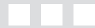
CHAPTER 5



Introduction to Seam

The preceding two chapters covered the EJB3 and JSF frameworks, which are the core components of Seam. In those chapters, you learned a simplistic way of designing both presentation and business logic. However, in order to have the JSF pages call the business logic, we had to go through JSF backing beans, the intermediate classes. Doing so often required adding code referencing the backing beans in `faces-config.xml`.

Now it is time to discuss Seam itself. In this chapter, I will show you how to eliminate the backing beans and call the EJB3's SB directly. In addition, I will also start the discussion on using Seam objects to help make common tasks simpler.

In the Figure 5-1 road map, you will see that our main focus is the Seam interception in every tier. To a lesser extent, our focus will be the EJB3 and JSF objects because they will be the target of our interception.

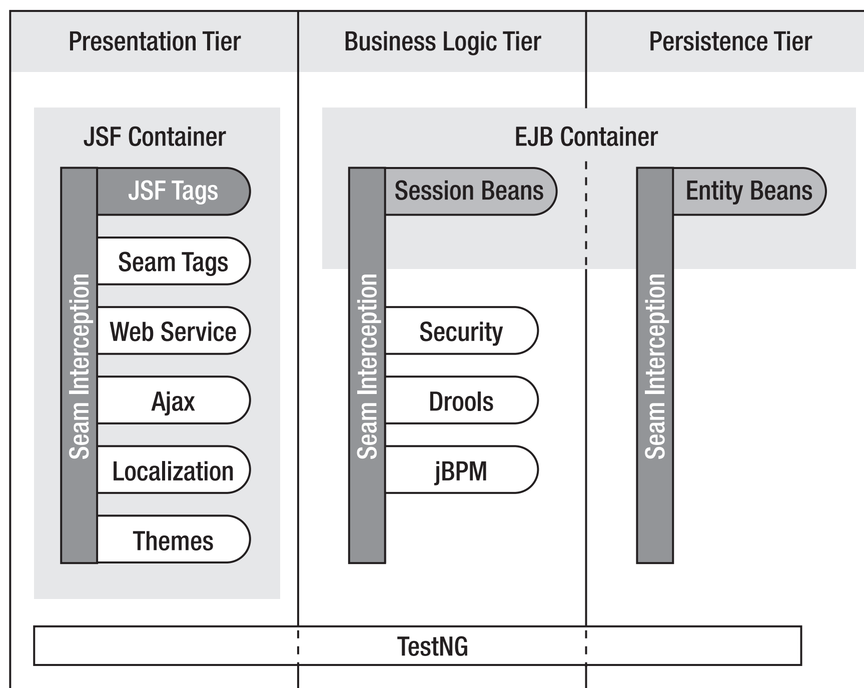


Figure 5-1. *The road map showing that our main focus will be Seam interception across the tiers*

This chapter discusses how we take the EJBs and JSF pages we created before and modify them to leverage Seam. As you will see, the end result will be less code required and an easy separation of barriers. Also, you will be able to see that not only does using Seam save time and space, but it adds functionality that we did not previously have.

The chapter starts off by explaining how to configure and download Seam. It then moves on to Seam's architecture, including an explanation of how it works and various high-level design aspects. You will learn about the injection and conversation mechanisms that Seam is known for. The chapter wraps up with a discussion of the basic components that you can use with Seam. Many of you will want to use these on a day-to-day basis.

■ **Note** Although this chapter refers only to the standard EJB3/JSF combination for Seam, there are other frameworks (for example, Hibernate) that can be used with Seam. These are discussed in later chapters.

What Is Seam?

Seam is a new application framework designed by JBoss, a division of Red Hat, to be integrated with many popular next-generation service-oriented architectures. This is achieved not by adding a heavy amount of code surrounding all the common architectures, but by sprinkling interceptors and annotations into already-existing classes. This keeps in line with the idea of using plain old Java objects (POJOs) in Java development by requiring less time for you to worry about the framework piece and leaving more time to spend actually developing the business functionality.

The obvious question you might ask is, "How does adding *more* into a potentially working model help save time?" Seam achieves this by eliminating the need for "plumbing" code. Essentially, we are allowing Seam to handle the plumbing and to have the business logic interact with the JSF pages themselves. One of the nice things about Seam is that even if you already have code you want to use, you will be able to keep your existing EJBs and JSF pages, thus being able to maintain a mixed environment if you choose. Seam does this by integrating with existing layers, as shown in Figure 5-2.

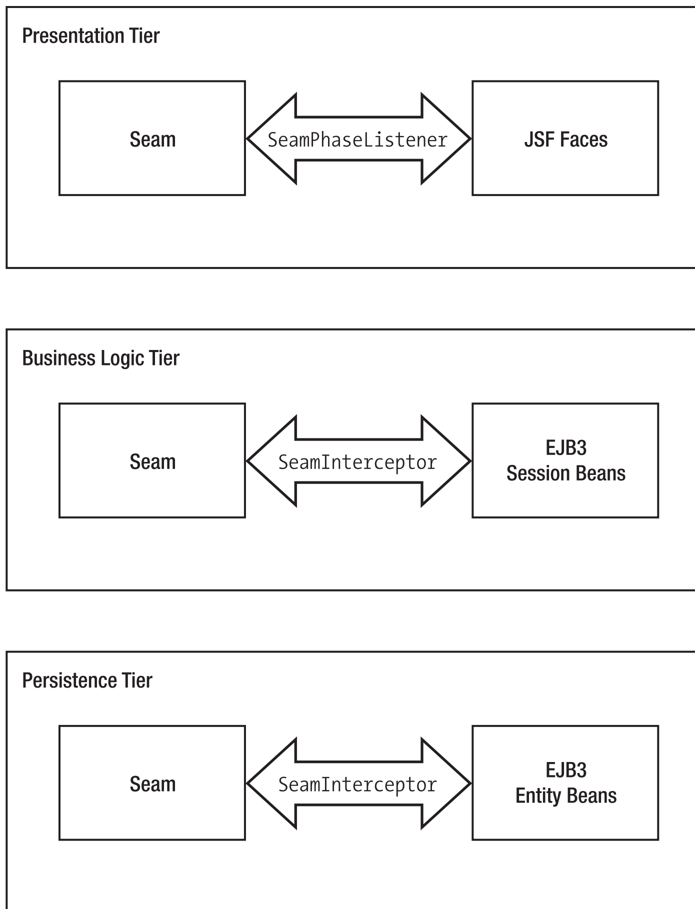


Figure 5-2. *Diagram of Seam intermixing with the various tiers*

Basic Seam Configuration

This section covers the configuration of Seam in an environment that is supporting EJB3, and the deployment of an EAR file.

Downloading Seam

Before configuring Seam, you first have to download the compressed Seam file. Seam is a product of JBoss and can be downloaded as a gun-zipped TAR file or as a ZIP file from `http://labs.jboss.com/portal/jbossseam/download/index.html`. This book uses the Seam 1.1.0 GA release.

The Seam download has many external library files associated with it, because of the large number of configurations possible with Seam. For right now, though, all you have to worry about is the basic configuration, so you will need only the `jboss-seam-ui.jar` and `jboss-seam.jar` files in the root directory of the downloaded file.

However, later in this chapter, you will also need the `jboss-seam-debug.jar` file to use Seam's debug mode.

Configuring Seam

Earlier I mentioned that there are many ways to configure Seam. For our first Seam applications, we are going to use the most basic configuration: the JSF—EJB3 configuration. This is the traditional way that Seam was designed to be set up. This minimalist configuration allows for coupling the presentation tier information to the persistence tier and will enable Seam to work with the examples in this chapter. Later in this chapter, I will discuss further modifications to make life easier in the Seam environment, and in later chapters you can add advanced options depending on the needs of your environment.

Updating XML Files

Let's begin. I will assume that you have an EAR file ready to go from Chapter 4. You will have to modify three of the existing files, one for the business logic and two on the presentation end. We will start with the `faces-config.xml` file in Listing 5-1.

Listing 5-1. *The faces-config.xml File After Adding the Seam Phase Listener*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config
PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
                            "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
    <lifecycle>
        <phase-listener>org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
    </lifecycle>

</faces-config>
```

This first modification integrates Seam with the phase life cycle of the JSF request life cycle. There are actually multiple class files to specify in the phase listener. The one you select depends on how you want to manage the transaction demarcation. I will explain the differences later; for right now, just stick with the basic phase listener.

Next we will move on to the `web.xml` file in Listings 5-2 and 5-3.

Listing 5-2. *In web.xml, Add a Context Parameter for State Saving*

```
<context-param>
            <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
            <param-value>client</param-value>
</context-param>
```

Another item that you need to add to your `faces-config.xml` file, depending on what JSF implementation you are using, is the setting of the state-saving method. If you are using Apache's MyFaces, Seam needs client-side state saving. For our initial examples we will be using MyFaces, so include the previous listing for now.

Listing 5-3. *In web.xml, Add the Listener to the Servlet Request Life Cycle*

```
<listener>
    <listener-class>
        org.jboss.seam.servlet.SeamListener
    </listener-class>
</listener>
```

In every framework that integrates with the web tier, you will have to add either a listener to the life cycle or a front controller servlet to be called by the Web. Seam solves this problem by using a listener to bootstrap the JSF servlet life cycle. The Seam listener is then responsible for moving the data across the tiers as well as for creation and destruction of context objects.

Adding Seam JAR Files

Now that you have your XMLs configured, there are two extra JAR files that you have to add to make the application work: `jboss-seam.jar` and `jboss-seam-ui.jar`. The `jboss-seam-ui.jar` file goes into the `WEB-INF/lib` directory. The `jboss-seam.jar` file belongs at the root level of the EAR file. The server will load up the `jboss-seam-ui.jar` automatically. However, you will have to specify the location of `jboss-seam.jar` in `ejb-jar.xml`, as shown in Listing 5-4.

Listing 5-4. *In ejb-jar, Add the jboss-seam.jar File*

```
<module>
    <java>jboss-seam.jar</java>
</module>
```

Finalizing the Setup

Now that you have your XML and JAR files set up, there is one final thing to do to make Seam work, and although this may seem trivial, if you do not do it you will run into a lot of errors that seem to make no sense at all. So the final step is the addition of the `Seam.properties` file to the EAR file. This file can be blank but it has to be there. I will explain later in this chapter what you can add to it.

Make these modifications to your files, compile them, and you should be good to go, ready to start using Seam. After the addition of the preceding files, your directory structure should look like the one in Figure 5-3, minus any JSP files.

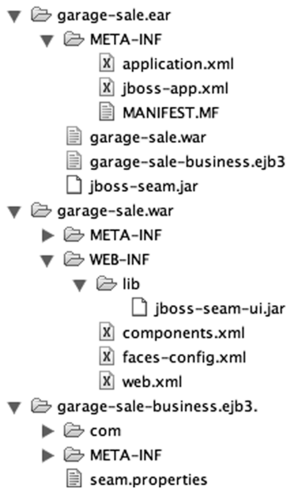


Figure 5-3. *The file structure of the EAR, EJB3, and WAR directories*

■ **Note** Seam has a generation program that provides a shortcut to get Seam up and running. For more information, consult the following websites: `http://www.integrallis.com` or the Apress website at `http://www.apress.com` in the Source Code/Download section.

First Example: Stateless Session Bean

Our first example application is the most basic one, a stateless session bean (SLSB). In this page we will be creating a SLSB that will add a `House` to the database. As I mentioned earlier, Seam does not really add any new code but takes your existing code and modifies it to integrate the business tier and presentation tier for you. So in this section, we will take the `HouseManagerAction` stateless bean from Chapter 4 and modify it for use with Seam.

For this example, we are going create a relatively simple SLSB that will add a `House` to the database by starting at a JSF page, setting properties on the `House`, and inserting the `House` into the database. In a typical web application, you would have the presentation tier call an action class, which in turn would call a utility class to do a JNDI lookup of the stateless session bean and to do a narrowing on it. At this point, you would have a reference to the home interface, and a method would be called to create the EJB remote. The EJB remote would then be brought to the action class for use, and finally back to the JSP. That is a lot of processing and essentially an extra class in the middle just to convert the request to data to call our session bean. By using Seam, we are going to do the same thing but not have what is essentially a middleman class (the action). We will modify our SLSB so that the JSF page can access it implicitly through Seam. Listing 5-5 shows an example of a SLSB class.

Listing 5-5. *Stateless Session Bean Class-Level Modifications by Seam*

```
@Stateless
@Name("salesManager")
@Interceptors(SeamInterceptor.class)
@JndiName("garage-sale/SaleManagerAction/local")
public class SaleManagerAction implements SaleManager {
    // Put the rest of the method in here
}
```

The first line of this should look familiar; it defines that this is a stateless bean. This is the only part of the class definitions that are not Seam specific.

The `@Name` attribute defines the Seam component name. This annotation is used by the JSF pages to reference the component or bean. This can of course lead to someone defining the same name in multiple beans. There is really no way to prevent this, or to specify which one will be the overwriting one. Your only indication will be a warning on the console.

The `@JndiName` annotation specifies the JNDI name that Seam will use to look up the EJB. This of course could get tedious to have in every single EJB, not to mention that the pattern could change per application server. Thankfully, there is a more global way to do this that I will explain later in this chapter.

The `@Interceptors` annotation is not a Seam-specific annotation, but we are using it for Seam. This annotation is needed for Seam to perform its bijection, validation, and so forth, by intercepting the invocation of the component. However, this is not required for EB because bijection and context demarcations are not defined. As with the `@JndiName` annotation, there is an easier way to do this that I will explain later in this chapter.

Now that you see how to start creating our Seam-modified session bean, let's go into some of the code itself. Listing 5-6 adds methods.

Listing 5-6. *Stateless Session Bean Method-Level Modifications by Seam*

```
@Stateless
@Name("salesManager")
@Interceptors(SeamInterceptor.class)
@JndiName("garage-sale/SaleManagerAction/local")
public class SaleManagerAction implements SaleManager {

    @PersistenceContext
    private EntityManager em;

    @In @Out
    private House house;

    public String addHouse() {
        em.persist(house);
        return "/homeSuccess.jsp";
    }
}
```

This is the body of our SLSB, with the necessary items in place to perform the house addition. As you can see, it does not take much code—a lot less than Struts would require. `EntityManager` is a normal component of the SLSB that I discussed in Chapter 4, so I will not explain that any further. The `House` object is an entity bean, with `@In` and `@Out` annotations. I will discuss these annotations further later in the chapter. For now, just realize that these annotations tell the Seam interceptors to populate that object and return it. Listing 5-7 shows our EB `House` object.

Listing 5-7. *Entity Bean Example*

```
@Entity
@Name("house")
public class House {

    private long houseId;
    private String address;
    private String city;
    private String state;
    private Date startTime;
    private Date endTime;
```

```
@Id @GeneratedValue
public long getHouseId() {
    return houseId;
}
public void setHouseId(long houseId) {
    this.houseId = houseId;
}

@NotNull(message="Address is required")
@Length(min=5, max=15, message="Address should be between 5 and 15")
public String getAddress() {
    return address;
}
public void setAddress(String address) {
    this.address = address;
}
public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
public Date getEndTime() {
    return endTime;
}
public void setEndTime(Date endTime) {
    this.endTime = endTime;
}
public Date getStartTime() {
    return startTime;
}
public void setStartTime(Date startTime) {
    this.startTime = startTime;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
```

```
    public String toString() {
        return houseId + ", " +address;
    }
}
```

Well, that is all that's required for the session bean to be ready to be used as a Seam component. As you can see, very little code was added, and what was added were annotations. This helps provide the robustness needed for a web application.

Now onto the changes needed by the JSF page. For reference, Listing 5-8 shows the JSF page from before with the Seam modifications needed.

Listing 5-8. *Seam Modifications to the JSF*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:view>

    <h:form>
        Please enter your address:<br/>
        <h:inputText value="#{house.address}" size="15"/><br/>

         <h:commandButton value="Add House" action="#{salesManager.addHouse}"/>
    </h:form>
</f:view>
```

You may be looking at this very hard, thinking, "I do not see the modifications." Well, you are correct. There are *no* modifications needed to the JSF page to make this work. This is the simplicity that Seam helps bring to a JSF environment. Now that being said, there are Seam-level tag libraries we can use to help make life even easier, but for this example those are not needed.

Well that's it. That's all you have to do to create your first Seam page (see Figure 5-4). In a bit we will get into some additional options for Seam and JSF pages.

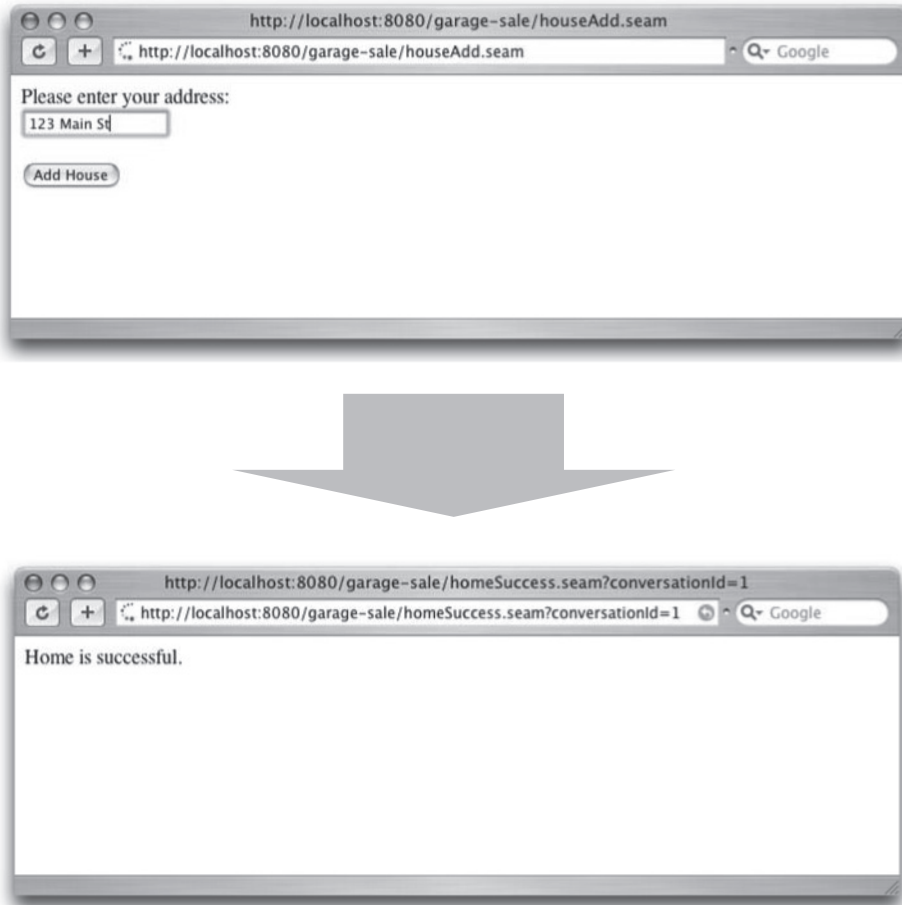


Figure 5-4. *Screen transition of entering a new home*

Architecture

The preceding example showed how easy it is to convert your EJBs and JSF pages to use Seam. As you saw, we added annotated POJOs to allow our objects to use the Seam architecture. You should find this easier than a traditional Java enterprise three-tier architecture.

However, this example raises two questions: what does Seam need with those annotations, and how does it put those together to create seamless application tiers? There are five main topics I will go over to explain how Seam works:

- POJOs and annotations
- Inversion of control and bijection

- Interceptors
- Seam context
- Three-tier architecture with Seam

POJOs and Annotations

Plain old Java objects have become the rage lately in Java development, and there is a good reason for that. They are lightweight service objects that can easily be plugged into any framework. The use of POJOs is one of the core values of many new architectures, and Seam is no different. Seam actually has no base classes to extend or interfaces to implement as do many other frameworks (think Struts and the `Action` interface). Every Seam-related object you create can be a POJO. You could of course use classes that are not POJOs, but there really is no need because POJOs give you straightforward code without the hassle of a lot of plumbing.

However, as is obvious, POJOs are not enough. Seam needs something else to at least help identify that the POJO is a Seam component. In older frameworks, XML files were used to identify these objects. However, that can get messy fast. Now, there are annotations.

As you saw in the first example in this chapter and in our Hello World application in Chapter 1, all Seam needs are POJOs—POJOS with annotations. Annotations provide a flexible way of adding functionality to your existing class structure.

Seam could not easily have been developed without this Java 5 addition. Had Seam's creators not gone this route, they would have had to base all the functionality entirely on XML-based files (which, given the number of things you can do with POJOs, would have gotten messy fast). Imagine in your XML file having to identify not only the class, but then which methods need validating, the type of validation, the time of input/output of objects—yick. So not only do annotations provide a more readable way of understanding the code, they also reduce the number of files needed. In fact, in some situations Seam uses already-existing annotations and then expands on the functionality for them. (The validation framework does this, as you will see this later in this chapter.)

This use of annotations is what sets Seam apart from other frameworks you have used in the past. This is the path that most frameworks in their next versions will be following as well—including Spring, Tapestry, and others.

Inversion of Control and Bijection

Inversion of control (IoC) is not a new concept, but a concept that until recently frameworks did not use. Frameworks such as Spring and Apache's HiveMind have made great use of it, and Seam is no exception in adding it to their repertoire.

Inversion of control, also known by many as *dependency injection*, takes the need to instantiate objects away from the normal user calls. Instead, you allow the container to handle the creation of the component and its subcomponents. Seam is no different in using dependency injection to inject objects. However, Seam takes it to the next step by allowing the injection to go both ways.

IoC is needed because of the nature of Seam. Because we don't use any JSF action classes to translate our presentation tier requests into objects and then set them on the EJB and call the EJB, we have to use IoC to make up for it. This helps us save time and space by letting the container manage the objects.

The usage of IoC in Seam is often referred to as *bijection*, because the injection is two-way (injection and "outjection"). You can specify the direction of the components. However, bijection is so much more in Seam than it is in most typical IoC patterns. Seam takes IoC and expands it by making it dynamic, contextual, and bidirectional, and allowing assembly by the container.

So where can you use bijection? Bijection can be used on any object that is a Seam object. Remember, a Seam object is any object that you have defined with an `@Name` annotation on the class. You can then biject your Seam objects into any SB or JavaBean. However, you cannot biject objects into your EB. This is because the domain model should be independent of business logic and because EBs are instantiated at the application level and Seam could not intercept them anyway.

Seam performs bijection by dividing it into two areas: one going into the system and one going out. As I have said, these are more commonly defined as injection and outjection, and we use `@In` and `@Out` for these, respectively.

You got a brief taste of bijection in the first example. Now let's take a look at what exactly is happening and the options on those annotations. In this example, we have an EB Seam object called `House`:

```
@In @Out
private House house;
```

This code specifies that your EB `House` will be a variable that can be injected in and out of the SLSB and back to the JSF page.

The `@Out` annotation has the following parameters:

`required`: This parameter specifies whether the field coming in/out should be created. By default, this is set to `true` and will automatically be created, or you can specify `required=false` to not have it automatically created.

`scope`: This is used to specify the scope of a non-Seam component. This is not necessary if you are referencing objects that you specified with a Seam name. This is more for objects such as strings, lists, and so forth.

`value`: This is the context variable name, thus the name you are going to use when referencing this component either in other beans or in your JSF page. This will default to the name of the field or the getter/setter method.

The `@In` annotation will have all the parameters of `@Out` as well as the `create` parameter:

`create`: This parameter specifies that a component should be created if the context variable is `null`.

■ **Note** You cannot do bijection directly in an interceptor. You would have to call the objects through other means in the interceptor.

BIJECTION WITH STATELESS SESSION BEANS

One thing that should be noted before we continue is that the bijection in Seam is actually very complex (in a good way). Case in point: how we deal with stateless session beans. If you have worked with them in the past, you are used to a simplistic model in which you send data to the SLSB, it performs some operation, and then you return data. You never have the SLSB access property-specific data on the SLSB. It is not safe to do so with SLSBs, unless the property is a global object such as a database connection or logger reference.

However, with Seam this methodology is thrown out the window a bit. Seam does not use a simplistic creation-time IoC. Instead the bijection happens at invocation. Therefore, the objects you are setting with `@In` or `@Out` to be injected or outjected are done at invocation, and when the method is complete, they are disinjected. This is what makes it safe for us in Seam to reference properties of the SLSB inside the methods.

Of course, along with this plus there's a minus. These SLSBs now become somewhat Seam specific, because you would never want to use them in a regular EJB3 container because their behavior would be different. In a non-Seam container, you would not be guaranteed that the properties set on the SLSB would then be the same SLSB that you are calling the method on.

Interceptors

So far, I have discussed the POJO objects that allow you to create both business and domain objects that Seam applies its functionality to. I also discussed using annotations for decorating the POJOs, indicating where we want to apply Seam functionality. So now the question becomes, “How does Seam use these annotations to provide functionality?”

After all, there are no parent interfaces or any classes that get called directly. The answer is, through a combination of interceptors. These interceptors are called directly via the code and the configuration file, or indirectly by classes used by existing interceptors.

There are many ways to wrap calls to the interceptor. In Seam we use Java EE's interceptor classes to wrap the POJOs. The interceptor classes use the `javax.interceptor.AroundInvoke` and `javax.interceptor.InvocationContext` annotations for creating the methods that should be called during that interception.

Seam Contexts

Previously I explained how IoC works and its use in the Seam framework. Now part of IoC is the life cycle of the injected object—obviously, it would be bad to have these objects around forever or to have them deleted at each request. This concept is the same with the POJOs themselves; they need to be kept around for sometimes more than one request. The few built-in contexts into the Servlet specification (Request, Session, Servlet context) hardly match everything you could need. Hence Seam has come up with a few more contexts to help you out. I cannot cover them all in detail in this chapter; however, you will see examples of all of them in this or later chapters.

So what exactly are contexts and why do you need them? Any experienced web developer or avid web surfer knows the need to maintain his website as the pages go on. If you are filling out a multipage loan form, you obviously want the information you stored on the first page to go to the second page and so on. You especially do not want to have that information destroyed because you tried to open a new browser window. Or, if you are a system administrator, you might want only certain users to have access to each page. As an even more-advanced example, you may want to have someone interact with the information you are using. You may need to have one person create a request and one person approve it. There are many of these concepts that web developers encounter on a daily or weekly basis.

On most traditional web applications, we handle these contexts by storing information in the request or session objects. Sometimes you have to get tricky and use tokens to prevent the user from using the Back button and refilling in data or using new browser windows. There are many different tricks to the trade. Well, Seam realized this and basically said enough is enough, and instead of forcing developers to add tricks ad hoc by using the request and session objects, just created new conversation states to use.

The following are the contexts available in Seam.

Stateless: The stateless pseudo context.

Event: This is more commonly known as a request scope. This will last the length of a single server request.

Page: This is the culmination of requests for a single Faces request. It will start when you invoke the action to take you to the page, and it lasts until the end of any action invoked from the page.

Conversation: This is for use on a series of requests from the same browser window that all are related to the same topic or conversation. A typical example is a wizard application for a loan.

Business Process: This is used for process management with tools such as jBPM. Applications that require multiple actors to use the same set of items in a process is an example. Chapter 7 covers this topic in greater detail. This context will span multiple conversations with multiple users.

Session: This is a traditional Session context.

Application: This is a traditional Servlet context.

How to Define Context Scope

Before I explain details about each context, you need to know how to set your objects to be those contexts. With Seam they refer to the context of an object as its *scope*; think of it as *the scope of the context*. It is pretty simple. You can define the scope either on your POJO itself, or on the attributes of the POJO by using the scope reference, as in Listing 5-9.

Listing 5-9. *Example of Using Scope*

```
Import org.jboss.seam.ScopeType;

@Scope(ScopeType.Stateless)
public class MyBean {

@Out(scope = ScopeType.Stateless)
String name;

}
```

As you see, we have defined the bean itself as a stateless scope, and we have defined the outjection of `name` in a stateless scope as well. Under normal circumstances, you can easily define just one, not both.

Now let's delve into more details of each scope type.

Stateless

Stateless contexts are for contexts that are, as you may have guessed, stateless. There really is not much to discuss or show. These are used for a totally stateless object such as a stateless session bean.

Event

For most of your requests that go from one page to another page, this is the context you will be using. This is considered the "narrowest" context. Event contexts are held for one cycle of one request. This can be as simple as going to a list page or just loading the home page.

Page

The Page context is a relatively simple concept; it is when a component is tied to one particular page. Conversely, the page then has access to all the components that referenced it. You will want to use this on pages where you need to persist the components upon multiple subsequent calls to the same page.

Conversation

Have you ever been to a website that requires multiple form submissions and wanted to try multiple scenarios? Suppose you are on a vacation-planning website and you have to go through multiple selections—for example, selecting your airplane travel, hotel, car rental, and maybe even meals. Well, you are not sure what option you want to pick. So you open up your tab-allowed browser such as Mozilla's Firefox and create multiple travel plans. As you may have experienced, there can be one big problem: the site uses session-scoped data and now some of those sessions are overlapping. So you will get a final page with data from different parts. Now there are ways around this, but they have to be programmatically included in the code. That can be a pain.

As a solution, Seam has come up with this idea of a *conversation*, also known as a *workspace*. Conversations are essentially the context every call will initially be in. Most normal conversations will last the duration of one call. However, with Seam we can upgrade these conversations to long-running conversations. Once upgraded, these conversations will last multiple page calls, and in fact will be tied to `conversationId`. Creating the long-running conversations is basically a way of using the same named context data multiple times. Think in terms of having an `HttpSession` object with a name, but then if you wanted to start a separate path in the system, you could re-create it again without losing your data from the previous session. Also, another great feature is that the user can access any of these conversations on any of

the pages. So let's say that when you were booking your travel, you could still display data from your other selections on that page—for example, showing the total price thus far.

Business Process

The business process mechanism is a way of managing interactions across multiple screens and multiple users all joined together. Because Seam is a JBoss product, it is only natural for it to use the JBoss jBPM (Business Process Manager). Explaining how to use and set up jBPM is too complex for this chapter. However, we will delve into it more in Chapter 7.

Session

This is like a traditional `HttpSession` object. This is necessary when you have SFSBs that need to have their data persisted across multiple session requests. For example, often you will have a request for a list page that you want to make changes to, and you may not want to lose the persistable list data. So you store the data to the Session context, and the data will be persisted until the Seam contexts are destroyed.

Application

This is like the traditional `ServletContext` holding information that is available to all the users. Because this is like the Servlet context, there is no tracking of individual web requests from specific users.

Three-Tier Architecture with Seam

So now that we have all of the major players in place, it is time to put this all together into a functioning Seam application. In this section, I am going to explain how Seam with its interceptors defines the three-tier architecture. This interception for the most part will be transparent to the user. Figure 5-5 presents a diagram of this interception.

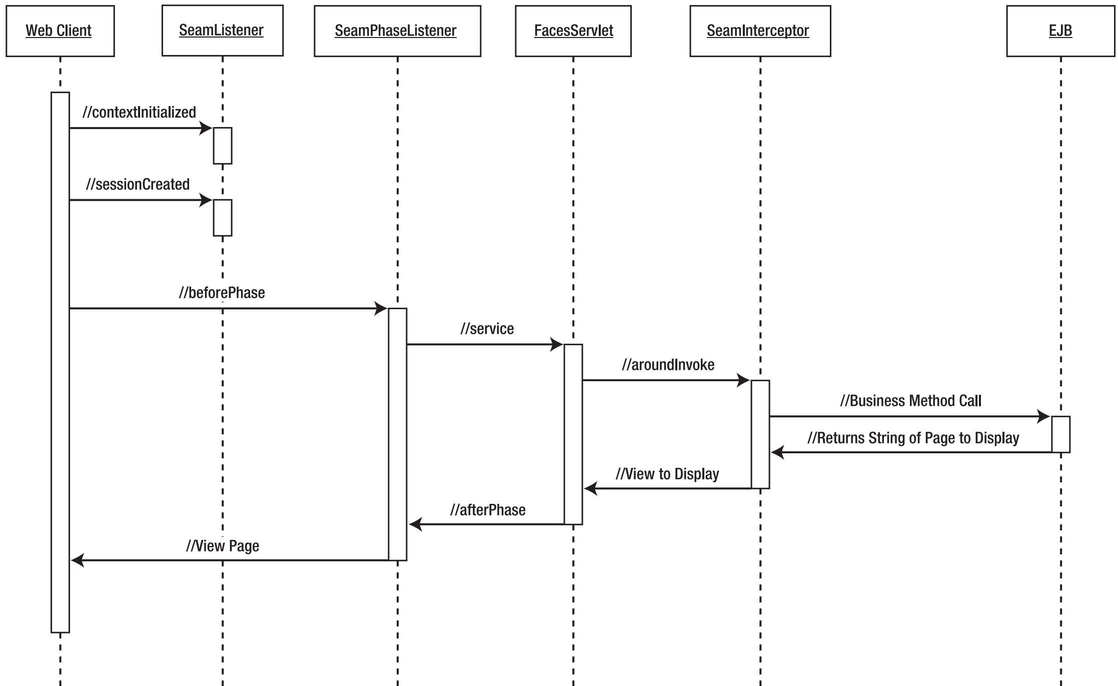


Figure 5-5. *The sequence diagram of an initial full life-cycle call with Seam*

Seam's Integration with the MVC

Most frameworks integrate directly by having you call their framework-specific servlets to integrate with the architecture. Seam is different; it controls items by adding listeners and life cycles into the request. This allows us to maintain the normal life cycle of a JSF request. You saw this already earlier in the chapter when I presented Seam's basic configuration. Here I will explain it in a bit more detail.

Before I start explaining how Seam integrates with the various areas, you need to be aware of a central class: `org.jboss.seam.context.Lifecyle`. This is the class that will keep our contexts straight. Contexts will handle state in the web tier in a more advanced way than a standard request and session object.

As you may recall, in the `web.xml` a listener (`org.jboss.seam.servlet.SeamListener`) was added. You also see this listener being the first thing called in our sequence diagram. This listener is called only at the start of a new session. This will set `ServletContext` and `Session` to the `Lifecycle` object for manipulation later.

Now you see the next object called is `SeamPhaseListener`. The `SeamPhaseListener` object is called in connection with `FacesServlet`. As you saw in `faces-config.xml` earlier, the `SeamPhaseListener` is part of the life cycle for `FacesServlet`. This again is used to control much of the context and to store the request state. This listener is needed to help move

the data from the presentation to the business logic tier. In a typical JSF life cycle, you would use backing beans. Now instead of having to worry about your backing beans needing to translate the data and call EJBs, Seam will handle this directly for us.

Seam's Integration with the EJB3

Figure 5-6 represents the integration of Seam with the EJB3 POJO.

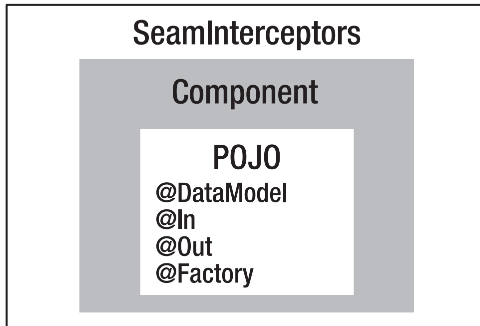


Figure 5-6. *Representation of the layers of the EJB POJO*

So now that you understand the web tier's integration, you're ready to learn about the business logic tier EJB integration. The business logic tier integration uses `SeamInterceptor` to wrap around its call to the EJB. This interceptor wraps around any call to the POJO EJB. This indirectly accesses a `Component` object stored in the Application context, all the while wrapping the call in `Lifecycle`. This `Component` object contains in it the EJB that you are accessing. It is much more than a simple wrapper object. This is the object that holds all the field-level objects—from the injected fields, to the data model objects, to the validators—basically, anything associated with the POJO's attribute-level Seam annotations. Also, another big extra is the inclusion of additional default interceptors. These interceptors handle everything from bijection to validation to transaction management. Additionally, you are able to add your own interceptors into the POJOs or even additional default interceptors. The interceptors are defined in the `org.jboss.seam.interceptors` package.

Put It All Together

Putting it all together, you have the ability to initiate calls from the JSP page to the EJB without having to worry about the middle-management bean. At the same time, you are also not losing any functionality.

JSF PAGES

I have discussed the changes to the business logic tiers. Now I want to briefly mention the presentation tier. All the JSF tags you are currently using can be kept without any modifications. There are some custom tag libraries from Seam; in general, these are tied to components used on the business logic tier (for example, `DataModel`). Throughout this chapter and the rest of the book, we will be using a mixture of JSF tags and Seam tags for our pages.

Components

Hopefully by now you have a very basic understanding of using Seam with the code you have created in Chapters 3 and 4. This section covers additional tools to help you create your Seam pages, including logging and debugging. In addition, I am going to introduce Seam-level components that will help make your pages more robust—for example, data models and validation components. By the end of this section, you will know enough to start writing even more-complex Seam pages.

Seam Configuration Options

As you may have noticed, these JSF pages tend to get a lot of annotations at the class level, and this can get quite messy. What is worse is that some of those annotations are quite repetitive. The interceptor is required on every single page that you want to become a Seam page. In addition, the JNDI name has to be defined on each page, and worse yet, the JNDI name is application server–specific. So take a look at the following code in Listing 5-10.

Listing 5-10. *The Standard Code We Have Been Using for Our Classes*

```
@Stateless
@Name("houseManager")
@JndiName("garage-sale/HouseManagerAction/local")
@Interceptors(SeamInterceptor.class)
public class HouseManagerAction implements HouseManager {
    // Add the rest of the code here
}
```

By modifying two configuration files, we can eliminate having to use `@JndiName` and `@Interceptors`.

JNDI Name

You can define the JNDI name in `components.xml`, which is placed in the `WEB-INF` directory. You are going to use a wildcard expression for the pattern of the EJB name. Listing 5-11 contains an example of defining a `jndiPattern` to be used for JBoss. Consult your documentation if you plan to use a different application server.

Listing 5-11. *Our components.xml file with the JNDI Name Pattern*

```
<components>
    <component name="org.jboss.seam.core.init">
        <property name="debug">true</property>
        <property name="myFacesLifecycleBug">true</property>
        <property name="jndiPattern">garage-sale/#{ejbName}/local</property>
    </component>
</components>
```

Seam Interceptor

There is a rather easy way to define the Seam interceptor, and unfortunately it uses a file that you may have thought you had gotten rid of: `ejb-jar.xml`. Because we define most EJB configurations now in annotations (optionally still definable in `ejb-jar`), we use a new attribute for EJB3: `interceptor-binding`. Here we just define `SeamInterceptor` for all the EJBs. Of course, if you wanted Seam to work only on a specific subset, you could specify that in Listing 5-12 as well.

Listing 5-12. *The ejb-jar.xml File with the Seam Interception*

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
   <interceptors>
     <interceptor>
       <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
     </interceptor>
   </interceptors>
```

```
   <assembly-descriptor>
      <interceptor-binding>
         <ejb-name>*</ejb-name>
         <interceptor-class>
              org.jboss.seam.ejb.SeamInterceptor
         </interceptor-class>
      </interceptor-binding>
   </assembly-descriptor>
</ejb-jar>
```

So from now on in the examples, assume that we are using these two optional configurations.

Logging

One good habit that many, from beginners to experts, forget is proper logging. Logging is such a core fundamental to good programming that I wanted to include it sooner rather than later. One of the issues some people have is which logger to use. The main two are Log4J and Apache commons-logging. Another issue with logging is how to properly do it. Listing 5-13 shows an example of logging.

Listing 5-13. *Logging by Using LogFactory*

```
private static final Log log = LogFactory.getLog(SaleManagerAction.class);

public String addHouse() {
    if (log.isDebugEnabled()) {
        log.debug("House address to add is "+ house.getAddress());
    }
    em.persist(house);
    return "/homeSuccess.jsp";
}
```

That is the proper way to write a debug or info log message. Not only does creating the debug statement add many lines, but many developers (even experienced ones) screw up the instantiation of it. Often they will forget to declare the `Log` static or have to change the class name and forget to change it on the logger as well. This can cause all sorts of problems. Also, when debugging, you use `isDebugEnabled` mainly because creating the string that is accessing the object requires processing time, so often people forgo using `isDebugEnabled` and just use the debug statement.

So as you can see, there is a lot of extra effort here and room for mistakes. Fortunately, Seam realized this in advanced and used annotations and scriptlets, which

has made logging much simpler. Listing 5-14 shows the previous code written with Seam logging.

Listing 5-14. *Logging with Seam*

```
@Logger
private Log log;

 public String addHouse() {
    log.debug("House address to add is #{house.address}");
    em.persist(house);
    return "/homeSuccess.jsp";
}
```

Well, look at that—not only did we cut down on code, but we made it simpler. Now the obvious question is, "Did we lose any of the functionality?" The answer is no. Seam uses Apache commons-logging for its logging behind the scenes. The annotation itself then defines the static log element to be used in the page. Also, you notice that `isDebugEnabled` is no longer there either. This happens because, as you can see, the `debug` method is not retrieving the address via string concatenation of the object. So this way, the object is not being forced to resolve until well within the debug method; hence the `isDebugEnabled` method is no longer needed. All in all, this is a smooth way to perform debugging. Try adding some debug statements to your code now and check the `server.log` file for the output.

Debug Mode

Another nice feature of Seam is its *debug mode*. This idea of having easily debuggable abilities via the container is something that is catching on. Tapestry has an error page that details the error and stack trace and all the variables associated with the request. This is a bit different. Both are displayed on a web page built into the debugging software. With Seam, you merely go to an independent web page that shows you all the information. This is more for debugging complex problems that arrive with an application of this nature. As I said before, there are many Conversation contexts to use, so consequently knowing all of them and keeping track of them can be tricky. So what does the debug mode tell you? It tells you the following:

- Conversations
- Component
- Conversation context
- Business Process context

- Session context
- Application context

These are all associated with the current session you are in—which makes sense, because you would not necessarily want to see all the sessions. Figure 5-7 shows the debug screen.

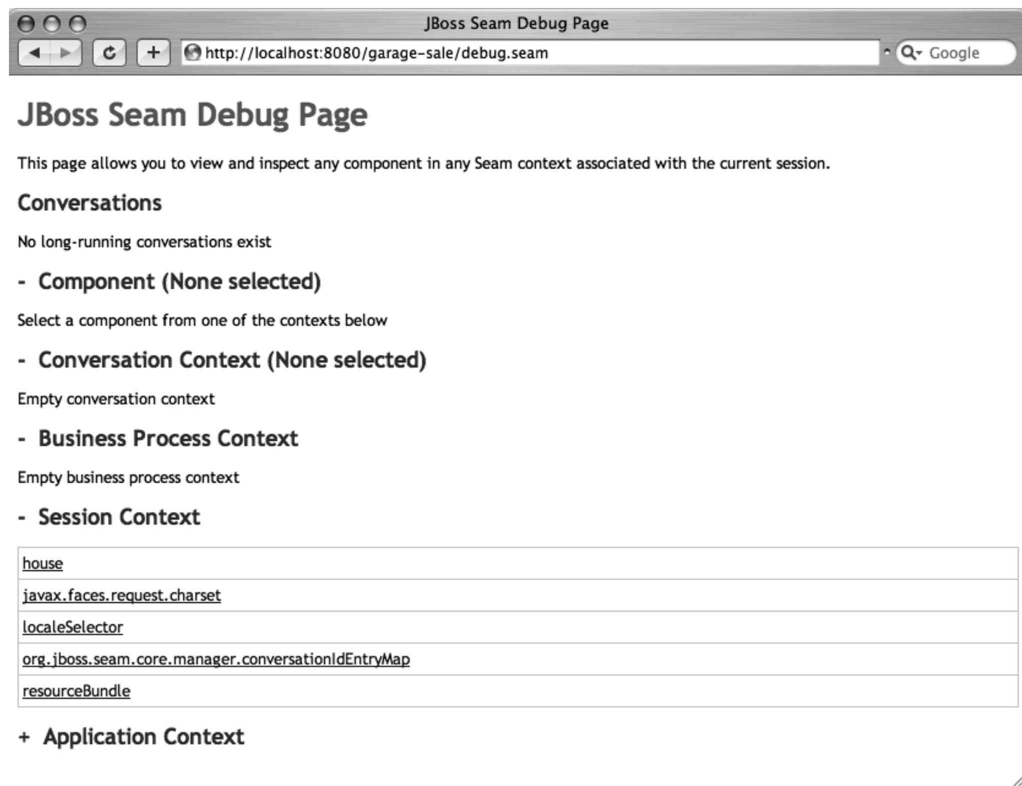


Figure 5-7. *Screen shot of Seam in debug mode*

How to Make This Work

Well hopefully you are thinking, "Wow, this is neat! How do I do it?" Well, it is quite simple. It requires one change to your `web.xml` file and the addition of four other JARs. The debug addition to `web.xml` is shown in listing 5-15.

Listing 5-15. *Add This to web.xml to Turn On Seam Debugging*

```
<context-param>
    <param-name>org.jboss.seam.core.init.debug</param-name>
    <param-value>true</param-value>
</context-param>
```

The JAR files are located in two areas. First, from the base of the download, you will find `jboss-seam-debug.jar`. You will take all the JAR files from the `facelets/lib` directory as well. Figure 5-8 shows the directory structure.

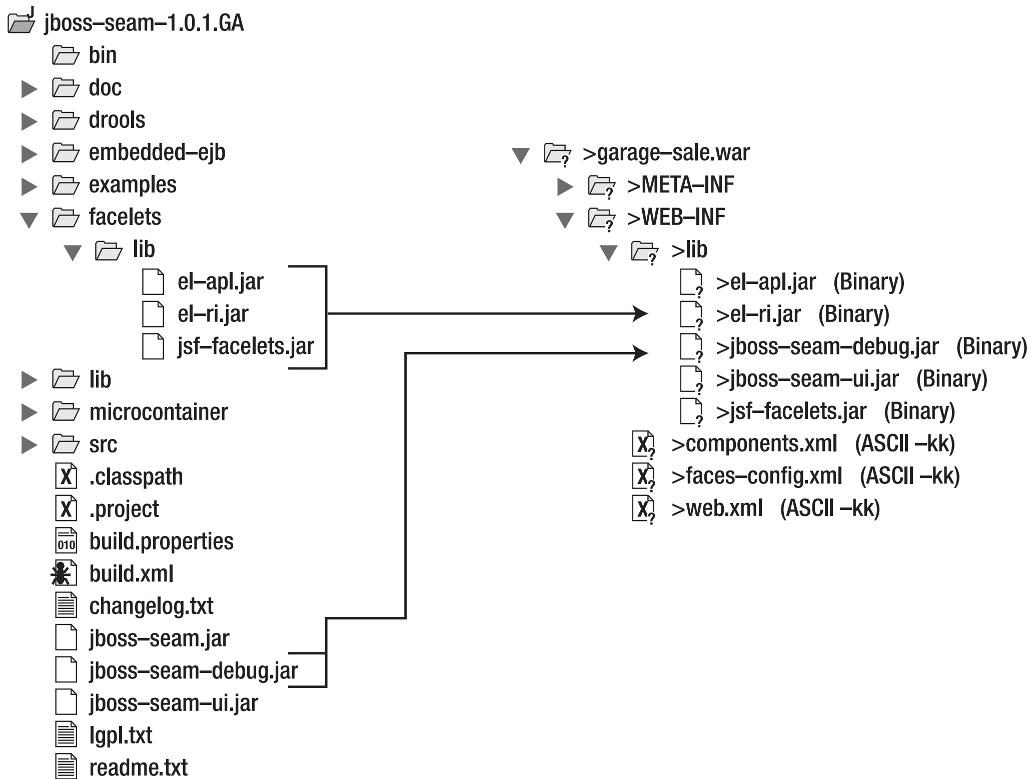


Figure 5-8. *Diagram of where to add the extra JAR files from and to*

Now all you have to do is open a separate web browser in the same session and call `http://localhost:8080/garage-sale/debug.seam`. The `localhost:8080` represents where you configured the server to run, `garage-sale` is the context root, `debug` is specified by default, and `.seam` is what you specified in the URL pattern for your Faces servlet mapping in the `web.xml` file. Listing 5-16 defines the additional entry for the URI.

Listing 5-16. *Web Configuration for the application.xml File*

```
<web>
    <web-uri>garage-sale.war</web-uri>
    <context-root>/garage-sale</context-root>
</web>
```

Listing 5-17 provides the definition for the web suffix.

Listing 5-17. *Configuration for web.xml*

```
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

Data Model

The *data model* is a useful set of Seam annotations and JSP tag libraries for processing lists on the presentation tier. Quite often you will have a list of items for a page and you will need to either edit parts of the list or delete parts of the list. This is actually quite simple. You get a list of items from the database. Then you display the list on the presentation tier with a link and an ID. When you click the link, you are taken back to the action page, where you can use that ID to either look up the item from the list or from the database. This is a basic operation that happens throughout many web applications the world over. This tends to become a routine, cumbersome process. Fortunately, Seam has some components to help you with it.

For our example, we will display a list of addresses and a Delete button for each to remove them from the database.

Seam has made this much simpler by adding framework pieces into an SFSB that makes life easier for the developer. All you have to do is specify three items in the SB. Specify the item that is the list, the individual item you want selected to be injected, and the factory method to instantiate the list.

In this example, we want to get a list of houses back and be able to edit them one at a time. The page will display with a list of text boxes that each have a Delete button. The user can hit the Delete button and delete any address desired. This code is built onto the same Garage Sale application we used earlier. Let's start off with the SFSB, which is a new item we have not discussed before.

POJO Service

Listing 5-18 shows our stateful session bean.

Listing 5-18. *The SFSB for Our Edit Action*

```
@Stateful
@Name("houseManagerEdit")
@Scope(ScopeType.SESSION)
public class HouseManagerEditAction implements HouseManagerEdit {
   //...
}
```

This style should look fairly familiar by now for defining our Seam `Session` objects. The new part this time is the `@Stateful` annotation, which of course denotes that we are using an SFSB.

The `@Scope` annotation is a Seam-specific annotation that sets the context for binding the instance of the POJO. In this case, we are binding to the Session context. The combination of making it stateful and setting the scope to `Session` is necessary to allow us to persist the list objects, so when we go back to the server we know which object of the list that the user selected. If we did not persist this in `Session`, the list would be lost and we could not persist it.

Now that we have our code to create the bean, let's add the guts of it, which gives the real functionality. We are going to define two objects to use: `DataModel` and `DataModelSelection`. The `DataModel` object will represent the list of items, and the `DataModelSelection` object will represent the selected item. Listing 5-19 displays the `DataModel` selection.

Listing 5-19. *The DataModel Selection Example*

```
@Stateful
@Name("houseManagerEdit")
public class HouseManagerEditAction implements HouseManagerEdit {

@PersistenceContext
private EntityManager em;

@DataModelSelection
@Out(required=false)
private House house;

@DataModel
private List<House> houses;
```

```
@Factory("houses")
public void findHomes() {
    List list = em.createQuery("From House hs order by hs.houseId").getResultList();
    houses = list;
}
}
```

@DataModel is a Seam annotation that allows us to use java.util.Collections from the EJB in the JSF <h:dataTable> tag on the front side. This is achieved by having Seam convert the list into an instance of javax.faces.model.DataModel. This can be a powerful component for use in an SFSB by allowing us to select and bring back objects into the bean for processing. However, the question remains, "How does the page know to populate that object?"

The @Factory annotation comes into play in telling the page how to initialize the houses bean. Seam uses this to tell it to instantiate the houses object and run this method when the presentation tier requests the houses object. As you can see, this method then sets our houses object by using a simple query from our entity bean. Note that this method will be called upon each request to it, regardless of whether you have a stateless or stateful bean.

The @DataModelSelection annotation tells Seam that this is the object to be injected from the list that is selected on the presentation tier. This object then can be used in your method that wants to use it and that is called from the JSF page. Listing 5-20 shows the method for performing the deletion.

Listing 5-20. *SFSB Delete*

```
public String remove() {
    em.remove(house);
    return null;
}
```

Here is our method we will call from the JSF page that will call for the deletion of the house from the database.

Listing 5-21 shows our change to make the persistence context extended.

Listing 5-21. *The Change of Our Persisentece Context*

```
import static javax.persistence.PersistenceContextType.EXTENDED;

@PersistenceContext(type=EXTENDED)
private EntityManager em;
```

We change `EntityManager` to `type=EXTENDED` in order to give it an extended persistence context. This keeps our house list in a managed state, and therefore subsequent calls to the page will not need to requery the list to get the full data.

JSF Presentation Tier

Now that we have covered the business logic object, let's go over the JSP that is going to display the list of houses in Listing 5-22.

Listing 5-22. *The JSP to Display the List of Addresses*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>

<f:view>

    <h:dataTable var="house" value="#{houses}" rendered="#{houses.rowCount>0}">

        <h:column>
            <f:facet name="header">
                <h:outputText value="Address"/>
            </f:facet>
            <h:outputText value="#{house.address}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Action"/>
            </f:facet>
            <s:link value="Delete" action="#{houseManagerEdit.update}"
                        linkStyle="button"/>
        </h:column>
    </h:dataTable>
</f:view>
```

This JSF is doing quite a bit here. As you can see, we use the `dataTable` discussed in Chapter 3. However, we introduce our first use of one of the Seam tag libraries. This creates a button link that will reference the specific list item. When the user clicks on this tag library, the page will call the SFSB, setting the `house` property with the item from that row selected. The action defines the name of the SFSB and the method on it to call. The method is a simple delete, as you would have had in any standard SFSB. This JSF page rendered with example data will produce the output in Figure 5-9.

Figure 5-9. *The display of our JSF page*

Bringing this all together, you should be able to see how easy it is to use Seam to create a presentation tier that display lists and allow us to perform operations on each individual item in that list.

Validation

Validation is an intricate part of any application, especially a web application. Unfortunately, the EJB3 specification alone does not contain any validation specs. However, the Hibernate framework does. The Hibernate validation framework is already part of JBoss, so we are able to easily use it. Figure 5-10 steps through our validation process for the garage sale house addition.

This is the validation for Seam, making use of the Hibernate validator. It is a relatively simple concept. You apply the Hibernate `@Valid` annotation to any attribute that needs to be validated on the POJO. If the action method you call from the JSF page has the `@IfInvalid` annotation on the method, Seam will attempt to validate all those Hibernate validations. If any of the validations error out, a message for the JSF page will be displayed. Now let's dive into the code changes.

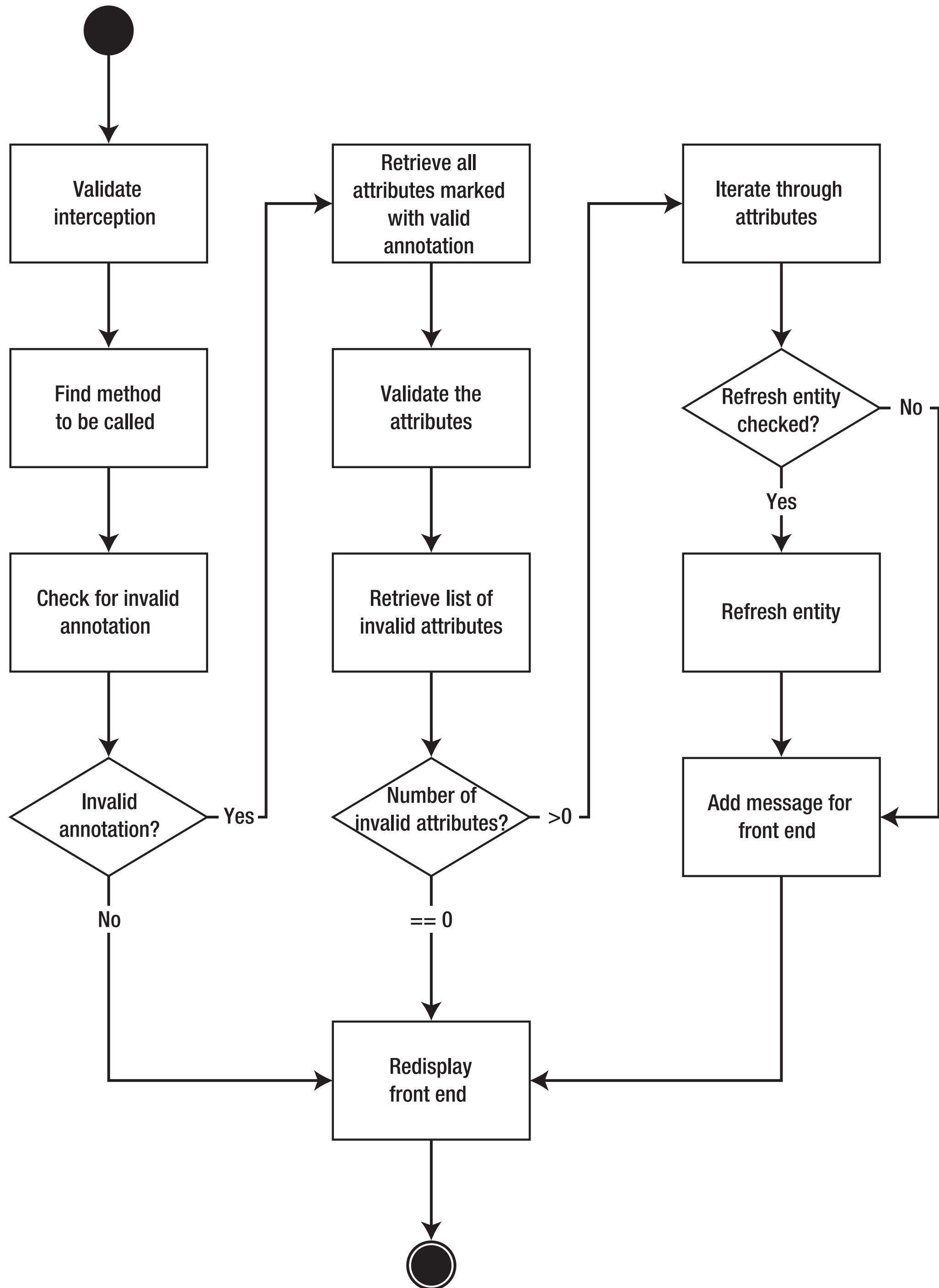


Figure 5-10. *Activity diagram of the validation mechanism in Seam*

Validation on the Domain Model

The validation specification works by allowing validation annotations on the domain-level objects. Because we are using a full EJB cadre for our framework, the EBs would be the domain-level objects. Thus, in order to implement the validation, all you would have to do is annotate the EB as in Listing 5-23.

Listing 5-23. *An Example of an Entity Bean with NotNull Validations*

```
import org.hibernate.validator.NotNull;

@Entity
public class House implements Serializable {

    private static final long serialVersionUID = -3823531857349759805L;

    private long houseId;
    private String address;

    @Id @GeneratedValue
    public long getHouseId() {
        return houseId;
    }
    public void setHouseId(long houseId) {
        this.houseId = houseId;
    }

    @NotNull(message="Address required")
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

As you can see, it is a fairly simple process. Just attach the validation on the getter of the value that you wish to validate. This then allows the framework to be able to determine whether this is a valid object. Table 5-1 shows the possible Hibernate validations.

Table 5-1. *Validation Annotations in Hibernate*

Annotation	Description
`@Length(min=, max=)`	Checks whether the string length matches the range
`@Max(value=)`	Checks that the value is less than or equal to the max
`@Min(value=)`	Checks that the value is greater than or equal to the min
`@NotNull`	Checks that the value is not null
`@Past`	For a date object, checks that the date is in the past
`@Future`	For a date object, checks that the date is in the future
`@Pattern(regex="regexp", flag=)`	For a string, checks that the string matches this pattern
`@Range(min=, max=)`	Checks whether the value is between the min and the max
`@Size(min=, max=)`	For collections, checks that the size is between the two
`@AssertFalse`	Asserts that the evaluation of the method is false
`@AssertTrue`	Asserts that the evaluation of the method is true
`@Valid`	For a collection or a map, checks that all the objects they contain are valid
`@Email`	Checks whether the string conforms to the email address specification

You will notice that in Listing 5-23 we also added a `message=` parameter to our `@NotNull` annotation. This allows us to pass a message back to the presentation tier specifying the error message.

Calling the Validator from the Business Tier

We have gone over how to define the validation on the domain objects, but have not yet gone over how to tell Seam when to validate the domain objects. The Hibernate validator can be called from different layers, thus making it so we can perform checks on the domain bean from layers not near the validator. This will work well for our needs.

The validation on the business logic tier is accomplished in two steps. First, you define what properties should be validated, and then you define which methods will trigger validation to be performed. Listing 5-24 defines the validation.

Listing 5-24. *A SLSB That Will Validate the House Before Adding It*

```
import org.hibernate.validator.Valid;
import org.jboss.seam.annotations.Outcome;

@Stateless
@Name("salesManager")
@Interceptors(SeamInterceptor.class)
@JndiName("garage-sale/SaleManagerAction/local")
public class SaleManagerAction implements Serializable, SaleManager {

    private static final long serialVersionUID = -5814583678795046052L;

    @PersistenceContext
    private EntityManager em;

    @Valid
    @In @Out
    private House house;

   public String addHouse() {
        em.persist(house);
        return "/homeSuccess.jsp";
    }
```

The `@Valid` annotation tells Seam which object needs to be validated. This annotation is not Seam specific; this is another Hibernate validator that Seam uses to determine that this object needs to be validated.

■ **Note** Earlier versions of Seam used the `@IfInvalid` annotation. This has been semi-deprecated in favor of using `<s:validateAll/>` in the JSF page.

Validation on the JSF Pages

So now you know how to set the domain objects to validate and how to trigger the validation on the business tier. Now let's go over how to add the code to our JSP to display the validation error. Listing 5-25 shows our modified code with validation.

Listing 5-25. *The JSP for Our houseAdd.jsp*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>


<f:view>
   <h:messages/></div>

    <h:form>
        <s:validateAll>
        Please enter your address:<br/>
        <h:inputText value="#{house.address}" size="15"/><br/>
        <h:commandButton value="Add House" action="#{salesManager.addHouse}"/>
        </s:validateAll>
    </h:form>
</f:view>
```

The only major addition you can see is the `<h:messages/>` tag, which will allow the display of any errors that occur. The Seam `<s:validateAll>` tag has also been added, to signify that these elements need validating.

Figure 5-11 shows the display that occurs when a user tries to add a house that doesn't meet the minimal validation rules.

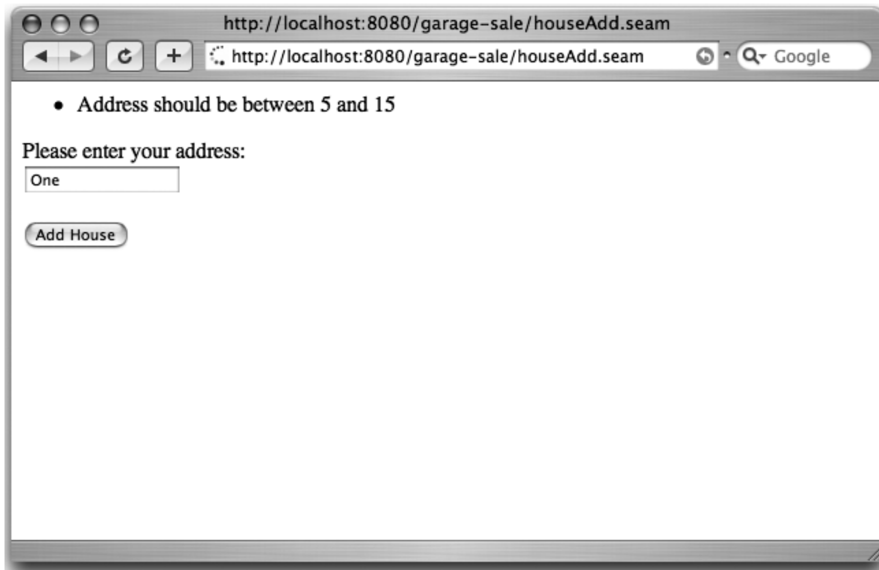


Figure 5-11. *House Add Page with an error*

Schema Generation

One final positive side about the validator mechanisms is in its use for schema generation. If you are having your schema automatically generated, the validation framework will add the appropriate not-nulls, lengths, and so forth to it. Obviously, not all the validations will be there, because constraints such as `@EmailChecks` are not normal database checks.

Summary

At this point, we have gone over the basics of MVC, JSF, EJB3, and now Seam. So you should have a basic understanding of how Seam works and how to write a basic Seam-enabled Java EE application. You have also learned about the different contexts that Seam uses to manage state. Although some of that information may be confusing right now, as you read the next two chapters, you will understand the usefulness of these contexts.

This chapter also covered a few additional components such as logging, data models, and debugging. These are all tools specific to Seam to help with web application development. There are of course many more, and some are tied to more-complex processes that you will read about later in this book. Most of these components are tied to higher-end processing.

In the next few chapters, you will learn about some progressively more-complex Seam processing.