Covers Seam 2

SEAM IN ACTION

SAMPLE CHAPTER

Dan Allen

FOREWORD BY Norman Richards

MANNING

Seam in Action

by Dan Allen

Chapter 7

Copyright 2009 Manning Publications

brief contents

PART 1 TEEING OFF WITH SEAM 1

1 ■ Seam unifies Java EE 3
2 ■ Putting seam-gen to work 29

PART 2 SEAM FUNDAMENTALS 81

3 ■ The Seam life cycle 83
4 ■ Components and contexts 130
5 ■ The Seam component descriptor 179
6 ■ Absolute inversion of control 219

PART 3 SEAM'S STATE MANAGEMENT 271

7 ■ The conversation: Seam's unit of work 273
8 ■ Understanding Java persistence 325
9 ■ Seam-managed persistence and transactions 352
10 ■ Rapid Seam development 380

PART 4 SINKING THE BUSINESS REQUIREMENTS 431

11 ■ Securing Seam applications 433
12 ■ Ajax and JavaScript remoting 475
13 ■ File, rich rendering, and email support 511

Part 3

Seam's state management

Part 1 presented the motivation for why Seam was created and demonstrated ways it simplifies development of web applications. You used seam-gen to quickly put together a Seam-based application and agile development environment. Part 2 got into the guts of Seam by teaching you to define and configure components and getting them to communicate. What sets Seam apart from other web-oriented frameworks is its focus on state management. This term may not mean much to you right now, but trust that it plays a key role in what you'll learn to master in the next three chapters: conversations, page flows, the extended persistence context, application transactions, and entity home components.

Chapter 7 introduces conversations as a way to effectively string together requests. You define conversation boundaries using a familiar declarative approach. You also learn to orchestrate a conversation with a page flow and to let the user multitask using workspaces.

Chapter 8 puts conversations aside initially to cover Java persistence, the ORM mechanism that translates Java objects to and from database records. The end of the chapter sees a return to state management when the extended persistence context in EJB 3 is introduced. This construct ensures persistent objects remain managed so that updates to the database require no programming, related objects can be fetched on demand, and database reads are kept to a minimum.

In chapter 9, you learn that conversations provide the perfect vehicle for extending the persistence context. This chapter introduces Seam-managed persistence as an alternative to its complement in Java EE, with a number of extensions weaved in. More important, propagation of the persistence context is handled transparently, a relief from complex rules in EJB 3. Seam also offers a unique approach to transactions by wrapping them around each request and facilitating application transactions that span multiple requests.

In chapter 10, you get to bring together everything you have learned about Seam and perform rapid development using the Seam Application Framework. This chapter helps you appreciate how much you have matured as a Seam developer throughout this part.

7 The conversation: Seam's unit of work

This chapter covers

- Managing state with a conversation
- Controlling long-running conversations
- Switching between workspaces
- Defining stateful page flows

Seam helps establish a rich user experience by stretching the boundaries of a unit of work to cover a use case—a determinate interchange between the user and the application. In this chapter, you'll learn how Seam's conversation context can host the working set of data needed to support this interchange. Seam's conversation is contrasted with traditional state management techniques, demonstrating how it both relieves the burden of handling this task and gives multipage interactions a formal representation in a web application. Conversations open the door to more advanced state management techniques—such as stateful page flows, nested conversations, and workspaces—that further enrich the user experience.

Conversations are one of Seam's crowning features, touching many areas of the framework and bridging earlier chapters with those that lie ahead. In fact, CRUD

applications created by seam-gen use conversations to manage the use case of adding and modifying an entity instance. To witness conversations in action before beginning this chapter, you can study them at work in the Open 18 application. This chapter starts off by defining a use case and then explores how aligning it with a stateful context can improve the user's experience.

7.1 Learning to appreciate conversational state

After returning from your golf getaway, you learn that, in your absence, someone has been having fun at your expense, courtesy of your credit card number. To reclaim your assets, you have to endure the disparate customer service process deployed by your bank. I'm sure this exchange will strike a familiar chord.

"How can I help you?"

You take this cue to launch into your rant about being a victim of fraud, your whereabouts during the previous week, and which charges you're disputing.

A brief pause is interrupted by, "Can I have your account number?"

Your momentum is temporarily interrupted as you wait for your call to be logged. The agent then informs you that you need to be transferred to the fraud department, which is properly equipped to deal with this matter. There's 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 began the day a victim of fraud. Now you have become a victim of a stateless process. Critical information about your situation failed to make the leap from the customer service representative to the fraud representative. This mishap could have been avoided had the two representatives engaged in a conversation during the switch to retain a record of your story. Unfortunately for you, they don't see the big picture. As soon as you're handed off, there's another call to answer.

That's exactly how requests are handled in a web application, which rests atop a stateless protocol (HTTP). The result is that data tends to be dropped between page requests, blind to the ongoing use case. Seam acknowledges that all requests occur in the scope of a conversation, that a conversation can span multiple requests, and that conversations are subsets of a user's session, as illustrated in figure 7.1.

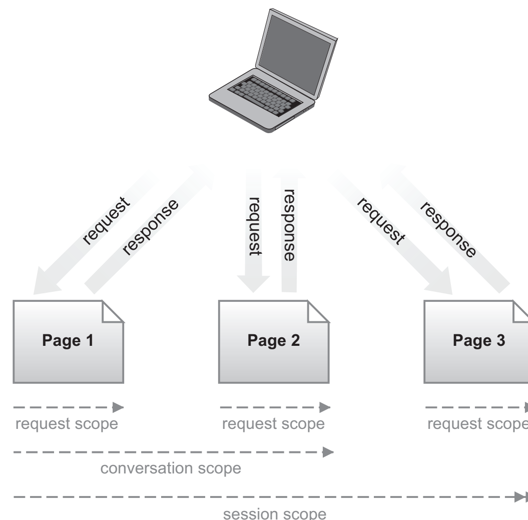

Figure 7.1 The conversation scope ties together individual requests, but is more granular than the session.

Through use of conversations, Seam supports building stateful interactions that bridge the gap between requests. As a result, state can be tracked until the user's goal is accomplished, not just to satisfy an atomic step along the way.

7.1.1 Redefining the unit of work

You may think of a unit of work in terms of a database transaction. The story just told is faithful to this definition at the expense of the caller's time. This short-lived unit of work is problematic because it's not stateful (in the long-running sense). Seam redefines the unit of work by looking at it from the user's perspective, coining what is known as a conversation. During a conversation, database transactions, page requests, and other atomic units of work may come and go, but the overall process isn't deemed complete until the user's goal is accomplished. In a conversation, the state is said to be extended. In the next chapter, you'll learn that the persistence context can also be extended to match this timetable.

The linkage between page requests in a conversation is established through the use of a special token and a partitioning of the HTTP session, which you'll learn about in section 7.2. The conversation's lifetime is controlled by declarative boundary conditions, covered in section 7.3. First, I want to focus on the challenge of establishing a well-defined stateful context in a web application and how this task has been traditionally handled. You could go so far as to say that propagating state in a web application is a downright burden. After studying some of the alternatives, I'll segue into how Seam's conversation provides relief.

7.1.2 The burden of managing state

All applications have state, even those classified as stateless (e.g., RESTful applications). Some applications stash state away in server-side contexts such as the HTTP session scope or the JSF page scope. So-called stateless applications just weave state into the URL or hide it away in hidden form fields. A majority of applications likely use a mix of these strategies. The real question is, as users traverse from one page to the next, how much support do they get from the framework for managing and accessing that state?

I'm sure that in the past you have worked very hard to save data between requests and subsequently prepare it for use in the business logic. Whether you wrote one of those applications that has more hidden form fields than visible ones or one with as many Struts `ActionForm` classes as business components, you have felt the burden of managing state. That's not to say that RESTful URLs, hidden form fields, request parameters, or cookies aren't viable. It's just that when they are means to an end, where the end is to restore the state from the previous request, they cause a lot of work. Seam attempts to make state readily available in contexts that closely represent the life cycle of that state (i.e., a use case). Not only does the conversation context blend state between requests for the duration of a use case, it avoids destroying the identity of objects as a result of serialization, which is the main problem with the traditional approaches.

PASSING DATA THROUGH THE REQUEST

One approach to propagating state is to send it along as part of the request in one of these forms:

- Request parameters (i.e., hidden form fields or the query string)
- As part of the URL (e.g., /course/view/9)
- Browser cookies

All of these options work by disassembling some server-side object into bite-sized chucks, tunneling those parts through the request as string values, then reassembling the objects on the server, as figure 7.2 illustrates. There are times when a RESTful URL, request parameter, or cookie is the right tool for the job. However, if you're working with any decent-size set of data, having to prepare this translation on every request is downright tedious.

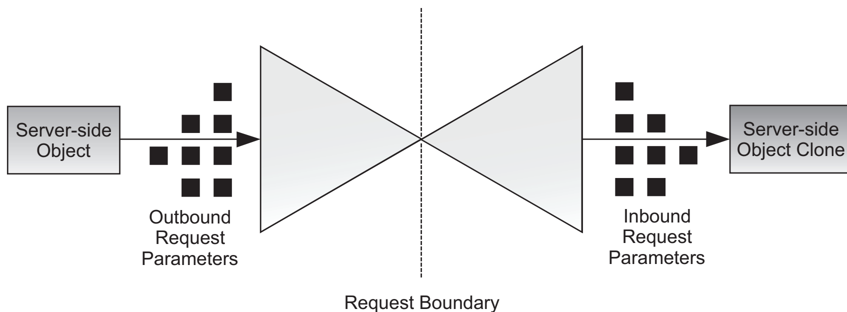

Figure 7.2 Passing an object using request parameters is akin to teleportation.

You learned in chapter 3 that Seam's page parameters offer some relief here by automatically translating objects to and from HTTP request parameters on a per-page basis. The downside to page parameters is that they are hardwired to the page definition. Since a page may be used in more than one set of circumstances, the configuration could result in data being propagated even when you don't need it; or worse, it gets in the way.

The biggest drawback of parameter-based propagation in general is that the server-side object loses its identity when it passes through this funnel. The object that's built on the server, beyond the request boundary, is a clone of the original object, possibly even a partial one. This cloning process makes it impossible to transfer a resource that relies on its identity being maintained, such as a persistence manager or managed entity instance. For an object's identity to be preserved, the object must be stored in a server-side context, such as the JSF UI component tree (which requires server-side state saving), the HTTP session, or as you'll soon learn, the conversation. Then it's only necessary to pass a token to the server to restore the context and the objects it contains.

STASHING STATE IN THE JSF UI COMPONENT TREE

You learned in chapter 4 that the root of the JSF UI component tree has an attribute map that can be used to save data across a JSF postback. Seam's page context and the `<t:saveState>` component tag from the MyFaces Tomahawk library both

offer transparent access to this map, among alternatives. As elegant as these abstraction layers may be, it doesn't diminish the fact that this map is merely the JSF equivalent of hidden form fields.

Using the UI component tree as a stateful context has problems that match those cited previously. First, you must reestablish the set of variables in the page context when the UI component tree is rebuilt (which happens during any postback request that issues a redirect or renders a different view). The second problem is that the UI component tree doesn't guarantee that the identity of the objects will be maintained. JSF state saving can be done either on the client side or the server side. When client-side state saving is used, the restored objects are clones of the originals, having the same problem as passing parameters through the request. If you don't have control over the state-saving setting, it's best not to rest an object's identity on such unstable ground.

The challenge of maintaining object identity is often solved by using the HTTP session.

STORING DATA IN THE HTTP SESSION

The HTTP session can be used to store arbitrarily complex objects while preserving their identity. This context is shared by all browser tabs and windows that restore the same session token and typically lasts on the order of hours or days. While this storage mechanism 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 of data that consumes excessive amounts of memory and complicates multiwindow application usage. Let's explore these issues.

The HTTP session is well suited for data that you want to retain across all requests made by a given user, such as the user's identity. However, the session isn't a good place to store data for a specific use case. It may seem harmless when the user is accessing the application from a single window—but what happens when the user spawns multiple tabs or windows? Because the session identifier is stored as a cookie in the user's browser, and most browsers share cookies between tabs and windows (herein referred to as tabs), the result is that the multiple tabs naively share the same session data. If the application doesn't recognize this sharing, it can cause data to be manipulated in conflicting ways.

NOTE The session identifier can also be passed through the URL, known as URL rewriting. When URL rewriting is used, links that contain the same session identifier restore the same session, even if opened in a new tab.

Consider the use case of updating a golf course record. Assume that the golf course is stored in the HTTP session while it's being modified. If you select a course to modify in one browser tab and then select a different course to modify in a second tab, the second course selection on the server overwrites the first. When you click Save in the first tab, assuming the changes are applied directly to the record in the session, you inadvertently modify the second course instance. Things get even trickier if you're working with a multipage wizard, since the leakage of data can be less apparent. Yet another problem is that data in the session isn't protected from concurrent use, so if two requests try to access session data simultaneously, it can lead to a race condition.

The most severe problem with the session scope is that it is mostly unmanaged. If objects continue to build up in the session, and there's no application-provided garbage collector to clean it out, memory leaks that impact the performance of the application will occur, thus affecting all of the users.

It's possible to work around the aforementioned problems by using the session with care or by putting in synchronization and locking code to prevent collisions, but that's a burden on you as a developer. In general, heavy use of the session scope is a common source of bugs, and the unexpected behavior it causes is often difficult to reproduce in a test environment.

NOTE Cookies have the combined problem of request parameters and session data. They only store string data, capped at a fixed size (~4K), and they cannot be partitioned by use case. Their utility is in identifying a repeat visitor or storing basic user preferences.

Although the existing storage options are workable, they aren't well suited for maintaining an isolated working set of data for a use case. Clearly there is room for a better solution. Surprisingly enough, that solution lies in the HTTP session. Despite my having just bashed the session for its weaknesses, it's not all bad. It simply needs to be partitioned and better managed. That's exactly what Seam does. The conversation context is designed to be a well-behaved abstraction layer over the HTTP session.

7.2 The conversation context

The conversation context is one of two Seam contexts introduced to serve business-world time frames as opposed to servlet life cycles (the other is the business process context). From reading the previous section, you should have a clear picture as to a conversation's purpose. In this section, you'll learn how it's maintained.

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 7.3. Seam uses the conversation context as the home for a working set of context variables.

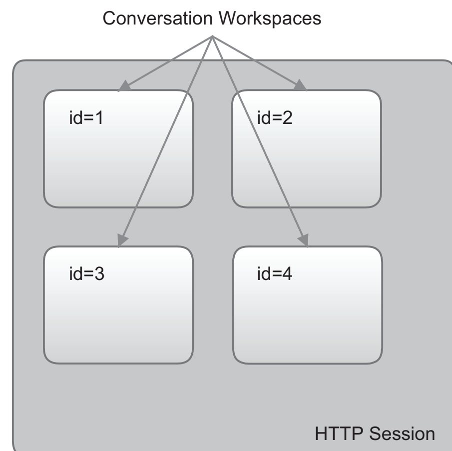

Figure 7.3 Conversation workspaces are isolated segments of the HTTP session, each assigned a unique identifier.

You may shudder at the mention of using the HTTP session to store the conversation, given the problems cited in the last section. However, a conversation doesn't suffer from the same problems as its parentage. First and foremost, the lifespan of a typical conversation is on the order of minutes, whereas a session can last on the order of hours. This difference is made possible by the fact that a conversation has its own distinct life cycle,

which Seam manages. Each conversation can have its own timeout period, which defaults to the global timeout setting, covered in section 7.3.5. Additionally, concurrent conversations are kept isolated, unlike session attributes, which just get jammed together in a single map.

Since conversations are stored in the session, two requirements must be met:

- Conversation-scoped components must implement `java.io.Serializable`.
- The session timeout, defined in web.xml, must exceed all conversation timeouts.

A conversation has a clear set of life-cycle boundaries that coincide with the boundaries of a use case. When the user triggers a condition that begins a conversation, a new managed 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 region of the session. The conversation id is passed on to the next request as a request parameter, hidden form field, or JSF view root attribute. Fortunately, propagation of the conversation id is handled transparently in a Seam application. As a result of the conversation id and session token being sent together to the server, the conversation is retrieved from the session and associated with the request.

NOTE Although the HTTP session is used as the storage mechanism for a conversation, understand that the memory footprint is strikingly low because conversations are aggressively managed. There's no chance of them lingering on to cause memory leaks.

The conversation context is an ideal place to store data that's needed over the extent of several pages. It leverages the session's ability to store arbitrarily complex objects while preserving object identity, but doesn't suffer from memory leaks or concurrency problems. What makes conversations truly unique is that they remain isolated from one another.

SOLVING THE MULTIWINDOW CONCURRENCY PROBLEM

Let's revisit the scenario in which different golf course records are being edited in separate browser tabs. This time around, we'll assume that a conversation is being used to manage each use case. The user begins by selecting a golf course in the first tab, which starts a new conversation and presents the user with an update form. The user then switches to the second tab and selects a different course, again resulting in a conversation being created and the rendering of an update form. Each tab now has its own conversation. The user then switches back to the first tab and clicks Save. The form values from that tab are sent to the server along with the conversation id. On the server, the conversation context is retrieved from the session using the conversation id. The course instance is pulled out of that conversation, the form values are applied to it, and finally it's synchronized to the database.

Although the two tabs are serving the same use case, with the same context variables, the data is kept isolated. Conversations don't suffer from leaky behavior because they aren't shared across all tabs and windows like the session. Instead, a conversation can be reserved for a single tab and restored on each request by passing the conversation id. As such, activity occurring in one tab doesn't affect other tabs (that use different

conversations). Although conversations prevent unwanted sharing of data between separate use cases, sharing data across multiple requests in the same use case is a desirable feature of the conversation.

A BUSINESS TIER CACHE

The conversation context provides a natural caching mechanism that's readily controlled from the application, allowing cached data to be relinquished or refreshed in accordance with the business logic. You can even provide the user with controls that force the data to be refreshed on demand. If the conversation is abandoned, it's not long before this state is cleaned up by Seam (unlike with state stored in the HTTP session).

Caching data is critical because it avoids redundant data inquiries. If you cache database result sets, it means that you don't have to consult the database again when there's no expectation that the data has changed. You should take advantage of this opportunity because, of all the tiers in your application, the database tier is the least scalable. Don't abuse the database by repeatedly asking it to retrieve the same data over and over again. When the application fails to track data that it was already fed, it hurts the performance of both the database and the application.

Reducing load on the database is one of the primary concerns of an ORM. An ORM supports two levels of caching. The first-level cache, known as the persistence context, holds the collection of all entities retrieved by a single persistence manager, which you'll learn about in chapters 8 and 9. If the persistence manager is scoped to the conversation, then the ORM works naturally to reduce database load.

The second-level ORM cache is shared by the persistence managers and holds persistent objects previously loaded through any persistence context. It is used as a way to reduce traffic between the application and the database. Employing an intelligent algorithm, it attempts to keep the cache in sync with the changes made to the database. However, regardless of how good this logic is, within the scope of a use case, it lacks the business-level insight to know exactly when data should be considered stale. Expecting the second-level cache to make up for the application's inability to retain data is a misuse of the technology.

The need for a stateful context acting as an intermediary between the browser and the database is especially important in the world of Web 2.0, where Ajax requests are sent to the server at a rate that far exceeds the previous usage pattern of web applications. Requests that leverage the conversation context save database hits and are faster since they're returning data that's held close at hand. The conversation plays another important role in Ajax: preventing the requests from accessing data concurrently.

PREVENTING CONCURRENCY PROBLEMS

Seam serializes concurrent requests that access the same conversation. This means only one thread is allowed to access a conversation at any given time. In pre–Web 2.0 applications, this might help deal with a double submit, but when Ajax starts firing off requests like they are going out of business, the likelihood of your data entering an inconsistent state as a result of concurrent access dramatically increases. Seam keeps those Ajax requests in line, so you can be confident that conversation-scoped data won't be modified by a second incoming request while the first request is being served.

Conversations fit very naturally with Ajax. 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 won't suffer. You'll learn more about how well Seam and Ajax fit together in chapter 12.

Having explored ways in which the conversation context solves the need for a user-focused stateful context, let's examine the types of data you might typically store in a conversation as you prepare to use it.

7.2.2 What you might store in a conversation

The conversation provides a way for data to be stashed away 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. Additional information is accumulated in the conversation as the user moves from screen to screen. There are four classifications of data that a working set is used to store, all of which are demonstrated in this chapter:

- *Nonpersistent data*—An example of nonpersistent data is a set of search criteria or a collection of record identifiers. The user can establish the state in one request and then use it to retrieve data in the next. This category also includes configuration data (such as the page flow definition).
- *Transient entity data*—A transient entity instance may be built and populated as part of a wizard. Once the wizard is complete, the entity instance is drawn from the working set and persisted.
- *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 form. When the user submits the form, the form values are applied to the entity instance that's stored in the working set (whose object identity has been preserved) and the changes are flushed to the database transparently.
- *Resource sessions*—The conversation context offers an ideal mechanism for maintaining a reference to an enterprise resource. For instance, the persistence context (a JPA `EntityManager` or Hibernate `Session`) can be stored in the conversation to prevent entity instances from becoming detached prematurely. The next several chapters focus on how conversations benefit persistence management.

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 how to control conversations by defining conversation boundaries.

7.3 Establishing conversation boundaries

The conversation context is unique from other Seam contexts you have used so far in that it has explicit boundaries dictated by application logic, as opposed to implicit

boundaries that correlate with a demarcation in the servlet or JSF life cycle. The boundaries of the conversation context are controlled using conversation propagation directives. This section introduces these directives and demonstrates how they're used to transition the state of a conversation and effectively manage its life cycle.

7.3.1 A conversation's state

A conversation actually has two states: temporary and long-running. There is also a third state, nested, which is a characteristic of the long-running state. Nested conversations are covered in section 7.4.2. Right now, I want to focus on the first two states.

Switching the state of a conversation is referred to as *conversation propagation*. When you set the boundaries of a conversation using the conversation propagation directives, you're not initiating and destroying the conversation, but rather transitioning it between a temporary and long-running state.

TEMPORARY VS. LONG-RUNNING CONVERSATIONS

Most of the time, when people talk about Seam conversations, they're referring to *long-running* conversations. The discussion in the early part of this chapter pertains to long-running conversations. A long-running conversation remains active over a series of requests in correlation with the transfer of the conversation id. In the absence of a long-running conversation, Seam creates a *temporary* conversation to serve the current request. A temporary conversation is initialized 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 achieving the same result as the flash hash in Ruby on Rails: transporting data across a redirect. In Seam, the temporary conversation carries conversation-scoped context variables across a redirect that may occur during a JSF navigation event. This works by maintaining the temporary conversation until the redirect is complete. So to clarify, a temporary conversation is destroyed after the *Render Response* phase ends, even if it's preceded by a redirect. The most popular use of a temporary conversation is to keep JSF messages alive during the redirect-after-post pattern, assuming those messages are registered using Seam's built-in, conversation-scoped `FacesMessages` component.

The other purpose of a temporary conversation is to serve as a seed for a long-running conversation. A long-running conversation is nothing more than a temporary conversation whose termination has been postponed. This postponement lasts from the time the *begin* conversation directive is encountered up until the *end* conversation directive is met. Instead of just surviving a navigation redirect, a long-running conversation is capable of surviving a whole series of user interactions. Only when the conversation reacquires the temporary state is it scheduled to be terminated. In Seam, every request is part of a conversation. You just have to decide how long you want that conversation to last.

CONVERSATION PROPAGATION DIRECTIVES

Learning to use long-running conversations involves learning the conversation propagation directives, listed in table 7.1, and how they transform a temporary conversation to and from a long-running conversation. You can think of conversation propagation

Table 7.1 A list of the conversation propagation directives

Propagation type	Description
begin	Promotes a temporary conversation to a long-running conversation. An exception is thrown if a long-running conversation is already active.
join	Promotes a temporary conversation to a long-running conversation. No action is taken 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, suspends it and adds a new, long-running conversation to the conversation stack. If a long-running conversation is not active, promotes the temporary conversation to a long-running conversation.
none	Abandons the current conversation. The previous conversation is left intact and a temporary conversation is created to serve the incoming request.

directives serving a parallel purpose for a conversation as transaction propagation directives do for a transaction.

The conversation propagation directives can be applied using the following means:

- Method-level annotations
- UI component tags
- Seam page descriptor
- Seam conversation API
- Stateful page flow descriptor (end conversation only)

These variants are provided to accommodate different usage and design scenarios, allowing you to establish conversation boundaries where it makes the most sense in your application. You'll learn to use the conversation propagation directives in the next section.

The conversation propagation directives dictate the life cycle of the conversation. Figure 7.4 diagrams this life cycle, showing how the state of the conversation changes during the request as a result of encountering a conversation propagation directive.

Let's step through the diagram in figure 7.4. At the start of the request, a long-running conversation is restored if the conversation id is detected among the request parameters. If the conversation id is absent or invalid, Seam initiates a new, temporary conversation. At any point during the processing of the request, the conversation may change state as the result of encountering a conversation propagation directive. The begin directive transitions a temporary conversation to long-running. The join directive has the same effect as begin, except that it can enter into an existing long-running conversation, whereas the begin directive raises an exception in this case. The nest directive can also begin a long-running conversation, but if one exists, a new conversation is created, temporarily suspending the existing one. The end directive sends the long-running conversation back to its temporary state. At the end of the request, the temporary conversation is destroyed, whereas the long-running conversation is tucked

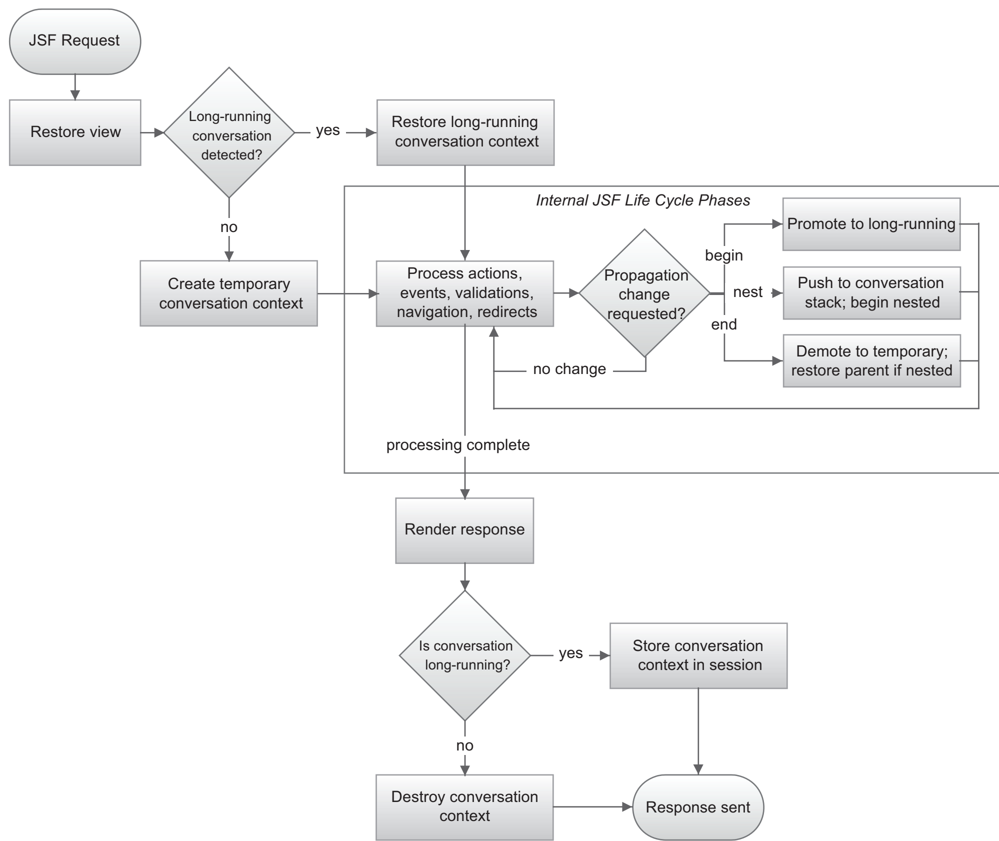

Figure 7.4 How the conversation propagation directives affect the conversation during a request

away in the HTTP session to be retrieved by a subsequent request. Although not shown, if the conversation being restored is invalid or has previously timed out, the user is notified and forwarded to a fallback page if one is configured.

Seam uses a built-in component to keep track of the conversation's state, including its relationships to nested and parent conversations. It's important to know that this component exists as you'll often find that you need to reference it.

THE CONVERSATION COMPONENT

Seam maintains the state of the conversation in an instance of the built-in conversation-scoped component named `conversation`. This component provides a wealth of information about the current conversation in the form of properties and exposes methods for acting on the conversation. These properties and methods are listed in table 7.2.

The most important property on the conversation component is the conversation id, which is typically accessed using the value expression `#{conversation.id}`. The conversation id is used to restore the conversation from the session at the beginning of a request. You'll see this expression used often in this chapter. The properties on the conversation component aid in making decisions about navigation or rendering

Table 7.2 The properties and methods on the built-in conversation component

Property	Type	Description
`id`	`String`	A value that identifies this conversation. This value is typically numeric, though business key identifiers are also supported.
`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 `<description>` 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 elapse after its last use for this conversation to be automatically garbage collected.
`longRunning`	`boolean`	A flag indicating whether this conversation is long-running.
`nested`	`boolean`	A flag indicating whether this conversation is nested.
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 only if one is not already active. This method is equivalent to using the join directive.	
`reallyBegin()`	Start a new long-running conversation without checking if one already exists. This method is equivalent to to using the begin directive.	
`beginNested()`	Start a nested conversation by branching off 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 an automatic redirect.	
`redirectToParent()`	Switch to the parent conversation, leaving the current conversation intact, and redirect to the last known view ID for the parent conversation.	

Table 7.2 The properties and methods on the built-in conversation component

Method	Purpose
`root()`	Switch to the root level conversation, leaving the current conversation intact. Does not trigger an automatic 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.

markup conditionally. The methods on the conversation component can change the conversation's state and are often used as page actions or as the action method on links and buttons.

With this component in hand, you're now ready to learn how to define the conversation boundaries. As you read through the next couple of sections, I encourage you to study the options you have for defining these boundaries, decide on a favorite approach, and try to adhere to it a majority of the time. Just because you can define each boundary a handful of ways doesn't mean you should use every single variation, at least not without good reason.

7.3.2 *Beginning a long-running conversation*

To demonstrate the use of a long-running conversation, let's work through an example of a multipage wizard that captures information about a golf course and adds it to the Open 18 directory. Entering data for a golf course can be quite intimidating, so a wizard is used to break up the form into short, logical steps, shown in figure 7.5. Each box in figure 7.5 represents a screen with 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's available when the final screen is complete and the course is persisted to the database.

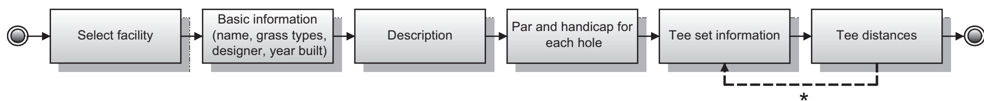

Figure 7.5 The wizard allows the user to enter a new golf course into the directory.

The textbook choice for starting conversations such as the golf course wizard is to add the `@Begin` annotation to the action 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>` page descriptor tag or a UI component tag is a more appropriate choice. The latter 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 need to begin the conversation only once, so these options are mutually exclusive. Let's start by looking at how to use the `@Begin` annotation.

AN ANNOTATION-BASED APPROACH

One of the method interceptors that Seam wraps around components is the `ConversationInterceptor`. This interceptor looks for the `@Begin` annotation on the component method that's being invoked. If it finds one, it converts the temporary conversation to long-running if the method completes successfully. See the accompanying sidebar.

> **Seam's definition of success**
>
> For many of Seam's annotations that designate an action to be performed, such as `@Begin` and `@End`, Seam only performs the action if the method completes successfully according to Seam's definition of success. For Seam to consider a method call successful, it must return without throwing an exception and it must return a non-null value unless it's a void method.

The `@Begin` annotation is outlined in table 7.3. The `pageflow` attribute is used to initiate a stateful page flow and is covered in section 7.6. The `flushMode` attribute is used to switch the flush mode of the persistence context when the conversation begins and can be used to initiate an application transaction as covered in chapter 9.

Table 7.3 The `@Begin` annotation

Name: Begin
Purpose: Instructs Seam to convert the temporary conversation to a long-running or nested state after this method is invoked successfully. See the sidebar "Seam's definition of success."
Target: `METHOD`

Attribute	Type	Function
`join`	`boolean`	A true value allows this method to be invoked even when the conversation is already long-running. If the value is false, and the conversation is long-running, an exception is thrown. If the conversation is temporary, this attribute doesn't apply. Default: false.
`nested`	`boolean`	A true value suspends the conversation, if it is long-running, and starts a new, nested long-running conversation. If the conversation is temporary, it is simply converted to long-running. This attribute is mutually exclusive with join. Default: false.
`pageflow`	`String`	The name of the page flow descriptor that is to be used to manage the stateful navigation for this conversation. Default: none.
`flushMode`	`FlushModeType`	Changes the flush mode of the Seam-managed persistence contexts in this conversation when the conversation begins. Default: `AUTO`.

Let's use the `@Begin` annotation to initiate a long-running conversation for the course wizard by placing it on the `addCourse()` method. This method is invoked at the start of the course wizard, beginning a long-running conversation. It also outjects the `course` property into the conversation context, making it available throughout the wizard:

```
@Name("courseWizard")
@Scope(ScopeType.CONVERSATION)
public class CourseWizard implements Serializable {
    @In protected EntityManager entityManager;
    @RequestParameter protected Long facilityId;
    @Out protected Course course;            <-- Outjects to conversation context

    @Begin public void addCourse() {         <-- Begins long-running conversation
        course = new Course();
        course.setFacility(entityManager.find(Facility.class, facilityId));
    }
}
```

To start the course wizard, the user navigates to the detail page of a facility, then clicks a command button that calls the `addCourse()` method and passes the id of the facility:

```
<s:button action="#{courseWizard.addCourse}" value="Add course...">
  <f:param name="facilityId" value="#{facilityHome.instance.id}"/>
</s:button>
```

The following navigation rule is defined to advance the user to the first screen in the wizard:

```
<page view-id="/Facility.xhtml">
  <navigation from-action="#{courseWizard.addCourse}">
    <render view-id="/coursewizard/basicCourseInfo.xhtml"/>
  </navigation>
</page>
```

The `@Begin` annotation isn't limited to action methods. You can also add it to a method used as a page action; combine it with a life-cycle annotation, such as `@Create`; or tie it to a `@Factory` method. These options allow the long-running conversation to start at various points in the Seam life cycle. Remember, it doesn't matter when the state of the conversation changes during the request, but rather what the state of the conversation is when the request is complete.

Instead of using a UI command button to activate the `addCourse()` method, you could trigger this method by registering it as a page action on the opening screen in the wizard:

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

In this case, when the servlet path `/coursewizard/basicCourseInfo.seam?facilityId=3` is requested in the browser, 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 waiting for a JSF postback. Starting a conversation from a page request may require additional logic either to prevent excessive conversation creation or to enable conversation joining, both of which are covered later on.

You also have the option of defining a factory method for the `course` context variable. When the `course` context variable is looked up by the first screen in the wizard through a factory, a long-running conversation can be started:

```
@Begin @Factory("course")
public void initCourse() {
    course = new Course();
    course.setFacility(entityManager.find(Facility.class, facilityId));
}
```

Factory methods are a good place to start conversations because they don't require user interaction, nor do they require XML to be defined in the page descriptor. The same goes for the `@Create` life-cycle method. You can also have the conversation begin the very first time the `CourseWizard` component is accessed:

```
@Begin @Create
public Course initCourse() {
    course = new Course();
    course.setFacility(entityManager.find(Facility.class, facilityId));
}
```

If you fancy those angled brackets, you might instead find the page descriptor to be an ideal place to begin a long-running conversation, which we look at next.

A PAGE-ORIENTED APPROACH

One way to start a long-running conversation in a page-oriented manner is by using a `@Begin` method as a page action. However, the task of starting a long-running conversation when a page is requested is so common that Seam includes two built-in options. You can either use the method-binding expression `#{conversation.begin}` in a page action or nest the `<begin-conversation>` page descriptor tag with a `<page>` node. The `<begin-conversation>` tag can also be used during a page transition.

Let's start by applying the `<begin-conversation>` tag to a page transition. We'll assume that the command button shown earlier is activated, but the action method doesn't have the `@Begin` annotation. Instead, the following navigation rule will start a long-running conversation, then direct the user to the first screen in the wizard:

```
<page view-id="/Facility.xhtml">
  <navigation from-action="#{courseWizard.addCourse}">
    <begin-conversation/>
    <redirect view-id="/coursewizard/basicCourseInfo.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/basicCourseInfo.xhtml view ID is requested. This is done either by using the `begin()` method on the built-in conversation component:

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

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

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

If a fine-grained page descriptor is associated with the first screen of the wizard, /coursewizard/basicCourseInfo.page.xml, you could exclude the `view-id` attribute, thus simplifying the declaration to

```
<page>
  <begin-conversion/>
</page>
```

The only downside of using the built-in page action alone is that you can't perform "prep work" in a component method before the first page is rendered. The `addCourse()` method, for instance, initializes and outjects the `course` context variable. The built-in page action is best suited for simply "turning on" a conversation.

The page descriptor offers a lot of control for defining conversation boundaries because you can distinguish between initial requests, postbacks, and even action method outcomes. However, you may want to be able to associate the start of 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 UI command component with conversation propagation capabilities by adding a nested `<s:conversationPropagation>` tag, or you can use one of Seam's command components, `<s:link>` or `<s:button>`, both of which have native support for conversation propagation. The propagation value, assigned using the `type` attribute on the `<s:conversationPropagation>` tag and the `propagation` attribute on the command tags, can be any of the values listed in table 7.1, and is therefore not limited to beginning a long-running conversation.

Seam reads a request parameter named `conversationPropagation` to decide how to address the current conversation. This request parameter is passed either through the query string of the URL or in the POST data. 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's 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 UI command component tag to give it conversation propagation abilities. Let's assume the facility list page includes a link in each row to start the course wizard for that facility (citing a slightly different use case here), passing the iteration variable, `_facility`, as a parameter:

```
<h:commandButton action="#{courseWizard.addCourse(_facility)}"
  value="Add course...">
  <s:conversationPropagation type="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 command tags to specify the conversation propagation using the `propagation` attribute:

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

In the event that you don't need to submit a form, the command tags are a great way to work with conversations because they have propagation controls built right in and they automatically pass along the conversation token. That adds to the previously mentioned benefits of the command tags to produce bookmarkable URLs (covered in chapter 3).

If you've been experimenting with the begin directive while reading this section, you may have encountered an error message reporting that a long-running conversation can't be created because one is already active. To remedy this problem, you need to know about conversation joining and how to enable it.

ENABLING CONVERSATION JOINING

By default, Seam only begins a long-running conversation if one isn't 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 throws an exception. That's because, by default, conversation joining is disabled.

This restriction could cause undue errors if it's possible for the user to navigate through one of the defined conversation boundaries after having already entered into a long-running conversation. For example, assume that you're 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, and validation errors are found, when JSF attempts to redisplay the page, an exception will be thrown because the `<begin-conversation>` element is once again encountered, this time in the presence of a long-running conversation.

There are numerous other situations where the user's action attempts to begin a new long-running conversation when one already exists. This situation isn't that rare in applications that use free-form navigation since the possible execution paths are numerous. Unless you have a good reason not to allow conversation joining, always use the join directive to avoid surprising your user with erroneous exceptions. The truth is, the join behavior should probably be the default. In the absence of a long-running conversation, the join directive acts just like the begin directive, without risking an exception.

If you're using the annotation-based approach to setting conversation boundaries, you enable joining of an existing long-running conversation by setting the `join` attribute on `@Begin` to true:

```
@Begin(join = true)
public void addCourse() { ... }
```

If you're using the page-oriented approach to mark the boundaries of your conversation, you add the `join` attribute to the `<begin-conversation>` element:

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

The `<begin-conversation>` element can be applied conditionally using the `if` attribute. The following declaration has the same effect as a join, made possible by consulting the conversation component to determine whether the conversation is long-running:

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

The support for conditions is useful as your pages mature, serving more variant use cases with different entrances and exits. Your only limit is what the EL can tell you.

If you're using the built-in conversation control `#{conversation.begin}` in a page action, you don't have to worry about the join flag since the method called by this expression is already "join safe."

Finally, if you're using the UI component tag approach, you enable conversation joining by setting the value of the `propagation` to `join`:

```
<s:button action="#{courseWizard.addCourse}" propagation="join"
  value="Add course...">
  <f:param name="facilityId" value="#{facilityHome.instance.id}"/>
</s:button>
```

I realize that all of these options may have been a tad overwhelming. My goal was to help you appreciate how flexible Seam is when it comes to setting the boundaries of a long-running conversation. As you might expect, there are just as many options for declaring the end of a long-running conversation. Since they follow the same patterns, learning to use them should be easy.

AUTHOR NOTE I have determined that controlling conversations using the page descriptor works best for page-oriented applications, whereas the annotations are best suited for single-page, Ajax-based applications. Despite my recommendation, there's no hard-and-fast rule.

All this work of striking up a long-running conversation is only useful if you can keep the conversation going. Let's see how the conversation is carried on to the next request.

7.3.3 Keeping the conversation going

Data stored in a long-running conversation is only available to a subsequent request if the long-running conversation holding that data is restored. The secret to restoring a conversation is to pass on its conversation id to the next request using the conversation token. Depending on the style of the request, the conversation token may be passed as a request parameter or tucked away in the JSF component tree. In either case, when Seam detects the conversation token in the request, the value is used to look up the existing long-running conversation from the session context and restore it, thus preventing a new temporary conversation from being spawned.

If passing the conversation token sounds like tedious work, there's good news. This task is handled automatically by any JSF postback or Seam UI command component.

Thus, pages don't even need to be aware of the fact that they're participating in a long-running conversation. We've sure come a long way from managing hidden form fields manually!

Let's consider the two styles of restoring the long-running conversation.

CONVERSATIONS ON POSTBACK

In the presence of a long-running conversation, Seam stores the conversation token in a page-scoped component, leveraging the state-saving feature of JSF. Any subsequent JSF postback gives Seam access to this page-scoped component and thus the conversation token. You don't have to change anything about how you define UI command components to enable this behavior:

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

Notice that there's 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. The hand-off takes place completely behind the scenes.

For JSF postbacks, passing the conversation id through the page scope is convenient, but what about non-JSF postbacks that don't carry the page scope? These requests require an alternative means of communicating the conversation token. That brings us to the conversation id parameter, which Seam uses to restore a conversation on a GET request.

RESTORING A CONVERSATION FROM A GET REQUEST

Seam can't read your thoughts, so unless you explicitly pass a conversation token to tell it which conversation to restore, a new temporary conversation is created. Earlier I told you that Seam's UI command components can be used to control the conversation. Knowing that these components are designed to issue a GET request rather than a JSF postback, it's clear that they must send along the conversation id in the URL. Thus, to restore a conversation from a GET request, you can simply use a Seam UI command component.

Let's say that, at any point in the course wizard, you want to allow users to be able to view a summary in a preview window of what they've entered so far. The following link will open the preview window and give it access to the long-running conversation used by the course wizard:

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

While I'm tempted to tell you not to worry about how this works, I trust that if you've gotten this far, you are one of those developers who wants to know (if you don't, skip on to the next section).

At the beginning of the Seam life cycle, Seam looks for the conversation id in a URL parameter. The default name for this parameter is `conversationId`. You also know that the current conversation id can be retrieved from the value expression `#{conversation.id}`. Putting these two facts together, you can manually assemble the link shown previously:

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

Alternatively, you can get a little help from JSF to build the link:

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

There's a serious flaw in the previous two links. Seam allows the name of the conversation parameter to be customized, but these links hard-code the default name, `conversationId`. The name used for the conversation token is set using the `conversationIdParameter` property on the built-in component named `manager`. You can override the name of the conversation token using this component configuration:

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

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:

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

I recommend that you stick to using Seam's UI command components unless you have a good reason not to. Then again, the conversation token is not just useful for creating links and buttons; it also lets you restore the conversation through alternate channels, such as Ajax requests and conversational web services. Remember that the conversation token is the key to the storage locker holding the conversation's working set of context variables.

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 it from another.

7.3.4 Enlisting objects in a conversation

You enlist objects in a conversation by storing them in 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?"

FINDING A CONVERSATION TO JOIN

As you've learned, there's always a conversation active during a request, whether it be temporary or long-running—but only one. A request can serve only a single conversation at a time, even though it's possible for concurrent conversations to exist in the background, as you'll discover later on. The conversation to which component instances and outjected context variables are bound is the one that's active for this request.

What you may find interesting—perhaps even surprising—is that a temporary conversation doesn't need to be converted to a long-running conversation before you

can start adding objects to it. Any variables added to the conversation while it is temporary remain part of the conversation once it transitions to long-running. The conversation state is simply an indicator that determines whether the conversation should be stored in the session (long-running) or removed (temporary) after the *Render Response* phase.

You should be able to put this knowledge together with what you've learned about component instantiation to conclude that when a conversation-scoped component is requested via its component name, an instance is created and attached to the active conversation. Table 7.4 shows how the conversation is populated as the course wizard is launched, assuming the `addCourse()` method is annotated with `@Begin`.

Table 7.4 How the conversation is populated when the course wizard is launched

Step	Description
1. User activates the JSF command button	The JSF life cycle is invoked, the `#{courseWizard.addCourse}` action is queued, and a temporary conversation is created. The *Invoke Application* phase is entered.
2. Action method is invoked	`CourseWizard` is instantiated and bound to the `courseWizard` context variable in the conversation context. The `addCourse()` method call begins. `Course` is instantiated and assigned to the protected field `course`. The `addCourse()` method call ends. `course` is outjected into the conversation context and the temporary conversation is promoted to a long-running conversation.
3. Navigation rule fires	The *Render Response* phase is entered and the first screen of the course wizard is rendered. The long-running conversation is stored in the HTTP session and the conversation id is stored in the JSF UI component tree.

When the initial request in the course wizard ends, there are two new context variables in the conversation: `courseWizard` and `course`. The `course` context variable is placed into the conversation scope after the `addCourse()` method completes as a result of outjection. The conversation scope is used since no scope is specified in the `@Out` annotation and the component is scoped to the conversation. For the duration of the wizard, the `course` context variable remains in the conversation and is progressively populated with each form submission as the wizard progresses.

CONVERSATIONS IN ISOLATION

If the user were to open a new browser tab and initiate the course wizard, the process in table 7.4 would occur in a parallel conversation (assuming, of course, that the URL requested didn't include the conversation token from the first tab). The conversation activity taking place in the second tab would happen in an isolated area of the session managing that conversation. With the course wizard under way in both tabs, the same two context variables, `courseWizard` and `course`, exist in each of the two conversations, but don't interfere with one another—two conversations, two sets of variables.

TIP You can see what context variables are stored in each conversation using the Seam debug page. Ensure that debug mode is enabled (see chapter 3) and then visit the servlet path `/debug.seam` to get the list of conversations. Click on a conversation to inspect it.

Note that the component that hosts the `@Begin` method doesn't have to be a conversation-scoped component. You could begin the conversation for the course wizard using an event-scoped component and set the scope of the outjection explicitly:

```
@Name("courseWizard")
public class CourseWizard {
    ...
    @Out(scope = ScopeType.CONVERSATION)
    protected Course course;
    ...
    @Begin public void addCourse() { ... }
}
```

In this case, only the `course` context variable is placed into the conversation after a method is invoked. The scope on the `@Out` annotation must be set to conversation to override the default of event inherited from the scope of the component.

Instead of beginning or ending a long-running conversation when a component method is invoked, you may simply want to verify that a long-running conversation exists before allowing the method to proceed. Let's see how to enforce this requirement.

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, annotate the component class or the method with the `@Conversational` annotation, summarized in table 7.5. Seam verifies that the conversation is long-running before permitting a call to the following method:

```
@Conversational public String submitBasicCourseInfo() { ... }
```

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`.

Table 7.5 The `@Conversational` annotation

Name:	Conversational
Purpose:	Specifies that this component or method is conversational and the method(s) can only be invoked within the scope of a long-running conversation
Target:	`TYPE` (class), `METHOD`

Using this annotation is superficial in most cases since there's likely more to the story than just the existence of a conversation. If you're simply trying to protect a sensitive area of the code, perhaps using this annotation makes sense. You can also enforce the presence of a long-running conversation when a view is rendered. This restriction is configured on the page node that matches the view ID being rendered:

```
<page conversation-required="true"
  no-conversation-view-id="/FacilityList.xhtml"> ... </page>
```

Once again, if a long-running conversation isn't active or has expired when the view ID is requested, the `org.jboss.seam.noConversation` event is raised. However, in this case, the user is redirected to the view ID defined in the `no-conversation-view-id` attribute. In section 7.6 you'll learn that a page flow is a much better way to enforce the existence of a conversation, whether it's at the method or view ID level.

One of the key benefits of conversations is that they can be cleaned up easily. Having learned to fear the session, you may be uneasy about letting data accumulate in the conversation. Let's first consider how to pull context variables out of a conversation and then move on to ending the conversation when the use case is finished.

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 the conversation is ongoing but there are certain conversation-scoped context variables that you no longer need, you can clear them out of the conversation in one of two ways:

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

Let's consider an example. In the course wizard, the `saveHoleData()` method outjects a temporary `TeeSet` instance to the `teeSet` context variable, which is used by the tee set form to capture information about the tee set. Only when that form is submitted with no validation errors is the `saveTeeSet()` method called, which appends the `TeeSet` to the managed `Course` instance held in the conversation. At that time, the `teeSet` context variable can be cleared from the conversation by assigning the `teeSet` property a null value. The required attribute on the `@Out` annotation is set to false to permit the value to be null:

```
@Out(required = false)
protected TeeSet teeSet;

public void submitHoleData() {
    teeSet = new TeeSet();
}

public void submitTeeSet() {
    course.getTeeSets().add(teeSet);
    teeSet = null;
}
```

Another way to clear a context variable from the conversation is to retrieve the conversation context via the Seam Context API and explicitly remove the context variable:

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

The best way to ensure that conversation-scope context variables are cleaned up is to just end the conversation. Leaving conversations active isn't dangerous like letting objects

linger in the session because Seam regularly cleans out stale conversations. However, ending a conversation may play an important role in wrapping up the use case.

7.3.5 Ending a long-running conversation

As you've learned, conversations are a managed region of the HTTP session. Thus, it's possible to terminate a conversation without destroying the entire session. 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. One use case for ending a conversation is to let the user cancel out of a form or wizard. In this case, you discard the conversation and return the user to a screen of your choice, perhaps by using a Seam UI command component:

```
<s:link view="/FacilityList.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 instead:

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

Pair this navigation rule with the following Seam UI command component:

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

If you were to instead use a UI command component, you would need to set the immediate attribute to true to prevent the form values from being processed:

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

Note that the term *end* is deceptive. Ending a conversation merely demotes it from long-running to temporary—it doesn't destroy it outright. It's terminated only *after* the view has been rendered. That means that whatever values were present in the conversation remain available in the *Render Response* phase that immediately follows the demotion.

If you want to part ways with the conversation before the next render, you set the `beforeRedirect` flag on the end conversation directive and then issue a redirect after demotion has taken place:

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

Having become a temporary conversation, it won't last through the redirect and the next page view will be using a fresh conversation. Be careful using the `beforeRedirect` flag, though, because you'll lose any JSF status messages that you added in the action method. An alternative is to use a confirmation page that displays the status messages. Navigating away from the confirmation page leaves behind the conversation that ended.

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 the use of the `@End` annotation. Let's place a command button on the last page that invokes the `save()` method:

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

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

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

The `course` context variable is available to the confirmation page since the conversation isn't ended prior to the redirect. If an exception is thrown, the conversation isn't ended at all and the previous page is redisplayed. The `@End` annotation is summarized in table 7.6.

Table 7.6 The `@End` annotation

Name: End
Purpose: Instructs Seam to convert the long-running conversation to a temporary state after this method is invoked successfully
Target: METHOD

Attribute	Type	Function
`beforeRedirect`	`boolean`	If set to true, instructs Seam to terminate the conversation prior to issuing a redirect. The default is to propagate the conversation across the redirect and terminate it once the response is complete. Default: false.

Alternatively, you may want to end the conversation using the `<end-conversation>` element in the page descriptor rather than using the `@End` annotation:

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

```
      <redirect view-id="/coursewizard/summary.xhtml"/>
    </rule>
  </navigation>
</page>
```

In this case, the conversation is maintained across the redirect. You can set the beforeRedirect attribute on the <end-conversation> element to true to have the conversation terminated before the redirect.

CONVERSATION TIMEOUTS

The other, less graceful way of terminating a conversation it to allow it to expire. The default timeout period of the conversation is stored in the built-in component named manager. The timeout is specified in milliseconds. The following component configuration overrides the default timeout of ten minutes, setting it instead to a period of one hour:

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

You can customize this value by setting the timeout attribute on the <page> node in the page descriptor. This lets you modify the timeout period per view ID, so you can give the user more time to fill out a more complex form:

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

You also have the option of assigning a timeout value for a particular conversation by calling the setTimeout() method on the built-in conversation component.

A conversation will eventually time out if the user walks away from the computer. It's also possible for the current long-running conversation to be abandoned as the result of ad hoc navigation, at which point it's also subject to timeout. Let's consider how conversations get abandoned and why it's not necessarily a bad occurrence. You'll also see how to suspend a conversation, just as a transaction might be suspended, to allow the user to perform more granular work in a nested conversation. The parent conversation is restored when the nested conversation ends.

7.4 Putting the conversation aside

So far we have talked about how to begin, restore, and end a long-running conversation, but what happens to the conversation when it's not propagated to the next request? In that case, the conversation simply sits idle in the background. You can think of it as stepping outside of the conversation. It's also possible to step out of a conversation by spinning off a nested conversation. Let's explore these two cases.

7.4.1 Abandoning a conversation

It's understandable why a conversation is abandoned when the user leaves the application. But there's also good reason to intentionally abandon a conversation. Although the stateful behavior that conversations provide for have nice benefits, sometimes the conversation just needs to be set aside to do something else. This section explores how to get away from the current long-running conversation to move on to a separate use case, regardless of whether there is intention of returning to it. Note, however, that

once a conversation is abandoned, it may eventually time out if not restored in a timely manner.

Suppose we want to give the user the option to pick up other tasks while in the midst of using the course wizard, perhaps even to start a second wizard process. When transitioning to the start of the wizard, you don't want to go into it using the existing conversation. You can break ties with that conversation by using the propagation directive `none`. For now, don't worry about how to get back to the partially complete wizard (that comes later when you study conversation switchers).

To disable propagation, you can use either the `<s:conversationPropagation>` component tag inside a UI command component or the `propagation` attribute if you're using a Seam command component tag. The `none` directive is necessary in both cases since the conversation token is added automatically by these tags. This directive prevents the conversation token from being appended, effectively warding off the conversation. The following link is used from within the wizard to start a new instance of the course wizard reusing the same facility:

```
<s:link action="#{courseWizard.addCourse}" propagation="none"
  value="Add course...">
  <f:param name="facilityId" value="#{course.facility.id}"/>
</s:link>
```

Another way to get away from the current long-running conversation is to use the `leave()` method on the conversation component as an action listener. This method can be used as an action listener thanks to the Seam EL, which makes the `ActionEvent` parameter optional. This method has the same effect as the `none` propagation directive:

```
<s:link action="#{courseWizard.addCourse}"
  actionListener="#{conversation.leave}"
  value="Add course...">
  <f:param name="facilityId" value="#{course.facility.id}"/>
</s:link>
```

Links created by `<h:outputLink>` have no awareness of the current long-running conversation, so abandoning a conversation in those cases is just a matter of leaving off the nested `<s:conversationId>` tag.

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 abandon it. However, if you're not ready to call it quits, then abandoning the conversation is the appropriate choice. Before deciding to abandon a conversation to allow the user to go off on a tangent, consider whether it's more appropriate to suspend the current long-running conversation by nesting a new long-running conversation within it.

7.4.2 Creating nested conversations

Nested conversations allow you to suspend a long-running conversation, isolating context variables within the scope of a new, self-contained conversation. A nested conversation maintains a reference to its parent conversation and can even access its context

variables. When the nested conversation ends, the parent conversation is automatically restored.

BRANCHING OFF THE MAINLINE

Nested conversations share similar semantics with child processes in an operating system. When you begin a nested conversation, you're effectively suspending the state of the current long-running conversation and starting a new long-running conversation with its boundaries. This branching process is illustrated in figure 7.6. As you can see, more than one nested (child) conversation can exist concurrently within a parent conversation. When the nested conversation is terminated, the parent conversation is popped back into place. When that happens, Seam may even redirect the user back to the page where the branch occurred, depending on the configuration. If the parent conversation is terminated, so are all of its children. Conversations can be nested to arbitrary depth, so a nested conversation can itself be a parent to another nested conversation. Seam maintains a stack of these nested conversations, which you learn about later in the section.

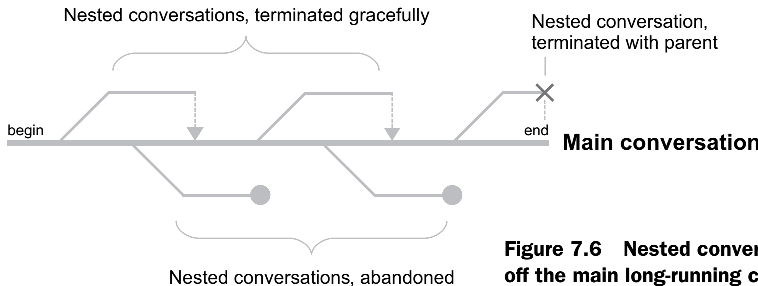

Figure 7.6 Nested conversations are branched off the main long-running conversation.

Context variables in the parent conversation are visible to the nested conversation, but the nested conversation cannot alter this set. In fact, if the nested conversation sets a context variable with the same name as one in the parent conversation, the result is that the variable in the nested conversation shadows its namesake in the parent conversation. Although it appears that the context variable has been reassigned, when the nested conversation ends, the shadowed value of the context variable is once again revealed.

NOTE Although the nested conversation cannot alter the set of context variables in the parent conversation, the objects bound to those variables are mutable.

It's time to start branching! Let's consider cases when nested conversations are useful.

WHEN TO USE NESTED CONVERSATIONS

You typically use a nested conversation when you want to allow users to maintain their proverbial spot in line while they go do something else. Keeping that spot in line means not destroying the existing conversation and perhaps sharing its position (its state).

NOTE When an isolated conversation is required, you should think hard about whether it's more appropriate to branch the current conversation (`propagation="nest"`) or abandon it (`propagation="none"`) and begin a brand-new conversation. Both have pluses and minuses.

Let's say that, while working through the course wizard, the user realizes the information about the course's facility is incorrect. While the error is fresh in the user's mind, you want the user to be able to pause the course wizard and go update the facility. As it stands now, editing a facility joins the existing conversation. When the user saves the facility, the conversation ends. However, you don't want the conversation for the course wizard to end too. Instead, you want to return users to the course wizard and let them continue as though they never left. Keeping the use cases isolated, yet linked, calls for a nested conversation.

A nested conversation is started whenever the nested propagation directive is encountered and is ended the same way as a regular conversation. To use a nested conversation for the facility editor, we need to change the FacilityEdit.page.xml descriptor so that a nested conversation is started upon entry to the page:

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

If there's no long-running conversation available at the time, the nested directive has the same effect as the begin directive. However, you can't join and nest at the same time. If the current conversation is long-running, Seam begins a nested conversation, even if the current conversation is itself a nested conversation. That means each time the user performs a postback, it's going to spawn yet another nested conversation. We prevent Seam from beginning a nested conversation if one already exists by using a conditional on the `<begin-conversation>` element in the FacilityEdit.page.xml descriptor:

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

Now, when the user clicks Save on the facility editor screen, the nested conversation is ended and the long-running conversation that manages the course wizard is restored.

WARNING Page flows, covered in section 7.6, enforce a strict navigation path. To spawn a nested conversation from a page flow, it must be configured as part of the page flow definition. Another option for initiating a parallel task is to abandon the page flow's conversation.

However, we're not quite done yet. The most critical part of letting users go off on a tangent is to return them to where they left off. If the navigation is unpredictable, users will be hesitant to branch out for fear they cannot easily get back to the current page. Let's see how to let users restore their place in line when the nested conversation is closed.

RETURNING TO THE BRANCH POINT

When a long-running conversation begins, Seam initializes a conversation stack and adds the conversation to the base of the stack. This first entry is known as the root conversation. Each time a branch occurs, Seam adds the nested conversation onto the stack. Thus, each entry added to the stack is a child of the previous. When a nested

conversation is ended, using the end propagation directive, Seam "pops" the conversation stack, restoring the previous entry—the parent conversation—into the foreground as the current conversation.

By maintaining the conversation stack, Seam relieves the burden on the developer of tracing the user's steps. But it does more than that. As part of this conversation stack, Seam keeps track of the last view ID visited by each conversation (at the branch point). This makes it possible to redirect the user back to the branch point when the nested conversation ends. However, Seam doesn't automatically perform this routing—it requires some work on your part. Fortunately, Seam offers assistance.

To get the user back on track, you can use the `endAndRedirect()` method on the conversation component (i.e., `#{conversation.endAndRedirect}`). This method ends the nested conversation and, if the view ID where the nested conversation was spawned is known, redirects the user back to that page. For instance, this method can be used as the action of the cancel button to get the user back to the current page in the course wizard:

```
<s:button action="#{conversation.endAndRedirect}" value="Cancel"
  rendered="#{conversation.nested}"/>
```

Combining this functionality with a form submission, such as the Save operation, requires calling the `endAndRedirect()` method from within the action method. This ensures that the end and redirect only occurs if the business logic completes. It also bypasses the navigation rules. For instance, you could weave this logic into the `update()` method on the `FacilityHome` component:

```
@In private Conversation conversation;

public String update() {
    String outcome = super.update();
    if (conversation.isNested()) {
        conversation.endAndRedirect();
    }
    return outcome;
}
```

This feature also comes in handy when you're developing breadcrumb-link navigation. For instance, suppose you want to allow users to spawn a nested conversation from the course page to view a related course. From there, the cycle may continue. To allow the user to back up to the previous course—without needing to use the browser back button—you can use the end-and-redirect behavior. Let's assume that the context variable `nearbyCourses` holds a list of courses in close proximity to the current course. To allow the user to navigate to one of these courses using a nested conversation, you create a link for each related course:

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

When the user navigates to a nearby course, the conversation is nested. On the course detail page, a link is provided to return the user to the previously viewed course if a nested conversation is active:

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

This nested conversation example is the first you have seen of shuffling conversations. Having suspended or idle conversations might make you nervous about the possibility of leaking memory. Although I've assured you that conversations are cleaned up when they time out, you may not be comfortable with the idea of having all these idle conversations lying around. Fortunately, Seam provides a way to allow the user to rediscover lost conversations and either return to them or end them manually. In the next section, you'll discover that leaving a long-running conversation and later switching back to it can be a natural part of how the application works.

7.5 Switching between conversations

Abandoning a conversation may sound remiss, but it can be a powerful tool. Keep in mind that your user is a person and people like to multitask. The popularity of browser tabs reflects this fact. When a conversation is abandoned, you don't have to consider it lost forever. It's just sitting behind the scenes waiting to be rediscovered, just like a background tab in a browser. Unless the abandoned conversation reaches its timeout period, it's 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*. Think of it as switching tabs in a browser. In this section, you're introduced to workspaces, how they're defined, and how you can provide a way for the user to jump between them.

7.5.1 The conversation as a workspace

A conversation is more than just a context. It also represents the user's workspace within the application. Not just any conversation can be a workspace, though. To be a workspace, a conversation must have a description, which you'll learn to assign in the next section.

If there were only one workspace (per user), there wouldn't be much need to assign it a description. We'd simply call it *the conversation*. The term *workspace* is significant because it's possible for a user to have multiple, parallel conversations. Since the browser window can only focus on one workspace at a time, the remaining workspaces exist in the background.

Workspace support is useful for two reasons. First, it allows the user to pause the current task and pick up something else with the intention of returning to the original task later. You have already seen an example of this in the nested conversation section. What you are about to learn is that the user can switch back to the original task without having to end the nested conversation. Workspace switching sanctions multitasking as a natural part of the application, rather than requiring the user to turn to browser tabs to get this feature.

Keeping conversations natural

Conversations can use a natural business key as the conversation token, supplied by an EL value expression, rather than a surrogate key generated by Seam. This configuration provides several benefits. First, since the key is derived from a business object, restoring the conversation happens automatically as a result of the user making the same selection in the UI, instead of the user needing to use a conversation switcher. This channeling minimizes the number of conversations created—another benefit. Finally, the conversation token is meaningful to both the user and the application. The trade-off is that it's no longer possible to have parallel conversations that operate on the same business object.

A natural conversation token configured for the course editor page might use this URL:

/open18/CourseEdit.seam?courseId=9

No much different, you say? Well, notice that the awkward `cid` parameter is absent. In this case, the `courseId` parameter serves as the conversation token. A natural conversation token is defined in the global page descriptor, assigned a name, and then assigned to the `<page>` node corresponding to the view ID on which it is used:

```
<conversation name="Course" parameter-name="courseId"
  parameter-value="#{courseHome.instance.id}"/>

<page view-id="/CourseEdit.xhtml" conversation="Course" ...</page>
```

When a page with a natural conversation is requested, and that page begins a long-running conversation, Seam sets the URL parameter accordingly. The only oddity with natural conversations is that you must educate JSF and Seam UI command components about the natural conversation, using the `<s:conversationName>` component tag and `conversationName` attribute, respectively. Here is an example of a JSF command button that joins into a natural conversation:

```
<h:commandButton action="#{courseHome.update}" value="Save" ...>
  <s:conversationName value="Course"/>
</h:commandButton>
```

By applying the UrlRewrite configuration described in chapter 3, you can get friendly URLs and stateful behavior at the same time, without having to worry about that pesky `cid` parameter. You can find several examples of natural conversation in the book's source code. For additional information, consult the Seam reference documentation.

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 will be inadvertently abandoned. By presenting users with a widget that lets them restore abandoned workspaces, you encourage users to finish what they started.

As you've 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 will be lost on users if you require them to specify a numeric ID

to continue a conversation. You need to provide users with a workspace switcher component that can be used to select a conversation. The options in the switcher should consist of friendly descriptions so that users can recognize the workspace and have motivation to return to it.

7.5.2 Giving conversations a description

A conversation is assigned a description, thus promoting it to a workspace, when the user navigates to a page with a description during a long-running conversation. A description is assigned to a page by populating the `<description>` element within the `<page>` node in either the Seam page descriptor (the stateless navigation model) or jPDL page flow descriptor (the stateful navigation model). When the current view ID matches that `<page>`, the value of the `<description>` element is assigned to the conversation. An example of a page with a description, defined in a Seam page descriptor, is shown here:

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

The same element is used in the stateful navigation model:

```
<pageflow-definition name="Course Wizard">
  ...
  <page name="basicCourseInfo"
      view-id="/coursewizard/basicCourseInfo.xhtml" redirect="true">
    <description>
      Course wizard (New course
      @ #{course.facility.name}): Basic information
    </description>
    ...
  </page>
</pageflow-definition>
```

Assigning a description to a conversation by way of assigning a description to a page may strike you as odd. Why not just assign a description to the conversation directly? Well, if you think about it, the state of a conversation changes over the course of its use. By describing the conversation only when it's created, the description quickly becomes outdated, failing to reflect the current state of the conversation. Conversations are shaped by their most recent page visit and the state of the system at the time the page is viewed. Therefore, it makes sense that the description is frequently updated. If the conversation is abandoned, the description reflects the last known state of the conversation and gives users an idea of where they'll be taken when the workspace is restored.

What makes the descriptions even more contextual and descriptive is that they can leverage EL value expressions. That might get you wondering when these descriptions

are evaluated. The description of a page is evaluated just prior to the page being rendered. You can see where this happens in the Seam life cycle by looking at table 3.2 in chapter 3.

By giving a conversation a description, it becomes a workspace (at least in the eyes of the user). It's one prerequisite for allowing it to appear in a conversation switcher component. The other prerequisite is that the conversation must be "switch enabled." Let's explore how a conversation is assigned this status.

ALLOWING THE SWITCH TO OCCUR

Just as the description of the conversation is updated as each page in the conversation with a description is requested, the view ID is updated as well. When a background conversation is restored using a switcher component, the conversation comes into the foreground and the user is redirected to the last view ID recorded for that conversation.

However, the view ID is only registered with the conversation if the corresponding `<page>` node supports switching, which is the default behavior. If switching is explicitly disabled, the conversation isn't made aware of the visit to the view ID:

```
<page view-id="/FacilityList.xhtml" switch="disabled">
  <description>Facility List</description>
  ...
</page>
```

The description and the view ID can be assigned to the conversation independently of one another. For instance, if switching is disabled, yet the `<page>` has a description, then the description of the conversation is still updated. Likewise, if the `<page>` supports switching, but there is no description, then only the view ID is recorded, leaving the conversation description as it was. If none of the pages the user visits support switching, then the workspace can't be restored because there's nowhere for the switcher component to redirect the user to. Thus, for a conversation to support switching, at least one view ID with switching enabled must be requested.

All that is left to enable conversation switching is to provide the user with a menu of available workspaces and a command to select one. Seam includes a handful of built-in components that aid in creating such a control.

7.5.3 *Using the built-in conversation switchers*

Workspaces are a new concept in web applications. To promote their use, Seam provides several built-in conversation switchers that you can drop into your application with little effort. Seam offers a simple select menu switcher, a more advanced table-based switcher, and a conversation stack that can be used for breadcrumb navigation. The first two components are used to switch between parallel conversations, while the latter is constrained to the ancestry of the current conversation. Let's start with the select menu.

THE BASIC CONVERSATION SWITCHER

Seam's built-in conversation switcher component, named `switcher`, is a ready-made component intended to be used with a `UISelectOne` component, such as `<h:selectOneMenu>`. This conversation switcher is a great place to start because it's

simple and unobtrusive. What's more important about it is that it helps raise awareness of the workspace construct. Users can see which workspace is currently active and get an inventory of the other active workspaces in their session, as figure 7.7 shows.

Figure 7.7 A basic conversation switcher that includes static outcomes for returning home and entering a new course.

Here is the markup that creates a switcher control that includes the list of workspaces:

```
<h:form id="switcher"> Workspace:
  <h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
    <f:selectItems value="#{switcher.selectItems}"/>
  </h:selectOneMenu>
  <h:commandButton action="#{switcher.select}" value="Switch"/>
</h:form>
```

The `#{switcher.selectItems}` value expression prepares a set of select menu items from the list of long-running conversations that support switching (i.e., workspaces). The value of the options are the conversation ids and the labels are the conversation descriptions. When the action `#{switcher.conversationIdOrOutcome}` is invoked, Seam uses the value of the selected option to locate the background conversation. The user is then redirected to the last view ID used by that conversation. When the switch occurs, the current conversation is abandoned.

TIP Notice that I used a standard UI command component to invoke the action method of the switcher component. In order for this component to work, it must submit the form so that the value selected in the `UISelectOne` component is captured. Seam UI command components would not work since they *do not* submit the form.

This component allows you to tack on your own options to the menu, which is where the `OrOutcome` part of the action method becomes relevant. If the selected value isn't numeric, the action method returns the selected value as a logical outcome and allows the JSF standard navigation rules to take effect. Let's add an outcome that returns the user to the home page and one to add a new facility:

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

You then need to define navigation rules that match the new outcomes. If the switcher is displayed on every page, the navigation rule must be global (match the view ID `*`). The navigation rule that matches the `addFacility` outcome is shown here:

```
<page view-id="*">
  <navigation from-action="#{switcher.select}">
    <rule if-outcome="addFacility">
      <redirect view-id="/FacilityEdit.xhtml"/>
    </rule>
  </navigation>
</page>
```

The auxiliary outcomes in this switcher have to be fairly rudimentary since they can't pass additional information and don't execute a dedicated action.

WARNING The only downside of switching to a background conversation is that any nonsubmitted form data on the foreground page is lost. You can work around this issue by using an Ajax component library, such as Ajax4jsf, to periodically synchronize the form values with the properties on the model.

As you can see, the basic conversation switcher component is straightforward. Although it gets the job done, it leaves a little to be desired. For one thing, it can only show the conversation description, even though a conversation entry has a lot more useful information. It doesn't let the user terminate background conversations either. 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 in the built-in Seam component named `conversationEntries`. These conversations are exported to the session-scoped context variable `conversationList` as a list of `ConversationEntry` objects, whose properties are shown in table 7.7. This list excludes all entries that aren't displayable by default. You can use the `conversationEntries` component to export a custom list.

Table 7.7 The properties on a conversation entry

Property	Type	Description
`id`	`String`	A value that identifies this conversation. The value is typically numeric, though "natural" identifiers are also supported.
`description`	`String`	The conversation description resolved from the EL value expression specified in the `<description>` nodes in the page descriptor.
`current`	`boolean`	A flag indicating whether this conversation entry is the current conversation.
`viewId`	`String`	The view ID that was rendered when this conversation was last used.
`displayable`	`boolean`	A flag indicating whether this entry is displayable, meaning it must be active and it must have a description.
`startDatetime`	`Date`	The timestamp when this conversation began.
`lastDatetime`	`Date`	The timestamp when this conversation was last restored.
`lastRequestTime`	`long`	The timestamp when this conversation was last restored.

Table 7.7 The properties on a conversation entry *(continued)*

Property	Type	Description
`timeout`	`Integer`	The timeout period that must elapse after its last use for this conversation to be automatically garbage collected.
`nested`	`boolean`	A flag indicating whether this conversation is nested.
`ended`	`boolean`	A flag indicating whether this conversation has ended.
`removeAfterRedirect`	`boolean`	A flag indicating that this conversation will be removed immediately after a redirect.

You may not want to use all of these properties in the display, but the information they provide can be useful for deciding how to render the list of conversations. In addition to these properties, each conversation entry has several built-in action methods, which are listed in table 7.8.

Table 7.8 The methods on a conversation entry that operate on the selected conversation

Action 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's necessary to select it again, if it still exists.

Armed with these properties and action methods, you're ready to construct your advanced workspace control. We'll be using a `UIData` component to present a list of workspaces, as shown in figure 7.8.

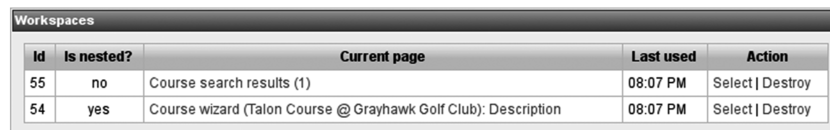

Figure 7.8 A list of workspaces in a user's session. The user can switch to one of the workspaces or destroy it.

The conversations in this table are sorted based on the last time they were used, with the most recent conversations appearing first. The JSF markup to produce this table is shown in listing 7.1.

Listing 7.1 A table-based conversation switcher component

```
<h:form id="workspaces">
  <rich:panel><f:facet name="header">Workspaces</f:facet>
    <s:span rendered="#{empty conversationList}">No workspaces</s:span>
```

```
    <rich:dataTable value="#{conversationList}" var="_entry"
      rendered="#{not empty conversationList}">
      <h:column><f:facet name="header">Id</f:facet>
        #{_entry.id}
      </h:column>
      <h:column><f:facet name="header">Is nested?</f:facet>
        #{_entry.nested ? 'yes' : 'no'}
      </h:column>
      <h:column><f:facet name="header">Description</f:facet>
        <h:commandLink action="#{_entry.select}"
          value="#{_entry.description}"/>
      </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"/> |
        <h:commandLink action="#{_entry.destroy}" value="Destroy"/>
      </h:column>
    </rich:dataTable>
  </rich:panel>
</h:form>
```

When a UI command link in one of the rows is activated, the appropriate action 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 (it is stored in the UI component tree).

The `select()` action method on the conversation entry works just like the basic switcher shown in the previous section. Seam issues a redirect to the last-used view ID in that conversation. When the `destroy()` method is invoked, Seam switches to the selected conversation and ends it. Recall that if you destroy a conversation, all of its descendants are terminated as well.

DEALING WITH DESTROY

While destroying a conversation appears straightforward, it carries with it some complexities. The `destroy()` method on the conversation entry restores the background conversation before terminating it, which means the terminated conversation remains available—along with all of its context variables—while the current view ID is rendered again. To get around this problem, you may want to add a navigation rule that terminates the conversation before redirect and issues a redirect back to the current view ID:

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

```
    </navigation>
  </page>
```

There is another, more serious problem, though. If the user destroys a background conversation while working in the context of a long-running conversation, the current long-running conversation is abandoned. The navigation rule just implemented makes matters worse since the current page is now rendered without a long-running conversation. If that page requires a long-running conversation (i.e., the `conversation-required` attribute on `<page>` is true), the user would be issued a warning and redirected to a fallback page.

I don't mean to scare you off by presenting these complications. It just leads to the following advice. Only allow background conversations to be destroyed from a page dedicated to displaying workspaces. Another way to let users destroy a workspace is to have them switch to it and then end the conversation in the normal way for that conversation, such as by clicking a cancel button. While you could develop a more intelligent destroy method than the one on the conversation entry, it's probably best to follow this advice.

TIP The table-based conversation switcher is a great way to test conversation timeouts. Open two tabs in your browser, one that you're using to test the application and one that displays the list of workspaces. You can destroy the active conversation in the workspace tab to see how the application behaves when the conversation lapses.

The conversation switcher just shown displays both top-level and nested conversations. It's also possible to create a switcher that moves solely along the ancestral chain of 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's possible to have a long chain of breadcrumbs. Seam exports a list of generational conversation entries to the session-scoped context variable `conversationStack`.

Your application must support a nested conversation model for the conversation stack to be populated with more than one entry. The example of navigating related golf courses presented earlier is a perfect use case for this component. The `conversationStack` context variable can be used in an iteration component to lay out the chain as a delimited list:

```
<h:form id="breadcrumbs" rendered="#{conversation.nested}">
  <s:span 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}"/>
    </ui:repeat>
  </s:span>
</h:form>
```

Here's an example of the output produced by this component. Each item is a link that restores the appropriate nested conversation:

```
Trail: Course search results (25) > Talon Course @ GrayHawk Golf Club >
  ➥Raptor Course @ GrayHawk Golf Club
```

The entries in the conversation stack are selected in the same way as the table-based conversation switcher. In fact, the only difference is that this list consists of hierarchical entries rather than parallel conversations.

You've now seen a couple of examples of how to use Seam's built-in conversation switchers. Once you're comfortable using them, you may decide to create more sophisticated or context-sensitive switchers to suit your needs. Recall that conversation switchers aren't the only way to control conversations. The built-in conversation component, whose properties and methods are summarized in table 7.7, can be used in both action methods and views to navigate between conversations related to the current one.

Workspaces and conversation switching offer a new, yet surprisingly refreshing, experience for the user. They can cut down on the proliferation of tabs in the user's browser because users never fear they are "losing their place" by taking a temporary detour. Instead, you can bring the power of tabs into the application.

There's one important aspect of conversations that has yet to be addressed: navigation. You probably agree that the course wizard could benefit from improved navigation control. I want to introduce you to how Seam combines conversations and page flows to provide stateful navigation. In the next section, we bolt Seam's page flow support onto the course wizard to ensure the user stays on the right track while populating the course data.

7.6 Driving the conversation with a page flow

There are two types of navigation models in Seam: stateless and stateful. Up to this point you've worked solely with the stateless navigation model. The stateless model is ideal if you don't want to enforce order on the user's actions. When the navigation's state carries meaning in the use case, as in the course wizard example, driving the conversation with a page flow is a good fit.

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. Understand that Seam is leveraging jBPM for its process definition language (jPDL) and interpreter, which together serve as a framework for building flow-based software modules. In this section you'll see how Seam uses the page flow module in jBPM. In chapter 14 (online), you'll learn to use jBPM to drive business processes.

TIP The JBossTools project includes a GUI page flow editor that can help visualize and maintain page flows like the one presented in this section.

A jPDL descriptor defines the page flow for a single conversation. The conversation and the page flow share the same life cycle. When we discuss page flows, you'll encounter references to a *process token*. During the conversation, a process token tracks

the user's place within the page flow. The process token always coincides with the currently rendered page. Navigations resulting from a user interaction are applied relative to the node at which the process token is positioned.

7.6.1 *Setting up a page flow*

The course wizard presented earlier is going to be refactored so that it's driven by a page flow named Course Wizard, following the steps in figure 7.5. The flow is defined in the jPDL descriptor courseWizard-pageflow.jpdl.xml. Rather than just dump the page flow on your lap at one time, I'm going to step through it in phases. The complete descriptor is available in the sample code at the book's website.

NOTE Page flows descriptors (*.jpdl.xml) are not hot deployable.

First things first: it is necessary to create and "install" the page flow for this example. The page flow descriptor must reside on the classpath. (For seam-gen projects, it should be placed in the resources folder.) Next, declare it in the Seam component descriptor:

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

As of Seam 2.1, files ending in .jpdl that reside on the classpath are detected by the deployment scanner and registered automatically, making this declaration unnecessary. With the page flow descriptor in place, you're ready to start populating it.

7.6.2 *Learning your way around a page flow*

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 enjoy with the other Seam descriptors. Here's the outer shell of the page flow descriptor for the course wizard:

```
<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="Course Wizard">
</pageflow-definition>
```

To put the page flow in motion, you have to create a process instance that manages it. Fortunately, Seam makes this task extremely easy.

STARTING THE FLOW

You begin a page flow using the same directive that you use to begin the conversation. Regardless of whether you're using the `@Begin` annotation, the `<begin-conversation>` page descriptor tag, or the Seam UI component tags, you specify the page flow definition

in the `pageflow` attribute. The value of this attribute is the name of the page flow, which was defined above. When the conversation begins, an instance of the page flow definition is created and the process token is advanced to the start node.

Here, the `@Begin` annotation has been augmented to initiate the Course Wizard page flow when the `addCourse()` method on the `CourseWizard` component is invoked:

```
@Begin(pageflow = "Course Wizard")
public void addCourse() {
    course = new Course();
    course.setFacility(entityManager.find(Facility.class, facilityId));
}
```

When the process instance that tracks the page flow is created, it immediately looks for a start node. There are two options: `<start-state>` or `<start-page>`. If you're starting a page flow from an action, you choose the `<start-state>` node. This approach has been chosen for the course wizard. An example of using the `<start-page>` node is shown later. The `<start-state>` of the course wizard page flow is defined as follows:

```
<start-state>
  <transition to="basicCourseInfo"/>
</start-state>
```

Let's now look at how navigation events are handled.

NAVIGATING TO A PAGE

The `<transition>` node is analogous to the `<rule>` node in the page descriptor. In this case, it's the `name` attribute that is matched against the outcome value—the return value—of the action method. If there's no outcome value, as is the case with the `addCourse()` method, the `<transition>` element without a `name` attribute is selected.

A transition implies a target. The `to` attribute specifies the name of the node to which to advance. There are four main nodes that can appear in the page flow definition after the start state. These nodes are summarized in table 7.9.

Table 7.9 The main nodes in the page flow descriptor

Node name	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 subpage flow
end-state	Terminates the process instance without ending the long-running conversation; typically used to end a subpage flow

The jPDL `<page>` node indicates which view ID should be rendered when the process token arrives. The `<page>` node is the "wait" state in the page flow process.

NOTE Don't confuse the `<page>` node from jPDL with the one used in the Seam page descriptor. They are not the same.

The following <page> node stanza renders the first screen in the course wizard, which is called on by the start state:

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

Notice the nested <transition> elements. Since <page> is a "wait" state node, it means that these transitions don't apply until an action is invoked from the view ID. You can think of them as *exit* transitions. We'll get back to those shortly.

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. The redirect also prevents the situation where the user hits the refresh button and is prompted with a confusing message about resubmitting post data. More about browser buttons in a moment.

AUTHOR NOTE The redirect functionality can also be declared using a nested <redirect/> element. I prefer the redirect attribute; I find it more intuitive since it is adjacent to the view-id attribute to which it applies.

Let's put this page flow configuration aside for now and look at how to start the flow the other way, beginning with the <start-page> node.

INITIALIZING A PAGE FLOW LAZILY

If the conversation that manages the page flow begins in the *Render Response* phase, perhaps by a factory, it isn't possible to invoke a navigation event at the start of the flow. Therefore, the start of the page flow must be declared using a <start-page> node.

Let's say that we want to start the course wizard by navigating the user directly to the first page. To support this starting point, the course context variable referenced on that page is created by a @Factory method, which also begins the long-running page flow:

```
@Out private Course course;
...
@Begin(pageflow = "Course Wizard")
@Factory("course")
public void initCourse() {
    course = new Course();
    course.setFacility(entityManager.find(Facility.class, facilityId));
}
```

In this scenario, the start of the page flow is declared using a <start-page>. The value of the view-id attribute must match the view ID of the first page in the course wizard:

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

Note that aside from the use of <start-page>, this element is equivalent to the <page> element configured in the first scenario. Now, on to the transitions.

7.6.3 Advancing the page flow

As mentioned earlier, page flow transitions work just like JSF navigation rules, chosen based on the outcome of the action method. In lieu of using an action method, the outcome can be specified as a literal value in the action attribute of the command component tag, which is the approach often used in page flows. Here are the buttons on the first page of the wizard:

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

When either button is activated, Seam locates the matching <transition> node and advances the token to the node whose name matches the value in the to attribute. In this case, the targets nodes are named description and cancel:

```
<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 begin with the cancel transition.

ENDING SO SOON?

The cancel transition advances to the <page> node named cancel. There, we see another familiar element, <end-conversation>. This element ends the conversation upon entering the <page> node. In this case, the conversation is ended prior to the redirect, which immediately follows. As a result, the conversation that served the page flow is wiped out before CourseList.xhtml is rendered. At this point, the process instance is effectively terminated (no <end-state> is needed).

The transition to holeData is unique in that it executes an action before advancing to the target node. Let's see what that is all about.

INVERTING THE CONTROL

Using an <action> node in a page flow is an inversion of the typical navigation mechanism in JSF. Rather than declaring an action method expression on a UI command component and following it with a navigation rule based on the action method's outcome, the outcome comes first and then an action method is invoked. What's nice about the inverted approach is that it abstracts the action method expression from the view. All the UI command component says is "next." The page flow descriptor takes it from there. You can use either approach.

Now it's decision time. Page flows can consult a component's state to determine which navigation path to follow, thus enabling conditional navigation.

PERFORMING LOGIC IN A TRANSITION

Although the game of golf is designed to level the playing field by giving men and ladies different par and handicap values, many courses don't make that distinction. Therefore,

when the user is presented with the form to enter the men's par and handicap data, a checkbox appears to indicate whether it's necessary to provide a different set of data for the ladies. The checkbox's state is consulted in the page flow to determine whether it's necessary to return to the holeData.xhtml page to capture the additional data:

```
<h:selectBooleanCheckbox rendered="#{gender == 'Men'}"
  value="#{courseWizard.ladiesDataUnique}" /> Unique data for ladies?
<h:commandButton action="Men" value="Next"
  rendered="#{gender == 'Men'}"/>
<h:commandButton action="Ladies" value="Next"
  rendered="#{gender == 'Ladies'}"/>
```

The decision of whether to return to the holeData.xhtml page is handled by the `decideHoleData` node. The value of the `expression` attribute on the `<decision>` node, which is a value expression, is resolved immediately upon entry and its value is used to determine where to transaction next:

```
<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 all of the data has been collected for the course, the user arrives at the review screen. The final two `<page>` nodes that wrap up the wizard 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(course.id)}"/>
  </transition>
  <transition to="review"/>
</page>

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

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

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

By putting the method-binding expression in the UI, we can leverage the transition action to set the newly established ID of the course so that the course can be displayed once the page flow is complete.

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

7.6.4 *Addressing the back button*

If you've heard it asked once, you've heard it a hundred times: "Can the back button be disabled?" Lucky are those who are so blissfully unaware. It's a stateless world and we have to learn to live in it. Fortunately, Seam addresses the back button "problem" not by disabling the back button, but by being smart enough to know what to do when it's used.

BACKING UP IN THE FLOW

If, during a page flow, users attempt 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 positioned. The same goes for when users click the refresh button and the browser attempts to resubmit the form. Of course, the refresh problem has already been solved by performing a redirect during the transition, but it's still nice to know that Seam prevents the double submit anyway.

Holding the user to the current page is the default behavior for a page flow. You may decide that it's permissible for users to back up in the flow to modify or review their work. If you want to sanction this behavior, you need to bake it into the page flow. You enable use of the 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>
```

This setting lets the user return to any page leading up to this page and step through the flow again. The only downside is that once you open this door, you have to deal with the possibility of the user executing parts of the page flow over again.

WHAT'S DONE IS DONE

The back button can be used to do more than just move around in the current conversation. Its most troubling aspect is that it allows the user to back up into an old conversation and attempt to interact with it again. Fortunately, Seam takes notice and scorns this behavior.

Let's say that the user previously posted a transaction that ends a conversation (perhaps submitting an order). If the user backs up to the form and tries to submit it again, Seam detects that the conversation has already ended and raises a warning. If a `no-conversation-view-id` is configured in the page or page flow descriptor, Seam also redirects the user to this fallback page. This check is applied to both regular conversations and conversations managed by a page flow.

That wraps up our introduction to page flows. Page flows have a wealth of additional features, including the ability to define subflows, 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. It's a lot to configure, but then again, if power is what you're after, it may be worth the trouble.

The course wizard is an example of a well-defined conversation, having explicit begin and end points and a logical progression in between. A conversation can also be combined with free-form interaction to let the user mold the state and direction of the user interface. An example of this type of conversation is presented in the next section.

7.7 Ad hoc conversations

While there are standard use cases that are modeled best using a page flow, such as a store checkout process or wizard-based form, the most popular web-based applications don't try to enforce a structure on the user. Instead, they let the user see and do everything at once. To support these nonlinear interactions, the state of the application needs to be tracked and managed. Once again we look to a conversation to handle this task. In this case, an ad hoc conversation is used, which is identified by its omission of a well-defined flow.

7.7.1 Open for business

I like to think that when an ad hoc conversation begins, it becomes open for business. Any widget on the page can offer to let the user engage in that conversation, to contribute, modify, or reduce its state. As before, this activity occurs independently of other conversations in the background or in other tabs.

A good example of an ad hoc conversation from the real world is a flight search engine. The conversation begins with a form that captures the most elemental criteria: the origin and destination cities and dates of travel. The initial search brings back a list of all matching flights. From that point, the user can tune a slew of additional criteria and watch the results change. But that's just the beginning. Other possible interactions include expanding the details about a flight, marking a flight for comparison, seeing flight trends for the current trip, or changing the currency displayed.

The conversation provides several benefits in this situation:

- Keeps track of the state of the data in the UI: selected, visible, or expanded
- Acts as a near cache to avoid database hits
- Maintains the persistence context to ensure entity instances remained managed

Although the JSF UI component tree was designed to support the first two cases, a conversation can supplement the UI component tree to give the state more longevity. The final point is covered in detail in the next two chapters.

To see these benefits in action, the comparison feature will be distilled from the flight search example and used in the golf course directory. An ad hoc conversation will host a collection of courses that the user marks. The selections are then compared side by side on a comparison page. A page action is used to begin (or join) a conversation when the /CourseList.xhtml page is requested:

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

Next, a link is added to each row in the courses table that lets the user mark the course for comparison:

```
<s:link action="#{courseComparison.mark}" value="Mark">
  <f:param name="courseId" value="#{_course.id}"/>
</s:link>
```

Although it's not shown here, you could also add a link to unmark a previously marked course. A minimal version of the component that manages the comparison, named `courseComparison`, is shown in listing 7.2.

Listing 7.2 A conversation-scoped component used for comparing courses

```
package org.open18.action;
import ...;

@Name("courseComparison")
@Scope(ScopeType.CONVERSATION)
public class CourseComparison implements Serializable {
    @In protected EntityManager entityManager;

    @RequestParameter protected Long courseId;

    @Out("readyToCompare")
    protected boolean ready = false;

    @DataModel("comparedCourses")
    protected Set<Course> courses = new HashSet<Course>();

    public void mark() {
        Course course = entityManager.find(Course.class, courseId);
        if (course == null) return;
        courses.add(course);
        ready = courses.size() >= 2;
    }
}
```

Each time a course is marked, both the `readyToCompare` and `comparedCourses` context variables are outjected to the conversation scope. Once at least two courses have been marked, the `readyToCompare` context variable will be set to true and a button can be added that takes the user to the comparison screen:

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

All that's left is to create the course comparison screen and show the courses.

7.7.2 *Show me what you've got*

The courses to be compared reside in the conversation once a course has been marked. When the user is taken to the course comparison page, it's just a matter of iterating over this collection to render the comparison.

```
<h:panelGrid columns="#{comparedCourses.rowCount + 1}">    <-- Adds extra column for labels
  <rich:panel>
    <f:facet name="header"> </f:facet>
```

```
    <div>Location:</div>
    ...
  </rich:panel>
  <c:forEach items="#{comparedCourses.wrappedData}" var="_c">
    <rich:panel>
      <f:facet name="header">#{_c.name}</f:facet>
      <div>
        #{_c.facility.city}, #{_c.facility.state}     <- Loads facility on demand
      </div>
      ...
    </rich:panel>
  </c:forEach>
</h:panelGrid>
```

The reference to facility from course is a lazy association. It can be loaded here since the persistence context is scoped to the conversation and therefore the course entities remain managed. You'll learn the importance of persistence context scoping in the next two chapters.

Since the CompareCourses.xhtml page requires that you have a conversation active, you may want to enforce this restriction in the page descriptor:

```
<page view-id="/CompareCourses.xhtml" conversation-required="true"
  no-conversation-view-id="/CourseList.xhtml"/>
```

In this section, you've learned how to use an ad hoc conversation that is capable of accumulating state until it's time for the user to act on it, such as to produce a report. This style of conversation is useful in situations where the possible interactions are numerous.

7.8 Summary

Users become frustrated when their story is forgotten, a far too frequent occurrence in call centers and web applications. If the application fails to track state, kicking the user back to the starting point as a result, the user will be ready to hang up on your application. Seam's conversation remedies this situation by propagating state held in one request to the next.

This chapter introduced the conversation as a stateful context in which context variables for a use case are stored. You learned two important things about a conversation: that it's a managed and isolated segment of the HTTP session, identified by its conversation id, and that it represents a unit of work from the perspective of the user. At times, a unit of work may span only one request, which you learned is modeled as a temporary conversation, ensuring that conversation-scoped variables are maintained until the view is rendered. To extend the unit of work across a sequence of pages, you learned that you must use a propagation directive that begins a long-running conversation. Toward the end of the chapter you learned that a long-running conversation can either be managed by a page flow descriptor or left open to be used in an ad hoc manner.

Much of the chapter was spent going over various options for switching a conversation between its three states: temporary, long-running, and nested. The options include annotations, page descriptor elements, UI component tags, and methods on

the built-in conversation component and conversation entry. The propagation directives are what set the conversation apart from other contexts covered so far in the book.

The discussion turned from singular to plural as you learned that you can have multiple conversations going at once, either sharing a nested relationship or as isolated background conversations. Seam acknowledges multitasking through the use of workspaces, providing several built-in conversation switchers that allow the user to restore previously abandoned conversations.

This chapter established a foundational knowledge of conversations, but it's really just the beginning. One of the primary uses of a conversation is to manage the persistence context. Before you can learn about Seam's pioneering work with the persistence context, you need to learn about Java persistence, which is where the next chapter picks up.

JAVA WEB DEVELOPMENT

SEAM IN ACTION

Dan Allen

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

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

What's Inside

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

About the Author

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

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

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

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

"Open this book first! An excellent resource."

—Peter Johnson
author *JBoss in Action*

"Explains, teaches, and instills confidence."

—Peter Pavlovich
Sr. Software Engineer, Kronos Inc.

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

—Michael Smolyak
Software Architect, DataSource, Inc.

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

—Kevin P. Galligan
CTO, Medical Research Forum

"Essential!"

—Ted Goddard
Chief Architect, ICEfaces.org

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

—Nikolaos Kaintantzisl
Software Engineer, Zühlke Engineering

ISBN-13: 978-1933988405
ISBN-10: 1933988401

MANNING $44.99 / Can $44.99 [INCLUDING EBOOK]