Covers Seam 2

# SEAM IN ACTION

BONUS CHAPTER

Dan Allen

FOREWORD BY Norman Richards

MANNING

***Seam in Action***

by Dan Allen

Bonus Chapter 14

# 14 Managing the business process

***This chapter covers***

- Demystifying BPM
- Initiating a business process
- Presenting actionable task lists
- Creating multiuser interactions

Business processes are all around us. They aren't confined to Fortune 500 companies or large government agencies. In the context of information technology, a business process simply describes the way that people and systems interact to accomplish a goal. A business process management (BPM) tool, such as jBPM, provides a means of modeling business processes and functions as a workflow engine to execute them.

It may astonish you to learn that creating a "friend" link, the central feature in all social networking tools, is an ideal use case for a business process. You'll even get to implement this process by the end of the chapter. In doing so, you'll discover that Seam makes it easy to adopt business processes by offering tight integration

with Seam components and the EL—the same style of integration that you have seen chapter after chapter in this book.

Many developers take one look at the term *business process* and immediately write it off as being too difficult or go so far as to discount it as being a product of SOA marketing hype. These developers steer clear of business processes and prefer the more traditional strategies to "play it safe. " Ironically, they end up performing the exact same work, albeit with far less structure and durability. You witness similar evasive behavior in golf players, who pretend to ignore the 1-iron in their bag. (If you're not familiar with golf, know that this golf club is notoriously hard to use effectively.) Instead, they play it safe by choosing more comfortable, less risky clubs. There is a famous joke by Lee Trevino, a former golf great, pertaining to this tall order:

> *If you are caught on a golf course during a storm and are afraid of being hit by lightning, hold up a 1-iron. Not even God can hit a 1-iron.*

Trevino makes a wisecrack about the general sense of fear, uncertainty, and doubt (FUD) shared by many golfers, even pros, when it comes to using the 1-iron. Fortunately for you, business processes don't require the same level of mastery as 1-irons. That doesn't eliminate the fact, though, that they're still something new to learn, which means time and effort, or even failure. That's where Seam comes into the picture. Thanks to Seam, business processes are just a combination of conversations, persistence, and security, with some wait states and user handoffs added to the mix. You should have no trouble making the leap to business processes as long as you're reasonably proficient with the first three.

This chapter focuses on BPM from the perspective of the jBPM library. jBPM is an open source process language engine that supports BPM. jBPM's reach extends beyond BPM, though. It also supports other process languages, such as Seam Page Flow, which you learned about in chapter 7, and BPEL. The most common language in jBPM is jPDL, the process language used for business processes in Seam.

The first part of this chapter will focus on bringing you up to speed with jBPM. You'll explore basic BPM terminology, see how a business process operates in jBPM, and learn how to set up the jBPM integration. The remainder of the chapter will take a hands-on approach, showing you how to craft a business process, create and operate the process declaratively with either Seam annotations or page descriptor tags, and, most importantly, presenting the user with tasks to perform the human interaction part of the process. This chapter should help you appreciate how attainable business processes are when coupled with Seam and how applicable they are to even the simplest of web applications.

## 14.1 Getting acquainted with BPM

The term *business process* may sound like a matter for upper management rather than for software developers. In particular, it gets a bad rap from the developer community for being paired with scary marketing terms like *service-oriented architecture (SOA)* and because the solutions have traditionally been so monolithic and expensive and have

led to vendor lock-in. To ease your qualms about business processes, this section will provide substantial meaning to the term, help acquaint you with BPM and the jBPM library, and give you an idea of when you might need to use a business process in a web application. I can assure you that business processes are relevant and useful for improving the quality and usability of your applications.

### 14.1.1 What is a business process?

Just as a page flow represents a fixed sequence of pages and navigation events, a business process represents a workflow of tasks. The difference is that a page flow is used to direct a single user through an individual task, whereas a business process accommodates an arbitrary number of users, and even computers, collaborating to accomplish an objective. The business process also maintains process-scoped data that is shared among the tasks but may not be relevant outside the process. While the business process is active, users participate by taking temporary ownership of the process to perform tasks, relinquishing control once they have played their part.

This isn't the first time that you have used jBPM in Seam. Seam also leverages jBPM to guide single-user page flows, which we covered in chapter 7. jBPM can be used for both purposes since it's designed to be a flexible, extensive process framework and isn't limited strictly to business processes. In fact, it's not uncommon to see jBPM used to guide a page flow for an individual task within a broader jBPM context that's executing the business process. Two different contexts, two different configurations. In this chapter, we focus on the use of jBPM to execute business processes.

As we established, a business process is a progressive sequence of tasks performed by one or more participants to accomplish a collaborative mission. For this collaboration to happen, there must be a director. That's where BPM comes in. The main purpose of a business process management (BPM) system is to facilitate the automation of these task handoffs to perpetuate a multiuser workflow. These interactions are arranged using a process language, which the BPM director uses as the score for conducting the flow of tasks.

The question that remains to be answered is, *why all the fireworks?*

### 14.1.2 The role of a business process language

The primary objective of a BPM tool, and jBPM in particular, is to serve as a framework for defining business processes using graph-oriented programming. In jBPM, processes are represented as graphs of nodes. The jBPM engine is agnostic to the language in which processes are defined, but most of the time you see processes defined using the XML-based jBPM Process Definition Language (jPDL), the main process language that ships with jBPM. Seam has settled on jPDL for defining both business process definitions and page flow definitions. Therefore, jPDL is the basis for the examples you'll see in this chapter.

The process graph always extends from a single starting point and consists of an arbitrary number of nodes and transitions leading to one or more endpoints, as illustrated

in figure 14.1. The possible paths along the way are dictated by the scenario being represented.

But this vague description of how a business process is represented doesn't provide motivation for why it's needed. The true goal of a business process is to be able to see how the code executes without having to dig into the program logic.

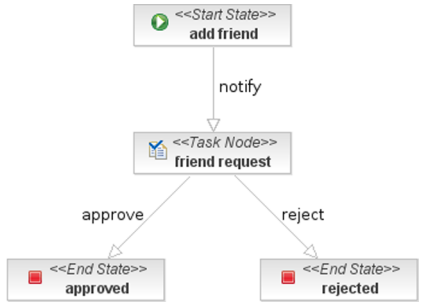


**Figure 14.1 The Add Friend business process represented visually in the process designer**

#### ESTABLISHING THE BIG PICTURE

When a business process is translated into an imperative language, such as Java, the big picture is quickly lost. This consequence is partially due to the fact that Java wasn't designed to be represented graphically. The main reason, though, is that imperative languages typically operate using a brief, linear stream of execution (i.e., a thread). You already learned in chapter 11 that imperative languages aren't well suited for the evaluation of rules; the better approach is a declarative language like Drools. The execution style of imperative languages is also at odds with business processes. Business processes are carried out in the real world where the speed of operation is many orders of magnitude slower than computer programs and where interruptions are commonplace. The result is that business processes spend more time waiting than executing.

Imperative languages (without continuations, such as Java) don't handle waiting well because they need somewhere to offload state held by the execution context when a wait is encountered. In addition negotiating a business process strictly in Java shifts the responsibility of establishing the correlation between nodes, and transferring state between isolated executions, on an external resource. It's common to see developers write this state out to a file system or database so that it's available on the next go-around. The result is a lack of clarity and no guarantee that the integrity of the process will be maintained.

What you want is a continuous execution context that sits on top of Java and can be suspended and resumed freely. That describes exactly what jBPM provides. You define the process in a process language, such as jPDL, and jBPM carries out the process. The process language, which in the case of jPDL is XML, can be viewed as a graph-based diagram and therefore gives the business analyst and the developer a common language to help improve communication between them, as illustrated in figure 14.2.

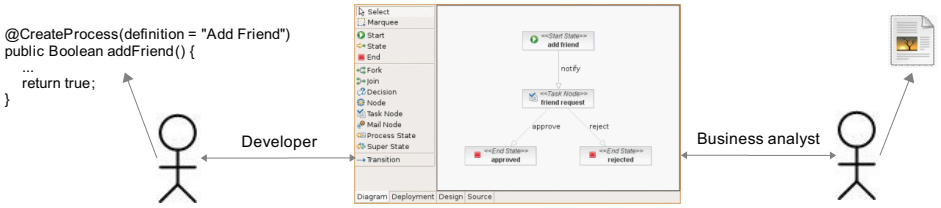


**Figure 14.2 The developer and the business analyst collaborating through a business process**

What's more important, though, is that the process definition can be consumed by the application and used to spawn a process instance for directing the short-lived executions of the imperative language. In turn, the completion of those tasks advances the execution of the graph. The big picture remains intact and you can monitor it by querying the jBPM process context.

#### A PERSISTENT FLOW

For the big picture to be maintained, the executions must be persisted during the wait states. That's what sets apart a business process from something like a servlet. At the end of a servlet request, a decision has to be made about what to keep and what to throw away. Some data is usually persisted; other data is stored in a longer-term scope such as the conversation or session. But when this happens, the current state of the flow is fragmented and will likely never be restored exactly how it was at that time.

The business process, on the other hand, doesn't live and die by the request. Nor by the conversation. Not even by a server restart. The current state—the execution of the process and all the variables within it—is persisted in its entirety into the database (or some other persistence storage mechanism). It can be continued after a couple of seconds, a couple of days, or even as long as a year or more! When it continues, it doesn't even have to be reactivated by the same participant. If there's one thing you must keep in mind, it's that business processes are very good at waiting.

As part of the execution state, it's possible to store arbitrary variables, represented as key-value pairs in the process instance. jBPM allows you to store just about any Java data type in the business process context. That list includes

- `java.lang.String`
- Primitives (`int`, `long`, etc.)
- Primitive wrappers (`java.lang.Integer`, `java.lang.Long`, etc.)
- `byte[]`
- `java.io.Serializable`
- Classes persistable with Hibernate

Most of the types in this list you'd expect to see. The entry you might find the most intriguing is the last one, classes persistable with Hibernate. jBPM uses Hibernate as its persistence storage manager, though independent from the persistence strategy that you use in your application. By using Hibernate, jBPM is able to support a wide array of databases thanks to the SQL abstraction provided by ORM. In addition, it allows you to define ORM mappings for application classes, which jBPM treats as entity classes and can therefore persist them (these can be different entities from those used in your application).

Given that business process instances can live for a long time, a risk exists that a serialized application object will become incompatible with newer versions of the underlying class. When jBPM discovers an entity instance (as defined in the jBPM Hibernate mapping configuration) in the business process context that it needs to store, it disassembles the object according to the mapping configuration and spreads

it across columns in the table rather than storing it as a serialized object in a single column. This allows jBPM to do a better and safer job of persisting application objects because disassembling the object using Hibernate mappings works purely off the state of the object and isn't contingent on deserialization. My advice is to always avoid serializing application objects and use Hibernate's data-oriented storage approach instead.

Since jBPM uses Hibernate as its persistence storage, there are some steps you'll need to follow to get Hibernate and the jBPM database configured. We'll go over the necessary configuration in the Seam jBPM integration section.

#### BUSINESS PROCESS EXECUTION

You, the developer, and the business analyst have sat down together and were able to come up with a definition of the business process in this common language. You get your XML and the business analysts get a pretty picture. But what now? You can't just throw this definition at Java and tell it to execute. Fortunately, Seam takes care of building a bridge between process definition and running code. We'll get to that shortly. But even with Seam handling the dirty work, it's important to understand the basic premise of how a process executes.

An instance of a process is created from a process definition in much the same way that a component instance is created from a component definition. The process definition describes what will happen and the process instance is the actual execution. The execution is simply a token pointing to one of the nodes in the definition. When the process is created, the token is placed at the start node (i.e., `<start-state>`). The token advances each time it's signaled, which can happen automatically, as the result of a task completing or in response to an external event. If the process definition is the highway, then the token is the car driving on it.

But thanks to Seam, you never have to deal with traffic by interacting with jBPM directly. Instead, you create an instance of a business process and advance the process execution using Seam components and a handful of annotations or page descriptor tags. In the next section, we study the unique approach that Seam takes to marrying business processes with its contextual container. After that, we move on to the steps involved in setting up the Seam BPM integration so that you're ready to start incorporating business process definitions into your Seam application.

### 14.1.3 Seam's declarative approach to BPM

BPM integration is what makes Seam stand out from just about all of the other web-oriented frameworks available today. The Seam development team recognized that single-user interactions only account for a portion of the use cases in enterprise applications. Multiuser processes are just as relevant and have a much greater impact on the application's effectiveness. That's why Seam does far more than just provide a jBPM module to nudge along the tasks in a business process. Seam folds business processes into the Seam container by relying on these key coordinators:

- *The business process context*—Seam assimilates the stateful context bound to the process instance as one of its own. Business process–scoped context variables are exchanged with the jBPM process context transparently, but are otherwise treated just like any other context variable. They can participate in bijection and can be resolved using EL notation. Value expressions can also be used in the process definition descriptor to resolve context variables from the Seam container.
- *Declarative business process controls*—A set of annotations and page descriptor tags are included that can initiate or resume a process instance, start and end tasks, and bring variables from an active process into scope, all without having to interact directly with jBPM. Seam also includes a set of UI controls for selecting the process or task instance on which to operate.
- *Identity management*—Similar to the `identity` component, Seam provides a basic component named `actor` for registering the actor id and group ids that will be passed on to the process instance to record the current participant. The `actor` component is typically populated during the authentication routine.
- *User code interceptor*—Seam traps executions within a business process instance and sets up the Seam contexts so that the executions have access to the Seam container.

The main reason business processes are so attainable in Seam is because you can use them without ever having to make your code interface with jBPM directly. You don't have to worry about how definitions are loaded, how processes are kicked off, or how signals are sent to the process instance to advance tasks. Seam handles all of those low-level details. Your application uses metadata to declare where, and under what conditions, you want these operations to occur.

But it's more than just an exercise in indirection. The important part is that there's no change necessary in the programming model in order to incorporate a business process. You have to perform some extra setup, but after that everything falls into step with what you've learned about Seam up to this point.

Seam components, or page descriptor tags, can control the progress of a business process just as if it were a long-running conversation. During the transitions, context variables move fluently from a web context into the business process context and back out again. The pattern of interaction used for business processes parallels that of conversations so closely that you'll likely experience a sense of déjà vu the first time you see a business process implemented in Seam. A Seam application that uses a business process is not fundamentally different from any other Seam application. You still have components and those components still have action methods that trigger bijection, start and stop conversations, perform work in transactions, and activate navigation.

With all of that lead-in, I'm sure you're ready to start exploring using business processes in Seam. Let's put the configuration in place and get started!

## 14.2 Setting up the jBPM integration

The process of setting up the jBPM integration is a bit more involved than many of the other integrations that Seam offers. Given that, the steps required to get jBPM operational are still quite reasonable by design.

### 14.2.1 jBPM a la carte

First, you must make the jBPM components available on your application's classpath. Seam relies on version 3.2.2 of the jBPM core library and the jBPM runtime engine, both of which are bundled in the jbpm-jpdl.jar distribution. You're also required to have the Hibernate 3 libraries on the classpath to support the persistence of process instances and variables. As you've come to expect, projects created using seam-gen already have these libraries in place. If you didn't start with seam-gen, you'll need to add them to your classpath.

The next step is to prepare the jBPM configuration to be compatible with Seam. jBPM is configured by including the jbpm.cfg.xml configuration file at the root of the classpath (e.g., the resources directory in seam-gen projects). If this file isn't found, the default configuration that's bundled with jBPM, jbpm-jpdl.xml, will be used. Out of the box, jBPM supports persisting executions to the database using Hibernate. However, since jBPM is designed to be used in a stand-alone environment, it also uses its own resource-local transactions when performing persistence operations. You don't want jBPM off in its own little world; instead, you want jBPM to participate in the global transaction managed by Seam. In fact, Seam forbids jBPM from using its own transactions.

To disable transaction management on the persistence service so that jBPM participates in Seam-managed transactions, populate the jbpm.cfg.xml as follows (this code is copied from the file src/test/integration/resources/jbpm.cfg.xml in the Seam distribution):

```
<jbpm-configuration>
  <jbpm-context>
    <service name="persistence">
      <factory>
        <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
          <field name="isTransactionEnabled"><false/></field>
        </bean>
      </factory>
    </service>
    <service name="tx"
      factory="org.jbpm.tx.TxServiceFactory"/>
    <service name="message"
      factory="org.jbpm.msg.db.DbMessageServiceFactory"/>
    <service name="scheduler"
      factory="org.jbpm.scheduler.db.DbSchedulerServiceFactory"/>
    <service name="logging"
      factory="org.jbpm.logging.db.DbLoggingServiceFactory"/>
    <service name="authentication"
      factory="org.jbpm.security.authentication.
```

```
    ➥DefaultAuthenticationServiceFactory"/>
    </jbpm-context>
  </jbpm-configuration>
```

As mentioned earlier, jBPM uses Hibernate for persistence. If you're already using the native Hibernate API in your application, you can simply append the jBPM mappings to that configuration. However, I strongly recommend that you set up a dedicated database, and thus a separate persistence unit, for storing the jBPM process definitions and executions. jBPM uses a lot of tables and you probably don't want them being mixed up with your schema. This decision affects how you set up your data sources. If you use a separate database, you'll need to make your data sources XA resources to enable two-phase commit in the transaction, which I present in the next section. You'll also learn how to get Hibernate to create the jBPM schema for you automatically.

### 14.2.2 Configuring the jBPM persistence unit

Let's first get the Hibernate configuration in place. The Hibernate configuration is very environment specific. By default, jBPM will look for hibernate.cfg.xml at the root of the classpath. This location can be overridden by specifying the `resource.hibernate.cfg.xml` property in the jBPM configuration. Another option is to direct the `DbPersistenceServiceFactory` to look in JNDI for the Hibernate session factory. You can register the Hibernate session factory with JNDI using the application server console or by setting up a Seam-managed session factory. To keep things simple, let's stick to the stand-alone Hibernate configuration.

For clarity, we use a custom name for the Hibernate configuration file loaded by jBPM. The jbpm.cfg.xml has been modified to include the `resource.hibernate.cfg.xml` property, which dictates the resource path of the Hibernate configuration file:

```
<jbpm-configuration>
  ...
  <string name="resource.hibernate.cfg.xml"
    value="jbpm-hibernate.cfg.xml"/>
</jbpm-configuration>
```

Create the jbpm-hibernate.cfg.xml file and place it at the root of the classpath. You'll populate this file with the data source and transaction settings as well as the mappings to the jBPM tables. An abbreviated version of the configuration is shown in listing 14.1. You can get the full listing from the book's source code, from the jBPM distribution, or from the file src/test/integration/resources/hibernate.cfg.xml in the Seam distribution. I've made some customizations to the contents of this file, which are described in a moment. Notice also that the property names don't have the optional `hibernate.` prefix that you're used to seeing.

**Listing 14.1 The Hibernate configuration for jBPM**

```
<hibernate-configuration>
  <session-factory>
```

```
    <property name="connection.datasource">          ❶
      open18JbpmDatasource
    </property>
    <property name="dialect">          ❷
      org.hibernate.dialect.H2Dialect
    </property>
    <property name="transaction.factory_class">          ❸
      org.hibernate.transaction.JTATransactionFactory
    </property>
    <property name="transaction.manager_lookup_class">          ❹
      org.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <property name="hbm2ddl.auto">update</property>          ❺
    <mapping resource="..."/>
    ...
  </session-factory>
</hibernate-configuration>
```

The Hibernate configuration is central to getting jBPM to participate in Seam-managed transactions. We set up a dedicated database to hold the jBPM tables with its own data source ❶. This data source is configured in the next step. The Hibernate dialect is autodetected for most databases, but must be specified for H2 ❷ until Hibernate 3.2.6. Since your application now has two data sources, the use of JTA transactions ❸ (and also XA data sources) is imperative. In order for Hibernate to locate the JTA transaction, the lookup strategy for the deployment environment ❹ must be specified. This configuration assumes the use of JBoss AS, but you can adjust it as needed if you're deploying to a different application server. To avoid having to create the jBPM tables yourself, Hibernate is configured to create or update them ❺ when the application starts. After deploying the first time, you can change the `hbm2ddl.auto` setting to `none` to ensure a quicker start. Next, let's take a look at the data source configuration for the jBPM database and ensure that both data sources can participate in an XA transaction.

### 14.2.3 Making the move to XA

The final step of the persistence configuration is to add a dedicated XA data source for jBPM and convert the existing application data source to XA as well. An XA data source is a data source that is capable of participating in an XA transaction,[1] a distributed transaction that spans multiple resources—either one or more databases or other resources, like JMS—and is coordinated by a JTA transaction manager. In an XA transaction, all resources are committed or roll back together by the transaction manager using a process known as a two-phase commit. A non-XA (local) transaction, used up to this point, only involves a single resource and therefore the resource is capable of handling the transaction itself (one-phase commit).

The configuration for XA data sources presented here is specific to JBoss AS and the H2 database. You'll need to follow the necessary steps to set up XA resources in the

[1] Read why XA transactions are needed more often than you think: http://weblogic.sys-con.com/node/44439.

JNDI for alternate environments. An indicator that your XA data sources aren't configured properly on JBoss AS is the appearance of the following error log message:

```
Adding multiple last resources is disallowed.
```

Modify your data source configuration file (e.g., files with the -ds.xml suffix) to use `<xa-datasource>` nodes for the two data sources. The complete contents of this file after the change is made are included in listing 14.2. In a seam-gen project, one of these files exists for each environment (i.e., dev or prod).

**Listing 14.2 XA data source definitions for the Open 18 application**

```
<datasources>
  <xa-datasource>
    <jndi-name>open18Datasource</jndi-name>
    <use-java-context>false</use-java-context>
    <track-connection-by-tx>true</track-connection-by-tx>
    <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    <xa-datasource-property name="URL">
      jdbc:h2:file:/home/twoputt/databases/open18-db/h2
    </xa-datasource-property>
    <user-name>open18</user-name>
    <password>tiger</password>
  </xa-datasource>

  <xa-datasource>
    <jndi-name>open18JbpmDatasource</jndi-name>
    <use-java-context>false</use-java-context>
    <track-connection-by-tx>true</track-connection-by-tx>
    <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    <xa-datasource-property name="URL">
      jdbc:h2:file:/home/twoputt/databases/open18-jbpm-db/h2
    </xa-datasource-property>
    <user-name>sa</user-name>
    <password></password>
  </xa-datasource>
</datasources>
```

jBPM is now ready to persist process definitions and executions using Hibernate. Although you can steer clear of using XA data sources by using the same persistence unit for both jBPM and your application, I find that XA data sources aren't that difficult to configure, and they save a lot of headaches when you're dealing with multiple transactional technologies.

### 14.2.4 Activating the jBPM Seam component

All that's left is to enable the jBPM component in Seam and load the process definitions. In the component descriptor, ensure that the component namespace `http://jboss.com/products/seam/bpm`, aliased as `bpm`, is declared and then activate the jBPM component by appending to the `<bpm:jbpm>` element as follows:

```
<bpm:jbpm>
  ...
```

```
  <bpm:process-definitions/>
</bpm:jbpm>
```

You're now ready to start developing business process flows and incorporating them with your components!

## 14.3 Bringing process to your application

We begin with a "Hello World" business process example to show how to control a process using Seam components and to introduce additional BPM concepts. But the most important part of this lesson is identifying when you need a business process, so let's consider this question before jumping into the development of the process.

The existence of the following two needs justifies the use of a business process:

- The process involves more than one participant.
- Data needs to be exchanged but doesn't have to be held permanently.

You could take a data-centric approach to satisfy these two needs, in which the state machine (i.e., the process instance) is implied by the state of records in the application's main database. However, what works against this solution is change. The business process may alter over time and the schema of the exchanged data may change. Using the BPM console (jbpm-console.war, which is included in the jBPM distribution) or the jBPM API (see the dvdstore example in the Seam distribution), you can upgrade the business process at runtime, thus avoiding the need to redeploy your application or incur the expensive cost of modifying your application's database schema. Processes are automatically or explicitly versioned and the system can run existing processes using the old version and new processes with the upgraded definition. As long as all your UI and related components can remain consistent, you can cleanly update your application. As mentioned earlier, you also avoid extending your core database schema since jBPM takes care of persisting the process data in designated tables (and perhaps a separate database).

To give you an easy start, we now create a variation on the classic "Hello World," a courier service. In this process, a user sends a message to another user, who then confirms receipt of the message. We begin by defining the business process in a process language.

### 14.3.1 Defining the business process

Every business process must have a name. We name our messaging business process *Courier*. The Courier business process involves two actors and one task. An instance of the process begins when one actor sends a message to another actor. The process has one task, which is assigned to the recipient. The purpose of this task is to acknowledge receipt of the message, effectively dismissing it. When the task is complete, the business process ends and the message is removed from the receiver's inbox.

The Courier business process definition, defined in a file named courier.jpdl.xml at the root of the classpath, is shown in listing 14.3.

**Listing 14.3 The Courier business process definition**

```
<process-definition xmlns="urn:jbpm.org:jpdl-3.2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
  urn:jbpm.org:jpdl-3.2
  http://docs.jboss.org/jbpm/xsd/jpdl-3.2.xsd"
 name="Courier">

 <start-state name="create">
  <transition name="send" to="inbox"/>
 </start-state>

 <task-node name="inbox">
  <task name="receive" description="#{message.content}">
   <assignment actor-id="#{message.recipient}"/>
  </task>
  <transition name="acknowledge" to="acknowledged"/>
 </task-node>

 <end-state name="acknowledged"/>

</process-definition>
```

The same process can be represented graphically using the jPDL process designer, as shown in figure 14.3. The jPDL process designer is an Eclipse plug-in developed as part of the jPDL project. It is bundled with both JBossTools and JBoss Developer Studio (JBDS).

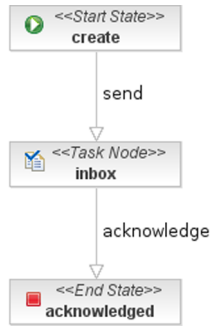


**Figure 14.3 The Courier business process represented visually in the process designer**

You then add this process definition to the `<bpm:jbpm>` component in the component descriptor:

```
<bpm:jbpm>
 <bpm:process-definitions>
  <value>courier.jpdl.xml</value>
 </bpm:process-definition>
</bpm:jbpm>
```

The Courier process will now be read when Seam initializes and loaded into the jBPM database.

We have to address two concepts in the Courier process: tasks and actors. Tasks represent work to be done by a user. Tasks are typically embedded in a task node, which contains one or more tasks to be performed. The process execution waits for the tasks in the task node to be completed, at which time the process execution continues. Actors are the users who perform the tasks and ultimately advance the process token.

When the process execution arrives at a task, an instance of the task is created. The task can be assigned to a particular actor or an actor group. In the latter case, all the actors in the group become candidates for performing the task. One of the actors in the group must step forward to take ownership of the task in order to complete it. We look at both models of assignment in the example. But first, we need to take a closer look at actors and actor groups and how they relate to users.

### 14.3.2 How to become an actor

Sorry, I can't tell you how to make it in Hollywood. But I can explain how to become an actor in the eyes of jBPM. An actor is anyone who participates in a business process, although you can't act until you're known to jBPM. Your ticket to fame is Seam. Seam, acting as your agent, negotiates a contract with jBPM to give you a unique actor id (e.g., a calling card), thus making you a candidate for gigs (e.g., performing tasks).

That leaves one question. How does Seam know what type of actor you want to be? You have to tell it. In chapter 11, you saw how Seam establishes a user's identity. Seam manages the JAAS subject for the current user in the session-scoped `identity` component, which is set up during the authentication routine. In the same way, Seam manages the jBPM actor for the current user in the session-scoped `actor` component, which is also set up during that routine.

**NOTE** Seam's `actor` component is not coupled to jBPM or the `identity` component. It's a simple POJO that holds an id and a collection of groups. However, since Seam only supports jBPM for business process management, it's safe to say that we're dealing with a jBPM actor. And while I show you how to assign an actor id and actor groups in a Seam authentication method, this setup doesn't have to be performed through Seam authentication.

There's typically a one-to-one mapping between a JAAS subject and a jBPM actor, as shown in figure 14.4. An actor has an id, which is equivalent to the user's login name, and a set of groups, which are equivalent to the user's roles. There's no hard-and-fast rule that says these two identities have to align in this way, but it's usually the most convenient approach. It's important that the actor id be unique among all participants and that it remain consistent throughout the lifetime of the business process.

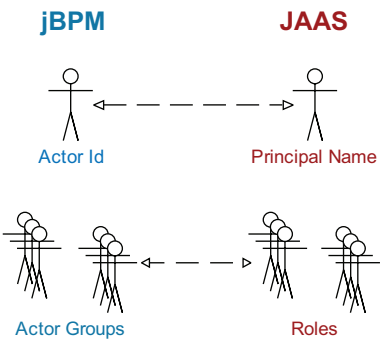


**Figure 14.4 The one-to-one mapping between a jBPM actor and a JAAS security principal**

Although a correlation exists between the two identities, you still have to tell Seam how to predicate it. That happens in the authentication method. At the same time that the security principals are being established, you also populate the `id` and `groupActorIds` properties of the actor component. Listing 14.4 modifies the authentication component presented in chapter 11 to include the setup of the jBPM actor.

**Listing 14.4 An authentication component that sets up a jBPM actor**

```
package org.open18.action;
import org.jboss.seam.bpm.Actor;
import ...;

@Name("authenticationManager")
public class AuthenticationManager {
```

```
@In private EntityManager entityManager;
@In private Identity identity;
@In private Actor actor;
@In private PasswordManager passwordManager;
@Out(required = false) private Golfer currentGolfer;

@Transactional public boolean authenticate() {
   try {
      Member member = (Member) entityManager.createQuery(
         "select m from Member m where m.username = :username")
         .setParameter("username", identity.getUsername())
         .getSingleResult();

      if (!validatePassword(identity.getPassword(), member)) {
         return false;
      }

      actor.setId(identity.getUsername());

      identity.addRole("member");
      actor.getGroupActorIds().add("actor");
      if (member.getRoles() != null) {
         for (Role role : member.getRoles()) {
            identity.addRole(role.getName());
            actor.getGroupActorIds()
               .add(role.getName());
         }
      }

      if (member instanceof Golfer) {
         currentGolfer = (Golfer) member;
         identity.addRole("golfer");
      }
      return true;
   } catch (NoResultException e) {
      return false;
   }
}

public boolean validatePassword(String password, Member m) {
   return passwordManager.hash(password).equals(m.getPasswordHash());
}
}
```

**Injects jBPM actor component**

**Sets actor id to username**

**Assigns default group actor id**

**Adds group actor for each role**

The jBPM actor representing the current user is now established. Next let's see how tasks are assigned to an actor or a group of actors.

### 14.3.3 Assigning tasks

If you look back at listing 14.3, you'll notice the `<assignment>` node nested within the `<task>` node in the process. The `<assignment>` node is used to assign a task to a user (i.e., an actor), in effect turning the process over to that user. Unless you're writing custom code to control the process, you always want to include a task assignment. Otherwise, no one can act on the tasks to move the process forward.

To assign a task to a particular actor, you use the `actor-id` attribute. If you want to assign the task to an actor group, you use the `pooled-actors` attribute. The `actor-id` attribute trumps the `pooled-actors` attribute.

You have a choice of using a literal value or an expression for either attribute. The downside to a literal value is that it essentially hardcodes the assignment into the process definition. To assign a task to an actor using a literal value, you just specify the actor's id:

```
<assignment actor-id="twoputt"/>
```

To make the assignment contextual, it's better to use an EL value expression. By using an expression, you're able to resolve any context variable in the Seam container that's in scope at the time of the assignment. A common case is to resolve a task to the current actor:

```
<assignment actor-id="#{actor.id}"/>
```

If you want to hand the task off to another user, which is the case with the Courier process, you just need to provide a value expression that resolves to the other user's actor id.

Group assignments differ in that they can accept multiple values. When using a literal expression, you must enter the group actor ids as a comma-delimited list. Here, the task is assigned to the pool of administrators and private facility owners:

```
<assignment actor-id="admin,privateFacilityOwner"/>
```

A value expression affords more flexibility. jBPM will accept a value expression that resolves to an array of strings, a collection of strings, or a comma-delimited list of values. The sole restriction with using an array or collection is that it must only contain string values.

Tasks are an opportunity for humans to interact in the business process. To help the human know what the task is, it's helpful to provide a description. The description is specified in the `description` attribute of the `<task>` node. The description can also be a literal value or a value expression.

In the Courier process, the description is equivalent to the content of the message. This choice is one of convenience. The description isn't the only way to store information in a task. Additional information can be stored in jBPM process variables. You'll see later that the business process context will be used to store the username of the message sender.

Task assignments, and the description associated with a task, will be important later when we look at task lists. Before we get there, though, we need to figure out how to start the business process and create task instances to be assigned.

### 14.3.4 Driving the business process

I'll agree with you if you say that thus far, this just looks like a lot of hand waving. That's one of the main things that turns off developers from using BPM. It can be disconcerting to anyone used to working in imperative languages. On its own, the business process doesn't appear to do anything. You can't just take the jPDL XML file and execute it as you can a typical "Hello World" example—at least, not yet. That's where Seam comes into the picture.

Seam is capable of driving the business process. This interaction is declared using annotations or page descriptor nodes. A business process instance is created when a `@CreateProcess` component method is invoked or a `<create-process>` page descriptor node is encountered during a navigation event. As is true of all Seam annotations that declare an action for Seam to conduct following the execution of a method, the action is only carried out if the method returns successfully according to Seam's definition of success (see section 7.3.2 of chapter 7).

A summary of the `@CreateProcess` annotation is provided in table 14.1.

**Table 14.1 The `@CreateProcess` annotation**

| | |
|---|---|
| **Name:** | CreateProcess |
| **Purpose:** | Creates an instance of the process definition with the name provided in the `definition` attribute |
| **Target:** | `METHOD` |

| Attribute | Type | Function |
|---|---|---|
| `definition` | `String` | The name of the business process definition. Default: none (required). |
| `processKey` | `String` (EL) | An EL expression that's used to assign a business key to this process instance to make it easier to locate for resuming the process. Default: empty string. |

The Courier process works best if activated in response to the user clicking the Send button. Therefore, we use the `@CreateProcess` annotation to put the process in motion.

You must provide a definition along with the `@CreateProcess`—or `<create-process>` node—to tell Seam which business process to create. The definition corresponds to the value of the `name` attribute on the `<process-definition>` node in the jPDL file. Here is the signature of the method we use to initiate the Courier process:

```
@CreateProcess(definition = "Courier")
public void send() {}
```

jBPM uses an internal token to keep track of the progression of the process instance. The token is always placed at the `<start-state>` node when the process is created. After creating an instance of a process, Seam signals the process instance to advance to the default transition defined by this initial node. The default transition is the first `<transition>` within a process node (`<start-state>`, `<task-node>`, etc.). In the Courier process, the default transition on the `<start-state>`, named *send*, advances the process token to the `<task-node>` named *inbox*. A task node represents one or more tasks that are to be performed by humans. So when execution arrives in a task node, task instances will be created in the task lists of the actors. After that, the node will behave as a wait state. In the Courier process, the inbox task node immediately assigns the task of receiving the message to the recipient actor. Once the recipient

acknowledges the message, the task completion will trigger, thus resuming the execution of the process.

Ah, now you're sensing some motion! But we picked up a little too much speed. Where is the message coming from and how is the process able to refer to it? Let's take a step back and look at where the `send()` method is called.

### 14.3.5 Initiating the handoff

Sending a message implies that it's first created. Thus, the real starting point of the process is the drafting of the message. The message will be stored in an instance of the `Message` class. The `Message` class has two fields, `content` and `recipient`, which are used to provide the task description and the actor id for the task assignment, respectively. The `Message` class has been declared as a Seam component for convenience. The `Message` class is shown here:

```
@Name("message")
public class Message {
   private String recipient;
   private String content;

   public String getRecipient() { return recipient; }

   public void setRecipient(String recipient) {
      this.recipient = recipient;
   }

   public String getContent() { return content; }

   public void setContent(String content) {
      this.content = content;
   }
}
```

The `message` component will be bound to a JSF form to capture the recipient and content of the message, shown next. The user must be authenticated to send a message in order to track the identity of the sender.

```
<rich:panel rendered="#{identity.loggedIn}">
  <f:facet name="header">Send message</f:facet>
  <h:form id="sendMessage">
    <h:panelGrid columns="2">
      <h:outputLabel for="recipient" value="To:"/>
      <h:selectOneMenu id="recipient" value="#{message.recipient}"
        required="true">
        <s:selectItems var="_actorId" value="#{availableActorIds}"
          label="#{_actorId}" noSelectionLabel="-- Select --"/>
      </h:selectOneMenu>
      <h:outputLabel for="content" value="Message:"/>
      <h:inputTextarea id="content" value="#{message.content}"
        required="true"/>
    </h:panelGrid>
    <h:commandButton action="#{courier.send}" value="Send message"/>
  </h:form>
</rich:panel>
```

The rendered output of this form is shown in figure 14.5.

For convenience, I use an `EntityQuery` from the Seam Application Framework to provide a list of available actors excluding the current actor. The result of the query is available under the context variable `availableActorIds`. The query and the result alias are defined in the component descriptor as follows:

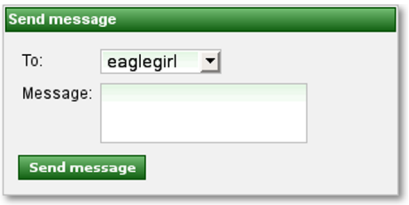


**Figure 14.5 The form for sending a message to a fellow member using the Courier process**

```
<framework:entity-query name="availableActorIdsQuery">
  <framework:ejbql>
    select m.username from Member m
    where m.username != #{identity.username}
  </framework:ejbql>
  <framework:order>m.username asc</framework:order>
</framework:entity-query>
<factory name="availableActorIds"
  value="#{availableActorIdsQuery.resultList}"/>
```

After the message is drafted and the recipient selected, the user clicks the Send message button, activating the `send()` method of the `Courier` component through the method expression `#{courier.send}`. The signature of the `send()` method on the `courier` component was shown earlier. The full implementation of the `courier` component, as it pertains to sending the message, is unveiled in listing 14.5.

**Listing 14.5 The component that starts the Courier business process**

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

@Name("courier")
public class Courier {
   @In protected Identity identity;
   @In protected FacesMessages facesMessages;

   @Out(scope = ScopeType.BUSINESS_PROCESS,
     required = false)
   protected String sender;

   @CreateProcess(definition = "Courier")
   public void send() {
      sender = identity.getUsername();
      facesMessages.add(
         "Your message has been sent to #{message.recipient}.");
   }
}
```

**Puts sender into process context**

**Creates instance of Courier process**

You may be wondering where the `message` component went. Well, when the form is submitted, the `message` instance is put into the event scope. When the business process is created, it immediately transitions to the task node and an instance of the receive task is created. Since the creation of the task happens within the context of the

JSF life cycle, the jBPM context can "see" the `message` context variable and use its properties to set up the task. This integration is made possible because Seam registers a jBPM-aware EL resolver.

Be aware that the `message` context variable isn't stored in the business process, even though it's referenced in the process definition. In contrast, the `sender` context variable is stored in the business process. That's because it needs to be available when the task is displayed to the user. Technically, the content of the message could also be stored in the business process context, but in this case, the description of the task is a more convenient place to store it.

At this point, an instance of the Courier process is active and waiting. It's waiting for the recipient to act on it. But for that to happen, recipients have to know that they have something to do. That brings us to task lists.

### 14.3.6 Tasks for homework

There are plenty of ways to occupy our time. To stay on task, we often must be given a specific list of things to do. That's the purpose of tasks lists. These BPM-style to-do lists are a way of providing users with a list of tasks that are on their plate so they can act on them.

Seam keys off the current actor id to set up task lists from the jBPM context. Seam can't look up tasks for anonymous users, so the prerequisite for using these tasks is that the user is authenticated—or at least the `#{actor.id}` expression resolves to a non-null value.

Seam's task lists are what make this integration shine because this would be boilerplate code that you'd have to write in any BPM application. Seam prepares several task list views filtered either by the current actor or the current actor's groups, which are listed in table 14.2. Each task list holds a collection of jBPM `TaskInstance` objects, which is how jBPM represents an active task. These task lists are used to present the user with a list of tasks that require action on the user's part. You can create additional task lists if the built-in ones don't suit your needs. These components merely reach under the covers and query the jBPM database using Hibernate to find task lists assigned to the current user.

**Table 14.2 The built-in jBPM task list components**

| Component name | Description |
|---|---|
| `taskInstanceList` | A collection of tasks assigned to the current actor |
| `taskInstancePriorityList` | A collection of tasks assigned to the current actor, ordered by ascending priority |
| `taskInstanceListForType` | A map of task lists keyed by the name of the task |
| `pooledTaskInstanceList` | A collection of tasks assigned to any actor groups to which the current actor belongs |

There are two factors to keep in mind when using the built-in task lists shown in table 14.2. First, the task lists aren't stored in a Seam context, so every time a task list is referenced, Seam queries the jBPM database.[2] As you learned in chapter 6, you can bind a result to a context variable using a factory. Second, with the exception of the `taskInstanceListForType`, the task lists return tasks at any stage in the process across all process instances. Even the `taskInstanceListForType`, despite its extra bit of granularity, can't distinguish between two tasks with the same name in two different process definitions. If you're only working with a single process with one task, as in the Courier process example, none of this matters. But as you start adding more processes to your application, there's a likely chance of overlap. I recommend using distinct task names across all business processes. But if you want more granularity in task selection, you can query the `JbpmContext` object directly.

What difference does it make if the tasks are all mixed together? If you're only displaying tasks to a user, it doesn't matter. The description of the task should be enough for the user to make the necessary distinction. However, if you want to allow the user to act on the task, you'll quickly discover that the action that you bind to it is specific to the type of task.

Let's put the task list into practice. In the Courier process, the task list serves as the user's inbox. After reading a message, the user can acknowledge that it's read by completing the task. To perform an action on a task, it must be placed "in scope." A convenient way of selecting a task is to use the Seam UI command components, `<s:link>` and `<s:button>`, which are capable of passing along the task selection via the `taskInstance` attribute. Each row in the inbox has the following button, where `#{_task}` represents one of the items in the task list:

```
<s:link action="#{courier.acknowledge}" taskInstance="#{_task}"
  value="Acknowledge"/>
```

It's permissible to use a Seam UI command component in this case since the expression in the `taskInstance` attribute is evaluated at the time the button (or link) is rendered. Following the evaluation, the `taskId` parameter is appended to the target URL.

In the same way, if you're using a standard JSF command link, you need to append the `taskId` parameter to the URL so that Seam can locate the task:

```
<h:commandLink action="#{courier.acknowledge}" value="Acknowledge">
  <f:param name="taskId" vaue="#{_task.id}"/>
</h:commandLink>
```

You can customize the name of the parameter that Seam uses to locate a task, but we get to that later. For now, we have what we need to put the task in scope so that work can be performed on it.

As mentioned earlier, each task in the list is an instance of the `TaskInstance` class from jBPM. A ton of good information is available from this object. The most relevant properties are shown in table 14.3.

---

[2] jBPM uses Hibernate's second level cache to minimize load on the database, but it still makes sense to avoid executing a query if it isn't necessary.

**Table 14.3 Commonly used properties on `TaskInstance`**

| Property name | Value type | Description |
|---|---|---|
| `name` | `String` | The name of the task, as provided on the `<task>` node |
| `description` | `String` | The description of the task, as provided on the `<task>` node |
| `priority` | `int` | The priority of the task, as provided on the `<task>` node |
| `variables` | `Map<String, Object>` | A map of all of the variables available in the current process instance |
| `create` | `Date` | The date on which this task was created |
| `dueDate` | `Date` | The due date, as provided on the `<task>` node |
| `actorId` | `String` | The actor assigned to work on this task |
| `previousActorId` | `String` | The previous actor, if any, that was assigned to work on this task |
| `processInstance` | `ProcessInstance` | A reference to the process instance, another jBPM class that provides access to additional process-related information, such as when the process was started, the version, and the complete process definition |

The most interesting aspect of `TaskInstance` is that it's a variable container. The variables are accessed using the `variables` property on this class. The `TaskInstance` exposes variables associated with the task as well as variables stored in the process, which is the parent context of the task. In fact, when transferring business process variables into the jBPM context, Seam registers them with the task instance if it's available, or the process instance otherwise.

Let's grab the receive tasks and use them to create a message inbox for the user. The `variables` property on `TaskInstance` will be used to fetch the sender of the message and the description of the task holds the content of the message. But before creating the table, we need to bind the receive tasks to an event-scoped variable using a factory since Seam doesn't store the task lists by default:

```
<factory name="cachedTaskInstanceListForType"
  value="#{taskInstanceListForType}"/>
<factory name="messagesFromCourier"
  value="#{cachedTaskInstanceListForType['receive']}"/>
```

Then, we can use the `messagesFromCourier` variable to populate the user's inbox:

```
<rich:dataTable var="_task" value="#{messagesFromCourier}"
  rendered="#{identity.loggedIn and not empty messagesFromCourier}">
  <h:column>
    <f:facet name="header">Message</f:facet>
```

```
    #{_task.description}
  </h:column>
  <h:column>
    <f:facet name="header">From</f:facet>
    #{_task.variables.sender}
  </h:column>
  <h:column>
    <f:facet name="header">Sent at</f:facet>
    <h:outputText value="#{_task.create}">
      <s:convertDateTime type="both" timeStyle="short"/>
    </h:outputText>
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <s:link action="#{courier.acknowledge}"
      taskInstance="#{_task}" value="Acknowledge"/>
  </h:column>
</rich:dataTable>
```

The Acknowledge button allows the user to perform work on the task. Before revealing that action, let's look at another way that tasks can be assigned to a user.

If the Courier process were modified to put messages in a pool—assigning them to an actor group rather than to an individual recipient—candidate actors would have to first assign the task to themselves in order for it to appear in their message inboxes. To have messages sent to the pool, simply apply the `#{message.recipient}` to the `pooled-actors` attribute on the `<assignment>` node rather than the `actor-id` attribute:

```
<task name="receive" description="#{message.content}" >
  <assignment pooled-actors="#{message.recipient}"/>
</task>
```

To support sending a "broadcast" message, you'll need to change the UI input for `#{message.recipient}` so that it offers a list of groups rather than a list of actor ids. If you want to offer support for sending messages to a group or an individual in the same business process, you'll need to add a decision node to the process that routes to a group or individual inbox (the first having an individual assignment and the second a pooled assignment). This routing is necessary since the `<assignment>` node isn't capable of conditional task assignment between an actor and a pool of actors.

```
<decision name="route">
  <transition name="actor" to="actor-inbox">
    <condition>#{!message.broadcast}</condition>
  <transition name="group" to="group-inbox"
    <condition>#{message.broadcast}</condition>
</decision>
```

When a message is sent to an actor group, it shows up in the `pooledTaskInstanceList` for any actor in the target group. You typically offer a command link or button in each row that users can use to assign one of the tasks to themselves. When rendering a group inbox, you usually don't reference process variables since the collection of tasks come from a variety of business processes having different sets of variables. Thus, the name of

the task is shown to help distinguish it from the others. This table uses the pooled task list directly, though I recommend always consulting the task lists through a factory.

```
<rich:dataTable var="_task" value="#{pooledTaskInstanceList}"
  rendered="#{identity.loggedIn and not empty pooledTaskInstanceList}">
  <h:column>
    <f:facet name="header">Type</f:facet>
    #{_task.name}
  </h:column>
  <h:column>
    <f:facet name="header">Description</f:facet>
    #{_task.description}
  </h:column>
  <h:column>
    <f:facet name="header">Created</f:facet>
    <h:outputText value="#{_task.create}">
      <s:convertDateTime type="both" timeStyle="short"/>
    </h:outputText>
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <s:link action="#{pooledTask.assignToCurrentActor}"
      taskInstance="#{_task}" value="Assign"/>
  </h:column>
</rich:dataTable>
```

When the user clicks the Assign button, the built-in `pooledTask` component assigns the selected task to the current actor as returned by `#{actor.id}`. After being assigned, the task is migrated into one of the other task lists based on its type, where the user can perform actual work on the task. You can think of the pooled task list as being focused on engaging a volunteer for a task. The individual and group task lists for the user twoputt are shown in figure 14.6. As you can see, twoputt has several messages waiting his acknowledgment and one group task that he can assign to himself.

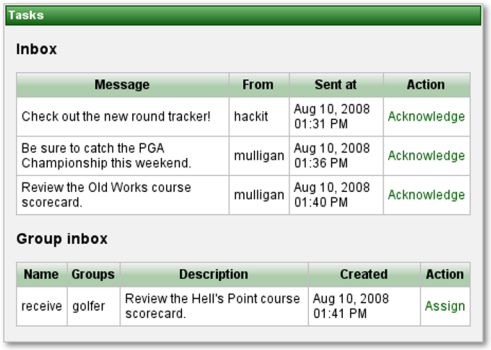


**Figure 14.6 A list of individual and group tasks available to one of the members**

As a variation, the `pooledTask` component can be used to have the pooled task assigned to another actor. You first add an input element (a select menu) and bind it to the `actorId` property of a new event-scoped component named `assignment`. Then add a command link before the input that reads "Assign to:":

```
<h:commandLink action="#{pooledTask.assign(assignment.actorId)}"
  value="Assign to:">
  <f:param name="taskId" value="#{_task.id}"/>
</h:commandLink>
```

```
<h:selectOneMenu value="#{assignment.actorId}" required="true">
  <s:selectItems var="_actorId" value="#{availableActorIds}"
    label="#{_actorId}"/>
</h:selectOneMenu>
```

There are other ways to specify the target actor. With the `assign()` method, you can refer to any context variable that's in scope by taking advantage of parameterized method calls in the JBoss EL.

The user has committed to performing the task. Let's see how tasks are completed.

### 14.3.7 Completing the task at hand

Tasks are controlled using the same declarative style used to create a process. A task is started using either the `@StartTask` annotation on a component method or the `<start-task>` page descriptor node. When the `@StartTask` annotation is used, the task is started only after the annotated method completes successfully.

Seam knows which task to start by using the value of the `taskId` request parameter to look up the task in the jBPM context. To customize the name of the parameter Seam uses to look up the task, you can specify an override using the `taskId` attribute on the annotation or page descriptor node. If the task is found, the task and the process to which it belongs are associated with the current conversation and the task is marked as started.

The `@BeginTask` annotation and `<begin-task>` page descriptor node offer a benign alternative to `@StartTask` and `<start-task>`. The two begin directives associate the selected task and the process to which it belongs with the current conversation but *don't* start the task, thus avoiding the exception in the event that the task is already started. The `@StartTask` and `@BeginTask` annotations are both summarized in table 14.4.

**Table 14.4 The `@BeginTask` and `@StartTask` annotations**

| | |
|---|---|
| **Name:** | BeginTask/StartTask |
| **Purpose:** | Starts a conversation and associates both the task instance and the owning process instance with that conversation. A channel is opened between the Seam business process context and the jBPM variable context associated with the process instance so that context variables can be exchanged between the two containers. The `@StartTask` marks the task as started once the method returns successfully, whereas the `@BeginTask` doesn't affect the task. |
| **Target:** | `METHOD` |

| Attribute | Type | Function |
|---|---|---|
| `taskIdParameter` | `String` | The name of the request parameter holding the id of the task to be resumed. Used as an alternative to `taskId`. Default: empty string (`taskId` attribute is used instead). |
| `taskId` | `String` (EL) | An EL expression that resolves to the id of the task instance to be resumed. Used as an alternative to `taskIdParameter`. Default: `#{param.taskId}`. |

**Table 14.4 The `@BeginTask` and `@StartTask` annotations *(continued)***

| | | |
|---|---|---|
| `pageflow` | `String` | The name of the page flow definition used to manage the navigation for this conversation. Default: empty string. |
| `flushMode` | `FlushModeType` | Changes the flush mode of any Seam-managed JPA `EntityManager` or Hibernate `Session` in this conversation for the remainder of the conversation's lifetime. Can be used to conduct an application transaction. Default: `FlushMode.AUTO`. |

A task is completed when a component method with an `@EndTask` annotation is invoked or an `<end-task>` page descriptor node is crossed. As with the `@StartTask` annotation, the `@EndTask` annotation marks the task as complete only if the method completes successfully. The `@EndTask` annotation can also dictate which named transition the process should follow upon exiting the task node. If the transition needs to be specified dynamically, you can inject the built-in `Transition` component, named `transition`, and establish the transition using this component's `setName()` method during the execution of the method. A summary of the `@EndTask` annotation is provided in table 14.5.

**Table 14.5 The `@EndTask` annotation**

**Name:** EndTask

**Purpose:** Ends the long-running conversation and disassociates the task from the current context. The process instance remains available until the conversation is cleaned up. If the method throws an exception or has a non-void return type and returns a null value, the task is not ended.

**Target:** `METHOD`

| Attribute | Type | Function |
|---|---|---|
| `transition` | `String` | The name of the business process transition that should be followed exiting the `<task>` node. Default: either the transition set on the `transition` component or the default transition. |
| `beforeRedirect` | `boolean` | A flag indicating whether or not the conversation should be removed before a redirect is issued. Default: false. |

The task directives serve as long-running conversation boundaries, offering a special type of conversation that has awareness of the overarching business process. (You use the task-oriented directives in place of the normal ones.) When a task is started by the request encountering a start or begin task directive, the task instance, process instance, and the variables in that process instance are all bound to the current conversation. Those variables remain accessible until the conversation ends or the conversation is associated with a different task. As with a normal begin conversation directive, the start and begin task directives are capable of activating the manual flush feature of Hibernate to instrument an application transaction while a task is being performed.

If you want to be able to associate the process with the current conversation without starting or resuming a task, you can use the `@ResumeProcess` annotation or `<resume-process>` page descriptor. Seam knows which process to resume by using the value of the `processId` request parameter to look up the process in the jBPM context. To customize the name of the parameter Seam uses to look up the process, you can specify an override using the `processId` attribute on the annotation or page descriptor node.

A summary of the `@ResumeProcess` annotation is provided in table 14.6.

**Table 14.6 The `@BeginTask` and `@StartTask` annotations**

| | |
|---|---|
| **Name:** | ResumeProcess |
| **Purpose:** | Associates an existing process instance with the current conversation. A channel is opened between the Seam business process context and the jBPM variable context associated with the process instance so that context variables can be exchanged between the two containers. `@StartTask` and `@BeginTask` also resume the business process. |
| **Target:** | `METHOD` |

| Attribute | Type | Function |
|---|---|---|
| `processIdParameter` | `String` | The name of the request parameter holding the id of the process to be resumed. Used as an alternative to `processId` and `processKey`. Default: empty string. |
| `processId` | `String` (EL) | An EL expression that resolves to the task instance to be resumed. Used as an alternative to `taskIdParameter` and `processKey`. Default: `#{param.processId}`. |
| `processKey` | `String` (EL) | An EL expression that resolves to the business key of the process to be resumed. Used as an alternative to `processIdParameter` and `processId`. The `definition` attribute must be specified in order to use this lookup method. Default: empty string. |
| `definition` | `String` | The name of the business process definition. Required for looking up the process instance by `processKey`. Default: empty string. |

Seam provides the `@Transition` annotation, which is placed on an action method, for signaling a process instance to transition the process token out of a node without a task or in lieu of the task being completed. The transition defined in the value of this annotation is executed after the method annotated with `@Transition` completes successfully. If a transition isn't provided by the annotation, either the transition defined on the built-in `transition` component or the default transition of the node is followed. A summary of the `@Transition` annotation is provided in table 14.7.

The Courier process doesn't take advantage of the conversation aspects of the task control at all when performing work on the task. In fact, the `acknowledge()` method performs the simplest possible task operation by starting and ending the task in the same operation.

**Table 14.7 The `@Transition` annotation**

| | |
|---|---|
| **Name:** | Transition |
| **Purpose:** | Sends a signal to the current process instance to perform a transition. If the method isn't successful, the signal isn't issued. |
| **Target:** | METHOD |

| Attribute | Type | Function |
|---|---|---|
| `value` | `String` | The name of the business process transition that should be followed. Default: either the transition set on the `transition` component or the default transition. |

```
@Name("courier")
public class Courier {
   ...
   @BeginTask @EndTask(transition = "acknowledge")
   public void acknowledge() {}
}
```

When the user clicks the Acknowledge button, bound to `#{courier.acknowledge}`, JSF invokes the `acknowledge()` method, which ends the task and removes it from the user's inbox. Completing the task also takes the process execution out of the wait state. From there, the process token follows the acknowledge transition, as specified in the `@EndTask` annotation, to the `<end-state>`, ending the process instance.

That's it! You've completed your first business process! Your professional life will never be the same. Even if it's not career changing, you'll likely find a good case soon enough that can benefit from a business process.

Are you ready to do it again? As promised, I step you through how to implement the "Add to friends" feature using a business process.

## 14.4 The process of making friends

When you become brainwashed by CRUD applications, you tend to overlook the opportunity to use a business process to manage the steps leading up to a database insert or update. Rather than adding special columns for indicating the state of a record (e.g., pending, awaiting approval), you can track the state of the record in a business process before it's saved and only save it once it's ready to be persisted permanently. The idea here to avoid expanding your schema to hold transient information (information about the record that isn't relevant outside the business process). It also avoids database records that aren't yet "official" from being mixed up in query results.

Consider the case of a user adding a friend in a social networking tool. The link between the users can't be made official until the other user confirms the friendship. The request must be persisted until the confirmation is made. To accommodate the storage of the pending request, your instinct may be to introduce a database table. Although that approach may work, it reduces the clarity of the overall process. It is also shortsighted. If more sophisticated requirements are introduced, one table becomes many, and it becomes harder and harder to figure out where the data is

located and how it plays into the process. A much better approach is to use a business process. Not only can the business process manage pending operations naturally, it can also deal with all of the notifications that need to be sent back and forth.

### 14.4.1 The foundation of a good friendship

Listing 14.6 shows the Add Friend business process described in jPDL. The process involves two actors and one task. The user making the friend request is the initiator and the user being added as a friend is the prospect. What sets the Add Friend business process apart from the Courier process is what happens in the transitions out of the task. If the friend request is approved, an instance of the Courier business process is kicked off to send a confirmation message back to the initiator. If the request is rejected, it's silently discarded. Two end states are reserved to accommodate the need for additional nodes in the future for the two different outcomes. For instance, down the road, an approval may lead to additional tasks for negotiating details about the friendship, thus extending the process.

**Listing 14.6 The Add Friend business process definition expressed in jPDL**

```
<process-definition ...
 name="Add Friend">

 <start-state name="add friend">
  <transition name="notify" to="friend request"/>
 </start-state>

 <task-node name="friend request">
  <task name="confirm friend"
   description="Confirm friend request from #{friendRequest.initiator}">
   <assignment actor-id="#{friendRequest.prospectiveFriend}"/>
  </task>
  <transition name="approve" to="approvalNotification"/>
  <transition name="reject" to="rejected"/>
 </task-node>

 <process-state name="approvalNotification">
  <sub-process name="Courier"/>
  <variable name="prospectiveFriend" mapped-name="sender"/>
  <transition to="approved"/>
 </process-state>

 <end-state name="approved"/>
 <end-state name="rejected"/>

</process-definition>
```

The graphical representation of the Add Friend process was presented as figure 14.1 earlier in the chapter.

Friend relationships are represented by the `FriendLink` entity, shown here:

```
@Entity
@Table(name="FRIEND_LINK", uniqueConstraints =
 @UniqueConstraint(columnNames = {"golfer_id", "friend_id"}))
```

```
public class FriendLink implements Serializable {
   private Long id;
   private Golfer golfer;
   private Golfer friend;

   @Id
   @GeneratedValue
   public Long getId() { return this.id; }
   public void setId(Long id) { this.id = id; }

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name="golfer_id")
   public Golfer getGolfer() { return this.golfer; }
   public void setGolfer(Golfer golfer) { this.golfer = golfer; }

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name="friend_id")
   public Golfer getFriend() { return this.friend; }
   public void setFriend(Golfer friend) { this.friend = friend; }
}
```

The friend links are then wired into the `Golfer` entity as a one-to-many property named `friendLinks`. A convenience method for adding a new friend link is also included.

**INFO** It is possible to eliminate this association entity, allowing the golfer entity to map directly to a collection of friends, by using join table association mappings. But the configuration proposed here will suffice for the purpose of this example.

Here are the relevant portions of the `Golfer` entity showing the additions:

```
@Entity
...
public class Golfer implements Serializable {
   ...

   private Set<FriendLink> friendLinks = new HashSet<FriendLink>();

   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY,
    mappedBy = "golfer")
   public Set<FriendLink> getFriendLinks() { return this.friendLinks; }

   public void setFriendLinks(Set<FriendLink> friendLinks) {
      this.friendLinks = friendLinks;
   }

   public void addFriend(Golfer friend) {
      FriendLink link = new FriendLink();
      link.setGolfer(this);
      link.setFriend(friend);
      friendLinks.add(link);
   }
}
```

The end goal of the Add Friend business process is to create two `FriendLink` instances, one for the golfer who initiates the request and an inverse one for the prospective friend,

thus establishing a two-way relationship between the golfers. But for that to happen, the friend request must be approved. Let's see how a friend request is created.

### 14.4.2 Can I call you my friend?

A friend request comes about as the result of wanting to stake the claim that someone is your friend. Otherwise, I could say that I am friends with Tiger Woods, which wouldn't be true (at least not yet).

To add a friend, a candidate first has to be selected. Locating nonfriends can be handled in a variety of ways. For the purpose of this example, let's put an "Add to friends" link on all the golfers' profile pages except for the profile page for the current golfer (distinguished by the `isNot()` helper method on `Golfer`). This link passes the id of the golfer to the addFriend.xhtml page using the request parameter named `friendId`. Here's the JSF fragment just described:

```
<s:button id="addFriend" value="Add to friends" view="/addFriend.xhtml"
 rendered="#{s:hasRole('golfer') and
   selectedGolfer.isNot(currentGolfer)}">
 <f:param name="friendId" value="#{selectedGolfer.id}"/>
</s:button>
```

Using what you learned about the `EntityHome` framework class in chapter 10, you can then create a `friendHome` component to manage the selected golfer. For conciseness, we take the XML approach. First, define a new `EntityHome` component and an alias for the managed instance in the component descriptor. Notice in this example that we can reuse the `Golfer` entity class to serve another purpose.

```
<framework:entity-home name="friendHome"
 entity-class="org.open18.model.Golfer"/>
<factory name="friend"
 value="#{friendHome.instance}"
 auto-create="true"/>
```

The next step is to configure a page parameter to read the `friendId` request parameter into the `#{friendHome.id}` value binding, which makes the golfer with that id available under the context variable `friend`. Create the page descriptor addFriend.page.xml in the web root and populate it with the following contents:

```
<page>
 <param name="friendId" value="#{friendHome.id}"
   converterId="javax.faces.Long"/>
 <restrict>#{s:hasRole('golfer')}</restrict>
</page>
```

The addFriend.xhtml page gives users a chance to verify the friend request they're about to make. When they click the Add Friend button, it executes the `#{friendRequester.addFriend}` action, which, as you'll see in a moment, kicks off the Add Friend business process. The body of the addFriend.xhtml page is shown here:

```
<richpanel>
 <f:facet name="header">Add #{friend.username} as a friend?</f:facet>
```

```
  <h:form>
    <p>You are about to add #{friend.username} as a friend.
      #{friend.username} will be asked to confirm that you're friends.
    </p>
    <p>
      Add a personal message: (leave blank to exclude)<br/>
      <h:inputTextarea value="#{friendRequester.message}"/>
       </p>
    <s:button view="/profile.xhtml" value="Cancel">
      <f:param name="golferId" value="#{friend.id}"/>
    </s:button>
    <h:commandButton action="#{friendRequester.addFriend}"
      value="Add Friend"/>
  </h:form>
</rich:panel>
```

That takes care of the UI. Let's now move on to the action component, `FriendRequester`, shown in listing 14.7. The `addFriend()` method is responsible for outjecting the `initiator` and `prospectiveFriend` context variables and initiating the Add Friend process. These two context variables are also outjected in the `friendRequest` property as a workaround for a bug in Seam 2.0 that prevented root context variables from being accessed in the business process definition directly.

**Listing 14.7 The action component for initiating the Add Friend process**

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

@Name("friendRequester")
public class FriendRequester {

   @In protected FacesMessages facesMessages;

   @In protected Golfer friend;                          (1)

   @In protected Identity identity;                      (2)

   @Out(scope = ScopeType.BUSINESS_PROCESS)
   protected String initiator;                           (5)

   @Out(scope = Scopetype.BUSINESS_PROCESS)
   protected String prospectiveFriend;                   (5)

   @Out(value = "personalMessage",
     scope = ScopeType.BUSINESS_PROCESS, required = false)
   protected String message;                             (3)

   @Out protected FriendRequest friendRequest;           (7)

   @BypassInterceptors
   public String getMessage() { return this.message; }

   @BypassInterceptors
   public void setMessage(String message) { this.message = message; }

   @CreateProcess(definition = "Add Friend")             (6)
   public void addFriend() {
      initiator = identity.getUsername();                (4)
      prospectiveFriend = friend.getUsername();
```

```
        friendRequest = new FriendRequest(initiator, prospectiveFriend);
        facesMessages.add("A friend request has been sent to " +
           "#{prospectiveFriend} for approval.");
    }
}
```

**NOTE** I'll admit that there is quite a lot of bijection going on in this component. Although it can be justified in this case, you do want to be sure to have fine-grained Seam components, especially when using bijection. If you try to mix too many functions in a single component, things end up getting pretty messy. Avoid monolithic components!

The Add Friend command button causes the `addFriend()` method to be invoked. The selected golfer is reestablished on the `friendHome` component as per the page parameter configuration in addFriend.page.xml. Hence, the context variable `friend` ❶, which resolves the expression `#{friendHome.instance}`, injects the selected golfer into the `friendRequester` component. The current user is the one issuing the friend request, so the user's authentication details ❷ are pulled in as well. A personal message can be included with the request. The personal message is bound directly to the `friendRequester` component through the `message` bean property and later exported to the business process as the variable `personalMessage` ❸ so that it can be displayed to the friend when confirming the request. The `@BypassInterceptors` annotation is placed on both accessor methods for `message` to prevent bijection. Accessor methods typically don't benefit from interception and can also interfere with the bijection constraints.

The `initiator` and `prospectiveFriend` ❹ properties are initialized to the username of the current user and the friend, respectively. Both of these context variables are referenced in the business process definition. The context variable `initiator`, as well as `personalMessage`, provide information to the friend regarding the friend request. Therefore, they're exported to the business process context ❺ that's started after the `addFriend()` method completes ❻, making them available for the duration of the process. The context variable `friendRequest` is only needed short-term for the purpose of creating and assigning the task. Since the task is created in the same request as the business process, the event scope suffices here ❼.

The task is now on the plate of the proposed friend. Will the friend accept?

### 14.4.3 Issuing the verdict

A friend or not a friend—that's the decision to be made for each friend request. The name given to this task—also considered its type—is *confirm friend*. Thus, the friend requests can be presented to the prospective friend using the `taskInstanceListForType` task list component. Let's first create a factory named `friendRequests` that uses the cached version of the task instance list:

```
<factory name="friendRequests"
  value="#{cachedTaskInstanceListForType['confirm friend']}"/>
```

Then refer to this list to render the list of requests in a data table and provide corresponding links for the user to review each request:

```
<h3>Friend requests</h3>
<h:dataTable var="_task" value="#{friendRequests}"
  rendered="#{identity.loggedIn and not empty friendRequests}">
  <h:column>
    You have a friend request from #{_task.variables.initiator}.
  </h:column>
  <h:column>
    <s:link action="#{friendDecision.review}"
      taskInstance="#{_task}" value="Review"/>
  </h:column>
</h:dataTable>
```

This time around, we get to see how the task plays a role in a conversation. The `#{friendDecision.review}` action begins a long-running conversation. It also begins the task. By beginning the task, the context variables stored in the business process are available for as long as the conversation is active, or the task is explicitly ended.

**NOTE** I chose to "begin" rather than "start" the task to avoid the potential for an errant exception since a task can only be put into the start state once. You only want to start a task when it would be considered an error to attempt to start it again. An example is a report in an issue-tracking system. Once started, the issue can't be started again. The UI would have to accommodate for the state of the task and offer the appropriate controls.

A decision of how to respond to the friend request can be made from the reviewFriendRequest.xhtml page, shown here:

```
<rich:panel>
  <f:facet name="header">Is #{initiator} your friend?</f:facet>
  <h:form id="friendDecision">
    <p><b>#{initiator}</b> wants to add you as a friend. We need you
      to confirm that you are, in fact, friends with #{initiator}.</p>

    <s:fragment rendered="#{not empty personalMessage}">
      <p>#{initiator} says, "#{personalMessage}"</p>
    </s:fragment>

    <p>To either approve or reject this friend request, please use
      one of the buttons below. Please note that #{initiator} will
      only be notified if you approve the friend request.</p>

    <s:button action="#{friendDecision.reject}" value="Reject"/>
    <s:button action="#{friendDecision.approve}" value="Approve"/>
  </h:form>
</rich:panel>
```

The rendered output of the decision form is shown in figure 14.7.

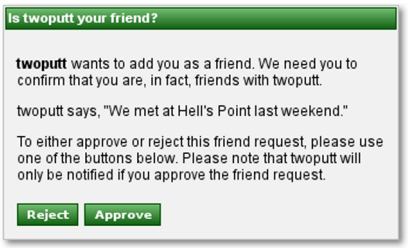


**Figure 14.7 The dialogue used to reject or approve a friend request**

The prospective friend is presented with both the username of the golfer requesting the friend link and the personal message, if provided, to help influence the decision. The friend can then choose to approve or reject the friend request. Either action will end both the

conversation and the task, though with different outcomes. The approve action, `#{friendDecision.approve}`, establishes a friend link between the two golfers and sends back a confirmation message using the Courier process, whereas the reject action, `#{friendDecision.reject}`, silently discards the request. Notice that neither of these actions requires specifying the task instance since it's already associated with the long-running conversation. The `friendDecision` component, where all of this activity occurs, is shown in listing 14.8.

**Listing 14.8 The conversational component that handles the friend request**

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

@Name("friendDecision")
@Scope(ScopeType.CONVERSATION)
public class FriendDecision implements Serializable {

   @In protected FacesMessages facesMessages;

   @In protected EntityManager entityManager;

   @In protected Identity identity;

   @In(scope = ScopeType.BUSINESS_PROCESS)
   protected String initiator;

   @Out(value = "message", required = false)
   protected Message verdict;

   @BeginTask
   public String review() {
      return "/reviewFriendRequest.xhtml";
   }

   @EndTask(transition = "approve")
   @Transactional
   public void approve() {
      Golfer golfer1 = (Golfer) entityManager.createQuery(
         "select g from Golfer g " +
         "where g.username = #{initiator}")
         .getSingleResult();
      Golfer golfer2 = (Golfer) entityManager.createQuery(
         "select g from Golfer g " +
         "where g.username = #{identity.username}")
         .getSingleResult();

      golfer1.addFriend(golfer2);
      golfer2.addFriend(golfer1);
      prepareMessages(true);
   }

   @EndTask(transition = "reject")
   public void reject() {
      prepareMessages(false);
   }

   private void prepareMessages(boolean approved) {
      verdict = new Message();
```

**Injects username of golfer requesting link**

❷

**Begins task and conversation**

**Approves link, ends conversation**

**Participates in global transaction**

**Adds companion FriendLink instances**

**Rejects link, ends conversation**

❶

```
        verdict.setRecipient(initiator);
        String resultStr = approved ? "accepted" : "turned down";
        verdict.setContent(identity.getUsername() +
           " has " + resultStr + " your friend request.");
        if (approved) {
           facesMessages.add("You're now friends with {0}.", initiator);
        }
        else {
           facesMessages.add("You have turned down the friend request " +
              "made by {0}.", initiator);
        }
     }
  }
```

If you look back at the Add Friend process definition, you'll see that the approve transition uses Courier as a subprocess. The `prospectiveFriend` is mapped directly onto the `sender` process variable using `<variable>`. Instead of the `message` context variable being populated by a JSF form, as you saw in the earlier example, it's prepared in code ❶ and outjected to the event scope ❷. The Courier process aggregates this information to create an approval message that is sent back to the initiator. The recipient will see the message in the `taskInstanceListForType['receive']` just as if it were any other message. The acknowledge action works as described earlier.

There's only one problem right now with the handling of this process: it's possible to issue multiple requests for the same link. Let's see how to detect duplicate tasks.

### 14.4.4 One task at a time, please

To avoid annoying the prospective friend—potentially jeopardizing the chance for a friendship—we want to prevent duplicate requests from being issued. That means peeking into the task list of the prospective friend. Doing so is going to require special interaction with the managed jBPM context component, `jbpmContext`.

The goal here is simple. We query the jBPM context for full task list of the prospective friend, identified by the actor id, and then look for a pending confirm friend task issued by the current user. The logic just described is performed by the `isDuplicateRequest()` method, shown below, which is added to the `friendRequester` component:

```
@In protected JbpmContext jbpmContext;

protected boolean isDuplicateRequest() {
   List<TaskInstance> tasks = jbpmContext.getTaskList(prospect);

   for (TaskInstance task : tasks) {
      if (task.getName().equals("confirm friend") &&
         task.getProcessInstance().getProcessDefinition()
            .getName().equals("Add Friend") &&
         task.hasVariable("initiator") &&
         task.getVariable("initiator").equals(initiator)) {
         return true;
      }
   }

   return false;
}
```

**A common thread in persistence context management**

The built-in `jbpmContext` component serves a similar purpose as the managed persistence context. It returns the native `JbpmContext`, but not before automatically associating its persistence context with the global transaction, registering transaction synchronizations, and ensuring that requests made to jBPM are serialized to avoid concurrency problems. However, since Seam doesn't manage the state of jBPM's persistence context, this component is scoped to the event rather than the conversation.

To make this method more efficient, you could craft a custom HQL query to run against the `jbpmContext`,[3] but the more rudimentary approach will do for now.

Let's put the check for a duplicate friend request to use. This is a good opportunity to demonstrate how to prevent the business process from starting even when a `@CreateProcess` method is invoked. The secret is in how this annotation is processed. In order for the business process to be started, the method marked with the `@CreateProcess` annotation must return without error, and unless it's a void method, it must return a non-null value. If you recall from chapter 3, this description is Seam's definition of success. If a duplicate friend request is detected, we add a message to display to the user and return null to prevent the process instance from being created. Listing 14.9 shows the modified `addFriend()` method from the `friendRequester` component.

**Listing 14.9 The method that checks for a duplicate friend request**

```
@CreateProcess(definition = "Add Friend")
@Transactional public Boolean addFriend() {
   initiator = identity.getUsername();
   prospectiveFriend = friend.getUsername();
   friendRequest = new FriendRequest(initiator, prospectiveFriend);
   if (isDuplicateRequest()) {
      facesMessages.add("You already have a friend request " +
         " issued to #{prospectiveFriend}.");
      return null;
   }
   else {
      facesMessages.add("A friend request has been sent to " +
      "#{prospectiveFriend} for approval.");
      return true;
   }
}
```

That wraps up the friend request. Thanks to the Courier and Add Friend business processes, the members of the Open 18 community can now interact. I'll leave it to you as an exercise to display the list of the golfer's friends (e.g., the friend space).

[3] The hibernate session can be accessed through the `getSession()` method on the `JbpmContext` object.

## 14.5 Other business

This chapter has focused on how to set up jBPM and how to use Seam to manipulate process and task instances from your application. As a trade-off, you have only been given a cursory look at the core jBPM features, namely, task nodes and transitions. jBPM offers extensive functionality, including forks and joins, rule-based decisions and assignments using Drools, a job scheduler, asynchronous tasks and continuations, email support, task notifications and reminders, exception handling, events, embedded scripts, and custom action handlers. Obviously, it would be out of scope to try to cover all of these features here. But I want to leave you with one last example that offers a glimpse of what else is possible within a business process.

### 14.5.1 The business of email confirmations

In chapter 11, we implemented a security measure known as CAPTCHA on the registration page to keep bots from flooding the system with spurious registration requests. Now we want to take that security measure a step further by requiring all newly registered golfers to respond to a confirmation email before being allowed to log in to the site. We can ensure that the database is kept clean of stale registrations by removing the account if the golfer doesn't respond to the email in a fixed amount of time. This calls for a business process!

Several things need to happen in registration confirmation business processes, some of which haven't been covered up to this point:

- Create a task to confirm registration and assign it to the new golfer.
- Send an email to the new golfer that contains a URL for completing that task.
- Activate a timer when the task begins and cancel the task if the timer expires.
- Ensure that if the golfer confirms the task, the timer is canceled.
- Cancel the registration if the timeout is exceeded.

These steps have been converted into a business process definition named Confirm Registration, shown in listing 14.10. Three new types of nodes are used in this process: `<event>`, `<script>`, and `<mail>`. The `<event>` nodes allow you to observe a jBPM event and invoke one or more actions. Events in jBPM are similar to component events in Seam, which you learned about in chapter 6. The built-in jBPM events are raised upon starting or ending a process; entering or leaving a node; creating, assigning, starting, or ending a task; and before and after a signal is fired, among others. Actions can be specified using the `<action>` node, which evaluates a method-binding expression or executes a custom class; the `<script>` node, which executes an inline BeanShell[4] expression; the `<mail>` node, which sends out an email; and the `<create-timer>` and `<cancel-timer>` nodes, which start and cancel a timer, respectively. Apart from the timer actions, all actions can be configured to execute asynchronously by setting the `async` attribute on the node to true. The `<script>` node is especially

[4] http://www.beanshell.org/intro.html

handy during development because it allows you to print messages to the console to verify that the business process is working properly (though certainly not a replacement for proper testing).

**Listing 14.10 The Confirm Registration process definition**

```
<process-definition ...
 name="Confirm Registration">

 <start-state name="register">
  <event type="after-signal">
   <script>System.out.println("A new golfer has registered");</script>
  </event>
  <transition name="request confirmation" to="confirm"/>
 </start-state>

 <task-node name="confirm">
  <task description="Respond to the registration confirmation email">
   <assignment actor-id="#{newGolfer.username}"/>
   <event type="task-assign">
    <mail async="true" to="#{newGolfer.emailAddress}">
     <subject>Confirm Registration</subject>
     <text><![CDATA[#{newGolfer.name},

Thanks for registering. Before you can login, you must verify your
e-mail address by clicking on the following link:

http://localhost:8080/open18/confirmRegistration.seam?
  ➥code=#{confirmationCode}&amp;taskId=#{taskInstance.id}

Open 18
...a place for golfers
Member Services]]></text>
    </mail>
   </event>
   <timer name="confirmation timeout"
    duedate="1 hour" transition="timed out">
    <script>
     System.out.println("Registration canceled");
     taskInstance.end();
    </script>
   </timer>
  </task>
  <transition name="timed out" to="end"/>
  <transition name="confirmed" to="end">
   <script>System.out.println("Registration confirmed");</script>
  </transition>
 </task-node>

 <end-state name="end"/>

</process-definition>
```

To activate this process, copy the contents of this listing into the file confirmregistration.jpdl.xml and register it in the `<bpm:process-definitions>` node in the Seam component descriptor as before.

When the confirmation task is created, an email is sent to the newly registered golfer. You learned how to configure an email session in Seam and to use Facelets templates to render and send emails in chapter 13. jBPM's email support is separate from Seam and therefore has its own configuration. In the Confirm Registration process, we've embedded the contents of the email directly in the process definition, but jBPM also supports externalizing email templates to a separate file.

Out of the box, emails are sent from localhost, which can be overridden by specifying the value of the `jbpm.mail.smtp.host` property in the jbpm.cfg.xml file. You can also set the email address of the sender in the `jbpm.mail.from.address` property:

```
<jbpm-configuration>
  …
  <string name="jbpm.mail.smtp.host" value="mail.open18.org"/>
  <string name="jbpm.mail.from.address" value="noreply@open18.org"/>
</jbpm-configuration>
```

Unfortunately, the email configuration in jBPM is primitive and can't accommodate sending messages using SMTP authentication as required by Gmail and covered in chapter 13. However, expect this to be supported in a future release.

All that's left now is to activate this business process during registration and to respond to the link that's sent in the email to confirm the registration. Listing 14.11 shows the `RegistrationConfirmation` class, which observes a synchronous event named `registrationRequested`, raised during registration, to initiate the registration confirmation process. The event is synchronous so that both the registration and the instantiation of the business process occur in the same JTA transaction using two-phase commit. The `RegistrationConfirmation` component also provides a method that is used to complete the task. The confirmation code is hardcoded in this listing, but should obviously be created dynamically in a real system.

**Listing 14.11 The action component that controls the Confirm Registration process**

```
package org.open18.notification;
import ...;

@Name("registrationConfirmation")
public class RegistrationConfirmation {
   @In protected FacesMessages facesMessages;

   @In(required = false, scope = ScopeType.BUSINESS_PROCESS)
   @Out(required = false, scope = ScopeType.BUSINESS_PROCESS)
   protected String confirmationCode;

   @RequestParameter("code") protected String confirmationCodeVerify;

   @CreateProcess(definition = "Confirm Registration")
   @Observer("registrationRequested")
   public void initiateConfirmation() {
      confirmationCode = "hole-in-one";
   }

   @BeginTask @EndTask(transition = "confirmed")
```

```
    public String confirm() {
       if (confirmationCodeVerify != null &&
          confirmationCodeVerify.equals(confirmationCode)) {
          facesMessages.add("Registration confirmed!");
          return "confirmed";
       }
       else {
          facesMessages.add("Invalid confirmation code.");
          return null;
       }
    }
}
```

Finally, you define a page action in confirmRegistration.page.xml that executes the `confirm()` method when the confirmRegistration.xhtml page is requested:

```
<page view-id="/confirmRegistration.xhtml"
 action="#{registrationConfirmation.confirm}"/>
```

There is one final integration I want to touch on before this chapter closes. One of the downsides of a business process is that it can often oversimplify the real world. For instance, a wide variety of factors may be involved in deciding which transition to follow or which actor should be assigned a task. In chapter 11, you learned that a declarative rules engine is the best choice for making complex decisions. Seam allows you to tap into this intelligence in your business process by using either the `DroolsDecisionHandler` or `DroolsAssignmentHandler` classes from the Seam API, which are applied to the `handler` attribute of the `<decision>` and `<assignment>` nodes, respectively. When using the decision handler, Seam inserts the `decision` object into the working memory and you set the name of the transition to follow in the conclusion of the rule using `decision.setOutcome("transition name")`. When using the assignment handler, Seam inserts the `assignable` object into the working memory, which you use in the conclusion of the rule to set the actor or pooled actors using `assignable.setActorId("actor id")` and `assignable.setPooledActors(new String[] {"actor group"})`, respectively.

Let's assume that we've developed a business process for pairing golfers together (think of it as a match.com for the country club crowd). A golfer initiates a business process to find a partner and the task is assigned to a candidate. But which candidate? We let the rules engine decide. Here's a snippet of what the task might look like:

```
<task name="review" description="Review pairing">
 <assignment handler="org.jboss.seam.drools.DroolsAssignmentHandler">
  <workingMemoryName>pairingRulesWorkingMemory</workingMemoryName>
  <assertObjects>
   <element>#{actor}</element>
   <element>#{date}</element>
   <element>#{skillLevel}</element>
  </assertObjects>
 </assignment>
 ...
</task>
```

Of course, you need to register the working memory invoked by the Seam's built-in Drools handlers in the Seam component descriptor. See the section *Using rules from a jBPM process definition* in the Seam reference documentation for instructions.

The examples in this chapter demonstrate that a business process doesn't have to be long and complicated to be useful. In picking these simple examples, my hope is to get you comfortable using business processes in your application. Once you start using them, I have no doubt that you'll begin leveraging the more advanced capabilities of jBPM. But you must crawl before you walk, and the examples in this chapter gave you an opportunity to make this progression.

## 14.6 Summary

Seam crushes the misconceptions that business processes are just marketing jargon. This chapter attempted to bring down the barriers to adopting business processes. A skeptical approach was chosen since business processes aren't as pervasive as single-user interactions in the history of web application development.

There are two important points you should take away from this chapter. Business processes *are* an important part of application development and Seam makes them no more difficult to use than single-user conversational page flows. The work of loading the business process definition, creating a process execution, putting the process instance in scope, transferring variables to and from the business process context, and sending signals to the process execution is hidden behind declarative controls (i.e., annotations and page descriptor tags); controls you should feel comfortable using from your conversation experience gained in chapter 7 and since.

Although Seam's business process annotations aren't bound to any specific BPM implementation, only the jBPM engine is supported by Seam at the time of this writing. Therefore, the chapter opened by providing an overview of BPM from the perspective of jBPM. You learned how a business process workflow differs from a page flow and the high-level goals of business process languages. The one point that should stand out above all others is that a business process aligns with the slower pace of the real world and is thus very good at waiting and persisting its state between transitions.

You were given a glimpse of Seam's declarative approach to controlling a business process using business process–related annotations and page descriptor tags. You were also introduced to the business process context, a new scope that transparently circulates context variables to the jBPM process context and back. You learned how to set up Seam's jBPM configuration; the most significant steps were adding an additional Hibernate persistence unit and making a move to XA data sources.

The rest of the chapter took a hands-on approach to learning how to define process in jPDL, the jBPM business process definition language supported by Seam. A comparison was made between jBPM actors and JAAS identities, and you got a chance to play with assigning tasks to these actors. You saw firsthand that business processes don't have to be long and involved, with lots of steps, forks, joins, and conditionals, to be useful. Just adding a friend in a social network tool can be represented as a business process; a mere message exchange between two users.

One of the reasons the jBPM integration is so successful is because Seam doesn't try to overstep its bounds in an attempt to be a BPM engine. Instead, Seam provides a support structure, consisting of components and contexts, for jBPM. Seam offers a similar strategy with its Spring integration, which you learn about in the next chapter. Rather than try to duplicate all of what Spring has to offer, Seam finds a way to offer its assistance to Spring so that you can leverage Spring's unique features while still being able to tap into Seam's contextual components.