelectTemplate);

    Map<String, String> hParams = new HashMap<String, String>();
    hParams.put("q", "electronics");
    hParams.put("start", "5");
    hParams.put("rows", "5");
    hParams.put("indent", "on");

    // just for debugging
    selectTemplate.executeGetMethod(hParams, new ResponseStringCallback() {
        public void doWithResponse(String response) throws IOException {
            logger.debug(response);
        }
    });

    marshallingSelectTemplate.executeGetMethod(hParams,
        new ResponseCallback<List<CatalogItem>>() {
            public void doWithResponse(List<CatalogItem> lCatalogItems) throws IOException {
                assertNotNull("Catalog item list is null.", lCatalogItems);

                int expectedCount = 5;
                assertEquals("Catalog item list should be '" + expectedCount + "'.", expectedCount,
lCatalogItems.size());

                CatalogItem item = lCatalogItems.get(0);

                logger.debug("id={}  manufacturer={}  name={} price={} inStock={} popularity={}",
                        new Object[] { item.getId(), item.getManufacturer(), item.getName(),
                                       item.getPrice(), item.isInStock(), item.getPopularity() });

...
            }
    });
}
```

Example 9 Excerpt from `SolrHttpClientTemplateIT.testPaginatedSearch()`

Update

Adds or updates a list of JavaBeans and then calls commit. The add or update isn't committed until a commit is sent.

```
public void testUpdate() {
...

    List<CatalogItem> lCatalogItems = new ArrayList<CatalogItem>();

    CatalogItem item = new CatalogItem();
    item.setId(CATALOG_ITEM_ID);
    item.setManufacturer(CATALOG_ITEM_MANUFACTURER);
    item.setName(CATALOG_ITEM_NAME);
    item.setPrice(CATALOG_ITEM_PRICE);
    item.setInStock(CATALOG_ITEM_IN_STOCK);
    item.setPopularity(expectedPopularity);

    lCatalogItems.add(item);

    marshallingUpdateTemplate.executePostMethod(lCatalogItems);

    // have to commit the updates
    updateTemplate.executePostMethod(COMMIT);

...

}
```

Example 10 Excerpt from `SolrHttpClientTemplateIT.testUpdate()`

3. Reference

Related Links

- Apache Solr [http://lucene.apache.org/solr/]
- Spring by Example Utils Module

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd enterprise/solr-client
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Spring by Example Utils 1.2.3
- Apache Solr 3.6.1

Part VI. Contact Application

The contact application has a DAO module, WS (Web Service) Beans module, Services module, REST Services module, Web Application module, and a shared test module.

The DAO module contains the schema, JPA entities, and uses Spring Data JPA [http://www.springsource.org/spring-data/jpa] repositories. It uses Spring Profiles to have either an HSQL or PostgreSQL JDBC connection.

The Services module is to contain any business logic. All of it's APIs use JAXB beans and Dozer [http://dozer.sourceforge.net/] is used to convert between this beans and the JPA entities. Security and transactions are configured in this layer.

The REST Services module contains the REST clients & controllers, as well as their Spring configurations. JSON and XML views are supported for requests. The Jackson JSON Processor [http://jackson.codehaus.org/] is used for JSON marshalling. It has the standard JSON media type and a custom one that also includes class information for more complex data models.

The contact webapp has a standard JSP UI, Sencha ExtJS [http://www.sencha.com/products/extjs/], and also a Sencha Touch [http://www.sencha.com/products/touch/] UI.

The DAO, Services, and REST Services all have an abstract test class for each module that each test extends. This way within each module, all tests have a shared context so Spring only has to load once. All of these tests use an in memory database and the REST Services have an embedded jetty server. REST Services tests can be run with clients using JSON or XML for marshalling.

Contact Application DAO

David Winterfeldt

2012

This is the DAO module for the Contact Application and has a basic JPA configuration with inheritance. It uses Spring Data JPA [http://www.springsource.org/spring-data/jpa] for it's repositories and to set audit information when persisting and entity. JPA optimistic locking has also been added. This module is based on Spring Data JPA.

1. Spring Configuration

The first Spring configuration file sets up the `DataSource`, what SQL scripts should be run at startup, and any property placeholder files that should be loaded.

META-INF/spring/db/dao-datasource-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.1.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-3.1.xsd">

    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close">
        <property name="driverClassName" value="${jdbc.driver.class}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="150" />
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
    </bean>

    <beans profile="hsql">   ❶
        <context:property-placeholder
location="classpath:/sql/hsql.properties,classpath*:META-INF/config/**/*.properties" /> ❷

        <util:list id="databaseScriptsList" value-type="org.springframework.core.io.Resource">
            <value>classpath:/sql/hsql/schema.sql</value>
            <value>classpath:/sql/hsql/security_schema.sql</value>
        </util:list>
    </beans>

    <beans profile="postgresql"> ❸
        <context:property-placeholder
location="classpath:/sql/postgresql.properties,classpath*:META-INF/config/**/*.properties,file:contact-app*.properties"
/> ❹

        <util:list id="databaseScriptsList" value-type="org.springframework.core.io.Resource" />
    </beans>

</beans>
```

```
```

❶ HSQL DB Spring profile.
❷ Context property placeholder is configured to explicitly load the *hsql.properties* file and load all properties under *META-INF/config*.
❸ PostgreSQL DB Spring profile.
❹ Context property placeholder is configured to explicitly load the *postgresql.properties* file and load all properties under *META-INF/config*. It also will load any property files from where the application is started that start with 'contact-app' and end with '.properties'.

The second Spring configuration file configures Spring Data JPA [http://www.springsource.org/spring-data/jpa], and it's entity auditing feature. It also configures transactions, the entity manager, and runs any SQL scripts added to the 'databaseScriptsList' bean. The persistence unit name for JPA is set using the properties files that were loaded in the Spring profiles in the previous Spring Configuration. There is a persistence unit configuration for HSQL DB and PostgreSQL in the *persistence.xml*.

META-INF/spring/db/dao-jpa-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jpa:repositories base-package="org.springbyexample.contact.orm.repository" />

    <!-- auditing -->
    <jpa:auditing auditor-aware-ref="auditorAware" />
    <bean id="auditorAware" class="org.springbyexample.contact.orm.entity.AuditorAwareImpl" />

    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager"
          p:entityManagerFactory-ref="entityManagerFactory"/>

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              p:dataSource-ref="dataSource"
          p:persistenceUnitName="${jpa.persistence.unit.name}" ❶
          depends-on="dataSourceInitializer">
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.show_sql">${jpa.show.sql}</prop>
                <prop key="hibernate.generate_statistics">false</prop>
                <prop
key="hibernate.cache.use_second_level_cache">${jpa.cache.use_second_level_cache}</prop>
                <prop
key="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</prop>
            </props>
```

```
        </property>
    </bean>

    <bean id="dataSourceInitializer"
          class="org.springframework.jdbc.datasource.init.DataSourceInitializer"
          p:dataSource-ref="dataSource">
        <property name="databasePopulator">
            <bean class="org.springframework.jdbc.datasource.init.ResourceDatabasePopulator"
                  p:scripts-ref="databaseScriptsList"
                  p:sqlScriptEncoding="UTF-8" />
        </property>
    </bean>

</beans>
```

❶ The persistence unit name is set using a property placeholder, which loaded the appropriate name based on the Spring DB profiles.

2. JPA Configuration

Two *persistence-unit* elements are defined. One for HSSQL DB and another for PostgreSQL.

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="hsql">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />

            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.EhCacheProvider" />
            <property name="jadira.usertype.autoRegisterUserTypes" value="true" />  ❶
            <property name="jadira.usertype.databaseZone" value="jvm" />
        </properties>
    </persistence-unit>
    <persistence-unit name="postgresql">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            <property name="hibernate.dialect"
value="org.springbyexample.orm.dialect.TableNameSequencePostgreSQLDialect" />

            <property name="hibernate.hbm2ddl.auto" value="validate" />
            <property name="hibernate.ejb.naming_strategy"
value="org.hibernate.cfg.ImprovedNamingStrategy"/>
            <property name="hibernate.max_fetch_depth" value="3" />
            <property name="jadira.usertype.autoRegisterUserTypes" value="true" />
            <property name="jadira.usertype.databaseZone" value="jvm" />
        </properties>
    </persistence-unit>
```

```
</persistence>
```

❶ Register Joda `DateTime` support.

3. JPA Configuration

These are the two DB specific properties files that are loaded by their respective Spring profiles in *dao-datasource-context.xml*.

sql/hsql.properties

```
jdbc.driver.class=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:contact
jdbc.username=sa
jdbc.password=

jpa.persistence.unit.name=hsql
jpa.show.sql=false

jpa.cache.use_second_level_cache=false
```

It is expected that the database is already initialzed before running the application. In the contact-app project there is a scripts directory that contains *init-postgres.sh* and *reinit-postgres.sh*. The first script will just create the DB, user, and schema. The second will drop the DB and user first, before creating everything.

sql/postgresql.properties

```
jdbc.driver.class=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/contact
jdbc.username=contact
jdbc.password=contact

jpa.persistence.unit.name=postgresql
jpa.show.sql=false  ❶

jpa.cache.use_second_level_cache=true
```

❶ This can be set to `true` by passing it in as a Java system property or by setting it in a *contact-app.properties* file.

4. Reference

Related Links

- Spring 3.1.x JPA Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/orm.html#orm-jpa]
- Hibernate [http://www.hibernate.org/]
- Apache Commons DBCP [http://commons.apache.org/dbcp]
- PostgreSQL [http://www.postgresql.org/]
- Spring Data JPA

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-dao
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Java Persistence API 2.0

Contact Application Web Service Beans

David Winterfeldt

2012

The WS (Web Service) Beans are used as the main model in the project, in and above the service layer. They are JAXB beans generated from XSDs. Besides always having a valid XSD to go with the XML generated, the generated Java classes have a fluent API for setting values (ex: `new Person.withId(1).withFirstName("John")`). They provide a way to decouple business logic and user facing APIs from a persistent store, multiple persistent stores, or 3rd party services.

1. Spring Configuration

The Spring OXM JAXB marshaller is defined, along with the packages it should manage.

META-INF/spring/marshaller/jaxb2-marshaller-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="contextPaths">
            <array>
                <value>org.springbyexample.schema.beans.entity</value>
                <value>org.springbyexample.schema.beans.person</value>
                <value>org.springbyexample.schema.beans.response</value>
            </array>
        </property>
    </bean>

</beans>
```

2. JAXB Configuration

Custom JAXB bindings for date and date time to use Jode `DateTime`.

jaxb-bindings.xjb

```
```

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
               jaxb:extensionBindingPrefixes="xjc"
               version="2.0">

    <jaxb:bindings>
        <jaxb:globalBindings>
            <jaxb:serializable/>

            <xjc:javaType adapter="org.springbyexample.jaxb.beans.adapter.DateXmlAdapter"
name="org.joda.time.DateTime" xmlType="xsd:date" />
            <xjc:javaType adapter="org.springbyexample.jaxb.beans.adapter.DateTimeXmlAdapter"
name="org.joda.time.DateTime" xmlType="xsd:dateTime" />
        </jaxb:globalBindings>
    </jaxb:bindings>

</jaxb:bindings>
```

Entity base model for persistent entities. It is defined as an *xsd:complexType* so it can be extended, but is annotated to have `XmlRootElement` so it can be serialized by JAXB if it is returned directly from a controller (as opposed to being part of another instance returned).

entity-base.xsd

```


<xsd:schema xmlns="http://www.springbyexample.org/schema/beans/entity"
            targetNamespace="http://www.springbyexample.org/schema/beans/entity"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
            xmlns:annox="http://annox.dev.java.net"
            jaxb:extensionBindingPrefixes="annox">

    <xsd:complexType name="pk-entity-base">
        <xsd:annotation>
            <xsd:appinfo>
                <annox:annotate>
                    <annox:annotate annox:class="javax.xml.bind.annotation.XmlRootElement"
name="pk-entity-base"/>
                </annox:annotate>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="id" type="xsd:int" />

            <xsd:element name="lock-version" type="xsd:int" />

            <xsd:element name="last-updated" type="xsd:dateTime" />
            <xsd:element name="last-update-user" type="xsd:string" />
            <xsd:element name="created" type="xsd:dateTime" />
            <xsd:element name="create-user" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Responses for single and multiple results. Abstract base classes are defined to provide a way to handle responses

more generically in the Java code.

Excerpt from *response-base.xsd*

```
<xsd:schema xmlns="http://www.springbyexample.org/schema/beans/response"
            targetNamespace="http://www.springbyexample.org/schema/beans/response"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
            xmlns:annox="http://annox.dev.java.net"
            jaxb:extensionBindingPrefixes="annox"
            jaxb:version="2.1">

    ...

    <xsd:complexType name="abstract-response">
        <xsd:annotation>
            <xsd:appinfo>
                <jaxb:class ref="org.springbyexample.schema.beans.response.AbstractResponse"/>
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:complexType>

    <xsd:complexType name="abstract-response-result">
        <xsd:annotation>
            <xsd:appinfo>
                <jaxb:class ref="org.springbyexample.schema.beans.response.AbstractResponseResult"/>
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:complexType>
    <xsd:complexType name="abstract-entity-response-result" abstract="true">
        <xsd:complexContent>
            <xsd:extension base="abstract-response-result">
                <xsd:sequence>
                    <xsd:element name="errors" type="xsd:boolean" />
                    <xsd:element name="message-list" type="message" minOccurs="0"
maxOccurs="unbounded" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="abstract-find-response-result">
        <xsd:annotation>
            <xsd:appinfo>
                <jaxb:class
ref="org.springbyexample.schema.beans.response.AbstractFindResponseResult"/>
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:complexType>
    <xsd:complexType name="abstract-entity-find-response-result" abstract="true">
        <xsd:complexContent>
            <xsd:extension base="abstract-find-response-result">
                <xsd:sequence>
                    <xsd:element name="errors" type="xsd:boolean" />
                    <xsd:element name="message-list" type="message" minOccurs="0"
maxOccurs="unbounded" />

                    <!-- primarily for paginated results -->
                    <xsd:element name="count" type="xsd:long" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="response-result">
        <xsd:annotation>
            <xsd:appinfo>
```

```
                <annox:annotate>
                    <annox:annotate annox:class="javax.xml.bind.annotation.XmlRootElement"
name="response-result"/>
                </annox:annotate>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="abstract-response">
                <xsd:sequence>
                    <xsd:element name="errors" type="xsd:boolean" />
                    <xsd:element name="message-list" type="message" minOccurs="0"
maxOccurs="unbounded" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="find-response-result">
        <xsd:complexContent>
            <xsd:extension base="response-result">
                <xsd:sequence>
                    <!-- primarily for paginated results -->
                    <xsd:element name="count" type="xsd:long" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="message">
        <xsd:sequence>
            <xsd:element name="message-type" type="message-type" />
            <xsd:element name="message" type="xsd:string" />
            <xsd:element name="message-key" type="xsd:string" />
            <xsd:element name="property" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:simpleType name="message-type">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="INFO" />
            <xsd:enumeration value="WARN" />
            <xsd:enumeration value="ERROR" />
            <xsd:enumeration value="FATAL" />
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

Person/Student/Professional complex types and their responses are defined.

person-base.xsd

```


<xsd:schema xmlns="http://www.springbyexample.org/schema/beans/person"
            targetNamespace="http://www.springbyexample.org/schema/beans/person"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
            xmlns:annox="http://annox.dev.java.net"
            jaxb:extensionBindingPrefixes="annox"
            xmlns:entity="http://www.springbyexample.org/schema/beans/entity"
            xmlns:response="http://www.springbyexample.org/schema/beans/response">

    <xsd:import namespace="http://www.springbyexample.org/schema/beans/entity"
schemaLocation="entity-base.xsd" />
    <xsd:import namespace="http://www.springbyexample.org/schema/beans/response"
```

```
schemaLocation="response-base.xsd" />

    <xsd:element name="person-response">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="response:abstract-entity-response-result">
                    <xsd:sequence>
                        <xsd:element name="results" type="person" />
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="person-find-response">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="response:abstract-entity-find-response-result">
                    <xsd:sequence>
                        <xsd:element name="results" type="person" minOccurs="0"
maxOccurs="unbounded" />
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="professional-response">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="response:abstract-entity-response-result">
                    <xsd:sequence>
                        <xsd:element name="results" type="professional" />
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="professional-find-response">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="response:abstract-entity-find-response-result">
                    <xsd:sequence>
                        <xsd:element name="results" type="professional" minOccurs="0"
maxOccurs="unbounded" />
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="person">
        <xsd:annotation>
            <xsd:appinfo>
                <annox:annotate>
                    <annox:annotate annox:class="javax.xml.bind.annotation.XmlRootElement"
name="person"/>
                </annox:annotate>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="entity:pk-entity-base">
                <xsd:sequence>
                    <xsd:element name="first-name" type="xsd:string" />
                    <xsd:element name="last-name" type="xsd:string" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="student">
```

```
        <xsd:annotation>
            <xsd:appinfo>
                <annox:annotate>
                    <annox:annotate annox:class="javax.xml.bind.annotation.XmlRootElement"
name="student"/>
                </annox:annotate>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="person">
                <xsd:sequence>
                    <xsd:element name="school-name" type="xsd:string" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="professional">
        <xsd:annotation>
            <xsd:appinfo>
                <annox:annotate>
                    <annox:annotate annox:class="javax.xml.bind.annotation.XmlRootElement"
name="professional"/>
                </annox:annotate>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="person">
                <xsd:sequence>
                    <xsd:element name="company-name" type="xsd:string" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

3. Reference

Related Links

- JAXB Reference Implementation Project [http://jaxb.java.net/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-ws-beans
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Contact Application Services

David Winterfeldt

2012

The service layer is a where business logic is located, and the persistent layer is converted to and from the JPA entities & JAXB beans using Dozer [http://dozer.sourceforge.net/].

This example doesn't have this, but the service layer is where general handling of exceptions and validation can be put in place using AOP. This way standard Spring runtime exceptions can be thrown, but before they would go to the REST controller layer they could be translated into i18n friendly user messages. Also validation can intercept requests and immediately return results before any of the actual business logic starts processing in a service.

1. Spring Configuration

This is the Spring Dozer Mapper configuration. More than one mapping file can be passed in or wild cards can be used to load mapper configs.

A larger application may have a need to have different dozer mappers with different rules, and they could all be defined here. Then Spring injection could be controlled with custom qualifiers. An example could be if you wanted to completely copy an existing contact, but exclude primary keys from being copied. Then this new copy could be persisted as a new record.

META-INF/spring/services/mapper-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <bean id="org.dozer.Mapper" class="org.dozer.spring.DozerBeanMapperFactoryBean">
        <property name="mappingFiles">
            <list>
                <value>classpath:/dozer/dozer-mappings.xml</value>
            </list>
        </property>
    </bean>

</beans>
```

Spring configuration for loading all converters and services.

META-INF/spring/services/services-context.xml

```
```

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan
base-package="org.springbyexample.contact.converter,org.springbyexample.contact.service" />

</beans>
```

Spring configuration for security. This has the authentication manager use the default JDBC user service and activates detection of the `@Secured` annotation on Spring beans.

META-INF/spring/security/security-context.xml

```
<?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:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <security:global-method-security secured-annotations="enabled" />

    <security:authentication-manager>
        <security:authentication-provider >
            <security:jdbc-user-service data-source-ref="dataSource" />
        </security:authentication-provider>
    </security:authentication-manager>

</beans>
```

2. Dozer Configuration

Custom mappings between classes and fields can be defined here, including referencing Spring custom converters. Currently there is just a custom converter for Joda `DateTime` registered.

dozer/dozer-mappings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://dozer.sourceforge.net
                              http://dozer.sourceforge.net/schema/beanmapping.xsd">
```

```
    <configuration>
        <custom-converters>
            <converter type="org.springbyexample.converter.DateTimeConverter">
                <class-a>org.joda.time.DateTime</class-a>
                <class-b>org.joda.time.DateTime</class-b>
            </converter>
        </custom-converters>
    </configuration>

</mappings>
```

3. Converter Code

There are two simple interfaces, `Converter` and `ListConverter`, used to copy to and from two different types classes. This provides a clear interface for the application to interact with even though Dozer [http://dozer.sourceforge.net/] is the primary implementation, it provides some flexibility for other possible implementations. Such as converting a JSON `String` to and from a class.

Simple interface for converting to and from any two classes.

```
public interface Converter<T, V> {

    /**
     * Converts from a domain model to the business model.
     */
    public V convertTo(T source);

    /**
     * Converts from a business model to the domain model.
     */
    public T convertFrom(V source);

}
```

Example 1 `Converter`

Simple interface for converting to and from lists of any two classes.

```
public interface ListConverter<T, V> extends Converter<T, V> {

    /**
     * Converts from a domain model list to a business model list.
     */
    public List<V> convertListTo(Collection<T> list);

    /**
     * Converts from a business model list to a domain model list.
     */
    public List<T> convertListFrom(Collection<V> list);

}
```

Example 2 `ListConverter`

Most converter implementations should just need to extend this class and pass in the appropriate parameters to the constructor.

```
public abstract class AbstractMapperListConverter<T, V> extends AbstractListConverter<T, V> {

    private final Mapper mapper;
    private final Class<T> tClazz;
    private final Class<V> vClazz;

    public AbstractMapperListConverter(Mapper mapper,
                                       Class<T> tClazz, Class<V> vClazz) {
        this.mapper = mapper;
        this.tClazz = tClazz;
        this.vClazz = vClazz;
    }

    @Override
    public V convertTo(T source) {
        Assert.notNull(source, "Source must not be null.");

        return mapper.map(source, vClazz);
    }

    @Override
    public T convertFrom(V source) {
        Assert.notNull(source, "Source must not be null.");

        return mapper.map(source, tClazz);
    }

}
```

Example 3 `AbstractMapperListConverter`

The `ContactConverter` converts to and from the entity and JAXB bean. It extends `AbstractMapperListConverter`, which basically just delegates to Dozer.

```
@Component
public class ContactConverter extends
AbstractMapperListConverter<org.springbyexample.contact.orm.entity.person.Person, Person> {

    @Autowired
    public ContactConverter(Mapper mapper) {
        super(mapper,
              org.springbyexample.contact.orm.entity.person.Person.class, Person.class);
    }

}
```

Example 4 `ContactConverter`

4. Persistence Service Code

Persistence Service Interface Code

There is a persistence interface and abstract base broken into read-only and persistent operations. For standard persistence based on Spring Data JPA [http://www.springsource.org/spring-data/jpa] repositories, just extending one of the base classes will handle basic persistence.

The find service takes a generic response and find response. The generic response is a wrapper, with messages, for a single result. The find response is for a standard and paginated search. It also can have messages.

```
public interface PersistenceFindService<R extends EntityResponseResult, FR extends
EntityFindResponseResult> {

    /**
     * Find a record with an id.
     */
    public R findById(Integer id);

    /**
     * Find all records.
     */
    public FR find();

    /**
     * Find a paginated record set.
     */
    public FR find(int page, int pageSize);

}
```

Example 5 `PersistenceFindService`

Besides a generic response and find response, the `PersistenceService` also takes the entity bean the persistence class handles.

```
public interface PersistenceService<V extends PkEntityBase,
                                    R extends EntityResponseResult, FR extends
EntityFindResponseResult>
        extends PersistenceFindService<R, FR> {

    /**
     * Creates a record.
     */
    public R create(V request);

    /**
     * Updates a record.
     */
    public R update(V request);

    /**
     * Deletes person.
     */
    public ResponseResult delete(Integer id);

}
```

Example 6 `PersistenceService`

Persistence Service Abstract Code

The abstract persistence find service adds another generic value representing a Spring Data JPA [http://www.springsource.org/spring-data/jpa] entity. It expects a `JpaRepository`, `Converter`, and a `MessageHelper` for it's constructor. The `MessageHelper` is just a helper bean for accessing the Spring `MessageSource`.

The `@Transactional` annotation is set on the class to be read-only. Any method will automatically have a read-only transaction in any subclass unless marked otherwise. The converter is used to convert the Spring Data JPA [http://www.springsource.org/spring-data/jpa] results into ws beans, then an abstract method is used to create the response.

```
@Transactional(readOnly=true)
public abstract class AbstractPersistenceFindService<T extends AbstractPersistable<Integer>, V
extends PkEntityBase,
                                                    R extends EntityResponseResult, FR extends
EntityFindResponseResult>
        extends AbstractService implements PersistenceFindService<R, FR> {

    protected final JpaRepository<T, Integer> repository;
    protected final ListConverter<T, V> converter;

    public AbstractPersistenceFindService(JpaRepository<T, Integer> repository, ListConverter<T, V>
converter,
                                          MessageHelper messageHelper) {
        super(messageHelper);

        this.repository = repository;
        this.converter = converter;
    }

    @Override
    public R findById(Integer id) {
        T bean = repository.findOne(id);
        V result = (bean != null ? converter.convertTo(bean) : null);

        return createResponse(result);
    }

    @Override
    public FR find() {
        List<V> results = converter.convertListTo(repository.findAll(createDefaultSort()));

        return createFindResponse(results);
    }

    @Override
    public FR find(int page, int pageSize) {
        Page<T> pageResults = repository.findAll(new PageRequest(page, pageSize,
createDefaultSort()));

        List<V> results = converter.convertListTo(pageResults.getContent());

        return createFindResponse(results, pageResults.getTotalElements());
    }

    /**
     * Create a response.
     */
    protected abstract R createResponse(V result);
```

```
    /**
     * Create a find response with the count representing the size of the list.
     */
    protected abstract FR createFindResponse(List<V> results);

    /**
     * Create a find response with the results representing the page request
     * and the count representing the size of the query.
     */
    protected abstract FR createFindResponse(List<V> results, long count);

    /**
     * Whether or not the primary key is valid (greater than zero).
     */
    protected boolean isPrimaryKeyValid(V request) {
        return DBUtil.isPrimaryKeyValid(request);
    }

    /**
     * Creates default sort.
     */
    private Sort createDefaultSort() {
        return new Sort("lastName", "firstName");
    }

}
```

Example 7 `AbstractPersistenceFindService`

The `AbstractPersistenceService` extends `AbstractPersistenceFindService` and adds the methods `create`/`update`/`delete`. The `create` & `update` methods are separated they can have different Spring Security [http://static.springsource.org/spring-security/site/] annotations applied, even though they both call into the same method for actually saving (`doSave`).

The `create`/`update`/`delete` methods also all are marked with `@Transactional` to override the default transactional configuration since they are not read-only transactions.

```
public abstract class AbstractPersistenceService<T extends AbstractPersistable<Integer>, V extends
PkEntityBase,
                                                  R extends EntityResponseResult, FR extends
EntityFindResponseResult>
        extends AbstractPersistenceFindService<T, V, R, FR>
        implements PersistenceService<V, R, FR> {

    protected static final String DELETE_MSG = "delete.msg";

    public AbstractPersistenceService(JpaRepository<T, Integer> repository, ListConverter<T, V>
converter,
                                      MessageHelper messageHelper) {
        super(repository, converter, messageHelper);
    }

    @Override
    @Transactional
    public R create(V request) {
        Assert.isTrue(!isPrimaryKeyValid(request), "Create should not have a valid primary key.");

        return doSave(request);
    }

    @Override
```

```
    @Transactional
    public final R update(V request) {
        Assert.isTrue(isPrimaryKeyValid(request), "Update should have a valid primary key.");

        return doSave(request);
    }

    @Override
    @Transactional
    public ResponseResult delete(Integer id) {
        return doDelete(id);
    }

    /**
     * Processes save.  Can be overridden for custom save logic.
     */
    protected R doSave(V request) {
        V result = null;

        T convertedRequest = converter.convertFrom(request);

        // issues with lock version updating if flush isn't called
        T bean = repository.saveAndFlush(convertedRequest);

        result = converter.convertTo(bean);

        return createSaveResponse(result);
    }

    /**
     * Processes delete. Can be overridden for custom save logic.
     */
    protected ResponseResult doDelete(long id) {
        repository.delete((int) id);
        repository.flush();

        return createDeleteResponse();
    }

    /**
     * Create a save response.
     */
    protected abstract R createSaveResponse(V result);

    protected ResponseResult createDeleteResponse() {
        return new ResponseResult().withMessageList(new Message().withMessageType(MessageType.INFO)
                .withMessageKey(DELETE_MSG).withMessage(getMessage(DELETE_MSG)));
    }

}
```

Example 8 `AbstractPersistenceService`

5. Contact Service Code Example

The `ContactService` could just extend the `PersistenceService`, but instead it overrides each method so every service can set it's own security rules. Without making any assumptions that they would all be consistent. If the application is known to have consistent security rules, they could be applied to the parent persistent interfaces and it wouldn't be necessary to apply them to each level.

```
public interface ContactService extends PersistenceService<Person, PersonResponse,
```

```
PersonFindResponse> {

    @Override
    @Secured ({ USER })
    public PersonResponse findById(Integer id);

    @Override
    @Secured ({ USER })
    public PersonFindResponse find();

    @Override
    @Secured ({ USER })
    public PersonFindResponse find(int page, int pageSize);

    @Override
    @Secured ({ USER })
    public PersonResponse create(Person person);

    @Override
    @Secured ({ USER })
    public PersonResponse update(Person person);

    @Override
    @Secured ({ ADMIN })
    public ResponseResult delete(Integer id);

}
```

Example 9 `ContactService`

The `ContactServiceImpl` is the implementation for the `ContactService`. It shows that if you needed to override the default behavior for `save`, the `doSave` method can be overridden. If the service wasn't also trying to handle saves/conversions for subclasses this wouldn't be necessary. Otherwise just the abstract methods for creating responses is needed.

```


@Service
public class ContactServiceImpl extends
AbstractPersistenceService<org.springbyexample.contact.orm.entity.person.Person, Person,
                                                                    PersonResponse,
PersonFindResponse>
        implements ContactService {

    private static final String SAVE_MSG = "contact.save.msg";

    private final StudentConverter studentConverter;
    private final ProfessionalConverter professionalConverter;

    @Autowired
    public ContactServiceImpl(PersonRepository repository,
                              ContactConverter converter,
                              StudentConverter studentConverter, ProfessionalConverter
professionalConverter,
                              MessageHelper messageHelper) {
        super(repository, converter, messageHelper);

        this.studentConverter = studentConverter;
        this.professionalConverter = professionalConverter;
    }

    @Override
    protected PersonResponse doSave(Person request) {
        org.springbyexample.contact.orm.entity.person.Person bean = null;
```

```
        if (request instanceof Student) {
            bean = studentConverter.convertFrom((Student) request);
        } else if (request instanceof Professional) {
            bean = professionalConverter.convertFrom((Professional) request);
        } else {
            bean = converter.convertFrom(request);
        }

        Person result = converter.convertTo(repository.saveAndFlush(bean));

        return createSaveResponse(result);
    }

    @Override
    protected PersonResponse createSaveResponse(Person result) {
        return new PersonResponse().withMessageList(new Message().withMessageType(MessageType.INFO)
                    .withMessage(getMessage(SAVE_MSG, new Object[] { result.getFirstName(),
result.getLastName()})))
                    .withResults(result);
    }

    @Override
    protected PersonResponse createResponse(Person result) {
        return new PersonResponse().withResults(result);
    }

    @Override
    protected PersonFindResponse createFindResponse(List<Person> results) {
        return new PersonFindResponse().withResults(results).withCount(results.size());
    }

    @Override
    protected PersonFindResponse createFindResponse(List<Person> results, long count) {
        return new PersonFindResponse().withResults(results).withCount(count);
    }

}
```

Example 10 `ContactServiceImpl`

6. Reference

Related Links

- Contact Application Web Service Beans
- Contact Application DAO
- Dozer [http://dozer.sourceforge.net/]
- Spring Security [http://static.springsource.org/spring-security/site/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-services
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Contact Application REST Services

David Winterfeldt

2012

REST services contains the Spring MVC Controllers and also the REST templates (clients) to access them. The clients are used in the unit tests. Also a client artifact is generated for other modules & external applications to use.

1. Spring Configuration

Spring MVC Configuration

The REST Controllers are loaded by *context:component-scan*, and a handler is created for them.

META-INF/spring/mvc/mvc-rest-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.contact.web.service"/>

    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"
/>

</beans>
```

Spring JSON Configuration

The Jackson JSON-processor [http://jackson.codehaus.org/] mapping and view are setup.

META-INF/spring/mvc/rest-json-converter-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       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.springframework.org/schema/context/spring-context.xsd
```

```
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">

    <bean id="abstractJacksonObjectMapper"
          class="org.springframework.beans.factory.config.MethodInvokingFactoryBean"
          abstract="true"
          p:targetMethod="disable">
        <property name="targetObject">
            <bean class="org.codehaus.jackson.map.ObjectMapper" />
        </property>
        <property name="arguments">
            <list>
                <util:constant
static-field="org.codehaus.jackson.map.DeserializationConfig.Feature.FAIL_ON_UNKNOWN_PROPERTIES"/>
            </list>
        </property>
    </bean>

    <bean id="abstractMappingJacksonHttpMessageConverter"
          class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"
          abstract="true"/>

    <bean id="abstractMappingJacksonJsonView"
          class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"
          abstract="true"
          p:extractValueFromSingleKeyModel="true"/>

    <bean id="jacksonObjectMapper" parent="abstractJacksonObjectMapper" />

    <bean id="mappingJacksonHttpMessageConverter"
          parent="abstractMappingJacksonHttpMessageConverter"
          p:objectMapper-ref="jacksonObjectMapper"
          p:supportedMediaTypes="application/json" />

    <bean id="mappingJacksonJsonView"
          parent="abstractMappingJacksonJsonView"
          p:objectMapper-ref="jacksonObjectMapper"
          p:contentType="application/json" />

</beans>
```

A custom JSON media type is configured that will serialize the JSON with class information. This is for more complex models to handle specifying specific instances of abstract classes. Ideally this custom media type can just be used by the UI and any external APIs will the standard media type that doesn't contain any Java class information.

META-INF/spring/mvc/rest-json-type-converter-context.xml

```


<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">

    <bean id="jacksonObjectMapperWithType" parent="abstractJacksonObjectMapper">
```

```
        <property name="targetObject">
            <bean class="org.springbyexample.web.servlet.mvc.json.RestObjectMapper">
                <property name="defaultTyping">
                    <bean class="org.springbyexample.web.servlet.mvc.json.ClassTypeResolverBuilder"
/>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="mappingJacksonHttpMessageConverterWithType"
          parent="abstractMappingJacksonHttpMessageConverter"
          p:objectMapper-ref="jacksonObjectMapperWithType"
          p:supportedMediaTypes="application/json-type" />

    <bean id="mappingJacksonJsonViewWithType"
          parent="abstractMappingJacksonJsonView"
          p:objectMapper-ref="jacksonObjectMapperWithType"
          p:contentType="application/json-type" />

</beans>
```

2. Persistence Marshalling Code

Marshalling service persistence interfaces that are separated into read-only and persistent operations, similar to the service layer.

The find persistence interface has constants for creating request paths and the basic find methods.

```


public interface PersistenceFindMarshallingService<R extends EntityResponseResult, FR extends
EntityFindResponseResult> {

    public final static String PATH_DELIM = "/";

    public final static String ID_VAR = "id";
    public final static String PAGE_VAR = "page";
    public final static String PAGE_SIZE_VAR = "page-size";

    public final static String PAGE_PATH = PATH_DELIM + PAGE_VAR;
    public final static String PAGE_SIZE_PATH = PATH_DELIM + PAGE_SIZE_VAR;
    public final static String PAGINATED = PAGE_PATH + PATH_DELIM + "{" + PAGE_VAR + "}" +
PAGE_SIZE_PATH + PATH_DELIM + "{" + PAGE_SIZE_VAR + "}";


    /**
     * Find by primary key.
     */
    public R findById(long id);

    /**
     * Find a paginated record set.
     */
    public FR find(int page, int pageSize);

    /**
     * Find all records.
     */
    public FR find();

}
```

```
```

Example 1 `PersistenceFindMarshallingService`

Save and delete are both overloaded so they would continue to work with existing backend tests, but also automatically work with Sencha stores.

```
public interface PersistenceMarshallingService<R extends EntityResponseResult, FR extends
EntityFindResponseResult, S extends PkEntityBase>
        extends PersistenceFindMarshallingService<R, FR> {

    /**
     * Save record.
     */
    public R save(S request);

    /**
     * Update record.
     */
    public R update(S request);

    /**
     * Delete record.
     */
    public ResponseResult delete(long id);

    /**
     * Delete record.
     */
    public ResponseResult delete(S request);

}
```

Example 2 `PersistenceMarshallingService`

3. Contact REST Code Example

The contact controller just needs to have an interface that extends `PersistenceMarshallingService` and a controller that implements that interface.

The marshalling service extends `PersistenceMarshallingService` and defines all the paths for the controller as constants.

```
public interface PersonMarshallingService extends PersistenceMarshallingService<PersonResponse,
PersonFindResponse, Person> {

    final static String PATH = "/person";

    public final static String FIND_BY_ID_REQUEST = PATH + PATH_DELIM + "{" + ID_VAR + "}";
    public final static String FIND_PAGINATED_REQUEST = PATH + PAGINATED;
    public final static String FIND_REQUEST = PATH;
    public final static String SAVE_REQUEST = PATH;
    public final static String UPDATE_REQUEST = FIND_BY_ID_REQUEST;
    public final static String DELETE_PK_REQUEST = FIND_BY_ID_REQUEST;
    public final static String DELETE_REQUEST = PATH;
```

```
}
```

Example 3 `PersonMarshallingService`

The controller primarily delegates to it's service and also defines a `@RequestMapping` for each method.

```
@Controller
public class PersonController extends AbstractController<Person, PersonResponse, PersonFindResponse>
        implements PersonMarshallingService {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    public PersonController(ContactService service) {
        super(service);
    }

    @Override
    @RequestMapping(value = FIND_BY_ID_REQUEST, method = RequestMethod.GET)
    public PersonResponse findById(@PathVariable(ID_VAR) long id) {
        logger.info("Find person.  id={}", id);

        return service.findById((int)id);
    }

    @Override
    @RequestMapping(value = FIND_PAGINATED_REQUEST, method = RequestMethod.GET)
    public PersonFindResponse find(@PathVariable(PAGE_VAR) int page,
                                   @PathVariable(PAGE_SIZE_VAR) int pageSize) {
        logger.info("Find person page.  page={}  pageSize={}", page, pageSize);

        return service.find(page, pageSize);
    }

    @Override
    @RequestMapping(value = FIND_REQUEST, method = RequestMethod.GET)
    public PersonFindResponse find() {
        logger.info("Find all persons.");

        return service.find();
    }

    @Override
    @RequestMapping(value = SAVE_REQUEST, method = RequestMethod.POST)
    public PersonResponse save(@RequestBody Person request) {
        Assert.isTrue(!isPrimaryKeyValid(request), "Create should not have a valid primary key.");

        logger.info("Save person.  id={}", request.getId());

        return service.create(request);
    }

    @Override
    @RequestMapping(value = UPDATE_REQUEST, method = RequestMethod.PUT)
    public PersonResponse update(@RequestBody Person request) {
        Assert.isTrue(isPrimaryKeyValid(request), "Update should have a valid primary key.");

        logger.info("Update person.  id={}", request.getId());

        return service.update(request);
    }

    @Override
    @RequestMapping(value = DELETE_PK_REQUEST, method = RequestMethod.DELETE)
```

```
    public ResponseResult delete(@PathVariable(ID_VAR) long id) {
        logger.info("Delete person.  id={}", id);

        return service.delete((int)id);
    }

    @Override
    @RequestMapping(value = DELETE_REQUEST, method = RequestMethod.DELETE)
    public ResponseResult delete(@RequestBody Person request) {
        Assert.isTrue((request.getId() > 0), "Delete should have a valid primary key");

        int id = request.getId();

        return delete((int)id);
    }

}
```

Example 4 `PersonController`

4. REST Client

Spring Configuration

The REST client configuration creates a `RestTemplate` and leverages the same marshallers that the server uses. The default configuration uses the JAXB marshaller, but if the Spring Profile for JSON is activated the JSON marshaller will be used for client requests.

META-INF/spring/client/rest-client-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">

    <import resource="classpath:/META-INF/spring/marshaller/jaxb2-marshaller-context.xml"/>

    <context:component-scan base-package="org.springbyexample.contact.web.client" />

    <bean id="httpClient" class="org.apache.http.impl.client.DefaultHttpClient">
        <constructor-arg>
            <bean class="org.apache.http.impl.conn.PoolingClientConnectionManager"/>
        </constructor-arg>
    </bean>

    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate"
          p:messageConverters-ref="messageConvertersList">
        <constructor-arg>
            <bean class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
                <constructor-arg ref="httpClient"/>
            </bean>
```

```
        </constructor-arg>
    </bean>

    <util:list id="messageConvertersList">
        <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter"
              p:supportedMediaTypes="application/xml">
            <property name="marshaller" ref="marshaller" />
            <property name="unmarshaller" ref="marshaller" />
        </bean>
    </util:list>

    <beans profile="rest-json">
        <import resource="classpath:/META-INF/spring/mvc/rest-json-converter-context.xml"/>

        <!-- since id is the same as XML list above, essentially overriding the other list -->
        <util:list id="messageConvertersList">
            <ref bean="mappingJacksonHttpMessageConverter"/>
        </util:list>
    </beans>

</beans>
```

Client Code

The `RestClient` configures the `RestTemplate` with the default credentials, and also can create URLs from a URI.

```


@Component
public class RestClient {

    final Logger logger = LoggerFactory.getLogger(getClass());

    private final RestTemplate template;
    private final RestClientProperties clientProperties;
    private final DefaultHttpClient httpClient;

    @Autowired
    public RestClient(RestTemplate template, RestClientProperties clientProperties,
                      DefaultHttpClient httpClient) {
        this.template = template;
        this.clientProperties = clientProperties;
        this.httpClient = httpClient;
    }

    @PostConstruct
    public void init() {
        setCredentials(clientProperties.getUsername(), clientProperties.getPassword());
    }

    /**
     * Gets rest template.
     */
    public RestTemplate getRestTemplate() {
        return template;
    }

    /**
     * Creates URL based on the URI passed in.
     */
    public String createUrl(String uri) {
```

```
        StringBuilder sb = new StringBuilder();

        sb.append(clientProperties.getUrl());
        sb.append(clientProperties.getApiPath());
        sb.append(uri);

        logger.debug("URL is '{}'.", sb.toString());

        return sb.toString();
    }

    /**
     * Set default credentials on HTTP client.
     */
    public void setCredentials(String userName, String password) {
        UsernamePasswordCredentials creds =
                new UsernamePasswordCredentials(clientProperties.getUsername(),
clientProperties.getPassword());
        AuthScope authScope = new AuthScope(AuthScope.ANY_HOST, AuthScope.ANY_PORT,
AuthScope.ANY_REALM);

        httpClient.getCredentialsProvider().setCredentials(authScope, creds);
    }


}
```

Example 5 `RestClient`

Abstract persistence client base for read-only operations.

```


public abstract class AbstractPersistenceFindClient<R extends EntityResponseResult, FR extends
EntityFindResponseResult>
        implements PersistenceFindMarshallingService<R, FR> {

    final Logger logger = LoggerFactory.getLogger(getClass());

    protected final RestClient client;

    private final String findByIdRequest;
    private final String findPaginatedRequest;
    private final String findRequest;
    protected final Class<R> responseClazz;
    protected final Class<FR> findResponseClazz;


    public AbstractPersistenceFindClient(RestClient client,
                                         String findByIdRequest, String findPaginatedRequest, String
findRequest,
                                         Class<R> responseClazz, Class<FR> findResponseClazz) {
        this.client = client;
        this.findByIdRequest = findByIdRequest;
        this.findPaginatedRequest = findPaginatedRequest;
        this.findRequest = findRequest;
        this.responseClazz = responseClazz;
        this.findResponseClazz = findResponseClazz;
    }

    @Override
    public R findById(long id) {
        R response = null;

        String url = client.createUrl(findByIdRequest);

        logger.debug("REST client findById.  id={}  url='{}'", id, url);
```

```
        response = client.getRestTemplate().getForObject(url, responseClazz, createPkVars(id));

        return response;
    }

    @Override
    public FR find(int page, int pageSize) {
        FR response = null;

        String url = client.createUrl(findPaginatedRequest);

        logger.debug("REST client paginated find.  page={}  pageSize={}  url='{}'",
                     new Object[] { page, pageSize, url});

        response = client.getRestTemplate().getForObject(url, findResponseClazz,
createPageVars(page, pageSize));

        return response;
    }

    @Override
    public FR find() {
        FR response = null;

        String url = client.createUrl(findRequest);

        logger.debug("REST client find. url='{}'", url);

        response = client.getRestTemplate().getForObject(url, findResponseClazz);

        return response;
    }

    /**
     * Create primary key request variables.
     */
    public Map<String, Long> createPkVars(long id) {
        return Collections.singletonMap(ID_VAR, id);
    }

    /**
     * Create page vars for a paginated request.
     */
    public Map<String, Integer> createPageVars(int page, int pageSize) {
        Map<String, Integer> result = new HashMap<String, Integer>();

        result.put(PAGE_VAR, page);
        result.put(PAGE_SIZE_VAR, pageSize);

        return result;
    }

}
```

Example 6 `AbstractPersistenceFindClient`

Abstract persistence client base for persistent operations.

```


public abstract class AbstractPersistenceClient<R extends EntityResponseResult, FR extends
EntityFindResponseResult, S extends PkEntityBase>
        extends AbstractPersistenceFindClient<R, FR>
        implements PersistenceMarshallingService<R, FR, S> {

    private final String saveRequest;
```

```
    private final String updateRequest;
    private final String deletePkRequest;
    private final String deleteRequest;

    public AbstractPersistenceClient(RestClient client,
                                     String findByIdRequest, String findPaginatedRequest, String
findRequest,
                                     String saveRequest, String updateRequest,
                                     String deletePkRequest, String deleteRequest,
                                     Class<R> responseClazz, Class<FR> findResponseClazz) {
        super(client,
              findByIdRequest, findPaginatedRequest, findRequest,
              responseClazz, findResponseClazz);

        this.saveRequest = saveRequest;
        this.updateRequest = updateRequest;
        this.deletePkRequest = deletePkRequest;
        this.deleteRequest = deleteRequest;
    }

    @Override
    public R save(S request) {
        R response = null;

        String url = client.createUrl(saveRequest);

        logger.debug("REST client save.  id={}  url='{}'", request.getId(), url);

        response =  client.getRestTemplate().postForObject(url, request, responseClazz);

        return response;
    }

    @Override
    public R update(S request) {
        R response = null;

        String url = client.createUrl(updateRequest);

        logger.debug("REST client update.  id={}  url='{}'", request.getId(), url);

        Map<String, Long> vars = createPkVars(request.getId());

        response = client.getRestTemplate().exchange(url, HttpMethod.PUT, null, responseClazz,
vars).getBody();

        return response;
    }

    @Override
    public ResponseResult delete(long id) {
        ResponseResult response = null;

        String url = client.createUrl(deletePkRequest);

        logger.debug("REST client delete.  id={}  url='{}'", id, url);

        Map<String, Long> vars = createPkVars(id);

        response = client.getRestTemplate().exchange(url, HttpMethod.DELETE, null,
ResponseResult.class, vars).getBody();

        return response;
    }

    @Override
    public ResponseResult delete(S request) {
        ResponseResult response = null;

        String url = client.createUrl(deleteRequest);

        int id = request.getId();
```

```
        logger.debug("REST client delete.  id={}  url='{}'", id, url);

        response = client.getRestTemplate().exchange(url, HttpMethod.DELETE, null,
ResponseResult.class).getBody();

        return response;
    }

}
```

Example 7 `AbstractPersistenceClient`

The abstract base classes handle everything once all the URIs and expected responses are correctly set in the constructor.

```
@Component
public class PersonClient extends AbstractPersistenceClient<PersonResponse, PersonFindResponse,
Person>
        implements PersonMarshallingService {

    @Autowired
    public PersonClient(RestClient client) {
        super(client,
              FIND_BY_ID_REQUEST, FIND_PAGINATED_REQUEST, FIND_REQUEST,
              SAVE_REQUEST, UPDATE_REQUEST, DELETE_PK_REQUEST, DELETE_REQUEST,
              PersonResponse.class, PersonFindResponse.class);
    }

}
```

Example 8 `PersonClient`

5. Reference

Related Links

- Contact Application Web Service Beans
- Contact Application DAO
- Contact Application Services
- Jackson Java JSON-processor [http://jackson.codehaus.org/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-rest-services
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Contact Application Webapp

David Winterfeldt

2012

The web application builds on all the other modules, so has very little configuration of it's own. It has a JSP UI, Sencha ExtJS [http://www.sencha.com/products/extjs/], and also a Sencha Touch [http://www.sencha.com/products/touch/] UI. The latter Sencha UIs both use the REST services.

1. Web Configuration

Two servlets are configured. The first one configures a JSP application and the second one is for the REST API.

Excerpt from */WEB-INF/web.xml*

```
<servlet>
    <servlet-name>simple-form</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/mvc/jsp-servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet>
    <servlet-name>spring-mvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/mvc/rest-servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
    <async-supported>true</async-supported>
</servlet>

<servlet-mapping>
    <servlet-name>simple-form</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>spring-mvc</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

2. Spring Configuration

All of the Spring configuration files in the root directory of 'WEB-INF/spring' are loaded in the root Spring context.

This configuration is for loading anything web related into the root context, in this case it's just the web security.

WEB-INF/spring/web-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath:/META-INF/spring/web/web-security-context.xml"/>

</beans>
```

Most of the main Spring configuration files from the different modules are loaded. Basic security, marshallers, DB related code, and the services layer.

WEB-INF/spring/services-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath*:/META-INF/spring/security/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/marshaller/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/db/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/services/**/*-context.xml"/>

</beans>
```

The REST API servlet configuration loads the REST controllers and their views & handlers.

WEB-INF/spring/mvc/rest-servlet-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath*:/META-INF/spring/mvc/**/*-context.xml"/>
```

```
    <import resource="classpath:/META-INF/spring/web/web-rest-servlet-context.xml"/>

</beans>
```

The JSP servlet loads the JSP controllers and configures it's static and dynamic resources. Wild card mappings are setup to serve all resources for Sencha ExtJS [http://www.sencha.com/products/extjs/] & Sencha Touch [http://www.sencha.com/products/touch/].

Also a static content zip is generated. So if a server like Apache is proxying the servlet container, it could also be configured to serve the static content.

WEB-INF/spring/mvc/jsp-servlet-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="org.springbyexample.contact.web.servlet.mvc" />

    <mvc:annotation-driven />

    <mvc:view-controller path="/index.html" />
    <mvc:view-controller path="/login.html" />
    <mvc:view-controller path="/logoutSuccess.html" />

    <mvc:resources mapping="/extjs/**" location="/extjs/" />
    <mvc:resources mapping="/touch/**" location="/touch/" />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
          p:definitions="/WEB-INF/tiles-defs/templates.xml" />

    <bean id="tilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:viewClass="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp" />

    <!-- Declare the Interceptor -->
    <mvc:interceptors>
        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
              p:paramName="locale" />
    </mvc:interceptors>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

3. Code Example

For the most part, the web application just takes all the other modules and their Spring config to load them in the webapp. The one exception is the implementation of the `ApplicationContextInitializer` that checks if anything has been set for the Spring Profiles system property. If nothing is set, the HSQL DB profile is used as the default Spring Profile. For production the PostgreSQL profile should be set.

```
public class ContactApplicationContextInitializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    private static final String SPRING_PROFILES_ACTIVE_PROPERTY = "spring.profiles.active";

    private static final String PROFILE_HSQL = "hsql";

    private final static String [] DEFAULT_ACTIVE_PROFILES = { PROFILE_HSQL };

    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        String springProfilesActive = System.getProperty(SPRING_PROFILES_ACTIVE_PROPERTY);

        if (StringUtils.hasText(springProfilesActive)) {
            logger.info("Using set spring profiles.  profiles='{}'", springProfilesActive);
        } else {
            applicationContext.getEnvironment().setActiveProfiles(DEFAULT_ACTIVE_PROFILES);

            logger.info("Setting default spring profiles.  profiles='{}'", DEFAULT_ACTIVE_PROFILES);
        }
    }

}
```

Example 1 `ContactApplicationContextInitializer`

4. Reference

Related Links

- Contact Application Web Service Beans
- Contact Application DAO
- Contact Application Services
- Contact Application REST Services
- Simple Spring Security Webapp

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

DB Setup Instructions

To setup the webapp to run with PostgreSQL, follow the steps below.

- Install PostgreSQL 9.0
- Create this file in your home directory so you don't have to enter the DB password.

  .pgpass

```
*:*:*:postgres:password
*:*:*:contact:contact
localhost:5432:*:postgres:password
```

- Initialize the DB:

```
$ cd app/contact-app
```

```
$ ./init-postgres.sh
```

  or for re-initializing an existing DB:

```
$ cd app/contact-app
```

```
$ ./reinit-postgres.sh
```

- Set the Java System Property when starting the servlet container for the Spring Profile.

  -Dspring.profiles.active=postgresql

- Create a property file with the JDBC URL to PostgreSQL in the directory where the servlet container is started.

  contact-app.properties

```
jdbc.url=jdbc:postgresql://$postgres_ip:5432/contact
```

Project Information

- Spring Framework 3.1.x

Contact Application Test

David Winterfeldt

2012

The DAO, Services, and REST Services all have a shared Spring Test context setup when the abstract test base for each module is extended. This will improve overall test performance since any setup by Spring will only be performed once for a suite of tests.

There is a contact-test module that is used to share abstract test base classes, constants used across layers, and any shared config. This is used instead of having each module generate it's own test artifact. When each there are a lot of test artifacts, inter-module dependencies can become more complicated.

1. Abstract Test Code

These are shared abstract base classes for tests in different modules to extend.

Abstract Code

The `AbstractProfileTest` sets up the main JUnit Spring test runner and sets the default Spring Active Profile to use the HSQL DB. All of the tests from each module extend this class.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles(profiles= { HSQL })
public abstract class AbstractProfileTest {

    /**
     * Setup the security context before each test.
     */
    @Before
    public void setUp() {
        doInit();
    }

    /**
     * Clear the security context after each test.
     */
    @After
    public void tearDown() {
        SecurityContextHolder.clearContext();
    }

    /**
     * Set the default user on the security context.
     */
    protected abstract void doInit();

}
```

Example 1 `AbstractProfileTest`

The DAO and Service tests extend this abstract class. It extends `AbstractProfileTest`, sets up the Spring transactional test configuration, and also configures a default security context. The Spring transactional test configuration provides automatic transaction rollback as each individual test finishes. Having the security context set before each test is to reset it to the default in case an individual test made any changes to test different users.

```
@TransactionConfiguration
@Transactional
public abstract class AbstractTransactionalProfileTest extends AbstractProfileTest {

    /**
     * Set the default user on the security context.
     */
    protected void doInit() {
        SecurityContextHolder.getContext().setAuthentication(
                new UsernamePasswordAuthenticationToken(DEFAULT_SECURITY_USER,
DEFAULT_SECURITY_USER_PASSWORD));
    }

}
```

Example 2 `AbstractTransactionalProfileTest`

2. DAO Test

The DAO tests and their test base are for testing the JPA entities & Spring Data JPA [http://www.springsource.org/spring-data/jpa] repositories against an in memory database.

Spring Configuration

The DAO test config is very simple and just loads all of the DB Spring configuration files. All Spring XML configuration files end in *-context.xml* and test ones end in *-test-context.xml*

dao-test-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath*:/META-INF/spring/db/**/*-context.xml"/>

</beans>
```

Abstract Code

Most of the test configuration was already configured in the parent test classes. All that is left is to configure the test Spring configuration to use by setting the `@ContextConfiguration`. As each test extending this class runs, it will automatically use this shared Spring test context.

```
@ContextConfiguration({ "classpath:/dao-test-context.xml" })
public abstract class AbstractRepositoryTest extends AbstractTransactionalProfileTest {

}
```

Example 3 `AbstractRepositoryTest`

Code Example

The person repository test is for the Spring Data JPA [http://www.springsource.org/spring-data/jpa] `PersonRepository`. There is in progress work for having `Person` subclasses work through the entire stack.

```
public class PersonRepositoryTest extends AbstractRepositoryTest {

    final Logger logger = LoggerFactory.getLogger(PersonRepositoryTest.class);

    @Autowired
    private PersonRepository personRepository;

    @Autowired
    private ProfessionalRepository professionalRepository;

    @Test
    public void testFindOne() {
        Person person = personRepository.findOne(FIRST_ID);

        testPersonOne(person);
    }

    @Test
    public void testFindAll() {
        Collection<Person> persons = personRepository.findAll();

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + EXPECTED_COUNT + ".", EXPECTED_COUNT,
persons.size());

        for (Person person : persons) {
            logger.debug(person.toString());

            if (FIRST_ID.equals(person.getId())) {
                testPersonOne(person);
            } else if (SECOND_ID.equals(person.getId())) {
                testPersonTwo(person);
            }
        }
    }

    @Test
    public void testFindByFirstNameLike() {
        List<Person> persons = personRepository.findByFirstNameLike("J%");
```

```
        int expectedCount = 2;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

        Person person = persons.get(0);

        testPersonOne(person);
    }

    @Test
    public void testFindByLastName() {
        List<Person> persons = personRepository.findByLastName(LAST_NAME);

        int expectedCount = 1;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

        Person person = persons.get(0);

        testPersonOne(person);
    }

    @Test
    public void testFindByAddress() {
        List<Person> persons = personRepository.findByAddress(ADDR);

        int expectedCount = 1;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

        Person person = persons.get(0);

        testPersonOne(person);
    }

    @Test
    public void testFindByAddressPage() {
        String firstName = "Jack";
        String lastName = "Johnson";
        String companyName = "Spring Pizza";

        int page = 0;
        int size = 10;

        for (int i = 0; i < 35; i++) {
            createProfessional(firstName, lastName, companyName, ADDR);
        }

        Page<Person> pageResult = personRepository.findByAddress(ADDR, new PageRequest(page, size));
        List<Person> persons = pageResult.getContent();

        int expectedCount = size;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

        // query last page
        page = pageResult.getTotalPages() - 1;
        pageResult = personRepository.findByAddress(ADDR, new PageRequest(page, size));
        persons = pageResult.getContent();

        // created 35 records with the same address, one existing
        expectedCount = 6;
```

```
        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

    }

    @Test
    public void testFindByName() {
        List<Person> persons = personRepository.findByName(FIRST_NAME, LAST_NAME);

        int expectedCount = 1;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());

        Person person = persons.get(0);

        testPersonOne(person);
    }

    @Test
    public void testSave() {
        String firstName = "Jack";
        String lastName = "Johnson";
        String companyName = "Spring Pizza";

        Person person = createProfessional(firstName, lastName, companyName, ADDR);

        // get PK of first address
        Integer addressId = person.getAddresses().iterator().next().getId();

        // test saved person
        testPerson(person,
                   firstName, lastName,
                   EXPECTED_ADDRESS_COUNT, addressId, ADDR, CITY, STATE, ZIP_POSTAL,
                   true, companyName);

        person = professionalRepository.findOne(person.getId());

        // test retrieved person just saved
        testPerson(person,
                   firstName, lastName,
                   EXPECTED_ADDRESS_COUNT, addressId, ADDR, CITY, STATE, ZIP_POSTAL,
                   true, companyName);

        Collection<Person> persons = personRepository.findAll();

        int expectedCount = EXPECTED_COUNT + 1;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());
    }

    @Test
    public void testUpdate() {
        Person person = personRepository.findOne(FIRST_ID);
        testPersonOne(person);

        String lastName = "Jones";
        person.setLastName(lastName);

        personRepository.saveAndFlush(person);

        person = personRepository.findOne(FIRST_ID);
        testPersonOne(person, lastName);
    }

    @Test
    public void testDelete() {
```

```
        personRepository.delete(FIRST_ID);

        // person should be null after delete
        Person person = personRepository.findOne(FIRST_ID);
        assertNull("Person is not null.", person);
    }

    /**
     * Create professional.
     */
    private Person createProfessional(String firstName, String lastName, String companyName,
                                      String addr) {
        Professional person = new Professional();
        Set<Address> addresses = new HashSet<Address>();
        Address address = new Address();
        addresses.add(address);

        address.setAddress(addr);
        address.setCity(CITY);
        address.setState(STATE);
        address.setZipPostal(ZIP_POSTAL);
        address.setCountry(COUNTRY);

        person.setFirstName(firstName);
        person.setLastName(lastName);
        person.setCompanyName(companyName);
        person.setAddresses(addresses);

        Person result = personRepository.saveAndFlush(person);

        return result;
    }

    /**
     * Tests person with a PK of one.
     */
    private void testPersonOne(Person person) {
        testPersonOne(person, LAST_NAME);
    }

    /**
     * Tests person with a PK of one.
     */
    private void testPersonOne(Person person, String lastName) {
        String schoolName = "NYU";

        Integer addressId = new Integer(1);

        testPerson(person,
                   FIRST_NAME, lastName,
                   EXPECTED_ADDRESS_COUNT, addressId, ADDR, CITY, STATE, ZIP_POSTAL,
                   false, schoolName);
    }

    /**
     * Tests person with a PK of two.
     */
    private void testPersonTwo(Person person) {
        String firstName = "John";
        String lastName = "Wilson";
        String companyName = "Spring Pizza";

        int expectedAddresses = 2;

        Integer addressId = new Integer(3);
        String addr = "47 Howard St.";
        String city = "San Francisco";
        String state = "CA";
        String zipPostal = "94103";

        testPerson(person,
```

```
                    firstName, lastName,
                    expectedAddresses, addressId, addr, city, state, zipPostal,
                    true, companyName);
    }

    /**
     * Tests person.
     */
    private void testPerson(Person person,
                            String firstName, String lastName,
                            int expectedAddresses, Integer addressId,
                            String addr, String city, String state, String zipPostal,
                            boolean professional, String professionName) {
        assertNotNull("Person is null.", person);

        assertEquals("firstName", firstName, person.getFirstName());
        assertEquals("lastName", lastName, person.getLastName());

        assertNotNull("Person's address list is null.", person.getAddresses());
        assertEquals("addresses", expectedAddresses, person.getAddresses().size());

...

        for (Address address : person.getAddresses()) {
            assertNotNull("Address is null.", address);

            if (addressId.equals(address.getId())) {
                assertEquals("address.id", addressId, address.getId());
                assertEquals("address.addr", addr, address.getAddress());

                assertEquals("address.city", city, address.getCity());
                assertEquals("address.state", state, address.getState());
                assertEquals("address.zipPostal" + zipPostal + "'.", zipPostal,
address.getZipPostal());
                assertEquals("address.country", COUNTRY, address.getCountry());

                testAuditable(address);
            }
        }

        testAuditable(person);
    }

    /**
     * Tests auditable entity.
     */
    private void testAuditable(AbstractAuditableEntity auditRecord) {
        assertNotNull("lastUpdated", auditRecord.getLastModifiedDate());
        assertNotNull("lastUpdatedBy", auditRecord.getLastModifiedBy());
        assertNotNull("created", auditRecord.getCreatedDate());
        assertNotNull("createdBy", auditRecord.getCreatedBy());
    }

}
```

Example 4 `PersonRepositoryTest`

3. Services Test

The Services tests and their test base are for testing the services use of the Spring Data JPA [http://www.springsource.org/spring-data/jpa] repositories, conversion between the entity and ws bean models, and business logic (if any). Also security and transactional configuration are loaded and used during the tests. All database operations are performed against an in memory database.

Spring Configuration

The Services Spring test configuration loads the main security, marshaller, DB, and services config. It doesn't use any special test configuration or mocks, but anything different from the standard production Spring configuration could be added here.

services-test-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath*:/META-INF/spring/security/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/marshaller/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/db/**/*-context.xml"/>
    <import resource="classpath*:/META-INF/spring/services/**/*-context.xml"/>

</beans>
```

Abstract Code

The `AbstractServiceTest` extends the shared transactional profile test abstract class. It sets the shared test configuration to *services-test-context.xml*.

```
@ContextConfiguration({ "classpath:/services-test-context.xml" })
public abstract class AbstractServiceTest extends AbstractTransactionalProfileTest {

}
```

Example 5 `AbstractServiceTest`

Code Example

The `ContactServiceTest` tests the basic usage of the `ContactService`. This primarily covers usage of it's JPA repository and conversion to and from entity beans & ws beans.

```
public class ContactServiceTest extends AbstractServiceTest {

    final Logger logger = LoggerFactory.getLogger(ContactServiceTest.class);

    @Autowired
    private ContactService service;
```

```
    @Test
    public void testFindOne() {
        PersonResponse response = service.findById(FIRST_ID);
        Person person = response.getResults();

        testPersonOne(person);
    }

    @Test
    public void testFindAll() {
        PersonFindResponse response = service.find();
        assertNotNull("Person response is null.", response);

        Collection<Person> persons = response.getResults();

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + EXPECTED_COUNT + ".", EXPECTED_COUNT,
persons.size());

        for (Person person : persons) {
            logger.debug(person.toString());

            if (FIRST_ID.equals(person.getId())) {
                testPersonOne(person);
            } else if (SECOND_ID.equals(person.getId())) {
                testPersonTwo(person);
            }
        }
    }

    @Test
    public void testCreate() {
        String firstName = "Jack";
        String lastName = "Johnson";

        PersonResponse response = createPerson(firstName, lastName);
        assertNotNull("Person response is null.", response);

        Person person = response.getResults();

        // test saved person
        testPerson(person,
                   firstName, lastName);

        PersonFindResponse findResponse = service.find();
        assertNotNull("Person response is null.", findResponse);

        Collection<Person> persons = findResponse.getResults();

        int expectedCount = EXPECTED_COUNT + 1;

        assertNotNull("Person list is null.", persons);
        assertEquals("Number of persons should be " + expectedCount + ".", expectedCount,
persons.size());
    }

    @Test
    public void testUpdate() {
        PersonResponse response = service.findById(FIRST_ID);
        assertNotNull("Person response is null.", response);

        Person person = response.getResults();

        testPersonOne(person);

        String lastName = "Jones";
        person.setLastName(lastName);

        service.update(person);
```

```
        response = service.findById(FIRST_ID);
        assertNotNull("Person response is null.", response);

        person = response.getResults();

        testPersonOne(person, lastName);
    }

    @Test
    public void testDelete() {
        service.delete(FIRST_ID);

        // person should be null after delete
        PersonResponse response = service.findById(FIRST_ID);
        assertNotNull("Person response is null.", response);

        Person person = response.getResults();

        assertNull("Person is not null.", person);
    }

    /**
     * Create person.
     */
    private PersonResponse createPerson(String firstName, String lastName) {
        Person person = new Person();

        person.setFirstName(firstName);
        person.setLastName(lastName);

        PersonResponse response = service.create(person);

        return response;
    }

    /**
     * Tests person with a PK of one.
     */
    private void testPersonOne(Person person) {
        testPersonOne(person, LAST_NAME);
    }

    /**
     * Tests person with a PK of one.
     */
    private void testPersonOne(Person person, String lastName) {
        testPerson(person,
                   FIRST_NAME, lastName);
    }

    /**
     * Tests person with a PK of two.
     */
    private void testPersonTwo(Person person) {
        String firstName = "John";
        String lastName = "Wilson";

        testPerson(person,
                   firstName, lastName);
    }

    /**
     * Tests person.
     */
    private void testPerson(Person person,
                            String firstName, String lastName) {
        assertNotNull("Person is null.", person);

        assertEquals("firstName", firstName, person.getFirstName());
        assertEquals("lastName", lastName, person.getLastName());
```

```
        testAuditable(person);
    }

    /**
     * Tests auditable entity.
     */
    private void testAuditable(PkEntityBase auditRecord) {
        assertNotNull("lastUpdated", auditRecord.getLastUpdated());
        assertNotNull("lastUpdatedBy", auditRecord.getLastUpdateUser());
        assertNotNull("created", auditRecord.getCreated());
        assertNotNull("createdBy", auditRecord.getCreateUser());
    }

}
```

Example 6 `ContactServiceTest`

4. REST Services Test

The REST Services tests and their test base are for testing the REST clients and services configuration, as well as their calls into the service layer. There is an in memory database and controllers are loaded in an embedded jetty server.

Spring Configuration

There are actually two main Spring contexts. One is the standard test context that just loads the REST clients and a properties file to configure the path to the REST APIs and any other client configuration information.

The other context is loaded in the `EmbeddedJetty` bean. It takes the list of Spring configuration files and loads them in a parent context, then a servlet child context is created. It also registers the Spring Security filter with the servlet handler.

rest-controller-test-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">

    <import resource="classpath*:/META-INF/spring/client/**/*-context.xml"/>

    <util:properties id="restProperties"
                     location="org/springbyexample/contact/web/service/ws.properties" />

    <bean class="org.springbyexample.web.service.EmbeddedJetty">
        <constructor-arg>
            <list>
                <value>/embedded-jetty-context.xml</value>
```

```
                <value>/META-INF/spring/security/**/*-context.xml</value>
                <value>/META-INF/spring/marshaller/**/*-context.xml</value>
                <value>/META-INF/spring/db/**/*-context.xml</value>
                <value>/META-INF/spring/services/**/*-context.xml</value>
                <value>/META-INF/spring/mvc/**/*-context.xml</value>

                <value>/mock-web-security-context.xml</value>
            </list>
        </constructor-arg>
    </bean>

</beans>
```

Abstract Code

The `AbstractRestControllerTest` sets up the shared Spring test configuration, and before each test resets the DB by re-initializing the schema and clearing the JPA entity manager cache.

```


@ContextConfiguration({
"classpath:/org/springbyexample/contact/web/service/rest-controller-test-context.xml" })
public abstract class AbstractRestControllerTest extends AbstractProfileTest {

    final Logger logger = LoggerFactory.getLogger(AbstractRestControllerTest.class);

    @Autowired
    private EmbeddedJetty embeddedJetty;

    /**
     * Reset the DB before each test.
     */
    protected void doInit() {
        reset();
    }

    /**
     * Reset the database and entity manager cache.
     */
    protected void reset() {
        resetSchema();
        resetCache();

        logger.info("DB schema and entity manager cache reset.");
    }

    /**
     * Resets DB schema.
     */
    private void resetSchema() {
        ApplicationContext ctx = embeddedJetty.getApplicationContext();
        DataSource dataSource = ctx.getBean(DataSource.class);
        @SuppressWarnings("unchecked")
        List<Resource> databaseScripts = (List<Resource>) ctx.getBean("databaseScriptsList");

        Connection con = null;
        ResourceDatabasePopulator resourceDatabasePopulator = new ResourceDatabasePopulator();

        try {
            con = dataSource.getConnection();

            resourceDatabasePopulator.setScripts(databaseScripts.toArray(new Resource[0]));

            resourceDatabasePopulator.populate(con);
```

```
        } catch (SQLException e) {
            logger.error(e.getMessage(), e);
        } finally {
            try { con.close(); } catch (Exception e) {}
        }
    }

    /**
     * Reset cache.
     */
    private void resetCache() {
        ApplicationContext ctx = embeddedJetty.getApplicationContext();
        EntityManagerFactory entityManagerFactory = ctx.getBean(EntityManagerFactory.class);

        Cache cache = entityManagerFactory.getCache();

        if (cache != null) {
            cache.evictAll();
        }
    }

}
```

Example 7 `AbstractRestControllerTest`

The `AbstractPersistenceFindControllerTest` provides an abstract base for testing a `PersistenceFindMarshallingService`.

```


public abstract class AbstractPersistenceFindControllerTest<R extends EntityResponseResult, FR
extends EntityFindResponseResult, S extends PkEntityBase>
        extends AbstractRestControllerTest {

    protected final Logger logger = LoggerFactory.getLogger(getClass());

    protected final int id;
    protected final long expectedCount;

    public AbstractPersistenceFindControllerTest(int id, long expectedCount) {
        this.id = id;
        this.expectedCount = expectedCount;
    }

    /**
     * Gets find client.
     */
    protected abstract PersistenceFindMarshallingService<R, FR> getFindClient();

    /**
     * Tests if record is valid.
     */
    protected void verifyRecord(S record) {
        verifyRecord(record, false);
    }

    /**
     * Tests if record is valid and can specify whether or not it was a save.
     */
    protected abstract void verifyRecord(S record, boolean save);

    @Test
    @SuppressWarnings("unchecked")
    public void testFindById() {
        R response = getFindClient().findById(id);
```

```
        assertNotNull("Response is null.", response);

        verifyRecord((S) response.getResults());
    }

    @Test
    @SuppressWarnings("unchecked")
    public void testPaginatedFind() {
        int page = 0;
        int pageSize = 2;

        FR response = getFindClient().find(page, pageSize);
        assertNotNull("Response is null.", response);

        assertEquals("count", expectedCount, response.getCount());

        assertNotNull("Response results is null.", response.getResults());
        verifyRecord((S) response.getResults().get(0));
    }

    @Test
    @SuppressWarnings("unchecked")
    public void testFind() {
        FR response = getFindClient().find();
        assertNotNull("Response is null.", response);

        assertEquals("count", expectedCount, response.getCount());

        assertNotNull("Response results is null.", response.getResults());
        verifyRecord((S) response.getResults().get(0));
    }

    /**
     * Tests if audit info is valid.
     */
    protected void verifyAuditInfo(DateTime lastUpdated, String lastUpdateUser,
                                   DateTime created, String createUser) {
        DateTime now = DateTime.now();

        assertNotNull("'lastUpdated' is null", lastUpdated);
        assertNotNull("'lastUpdateUser' is null", lastUpdateUser);
        assertNotNull("'created' is null", created);
        assertNotNull("'createUser' is null", createUser);

        assertTrue("'lastUpdated' should be before now.", (lastUpdated.isBefore(now)));
        assertTrue("'created' should be before now.", (created.isBefore(now)));
    }

}
```

Example 8 `AbstractPersistenceFindControllerTest`

The `AbstractPersistenceControllerTest` provides an abstract base for testing a `PersistenceMarshallingService`.

```


public abstract class AbstractPersistenceControllerTest<R extends EntityResponseResult, FR extends
EntityFindResponseResult, S extends PkEntityBase>
        extends AbstractPersistenceFindControllerTest<R, FR, S> {

    protected final Logger logger = LoggerFactory.getLogger(getClass());

    public AbstractPersistenceControllerTest(int id, long expectedCount) {
        super(id, expectedCount);
    }
```

```
    /**
     * Gets find client.
     */
    protected PersistenceFindMarshallingService<R, FR> getFindClient() {
        return getClient();
    }

    /**
     * Gets client.
     */
    protected abstract PersistenceMarshallingService<R, FR, S> getClient();

    /**
     * Create save request.
     */
    protected abstract S createSaveRequest();

    @Test
    @SuppressWarnings("unchecked")
    public void testSave() {
        S request = createSaveRequest();

        R response = getClient().save(request);

        assertNotNull("Response is null.", response);

        verifyRecord((S) response.getResults(), true);

        int expectedCount = 1;
        assertEquals("messageList.size", expectedCount, response.getMessageList().size());

        logger.info(response.getMessageList().get(0).getMessage());
    }

    @Test
    public void testDeletePk() {
        ResponseResult response = getClient().delete(id);

        assertNotNull("Response is null.", response);

        int expectedCount = 1;
        assertEquals("messageList.size", expectedCount, response.getMessageList().size());

        logger.info(response.getMessageList().get(0).getMessage());
    }

    /**
     * Tests if primary key is valid.
     */
    protected void verifyPrimaryKey(int id, boolean save) {
        if (!save) {
            assertEquals("'id'", id, this.id);
        } else {
            assertTrue("Primary key should be greater than zero.", (id > 0));
        }
    }

}
```

Example 9 `AbstractPersistenceControllerTest`

Code Example

Since all of the main testing for the `PersistenceMarshallingService` client is in the parent classes, just

the constructor and a few methods need to be implemented. The constructor takes the primary key used by find tests and the expected count for retrieving all records. The methods implemented are `getClient()` to return this test's client, the request to be saved, and a verification method.

```
public class PersonControllerTest extends AbstractPersistenceControllerTest<PersonResponse,
PersonFindResponse, Person> {

    @Autowired
    private PersonClient client = null;

    public PersonControllerTest() {
        super(1, 3);
    }

    @Override
    protected PersistenceMarshallingService<PersonResponse, PersonFindResponse, Person> getClient()
{
        return client;
    }

    @Override
    protected Person createSaveRequest() {
        return new Person().withFirstName(FIRST_NAME).withLastName(LAST_NAME);
    }

    @Override
    protected void verifyRecord(Person record, boolean save) {
        assertNotNull("Result is null.", record);

        verifyPrimaryKey(record.getId(), save);

        assertEquals("'firstName'", FIRST_NAME, record.getFirstName());
        assertEquals("'lastName'", LAST_NAME, record.getLastName());

        verifyAuditInfo(record.getLastUpdated(), record.getLastName(), record.getCreated(),
record.getCreateUser());

        logger.debug("id=" + record.getId() +
                     "  firstName=" + record.getFirstName() +
                     "  lastName=" + record.getLastName() +
                     "  lastUpdated=" + record.getLastUpdated() +
                     "  lastUpdateUser=" + record.getLastUpdateUser() +
                     "  created=" + record.getCreated() +
                     "  createUser=" + record.getCreateUser());
    }

}
```

Example 10 `PersonControllerTest`

5. Reference

Related Links

- Contact Application Web Service Beans
- Contact Application DAO
- Contact Application Services

- Contact Application REST Services

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd app/contact-app/contact-test
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Part VII. Spring dm Server

This section is on developing with the Spring dm Server and OSGi based development.

OSGi (Open Services Gateway interface) is a specification maintained by the OSGi Alliance [http://www.osgi.org/]. It was founded in March 1999. One of the goals of OSGi is to reduce complexity by having much better modularity and deployment of libraries and services. The increased modularity primarily comes from a much more fine grained classloading mechanism compared to a standard JVM or Java EE container. Libraries and packages needed at runtime must be explicitly defined and even the version or version ranges required can be specified. Also, what interfaces or classes your library exports are also clearly defined. A library (JAR archive) in OSGi terminology is referred to as a bundle.

There are a few different open source OSGi runtime environments. Some are Apache Felix [http://felix.apache.org/], Knopflerfish [http://www.knopflerfish.org/], and Equinox [http://www.eclipse.org/equinox/]. You may know Equinox [http://www.eclipse.org/equinox/] as being what the Eclipse IDE [http://www.eclipse.org/] is built on. Any Eclipse plugin is actually an OSGi bundle, so Eclipse has very good OSGi development support.

The Spring dm Server [http://www.springsource.com/products/suite/dmserver] is also an OSGi server and is actually built on top of Equinox [http://www.eclipse.org/equinox/]. Although many customizations and enhancements have been done to make it easier to manage and deploy bundles. Some of the enhancements are customizations to make it easier to deploy Spring bundles and also for deploying web bundles. In fact, standard WARs can be deployed as they are to any web container. Web modules can also be deployed as a thin WAR that uses OSGi imports to resolve any JARs instead of having them embedded in */WEB-INF/lib* and also a more OSGi like web solution which will be shown in the next basic example.

SpringSource [http://www.springsource.com/] has made a very nice plugin for developing OSGi bundles and working with the Spring dm Server [http://www.springsource.com/products/suite/dmserver] on the Eclipse IDE [http://www.eclipse.org/]. The plugin provides project templates for a basic bundle and also a type called PAR. A PAR is a way to group together related bundles and deploy them together. The plugin also has Server support in the IDE that allows deploying projects to the server. As changes are made to your project, it will automatically keep deploying any changes to the bundle to the server. Also, when managing the project any OSGi bundle from the SpringSource [http://www.springsource.com/] can be downloaded directly into the Spring dm Server [http://www.springsource.com/products/suite/dmserver].

If you haven't already installed the IDE support for this and would like to run the examples, please refer to the Setup Appendix.

Simple Message Service

David Winterfeldt

2008

There are three OSGi bundles in this example. Two are different versions of a bundle that exposes a message interface as an OSGi service. Each bundle depends on a different version of Commons Lang to demonstrate the side-by-side versioning capabilities of OSGi. The third bundle is a web module that displays the result from calling the OSGi message service. The example was made to be deployed on the Spring dm Server [http://www.springsource.com/products/suite/dmserver].

Note

Currently the examples are setup to only run from the Eclipse environment and to be deployed using the Spring dm Server [http://www.springsource.com/products/suite/dmserver]'s IDE integration. A Maven build will eventually be added.

1. Message Service Bundle 1.0

Manifest Configuration

The manifest defines the services name, description, it's symbolic name (will deployed under this name), version, and whatever libraries, bundles, and/or packages it imports. Also, whatever packages are exported are also defined.

Note

If when opening the manifest and viewing the Dependencies tab in the Eclipse IDE, there isn't an 'Import Bundle' section on the right and an 'Import Library' section underneath it the default Eclipse manifest editor may have been opened. To open the Spring manifest editor, right click, 'Open with'/'Other...', and then choose the 'Bundle Manifest Editor'.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-Name: Simple Service Bundle
Bundle-Description: Simple Service
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.springbyexample.sdms.message.messageService  ❶
Bundle-Version: 1.0.0  ❷
Import-Library: org.aspectj,
 org.springframework.spring
Import-Bundle: com.springsource.org.apache.commons.lang;version="[2.1.0,2.1.0]"  ❸
Export-Package: org.springbyexample.sdms.message.service;version="1.0.0"  ❹
```

❶ The symbolic name the bundle is deployed under and another bundle could use to reference it.
❷ The version the bundle is deployed under.
❸ The *Import-Bundle* property imports the package `com.springsource.org.apache.commons.lang`, version *2.1.0*.
❹ The *Export-Package* property exposes any packages for external use by other bundles. This exposes the package `org.springbyexample.sdms.message.service` which has the `MessageService` interface as version *1.0.0*.

Spring Configuration

By default any spring configuration files in the *META-INF/spring* will be processed (this is configurable). Following the naming conventions setup by Spring Dynamic Modules for OSGi(tm) Service Platforms [http://www.springsource.org/osgi] there are two Spring configuration files. One is called *bundle-context.xml* and the other is called *bundle-context-osgi.xml*. The first one is for standard Spring configuration and the latter is meant for Spring Dynamic Modules [http://www.springsource.org/osgi] specific configuration. Like exposing OSGi services or defining services as bean references.

The `MessageServiceImpl` is defined as a *messageService* bean which will be exposed as a service in the *bundle-context-osgi.xml* configuration file. Spring's *context:component-scan* could have also been used, but wasn't just to keep the example a little more clear.

src/main/resources/META-INF/spring/bundle-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="messageService"
          class="org.springbyexample.sdms.message.service.impl.MessageServiceImpl"
          p:message="Greetings!"/>

</beans>
```

The Spring Dynamic Modules [http://www.springsource.org/osgi]' namespace is setup as the default namespace for the configuration file. The *service* element exposes the *messageService* bean as an OSGi service under the interface `org.springbyexample.sdms.message.service.MessageService`.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

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

    <service  ref="messageService"
```

```
interface="org.springbyexample.sdms.message.service.MessageService"/>

</beans:beans>
```

Code Example

The code consists of the `MessageService` interface and the implementation `MessageServiceImpl`. It's just an interface to get a message and the implementation uses Commons Lang.

sdms/message-service/service-version1_0/src/main/java/org/springbyexample/sdms/message/service/MessageService.java

```
public interface MessageService {

    /**
     * Gets message.
     */
    public String getMessage();

}
```

Example 1 Message Service `MessageService` (version 1.0)

This implementation uses `FastDateFormat` from Commons Lang 2.1.0 to display a formatted date after the message that will be injected from the Spring configuration.

sdms/message-service/service-version1_0/src/main/java/org/springbyexample/sdms/message/service/impl/MessageServiceImpl.java

```
public class MessageServiceImpl implements MessageService {

    private String message = null;

    /**
     * Gets message.
     */
    public String getMessage() {
        StringBuffer sb = new StringBuffer();

        sb.append(message);
        sb.append("  [");
        sb.append(FastDateFormat.getInstance("MM/dd/yy").format(new Date()));
        sb.append("]");

        return sb.toString();
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Example 2 Message Service `MessageServiceImpl` (version 1.0)

2. Message Service Bundle 1.1

Manifest Configuration

This is the same as the *1.0* version of the Message Service bundle, but it has it's version set to *1.1.0* on the *Bundle-Version* property and also depends on Commons Lang 2.4.0.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-Name: Simple Service Bundle
Bundle-Description: Simple Service
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.springbyexample.sdms.message.messageService  ❶
Bundle-Version: 1.1.0  ❷
Import-Library: org.aspectj,
 org.springframework.spring
Import-Bundle: com.springsource.org.apache.commons.lang;version="[2.4.0,2.4.0]"  ❸
Export-Package: org.springbyexample.sdms.message.service;version="1.1.0"  ❹
```

❶ The symbolic name the bundle is deployed under and another bundle could use to reference it.
❷ The version the bundle is deployed under.
❸ The *Import-Bundle* property imports the package `com.springsource.org.apache.commons.lang`, version *2.4.0*.
❹ The *Export-Package* property exposes any packages for external use by other bundles. This exposes the package `org.springbyexample.sdms.message.service` which has the `MessageService` interface as version *1.1.0*.

Spring Configuration

The version 1.1 of `MessageServiceImpl` has a different message than version 1.0. Also the implementation uses a Commons Lang 2.4.0 specific method call that is appended to the message.

src/main/resources/META-INF/spring/bundle-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="messageService"
          class="org.springbyexample.sdms.message.service.impl.MessageServiceImpl"
          p:message="Greetings from Spring by Example!"/>
```

```
</beans>
```

The *messageService* bean is exposed as a service under the interface `org.springbyexample.sdms.message.service.MessageService`.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

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

    <service  ref="messageService"
interface="org.springbyexample.sdms.message.service.MessageService"/>

</beans:beans>
```

Code Example

The `MessageService` interface is identical to the one in the Message Service Bundle 1.0 class. So the Message Service Web Module can dynamically use either implementation's service.

sdms/message-service/service-version1_1/src/main/java/org/springbyexample/sdms/message/service/MessageService.java

```
public interface MessageService {

    /**
     * Gets message.
     */
    public String getMessage();

}
```

Example 3 Message Service `MessageService` (version 1.1)

This implementation uses `FastDateFormat` from Commons Lang 2.4.0 to display a formatted date after the message that will be injected from the Spring configuration. It also uses `StringUtils.length(String)` to add the message length at the end of the message. This method was adding Commons Lang 2.4.0. If you try to use this method in the Message Service Bundle 1.0 implementation, you will see that it won't be able to find that method on `StringUtils`.

sdms/message-service/service-version1_1/src/main/java/org/springbyexample/sdms/message/service/impl/MessageServiceImpl.java

```
public class MessageServiceImpl implements MessageService {

    private String message = null;

    /**
     * Gets message.
     */
    public String getMessage() {
        StringBuffer sb = new StringBuffer();

        sb.append(message);
        sb.append("  [");
        sb.append(FastDateFormat.getInstance("MM/dd/yyyy").format(new Date()));
        sb.append("]");

        // add Commons Lang StringUtils.length(String) which was specifically added in Commons Lang
2.4.0)
        sb.append("  (message length=");
        sb.append(StringUtils.length(message));
        sb.append(")");

        return sb.toString();
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Example 4 Message Service `MessageServiceImpl` *(version 1.1)*

3. Message Service Web Module

Manifest Configuration

The web module's manifest is very similar to the other bundles, but it imports the taglib and servlet bundles. Also instead of exporting a package, it imports the Message Service package exported by the Message Service bundles and specifies that is can use version 1.0.0 through 1.1.0.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-Name: Simple Service Web Module
Bundle-SymbolicName: org.springbyexample.sdms.message.webModule  ❶
Bundle-Version: 1.0.0  ❷
Bundle-Vendor: Spring by Example
Bundle-ManifestVersion: 2
Import-Bundle: com.springsource.org.apache.taglibs.standard,
 com.springsource.javax.servlet
Import-Library: org.aspectj;version="[1.6.0,1.7.0)",
 org.springframework.spring;version="[2.5.5,3.0.0)"
Module-Type: Web  ❸
Web-ContextPath: message  ❹
Web-DispatcherServletUrlPatterns: *.html  ❺
```

```
Import-Package: org.springbyexample.sdms.message.service;version="[1.0.0,1.1.0]"  ❻
```

❶ The symbolic name the bundle is deployed under and another bundle could use to reference it.
❷ The version the bundle is deployed under.
❸ Declares the module type indicating this is a web module. This is used by the Spring dm Server [http://www.springsource.com/products/suite/dmserver].
❹ The *Web-ContextPath* property configures the web application's context path.
❺ The *Web-DispatcherServletUrlPatterns* property configures the dispatch servlet to route any matches for the pattern '*.html' to controllers.
❻ The *Import-Package* property imports the package `org.springbyexample.sdms.message.service` which was exported by the message service bundle.

Spring Configuration

The *bundle-context.xml* isn't used in this example, but *bundle-context-osgi.xml* makes a reference to the message bundle service and exposes it as a bean. The Spring web configuration is in *webmvc-context.xml*.

The reference to the OSGi service `MessageService` is exposed as a bean named *messageService*.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

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

    <reference id="messageService"
interface="org.springbyexample.sdms.message.service.MessageService"/>

</beans:beans>
```

This is just a standard Spring MVC configuration and the controller is register using *context:component-scan*.

src/main/resources/META-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.sdms.web.message.mvc" />
```

```
    <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
        <property name="interceptors" ref="localeChangeInterceptor"/>
    </bean>

    <!-- Enables annotated POJO @Controllers -->
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

    <!-- Enables plain Controllers -->
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/WEB-INF/jsp/"
          p:suffix=".jsp"/>

    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <value>messages</value>
        </property>
    </bean>

    <!-- Declare the Interceptor -->
    <bean id="localeChangeInterceptor"
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="locale"/>
    </bean>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

JSP Example

To reduce the complexity of the example the controller just puts the message into the `Model` and is rendered on the page using JSTL.

src/main/resources/MODULE-INF/WEB-INF/jsp/display/index.jsp

```
<html>
<head>
<title>Message</title>
</head>
<body>
<h1>Message</h1>

<p>${message}</p>
</body>
</html>
```

Code Example

The controller has the message service defined in *bundle-context-osgi.xml* injected using `@Autowired`. The

`display` method maps to */display/index.html"* and puts the results of the message service into the `Model` under the key 'message'.

sdms/message-service/service-web-module/src/main/java/org/springbyexample/sdms/web/message/mvc/MessageController.java

```
@Controller
public class MessageController {

    @Autowired
    protected MessageService messageService = null;

    /**
     * Displays RMI info.
     */
    @RequestMapping(value="/display/index.html")
    public void display(Model model) {
            model.addAttribute("message", messageService.getMessage());
    }

}
```

Example 5 Mesage Service `MessageController`

4. Reference

Related Links

- OSGi Alliance [http://www.osgi.org/]
- Spring dm Server [http://www.springsource.com/products/suite/dmserver]
- Equinox [http://www.eclipse.org/equinox/]
- Adrian Colyer's blog 'Why should I care about OSGi anyway?' [http://blog.springsource.com/2008/05/15/why-should-i-care-about-osgi-anyway/]
- Rob Harrop's blog 'Getting started with SpringSource dm Server' [http://blog.springsource.com/2008/10/22/getting-started-with-springsource-dm-server/]

Project Setup

The project is available to checkout from an anonymous Subversion checkout.

Command Line Example

This will checkout all three projects, but each one could be checked out individually too.

```
$ svn co http://svn.springbyexample.org/sdms/simple-message-service/tags/1.0/ simple-message-service
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 2.5.x
- Spring dm Server 1.0

Simple Spring MVC

David Winterfeldt

2008

There are three OSGi bundles in this example that can either be deployed individually or as part of a PAR bundle. A PAR bundle groups together your other bundles and deploys them so they aren't visible outside the PAR. In this example since a `javax.sql.DataSource` is exposed an OSGi service, this feature is very useful since it's quite likely at some point to want to deploy multiple services of a similar type, but each one may be specific to an application.

This example has a `DataSource` bundle that exposes a `DataSource` OSGi service for an in memory HSQL DB. There is a JPA Bundle that exposes a Person service and uses the `DataSource` service. The final bundle is a web application using Spring MVC to create, edit, update, delete, and search using the Person service. The example was made to be deployed on the Spring dm Server [http://www.springsource.com/products/suite/dmserver].

Note

Currently the examples are setup to only run from the Eclipse environment and to be deployed using the Spring dm Server [http://www.springsource.com/products/suite/dmserver]'s IDE integration. A full Maven build will eventually be added.

The *simple-spring-mvc-par* project has a Maven build that will download and install all the libraries and bundles needed for this example. It will install in them in '${sdms.home}/repository/libraries/usr' and '${sdms.home}/repository/bundles/usr'. Just change the property 'sdms.path' in the pom.xml to the Spring dm Server [http://www.springsource.com/products/suite/dmserver] location used by Eclipse and run 'mvn package'.

1. Simple Spring MVC PAR

Manifest Configuration

There basically isn't very much to the PAR bundle since it's just a a way to group together other bundles. If you telnet into the server you'll be able to see the synthetic context the bundles were deployed under based on the PAR's bundle name. Below is a partial display of the results from running the **ss**.

```
$ telnet localhost 2401
```

```
osgi> ss
...
91 ACTIVE
org.springbyexample.sdms.simpleForm-1-org.springbyexample.sdms.simpleForm-synthetic.context_1.0.0
92 ACTIVE
org.springbyexample.sdms.simpleForm-1-org.springbyexample.sdms.simpleForm.datasource_1.0.0
93 ACTIVE
org.springbyexample.sdms.simpleForm-1-org.springbyexample.sdms.simpleForm.person_1.0.0
94 ACTIVE
org.springbyexample.sdms.simpleForm-1-org.springbyexample.sdms.simpleForm.webModule_1.0.0
```

```
...
```

Note

If when opening the manifest and viewing the Dependencies tab in the Eclipse IDE, it doesn't say 'PAR Editor' at the top, the default Eclipse manifest editor may have been opened. To open the Spring manifest editor, right click, 'Open with'/'Other...', and then choose the 'PAR Manifest Editor'.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Application-SymbolicName: org.springbyexample.sdms.simpleForm  ❶
Application-Version: 1.0.0
Application-Name: Simple Spring MVC PAR
Bundle-Vendor: Spring by Example
```

❶ The symbolic name the PAR bundle is deployed under.

2. Simple Spring MVC `DataSource` Bundle

The `DataSource` bundle creates an in memory HSQL database and creates a connection pool that is exposed as an OSGi `DataSource` service.

Manifest Configuration

The manifest defines the bundles symbolic name, and imports `javax.sql`, Spring, HSQL DB, and Commons DBCP.

Note

If when opening the manifest and viewing the Dependencies tab in the Ecilpse IDE, there isn't an 'Import Bundle' section on the right and an 'Import Library' section underneath it the default Eclipse manifest editor may have been opened. To open the Spring manifest editor, right click, 'Open with'/'Other...', and then choose the 'Bundle Manifest Editor'.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Simple Spring MVC Form DataSource Bundle
Bundle-Description: Simple Spring MVC Form DataSource
Bundle-SymbolicName: org.springbyexample.sdms.simpleForm.datasource  ❶
Bundle-Version: 1.0.0  ❷
Bundle-Vendor: Spring by Example
Import-Package: javax.sql  ❸
```

```
Import-Library: org.springframework.spring;version="[2.5.5,3.0.0)"  ❹
Import-Bundle: com.springsource.org.hsqldb;version="[1.8.0,2.0.0)",
 com.springsource.org.apache.commons.dbcp;version="[1.2.2.osgi,1.3.0)"  ❺
```

❶ The symbolic name the bundle is deployed under and another bundle could use to reference it.
❷ The version the bundle is deployed under.
❸ Imports the package `javax.sql`.
❹ The *Import-Library* property imports all the related Spring bundles for the latest version of the library that can be resolved based on the version range.
❺ Imports the HSQL DB and Commons DBCP bundles, retrieving the newest version possible based on the version ranges specified.

Spring Configuration

By default any spring configuration files in the *META-INF/spring* will be processed (this is configurable). A connection pool is created using Commons DBCP to an HSQL DB and then the pool is exposed as an OSGi service.

src/main/resources/META-INF/spring/bundle-context.xml

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

        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
                <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
                <property name="url" value="jdbc:hsqldb:mem:person" />
                <property name="username" value="sa" />
                <property name="password" value="" />
        </bean>

</beans>
```

The Spring Dynamic Modules [http://www.springsource.org/osgi]' namespace is setup as the default namespace for the configuration file. The *service* element exposes the *dataSource* bean as an OSGi service under the interface `javax.sql.DataSource`.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xsi:schemaLocation="http://www.springframework.org/schema/osgi
                                 http://www.springframework.org/schema/osgi/spring-osgi-1.0.xsd
                                 http://www.springframework.org/schema/beans
                                 http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
    <service ref="dataSource" interface="javax.sql.DataSource" />

</beans:beans>
```

3. Simple Spring MVC Person DAO Bundle

The bundle uses JPA to create a Person service. The `Person` has a one to many relationship to `Address`, but to keep the example simple only the `Person` is used by the web module. JPA is using Hibernate for persistence.

Manifest Configuration

The bundle imports `javax.sql`, HSQL DB, Spring, and Hibernate. It also exports the package containing the Hibernate persistence beans and the Person DAO interface.

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Simple Spring MVC Person Bundle
Bundle-Description: Simple Spring MVC Person
Bundle-SymbolicName: org.springbyexample.sdms.simpleForm.person  ❶
Bundle-Version: 1.0.0  ❷
Bundle-Vendor: Spring by Example
Import-Package: javax.sql  ❸
Import-Bundle: com.springsource.org.hsqldb;version="[1.8.0,2.0.0)"  ❹
Import-Library: org.springframework.spring;version="[2.5.5,3.0.0)",
 org.hibernate.ejb;version="[3.3.2.GA,3.3.2.GA]"  ❺
Export-Package: org.springbyexample.sdms.simpleForm.orm.bean;version="1.0.0",
 org.springbyexample.sdms.simpleForm.orm.dao;version="1.0.0"  ❻
```

❶ The symbolic name the bundle is deployed under.
❷ The version the bundle is deployed under.
❸ Imports the package `javax.sql`.
❹ Imports the HSQL DB bundle.
❺ The two libraries for Spring and Hibernate are resolved based on their version ranges.
❻ The package containing the persistence beans and the one with the Person DAO interface are both exported under the version *1.0.0*.

JPA Configuration

The JPA configuration sets up Hibernate as the persistence provider and registers the two entity classes `Person` and `Address`.

src/main/resources/META-INF/persistence.xml

```
```

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
             version="1.0">
    <persistence-unit name="personDatabase">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>org.springbyexample.sdms.simpleForm.orm.bean.Person</class>
        <class>org.springbyexample.sdms.simpleForm.orm.bean.Address</class>

        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <!--
                Note: setting 'hibernate.hbm2ddl.auto' to 'create' will result in
                'import.sql' (in the root of the classpath) being used to populate
                the DB
            -->
            <property name="hibernate.hbm2ddl.auto" value="create" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
        </properties>
    </persistence-unit>
</persistence>
```

Spring Configuration

The *context:component-scan* is used to Person DAO implementation, and *tx:annotation-driven* configures Spring to process transaction annotations. A JPA transaction manager is created and JPA with i and also with the `DataSource` from the OSGi service.

src/main/resources/META-INF/spring/bundle-context.xml

```
<?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:tx="http://www.springframework.org/schema/tx"
       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.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

        <context:component-scan base-package="org.springbyexample.sdms.simpleForm.orm.impl"/>

        <tx:annotation-driven />

        <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory"/>
        </bean>

    <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
            <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
```

```
    </bean>

</beans>
```

The `javax.sql.DataSource` exposed by the previous bundle is referenced, and the Person DAO implementation is registered as an OSGi service.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xsi:schemaLocation="http://www.springframework.org/schema/osgi
                                 http://www.springframework.org/schema/osgi/spring-osgi-1.0.xsd
                                 http://www.springframework.org/schema/beans
                                 http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <reference id="dataSource" interface="javax.sql.DataSource" />

    <service ref="personDao" interface="org.springbyexample.sdms.simpleForm.orm.dao.PersonDao" />

</beans:beans>
```

Code Example

The Person entity uses `javax.persistence` annotation for configuration.

Excerpt from *sdms/simple-spring-mvc/simple-spring-mvc-person/src/main/java/org/springbyexample/sdms/simpleForm/orm/bean/Person.java*

```
@Entity
@Table(name="PERSON")
public class Person implements Comparable<Person>, Serializable {

    ...

    private Integer id = null;
    private String firstName = null;
    private String lastName = null;
    private Set<Address> addresses = null;
    private Date created = null;

    /**
     * Gets id (primary key).
     */
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

    /**
```

```
     * Sets id (primary key).
     */
    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * Gets first name.
     */
    @Column(name="FIRST_NAME")
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Gets last name.
     */
    @Column(name="LAST_NAME")
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets list of <code>Address</code>es.
     */
    @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    @JoinColumn(name="PERSON_ID", nullable=false)
        public Set<Address> getAddresses() {
                return addresses;
        }

    /**
     * Sets list of <code>Address</code>es.
     */
        public void setAddresses(Set<Address> addresses) {
                this.addresses = addresses;
        }

    /**
     * Gets date created.
     */
        public Date getCreated() {
                return created;
        }

    /**
     * Sets date created.
     */
        public void setCreated(Date created) {
                this.created = created;
        }

        public Address findAddressById(Integer id) {
            Address result = null;

            if (addresses != null) {
                for (Address address: addresses) {
                    if (address.getId().equals(id)) {
```

```
                        result = address;

                        break;
                    }
                }
            }

            return result;
        }

    ...

}
```

Example 1 `Person` Entity

The Person DAO implementation uses the JPA `EntityManager` for managing a Person.

sdms/simple-spring-mvc/simple-spring-mvc-person/src/main/java/org/springbyexample/sdms/simpleForm/orm/impl/PersonDaoImpl.ja

```


@Repository("personDao")
@Transactional(readOnly = true)
public class PersonDaoImpl implements PersonDao {

    private EntityManager em;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Person findPersonById(Integer id) {
        return em.find(Person.class, id);
    }

    /**
     * Find persons using a start index and max number of results.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons(final int startIndex, final int maxResults) {
        return em.createQuery("select p from Person p order by p.lastName, p.firstName")
            .setFirstResult(startIndex).setMaxResults(maxResults).getResultList();
    }

    /**
     * Find persons.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersons() {
        return em.createQuery("select p from Person p order by p.lastName,
p.firstName").getResultList();
    }

    /**
     * Find persons by last name.
     */
    @SuppressWarnings("unchecked")
    public Collection<Person> findPersonsByLastName(String lastName) {
        return em.createQuery("select p from Person p where p.lastName = :lastName order by
p.lastName, p.firstName")
            .setParameter("lastName", lastName).getResultList();
```

```
    }

    /**
     * Saves person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person save(Person person) {
        return em.merge(person);
    }

    /**
     * Deletes person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void delete(Person person) {
        em.remove(em.merge(person));
    }

    /**
     * Saves address to person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person saveAddress(Integer id, Address address) {
        Person person = findPersonById(id);

        if (person.getAddresses().contains(address)) {
            person.getAddresses().remove(address);
        }

        person.getAddresses().add(address);

        return save(person);
    }

    /**
     * Deletes address from person.
     */
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public Person deleteAddress(Integer id, Integer addressId) {
        Person person = findPersonById(id);

        Address address = new Address();
        address.setId(addressId);

        if (person.getAddresses().contains(address)) {
            person.getAddresses().remove(address);
        }

        return save(person);
    }

}
```

Example 2 Person DAO Implementation

4. Simple Spring MVC Web Module

The web module has a simple form for creating and editing a Person, and also has a basic search page. The application also has internationalization and uses Tiles for templating.

Manifest Configuration

src/main/resources/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Simple Spring MVC Web Module
Bundle-Description: Simple Spring MVC Web Module
Bundle-SymbolicName: org.springbyexample.sdms.simpleForm.webModule  ❶
Bundle-Version: 1.0.0  ❷
Bundle-Vendor: Spring by Example
Module-Type: Web  ❸
Web-ContextPath: simple-form  ❹
Web-DispatcherServletUrlPatterns: *.html  ❺
Import-Bundle: com.springsource.org.apache.taglibs.standard,
 com.springsource.javax.servlet
Import-Library: org.aspectj;version="[1.6.0,1.7.0)",
 org.springframework.spring;version="[2.5.5,3.0.0)",
 org.hibernate.ejb;version="[3.3.2.GA,3.3.2.GA]",
 org.apache.tiles;version="[2.0.5.osgi,2.0.5.osgi]"
Import-Package: org.springbyexample.sdms.simpleForm.orm.bean;version="[1.0.0,1.1.0]",
 org.springbyexample.sdms.simpleForm.orm.dao;version="[1.0.0,1.1.0]"  ❻
```

❶ The symbolic name the bundle is deployed under.
❷ The version the bundle is deployed under.
❸ Declares the module type indicating this is a web module. This is used by the Spring dm Server [http://www.springsource.com/products/suite/dmserver].
❹ The *Web-ContextPath* property configures the web application's context path.
❺ The *Web-DispatcherServletUrlPatterns* property configures the dispatch servlet to route any matches for the pattern '*.html' to controllers.
❻ The *Import-Package* property imports the the persistence beans and the Person DAO interface from the Person DAO bundle.

Spring Configuration

The *bundle-context.xml* isn't used in this example, but *bundle-context-osgi.xml* makes a reference to the Person service and exposes it as a bean. The Spring web configuration is in *webmvc-context.xml*.

The reference to the OSGi service `PersonDao` is exposed as a bean named *personDao*.

src/main/resources/META-INF/spring/bundle-context-osgi.xml

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

    <reference id="personDao" interface="org.springbyexample.sdms.simpleForm.orm.dao.PersonDao"/>

</beans:beans>
```

```
```

This standard Spring MVC configuration file scans for controller classes, creates handlers, configures Tiles, and also internationalization. The *classnameControllerMappings* bean enables convention based mappings for reduced configuration. It is configured to be case sensitive, so a controller called `StudentPersonController` would map to the URL '/studentPerson'. Although it defaults to case insensitive and would then map to the URL '/studentperson'. The default handler is set with the `UrlFilenameViewController`, which will handle any requests not handled by a convention based controller.

src/main/resources/META-INF/spring/webmvc-context.xml

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.sdms.simpleForm.web.mvc" />

    <!-- Enables /[resource]/[action] to [Resource]Controller class mapping -->
    <bean id="classnameControllerMappings"
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"
          p:order="1"
          p:interceptors-ref="localeChangeInterceptor"
          p:caseSensitive="true">
        <property name="defaultHandler">
            <bean class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />
        </property>
    </bean>

    <!-- Enables annotated POJO @Controllers -->
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

    <!-- Enables plain Controllers -->
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

    <bean id="tilesConfigurer"
             class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
           <property name="definitions">
                  <list>
                         <value>/WEB-INF/tiles-defs/templates.xml</value>
                  </list>
           </property>
    </bean>

    <bean id="tilesViewResolver"
             class="org.springframework.web.servlet.view.UrlBasedViewResolver"
          p:order="2"
             p:viewClass="org.springframework.web.servlet.view.tiles2.TilesView" />

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
          p:basenames="messages" />

    <!-- Declare the Interceptor -->
    <bean id="localeChangeInterceptor"
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
          p:paramName="locale" />

    <!-- Declare the Resolver -->
```

```
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

JSP Example

A simple person form using Spring's custom JSP tags.

src/main/resources/MODULE-INF/WEB-INF/jsp/person/form.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1><fmt:message key="person.form.title"/></h1>

<c:if test="${not empty statusMessageKey}">
    <p><fmt:message key="${statusMessageKey}"/></p>
</c:if>

<c:url var="url" value="/person/form.html" />
<form:form action="${url}" commandName="person">
    <form:hidden path="id" />

    <fieldset>
        <div class="form-row">
            <label for="firstName"><fmt:message key="person.form.firstName"/>:</label>
            <span class="input"><form:input path="firstName" /></span>
        </div>
        <div class="form-row">
            <label for="lastName"><fmt:message key="person.form.lastName"/>:</label>
            <span class="input"><form:input path="lastName" /></span>
        </div>
        <div class="form-buttons">
            <div class="button"><input name="submit" type="submit" value="<fmt:message
key="button.save"/>" /></div>
        </div>
    </fieldset>
</form:form>
```

This is the search page and next to each record displayed is an 'edit' and 'delete' link.

src/main/resources/MODULE-INF/WEB-INF/jsp/person/search.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<h1><fmt:message key="person.search.title"/></h1>

<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
```

```
    </tr>
<c:forEach var="person" items="${persons}" varStatus="status">
    <tr>
        <c:set var="personFormId" value="person${status.index}"/>

        <c:url var="editUrl" value="/person/form.html">
            <c:param name="id" value="${person.id}" />
        </c:url>
        <c:url var="deleteUrl" value="/person/delete.html"/>
        <form id="${personFormId}" action="${deleteUrl}" method="POST">
            <input id="id" name="id" type="hidden" value="${person.id}"/>
        </form>

        <td>${person.firstName}</td>
        <td>${person.lastName}</td>
        <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <a href="javascript:document.forms['${personFormId}'].submit();"><fmt:message
key="button.delete"/></a>
        </td>
    </tr>
</c:forEach>
</table>
```

Below is an example of an alternative way to handle a delete following a REST style approach, although the following JavaScript won't work in IE. A library like Dojo should be used for the AJAX call to make the JavaScript browser independent.

The delete method in the controller would be made to only accept an HTTP DELETE. A standard HTML form can't submit this type of request, but JavaScript's `XMLHttpRequest` can. Ideally in a more complex example, Spring JS would be used to decorate the link with JavaScript and there would be a URL that would still function with a standard GET or POST in case the user has JavaScript disabled in their browser.

The delete link calls the JavaScript function `deletePerson` passing in the URL that should be called by `XMLHttpRequest` using an HTTP DELETE (instead of the usual GET or POST). Before the delete is processed it is confirmed by a JavaScript popup.

Alternative search JSP

```


<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<script language="javascript">
//<!--
function deletePerson(url){
    var confirmed = confirm('<fmt:message key="person.form.confirmDelete"/>');

    if (confirmed) {
        var request = new XMLHttpRequest();
        request.open('DELETE', url, true);
        request.onreadystatechange = function () { window.location.reload(true); };
        request.send(null);
    }
}
// -->
</script>

<h1><fmt:message key="person.search.title"/></h1>
```

```
<table class="search">
    <tr>
        <th><fmt:message key="person.form.firstName"/></th>
        <th><fmt:message key="person.form.lastName"/></th>
    </tr>
<c:forEach var="person" items="${persons}">
    <tr>
        <c:url var="editUrl" value="/person/form.html">
            <c:param name="id" value="${person.id}" />
        </c:url>
        <c:url var="deleteUrl" value="/person/delete.html">
            <c:param name="id" value="${person.id}" />
        </c:url>

            <td>${person.firstName}</td>
        <td>${person.lastName}</td>
            <td>
            <a href='<c:out value="${editUrl}"/>'><fmt:message key="button.edit"/></a>
            <a href="javascript:deletePerson('<c:out value="${deleteUrl}"/>')"><fmt:message
key="button.delete"/></a>
        </td>
    </tr>
</c:forEach>
</table>
```

Code Example

Because of the *classnameControllerMappings* bean, the controller is automatically mapped to handle '/person/*' based on it's name (ex: [Path]Controller). Then the method names are used to match the rest of the path. So, '/person/form' will match the `form` method. The `@ModelAttribute` will be called before each path mapped to the controller. For example, when '/person/form' is called, the `newRequest` method will be called first so an instance of `Person` is available to bind to the form.

The search method doesn't currently use the person model attribute from the `newRequest` method, but in a more advanced example it likely would. For example using the first and last name fields to perform a like search in the database.

sdms/simple-spring-mvc/simple-spring-mvc-web-module/src/main/java/org/springbyexample/sdms/simpleForm/web/mvc/PersonContro

```
@Controller
public class PersonController {

    private static final String SEARCH_VIEW_KEY = "redirect:search.html";
    private static final String SEARCH_MODEL_KEY = "persons";

    @Autowired
    protected PersonDao personDao = null;

    /**
     * For every request for this controller, this will
     * create a person instance for the form.
     */
    @ModelAttribute
    public Person newRequest(@RequestParam(required=false) Integer id) {
        return (id != null ? personDao.findPersonById(id) : new Person());
    }

    /**
```

```
     * <p>Person form request.</p>
     *
     * <p>Expected HTTP GET and request '/person/form'.</p>
     */
    @RequestMapping(method=RequestMethod.GET)
    public void form() {}

    /**
     * <p>Saves a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/form'.</p>
     */
    @RequestMapping(method=RequestMethod.POST)
    public void form(Person person, Model model) {
        if (person.getCreated() == null) {
            person.setCreated(new Date());
        }

        Person result = personDao.save(person);

        // set id from create
        if (person.getId() == null) {
            person.setId(result.getId());
        }

        model.addAttribute("statusMessageKey", "person.form.msg.success");
    }

    /**
     * <p>Deletes a person.</p>
     *
     * <p>Expected HTTP POST and request '/person/delete'.</p>
     */
    @RequestMapping(method=RequestMethod.POST)
    public String delete(Person person) {
        personDao.delete(person);

        return SEARCH_VIEW_KEY;
    }

    /**
     * <p>Searches for all persons and returns them in a
     * <code>Collection</code>.</p>
     *
     * <p>Expected HTTP GET and request '/person/search'.</p>
     */
    @RequestMapping(method=RequestMethod.GET)
    public @ModelAttribute(SEARCH_MODEL_KEY) Collection<Person> search() {
        return personDao.findPersons();
    }

}
```

Example 3 Simple Spring MVC `PersonController`

5. Reference

Related Links

- OSGi Alliance [http://www.osgi.org/]
- Spring dm Server [http://www.springsource.com/products/suite/dmserver]

- Equinox [http://www.eclipse.org/equinox/]
- Adrian Colyer's blog 'Why should I care about OSGi anyway?' [http://blog.springsource.com/2008/05/15/why-should-i-care-about-osgi-anyway/]
- Rob Harrop's blog 'Getting started with SpringSource dm Server' [http://blog.springsource.com/2008/10/22/getting-started-with-springsource-dm-server/]

Project Setup

The project is available to checkout from an anonymous Subversion checkout.

Command Line Example

This will checkout all four projects, but each one could be checked out individually too.

```
$ svn co http://svn.springbyexample.org/sdms/simple-spring-mvc/tags/1.0.3/ simple-spring-mvc
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 2.5.x
- Spring dm Server 1.0

Part VIII. Modules

Reusable libraries/modules released under the Apache License, Version 2.0 [http://www.apache.org/licenses/LICENSE-2.0.txt] and developed with Java 6.

Module Summary

David Winterfeldt

2008

Reusable libraries/modules released under the Apache License, Version 2.0 [http://www.apache.org/licenses/LICENSE-2.0.txt] and developed with Java 6.

1. Downloads

Custom *ServletContext* Scope Module

The Spring by Example Custom *ServletContext* Scope module is a custom scope implementation for providing *ServletContext* (web application) scoped beans.

- Custom *ServletContext* Scope Module
- Custom *ServletContext* Scope Module Site [http://springbyexample.org/maven/site/sbe-servlet-context-scope/1.0.2/sbe-servlet-context-scope/]
- Download
  - Maven Dependency Instructions or Custom *ServletContext* Scope Module 1.0.2 JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-servlet-context-scope/1.0.2/sbe-servlet-context-scope-1.0.2.jar]

Custom Thread Scope Module

The Spring by Example Custom Thread Scope module is a custom scope implementation for providing thread scoped beans.

Note

See `org.springframework.context.support.SimpleThreadScope`, which was added in Spring 3.0, for a Spring Framework implementation. Although the Spring version doesn't support destruction callbacks (this implementation does when using a custom Runnable).

- Custom Thread Scope Module
- Custom Thread Scope Module Site [http://springbyexample.org/maven/site/sbe-thread-scope/1.0.2/sbe-thread-scope/]
- Download
  - Maven Dependency Instructions or Custom Thread Scope Module 1.0.2 JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-thread-scope/1.0.2/sbe-thread-scope-1.0.2.jar]

Dynamic Tiles Module

The Dynamic Tiles 2 module dynamically renders a Tiles 2 template with Spring MVC. Any request coming in mapped for Tiles processing will use the default template and dynamically insert the body based on the URL. There is support for AJAX and Spring Web Flow requests to render a fragment of a URL (based on `AjaxTilesView` from Spring JS and `FlowAjaxTilesView` from Spring Web Flow).

- Dynamic Tiles Spring MVC Module
- Dynamic Tiles Spring MVC Module Site [http://springbyexample.org/maven/site/sbe-dynamic-tiles2/1.2.1/sbe-dynamic-tiles2/]
- Download
  - Maven Dependency Instructions or Dynamic Tiles Module 1.2.1 JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-dynamic-tiles2/1.2.1/sbe-dynamic-tiles2-1.2.1.jar]

Spring by Example JCR Module

The Spring by Example JCR module uses Spring Modules JCR (Java Content Repository) module. Currently the utilities provide a way to recurse through the repositories nodes using JcrTemplate and a custom node callback for each matching node found while recursing the repository. This example uses Apache Jackrabbit for the Java Content Repository which is the reference implementation for JSR-170.

- Spring by Example JCR Module
- Spring by Example JCR Module Site [http://springbyexample.org/maven/site/sbe-jcr/1.0.2/sbe-jcr/]
- Download
  - Maven Dependency Instructions or Spring by Example JCR Module 1.0.2 JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-jcr/1.0.2/sbe-jcr-1.0.2.jar]

Spring by Example Utils Module

The Spring by Example Utils module currently has the `HttpClientTemplate` and `HttpClientOxmTemplate` which are light wrappers on top of Apache's `HttpClient` providing Spring style code based templating. The latter template provides marshaling and unmarshalling of XML based requests.

Built on top of the `HttpClientTemplate` and `HttpClientOxmTemplate`, the `SolrOxmClient` provides for easier client communication with Apache Solr [http://lucene.apache.org/solr/]. Solr [http://lucene.apache.org/solr/] provides an XML based API over HTTP to the Apache Lucene [http://lucene.apache.org/] search engine. `SolrOxmClient` marshalls/unmarshalls searches and updates to and from a JavaBean. It also allows calls to commit, rollback, and optimize.

The Logger `BeanPostProcessor` provides logger creation and injection based on reflection, interfaces, or annotations.

The `ImageProcessor` [http://springbyexample.org/maven/site/sbe-util/1.2.3/sbe-util/apidocs/org/springbyexample/util/image/ImageProcessor.html] is a utility to process images and currently can help scale and image from one size to another either using a

`java.io.File` or a `java.io.InputStream` to a `java.io.OututStream`.

- Spring by Example Utils Module
- Spring by Example Utils Module Site [http://springbyexample.org/maven/site/sbe-util/1.2.3/sbe-util/]
- Download
  - Maven Dependency Instructions or Spring by Example Utils 1.2.3 Module JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-util/1.2.3/sbe-util-1.2.3.jar]

Spring by Example Web Module

There is a Spring GWT Controller for standard GWT usage and also Spring Bayeux integration for using Comet on Jetty.

There is an `ImageInterceptor` that intercepts a request and looks in a directory that matches the requests relative path. Currently it only matches one image extension type which defaults to '.jpg'. It generates a thumbnail if one doesn't exist and also makes a list of available thumbnails and images.

- Spring by Example Web Module
- Spring by Example Web Module Site [http://springbyexample.org/maven/site/sbe-web/1.1.3/sbe-web/]
- Download
  - Maven Dependency Instructions or Spring by Example Web Module 1.1.3 JAR [http://springbyexample.org/maven/repo/org/springbyexample/sbe-web/1.1.3/sbe-web-1.1.3.jar]

Spring by Example Custom *ServletContext* Scope Module

David Winterfeldt

2008

The Spring by Example Custom *ServletContext* Scope module is a custom scope implementation for providing *ServletContext* (web application) scoped beans.

It can be useful for sharing a Spring bean between web applications if your servlet container has cross context support. Tomcat has cross context support that can be turned on by setting *crossContext="true"* on the *Context*. See Tomcat's context documentation for more details.

Note

Even though a bean can be shared between web contexts, only classes loaded by a parent classloader can be shared. The easiest thing to do is either share data as XML or in a `Map` or `String` (something JVM's core). Then there won't be any `ClassCastExceptions`. Or the server can be configured to have the shared class in the parent classloader. For example, in Tomcat the jar with the shared class could be put in Tomcat's lib directory, but this should be avoided as much as possible since it will cause your web applications to be tightly coupled together.

1. Spring Configuration

The custom `ServletContextScope` is registered with the `CustomScopeConfigurer` under the key 'servletContext'. The key can be used to specify the scope on a bean just like the default scopes 'singleton' or 'prototype'.

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="servletContext">
                <bean class="org.springbyexample.web.context.ServletContextScope" />
            </entry>
        </map>
    </property>
</bean>
```

This is the same, but it specifies a specific context to use for storing and retrieving values.

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
```

```
    <property name="scopes">
        <map>
            <entry key="servletContext">
                <bean class="org.springbyexample.web.context.ServletContextScope" />
                    <property name="context" value="/advanced-form" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

2. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
    <groupId>org.springbyexample</groupId>
    <artifactId>sbe-servlet-context-scope</artifactId>
    <version>1.0.2</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

3. Reference

Related Links

- Custom *ServletContext* Scope Site [http://springbyexample.org/maven/site/sbe-servlet-context-scope/1.0.2/sbe-servlet-context-scope/]
- Spring 3.1.x Custom Scopes Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-factory-scopes-custom]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-servlet-context-scope
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Spring by Example Custom Thread Scope Module

David Winterfeldt

2008

The Custom Thread Scope module is a custom scope implementation for providing thread scoped beans. Every request for a bean will return the same instance for the same thread. A `Runnable` must be wrapped in a `ThreadScopeRunnable` if destruction callbacks should occur on a thread scoped bean.

Note

See `SimpleThreadScope` [http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/context/support/SimpleThreadScope.ht which was added in Spring 3.0, for a Spring Framework implementation Although the Spring version doesn't support destruction callbacks (this implementation does when using a custom Runnable).

1. Spring Configuration

The *threadCounter* bean is set to use the custom `ThreadScope` and the `CustomScopeConfigurer` registers the custom scope.

```
<bean id="threadCounter"
      class="org.springbyexample.bean.scope.thread.Counter"
      scope="thread" />

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread">
                <bean class="org.springbyexample.bean.scope.thread.ThreadScope"/>
            </entry>
        </map>
    </property>
</bean>
```

2. Code Example

Below is the code for the `ThreadScopeRunnable` class. It can be used to wrap another `Runnable` so when it's finished running it clears the current thread scoped variables in order for destruction callbacks to run. There is also a `ThreadScopeCallable` class for wrapping a `Callable`. If you have your own custom `Thread` implementations, they can call `ThreadScopeContextHolder.currentThreadScopeAttributes().clear()` directly.

```
public class ThreadScopeRunnable implements Runnable {

    protected Runnable target = null;

    /**
     * Constructor
     */
    public ThreadScopeRunnable(Runnable target) {
        this.target = target;
    }

    /**
     * Runs <code>Runnable</code> target and
     * then afterword processes thread scope
     * destruction callbacks.
     */
    public final void run() {
        try {
            target.run();
        } finally {
            ThreadScopeContextHolder.currentThreadScopeAttributes().clear();
        }
    }

}
```

3. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
    <groupId>org.springbyexample</groupId>
    <artifactId>sbe-thread-scope</artifactId>
    <version>1.0.2</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

4. Reference

Related Links

- Custom Thread Scope Site [http://springbyexample.org/maven/site/sbe-thread-scope/1.0.2/sbe-thread-scope/]
- Spring 3.1.x Custom Scopes Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-factory-scopes-custom]
- Spring JIRA 2581 - Provide out of the box implementation of the thread scope [http://jira.springframework.org/browse/SPR-2581]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-thread-scope
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Spring by Example's Dynamic Tiles 2 Spring MVC Module

David Winterfeldt

2008

The Dynamic Tiles 2 Spring MVC Module [http://springbyexample.org/maven/site/sbe-dynamic-tiles2/1.2.1/sbe-dynamic-tiles2/], version 1.2, dynamically renders a Tiles 2 [http://tiles.apache.org/] template with Spring MVC. Any request coming in mapped for Tiles processing will use the default template and dynamically insert the body based on the URL.

Besides basic support for rendering dynamically rendering Tiles templates, there is also support for rendering Tiles fragments like `AjaxTilesView` and `FlowAjaxTilesView` in Spring JS and Spring Web Flow

1. Spring Configuration

The *tilesConfigurer* bean initializes tiles with all the tiles configuration files (more than one can be specified). The *tilesViewResolver* bean defines using `DynamicTilesView` which uses the url to lookup the Tiles definition, dynamically insert the body into the definition, and render it.

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

    <bean id="tilesConfigurer"
     class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
        <property name="definitions">
            <list>
                <value>/WEB-INF/tiles-defs/templates.xml</value>
            </list>
        </property>
    </bean>

    <bean id="tilesViewResolver"
     class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass"
value="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView" />
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/**/*.html">viewController</prop>
            </props>
        </property>
    </bean>

    <bean id="viewController"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

```
</beans>
```

For the AJAX requests to work correctly an `AjaxUrlBasedViewResolver` must be configured and `FlowAjaxDynamicTilesView` as the view class.

```
<bean id="tilesViewResolver"
      class="org.springframework.js.ajax.AjaxUrlBasedViewResolver">
    <property name="viewClass"
value="org.springbyexample.web.servlet.view.tiles2.FlowAjaxDynamicTilesView" />
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

2. Tiles XML Configuration

The Tiles '.mainTemplate' sets up the default layout. Each page is dynamically rendered by dynamically setting the body on the derived Tiles definition. Which in this case is '.mainTemplate'.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

    <!-- Default Main Template -->
    <definition name=".mainTemplate" template="/WEB-INF/templates/main.jsp">
        <put-attribute name="title" value="Simple Tiles 2 Example" type="string" />
        <put-attribute name="header" value="/WEB-INF/templates/header.jsp" />
        <put-attribute name="footer" value="/WEB-INF/templates/footer.jsp" />
        <put-attribute name="menu" value="/WEB-INF/templates/menu.jsp" />
        <put-attribute name="content" value="/WEB-INF/templates/blank.jsp" />
    </definition>

</tiles-definitions>
```

3. Tiles JSP Example

This JSP has the main layout for where the header, footer, menu, and body are located. They are inserted using Tiles custom JSP tags.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>

<html>
<head>
<title><tiles:getAsString name="title" /></title>
<link rel="stylesheet" type="text/css" href="<c:url value="/css/main.css"/>" />
</head>
<body>
<div id="header">
    <div id="headerTitle"><tiles:insertAttribute name="header" /></div>
</div>
<div id="menu">
    <tiles:insertAttribute name="menu" />
</div>
<div id="content">
    <td><tiles:insertAttribute name="content" />
</div>
<div id="footer">
    <tiles:insertAttribute name="footer" />
</div>
</body>
</html>
```

4. `DynamicTilesView`

Processing Order

If a request is made for '/info/index.html', Spring will pass 'info/index' into the view. The first thing will be to look for 'info/index' as a Tiles definition. Then a template definition of '.info.mainTemplate', which if found will dynamically have a body set on this definition. If the previous aren't found, it is assumed a root definition exists. This would be '.mainTemplate'. If none of these exist, a `TilesException` will be thrown.

1. Check if a Tiles definition based on the URL matches.
2. Checks if a definition derived from the URL and default template name exists and then dynamically insert the body based on the URL.
3. Check if there is a root template definition and then dynamically insert the body based on the URL.
4. If no match is found from any process above a `TilesException` is thrown.

The following are the default values for determining a Tiles definition for a request. If these aren't acceptable, they can be changed using `TilesUrlBasedViewResolver` which is a subclass of `UrlBasedViewResolver`. Or for AJAX support `TilesAjaxUrlBasedViewResolver`, which is a subclass of `AjaxUrlBasedViewResolver`, can be used.

Table 1. `DynamicTilesView` Defaults

Property	**Default Value**
tilesDefinitionName	mainTemplate

Property	Default Value
tilesBodyAttributeName	content
tilesDefinitionDelimiter	.

```
<bean id="dynamicTilesViewResolver"
      class="org.springbyexample.web.servlet.view.tiles2.TilesUrlBasedViewResolver">
    <property name="viewClass" value="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView"
/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
    <property name="tilesDefinitionName" value="root" />
    <property name="tilesBodyAttributeName" value="content" />
    <property name="tilesDefinitionDelimiter" value="-" />
</bean>
```

5. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
        <groupId>org.springbyexample</groupId>
        <artifactId>sbe-dynamic-tiles2</artifactId>
        <version>1.2.1</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

6. Reference

Related Links

- Dynamic Tiles 2 Site [http://springbyexample.org/maven/site/sbe-dynamic-tiles2/1.2.1/sbe-dynamic-tiles2/]
- Spring 3.1.x Tiles Documentation

http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/view.html#view-tiles]

- Tiles 2 [http://tiles.apache.org/]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-dynamic-tiles2
```

Dynamic Tiles 2 Webapp Example

```
$ svn co http://svn.springbyexample.org/web/dynamic-tiles2-webapp/tags/1.2/ dynamic-tiles2-webapp
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Tiles 2

Spring by Example JCR Module

David Winterfeldt

2008

The Spring by Example JCR module uses Spring Modules JCR (Java Content Repository) module. Currently the utilities provide a way to recurse through the repositories nodes using JcrTemplate and a custom node callback for each matching node found while recursing the repository. This example uses Apache Jackrabbit for the Java Content Repository which is the reference implementation for JSR-170.

1. Spring Configuration

The first bean definition defines defines the Jackrabbit repository by specifying the configuration file to use and the location of the repository. If the repository doesn't already exist, it will be created on startup. The next bean creates a session factory based on the respository and the next one creates a `JcrTemplate` using the session factory.

```
<?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"
       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.springframework.org/schema/context/spring-context.xsd">

    <!-- Register Annotation-based Post Processing Beans -->
    <context:annotation-config />

    <!-- Scan context package for any eligible annotation configured beans. -->
    <context:component-scan base-package="org.springbyexample.jcr" />

    <!-- normal repository -->
    <bean id="repository"
        class="org.springmodules.jcr.jackrabbit.RepositoryFactoryBean">
        <!-- normal factory beans params -->
        <property name="configuration" value="classpath:/jackrabbit-repository.xml" />
        <!-- use the target folder which will be cleaned  -->
        <property name="homeDir" value="file:./target/repo" />
    </bean>

    <bean id="sessionFactory"
          class="org.springmodules.jcr.jackrabbit.JackrabbitSessionFactory">
        <property name="repository" ref="repository" />
    </bean>

    <bean id="jcrTemplate" class="org.springmodules.jcr.JcrTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
        <property name="allowCreate" value="true" />
    </bean>

    <bean id="processor" class="org.springbyexample.jcr.JcrContentExporter">
        <property name="contentRecurser" ref="jcrContentRecurser" />
        <property name="rootFolder" value="./target/repo_export" />
    </bean>

</beans>
```

2. Code Example

The example shows calling `JcrContentExporter` which is defined in the Spring configuration. It just needs a reference to the `JcrContentRecurser`, which is automatically loaded by the *context:component-scan* since it has an `@Component` annotation, and also a file path where to export all content nodes in the repository.

This is an excerpt from `JcrContentExporterIT`.

```
@Autowired
private JcrContentExporter processor = null;

...

public void testJcrContentProcessor() {
...

    List<String> lResults = processor.process();

...
}
```

Example 1 Content Exporter

This example is from `JcrContentExporter`. It processes each content node and writes the content to an export directory, but obviously any type of processing can be done on the content node inside the callback.

For every content node found while recursing the repository `JcrNodeCallback.doInJcrNode(Session session, Node node)` is called. The file name and relative path from the root node is derived and then the content is retrieved and written to the export directory.

```
public List<String> process() {
    final List<String> lResults = new ArrayList<String>();

    contentRecurser.recurse(new JcrNodeCallback() {
        public void doInJcrNode(Session session, Node node)
                throws IOException, RepositoryException {
            // file name node is above content
            String fileName = node.getParent().getName();
            // file path is two above content
            String path = node.getParent().getParent().getPath();

            logger.debug("In content recurser callback." +
                         "  fileName='{}' path='{}'", fileName, path);

            File fileDir = new File(rootFolder + path);
            File file = new File(fileDir, fileName);

            Property dataProperty = node.getProperty(JcrConstants.JCR_DATA);

            InputStream in = null;
            OutputStream out = null;

            try {
                FileUtils.forceMkdir(fileDir);

                in = dataProperty.getStream();
                out = new FileOutputStream(file);
```

```
                IOUtils.copy(in, out);

                lResults.add(path + File.pathSeparator + fileName);
            } finally {
                IOUtils.closeQuietly(in);
                IOUtils.closeQuietly(out);
            }
        }
    });

    return lResults;
}
```

Example 2 Using `JcrContentRecurser`

The previous example of `JcrContentRecurser` is a very simple subclass of `JcrRecurser`. If it no specific node matching is set on it, it's callback will be called for every node in the repository. But a `Set`of specific node names to match can be passed in either through the constructor or methods.

Below is `JcrContentRecurser` which just sets the content node as the only node to have callbacks performed.

```
public class JcrContentRecurser extends JcrRecurser {

    final Logger logger = LoggerFactory.getLogger(JcrContentRecurser.class);

    /**
     * Constructor
     */
    public JcrContentRecurser() {
        super(JcrConstants.JCR_CONTENT);
    }

}
```

Example 3 Using `JcrRecurser`

3. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
    <groupId>org.springbyexample</groupId>
    <artifactId>sbe-jcr</artifactId>
    <version>1.0.2</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
```

```
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

4. Reference

Related Links

- Spring by Example JCR Site [http://springbyexample.org/maven/site/sbe-jcr/1.0.2/sbe-jcr/]
- Spring Modules [http://www.springsource.org/spring-modules]
- Apache Jackrabbit [http://jackrabbit.apache.org/]
- Integrating Java Content Repository and Spring [http://www.infoq.com/articles/spring-modules-jcr]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-jcr
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Spring by Example Utils Module

David Winterfeldt

2008

The Spring by Example Utils module currently has the `HttpClientTemplate` and `HttpClientOxmTemplate` which are light wrappers on top of Apache's `HttpClient` providing Spring style code based templating. The latter template provides marshaling and unmarshalling of XML based requests.

The Logger `BeanPostProcessor` provides logger creation and injection based on reflection, interfaces, or annotations.

The `ImageProcessor` [http://springbyexample.org/maven/site/sbe-util/1.2.3/sbe-util/apidocs/org/springbyexample/util/image/ImageProcessor.html] is a utility to process images and currently can help scale and image from one size to another either using a `java.io.File` or a `java.io.InputStream` to a `java.io.OututStream`.

1. `HttpClientTemplate`

`HttpClientTemplate` which uses Apache's `HttpClient` to process HTTP requests and receive the data as a `String`, `InputStream`, or `byte[]`.

Spring Configuration

Basic Configuration

Set a default URI for the `HttpClientTemplate`.

HttpClientTemplateTest-context.xml

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

    <bean id="httpClient" class="org.springbyexample.httpclient.HttpClientTemplate">
        <property name="defaultUri">
            <value><![CDATA[http://localhost:8093/test]]></value>
        </property>
    </bean>

</beans>
```

HTTP Authorization Configuration

The `HttpClient` instance used by `HttpClientTemplate` could be set during configuration with authorization, but all of the default authorization classes must be set using constructors. The Spring by Example authorization beans allow the use of setters so are a little more Spring friendly. The example below will use the userName and password when challenged for authentication by a request to *localhost:8093*.

When accessing a site that will challenge any request, *authenticationPreemptive* can be set to `true`. Then the credentials will be sent along with every request instead of a challenging needed to be issues before the credentials are sent.

HttpClientTemplateAuthTest-context.xml

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

    <bean id="httpClient" class="org.springbyexample.httpclient.HttpClientTemplate">
        <property name="defaultUri">
            <value><![CDATA[http://localhost:8093/admin/test]]></value>
        </property>
        <property name="credentials">
            <list>
                <bean class="org.springbyexample.httpclient.auth.Credentials">
                    <property name="authScopeHost" value="localhost" />
                    <property name="authScopePort" value="8093" />
                    <property name="userName" value="jsmith" />
                    <property name="password" value="password" />
                </bean>
            </list>
        </property>
    </bean>

</beans>
```

Code Example

Excerpts from `HttpClientTemplateTest`.

```
template.executeGetMethod(new ResponseStringCallback() {
    public void doWithResponse(String response) throws IOException {
        ...

        logger.debug("HTTP Get string response. '{}'", response);
    }
});
```

Example 1 HTTP Get Response with a `String`

```
template.executeGetMethod(new ResponseStreamCallback() {
```

```
    public void doWithResponse(InputStream in) throws IOException {
        String response = IOUtils.toString(in);

        ...

        logger.debug("HTTP Get stream response. '{}'", response);
    }
});
```

Example 2 HTTP Get Response with an `InputStream`

```
template.executeGetMethod(new ResponseByteCallback() {
    public void doWithResponse(byte[] byteResponse) throws IOException {
        String response = new String(byteResponse);

        ...

        logger.debug("HTTP Get byte response. '{}'", response);
    }
);
```

Example 3 HTTP Get Response with a Byte Array

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put(GET_PARAM_INPUT_KEY, GET_PARAM_INPUT_VALUE);

template.executeGetMethod(hParams,
    new ResponseStringCallback() {
        public void doWithResponse(String response) throws IOException {
            ...

            logger.debug("HTTP Get with params string response. '{}'", response);
        }
});
```

Example 4 HTTP Get with Parameters & Response with a `String`

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put(LOWER_PARAM, POST_NAME);

template.executePostMethod(hParams,
    new ResponseStringCallback() {
        public void doWithResponse(String response) throws IOException {
            ...

            logger.debug("HTTP Post string response. '{}'", response);
        }
});
```

Example 5 HTTP Post with Parameters & Response with a `String`

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put(LOWER_PARAM, POST_NAME);

template.executePostMethod(hParams,
    new ResponseStreamCallback() {
        public void doWithResponse(InputStream in) throws IOException {
            String response = IOUtils.toString(in);

            ...

            logger.debug("HTTP Post stream response. '{}'", response);
        }
});
```

Example 6 HTTP Post with Parameters & Response with an `InputStream`

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put(LOWER_PARAM, POST_NAME);

template.executePostMethod(hParams,
    new ResponseByteCallback() {
        public void doWithResponse(byte[] byteResponse) throws IOException {
            String response = new String(byteResponse);

            ...

            logger.debug("HTTP Post byte response. '{}'", response);
        }
});
```

Example 7 HTTP Post with Parameters & Response with a Byte Array

A post sending data, like XML, in the request for the server to process and return results.

```
protected final static String POST_DATA_INPUT = "<message>Greetings</message>";

...

template.executePostMethod(POST_DATA_INPUT,
    new ResponseStringCallback() {
        public void doWithResponse(String response) throws IOException {
            ...

            logger.debug("HTTP string data post string response. '{}'", response);
        }
});
```

Example 8 HTTP Post with Data & Response with a `String`

2. `HttpClientOxmTemplate`

`HttpClientOxmTemplate` which uses Apache's HttpClient [http://hc.apache.org/] to process HTTP requests and Spring Web Service's OXM [http://static.springframework.org/spring-ws/site] to marshall and unmarshall the requests.

Spring Configuration

Define a marshaller/unmarshaller and set it on `HttpClientOxmTemplate` along with it's default URI.

HttpClientOxmTemplateTest-context.xml

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

    <bean id="jaxbMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="contextPath" value="org.springbyexample.schema.beans"/>
    </bean>

    <bean id="httpClient" class="org.springbyexample.httpclient.HttpClientOxmTemplate">
        <property name="defaultUri">
            <value>http://localhost:8093/test</value>
        </property>
        <property name="marshaller" ref="jaxbMarshaller" />
        <property name="unmarshaller" ref="jaxbMarshaller" />
    </bean>

</beans>
```

Code Example

The instance of Persons is marshalled and the XML is posted to the default uri and the callback has the XML result unmarshalled back to an instance of `Persons`. The class passed in and returned in this example match, but they don't need to since a different marshaller and unmarshaller can be used.

```
template.executePostMethod(persons, new ResponseCallback<Persons>() {
    public void doWithResponse(Persons persons) throws IOException {
        ...

        Person result = persons.getPerson().get(0);

        ...

        logger.debug("id={}  firstName={}  lastName={}",
                     new Object[] { result.getId(),
                                    result.getFirstName(),
                                    result.getLastName()});
    }
});
```

Example 9 Excerpt from `HttpClientOxmTemplateTest.testStringDataPostMethodWithStringResponse()`

3. `SolrOxmClient`

The `SolrOxmClient` provides for easier client communication with Apache Solr [http://lucene.apache.org/solr/]. It marshalls/unmarshalls searches and updates to and from a JavaBean. It also allows calls to commit, rollback, and optimize. It is built on top of `HttpClientTemplate` and `HttpClientOxmTemplate`.

Note

The Spring configuration and code are from the Solr Client example and are not part of this module.

Spring Configuration

A base URL to Solr [http://lucene.apache.org/solr/] and a marshaller and unmarshaller must be defined to use it. To read more about the test look at the Solr Client example.

HttpClientOxmTemplateTest-context.xml

```
<bean id="solrOxmClient" class="org.springbyexample.httpclient.solr.SolrOxmClient"
      p:baseUrl="http://${solr.host}:${solr.port}/solr"
      p:marshaller-ref="catalogItemMarshaller"
      p:unmarshaller-ref="catalogItemMarshaller" />
```

Code Example

Any Solr [http://lucene.apache.org/solr/] search parameter can be passed in and the results are unmarshalled into a specific JavaBean.

```
List<CatalogItem> lCatalogItems = client.search(SEARCH_QUERY_PARAM);
```

Example 10 Excerpt from `SolrOxmClientIT.testSearch()`

To pass in other parameters to the search, a `Map` can be passed in. The query is specified by the 'q' key, and the 'start' & 'rows' indicate what range of the results to return.

```
Map<String, String> hParams = new HashMap<String, String>();
hParams.put("q", "electronics");
hParams.put("start", "5");
hParams.put("rows", "5");
```

```
List<CatalogItem> lCatalogItems = client.search(hParams);
```

Example 11 Excerpt from `SolrOxmClientIT.testPaginatedSearch()`

Updates a list of `CatalogItem` instances.

```
client.update(lCatalogItems);
```

Example 12 Excerpt from `SolrOxmClientIT.testUpdate()`

4. Logger `BeanPostProcessor`

There are different logger `BeanPostProcessors` to dynamically inject a logger based on reflection, interfaces, or annotations. Many loggers are defined with a static reference to the a class for each class and a developer cuts & pastes each logger definition. This can be error prone and the Logger `BeanPostProcessors` help deal with this problem.

Spring Configuration

All of the logger `BeanPostProcessors` will ignore any logger that isn't defined in the `Map` of logger factories. The defaults defined are SLF4J, Apache Commons Logging, Log4J, and JDK Logging. A custom list can be set using the *loggerFactoryMap* property.

Table 1. Default Loggers

Logger	**Logger Factory Static Method**
`org.slf4j.Logger`	`org.slf4j.LoggerFactory.getLogger`
`org.apache.commons.logging.Log`	`org.apache.commons.logging.LogFactory.getLog`
`org.apache.log4j.Logger`	`org.apache.log4j.Logger.getLogger`
`java.util.logging.Logger`	`java.util.logging.Logger.getLogger`

The `LoggerBeanPostProcessor` uses reflection to set a logger on any method that matches the specified method and the logger being set on it matches the defined loggers and logger factories.

Reflection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="org.springbyexample.util.log.LoggerBeanPostProcessor">
        <property name="methodName" value="setCustomLogger" />
    </bean>

    <bean id="loggerBean" class="org.springbyexample.util.log.LoggerBean" />

</beans>
```

The `LoggerAwareBeanPostProcessor` only sets subclasses of the `LoggerAware` interface. The current interfaces are `Slf4JLoggerAware`, `CommonsLoggerAware`, `Log4JLoggerAware`, and `JdkLoggerAware`.

Interfaces

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

    <bean class="org.springbyexample.util.log.LoggerAwareBeanPostProcessor" />

    <bean id="slf4jLoggerBean" class="org.springbyexample.util.log.Slf4JLoggerBean" />

    <bean id="commonsLoggerBean" class="org.springbyexample.util.log.CommonsLoggerBean" />

    <bean id="log4jLoggerBean" class="org.springbyexample.util.log.Log4JLoggerBean" />

    <bean id="jdkLoggerBean" class="org.springbyexample.util.log.JdkLoggerBean" />

</beans>
```

The `AnnotationLoggerBeanPostProcessor` can be used to inject loggers into fields and methods that have the `AutowiredLogger` annotation.

Annotations

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

    <bean class="org.springbyexample.util.log.AnnotationLoggerBeanPostProcessor" />

    <bean id="loggerBean" class="org.springbyexample.util.log.AnnotationLoggerBean" />

</beans>
```

Code Example

Examples of the beans having loggers inject by the `BeanPostProcessors` defined in the Spring configurations above.

```
public class LoggerBean {

    Logger logger = null;

    /**
     * Gets SLF4J logger.
     */
    public Logger getCustomLogger() {
        return logger;
    }

    /**
     * Sets SLF4J logger.
     */
    public void setCustomLogger(Logger logger) {
        this.logger = logger;
    }

}
```

Example 13 Reflection

```
public class Slf4JLoggerBean implements Slf4JLoggerAware {

    Logger logger = null;

    /**
     * Gets SLF4J logger.
     */
    public Logger getLogger() {
        return logger;
    }

    /**
     * Sets SLF4J logger.
     */
    public void setLogger(Logger logger) {
        this.logger = logger;
    }

}
```

Example 14 Interfaces

```
@AutowiredLogger
final Logger logger = null;
```

```
@AutowiredLogger
public void setMethodLogger(Logger methodLogger) {
    this.methodLogger = methodLogger;
}
```

Example 15 Annotations

5. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
    <groupId>org.springbyexample</groupId>
    <artifactId>sbe-util</artifactId>
    <version>1.2.3</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

6. Reference

Related Links

- Spring by Example Utils Site [http://springbyexample.org/maven/site/sbe-util/1.2.3/sbe-util/]
- Apache's `HttpClient` [http://hc.apache.org/]
- Solr Client

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-util
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x
- Apache Commons `HttpClient` 3.1

Spring by Example Web Module

David Winterfeldt

2008

There is a Spring GWT Controller for standard GWT usage and also Spring Bayeux integration for using Comet on Jetty.

There is an `ImageInterceptor` that intercepts a request and looks in a directory that matches the requests relative path. Currently it only matches one image extension type which defaults to '.jpg'. It generates a thumbnail if one doesn't exist and also makes a list of available thumbnails and images.

1. Spring GWT Controller

Spring Configuration

Even though the service controller is annotation-based, since GWT calls RPC methods using reflection, the mapping has to be manually set using the `SimpleUrlHandlerMapping`.

```
<?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"
    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.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springbyexample.web.gwt.server" />

    <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="order" value="0" />
        <property name="mappings">
            <value>
                /person/service.do=serviceController
            </value>
        </property>
    </bean>

</beans>
```

Code Example

```
@Controller
public class ServiceController extends GwtController implements Service {

    final Logger logger = LoggerFactory.getLogger(ServiceController.class);

    private static final long serialVersionUID = -2103209407529882816L;

    @Autowired
    private PersonDao personDao = null;

    /**
     * Finds person within a range.
     */
    public Person[] findPersons(int startIndex, int maxResults) {
        Person[] results = null;

        List<Person> lResults = new ArrayList<Person>();

        Collection<org.springbyexample.orm.hibernate3.annotation.bean.Person> lPersons =
personDao.findPersons(startIndex, maxResults);

        for (org.springbyexample.orm.hibernate3.annotation.bean.Person person : lPersons) {
            Person result = new Person();
            result.setId(person.getId());
            result.setFirstName(person.getFirstName());
            result.setLastName(person.getLastName());

            lResults.add(result);
        }

        return lResults.toArray(new Person[]{});
    }

}
```

2. Spring Bayeux Integration for Comet on Jetty

Web Configuration

The `SpringContinuationCometdServlet` is configured to handle all Comet requests. It only needs the *asyncDeliver* value set on it and the other values you would normally set on `ContinuationCometdServlet` are all set on `SpringContinuationBayeux` in the Spring configuration.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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"
         version="2.5">
    <display-name>Spring Cometd Test WebApp</display-name>
```

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/web-application-context.xml</param-value>
    </context-param>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <servlet>
        <servlet-name>cometd</servlet-name>
<servlet-class>org.springbyexample.cometd.continuation.SpringContinuationCometdServlet</servlet-class>
        <init-param>
            <param-name>asyncDeliver</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>cometd</servlet-name>
        <url-pattern>/cometd/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Spring Configuration

```


[
    {
        "channels": "/**",
        "filter"  : "org.mortbay.cometd.filter.NoMarkupFilter",
        "init"    : {}
    },

    {
        "channels": "/chat/*",
        "filter"   : "org.mortbay.cometd.filter.RegexFilter",
        "init"    : [
                        [ "[Ss]pring [Bb]y [Ee]xample","Spring by Example" ],
                        [ "[Ss][Bb][Ee]","Spring by Example" ]
                    ]
    },

    {
        "channels": "/chat/**",
        "filter"   : "org.mortbay.cometd.filter.RegexFilter",
        "init"    : [
                              [ "teh ","the "],
                              [ "sring ","spring "]
                    ]
    }

]
            ]]>
        </value>
    </property>
</bean>
```

Code Example

The only difference in this compared to a standard `BayeuxService` is that this Bayeux implementation is instantiated by Spring and the Bayuex instance is injected into it's constructor.

```
@Component
public class ChatService extends BayeuxService {

    final Logger logger = LoggerFactory.getLogger(ChatService.class);

    final ConcurrentMap<String, Set<String>> _members = new ConcurrentHashMap<String,
Set<String>>();

    /**
     * Constructor
     */
    @Autowired
    public ChatService(Bayeux bayeux) {
        super(bayeux, "chat");
        subscribe("/chat/**", "trackMembers");
    }

    /**
     * Tracks chat clients.
     */
    public void trackMembers(Client joiner, String channel,
                             Map<String, Object> data, String id) {
        if (Boolean.TRUE.equals(data.get("join"))) {
            Set<String> m = _members.get(channel);

            if (m == null) {
                Set<String> new_list = new CopyOnWriteArraySet<String>();
                m = _members.putIfAbsent(channel, new_list);
                if (m == null) {
                    m = new_list;
                }
            }

            final Set<String> members = m;
            final String username = (String) data.get("user");

            members.add(username);

            joiner.addListener(new RemoveListener() {
                public void removed(String clientId, boolean timeout) {
                    members.remove(username);

                    logger.info("members: " + members);
                }
            });

            logger.info("Members: " + members);

            send(joiner, channel, members, id);
        }
    }

}
```

3. Image Interceptor

Used for generic thumbnail processing. If a request for '/pics/newyork.html' is received, the image interceptor will look for a directory at '/var/www/html/images/newyork'. If the directory exists generate any missing thumbnails and have the view rendered by the *imageViewName* property.

Spring Configuration

```
<?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"
       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.springframework.org/schema/context/spring-context.xsd">

    <!--Setup interceptor for annotation-based controllers -->
    <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="localeChangeInterceptor"/>
                <ref bean="imageInterceptor"/>
            </list>
        </property>
        <property name="defaultHandler">
            <bean class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />
        </property>
    </bean>
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

    <context:annotation-config />

    <bean id="tilesConfigurer"
          class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
        <property name="definitions">
            <list>
                <value>/WEB-INF/tiles-defs/templates.xml</value>
            </list>
        </property>
    </bean>

    <bean id="dynamicTilesViewResolver"
          class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass"
value="org.springbyexample.web.servlet.view.tiles2.DynamicTilesView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <value>messages</value>
        </property>
    </bean>

    <!-- Declare the Interceptor -->
    <bean id="localeChangeInterceptor"
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="locale"/>
    </bean>

    <!-- Declare the Resolver -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
```

```
    <bean name="imageInterceptor" class="org.springbyexample.web.servlet.image.ImageInterceptor">
        <property name="imageProcessor">
            <bean class="org.springbyexample.util.image.ImageProcessorImpl" />
        </property>
        <property name="rootImagePath" value="/var/www/html" />
        <property name="imageViewName" value="/pics/generic_display" />
    </bean>

</beans>
```

JSP Example

JSP referenced by the *imageViewName* property defined in the *imageInterceptor* bean.

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<table>
<c:forEach var="item" items="${imageList}" varStatus="status">
    <c:if test="${(status.index % rowWidth) == 0}">
        <tr>
    </c:if>

    <td>
        <a href="${item.imagePath}"><img src="${item.thumbnailPath}"></a>
    </td>

    <c:if test="${(status.index % 3) == (rowWidth -1)}">
        </tr>
    </c:if>
</c:forEach>
</table>
```

4. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
    <groupId>org.springbyexample</groupId>
    <artifactId>sbe-web</artifactId>
    <version>1.1.3</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
```

```
    </repository>
</repositories>
```

5. Reference

Related Links

- Spring by Example Web Module Site [http://springbyexample.org/maven/site/sbe-web/1.1.3/sbe-web/]
- Spring 3.1.x MVC Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html]
- GWT (Google Web Toolkit) [http://code.google.com/webtoolkit/]
- GWT Widget Library [http://gwt-widget.sourceforge.net/] (see `GWTSpringController`)
- Jetty Continuations [http://docs.codehaus.org/display/JETTY/Continuations]
- Simple GWT Spring Webapp

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-web
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Spring Modules Validation

David Winterfeldt

2009

The Spring Modules [http://www.springsource.org/spring-modules] project has a number of subprojects, including validation. This module is based on Spring Modules [http://www.springsource.org/spring-modules] Validation Version 0.9 and has a number of enhancements to the Valang part of the project.

Thanks to everyone that has worked on this project previously and currently.

1. Valang

Valang is short for **Va**-*lidation* **Lang**-*uage*. It provides a very readable language for expressing validation rules, as well as providing the ability to add custom functions to the language. Once the `ValangValidator` is configured, using it isn't different than any other Spring Validator since it implements `org.springframework.validation.Validator`.

Below is a list of current enhancements.

Version 0.91

- Bytecode generation added to DefaultVisitor as a replacement for reflection accessing simple properties (BeanPropertyFunction) for a significant performance improvement.
- Basic enum comparison support. In the expression below the personType is an enum and the value `STUDENT` will be convereted to an enum for comparison. The value must match an enum value on the type being compared or an exception will be thrown.

  ```
  personType EQUALS ['STUDENT']
  ```

  For better performance the full class name can be specified so the enum can be retrieved during parsing. The first example is for standard enum and the second one is for an inner enum class .

  ```
  personType                                                                EQUALS
  ['org.springmodules.validation.example.PersonType.STUDENT']
  ```

  ```
  personType                                                                EQUALS
  ['org.springmodules.validation.example.Person$PersonType.STUDENT']
  ```

- Where clause support. In the expression below, the part of the expression `price < 100` will only be evaluated if the `personType` is 'STUDENT'. Otherwise the validation will be skipped.

  ```
  price < 100 WHERE personType EQUALS ['STUDENT']
  ```

Note

Support for the where clause has not been added to the JavaScript custom tag currently.

- Improved performance of 'IN'/'NOT IN' if comparing a value to a java.util.Set it will use Set.contains(value). Static lists of Strings (ex: 'A', 'B', 'C') are now stored in a Set instead of an ArrayList.
- Functions can be configured in Spring, but need to have their scope set as prototype and use a FunctionWrapper that is also a prototype bean with *<aop:scoped-proxy>* set on it.
- Removed servlet dependency from Valang project except for the custom JSP tag ValangValidateTag needing it, but running Valang no longer requires it. This involved removing ServletContextAware from it's custom dependency injection. If someone was using this in a custom function, the function can now be configured directly in Spring and Spring can inject any "aware" values.
- Changed logging to use SLF4J api.

Version 0.92

- Removed custom dependency injection since functions can be configured in Spring.
- Added auto-discovery of FunctionWrapper beans from the Spring context to go with existing auto-discovery of FunctionDefinition beans.

Version 0.93

- Made specific comparison classes for each operator for a performance improvement.
- Changed IS WORD and IS BLANK to use Commons Lang StringUtils, which will change the behavior slightly but should be more accurate to the description of the validation.
- Change Operator from interfaces to an enum and removed OperatorConstants.
- Fixed bytecode generation to handle a `Map`, a `List`, and an Array.

Version 0.94

- Upgraded to Spring 3.0 and changed group & artifact IDs to match standard Spring naming conventions.

Version 0.95

- Upgraded to Spring 3.1 and minor improvements to bytecode generation.

Rule Syntax

The basic construction of a Valang rule is to have it begin and end with a brace. Within the braces, the default property name for the rule is specified first. Then the Valang expression, followed by the default error message. These are all the required values for a Valang rule. The other optional values for a rule are the error message key and arguments for it. Each of the values of the rule are delimitted by a colon.

{ <property-name> : <expression> : <default-error-message> *: <error-message-key> : <error-message-args>* }

Table 1. Rule Syntax

Rule Value	Description	Required
property-name	This is the default property of the bean being targeted for validation, and can be referred to with the shortcut `?` in an expression.	`true`
expression	The Valang expression.	`true`
default-error-message	The default error message. If this isn't needed, it can be left blank even though it's required.	`true`
error-message-key	The message resource key for the i18n error message.	`false`
error-message-arg	If the *error-message-key* is specified, arguments for the error message can also be set as the final value of the rule. This accepts a comma delimited list of values.	`false`

Expression Syntax

The expression language provides an English like syntax for expressing validation rules. There are a number of operators for comparing a value to another. Logical expressions, resulting in `true` or `false`, can be grouped together with parentheses to form more complex expressions.

Just to give some context to the explanation of all the rules, below is a simple example. The bean being validated has the properties `getFirstName()`, `getLastName()`, and `getAge()`. The first two return a `String` and the last returns an `int`. The default property is 'firstName', which is referred to by the question mark. The first part of the rule enclosed in parentheses checks if the first name is either 'Joe' or it's length is greater than 5. The next part checks if the last name is one of the values in the list, and the final part checks if the age is over 18.

```
(? EQUALS 'Joe' OR length(?) > 5) AND lastName IN 'Johnson', 'Jones', 'Smith'
AND age > 18
```

Operator Syntax

The parser is not case sensitive when processing the operators.

Table 2. Expression Operators

Comparison Operator	Description	Supports	Example
= | == | IS | EQUALS	Checks for equality.	Strings, booleans, numbers, dates, and enums.	`firstName EQUALS 'Joe'`

Comparison Operator	Description	Supports	Example
!= | <> | >< | IS NOT | NOT EQUALS	Checks for inequality.	Strings, booleans, numbers, dates, and enums.	`firstName NOT EQUALS 'Joe'`
> | GREATER THAN | IS GREATER THAN	Checks if a value is greater than another.	Numbers and dates.	`age > 18`
< | LESS THAN | IS LESS THAN	Checks if a value is less than another.	Numbers and dates.	`age > 18`
>= | => | GREATER THAN OR EQUALS | IS GREATER THAN OR EQUALS	Checks if a value is greater than or equal to another.	Numbers and dates.	`age >= 18`
<= | =< | LESS THAN OR EQUALS | IS LESS THAN OR EQUALS	Checks if a value is less than or equal to another.	Numbers and dates.	`age <= 18`
NULL | IS NULL	Checks if a value is `null`.	Objects.	`firstName IS NULL`
NOT NULL | IS NOT NULL	Checks if a value is not `null`.	Objects.	`firstName IS NOT NULL`
HAS TEXT	Checks if the value has at least one non-whitespace character.	Strings.	`firstName HAS TEXT`
HAS NO TEXT	Checks if the value doesn't have a non-whitespace character.	Strings.	`firstName HAS NO TEXT`
HAS LENGTH	Checks if the value's length is greater than zero.	Strings.	`firstName HAS LENGTH`
HAS NO LENGTH	Checks if the value's length is zero.	Strings.	`firstName HAS NO LENGTH`
IS BLANK	Checks if the value is blank (`null` or zero length).	Strings.	`firstName IS BLANK`
IS NOT BLANK	Checks if the value isn't blank (not `null`, length greater than zero).	Strings.	`firstName IS NOT BLANK`
IS UPPERCASE | IS UPPER CASE | IS UPPER	Checks if the value is uppercase.	Strings.	`firstName IS UPPERCASE`
IS NOT UPPERCASE | IS NOT UPPER CASE | IS NOT UPPER	Checks if the value isn't uppercase.	Strings.	`firstName IS NOT UPPERCASE`
IS LOWERCASE | IS	Checks if the value is	Strings.	`firstName IS`

Comparison Operator	Description	Supports	Example
LOWER CASE | IS LOWER	lowercase.		`LOWERCASE`
IS NOT LOWERCASE | IS NOT LOWER CASE | IS NOT LOWER	Checks if the value isn't lowercase.	Strings.	`firstName IS NOT LOWERCASE`
IS WORD	Checks if the value has one or more letters or numbers (no spaces or special characters).	Strings.	`firstName IS WORD`
IS NOT WORD	Checks if the value doesn't have one or more letters or numbers (no spaces or special characters).	Strings.	`firstName IS NOT WORD`
BETWEEN	Checks if a value is between two other values.	Numbers and dates.	`age BETWEEN 18 AND 65`
NOT BETWEEN	Checks if a value isn't between two other values.	Numbers and dates.	`age NOT BETWEEN 18 AND 65`
IN	Checks if a value is in a list.	Strings, booleans, numbers, dates, and enums.	`firstName IN 'Joe', 'Jack', 'Jane', 'Jill'`
NOT IN	Checks if a value isn't in a list.	Strings, booleans, numbers, dates, and enums.	`firstName NOT IN 'Joe', 'Jack', 'Jane', 'Jill'`
NOT	Checks for the opposite of the following expression.	Any expression.	`NOT firstName EQUALS 'Joe'`
!	Changes a boolean expression to it's opposite.	Booleans	`matches('\\s+', firstName) IS !(TRUE)`
AND	Used to join together the logical comparisons on either side of the operator. Both must evaluate to `true`.	Any expression.	`firstName EQUALS 'Joe' AND age > 21`
OR	Used to join together the logical comparisons on either side of the operator. Only one must evaluate to `true`.	Any expression.	`firstName EQUALS 'Joe' OR age > 21`
WHERE	If the where expression is true, then the main	Any expression.	`firstName EQUALS 'Joe' WHERE age >`

Comparison Operator	Description	Supports	Example
	expression for validation is performed. Otherwise it isn't evaluated and no errors are generated.		`21`
this	A reference to the bean passed in for validation, which could be passed into a custom function for example.	Any expression.	`isValid(this) IS TRUE`

Literal Syntax

Table 3. Literals

Literal Type	Description	Example
String	`String` literals are surrounded by single quotes.	'Joe'
Numbers	Numbers can be expressed without any special syntax. Numbers are all parsed using `BigDecimal`.	1, 100, 0.73, -2.48
Dates	Date literals are surrounded by brackets. These are the supported formats supported by the `DefaultDateParser`. *yyyyMMdd*, *yyyy-MM-dd*, *yyyy-MM-dd HH:mm:ss*, *yyyyMMdd HHmmss*, *yyyyMMdd HH:mm:ss*, *yyyy-MM-dd HHmmss*	[20081230], [2008-12-30], [2008-12-30 12:20:31]
Booleans	There are four different constants for boolean values. The values 'TRUE' and 'YES' represent `true`, and the values 'FALSE' and 'NO' represent `false`	TRUE, YES, FALSE, NO
Enums	Enums are surrounded by bracket and single quotes. If the full path to the enum isn't specified, it will be resolved when the expression is evaluated by looking up the enum value from enum on the opposite side of the expression.	['FAIL'], ['org.springmodules.validation.valang.CreditStatus.FAI ['org.springmodules.validation.valang.Person$CreditRat

Mathematical Operator Syntax

Valang supports basic mathematical formulas based on numeric literals and property values.

Table 4. Mathematical Expression Operators

Mathematical Operator	Description	Example
+	Addition operator.	`price + 12`
-	Subtraction operator.	`price - 12`
*	Multiplication operator.	`price * 1.2`
/ | DIV	Division operator.	`price / 2`
% | MOD	Modulo operator.	`age % 10`

Property Syntax

Valang supports standard property and nested property access to the bean passed in for validation.

Table 5. Property Syntax

Property Type	Description	Example
Standard	Using standard JavaBean property notation, a value from the bean being validated may be retrieved. The `address` represents `getAddress()` on the bean.	`address IS NOT NULL`
Nested	Using standard JavaBean property notation, a nested value from the bean being validated may be retrieved. The `address.city` represents `getAddress().getCity()` on the bean.	`address.city IS NOT BLANK`
List	From an array, `List`, or `Set`, a value from it can be returned by specifying it's index. Only arrays and lists are supported by bytecode generation.	`addresses[1] IS NOT NULL`
Map	From a `Map`, the value based on the key specified is retrieved.	`addresses[home] IS NOT NULL`

Functions

These are built in functions that come with Valang. The function framework is pluggable, so it's easy to add custom functions. Adding custom functions will be covered in the next section.

Table 6. Functions

Function	Description	Example
length | len | size | count	Returns the size of a collection or an array, and otherwise returns the length of string by called `toString()` on the object.	`length(firstName) < 20`
match | matches	Performs a match on a regular expression. The first argument is the regular expression and the second is the value match on.	`matches('\\w+', firstName) IS TRUE`
email	Checks if the value is a valid e-mail address.	`email(email) IS TRUE`
upper	Converts the value to uppercase.	`upper(firstName) EQUALS 'JOE'`
lower	Converts the value to lowercase.	`lower(firstName) EQUALS 'joe'`
resolve	Wraps a string in `DefaultMessageSourceResolvable`	`resolve('personForm.firstName') EQUALS 'First Name'`
inRole	Checks if the user authenticated by Spring Security is in a role.	`inRole('ADMIN') IS TRUE`

Custom Functions

Custom functions can either be explicitly registered or instances of `FunctionDefinition` and `FunctionWrapper` are automatically registered with a `ValangValidator`. If just specifying a class name, it must have a constructor with the signature `Function[] arguments, int line, int column`. The `FunctionWrapper` is specifically for Spring configured functions. If the `Function` in a `FunctionWrapper` takes any arguments, it must implement `ConfigurableFunction` which allows the parser to configure the arguments, line number, and column number. Otherwise the line & column number will not be set on a Spring configured function.

Note

It's important for a `FunctionWrapper` around a custom `Function` to be of the scope prototype as well as the `FunctionWrapper`. Also the `FunctionWrapper` must have *<aop:scoped-proxy/>* defined so each call to it will get a new instance of the function. This is because as the validation language is parsed a new instance of a function is made each time and has the arguments specific to that function set on it.

Spring Configuration

The example below shows how to explicitly register a custom function directly with a validator. The custom functions 'validLastName' and 'creditApproval' are registered on the `customFunctions` property as a `Map`. The key is the name of the function to be used in the validation language and the value if the function being registered, which can either be the fully qualified name of the class or an instance of `FunctionWrapper`.

ValangValidatorCustomFunctionTest-context.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="creditApprovalFunction"
          class="org.springmodules.validation.valang.CreditApprovalFunction"
          scope="prototype">
        <property name="creditRatingList">
            <list>
                <value>GOOD</value>
                <value>EXCELLENT</value>
            </list>
        </property>
    </bean>

    <bean id="personValidator" class="org.springmodules.validation.valang.ValangValidator">
        <property name="className" value="org.springmodules.validation.valang.Person"/>
        <property name="customFunctions">
            <map>
                <entry key="validLastName">
                    <value>org.springmodules.validation.valang.ValidLastNameFunction</value>
                </entry>
                <entry key="creditApproval">
                    <bean class="org.springmodules.validation.valang.functions.FunctionWrapper"
                          scope="prototype">
                        <aop:scoped-proxy/>

                        <property name="function" ref="creditApprovalFunction" />
                    </bean>
                </entry>
            </map>
        </property>
        <!--
            Final validation tests that the aop:scoped-proxy is working since if the same instance
            of CreditApprovalFunction is used it will be set to a failing value for both sides of
the or.
            While if two instances are made the first condition should pass while the second will
fail.
        -->
        <property name="valang">
            <value><![CDATA[
              { lastName : validLastName(?) is true : '' }
              { lastName : creditApproval(age, creditRating) is true : '' }
              { lastName : validLastName(?) is true AND creditApproval(age, creditRating) is true :
'' }
              { lastName : validLastName(?) is true AND
                           (creditApproval(age, creditRating) is true OR
                            creditApproval(age,
['org.springmodules.validation.valang.Person$CreditRating.FAIR']) is true) : '' }
            ]]<</value>
        </property>
    </bean>
```

```
</beans>
```

Instances of `FunctionDefinition` and `FunctionWrapper` are automatically registered with a `ValangValidator` The custom functions 'validLastName' and 'creditApproval' are registered. If a `FunctionWrapper` doesn't have a function name specified, the name of the bean will be used for the function name.

ValangValidatorCustomFunctionDiscoveryTest-context.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springmodules.validation.valang.functions.FunctionDefinition"
          p:name="validLastName"
          p:className="org.springmodules.validation.valang.ValidLastNameFunction"/>

    <!-- Uses bean name for function name if not explicitly set on the wrapper -->
    <bean id="creditApproval"
          class="org.springmodules.validation.valang.functions.FunctionWrapper"
          scope="prototype">
        <aop:scoped-proxy/>

        <property name="function">
            <bean id="creditApprovalFunction"
                  class="org.springmodules.validation.valang.CreditApprovalFunction"
                  scope="prototype">
                <property name="creditRatingList">
                    <list>
                        <value>GOOD</value>
                        <value>EXCELLENT</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="personValidator" class="org.springmodules.validation.valang.ValangValidator">
        <property name="className" value="org.springmodules.validation.valang.Person"/>
        <!--
            Final validation tests that the aop:scoped-proxy is working since if the same instance
            of CreditApprovalFunction is used it will be set to a failing value for both sides of
the or.
            While if two instances are made the first condition should pass while the second will
fail.
        -->
        <property name="valang">
            <value><![CDATA[
              { lastName : validLastName(?) is true : '' }
              { lastName : creditApproval(age, creditRating) is true : '' }
              { lastName : validLastName(?) is true AND creditApproval(age, creditRating) is true :
'' }
              { lastName : validLastName(?) is true AND
                           (creditApproval(age, creditRating) is true OR
                            creditApproval(age,
```

```
['org.springmodules.validation.valang.Person$CreditRating.FAIR']) is true) : '' }
            ]]<</value>
        </property>
    </bean>

</beans>
```

Code Example

Checks if the last name is in a list, and if it isn't `false` is returned.

```
public class ValidLastNameFunction extends AbstractFunction {

    final Logger logger = LoggerFactory.getLogger(ValidLastNameFunction.class);

    final Set<String> lValidLastNames = new HashSet<String>();

    /**
     * Constructor
     */
    public ValidLastNameFunction(Function[] arguments, int line, int column) {
        super(arguments, line, column);
        definedExactNumberOfArguments(1);

        lValidLastNames.add("Anderson");
        lValidLastNames.add("Jackson");
        lValidLastNames.add("Johnson");
        lValidLastNames.add("Jones");
        lValidLastNames.add("Smith");
    }

    /**
     * Checks if the last name is blocked.
     *
     * @return      Object      Returns a <code>boolean</code> for
     *                          whether or not the last name is blocked.
     */
    @Override
    protected Object doGetResult(Object target) {
        boolean result = true;

        String symbol = getArguments()[0].getResult(target).toString();

        if (!lValidLastNames.contains(symbol)) {
            result = false;
        }

        return result;
    }

}
```

Example 1 `ValidLastNameFunction`

The function checks if a person can get credit approval. Their credit rating is checked against a list only if they are over 18 years old.

```
public class CreditApprovalFunction extends AbstractFunction
        implements ConfigurableFunction {

    final Logger logger = LoggerFactory.getLogger(CreditApprovalFunction.class);

    Set<Person.CreditRating> lCreditRatings = new HashSet<Person.CreditRating>();

    /**
     * Constructor
     */
    public CreditApprovalFunction() {}

    /**
     * Constructor
     */
    public CreditApprovalFunction(Function[] arguments, int line, int column) {
        super(arguments, line, column);
        definedExactNumberOfArguments(2);

        lCreditRatings.add(Person.CreditRating.FAIR);
        lCreditRatings.add(Person.CreditRating.GOOD);
        lCreditRatings.add(Person.CreditRating.EXCELLENT);
    }

    /**
     * Gets number of expected arguments.
     * Implementation of <code>ConfigurableFunction</code>.
     */
    public int getExpectedNumberOfArguments() {
        return 2;
    }

    /**
     * Sets arguments, line number, and column number.
     * Implementation of <code>ConfigurableFunction</code>.
     */
    public void setArguments(int expectedNumberOfArguments, Function[] arguments,
                             int line, int column) {
        // important to set template first or can cause a NullPointerException
        // if number of arguments don't match the expected number since
        // the template is used to create the exception
        super.setTemplate(line, column);
        super.setArguments(arguments);
        super.definedExactNumberOfArguments(expectedNumberOfArguments);
    }

    /**
     * Sets valid credit rating approval list.
     */
    public void setCreditRatingList(Set<Person.CreditRating> lCreditRatings) {
        this.lCreditRatings = lCreditRatings;
    }

    /**
     * If age is over 18, check if the person has good credit,
     * and otherwise reject.
     *
     * @return      Object      Returns a <code>boolean</code> for
     *                          whether or not the person has good enough
     *                          credit to get approval.
     */
    @Override
    protected Object doGetResult(Object target) {
        boolean result = true;

        int age = (Integer) getArguments()[0].getResult(target);
        Person.CreditRating creditRating = (Person.CreditRating)getArguments()[1].getResult(target);

        // must be over 18 to get credit approval
```

```
        if (age > 18) {
            if (!lCreditRatings.contains(creditRating)) {
                result = false;
            }
        }

        return result;
    }

}
```

Example 2 `ConfigurableFunction`

Bytecode Generation

If the validator will only be used to validate a specific class, the property 'className' can be specified to avoid reflection. If it's set, a custom `Function` will be generated that directly retrieves a property to avoid reflection. This provides a significant performance improvement if that is a concern, which typically isn't if the validation is being used to validate a web page since the delay is so small either way.

Note

Only a `Map`, a `List`, or an Array is supported by bytecode generation, not a `Set`. Primitive arrays currently aren't supported, but any object one is. Also, nested properties are currently not supported.

This is a small excerpt from the logging of the performance unit test. As you can see from the logging, as the validator is initialized it generates bytecode and shows for which class and method, as well as what the generated class name is. The package and name of the original class is used and then has what property is being retrieved appended along with 'BeanPropertyFunction$$Valang' to make a unique class name to try to avoid any collisions.

```
DefaultVisitor - Generated bytecode for org.springmodules.validation.valang.Person.getLastName()
    as 'org.springmodules.validation.valang.PersonLastNameBeanPropertyFunction$$Valang'.
DefaultVisitor - Generated bytecode for org.springmodules.validation.valang.Person.getAge()
    as 'org.springmodules.validation.valang.PersonAgeBeanPropertyFunction$$Valang'.
DefaultVisitor - Generated bytecode for org.springmodules.validation.valang.Person.getCreditRating()
    as 'org.springmodules.validation.valang.PersonCreditRatingBeanPropertyFunction$$Valang'.
DefaultVisitor - Generated bytecode for org.springmodules.validation.valang.Person.getFirstName()
    as 'org.springmodules.validation.valang.PersonFirstNameBeanPropertyFunction$$Valang'.
DefaultVisitor - Generated bytecode for org.springmodules.validation.valang.Person.getCreditStatus()
    as 'org.springmodules.validation.valang.PersonCreditStatusBeanPropertyFunction$$Valang'.
ValangValidatorPerformanceTest - Took 7098.0ns.
ValangValidatorPerformanceTest - Took 2124.0ns.
ValangValidatorPerformanceTest - Message validator took 7098.0ns, and bytecode message valdiator
took 2124.0ns.
```

Results from `ValangValidatorPerformanceTest` which was run on a Macbook Pro (2.3GHz Intel Core i7 with 8 GB RAM with OS X 10.6.8) with Java 6. All the expressions are identical, but adjusted to either retrieve the values being compared from a JavaBean, `Map`, `List`, or an array.

Table 7. Bytecode Generation Performance Comparison

Expression	Reflection	Bytcode Generation
{ lastName : validLastName(?) is true AND creditApproval(age, creditRating) is true WHERE firstName IN 'Joe', 'Jack', 'Jill', 'Jane' AND creditStatus IN ['org.springmodules.validation.valang.CreditStatus.PENDING'], ['org.springmodules.validation.valang.CreditStatus.FAIL'] AND creditRating EQUALS ['org.springmodules.validation.valang.Person$CreditRating.EXCELLENT'] AND age > 18 : " }	1176ns	327ns
{ mapVars[lastName] : validLastName(?) is true AND creditApproval(mapVars[age], mapVars[creditRating]) is true WHERE mapVars[firstName] IN 'Joe', 'Jack', 'Jill', 'Jane' AND mapVars[creditStatus] IN ['org.springmodules.validation.valang.CreditStatus.PENDING'], ['org.springmodules.validation.valang.CreditStatus.FAIL'] AND mapVars[creditRating] EQUALS ['org.springmodules.validation.valang.Person$CreditRating.EXCELLENT'] AND mapVars[age] > 18 : " }	905ns	48ns
{ listVars[1] : validLastName(?) is true AND creditApproval(listVars[2], listVars[4]) is true WHERE listVars[0] IN 'Joe', 'Jack', 'Jill', 'Jane' AND listVars[3] IN ['org.springmodules.validation.valang.CreditStatus.PENDING'], ['org.springmodules.validation.valang.CreditStatus.FAIL'] AND listVars[4] EQUALS ['org.springmodules.validation.valang.Person$CreditRating.EXCELLENT'] AND listVars[2] > 18 : " }	575ns	43ns
{ vars[1] : validLastName(?) is true AND creditApproval(vars[2], vars[4]) is true WHERE vars[0] IN 'Joe', 'Jack', 'Jill', 'Jane' AND vars[3] IN ['org.springmodules.validation.valang.CreditStatus.PENDING'], ['org.springmodules.validation.valang.CreditStatus.FAIL'] AND vars[4] EQUALS ['org.springmodules.validation.valang.Person$CreditRating.EXCELLENT'] AND vars[2] > 18 : " }	563ns	40ns

Spring Configuration

By specifying the 'className' property, bytecode will be generated for each method being called to avoid reflection. This gives a significant performance improvement.

Excerpt from *ValangValidatorCustomFunctionTest-context.xml*

```
<!--
    Only perform validation if valid first name, credit status is failed or pending,
    and the credit rating is excellent where the person's age is over 18.
-->
<bean id="expression" class="java.lang.String">
    <constructor-arg>
        <value><![CDATA[
          { lastName : validLastName(?) is true AND creditApproval(age, creditRating) is true
                       WHERE firstName IN 'Joe', 'Jack', 'Jill', 'Jane' AND
                             creditStatus IN
['org.springmodules.validation.valang.CreditStatus.PENDING'],
['org.springmodules.validation.valang.CreditStatus.FAIL'] AND
                             creditRating EQUALS
['org.springmodules.validation.valang.Person$CreditRating.EXCELLENT'] AND
                             age > 18 : '' }
        ]]<</value>
    </constructor-arg>
</bean>

...

<bean id="bytecodePersonValidator" class="org.springmodules.validation.valang.ValangValidator">
    <property name="className" value="org.springmodules.validation.valang.Person"/>
    <property name="valang" ref="expression" />
</bean>
```

Date Examples

The default date parser provides support for a number of different date literals, and also has support for shifting and manipulating dates. Below are a few examples, but see the `DefaultDateParser` for more detailed information.

Spring Configuration

ValangValidatorDateTest-context.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="personValidator" class="org.springmodules.validation.valang.ValangValidator">
        <property name="className" value="org.springmodules.validation.valang.Person"/>
        <property name="valang">
```

```
            <!--
                Third to last validation shifts '2008-12-30<y' to '2008-01-01 00:00:00'
                Second to last validation shifts '2005-04-09 23:30:00<M+10d+8H' to '2005-04-11
08:00:00'.
                Last shifts '2009-02-06 00:00:00<M+20y' to '2029-02-01 00:00:00'.
            -->
            <value><![CDATA[
              { lastUpdated : ? > [20081230] : '' }
              { lastUpdated : ? > [2008-12-30] : '' }
              { lastUpdated : ? > [2008-12-30 12:20:31] : '' }
              { lastUpdated : ? > [20081230 122031] : '' }
              { lastUpdated : ? > [20081230 12:20:31] : '' }
              { lastUpdated : ? > [2008-12-30 122031] : '' }

              { lastUpdated : ? BETWEEN [20081230] AND [2009-02-06 00:00:00<M+20y] : '' }

              { lastUpdated : ? > [2008-12-30<y] : '' }
              { lastUpdated : ? > [2005-04-09 23:30:00<M+10d+8H] : '' }
              { lastUpdated : ? < [2009-02-06 00:00:00<M+20y] : '' }
            ]]</value>
        </property>
    </bean>

</beans>
```

2. Download

If you are using Maven, you can use the Maven dependency below and add a repository definition for the Spring by Example Maven repository. Otherwise you can download the jar directly from the Spring by Example Maven repository (http://www.springbyexample.org/maven/repo/).

```
<dependency>
        <groupId>org.springbyexample</groupId>
        <artifactId>sbe-validation</artifactId>
        <version>0.95</version>
</dependency>

<repositories>
    <repository>
        <id>springbyexample.org</id>
        <name>Spring by Example</name>
        <url>http://www.springbyexample.org/maven/repo</url>
    </repository>
</repositories>
```

3. Reference

Related Links

- Spring by Example Validation Site [http://springbyexample.org/maven/site/sbe-validation/0.95/sbe-validation/]

- Spring 3.1.x Validation Documentation [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/validation.html]

Project Setup

Follow the Project Checkout instructions for git, then go to the project (relative path below).

```
$ cd modules/sbe-validation
```

General Setup Instructions

General instructions for checking out the project with Eclipse and building with Maven.

Example Project Setup

Project Information

- Spring Framework 3.1.x

Appendix A. Setup

A.1. Project Setup

The latest Spring by Example repository is available at https://github.com/spring-by-example/spring-by-example.

Older projects can be checked out directly from the Subversion repository located at http://svn.springbyexample.org/.

The projects were setup to be imported as Maven projects into Eclipse, but because of some issues some Eclipse files are checked for reference. Maven can also be used to generate IntelliJ project files.

```
$ mvn idea:idea
```

Basic Setup

1. Java 6 or higher installed with JAVA_HOME set.
2. Download and install to use Maven from the command line. Maven [http://maven.apache.org/] (version 3.x or higher).
3. Install Git [http://git-scm.com/].

Example Environment Settings

These are the *~/.bash_profile* settings for a Mac.

```
JAVA_HOME=`/usr/libexec/java_home -v 1.6+`
MAVEN_OPTS="-Xms256m -Xmx1024m -XX:MaxPermSize=256M"

PATH=$PATH:JAVA_HOME/bin

export JAVA_HOME MAVEN_OPTS
```

Project Checkout

Checkout the project using Git [http://git-scm.com/] from Spring By Example Repository [https://github.com/spring-by-example/spring-by-example]. The first command will checkout all of the current Spring by Example projects, and the second will checkout the specific tag that corresponds to this document.

```
$ git clone git://github.com/spring-by-example/spring-by-example.git
$ cd spring-by-example
$ git checkout sbe-1.2.2
```

SpringSource Tool Suite Setup

Download and install the SpringSource Tool Suite Download [http://www.springsource.com/products/sts]. It's free and simplifies the IDE setup. The SpringSource Tool Suite [http://www.springsource.com/products/sts] is based on the Eclipse IDE [http://www.eclipse.org/], but has all the SpringSource [http://www.springsource.com/] plugins installed as well as others like for AspectJ [http://www.eclipse.org/aspectj/] and Maven [http://maven.apache.org/].

- SpringSource Tool Suite Download [http://www.springsource.com/products/sts]

General Eclipse IDE Setup

Project Checkout

1. Navigate to the projects trunk using the Subversion repository browser.
2. Right click on 'trunk', select 'Checkout...', click on the 'Finish' button.

Web Setup

1. Download and install a compliant Web Application Server (like Tomcat [http://tomcat.apache.org/] or the VMware vFabric tc Server [http://www.vmware.com/products/application-platform/vfabric-tcserver/overview.html]).
2. Show Sever View
   - Window/Show View/Others...
   - Server/Servers
3. Right click in the Server View and select 'New/Server'.
   - Choose the installed server (for example Tomcat [http://tomcat.apache.org/] if that is the server you chose to install).
4. Right click on project and select 'Run As'/'Run on Server'.
   - Choose the server to run the project on.
   - Click on Next.
   - Click on Finish.
   - This should launch the application and also open up a browser window in Eclipse pointing to the webapps home page.

Appendix B. Author Bios

B.1. David Winterfeldt

Introduction

David has been doing software development for over 20 years. He's been using Java since 1998 and involved in using Open Source almost as long. David has focused on Web and Enterprise development for most of his career, and started working with the Spring Framework in 2006. He started working with Spring 2.0 towards the end of 2006 and really enjoy working with the Spring Framework. He really enjoys it because it not only saves time, but encourages better design and code reuse through loosely coupled components.

David started Spring by Example [http://www.springbyexample.org/] to post different examples he had been doing, and also to become involved again with Open Source projects. Spring by Example [http://www.springbyexample.org/] is a general resource for Spring and should ultimately save developers time.

Currently David works at VMware on the VMware vFabric Application Director [http://www.vmware.com/products/application-platform/vfabric-application-director/overview.html] project. It enables developers and organizations to deploy applications to the cloud by having a logical abstraction for software services and application topologies. This allows an application to be easily deployed multiple times to different environments.

David is also a committer on Struts [http://struts.apache.org/] and Commons Validator [http://commons.apache.org/validator/], but is no longer active on either. He was also the creator of Commons Validator [http://commons.apache.org/validator/].

Technical Expertise

- Publications
  - Co-author of Spring In-depth, In Context [http://www.springindepth.com/], which is a free, but incomplete (stopped due to time constraints), online book for the Spring Framework.
  - Co-author of the validation chapter in Struts in Action: Building Web Applications with the Leading Java Framework, Manning Publications (November 2002) [http://www.manning.com/husted/].
- Conference Speaker
  - Speaker at SpringOne 2GX 2010 [http://www.springone2gx.com/] presenting Killer Flex RIAs with Spring ActionScript [http://www.springone2gx.com/conference/chicago/2010/10/session?id=19344].
  - Speaker at SpringOne Americas 2008 [http://www.springone2gx.com/] presenting Case Study: GWT & Comet Integration with the Spring Framework at NYSE [http://www.springone2gx.com/m/mobile/presentation.jsp?showId=172&presentationId=12842].
- Open Source Contributions
  - Original developer of Commons Validator and its integration into Struts.
  - Committer on Struts (http://struts.apache.org) and the Commons Validator (http://jakarta.apache.org/commons).
  - Webmaster of Spring by Example (http://www.springbyexample.org [http://jakarta.apache.org/commons]).
- Architecture & Design
  - Struts Validator & Commons Validator with matching client side JavaScript validation
  - Scalable E-mail Framework with scheduling, retry process, template processing, and dynamic population processing
  - Keyword based E-mail Alert System for online articles
  - Scalable Publishing System with configurable publishing processes
  - Multi-threaded, multi-tier servers with custom thread pools and tcp/ip socket connections for a client entry prototype system.
- Programming Languages
  - Java 1.0/1.1.x/1.2.x/1.3.x/1.4.x/1.5/1.6, Perl, JavaScript, SQL, PL/SQL
- Java EE
  - JDBC, RMI, JavaMail, JMS, EJBs (Stateless, Session, Message Driven, Entity), RMI, JSPs, Servlets, Web Services, XML, JTA
- Application Servers
  - Jetty 6.x, Tomcat 3.x/4.x/5.0/5.5/6.0, Spring dm Server, JBoss, BEA Weblogic Application Server

4.0/4.5/5.1/7.0

- Databases
  - Oracle 8i/9i, DB2, MS SQL*Server 6.0/6.5/7.0, MySQL, PostgreSQL, MS Access, FoxPro
- Open Source Frameworks
  - Spring Framework, Spring Security, Spring Web Flow, Spring Web Services, Struts, GWT, Hibernate, Velocity Template Engine, Axis, OSGi
- Open Source Projects
  - Ant, Maven, AspectJ, Jakarta Commons BeanUtils, Jakarta Commons Betwixt, Jakarta Commons Collections, Jakarta Commons Digester, Jakarta Commons IO, Jakarta Commons Lang, Jakarta Commons Validator, Jakarta ORO, Jakarta POI, Jakarta Regexp, Tiles, Log4J, LOGBack, Xalan, Xerces, JUnit, HttpUnit, Cactus, XDoclet
- Miscellaneous
  - Windows 2000 Server, Windows XP, Mac OS X, Solaris, Linux, Eclipse IDE

Experience

- Lead Architect: Designing systems/frameworks, preparing projects and frameworks for developers on new projects, providing instruction and assistance to developers.
- Ability to interact with Business Analysts, and Clients to gather requirements and implement.
- Performance Enhancements: database performance, adding multi-threading, optimizing code.
- Interaction with clients to define specifications, gather requirements, and communicate & resolve issues.
- Documentation: Javadocs, Wiki based documentation for developers, technical specifications.
- Interpreting specifications and requirement documents into working systems.
- Ability to learn new technologies, investigate issues, resolve development issues.
- Working with distributed team environments.

Sites & Blogs

- Spring by Example - http://www.springbyexample.org/
- Blog - http://davidwinterfeldt.blogspot.com/
- Spring by Example Blog - http://springbyexample.blogspot.com/
- Twitter - http://twitter.com/dwinterfeldt

Contact Info

E-mail: <dwinterfeldt@springbyexample.org>