

Covers Seam 2

# SEAM IN ACTION

BONUS CHAPTER

Dan Allen

FOREWORD BY Norman Richards





***Seam in Action***  
by Dan Allen  
Bonus Chapter 15

Copyright 2009 Manning Publications

# 15

## *Spring integration*

---

### ***This chapter covers***

- Resolving Spring beans from Seam
- Building Spring–Seam hybrid components
- Letting Seam manage persistence for Spring
- Using Spring transactions in Seam

If you look inside the bag of any nonsponsored golfer, you're likely to find a wide variety of equipment brands. A golfer may feel that a Nike putter provides the best touch on the greens, that a TaylorMade driver helps stay out of the woods, and that Cleveland irons are the most forgiving of a bad swing. Although the reason behind the diversity is partially mental, no doubt, the dominant factor is that equipment companies specialize in different products. One company may not have the same passion for designing perfect putters as they do for designing distance drivers. As a result, one brand doesn't always suffice. The same can be said of a developer's toolbox.

To foster such diversity, Seam provides a module for integrating the Spring Framework into your Seam application—and trust me, this is more than just an academic exercise. You just read 14 chapters detailing features that Seam fits under a single, accessible component-based model, empowering you to create stateful enterprise applications with tremendous agility. Still, Spring offers up a plethora of

features that you won't find in Seam. It's likely that these features—such as lightweight remoting, advanced aspect-oriented programming (AOP) declarations, template classes, and emulated container resource injections, to mention a few—simply aren't the passion of Seam. This disparity isn't necessarily a shortcoming. You don't have to throw out Seam, and the integrations that come with it, just because it's missing a feature that Spring boasts. Instead, you should recognize that these differences translate into more tools for your toolbox. To that end, Seam fully sanctions the integration with Spring. Seam and Spring are certainly not mutually exclusive technologies, and you might even find that their combination offers more value than either one standing alone. In reality, though, the primary goal of the integration is to assist you in migrating your applications from Spring to Seam to take advantage of Seam's state management.

In this chapter, you'll learn how to leverage the Spring container from Seam and vice versa. The integration starts with a lightweight variable resolver bridge. You're then introduced to the Spring–Seam hybrid component, which is an object instance that benefits from functionality provided by both the Seam and Spring containers. You'll also learn how to infuse state into traditionally stateless Spring beans by allowing them to reside in Seam contexts. Finally, you'll learn how to integrate Seam and Spring at the most basic level by having them share persistence managers and transactions.

By the end of the chapter, you'll walk away an enlightened developer, no longer interested in the trite *Spring versus Seam* debates, but rather looking for more ways to extract value from the unmatched features of both frameworks. To you, it's all gravy!

I want to start by showing you how to integrate Seam and Spring from the API level, a concept I label *POJO integration*. Indeed, the best integration is no integration at all! After that, we get more serious and move on to learning how to get the two containers to play nice with each other and share resources.

## **15.1** *POJO integration*

Before we go diving into the technical challenges of getting the two containers talking to each other, let's first consider if it's even necessary to go down that road. By jumping straight into container integration, we're overlooking the most valuable feature each has to offer. All the hubbub about POJO frameworks stems from the fact that the POJOs themselves can be used in isolation. We need to practice what we preach and actually use them that way. Spring POJOs can be instantiated using the Java `new` operator and configured through their bean properties. The Spring container is *not* required to use them. They are autonomous—free spirits. They can just as easily call Seam their master as they do now with Spring. In fact, letting a class roam between containers is a great exercise to validate its “POJO rating”—the degree to which a class can stand alone. It's a judicious test to see if we're truly getting our money's worth for going POJO. If the Spring designers' word holds true, we just got a whole new library of components to incorporate into our Seam applications.

### 15.1.1 Assimilating Spring POJOs into Seam

As suggested in the introduction, POJO integration is nothing more than taking a POJO class from another framework library, in this case Spring, and declaring and configuring it as a Seam component. Assimilating a class into the Seam container can be done either by extending the class and adding a `@Name` annotation to the child class or by declaring the original class as a Seam component in the component descriptor (e.g., `components.xml`). You've seen POJO-based component declarations many times throughout this book, so this task should be nothing new.

By turning the class into a Seam component, the POJO gets all the nice enhancements that existing Seam components are already enjoying, such as bijection, statefulness, component life-cycle hooks, component events, custom interceptors, declarative transaction boundaries, security, remoting, and asynchronicity. To get some of these features, you have to extend the Spring class to add the appropriate Seam annotations.

Which classes of Spring can be assimilated into Seam? Well, it depends on their POJO rating, of course! As long as the class can act as a regular object, meaning it doesn't attempt to call out to the Spring container or depend on any Spring container magic (at least that we can't emulate), it's a viable candidate. Here's a small sampling just to get you thinking:

- *Custom*—Classes in your application being used as Spring beans
- *Lightweight remoting*—`HttpInvoker`, `Hessian`, `Burlap`, and `POJO`
- *JDBC*—`JdbcTemplate` and `SimpleJdbcTemplate`
- *JMS*—`JmsTemplate`
- *iBatis*—`SqlMapClientFactoryBean` and `SqlMapClientTemplate`
- *JNDI*—`JndiObjectLocator`
- *JMX*—`MBeanExporter`

Notice that I didn't put `JpaTemplate` or `HibernateTemplate` on this list. It's true that these two classes are POJO candidates, but you tend not to need them now that you have Seam-managed persistence contexts and the Seam Application Framework components, `Home` and `Query`. If you still find value in these two POJO candidates, section 15.4 will be of interest, which shows how to incorporate Seam-managed persistence contexts into your Spring beans so that the JPA and Hibernate template classes can leverage the features of an extended persistence context.

Let's put POJO integration into practice by configuring an `HttpInvoker` client component in Seam to talk to a remote Spring service. I'm jumping right into lightweight remoting services for three reasons: Spring is good at them, they provide an example of a feature that Spring offers but Seam doesn't, and all the interesting applications these days are mashups. Remote method execution is the key to inexpensive mashups, and Spring makes it ridiculously easy to develop. Don't skip over this example if you're not yet using remote calls in your application. The example perfectly demonstrates POJO integration, and you may even decide to add remoting to your application after seeing it here in action.

### 15.1.2 A lightweight integration example

In this example, you're going to have a chance to see the POJO integration from three different angles. From one view, you'll see how you can use a Spring Framework class directly as a Seam component to consume a lightweight remoting service. On the flip side, you'll see how this service is exported by the Spring container. Finally, you'll see how the two POJO components interact with each other through a container-agnostic remoting protocol. The loose coupling present in this example demonstrates the sheer value of POJO-based frameworks.

#### OPENING THE DOOR TO SPRING

Before we get started developing the remoting client, it's necessary to get the configuration in order. The only requirement for performing POJO integration with Spring is to put the Spring JAR file on your classpath. Fortunately, seam-gen already did this task for you when it set up your project. See section A.4 of appendix A for details on how to add a new JAR file to the project deployment and to the IDE build path. With that taken care of, let's get to work!

The Open 18 members have been practicing hard and they are eager to put their skills to work playing in tournaments to win money and fame. The Open 18 application will be enhanced to show a calendar of upcoming tournaments and relevant information. However, the tournament database is maintained by a partner of ours, so we can't merely reverse-engineer the tables and make a CRUD task out of it, as we've done in previous chapters. Instead, we access our partner's remote service endpoint, exposed using Spring's `HttpInvokerServiceExporter`, to gather the information. Although I chose `HttpInvoker` for this example, the configurations for consuming services exposed via the Hessian, Burlap, or RMI exporters are almost identical, so you're really getting four for one.

#### GOING TO THE SOURCE OF THE SERVICE

To add clarity to the client-side requirement, we visit the development shop of our partner and check out the Spring application that exports the tournament service. While we're there, we grab the JAR containing the service interface and model class so that we can interact with our partner's endpoint. We don't need the implementation of the service, of course, since that's all handled on the other side of the fence.

After some brief introductions, we begin by pulling up the model class, `Tournament`, shown in listing 15.1. This class holds the tournament details.

#### Listing 15.1 The class that models a golf tournament

```
package org.open18.partner.model;

import ...;

public class Tournament implements Serializable {
    private Long id;
    private String name;
    private Date startDate;
    private Date endDate;
}
```

```

private String hostFacilityName;
private String hostFacilityLocation;
private Date entryDeadline;
private List<String> sponsors;
private List<String> benefitingCharities;
private String summary;
private String contact;
private Double entryFee;
private String purse;
private String phone;
private String website;
private String email;

// getters and setters hidden
}

```

The `Tournament` class could just as easily have JPA annotations on it, but since annotations are just metadata, they don't affect how the class is used in a remote service. Information about the tournaments in the partner's database is exposed through the `TournamentService` interface:

```

package org.open18.partner.service;

import ...;

public interface TournamentService {
    List<Tournament> getUpcomingTournaments();
}

```

One of the developers reveals to us how the implementation of this service is defined in his Spring configuration file. The definition is simplified for this example, shown here without the transaction and security proxies that would normally be used:

```

<bean id="tournamentService"
      class="org.open18.partner.service.impl.TournamentServiceImpl"/>

```

The `HttpInvokerServiceExporter` passes on calls to this service implementation by responding to HTTP requests (no surprise there). To prepare to accept requests, this service must be connected to a servlet, the Spring MVC `DispatcherServlet` to be exact. The servlet is defined in our partner's `web.xml` descriptor:

```

<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.service</url-pattern>
</servlet-mapping>

```

This configuration says that URLs ending in `*.service` should be treated as remote method invocations and should be passed on to the appropriate Spring bean. The

mapping from URL to bean ID is defined in the `dispatcher-servlet.xml` descriptor, a file whose name is derived by combining the name of the servlet with the suffix `-servlet.xml`:

```
<bean id="urlMapping" class=
  "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/tournament.service">httpTournamentService</prop>
    </props>
  </property>
</bean>
```

Finally, the `httpTournamentService` bean is defined by exporting the tournament service using the `HttpInvokerServiceExporter` in the main Spring configuration file:

```
<bean id="httpTournamentService" class=
  "org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="tournamentService"/>
  <property name="serviceInterface"
    value="org.open18.partner.service.TournamentService"/>
</bean>
```

That completes the configuration necessary to export the `TournamentService` interface over the `HttpInvoker` channel. Fortunately, we don't have to write (or maintain) the code we've seen thus far. That's the responsibility of our partner. However, it would be nice to be able to run it in order to test the client we write. To help us with testing, our partner has provided us with a deployable archive, `open18-partner.war`, that we can run locally. It responds with canned data, which is fine for our needs. When run locally, the tournament service is available at the following URL:

```
http://localhost:9090/open18-partner/tournament.service
```

We pack our bags with JAR and WAR archives in hand and head on home.

### **MAKING THE REMOTE CALL**

Back at the lab, we're ready to put this service to the test using nothing more than POJO integration. If we were using Spring, the `HttpInvoker` client would be handled by the following bean definition in the client application:

```
<bean id="tournamentService" class="org.springframework.
  remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
    value="http://localhost:9090/open18-partner/tournament.service"/>
  <property name="serviceInterface"
    value="org.open18.partner.service.TournamentService"/>
</bean>
```

However, we aren't using a Spring application context (at least, not yet). Instead, we want to mirror this configuration as a Seam component declaration:

```
<component name="tournamentService" class="org.springframework.
  remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl">
    http://localhost:9090/open18-partner/tournament.service
```

```

</property>
<property name="serviceInterface">
  org.open18.partner.service.TournamentService
</property>
</component>

```

Looks quite similar, doesn't it? There are two roadblocks that prevent the `HttpInvokerProxyFactoryBean` component from operating properly in Seam. Pay close attention, because what I'm about to tell you will be relevant to nearly every Spring–Seam POJO integration: First, the `HttpInvokerProxyFactoryBean` is *both* an initializing bean and a factory bean. Initializing beans in Spring, as part of their `InitializingBean` contract, define the method `afterPropertiesSet()`, which Spring's built-in bean postprocessors call after the bean is instantiated. Seam will need to emulate this postconstruct hook. Second, when a Spring factory bean is injected or accessed, it resolves to the object that it produces. Sounds a lot like a Seam manager component—a component with an `@Unwrap` method—doesn't it? What we need to do is create an adapter for `HttpInvokerProxyFactoryBean` that can satisfy both of these behaviors when used as a Seam component. The adapter simply extends the Spring class, augmenting the two relevant methods with the necessary Seam annotations, `@Create` and `@Unwrap`:

```

package org.open18.spring.adapter;

import org.springframework.remoting.httpinvoker.
    ↳HttpInvokerProxyFactoryBean;

@Scope (ScopeType.APPLICATION)
public class HttpInvokerProxyFactoryBeanAdapter
    extends HttpInvokerProxyFactoryBean {

    @Create public void afterPropertiesSet () {
        super.afterPropertiesSet ();
    }

    @Unwrap public Object getObject () {
        return super.getObject ();
    }
}

```

We modify the definition of the component that consumes the `HttpInvoker` service to use this adapter:

```

<component name="tournamentService" class="org.open18.spring.adapter.
    ↳HttpInvokerProxyFactoryBeanAdapter">
  <property name="serviceUrl">@tournamentServiceUrl</property>
  <property name="serviceInterface">
    org.open18.partner.service.TournamentService
  </property>
</component>

```

We can't let this definition rest just yet. Hardcoding the service URL in the component definition is a bad idea. That's why I made the value of the `serviceUrl` property a replacement token, highlighted in bold in the previous snippet. The value of this

token is fulfilled by a corresponding property key in `components.properties` and can therefore be controlled by the build process:

```
tournamentServiceUrl=\
http://localhost:9090/open18-partner/tournament.service
```

The only work that remains is to invoke the `getUpcomingTournaments()` method on the client and display the calendar on a page in the application. The tournament results can be captured using a factory:

```
<factory name="upcomingTournaments" scope="event"
value="#{tournamentService.upcomingTournaments}"/>
```

Alternatively, you could inject the `tournamentService` component into another component and export the result using the `@Factory` annotation. Injecting the client into a component allows you to do more sophisticated logic or just avoid the XML. For instance, each record may contain a host facility id that corresponds to a facility in the Open 18 database. You could merge that information into each record before the factory call is complete. Either way, it's important to use a factory because you want to avoid invoking the remote service more than once per request because the call can be expensive and it's just unnecessary chatter.

The final step is to display the results on the tournament calendar page, `tournaments.xhtml`:

```
<h:dataTable var="_tournament" value="#{upcomingTournaments}">
  <h:column>
    <strong>#{_tournament.name}</strong>
    @ #{_tournament.hostFacilityName}, #{_tournament.hostFacilityLocation}
    #{_tournament.startDate} to #{_tournament.endDate}
  </h:column>
</h:dataTable>
```

That wraps up your first POJO integration! Not only did you get a chance to see a Spring class used directly in Seam, but you had the opportunity to implement a lightweight web service at the same time. I encourage you to use this example as an inspiration to try out other POJO integrations, perhaps drawing from the list provided earlier.

Although the technique just demonstrated is useful, you do lose out on the enhancements that the Spring container provides. Next we examine how to access Spring-configured beans from Seam using a variable resolver bridge. That, of course, means that we need to start the Spring container, specifically the `WebApplicationContext` variant. For now, we assume that it's started in the typical way and the Spring beans that we'll reference are those loaded by this Spring container implementation.

## 15.2 *Poor man's integration*

Exhibiting one of the great virtues of a programmer, laziness (and perhaps a little impatience as well), we'll see how far we can get piggybacking on the Spring-JSF integration to allow Seam to locate Spring beans. Seam doesn't have to invest any special

configuration to make this integration possible because it relies on the Spring-aware JSF variable resolver to do the legwork. That is why I term this the “poor man’s integration.”

### 15.2.1 Building bridges with variable resolvers

The Spring-JSF integration acts as a bridge between the Seam and Spring containers. You build this bridge by registering a custom JSF variable resolver, included with the Spring distribution, in the JSF configuration file (i.e., /WEB-INF/faces-config.xml):

```
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
</faces-config>
```

In JSF, a variable resolver represents a pluggable, chaining mechanism for resolving a top-level EL variable reference (i.e., `#{variableName}`) at evaluation time. Spring’s delegating variable resolver taps into this mechanism to make Spring beans available to JSF by mapping the variable name to a bean ID. For instance, the value expression `#{tournamentManager}` would resolve to the following Spring bean:

```
<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl"/>
```

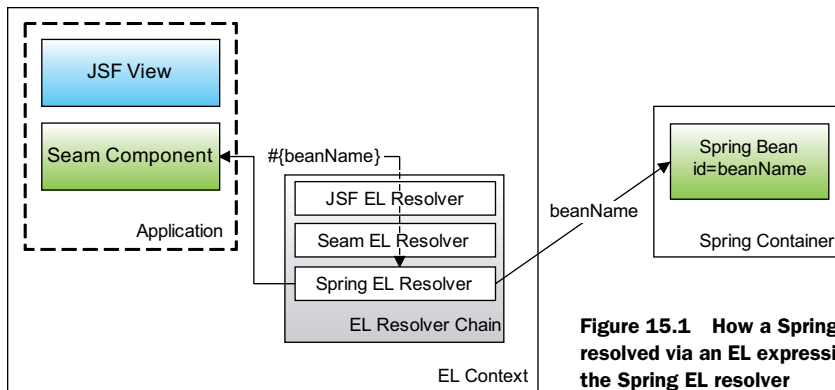
When a value expression is resolved, the registered variable resolver chain is activated. The `DelegatingVariableResolver` sits at the forefront of this chain, and is thus consulted first. The Spring-aware variable resolver starts by delegating to the default variable resolver to allow JSF managed beans to be resolved. If that search turns up empty, it then looks for Spring beans that match the name being resolved, returning a value if it finds one or null otherwise.

The reason this bridge works is because Seam relies on the JSF variable resolver mechanism under the covers to translate unified EL expressions. If you’ve made the move to Spring 2.5, you can take advantage of Spring’s custom EL resolver instead. The EL resolver is registered in place of the variable resolver in the JSF configuration file:

```
<faces-config>
  <application>
    <el-resolver>
      org.springframework.web.jsf.el.SpringBeanFacesELResolver
    </el-resolver>
  </application>
</faces-config>
```

Figure 15.1 helps visualize how a Spring bean is resolved from an EL expression through the Spring EL resolver, after no match is found by either the JSF or Seam EL resolvers.

On the surface, the variable resolver and the EL resolver appear to be the same, both providing access to Spring Beans through the use of value expressions. However, the latter gives you full EL capabilities, as discussed in the sidebar “Variable resolvers vs. EL



**Figure 15.1** How a Spring bean is resolved via an EL expression using the Spring EL resolver

resolvers,” including the enhancements added by Seam’s EL interpreter and those provided by the JBoss EL, both of which were covered in section 4.7.2 in chapter 4. In essence, access to Spring beans using EL notation is on par with the access you have to Seam components.

### Variable resolvers vs. EL resolvers

It’s easy to confuse a variable resolver and an EL resolver, since the behavior of each is nearly identical in some contexts. An EL resolver combines the functionality of a variable resolver and a property resolver into a single, “unified” API. On top of that, it offers a lot more power to the interpreter, elevating the EL notation to the level of a mini-scripting language. Where this expressiveness has been put to use is the parameterized binding expression support in the JBoss EL resolver.

The EL resolver became part of JSF 1.2. Seam uses EL resolvers, as opposed to variable resolvers, to process value- and method-binding expressions. If you haven’t yet moved to Spring 2.5, you have to rely on a variable resolver to look up Spring beans from within JSF. By switching over to Spring 2.5, you can take advantage of the new EL resolver. Either one works for the poor man’s integration, and that’s all that matters right now.

There’s one potential limitation of using the JSF variable and EL resolver bridges: they only work in the context of the JSF life cycle. If you recall from chapter 3, the Seam life cycle is broader than its JSF counterpart, even encompassing non-JSF requests. Clearly the JSF resolvers won’t be available in those cases. You often find yourself in this predicament when using Seam’s JavaScript remoting. Requests issued by a JavaScript remoting call don’t trigger the JSF life cycle.

Fortunately, Seam anticipates this need by mocking an EL context outside of JSF. Seam’s own `SpringELResolver` is registered with this mock context so that Seam is able to locate Spring beans during non-JSF requests using an approach comparable to that done by `DelegatingVariableResolver` and `SpringBeanFacesELResolver`. The











































































