

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

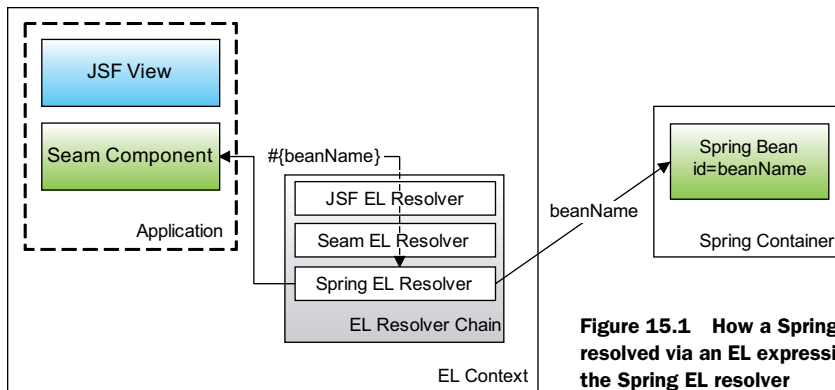


Figure 15.1 How a Spring bean is resolved via an EL expression using the Spring EL resolver

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

SpringELResolver is installed automatically with the mock EL context if Spring is detected on the classpath. Note that you do need to have the IoC module JAR file on your classpath, `jboss-seam-ioc.jar`, which projects created by `seam-gen` already include.

What makes hooking up the Spring container to the unified EL so powerful is that the caller has no knowledge that it's getting a bean from the Spring container, as opposed to the Seam container or the JSF managed-bean container, when an expression is resolved. That's where the EL gets the label "unified." With the `DelegatingVariableResolver` covering JSF requests and the `SpringELResolver` covering non-JSF requests, the poor man's integration is complete. Now we just need to start making use of it!

15.2.2 Delegating work to Spring beans

As it turns out, our preference for Seam has rubbed off on our partner. The developers called us back into their office because they like what Seam has to offer and they want to use it in their new tournament management user interface. But they need us to help them get started. They plan on using Seam components as action handlers to support the view. However, they don't want to have to give up the Spring beans in the business layer that manage the tournament database. Therefore, the action components will need to access these Spring beans. They also need Spring to keep the tournament service going for partners like us. I'd say this is an excellent case for some poor man's integration.

Our partner's application currently uses JSF but not Seam. In fact, the application already has the `DelegatingVariableResolver` (or `SpringBeanFacesELResolver`) installed. And since the application is built using Maven 2, upgrading to Seam is just a matter of adding a couple of library dependencies to the build file and putting the typical Seam configurations in place. Let's start by looking at the necessary modifications to the Maven 2 POM file.

The Seam developers recommend that if you're going to use Seam, you allow your project's POM to extend the Seam parent POM, which gets pulled from the JBoss Maven repository. See section A.5 of appendix A for details. By doing this, you don't have to worry about supplying any of the versions of libraries on which Seam depends. The versions are maintained as part of the Seam project, a continuation of Seam's goal to be a comprehensive integration platform. You aren't required to use this approach, though, and can certainly provide version-complete definitions for each dependency. Using the Seam parent POM is just more convenient, and therefore we use it here.

Our partner's developers recognize the benefit of using the library dependency configuration maintained by the Seam project members, so the first step to get them set up is to have them add the parent POM declaration in their project POM file:

```
<parent>
  <groupId>org.jboss.seam</groupId>
  <artifactId>parent</artifactId>
  <version>2.0.3.GA</version>
</parent>
```

The next step is to add the Seam dependencies. Maven's transitive dependency mechanism will import all of the libraries that Seam needs, so all we have to do is add the Seam artifacts themselves. The dependencies we need for the remaining examples in this chapter are the Seam core, the UI module, the IOC module (Spring support), and Facelets:

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam</artifactId>
</dependency>
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ui</artifactId>
</dependency>
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ioc</artifactId>
</dependency>
<dependency>
  <groupId>com.sun.facelets</groupId>
  <artifactId>jsf-facelets</artifactId>
</dependency>
```

You may find it necessary to tweak some of the transitive dependencies depending on which environment you'll be deploying to. If you have used Maven for any length of time, you should be familiar with this exercise.

That takes care of the dependencies. Next you need to get Seam registered in the application. Refer back to chapter 3 to find out how to register the `SeamListener`, the `SeamFilter`, and the `SeamResourceServlet`. Technically, only the `SeamListener` is required, but it's good to get all the configuration out of the way right up front so that you don't have to worry about it midway through developing the application.

Once our partner's application has been supplemented with the Seam libraries and configurations, its developers are ready to develop their first Seam component. We start them off simple by helping them to create a Seam component that captures tournament details from a form and passes it on to the business layer to be stored in the database. The same `Tournament` entity class is used as before. The business layer object that manages `Tournament` instances implements the interface `TournamentManager`. The implementation for this interface is defined in their Spring configuration file as follows:

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

NOTE In seam-gen projects, classes referenced in the Spring configuration file must be placed under the `src/model` build path. Of course, these classes are not hot deployable.

The transaction and security layers that wrap the `tournamentManager` bean aren't shown, but they don't affect the nature of this integration. That's because the

DelegatingVariableResolver (or SpringBeanFacesELResolver) only sees the final result and is blind to any proxying. We discuss bean proxies later on.

With the business object in place, we must develop a stateful Seam component that handles the form actions and delegates the CRUD operations to the business object. The TournamentAction component, shown in listing 15.2, serves as the action component. It gets a reference to the TournamentManager Spring bean using @In. The EL expression used in the @In annotation, #{tournamentManager}, resolves to an equivalently named bean in the Spring container, courtesy of the delegating variable resolver.

Listing 15.2 The action component for managing a golf tournament

```
package org.open18.partner.action;

import ...;

@Name("tournamentAction")
@Scope(ScopeType.CONVERSATION)
public class TournamentAction implements Serializable {
    @In private FacesMessages facesMessages;

    @In("#{tournamentManager}")
    private TournamentManager tournamentManager;

    @Out private Tournament tournament;

    @Out private boolean managed;

    @Begin
    public String create() {
        tournament = new Tournament();
        return "/tournamentEditor.xhtml";
    }

    @Begin(join = true, flushMode = FlushMode.MANUAL)
    public String edit(Tournament selectedTournament) {
        tournament = tournamentManager.get(selectedTournament.getId());
        managed = true;
        return "/tournamentEditor.xhtml";
    }

    @Begin(flushMode = FlushMode.MANUAL)
    public String view(Tournament selectedTournament) {
        tournament = tournamentManager.get(selectedTournament.getId());
        return "/tournament.xhtml";
    }

    @End
    public String save() {
        if (tournamentManager.exists(tournament)) {
            facesMessages.add(FacesMessage.SEVERITY_ERROR,
                "A tournament with that name, date, and location " +
                "already exists in the database.");
            return null;
        }

        tournamentManager.save(tournament);
        return "/tournament.xhtml";
    }
}
```

```

    }

    @End
    public String delete() {
        tournamentManager.remove(tournament.getId());
        return returnToList();
    }

    @End
    public String cancel() {
        return tournament.getId() != null ?
            "/tournament.xhtml" : returnToList();
    }

    @End
    public String returnToList() {
        return "/tournaments.xhtml";
    }
}

```

When the tournament editor form is submitted, the `Tournament` instance is injected into the action component, the component uses the manager to validate that the tournament isn't a duplicate, and finally, the component passes the tournament to the manager to be saved. The component assumes that the manager will flush the persistence context. There are other bits of functionality to inform the user how things went and to direct traffic. The key feature to pay attention to is that all the back-end work is being delegated to the Spring bean. The action component is just a facilitator. Granted, we're back to using a middleman, which is the layer that Seam tries to eliminate. However, we can't yet use the Spring bean directly in the UI since it can't access components in the Seam container and can't perform bijection. The next level of integration is upgrading the Spring bean so that it can be used both as an action component and a business object, similar to what's done with EJB 3 components. We get to that shortly.

Spring beans can be referenced using a value or method expression anywhere that EL notion is accepted, including JSF views, component descriptors (`components.xml` or `*.component.xml`), page descriptors (`pages.xml` or `*.page.xml`), `@In` annotations, or in jBPM page flows or process descriptors. However, keep two important things in mind. First, referencing a service method using a value expression (`{tournamentManager.all}`) in a JSF view can be a performance risk. I see this done a lot, and it's a bad practice. The getter method will likely be executed many, many times in one request, potentially leading to excessive database access. It's much better to use a factory or a collaborator component to ensure that the call to the business method is only made when needed.

Also keep in mind that most Spring beans are singletons and thus agnostic to state. This inability to operate on contextual variables renders their use somewhat limited in the scope of a stateful process such as a conversation. If you want to be able to continue using all the nice features provided by the Spring container—such as advanced AOP and XML-declared transaction boundaries—but want to weave functionality into these beans that's provided by Seam—such as stateful scopes, bijection, and extended

persistence contexts—then you’ve outgrown the poor man’s integration. If you find yourself in this predicament, it’s time to take the integration more seriously (and graduate from the laziness).

While the poor man’s integration may suffice for coarse-grained sharing of objects, you have much to gain by establishing a genuine integration between the two containers. The goal is to have the components recognize the presence of each other, rather than having the EL resolver mediate their interaction. In the next section, you’ll learn to create Spring–Seam hybrid components and how to let Spring beans live in Seam scopes. But don’t devalue the purpose of the poor man’s integration. It still has its uses, as you’ll discover later in this chapter.

15.3 Spring and Seam, working together

Spring and Seam can peacefully coexist in the same application while remaining mostly oblivious of each other. However, to allow cross-breeding to occur, it’s essential to have the two containers start up simultaneously rather than in isolation. Otherwise, you could face an impossible situation where each container has a dependency on a startup component from the other, preventing either container from being able to load first. Fortunately, Seam can facilitate a tandem startup, taking care of resolving any circular prerequisites.

15.3.1 Bootstrapping Spring and Seam simultaneously

Both Seam and Spring provide servlet context listeners that are used to bootstrap the respective containers. If you register both listeners in the same Web application, the containers are loaded based on the order of the listeners in the web.xml descriptor. Instead of betting that load order alone will guarantee a successful startup, it’s better to give one container the responsibility of booting the other. Seam steps forward and offers the `ContextLoader` component, configured using the `<spring:context-loader>` element in the component descriptor, to handle this task. This built-in Seam component performs the same work as Spring’s `ContextLoaderListener`. In fact, the Spring container won’t even recognize that it’s being started through a different mechanism. If you’re going to be using Spring and Seam together in an application, I *strongly* recommend that you leverage this boot configuration.

To activate the `ContextLoader` component, you first register the component namespace `http://jboss.com/products/seam/spring`, prefixed as `spring`, in the component descriptor. You then add the following component definition, shown in bold:

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ...
  xmlns:spring="http://jboss.com/products/seam/spring"
  xsi:schemaLocation="
    ...
    http://jboss.com/products/seam/spring
    http://jboss.com/products/seam/spring-2.0.xsd
```

```

    http://jboss.com/products/seam/components
    http://jboss.com/products/seam/components-2.0.xsd">
<spring:context-loader/>
</components>

```

If defined without any properties, it mimics the behavior of Spring's `ContextLoaderListener` by assuming that the Spring configuration is located at the resource path `/WEB-INF/applicationContext.xml`. With Seam assuming control of the Spring startup, the `contextConfigLocation` servlet context parameter in the `web.xml` descriptor that's used by Spring is no longer needed.

If your Spring configurations are stored elsewhere, you can set one or more Spring configuration file locations on the `configLocations` property of the `ContextLoader` component. The `configLocations` property is a String array and can be configured in the typical way for a multivalued component property.

Here are several customized configurations for the `ContextLoader` component. The first uses a single Spring configuration file in a nondefault location:

```
<spring:context-loader config-locations="/WEB-INF/spring-beans.xml"/>
```

The second divides the Spring configuration files up into layers and positions them on the classpath rather than in the web configuration directory:

```
<spring:context-loader
  config-locations="classpath:spring-beans-persistence.xml
  ──>classpath:spring-beans-business.xml"/>
```

You can also break the value of the `configLocations` property into child elements:

```
<spring:context-loader>
  <spring:config-locations>
    <value>classpath:spring-beans-persistence.xml</value>
    <value>classpath:spring-beans-business.xml</value>
  </spring:config-locations>
</spring:context-loader>
```

Putting Seam in charge of loading the Spring container also makes it easy to integration-test Seam applications that rely on Spring beans. When Seam's integration test infrastructure loads the Seam container, it transitively loads the Spring container. Seam's integration-test infrastructure is activated by writing tests that extend the `SeamTest` base class.

Using Seam and Spring in the same application goes beyond just divvying up your managed objects between two containers. What this integration affords you is the possibility of promoting Spring beans into Seam components at the level of the configuration metadata to form a Spring–Seam hybrid component, which you'll learn about in the next section. The simultaneous loading of the two containers permits this type of integration because the Seam container is active and ready to register additional components at the time the Spring configuration file is parsed.

The two containers can now peacefully coexist regardless of what integration requirements we throw at them. And here they come! We start by promoting Spring

beans to Seam components, rather than solely relying on the variable resolver trick covered in section 15.2.

15.3.2 Spring–Seam hybrid components

In the POJO integration section, you discovered that there’s a certain amount of magic behavior that Spring adds to its beans, in the form of `BeanPostProcessors`. If you register a Spring Framework class that relies on functionality provided by a postprocessor directly as a Seam component, you take on the burden of emulating the work done by the Spring container. It would be better to let Spring do the necessary prep work and then have Seam come along and consume the final product.

That perfectly describes the behavior of a Spring–Seam hybrid component. Seam provides an XML tag that’s used in a Spring bean definition to augment the Spring bean so that it can be treated as a first-class Seam component. Like other Seam components, the hybrid is given a name, registered with the Seam container, and wrapped with Seam interceptors when instantiated. Let’s see how Seam permits you to weave this functionality into the definition of a Spring bean.

PREPARING SPRING TO GIVE AWAY BEANS

To get started with creating Spring–Seam hybrid components, there are two requirements your application must meet. You must first include the Seam IoC integration library, `jboss-seam-ioc.jar`, on your classpath, which `seam-gen` takes care of for you. If you’re using a Maven 2 project, the dependency was prepared in section 15.2.2. The next step is to register the Seam XML namespace `http://jboss.com/products/seam/spring-seam`, prefixed as `seam`, in the Spring configuration file, as shown here:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ...
  xmlns:seam="http://jboss.com/products/seam/spring-seam"
  xsi:schemaLocation="
    ...
    http://jboss.com/products/seam/spring-seam
    http://jboss.com/products/seam/spring-seam-2.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <!-- bean definitions -->
</bean>
```

Note that we’re now dealing with Spring namespaces that are used in the Spring configuration file, not the Seam namespaces that were shown earlier. This namespace declaration imports the custom tags from the `seam` namespace (e.g., `<seam:component>`) into your Spring configuration. The custom tags in the `seam` namespace do more than just reduce keystrokes and provide implicit validation of property names. The key feature of using custom tags in a Spring configuration file is the bean creation hook captured by an implementation of Spring’s `NamespaceHandler` class. This implementation is bound to a particular XML namespace prefix. The `NamespaceHandler` parses the XML elements that it matches, either to produce a Spring bean, decorate an existing bean, or register functionality with the Spring container.

XML Schema and type-safe configuration

One of the most anticipated features of Spring 2.0 was the introduction of custom XML namespace support. XML Schema-based configuration allows for the use of domain-specific tags (e.g., `<jee:jndi-lookup>`) instead of the generic and more verbose `<bean>` tags. Seam made a similar move from generic `<component>` tags to schema-based component declarations (e.g., `<core:manager>`). The custom tags offer a sort of “type-safe” XML that simplifies the definition, makes it easier to read, and limits the available property names and property types to what the underlying class supports.

Unfortunately, to a newcomer, the XSD declarations at the top of the XML file are downright daunting. Normally, I prefer to do without such XML Hell. However, in this case, the benefits outweigh the negatives because any XSD-aware tool can immediately provide code completion of XML tags without any knowledge of the framework. That sure beats waiting for a vendor to create a tool!

Seam supplies a `NamespaceHandler` implementation to process tags in the Spring configuration file that use the Seam namespace. One such tag, `<seam:component>`, is used to decorate a Spring bean as a `SpringComponent`, the formal class name of a Spring–Seam hybrid. This component is then offered as an authentic Seam component to the Seam container. Seam uses the `SpringComponent` as a wrapper that tunnels through to the Spring container to locate the underlying Spring bean instance. The `NamespaceHandler` can also handle the `<seam:instance>` and `<seam:configure-scopes>` tags, which are addressed in section 15.3.3.

NOTE If you’re not using Seam to start Spring (though I can’t think of any reason why you would resist doing so) and you plan to use Spring–Seam hybrid components, you have to be certain that the `SeamListener` declaration appears before the `ContextLoaderListener` in the `web.xml` descriptor. This ordering is necessary so that the Seam container is active and ready to handle the callbacks from its custom `NamespaceHandler` when the Spring configuration file is processed. However, you may still encounter other, more complex reference problems during startup. Therefore, I strongly urge you to use Seam to start the Spring container.

Let’s see how to incorporate the `<seam:component>` tag into a Spring bean definition.

DRESSING UP A SPRING BEAN AS A SEAM COMPONENT

Seam is able to decorate a Spring bean as a Seam component by nesting a `<seam:component>` inside the `<bean>` element that defines the Spring bean. This setup is different from the EL resolver approach mentioned earlier because it allows the Spring bean to become a first-class citizen in the Seam container. This gives you much more control over how the Spring bean behaves under the Seam runtime. Figure 15.2 illustrates how a bean definition containing a nested `<seam:component>` tag is registered with both the Spring and Seam containers. Note that the Spring container isn’t aware

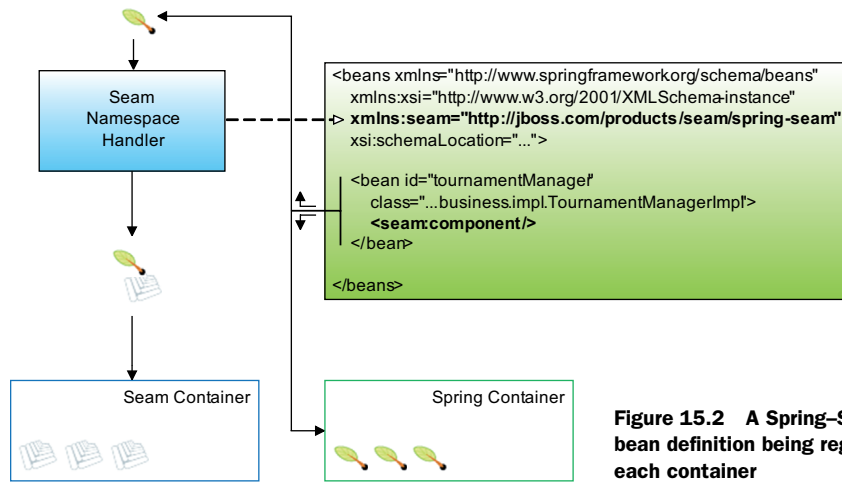


Figure 15.2 A Spring–Seam hybrid bean definition being registered in each container

of the hybrid. Thus, the hybrid must always be accessed through the Seam container or, as you discover later, the `<seam:instance>` tag.

Let's explore this integration by putting it into practice. The `tournamentManager` bean presented earlier has been promoted to a Seam component through the use of a nested `<seam:component>` tag:

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

This bean can now be injected into another Seam component using `@In` without having to make reference to it using the EL:

```
@In private TournamentManager tournamentManager;
```

Seam derives the name of the Seam component from the Spring bean identifier, which may be supplied as either the `id` or `name` attribute of the `<bean>` element. If for some reason you don't want to use the ID of the Spring bean as the name of the Seam component, you can provide an alternative name using the `name` attribute of the `<seam:component>` tag, as shown here:

```
<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl">
  <seam:component name="tournamentManagerSpring" />
</bean>
```

By default, the Seam component is scoped to the stateless context (i.e., `ScopeType.STATELESS`). The stateless context is selected as the default Seam scope because, more times than not, the Spring bean is maintained as a singleton in the Spring container. Thus, there's no benefit in storing the bean in a real Seam scope, as it already lives for the duration of the application's lifetime. The stateless context is also used to step aside and let Spring manage the scope of the bean, if the bean

isn't a singleton. For instance, Spring now supports custom scopes, such as the flow scope from Spring Web Flow. Use of the stateless context prevents Seam from interfering with Spring's plans for the bean and avoids the case where Seam is holding a stale reference to a bean. However, if the Spring bean is a prototype, it will be instantiated on each lookup. In that case, it's appropriate to store the bean instance in a Seam context.

You specify the Seam context in which to store a Spring bean using the `scope` attribute on the `<seam:component>` tag nested within the `<bean>` definition. The scope names accepted in this attribute are the same as those used in the Seam component descriptor and must be typed in all uppercase letters. Keep in mind that to store a Spring bean in a Seam scope, the Spring bean must be a prototype bean (i.e., assigned the prototype scope).

SCOPING A SPRING BEAN

To demonstrate storing a Spring bean in a Seam scope, let's define a search criteria bean, named `tournamentSearchCriteria`, that's used in conjunction with the `tournamentManager` bean to filter the collection of tournaments displayed in the UI. We store it in the conversation scope to ensure that the search criteria is maintained for the duration of the user's interaction with the search form. The Spring bean definition for the criteria bean is shown here:

```
<bean id="tournamentSearchCriteria"
  class="org.open18.partner.search.TournamentSearchCriteria"
  scope="prototype">
  <seam:component scope="CONVERSATION"/>
</bean>
```

Each time a prototype bean is requested from the Spring container, Spring allocates a new instance. But because the resolved bean is stored in a conversation-scoped context variable in this case, Seam only asks Spring to create an instance once per user conversation. It's an ideal marriage of the functionality from each container. In section 15.3.3, you'll learn another way to scope the Spring bean to a Seam context using Spring's custom scoping mechanism, and you'll also study the dangers of scope impedance. For now, let's get back to the `tournamentManager` bean, because we need to add some capabilities to it that are going to affect how it's wrapped by Seam.

DEALING WITH AOP PROXIES AND COVERT BEANS

Our earlier definition of `tournamentManager` is oversimplified because it doesn't have transactional capabilities. Spring prides itself on providing declarative services, such as transactions, that are completely transparent to the bean on which they're registered. However, it's important to understand how these services are applied because Spring's proxy mechanism has the potential to throw a wrench into the Spring–Seam integration. You'll see an example of this problem and learn how to work around it. In listing 15.3, transaction advice has been registered on the `tournamentManager` bean using Spring's AOP configuration.

Listing 15.3 Adding transaction advice to a Spring bean with AOP

```

<tx:advice id="defaultTxAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
<aop:config proxy-target-class="true">
  <aop:advisor id="tournamentManagerTx" advice-ref="defaultTxAdvice"
    pointcut=
      "execution(* org.open18.partner.business.TournamentManager.*(..))"/>
</aop:config>

<bean id="entityManagerFactory" ...>
  ...
</bean>

<bean id="transactionManager"
  class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

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

```

Defines advice

Applies advice

Receives advice

You may not be a big fan of AOP pointcuts, but bear with me for the time being. This configuration essentially tells Spring's PlatformTransactionManager implementation for JPA, JpaTransactionManager, to apply transactions around all of the methods of classes that implement the TournamentManager interface. Any method beginning with get uses a read-only transaction (no flushing of the persistence context, if available), whereas the remaining transactions force a flush of the persistence context prior to committing. So far, this is just standard Spring behavior and doesn't affect how the Seam component is derived from the target bean. (The complete JPA configuration is presented later in listing 15.7.)

The setting that does affect how the Seam component is created is the proxy-target-class attribute on the <aop:config> element, highlighted in listing 15.3 in bold. The point I am about to make is very important: *Seam cannot adopt JDK proxied Spring beans. Seam can only deal with Cglib and Javassist instrumented classes.*

Huh? If you're new to dynamic proxying and bytecode manipulation, this may sound like gibberish to you. Basically, as part of the AOP mechanism, Spring wraps your Spring bean in an interceptor chain that can trap method calls and add functionality before and after each invocation. This technique is exactly what Seam does to add capabilities, such as bijection, to its components. Seam registers interceptors on components using runtime bytecode enhancement provided by the Javassist library.¹ Spring, on the other hand, uses JDK dynamic proxies by default. The JDK proxying

¹ For EJB 3 components, Seam uses a proxy to register client-side interceptors and registers server-side interceptors with the EJB interceptor chain.

infrastructure is only capable of creating proxies for interfaces, whereas the bytecode enhancement libraries, including Javassist and Cglib, can work directly on classes.

Suffice to say, Seam can't work with the Spring bean if it uses JDK dynamic proxying. This issue doesn't come up when accessing a Spring bean through the EL resolver because Seam only acts as a client, invoking its methods. Here, the Spring bean is being converted to a Seam component, so Seam is involved in the internals of the bean creation process and needs the extra level of visibility.

Why doesn't Seam support JDK dynamic proxies?

A JDK dynamic proxy is a dynamically generated class that implements a set of interfaces specified at runtime. A dynamic proxy class can't be cast to the target bean's class, nor does it provide access to the target bean (i.e., the proxied object). The proxy just happens to implement the same interface(s) as the target bean. Cglib proxies, on the other hand, can be treated more or less like the target bean itself. Cglib creates a new concrete class on the fly that is a subclass (in other words, a class that extends the target bean's class). Because Seam needs direct access to the target bean to expose it as a Seam component, it's mandatory that the target bean be enhanced using Cglib rather than act through a JDK dynamic proxy.

The situation is this: Seam can't handle JDK proxies. Spring can't enhance beans using Javassist. To meet in the middle, Spring can use runtime bytecode enhancement provided by Cglib, which is enough to make Seam happy. The switch to this alternate proxy provider happens on a per-bean basis. In the case of the `<aop:config>` element, that switch is made using the `proxy-target-class` attribute. Despite the horrible name for this attribute, a value of `true` enables Cglib proxies rather than dynamic JDK proxies. If you don't make this switch, you'll get the following exception when you try to make a proxied Spring bean into a Spring–Seam hybrid:

```
Caused by: java.lang.RuntimeException: Seam cannot wrap JDK proxied IoC beans.
Please use CGLib or Javassist proxying instead at
    org.jboss.seam.ioc.ProxyUtils.enhance(ProxyUtils.java:52)
    ...
```

What if pointcuts just aren't your cup of tea? You can opt, instead, to use Spring's `TransactionProxyFactoryBean` to wrap your service layer object in transactions. This bean has a similar switch for its proxy setting, but that isn't the only issue you'll encounter. You now have a new problem. When you use a proxy factory bean, the class name of the target object is no longer on the `<bean>` definition. That means you have to tell Seam explicitly what the class name of the target bean is using the `class` attribute on `<seam:component>`. Let's take a look.

Listing 15.4 shows an example of the `tournamentManager` bean that is semantically equivalent to listing 15.3 but that uses a transaction proxy wrapper instead of AOP pointcuts. Its `proxyTargetClass` property is set to `true` to enable Cglib proxying.

Listing 15.4 Using a Spring proxy factory bean to avoid pointcut expressions

```

<bean id="tournamentManager"
  class="org.springframework.transaction.interceptor.
  ↳TransactionProxyFactoryBean">
  <seam:component
    class="org.open18.partner.business.impl.TournamentManagerImpl"/>
  <property name="proxyTargetClass" value="true"/>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
  <property name="proxyInterfaces"
    value="org.open18.partner.business.TournamentManager"/>
  <property name="target">
    <bean class="org.open18.partner.business.impl.TournamentManagerImpl"/>
  </property>
</bean>

```

You can see in listing 15.4 that the `<seam:component>` element is now using the `class` attribute to specify the name of the class that implements the service layer interface. But notice that the class is also supplied in the `target` property of the bean definition. Why specify it in both places?

The implementation class must be stated explicitly so that Seam knows what type of object the Spring factory bean produces. This information is buried deep inside the `<bean>` definition, where Seam is unable to find it. Without this information, Seam can't perform operations that require knowledge of the hybrid component's type, such as injecting it into the property of another component. What's worse is that without this hint, Seam actually gets the type wrong since it looks at the `class` attribute on the `<bean>` definition to determine the component's type. Unless told otherwise, Seam would assume that the class of the `tournamentManager` component is `TransactionProxyFactoryBean`—hence the requirement.

This problem also comes up when using Spring's bean definition inheritance feature (the `parent` attribute on the `<bean>` definition). When the `<seam:component>` is used in a child bean definition, you once again have to educate Seam about which type to expect. This incapability with bean inheritance will likely be resolved soon, since all of the information Seam needs is there in the bean definition hierarchy—it's just a matter of Seam searching it out. However, the point remains that if there's ever a case where Seam can't determine the type, or gets the type wrong, you can supply an override using the `class` attribute.

Although the `class` attribute is required to get the hybrid component registered properly, Seam has no problem working with a Spring factory bean once the hybrid component is set up. When you inject the `tournamentManager` component into a `TournamentManager` property on another Seam component, Seam correctly unwraps the factory bean to get to the underlying proxied target object, as opposed to trying to

inject the factory bean itself. What you also might find interesting is that Seam can correctly apply component configuration properties to the target bean.

That gets us started with creating Seam components with the `<seam:component>` tag. Let's give more thought to the significance of a Spring bean becoming a Seam component and learn how to fine-tune the behavior of the Seam component that it generates.

ON BECOMING A SEAM COMPONENT

As was mentioned earlier, when a `<seam:component>` tag is detected inside of a `<bean>` element, Seam's `NamespaceHandler` implementation wraps the Spring bean in an instance of `SpringComponent` and registers it with the Seam container, making the Spring bean a full-fledged Seam component. That means that when a method is invoked on the Spring bean, all the normal Seam interceptors are applied. Think about that for a moment. You can now use all of Seam's bijection goodness that was covered in chapter 6 with your Spring beans!

Let's assume that a Spring bean named `tournamentDao`, responsible for performing searching against the database, is also set up as a Spring–Seam hybrid component as explained earlier. Listing 15.5 shows how this data access object can be injected into the tournament manager business object using Seam's `@In` annotation, an alternative to using Spring's dependency injection mechanism. Additionally, Seam's `@Logger` annotation is used to instantiate and inject a logger. Be *very* careful though, because you can't inject stateful components into a Spring singleton using bijection unless you're willing to synchronize the component. That's because bijection is not thread-safe. This problem is discussed in more detail in section 15.5.

Listing 15.5 Using bijection in a Spring bean

```
public class TournamentManagerImpl implements TournamentManager {
    @Logger private Log log;

    @In private TournamentDao tournamentDao;

    public List<Tournament>
        find(TournamentSearchCriteria criteria) {
        if (criteria == null) {
            log.debug("Fetching all tournaments");
            return tournamentDao.getAll();
        }
        else {
            log.debug("Searching for tournaments using criteria object");
            return tournamentDao.queryForList(
                criteria.getSearchPattern(),
                criteria.getSortBy(),
                criteria.getPageSize(),
                criteria.getPage());
        }
    }
}
```

Now we need to supply the search criteria argument to the finder method and capture the result. That is done through the use of a factory defined in the Seam component

descriptor. The finder method is referenced as a value expression and the return value is bound to a Seam context variable using a factory:

```
<factory name="tournamentResults"
  value="#{tournamentManager.find(tournamentSearchCriteria) }"/>
```

Notice that the value expression takes advantage of the JBoss EL, which allows a parameter to be passed to a method, in this case the conversation-scoped search criteria. It's now possible to iterate over the search results in the user interface by referencing the value expression `#{tournamentResults}` in the UI. You could also take advantage of additional Seam annotations, such as `@Factory`, `@Restrict`, `@RaiseEvent`, `@Observer`, and `@Transactional`, to name a few.

TUNING THE HYBRID COMPONENT

If you'd rather not dirty your Spring beans with Seam interceptors, preferring to leave your Spring beans pristine, you can disable Seam interceptors by setting the `intercept` attribute of `<seam:component>` to `false`. Disabling interceptors can improve performance, especially if you aren't going to be using any of the functionality they provide anyway. The change to the definition of `tournamentManager` is shown in bold (we're assuming that the transaction boundaries are being weaved in using the AOP pointcut configuration):

```
<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl">
  <seam:component intercept="false"/>
</bean>
```

With or without interception enabled, you can still use Seam's component configuration mechanism to set initial property values on a Spring–Seam hybrid component. However, customizing properties through Seam should probably be done sparingly (perhaps only for environment-specific overrides) since it can confuse other developers—or perhaps even you, down the road—when attempting to get a complete picture of the component definition. I recommend that you use Spring's property configuration mechanism instead.

Spring–Seam hybrid components, by default, are configured to be autocreated. However, in this case, the term is a bit misleading. That's because the Spring bean may already be initialized at the time the component name is first requested, perhaps because it was created when the Spring container started. But Seam still considers the context variable uninitialized at this point, because Seam hasn't yet consulted the Spring container to determine if the bean has been created. Thus, the auto-create attribute on `<seam:component>` really means “attempt to resolve the Spring bean without being told to do so.” If this isn't the desired behavior, you can disable it by changing the value of the auto-create attribute of `<seam:component>` to `false`:

```
<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl">
  <seam:component auto-create="false"/>
</bean>
```

The auto-create setting is especially relevant considering that most Spring–Seam hybrid components are stateless, resulting in a context variable that’s never permanently initialized. Thus, you’ll likely always want the value of the auto-create attribute to be true. If its value is false, you must use the create flag on the @In annotation when referencing the hybrid’s component name. EL value expressions that reference the component name aren’t affected, since they perform the lookup implicitly.

Upon the component name being requested, if the create flag is true and if the Spring bean exists, it will be returned to Seam. If the create flag is true and the Spring bean doesn’t exist, the Spring container will attempt to create it, and then return the instance to Seam. If the create flag is false, no attempt will be made to communicate with the Spring container.

Autocreate components and container initialization

When the two containers are started in isolation, the load order again reaches a deadlock when autocreate hybrid components are formed from Spring singleton beans. If the Spring container is started first, then when the autocreate component is registered, an error will be thrown that complains that you’re attempting to invoke a Seam component outside of an initialized application. If the Seam container is started first, then when the autocreate component is registered, an error will be thrown stating that the application scope isn’t active. In both cases, the Spring initialization process is operating outside the boundaries of the Seam initialization process. To remedy this problem, and any other intercontainer dependencies, it’s best to allow Seam to start Spring.

THE BEST OF BOTH WORLDS

You can now use Spring beans as needed in your Seam application. The Spring Framework is filled with potential Seam components and convenient functionality; I’d be a fool to tell you otherwise. Supplementing Seam with Spring will likely prove to be very valuable to the success of your application. Why choose?

In fact, Seam can even make Spring beans better by teaching them a thing or two about maintaining application state.

15.3.3 Spring, meet state

So far, creating Spring–Seam hybrid components meant typing `<seam:component>` an awful lot. That’s kind of a downer; after all, the goal of Seam is to reduce typing, and this tag is just adding to the pile of XML mess in Spring. Fortunately, there’s an easier way to achieve the same end that leverages another highly anticipated feature of Spring 2.0: custom scopes. Custom scopes expand the choices for where instances of Spring beans can be stored.

While Spring added the mechanism to support custom scopes, the built-in implementations are stuck in the past, only covering the scopes in the Servlet API (*request*, *session*, and *application*), plus a stateless scope named *prototype*. These scopes are ill suited

for modern business applications. To bring Spring up with the times, Seam registers a custom scope handler that allows Spring beans to be stored in Seam contexts. Uses include the (temporary) *conversation* scope for implementing the redirect-after-post pattern (sometimes called a *flash*); the (long-running) *conversation* scope for single-user page flows; and the *business process* scope to support interactions from multiple users over an extended period of time. At last, Spring beans can be stateful without having to resort to the crutch of the HTTP session!

As it turns out, specifying a Seam scope on a Spring bean has the same effect as applying a nested `<seam:component>` tag to the bean definition in that it results in a Spring–Seam hybrid component. A two-for-one deal! To take advantage of this feature, you first need to register the Seam scope handler in any one of the Spring configuration files (but not more than once) as follows:

```
<seam:configure-scopes/>
```

Of course, that configuration file must also declare the Seam namespace on the root element, which was explained earlier.

To assign a Seam scope to a Spring bean, thus creating a Spring–Seam hybrid component, you simply set the scope attribute on a Spring `<bean>` definition—or any custom namespace element in the Spring configuration that supports the scope attribute—to one of the Seam scopes. The name of the scope is not case sensitive.

To distinguish Seam scopes from scopes provided by other frameworks, the scope name is prefixed with a namespace, which is `seam.`, by default. Putting that together, you create a conversation-scoped component by using the value `seam.CONVERSATION` in the scope attribute of the Spring bean definition. You can customize the namespace prefix that’s used by supplying an override to the `prefix` attribute of the `<seam:configure-scopes>` declaration:

```
<seam:configure-scopes prefix="org.jboss.seam.ScopeType."/>
```

In this case, to scope a hybrid component to the conversation, the value of the scope attribute would be set to `org.jboss.seam.ScopeType.CONVERSATION`, which happens to be the fully qualified name of the enum constant for the conversation scope.

The read and write operations for the `Tournament` entity are handled by the `tournamentManager` bean, a classic singleton Spring bean. It’s not very interesting in our pursuit of state. However, one of the beauties of tying Spring and Seam together is that stateless Spring beans, such as `tournamentManager`, can be infused with state through the use of other stateful beans. Let’s refactor the tournament search criteria object to take advantage of Seam’s extension to Spring’s custom scopes. For now, we’re sticking with the default namespace prefix for the scope name:

```
<bean id="tournamentSearchCriteria"
  class="org.open18.partner.search.TournamentSearchCriteria"
  scope="seam.CONVERSATION"/>
```

When declaring Spring–Seam hybrids using the scope attribute, you do lose some flexibility over the configuration of the component. While setting the scope is in, what

is out is the ability to set the autocreate flag, provide an explicit class, supply a custom component name, and disable Seam interceptors. Fortunately, there are other ways to configure some of these settings, which you'll see in a moment. But remember that the idea here is to use the `<bean>` tag without any additional interference, aside from the value of the scope attribute. This approach is just a shorthand for the more formal `<seam:component>` declaration, which gives you the extra power if you need it, including the ability to specify the Seam scope.

The autocreate flag is set to false by default for components configured using the scope attribute. However, this setting can be controlled globally using the `default-auto-create` attribute on the `<seam:configure-scopes>` tag:

```
<seam:configure-scopes default-auto-create="true"/>
```

For the duration of this chapter, I assume the use of this setting so that Spring initializes the bean for Seam when it's requested.

There's only way to disable interceptors for components configured using the scope attribute: by adding Seam's `@BypassInterceptors` annotation on the class:

```
@BypassInterceptors
public class TournamentManagerImpl implements TournamentManager { ... }
```

Now that you have your Seam-scope Spring bean defined, you're going to want to start using it in other beans. But you have to be careful mixing singletons and stateful objects, which the next section helps you sort out.

15.3.4 *Finding the right balance with injection*

When it comes time to wiring an instance of a Seam component into a Spring bean, you have one or two options, depending on the characteristics of the managed objects. Seam-scoped Spring beans can be injected into the property of another Spring bean using the familiar `<ref bean="beanId">` (or equivalent) syntax on the bean definition in the Spring configuration file. In this case, Spring is transparently reaching into the Seam container to grab the instance of the bean out of a Seam scope. Later on, you'll learn that you can also inject native Seam components in this way, as well as through an analogous tag in the Seam namespace. For Spring–Seam hybrids, you can inject native Seam components and other Spring–Seam hybrids into the property using the `@In` annotation.

That brings us to an important point about injection that I'd like to draw your attention to. While a Spring–Seam hybrid can leverage the injection mechanism from either container, you can quickly get into trouble if you don't understand how and when the injections are applied in each case. For instance, let's assume you're injecting the search criteria into the `tournamentManager` bean, as opposed to through a method parameter as shown earlier. You might decide to use a bean reference when initializing the property in the bean definition:

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

```

<property name="tournamentSearchCriteria"
  ref="tournamentSearchCriteria"/>
</bean>

```

You could also accomplish the injection by using the `@In` annotation instead:

```

public class TournamentManagerImpl implements TournamentManager {
    @In private TournamentSearchCriteria tournamentSearchCriteria;
    ...
}

```

The manner in which these injections are applied by the two containers is very different. And *neither* one is appropriate for injecting a stateful component into an application-scoped singleton! Spring injects dependencies once at creation time, whereas Seam performs injection dynamically, before every method invocation. This difference in design philosophy can fall heavily on your shoulders as a developer, leading to a paradox known as *scope impedance*. If you declare your injections carelessly, it can cause serious problems. In the next section, you'll learn how to work around this potential pitfall.

INJECTION AND SCOPE IMPEDANCE

Scope impedance is the situation where a bean outlives its intended lifetime as a result of attaching itself to a longer-lived component. Having the Seam scopes at hand when authoring your Spring configuration, you may be tempted to define a bean in a narrow scope and inject it into a Spring singleton, which is scoped to the lifetime of the container (the application context), using `<ref bean="beanId">`. Without any other provisions, this leads to scope impedance. Let's explore this problem using the wiring of the search criteria to the manager bean.

In this scenario, we're dealing with two beans, one stored in a wider scope than the other. The wider-scoped bean, `tournamentManager`, is the application-scoped singleton. The narrow-scoped bean, `tournamentSearchCriteria`, is bound to the conversation scope. The conversation-scoped bean is a dependency of the singleton. When the singleton is instantiated, the conversation-scoped bean is injected into one of the singleton's properties. However, once the conversation (or request if the conversation is temporary) ends, the singleton will still hold a reference to the conversation-scoped bean. Now the conversation-scoped bean has outlived its designed lifetime. Not only that, it's being shared among all users of the application because it's attached directly to an application-scoped singleton! (The same goes for request and session-scoped components as well.)

So why isn't Seam managing the narrower-scoped component so that it's cleaned up after the scope ends as it normally would be? The reason this happens is because the component instance is assigned directly to a property of the singleton. When the scope ends, Seam merely removes component instances from that context. The property values of the singleton objects aren't affected. Since the singleton is maintaining a reference to the component instance, the object is never destroyed. On the next request (or conversation), the singleton retains the reference to the stale instance. As a result, the singleton may make inappropriate decisions based on the state of the

stale instance, which is especially problematic if the next request doesn't come from the same user! This situation not only causes unexpected behavior, but can also compromise security. Oh my!

WHAT ABOUT BIJECTION?

Since you know that bijection is activated on every method call on a component, you may decide to make the switch to using the `@In` annotation on the properties of the class rather than using a `<ref>` element in the Spring bean definition (if you don't mind using Seam annotations in your Spring beans). That definitely gets you out of the scope impedance predicament. However, bijection is not thread-safe, which means that injecting stateful components using bijection is only safe for synchronized components.

So how does Seam do it? you might ask. Seam serializes access to session and conversation-scoped components automatically, and request-scoped components are naturally thread-safe, since they only service an incoming request thread (single-threaded). However, Seam doesn't serialize access to application-scoped components, because that would cause an incredible bottleneck in the application. Unfortunately, using `@In` on application-scoped Spring–Seam hybrid components (or regular Seam components, for that matter) is only appropriate if the method call is synchronized. Otherwise, you're either going to allow multiple requests to access the same injected references simultaneously, or if you're injecting other singletons, you may still get a `NullPointerException` because of disinjection if the methods calls overlap.

The simple solution to this problem is to break the habit of using application-scoped (singleton) business components, as Spring has incorrectly trained you to do. You lose a lot of the benefit of an object-oriented language when you choose to use stateless components. Make your business components request-, conversation-, or, dare I say, session-scoped objects and you can leverage stateful mechanisms such as bijection.

But before we close the door on stateless components, let's try to figure out a way to safely inject a stateful component into a pure, application-scope Spring bean. For that, we need a solution that comes from the Spring configuration.

PROXIES TO THE RESCUE

I may have scared you by mentioning the potential of scope impedance to create a security hole, especially since this situation is such an easy trap to fall into. Fortunately, Seam offers an elegant safety net to help you avoid both scope impedance and thread-safety problems. Instead of injecting a bean instance directly using a `<ref>` element, you inject a proxy using `<seam:instance>`, several examples of which are shown in the following bean definition:

```
<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl"
  scope="seam.APPLICATION">
  <property name="tournamentDao" ref="tournamentDao"/>
  <property name="tournamentSearchCriteria">
    <seam:instance name="#{tournamentSearchCriteria}" proxy="true"
      type="org.open18.partner.search.TournamentSearchCriteria"/>
  </property>
  <property name="facesMessages">
```

```

    <seam:instance name="#{facesMessages}" proxy="true"
      type="org.jboss.seam.faces.FacesMessages"/>
  </property>
  <property name="currentUser">
    <seam:instance name="#{currentUser}" proxy="true" scope="SESSION"
      type="org.open18.partner.model.User"/>
  </property>
</bean>

```

Each time the proxy is referenced, it consults the Seam container to ensure it has the proper component instance, and not a stale one from a previous request or from another simultaneous request. Even though the properties on the wider-scoped components still remain unaffected when narrower scopes end, the narrow-scoped instance can now be garbage collected, because the wider-scoped component doesn't hold a direct reference to it. The proxy acts as a weak reference, which is perfect for avoiding scope impedance. In this way, you're effectively infusing state into stateless objects.

The only major downside of `<seam:instance>` is that it suffers from a bit of configuration impedance with Spring. That's because Spring may try to resolve the proxy at container startup to determine the type of the property being injected. To avoid this problem, always explicitly state the type using the `type` attribute of `<seam:instance>`. Unfortunately, this is a design issue that hasn't been worked out at the time of this writing. It might take some trial and error to learn to use this tag effectively.

Seam's `<seam:instance>` vs. Spring's `<aop:scoped-proxy>`

If you're familiar with Seam 2.0 custom scopes, you may recognize that the `<seam:instance>` tag serves the same purpose as the `<aop:scoped-proxy>` tag. The `<aop:scoped-proxy>` tag was also developed to counteract scope impedance for Spring's native scopes. However, this built-in Spring tag isn't compatible with Spring–Seam hybrid components. When a hybrid component is being injected, you must use `<seam:instance>` in place of `<aop:scoped-proxy>` to elicit this behavior.

To summarize, scope impedance arises when using Spring's static dependency injection to wire components in different scopes. Thread safety is violated when you inject stateful components into singletons without using a proxy.

Now that we're on the topic of the `<seam:instance>` tag, you'll be interested to know that it can also be used to inject native Seam components into Spring beans. Let's see how Spring can benefit from already existing Seam components.

15.3.5 Donating Seam components to the Spring container

All we have done so far is take, take, take. It's time to start giving back to Spring. After all, if Spring components are useful to Seam, then the opposite is likely true as well. The `<seam:instance>` tag works for any Seam component, not just Spring–Seam hybrids. But here is the real kicker. The `<seam:instance>` tag can also be used to inject the result of an expression language (EL) value expression into the property of

a Spring bean! That makes the possibilities of what Seam can contribute to Spring beans virtually limitless.

You can think of the `<seam:instance>` tag as an alternative to the `@In` annotation from `bijection`, particularly when the `proxy` attribute is set to `true`. The `name` attribute can either be a context variable name or a value expression. The `scope` attribute narrows the search for the context variable to a single scope. (Note that you use the literal name of the scope, rather than the one with the prefix like you do on the Spring bean definition.) If the `scope` attribute is excluded, Seam performs a hierarchical context search. If the `create` attribute is specified, and its value is `true`, a new instance of the component will be created if one doesn't exist. If this value isn't used, or if it is `false`, a new instance won't be allocated if a value hasn't already been assigned to the context variable. Recall that value expressions imply that the `create` flag is `true`, regardless. The `proxy` attribute is the key to preventing scope impedance, but do keep in mind that it adds an extra bit of overhead.

Setting up a `<seam:instance>` declaration for a component may not seem so bad if you have to do it once, but having to retype the declaration each time you want to inject a given Seam component can be a drag, not to mention error-prone and non-transparent. That's why the `<seam:instance>` element was designed to double as an alias tag. When used as a top-level element, the `id` attribute of `<seam:instance>` can be used to create a mediator bean. To the Spring container, this bean is just like any other Spring bean. Under the covers, `<seam:instance>` results in the creation of a `SeamFactoryBean`, which works just like any other Spring factory bean in that the underlying target object is resolved when the factory bean is injected. In this case, what is injected is a Seam component proxy. Listing 15.6 gives examples of how all of the references used in this chapter can be converted into aliases and injected into the application-scoped `tournamentManager` singleton bean without fearing scope impedance or thread-safety problems.

Listing 15.6 Defining and using aliases for Seam component proxies

```
<bean id="tournamentSearchCriteriaTarget"
  class="org.open18.partner.search.TournamentSearchCriteria"
  scope="seam.CONVERSATION"/>

<seam:instance id="tournamentSearchCriteria"
  name="tournamentSearchCriteriaTarget" proxy="true"/>

<seam:instance id="facesMessages" name="#{facesMessages}" proxy="true"
  type="org.jboss.seam.faces.FacesMessages"/>

<seam:instance id="currentUser" name="#{currentUser}" proxy="true"
  scope="SESSION" type="org.open18.partner.model.User"/>

<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl"
  scope="seam.APPLICATION">
  <property name="tournamentDao" ref="tournamentDao"/>
  <property name="tournamentSearchCriteria"
```

```
    ref="tournamentSearchCriteria"/>
<property name="facesMessages" ref="facesMessages"/>
<property name="currentUser" ref="currentUser"/>
</bean>
```

By setting up an alias, you're free to inject the component into the properties of other Spring beans using the `<ref>` tag, which is blissfully unaware that it's actually injecting a Seam component, possibly even one that is proxied. This usage pattern, in my opinion, is the best way to prevent errors and steer clear of scope impedance. I'd even recommend creating a separate Spring configuration file to hold these alias definitions. Doing so will make it easy to perform functional tests on the Spring POJOs in the absence of the Seam container, since the main file won't have `<seam:instance>` references scattered all over the place. You can easily swap implementations of the referenced beans. It's sort of like a POXD (plain old XML document).

As an alternative to injecting a component proxy into the property of a Spring singleton, you can look up the Seam component from the Seam container and assign the result to a local variable.

15.3.6 Scoped singletons

To avoid direct interaction with the Seam container, you add a static lookup method to the component class that reaches into the Seam container and grabs the component instance by name. Here's the declaration of the static lookup method for the tournament search criteria and an example of its use in the `TournamentManagerImpl` class:

```
public class TournamentSearchCriteria implements Serializable {
    ...

    public static TournamentSearchCriteria instance() {
        return (TournamentSearchCriteria)
            Component.getInstance("tournamentSearchCriteria");
    }
}

public class TournamentManagerImpl implements TournamentManager {
    ...

    public List<Tournament> find() {
        TournamentSearchCriteria criteria =
            TournamentSearchCriteria.instance();
        ...
    }
}
```

The technique used here is known as a *scoped singleton*. Rather than the singleton being instantiated once and shared across the entire application, it's instantiated once and bound to its corresponding stateful scope. That means there's one instance in that particular scope, but there may be many instances across the whole application. For instance, if the stateful component is session-scoped, there can be one instance per user session. The *scoped singleton* allows you to avoid the multi-threaded issues discussed earlier. In addition, since the result of the lookup is

assigned to a local variable, there's no chance of scope impedance. The only downside is that the static instance method relies on the Seam container being available, which can make testing more challenging. However, you can design the lookup so that it has a fallback in a test environment.

One of the prime motivations for the Spring–Seam integration is to be able to donate the Seam-managed persistence context to the Spring container. For this use case, Seam offers a tighter integration than what is possible with the `<seam:instance>` tag. Let's take a look.

15.4 **Learning to share the persistence context**

One of Seam's best features—its conversation-scoped persistence context—can also solve one of Spring's greatest weaknesses. In fact, this feature is by far the most compelling reason for Spring developers to incorporate Seam into their development toolbox and truly makes this integration worthwhile.

For everything except extending the persistence context, the template classes in Spring's object-relational mapping (ORM) module offer a convenient way to manage the Hibernate `Session` or Java Persistence API (JPA) `EntityManager`. But where Spring's ORM support breaks down is in its *mishandling* of the persistence context. Spring can scope the persistence context anywhere from a single operation (method call) to an entire request. But out of the box, it can't go any further. Spring Web Flow, an extension module for Spring, does offer a solution to manage an extended persistence context over the course of a flow, but Seam's persistence context management is arguably more versatile overall.

Let's look more closely at how Spring shanks the persistence context before diving into the configuration necessary to make the exchange of the Seam-managed persistence context possible. The way the persistence manager (the generic term for a Hibernate `Session` or JPA `EntityManager` established in chapter 8) is handled turns out to be the biggest challenge in the effort to integrate Spring and Seam. It's akin to a custody battle over a child in a divorce. But in this case, learning how to share the child (the persistence manager) is what ends up saving the marriage.

15.4.1 **The problem with Spring's thread-local approach**

Spring's ORM support for JPA and Hibernate improved dramatically in Spring 2.0. The JPA support framework can now assume responsibility for injecting an `EntityManager` instance into a `@PersistenceContext` field on a bean in a non-Java EE environment. (The annotation would typically be handled by the Java EE container.) The Hibernate package now leverages the thread-local Hibernate session as returned by the `getCurrentSession()` method on the `SessionFactory`, by implementing the `CurrentSessionContext` contract rather than by introducing yet another proxy layer. However, Spring still imposes one key limitation as custodian of the persistence manager, which interferes with its function in stateful applications: Spring scopes the persistence context to the atomic transaction rather than to the business use case (e.g., the conversation).

REQUEST-SCOPED PERSISTENCE CONTEXTS

Let's start from the viewpoint of the `OpenEntityManagerInViewFilter`. Why is this filter needed in a Spring application? While persistence unit runtimes (e.g., `EntityManagerFactory`) are thread-safe, persistence managers (e.g., `EntityManager`) are not. The former can be stored in the application scope as a singleton. The latter must be reserved for a given use case, which has traditionally been a single request. This filter binds the persistence manager to the thread that's processing the request so that the persistence context remains available for all persistence operations in that request. Without this filter, the best Spring can do is ensure that the persistence manager remains available for the lifetime of the transaction. The worst approach you can take with Spring is to allow the persistence manager to run free (not even binding it to a transaction), in which case a new instance is created for each persistence operation!

As you learned in chapter 9, the persistence context is one of the main reasons for using ORM, and to maximize its value, you need to treat it right. The `OpenEntityManagerInViewFilter` is at least a step in the right direction. With the filter in place, Spring scopes the persistence manager to the thread using the `bindResource()` method of the `TransactionSynchronizationManager`, a page from Hibernate's play-book. (Hibernate uses a thread-local variable to which it binds the `Session` instance, although the `OpenSessionInViewFilter` is still available as an alternative, serving the same role as its JPA counterpart.)

This thread-bound approach gives you the flexibility to lazy-load associations in the view layer from the same persistence manager that retrieved the primary entity instance (meaning that the instances don't become detached at any point during the request). However, the benefit dies with the thread (and, in turn, the request).

Chapter 9 covered the limitations of the Open Session in View pattern in great detail and how the following two problems are solved in Seam using global transactions and an extended persistence context:

- Lazy loading occurs nontransactionally.
- Entity instances are detached before the next request.

The net effect is that Spring lacks the comprehensive request life-cycle management that Seam boasts. To solve the first problem, you learned that Seam uses at least two transactions per request. As a result, lazy loading can take place in a read-only transaction designated solely for the rendering of the view (the JSF *Render Response* phase). Unfortunately, the problem of nontransactional access can go unnoticed in development because it doesn't cause an exception. The second problem, however, can hardly be avoided. When the persistence context is closed prematurely at the end of the request, entity instances propagated through to the next request are abandoned by their persistence manager, and can only contribute to the new persistence context through the use of (dirty) merge operations. You also learned that with detached instances floating around, you run the risk of encountering exceptions that occur as the result of violating the unique entity contract (recall that only a single instance of an entity class with a given identifier is allowed to be associated with a persistence context

at any given time). As the application grows in complexity, the chance of running into this problem increases dramatically.

Fortunately, by moving to Seam-managed persistence, we can say goodbye to the problems of using the persistence context in Spring and the haphazard fix provided by the `OpenEntityManagerInViewFilter`. In making this move, we don't want to lose out on all the Spring classes that make working with Hibernate and the various JPA vendor implementations so easy. Once again, we want the best of both worlds. Seam's inversion of control (IoC) support answers this call. The first step is to get Seam's foot in the door by leveraging Spring's persistence configuration.

15.4.2 *Sharing the duty of persistence management*

How you choose to make the Seam-managed persistence context available to Spring depends on where you want to do the majority of the configuration. The end goal—to replace Spring's thread-bound approach with Seam's conversational one—is all that matters.

Except for differences in naming, the steps for incorporating a Seam-managed `EntityManager` and a Seam-managed Hibernate `Session` into Spring are almost identical. We focus on JPA first and then turn our attention to Hibernate. I must warn you that the first time you see the full handoff in place, it can appear a bit confusing. But by following the steps laid out here, you'll have it worked out in no time.

STEP 1: ENTITYMANAGERFACTORY CONFIGURATION

In the first step, I assume that you covet your persistence configuration in Spring and still want to use it. It's hard to deny that Spring does a great job of consolidating the persistence configuration, making it just another Spring bean definition. You don't have to worry about JNDI resources or special configuration files for each component. It's all there in the Spring configuration file.

If you're using Spring to bootstrap the `EntityManagerFactory`, you set up the `DataSource` and the `EntityManagerFactory` in the Spring configuration file in the typical way. Since the tournament manager web-based application is based on a Spring back end, let's assume the JPA configuration shown in listing 15.7 is already in place. In this case, Spring has primary custody of the persistence unit.

Listing 15.7 Initial Spring JPA configuration

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.
  ↳DriverManagerDataSource">
  <property name="driverClassName" value="org.h2.Driver"/>
  <property name="url"
    value="jdbc:h2:file:/home/twoputt/databases/tournament-db/h2"/>
  <property name="username" value="open18"/>
  <property name="password" value="tiger"/>
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.
  ↳LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="tournament"/>
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaDialect">
```

```

    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
  </property>
  <property name="jpaVendorAdapter">
    <bean
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="true"/>
    </bean>
  </property>
  <property name="jpaProperties">
    <map>
      <entry key="hibernate.dialect"
        value="org.hibernate.dialect.H2Dialect"/>
      <entry key="hibernate.format_sql" value="true"/>
      <entry key="hibernate.hbm2ddl.auto" value="create-drop"/>
    </map>
  </property>
</bean>

```

Since the `DataSource` and the `PersistenceProvider` are both defined in the Spring configuration, you can leave them out of the persistence unit deployment descriptor (e.g., `persistence.xml`), which is nothing more than a required placeholder, shown here:

```

<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="tournament" transaction-type="RESOURCE_LOCAL"/>
</persistence>

```

With the configuration for the `EntityManagerFactory` in place, the handoff begins. Seam needs to get ahold of the Spring-managed `EntityManagerFactory` and use it to create Seam-managed `EntityManager` instances, which get bound to Seam's conversation context. That means you need to be able to expose the `EntityManagerFactory` Spring bean to the Seam container.

The best approach here is to use poor man's integration. I make this recommendation for this part of the handoff because it's an easier route than exposing a Spring factory bean—`LocalContainerEntityManagerFactoryBean`, in this case—as a Spring-Seam hybrid component. All the wrapping and proxying needed to handle a Spring factory bean is a minefield of potential exceptions. If the variable/EL resolver approach works, take it. Besides, you don't need the `EntityManagerFactory` to be a Seam component, since it's only needed for what it produces: `EntityManager` instances.

With that said, Seam resolves the `EntityManagerFactory` from the Spring container using the value expression `#{entityManagerFactory}` and supplies it to the `<persistence:managed-persistence-context>` element in the Seam component descriptor:

```

<persistence:managed-persistence-context name="entityManager" entity-
  manager-factory="#{entityManagerFactory}" auto-create="true" />

```

If you're working with multiple persistence units (e.g., databases), you'll need one of these handoffs for each `EntityManagerFactory` bean definition (e.g., `entityManagerFactory2`, `entityManagerFactory3`).

The `LocalContainerEntityManagerFactoryBean` isn't out of the picture, as you can see. Seam still uses it to produce an `EntityManagerFactory` that it can then use to create the conversation-scoped `EntityManager`. Seam becomes the only consumer of this bean, though. To support autowiring of Spring beans that rely on an `EntityManagerFactory`, or to save yourself from having to change the explicit references to the `EntityManagerFactory` on those same beans if you're not using autowiring, you need to rename the native factory bean to `entityManagerFactorySpring` and reserve the name `entityManagerFactory` for the Seam wrapper, as shown here:

```
<bean id="entityManagerFactorySpring" class="org.springframework.orm.jpa.
    LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="tournament"/>
  <property name="dataSource" ref="dataSource"/>
  ...
</bean>
```

All that's left is to configure the `SeamManagedEntityManagerFactoryBean`. Since this factory bean produces an `EntityManagerFactory` wrapper that distributes the Seam-managed persistence context, the only property it requires is `persistenceContextName`, whose value is the component name assigned to the `<persistence:managed-persistence-context>` in Seam's component descriptor. Notice that this Seam component references the Spring factory bean that has been reserved for this handoff:

```
<persistence:managed-persistence-context name="entityManager"
  entity-manager-factory="#{entityManagerFactorySpring}"
  auto-create="true" />
```

The name of the Seam-managed persistence context is now assigned to the new `entityManagerFactory` bean definition. This bean gives Spring access to the conversation-scoped persistence manager:

```
<bean id="entityManagerFactory"
  class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
</bean>
```

If you decide not to use the naming convention suggested here, make sure that whenever you inject a bean reference into an `EntityManagerFactory` property, such as on a `JpaTemplate` or `JpaDaoSupport` bean, you're using the bean definition for the `SeamManagedEntityManagerFactoryBean`. With that, you're done! You can now use a Seam-managed, conversation-scoped `EntityManager` in Spring beans, Seam components, or Spring–Seam hybrid components.

What follows in the remainder of this section is a bonus round: you'll learn how to take advantage of Spring's emulated resource injections to make your Spring beans appear more like Java EE components by using the `@PersistenceContext` annotation.

EMULATING RESOURCE INJECTIONS

Spring boasts the ability to emulate the behavior of the Java EE container by injecting a persistence manager into a property of a bean annotated with `@PersistenceContext`. This feature is something that even Seam doesn't support, since Seam's position is to

step out of the way and allow Java EE annotations to be handled by the Java EE container. Thus, this is a useful feature to leverage from Spring if you're interested in using this annotation in a stand-alone environment.

Spring's `LocalContainerEntityManagerFactoryBean` loads the persistence unit configuration from the persistence unit descriptor (e.g., `/META-INF/persistence.xml`) whose name is equivalent to the value of the `persistenceUnitName` property, which is set in the bean definition. If a persistence unit name isn't provided, the first `<persistence-unit>` configuration in the persistence unit descriptor is selected. The injection of the persistence manager into properties of Spring beans annotated with `@PersistenceContext` is handled by the Java EE persistence annotation `BeanPostProcessor`, which must be registered in the Spring configuration file as follows:

```
<bean class="org.springframework.orm.jpa.support.
    ↳PersistenceAnnotationBeanPostProcessor"/>
```

When Spring encounters a `@PersistenceContext` annotation on a bean property, it uses the value of the annotation's `unitName` attribute to locate an `EntityManagerFactory`-producing factory bean with a matching `persistenceUnitName` property value, or matching against the bean ID as a fallback. If a `unitName` attribute isn't specified in the annotation, Spring uses the first `EntityManagerFactory`-producing factory bean defined in the Spring configuration. An example of using the `@PersistenceContext` to inject a transaction-scoped `EntityManager` into a Spring bean is shown here:

```
public class JpaTournamentDao implements TournamentDao {
    @PersistenceContext(unitName = "tournament")
    private EntityManager entityManager;
    ...
}
```

If there's only a single persistence unit in your application, you can safely leave off the `unitName` attribute. Note that you don't have to worry about thread-safety here, as the `EntityManager` instance that Spring injects is merely a proxy to the real transaction-scoped `EntityManager`. It would pose a thread-safety issue if you set the value of the `type` attribute to `PersistenceContextType.EXTENDED`.

With some additional configuration, the `SeamManagedEntityManagerFactoryBean` can be made to step into the middle of this lookup logic to ensure that the `EntityManager` injected is the conversation-scoped one managed by Seam. The `SeamManagedEntityManagerFactoryBean` is unique in that it doesn't require a persistence unit name like its Spring counterpart; this is because it's merely a wrapper around an existing Seam-managed `EntityManager`. Therefore, the `persistenceUnitName` property on this factory bean definition can be assigned an arbitrary value as a way of exporting an artificial persistence unit. By default, the value of the `persistenceUnitName` property is made to be equivalent to the value assigned to its sibling `persistenceContextName` property. You'll likely want to choose a more meaningful name—perhaps the suffix `:seam` as the following stanza shows—so that it's clear that the value of `persistenceUnitName` is both artificial and managed by Seam:

```

<bean id="entityManagerFactory"
  class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
  <property name="persistenceUnitName" value="tournament:seam"/>
</bean>

```

You can now inject the Seam-managed persistence context into any Spring bean by supplying this artificial persistence unit name to a `@PersistenceContext` annotation:

```

public class JpaTournamentDao implements TournamentDao {
    @PersistenceContext(unitName = "tournament:seam")
    private EntityManager entityManager;
    ...
}

```

However, you have to be careful! The `EntityManager` injected in this case isn't a proxy, but instead an actual `EntityManager` instance held in Seam's conversation scope. Another way to say this is that you're now dealing with an extended persistence context. An extended persistence context is a stateful object. Therefore, you can't use this annotation to inject an extended persistence context into a stateless, application-scoped singleton! What you'd need to do is alter your bean definition for `JpaTournamentDao` to make it stateful, perhaps binding it to the Seam conversation scope. The same must be done for `TournamentManagerImpl` to ensure that the entire stack is stateful:

```

<bean id="tournamentDao"
  class="org.open18.partner.dao.jpa.JpaTournamentDao"
  scope="seam.CONVERSATION"/>

<bean id="tournamentManager"
  class="org.open18.partner.business.impl.TournamentManagerImpl"
  scope="seam.CONVERSATION">
  <property name="tournamentDao" ref="tournamentDao"/>
  ...
</bean>

```

If you want to use the `@PersistenceContext` without specifying the `unitName` attribute, and have it select the Seam-managed persistence context, you can set the default persistence unit name on the persistence annotation `BeanPostProcessor` definition:

```

<bean class="org.springframework.orm.jpa.support.
  ► PersistenceAnnotationBeanPostProcessor">
  <property name="defaultPersistenceUnitName" value="tournament:seam"/>
</bean>

```

The `@PersistenceContext` can now be used in its most basic form:

```

public class JpaTournamentDao implements TournamentDao {
    @PersistenceContext
    private EntityManager entityManager;
    ...
}

```

There you have it! You now have seamless integration of the Seam-managed persistence context between Seam components and Spring beans. What's better is that

Spring doesn't have to worry about managing the life cycle of the persistence manager (no `OpenEntityManagerInViewFilter`). The conversation controls in Seam dictate how long the persistence manager stays open.

Using Seam-managed persistence in Spring makes it possible to work with extended persistence contexts, even though this is discouraged by Spring. The Spring documentation basically tells you to stay away from extended persistence contexts because they're used for stateful components. But you now know that stateful components are easy to create in Spring using the Spring–Seam integration! And, thanks to the low-level sharing of the persistence context, both component types end up operating on the active persistence context, relieving the database of unnecessary calls and ensuring object identity throughout the transaction, and beyond.

If you're using the native Hibernate API in your application, the persistence manager handoff works in exactly the same way. We can use the Hibernate configuration to review the mechanics of the integration just described.

15.4.3 Repeating the integration for Hibernate sessions

A crowning feature of Spring is that to move from one implementation to another, "it's just a name change." You're going to see that this is indeed the case as we move from the JPA Spring–Seam persistence integration to the Hibernate variant. The `DataSource` configuration remains unchanged, so we start by declaring the factory bean for the Hibernate `SessionFactory` in the Spring configuration:

```
<bean id="sessionFactorySpring" class="org.springframework.orm.
    ↳hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="annotatedClasses">
    <list>
      <value>org.open18.partner.model.Tournament</value>
    </list>
  </property>
  <property name="exposeTransactionAwareSessionFactory" value="false"/>
</bean>
```

Next, the Seam-managed Hibernate session is set up in the Seam component descriptor, resolving the `sessionFactorySpring` bean using the Spring-aware variable resolver:

```
<persistence:managed-hibernate-session name="hibernateSession"
  session-factory="#{sessionFactorySpring}" auto-create="true"/>
```

If the Hibernate `SessionFactory` is bootstrapped by Seam, the variable resolver bridge isn't required since Seam will have access to the `SessionFactory` in its own container.

Finally, here's the factory bean that's responsible for rewrapping the Seam-managed Hibernate Session using the `SeamManagedSessionFactoryBean`:

```
<bean id="sessionFactory" class="org.jboss.seam.ioc.spring.
    ↳SeamManagedSessionFactoryBean">
```

```
<property name="sessionName" value="hibernateSession"/>
</bean>
```

Once again, there you have it! You now have a Seam-managed Hibernate session that can be shared by Seam and Spring. Any call to `SessionFactory#getCurrentInstance()` will return the Seam-managed persistence manager. You should avoid using the `sessionFactorySpring` bean since its purpose is simply to bootstrap the persistence unit (e.g., `hibernate.cfg.xml`) and pass the runtime configuration (`SessionFactory`) to Seam to create conversation-scoped persistence managers.

While the persistence context is now being shared between the two frameworks, the transaction managers aren't aligned. Given that Seam automatically wraps each request in two-phased transactions, this can result in two isolated transactions going at once, despite only having one transactional resource! It's just as critical to share transactions between the two containers as it is to share the persistence context.

15.5 Turning transactions over to Spring

Transactions are the type of advice that everyone wants to give, yet no one can agree upon. The persistence manager (JPA or Hibernate), JTA, the EJB container (CMT or BMT), and Spring are all positioned to manage transactions, yet each has its own way of performing the task. What you need is an interface that hides the transaction platform choice and ensures that, regardless of the implementation, transactions are controlled and handled consistently. Since the focus of this chapter is on integrating Seam and Spring, this section will look at how Seam can utilize Spring's transaction manager to centralize transaction management, the motivation for doing so, and how to configure the handoff.

15.5.1 Why use Spring's transaction manager in Seam?

Both Seam and Spring offer an abstraction layer that normalizes the differences across the transaction platforms and provides a unified transaction interface and declarative controls to begin, commit, and rollback transactions. The abstraction layer also coordinates the transaction completion events (notifications) with registered synchronizations (interested parties).

If Seam is capable of providing this abstraction, why do we want to use Spring's abstraction layer on top of it? As the Spring reference manual states, "One of the most compelling reasons to use the Spring Framework is the comprehensive transaction support." Aha! Spring's transactions could very well be the reason you're attempting to use Spring with Seam in the first place.

Spring's transaction support is well integrated into the framework. You can control transactions either through abstract base classes, annotations, or AOP advice. If desired, you can keep all mention of transactions out of your Java code and declare the transaction boundaries, isolation levels, and read-write behavior entirely in XML. Spring POJOs are in no way tied to a specific transaction strategy. As far as the POJO is concerned, a JTA transaction is indistinguishable from a resource-local transaction.

But what makes Spring’s transaction manager even more compelling is that it erases the need to use JTA in scenarios where you’re dealing with a single transactional resource. Spring’s transaction manager is capable of performing the same context propagation and synchronization of transactions as performed by JTA. This independence allows you to operate outside of a Java EE container.

Finally, Spring’s `PlatformTransactionManager` supports several useful, albeit non-standard, transaction extensions such as nested transactions, transaction suspension, and other vendor-specific features. In summary, Spring’s transaction manager is just as capable as what Seam provides and may be more compelling because it’s already well integrated into your application.

That brings us back to the fact that both Seam and Spring boast a simple yet flexible abstraction layer for managing transactions, which is great if you’re using one container or the other. However, if you’re using both containers at once, as you’ve learned to do in this chapter, you still face the problem of coordinating transactions across components in the two containers.

15.5.2 *Seam’s double transaction abstraction*

The problem of having two capable solutions vying to solve the same problem is mitigated by sharing resources between the two containers, in this case the transaction manager. Spring can’t use Seam’s transaction abstraction layer because, as explained earlier, Spring doesn’t have a mechanism for tapping into Seam’s infrastructure. Seam, on the other hand, can call out to Spring. Thus, to merge the transaction capabilities between containers, Seam delegates to the transaction manager in Spring just as it would a JTA or resource-local transaction, as illustrated in figure 15.4.

As you learned in chapter 9, Seam abstracts all transaction operations behind its own `UserTransaction` interface—an extension of the Java EE standard `UserTransaction`

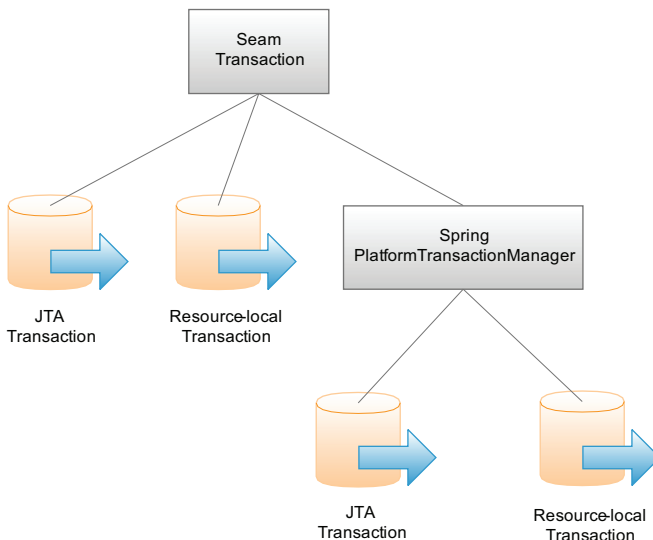


Figure 15.4 Spring’s platform transaction manager acts as an intermediary between Seam’s transaction component and the underlying transactional resource (JTA or resource-local).

interface. This interface acts as a translation layer for the native transaction platform. Earlier in this book you learned that there are implementations of this interface for the `JTAUserTransaction` and the resource-local (JPA or Hibernate) transaction. But there's also an implementation that wraps Spring's `PlatformTransactionManager`. The Spring transaction component delegates transaction operations to Spring's transaction manager, which in turn delegates the operations to the transaction platform that's configured in the Spring container, resulting in a double abstraction. In return, synchronizations can be registered with the Spring transaction manager through the Spring transaction component in order to forward transaction events back to the Seam container. This coordination is almost identical to how Seam handles CMT.

The active implementation of the `UserTransaction` interface is stored under the context variable `transaction` and can be dually retrieved using `Transaction.instance()`. By default, Seam uses the JTA transaction component, but you activate the transaction component that delegates to the Spring `PlatformTransactionManager` by declaring it in the component descriptor.

15.5.3 Activating the Spring transaction component

To switch the transaction strategy in Seam to the Spring `PlatformTransactionManager`, you declare the following component in the Seam component descriptor:

```
<spring:spring-transaction platform-transaction-manager="#{txManager}">
```

The Spring transaction manager, named `txManager` in this case, is supplied as a value expression and accessed through the poor man's integration (the EL resolver bridge). You only have to supply this value expression if the name of the Spring transaction manager bean is not `transactionManager`, which is the name Seam will look for by default.

If the Spring transaction manager uses a resource-local transaction strategy (e.g., `JpaTransactionManager`), the `join-transaction` attribute must be set to `false` on the Seam transaction component:

```
<spring:spring-transaction platform-transaction-manager="#{txManager}"
  join-transaction="false">
```

The `joinTransaction` is set to `false` in this case because resource-local transactions are controlled by the resource itself (e.g., `EntityManager`) and therefore don't have to use transaction enlistment (i.e., joining the JTA transaction). An example of a Spring transaction manager using a resource-local transaction strategy is shown here:

```
<bean id="txManager"
  class="org.springframework.orm.jpa.JpaTransactionManager"/>
```

Keep in mind that you should only use the resource-local transaction strategy if you're working with a single transactional resource (such as a single persistence unit). If you're using multiple transactional resources, or prefer to leverage the container-managed transaction service, you'll need to define the JTA transaction strategy in the Spring configuration file, shown here:

```
<bean id="txManager"
  class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

When using JTA transactions, the resource has to request to join the transaction. To ensure that happens, the `join-transaction` attribute on the Seam transaction manager can once again be excluded or set to true (the default):

```
<spring:spring-transaction platform-transaction-manager="#{txManager}">
```

For you to enlist multiple resources in a global JTA transaction, they must be XA resources (capable of performing a two-phase commit). A thorough discussion of two-phase commit transaction semantics is beyond the scope of this chapter.

Spring is adept at locating the JTA `UserTransaction` for the popular vendor environments without any special configuration. To use nested transactions or perform transaction suspension with JTA (`REQUIRES_NEW`, `NOT_SUPPORTED`), you'll need to supply the vendor-specific JNDI name of the `TransactionManager` implementation explicitly, shown here in bold:

```
<bean id="txManager"
  class="org.springframework.transaction.jta.JtaTransactionManager">
  <property name="transactionManagerName"
    value="javax.transaction.TransactionManager"/>
  </property>
</bean>
```

The `TransactionManager` interface isn't required by Java EE, even though this interface is part of JTA. However, all Java EE servers expose it as an extension to Java EE and Spring takes advantage of this fact. The way I see it, although the `TransactionManager` interface isn't part of the public Java EE API, you might as well take what you can get if you're going to go to all the trouble of using Spring.

TRANSACTION MANAGEMENT AND THE CONVERSATION CONTEXT

As you learned in chapter 9, Seam wraps each request in two global transactions. If you're using the resource-local transaction strategy and a Seam-managed persistence context in Spring, a conversation must be active before the first global transaction can be opened by Seam at the beginning of the request. This prerequisite is due to the fact that the conversation holds the resource that is managing the transaction (e.g., `EntityManager`).

In contrast, a JTA transaction isn't reliant on the presence of the transactional resource, and therefore doesn't require the conversation to be active in order to begin. Figure 9.1 from chapter 9 shows the point in the Seam life cycle at which the global JTA transaction begins compared to when the global resource-local transaction begins.

When using the JTA transaction strategy, or a resource-local transaction that doesn't use a Seam-managed persistence context, you can provide Seam with a hint that the transaction manager isn't dependent on the conversation context by setting the `conversation-context-required` property to false on the Spring transaction component:

```
<spring:spring-transaction platform-transaction-manager="#{txManager}"
  conversation-context-required="false">
```

The default value of the `conversation-context-required` is true, which will force Seam to wait until the conversation has started to begin the first global transaction.

You've tackled the two toughest integration challenges when bringing together any two frameworks: transactions and persistence. With these two critical resources being shared between the Seam and Spring containers, you should feel comfortable using objects managed by either container without fear of low-level conflicts. This integration allows you to view Seam and Spring as a single container.

15.6 Summary

This chapter set out to prove that Seam and Spring aren't mutually exclusive frameworks and that it's possible to take advantage of the strengths of each container. Spring is one of the most valuable libraries in the Java language. A commitment to Seam need not take Spring's benefits away from you.

At the most basic level, you can apply POJO integration, which treats Spring just like any other utility library, offering POJOs that can be loaded into the Seam container without having to start up a Spring application context.

Moving up the integration chain, you learned about poor man's integration, which gives Seam access to Spring beans through the use of Spring-aware variable resolvers. When that wasn't sufficient, you discovered how to create a Spring–Seam hybrid component that has knowledge of both containers, thus boasting a dual identity, by first starting the Spring and Seam containers in tandem and then enhancing the Spring bean definition with custom XML tags in the Seam namespace. This setup allows you to inject native Seam components into Spring beans and Spring–Seam hybrids into Seam components.

Although the two-way interchange of Spring beans and Seam components appears straightforward, you learned the potential for scope impedance when injecting stateful components into application-scoped singletons, the danger it can cause, and how Seam alleviates the problem by injecting lookup proxies. Even with this safety net, integrating Seam and Spring forces you to think more about the scope of components, something that's new to those who have long relied on Spring's primarily stateless behavior. While Spring 2.0 opened the door for custom scopes, Seam delivered the goods by allowing Spring beans to be scoped to Seam's rich set of contexts.

The toughest integration challenge for any two containers is learning how to share persistence contexts and transactions. In business applications, these two resources are pervasive. Without deep integration, the true substance that allows you to move freely between the two frameworks is lacking. Seam offers an entire package of built-in components to enable the sharing of these resources. Once the integration is complete, you're able to take advantage of extended persistence contexts managed by Seam while using the flexible template classes in the Spring arsenal, with transactions being stretched across the whole lot.

The Spring Framework is a tremendously useful tool in a Seam developer's toolbox. You may choose to integrate Spring beans in the runtime application or even just

for the purpose of testing. Indeed, Spring is a great way to test Java EE components in isolation because it can emulate the behavior of the container (i.e., @Persistence-Context injection). In a world rife with framework debates, it's easy to forget that all of this software is available for you to use. You now have the ability to inject collaborator components across framework lines, exercising the combined capabilities of the two containers, and utilizing the best tool for the job. *Just watch out for that scope impedance and thread safety!*

Good luck on your next Spring–Seam hybrid application!

SEAM IN ACTION

Dan Allen



Seam, an innovative Java EE framework, reinvents Java-based web development. Using simple Java objects, pre-built widgets, and very little XML, Seam's straightforward architecture and API properly manage persistence and provide a single development approach for both UI and business components. Seam works in any EE container, and its JSF-based approach makes Ajax easy to implement.

SEAM in Action is a detailed introduction to Seam for Java EE developers. In the book, you'll use seam-gen to build a basic application and follow it to learn how Seam automates non-core tasks through configuration by exception, Java 5 annotations, and aspect-oriented programming. You'll master key techniques for Spring integration, JavaScript remoting, business processes (jBPM) and stateful page flows (jPDL), and more.

What's Inside

- Get started quickly using seam-gen
- Master Seam annotations, components, and bijection
- Unlock Seam's new persistence model
- Make your applications rich (and hopefully yourself, too!)

About the Author

Dan Allen is a Seam committer, passionate open source advocate, and dedicated mentor with over eight years of development experience using Java frameworks and web technologies.

For online access to the author, code samples, and (for owners of this book) a free ebook, go to www.manning.com/SeaminAction

"*Seam in Action* captures the spirit of Seam and shows you how to use it the way we on the Seam team intended."

—From the Foreword by Norman Richards, Senior Engineer, Red Hat

"Open this book first! An excellent resource."

—Peter Johnson
author *JBoss in Action*

"Explains, teaches, and instills confidence."

—Peter Pavlovich
Sr. Software Engineer, Kronos Inc.

"... explains all technical aspects and the rationale of Seam's design."

—Michael Smolyak
Software Architect, DataSource, Inc.

"I've read all Seam books and still learned a lot from this one."

—Kevin P. Galligan
CTO, Medical Research Forum

"Essential!"

—Ted Goddard
Chief Architect, ICEfaces.org

"Easy to read, easy to understand—the best way to get going."

—Nikolaos Kaintantzisl
Software Engineer, Zühlke Engineering

