Integrating Spring with the Collaborative Infrastructure

Author: Ron van Hoof
Version: 1.0 12/16/2009

Table of Contents

1 Introduction 1
2 Spring Application Framework 1
3 Collaborative Infrastructure and Spring Integration 3
3.1 CI ASP Hooks 3
3.2 CI Spring Interface Hook 4
3.3 Springifying CI Actors 6
4 References 9

1 Introduction

The Collaborative Infrastructure provides a communication infrastructure (CI) to allow components (actors) to easily collaborate with one another as part of a distributed system in a structured agent-oriented fashion. To allow an actor developer to focus the development of a CI-based enterprise application on its business logic it is beneficial to integrate the collaborative infrastructure with an application framework that manages most of the plumbing for interacting with enterprise technologies such as relations between objects, database interactions, transactions, security. This article describes how the Spring application framework can be integrated with the Collaborative Infrastructure to simplify the development of enterprise applications using the CI.

2 Spring Application Framework

Spring is an open source application framework for the Java platform. It is a technology dedicated to enabling you to build applications using Plain Old Java Objects (POJOs). Spring's main aim is to make J2EE easier to use and promote good programming practice. It does this by enabling a POJO-based programming model that is applicable in a wide range of environments.

Central to the Spring framework is its Inversion of Control (IoC) container, which provides a consistent means of configuring and managing Java objects using callbacks. The container is responsible for managing object lifecycles: creating objects, calling initialization methods, and configuring objects by wiring them together.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

Objects created by the container are also called Managed Objects or Beans. Typically, the container is configured by loading XML files containing *Bean definitions* which provide the information required to create the beans.

Objects can be obtained by means of *Dependency lookup* or *Dependency injection*. Dependency lookup is a pattern where a caller asks the container object for an object with a specific name or of a specific type. Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods.

The Spring application framework comprises several modules that provide a range of services:

- Inversion of Control container: configuration of application components and lifecycle management of Java objects
- Aspect-Oriented Programming (AOP): enables implementation of cross-cutting routines
- Data Access: working with relational database management systems on the Java platform using JDBC and object-relational mapping (ORM) tools
- Transaction Management: unifies several transaction management APIs and coordinates transactions for Java objects
- Model-View-Controller (MVC): an HTTP and Servlet-based framework providing hooks for extension and customization
- Authentication and Authorization: configurable security processes that support a range of standards, protocols, tools, and practices via the Spring Security sub-project (formerly Acegi)
- Testing: support classes for writing unit tests and integration tests

The Spring application framework has become a popular framework amongst developers for developing enterprise applications for several reasons:

1. It addresses important areas that many other popular frameworks don't. Spring focuses around providing a way to manage your business objects.
2. Spring is both comprehensive and modular. Spring has a layered architecture, meaning that you can choose to use just about any part of it in isolation, yet its architecture is internally consistent. You get maximum value from your learning curve. You might choose to use Spring only to simplify use of JDBC, for example, or you might choose to use Spring to manage all your business objects.
3. Spring is designed from the ground up to help you write code that is easy to test. Spring is an ideal framework for test driven projects.
4. Spring is an increasingly important integration technology, its role recognized by several large vendors.

Some of the architectural benefits of Spring:

- Spring can effectively organize the middle-tier objects, whether or not you choose to use EJB. Spring takes care of plumbing that would be left up to the developer if only Struts or other

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

frameworks geared to particular J2EE APIs were used. And while it is perhaps most valuable in the middle tier, Spring's configuration management services can be used in any architectural layer, in whatever runtime environment

- Spring can eliminate the proliferation of Singletons seen on many projects.
- Spring can eliminate the need to use a variety of custom properties file formats, by handling configuration in a consistent way throughout applications and projects. Ever wondered what magic property keys or system properties a particular class looks for, and had to read the Javadoc or even source code? With Spring you simply look at the class's JavaBean properties or constructor arguments. The use of **Inversion of Control** and **Dependency Injection** (discussed below) helps achieve this simplification.
- Spring can facilitate good programming practice by reducing the cost of programming to interfaces, rather than classes, almost to zero.
- Spring is designed so that applications built with it depend on as few of its APIs as possible. Most business objects in Spring applications have no dependency on Spring.
- Applications built using Spring are very easy to unit test.
- Spring can make the use of EJB an implementation choice, rather than the determinant of application architecture. You can choose to implement business interfaces as POJOs or local EJBs without affecting calling code.
- Spring helps you solve many problems without using EJB. Spring can provide an alternative to EJB that's appropriate for many applications. For example, Spring can use AOP to deliver declarative transaction management without using an EJB container; even without a JTA implementation, if you only need to work with a single database.
- Spring provides a consistent framework for data access, whether using JDBC or an O/R mapping product such as TopLink, Hibernate or a JDO implementation.
- Spring provides a consistent, simple programming model in many areas, making it an ideal architectural "glue." You can see this consistency in the Spring approach to JDBC, JMS, JavaMail, JNDI and many other important APIs.

3 Collaborative Infrastructure and Spring Integration

3.1 CI ASP Hooks

The CI offers two approaches to making actors available in its infrastructure. Actors can be loaded into an Actor Hosting Environment (AHE). The AHE uses an XML based descriptor to configure it and to configure the actors that need to be loaded into the AHE and managed by the AHE. The AHE is a wrapper for a CI Actor Service Provider (ASP). The ASP is the component that manages the creation of the CI services (directory, transport, data distribution, management, translation), registration of actors, and that makes the CI services available to the registered actors. The second approach is to embed a CI Actor Service Provider into an application. This application in turn would be responsible for loading and managing actors.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

The CI Actor Service Provider provides an API to hook in any external services or frameworks that need to be integrated. To develop such an ASP hook a component needs to be developed that implements the `IASPHook` interface. The hook(s) need to be registered with the ASP by listing them in the ASP's descriptor file.

```
<ASP_HOOKS>
  <ASP_HOOK>gov.nasa.ci.MyHook</ASP_HOOK>
</ASP_HOOKS>
```

A hook allows for the initialization prior to any services having been loaded, after services have been loaded, and allows for shutdown or cleanup for the hook either prior to services having been shut down or after the services have been shut down.

3.2 CI Spring Interface Hook

To integrate Spring into the Collaborative Infrastructure we developed a CI Spring Interface hook. This ASP hook creates and initializes a Spring Application Context container.

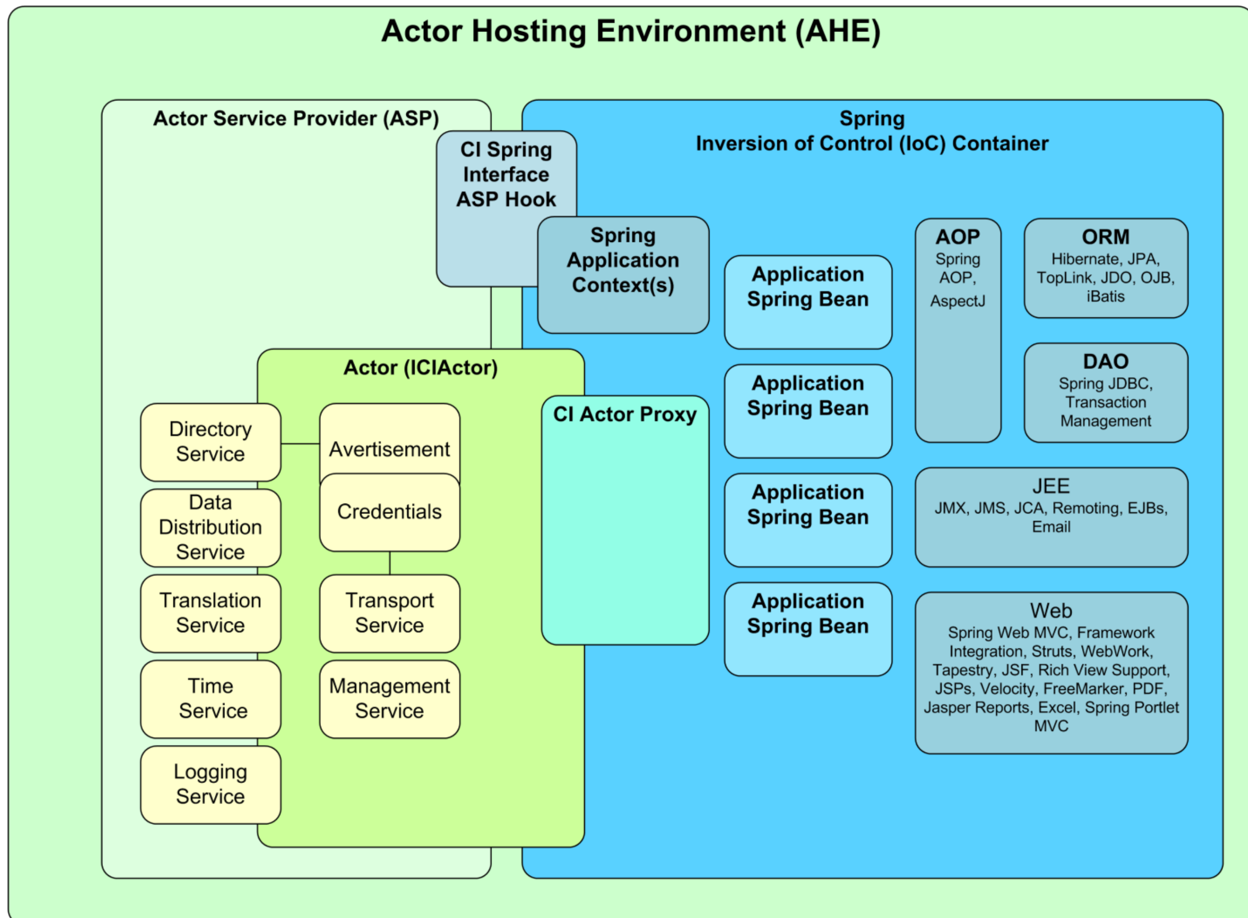

Figure 1: CI Spring Integration

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

This hook and other supporting integration classes are packaged in the cispringinterface.jar file. To ensure that this hook gets loaded by the ASP in an Actor Hosting Environment add the following hook in the ASP's descriptor:

```
<ASP_HOOKS>
  <ASP_HOOK>gov.nasa.ci.spring.CISpringInterface</ASP_HOOK>
</ASP_HOOKS>
```

This hook supports two configuration properties that can be declared in the `PROPERTIES` section in the ASP descriptor.

1. `ci.spring.selector` this property's value specifies the location of the resource(s) which will be read and combined to form the definition of the BeanFactoryLocator instance. It specifies a bean reference context that lists the application contexts that need to be loaded. By default the value for this property when not specified is `classpath*:beanRefContext.xml`. This value means that all beanRefContext.xml files anywhere in the classpath need to be loaded. The value must be a URL or a "claspath:" or "classpath*:" pseudo URL.
2. `ci.spring.context` this property's value is a comma-delimited list of the applications contexts that need to be initialized. The names listed are the names of the contexts listed in the file(s) defined for the ci.spring.selector. Initializing such a context causes for Spring to parse and process the application context files and create and initialize the listed Spring beans.

The contents of a beanRefContext.xml file would be similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<!-- load a hierarchy of contexts, although there is just one here -->
<beans>
  <bean id="application-context"

class="org.springframework.context.support.ClassPathXmlApplicationContext"
>
    <constructor-arg>
      <list>
        <value>/applicationContext.xml</value>
      </list>
    </constructor-arg>
  </bean>
</beans>
```

The id `application-context` is user defined and would be the name of the context to list for the `ci.spring.context` property to have it initialized. When this context is initialized by Spring it will locate, load, and process the file `applicationContext.xml`. A `ClassPathXmlApplicationContext` is configured to be used to locate the file so this file must be found in the classpath. Alternative application context loaders are:

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

FileSystemXmlApplicationContext and XmlWebApplicationContext. The XmlWebApplicationContext is only to be used when running in a servlet container such as Tomcat.

The applicationContext.xml file is the main configuration file for configuring a Spring application. It is the file in which all beans are declared and configured. You can name this file anything you like and list the appropriate filename in the beanRefContext.xml file. An example of such an applicationContext.xml file.

```
<?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:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <bean id="kfxParser"
      class="gov.nasa.arc.ci.spring.parser.KfxParser">
    <property name="kfxFileService" ref="kfxFileService" />
  </bean>

  <bean id="kfxFileService"
      class="gov.nasa.arc.ci.spring.common.KfxFileService">
    ...
  </bean>
</beans>
```

3.3 Springifying CI Actors

With the CI Spring Interface hook in place the Spring application framework is integrated and is able to load Spring beans. This however does not yet allow us to configure CI actors with Spring.

Beans configured in the Spring application context file are managed by the Spring container including its creation. CI Actors are non-spring beans managed by the CI Hosting Environment including their creation. It is therefore not possible to just declare a CI actor as a regular bean since it would cause the Spring container to try to instantiate the CI Actor when it processes the bean declaration.

To make a CI Actor configurable with Spring two things need to happen:

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

1. Declare the CI Actor as an abstract bean. Declaring a bean abstract indicates to Spring not to instantiate the bean.
2. Annotate the CI Actor with the `@Configurable("<bean id>")` annotation. By means of an AspectJ aspect Spring will configure the actor when the actor is instantiated by the Actor Hosting Environment.

Declaring a CI actor as an abstract bean does allow us to configure the actor, declaring properties and injecting references to other beans. However the problem with an abstract bean is that it cannot be injected into Spring beans, i.e. we cannot have Spring beans reference the actor and this is definitely something we want to be able to do to allow other components to access the CI services via the actor.

To solve this problem the following five activities need to be performed for each actor that needs to be springified in addition to the abstract bean declaration and annotating the CI actor:

1. Declare a Java interface for your actor extending the ICIActor interface (gov.nasa.ci.common.actor). Declare any actor specific business methods in this interface. This interface is the contract for your actor and specifies the methods available. The ICIActor interface already declares all of the CI specific methods.
2. Declare a Java proxy class for your actor that extends the CIActorProxy class and that implements the Java interface declared for your actor. Only your own actor specific business methods need to be implemented and they should delegate to the actual actor's implementation. The getActor() method returns a reference to the proxied actor. This actor reference is set via dependency injection in Spring.
3. Have your actor class extend the AbstractSpringCIActor class and implement the actor's interface. By extending the AbstractSpringCIActor class you enable for the dependency injection of the proxy into the actor. When this injection takes place the actor will set a reference to itself in the proxy. It won't keep a reference to the proxy.
4. Declare the proxy for the actor as a Spring bean in the bean application context file.
5. Configure the injection of the proxy into the actor.

The following example code summarizes all activities required to springify an actor.

1. Create the interface for the actor

```
package gov.nasa.ci.myactor;
import gov.nasa.ci.common.actor.ICIActor;
public interface IMyActor extends ICIActor {
      public void doSomething();
} // IMyActor
```

2. Create the actor and make it Spring configurable and proxy injectable

```
package gov.nasa.ci.myactor;
import gov.nasa.ci.spring.actor.AbstractSpringCIActor;
import org.springframework.beans.factory.annotation.Configurable;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
@Configurable("myActorImpl") // myActorImpl is the assigned bean id
public class MyActor
  extends AbstractSpringCIActor
  implements IMyActor {
      public void doSomething() {
           System.out.println("did something");
      } // doSomething
      ... all other required CI methods ...
} // MyActor
```

3. Create the proxy for the actor

```
package gov.nasa.ci.myactor;
import gov.nasa.ci.common.actor.CIActorProxy;
public class MyActorProxy extends CIActorProxy implements IMyActor {
      public void doSomething() {
           ((IMyActor)getActor()).doSomething();
      } // doSomething
} // MyActorProxy
```

4. Configure the actor in the Spring application context file

```
<!-- Declare the actual actor as an abstract bean -->
<bean id="myActorImpl"
  class="gov.nasa.ci.myactor.MyActor"
  abstract="true">
      <!-- inject the actor proxy into the actor to have the actor
           reference set in the proxy -->
      <property name="actorProxy" ref="myActor" />
</bean>

<!-- Declare the proxy for the actor. Use the bean id for this
     proxy in all beans that need a reference to the actor -->
<bean id="myActor"
  class="gov.nasa.ci.myactor.MyActorProxy">
</bean>
```

When the Actor Hosting Environment is started it will configure an Actor Service Provider using the ASP's descriptor. The Actor Service Provider sees that a hook is declared in its descriptor and instantiates the CISpringInterface hook. This hook loads the beanRefContext.xml file after all CI services have been created and initialized. The ci.spring.contexts specifies as value the application context that needs to be initialized that declares the Spring beans. Spring will load and initialize the application context. The only bean listed that can be initialized is the actor's proxy ("myActor"). This proxy is instantiated. The actual actor's bean is abstract and can therefore not be instantiated. The Spring application framework has now been initialized and configured as has the CI Actor Service Provider. The AHE now loads the

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

descriptors for the actors listed in its descriptor file. Only the MyActor.ciax file is listed for the MyActor actor. It loads the actor's descriptor file and retrieves the class name for that actor, then instantiates that actor. Because of the Configurable annotation Spring will immediately after instantiation of the Actor class initiate the configuration of that actor. The only configurable action to perform is to inject the proxy into the actor. Spring will invoke the setActorProxy method on the actor and pass it a reference to the actor proxy (myActor) it instantiated. The actor in invokes as a result of the injection the setActor method on the proxy to give it a reference to the actual actor so that the proxy can properly delegate all method invocations supported in the actor's contract (interface) to the actual actor. This completes the startup and makes the actor (or really its proxy) available as a Spring bean to other Spring beans.

4 References

Spring manual:
http://static.springsource.org/spring/docs/2.5.x/reference/introduction.html#introduction-overview

Introduction to the Spring Framework by Rod Johnson:
http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.