

Covers Seam 2.0

SEAM

in Action

Dan Allen

MEAP

Unedited Draft

 MANNING



Table of Contents

Part 1 - Teeing off with Seam

- 1 Seam unifies Java EE
- 2 Putting seam-gen to work

Part 2 – Seam fundamentals

- 3 The Seam life cycle
- 4 Components and contexts
- 5 The Seam component descriptor
- 6 Absolute inversion of control

Part 3 – How Seam manages state

- 7 The conversation: Seam's unit of work
- 8 Understanding Java persistence
- 9 Seam-managed transactions and persistence
- 10 Rapid Seam development

Part 4 – Sinking the business requirements

- 11 Securing Seam applications
- 12 Ajax and JavaScript remoting
- 13 File uploads, rich rendering, and email support
- 14 Managing the business process
- 15 Spring integration

Appendix A. Seam starter set

Appendix B. Seam annotations quick reference

Appendix C. JSF component libraries



MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Seam unifies Java EE

Is JSF worth a second look? Is EJB really fixed? Is it worth sticking with Java rather than jumping ship for Ruby on Rails?

With the release of JBoss Seam 2.0, you can now confidently answer *yes* to all of these questions. Seam is a progressive application framework for Java EE that makes writing web-based applications easier by finally delivering on the promise of a unified component architecture. Seam builds on the innovative changes in Java EE brought about by the Enterprise JavaBeans (EJB) 3 specification. These changes include favoring annotations over container interfaces and relying on configuration by exception rather than verbose and laborious XML descriptors. Seam tares down Java EE's remaining heavyweight legacy by spreading EJB 3's pivotal changes across the platform, leveraging more annotations, more configuration by exception and extending the platform as designed, weaving functionality into the JavaServer Faces (JSF) life cycle, and using the unified EL to allow these technologies to communicate. With Seam, the pain typically associated with using Java EE has vanished and JSF, in particular, appears completely revamped and worthy of attention.

In this chapter, you discover why Seam is the most exciting technology that has landed in Java's turf since its inception and the reasons why you should make Seam your framework of choice. I demonstrate how Seam solves your current problems with the Java EE platform by blending innovative concepts with existing standards. In a world inundated with frameworks, JBoss Seam is the unframework. It does not force a new programming model on you. Instead, Seam pulls together the standard Java EE APIs, makes them more accessible, functional, and attractive, and then finishes them off with modern upgrades such as page flows, JavaScript remoting, PDF rendering, email composition, charting, file upload management, business processes, and Groovy integration. Like a classic car, underneath the hood Seam has all the muscle of Java EE, but on the surface it appears stunning and elegant.

Putting Seam's strengths aside, the fact remains that there are many qualified frameworks that you have to choose from. In the next section, I provide you with advice that can hopefully put an end to your search and move you towards developing your application. Despite the fact that no one can tell you what framework is right for you, you are probably going to ask anyway, right? Don't worry, I came prepared.

1.1 Which framework should I use?

In a world full of framework options, how do you choose one? There are so many frameworks available for the Java platform, some proven, some promising, that the decision is downright agonizing! Does figure 1.1 speak to you?

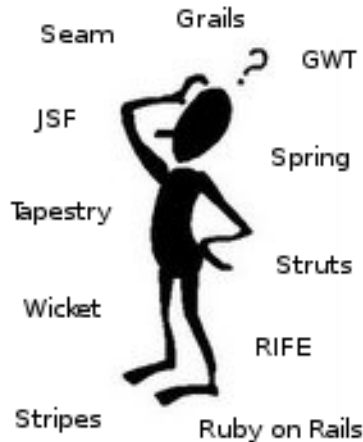


Figure 1.1 The great framework decision.

The choice is so bewildering that the framework inquiry is now the dominant greeting exchanged between developers at conferences. While the question "What do you do?" may have traditionally served in the role of sizing up a person's abilities, these days you are judged based on the merit of what framework you use for software development (or the advice that you can give pertaining to that choice). Just when you've made a decision, a new framework arrives on the scene promising to bury its predecessors.

These choices can be harmful, especially to productivity. Barry Schwartz argues in [The Paradox of Choice](#) that having a bewildering array of options floods our already exhausted brains. The result is that your ability to write a quality application stalls. You keep believing that the best framework is the one you haven't tried yet. As a consequence, you spend more time researching frameworks than you do designing functional applications. The search *consumes* you. You develop a false sense of how busy you are. While you may appear busy, the fact is, you aren't accomplishing much.

If any of these choices were truly satisfying, than you probably would not be reading this book. You would already have a set of tools that you know, beyond all doubt, allows you to be highly productive. But, you don't, do you? You are still searching for a framework that is new, yet familiar. Lightweight, yet powerful. You are in need of a platform that integrates the vast landscape of Java technologies into a unified stack. Seam might be just the framework you are looking for.

1.2 Choosing Seam

You might be tempted to think that Seam is just another web framework, competing in an already flooded market. In truth, to tag Seam as a web framework is quite unfitting. Seam is far more broad than a traditional web framework, such as Struts, and is better described as an application stack.

1.2.1 A complete application stack

Let's consider the distinction between an *application stack* and a *web framework*. Web frameworks are analogous to the guests that show up just in time for dinner and then leave immediately after eating. They entertain and soak up the limelight, but they are mostly unhelpful. They go out the same way they arrived, with lots of commotion. An application stack, on the other hand, is like the person who helps to plan the dinner party, shops for the groceries, cooks, sets up, serves, makes the coffee, and

then ultimately cleans up when it is all over. They are steadfast and resourceful. Sadly, their work goes mostly unrecognized.

In a world where everyone wants to be a rock star (i.e. web framework), Seam is your practical sidekick, your sous chef. The Seam stack includes the framework, the libraries, the build script and project generator, the IDE integration, a base test class, an embedded JBoss container, and integrations with other technologies. Seam is a certainly hard worker. Figure 1.2 gives a sample cross-section of the technologies that Seam is capable of pulling together in a typical application.

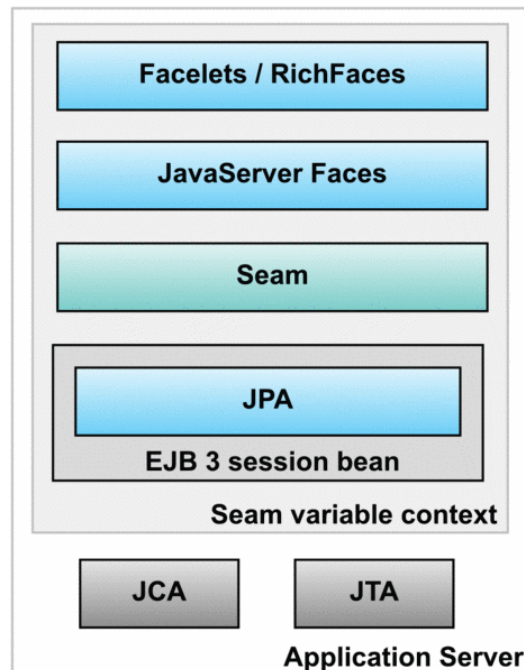


Figure 1.2 A cross-section of the technologies incorporated in the Seam stack.

While this stack gives you an idea of the technologies used in a Seam application, it does not give you a clear picture of Seam's purpose and why it exists. To understand why Seam was created, you have to recognize the challenge that it faced. Although the Java EE 5 release took a gigantic step towards establishing an agile platform for enterprise Java development, it left behind a rather significant gap between the component-based web tier managed by JSF and the component-based business-tier managed by EJB 3. A bridge was needed.

1.2.2 Why Seam was created

The Java EE 5 specification incorporates two key component architectures (specifications for creating reusable objects) for creating web-based business applications: JavaServer Faces (JSF) 1.2 and Enterprise JavaBeans (EJB) 3. JSF is the standard presentation framework for the web-tier that provides both a user-interface component model and a server-side event model. EJB 3 is the standard programming model for creating secure and scalable business components that access transactional resources. EJB 3 also encompasses the Java Persistence API (JPA), which defines a standard persistence model for translating data between a relational database and Java entity classes.

Aside from their residence in the Java EE 5 specification, the two architectures just mentioned share little resemblance, their backs facing each other like two sides of a coin. This communication barrier casts a shadow on the tremendous potential each of these technologies has. While it's true that

developers were able to get these two Java EE tiers to work together, it required a lot of "glue" code. Seam absorbs that responsibility and fits JSF and EJB 3 together, thus ironing out one of the roughest spots in the Java EE 5 specification and completing the missing link in the evolution of the Java EE platform. As such, Seam has positioned itself as the prototype for future Java EE specifications. Three such specifications are JSR 299 (Web Beans), JSR 314 (JavaServer Faces 2.0), and JSR 303 (Bean Validation). Seam isn't just about JSF and EJB 3, though. You can swap in alternative view technologies such as Wicket, Tapestry, GWT and Flex in place of JSF. Seam also boasts integration with the Spring container.

With that said, becoming the future of Java EE and an integration point for many open source technologies is not why Seam was created. That just happens to be a celebrated outcome. As with most software projects, Seam came about because of a single developer's itch.

As the story goes, Gavin King was fed up (and that doesn't take much) with the way developers were incorrectly using Hibernate, trapping it in a stateless design as popularized by the Spring Framework. Recognizing that the missing integration between JSF and EJB 3 would only lead to further abuse of Hibernate as a JPA provider, he decided to step in and build a foundation that would allow the persistence context (Hibernate `Session` or JPA `EntityManager`) to transcend layers and permit stateful session beans to respond directly to JSF UI components. The name "Seam" was chosen because this foundation brings JSF and EJB 3 together and teaches them to play nicely together in the same sandbox. In addition, Seam encourages the adoption of a stateful, yet efficient, architecture. As a result, applications built on Seam have effortless continuity from one user interaction (or event) to the next, a feature which is labeled a web conversation. The keen focus on variable scoping is what makes Seam contextual.

In the process of solving the mismatch between JSF and EJB 3, the Seam architects broadened the solution and created a universal component model, bringing the implicit services provided by the EJB 3 programming model, such as transactions, interceptors and security, to non-EJB components such as JavaBeans and Spring beans. For non-EJB components, Seam takes on the role of processing the Java EE annotations—or synonyms of these annotations from the Seam API—and weaving in the enterprise services. What that means is that you *do not* have to rely on an EJB 3 container to leverage the benefits the EJB 3 provides. In fact, you don't even need the JBoss Application Server to use Seam, despite what you may have heard.

1.2.3 Debunking the "vendor lock-in" myth

I don't want to be shy about addressing the myth that Seam is JBoss-focused technology or that by using Seam, you get locked into JBoss. Seam is no more a JBoss technology than Struts is an Apache technology or Spring is a SpringSource technology. An examination of the most successful complex projects in enterprise Java, such as Spring, Hibernate, Eclipse, JBoss AS, and even the Java EE platform, reveals that these projects are supported by organizations with paid developers. The projects in the JBoss Labs, such as Seam, are open source and can be whatever you, the community¹, drives them to be. While the projects may be operated under JBoss/RedHat's roof, the source code is yours to copy, share and modify. Specifically, JBoss Seam is licensed under the Lesser GNU Public License (LGPL), which is considered one of the more flexible options.

If you use Seam, you aren't stuck having to deploy to the JBoss Application Server either. A lot of effort has gone into ensuring that Seam is compatible and will run on all major application servers, including BEA Weblogic, IBM Websphere, Oracle OC4J, Apache Tomcat, and Glassfish. And it's

1 <http://www.seamframework.org> is the main communitie site for Seam

more than about deploying to major application servers. The improvements that Seam introduces are being contributed back into the platform as a standard using the Java Community Process (JCP) as a vehicle and summarized in the Java Specification Request (JSR) 299: Web Beans. The purpose of this JSR is to unify the JSF managed bean component model with the EJB component model, resulting in a significantly simplified programming model for web-based applications.

Seam is about standards blended with innovation. By signing your name on the dotted line and choosing Seam as your framework, you are not locking yourself in to a JBoss technology. Once the Web Beans JSR is accepted, as well as any others originating from the Seam project, any vendor can provide their own implementation. It is not a closed platform.

With an understanding of why Seam exists, and faith that you are not getting locked into JBoss by choosing this technology, you now need to consider if Seam is the right framework for you based on technical merit. After all, Seam may have saved Java EE, but can it fit the bill as your development framework of choice?

1.2.4 Making the case for Seam

Is there really a need for another application framework? Wasn't Spring supposed to be the one framework to rule them all? I will let the success of Ruby on Rails, and the wave of Java developers flocking to it, reveal that the need remains for a suitable Java application framework—or, in some developers' minds, an entire programming environment. So, should you follow the crowd? My advice is to look before you leap.

Promising that a framework will make the job of developing applications simpler is just lip service. Just because you are able to create a throwaway blog application with a framework doesn't make it viable. To earn the right to be called enterprise software, the framework has to stand up to the challenges of the real world, warts and all, and help the developer create well-designed, robust, and readable code. That is Seam's goal. Seam eliminates complexity and makes proven libraries more accessible. Seam doesn't turn its back on the pervasive Java EE platform, but rather serves as the glue that makes it truly integrated. Rather than encourage you to forget everything you know, Seam finds a way to allow you to use the Java EE services in a more agile way, while also providing enough new toys, in the form of extensions and third-party integrations, to make using it fun and interesting.

Below is a small sampling of the many improvements that Seam brings to the Java EE platform, all of which succeed in making the platform simpler:

- Eliminates the shortcomings in JSF, such as the lack of pre-render page actions, that have been the subject of countless rants
- Mends the communication between JSF and EJB 3 session beans / Spring beans
- Collapses unnecessary layers and cuts out passive middle-man components
- Offers a solution for contextual state management, discouraging the use of the stateless architecture (i.e. procedural business logic)
- Manages the persistence context (Hibernate `Session` or JPA `EntityManager`) to avoid lazy initialization exceptions in the view and in subsequent requests
- Provides a means for extending the persistence context throughout a use case
- Connects views together with stateful page flows
- Brings business processes to the web application world
- Plugs in a POJO-based authentication and authorization mechanism that is enforced at the JSF view ID level, accessible via the EL, and can be extended using declarative rules

- Provides an embedded container for testing in non-Java EE environments
- Delivers more than 30 reference examples with the distribution

As you can see, Seam is not shy about addressing problems in the platform, particularly those with JSF. For many, bullet point one is enough to justify the need for this framework if you are already committed to JSF. JSF can be quite painful without Seam's aid. The second point justifies Seam's usefulness in standards-based environments. But, Seam doesn't stop there. It advocates simpler architectures by encouraging developers to collapse unneeded layers and feel comfortable using long-running state. Seam does more than just improve the programming model. It provides a tool to build out the scaffolding of an application and generate CRUD functionality from an existing database schema, makes integration testing easy, and serves up Ajax in a variety of ways.

1.3 Seam's approach to unification

Seam revitalizes the standard Java EE platform by putting an end to its divergence and unifying its components, filling in the voids for which it is often criticized, making it more accessible, extending its reach to third-party frameworks and libraries, and form fitting them all together as a well integrated and consistent stack. While the features of Seam are vast, Seam's core mission is getting JSF, JPA, and POJO components to work together so that the developer's focus can be placed on building the application, not on integrating unallied technologies.

1.3.1 Seam Integrates JSF, JPA and POJO components

Getting technologies to work with one another is more than just having them pass messages. It's about creating an interaction that blurs the boundary between them, making them act as a single, unified technology. Seam achieves this integration by fitting EJB 3 up against the web-tier, finding a place for JPA, and scrapping the ineffectual JSF managed bean container. After reviewing how Seam tackles these challenges, you get a chance to determine which Seam stack is right for you.

Helping out a web-challenged EJB 3

By design, EJB components are not intended to be called on directly from JSF. It's great that EJB components are scalable, transactional, and secure, but it doesn't do much good if they are completely isolated from the web-tier, and in turn from JSF. This isolation makes them of limited use in web-applications because of the complexity involved to integrate them. They are not able to access data stored in any of the web-tier scopes (request, session, and application) or the JSF component tree, thus impairing their insight into essential parts of the application. (In practice, you really just need the EJB 3 components to have access to the conversation scope). Also, it's easy to get into trouble with concurrency when using EJB components from the web-tier. For instance, the Java EE container is not required to serialize access to the same stateful session bean, leaving it up to the developer to take care of this task or catch the exception that can result. There are also complexities that arise when dealing with non-thread-safe resources such as the JPA `EntityManager`. The only way the developer can safely use EJB components in the web-tier is by interfacing with an adapter layer.

Seam gives EJB 3 components access to web-tier scopes, offers a way to manage the state of these components so that they can be used safely in the web-tier, and even serializes access to stateful components to make concurrency issues a responsibility of the infrastructure and not the developer. Also, there is never a question about thread-safety accessing non-thread-safe resources since Seam handles the scoping properly.

Turning the tables, JSF faces equivalent challenges accessing business-tier components.

Hooking JSF to a better backend

JSF has its own "managed" bean container that is configured using a verbose XML descriptor rather than annotations and has a very limited dependency injection facility. While JSF managed beans can be stored in the web-tier contexts, they are barren objects, lacking scalability, transaction atomicity, and security (probably why they are termed beans and not components). They must reach out to an EJB 3 component to attain these business services. What you find is that you are stuck creating this facade layer to bridge EJB 3 components to the user interface that acts on them.

To correct this mismatch, Seam allows JSF UI components to tap right into the EJB layer by allowing EJB 3 components to stand in as JSF "backing" beans and action listeners. There is no longer a need for the managed bean facade layer and its verbose XML descriptor. By eliminating the complexity caused by the mismatch, it encourages developers to relax stringent mandates on overarchitected designs.

Which Seam are you?

Seam is not just a collection of classes and artifacts that get dropped on your desk with the disclaimer "some assembly required." The key to Seam's success is that it offers a handful of well-tested bundles that work fluently. You can liken it to the simplicity of buying a Mac when compared to buying a Dell. When you buy a Dell, you can customize the assembly down to the last stick of RAM. You get a product customized exactly to your needs, but getting there requires a lot of thought and effort on your part. Buying a Mac is much simpler in comparison. You choose between a laptop and a notebook, and then you select a screen size. Everything else is just details that Apple works out for you. Seam has a comparable set of options. You chose a state provider and a persistence provider (and, down the road, a web framework). Everything else is just details that the Seam developers work out for you. By removing the burden of too many choices, Seam can make life for the developer simpler.

The two main technology choices in a Seam application, summarized in figure 1.3, are the state provider and the persistence provider. The state provider is the technology that handles the application logic and responds to events in the UI. The persistence provider transports data to and from persistence storage. Seam manages the persistence provider to allow for the persistence context to be extended across a series of pages and shared amongst multiple components.

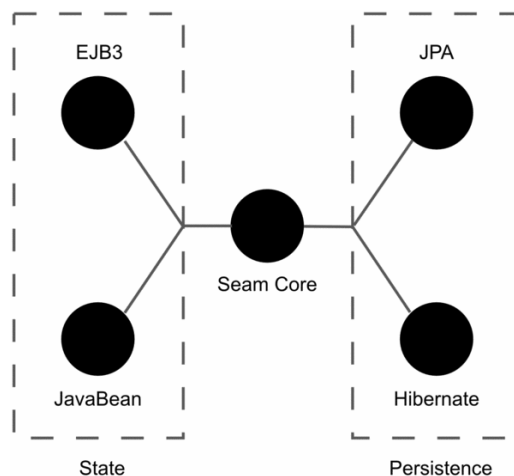


Figure 1.3 Seam's stack matrix, with options for state and persistence provider.

In looking at these options, you may be surprised to discover that EJB 3 is not required to use Seam. You can use Seam with basic JavaBeans and Hibernate and still not lose out on functionality. Note that JavaBean really means non-EJB component, so this option encapsulates Spring beans. Another popular bundle is to adopt JPA without using EJB 3 session beans, which is the bundle used in the example application in this book.

In order to make any of these technologies work together, there needed to be a way to integrate the various containers. EJB 3 has its container. JSF has one two. Spring is yet another. Once again, the task of integrating them falls on the shoulders of the developer. The need for a central integration point gave rise to Seam's contextual component model.

1.3.2 The contextual component model

At the heart of Seam is the contextual component model. Before your eyes gloss over, give me three short sentences to make this term meaningful to you. (1) Seam is a factory that constructs objects according to component definitions. (2) After creation, each object is stored in the container under one of several contexts (i.e. variable scopes) with varying lifetimes, making the objects contextual and capable of holding state (i.e. stateful). (3) Seam promotes the interaction of these stateful objects across contexts, assembling them together according to metadata associated with their respective classes. Chapter 4 explores components and contexts in depth and gives you an opportunity to learn how they are used in an application.

In this section, you learn how this model provides the basis for the unification of the technologies previously discussed. The unification is facilitated by a combination of the component registry, annotations, configuration by exception, method interceptors, and the unified expression language (EL).

A central component registry

Seam rakes in all of the Java EE components into a central registry, whether they are EJB session beans, JavaBeans, Spring beans or JPA entities. Any technology incorporated into the Seam stack can look to the Seam container to retrieve instances of the components by name and collaborate with the container to exchange state. Technologies which have access to the container include Seam components, JSF view templates, jBPM process definitions, jPDL page flow definitions, JBoss Rules, Spring beans, JavaScript, and more. Seam's container also unifies the variable scopes of the servlet API and introduces two additional stateful scopes, conversation and business process, that are better aligned with supporting user interactions.

Of course, components aren't just going to fall into this registry, they have to be recruited. Seam scours the classpath and enlists any class that contains a marker annotation, discussed next, that identifies it as a Seam component.

Annotations over XML

One way that Seam cuts down on the configuration overhead of Java EE is by eliminating needless XML. Although once thought to be desirable because of its flexibility, XML is an external configuration and quickly becomes out of sync (and out of touch) with the application logic. Seam brings configuration back in line with the code where it is easier to locate and can be refactored.

When the temptation arises to define JSF managed beans in XML, Seam just says "No", following the advice of figure 1.4. Seam reduces the declaration of a component to a single annotation, `@Name`, placed above the class definition. Seam components can take the place of JSF managed beans.

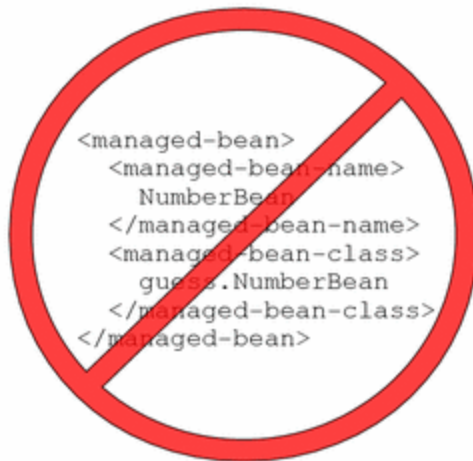


Figure 1.4 Seam cuts down on superfluous XML configuration that is difficult to keep in sync with the source code.

If you are dedicated enough, you can avoid the use of XML in Seam altogether, which is quite surprising given the number of places it *could* be warranted. Seam only resorts to XML when annotations do not suffice or to isolate deployment overrides. Moving to annotations is more than just improving the efficiency of keystrokes. Annotations are the central piece of Seam's configuration by exception strategy, conserving keystrokes until they are really necessary.

Configuration by exception

A good way to describe configuration by exception is by saying that the software is "opinionated." The general idea is that the framework happily prefers to operate as designed. The more you embrace the defaults, the less work you have to do. You are only required to step in and play a part when the software needs to do something different than the typical behavior.

In Seam, configuration by exception goes hand-in-hand with annotations. The annotations give Seam a hint to apply behavior and Seam tries to assume as much as possible about the declaration by relying on sensible defaults and standard naming conventions to keep your load light. In this way, Seam offers a nice balance between explicit declarations and assumed functionality.

While annotations cut down on keystrokes, there is more to annotations than just the elimination of XML. Annotations supply extra metadata to the class definition, where it is easier to find when compared with external descriptors and stays with the component throughout its lifetime.

Decorating components with services

Since components are requested through the Seam container, Seam has an opportunity to manage the instances throughout their life cycle. Seam wires the object with interceptors, wrapping it in a shell known as an object proxy, before handing down the newly created instance. This allows Seam to act as the object's puppeteer, pulling on its strings during each method call to modify its behavior, as depicted in figure 1.5. Interceptors account for much of the implicit logic in Seam that makes it "just work". Examples include beginning and committing transactions, enforcing security, and getting objects to socialize with one another. Annotations on the class definition give the interceptors a hint of how to apply the extra functionality, if for some reason it cannot be implied or needs to be different from the default behavior.

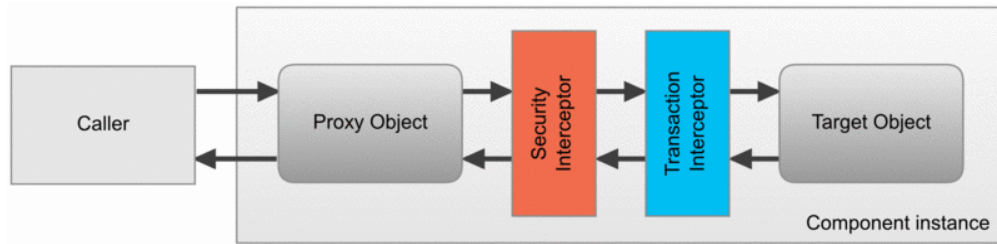


Figure 1.5 Interceptors trap method calls and perform cross-cutting logic around a method invocation.

The final piece to the unification puzzle is giving the application a way to access components in the container using a universal syntax. That is the role of the unified EL.

Extending the reach of the unified EL

The unified EL is a expressive syntax used to resolve variables and bind components to properties and methods on JavaBeans. It was introduced to better integrate JSF with JavaServer Pages (JSP). It is also used by JSF to lookup managed beans and other objects stored in web-tier scopes and is the basis for the JSF binding mechanism. It's impact, however, is far more widespread thanks to its pluggable design.

The EL is an open API that allows custom resolvers to be registered, thus turning the EL into a variable hub. Conversely, any layer of the code that wants to tap into the EL unified variable context can do so using its API. Thus, the EL frees you from having to develop a custom bridge between the variable contexts of the different technologies in your application. Although you are used to seeing the EL only in the view, there really isn't anything web-specific about it.

Seam takes advantage of the EL in two ways. First, Seam registers a custom EL resolver that is aware of the Seam container. This allows Seam components to be accessed using EL notation from anywhere in the application where the EL is available (which is pretty much everywhere). Secondly, Seam makes heavy use of the EL under the covers, allowing EL notation to be used in annotations, configuration descriptors, message strings, EJBQL queries, page flow definitions, and even business processes. With Seam, the EL truly is unified.

Despite all that has been said about Seam, nothing speaks to a programmer like lines of code. To help demonstrate why Seam is a sound choice and how it saves you valuable development time, I am going to whet your appetite with a brief example. I don't want to go into too much detail and spoil your appetite before chapter 2, when you get a chance to sink your teeth into Seam by building out an entire application with just a couple of commands.

1.4 Your first swings with Seam

To demonstrate some of the core principles of Seam, I am going to step you through a basic application that manages a collection of golf tips. Don't worry about trying to understand everything that you see here. Instead, focus on how Seam relies on annotations to define components, how it pulls the layers of the application together with its unified component model, and how it maintains a high signal-to-noise ratio in the business logic. I demonstrate a densely packed set of features in this example, so don't think that you have to use all of these techniques in order to use Seam.

We all want to be better golfers (at least, those of us who torture ourselves with the sport). Focusing on a simple golf tip can help to shave a couple of strokes off your round. To keep track of

the tips that you collect from the pros, buddies, and articles, you are going to slap together a Seam application that reads and writes these tips to a database. Aside from the deployment artifacts, which aren't going to be considered in this example, there are only a handful of files that you need to create to get this application functioning.

1.4.1 Entity classes a backing beans

I will start by discussing the `GolfTip` JPA entity class, shown in listing 1.1. In a Seam application, entity classes serve two purposes. Their primary role is to carry data to and from the database. The object-relational mapping mechanism, as this is called, is not part of Seam, per se. That work is handled either by JPA (the standard Java persistence framework) or Hibernate, though you will discover later that Seam helps manage the interaction with these two persistence frameworks.

In a Seam application, entity classes serve as form "backing" beans (akin to a Struts `ActionForm`) to capture input from the user, thus replacing the need for developing a dedicated "backing" bean class. An entity class becomes a candidate for use in a JSF view if it has a `@Name` annotation on its class definition, a condition which is satisfied by the `GolfTip` class in listing 1.1. You then bind the form inputs to properties on the entity class and JSF handles the necessary conversions.

Listing 1.1 The JPA entity class that represents a golf tip.

```
@Entity #2
@Name("tip") #1
public class GolfTip implements Serializable {
    @Id @GeneratedValue #3
    protected Long id;

    protected String author;

    protected String category;

    protected String content;

    // getters/setters for author, category and content not shown
}
```

Cueballs in code and text

The keywords prefixed with the `@` symbol are Java 5 annotations. The `@Name` annotation [#1], shown in bold, is a Seam annotation that registers the `GolfTip` class as Seam component named `tip`. Whenever the context variable `tip` is requested from the Seam container, Seam creates a new instance of the `GolfTip` class, binds the instance to the `tip` context variable in the conversation context (the default scope for entity classes), and returns the instance to the requester.

The remaining annotations in this class pertain to JPA. The `@Entity` annotation [#2] associates the `GolfTip` class with a database table by the same name. The `@Id` annotation [#3] indicates to JPA which property is to be used as the primary key. The `@GeneratedValue` annotation [#3] enables automatic surrogate key generation in the database. All of the other properties on the class (`author`, `category`, and `content`) are automatically mapped to columns with the same name as the respective property in the `GolfTip` table, following configuration by exception semantics.

As you can see, using the `@Name` annotation gives you one less file to worry about (that of the JSF managed bean facility and its verbose XML dialect). Staying away from the managed bean configuration is one of the early benefits of moving to Seam components. Another compelling advantage of moving to Seam is being able to bind the action of UI command component to an action listener method on transactional business object.

1.4.2 POJO components as action listeners

As with entity classes, there is no need to create a dedicated managed bean to act as a mediator between the JSF page and the service object in Seam. Instead, the service object can respond directly to an action invoked in the user interface. At first, that might sound like a really bad idea because it causes tight coupling between the user interface and the application logic. Seam prevents this coupling by acting as the mediator itself. The action listener component does not have to contain a single reference to a JSF resource. In fact, in chapter 3, you discover that the method's return value does not have to correlate with a navigation rule, which is a typical requirement of JSF managed beans. This example relaxes the separation from JSF to keep the number of classes to a minimum.

In the golf tips application, the tip `TipAction` class, shown in listing 1.2, is declared as a Seam component using the `@Name` annotation and is thus capable of having its methods bound to user interface controls. It handles the add and delete operations in the golf tips interface.

Listing 1.2 The action listener for the JSF view.

```
@Name("tipAction") #1
public class TipAction {
    @In #2
    EntityManager entityManager;

    @In #3
    FacesMessages facesMessages;

    @DataModel(scope = ScopeType.PAGE) #4
    private List<GolfTip> tips;

    @DataModelSelection
    @Out(required = false) #5
    private GolfTip activeTip;

    @Factory("tips") #4
    public void retrieveAllTips() {
        tips = entityManager.createQuery("from GolfTip")
            .getResultList();
    }

    public void add(GolfTip tip) {
        entityManager.persist(tip); #7
        activeTip = tip; #6
        facesMessages.add(
            "Thanks for the tip, #{activeTip.author}!"); #6
        retrieveAllTips();
    }
}
```

```

public void delete() {
    activeTip = entityManager.find(
        GolfTip.class, activeTip.getId());
    entityManager.remove(activeTip);
    facesMessages.add("The tip contributed by " +
        "#{activeTip.author} has been deleted.");      #6
    retrieveAllTips();
}
}

```

Cueballs in code and text

Like the `GolfTip` entity class, the `@Name` annotation [#1] designates the `TipAction` class as a Seam component, this time scoped to the event context (the default scope for JavaBean components). What really sets this component apart from the `GolfTip` entity class is that it is capable of receiving references to dependent components dynamically on account of the `@In` annotation being placed above certain fields of the class, a mechanism known as bijection. The two dependent components are the JPA `EntityManager` [#2] and the built-in JSF messages manager [#3]. This component also prepares and exposes the collection of tips to the JSF view [#4], captures the record selected in the UI [#5], and can reference that selected record in JSF status messages to be displayed to the user [#6]. All of these features are available without having to write a single line of custom code.

The `TipAction` component packs a lot of functionality in a very limited amount of space. What I want you to recognize is that, aside from the annotations, there is very little evidence of framework code in this class. The only code that you are required to write is that which reads, persists, and removes tips from the database by interacting with the JPA `EntityManager`. Even that code is probably better off in a data access object, which itself can be a Seam component. The focus of this example is on frugality. There is no infrastructure logic that reads request parameter values or sets request or session attributes. Instead, it only contains pure business logic.

1.4.3 Binding components to the view


Seam bridges the layers in the golf tips application by binding both the properties of the entity class and the action handler method to elements in the JSF view. Figure 1.6 shows the rendered `golftips.xhtml` template along with the value- and method-binding expressions that are associated with the page elements of interest.

NOTE

The file extension `.xhtml` indicates that this file is a Facelets template. Facelets is an alternative view handler for JSF that was created to alleviate the mismatch between the JSF and JSP life cycles. Facelets is the preferred view technology for Seam applications and is used throughout the book.


Use this figure to follow along with the discussion of how the JSF view interacts with the Seam components in the server.

Share your golf wisdom!


 Thanks for the tip, Jack Nicklaus!

Golf tips


Tiger Woods on The Swing

Shake hands with the target. 

Tommy Twoputt on Putting

Use one basic motion around the green. 

Jack Nicklaus on The Swing

The single most important maneuver in golf is the set-up. 

Do you have golf wisdom to share?

Author *

Category *

Content *

* required fields

Figure 1.6 The golf tips page, which renders the collection of tips at the top and a form for contributing a new tip at the bottom.

Start by focusing your attention on the form that is used to submit a new tip at the bottom of the page. Each of the input elements are bound to properties on the `GolfTip` entity class using expression language (EL) notation (e.g. `#{tip.author}`). When used in the `value` attribute of an input element, the EL notation acts as a value-binding expression. It captures the form value and transfers it to an instance of the `GolfTip` entity class as part of the JSF life cycle. Here is the (slightly trimmed down) fragment of the JSF template that renders the form:

```
<h:form>
  <h3>Do you have golf wisdom to add?</h3>
  <div class="field">
    <h:outputLabel for="author">Author:</h:outputLabel>
    <h:inputText value="#{tip.author}"/>
  </div>
  <div class="field">
    <h:outputLabel for="category">Category:</h:outputLabel>
    <h:selectOneMenu value="#{tip.category}">
      <f:selectItem itemValue="The Swing"/>
      <f:selectItem itemValue="Putting"/>
      <f:selectItem itemValue="Attitude"/>
    </h:selectOneMenu>
  </div>
</h:form>
```

```

</div>
<div class="field">
  <h:outputLabel for="content">Advice:</h:outputLabel>
  <h:inputTextarea value="#{tip.content}"/>
</div>
<div class="actions">
  <h:commandButton action="#{tipAction.add(tip)}"
    value="Submit Tip"/>
</div>
</h:form>

```

Seam makes the association between the value-binding expressions in the input fields and the `GolfTip` entity class through the context variable `tip`. The `@Name` annotation on the `GolfTip` class binds the class to the `tip` context variable. When the `tip` context variable is referenced by a value expression in the JSF template (`#{tip.*}`), Seam instantiates the `GolfTip` class and stores the instance in the Seam container under the variable name `tip`. All the value expressions that reference the `tip` context variable are bound to that same instance of the `GolfTip` class. When the form is submitted, the input values are transferred to the properties of the unsaved entity instance.

Let's consider what happens when the form is submitted. The method binding expression specified in the `action` attribute of the submit button, `#{tipAction.add(tip)}`, indicates that the `TipAction` component serves as the action handler for this form. Notice that this method expression actually passes the `GolfTip` instance associated with the `tip` context variable directly into the action method as its sole argument. Parameterized method-binding expressions are provided as an enhancement to JSF as part of Seam. You aren't required to use this feature, but it helps with readability.

1.4.4 Retrieving data on demand

What makes Seam so powerful is that it includes a mechanism for initializing a variable on demand. The top half of the screen in figure 1.6 renders the collection of tips in the database using the following markup:

```

<rich:dataGrid var="_tip" value="#{tips}" columns="1">
  <rich:panel>
    <f:facet name="header">
      <h:outputText value="#{_tip.author} on #{_tip.category}"/>
    </f:facet>
    <h:outputText value="#{_tip.content}"/>
    <h:commandLink action="#{tipAction.delete}">
      <h:graphicImage value="/images/delete.png" style="border: 0;"/>
    </h:commandLink>
  </rich:panel>
</rich:dataGrid>

```

The focal point of this markup is the `#{tips}` value expression. Notice that `tips` is not the name of one of the Seam components in the golf tips application. However, it is referenced in the `value` attribute of the `@Factory` annotation above the `retrieveAllTips()` method of the `TipAction` class from listing 1.2. The purpose of this method is to initialize the value of the `tips`

context variable when it is requested. Subsequent requests for the same variable return the calculated value rather than causing the method to be invoked again.

But, hold on a minute. The `retrieveAllTips()` method doesn't return a value. How is the value passed back to the view renderer? That's where things get a little tricky. After executing this method, Seam exports properties of the component that are annotated with either `@Out` or `@DataModel` to the view. Seam notices that the `@DataModel` annotation is assigned to the `tips` property on the `TipAction` component. That tells Seam not only to export its value to the `tips` context variable, but to first wrap the value in a JSF `DataModel` instance. The view iterates over this wrapped collection to render the data grid. The reason the collection is wrapped in a `DataModel` is to enable clickable lists to support the delete functionality.

1.4.5 Clickable lists

The scope specified on the annotation is `ScopeType.PAGE`, which instructs Seam to store the collection of tips in the JSF component tree. Since the data model is being stored in the JSF component tree, it is made available to any JSF action that is invoked from that page (resulting in a "postback").

The command link bound to the `#{tipAction.delete}` method expression is an example of an action that benefits from the propagation of this data model through the component tree. When the user clicks on one of the delete buttons, the data model is passed along with the event and JSF ensures that the activated row correlates with the appropriate item in that data model. This is where the complement to the `@DataModel` annotation, the `@DataModelSelection` annotation, is used. This annotation captures the selected row in the tips data model and injects it into the property over which it resides. With the work of assigning the tip from the activated row to a property of the component already taken care of, all the action listener method has to do is pass that reference to the JPA `EntityManager` to have it removed from the underlying database. Once again, no custom coding is required other than the business logic.

All that is left is to write a quick end-to-end test to ensure that we can save a new tip and that it can be subsequently retrieved.

1.4.6 Integration tests designed for JSF

The area of development that has routinely slowed down Java EE developers most often is testing. Even if you have never written a test, you are still testing. You test your code every time you redeploy your application or restart the application server to view the result of your latest modifications. It's just really slow and boring to do it that way. These days, testing is an integral part of any application development, and no framework is complete without an environment that allows you to test "outside of the container." Seam once again demonstrates its simplicity by exposing a single test class that can handle all of the integration testing needs in a Seam-powered application.

To make integration testing of JSF actions a breeze, Seam provides a base test class that sets up a standalone Java EE environment and executes the JSF life cycle within the test cases. The test infrastructure is driven by TestNG², a modern unit testing framework that can be configured using annotations. Although TestNG does not require you to inherit from a base test class, Seam's testing framework uses this approach to setup the fixture needed to bootstrap the embedded Java EE environment and the JSF context.

The test class `GolfTipsTest` in listing 1.4 simulates the initial request for the golf tips page and the subsequent form submission to add a new tip. The code in the test is invoked nearly identically to when it is used in the deployed application.

Listing 1.4 An end to end test of the golf tips application using the Seam test framework.

```
public class GolfTipsTest extends SeamTest {

    @Test #1
    public void testAddTip() throws Exception {

        new NonFacesRequest("/golftips.xhtml") {
            protected void renderResponse() throws Exception {
                Object value = getValue("#{tips}"); #2
                assert value != null && value instanceof DataModel;
                DataModel tips = (DataModel) value;
                assert tips.getRowCount() == 0;
            }
        }.run();

        new FacesRequest("/golftips.xhtml") {
            protected void updateModelValues() throws Exception {
                setValue("#{tip.author}", "Ben Hogan"); #3
                setValue("#{tip.category}", "The Swing");
                setValue("#{tip.content}",
                    "Good golf begins with a good grip.");
            }

            protected void invokeApplication() throws Exception {
                invokeMethod("#{tipAction.add(tip)}"); #4
            }

            protected void renderResponse() throws Exception {
                Object value = getValue("#{tips}"); #5
                assert value != null && value instanceof DataModel;
                DataModel tips = (DataModel) value;
                assert tips.getRowCount() == 1;
                List<FacesMessage> messages =
                    FacesMessages.instance().getCurrentMessages();
                assert messages.size() == 1;
                assert messages.get(0).getSummary() #6
                    .equals( "Thanks for the tip, Ben Hogan!" );
            }
        }.run();
    }
}
```

<Annotation #1> Designates a TestNG test method
<Annotation #2> Initial retrieval of tips, expecting 0
<Annotation #3> User filling out form
<Annotation #4> User clicking submit button
<Annotation #5> Subsequent retrieval of tips, expecting 1
<Annotation #6> Verify message with interpolated value

[Cueballs in code](#)

This tests both the initial rendering of the JSF view and the subsequent JSF action triggered from the rendered page. The first request is a HTTP GET request, which occurs when the user first requests the golf tips page. This part of the test verifies that when the tips are retrieved in the *Render Response* phase, Seam properly resolves a `DataModel`, but the collection underlying that model is empty. The second part of the test simulates the user submitting the form to create a new tip. The *Update Model Values* phase emulates the work JSF does to bind the input values to the value expressions. The method expression that is bound to the submit button is then forcefully invoked. Because Seam automatically wraps the *Invoke Application* phase in a transaction, there is no need to worry about beginning and committing the transaction. Finally, in the *Render Response* phase, the test verifies that when the tips are retrieved once again, that exactly one tip is found and that the message to display to the user is present and the author's name has been interpolated properly. This test is intentionally terse. Of course, there are many other cases that could be verified. Focus instead on how easy it is to exercise a Seam application using this simple test framework.

Hopefully the golf tips application has given you an idea of what Seam is all about. You should now have a general understanding of how Seam simplifies your application and saves you time by relying on a centralized container, annotations, configuration by exception and the unified EL. That is the essence of Seam. I now want to give you an idea of what else Seam offers you before you begin your journey down the road to becoming a Seam master.

1.5 Seam's core competencies

Throughout this chapter, there has been a lot of discussion about how Seam resolves issues in Java EE. I want to leave you with an understanding of how Seam is going to help your development process. Given how much Seam has to offer, this was a challenging exercise, but I have been able to summarize its benefits into three core competencies. Seam offers a better JSF, allows you to get rich quick, and fosters an agile environment.

1.5.1 Offers a better JSF

Seam does a lot to improve on the design of JSF. Although JSF isn't without flaws, it was selected as the main presentation framework in Seam because of its extensible request life cycle and strong UI component model. Seam taps into this potential and succeeds in making JSF a very compelling technology choice for creating web-based interfaces. While it's true that Seam supports alternative view technologies, this book primarily focuses on using Seam with JSF. Much of this coverage comes in chapter 3, which explains how Seam decorates and extends the JSF life cycle.

Seam's JSF enhancements

Seam's most recognizable improvement to JSF is eliminating the requirement to declare managed beans in the JSF descriptor. In addition, Seam adds a rich set of page-oriented functionality, covered in chapter 3, that makes the navigation rules in the JSF descriptor obsolete as well. These features include:

- pre-render page actions
- managed request parameters (for a given page)
- intelligent stateless and stateful navigation
- transparent JSF data model and data model selection handling

- fine-grained exception handling
- view-level security (per view ID)
- annotation-based form validation
- bookmarkable command links (solving the "everything is a POST" problem)
- entity converter for pick lists
- conversation controls
- prevents lazy initialization exceptions and non-transactional data access in the view

Part of the cleaning out process of JSF involved purging passive connector beans that did nothing more than adapt the UI with the backend action handlers.

Eliminating connector beans

Any Seam component can serve as a backing bean. Figure 1.7 shows the design of an interaction between a UI form and an EJB 3.0 session bean (or regular JavaBean) that completely eliminates the need for the connector bean. The form controls are bound directly to the entity class and the session bean handles the action to persist the data.

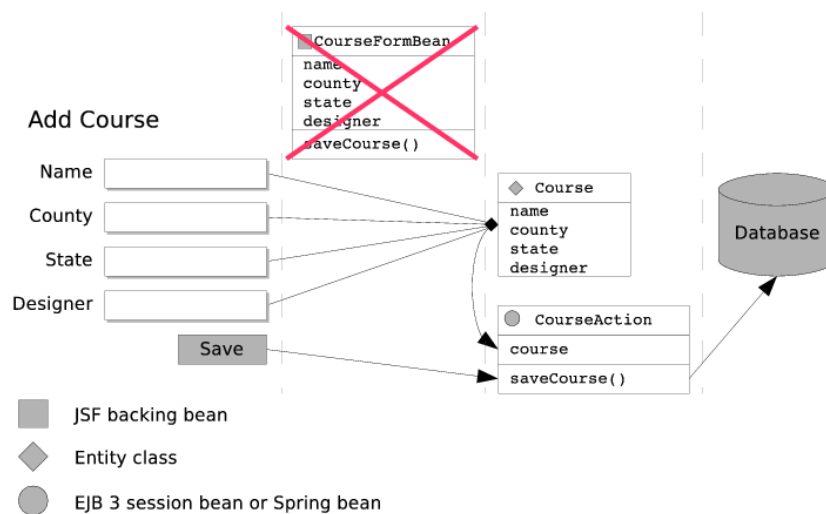


Figure 1.7 Seam cuts out the middle-man by eliminating the need for a JSF backing bean. Instead, the entity class and the EJB 3.0 session bean work together to capture data from the UI and handle the event to persist the data.

By cutting out the middleman, not only does Seam allow you to eliminate a class that you have to write and maintain, but it allows you to cut back on the number of layers, allowing your applications to become more light weight.

Aside from providing universal access to components, the Seam container augments the coarsely-grained scopes in the Java servlet specification—request, session and application—to include scopes that make more sense from the perspective of the application user. Seam offers two "stateful" contexts that are used to support single and multi-user pages flows in an application.

Stateful variable scopes

One of the main challenges with developing applications that are delivered over the web is learning how to efficiently propagate data from one page to the next—so called *state management*. The two

goto options are hidden form fields or the HTTP session. The first is cumbersome for the developer, the second eventually eats through precious server resources and hurts an application's ability to scale.

Seam addresses need for stateful variable scopes whose lifetime aligns with user interactions by adding the conversation context and business process context to the standard web scopes. The conversation scope, covered in chapter 7, maintains data for a single user across a well-defined series of pages while the business process scope, covered in chapter 14, is used to manage data that supports multi-user flows complete with wait states. The relationship between the lifetime of the scopes managed by the Seam container is illustrated in figure 1.8.



Figure 1.8 The lifetimes of the six scopes in a Seam application. The standard scopes are represented by dashed lines, while the scopes that Seam contributes are shown as solid lines. The business process scope is persisted to a database and can thus outlive the application scope when the application server restarts.

The conversation context is tremendously important in Seam not only because it is so unique and gives the user a better experience, but because it makes working with an Object-Relational Mapping (ORM) tool easier on the developer.

Extending the persistence context

When you talk to the database using ORM, you use a persistence manager (i.e. JPA `EntityManager` or Hibernate `Session`). Each instance of a persistence manager maintains an internal persistence context, which is an in-memory cache of entity instances that have been unmarshalled from the database. Given that databases are among the most expensive and heavily used resources in your server room, you want to leverage this in-memory cache as much as possible to avoid redundant queries. Extending the persistence context across the entire request is a step in the right direction (the so-called Open Session in View Pattern), but having it extend across multiple page requests is even better. Prior to Seam, there was just no good place to stick it, and as a result, each request reset the persistence context to a blank slate.

Seam takes control of the persistence manager, storing it in the conversation context, and is thus able to carry it, along with its persistence context, across the duration of an entire user case, potentially spanning more than one request, as shown in figure 1.9. Extending the persistence context across the three operations in this feature allows the entity instance to be retrieved only a single time. The remaining operations perform on that instance. This ensures object identity and can guarantee atomicity of the operation.

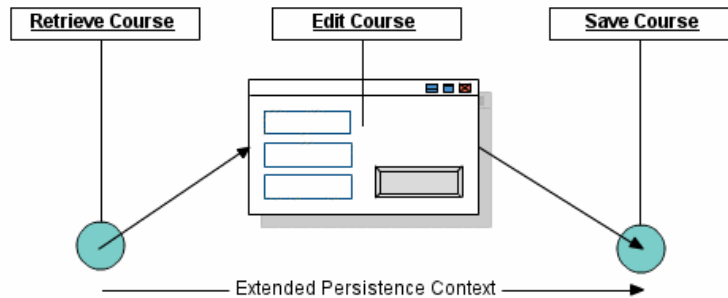


Figure 1.9 Using the extended persistence context to keep an object in scope for an entire use case, even across multiple page views. The extended persistence context avoids having to merge detached entity instances.

With Seam in control of the persistence manager, lazy initialization exceptions (LIE) are also a thing of the past since the persistence manager remains open throughout the use case and can thus load additional records as needed. The conversation and persistence context fit together so naturally that the conversation has been dubbed Seam's unit of work. You learn all about the interaction between the two in part 3.

1.5.3 Gets you rich quick

Seam gives you tools to build rich, Web 2.0 applications or to gently weave this richness into an existing page-oriented application. Lately, the term "rich" has become synonymous with a desktop-like experience in the web browser driven by Ajax. There are two approaches you can take to incorporate Ajax into a Seam application. You can use Ajax-enabled JSF components, such as RichFaces or ICEFaces, or you can invoke server-side components directly from the browser using JavaScript remoting. Seam extends the meaning of rich to incorporate media such as PDFs, charts, and graphics.

Tapping into the JSF ecosystem

Web user interfaces are getting more and more sophisticated and it is unreasonable to think that you can code the XHTML and JavaScript from scratch and get the job done cheaply. You need to build on what others have done. That is one of the primary goals of JSF and why Seam went with JSF as the primary user interface framework.

JSF is all about putting widgets on the screen. It decouples the design of a UI component from its use. Similar to widgets in Swing, JSF components are general solutions to common controls. This time, the vendors really did come through. There are loads of component libraries for JSF that range from basic data tables, to tree structures, to drag and drop targets. A sampling of these component libraries can be found in appendix C.

Historically one of the most entangled parts of an enterprise application is the UI (let's share hideous JSP files). By moving to JSF, the UI becomes a much simpler place. You really don't even need a WYSIWYG IDE because visualizing what these components render is quite reasonable. They are human-friendly, rather than tool-friendly. With JSF, the user-interface finally has an API too.

While JSF has its place, if you are looking for a lighter way to communicate with the server, Seam's JavaScript remoting library is a great alternative.

JavaScript remoting

Invoking server-side components from JavaScript in Seam couldn't be easier. You simply add the `@WebRemote` annotation to the Seam component method that you want to call from JavaScript, import the remoting library into the web page, and then invoke the component method using a JavaScript client stub of the component. Seam handles the rest. JavaScript remoting opens the door to creating single-page applications with Seam.

Although Ajax gets most of the attention these days when web-applications are discussed, there are other ways to make your application rich. These fall under the heading of rich media.

Creating rich media

Seam is very adept at generating a variety of rich media. Seam uses the Facelets view library to support alternate output based on XHTML templates, including PDF documents, RTF documents, charts, and multipart emails with attachments that include the previous items. With the addition of two JSF component tags, Seam can accept file uploads without any custom, low-level coding and can render dynamic graphics. All of these tasks are typically passed off by web frameworks to third-party libraries. While it's true that Seam leverages functionality provided by libraries such as iText and jFreeChart, the delegation is abstracted away. You are provided with a consistent approach, based on Facelets composition templates, that allows these features to be a native part of your Seam application.

1.5.3 Fosters an agile environment

In addition to being a framework, Seam also provides a collection of tools that help you setup a project, generate code, and develop in an incremental manner.

Project generator

One of the main highlights of Seam is its project generator, seam-gen. This tool serves two main functions. It sets up the structure of a Seam-based project, complete with a build script, environment profiles, and all of the libraries and configurations required to start developing your application. It's the best way to get started with Seam if you are new to the framework. The seam-gen tool can also create an application prototype by reverse-engineering a database schema and generating all of the necessary artifacts to create, read, update, and delete data in that database. In chapter 2, you learn all about seam-gen and use it to create a complete golf course directory web application.

Hot deployment

Seam goes to great lengths to enable "instant change" in the develop cycle. Seam initializes a hot redeployable classloader that is capable of reloading changed Java classes files just like they were JSP files. The project build script takes care of incrementally compiling the changed source files and shipping them off to the server. That means that you can change a Java file and a second later the change you made is reflected in the application. When the modified files are moved into the hot deployment area of the application server, the application server does not restart, nor does the application reload. This feature applies to Seam configuration descriptors and uncompiled Groovy scripts as well. You can finally match the change-view-change-view cycle that was previously only available with scripting languages such as PHP and Ruby.

Seam debug page

While developing your application, bad stuff happens. As a result, you get exceptions. Rather than always having to race to the log file to find the cause, Seam gives you a head start. When you run Seam in debug mode, any exception that occurs will be caught and summarized on a special debug page. In addition to the exception, this page gives you a snapshot of the JSF component tree and any Seam component instances that are present at the time of the exception.

You don't even have to wait for an exception to occur to use this page. When the debug page is accessed directly, it renders a list of all conversations and sessions that are currently active. You can then drill down on any of the active contexts to inspect the component instances that are stored in them.

Testing without deploying

The primary reason developers grew wary of the standard Java EE platform was because of its inability to operate in isolation. Testing an application meant packaging it up and shipping it off to a Java EE compliant application server, a costly process.

In order to work around this problem, developers adopted the POJO programming model, which encourages you to design code in such a way that it can be tested in isolation from the container and its services. While POJOs are definitely a good thing, and encourage proper unit testing, there is no replacement for integrating your components in a real environment to ensure that they work together. Previously, that meant deploying to the application server once again. Seam has a better solution.

To support integration test environments (and also deployment to non-Java EE containers, such as Tomcat), Seam ships with the embedded JBoss container. This portable container bootstraps a Java EE environment to support services such as JNDI, JTA, JCA, and JMS in a standalone environment. With these services up and running, you can test your application in place without having to deploy to a container. Seam supports this testing scenario by bootstrapping the embedded JBoss container as part of its single class integration framework, demonstrated back in section 1.4.6. This test infrastructure should prevent you from having to deploy over and over again to verify that your action handlers talk properly to your persistence layer and so on.

Between the incremental hot deployment support and the in place testing infrastructure, your valuable time should rarely be wasted when working on a Seam application. If its your business logic that is hanging you up, unfortunately there is not much Seam can do to help you there. That's all you.

1.6 Summary

The enthusiasm for Ruby on Rails was a real wake up call for the Java EE platform. It enlightened developers to the fact that sacrifice is not a prerequisite for creating a successful application. Developers no longer wanted to tolerate the burden of "XML situps"³ and over-engineered flexibility. In response, the Seam founders banded together best-of-breed Java EE technologies and created an agile platform that takes a bold stance against Java EE's formalities, cutting back the XML descriptor overgrowth, accentuating the platform's recent adoption of annotations and configuration by exception, and embracing the expressive syntax embodied by the EL, Facelets, and Groovy. With Seam, creating applications in Java becomes exciting again, whether you are a front-end designer, back-end developer, or jack-of-all-trades. Best of all, you can be confident that applications built with

Seam are scalable because the Java EE platform has proven itself in this regard, giving you productivity without sacrificing performance.

First and foremost, Seam makes the task of defining and accessing stateful business-logic components simple, regardless of whether they are EJB or non-EJB components. A basic `@Name` annotation atop a class gains it admission into Seam's contextual container. The container wrap these components in method interceptors, enabling enterprise services, such as transactions, security, and component assembly, to be declared with equivalent ease by applying an annotation at the class, method, or field level. Seam grants the technologies that it integrates access to the components in this container, primarily through the use of the unified EL. This arrangement facilitates the use of JPA entity classes as "backing" beans in a JSF form, EJB session beans or transactional JavaBeans as action listeners on a JSF UI component, and variables to be resolved on demand using Seam's factory or manager mechanism.

An important aspect of the container is its state management capabilities. It consolidates the variable scopes in JSF with two of its own business-oriented scopes. Seam understands variable scoping and helps components from different scopes to work with one another without violating thread-safety. Of particular note, Seam can extend the lifetime of the persistence manager across multiple page requests to reduce load on the database and eliminate complexities with using ORM in web-applications.

If you picked up this book because you believe that there is a better framework choice out there for you (and you are not yet using Seam), my promise to you is that Seam is worth checking out and that the time you spend reading this book will be worthwhile. But, merely knowing what framework someone recommends is not enough to decide to use it. You have to know *why* a person prefers a particular framework. In this book, I share with you my extensive knowledge of Seam and explain to you why I find it to be a compelling choice. As you read along, I encourage you to develop your own reason for choosing Seam.

The key to agile development with Seam begins with the project generator, seam-gen. In the next chapter, you learn how to use this tool to develop an entire application from scratch with only a handful of keystrokes. Once the application is in place, I walk you through the project structure, show you how to get it setup in your IDE, and demonstrate incremental hot deployment. While you must turn over some control when you opt to go with seam-gen, you quickly find that you don't miss the work.