CHAPTER 1

Introducing Seam

The first thing to understand about Seam is that it's a framework. Actually, it's a framework that sits on top of another framework (Java EE), and that framework sits on top of another one (Java). But don't get distracted by that just yet.

That word "framework" is a broad one, adopting many meanings depending on how it's used (and who is using it). In this case, I mean "framework" in a typical software technology sense: Seam knits together a set of APIs and services into an environment that makes it easy (or easier) to write Java EE web applications.

A framework typically "makes it easier" to do something by simplifying common tasks and providing built-in utilities that you'd otherwise have to write yourself. Seam is no different. Seam is based on Java EE, so it satisfies its framework duties in two fundamental ways:

- *Seam* ***simplifies*** *Java EE*: Seam provides a number of shortcuts and simplifications to the standard Java EE framework, making it even easier to effectively use Java EE web and business components.

- *Seam* ***extends*** *Java EE*: Seam integrates a number of new concepts and tools into the Java EE framework. These extensions bring new functionality within the Java EE framework.

You'll get familiar with Seam in this chapter by briefly examining each of these aspects. In the rest of this chapter, I'll list for you the various services and utilities that Seam provides. In the chapters that follow, you'll see these services in action directly, applied in application development cases.

Seam Simplifies Java EE

The standard Java EE environment consists of the Java Standard Edition (Java SE) with all of its APIs (JDBC for database access, JAXP for XML processing, etc.) supporting all of the enterprise-level capabilities of Java EE (JSF/JSP/servlets for web components, JAX-WS for web services, etc.). Your application components are then built directly on top of this overall framework, as depicted in Figure 1-1.

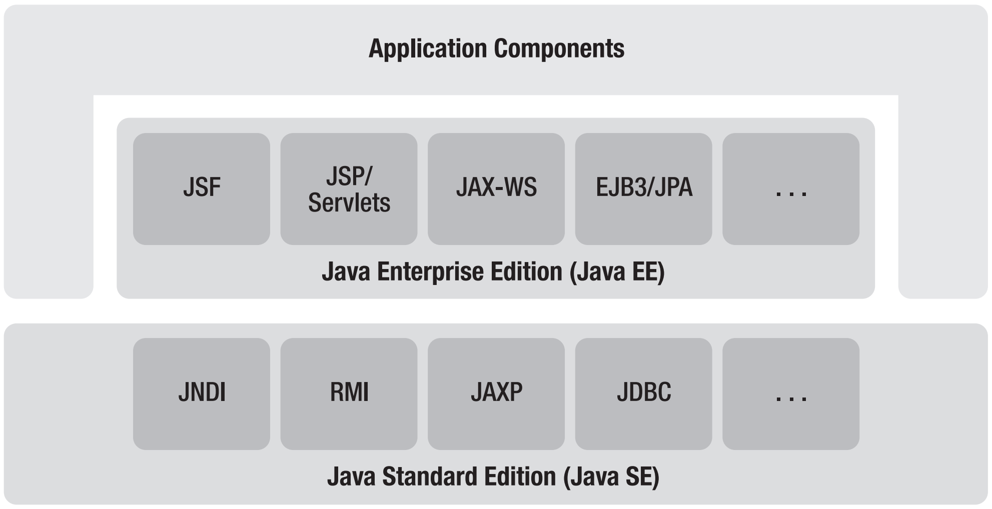


Figure 1-1. *Standard Java EE framework*

In addition to the APIs and component types depicted in Figure 1-1, Java EE also provides the deployment services, runtime security, and other services you need to create effective applications. And Java EE provides a number of improvements over its predecessor framework, J2EE, for example:

- Java 5.0 annotations are integrated liberally throughout the APIs in Java EE, giving you the option of using either externalized XML deployment data or embedded code annotations.
- The JavaServer Faces (JSF) 1.2, Java API for XML-based Web Services (JAX-WS) 2.0, and Enterprise JavaBeans (EJB) 3.0 APIs offer easier programming models than their J2EE predecessors, allowing you to implement most web, web service, and business components using simple JavaBeans.
- EJB 3.0 eliminates the need for many of the interfaces and other artifacts required in earlier versions of EJB, in most situations.

Even with the improvements delivered with Java EE, the JBoss Seam team saw room for simplifying things even further. Figure 1-2 depicts the Seam framework layered between your application code and the Java EE framework.

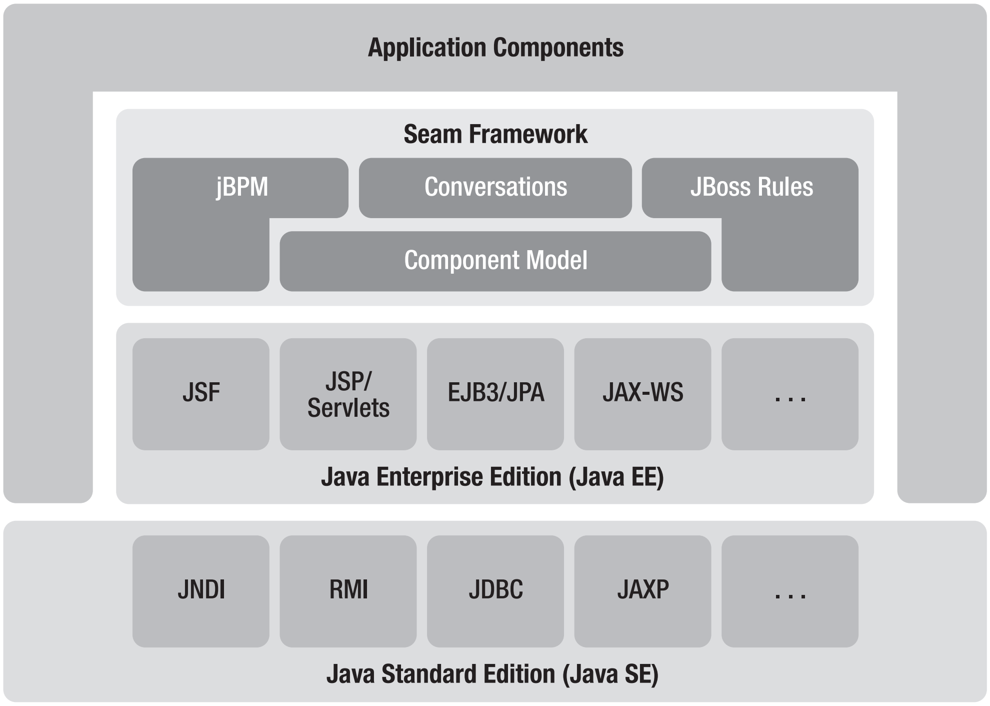


Figure 1-2. *Seam-enhanced Java EE framework*

The Seam Component Model

The simplifications provided by Seam stem mostly from the Seam component model—this component model can be considered, in essence, as an extension of the component model used for JSF managed beans. But it can be used for more than just web tier components, as you'll see in later chapters.

A key benefit provided by Seam's component model is the direct use of EJB components as backing beans for JSF pages. JSF's standard model allows for regular JavaBeans to be used as managed beans, configured in the JSF `faces-config.xml` file. EJB components can be invoked from the managed bean's callback methods, serving as a façade for the EJB component. Seam provides a direct bridge between JSF's component model and the EJB component model, allowing you to use an EJB directly as a JSF managed bean. This eliminates the need for extraneous façade beans when all you require is a single EJB.

Another simplification provided by Seam is the ability to use code annotations to directly bind beans to JSF component names, rather than writing `managed-bean` entries in the `faces-config.xml` file. The Seam component model includes annotations that can be used to link an instance of a bean directly to a JSF managed bean name. When the name is used in a JSF (one of its properties is used as the value of an HTML input field, for example), the bean instance will automatically be initialized, if necessary, and used as the backing bean for the JSF. There's no need to connect the bean to the JSF managed bean name using `faces-config.xml`.

The Seam component model also supports a more general version of dependency injection, called *bijection*. Standard dependency injection involves a one-time initialization of a bean reference within a component, typically done by some kind of container or other runtime service. Seam bijection extends this to support the following:

- *Two-way propagation of references*: A component can have a reference injected by the container, and a component can also "outject" a reference to the enclosing context as well.
- *Dynamic updates*: Instead of doing one-time injection of references, bijection is done on each invocation of the component. This is key in the Seam component model, since components can be stateful, and therefore they and their dependent beans can evolve across invocations.
- *Multiple contexts*: Dependencies (incoming and outgoing) can be established across multiple Seam contexts, rather than being forced to exist within a single context. So a session-scoped component can inject request-scoped beans and outject application-scoped beans, for example.

This may all sound a bit esoteric at this point, but the value of these features in the Seam component model will be clear once I show you some example code.

Running Example: A Gadget Catalog

The example we're going to use for much of the book is an online catalog of high-tech gadgets (mobile phones, laptops, digital media players, etc.). In coming chapters, we'll build up this application from the simple data entry tool described here into something a bit more interesting, and we'll also build solutions to other real-world cases, using the various capabilities of the Seam framework. But for now, we'll start with a very simple application that can only do two things:

- Display a list of gadgets contained in the catalog.
- Allow the user to enter a new gadget into the catalog.

At this point, our model for the application will be painfully simple: a gadget will only consist of a description (e.g., "Acme Powertop X1 Laptop") and a type (e.g., "laptop"). The data about these gadgets will be stored and managed in a relational database. The page-flow for the user interface will be equally simplistic: a main page will display the list of gadgets in the database and offer a single option to add a new gadget to the database. This option will bring the user to an entry form that prompts for the necessary attributes, and on submission the new gadget will be stored in the database, and the updated list of gadgets will be displayed again.

We can represent the "solution design" at this point with a pageflow diagram and a relational database diagram. The pageflow for the first iteration of the Gadget Catalog is shown in Figure 1-3, and the database structure (such as it is) is shown in Figure 1-4.

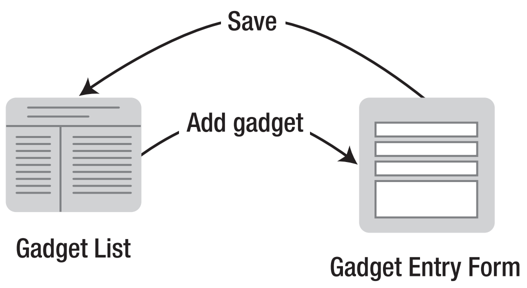
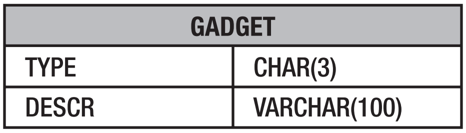


Figure 1-3. *Gadget Catalog pageflow*



GADGET	
TYPE	CHAR(3)
DESCR	VARCHAR(100)

Figure 1-4. *Gadget Catalog database*

Now all we have to do is build it. As a point of reference, let's first see what the Gadget Catalog looks like in the vanilla Java EE framework.

The Gadget Catalog Without Seam

The code for this example can be found in the code bundle for this book, under the `intro-JavaEE` subdirectory. In the Java EE framework, without JBoss Seam, the customary approach to implement the Gadget Catalog is to use JSF for the UI in conjunction with EJBs for the backing business logic and persistence.

To start, we'll implement an EJB 3.0 entity bean to represent the gadgets to be stored in the `GADGET` table. Listing 1-1 shows the `Gadget` bean. This is a simple EJB 3.0 entity bean that is mapped to the GADGET table using the EJB @Table annotation. The bean has two persistent properties: the description property is mapped to the DESCR column, and the type property is mapped to the TYPE column.

Listing 1-1. Gadget *Entity EJB*

```
@Entity
@Table(name="GADGET")
public class GadgetBean implements Serializable {
     private String mDescription = "";
     private String mType = "";

     public GadgetBean() { }

     @Id
     @Column(name="DESCR")
     public String getDescription() {
         return mDescription;
     }

     public void setDescription(String desc) {
         mDescription = desc;
     }

     @Id
     @Column(name="TYPE")
     public String getType() {
         return mType;
     }

     public void setType(String t) {
         mType = t;
     }
}
```

■Practical Tip Be careful about SQL reserved words used as EJB entity bean class or property names. Persistence engines may try to map them directly to auto-generated columns/tables, resulting in unexpected `SQLExceptions`. Notice that we called our `GadgetBean` property "description", rather than "desc". This is longer to type, but "desc" is reserved in some databases. If you decided to auto-generate the schema, a property called "desc" could be mapped into a column named "DESC", and problems could ensue. We're being extra careful here by using explicit `@Column` EJB3 annotations to map the properties to columns in our database model, so even if we auto-generated the schema (as we do in the sample code provided in the book's code bundle), we're sure not to run into issues.

In order to implement the functionality we've laid out for our Gadget Catalog, we'll need to be able to get a list of all gadgets currently in the database, and we'll need to be able to add a new `Gadget` to the database. Using a fairly typical "session façade" pattern for EJBs, we create a `GadgetAdminBean` session EJB to provide these functions. The code for this is shown in Listing 1-2.

Listing 1-2. `GadgetAdminBean` *Session EJB*

```
@Stateless
public class GadgetAdminBean implements IGadgetAdminBean {
     @PersistenceContext(unitName="gadgetDatabase")
     private EntityManager mEntityManager;

     /** Retrieve all gadgets from the catalog, ordered by description */
     public List<GadgetBean> getAllGadgets() {
         List<GadgetBean> gadgets = new ArrayList<GadgetBean>();
         try {
             Query q =
                  mEntityManager.createQuery("select g from GadgetBean " +
                                            "g order by g.description");
             List gList = q.getResultList();
             Iterator i = gList.iterator();
             while (i.hasNext()) {
                 gadgets.add((GadgetBean)i.next());
             }
         }
```

```
        catch (Exception e) {
            e.printStackTrace();
        }
        return gadgets;
    }

    /** Insert a new gadget into the catalog */
    public void newGadget(GadgetBean g) {
        try {
            mEntityManager.persist(g);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This session EJB uses standard EJB 3.0 and Java Persistence API (JPA) calls to implement the required functions. We've marked this as a stateless session bean using the EJB 3.0 `@Stateless` annotation on the class declaration. We're also using the JPA `@PersistenceContext` annotation to inject a JPA `EntityManager` into this session bean, allowing us to perform the persistence operations necessary to query and insert into the gadget database. We're referencing a persistence unit named "gadgetDatabase", so we'll need to define a persistence unit with this name in the `persistence.xml` deployment file when we package up these EJBs.

The `getAllGadgets()` method loads the entire Gadget Catalog using a JPA query created from the `EntityManager`. The `newGadget()` method persists a new gadget (in the form of a `GadgetBean`) using the `EntityManager`.

These two EJBs seem to take care of our current needs in terms of persistence operations, so now we can turn our attention to the UI. To implement the UI we specified in the pageflow design earlier, we create two JSF pages, one for each of the pages we specified. The first JSF page displays the list of gadgets in the database along with a link to create a new gadget. In building these pages, let's assume we can access the persistence functionality we built earlier through a JSF managed bean named "gadgetAdmin". Our gadget list JSF is shown in Listing 1-3. It simply uses a JSF data table component to iterate through the gadgets returned from the `getAllGadgets()` operation on the `gadgetAdmin` bean, displaying each gadget as a row in a table. Then, at the bottom of the table, we generate a link that invokes a JSF action named "addGadget".

Listing 1-3. *Gadget List JSF Page*

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

<html>
<head>
    <title>Gadget List</title>
</head>

<body>
    <f:view>
        <h:messages/>
        <!--  Show the current gadget catalog -->
        <h:dataTable value="#{gadgetAdmin.allGadgets}" var="g">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Type" />
                </f:facet>
                <h:outputText value="#{g.type}" />
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Description" />
                </f:facet>
                <h:outputText value="#{g.description}" />
            </h:column>
        </h:dataTable>
        <h:form>
            <!-- Link to add a new gadget -->
            <h:commandLink action="addGadget">
                <h:outputText value="Add a new gadget" />
            </h:commandLink>
        </h:form>
    </f:view>
</body>
</html>
```

The `addGadget` action is supposed to bring us to the second page in our pageflow, the gadget entry form. The JSF page that implements this, `addGadget.jsp`, is shown in Listing 1-4.

Listing 1-4. *Gadget Entry JSF Page*

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

<html>
<head>
    <title>Add a Gadget</title>
</head>

<body>
    <f:view>
        <h:form>
            <table border="0">
                <tr>
                    <td>Description:</td>
                    <td>
                        <h:inputText value="#{gadget.description}"
                                     required="true" />
                    </td>
                </tr>
                <tr>
                    <td>Type:</td>
                    <td>
                        <h:selectOneMenu value="#{gadget.type}"
                                         required="true">
                            <f:selectItems value="#{gadgetAdmin.gadgetTypes}" />
                        </h:selectOneMenu>
                    </td>
                </tr>
            </table>
            <h:commandButton type="submit" value="Create"
                             action="#{gadgetAdmin.newGadget}" />
        </h:form>
    </f:view>
</body>
</html>
```

This page generates a simple entry form that prompts the user for a type and description for a new gadget for the catalog. The description field is a simple text entry field, while the type is a drop-down menu populated with allowed values from the `GadgetTypes` enumeration. Both fields are bound to properties on a new managed bean named "gadget". At the end of the form is a submit button that invokes the `newGadget()` operation on the `gadgetAdmin` managed bean.

At this point, as with any JSF application, we need to wire the JSF managed beans to classes in our model. We could try to associate the `gadgetAdmin` bean with an instance of our `GadgetAdminBean` session EJB and the `gadget` bean to our `GadgetBean` entity EJB, using entries in our `faces-config.xml` like this:

```
<faces-config>
     <managed-bean>
         <managed-bean-name>gadget</managed-bean-name>
         <managed-bean-class>GadgetBean</managed-bean-class>
         <managed-bean-scope>session</managed-bean-scope>
     </managed-bean>

     <managed-bean>
         <managed-bean-name>gadgetAdmin</managed-bean-name>
         <managed-bean-class>GadgetAdminBean</managed-bean-class>
         <managed-bean-scope>session</managed-bean-scope>
     </managed-bean>
</faces-config>
```

But what you'll find is that this won't work, at least not the way you'd expect. In JSF, managed beans are expected to be simple JavaBeans, and they'll be managed that way at runtime by the JSF container. When the `gadget` or `gadgetAdmin` beans are created and used at runtime, the JSF container won't follow the rules for EJB components when handling them. It won't, for example, use the EJB container to obtain an instance of the `GadgetAdminBean`, as you should for any session bean. Instead, it will try to construct instances of `GadgetAdminBean` directly, outside of the EJB container and all of its services. This obviously isn't what we want—we defined `GadgetBean` and `GadgetAdminBean` as EJB components because we wanted them to use the persistence and transaction management services of the EJB container.

In order to integrate our EJB components into our JSF pages, we need to create JavaBean wrappers that the JSF components can use directly. These JavaBean wrappers will then interact with the EJB components to get the gadget persistence done.

First, we'll have to write a simple JavaBean version of our `GadgetBean` entity EJB that can be used as the `gadget` managed bean. This JavaBean class, `Gadget`, is shown in Listing 1-5.

Listing 1-5. *Simple JavaBean for Gadget Values*

```
public class Gadget {
     private String mDescription;
     private GadgetType mType;

     public String getDescription() { return mDescription; }
     public void setDescription(String desc) { mDescription = desc; }

     public String getType() { return (mType != null ? mType.name() : null); }
     public void setType(String t) { mType = GadgetType.valueOf(t); }
}
```

This JavaBean just carries the value of the gadget's properties between the JSF in the UI and the `GadgetBean` in the EJB container. It has the same two properties as our `GadgetBean` entity EJB, naturally.

Now, we need a JavaBean façade for our `GadgetAdminBean` session EJB. We'll be using this bean to implement JSF actions, so we call it "GadgetAdminAction". The code for this wrapper is shown in Listing 1-6.

Listing 1-6. *Action Wrapper for* `GadgetAdminBean`

```
public class GadgetAdminAction {
     @EJB
     private IGadgetAdminBean mGadgetAdmin;

     // Managed property for our JSF action, populated with the
     // Gadget being operated in the current request (added/deleted/edited)
     private Gadget mGadget;
     public Gadget getGadget() { return mGadget; }
     public void setGadget(Gadget g) { mGadget = g; }

     /** Facade to the newGadget operation on the GadgetAdminBean */
     public String newGadget() {
         // Convert the Gadget into a GadgetBean and persist it
         GadgetBean bean = gadgetToBean(mGadget);
         mGadgetAdmin.newGadget(bean);
         return "success";
    }

    /** Facade to the getAllGadgets operation on the GadgetAdminBean */
     public List<Gadget> getAllGadgets() {
        List<Gadget> gadgets = new ArrayList<Gadget>();
        List<GadgetBean> beans = mGadgetAdmin.getAllGadgets();
        Iterator i = beans.iterator();
        while (i.hasNext()) {
            Gadget g = beanToGadget((GadgetBean)i.next());
            gadgets.add(g);
        }
        return gadgets;
    }

    public Map<String,String> getGadgetTypes() {
        Map<String,String> types = new HashMap<String,String>();
        for (GadgetType value : GadgetType.values()) {
            types.put(value.label(), value.name());
        }
        return types;
    }

    /** Convert a Gadget JavaBean to a GadgetBean EJB */
    private GadgetBean gadgetToBean(Gadget g) {
        GadgetBean bean = new GadgetBean();
        bean.setDescription(g.getDescription());
        bean.setType(g.getType());
        return bean;
    }

    /** Convert a GadgetBean EJB to a Gadget JavaBean */
   private Gadget beanToGadget(GadgetBean g) {
        Gadget ig = new Gadget();
        ig.setDescription(g.getDescription());
        ig.setType(g.getType());
        return ig;
    }
}
```

The `GadgetAdminAction` wrapper bean does two things: it converts `GadgetBean` entity EJBs into `Gadget` JavaBeans for the JSF components with the `beanToGadget()` utility method and also converts `Gadget` beans from the JSF beans back into `GadgetBean` EJBs to persist them with the `gadgetToBean()` utility method. It also has a method for each operation on the `GadgetAdminBean` that we want to invoke as actions. The `getAllGadgets()` method invokes the `GadgetAdminBean.getAllGadgets()` method and converts the list of `GadgetBean` references into `Gadget` instances. The `newGadget()` method takes the managed

Gadget property and converts it into a new GadgetBean, and then passes it to the GadgetAdminBean.newGadget() method to be persisted.

Finally, we can wire these JavaBean wrappers into the JSF UI as managed beans, using the faces-config.xml file shown in Listing 1-7.

Listing 1-7. *JSF* faces-config.xml *for Java EE Gadget Catalog*

```
<faces-config>
    <managed-bean>
        <managed-bean-name>gadget</managed-bean-name>
        <managed-bean-class>Gadget</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>gadgetAdmin</managed-bean-name>
        <managed-bean-class>GadgetAdminAction</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>gadget</property-name>
            <value>#{gadget}</value>
        </managed-property>
    </managed-bean>

    <navigation-rule>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/listGadgets.jsf</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-action>addGadget</from-action>
            <to-view-id>/addGadget.jsf</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Notice that, in addition to using the Gadget JavaBean as the gadget managed bean and GadgetAdminAction as the gadgetAdmin managed bean, we've also defined the pageflow in a navigation-rule section. Any "success" outcome should take the user back to the gadget list page, and the addGadget action we referenced in the link on the listGadget.jsp page should take the user to the addGadget.jsp JSF page.

The Gadget Catalog with JBoss Seam

The last few steps that we were forced to carry out in creating the plain Java EE version of the Gadget Catalog provide some of the motivation for the simplifications introduced in JBoss Seam. If you look back at the previous section, you'll notice that we had all the required persistence and UI functionality for the Gadget Catalog defined as of Listing 1-4. Everything that follows in the section is just overhead that's needed in order to bridge from the JSF components to the EJB components, and to configure everything. This "overhead" includes the following:

- The `Gadget` and `GadgetAdminAction` JavaBean wrapper classes
- The managed bean declarations in the `faces-config.xml`

The JBoss Seam project team saw these steps as wasted effort to be eliminated. Creating the JavaBean wrappers to integrate the JSF and EJB component models doesn't add anything to the functionality of the Gadget Catalog; it's just an implementation detail required because Java EE doesn't provide its own bridge between these two component models. And their philosophy on the `managed-bean` entries in the `faces-config.xml` file is that these represent missing code annotations in the JSF model. In EJB, virtually all of the details in the `ejb-jar.xml` deployment descriptors can (optionally) be replaced with code annotations in EJB 3. Why not give JSF programmers the same option with all those `faces-config.xml` options?

JBoss Seam eliminates both of these issues, making our Gadget Catalog simpler to implement when we use a Java EE server enhanced with Seam. First off, we can use our `GadgetBean` and `GadgetAdminBean` EJB components directly as managed beans within JSF pages. We no longer need the JavaBean wrapper classes, because Seam provides a bridge between the JSF and EJB component models.

Second, Seam provides a `@Name` annotation that can be inserted into our EJBs to specify their JSF managed bean names. Our updated `GadgetBean` EJB is shown in Listing 1-8.

Listing 1-8. *Seam-Enhanced GadgetBean Entity EJB*

```
@Entity
@Table(name="GADGET")
@Name("gadget")
public class GadgetBean implements Serializable {
     private String mDescription = "";
     private String mType = "";

     public GadgetBean() { }

     @Id
```

```
    @Column(name="DESCR")
    public String getDescription() { return mDescription; }
    public void setDescription(String desc) {mDescription = desc; }

    @Id
    @Column(name="TYPE")
    public String getType() { return mType; }
    public void setType(String t) { mType = t;}
}
```

The only difference in the Seam-enabled version is the `@Name("gadget")` annotation at the start of the class. This annotation is equivalent to the "gadget" `managed-bean` entry in the `faces-config.xml` file shown in Listing 1-7. We can eliminate the "gadgetAdmin" entry as well if we put a corresponding `@Name` annotation on the `GadgetAdminBean` EJB class.

In summary, Seam has helped us to significantly simplify the implementation of our little application. There are fewer objects in our model, and the relationship between the UI views and the objects in the model makes more sense. You'll experience this again in Chapter 3, where we use an expanded version of our Gadget Catalog to examine the component model in JBoss Seam. There, I compare the object models with and without Seam, and show you direct evidence of the benefits of linking the EJB and JSF component models. That chapter also examines a number of other features of the Seam component model and the benefits that they bring.

Seam Extends Java EE

The previous section gave you a sense for the ways that JBoss Seam simplifies Java EE development, especially when it comes to applications using JSF and EJB components. This section quickly surveys the various extended capabilities that Seam provides in its framework. These capabilities are pretty compelling in their own right, even if the programming simplifications described earlier don't interest you.

Seam Component Services

I mentioned the Seam component model in the previous section because it serves as the basis for the JSF/EJB simplifications provided by Seam. The Seam component model also provides a number of powerful extended services on top of the Java EE framework.

All of the Seam component services discussed next will be explored in more detail in Chapter 3.

Seam Component Contexts

As with most Java EE component models, Seam components support various runtime contexts or scopes. Seam components support the typical contexts found in web components (request, page, session, and application scopes). But the model adds several additional contexts that can be useful in enterprise applications.

Seam components support an explicit representation of a stateless context, for example. This context isn't usually represented explicitly, because components with stateless context really don't have any context (context refers to the scope of the component state, and stateless components by definition have no state).

In addition, Seam components also have contexts for conversation scope and business process scope. The conversation context is a critical piece of Seam's overall web application model, so I discuss it next in its own section. I also devote Chapter 4 to the conversation model in Seam because of its importance in Seam and the many practical benefits it can bring to your application development.

The business process context represents state for a (possibly long-running) business process. Since business processes can run much longer than the lifetime of a request, a session, or even the application server itself, the business process context uses persistence services for its state data. Business processes also involve multiple users, potentially, so the context supports concurrent access as well.

Seam Conversations

Seam conversations are a very interesting and potentially powerful concept provided by the framework. One way to describe Seam's conversation concept is that it is a truly user-oriented transaction. Persistence transactions (like those managed in a JTA context) are typically defined around data consistency concerns, and business process transactions typically are defined around task dependencies and process structural consistency. Seam conversations provide another dimension of transactions, defined by what a user needs/wants to do within an application.

Another, less formal but more palatable, way to describe conversations is that they provide another layer between request scope and session scope in web applications. A conversation can group together data across multiple web requests, and it can track multiple groupings like this within a single user session.

Seam Events

JSF provides an event model in its component model, but Seam introduces two new types of events that can be utilized within Seam applications: *page actions* and *component-driven* events.

Seam page actions are events that you can have triggered after a user makes a web request, but before the requested web component or page is rendered. You can specify page actions that are to be invoked on request to specific views, or use wildcards to cover groups of views. Page actions are implemented by component operations (JSF managed bean methods and/or Seam component methods).

Seam component-driven events provide a general event notification scheme for Seam components. A set of named events can be defined, and component methods can be registered as "callbacks" on these events. Any component can trigger an event anywhere in its business code, and any component operations on the notification list will be invoked.

Integrated Pageflow with jPDL

Seam integrates jBPM into the framework in two ways. First, it's used to implement a rich pageflow capability. The jBPM Process Definition Language (jPDL) supports more complex and robust pageflows than JSF navigation rules can provide. jPDL pageflows are stateful, in the sense that the flow is defined as transitions between explicit, named states. jPDL pageflows are much more structured than JSF navigation rules—they define starting states, transitions, intermediate state nodes, and end states. And jPDL pageflows can be more explicit and externalized than JSF pageflows—transitions are defined around component actions and their outcomes, state transitions can themselves trigger state changes by invoking component methods, and so on.

You'll learn more about the pageflow capabilities provided by Seam and jPDL in Chapter 5.

Integrated Business Processes with jBPM and JBoss Rules

jBPM is also used in Seam to provide support for business process management in Seam applications. Business processes merge pageflow, transactions, task definitions, and rules to provide a way to define structured task flows. A business process defines what needs to be done and in what order, who needs to perform specific tasks or task types, and the rules under which all this happens.

In addition to jBPM, Seam also integrates the JBoss Rules framework into its business process services. These two capabilities work hand-in-hand to allow you to define and implement business processes in your Seam application.

You'll explore Seam's business process and rules engine support in Chapter 7.

Rich Internet Applications (aka Web 2.0)

You may have noticed the “Web 2.0” buzz word in the title of this book. I didn’t add that just to sell books—Seam provides some powerful facilities for integrating rich Internet UIs into your Seam component model, providing interesting ways to plug your business logic directly into your client-side UI.

Seam supports rich Internet applications built using Asynchronous JavaScript and XML (AJAX) in two fundamental ways. First, it supplies a remoting capability for Seam components (mentioned earlier in this chapter) that allows JavaScript code to invoke your components directly from the client side of the web UI. Second, Seam supplies a set of JavaScript “shim” code that makes it easy for you to make use of this component remoting capability in AJAX contexts.

You’ll explore this aspect of Seam in Chapter 8.

Read On

I largely mapped out the bulk of the book in the previous sections of this chapter. Chapter 3 provides a basic, practical overview of the core capabilities of Seam and its component model by creating an extended version of our Gadget Catalog application. Chapter 4 looks at Seam’s conversation concept in depth, and how it can be used to more easily manage complex user interactions and easily support multiple concurrent “workspaces” for users. Chapters 5 through 8 explore specific extended capabilities that Seam provides within its framework (specifically, pageflow with jPDL, security services, business processes with jBPM and Rules, and rich Internet UIs with component remoting and AJAX).

But first, in Chapter 2, I discuss another practical issue related to any framework, including Seam. That is the installation, configuration, and debugging of Seam applications. These topics are a necessary evil that comes with developing applications using a framework. If you’d prefer to blissfully ignore these topics for now and jump right into the “fun stuff,” you can safely proceed to Chapter 3 and get your teeth into the Seam component model right away. You should read Chapters 3 and 4 before proceeding on to the rest of the book, however, because the Seam component and conversation models provide the backdrop for all of the other capabilities integrated into the framework. And like it or not, if you do start using Seam “for real,” you’ll find yourself returning to Chapter 2 for assistance with the administrative details of Seam.

Summary

In this chapter, you've had a brief (but hopefully motivating) introduction to the capabilities JBoss Seam brings to the table when building Java EE applications. You saw how Seam simplifies Java EE by bridging its EJB and JSF component models, and how Seam extends Java EE with a number of additional capabilities, like structured pageflow, business process management, and rich web application support. You were also introduced to the Gadget Catalog application, which I'll use throughout the rest of the book to demonstrate all of these areas of Seam.