

Covers Seam 2.0

SEAM

in Action

Dan Allen

MEAP

Unedited Draft

 MANNING



Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

Table of Contents

Part 1 - Teeing off with Seam

Chapter 1 Seam unifies Java EE

Chapter 2 Putting Seam Gen to Work

Part 2 – Seam fundamentals

Chapter 3 The Seam life cycle

Chapter 4 Components and Contexts

Chapter 5 The Seam component descriptor

Chapter 6 Absolute inversion of control

Part 3 – How Seam manages state

Chapter 7 The conversation: Seam's unit of work

Chapter 8 Understanding Java persistence

Chapter 9 Seam-managed transactions and persistence

Chapter 10 Rapid Seam development

Part 4 – Sinking the business requirements

Chapter 11 Securing Seam applications

Chapter 12 Ajax and JavaScript remoting

Chapter 13 Managing business processes

Chapter 14 File, email and rich rendering support

Chapter 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

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

The conversation: Seam's unit of work

After returning home from your golf getaway, you learn that, in your absence, someone has been having fun at your expense, swiping your credit card around town. To reclaim your assets, you have to endure the disparate customer service process deployed by your bank. I'm sure this exchange will sound eerily familiar.

"How can I help you?"

This is your cue. You launch into your rant about being a victim of fraud, your whereabouts during the previous week, and which charges you are disputing.

Then silence.

"Can I have your account number?"

Your momentum is temporarily interrupted as you cut through the red tape. The agent then informs you that you need to be transferred to the fraud department, which is more equipped to deal with this matter. There is no chance to object as the switch happens without delay. A new voice appears on the other end of the line.

"How can I help you?"

Sigh. Time to start over from the beginning.

You started the day as a victim of fraud. Now you have become the victim of a stateless process. Critical information failed to make the leap between the customer service representative and the fraud representative. This mishap could have been avoided had the two representatives engaged in a conversation during the switch to retain the record of your story. Keeping track of state within the context of a use case is referred to as stateful behavior, something that would surely benefit this process.

In this chapter, you learn how Seam encourages stateful behavior through the use of web conversations. A conversation encompasses two forms of state to help manage a use case, data and navigation, as illustrated in figure 7.1. A use case represents the ultimate goal the user is trying to accomplish, not just an atomic stage along the way.

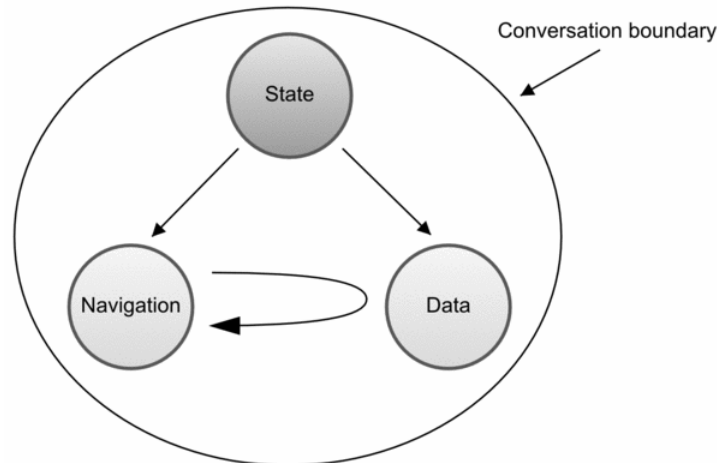


Figure 7.1 A conversation is a means of holding state. State is divided into data and navigation.

The chapter begins by establishing what it means to have a conversation in a web application and the difficulties of propagating a working set of data in the stateless HTTP environment. You are then introduced to Seam's conversation scope as a convenient place to store data across page views, even in light of ad-hoc requests. You learn the multitude of ways to control the boundaries of a conversation through the use of Seam directives. You then learn how combine a conversation with a stateful page flow to constrain the user to a predetermined track for the duration of a use case. Page flows represent the navigational aspect of conversations, which draw on data available in the conversation to make navigation decisions, as figure 7.1 suggests.

By the end of the chapter, you will have learned how to make effective use of the conversation state to retain short-term, yet critical, information that users provide, keeping them content and in stride. Let's begin by exploring ways in which state is maintained in a web application and how Seam can help manage the task.

7.1 Conversational state in a web application

One of Seam's primary goals is to focus on the user experience and to that end, conversations are a critical part of how Seam functions. In this section, you learn how Seam redefines an application's unit of work to be that of a conversation, thus giving the use case—a single user's interaction with the application—a formal representation within the framework.

As part of establishing this unit of work, the application must store a working set of data that supports the goal at hand. The traditional approaches to tracking state in web application are contrasted with Seam's conversations. The conversation context promises to relieve the burden of tracking state and even opens the door to more sophisticated functionality such as nested conversations and parallel workspaces. Let's see how this new context helps to focus the application on the user's interaction.

7.1.1 Redefining the unit of work

In technical terms, a unit of work is often defined to be an atomic database transaction. The story told at the beginning of this chapter is faithful to this definition at the expense of the caller's time. But

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

technical definitions don't solve a user's problem. The customer service representative records the call in one system and passes the caller off to a different system, writing off the job as finished. Clearly, this short-lived unit of work is not conducive to a good user experience. It is not stateful, meaning it is too fine-grained and does not propagate its memory. As far as the user is concerned, the use case is far from over at this point. From the user's standpoint, the unit of work extends from the time the call is made to when it is complete, which encompasses the interaction with the first and second representative and ultimately the resolution.

Conversations reflect the way that a user perceives the application. In a conversation, database transactions, page requests, and other fine-grained units of work may come and go, but the use case does not end until the goal at hand is accomplished. In a web application, these smaller units of work are divided up by individual page views. A conversation represents a linkage between two or more page views, as depicted in figure 7.2. This linkage is established through the use of a special token and a creative use of the HTTP session, which you learn about later in the chapter. The conversation scope is longer than the request scope but (much) shorter than the session scope. How long the conversation lasts is determined by the boundary conditions of the use case. In section 7.3 you will learn how these boundary conditions are defined.

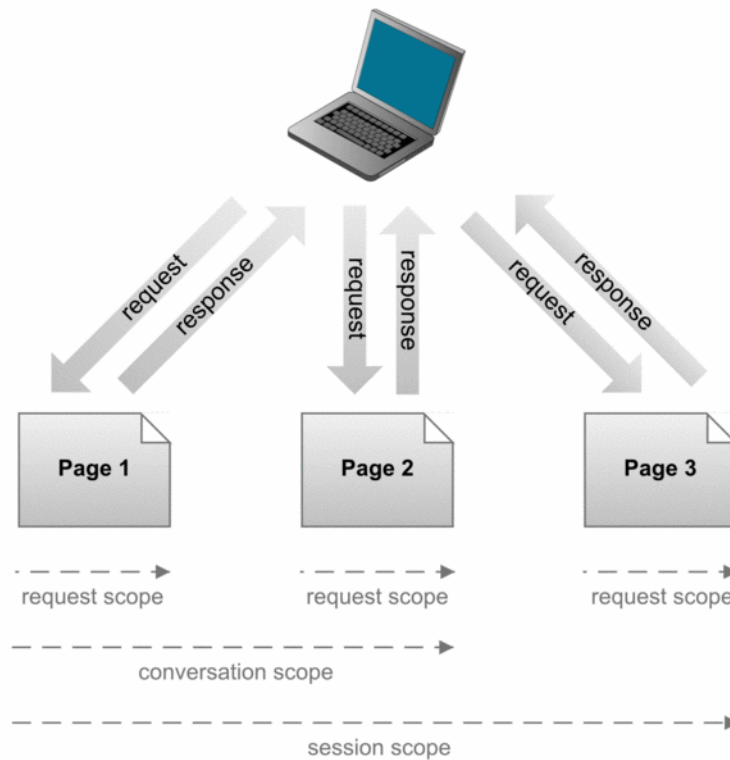


Figure 7.2 The conversation scope spans multiple page views but does not last as long as the session scope.

To put theory in to practice, let's consider an example whereby a conversation is used to allow the user to search and compare golf courses. The user starts on the course directory page, which is used to filter the list of courses for the purpose of selecting a course. Depending on the mode held in the conversation scope, the selection either results in the detail screen being displayed or it queues the

course to be compared. When the user is ready to compare the course, the Compare button is clicked and the comparison screen is displayed, showing each selection side by side. Upon returning to the directory page, the user finds that the selections have been retained. As long as the user stays within the boundaries of the conversation, the use case is considered active and the application remembers its state, which is stored in a working set of data. Let's take a closer look at the working set and consider ways it can help support a use case.

7.1.2 Building a working set of data

The need for a working set arises when the application must perform work that spans multiple requests—which Seam terms a conversation. To provide continuity, data is stashed into the conversation's working set during user "think" time—the time after the response is sent to the browser but before the user activates a link or submits a form. Information is accumulated in the working set as the user moves from screen to screen.

There are four general categories of data that a working set is used to store, all of which we be demonstrated as the chapter proceeds:

- *non-persistent data* – An example of non-persistent data is a search criteria or a selection of records. The user can establish the state in one request and then operate on it in another.
- *transient entity data* – A transient entity instance may be built and populated as part of a wizard and thus won't be ready to be persisted until the final screen is reached. Once the wizard is complete, the transient entity instance is drawn from the working set and made persistent.
- *managed entity data* – The working set provides an ideal way to work with database-managed entity data for the purpose of updating its fields. The entity instance is placed into the working set and then overlaid on a UI form when the user clicks on Edit. When the user clicks Save, the changes made to the form are applied to the entity instance stored in the working set (whose object identity has been preserved) and the instance is synchronized with the database.
- *resource sessions* – No example is more relevant to developers than the persistence context (JPA `EntityManager` or Hibernate `Session`). As part of a working set, the persistence context can be kept open to prevent entity instances from becoming detached during the course of the use case. The next three chapters focus on how conversations benefit persistence.

Having stateful data close at hand sounds convenient, but managing it with the Java Servlet API alone is different story. Far too much effort has gone into the task of carrying information from one page to the next since the dawn of web applications. In fact, propagating state has become a downright burden. Let's weigh the existing options and then see how Seam's conversation scope is able to offer transparent management of this state without the headache.

7.1.3 The burden of managing state

To support conversations, there must be a place for the working set to live and a means to propagate it from one page to the next. The goal of migrating data from page to page is not a new concept in web-based applications. In fact, it's often one of the most frequently discussed topics in the design phase or

thereafter. I'm sure just about every web developer has used one or more of the following choices from the Java Servlet API or HTTP protocol at one point or another to send data along to the next request:

- hidden form fields
- request parameters (i.e. the query string)
- HTTP session attributes
- cookies

I'm not going to stand here and say that none of these options are viable. Obviously, all of them have been used numerous times in the past to create functional applications. But, they each have their limitations that seek to slow you down on every project you take on, bringing along with them varying degrees of pain. After looking at some of the problems inherent in these existing options, you will learn about a fifth option, the conversation scope, which seamlessly ties together the state from one request to the next.

Passing data through the request

Hidden form fields and URL parameters are merely two sides of the same coin, both applied through the use of HTTP request parameters. The former is used when passing data through a form submission and the latter when appending data to a URL in a link. All the data is transferred out in the open as string values.

You learned in chapter 3 how Seam helps to propagate data from page to page using page parameters. While page parameters are helpful at times, if used in excess, managing them can quickly become as burdensome as managing hidden form fields. Since page parameters are hardwired to a given page, you end up having to mix parameters need to serve different use cases together. In addition, you run into situations where, when you link to a page with page parameters, you get more than you bargained for and it becomes hard to shake off the unwanted state.

The main weakness of request parameters is that the values are disassembled as they cross the request boundary and then reassembled once they arrive at the server, as depicted in figure 7.3. However, when an object passes through this funnel, it loses its identity. What exists on the server after the request is a clone of the original object, possibly even a partial one. This makes it impossible to transfer managed entity instances and resource sessions, which cannot be serialized into string values without jeopardizing their integrity.

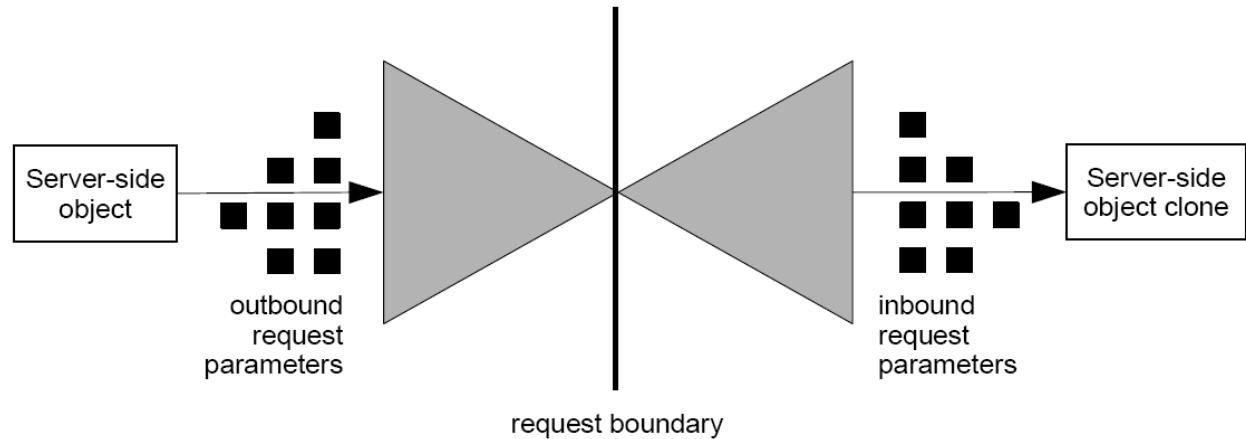


Figure 7.3 Passing an object using request parameters is akin to teleportation.

If maintaining object identity is important, or the values being propagated cannot be broken down into chunks to be passed through this funnel, it is necessary to keep them intact by storing them in the HTTP session.

Storing data in the HTTP session

The HTTP session can be used to store arbitrarily complex values in their original state (maintain object identity) and have those values be available to any subsequent request made by the same browser. While this sounds ideal, it's unfortunately too good to be true. The main downfall of the HTTP session is that it quickly becomes a tangled mess and can interfere with multi-window application usage.

Consider the case in which the session is used to store the golf course entity instance while the record is being updated in an editor. If you select a course to be edited in one browser tab (or window) and then select a different course to be edited in another browser tab, when you click Save in the first tab, you may inadvertently apply the changes to the wrong course instance. Things get even trickier if you are working with a multi-page wizard, since the leakage of data can be less apparent. It's possible to work around these problems by using the session carefully or putting in extra validation checks to ensure that cross-tab leakage is not occurring, but that is more of a burden on you as a developer.

The session scope has other problems due to the fact that it is unmanaged. If objects continue to build up in the session, and there is no application-provided garbage collection routine in place to clean them up, it can lead to memory leaks. Access to objects in the session are also not protected from concurrent use. In general, heavy use of the session scope is a common source of bugs and the unexpected behavior caused by state collision can be very difficult to reproduce in a test environment.

Although not discussed here in detail, cookies share the combined problems of request parameters and session data. They can only store string data, and even then only a limited size of data, and they are not easily partitioned by use case. While these existing storage options are workable, they are not well suited to serve a use case. Clearly there is room for a better solution. For as much as I just discredited the HTTP session as an option for storing state, it's not all bad. It just needs to be better managed. That's exactly what Seam does to give rise to the conversation context.

7.2 The conversation context

As presented in chapter 4, the conversation context is one of two contexts Seam introduces to serve business-world time frames as opposed to servlet life cycles (the other being the business process context). From reading the previous section, you should have a very clear picture as to the purpose of the conversation context for serving a multi-page user interaction. In this section, you learn of the conversation's identity.

7.2.1 Carving a workspace out of the HTTP session

The conversation context is carved out of the HTTP session to form an isolated and managed memory segment, as illustrated in figure 7.4. Seam uses the conversation context as the home for a working set of context variables.

You may shudder at any mention of using the HTTP session for data storage, given the problems that were cited in the last section. But, you should not confuse a conversation's working set with that of session attributes. What makes a conversation different from its parentage is that it has its own distinct life cycle. It is much shorter lasting than session (typically on the order of minutes), capable of being terminated or timing out separately from the session in which it lives, pertains only to the current context, and is kept isolated from other conversations in the same session so as not to interfere with them.

The key distinction between the session and a conversation is that a user can have many conversations while there is only one HTTP session. From the user's vantage point, a working set is viewed as a *workspace*. It represents state for one of the multiple tasks the user may be performing at any given time.

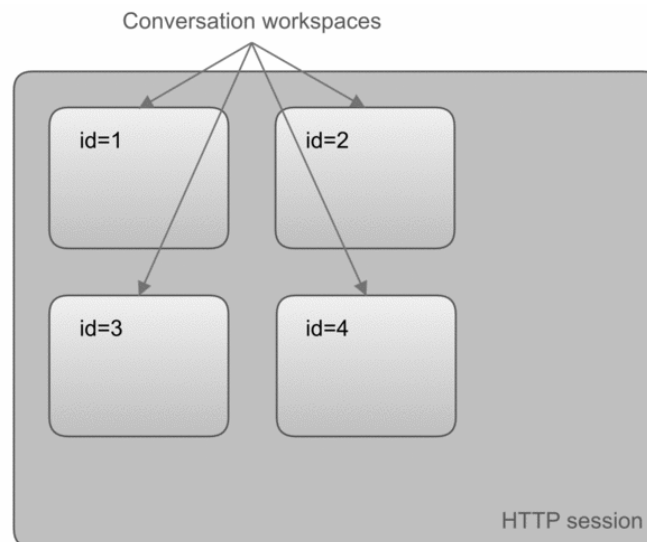


Figure 7.4 Conversation workspaces are isolated segments of the HTTP session, each assigned a unique identifier.

A conversation has a clear set of life cycle boundaries that typically coincide with a single use case, represented by the interaction with one or more JSF view IDs—or pages. You will learn how to define these boundaries in the next section. When the user triggers a condition that begins a

conversation, a new area of the HTTP session is sectioned off and dedicated to that conversation. A unique identifier, known as the *conversation id*, is generated and associated with this segment of the session. This token is passed as a request parameter, hidden form field, or JSF view root attribute to retrieve the memory segment and associate it with the current request.

The conversation context is the ideal place to store data that is needed over the course of several pages. It enables you to hang on to that data, keeping it in scope while you are working on it. All you have to do to use it is keep track of the single conversation id request parameter, which you will learn is handled for you automatically in most cases. Storing data in the conversation context has the same advantage of using the HTTP session in that arbitrarily complex objects can be preserved without them losing their object identity, unlike with hidden form fields or URL parameters. This enables you to easily migrate objects, such as the persistence manager, between requests. By the same token, the conversation context does not get out of hand—it does not become a tangled mess or suck up tons of memory.

What makes conversations truly unique is that they can coexist with one another, allowing the same context variable names to be used in isolated memory regions.

Solving the concurrency problem

The HTTP session is notorious for wrecking havoc on multi-window behavior. The session identifier is stored as either a cookie in the user's browser or appended to each form target or link URL generated by the application (URL rewriting). Many browsers will share cookies between tabs and windows—the viewport herein referred to as a tab. When multiple tabs naively share the same session data, it can quickly lead to confusing and undesirable behavior as the same data is often manipulated in conflicting ways.

NOTE

Even when URL rewriting is used, spawning new tabs from links may carry over the session identifier. The result is that each tab is accessing the same HTTP session on the server.

Conversations do not suffer the same leaky behavior as session data across multiple tabs because, in addition to requiring the session identifier, they also require that a conversation id be sent with the request. The conversation id gives the request access to an isolated area of the session.

Let's revisit the scenario in which different golf course records are being edited in separate tabs, but this time consider how the conversation context can be used to keep the entity instances isolated. The user selects a golf course in the first tab, which starts a new conversation and presents the user with an editor form. Then the user switches to the second tab and selects a different course, again resulting in a conversation being created the rendering of an editor form. Each tab has its own conversation. Next, the user switches back to the first tab and clicks Save. The updates are sent to the server along with the conversation id which was automatically added to the form—technically it is stored as an attribute in the JSF view root. The conversation id is used to activate the conversation created by that tab, the entity instance is retrieved from the working set in that conversation, the changes are applied to the instance, and it is synchronized to the database.

Although the two tabs are serving the same use case, with the same context variables, data is not shared between them. When the record from the first tab is saved, the state held by the conversation associated with the second tab is not affected. The important point to be made is that these context

variables do not collide. Section 7.4 demonstrates how it is possible for the user to toggle between concurrent workspaces using a conversation switcher component.

Without the help of conversations, it becomes a real burden on the developer to track data, keep it in scope, and prevent it from interfering with other parallel activity in the user's session. But conversations are more than just working sets of context variables. Let's consider the other benefits that come about as a result of having stateful behavior.

7.2.2 More than just a context

In one sense, the conversation is just another one of Seam's contexts whose boundaries are controlled by the application logic. Aside from just providing a place to store context variables, the conversation can be used to accomplish broader goals thanks to its awareness of the application's state.

Starting from the beginning

Conversations are closely tied to navigation. The pages involved in a conversation can either be requested ad-hoc without a definitive navigation path or they can be linked together as part of a page flow. In either case, you can restrict access to certain pages and components in the event that a conversation is not active.

If you are using stateless navigation, you can redirect the user to a starting page if they try to access a page that requires a conversation to be active. If stateful navigation is being used, a similar restriction can be enforced. Stateful navigation gives the added control of constraining the user to a given page sequence with the boundaries of an active conversation. As you can see, conversations are useful both from a state management perspective and as a navigation and page flow device. (some rework needed)

Business caching

Conversations can also serve as a domain-specific cache. A point brought up earlier, stateful architecture ensures that data retrieved from persistence storage or service endpoints is maintained until the application is ready to relinquish it. That means there is no need to repeatedly consult the data provider for same static information. You look up the data once and then hang on to it until you consider it to be stale. I refer to this natural caching mechanism as the *business cache*.

Generic caching systems, such as a second-level ORM cache, must perform their work based on an intelligent algorithm that will allow them to hold data for as long as possible, but not so long that it becomes inconsistent with the underlying data storage. Your application, on the other hand, knows intimate details about the data and can perform checks relevant to the current use case. In short, there is no one more suitable for deciding when to keep data and when to throw it away than your own business logic. You can even provide the user with controls to help the application decide when data should be fetched again. Keep in mind that using a second-level ORM cache for maintaining conversational data is a misuse of the technology.

You should take advantage of the conversation scope to reduce load on the database. Of all the tiers in your application, it is the *least* scalable tier. You don't want to abuse it. Don't make the database pay for the application failing to track data that it was already fed. A server-side memory will provide much needed relief to the database. Especially in the world of Web 2.0 in which Ajax requests

are fired off to the server at a rate that far exceeds the previous usage pattern of web applications. Ajax brings up another important issue that is often ignored, synchrony.

Serialized requests

Seam serializes concurrent requests that access the same conversation. This means only one thread is allowed to access a conversation at any given time. Therefore, you can be confident that data stored in the conversation will not be modified by another incoming request while the original request is being served. This safety net may not be so important in pre-Web 2.0 applications, but when Ajax starts firing off requests like they are going out of business, the likelihood of having a collision that causes the data to enter an inconsistent state dramatically increases. Because Seam serializes access to the conversation, you do not have to worry about such collisions. It's one at a time.

The combination of serialized access and stateful behavior drastically minimizes the risk of using Ajax in your application. With these mechanisms in place, you can rest assured that performance and data consistency will not suffer. Conversations fit so naturally with Ajax that you will likely discover that no other solution is as viable. You will learn more about using Ajax with Seam in chapter 12.

Ajax isn't the only Web 2.0 mechanism that benefits from stateful behavior. Web services can also tap into the conversation scope.

Stateful web services

Conversations can be used for interactions that don't have access to the HTTP session, or which have traditionally been stateless. The conversation id can be passed along with the web service header to allow subsequent requests to access the state accumulated in the HTTP session on the server. A web service can either be the formal definition or a more lightweight interface such as a RESTful URL. Either way, subsequent requests can be linked to the same server-side working set.

In this section you learned what we mean when we say "conversation": a context for keeping data in scope for the duration of a use case and a means of enabling stateful behavior in your applications. The next step is to learn about the conversation life cycle and then how to control conversations by defining conversation boundaries.

7.3 Establishing conversation boundaries

The conversation context is unique from other Seam contexts in that it has explicit boundaries dictated by application logic, as opposed to implicit boundaries that correlate with some demarcation in the servlet or JSF life cycle. The application logic controls the boundaries of the conversation context by using conversation propagation directives. This section introduces these directives and demonstrates how they can be used to control the life cycle of a long-running conversation.

7.3.1 Conversation propagation

There are actually three conversation states: temporary, long-running, and nested. Switching the state of a conversation is referred to as conversation propagation. We are going to focus on nested conversations later on. For now, I want to expound the distinction between a temporary and a long-running conversation. When you set the boundaries of a conversation using the conversation

propagation directives, you are not initiating and destroying the conversation, but rather transitioning it between the temporary and long-running state.

Temporary versus long-running conversations

Most of the time, when people talk about Seam conversations, they are talking about *long-running* conversations. The discussions in the early part of this chapter all have to do with long-running conversations. As described, a long-running conversation remains active over a series of JSF requests.

In the absence of a long-running conversation, Seam will create a *temporary* conversation for the purpose of serving a single request. A temporary conversation is created immediately following the *Restore View* phase of the JSF life cycle and is destroyed after the *Render Response* phase.

You can think of a temporary conversation as serving the same purpose as the flash hash in Ruby on Rails. In that regard, the role of the temporary conversation is to transport conversation-scope context variables across a navigation redirect. As a primary example, the temporary conversation is how Seam manages to keep JSF messages alive during the redirect-after-post technique, as long as they are added using the conversation-scoped `FacesMessages` component. So, the temporary conversation really is destroyed only after the *Render Response* phase, even if a redirect proceeds it.

The other purpose of a temporary conversation is to serve as a seed for a long-running conversation. What you will learn is that a long-running conversation is nothing more than a temporary conversation whose termination has been suspended—as dictated by a *begin* conversation directive—until an explicit *end* conversation directive has been encountered. Instead of just surviving a navigation redirect, a long-running conversation is capable of surviving a whole series of user interactions. Only when it reacquires its designation as temporary can it be terminated. Therefore, learning to use long-running conversations is about learning how to use the conversation propagation directives and how they transform a temporary conversation to a long-running conversation and back.

The conversation life cycle

The life cycle of a conversation is controlled using the conversation propagation directives. There are five conversation propagation directives, as shown in table 7.1.

Table 7.1 A list of the conversation propagation directives.

Propagation type	Description
begin	Promotes a temporary conversation to a long-running conversation. If a long-running conversation is already active, an exception will be thrown.
join	Promotes a temporary conversation to a long-running conversation if a long-running conversation is not already active. An exception will not be thrown if a long-running conversation is already active.
end	Demotes a long-running conversation to a temporary conversation.
nest	If a long-running conversation is active, suspend it and add a new, long-running conversation to the conversation stack. If a long-running conversation is not active, promote the temporary conversation to a long-running conversation.
none	Abandon the current conversation. The previous conversation is left intact and a new, temporary conversation is created to serve the request.

The *begin* and *join* directives are logistically the same, the *join* directive merely asserting that a long-running conversation is not active before converting the temporary conversation into a long-

running one. The last two directives allow you to step out of the context of the current conversation and are covered in section 7.5.

The conversation propagation directives can be applied using any of the following:

- Method-level annotations
- UI component tags
- Seam pages descriptor
- Seam conversation API
- Stateful page flows (jPDL) (*end conversation only*)

All of these options will be presented when exploring the conversation directives. You are given a plethora of options so that you can associate the conversation boundary with the most logical point in the code execution, whether it be the invocation of a component method, the activation of a link or button in the UI, the request for a page, or, if none of those options apply, an arbitrary point as controlled using the Seam conversion API.

What this flexibility means for you is that you don't have to go out of the way to define conversation boundaries. Hopefully, though, you will establish a favorite approach and adhere to it a majority of the time. Just because you have all of these options doesn't mean you should use every last one of them.

Figure 7.5 diagrams the conversation life cycle, which shows how the state of the conversation may change during the processing of the request as a result of a conversation propagation directive being encountered.

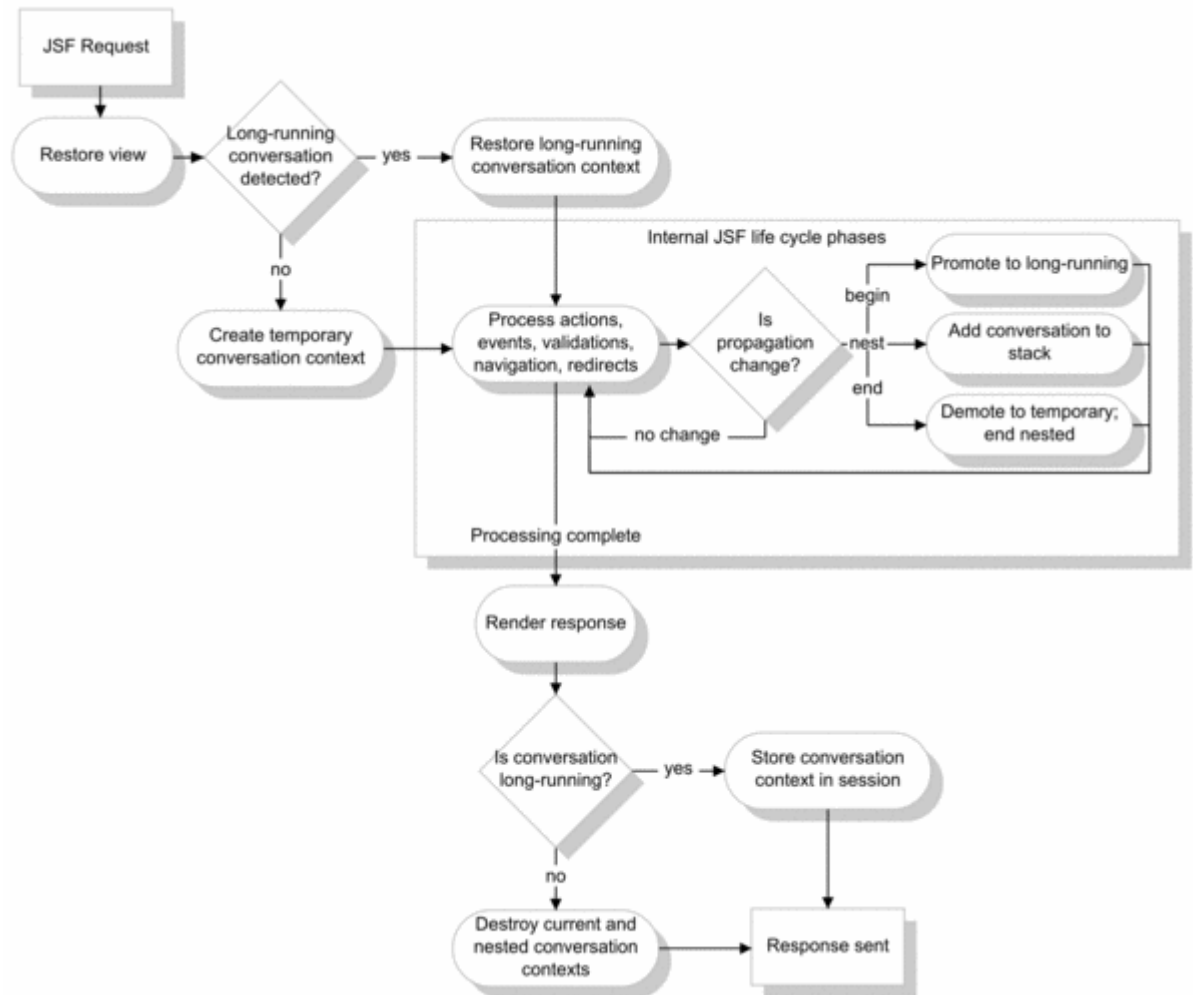


Figure 7.5 The conversation life cycle over the course of a single JSF request. The propagation directives toggle the current conversation between the temporary and long-running state.

An existing long-running conversation is restored if its conversation id is detected in the request. Although not shown in this diagram, if the long-running conversation being restored is invalid or has previously timed out, Seam either forwards the user to a fall back page or silently ignores the restore request. If a long-running conversation is not detected, or it is invalid, Seam creates a temporary conversation. At any point during the processing of the request, a conversation propagation directive may be encountered. If this happens, the temporary conversation may be promoted to a long-running conversation or vice versa. At the end of the request, the temporary conversation is destroyed, whereas the long-running conversation is tucked away in the session to be retrieved by a subsequent request.

NOTE

Pay particular attention to how a conversation is ended, as this can be a bit difficult to grasp at first. The term "end" is deceptive. When a conversation is ended, it isn't destroyed outright. Instead, it is demoted to a temporary conversation to be terminated at the end of the request, after the view has been rendered. Demoting a conversation does not result in the context being cleared. Therefore, the conversation-scoped context variables that were available during the long-running

conversation are still available in the *Render Response* phase that immediately follows the demotion of the long-running conversation. If you want to part ways with the conversation altogether, you set the "before redirect" flag on the end conversation directive and then issue a redirect after demotion has taken place. Having become a temporary conversation, it will not last the redirect and the next page view will use a fresh conversation.

Now that you have a clear picture of the long-running conversation and its origin, let's look at how you set the boundaries of a long-running conversation with the conversation propagation directives, starting at the beginning.

7.3.3 Beginning a long-running conversation

To demonstrate the use of a long-running conversation, we will work through several examples. The first example is a multi-page wizard that is used to capture information about a golf course and add it to a directory. A complete golf course record can be very intimidating, so the wizard format is used to break up the form into bite-size pieces as shown in figure 7.6.

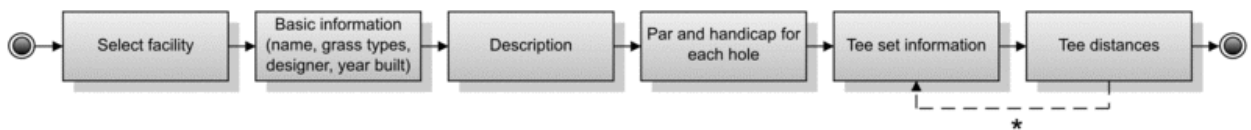


Figure 7.6 The screens in the wizard for entering a new golf course into the directory.

Each box in figure 7.6 represents a screen containing a form to fill out. As the user moves from screen to screen, the information from previous screens must be accumulated and stored so that it is available when the final screen is complete and the course is persisted to the database.

The textbook choice for starting conversations such as the golf course wizard is to add the `@Begin` annotation to the action handler method that spawns the wizard. However, there may be times when you need to start a conversation from a GET request, in which case either the `<begin-conversation>` pages descriptor tag or a UI component tag is a more appropriate choice. The later two may be attractive to you if you prefer to keep your navigation controls out of Java code or you need more fine-grained control over the conversation boundaries. As you read through this section, keep in mind that you only need to begin the conversation once, so these options are mutually exclusive (unless conversation joining is enabled, which is covered at the end). Let's begin with Seam's pride and joy, the annotation-based approach.

An annotation-based approach

One of the interceptors that Seam wraps around components is the `ConversationInterceptor`. This interceptor looks for the `@Begin` annotation on the method and if it finds one, it converts the temporary conversation to long-running after executing the method. The `addCourse()` method on the `CourseWizard` component may be used to begin a long-running conversation for the course wizard. Invoking the `addCourse()` method will in turn outject the `newCourse` context variable to be used by the JSF form:

```
@Name("courseWizard")
@Scope(ScopeType.CONVERSATION)
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```

public class CourseWizard implements Serializable {
    @Out private Course newCourse;

    @Begin
    public void addCourse() {
        newCourse = new Course();
    }
}

```

In the first variation, the long-running conversation will be started as a direct result of a user interaction by attaching the `addCourse()` method to a JSF command button component tag:

```

<h:commandButton action="#{courseWizard.addCourse}"
    value="Add course..."/>

```

To advance to the first page in the wizard requires the following navigation rule:

```

<page view-id="/CourseList.xhtml">
    <navigation from-action="#{courseWizard.newCourse}">
        <render view-id="/selectFacility.xhtml"/>
    </navigation>
</page>

```

Instead of using a JSF command button and a navigation rule, you could trigger the `addCourse()` method using a page action from the first screen in the wizard:

```

<page view-id="/coursewizard/selectFacility.xhtml"
    action="#{courseWizard.addCourse}"/>

```

In this case, when the servlet path `/coursewizard/selectFacility.seam` is requested, the `addCourse()` method is invoked and a long-running conversation is started. The benefit of using a page action is that it can start a long-running conversation from a bookmarked page or direct link, rather than relying on a JSF postback.

NOTE

There are two conditions that prevent the `@Begin` annotation from starting a long-running conversation. If the action handler method has a non-void return type and the method returns a null value, then the `@Begin` annotation is ignored. The conversation also does not begin if an exception is thrown. In all other cases, the conversation will begin. The same rules apply to methods marked with the `@End` annotation, which is used to demote a long-running conversation.

The `@Begin` annotation is outlined in table 7.2. The `pageflow` attribute is used to initiate a stateful page flow and is covered in section 7.4. The `flushMode` attribute is used to set the flush mode of the persistence context while the conversation is active and is covered in chapter 9.

Table 7.2 The `@Begin` annotation marks the start of a long-running conversation.

Name:	<code>Begin</code>
--------------	--------------------

Purpose: Indicates that a long-running conversation should begin when this method is invoked. To take effect, the method must complete successfully and have either a void return type or return a non-null value.

Target: METHOD

Attribute	Type	Function
join	boolean	A true value allows this method to be invoked even when a long-running conversation is active. Without this override, an exception is thrown indicating that a long-running conversation has already begun.
nested	boolean	A true value causes the active long-running conversation to be suspended and starts a new, nested long-running conversation. The default is to begin a non-nested conversation. This attribute is mutually exclusive with join.
pageflow	String	The name of the jPDL page flow for used to manage this conversation.
flushMode	FlushModeType	Changes the flush mode of the managed JPA <code>EntityManager</code> or Hibernate <code>Session</code> for the duration of the conversation.

The `@Begin` annotation isn't limited to methods acting as action listeners. You can also combine this annotation with either a life-cycle annotation, such as `@Create`, or the `@Factory` annotation. In either case, the long-running conversation is started at an arbitrary point in the Seam life cycle. But, as you know, it is never to late to begin a long-running conversation.

As an example, you could define a factory method for the `newCourse` context variable, which would be referenced in the first screen in the wizard, and designate it as the start of a long-running conversation:

```
@Out private Course newCourse;

@Begin
@Factory("newCourse")
public void initCourse() {
    newCourse = new Course();
}
```

Factory methods are a good place to start conversations because they don't require user interaction, but it doesn't require any XML to be defined in the pages descriptor either. The same goes for the `@Create` life cycle method. You can also have the conversation begin the very first time the `CourseWizard` component is used, regardless of which action listener method is invoked:

```
@Begin
@Create
public Course initCourse() {
    newCourse = new Course();
}
```

Now, if you prefer your code with a side of angled-brackets, you might instead find the pages descriptor to be an ideal place to begin a long-running conversation.

A page-oriented approach

You just witnessed how it is possible to start a long-running conversation by using a page action to execute a method annotated with `@Begin`. This task is so common that Seam includes a built-in page action in the form of either the method binding expression `#{conversation.begin}` or the `<begin-conversation>` tag. In addition to page requests, the `<begin-conversation>` tag can be used to begin a conversation during a page transition, where a page action cannot reach.

Let's start by applying the `<begin-conversation>` tag to a page transition. We'll assume that the command button shown earlier is activated, only the action handler method does not make use of the `@Begin` annotation. In response to that action, the following navigation rule will start a long-running conversation and take the user to the `/coursewizard/selectFacility.xhtml` page:

```
<page view-id="/CourseList.xhtml">
  <navigation from-action="#{courseWizard.addCourse}">
    <begin-conversation/>
    <redirect view-id="/coursewizard/selectFacility.xhtml"/>
  </navigation>
</page>
```

If you want to preclude the use of the command button, you can instead declare the long-running conversation to begin when the `/coursewizard/selectFacility.xhtml` view ID is requested. This is done either by using the built-in page action for beginning a conversation, which is part of the Seam conversation API:

```
<page view-id="/coursewizard/selectFacility.xhtml"
  action="#{conversation.begin}"/>
```

or by nesting the `<begin-conversation>` tag directly inside the `<page>` node:

```
<page view-id="/coursewizard/selectFacility.xhtml">
  <begin-conversation/>
</page>
```

If there is a fine-grained page descriptor associated with the first screen of the wizard, you could exclude the `view-id` attribute, thus simplifying the declaration to:

```
<page>
  <begin-conversation/>
</page>
```

The only downside of using the built-in page action is that you cannot perform "prep work" in a component method before the first page is rendered, which in the examples above was to initialize and object the `newCourse` context variable. As you will see in a later example, the built-in page action is best suited for making a conversation generally available without a specific goal or action in mind.

The pages descriptor offers a lot of control for defining conversation boundaries because you can distinguish between initial requests, postbacks, and even action listener outcomes. However, you may want to be able to associate the start a long-running conversation with a link or button directly. For

that, you can use any of the tags from Seam's JSF component library to control the boundaries of the conversation.

A UI component tag approach

As a third option, you can begin a long-running conversation using one of Seam's UI component tags. You can either enhance an existing `UICommand` component with conversation propagation capabilities by adding a nested `<s:conversationPropagation>` tag or you can use one of Seam's link tags, `<s:link>` or `<s:button>`, which both have native support for conversation propagation. The value of the `propagation` attribute on all three of these tags can be any of the values listed in table 7.1 and is therefore not constrained in its use to begin conversations.

The UI component tag approach works by passing the `conversationPropagation` a request parameter in the URL. However, rather than having to add this parameter yourself, you can use the component tags cited here to add it for you, which abstracts away the name so that it is not hard-coded in your source code. If you need to use a custom `UICommand` component or submit a JSF form when starting the conversation, you can add a nested `<s:conversationPropagation>` tag to any `UICommand` component tag:

```
<h:commandButton action="#{courseWizard.addCourse}"
  value="Add course...">
  <s:conversationPropagation propagation="begin"/>
</h:commandButton>
```

Instead of using the `<s:conversationPropagation>` tag as a nested element in a regular JSF component tag, you can use the Seam link tags to specify the conversation propagation using the `propagation` attribute:

```
<s:button action="#{courseWizard.addCourse}" propagation="begin"
  value="Add course..." />
```

As long as you don't need to submit form data, the `<s:link>` and `<s:button>` tags are great alternatives to use when working with conversations. The advantage is that they have conversation controls built right in. In the next section you will see that they are also capable of passing the conversation token to the next request. The conversation abilities add to the benefit of producing RESTful URLs that can be reliably bookmarked, which was covered in chapter 3.

If you have been experimenting with the `begin` directive while reading through this section, you may have run into an exception stating that a long-running conversation is already active. To remedy this problem, you need to learn about how to join an existing conversation.

Enabling conversation joining

By default, Seam only begins a long-running conversation if one is not already in play. A long-running conversation is active if it was restored from a previous request or if a `begin` directive has already been encountered. In either case, if an attempt to cross the `begin` threshold happens again in the same request, Seam will throw an exception. It is said that when the `begin` directive is used, conversation joining is disabled.

This restriction could cause undue errors if it is possible for the user to navigate through one of the defined conversation boundaries after having already entering into a long-running conversation. For example, assume that you are using the `<begin-conversation>` page directive to begin a long-running conversation when the first page in the course wizard is requested. If the user submits the form on that page with validation errors, then when JSF attempts to redisplay the page, an exception will be thrown because the `<begin-conversation>` tag is once again encountered, this time in the presence of a long-running conversation.

You may run into other situations where the user is in the middle of a long-conversation and manages to click on a link that attempts to begin a new long-running conversation on top of the old one. This case is not that rare since the possible execution paths are numerous in an application that uses free-form navigation. You probably don't want to surprise your user with erroneous exceptions because they traversed a path that you did not anticipate. Therefore, unless you have good reason to assert that a long-running conversation is not yet active, I recommend that you use the join directive rather than the raw begin directive. In the absence of a long-running conversation, the join directive acts just like the begin directive anyway (without the risk of an exception).

To enable joining of an existing long-running conversation in the event that one already exists, or to begin one if it does not, you add the `join` attribute to the `@Begin` annotation if you are using the annotation-based approach:

```
@Begin(join = true)
public void addCourse() {
    course = new Course();
}
```

Likewise, if you are using the page-oriented approach to mark the boundaries of your conversation, then you add the `join` attribute to the `<begin-conversation>` tag:

```
<page view-id="/CourseList.xhtml">
  <navigation from-action="#{courseWizard.addCourse}">
    <begin-conversation join="true"/>
    <redirect view-id="/coursewizard/selectFacility.xhtml"/>
  </navigation>
</page>
```

If you are using the built-in conversation control `#{conversation.begin}` in a page action, there is nothing you have to change since this method is already "join safe". Finally, if you are using the UI component tag approach, you would switch the value of the `propagation` attribute from `begin` to `join`:

```
<s:button action="#{courseWizard.addCourse}" propagation="join"
  value="Add course..."/>
```

I realize that all of the options just presented to you may have been a tad overwhelming. You can finally breathe knowing that I am turning off the fire hose. I laid out all of these options so that you can get an appreciation for how flexible Seam is when it comes to setting the boundaries of long-

running conversations. As you might expect, there are just as many options for declaring the end of a long-running conversation as there is to begin one. However, since they follow the same patterns, it should be easy to pick up, just as it was to switch from begin to join.

AUTHOR CHOICE

I find the best approach for controlling conversations is to use the pages descriptor. It has the benefit of making your components more reusable because they aren't tied to conversation boundaries. In one scenario, you may want a long-running conversation to begin when a component method is invoked, while in other cases you don't want the invocation to have an affect on the current conversation. The pages descriptor tags even support conditional conversation boundaries, as you learn next.

Now for the bonus material! The `<begin-conversation>` directive supports a conditional clause using the `if` attribute. The value of the `if` attribute is a boolean value expression that is evaluated to determine whether or not the conversation propagation should be applied. The following declaration has the same effect as a join because it checks against the current conversation's long-running status:

```
<page view-id="/CourseList.xhtml">
  <navigation from-action="{courseWizard.addCourse}">
    <begin-conversation if="{!conversation.longRunning}"/>
    <redirect view-id="/coursewizard/selectFacility.xhtml"/>
  </navigation>
</page>
```

The condition on the `<begin-conversation>` tag becomes very useful as your pages mature to serve more variant use cases with different entrances and exits.

All this work of creating a long-running conversation is only useful if you can restore it. Let's see what you need to do to ensure that you can continue using the conversation on the next request.

7.3.4 Keeping the conversation going

Now that you have learned how to strike up a conversation, the next logical step is to learn how to keep the conversation going. By that, I mean how to restore the conversation on a subsequent request. The secret to restoring a conversation is to pass its conversation id on to the next request using the conversation token. Depending on the style of the request, this token may be passed as a request parameter or tucked away in the JSF component tree. Either way, when Seam detects the conversation token in the request, it uses its value to restore the existing long-running conversation rather than spawning a new temporary conversation to handle the request.

If passing the conversation token sounds like tedious work, there is good news for you. This task is handled automatically. In the presence of long-running conversation, Seam will add the conversation token to the JSF component tree so that is available on any subsequent JSF postback. For instance, to submit the form to select a facility and advance to the basic course information screen while, at the same time, retaining the long-running conversation, you can simply use a JSF command button:

```
<h:commandButton action="{courseWizard.selectFacility}"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```
value="Next &gt;&gt;" />
```

Notice that there is no indication of the conversation token. In fact, if you view the source of the rendered page, you won't find the conversation token there either. Seam works by placing the conversation token into a JSF component tree attribute. On a postback, when the component tree is restored, Seam picks the conversation id off the component tree and uses it to restore the long-running conversation.

That takes care of JSF postbacks, but what about non-JSF postbacks that do not have access to the JSF component tree? Seam can restore a conversation from a GET request by reading a special request parameter.

Restoring a conversation from a GET request

The conversation is restored automatically on JSF postbacks because Seam is able to leverage the state-saving ability of the JSF component tree by storing the conversation id in the JSF component tree. A GET request, on the other hand, is always a fresh start. Seam cannot read your thoughts, so unless you explicitly tell Seam which conversation to restore, a new temporary conversation is created to handle the request.

To have Seam restore a long-running conversation, all you have to do is pass the conversation token along with the request as a URL parameter. The default name for this parameter is `conversationId`. You can access the current conversation id using the value expression `#{conversation.id}`, which references the built-in conversation component bound to the `conversation` context variable.

Let's say that, at any point in the course wizard, you want to allow the user to be able to view a summary of what they have entered in a preview window. You give that page access to the long-running conversation associated with the wizard using the following hyperlink:

```
<a href="preview.seam?conversationId=#{conversation.id}"
  target="_blank">Preview</a>
```

You may prefer to use JSF component tags rather than raw HTML tags. In that case, you can use the `<h:outputLink>` tag, defining the `conversationId` request parameter using a nested `<f:param>` tag:

```
<h:outputLink value="preview.seam" target="_blank">
  <f:param name="conversationId" value="#{conversation.id}" />
  <h:outputText value="Preview" />
</h:outputLink>
```

There is one serious flaw in the previous two links. Seam allows the name of the conversation token to be customized, but these links hard-code the default name, `conversationId`. The name used for the conversation token is set using the `conversationIdParameter` on the built-in conversation manager component, bound to the manager context variable. You can override the name of the conversation token using the following component configuration:

```
<core:manager conversation-id-parameter="cid" />
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

With this override in place, any links with a hard-coded `conversationId` parameter will no longer perpetuate the long-running conversation. Fortunately, Seam provides a special `UIParameter` component tag, `<s:conversationId>`, that can be used to add the conversation id parameter to the parent JSF link component, replacing the need for the nested `<f:param>` tag in the previous example:

```
<h:outputLink value="preview.seam" target="_blank">
  <s:conversationId/>
  <h:outputText value="Preview"/>
</h:outputLink>
```

But, Seam's link tags, `<s:link>` and `<s:button>`, are a much better equipped for this purpose. They are aware of the presence of a long-running conversation and will add the conversation token to the URL that they generate, saving you from having to remember to include the `<s:conversationId>` component in the link:

```
<s:link view="preview.xhtml" target="_blank" value="Preview"/>
```

Keep in mind that Seam's link tags do not submit the JSF form and do not restore the JSF component tree, so they would not be useful for advancing the course wizard. But for basic linking, they are the best choice for navigating within the context of the long-running conversation and have the added benefit of being able to control the conversation using the `propagation` attribute.

These conversation token is not only useful for links and buttons, but also to control how long-running conversations are restored through alternate channels, such as Ajax requests and conversational web services. The conversation token is the key to the storage locker holding the context variables in a given conversation.

You have now struck up a long-running conversation and learned how to navigate within its boundaries, let's consider how to take advantage of this working set by contributing to it on one screen and accessing the values it holds on adjacent screens.

7.3.5 Enlisting objects in a conversation

You enlist objects, typically component instances or outjected variables, in a conversation by storing them into the conversation context. When you see that a component is scoped to the conversation context, or that a value is outjected to the conversation context, you might think to yourself, "But which conversation?"

As you know, there is always a conversation active during a JSF/Seam request and that conversation is either temporary or long-running. It's possible that multiple parallel conversations exist in the server's memory, but never mind those for right now because a request can only serve a single conversation at a time. So, the answer to your question is the conversation that is active at the time the conversation-scoped component is accessed, or the outjection occurs, depending on which case you are asking about.

What you may find interesting—perhaps even surprising—is that a temporary conversation does not need to be converted to a long-running conversation before you can start using it as one. You can

access conversation-scoped components and object to the conversation-scope, thus contributing to the conversation context, while the conversation is still temporary. Any variables added to the conversation context while the conversation is temporary remain part of the conversation context once the conversation transitions to long-running. In fact, the conversation status is nothing more than a marker which dictates whether to clean up the conversation context or store it in the session as a working set after the *Render Response* phase is complete.

Attaching to the active conversation

You should be able to put this knowledge together with what you have learned about component instantiation to recognize that when a conversation-scoped component is requested via its context variable name, the instance is attached to the active conversation. To put substance to this, I will talk through what happens when the course wizard is launched. Let's assume that the `CourseWizard` class is defined as follows:

```
@Name("courseWizard")
@Scope(ScopeType.CONVERSATION)
public class CourseWizard implements Serializable {
    @Out private Course newCourse;

    @Begin
    public void addCourse() {
        newCourse = new Course();
    }
}
```

NOTE

Conversation-scoped components should implement the `java.io.Serializable` interface since they will be stored in the HTTP session between requests. Implementing the `java.io.Serializable` interface allows the object to properly serialize when using distributed sessions.

When the `addCourse()` method is invoked either from a JSF command button or a page action, the conversation context is populated according to table 7.3.

Table 7.3 The steps by which the context variables are bound to the conversation serving the course wizard.

Step	Description
1. User activates JSF command button	JSF life cycle is invoked, <code>#{courseWizard.addCourse}</code> action queued temporary conversation is created <i>Invoke Application</i> phase is entered, action is processed <code>CourseWizard</code> is instantiated and bound to <code>courseWizard</code> context variable in conversation context
2. Action handler <code>addCourse()</code> invoked	<code>Course</code> is instantiated and assigned to private field <code>newCourse</code> <code>newCourse</code> is objected into conversation context temporary conversation promoted to long-running conversation
3. Navigation rule fires	<i>Render Response</i> phase is entered first screen of course wizard is rendered long-running conversation stored in JSF component tree

When the first screen of the course wizard comes to rest, there are two context variables in the conversation, `courseWizard` and `newCourse`. The `newCourse` context variable is outjected to the conversation scope since the owning class, `CourseWizard`, is a conversation-scoped component. For the duration of the wizard, the `newCourse` context variable is progressively populated by each screen.

If a parallel conversation were taking place in a separate tab, and the same action handler invoked, the process described above would be executed, only it would occur in another isolated area of the session managing a separate conversation. The same two context variables, `courseWizard` and `newCourse`, would exist in that conversation without interfering with the context variables in the first conversation. Two conversations, two sets of variables.

TIP

You can inspect which context variables are available in the conversation by using the Seam debug page. Ensure that debug mode is enabled (see chapter 5) and then visit the servlet path `/debug.seam`. You can use this page to inspect all active conversations as well as the session and application scope.

You should note that the component that hosts the `@Begin` method does not have to be a conversation-scoped component. You could begin the conversation for the course wizard using the following definition of the `CourseWizard` class:

```
@Name("courseWizard")
public class CourseWizard {
    @Out(scope = ScopeType.CONVERSATION) private Course newCourse;

    @Begin
    public void addCourse() {
        newCourse = new Course();
    }
}
```

In this case, the `newCourse` context variable has to be explicitly pushed out to the conversation context since the owning component is no longer scoped to that context. Thus, only the `newCourse` context variable is present in the conversation. To ensure that subsequent action handlers have access to this context variable, you may want to enforce the presence of a long-running conversation.

Making the conversation a prerequisite

If you want to enforce that a component or method only be used within the scope of a long-running conversation, then you annotate the component class or the method with the `@Conversational` annotation. For instance, you could declare that the `submitBasicInfo()` method only be used in the context of a long-running conversation:

```
@Conversational
public String submitBasicInfo() {
    // ...
}
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

If an attempt is made to execute a `@Conversational` method outside the presence of a long-running conversation, Seam will raise the `org.jboss.seam.noConversation` event and then throw the runtime exception `NoConversationException`.

Using this annotation is superficial in most cases since there is likely more to the story than just the existence of a conversation. In the next section we are going to see how a conversation page flow is a much better way to enforce this requirement. However, if you are trying to protect a very sensitive area of the code than this simple annotation might make sense.

The course wizard offers a nice simple example to get started with conversations, but it could really benefit from a page flow and it also falls short on demonstrating the true convenience of having a working set of context variables. While the user is working through the course wizard, let's consider the example of comparing courses.

Learning to use the working set

In the course wizard, the conversation context relieves you of having to carry along the course data using hidden fields as you progress from screen to screen, but without having to use the crutch of the HTTP session. In this next example, the conversation context will be used in a non-linear fashion to build up a selection of courses which are then compared side-by-side on a comparison screen. This activity will be supported by the conversation-scoped `CourseComparison` component.

To capture a selection of courses, we first need to "open" a working set. That is done by beginning a long-running conversation when the `CourseList.xhtml` page is requested using the following stanza in the `CourseList.page.xml` descriptor:

```
<begin-conversation join="true"/>
```

Next, we need to enable data model selection on the `CourseList.xhtml` page. To do that, we first need to put the result list in the page scope so that it remains stable on postback. The `courses` context variable is populated using a factory method on `CourseList`:

```
@Name("courseList")
public class CourseList extends EntityQuery<Course> {
    ...
    @Factory(value = "courses", scope = ScopeType.PAGE)
    public List<Course> getResultList() {
        return super.getResultList();
    }
}
```

Next, the `courses` context variable is used in the value attribute of the `<rich:dataTable>` and the table is wrapped in an `<h:form>`:

```
<h:form id="courses">
    <rich:dataTable id="courseList" var="course" value="#{courses}"
        rendered="#{not empty courses}">
        ...
    </rich:dataTable>
</h:form>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```
</rich:dataTable>
</h:form>
```

With the table setup, we need to enable a course to be selected. We don't want to interfere with the Select link that takes the user to the course detail page, so we will introduce the concept of modes. When comparison mode is enabled, the Select link adds the selected course to the comparison queue. When comparison mode is disabled, the Select link will have to existing behavior. The mode is toggled using the `enable()` and `disable()` methods on `CourseComparison`. The `comparisonModeEnabled` context variable is outjected to the conversation scope to indicate whether the mode is active. Another variable named `readyToCompare` indicates whether there is at least two courses selected. To capture the course selection, a parameterized action handler `select(Course)` on `CourseComparison` is used. The `CourseComparison` component is shown in listing 7.1.

Listing 7.1 A conversation-scoped component to support the comparing courses use case

```
package org.openl8.action;
import ...;

@Name("courseComparison")
@Scope(ScopeType.CONVERSATION)
public class CourseComparison implements Serializable {
    @Out private boolean comparisonModeEnabled = false;

    @Out private boolean readyToCompare = false;

    @Out private List<Course> comparedCourses =
        new ArrayList<Course>();

    public void select(Course course) {
        if (comparedCourses.contains(course)) {
            return;
        }
        comparedCourses.add(course);
        if (comparedCourses.size() > 1) {
            readyToCompare = true;
        }
    }

    public void enable() {
        comparisonModeEnabled = true;
        if (courses.size() > 1) {
            readyToCompare = true;
        }
    }

    public void disable() {
        comparisonModeEnabled = false;
        readyToCompare = false;
    }
}
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```
}  
}
```

The CourseList.xhtml requires three additions. First, buttons to enable and disable comparison mode:

```
<s:button id="enableCompare" action="#{courseComparison.enable}"  
  value="Enable comparison mode" rendered="#{not  
comparisonModeEnabled}"/>  
  
<s:button id="disableCompare" action="#{courseComparison.disable}"  
  value="Disable comparison mode" rendered="#{comparisonModeEnabled}"/>
```

Next, a page fragment that lists the courses to be compared:

```
<s:fragment  
  rendered="#{comparisonModeEnabled and comparedCourses.size gt 0}">  
  <rich:panel>  
    <f:facet name="header">Courses selected for comparison</f:facet>  
    <ui:repeat var="_course" value="#{comparedCourses}">  
      <div>#{_course.name}</div>  
    </ui:repeat>  
  </rich:panel>  
</s:fragment>
```

Finally, the Select link in the course list table needs to be conditionally rendered to either be a drill down link or a link to select a course for comparison, depending on the mode:

```
<s:link id="course" view="/#{empty from ? 'Course' : from}.xhtml"  
  value="Select" rendered="#{not comparisonModeEnabled}">  
  <f:param name="courseId" value="#{course.id}"/>  
</s:link>  
<h:commandLink action="#{courseComparison.select(course)}"  
  value="Select" rendered="#{comparisonModeEnabled}"/>
```

With comparison mode enabled, each time the Select link is clicked for a previously unselected course, it is added to the list of compared courses in the conversation context. You are free to search, sort, and paginate the course list without worry of tracking the selected courses. No hidden form fields, no `<t:saveState>`. When the user is ready to compare the courses, they navigate to the CompareCourses.xhtml page using the following Seam link tag which sends the conversation token along with the request:

```
<s:button value="Compare" view="/CompareCourses.xhtml"  
  rendered="#{readyToCompare}"/>
```

The comparison page has full access to the conversation-scope variables established on the course list page and can direct the user back to the course list page to select additional courses. The course comparison is as follows:

```

<rich:panel>
  <f:facet name="header">Compare Courses</f:facet>
  <h:panelGrid columns="#{comparedCourses.size + 1}">
    <rich:panel>

      <f:facet name="header">&#160;</f:facet>
      <div>Location:</div>
      <div>Type:</div>
      <div>Holes:</div>
      <div>Green Fee:</div>
      ...
    </rich:panel>
    <c:forEach items="#{comparedCourses}" var="_course">
      <rich:panel>
        <f:facet name="header">#{_course.name}</f:facet>
        <div>#{_course.facility.city}, #{_course.facility.state}</div>
        <div>#{_course.facility.type}</div>
        <div>#{_course.numHoles}</div>
        <div>#{_course.facility.greenFee}</div>
        ...
      </rich:panel>
    </c:forEach>
  </h:panelGrid>
</rich:panel>
<div class="actionButtons">
  <s:button view="/CourseList.xhtml" value="Add courses"/>
</div>

```

Notice the traversal of the facility association. The persistence context is scoped to the conversation, allowing lazy associations to be loaded in the view since the Course entity instance never enters a detached state. You are going to learn how this works in chapters 8 and 9 on managed persistence.

Since the CompareCourses.xhtml page requires that you have a conversation active and that at least two courses are selected, you may want to enforce these restrictions in the CompareCourses.page.xml descriptor:

```

<page conversation-required="true"
  no-conversation-view-id="/CourseList.xhtml"
  action="#{courseComparison.validate}">
  <navigation from-action="#{courseComparison.validate}">
    <rule if-outcome="invalid">
      <redirect view-id="/CourseList.xhtml">
        <message severity="warn">
          You must select at least two courses.
        </message>

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```
        </redirect>
    </rule>
</navigation>
</page>
```

If the page is requested and a long-running conversation is not active or has expired, the `org.jboss.seam.noConversation` event is raised and the user is redirected to the view ID defined by the `no-conversation-view-id` attribute. If there is a long-running conversation, then the `#{courseComparison.validate}` page actions gets its chance to validate the request.

Having learned to fear the session, you may be uneasy about letting data accumulate in the active conversation. Apart from ending the conversation, which ultimately destroys the conversation context, you may want to be able to remove a particular context variable.

Unregistering conversation-scoped context variables

Any objects associated with a conversation are held in the conversation context until the conversation ends or the object is explicitly removed from the conversation context. If you get to a point where a context variable is no longer needed and is just wasting space in the conversion context, you can remove it in one of two ways:

- Set the property annotated with `@Out(required = false)` to null
- Remove the context variable using the Seam Context API

For instance, in the course wizard, you don't want to append a new tee set to the `Course` instance until it has been validated. Thus, the tee set information form (the fifth screen in the wizard) works on temporary `TeeSet` instance bound to the `newTeeSet` context variable. When the form is submitted and passes validation, the `newTeeSet` context variable can be assigned to the `Course` instance and discarded. Setting the outjected property to null accomplishes this task:

```
@Out(required = false) private TeeSet newTeeSet;

public void saveHoleData() {
    newTeeSet = new TeeSet();
}

public void saveTeeSet() {
    newCourse.getTeeSets().add(newTeeSet);
    newTeeSet = null;
}
```

Another way to accomplish this task, and may be better suited if not using outjection, is to retrieve the conversation context via the Seam Context API and manually remove the context variable from it:

```
Contexts.getConversationContext().remove("newTeeSet");
```

The best way to ensure that conversation-scope context variables are cleaned up is to just end the conversation. Leaving conversations active is not as dangerous as letting objects linger in the session, but is still a good idea to prevent excess consumption of memory.

7.3.6 Ending a long-running conversation

As you have learned, conversations are a managed region of the HTTP session. Thus, it is possible to terminate a working set without destroying the entire conversation. A conversation can either be ended explicitly using an end propagation directive or it can be automatically garbage collected by Seam when its idle time exceeds the timeout value of the conversation.

The end propagation directive is used in the same way as the begin directive. The most obvious use case for ending a conversation is when the user wants to cancel out of a form or wizard. In this case, you want to discard the conversation and return the user to the home page. The easiest way to do this is to use the `<s:link>` component tag.

```
<s:link view="/CourseList.xhtml" propagation="end" value="Cancel"/>
```

However, if you prefer to keep your conversation directives out of the JSF views, you can use the `pages.xml` configuration.

```
<page view-id="/coursewizard/*">
  <navigation>
    <rule if-outcome="cancel">
      <end-conversation/>
      <redirect view-id="/CourseList.xhtml"/>
    </rule>
  </navigation>
</page>
```

You pair this navigation rule with the following simplified link:

```
<s:link action="cancel" value="Cancel"/>
```

You could also use a regular JSF command link, but you must be sure to set the `immediate` flag to `true` so that the action is handled prior to applying any validations. The `<s:link>` and `<s:button>` components do not have this requirement since they do not submit the form.

```
<h:commandLink action="cancel" immediate="true" value="Cancel"/>
```

Ending a conversation merely demotes it from long-running to temporary. That means that whatever values were present in the conversation will still be available when rendering the ensuing page. This might be desirable if the next page is a confirmation screen that shows a summary of the persisted data. However, if you really want to wipe out all evidence of the conversation before rendering the next page, you need to instruct Seam to terminate the conversation prior to issuing the redirect. The `pages.xml` configuration has been updated to reflect this change.

```
<page view-id="/coursewizard/*">
  <navigation>
    <rule if-outcome="cancel">
      <end-conversation before-redirect="true"/>
      <redirect view-id="/CourseList.xhtml"/>
    </rule>
  </navigation>
</page>
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```

    </rule>
  </navigation>
</page>

```

By setting the `before-redirect` attribute to `true` on the `<redirect>` element, the context variables that were active during the wizard are no longer available to the `CourseList.xhtml` page when it is rendered.

Assume that the user made it all the way through the wizard and is ready to save the new course. This case is perfect for showing off the `@End` annotation. Let's place a command button on the last page that will be used to invoke the `save()` method on the `CourseWizardAction` component.

```
<h:commandButton action="#{courseWizard.save}" value="Save"/>
```

Next, you place the `@End` annotation on the `save()` method so that the conversation is demoted to temporary when the method call is complete:

```

@End
public String save() {
    try {
        // ...business logic...
        entityManager.persist(newCourse);
        FacesMessages.instance().add(
            "#{newCourse.name} has been added to the directory.");
        return "success";
    } catch (Exception e) {
        FacesMessages.instance().add(
            "Saving the course failed.");
        return "failure";
    }
}

```

The `@End` annotation is summarized in table 7.4.

Table 7.4 The `@End` annotation is used to transition the long-running conversation to temporary.

Name:	End	
Purpose:	Indicates that a long-running conversation should end when this method is invoked. To take effect, the method must have a void return type or return a non-null outcome when invoked.	
Target:	METHOD	
Attribute	Type	Function
<code>beforeRedirect</code>	boolean	If set to true, instructs Seam to terminate the temporary conversation prior to issuing a navigation redirect. The default is to propagate the conversation across the redirect and terminate it once the response is complete.

Alternatively, you may want to end the conversation using the `<end-conversation>` tag in the `pages.xml` rather than by using the `@End` annotation.

```
<page view-id="/coursewizard/*">
```

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=354>

```

<navigation from-action="#{courseWizard.save}">
  <rule if-outcome="success">
    <end-conversation/>
    <redirect view-id="/coursewizard/summary.xhtml"/>
  </rule>
</navigation>
</page>

```

The newCourse context variable is still available on the summary page since the conversation is not ended prior to the redirect.

HINT

You want to be careful using the `beforeRedirect` flag because you will lose any JSF messages that you added in the action handler method. An intermediate screen is recommended if you must display messages.

The other, less graceful way of terminating a conversation is to allow it to expire. The default timeout period of the conversation is assigned using the built-in conversation manager component, bound to the `manager` context variable. The timeout is specified in milliseconds. The following stanza sets the timeout period to one hour:

```
<core:manager conversation-timeout="3600000"/>
```

You can also customize this value per-conversation by setting the `timeout` attribute on the `<page>` node in the pages descriptor. What this allows you to do is modify the timeout period per screen, to perhaps give the user more time to fill out a more complex form:

```
<page view-id="/coursewizard/holeData.xhtml" timeout="7200000"/>
```

You can also manually set the value by using the `setTimeout()` method on the built-in conversation component, which is bound to the `conversation` context variable.

Generally, you want to avoid abandoning conversations because it makes the session larger than it has to be. The important point to take away, however, is that doing so is not dangerous. In fact, it may even be desirable if your intent is to be able to switch between existing long-running conversations, which are covered in section 7.6.

All this time you have spent with the course wizard example has probably gotten you thinking that this use case would be better served by a page flow. Your right! Let's hook up the course wizard to Seam's jBPM-based page flow support. **(these last three paragraphs are choppy)**

7.4 Driving the conversation with a page flow

There are two types of navigation models in Seam, stateless and stateful. Up to this point you have worked solely with the stateless navigation model. The stateless model is ideal if you want the user to be able to perform steps in no particular order. The course comparison offered a good example of that type of use case. But when the path to the end goal is very clear, like in the course wizard example, it makes far more sense to drive it with a page flow.

In Seam, page flows are implemented using a special integration with the jBPM library. It may seem like overkill to use a Business Process Management (BPM) library to control a page flow. But, understand that Seam is leveraging jBPM for its generic process definition language (jPDL) and interpreter, which together serve as a framework for building flow-based software modules. It is on this framework that Seam has built its page flow module. You will learn more about using jBPM to drive actual business processes in chapter 13.

7.4.1 Learning to use a page flow

A jPDL descriptor defines the page flow for a single conversation. We will use a page flow named `courseWizard`, defined in the `courseWizard.jpdl.xml` descriptor, to drive the course wizard and even perform some logic based on the user's input. Rather than just dump the whole descriptor on your lap, I am going to step through it piece by piece so that you can better understand how it works. The complete descriptor is available in the source code that accompanies the book.

First things first, it is necessary to "install" the page flow, `courseWizard.jpdl.xml`, in the Seam conversation descriptor:

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>courseWizard.jpdl.xml</value>
  </bpm:pageflow-definitions>
</bpm:jbpm>
```

The page flow descriptor can be located anywhere on the classpath. For seam-gen projects, it should be placed in the resources folder at the root of the project tree. With the file in place, you are ready to start populating it.

The root tag of a page flow is `<pageflow-definition>`. The name of the page flow is defined in the name attribute on this node. Seam provides an XSD schema for the page flow descriptor so that you get all of the tag completion goodness that you are enjoying with the other Seam descriptors. The outer shell of the page flow descriptor for the course wizard appears is shown here:

```
<pageflow-definition
  xmlns="http://jboss.com/products/seam/pageflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://jboss.com/products/seam/pageflow
    http://jboss.com/products/seam/pageflow-2.0.xsd"
  name="courseWizard">
  ...
</pageflow-definition>
```

To put the page flow in motion, you have to create a process instance that will manage it. Fortunately, Seam makes this task extremely easy. Since a page flow is typically mapped one-to-one with a conversation, you begin a page flow using the exactly same directive that you use to begin the conversation. Regardless of whether you are using the `@Begin` annotation, the `<begin-conversation>` pages descriptor tag, or the Seam UI component tags, you specify the page flow

definition in the `pageflow` attribute. When the conversation begins, an instance of the page flow definition will be created and the process token is advanced to the start node. More on that shortly.

Here, the `CourseWizard` component has been augmented to the page flow to the conversation that is started by the `addCourse()` action handler method:

```
@Begin(pageflow = "courseWizard")
public void addCourse() {
    newCourse = new Course();
}
```

I want to encourage you by saying that the hardest part of page flows is the first step. Once you get beyond that, I am positive that you will pick up on page flows without any trouble. When the process instance, which is used to track the state of the page flow, is created, it looks for a starting node. You have two choices, `<start-state>` or `<start-page>`. If you are starting a page flow from an action—meaning at a point in the Seam life cycle where a navigation event can still occur—then you use the `<start-state>` node. This node is the most appropriate choice for the course wizard. We will consider when the `<start-page>` node is used next. The page flow definition for the course wizard begins as follows:

```
<pageflow-definition name="courseWizard">
  <start-state>
    <transition to="basicCourseInfo"/>
  </start-state>
</pageflow-definition>
```

The `<transistion>` node is analogous to the `<rule>` node in the pages descriptor in that it matches an outcome value—the return value of an action handler method—specified in the `name` attribute. However, if there is no outcome value, which is the case with the `addCourse()` method, then the `name` attribute can be omitted.

A transition implies that there must be a target. The `to` attribute specifies the name of the node to advance to next. There are four main nodes that can appear in the page flow definition after the start state. These nodes are summarized in table 7.5.

Table 7.5 The main nodes in the page flow descriptor

Node	Purpose
page	Renders a JSF view and declares transitions that are used upon exiting that view
decision	Evaluates an EL expression and follows a declared transition based on the result
process-state	Used to spawn a sub-page flow
end-state	Terminates the process instance without ending the long-running conversation; typically used to end a sub-page flow

The `<page>` node is what ultimately leads to the next JSF view in the flow being rendered and is thus the "wait" state in the process. Do not confuse it with the `<page>` node in the pages descriptor as it does not have the same semantics. The `<page>` node indicates a JSF view ID should be rendered

when the process token arrives at the node. The following `<page>` node renders the first screen in the course wizard, coming out of the start state:

```
<page name="basicCourseInfo"
  view-id="/coursewizard/basicCourseInfo.xhtml" redirect="true">
  <transition name="cancel" to="cancel"/>
  <transition name="next" to="description"/>
</page>
```

If the `redirect` attribute is included on the `<page>` node and has a value of true, then Seam performs a redirect prior to rendering the page. Doing so resets the URL in the browser so that it reflects the current page and avoids the confusing message about resubmitting post data that is thrown up as the result of using the browser's "refresh" button. I will mention more about browser buttons in a moment.

NOTE

You may also specify the redirect functionality using a nested `<redirect/>` element. Which you choose is up to you, but I prefer the `redirect` attribute since it is adjacent to the `view-id` attribute to which it applies.

If the conversation, and the page flow association with it, is started from the *Render Response* phase, then it is not possible to invoke a navigation event. Therefore, the beginning of the page flow is specified using a `<start-page>` node. The `<start-page>` node acts exactly like a `<page>` node, only it has the special meaning of being the first page. Consider the case in which the `newCourse` context variable is created by a `@Factory` method and that method is what begins the conversation:

```
@Begin(pageflow = "courseWizard")
@Factory("newCourse")
public void initCourse() {
    newCourse = new Course();
}
```

Rather than executing the `addCourse()` action handler, the user is directed to the `/coursewizard/basicCourseInfo.xhtml` screen directly. Then, the start of the page flow is defined as follows:

```
<start-page name="basicCourseInfo"
  view-id="/coursewizard/basicCourseInfo.xhtml">
  <transition name="cancel" to="cancel"/>
  <transition name="next" to="description"/>
</start-page>
```

There is one important bit of information that you must know to make sense of a `<page>` node (as well as the `<start-page>` node). This node bundles configuration for entering the node and leaving the node. Okay, so that isn't so different from other process nodes. But what makes the `<page>` node so confusing to grasp at first is that it is a wait state node, which means that the entry configuration is processed in one request and the exit configuration is processed in another. Thus, the

`<transition>` nodes in the `<page>` node shown above apply to the outcomes of actions on the `basicCourseInfo.xhtml` view. While the user is filling out the form, the process instance is paused.

There is nothing special about the JSF views that participate in a page flow. The only difference is how the `UICommand` components are defined. As mentioned earlier, transitions are chosen based on the outcome of an action. You have a number of options on how to arrive at that outcome. Let's start with the literal approach, which is used on the first screen in the wizard. There are two command buttons, one that cancels the wizard and one that advances to the next screen:

```
<s:button id="cancel" action="cancel" value="Cancel"/>
<h:commandButton id="next" action="next" value="Next"/>
```

When either of these two buttons are activated, Seam merely matches the literal values of the action with the transitions defined on the `<page>` node and advances the token to the named nodes, shown here:

```
<page name="description"
  view-id="/coursewizard/description.xhtml" redirect="true">
  <transition name="cancel" to="cancel"/>
  <transition name="next" to="holeData">
    <action expression="#{courseWizard.prepareHoleData}"/>
  </transition>
</page>

<page name="cancel" view-id="/CourseList.xhtml" redirect="true">
  <end-conversation before-redirect="true"/>
</page>
```

Now things are starting to get interesting. Let's first focus on the `cancel` transition, which transitions to the `<page>` node named `cancel`. In the `cancel` `<page>` node, we see another familiar element, `<end-conversation>`. The `<end-conversation>` element is interpreted in the same way as when it is used with a `<rule>` element in the pages descriptor. It is applied upon entering the `<page>` node just like the `redirect` attribute. Since the two are combined in this case, when the `CourseList.xhtml` page is rendered, the conversation that served the page flow will be completely wiped out, along with all of its context variables. That also means that the process instance is terminated. This termination happens even in the absence of the `<end-state>` node since the process instance is bound to the conversation.

The `description.xhtml` page has the same two buttons as on the first screen. The `cancel` button behaves as before. The `next` button, however, behaves differently. In the process of transitioning to the `<page>` node named `holeData`, an action is executed. This behavior is the opposite of what you are used to from using stateless navigation in that the action is executed *after* the transition is decided. The reason for this ordering is to keep the method binding expressions out of the UI so that only the literal outcome value must be supplied in the command button. As an alternative, you could use the method binding expression in the `action` attribute of the command button:

```
<h:commandButton id="next" action="#{courseWizard.prepareHoleData}"
  value="Next" />
```

The `prepareHoleData()` method would need to return the string value `next`, which then allows the `<transition>` to return to its basic form:

```
<transition name="next" to="holeData"/>
```

Now its time for some decisions to be made. Although the game of golf is designed in such a way as to give men and ladies different par and handicap values, many courses don't make that distinction. Therefore, the user will be allowed to enter the men's par and handicap data and then asked whether it is necessary to provide different data for ladies. The answer to that question dictates whether or not to return to the `holeData.xhtml` view to capture the additional information. This decision is handled by the `decideHoleData` node, which either transitions back to the `holeData` `<page>` node or on to the next step in the flow (in this case, `teeSet`):

```
<page name="holeData"
  view-id="/coursewizard/holeData.xhtml" redirect="true">
  <transition name="cancel" to="cancel"/>
  <transition name="Men" to="decideHoleData">
    <action expression="#{courseWizard.submitMensHoleData}"/>
  </transition>
  <transition name="Ladies" to="teeSet">
    <action expression="#{courseWizard.submitLadiesHoleData}"/>
  </transition>
</page>

<decision name="decideHoleData"
  expression="#{courseWizard.ladiesHoleDataRequired}">
  <transition name="true" to="holeData"/>
  <transition name="false" to="teeSet"/>
</decision>
```

Once again, we see the inverted actions that are applied during the transition from the Men and Ladies outcomes. After the men's hole data is submitted, a decision has to be made as to whether to return to the `holeData.xhtml` screen to capture data for the ladies. The command buttons on the `holeData.xhtml` page are conditionally rendered based on the value of the gender context variable:

```
<h:commandButton id="nextMen" action="Men" value="Next"
  rendered="#{gender eq 'Men'}"/>
<h:commandButton id="nextLadies" action="Ladies" value="Next"
  rendered="#{gender eq 'Ladies'}"/>
```

The page flow continues on in this manner until the review screen, where the user can check the information one last time before persisting it. The final two pages are defined as follows:

```
<page name="review"
  view-id="/coursewizard/review.xhtml" redirect="true">
  <transition name="cancel" to="cancel"/>
```

```

    <transition name="success" to="end">
      <action expression="#{courseHome.setCourseId(newCourse.id)}"/>
    </transition>
    <transition name="failure" to="review"/>
  </page>

  <page name="end" view-id="/Course.xhtml" redirect="true">
    <end-conversation/>
  </page>

```

The review screen assumes the use of an action handler in the UI command button, since the transitions are setup to handle the outcome of that method:

```
<h:commandButton id="save" action="#{courseWizard.save}" value="Save"/>
```

Putting the method binding expression in the UI is acceptable here since the review screen is not likely to change its purpose. It allow allows us to leverage the transition action to set the newly established id of the course so that it can be displayed after the page flow is complete. The rendering of the course is handled by the `CourseHome` component, bound to the `courseHome` context variable.

You have now completed your very first page flow! While the excitement is still present and page flows are fresh in your mind, I want to address to additional features. First, let's talk about those pesky browser buttons, back and refresh.

7.4.2 Backing up In the flow

If you have heard it asked once, you have heard it a hundred times. "Can we disable the back button?" Lucky are those who are so blissfully unaware. It's a stateless world and we have to learn to live in it. Fortunately, Seam specializes in bring state to that world and thus, you can appreciate that Seam has taken care of this problem, not by disabling the back button, but by being smart enough to now when it is used.

If, during a page flow, the user attempts to return to an earlier page and resubmit the form, Seam will gracefully redirect them to the current page—the `<page>` node where the process token is resetting in its "wait" state. That also applies to when the user hits the refresh button and the browser attempts to resubmit the form. Of course, that problem has already been solved by performing a redirect during the transition, but it's still nice to know that Seam has that base covered. This describes the default behavior of a page flow.

But, you may decide that it is alright for users to back up and change their answers. If you want to allow this behavior as a way for users to check their work, I encourage you to bake it into the page flow. If you don't want to go through that extra effort and there is no harm done in allowing them to resubmit forms that have already been submitted, then you can enable back button by setting the back attribute on the `<page>` node to enabled:

```

<page name="review" view-id="/coursewizard/review.xhtml"
  redirect="true" back="enabled">
  ...

```

</page>

The only downside to this setting is that it allows the user to back up to any page leading up to this page. Once you open the door, it remains wide open.

7.4.3 Delegating to subflows

In addition to a linear page flow, it is possible to spawn a sub-page flows. A sub-page flow is a good way to synthesize a page flow from more elemental parts so as to enable sharing of common page flow logic (DRY). In the current implementation, we assume that the course wizard is launched from the facility page so that it can be immediately associated with the new course. If we want to allow the user to launch the flow from an arbitrary point in the application and let them select a facility as the first step, we need to incorporate that logic into the flow.

First, we change the `addCourse()` method to return an outcome based on whether a facility is available and has thus been selected:

```
@In(required = false)
private FacilityHome facilityHome;

@Begin(pageflow = "courseWizard")
public void addCourse() {
    newCourse = new Course();
    newCourse.setFacility(facilityHome.getDefinedInstance());
    return newCourse.getFacility() != null ?
        "facilitySelected" : "facilityNotSelected";
}
```

Next, we modify the page flow to handle this initial outcome and transition into the sub-process for selecting a facility if necessary:

```
<start-state>
  <transition name="facilitySelected" to="basicCourseInfo"/>
  <transition name="facilityNotSelected" to="selectFacility"/>
</start-state>

<process-state name="selectFacility">
  <sub-process name="selectFacility"/>
  <transition to="basicCourseInfo"/>
</process-state>
```

When the `selectFacility` sub-process ends, the process will transition to the `basicCourseInfo` <page> node. Of course, its possible to do more sophisticated logic using transition actions and <decision> nodes following the return to the page flow.

All that is left is to build the `selectFacility` process, which is defined in the `selectFacility.jpdl.xml` descriptor. Note that you will need to declare this descriptor in component descriptor above the page flow definition in which it is used:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```

<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>selectFacility.jpdl.xml</value>
    <value>courseWizard.jpdl.xml</value>
  </bpm:pageflow-definitions>
</bpm:jbpm>

```

The selectFacility page flow takes advantage of the FacilityList.xhtml page, which allows the user to search, sort and paginate to find a facility to select.

```

<pageflow-definition name="selectFacility">
  <start-state>
    <transition to="facilityList"/>
  </start-state>
  <page name="facilityList"
    view-id="/FacilityList.xhtml" redirect="true">
    <transition name="select" to="end">
      <action expression="#{courseWizard.assignFacility}"/>
    </transition>
  </page>
  <end-state name="end"/>
</pageflow-definition>

```

Note that the `<end-state>` is used in this page flow to signal the end of the process and return to the parent flow. We do not want to end the conversation since the main page flow is not yet complete. The `assignFacility()` method merely grabs the defined facility instance from the `FacilityHome` component and assigns it to the `newCourse` instance:

```

public void assignFacility() {
    newCourse.setFacility(facilityHome.getDefinedInstance());
}

```

What's nice about having split off the select facility logic is that it allows you expand that part of the flow in the future to perhaps allow the user to create a new facility instead of selecting an existing one. As far as the main flow is concerned, that's all just low-level details.

That wraps up the introduction to page flows. And what an introduction it has been. In this tutorial, you have had a chance to see a couple of the more advanced features. Trust that there is a wealth of additional features that were not covered here that are left for you to discover, including the ability to execute actions in more places, configure fall back pages when the conversation times out, set the timeout per page, end tasks, initiate a business process, and even tap into the native extension points of the jPDL. You can seriously micromanage the user's interaction with your system using page flows. The XML can get quite verbose, but then again, if power is what you are after, it may be worth the trouble.

Remember, though, that your user is a person and people like to multi-task. The popularity of browser tabs reflects this fact. Just because the page flow represents the most logical progression through the application doesn't mean the user is necessarily going to follow it without interruption. In the next section you learn how to allow the user to step out of a conversation to perform other tasks.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

Then, in section 7.6 you learn to create widgets that give the user the ability to switch between parallel conversations that have been spawned.

7.5 Stepping out of the conversation

So far we have talked about propagating the conversation, but what about the case when you want to disable propagation? Disabling propagation is not the same thing as ending a long-running conversation. Instead, you can think of it as abandoning the existing conversation, or stepping outside its bounds.

7.5.1 Abandoning a conversation

You may be wondering why you would want to avoid propagating a conversation. The need is actually more common than you think. While the stateful behavior that conversations provide is a drastic improvement over the stateless approach, sometimes the conversation can have too good of a memory. After you take a bad shot in golf, you are told to let it go. But it's easier said than done. While you are trying to focus on your next shot, you just cannot seem to shake off the memory of the prior shot. Conversations can be the same way. Sometimes you just want to get away from one, yet it just wants to keep holding on. This section shows you how to stop it in its tracks so that you can move on to the next use case.

Let's assume there you want to put a link to start the course wizard on a page that uses a conversation, such as the `CourseList.xhtml` page or perhaps even somewhere in the middle of the course wizard itself. When you start the wizard, you don't want to keep using the same conversation. Instead, you want to leave the existing conversation so that a new conversation will be created to handle the page flow. For now, don't worry about how to get back to the partially complete wizard as you will learn later how to do that using a conversation switcher. Let's just break ties with that conversation without looking back. Once again, we can use the `<s:conversationPropagation>` tag, only this time we are using it to prevent conversation propagation:

```
<h:commandLink action="#{courseWizard.addCourse}">
  <s:conversationPropagation propagation="none"/>
  Add course...
</h:commandLink>
```

Being the wise and savvy Seam developer that you are, you decide to cut down on keystrokes and use the `<s:link>` tag to trim the size of the markup and make a bookmark-friendly link:

```
<s:link action="#{courseWizard.addCourse}" propagation="none"
  value="Add course..."/>
```

The `none` directive is necessary in cases where the conversation token is added automatically, such as with command components and Seam link components. Another way to get away from the current long-running conversation is to use the no-arguments `leave()` method on the built-in conversation component in an action listener. This method can be used as an action listener thanks to the Seam extended EL. Calling this method has the same effect as the `none` propagation directive:

```
<h:commandLink actionListener="#{conversation.leave}"
  action="#{courseWizard.addCourse}">
  Add course...
</h:commandLink>
```

Non-JSF links have no awareness of the current long-running conversation, so abandoning a conversation in those cases is just a matter of leaving off the `<s:conversationId>` tag.

When you just cannot seem to shake a conversation, setting the propagation directive to `none` will be sure ward it off. When a conversation is abandoned, it becomes subject to conversation timeouts.

If you get to the end of the use case and no longer need the current long-running conversation, it's usually best to end the conversation properly, rather than abandoning it. However, if you just want to put the conversation to the side for the moment, then abandoning the conversation is the appropriate choice. But, before you go abandoning conversations, consider whether it is more appropriate to suspend the current long-running conversation by nesting a new long-running conversation within it. That way, when the nested conversation is finished, you can return to the parent conversation since the association is still established. You will see that this mechanism is almost identical to how sub-page flows work.

7.5.2 Creating nested conversations

Nested conversations allow you to isolate context variables within the scope of a conversation in the same way that conversations allow you to isolate context variables within the session scope.

If you are familiar with how child processes work on a Unix system, you will find that nested conversations have similar semantics and share the same relationships with their parents. When you begin a nested conversation, you are effectively suspending the state of the active long-running conversation and starting a brand new long-running conversation with its boundaries. What makes this nested conversation different from just another primary conversation is that it has read-only access to the context variables in its parent, just as a child process inherits the environment of its parent. Another similarity that these conversations share with child processes is that changes made to the nested conversation do not disrupt the state held in the parent conversation. Finally, conversations can be nested to arbitrary depth, opening up the possibility that a parent of a nested conversation can itself be a nested conversation.

Branching off the mainline

Nested conversations begin by branching off the main conversation, as shown in figure 7.7. As you can see, multiple nested conversations can exist concurrently within a parent conversation.

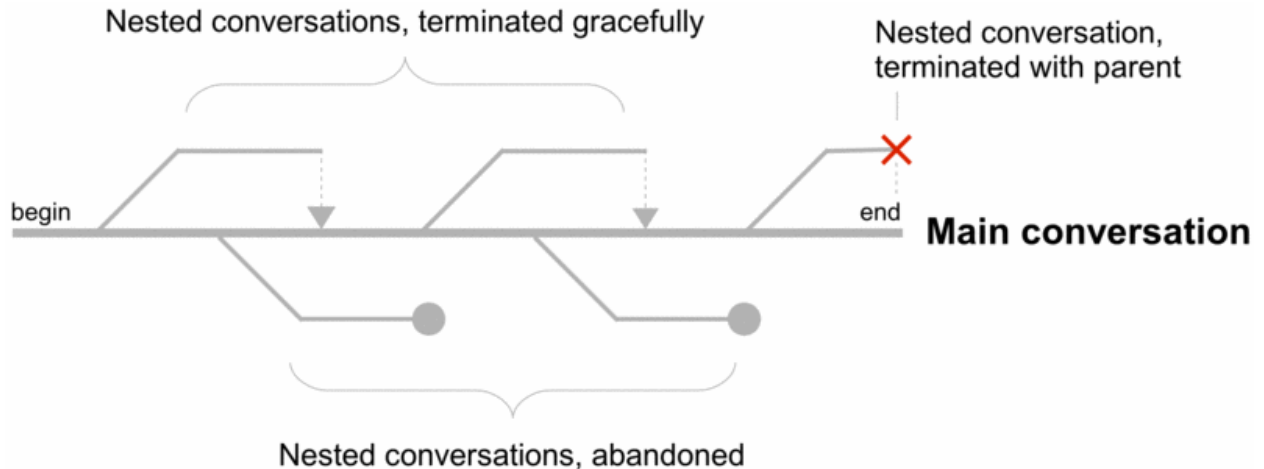


Figure 7.7 Nested conversations are branched off of the main long-running conversation.

Figure 7.7 also highlights the three ways to leave a nested conversation. The conversation can be terminated by:

- using an end propagation directive
- disabling propagation or using the browser's "back" button
- ending the parent conversation (*which terminates its nested conversations*)

When the nested conversation is terminated or abandoned, the main conversation is popped back into place. Not only is it possible to return to the main conversation when the nested conversation and related page flow ends, but you can also restore the last page visited by the main conversation.

This conversation stack "memory" takes a lot of burden off your plate of trying to keep track of where the user has been and to where they should be routed when they come off a tangent. Let's consider when you might use a nested conversation.

The need for nested conversations

You typically use a nested conversation when you want to allow the application to move along a tangential page flow without affecting the main conversation. When determining if a nested conversation is necessary, you keep the end in mind. If you need the application to perform a task within the boundaries of a conversation, but you don't want to destroy the conversation-scoped data that is available when the task begins, then it is appropriate to descend into a nested conversation.

Let's consider the example of comparing courses. Assume that while comparing courses side-by-side, the user recognizes that the information about one of the courses is incorrect. The user disables comparison mode and selects the course of interest with the intention of updating that course. When the course information is overlaid on the form, a conversation is started to keep a reference to the course instance and the persistence context that manages it. When the save button is activated, the course is updated, synchronized with the database, and the conversation is ended. Unfortunately, this means that the course comparison selections are also wiped out.

To prevent the course editor from affecting the active long-running conversation, you want to instead use a nested conversation for this purpose. A nested conversation is started when the nested propagation directive is encountered.

The CourseEdit.page.xml descriptor, which was generated by seam-gen for controlling the course editor page, begins a conversation upon entry:

```
<begin-conversation join="true"/>
```

In order to make the editor perform its work within a nested conversation, we switch to using the nested propagation directive:

```
<begin-conversation nested="true"/>
```

If there is no long-running conversation available at the time, the nested directive has the same affect as the begin directive. However, you cannot join and nest at the same time, so each time the user performs a postback, it is going to spawn yet another nested conversation. We prevent Seam from beginning a nested conversation using a conditional, only beginning a nested conversation if one is not already active:

```
<begin-conversation nested="true" if="#{!conversation.nested}"/>
```

When the user submits the editor form, the nested conversation is ended and the parent conversation is restored as the active long-running conversation, thus allowing the user to continue comparing the same set of courses that were previously selected.

One thing to note is that a nested conversation cannot modify the context variables in the parent conversation, but it does have read access to them. This is done to protect the integrity of the parent conversation while the nested conversation is in play.

The most critical part of letting the user go off on a tangent is to return them to the correct spot when they are done. If the navigation is unpredictable, the user will be afraid to leave the current page. Let's look at how to close the nested conversation and return back to its parent.

Popping out of a nested conversation

When a nested conversation is ended, Seam "pops" the conversation stack and the parent conversation is once again active in read-write mode. A nested conversation is ended in exactly the same way as a primary conversation is ended, by using a variant of the end propagation directive. Rather than terminating the long-running conversation outright, the nested conversation is flushed and the parent conversation is restored as the active long-running conversation. On the other hand, if the parent conversation were to be terminated first, then all nested conversations which it owns will also be terminated.

Left up to your own home-grown logic, you may find it is difficult to determine where to navigate when the nested conversation is ended. To help your application find its way, Seam provides a built-in method on the conversation component, bound to the `conversation` context variable, that can be used to terminate the nested conversation and redirect back to the page from which the nested conversation was spawned.

Using the restore page feature of nested conversations is most useful when you are developing a breadcrumb-link interaction. For instance, let's assume that you want to allow users to branch off of the detail page of a course to view related courses. From the page of the related course, they may

repeat the process. To allow them to go back to the previous course and back up the chain—without needing to use the browser back button—you can use nested conversations.

Let's assume there is a context variable `nearbyCourses` that is used to display a list of courses in close proximity to the current course. To allow the user to navigate to one of these courses, you place links on the page:

```
<ui:repeat var="_course" value="#{nearbyCourses}">
  <s:link view="/Course.xhtml" value="#{_course.name}"
    propagation="nested">
    <f:param name="courseId" value="#{_course.id}"/>
  </s:link>
</ui:repeat>
```

Notice the use of the nested propagation directive. When the user visits the nearby course, it is now in the context of a nested conversation. On the course detail page, we can now add a link that allows them to return to the previously viewed course if a nested conversation is detected:

```
<s:link action="#{conversation.endAndRedirect}"
  value="Return to previous" rendered="#{conversation.nested}"/>
```

The built-in action to switch back to the parent conversation is the first example you have seen of switching between conversations. Except, in this case, the nested conversation is simultaneously being terminated. In order to toggle between conversations, you have to be willing to temporarily leave one abandoned as you move to the next one. When you do that, you need to give the user controls to view the other conversations and hence return to them. That is where the conversation switchers come into play.

7.6 Switching between conversations

Abandoning a conversation may sound remiss, but it can actually be a very powerful tool for allowing the user to multitask. When a conversation is abandoned, it's not lost forever. It's just sitting behind the scenes waiting to be rediscovered. Until an abandoned conversation reaches its timeout period, it is possible to restore it using a conversation switcher widget. Switching between existing long-running conversations in the same browser window is referred to as *workspace management*. In this section, you are introduced to workspaces, how they are defined, and how you can allow the user to toggle between them.

7.6.1 The conversation as a workspace

To emphasize a point I made earlier, conversations are more than just a context. You will now learn that a conversation can also represent a user's workspace. A workspace is a named conversation. But if there was only one workspace, there wouldn't be much need for a name and we would just call it a conversation. The distinction arises when the user has more than one parallel conversations going at once. The browser can only display one page at a time and thus only focus on one conversation at a time. (JavaScript Remoting is an exception to this rule, which is covered in chapter 12).

Workspaces are useful for two reasons. First, they allow a user to pause the current task and pick up with something else with the intention of picking up with the original task later. For instance, while working through the course wizard, the user may need to return to the directory to lookup information on another course. Instead of having to open another browser tab, the user can switch tasks from within the application without losing the progress made thus far in the course wizard. In this sense, the workspace gives the application native support for tabs, sanctioning the usage scenario that the user is going to do anyway.

A workspace is also useful for limiting the number of active long-running conversations. Because users are going to inevitably perform ad-hoc navigation, conversations are going to be abandoned inadvertently. By presenting the user with a widget that allows them to acknowledge the presence of these abandoned workspaces, you encourage the user to once again give them attention and perhaps end them cleanly.

As you have learned, the application tracks and restores conversations using a conversation token, which passes along the value of the conversation id. The task of switching workspaces is going to be lost on the user if you require them to specify a numeric id to continue a conversation. Instead, you need to be able to provide a friendly name so that they recognize the workspace, thus giving them motivation to return to it. But names are more than just friendly. A conversation is not a workspace unless it has a name assigned to it.

7.6.2 Giving conversations a name

To name a conversation, you assign a name to each view ID that is used within the context of a long-running conversation. This point may be a bit counter-intuitive at first. Why would you name the page rather than the conversation itself?

If you step back and think about it, the state of a conversation changes over the course of its use. By naming the conversation only once, when it is created, that name will be too general, failing to reflect the "current" state of the conversation. Conversations are really shaped by the current page and the navigation that brought the user to that point. Even then, a page name may not help them to understand the context of that page in the conversation. Thus, page names can be defined using the EL, allowing you to tap into the conversation working set to derive a contextual name.

Page names can be assigned both in the stateless navigation model and the stateful navigation model. In both cases, the name is defined using a nested `<description>` tag within the `<page>` node. The name of the workspace is dynamic, updated to reflect the last page that has a description. Here, the terms name and description are used interchangeably.

Let's start by focusing on the stateless model. For the course comparison example, a description is assigned to each of the `<page>` nodes in the various page descriptors, shown here populated with the `view-id` attribute for clarity:

```
<page view-id="/CourseList.xhtml">
  <description>
    Course search results ({courseList.resultList.size})
  </description>
</page>

<page view-id="/CompareCourses.xhtml">
```

```

<description>
  Compare courses ({courseComparison.nameList})
</description>
</page>

```

Page nodes in the stateful navigation model can also be assigned names. It is especially important to assign descriptions in the stateful model so that it is clear which node is current in the "wait" state:

```

<pageflow-definition name="courseWizard">
  ...
  <page name="basicCourseInfo"
    view-id="/coursewizard/basicInfo.xhtml" redirect="true">
    <description>
      Course wizard (New course @ {newCourse.facility.name}):
      Basic information
    </description>
    ...
  </page>
  <page name="description"
    view-id="/coursewizard/description.xhtml" redirect="true">
    <description>
      Course wizard ({newCourse.name}): Description
    </description>
    ...
  </page>
  ...
  <page name="review"
    view-id="/coursewizard/review.xhtml" redirect="true">
    <description>Course wizard ({newCourse.name}):
Review</description>
    ...
  </page>
  ...
</pageflow-definition>

```

The description of a page is evaluated just prior to being rendered. You can see where this happens in the Seam life cycle by looking at table 3.2 in chapter 3. The description is then stored in the conversation. If the conversation is abandoned, the description will remain fixed so that when the list of conversations is presented to the user, the description will reflect its last known state.

Allowing the switch to occur

Before we get into the conversation switcher components, it is important to know what makes a workspace a candidate for switching. In Seam terms, what makes a workspace "switch enabled"?

First, the user must encounter a page with a description to promote a conversation to a workspace. If the user does not visit a page that has a description within the context of a given conversation, that conversation cannot be assigned a name and therefore cannot be restored using a workspace switcher.

To return a given view ID when the workspace is restored, the page must allow switching, which is controlled via the `<page>` node (in both the stateless and stateful models). If the view ID does not

support switching, then the last view ID that does will be restored. Switching is disabled using the switch attribute on the `<page>` node:

```
<page view-id="/FacilityList.xhtml" switch="disabled">
  <description>Select a facility</description>
  ...
</page>
```

The `<page>` can still update the name of the workspace, it is just not used when the user is returned to the workspace. The next step is creating a UI control that presents the user with a list of workspaces and allows them to select one, thus making it the active workspace, or conversation. Seam includes a handful of built-in components that aid in creating such a control.

7.6.3 Using the built-in conversation switchers

Workspaces are a new concept in web applications and it is going to take some time for developers to pick up on them. To encourage their use, Seam provides several built-in conversation switchers that you can drop into your application with very little effort. Seam provides a simple select-one menu switcher, an advantage table-based switcher, and an unordered list of breadcrumbs. The first two list all active, parallel conversations and the later represents just the active conversations stack. Let's start simple.

The basic conversation switcher

Seam's built-in conversation switcher component, `switcher`, is a ready-made component intended to be used by a `UISelectOne` JSF component, such as `<h:selectOneMenu>`. Listing 7.2 shows an example of how the `switcher` component is used in a JSF template.

Listing 7.2 A Seam conversation switcher

```
<h:form>
  <h:selectOneMenu value="#{switcher.conversationIdOrOutcome}"> #1
    <f:selectItems value="#{switcher.selectItems}" /> #2
  </h:selectOneMenu>
  <h:commandButton action="#{switcher.select}" value="Switch" /> #3
</h:form>
<Annotation #1> The current conversation
<Annotation #2> The set of conversations
<Annotation #3> The action that initiates the switch
```

The value of each select option that Seam generates is a conversation id. The names are the conversation descriptions. When the action handler is invoked, Seam uses the selected value to lookup the conversation and redirects to it, abandoning the previous conversation in the process.

HINT

Notice that I did not use `<s:link>` or `<s:button>` to invoke the action handler of the switcher component. In order for this component to work, the form must be submitted so that the

value selected in the `UISelectOne` component can be captured. The `<s:link>` and `<s:button>` components *do not* submit the form, even if they are enveloped within it.

You can also tack on your own options, which is where the `OrOutcome` part of the method becomes relevant. If the selected value is not numeric, then the action handler will return the selected value as a logic outcome so that the standard navigation rules take affect. Let's add an outcome for the user to return to the homepage and an outcome to start a new course entry wizard.

```
<h:form>
  <h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
    <f:selectItem itemLabel="Return home" itemValue="home"/>
    <f:selectItem itemLabel="Enter new course" itemValue="addCourse"/>
    <f:selectItems value="#{switcher.selectItems}"/>
  </h:selectOneMenu>
  <h:commandButton action="#{switcher.select}" value="Switch"/>
</h:form>
```

To support the `addCourse` outcome, you will need to add a global navigation rule that launches the user into the course entry wizard. In this case, the join isn't really necessary since the switcher disables conversation propagation.

```
<page view-id="*">
  <navigation from-action="#{switcher.select}">
    <rule if-outcome="addCourse">
      <begin-conversation/>
      <redirect view-id="/coursewizard/basicInfo.xhtml"/>
    </rule>
  </navigation>
</page>
```

[[screenshot]]

That wraps up what you can do with the switcher component. It is effective in switching between conversations, but it leaves a little to be desired. For one, it can only show the description, even though a conversation entry has a lot more useful information that may help the user understand the situation. Additionally, it does not allow the user to end conversations. Both of these features are supported by the built-in conversation list component.

A more powerful conversation switcher

Seam maintains a list of all long-running conversations and metadata about each one. Seam exposes this list as a collection of workspaces via the built-in Seam component, `conversationList`. This list is built from the collection of conversation entries maintained by the built-in `ConversationEntries` component, which can be used to develop your own custom switcher. Each entry in this collection has a lot more information than just the conversation id and description. The properties of a conversation entry, or workspace, are provided in table 7.6.

Table 7.6 The properties of a single conversation entry that may be used when displaying the conversation list.

Property	Type	Description
----------	------	-------------

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=354>

id	String	A value that identifies this conversation. This value is typically numeric, though business key identifiers are also supported.
description	String	The descriptive name of the conversation evaluated from the expression value specified in the <code>description</code> node of the page entry.
current	boolean	A flag indicating whether or not this conversation entry is the current conversation.
viewId	String	The last JSF view ID that was rendered while this conversation was active.
displayable	boolean	A flag indicating whether this entry is displayable, meaning it must still be active and it must have a description. The <code>conversationList</code> component automatically excludes all entries that are not displayable.
startDatetime	java.util.Date	The timestamp when this conversation began.
lastDatetime	java.util.Date	The timestamp when this conversation was last restored.
lastRequestTime	long	The timestamp when this conversation was last restored.
timeout	Integer	The timeout period that must be elapsed after its last use for this conversation to be automatically garbage collected.
nested	boolean	A flag indicating whether or not this conversation is nested.
ended	boolean	A flag indicating whether or not this conversation has ended.
removeAfterRedirect	boolean	A flag indicating that this conversation will be removed immediately after a redirect.

While you may not want to display all of these properties to the user, they can be useful when deciding how to display the list. In addition to these properties, each conversation entry also has several built-in action handlers. Table 7.7 lists the action handlers and their purposes.

Table 7.7 Action handlers on a single conversation entry that act on the selected conversation.

Action handler method	Purpose
select()	Selects the conversation entry, making it the current conversation, and redirects to the last rendered view ID when that conversation was active. The previous conversation is abandoned.
destroy()	Selects the conversation entry and ends that conversation. The previous conversation is abandoned, so it is necessary to select it again, if it still exists.

Armed with these properties and action handlers, you are ready to construct your advanced workspace control. Here we will use the `UIData` component, `<h:dataTable>`, to provide a list of active conversations. Alternatively, you can use any `UIData` component of your choice. The conversations are sorted based on the last time they were used, with the most recent conversations appearing first.

```
<h:form id="workspaces">
  <h:outputText value="There are no active workspaces"
    rendered="#{empty conversationList}"/>
  <h:dataTable value="#{conversationList}" var="entry"
    rendered="#{not empty conversationList}">
    <h:column>
      <f:facet name="header">Workspaces</f:facet>
      <h:commandLink action="#{entry.select}"
value="#{entry.description}"
    rendered="#{not entry.current}"/>
      <h:outputText value="#{entry.description}"
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```

        rendered="#{entry.current}"/>
    </h:column>
    <h:column>
        <f:facet name="header">Last used</f:facet>
        <h:outputText value="#{entry.lastDatetime}"
            rendered="#{not entry.current}"/>
        <s:convertDateTime type="time" pattern="hh:mm a"/>
    </h:outputText>
        <h:outputText value="current" rendered="#{entry.current}"/>
    </h:column>
    <h:column>
        <f:facet name="header">Action</f:facet>
        <h:commandLink action="#{entry.select}" value="Select"
            rendered="#{not entry.current}"/>
        <h:commandLink action="#{entry.destroy}" value="Destroy"/>
    </h:column>
</h:dataTable>
</h:form>

```

[[considering the use of cueballs]]

When a command link in one of the rows is activated appropriate action handler method, either `select()` or `destroy()`, is called on the conversation entry associated with that row. JSF is able to locate the appropriate conversation entry to invoke because the `conversationList` is a page-scoped component and is therefore available on a JSF postback because it is stored along with the component tree.

HINT

If you want to use `<s:link>` or `<s:button>` to operate on the entry, you will need to use bijection to expose the `conversationList` as a JSF `DataModel` using `@DataModel` and then capture the selected row using `@DataModelSelection`. You will need to move the `select()` and `destroy()` action handlers to your custom component, and then have them delegate the work of selecting or destroying the selected entry to the entry itself. The only downside is that your custom component becomes tightly coupled to the Seam API from having to import, and operate on, the `ConversationEntry` class.

The `select()` action handler on the conversation entry works the same as the equivalent method on the `switcher` component. Seam will issue a redirect to the last used view ID. The `destroy()` component is a bit trickier. When the `destroy()` method is invoked, Seam will switch to the selected conversation and issue an end operation with the before redirect flag disabled.

If you put the conversation switcher on all screens, you may run into the situation where you are trying to destroy the conversation that is currently active, which can lead to bizarre behavior. You have two options. You either disallow terminating the current conversation or you have to put some navigation rules in place to handle the event gracefully.

Without any navigation rules in place, JSF will attempt to restore the same view again, with the terminated conversation lingering on during the rendering of that view. The result can be very bizarre because the screen may still show data from that conversation, even though it is no longer available. To ensure that the conversation is ended prior to rendering the page again, you will want to install the following global navigation rule.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

```
<page view-id="*">
  <navigation from-action="#{entry.destroy}">
    <end-conversation before-redirect="true"/>
    <redirect/>
  </navigation>
</page>
```

The `<end-conversation>` tag is not redundant. It is there to change the semantics of how the conversation is ended. The empty `<redirect>` tag simply issues a redirect to the same view ID. With this rule in place, destroying a workspace will result in the page being rendered with no conversation active. A `conversation-required` on the page being redisplayed can prevent any undesirable behavior. Unfortunately, without installing some of your own logic, it is not possible to restore the previous conversation that was active when the `destroy()` method was triggered using the built-in component.

NOTE

The only downside of switching to another conversation is that any non-submitted form data will be lost. You can work around this issue by using an Ajax component library such as Ajax4jsf that can periodically synchronize the state of a form with the server. That way, when it comes time to abandon a conversation, the partially complete form data is already preserved in the server-side model.

Workspaces truly are an innovative concept. Their use can cut down on the proliferation of tabs in the user's browser because the user will not feel like they are "losing their place" by taking a temporary detour.

TIP

The conversation switcher is a great way to test conversation timeouts. Open two tabs in your browser, one that you are using to test the screens in the application and one which displays the list of workspaces. Use the workspace tab to destroy the conversation that is active in the application tab to emulate the conversation reaching its timeout point.

The conversation switcher just shown displays both top-level and nested conversations. It is also possible to create a switcher that moves solely along the ancestral chain of a series of nested conversations using the conversation stack component.

Tracing your steps with breadcrumbs

Breadcrumb navigation complements switching between parallel conversations. Each breadcrumb represents a point where the conversation was branched to create a nested conversation. Since conversations can be nested to arbitrary depth, it is possible to have a long chain of breadcrumbs from which to choose. Seam makes generational conversations available via the built-in `conversationStack` component.

Your application must support a nested conversation model for this component to be populated. Navigating between related golf courses is a perfect use case for this component. We will use the Facelets tag `<ui:repeat>` to create a delimited linear chain.

```

<h:form id="breadcrumbs">
  <s:fragment rendered="#{not empty conversationStack}">
    Trail:
    <ui:repeat value="#{conversationStack}" var="entry">
      <h:outputText value=" > " rendered="#{entry.nested}"/>
      <h:commandLink action="#{entry.select}"
value="#{entry.description}"
      rendered="#{not entry.current}"/>
      <h:outputText value="#{entry.description}"
      rendered="#{entry.current}"/>
    </ui:repeat>
  </s:fragment>
</h:form>

```

NOTE

If the conversation is not nested, then the conversation stack will have only one entry. If you don't want to display the conversation stack in this case, you can use the `#{conversation.nested}` expression to enable a conditional check.

As you can see, the conversation stack is used in exactly the same way as the conversation list component. In fact, the only difference is that it just consists of hierarchical entries rather than parallel conversations. You could provide a `destroy()` action as well, which would terminate the selected entry and all of its descendants.

While Seam provides you with built-in components to either switch between parallel conversations or ancestor conversations, it does not include a component that allows you to view both simultaneously. Clearly, providing such a workspace switcher would provide ultimate control over the existing workspaces. Let's tap into the Seam's conversation API to make it happen.

7.6.3 Taking control of the current conversation

With all the attention on the other active conversations, the current conversation—the one servicing the current request—is feeling left out. Seam gives you a wealth of properties and controls that are related to the current conversation, just as it does for each entry in the collection of active long-running conversations. Having information about the current conversation can help tremendously when making decisions about navigation or rendering markup on the page. The built-in controls can be useful as page actions or to serve as action listeners for links and buttons. These methods offer another alternative for controlling the boundaries of the current long-running conversation.

You can access information about the current conversation, not surprisingly, by using the `conversation` component. You have already seen a couple of this component's properties used in this chapter. The value expression `#{conversation.id}` was used to reference the id of the current long-running conversation in order to propagate the conversation across a non-JSF request. Table 7.8 provides a complete list of properties that are available on this component.

Table 7.8 The properties of the current conversation available under the context variable named conversation.

Property	Type	Description
id	String	A value that identifies this conversation. This value is typically numeric, though business key identifiers are also supported.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>

parentId	String	The conversation id of the parent conversation of this nested conversation.
rootId	String	The conversation id of the primary (top-level) conversation of this nested conversation.
description	String	The descriptive name of the conversation evaluated from the expression value specified in the <code>description</code> node of the page entry.
viewId	String	The last JSF view ID that was rendered while this conversation was active.
timeout	Integer	The timeout period that must be elapsed after its last use for this conversation to be automatically garbage collected.
longRunning	boolean	A flag indicating whether or not this conversation is long-running. Conversations scheduled to be removed after a redirect are not considered to be long-running.
nested	boolean	A flag indicating whether or not this conversation is nested.

The conversation component also has a number of built-in action handlers that can be used to control the current conversation, or to switch to a related conversation. These action handlers are listed in table 7.9. Remember, these action handlers can be conveniently used as page actions defined on page nodes in the pages.xml configuration.

Table 7.9 Action handlers on the conversation context variable that can be used to control the current conversation.

Action handler method	Purpose
redirect()	Switch back to the last known view ID for the current conversation.
endAndRedirect()	End the current nested conversation and redirect to the last known view ID for its parent conversation.
endBeforeRedirect()	End the current conversation and set the before redirect flag to true. Does not trigger an automatic redirect.
end()	End the current conversation and set the before redirect flag to false. Does not trigger an automatic redirect.
leave()	Step out of the current conversation. A new temporary conversation will be initialized and used for the duration of the request.
begin()	Start a new long-running conversation if one is not already active. This method is the equivalent of setting join to true.
reallyBegin()	Start a new long-running conversation regardless if one already exists. This method is equivalent to setting join to false.
beginNested()	Start a nested conversation by branching off of the current long-running conversation. If a long-running conversation is not active, an exception will be thrown.
pop()	Switch to the parent conversation, leaving the current conversation intact. Does not trigger a redirect.
redirectToParent()	Switch to the parent conversation, leaving the current conversation intact, and redirect to the last known view ID for the parent conversation.
root()	Switch to the root level conversation, leaving the current conversation intact. Does not trigger a redirect.
redirectToRoot()	Switch to the root level conversation, leaving the current conversation intact, and redirect to the last known view ID for the root conversation.

There are likely more controls here than you may ever use. But that is a good thing, because it means that if you have some unique circumstance that requires you to manipulate the current conversation, or switch to a related conversation, the groundwork has already been laid.

Conversation switching is likely going to be a new, yet surprisingly refreshing, experience for the user. If instrumented correctly, they will not have to resort to the crutch of multiple tabs to be

productive. Instead, you can bring the power of tabs into the application by supporting multiple workspaces and providing the user with a control to move between them.

7.7 Summary

Users become frustrated when their story is forgotten. It forces them to have to go over the same information again and again to keep the application up to speed, rather than enabling them to pick up where they last left off. As a result of this failure to track state, the user is kicked back to the starting point and ready to hang up on your application. That is the void conversations are designed to fill.

Conversations are one of the crowning features of Seam. They reach into almost every aspect of the framework. That explains why it has been so difficult to refrain from presenting the material in fragments up until now. Hopefully, those missing pieces came into line for you in this chapter.

You learned in this chapter how conversations are more than just a context. Instead, they are units of work from the perspective of the user. At times, a unit of work may only span one request, which Seam supports using a temporary conversation. The temporary conversation ensures that context variables scoped to the conversation are held until the response is rendered, even if the life cycle is unhinged by a redirect—known as the "flash" scope in some other frameworks.

That discussion gave rise to the long-running conversation, which extends the conversation to span multiple pages, as dictated by well defined boundary points. You learned that long-running conversations are formed by promoting a temporary conversation to long-running when a "begin" conversation propagation directive is encountered. You studied the wide variety of propagation directives, which include annotations, in the pages configuration, using UI component tags, or via the Seam API. You came to understand that a conversation is really just a segment of the HTTP session and that the reference to that memory segment is passed along as a token in the request. The process of how a long-running conversation ends was explained, which happens either by demoting it to a temporary conversation when an "end" conversation propagation directive is encountered or because it times out from lack of use, and automatically garbage collection mechanism. What you were able to take away from this discussion is that it is no longer your responsibility to manage stateful data tucked away in the HTTP session.

The discussion then turned from singular to plural as you learned that you can have multiple conversations going at once, either sharing a nested relationship or isolated from one another in the session so as to prevent interference. This discussion gave rise to the concept of workspaces, which you learned represent Seam's awareness of concurrent conversations.

Knowing that users of your application are multitaskers, you learned how to extend the concept of a "unit of work" to be multiple units of work by leveraging Seam ability to switch between workspaces. This feature enables you to provide your users with conversation switching controls so that they can pick up where they left off amongst the parallel units of work. While Seam provides a number of built-in conversation controls, you learned to tap into the rich conversation API and the RichFaces component palette to create a custom conversation switcher control.

By no means did this chapter exhaust everything there is to say about conversations. This chapter is just the beginning. But it certainly builds a strong foundation. What you have learned in this chapter is close to everything you need to know about controlling conversations. From here on out, you are going to learn more about what you can do to use conversations. Conversations play a central role in Seam's managed persistence support, providing an ideal place to store the persistence manager to take

advantage of its in memory representation of the database. Before you can learn how conversations and persistence play off of one another, you first need to take a lesson in understanding Java persistence so as to develop a vocabulary that is used to discuss Seam's pioneering work in the persistence field.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=354>