

Seam and Spring Comparison

Andy Gibson

Seam and Spring Comparison

Andy Gibson

Copyright © 2008 Andy Gibson

Table of Contents

1. Introduction	1
1.1. Overview	1
1.1.1. The Problems Faced	1
1.1.2. Stateful Applications	1
1.1.3. Contextual Navigation	2
1.1.4. Other problems	2
1.1.5. Who this is for	2
1.1.6. Disclaimer	2
1.2. Comparison of Seam and Spring	2
1.2.1. Flows and Conversations	3
1.2.2. Seam	3
1.2.3. Spring	3
1.2.4. Differences in Style	4
1.3. The Test Application	5
2. Implementation with Spring	6
2.1. Initializing the project	6
2.2. Writing some code	6
2.3. Writing View Pages	7
2.4. Listing issues	13
2.5. Handling Issues	15
2.6. Managing Persistence Contexts	16
2.6.1. PC Scope Issues	16
2.6.2. Practical PC Management	16
2.7. Multi-Page Flows	18
2.8. Lookup Management	20
2.9. Creating Entities	20
2.10. Exception Handling	23
2.11. Extras	23
3. The Seam Application	25
3.1. Initializing the project	25
3.2. Seam State Management	25
3.3. Getting Started	26
3.4. Editor Pages	29
3.5. Handling Navigation	33
3.6. The Seam Application Framework	35
3.7. Issues Viewer and Editor	37
3.8. Selecting Projects	46
3.9. The Seam Framework Approach	49
3.10. Other Features	49
4. Summary	52
4.1. Comparing Solutions	52
4.1.1. Configuration	52
4.1.2. Flow Variable Declarations	52
4.1.3. Conversational URLs	53
4.1.4. Page flows	53
4.1.5. Documentation and Examples	54
4.1.6. A Complete Solution?	55
4.1.7. Layering and Decoupling	55
4.1.8. Taking the future into account	56
4.1.9. Usability	56
4.2. Summing it up	57

5. The Wicket Factor	58
5.1. Getting Started	58
5.2. Listing Issues	65
5.3. Editing Projects	68
5.4. Enhancing The Application	74
5.4.1. Entity Drop Downs	74
5.4.2. Selecting Projects	76
5.5. Summing Up Wicket	79
5.5.1. Choosing Between Them	81

List of Figures

2.1. Project listing using Spring	9
2.2. Message after editing the project	12
2.3. Message displayed after cancelling changes	13
3.1. Configuring the project Wizard	25
3.2. Configuring Seam and the DB Connection	25
3.3. Seam Code Assist	28
3.4. List of projects in Seam	28
3.5. Editing a project with Seam	33
3.6. Propagating cancelled messages to the users	34
3.7. Propagating save messages to the user	34
3.8. Create new project	35
3.9. Save new project	35
3.10. View new project in list	35
3.11. Issue view	39
3.12. Issue editor	43
5.1. Drop downs using Wicket	75
5.2. Wicket drop downs with properly titled choices	76

List of Examples

2.1. ProjectDao interface	6
2.2. ProjectDaoBean implementation	7
2.3. Spring Bean Definition	7
2.4. projects.xml flow definition.	8
2.5. projects.xhtml page	9
2.6. projectEdit.xml subflow.	10
2.7. projectEdit.xhtml page	11
2.8. MessageWriter.java helper class	12
2.9. Adding expressions to generate messages on transitions.	12
2.10. projectView.xml web flow.	13
2.11. Listing issues in projects.xhtml	14
2.12. Refreshing the entity in a flow.	14
2.13. Putting the list of issues into a flow as a dataModel.	15
2.14. IssueDao interface.	15
2.15. Putting the issues into a flow based on query results.	15
2.16. Using our new issues variable in the data table.	16
2.17. Refreshing the entity on a save action.	17
2.18. Refreshing the issue by re-loading it as a new instance.	17
2.19. Refreshing / loading the entity.	17
2.20. Refreshing the entity and putting it into the flow	17
2.21. Flow controlled refresh of the entity on the on-exit phase in projectView flow.	18
2.22. Transition to navigate to the project selection page.	19
2.23. The project selection state in the edit issue flow.	19
2.24. projects.xhtml column containing a link for selecting a project	19
2.25. projects.xhtml button for cancelling project selection.	20
2.26. End states in projects.xml flow for selecting or cancelling project selection.	20
2.27. Code in projectDao to find or create a project.	21
2.28. Transition in projects.xml for creating a new project.	21
2.29. issueDaoBean.java method to find or create an issue.	22
2.30. Button in projectView.xhtml to create a new issue.	22
2.31. Transition in projectView.xml flow to create a new issue.	22
2.32. Subflow state for creating a new issue for the project	23
3.1. EJB Local interface for our bean	26
3.2. Implementation of our stateful EJB	27
3.3. JSF page for displaying the projects	28
3.4. Column for the edit link in projects.xhtml	29
3.5. Simple JSF project viewer	29
3.6. ProjectHome interface for handling project instances.	30
3.7. ProjectHome implementation	31
3.8. Project view page parameter configuration from pages.xml	32
3.9. JSF for the back and edit buttons in projectView.xhtml	32
3.10. projectEdit.xhtml page.	33
3.11. The save and cancel buttons in projectEdit.xhtml	33
3.12. Navigation using pages.xml	34
3.13. The add project button on the project.xhtml page.	35
3.14. Issues data table	36
3.15. Define issues factory in components.xml	36
3.16. Issue query defined in components.xml	37
3.17. issueView.xml code.	38
3.18. Defining issueHome and a factory for issue in components.xml	38
3.19. Java code implementation of IssueHome	39

3.20. Binding parameter to entity home id property in pages.xml .	39
3.21. Defining navigation for issueView.xhtml in pages.xml .	40
3.22. issueEdit.xhtml page using the seam decorator layout components	41
3.23. Status selection using Seam selectItems components.	42
3.24. EntityQuery for grabbing a list of the issue status objects.	42
3.25. Implementation of the issueStatusQuery .	43
3.26. pages.xml parameter and conversation definition for issueEdit.xhtml .	43
3.27. issueEdit.jpdl.xml page flow for editing the issue.	44
3.28. Adding the pageflow to components.xml	44
3.29. Invoking the page flow when the conversation is started.	44
3.30. JSF for the save and cancel buttons in issueEdit.xhtml	45
3.31. Code listing for EntityHomeExtended	45
3.32. Framework EntityHome using our extended class.	46
3.33. JSF for the cancel button with our new method	46
3.34. Adding link to issueEdit.xhtml to select a project.	46
3.35. Additional interface methods for project selection on the projectList interface.	47
3.36. Implementation of project selection code in projectListBean .	47
3.37. Adding the transition for selecting a project for the issue.	48
3.38. Column for selecting a project in the table.	48
3.39. Transition to set the project when selected	48
3.40.	49
5.1. WicketTrackerApplication.java	58
5.2. Web.xml	59
5.3. ProjectViewPage.html interface	59
5.4. ProjectViewPage.java class	60
5.5. IssueViewPage.java	61
5.6. IssueViewPage.html	62
5.7. ProjectListPage.html	63
5.8. ProjectListPage.java	64
5.9. WicketTrackerApplication.java	65
5.10. IssueListPanel.html	66
5.11. IssueListPanel.java	67
5.12. Additional markup in ProjectViewPage.html for the issues list panel.	68
5.13. Additional code in ProjectViewPage.java for the issues list panel in initPage()	68
5.14. ProjectEditPage.html	69
5.15. ProjectEditPage.java	70
5.16. ProjectEditForm inner class in the ProjectEditPage class.	71
5.17. IssueEditPage.html	72
5.18. IssueEditPage.java	73
5.19. Adding issue status drop downs IssueEditPage.java .	75
5.20. HTML to use a select instead of text in IssueEditPage.html	75
5.21. Adding the IChoiceRenderer to IssueEditPage.java	76
5.22. Adding the select link in the ProjectListPage.java .	77
5.23. ProjectListPage.html	78
5.24. Adding the project change button to our IssueEditPage.html .	78
5.25. Adding the project selection button in ProjectEditPage.java .	79

Chapter 1. Introduction

This article compares application development using either Seam [<http://www.seamframework.org/>] or Spring Web Flow [<http://www.springsource.org/webflow>] . We'll compare the two by developing a common project and examining different aspects of development. In this first part, we'll look at the histories and the differences between Seam and Spring and start looking at the example application we want to develop, and issues that we might face. These articles are not meant to be a how-to, but are meant to describe the higher level functional differences between the two frameworks as far as they attempt to solve the same problems.

1.1. Overview

In many examples that demonstrate the use of web frameworks, the authors typically take a simple example such as Hello World, or displaying a list of items created in an array or perhaps read from a database and write a form for creating a new item in the database. The strange reality is that most of these are simple enough that we don't need any complex framework to make the example verbose. We turn to frameworks to solve the complex, the monotonous and the things we do daily and yet the only demonstrations we get are mostly for things we might never even do in simple code. Hopefully, these examples might demonstrate how these frameworks can solve the practical problems you meet on a daily basis or have to code around such as contextual navigation and state management.

1.1.1. The Problems Faced

Let's try an outline some of the complexities we face as application developers when writing complete CRUD type applications.

1.1.2. Stateful Applications

Web development is stateless by nature and the process of making it appear it has state is difficult and often done so using different techniques and often with flawed results. One could use either the session or the database to hold the state, but there are a number of problems with that approach both at an application and server level:

- If the user forgets to push data back into the session after changing it there is a danger that changes to mutable objects will be lost from one request to another. There is also no dirty checking for determining which objects need to be replicated.
- Objects in the session are stored by name, which could cause problems when the same page is opened in two browser windows or tabs. There is no way to isolate data to a page, all user data is session scoped.
- Reading objects back and forth between the database per request soon makes the database a performance bottleneck. You also lose your optimistic locking since you are always working on the most recent version of the object, not the version when the user started editing it.
- In an AJAX world where a page may be submitted every few seconds our problems have multiplied because now we are communicating with the server several times instead of once per interaction.

One answer to all these problems lies in using a conversation which is a unit of work done by the user where the server manages the state of the objects involved in the conversation. Data held in the conversation is isolated from the data in other conversations. How the conversation data is stored is really up to the implementation.

As part of the conversation solution, we can also solve problems relating to persistence managers such as the management of entities, the prevention of lazy initialization exceptions and dealing with detached entities. This eliminates the need for the `OpenSessionInView` type solutions to solve these problems.

1.1.3. Contextual Navigation

We also seek to solve the problems of contextual navigation. Typically, navigation from one page to another is done via a static link, usually involving passing parameters. However, this mechanism doesn't let us easily provide contextual navigation. If we have a page that lets you search for an object, we might want to re-use that page to let users select objects for use elsewhere in the application. One example might be searching for and selecting a customer to be sent an order. We can re-use the page to allow the user to search for customers, or in other places where we want to select a customer. This kind of functionality has been used and re-used in thick client apps for years. The goal is for the search form to know as little as possible about where it needs to return to in order to have a loosely coupled and easily re-usable form.

In order to do this we can make the customer search page navigation contextual so the page we are taken to depends on the process that we are currently executing. These processes are defined as work flows and are used to determine which page we navigate to based on the outcome from the page we are on. As part of contextual navigation, we may also need to pass data from one page to another which also involves knowing what process you are active in and where the data needs to go.

1.1.4. Other problems

There are a number of other issues with web development that both of these frameworks solve, whether it is the propagation of messages from one page to another or providing Ajax functions to JSF. It also includes less visual elements such as gracefully handling exceptions in business objects. We will also consider these additional benefits to the core functions that these frameworks provide.

1.1.5. Who this is for

These articles are designed for someone who is considering Seam or Spring for web application development using JSF. The reader should be familiar with JSF since it is the presentation framework used in both examples. For the Spring example, it is assumed the user has some familiarity with Spring, and is able to configure beans for use in Spring. I have gone into a little more detail in the Seam example because most readers will have less familiarity with Seam.

1.1.6. Disclaimer

I have been using Seam for a couple of years now and have been active on their forums and in raising issues with Seam and with JBoss Tools. One could say I am a Seam user, but I do spend time using alternatives.

1.2. Comparison of Seam and Spring

These two frameworks are considered from the view of building a web application which requires both complex state management and navigation, even though our example is still fairly trivial. There are a number of frameworks which can be used to build data driven web sites, but many frameworks do not tend to provide functionality to solve these problems easily. You can solve these problems with other frameworks, especially if you use add-ons, but the goal here is to consider the two cohesive single stacks as out-of-the box solutions.

1.2.1. Flows and Conversations

Seam refers to a conversation as a series of stateful interactions between the user and the server. Within that conversation, you can have a nested conversation which is a child-conversation that starts within the context of the parent conversation. Objects in the nested conversation can see the objects in the parent conversation, but not vice versa.

Spring uses the concepts of flows and conversations. A flow is a set of stateful interactions between the user and server and can invoke nested sub-flows. A conversation in Spring is a term which encapsulates the top level flow from start to finish, and may or may not involve one or more subflows. Each flow can define variables that are scoped to the flow or conversation scoped and therefore available in all flows in that conversation.

Where possible, this document has tried to use the correct terms for the framework being discussed.

1.2.2. Seam

Seam is a project developed by Gavin King under JBoss that sought to unify numerous standard technologies (EL, JPA, JSF, EJB) as well as non-standard ones (Facelets, iText, Jbpm, Richfaces) and to solve the problem of stateful development. As part of the Seam Framework, Gavin really introduced the concept of conversations into the web framework discussion. Not only did it provide the 'seams' between these technologies but it also covered a number of shortcomings (perceived or factual) of those technologies such as the Lazy Initialization Exceptions from Hibernate and the difficulty of using parameters in JSF.

Another goal for Seam was to try and create a single stack of libraries and tools for building applications and interweave them with Dependency Injection (and also bijection) and EL expressions. With Seam, the whole is far greater than the sum of the parts as it works towards a deeper and consistent integration of the individual pieces.

While most users may gravitate towards using Seam on a JBoss Application Server with Hibernate as the JPA solution, users are not tied to these implementations. You can use Tomcat with the JBoss Embedded EJB container or any other EJB container. However, straying from the most common configuration can mean you are in a minority and fewer people have experience with your configuration and any problems you run into. The inverse of this is also true. Since there is a very common configuration that most people use, there are plenty of people who have experience with that configuration, and any problems you might run into. This is just one of the benefits of using the default stack.

Seam is open source and also has commercial backing as it is part of JBoss Enterprise Application Platform (EAP). The libraries are identical to the open source versions except that the pieces of the EAP (which includes Seam) has been tested and certified to work well together by JBoss. The same applies to the IDE plug-ins. The JBoss Tools plugins work the same as the commercial JBoss Developer Studio, except the commercial version includes the EAP and has a slightly more complete setup initially. However, there is nothing that you can't configure yourself in minutes.

1.2.3. Spring

I'm sure most will need no introduction to Spring which was created by Rod Johnson in response to the difficulties and complexities people faced with J2EE prior to EJB 3.0. Spring stayed outside of the standards to create their own framework for developing J2EE applications. Spring lets developers use XML or annotations to specify beans available to the application and automatically inject other beans to provide functionality

The environment that is usually rigid in EJB containers can be specified dynamically in the Spring environment. For example, you can specify a standard transaction manager or subclass it and use that

instead or even write your own from scratch (good luck). This concept is really nice and used throughout the Spring platform from data access to transaction management to managing web page controllers in Spring MVC and Web Flow. This makes the Spring approach more transparent and less like a black box that the EJB containers represent and only forces developers to include and setup the pieces that will be used.

While EJB may be a standard, there are differences between implementations which is not an issue for Spring even when using different containers since you are using the same Spring container in each actual container.

The downside (if you can call it that) is that Spring is in essence proprietary. While it is open source and freely available, there is only one implementation of it, there is only one controller of it (SpringSource) and the direction Spring will take will depend on what side of the bed Rod Johnson gets up on in the morning.

For years, Spring has advocated a stateless approach to software development and has had no real thrust in the area of stateful web development other than to use the session and rely on re-fetching data from the database on each request, as well as providing an interface for using stateful EJBs.

Spring Web Flow is the part of Spring that brings stateful development and workflow management to Spring. It is designed as another add-on to the Spring platform, this time integrating with Spring MVC. The particular aspect of Spring Web Flow we will be looking at is the Java Server Faces focused piece called SpringFaces.

1.2.4. Differences in Style

While Spring offers the flexibility of swapping out one technology for another (except for Spring itself), Seam commits wholeheartedly to the technology set that it has chosen which are mostly standards. Some people may not consider the issue of standards to be a valuable one, but it is an issue that could get Seam a foot in the door of some of the bigger shops. However, there is rigidity in there that prevents you from adjusting parts of the framework like Spring can.

Spring requires you to define beans to provide the environment in which your application will run. For the most part, these beans are stateless singletons. State is held completely in the web flow by defining any stateful objects (i.e. Entities) there, and then passing them to and from the stateless beans (i.e. a Dao).

Seam on the other hand can use this method, but it also tends to worry less about the layering of applications. It's not that it ignores good design, or promotes bad design, it's simply that it isn't really needed. Typically, an entity might be held by a Dao type object, which can also contain methods for JSF event handlers. Seam makes great use of EL (Expression Language) expressions and uses those to decouple the view and logic from the implementation. You wouldn't want your view methods integrated with business methods.

Seam is geared towards using the JSF framework for presentation which again, is the standard component framework. Spring Web Flow has really made efforts to support JSF as well as their own Spring MVC framework. Seam is also diversifying since version 2.1.0 has Wicket support and there is also information out there on using it with Google Web Toolkit (GWT). Exadel also have produced tools for using Seam with Adobe Flex called Flamingo. JSF however is still the strongest view technology out there behind the original struts and still the main focus of Seam. The level of integration these different view technologies will have with Seam or Spring Web Flow may vary however.

Seam takes a global approach to the application in terms of defining stateful data, where it comes from, and what parameters it is derived from. Spring on the other hand takes a localized approach where data is only declared in the flows themselves and passed to and from subflows. The two frameworks differ greatly in the choice of mechanism and both have the pros and cons.

This could create problems in the Seam application since one name change can cause changes across the whole application. It could also create a duplication of data definitions which is why naming is important.

Spring localized data could run into problems if a new variable is given the name of existing data introduced in a parent flow but is shared among the nested subflows which are unaware of the variable name in a parent flow.

Note that when we talk about globally declared data, we are NOT talking about global instances of data (i.e. it is not a global variable), merely that the declaration of binding a name to a class is global in much the same way that Spring declares beans globally within the context of the Spring container.

In terms of IDE support, Seam has JBoss Developer Studio (JBDS) which it has released as a commercial open source product and also has made the plugins available free as open source. This is really a very good development tool for the Seam stack, with a visual page editor and bean code completion nearly everywhere. While the visual IDE may not be everyone's cup of tea, the code completion when coding by hand in the editor is priceless. Spring has the Spring IDE plugins which help when writing bean definitions and even web flow definitions including a visual flow editor, both of which are excellent. JBDS also provides code assist for JSF tags, and attributes using Seam component definitions.

1.3. The Test Application

Our test application will consist of a simple issue tracker. We have a set of projects, each of which has a set of related technical support issues. Each issue has a title, a description and a status represented by a status entity. We have one page which lists the projects, and one page which lets us edit the project, and another which lets us view the project. The project viewer lists the issues with links to edit or add a new issue and we can edit or view the issues. The issue editor will contain a drop down for the issue status value and also, we will eventually have a link on the project field so we can go to our project list and select a project for this issue. This project selection should re-use the project list and demonstrate making a page part of a workflow.

The goal is to make the pages as accessible as possible so that we can get to any page without having to go through another page first. So for example, we can go straight to `/projectView?projectId=3` without having to go to the projects list and selecting project 3. This is somewhat essential in the spirit of loose coupling since we never know where we might want to display those project details from. Both frameworks offer the option of just passing data objects around, but we want to demonstrate loose coupling, especially with page parameters in JSF.

The goal is to mostly use features out of the box which means as little changes to the actual supplied framework as possible. This includes writing additional classes to help out the core framework. Our coding should be limited to solving the use case. Again, this is not so much a how-to, but a description of the frameworks and to demonstrate the differences. By default I've made the entity models use lazy loading everywhere, mainly so I can demonstrate some solutions to Lazy Initialization Exceptions and because I can use eager fetching in the queries if needed.

Chapter 2. Implementation with Spring

2.1. Initializing the project

The Spring application is being developed in a clean version of Eclipse with the Spring IDE plug-ins installed. I'm using Spring 2.5.4 and Web Flow 2.0, and most of the required libraries are coming from the spring dependencies.

To Start with, I'll create a new plain old vanilla dynamic web application. I could add in the JPA or Faces facets, but then I have to start configuring the libraries in Eclipse and it's just easier to do it manually since I'll be adding all my other libraries manually. I added the elements to `web.xml` to configure faces, facelets, spring, and spring web flow. As I deploy it I need to add libraries for the features, so I gradually add the 30 or so needed libraries. I then add the `persistence.xml` file to the classpath, and my `log4j.xml` config file. I configure the `applicationContext.xml` to include three other config files, one for JPA and one for Spring Web Flow, and the other for my actual beans. I'm using MySQL for the database and using JPA in both examples. It can take anywhere from 15 minutes to an hour to get the application deployed and running depending on experience and how much trial and error you employ in finding out which jars you need. Having been through this a number of times before, it took me closer to 15 since I just copied over most of the required files.

Once I have the application created, and I can deploy and run it, I create the model, namely the `Project`, `Issue` and `IssueStatus` classes with the JPA annotations. For simplicity, I am using the `create-drop` JPA functions to create the database schema and also using `import.sql` to insert some test data.

2.2. Writing some code

We also have a `projectDao` interface that contains the methods for dealing with `Project` objects.

Example 2.1. ProjectDao interface

```
public interface ProjectDao {  
  
    Project findProject(Long projectId);  
    Project refresh(Project project);  
    List<Project> findProjects();  
    void save(Project project);  
}
```

This is implemented in a class called `ProjectDaoBean`

Example 2.2. ProjectDaoBean implementation

```
public class ProjectDaoBean implements ProjectDao,Serializable {

    @PersistenceContext
    private transient EntityManager entityManager;

    @Transactional
    public Project findProject(Long projectId) {
        return entityManager.find(Project.class, projectId);
    }

    @Transactional
    public List<Project> findProjects() {
        return entityManager.createQuery("select p from Project p").getResultList();
    }

    @Transactional
    public Project refresh(Project project) {
        entityManager.refresh(project);
        return project;
    }

    @Transactional
    public void save(Project project) {
        entityManager.persist(project);
    }
}
```

This is really simple code, and shouldn't need too much explanation. The entity manager is defined as transient since spring web flow will complain about the entity manager not being serializable as it tries to serialize the bean. We add the `projectDao` bean to the Spring Bean configuration.

Example 2.3. Spring Bean Definition

```
<bean name="projectDao" class="org.issue tracker.dao.ProjectDaoBean"/>
```

2.3. Writing View Pages

Once our application is up and running, it is time to start creating some pages. For reasons that will become clear later on, we will put our project list in its own page flow. We create a new folder `WEB-INF\WebContent\flows\projects\` and in it we add a new file called `projects.xml` which will contain our new web flow.

Example 2.4. projects.xml flow definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      start-state="projects">

  <view-state id="projects">
    <on-render>
      <evaluate expression="projectDao.findProjects()"
        result="flowScope.projects" result-type="dataModel" />
    </on-render>

    <transition on="edit" to="projectEdit" />
    <transition on="view" to="projectView" />

  </view-state>

  <subflow-state id="projectEdit" subflow="projectEdit">
    <input name="projectId" value="projects.selectedRow.id" />
    <transition on="cancel" to="projects"/>
    <transition on="save" to="projects"/>
  </subflow-state>

  <subflow-state id="projectView" subflow="projectView">
    <input name="projectId" value="projects.selectedRow.id" />
    <transition on="close" to="projects"/>
  </subflow-state>

</flow>
```

SWF lets you customize the names for flows, pages and the url used to invoke it, but it uses sensible defaults such as the state name. For now, I've used this default. The start view is the view state called `projects`, which calls `findProjects()` on my stateless bean when the page is rendered, and wraps the result in a JSF data model and puts it in a `flowScope` variable. The flow scope is a scope which lasts for the life of the flow, and `projects` is the name of the variable. Note that we can now refer to this variable as `#{projects}` since all scopes will be searched until the variable is found. We have two transitions from the `projects` page, `edit` and `view` which takes us to the edit and view states which are defined as subflows. We could define the edit/view process as part of this `projects` flow, but the problem with that is we cannot re-use those flows from other places if we do that. Since we want to provide direct access to the view and edit pages via a URL, we have chosen to put them in separate flows.

In each subflow, we define an input variable called `projectId` which is the Id of the selected project. Since we use a `dataModel` as the source of data for the table, we get clickable tables where the selected row in the data model is set based on the item clicked in the table that caused the postback.

Example 2.5. projects.xhtml page

```
<h:messages globalOnly="true" styleClass="message" />

<h:form>

  <h:dataTable value="#{projects}" var="v_project">
    <h:column>
      <f:facet name="header">ID</f:facet>
      <h:outputText value="#{v_project.id}" />
    </h:column>

    <h:column>
      <f:facet name="header">Title</f:facet>
      <h:commandLink value="#{v_project.title}" action="view"/>
      (<h:commandLink value="Edit" action="edit"/>)
    </h:column>
  </h:dataTable>

</h:form>
```

Figure 2.1. Project listing using Spring

We reference the variable `#{projects}` which refers to the projects variable we set up in the on-render stage in the page flow. We display the title of the project as a link which returns the action `view` and we have an edit link which returns the `edit` action. These actions mean nothing in the page itself, they only have meaning in the faces navigation, or in our case, the Spring flow. Looking at the flow, an edit or view action transitions to the subflows to edit or view the project and they have the project Id passed in to them. With that in mind, lets look at the project edit subflow.

Example 2.6. projectEdit.xml subflow.

```

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      start-state="projectEdit">

  <persistence-context />
  <input name="projectId" value="requestScope.projectId" />

  <on-start>
    <evaluate expression="projectDao.findProject(projectId)"
      result="flowScope.project">
    </evaluate>
  </on-start>

  <view-state id="projectEdit">

    <transition on="save" to="save">
      <evaluate expression="projectDao.save(project)" />
    </transition>
    <transition on="cancel" to="cancel" />
  </view-state>

  <end-state id="save" view="externalRedirect:/projects" commit="true"/>
  <end-state id="cancel" view="externalRedirect:/projects" />

</flow>

```

The `persistence-context` element at the start lets us use the same entity manager for this flow. This means we can load, modify and save the entity in this flow and it will use the same entity manager so we don't have to worry about the entity becoming detached. The `input` element declares `projectId` as an input value. The `projectId` variable can be passed in from a parent flow, but if it is not, then the value is assigned from the request parameter called `projectId`. This means if we enter the project edit page via a URL as opposed to a parent flow, we can pass in the project Id as a parameter and it will be assigned to the `projectId` variable. This gives us two ways to enter a page but a singular way to get the `projectId` in the flow that we use to get the project to edit. The `on-start` element is evaluated when we first start the flow. Here, we call the `findProject()` method on the project Dao and pass in the `projectId` value. The result, which is the project we will be editing, is put into the `project` variable in the flow scope. In our project edit page, we can reference the `#{project}` variable and it will receive the value of our `project` variable defined in the web flow.

Example 2.7. projectEdit.xhtml page

```
<h:messages globalOnly="true" styleClass="message" />

<h:form>

  <h:panelGrid columns="2">
    Project Id : <h:outputText value="#{project.id}" />
    Title : <h:inputText value="#{project.title}" />
  </h:panelGrid>
  <h:commandButton action="save" value="Save" />
  <h:commandButton action="cancel" value="Cancel" />

</h:form>
```

Again, we have `save` and `cancel` actions which only have meaning if we look back at our flow. The `save` transition calls the expression `projectDao.save(project)` which is used to save the project to the database. It then transitions to the `save` end-state node which has a `commit` attribute set to `true` so the entity manager commits any changes. The view for the `end-state` is set to redirect to a view outside of the flow. The `cancel` transition is virtually identical except that it doesn't call `save` changes or `commit` changes at the end state. Note that we might run this flow from a parent flow, or this flow might run as a top level flow. While this is useful from the perspective of having re-usable code, it may cause problems in determining when the right time to commit the changes are. If this subflow is part of a larger process, we don't want to commit right away, but if it is also used as a process on its own, then there may be good reasons to perform the commit here.

If we go to the projects page, we can click on the edit link, and it will go to the project edit page. Clicking `cancel` or `save` returns us to the project list page. If we enter the URL `/projectEdit?projectId=2`, it will edit the second project. If we click `save` or `cancel`, the flow will end and we will be redirected back to the projects list page. One problem we do have is returning messages regarding the success or failure of our actions. The faces messages mechanism has been abstracted through Spring Web Flow, although it can still be used directly. Faces messages do not survive a redirect which both Seam and Spring work around, however Spring does not propagate faces messages outside of a flow since it uses a flash scope. Let's add a simple message handler by using a message writer bean which is defined as follows :

Example 2.8. MessageWriter.java helper class

```
public class MessageWriter {

    public void info(MessageContext ctx,String message) {
        ctx.addMessage(new MessageBuilder().info().defaultText(message).build());
    }

    public void error(MessageContext ctx,String message) {
        ctx.addMessage(new MessageBuilder().error().defaultText(message).build());
    }

    public void savedChanges(MessageContext ctx) {
        info(ctx, "Saved Changes");
    }

    public void cancelledChanges(MessageContext ctx) {
        info(ctx, "Changes cancelled");
    }
}
```

This is a really simple class to put messages into the flow. The MessageContext object that is passed in is the instance of the message context used by spring and is an abstraction of the faces message object. We add method calls to the flow to generate messages in response to certain actions.

Example 2.9. Adding expressions to generate messages on transitions.

```
<view-state id="projectEdit">

    <transition on="save" to="save">
        <evaluate expression="projectDao.save(project)" />
        <evaluate expression="messageWriter.savedChanges(messageContext)" />
    </transition>
    <transition on="cancel" to="cancel">
        <evaluate expression="messageWriter.cancelledChanges(messageContext)" />
    </transition>
</view-state>
```

When we save or cancel the project, the message for the user is pushed into the message context for display on the next page. However, as stated earlier, the messages do not survive the end of a flow which means that those messages will not be displayed if you enter the projectEdit page as a top level workflow.

Figure 2.2. Message after editing the project

Now let's add a flow for viewing the project, everything is almost the same, we use the same mechanism to get the projectId and load the project. The only real difference is that we have a transition to edit the project from the view page.

Example 2.10. projectView.xml web flow.

```
<input name="projectId" value="requestScope.projectId" />

<on-start>
  <evaluate expression="projectDao.findProject(projectId)"
    result="flowScope.project">
  </evaluate>
</on-start>

<view-state id="projectView">

  <transition on="close" to="close" />
  <transition on="edit" to="edit"/>
</view-state>

<subflow-state id="edit" subflow="projectEdit">
  <input name="projectId" value="project.id" />
  <transition on="save" to="projectView"/>
  <transition on="cancel" to="projectView"/>
</subflow-state>

<end-state id="close" view="externalRedirect:/projects" />
```

From the view page, we can call the edit subflow to invoke the same edit subflow which brings re-use to our flows. Again, we just need to pass in the `projectId` value like we did from the projects page. Also, since we invoke the editor as a subflow, when we return, we return to the view flow which means it will display any messages that were generated from the edit page.

Figure 2.3. Message displayed after cancelling changes

2.4. Listing issues

Now let's consider the project issues. We want to display the issues for each project on the view page in a kind of master detail fashion. For now, we will take the easy route and just connect the data table to the `#{project.issues}` value.

Example 2.11. Listing issues in projects.xhtml

```

<h:dataTable value="#{project.issues}" var="v_issue">
  <h:column>
    <h:outputText value="#{v_issue.id}" />
  </h:column>

  <h:column>
    <h:commandLink value="#{v_issue.title}" action="view" />
    (<h:commandLink value="edit" action="edit" />)
  </h:column>
</h:dataTable>

```

If we open this page as-is, we will get an error because of Lazy Initialization of the issues on the project object. The reason for this is because the entity is detached at the point of rendering the page. In order to fix this, we simply add the `persistence-context` element at the top of our project view flow. If we re-load the page, it works, however we have introduced another problem. If you click the edit button from the project view page, make changes and save it, you are returned to the project view page, the saved changes message appears but...the project that we edited has not changed. Click the close button and go back to the projects page. You should see the changes in the list of projects, so the problem was local to the project view page.

The problem is that we used the same persistence context in the view flow. The project entity was loaded when we first went to the project view page. We edited the project, and returned to the project view page. When we started the view flow, we searched for the project entity based on the project Id and put it into `flowScope`. When we return from the project edit page, that value is still in flow scope and will be re-used. To get around this, we would need to refresh the project entity if the return value from the editor subflow is `save`.

Example 2.12. Refreshing the entity in a flow.

```

<subflow-state id="edit" subflow="projectEdit">
  <input name="projectId" value="project.id" />

  <transition on="save" to="projectView">
    <evaluate expression="projectDao.refresh(project)" />
  </transition>

  <transition on="cancel" to="projectView" />
</subflow-state>

```

This isn't ideal, since you also have problems with regards to refreshing the page. If you display the project view page while changing the project data in either another window or directly in the database, the same values are displayed in the refreshed project view page. To me, this is wrong as in general, in an entity view, each refresh should literally refresh the data. I'm sure there is a way around this using web flow that I just haven't come across. In theory, I should be able to put the evaluation expression in an `on-render` tag in the project view page so the item is refreshed on each page refresh. However, when I tried this, I got errors since web flow had problems with knowing about or seeing the `projectId` value.

2.5. Handling Issues

Now let's look at the view and edit pages for the issues. They are pretty much the same as the view and edit pages for the projects, except the names of parameters and variables, so I won't reproduce the code here.

However, we do need to consider one issue. If we are to display a list of issues for a project in the project view page and allow the selection of that project, we need to wrap it in a `dataModel`, which isn't done currently since we are using `#{project.issues}` to get the list of issues. We have three choices here. The first is to simply use a GET request passing the `issueId` as a parameter. However, request would take us out of any workflows we might be in. Alternatively, we could wrap the current list of issues in a data model.

Example 2.13. Putting the list of issues into a flow as a `dataModel`.

```
<evaluate expression="project.issues"
  result="flowScope.issues" result-type="dataModel">
```

Adding this to the start of the `projectView` flow wraps the list of issues from the project attribute in a data model. Alternatively, we can create a separate function which will allow us to get a list of issues independently from the project entity. This is probably a better solution since a project may end up with thousands of issues which we will want to paginate without having to load them all. However, using this solution changes a number of things for us as we will see. First off, we'll implement an bean that has an `IssueDao` interface.

Example 2.14. `IssueDao` interface.

```
public interface IssueDao {
    public List<Issue> findIssuesForProject(Long projectId);
    public Issue findIssue(Long issueId);
    public void refresh(Issue issue);
    public void save(Issue issue);
}
```

I won't include the implementation since it is fairly straightforward, but we call it `IssueDaoBean` and add it as a spring bean called `IssueDao`.

At the top of our `projectView` flow, we add a call to find the issues for a project and put the results into a flow scoped data model called `issues`.

Example 2.15. Putting the issues into a flow based on query results.

```
<evaluate expression="issueDao.findIssuesForProject(projectId)"
  result="flowScope.issues" result-type="dataModel">
</evaluate>
```

In our web page, where the `dataTable` was using `#{project.issues}` , we change it to just `#{issues}` .

Example 2.16. Using our new `issues` variable in the data table.

```
<h:dataTable value="#{issues}" var="v_issue">
```

Now that we independently fetch the list of issues, we can technically remove the `persistence-context` element. If you recall, we added this because we wanted to fetch the list of issues in the rendering phase and didn't want to cause a Lazy Initialization Exception. Since we know we have already fetched the data we need at the start of the flow, we no longer need it. However, if we need to display the status entity of the issue, we would need the same PC to be available on the rendering phase assuming the status is fetched lazily. To solve this, we have two options. Either keep the persistence context for the duration of the flow, or explicitly load the status entities when we get the list of issues in the query which is a better option. However, for now, let's keep the persistence context for the duration of the flow. There are issues associated with unnecessarily keeping a persistence context which we'll see in the next section.

2.6. Managing Persistence Contexts

In Spring Web Flow, Persistence Contexts (PCs) can last the duration of the flow, or for the duration of the request. It is possible to have a PC scoped to last for a conversation but this is not yet implemented. Whether you use the long or short PC scopes, they both introduce different problems. Note that some of these same problems apply to Seam also where a PC can be conversational or non-conversational depending on whether a long running conversation is active or not.

2.6.1. PC Scope Issues

Short PC scopes can result in detached entities that need re-attaching when persisting changes. The problem with this is that you lose optimistic locking since the version you started out with could be different than the version you ultimately save with if another user has modified the entity.

While it may seem tempting to always use a longer PC scope, such as one scoped to the flow, there are different problems associated with it. For example, in a view page with a flow scoped PC, when you refresh the page, an attempt is made to refresh the data you are viewing. However, because you are using a flow scoped PC, the same instance of the data is returned from the PC. This means that you would manually need to trigger a refresh of the data by calling the refresh method on the persistence context.

2.6.2. Practical PC Management

So, now we have our issue edit/view pages up and running, let's look through a couple of scenarios that we might encounter. We can remove the `persistence-context` element from the issue view flow since we don't need to keep our PC around. If we try and edit the issue and save it, when we get back to the `issueView` flow, we run into a problem. When the outcome was `save` , we refreshed the `issue` entity to get the latest version using the variable in the flow.

Example 2.17. Refreshing the entity on a save action.

```
<subflow-state id="edit" subflow="issueEdit">
  <input name="issueId" value="issue.id" />
  <transition on="save" to="issueView">
    <evaluate expression="issueDao.refresh(issue)"/>
  </transition>
  <transition on="cancel" to="issueView" />
</subflow-state>
```

Because we are not using the same persistence context for the duration of the flow, we get an 'Entity not managed' exception because we are refreshing the entity against a different PC. We can solve this a few ways. One is to change the call to refresh to a call to find the issue.

Example 2.18. Refreshing the issue by re-loading it as a new instance.

```
<evaluate expression="issueDao.findIssue(issue.id)" result="flowScope.issue">
```

This calls the find method using the current PC and puts the value in the `issue` variable in the flow scope. Another way is to write a smarter refresh method in the Dao. We can't just call `find()` since we will get the stale version back, so we check if it is in the PC and if so, we refresh it. Otherwise we call the `find()` method to get an instance back.

Example 2.19. Refreshing / loading the entity.

```
public Issue refresh(Issue issue) {
    if (entityManager.contains(issue)) {
        entityManager.refresh(issue);
        return issue;
    } else {
        return findIssue(issue.getId());
    }
}
```

This way, if the issue is not part of the current PC, then it loads a new one using the ID of the old one. If it is a part of the PC, then it simply refreshes it. The returned value is pushed into the flow scope `issue` variable.

Example 2.20. Refreshing the entity and putting it into the flow

```
<evaluate expression="issueDao.refresh(issue)" result="flowScope.issue">
```

The advantage of this method is that it will work whether you are using flow scoped persistence contexts or not and you can change between the two without requiring any additional changes (toggling between the `find()` and `refresh()` methods on the `issueDao`).

Incidentally, this kind of feature can also be used in the `projectView` page where we list the issues. When the user views or edits an issue we need to refresh the issue in the list on the project page since it may have changed (the user may click view issue, then edit the issue, and then end up back to the project view page). To ensure that we refresh the issue, we can add an `on-exit` element to the `issueView` and `issueEdit` states so that we can refresh the item in the issue list.

Example 2.21. Flow controlled refresh of the entity on the on-exit phase in projectView flow.

```
<subflow-state id="issueView" subflow="issueView">
  <input name="issueId" value="issues.selectedRow.id" />
  <transition on="save" to="projectView" />
  <transition on="close" to="projectView" />
  <on-exit>
    <evaluate
      expression="issueDao.refresh(issues.selectedRow)">
    </evaluate>
  </on-exit>
</subflow-state>
```

This works great, when the user views an issue and we come back to the project view flow, just that one selected issue is refreshed.

One nice aspect of Spring Web Flow is the amount of control you can have over the data in your flows, the scope of it, and the scope of the persistence contexts. Also, everything is local to the flow itself, the data is declared in the flow, and inserted into a scope according to the flow definition. This means your flows can be strongly decoupled from each other, and also from your code since most code is stateless. State is handled by declaring stateful variables in the flow itself.

I can imagine situations where this will result in problems though if careful variable naming is not used. The fact that variables can be declared as flow scoped as opposed to conversationally scoped will reduce these incidents if users can limit the scope to the flow

2.7. Multi-Page Flows

Let's look at one more scenario that is a little more complex. Let's say that as part of editing an issue we want to select a different project for the issue. For this, we can re-use the project list page, but this time the list will have a button to select the project for the issue we are editing. When a project is selected, we should come straight back to the issue editing page, and the project should be selected as the issue's project. It is feasible that such a listing page would be used in a number of places throughout the application making the re-use of this page important. We don't want any hard coding or hacks in the page to indicate where to navigate to next, we want to use the page flow functions to control our navigation. The other aspect of this problem is that we need to pass the selected project back to the issue edit flow. To start with, we'll add a `selectProject` transition to our `issueEdit` flow.

Example 2.22. Transition to navigate to the project selection page.

```

<view-state id="issueEdit">
...
...
<transition on="selectProject" to="selectProject" />
...
...
</view-state>

```

This transitions to our select project sub flow which is the `projects` flow we created at the beginning. Again, the goal here is to re-use elements we have already built. This flow state is also responsible for any changes to our issue entity due to the project selection.

Example 2.23. The project selection state in the edit issue flow.

```

<subflow-state id="selectProject" subflow="projects">
  <input name="doSelect" value="true" type="java.lang.Boolean" />
  <transition on="selectProject" to="issueEdit">
    <set name="issue.project"
      value="currentEvent.selectedProject">
    </set>
  </transition>
  <transition on="selectCancel" to="issueEdit" />
</subflow-state>

```

Caution

As of Web Flow 2.0.1, `currentEvent.selectedProject` should be replaced by `currentEvent.attributes.selectedProject` due to changes in how SWF accesses the attributes in the current event.

What we are doing is calling the `projects` flow and pushing a variable called `doSelect` in and setting it to `true`. If the subflow returns `selectProject` then we expect the subflow to pass out a value called `selectedProject` which we assign to the `issue.project` value.

In our `projects.xhtml` page, we add a new column to the list of projects to contain the button to select a project. This column should only be rendered if the `doSelect` flag is set to `true`. This flag is only set to `true` when we are calling the flow from another flow for the purpose of project selection.

Example 2.24. `projects.xhtml` column containing a link for selecting a project

```

<h:column rendered="#{doSelect == true}">
  <f:facet name="header">Select</f:facet>
  <h:commandLink value="Select" action="selectProject" />
</h:column>

```

At the bottom of the table we add another button to cancel the selection if the user decides not to change the project. Again, rendering this button depends on the `doSelect` flag being set.

Example 2.25. `projects.xhtml` button for cancelling project selection.

```
<h:commandButton rendered="#{doSelect == true}" value="Cancel" action="selectCancel"/>
```

The select and cancel buttons returns the actions `selectProject` and `selectCancel` respectively. These actions are handled by the `projects` page flow. As a part of selection process, we need to pass the selected project out of the flow and into the parent flow. We do this using a new `end-state` which outputs the selected row. We also have an end state for the cancel action which does not output the selected project.

Example 2.26. End states in `projects.xml` flow for selecting or cancelling project selection.

```
<end-state id="selectProject">
  <output name="selectedProject" value="projects.selectedRow" />
</end-state>
<end-state id="selectCancel" />
```

If the user cancels the selection, then we just go to the end state, but if they click select, then we pass back the `projects.selectedRow` value in a variable called `selectedProject`.

What is really great about this is I can start any number of flows and they will manage themselves correctly. I can navigate the pages in any order by editing and viewing projects or issues, changing the project for an issue but editing the project before selecting it. As I save and close my way back down the flow stack everything works perfectly. My data is isolated between different conversation and I don't worry about overwriting values.

2.8. Lookup Management

Unfortunately, there is no provision for using a drop down in Jsf without resorting to manually handling `selectItems` and taking responsibility for mapping the selected item to a scoped list within the flow.

2.9. Creating Entities

Let's look at the task of creating a new project and a new issue. For this, we need to determine that the project or issue Id is blank, and thus create a new one. I found this to be a somewhat thorny problem to a degree. The most obvious way was to use the `findProject(projectId)` to return a project, or a new project if `projectId` is null or 0. Note that this is a quick solution and you may want to use an alternative mechanism for production. Especially since security authorization will be involved in most cases to determine whether the user can view or create the item in question.

Example 2.27. Code in projectDao to find or create a project.

```
public Project find(Long projectId) {
    if (projectId == null || projectId == 0) {
        return new Project();
    } else {
        return entityManager.find(Project.class, projectId);
    }
}
```

Now we just add a "new" button on the project list and make it call the sub flow for editing the project

Example 2.28. Transition in projects.xml for creating a new project.

```
<view-state id="projects">
...
...
...
    <transition on="new" to="projectNew" />
...
...
...
</view-state>

<subflow-state id="projectNew" subflow="projectEdit">
    <transition on="cancel" to="projects" />
    <transition on="save" to="projects" />
</subflow-state>
```

Note that we do not pass a projectId in this time. If we run this, we can click new, edit a new project and save it. Note also that this fits in with our existing project edit workflow without needing any changes.

I can view an existing project, edit one of the issues, and decide to change the project. Rather than pick an existing project, I can create a new project, save it, and then select it as the project for the issue I'm editing. Also, I could cancel the changes to my issue and the project will still be saved. The beauty of it is that since the flows are self contained, data is isolated and I don't have to worry about variable names overlapping or the flows becoming intertwined. It just works without any changes to the other flows.

For issues, we have a tougher challenge. If the issue Id is null, we cannot just create a new issue, we need to have a project to attach it to. We add a new method to the issueDao called findOrCreateIssue into which we pass the issueId and any projectId we have.

Example 2.29. `issueDaoBean.java` method to find or create an issue.

```
public Issue findOrCreateIssue(Long issueId, Long projectId) {
    if (issueId == null || issueId == 0) {
        if (projectId == null || projectId == 0) {
            //error!
            return null;
        } else {
            Issue issue = new Issue();
            issue.setProject(entityManager.find(Project.class, projectId));
            return issue;
        }
    } else {
        return findIssue(issueId);
    }
}
```

This should take care of creating a new issue or project when we don't pass an Id in. There is the question of security and exception handling which I'll get to later.

For now, let's continue with the problem of adding issues. In the project view page, we add a button to add a new issue to the project.

Example 2.30. Button in `projectView.xhtml` to create a new issue.

```
<h:commandButton action="issueNew" value="New Issue" />
```

In the `projectview` flow, we add a transition to the view for creating new issues.

Example 2.31. Transition in `projectView.xml` flow to create a new issue.

```
<transition on="issueNew" to="issueNew" />
```

This goes to a subflow for editing issues. As part of this subflow, we pass in the value `project.id` as the ID of the project we want the new issue for.

Example 2.32. Subflow state for creating a new issue for the project

```
<subflow-state id="issueNew" subflow="issueEdit">
  <input name="projectId" value="project.id" />
  <transition on="save" to="projectView" />
  <transition on="cancel" to="projectView" />
  <on-exit>
    <evaluate
      expression="issueDao.findIssuesForProject(project.id)"
      result="flowScope.issues" result-type="dataModel">
    </evaluate>
  </on-exit>
</subflow-state>
```

When we return and exit that subflow, we refresh the list of issues for the project. Again, the level of decoupling between the two flows is excellent, we pass in what we need to and we check for the responses we expect.

At this point, it looks like we've completed the demo application. While the example is not that complex, it should give you an idea of how to write a stateful CRUD application with Spring and Spring Web Flow.

2.10. Exception Handling

In our application, when an error is reached, we should be throwing an exception which should then be caught and our flow redirected, and/or an error message displayed. In particular I'm thinking of the cases where we want to load an issue or project, and there is no Id, or the item for that Id does not exist anymore. There may also be cases where the user may be creating a new item when they don't have security rights to do so. There exists mechanisms for handling exceptions which lets us write exception handling classes to handle the exception and if possible add messages to be displayed on the current view.

However, we call these functions on the start of the flow at which point, the view state is not available to the flow, and the faces context is not available, therefore if we do find an exception, there isn't much we can do about it. We could make our `findOrCreateIssue()` call in the `on-render` phase of a state, but we have problems there since it cannot see the `projectId` or `issueId` values properly. It seems like these values have a very short scope.

Overall, the exception handling seems limited (almost non-existent) without writing custom exception handlers for the view. Again, this is done through the bean mechanism so there is plenty of chance to write handlers that uses lists of re-usable handlers to process exceptions. While this does offer a flexible system, it would be nice to see some default in there for error handling, even if it is just staying on the page, and pushing the exception message into the `messageContext`. I think though that there are more complications involved in that though. The help documents and examples don't cover exception handling very well, but I'm sure we'll see more on the topic from Spring.

2.11. Extras

Spring Web Flow does come with a number of ajax related JSF components that can allow you to limit the areas that are re-rendered on submission which is quite nice. It also provides some decorations for existing

DOM nodes that can apply client side validation which is also nice. JSF has a number of existing Ajax frameworks out there that can achieve some of this, however they may be considered heavyweight and loaded with unneeded components, so Spring's Ajax controls offer a lighter alternative if you wanted to apply a simple ajax solution to your application.

Spring Security can also be used with flows and you can apply authentication at the flow, view and even transition levels.

Flows also have an inheritance mechanism which allows you to let one flow inherit from another. While the inheritance is a little less flexible than object inheritance, this is still a great feature for some of those views that are often used. This could easily make up for some of those cases where the data factory methods needs to be repeated in similar flows. For example, the project view and edit flows both need to grab an instance of an object based on the project Id parameter.

IDE support is missing in some areas, although mostly in the non-Spring areas. The JSF page editor is missing the ability to auto complete code for beans, and there are places where I expected auto-completion in the flow editors, but didn't get it. The web flow editor in this version was still thinking in terms of Web Flow 1.0, although it was still able to come up with diagrams for my flows. Auto completion for the Spring elements worked great, from specifying beans to writing flows and auto-completing the list of states available to transition to. For me, not having auto complete in the JSF editor is a pain.

Spring Web Flow offers some flexibility with the Persistence Context. The `Persistence-Context` element at the start of flows lets us define whether the PC is event, flow or conversation scoped. Conversation scoped is not currently implemented. This could cause some problems if you are pushing entities from one flow to another since they could have different PCs and the entity would be detached in the new flow.

Overall, this is a great framework. Version 2.0 included a heavy re-write to make it work much more amicably with JSF and it shows. The flow language is clean and straightforward, although it does tend towards some repetition. Having default transitions might be a nice addition to both ease repetition and also avert disaster in the event of changes to a subflow which returns an unexpected value to a parent flow. It is great how easily the flows can be written so you can take a navigation path that is an endless circle, and still come back out the way you came in without data getting overwritten. The only other criticism is that it does make a mess of your URLs by adding large flow state values in there. As a newly re-written framework, there are plenty of places where the documentation is lacking, especially for newcomers.

The data management and scoping is really nice, the localization of the definition to the flows is great, even if it is at the expense of defining things multiple times for similar flows. The IoC and Dependency Injection pieces of this solution probably need no introduction as they are provided by the Spring core.

Chapter 3. The Seam Application

3.1. Initializing the project

The Seam application is being developed using a clean eclipse and the latest JBoss Tools plugins. Getting started with the project is a breeze. We start by defining the database connection and the JBoss app server to use in our project wizard. We start the project wizard which asks questions about the project, any datasources, deployment options, and server runtimes and it creates everything you need to create a new project.

Figure 3.1. Configuring the project Wizard

Figure 3.2. Configuring Seam and the DB Connection

Since I chose the EAR deployment over the WAR deployment, we end up with a set of projects:

- The WAR project
- The EJB project
- The EAR Project that it is all wrapped up in
- The Test Project

We have no jars to copy over, and no configuration files to set up, the project is ready to deploy and run. JBoss have done really well smoothing this piece out and letting users get up and running quickly.

3.2. Seam State Management

In Seam, state management works a little differently. We don't need to put every page into a work flow just to manage state. With Seam, state management is always present in the form of conversations. Conversations can either be default or long running. The default conversation lasts just for this page and into the next page. A long running conversation lasts until you explicitly end it. All pages run in a conversation whether it is the default temporary conversation or whether it is a long running conversation. While it may seem intrusive, it is rarely noticeable and is actually a good thing as we will see later.

Once we start a long running conversation, any conversationally scoped data that is available in the conversation at the point the conversation started, as well as any conversational scoped data we add to the conversation, will remain in that conversation until it ends.

Conversations can be started in a number of ways, by calling methods on Seam components annotated with a `@begin`, or by using page metadata in `pages.xml` or from a link using the `s:link` or `s:button` Seam JSF components which have a `propagation` value set to `true`.

First off, we'll be using the same model as the Spring sample project. Regarding the Dao pattern of layering, Seam offers a few different ways of structuring your application. Seam tends towards using 'thicker'

stateful objects, objects that hold references to stateful data as well as providing action methods on those objects.

Seam also comes with classes for the "Seam Application Framework" which is a built-in framework for creating simple persistence and query components. We'll cover both options here since it really is almost two separate features. There's also the question of whether we use EJB or POJOs since Seam supports both out of the box. The draw back to this is that even if you use POJOs, you still require an EJB 3.0 environment to run it, even if it is the embedded EJB environment in a non-EJB container.

Seam works on a pull model rather than a push model. When a context variable is requested, Seam searches for the variable in the available contexts such as request scope, page scope, the conversational scope, and then session and application scopes. If it does not find it, it looks into its internal metadata to see if a factory method is registered for the variable name. If so, the factory method is called, and the variable is put into whatever scope the factory method or outjection specified.

3.3. Getting Started

We'll start with our list of projects, for which we will create a `ProjectListActionBean` which will be an EJB. This bean will use a query for projects, and will be used for other functions later on. First we define an interface for the bean.

Example 3.1. EJB Local interface for our bean

```
@Local
public interface ProjectListAction {

    List<Project> getProjects();
    void remove();

}
```

We implement this interface in the class `ProjectListBean` and define it as a Seam component called `projectList`.

Example 3.2. Implementation of our stateful EJB

```
@Name("projectList")
@Scope(ScopeType.CONVERSATION)
@Stateful
public class ProjectListBean implements ProjectList, Serializable {

    @Logger
    private Log log;

    @In
    private EntityManager entityManager;

    private List<Project> projects;

    @SuppressWarnings("unchecked")
    @Factory("projects")
    public List<Project> getProjects() {
        if (projects == null) {
            log.debug("Getting projects");
            projects = entityManager.createQuery("select p from Project p")
                .getResultList();
        }
        return projects;
    }

    @Remove @Destroy
    public void remove() {

    }
}
```

Here we declare our `ProjectListBean` class to be a Seam component called `projectList` and it is declared as a stateful EJB. The `@Name` annotation is used to specify the name of the component to Seam and the `@Stateful` annotation is used to indicate that it is a stateful EJB. The `@Factory` annotation on the `getProjects()` method is used to declare a factory method in Seam. If a variable called `projects` is requested and not found, Seam would call this method and the results will be put into a context variable called `projects`. The scope of the results depends on the scope specified in the factory annotation if there is one. If not, it defaults to the scope of the component that contains the factory method (in this case, conversational scope).

The entity manager is a special seam component that is injected like other seam components. The `@Logger` annotated member is a Seam logger which is a standard logger that is automatically injected without the explicit code to obtain a logger per class type. It also wraps calls to the actual logger and lets you use EL syntax within the logged message such as "Saving project #{project.title}" and the values will be substituted. It is these touches throughout Seam that make it a very cohesive framework.

For the projects page, we can use the same HTML from the Spring application.

Example 3.3. JSF page for displaying the projects

```

<h:dataTable value="#{projects}" var="v_project">
  <h:column>
    <f:facet name="header">ID</f:facet>
    <h:outputText value="#{v_project.id}" />
  </h:column>

  <h:column>
    <f:facet name="header">Title</f:facet>
    <s:link value="#{v_project.title}" view="/projectView"
      propagation="none">
      <f:param value="#{v_project.id}" name="projectId" />
    </s:link>
  </h:column>

  <h:column>
    <s:link value="Edit" view="/projectEdit" propagation="none">
      <f:param value="#{v_project.id}" name="projectId" />
    </s:link>
  </h:column>
</h:dataTable>

```

When we use `s:link` or `s:button` Seam JSF controls, we are using controls that use GET instead of POST. They also provide us with conversation management attributes which is what the `propagate` attribute is. Since each page runs in a conversation, even if it is a short lived one, there is a chance that the next page will re-use the same conversation as the last one. By setting `propagation` to `none`, we specify that the conversation is not shared between the two pages, and a new one is required for the new page. Also, as this was typed out the code complete features in the page editor and the WYSIWYG editor enabled me to quickly pick my backing bean names.

Figure 3.3. Seam Code Assist

Also note that we are able to use code completion on JSF tags, so we just enter `"h:output"` it will suggest a number of options. If we select the `outputText` element, it will add a value attribute to put the text value into. This value attribute also has auto completion, just like almost every other attribute and JSF element. The Exadel and JBoss Tools team have done an exceptional job with these tools.

If we start the server and go to our page, we will see a list of our projects. If you make a slight change to the page and refresh the page, it will take a couple of seconds for the change to be published to the server. It is an annoyance, but it is tempered slightly by the fact that you can do a preview in the IDE so you don't have to launch it in the browser to see every mistake you've made. For some reason, this delay seems longer on my machine than others I have seen.

Figure 3.4. List of projects in Seam

3.4. Editor Pages

You will notice that to list the projects, we haven't used any kind of flows yet. As we discussed earlier, every page runs in a conversation even if it is one that only lasts from this page to the next. Furthermore, we haven't declared any DataModels yet. For now, we can use page parameters to pass the `projectId` over to the edit page. To make this easier, we use some Seam provided JSF controls. The `s:link` and `s:button` components provide us with a link and button component that we can use to call pages, and pass parameters easily. These requests are done as GET requests as opposed to a POST that JSF would normally use.

In our JSF page, we define the column for the edit link using a Seam `s:link` component.

Example 3.4. Column for the edit link in `projects.xhtml`

```
<h:column>
  <s:link value="Edit" view="/projectEdit" propagation="none">
    <f:param value="#{v_project.id}" name="projectId" />
  </s:link>
</h:column>
```

When rendered in a page, this becomes `/projectEdit.seam?projectId=3` and acts as a GET request into `projectEdit.xhtml`. The reason we have the `propagation="none"` attribute is because by default, the `s:link` and `s:button` components propagate the conversation to the next page. In this case, we don't really want to. We want the edit page to grab a fresh instance of the project we are about to edit. We use the same attribute in the view link in the project title column.

Now lets look at the view and edit page, starting with the view page. We can just whip one up with a simple panel grid.

Example 3.5. Simple JSF project viewer

```
<h:panelGrid columns="2">
  Project Id : <h:outputText value="#{project.id}" />
  Title : <h:outputText value="#{project.title}" />
</h:panelGrid>
```

Now lets consider the problems we face to generate the variable `project`. We need to take the parameter, and load the project based on the parameter and push it out as a scoped variable called `project`.

We'll do this the most direct way for now to give you an idea of how straightforward dealing with Seam components can be.

We'll create a new class called `ProjectHomeBean` which we will use to handle the parameter and the instance of the project. First we define a local interface for it called `ProjectHome`.

Example 3.6. ProjectHome interface for handling project instances.

```
@Local
public interface ProjectHome {

    Project getProject();
    void setProject(Project project);
    Long getProjectId();
    void setProjectId(Long id);
    public void remove();
    String save();
    String cancel();

}
```

Next, we create a class called `ProjectHome` which implements this interface. If we wanted to make this a javabean POJO, we could omit the interface and the stateless annotation and Seam would use this bean the same. You would however have to add transaction annotations similar to the Spring bean ones.

```
@Stateful
public class ProjectHomeBean implements ProjectHome {
```

```
    @In                                The Seam Application
    private EntityManager entityManager;
```

Example 3.7. ProjectHome implementation

```
private Logger log;

private Long projectId;

private Project project;

private boolean hasProjectId() {
    return (projectId != null && projectId != 0);
}

    @Factory("project")
public Project getProject() {
    if (project == null) {
        if (hasProjectId()) {
            project = entityManager.find(Project.class, projectId);
        } else {
            project = new Project();
        }
    }
    return project;
}

public Long getProjectId() {
    return projectId;
}

public String save() {
    entityManager.persist(project);
    entityManager.flush();
    FacesMessages.instance().add(
        "Saved changes to project #{project.title}");
    return "save";
}

public void setProject(Project project) {
    this.project = project;
}

public void setProjectId(Long id) {
    this.projectId = id;
}

public String cancel() {
    entityManager.refresh(project);
    return "cancel";
}

    @Remove @Destroy
public void remove() {
}
}
```

Notice that we added a `@Factory` annotation onto the `getProject()` method. This annotation marks this method as the one to call anytime we need a variable called `project`.

When we save the project, we also add a `FacesMessage` in code. Notice how we are able to use the EL expression `#{project.title}` in the message. We'll add a message for the cancel method an alternate way.

In order to load the project we need to somehow bind the `projectId` parameter with the `projectId` member in this component. We can do this using `pages.xml`. `pages.xml` is one (or more) xml files for providing additional meta data about each page to Seam. When you create an application using the Seam-Gen tools or JBDS, Seam creates an instance of this file with the default exception handling information included to which you can add your page information. Alternatively, you can have multiple files that deal with logical sections of your application. The information contained can related to page parameters, conversations and page flows that start or end on a given page, page navigation information for a given page, actions to execute when a certain page is requested, and even whether a login is required for access to certain pages which can be defined using wildcards.

Example 3.8. Project view page parameter configuration from `pages.xml`

```
<page view-id="/projectView.xhtml">
    <param value="#{projectHome.projectId}" name="projectId"/>
</page>
```

This tells Seam that every time it renders this page, it sets the value of `#{projectHome.projectId}` to the value of the `projectId` parameter. This binding is two way, in that when we move to the next page, we put the value of `#{projectHome.projectId}` in a parameter called `projectId` and send it to the next page.

If we load our project view page, we can see the project is loaded correctly. If we click the browser refresh, we can see from the log that the page gets a fresh instance of the project entity which is good. We refresh the page to see the latest instance, not just to be sent the same output. Now let's consider the buttons to navigate to the edit page, or back to the project list.

Example 3.9. JSF for the back and edit buttons in `projectView.xhtml`

```
<h:panelGrid columns="2">
    <s:button value="Back" view="/projects.xhtml" />
    <s:button value="Edit" view="/projectEdit.xhtml">
        <f:param name="projectId" value="#{project.id}" />
    </s:button>
</h:panelGrid>
```

Here, we explicitly define the views we are going to with any parameters needed. This is one way of handling navigation. Because conversations are omnipresent, we don't have to rely on conversation mechanisms for passing data around, we can go back to page parameters. This is also beneficial because we can easily create loosely coupled pages. The edit page itself is fairly simple and is almost identical to the view page.

Example 3.10. projectEdit.xhtml page.

```
<h:panelGrid columns="2">

    Project Id : <h:outputText value="#{project.id}" />
    Title : <h:inputText value="#{project.title}" />

</h:panelGrid>
```

The next question to ask is where does the `project` variable come from this time? The answer is simple, we just re-used the same code we used for the viewer. When the page is loaded and JSF looks for a variable called `project`, Seam will use the same factory method to create the project using the `projectId` parameter. We can deploy this page and run it right away without any further coding. This is one of the benefits of Seam variables being defined application wide.

Figure 3.5. Editing a project with Seam

We can also access this page directly with a url using parameters such as `/projectEdit.seam?projectId=2`.

3.5. Handling Navigation

Let's consider the save and cancel buttons and how we act on them, and navigate to our next page based on the button clicked. We define our two buttons with actions calling the methods on the backing beans.

Example 3.11. The save and cancel buttons in projectEdit.xhtml

```
<h:panelGrid columns="2">
<h:commandButton value="Save" action="#{projectHome.save}" />
  <h:commandButton value="Cancel" action="#{projectHome.cancel}" />
</h:panelGrid>
```

Our save button calls the `projectHome.save()` method which saves the project. The cancel button calls the cancel method which simply refreshes the value of the project we are editing. These methods return strings for the navigation handlers to process.

Notice that we switched from using `s:button` to `h:commandButton`. Remember `s:button` and `s:link` perform a GET request without the POST. If we used the seam components we wouldn't post our changes back to the server. Technically, we could use a `s:button` for the cancel button since we don't care whether the changes are posted back. Notice that we use the `immediate="true"` attribute for the command button since this bypasses the validation steps which we don't care about since we are cancelling changes.

Normally, in JSF, navigation is handled using the JSF navigation rules. Seam offers us multiple additional ways to handle navigation. The first is in the `JSF faces-config.xml` which I've not used for a while. Next we have `pages.xml` which is Seam specific. We can simply add on the navigation rules to our page information in `pages.xml`

Example 3.12. Navigation using pages.xml

```
<page view-id="/projectEdit.xhtml">
  <begin-conversation join="true" flush-mode="manual"/>
  <param value="#{projectHome.projectId}" name="projectId"/>
  <navigation>
    <rule if-outcome="save">
      <end-conversation/>
      <redirect view-id="/projectView.xhtml">
        <param name="projectId" value="#{project.id}"/>
      </redirect>
    </rule>

    <rule if-outcome="cancel">
      <end-conversation/>
      <redirect view-id="/projectView.xhtml">
        <message>Cancelled Changes to project #{project.title}</message>
        <param name="projectId" value="#{project.id}"/>
      </redirect>
    </rule>

  </navigation>
</page>
```

This page definition does a number of things. When we enter this page, we want to start a conversation. Since we specified the `join="true"` attribute, then if one already exists we join that existing conversation.

We then have some navigation rules, one for `save` and the other for `cancel` which were the two outcomes from the `save` and `cancel` methods called from our `projectEdit.xhtml` page. If the user saves the project, `save` is returned to the navigation handler so we end the conversation and redirect to the view page passing in the project Id as a parameter. If they cancel the changes, the `cancel` method is called which returns `cancel`. Navigationally, the same thing happens except we also pass a message along with it. This message is rendered on the page that we are redirected to. Notice how we were able to include the EL expression `#{project.title}` into the message to give the name of the project we were editing. The only reason the message is here is to demonstrate the number of different ways we can do simple things with Seam.

If we run this, edit a project, and cancel the changes we get the message appearing once we have cancelled the changes.

Figure 3.6. Propagating cancelled messages to the users

We came back to the view page with a message to display. If we edit the project and save the changes, the message is added from the `saveProject()` method in our bean.

Figure 3.7. Propagating save messages to the user

Before we move on, let's consider one more scenario, the adding of a new project to the list. We do this by navigating to the `projectEdit` page with no `projectId` parameter. By default, we create a new project in our `project` factory method if no parameter is passed in to the bean.

We can add the create button on our projects page.

Example 3.13. The add project button on the `project.xhtml` page.

```
<s:button value="New Project" view="/projectEdit.xhtml" propagation="none"/>
```

This was pretty straight forward, and our `projectHome` component has handled the cases where we want to view, edit or create a new project instance giving us quite a bit of code re-use without any additional code or configuration needed. All we need to do is request the `project` variable and Seam will provide it for us. One real benefit here is that we can move the factory method from one bean to another and Seam will automatically use that bean instead. We have created a de-coupling between our view and the source from which we obtain data. To change the source of our `project` entity in spring would require us changing all the spring web flows that called that one method to generate the project.

Figure 3.8. Create new project

Figure 3.9. Save new project

Figure 3.10. View new project in list

3.6. The Seam Application Framework

This section introduces a couple of generic components which can be used to build typical CRUD querying and entity handling functions. Note that these components make up the Seam Application Framework, which is just a small (and optional) part of the Seam framework itself. These are really just helper classes that make it really easy to build CRUD and query components.

Seam typically blurs the lines on typical application layering. Our Dao layer has become a couple of beans which hold state, and are themselves stateful. You can choose to create a Dao or service layer and inject it into beans that manage state, but it is somewhat unnecessary. Our data is decoupled from our view through the use of EL. Some people might be wondering where the application layers are and having fits at the notion of having a single bean handle view events and data access. There are good reasons why it makes sense to flatten things down since the flatter version handles 95% of situations, and it is simple to refactor out these pieces if you need to re-use them or access them as separate layers. Also, since the flatter version requires almost no code, it raises the question of whether code that is never written can be shareable?

Seam embraces this notion of thicker stateful beans wholly to the extent of providing generic versions of beans called `EntityHome` and `EntityQuery`. These components make up the Seam Application Framework and are used to provide simple and easy classes to fetch, create and persist entities and to query for entity instances respectively. First, we'll take a look at the `EntityQuery` as we determine how we

are going to display the list of issues for a project. To display the list of issues, we add a data table that connects to our list of issues.

Example 3.14. Issues data table

```
<h:dataTable value="#{issues}" var="v_issue">
  <h:column>
    <h:outputText value="#{v_issue.id}" />
  </h:column>

  <h:column>
    <s:link value="#{v_issue.title}" view="/issueView.xhtml">
      <f:param value="#{v_issue.id}" name="issueId" />
    </s:link>
  </h:column>

  <h:column>
    <s:link value="edit" view="/issueEdit.xhtml">
      <f:param value="#{v_issue.id}" name="issueId" />
    </s:link>
  </h:column>
</h:dataTable>
```

Now we just need to determine how we obtain the value of `issues` one easy way would be to create a factory method in `components.xml` which is the configuration file used to define Seam components in XML instead of annotations. We can also use this file to specify factory methods which we do here for the value `issues`.

Example 3.15. Define `issues` factory in `components.xml`

```
<factory name="issues" value="#{project.issues}" scope="conversation"/>
```

When we open the `projectView` page, since `issues` is not defined, Seam evaluates the expression `#{project.issues}` and puts the results into a Seam context variable called `issues` in the chosen (or default) scope. Note that we don't have any problems regarding Lazy Initialization Exceptions, because Seam gives us two transactions during the request - reponse process. The first transaction spans the application invocation while the second spans the render response phase of the JSF lifecycle. This way, any problems that might occur from a commit happen during the application invocation and not after the response render where you just rendered a message to the user that everything is ok.

Factory methods are a really nice way to let us create ways to access data in a quick and dirty fashion. For smaller result sets, this may be acceptable, but for many cases, we want to use a method that is more controllable, and lets us use pagination on the returned data. For this, we could create a method on a Dao to return the set of results based on the current page, and the page size, or we can use an `EntityQuery`. The `EntityQuery` has methods for managing ordering and pagination built-in and can be used by each query.

For now, we will define our entity query in `components.xml`.

Example 3.16. Issue query defined in components.xml

```
<fwk:entity-query name="issuesQuery" max-results="10" scope="CONVERSATION">
  <fwk:ejbql>from Issue</fwk:ejbql>
  <fwk:restrictions>
    <value>project.id = #{project.id}</value>
  </fwk:restrictions>
</fwk:entity-query>
```

This query creates a list of issues that belong to the project identified by `#{project}` by matching the `id` attribute of the `project`, again by making use of EL expressions to great effect. If we were to add pagination to our list of issues, we could use the `EntityQuery` methods for `nextExists()` and `previousExists()`, and call the `next()` and `previous()` methods on our query. My article on codeless ordered and paginated tables [<http://www.andygibson.net/blog/index.php/2008/10/02/codeless-ajax-ordered-and-paginated-tables-in-seam/>] describes how to use the entity query to great effect in producing highly interactive tables.

Since we are driving our list of issues by the value of `#{project}` we need to make sure this value is valid on each refresh which with the current implementation, it is. However, it does raise a problem with Seam, we define our `issues` query in terms of the current value of `project`. As developers, we need to be mindful that anytime we use this query, we need to have a valid value for `project`. In most cases, it shouldn't be a problem, but for larger applications, I can see some confusion in keeping track of dependencies. Compare this to Spring where the query and the `project id` is local to the flow as opposed to globally declared. If the `project` value is not defined, the restriction will not be included in the query and all issues will be returned by the query.

3.7. Issues Viewer and Editor

Now let's consider the edit/view pages for the issues. This time we'll use the `EntityHome` piece of the Seam framework so you can see how simple it can be to set up entity management using it. This is not required for using Seam, but is simply a generic entity manager that just contains the code that you would typically end up writing anyway. Early examples for Seam do not use these objects, nor did early versions of the Seam-Gen tools. However, this method of creating applications is becoming almost the standard with Seam.

The code is very similar to the code we wrote for the `ProjectHome` bean class, except that the `EntityHome` code is already written and ready for us to use in our subclass. First, let's take a look at our page code that will use our entity.

Example 3.17. issueView.xml code.

```
<h:panelGrid columns="2">
    Issue Id : <h:outputText value="#{issue.id}" />
    Title : <h:outputText value="#{issue.title}" />
    Project : <h:outputText value="#{issue.project.title}" />
    Status : <h:outputText value="#{issue.status.title}" />
    Description : <h:outputText value="#{issue.description}" />
</h:panelGrid>

<h:panelGrid columns="2">
    <s:button value="Back" action="back" />
    <s:button value="Edit" action="edit" />
</h:panelGrid>
```

Now let's consider where the `#{issue}` value is coming from. Since we are using `EntityHome`, we can define it in `components.xml` using XML.

Example 3.18. Defining issueHome and a factory for issue in components.xml

```
<fwk:entity-home name="issueHome"
    entity-class="org.issuetracker.model.Issue"
    scope="conversation" />

    <factory name="issue" value="#{issueHome.instance}" scope="conversation"/>
```

This sets up an `EntityHome` component called `issueHome` which will handle the persistence for a class of type `org.issuetracker.model.Issue`. We also defined a factory method to be called when the `issue` context variable doesn't exist. We simply get the value of `issueHome.instance` which will return the instance of the `Issue` entity we are currently working with. Note that this all uses lazy initialization, so if the `issue` variable doesn't exist, Seam calls the factory method for it which is `#{issueHome.instance}`. If `issueHome` doesn't exist, then it is created as per our definition in `components.xml`. When the `instance()` method is called, if the value held by this `EntityHome` component is null, it attempts to load one if it has an `Id` set, or create a new one if it doesn't.

Alternatively, we could write this easily in code using the following

Example 3.19. Java code implementation of IssueHome .

```
@Name("issueHome")
@Scope(ScopeType.Conversation)
public IssueHome extends EntityHome<Issue> {

    public Long getIssueId() {
        return (Long) getId();
    }

    public void setIssueId(Long id) {
        setId(id);
    }

    @Factory("issue")
    public Issue getIssue() {
        return getInstance();
    }
}
```

This few lines of code lets us implement the same thing we did in xml. The benefits here is that the Ide has better auto-completion when you use the code version, you can provide a type for the id so you no longer need to add the converter in pages.xml for assigning parameters, and you can optionally add backing bean methods if you choose to. Note that we also put the factory method here to generate the issue variable. You can also add code to handle security issues in the event that no issue id is supplied and the user doesn't have rights to create an issue. In order to use the java code mechanism, you will need to delete the definition from components.xml since they share the same component name.

The only other piece we need to deal with is passing the `issueId` parameter to the `issueHome` component so we know the id of the issue we are dealing with. For this, we will use `pages.xml` to provide the parameter to the `EntityHome` .

Example 3.20. Binding parameter to entity home id property in pages.xml .

```
<page view-id="/issueView.xhtml">
    <param value="#{issueHome.id}" name="issueId" converterId="javax.faces.Long"/>
</page>
```

Figure 3.11. Issue view

Let's consider our navigation for the back and edit buttons. This time, lets determine the navigation from the page by using action strings set on the buttons, similar to what we used with Spring. In `pages.xml` , we can extend the page definition to include the navigation.

Example 3.21. Defining navigation for `issueView.xhtml` in `pages.xml` .

```
<page view-id="/issueView.xhtml">
  <param value="#{issueHome.id}" name="issueId" converterId="javax.faces.Long"/>
  <navigation>
    <rule if-outcome="back">
      <redirect view-id="/projectView.xhtml">
        <param name="projectId" value="#{issue.project.id}"/>
      </redirect>
    </rule>

    <rule if-outcome="edit">
      <redirect view-id="/issueEdit.xhtml">
        <param name="issueId" value="#{issue.id}"/>
      </redirect>
    </rule>
  </navigation>
</page>
```

Here, we have defined the pages to go to when the back or edit buttons are clicked. Using this technique allows us to decouple the navigation from the view, with the navigation being determined based on the outcome from the view. This is important if we are using workflows since a 'back' button may take us to different places depending on the work flow we are currently in.

Now lets create the edit page. Note that again, we re-use our `IssueHome` component to handle the `Issue` object persistence tasks for us. We just need to provide the `issueId` in the parameter.

For the page itself, we will use something a little different that comes with Seam in order to handle layout this time.

Example 3.22. issueEdit.xhtml page using the seam decorator layout components

```

<s:decorate template="/layout/edit.xhtml">
  <ui:define name="label">Issue Id : </ui:define>
  <h:outputText value="#{issue.id}" />
</s:decorate>

<s:decorate template="/layout/edit.xhtml">
  <ui:define name="label">Title : </ui:define>
  <h:inputText value="#{issue.title}" required="true" />
</s:decorate>

<s:decorate template="/layout/edit.xhtml">
  <ui:define name="label">Project : </ui:define>
  <h:outputText value="#{issue.project.title}" />
  <h:commandLink action="selectProject" value="change" />
</s:decorate>

<s:decorate template="/layout/edit.xhtml">
  <ui:define name="label">Status : </ui:define>
  <h:selectOneMenu value="#{issue.status}">
    <s:selectItems value="#{issueStates}" var="v_status"
      label="#{v_status.title}" />
    <s:convertEntity />
  </h:selectOneMenu>
</s:decorate>

<s:decorate template="/layout/edit.xhtml">
  <ui:define name="label">Description : </ui:define>
  <h:inputTextarea value="#{issue.description}" />
</s:decorate>

<s:div styleClass="buttonSet">
  <h:panelGrid columns="2">
    <h:commandButton value="Save" action="#{issueHome.update}" />
    <h:commandButton value="Cancel" action="#{issueHome.cancelChanges()}" />
    <h:commandButton value="dirty" action="dirty" />
  </h:panelGrid>
</s:div>

```

Well this looks a bit different! This uses the `s:decorate` tag that Seam provides. It lets you use a Facelet template to decorate a JSF tag. It extends the Facelets `decorate` tag by making the values `#{invalid}` and `#{required}` available inside of the template. These values represent whether the input control has a validation error or if it is required. This way we can display the field differently if it is invalid or is required. You can easily write your own `edit.xhtml` template or use the Seam version. In this case, the Seam version uses CSS to position the form controls. The decorator also automatically wraps the control in Seam validation tags which automatically performs validation using the hibernate validators on the model. This reuse of validation annotations is another strong point for Seam.

Besides the strange layout method, this form is fairly straightforward. However, the one piece we should take special note of is the drop down selection for the issue status values.

Example 3.23. Status selection using Seam `selectItems` components.

```
<s:decorate template="/layout/edit.xhtml">
<ui:define name="label">Status : </ui:define>
  <h:selectOneMenu value="#{issue.status.title}">
    <s:selectItems value="#{issueStates}" var="v_status"
label="#{v_status.title}" />
    <s:convertEntity />
  </h:selectOneMenu>
</s:decorate>
```

Seam provides JSF tags to allow you to bind drop down lists or listboxes to entity lists without requiring you to wrap them in JSF select items. It also provides a converter that uses the default `entityManager` to convert the selection to an entity. This makes lookup lists from entities really simple to implement. In order for this to work, I need to create a list of the issue status values called `issueStates`. I could use an Entity Query defined in `components.xml`.

Example 3.24. EntityQuery for grabbing a list of the issue status objects.

```
<fwk:entity-query name="issueStatusQuery">
  <fwk:ejbql>from IssueStatus</fwk:ejbql>
</fwk:entity-query>

<factory name="issueStates" value="#{issueStatusQuery.resultList}"/>
```

Instead of "programming in XML", I'm going to create the same class in code by writing a class that extends the `EntityQuery` the same way we did for the `IssueHome` bean.

Example 3.25. Implementation of the `issueStatusQuery`.

```
@Name("issueStatusQuery")
@Scope(ScopeType.CONVERSATION)
public class IssueStatusQuery extends EntityQuery<IssueStatus> {

    @Override
    public String getEjbql() {
        return "from IssueStatus";
    }

    @Factory("issueStates")
    public List<IssueStatus> getIssueStatuses() {
        return getResultList();
    }
}
```

It takes only a little more code to define this in Java as opposed to using XML, and we get exactly the same result. Again, the benefit to doing it here is that we can add additional methods on there, as well as type safety, and also better auto completion in the IDE.

One last thing we need to add is to map the `issueId` parameter to the `issueHome.id` value for the `issueEdit.xhtml` page. This is a step I usually forget until I go to the page and am greeted with a blank page.

Example 3.26. `pages.xml` parameter and conversation definition for `issueEdit.xhtml`.

```
<page view-id="/issueEdit.xhtml">
<begin-conversation/>
    <param value="#{issueHome.id}" name="issueId"
        converterId="javax.faces.Long"/>
</page>
```

Figure 3.12. Issue editor

The drop down is selecting the current value correctly, and as we'll see, it will put any newly selected value back in the model. Let's deal with the save and cancel buttons.

This time, we'll use the string results from the actions to perform navigation. Also, bear in mind we will later be adding the code to select which project this issue belongs to. For this reason, we'll make this process a work flow, or as Seam calls them page flows which is the third and final mechanism we have for handling navigation.

We'll create the pageflow file called `issueEdit.jpdl.xml`.

Example 3.27. issueEdit.jpdl.xml page flow for editing the issue.

```
<?xml version="1.0" encoding="UTF-8"?>
<pageflow-definition xmlns="http://jboss.com/products/seam/pageflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jboss.com/products/seam/pageflow
    http://jboss.com/products/seam/pageflow-2.0.xsd"
  name="issueEdit">
  <start-page view-id="/issueEdit.xhtml" name="issueEdit">
    <transition name="updated" to="endState"/>

    <transition name="cancel" to="endState"/>
  </start-page>

  <page name="endState" view-id="/issueView">
    <end-conversation />
  </page>
</pageflow-definition>
```

In order to incorporate this pageflow into the Seam application, we need to add it to `components.xml` in the place defined in the default version of the file.

Example 3.28. Adding the pageflow to components.xml

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

In `pages.xml` we alter the `begin-conversation` attribute to include the name of the page flow to use when we start the conversation. The page flow typically operates as part of a conversation, with all data in the page flow being conversational data held in the conversation.

Example 3.29. Invoking the page flow when the conversation is started.

```
<page view-id="/issueEdit.xhtml">
  <begin-conversation join="true" pageflow="issueEdit" flush-mode="manual"/>
  <param value="#{issueHome.id}" name="issueId" converterId="javax.faces.Long"/>
</page>
```

We also specify that the `flush-mode` on this conversation is set to `manual` which means that changes to our entities are not flushed until we manually call the `flush` method. This is similar to adding the `persistence-context` tag to the Spring flows making the PC last for the duration of the conversation.

We add two buttons in `issueEdit.xhtml` for saving and cancelling changes.

Example 3.30. JSF for the save and cancel buttons in `issueEdit.xhtml`

```
<h:commandButton value="Save" action="#{issueHome.update}" />
<h:commandButton value="Cancel" action="cancel"/>
```

When the user clicks the save button, we call the update method of the `issueHome` component. This persists our changes and flushes the persistence context. It also adds a faces message indicating the instance has been saved. The update method returns a string of `updated` which we respond to in the pageflow. In the event of an `updated` or `cancel` string action, we navigate to the `endState` which is the issue view page.

Notice that we don't pass any parameter to the issue view page used for the end state. The reason is because even though we end the conversation, the conversation is re-used in the next page, so all the context variables of the conversation are still available when we get to the issue view page. This means we don't need to re-load the issue to view it since there is already a context variable called `issue`. Besides the fact that I'm not a fan of this implicit passing of data, it also can be problematic. What if we made changes to the issue that did get posted back to the server as part of the user interaction and the current instance of the `issue` entity is dirty? We need to refresh it somehow. We can actually view this problem in action by adding a button that does nothing on the edit page. If you edit the issue title, click the button (which posts changes but stays on the page), click cancel, then the issue view page displays the changed issue with the wrong title.

Fixing this problem is simple - kind of. We need to either invalidate both the `issue` variable and the value of `issueHome.instance` which is where it will be obtained from when it is not found. Remember, our home bean is stateful so even though the `issue` variable can be invalidated, it is pointless if Seam is just going to fetch the dirty copy from `issueHome.instance` via the factory method.

Alternatively, we can just refresh the current instance of the `issue` we are using. The other option is to use events. Events can be raised from numerous places, either via code or as part of the page flow or `pages.xml` navigation. We might raise an event for when we cancel changes to the Issue. We could then outject a null `issue` value or just set it in code, and invalidate the issue instance held in the `issueHome` component. However, it seems a little complex to create event listeners for every kind of entity we want to edit.

Unfortunately, the default implementation of `EntityHome` does not contain a `refresh` or `cancelChanges` method in which we can do any of this. We can extend the `EntityHome` class to include this, you can use your own version instead of the built in version because the framework tags have a `class` attribute that lets you specify the class to use for the `EntityHome`.

To do this, we create class called `EntityHomeExtended` which extends the `EntityHome`.

Example 3.31. Code listing for `EntityHomeExtended`

```
public class EntityHomeExtended<E> extends EntityHome<E> {

    public String cancelChanges() {
        getEntityManager().refresh(getInstance());
        return "cancel";
    }
}
```

We could have expanded on this so it follows the standards set in the `persist()` and `update` methods with regards to raising events and putting messages in the `FacesMessages`. To use our new class, we just change the `issueHome` definition in `components.xml`.

Example 3.32. Framework `EntityHome` using our extended class.

```
<fwk:entity-home class="org.seamtracker.session.EntityHomeExtended"
  name="issueHome"
  entity-class="org.issuetracker.model.Issue"
  scope="conversation" />
```

Alternatively, if you used the java code version of `IssueHome`, we just extend it from the `ExtendedEntityHome`. Finally, we amend our page to call the new method when the user clicks cancel.

Example 3.33. JSF for the cancel button with our new method

```
<h:commandButton value="Save" action="#{issueHome.update}" />
<h:commandButton value="Cancel" action="#{issueHome.cancelChanges()}" />
```

Now we can make changes, click cancel, and any changes to the issue will be undone with the entity being refreshed by our new method.

3.8. Selecting Projects

Now let's look at the issue of changing the project for the issue. First, let's add a link to change the project. We put this next to the project name display in the `issueEdit.xhtml`.

Example 3.34. Adding link to `issueEdit.xhtml` to select a project.

```
<h:outputText value="#{issue.project.title}" />
<h:commandLink action="selectProject" value="change" />
```

Now we need to consider a few things. Until now, we have not used `DataModel` in our pages, mainly because we haven't needed to since we have been using parameters for passing data around. In order to select a project, we probably want to use a `DataModel` to create a clickable list. We also need to have some kind of flag that will let us indicate on the project list page that we want to select a project. Since we have our `ProjectListBean`, we can use that to hold the flag since the whole bean is stateful. We add some methods to the interface to handle these tasks.

Example 3.35. Additional interface methods for project selection on the projectList interface.

```
Boolean getDoSelect();
void setDoSelect(Boolean doSelect);
public Project getSelectedProject();
public void setSelectedProject(Project selectedProject);
```

We then implement these methods on the projectListBean .

Example 3.36. Implementation of project selection code in projectListBean .

```
private Boolean doSelect;

@DataModelSelection
private Project selectedProject;

public Boolean getDoSelect() {
    return doSelect;
}

public void setDoSelect(Boolean doSelect) {
    this.doSelect = doSelect;
}

public Project getSelectedProject() {
    return selectedProject;
}

public void setSelectedProject(Project selectedProject) {
    this.selectedProject = selectedProject;
}

/--pre existing code
@Factory("projects")
@DataModel
public List<Project> getProjects() {
```

The doSelect flag is used to determine whether the projects page is displayed in selection mode, or normal browsing mode. As we added a @DataModel annotation to the getProjects method, when Seam fetches the list of projects it wraps the list in a DataModel , and we can have a list of entities which can be used as clickable links. When a DataModel row link is clicked, Seam can automatically push the entity into a property which has been annotated with the @DataModelSelection . By default the single DataModelSelection is populated from the single DataModel clicked item. If you have more than one DataModel on a bean then Seam cannot assume the defaults, and you have to start naming the data models and the associated datamodel selections.

In our pageflow, we add the transition and the page for selecting the project, and the method call to set the project value on the issue entity.

Example 3.37. Adding the transition for selecting a project for the issue.

```
<start-page view-id="/issueEdit.xhtml" name="issueEdit">
  <transition name="updated" to="endState" />
  <transition name="cancel" to="endState" />

  <!-- new transition -->
  <transition to="projectState" name="selectProject">
    <action expression="#{projectList.setDoSelect(true)}" />
  </transition>
</start-page>

<page name="projectState" view-id="/projects.xhtml">
  <transition name="cancelSelection" to="issueEdit" />
  <transition name="selected" to="issueEdit">
    <action expression="#{issue.setProject(projectList.selectedProject)}" />
  </transition>
</page>
```

When we click on the select project link, we set the `doSelect` flag on the `projectList` bean to true. When we get to the projects page, we use the `projectList` bean that we created in the issue edit page to display the projects. When displaying the projects, we query the select flag to determine whether the projects can be selected. If so, we display the select link in a column in the data table.

Example 3.38. Column for selecting a project in the table.

```
<h:column rendered="#{projectList.doSelect == true}">
  <f:facet name="header">Select</f:facet>
  <s:link value="Select" action="selected"/>
</h:column>
```

We only render the column if the `doSelect` flag is set and the link returns an action of `selected`. Going back to our page flow, in the `projectState` state, we have a transition for `selected` that sets the project for the current issue to the selected project value in the `projectList` bean.

Example 3.39. Transition to set the project when selected

```
<transition name="selected" to="issueEdit">
  <action expression="#{issue.setProject(projectList.selectedProject)}" />
</transition>
```

If the action is `selected` then we first call the `setProject()` method on the issue, and set it to the current `selectedProject` on the `projectList` bean. We then transition to the `issueEdit` node which is the `issueEdit` page.

One annoying problem I found with Seam pageflow is that I had a couple of errors in my code which resulted in an endless loop for the navigation handler. Not only is it annoying but it is hard to pull out the real cause of the problem since it gets scrolled off the console display if it is displayed at all.

3.9. The Seam Framework Approach

The approach the Seam team took to letting you interface with the pieces of the Seam Framework was to instantiate them as Seam components just like the components you write. That means, as the manual says, you get to work with "one kind of stuff". For example, in your web page, you can optionally render a 'debug panel' which includes all sorts of framework related information such as:

Example 3.40.

```
#{conversation.id}  
#{conversation.isLongRunning}  
#{pageFlow.hasDefaultTransition}
```

These components are also easily available in code by either injecting them, or using the static access method. Just about every piece of the Seam Framework is accessible using these three methods which makes the Framework very accessible.

```
@In  
Private Conversation conversation
```

Or using :

```
PageFlow.getInstance();
```

Since these components are exposed via EL, and Seam tries to work in EL in almost every place it can (pageflows, messages, debug log messages, queries, sql) you can access these components from just about anywhere. Also, because the tooling the IDE team have built gives you plenty of EL code completion, the "one kind of stuff" rule means you also get auto-completion on these framework components.

Additionally, the system components can be replaced with your own versions if you chose to. There is precedence to component installation which lets you override the default implementation with your own. This component precedence is also used to instantiate Mock or test components when running tests instead of the actual components.

3.10. Other Features

Seam comes packed with a bunch of additional features from PDF and email generation using JSF tags to Seam Remoting. It extends the EL expressions used in JSF to allow the use of parameters in calls. It includes JSF components that provide GET request features as opposed to JSF's POST only implementation. It comes with the ability to integrate Spring beans into your Seam application as first class Seam components, as well as the ability to integrate with GWT. Seam also allows you to create conversational web services in which you can hold a stateful conversation with the server from a variety of clients. Seam also allows for the injection of the POJO Cache as a Seam component, and also allows for the caching of rendered JSF content fragments with a single JSF tag. Seam comes bundled with Ajax4Jsf and the Richfaces JSF component set which gets you up and running and creating ajaxified applications in no time.

Seam also comes with Seam-Gen than not only generates the basic application, but can be used to generate forms and JPA entity objects from data tables as you want to add new forms. While this works pretty well, and can speed things up a bit and can get you going, there can be problems with different ideas on how

tables should be named and we are getting into code generator territory which means you start losing your grip on how your app is written. However, the basic Seam-Gen function of creating an app that is ready to be deployed works very well and takes a minimalist approach to code generation.

The number of options available regarding security is lacking, but as of writing (June 2008) it is undergoing a serious overhaul and from first glances, it looks like it will be almost on par with Spring Security. Like most features of Seam, security is available from EL expressions, and also can be applied at the page level, method level, bean level or HTML component level.

Seam comes with a transaction manager that solves problems relating to committing changes and then needing a transaction when rendering a view. Typically with web frameworks if you use the Open Session In View solution to the lazy initialization exception, you have a transaction that spans from the start of the request to the end of the request when the response has rendered. However, what if you save data in the action, and render a success message only to find that you have an error when you commit the data in the transaction? If you commit the transaction after the bean action, you could get an LIE if you require additional pieces of the object graph to render the response. Seam's answer to this is to use two transactions. The first is used to wrap any persistence that takes place in the bean action. The second is used to provide any additional object loading in the rendering phase of the request. Also, if something fails in the update during the bean action, we find out before we are rendering a response, and we can actually let the user know.

Seam also provides process flow management so you can create work flows that last longer than any user session or conversation as well as incorporating JBoss Rules for defining business rules. However, I have had little experience with either of these two Seam functions.

`Pages.xml` lets you define pages to default to in the case of exceptions, as well as defining what actions to take on a specific page if there is no active conversation, and whether a conversation is required. This can take care of the issue of the double submit problem since you can't go back and reopen a conversation that closed when you clicked submit the first time round.

Seam really does tie together a number of good functions and features in a single stack which is something I think Java has been missing for a while. What's more, it isn't just a collection of distributed libraries, they are pulled in and made a part of the Seam framework. This was one of the goals behind the Seam framework, to make the whole far greater than the sum of its parts. Yes, in other applications, you can use iText for PDF generation, but only Seam lets you access it through JSF components using the same Facelets and EL data access mechanisms that you have already set up. Seam embraces and provides plenty of value add ons for these libraries. Seam also appears to provide a lot more flexibility than Spring on how you get things done by offering multiple ways to do the same thing. The downside to this is that there is no 'best practices' way of doing things and some ways *could* be more problematic in the long run and as applications grow. The demos were mostly written before the Seam Application Framework was finalized, and the framework is now the recommended method while their Hotel booking example still focuses on passing objects around between beans which I am not a fan of.

JBoss worked with Exadel on making their JSF editor tools and Richfaces a part of the JBoss offerings, and includes seamless integration with Seam. Exadel have now also come out with Flamingo which lets you use Seam with Adobe Flex.

Seam has sophisticated conversation management using a single parameter to keep track of the conversation for each page. Seam also supports the notions of workspaces whereby you can have multiple conversations running at the same time and choose between them. Each conversation can be given a description which can be used to distinguish between the conversations. For example, if you search for hotels, and decide to open up a booking form in a new window for 4 of them, you create 4 workspaces that you can switch between. Each one has an assigned description such as `Book Hotel #{hotel.name}` so the workspace description has meaning. You can also use business conversation Ids so if your URL is `/projectEdit.seam?projectId=12` you can tell Seam to use the project Id value as the

conversation Id for a given page. The advantage here is that if you navigate off that page, and go back to `/projectEdit.seam?projectId=12` you will end up back in the same conversation. Seam also lets you assign a timeout value for conversations and if needed, you can assign a different timeout value for individual conversations. Tie this in with RESTful Urls using URL rewriting which now comes with Seam, and you can use `/projects/12/edit` as your URL and not only will it manage your conversation for you, but it will give you a bookmarkable URL with JSF.

Seam also has a great Events system which lets you trigger actions in backing beans in response to events raised by your application, or events fired by the Seam framework itself. These events work on a subscriber basis so adding another observer to the event is as simple as annotating a Seam component with `@Observer([eventName])`. You can also raise your own events and add observers to them so you could have a bean that receives a specific event (i.e. closing a case) to send an email to a supervisor. Data can also be passed in events so the particular case can be passed to the event observer.

The other aspect of Seam is as a conversational backend for non-JSF front ends. I haven't really played with this that much, but there is a Wicket-Seam interface in the works, there have been a number of posts about getting GWT to work with Seam, and Exadel have released Flamingo which is their Seam interface for Adobe Flex. Seam lets you write conversational Web Services which is another way to get more mileage out of the code you write.

Seam Remoting lets you call your server side Seam Components from Javascript with a couple of bean annotations and javascript include files added to the web page. This opens up plenty of possibilities for creating lightweight sites that run mostly on the browser with only a minimal amount of server interaction. Not only can you call business bean methods but also gain access to your domain model from javascript. You can also use conversations in your interactions with the server.

Chapter 4. Summary

First off, these are two excellent solutions to have to choose between. Most of the problems with basic CRUD web development can be solved with either of these frameworks, and it's fairly safe to say that there are no real reasons not to use either of these.

4.1. Comparing Solutions

Let's start by recapping some of the differences between the two and noting their respective strengths and weaknesses in different areas

4.1.1. Configuration

Seam comes with Seam-Gen which is a command line tool to get you started writing Seam applications. Alternatively, you can use the JBoss Tools / JBoss Developer Studio tools to create applications. Seam requires you to list any page flows in the `components.xml` file. Annotations are usually used to name components in source, but that too can be done in `components.xml`. Pageflows are invoked by name when a conversation is started.

Spring essentially leaves application creation up to the developer which isn't surprising given that Spring gives you unlimited options on the libraries you can use with it. Web Flow configuration involves a number of different classes and beans. It also requires you to be running Spring MVC underneath it all. It wouldn't be a Spring solution if you couldn't interchange nearly each and every part of the solution. For that reason, I'm sure there are no limits on how you could set up the mappings between URLs and flows. The default is to map URLs to flows with the option of using wildcards. Spring beans are defined either by annotation or using xml configuration files. These can be imported into the spring config files so there is no rigid structure in terms of content or location.

4.1.2. Flow Variable Declarations

In addition to the stateless, session and application scopes, both frameworks offer additional scopes. Spring offers a flash, view, flow or conversation scope, with the conversation spanning one or more flows. Seam lets you define a conversation scope which is equivalent to SWF's flow scope, or a page scope which holds the object in the page and appears similar to Spring's view scope. As a modifier to the Seam conversation scope, you can indicate that an object is created per nested conversation giving it a scope limited to that conversation, similar to Spring's flow scope.

Spring lets you wrap values in a `JSF DataModel` which can be specified in the flow. Seam also enables you to easily do so using an annotation which requires that you have an actual class to annotate. If you define a query in `components.xml` then there is no way to wrap the results in a `DataModel`.

Regarding instance declaration, Spring uses a 'push' convention over 'pull' while Seam can use both although it tends towards the pull convention. With Spring, you need to declare a data instance in the flow before you use it, while with Seam, you typically declare how that data is created with a factory annotation before you can use it. When it is requested (usually from a reference on a JSF page), the data instance is created and put into one of the Seam variable contexts (depending on the assigned scope). Alternatively, you can just outject object instances from classes to emulate a push technique, although this technique requires you to call a method call on the bean to trigger the outjection process.

Seam uses a global approach to defining it's named beans which gives you the benefit of being able to define beans once and use them anywhere in the application. Seam also allows you to create multiple instances of the same class with different names and possibly different scopes. Defining a component

with `perNestedConversation` gives you most of the power of Spring's declaration within the flow. However, this cannot easily be applied to the persistence context which by default, only allows one instance per conversation stack.

Since Spring defines the data in the flow itself, it gives you slightly more control on a per flow basis, but it could lead to repetition and the possibility of variable names stepping in each others toes as one flow invokes another flow. However, this could be managed by scope control of the variables created and mostly using the more restrictive flow scope, and only using conversational scope when needed. Flow inheritance could solve the repetition issue, but it might be overkill to create a flow with common code in just to be inherited by two flows (i.e. edit and view CRUD pages).

4.1.3. Conversational URLs

Seam offers natural conversation urls where the conversation is identified by a an identifier that is contextual. By default, a Seam url with a conversation is `partEdit.seam?partId=1234?cid=5` where the conversation is identified by the `cid` parameter. You can define a natural conversation for that page that makes the conversation Id defined by the part number parameter instead so rather than have a synthetic conversation id of 5 passed in a parameter called `cid`, you will have a contextual conversation id defined by the part number in the `partId` parameter. This leaves you with a URL of `/partEdit.seam?partId=1234`, or, if you use the `UrlRewriter` a RESTful URL of `/parts/edit/1234`. Spring appends something like `?execution=e3s1` on to your URLs which can be quite ugly. The 's' part of the URL indicates the state and increments for each page in the flow. It is this that identifies each state point in the flow allowing us to use the back button to get to any point in the flow and essentially, travel back in time to that point, and resume the flow from that point. Seam just has one conversation that spans all pages that participate in it and only one set of state which is the current state.

4.1.4. Page flows

Both solutions provided an easy way to write navigation rules and flows, and took very different approaches to it.

Spring navigation is more decoupled from the view than Seam. Spring favors returning action strings like `save`, `cancel`, or `select` from the view and executing code within the flow based on those actions. It is probably the more technically correct way to do it from the sense of decoupling the view from the navigation and business logic. While the existing navigation language is simple, but fairly adequate, the Web Flow team has talked about adding new features based upon user feedback. On important feature I found was the ability to put as many commands as you wanted in the transition segments.

Seam gives developers a choice of navigation methods. While you can use the JSF based navigation, it is much easier to use one of the two alternatives. If you use `pages.xml` for navigation, you navigate based on string actions, on logical statements, and it also depends on what method the string action was returned from. However, you cannot call methods in response to those string actions as part of the navigation. Typically this means your view has to call a method to perform an action and you navigate based on the outcome of that action. In order to use string actions in your view and call methods on the back end, you must use page flows like Spring Web Flow does. Having multiple navigation methodologies might seem overkill and complex, but it does offer solutions of different degrees. The page flow language does feel rigid, and unforgiving in many ways. There's no way to make multiple method calls, there's no way to outject values to the parent conversation. It often feels like you are digging for ways to do things which should be intuitive.

Another issue is that if you navigate through the flow, and click the back button and try to take another path, you get `Illegal Navigation` error messages. This is because the flow isn't synchronized with the view. If you navigate from page A to page B, and click the back button and try to navigate off page A using action string "XYZ", the pageflow handler is still on page B, and since page B doesn't know about action

string "ABC", it throws an illegal navigation error. Spring has a finer granularity in storing its state. When it appends the execution info in the url, it is of the form `execution=e3s4`. When you navigate through the flow the number after the 's' increments, so Spring stores not only the state for the flow, but also for each state of the flow on a page by page basis. This is handy for avoiding problems like the one shown above, but can come with it's own set of headaches. Spring however lets us discard or invalidate history on a transition by transition basis.

One thing that should be noted is that it appears that Spring Web Flow cannot be used for pages not included in a web flow. If you have a page that is not in a page flow and you want to display some data, then you still need to provide some way to generate that data and bind it to the EL expression for JSF to display. For this reason, any time you want to use Web Flow for passing data to JSF, it must be done within a flow. While there is a JSF variable resolver for Spring, it means you have to learn and use a whole different technique for implementing JSF pages with Spring. On the other hand, you could also use this as an opportunity to delve in to some Spring MVC for those pages since you already have most if not all Spring MVC elements installed in your application.

Seam on the other hand is designed to be used with varying degrees of intrusion into your pages. You don't have to start a conversation and a pageflow in order to use the Seam defined variables. The expression `#{projects}` works in any page and even outside of pages because the declaration is application wide. This can even make your java code conversational since it always executes within a conversation. EL expression values can be used for error messages, navigation rules, in any email or PDFs you generate and even logging. Overall, Seam delivers a more unified and integrated approach to fetching and generating data than Spring.

One point where Spring shines is the ability to localize the data in a flow resulting in cleaner nested flows. With Seam, data is declared globally, there is the `@PerNestedConversation` annotation that can let you define new instances of variables in a nested conversation which is a half measure to achieving the same functionality. However, passing data to a nested or parent conversation is not easily done. Part of this probably stems from Seam's perception of nested conversations. To Seam, a nested conversation is meant to be a part of the parent conversation, not a separate action. I.e. choosing a hotel room type in the nested flow for a hotel booking that is in the parent flow. Spring on the other hand can let you use subflows as totally separate and encapsulated processes such as editing an issue, going to select a project for the issue, and then choosing to add a new project). This behavior is very thick-client-ish since it gives the users a lot of flexibility. However, in Seam, as soon as you go to create a new project, it would look up the `project` variable and find the instance from the parent conversation, thus preventing you from adding a new one. You would then have to add the `PerNestedConversation` annotation to the `projects` bean which could introduce problems of its own. While this is very potential problem, it isn't a big one. Interfaces that let users go round in endless circles usually end up with the user getting lost and to some degree should be somewhat constrained. However, it was impressive going round in circles in the Spring implementation and having Spring keep track of my flows and data instances and shuffling up and down the conversation stack with ease.

4.1.5. Documentation and Examples

Seam has excellent documentation, and a large number of Samples, with a 650+ page PDF manual starting with simple examples and documenting most of the features with a mini example of the syntax for each. The documentation for core Spring is excellent, however the documentation for Web Flow is lagging behind with most functions having little coverage. This is understandable to a degree, and Seam documentation was lacking (although a little better than SWF's current state) for the first few releases until Gavin took the time to document and it grew from 250 pages in 1.2 to 350 pages in version 2.0.1 to 650+ in 2.1.0. However, poor documentation can be annoying, especially after I wasted a couple of hours trying to get the expression `currentEvent.entity` to work when changes in the web flow API made in May 2008 required it to be `currentEvent.attributes.entity`. The documentation, as of January 2009 still reflects the old version. It wasn't until I read the release notes that I found out about the change. Spring

has only a few examples, mainly a duplication of the Seam booking application, with one version for each of the different web flow technologies. Since many people use the examples as a basis for finding solutions to their own problems, fewer examples means less chance that people will find answers within them.

4.1.6. A Complete Solution?

It might be unfair to compare how far these two products go to provide a complete solution since Spring isn't aiming to do that. Spring offers page flow control and stateful data management on top of its IoC container and other core functions. It doesn't strive to be a complete solution, and it expects you to add in any additional nuts and bolts (i.e. security, Ajax Frameworks, iText or email generation) yourself, and either implement your own Spring integration with third party libraries or hope the libraries already have it built in. Either way adding the pieces to the Spring stack would take time and the integration probably won't be as seamless.

Seam on the other hand aims to be a complete software stack. Cynics might disagree with the idea of someone else choosing which libraries they use for development since in this day and age it almost feels like Framework and library selection should be a full time job for Java developers. However, the Seam team has strived to create a stack that really does deliver almost everything a developer could need. Most components could be replaced if needed, but out of the box, everything works together nicely. If you want to get going and start writing an application with Seam and the JBoss Tools IDE, you can find yourself very productive in less than a minute, complete with IDE integration and hot deploy.

4.1.7. Layering and Decoupling

Both frameworks let you design layered applications in order to separate concerns. They both let you create Dao or Service beans which provide generic services to more function specific beans. Seam has come under some fire due to the lack of apparent layering in the demo applications. However, the lack of layering is by design in the demos (they don't really need it), and not because Seam is incapable of layering. You can add as much layering as your sanity will allow under Seam without a problem.

One advantage of JSF is the ability to decouple the view from the back end objects via the use of EL to loosely couple beans to view elements. Seam expands on this with contextually named data that results in less coupling between the view and the data in the conversation. If we use the expression `#{customers}` in several pages, our view is only bound to this expression. Seam then binds this expression to a method or object using a factory. At a later date, we can change the factory for the `customers` expression to another method, or another bean, or even another layer in our applications and all bindings in our view will get the data from our new source.

With Spring, because all of our variable declarations are in the flows themselves, if we want to fetch our customers from a different location, we need to change the declaration in each flow that uses it. This is because each flow couples that variable name to a method. Spring can eliminate the problem by layering the application and having a `customerDao` with a `getCustomers` method which is then coupled to the expression `customers` in the flow. To change the source of the data, you need change the implementation in the Dao. This would require a little pre-thought before implementation. However, it does demonstrate one reason why Seam doesn't require as much layering as it has an extra layer (Seam itself) between the view and the backing beans.

Another aspect of this is code portability. Spring maintains the same Model/Dao layering where the Dao returns model data, and model data is put into the flow under a variable name which makes it accessible to the view. Spring makes it fairly easy to take that Model/Dao code and use it in another application or test environment, as well as keeping things lightweight. Seam on other hand promotes thicker classes in the conversation. For example, the `EntityHome` beans contains the reference to the model entity instance, as well as all the Dao functionality including a reference to the `EntityManager`. While this is still somewhat portable, there is a higher degree of integration of the code with Seam than with Spring's 'Roll

Your Own' data access approach. You can of course roll your own data access layer with Seam, but then you lose the ease of using the EntityHome/Query beans.

4.1.8. Taking the future into account

Of course, one factor in all of this is how well the frameworks will weather the future. Nobody wants to adopt a framework that will be extinct in a couple of years. Spring has somewhat of an advantage here in that there are plenty of Spring users already in existence, although few of those are Web Flow users. One might compare that to the number of EJB users versus the number of Seam users since there are plenty of people use EJB but without Seam.

Gavin King is working on creating the Web Beans JSR (JSR-299) which will seek to make a standard out of the component declaration model used in Seam and Google Guice. This is not a case of making Seam a standard since I believe there are a number of differences between Seam and Web Beans. However, once Web Beans is out, Seam will probably move towards being a Web Beans implementation and is probably due to become the reference implementation for the JSR. Despite building the framework around standards, there is support for Wicket, Adobe Flex and GWT. A web beans standard can likely help those integrations by standardizing the use of contextual named components. Web beans will also become a more pojo oriented framework which could attract those that run screaming at the sight of EJBs. However, since Web Beans is a standard, like all standards, it will not be as complete as either Seam or Spring as a solution. One concern for Seam could be that once Web Beans comes out, will we start seeing Web Beans based stateful frameworks without all the additional features that Seam has (pageflows, security, deep EL integration etc) that might be more attractive as Seam-Lite.

Spring have been moving ahead with their own efforts to create a single full stack with the Spring Application Platform focused around an OSGI based application server. They are passing up on the standards and sticking with their own solutions. Interestingly enough Spring are pushing their own application servers for use with their own Spring stack, while the Seam team are making efforts to promote cross server usage of Seam.

While many have aversions to standards (rational and otherwise), there is a varying degree of benefit to adopting standard technologies. Seam is all about standards, and will probably continue to be a framework built on standards. Spring on the other hand caters to the standards when they think it is prudent (i.e. plenty of JSF integration with Spring Web Flow), and ignore them when they think it is not. Support for Spring Web Flow with other frameworks is more likely to come from the other frameworks trying to integrate with Spring rather than the Spring Team reaching out to other frameworks. The Seam team has made a number of efforts to include other frameworks (i.e. Wicket, GWT and even spring), as well as other app services (WebSphere OC4j, Glassfish).

4.1.9. Usability

Sometimes, SWF feels like a more professional and polished product with professional developers building real world products in mind. As such it seems not to suffer from some of the pains that Seam has, many of which can be overcome, but some that require a little work. Even when things go wrong in Spring, you get a fairly clear error message most of the time (but not always!), sometimes even a suggestion on how to fix it. With Seam, you often have to first decode the exception to determine where things went wrong, and then fix the application. For example, if you have an error in your pageflow, it will often loop endlessly complaining that the workflow hasn't started. Somewhere in there (or at the top of the log) is a helpful error message, but only if you are lucky and only if you dig for it. Usually the error messages are cryptic and I find myself struggling to remember what kind of problem is represented by the symptoms I am seeing rather than having a message telling me what happened. However, these are relatively small problems and somewhat superficial even if they are frustrating. Support on the groups is often very helpful since everyone is using the same product stack (in most cases, the same versions too), and not mixing and matching libraries and library versions which can be a problem with Spring.

4.2. Summing it up

These are two very good frameworks, with only small issues between the two which makes any kind of summary or conclusion difficult to determine.

To get the obvious out of the way, Seam is jam packed full of features which many people might not appreciate and consider Seam to be bloated or heavyweight which seems nonsensical given that Seam is designed to be a complete solution in a box. That said, here we are looking at how well the frameworks tackle most of the work we do which is CRUD with logical flows between pages.

While Spring Web Flow might offer a more lightweight solution as an optional plugin to your web framework stack, it lacks the same deep integration that Seam has. Like many Spring libraries which can be bolted on to your projects it takes a little work and sometimes even some code to get the different pieces working together.

If you are using JSF and have no problems using EJB3.0 (but not requiring it) with a default stack of JBoss AS and Hibernate then Seam is well worth taking a look. I think Seam as a back end to JSF can be easier to use with the EL integration and the narrower focus of the framework makes it more powerful out of the box. There are still some points which need polishing up on, but for the most part they can be worked around.

If you favor Spring libraries then you can feel right at home with Spring Web Flow as it will meet most of your needs and integrates the same way most of the other Spring plugins do. The weakest points are exception handling, passing messages outside the flow and the issue of choosing between using a flow for each page or using another technique to create non-flow JSF pages. Any weak or missing points can probably be resolved through custom code.

The Spring solution isn't as well integrated as the Seam solution which is good and bad for both. When Seam makes a mistake it is hammered all the way through the framework while Spring requires you to do more work but gives you a little more control on a few features. However, in some areas, Spring provides no support for some features and expects the user to handle those pieces. JBoss has some potential to add features to Seam that can bring some aspects up to par with that of Spring Web Flow. For example, defining variables in a pageflow or even in `pages.xml` making it local to that flow or page, giving the persistence context more flexible scopes, and allowing multiple action calls during navigation transitions. These are probably not without backwards compatibility issues, but it would allow Seam to take on some of SWF's strengths.

IDE support for Seam really shines since they partnered with Exadel and the code suggestion for Seam components (even EL expression within Java code) is wonderful. Spring also has IDE plugins but again suffers from the lack of deep integration in the Spring framework. This idea of a full stack offering deep integration while the lighter stack offers less integration is a common theme between these two solutions.

Regardless, both projects have been well received, and appropriately so as these are two excellent and freely available tools. While they both have strengths and weaknesses, they are both great frameworks.

Chapter 5. The Wicket Factor

After completing this set of articles, I decided to take a look at this same project and try and implement it with a framework that didn't provide conversational and flow functionality. Rather than start from scratch with something like PHP or Spring MVC which would make the statefulness completely manual, I decided to try and write it in Apache Wicket [<http://wicket.apache.org/>] which is another framework I'm a fan of. Wicket is a framework which totally abstracts the HTTP and web mechanisms allowing the developer to write their application using pure java and Object Oriented code, more like a Swing Application.

This isn't meant to be a strict head to head against Seam and Spring for a number of reasons. First, Seam and SWF both offer built-in functionality to solve the problems we are trying to work around. Wicket just gives you the environment to build your own solutions to those problems. The value-add from Wicket is that it stays out of your way and only acts as a very thin layer between your OO Java code and the server. Because Seam and Spring Web Flow were supposed to handle the issues of creating stateful web applications, I had a 'rule' that the applications should be mainly built using the features out of the box rather than those which were created from code. With Wicket, just about every page you write involves writing some code so that rule doesn't quite apply.

If there is any comparison it would be how easy it is to implement similar functionality using Wicket. Granted, there is a lot of code in Seam and SWF that does a lot of work, however I think it quite likely that implementing lightweight alternatives to some of the most used core functionalities of these frameworks is possible with Wicket.

5.1. Getting Started

Creating an application with Wicket is simple, simpler than the other two frameworks especially if you are using Maven. The Wicket web site also includes instructions on starting a Wicket project with Maven. To set up the application in Eclipse, just create a new web application, and create a subclass of a wicket `WebApplication`. The only requirement here is that you implement the `getHomePage` method that returns the default page class. Only a minimal amount of xml needs to be added to `web.xml` to add the Wicket servlet and give the class name of the Web Application class. There is no other XML configuration required for Wicket.

Example 5.1. `WicketTrackerApplication.java`

```
public class WicketTrackerApplication extends WebApplication {
    @Override
    public Class getHomePage() {
        return ProjectListPage.class; //we'll define this later
    }
}
```

Example 5.2. Web.xml

```
<filter>
  <filter-name>WicketTrackerApp</filter-name>
  <filter-class>
    org.apache.wicket.protocol.http.WicketFilter
  </filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>
      org.wicketracker.web.WicketTrackerApplication
    </param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>WicketTrackerApp</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Wicket pages work by defining a page class and some HTML with the same file name for each page in the application. The HTML and the class reside in the same source package, but Wicket can be configured to keep the HTML in another location. In a Wicket page, the HTML contains the markup and in the corresponding class, you create java-side components to be bound to the markup. The concept is easy to understand once you see an example. We'll start with our project view page which consists of the `ProjectViewPage.java` class and the markup `ProjectViewPage.html`. I've also created a simple template called `BasePage` which also consists of some HTML and a class from which our pages will extend. This is markup inheritance and is how Wicket handles templating.

Example 5.3. ProjectViewPage.html interface

```
<html>
<body>
<wicket:extend>
  <table>
    <tr>
      <td class="label">ID</td>
      <td><span wicket:id="id">{id}</span></td>
    </tr>

    <tr>
      <td class="label">Title</td>
      <td><span wicket:id="title">{title}</span></td>
    </tr>
  </table>
</wicket:extend>
</body>
</html>
```

I used a table for the layout simply because I didn't want to clutter the HTML with additional markup to make things lay out nicely. Here we have the Id and Title captions created as spans that contain `wicket:id` attributes. These attributes provide a way to bind the markup to the server side components we will create in our backing java code.

Example 5.4. `ProjectViewPage.java` class

```
public class ProjectViewPage extends BasePage {

    public ProjectViewPage(PageParameters parameters) {
        Long id = ParamUtils.getLongObject(parameters, "projectId");
        Project project = EMF.createEntityManager().find(Project.class, id);
        add(new Label("id",project.getId().toString()));
        add(new Label("title",project.getTitle()));
    }
}
```

In the Java class for the project view page, we get the `projectId` from the page parameters, load the project and then create the Wicket label components using the `project` object to provide values for the components.

We can create a similar page with the same kind of code for the `Issue` object which we will call `IssueViewPage`. When we get to them, the edit pages for the project and issues will be somewhat similar which makes you realize that one key fundamental to developing with Wicket is that you can leverage all the best Java design practices to reduce complexity and repetition. For example, the code to grab the Id from the parameters and load the object could be genericized including handling errors for missing parameters or missing entities. When you start using Wicket, you almost get a case of code freeze as you start thinking of the possibilities for creating very re-usable code and mini-frameworks with just a small amount of code. In these examples, I have opted not to go a generic route but have focused on the different ways that Wicket will let you do certain things.

One common feature in Wicket is the concept of models which provides Wicket components with source data for display. Initially, you might find yourself just passing the String values to the component as the model like we did in the `projectView` page. Models can also be objects that implement the `IModel` interface. This simple interface provides a mechanism for the component to consume the model as needed from the implementation. The implementation could just hold a simple object reference, or it could trigger the fetching of data from the database or some external source. Again, the abstraction leads to unlimited possibilities.

I implemented the `IssueView` page a little differently by using a `CompoundPropertyModel` which can be applied to a parent container such as a page, form or panel and be shared with the child components in that container. The child components take their values from the model using the reflected property values based on the component ids. The `CompoundPropertyModel` becomes far more useful when we are editing values and it provides the two way binding by reading the values from the model and pushing them back into the model on the update.

Example 5.5. IssueViewPage.java

```
public class IssueViewPage extends BasePage {

    public IssueViewPage(PageParameters parameters) {
        Long id = ParamUtils.getLongObject(parameters, "issueId");
        initPage(id);
    }

    public void initPage(final Long id) {

        Issue issue = EMF.createEntityManager().find(Issue.class, id);

        IModel compound = new CompoundPropertyModel(issue);
        setModel(compound);

        add(new Label("title"));
        add(new Label("description"));
        add(new Label("id"));
        add(new Label("project.title"));
        add(new Label("status.title"));
    }
}
```

Example 5.6. IssueViewPage.html

```
<html>
<body>
<wicket:extend>

<table>
<tr>
<td class="label">ID</td>
<td><span wicket:id="id">{id}</span></td>
</tr>

<tr>
<td class="label">Title</td>
<td><span wicket:id="title">{title}</span></td>
</tr>

<tr>
<td class="label">Description</td>
<td><span wicket:id="description">{description}</span></td>
</tr>

<tr>
<td class="label">Project</td>
<td><span wicket:id="project.title">{project}</span></td>
</tr>

<tr>
<td class="label">Status</td>
<td><span wicket:id="status.title">{status}</span></td>
</tr>
</table>

</wicket:extend>
</body>
</html>
```

Note that the text in curly braces acts as a simple visual placeholder for the content while we edit the page and serves no purpose. We call our `initPage` method to setup the page components, and since we are binding the form to a `ComponentPropertyModel` we need to give the components and the associated markup the same `wicket:id` attribute as the name of the properties they will be bound to. We also have properties on the entity that are objects (i.e. `project` and `status`) so we need to name our wicket components like we would access the property in java or JSF i.e. `project.title`. We bind our model to the top level page and the child components use that model to obtain the values based on their name. There is a logical process to locating a model for a component which involves seeing if one is attached to the component, and then searching up the component hierarchy until it finds one. In this case, it finds the compound property model and uses its component name to get the actual value. If you aren't a fan of reflection or you have some more complex needs, you can always just push the values into the component like we did for the project view page. Note though that you would also have to handle extracting the values from the components when the page was submitted.

Now let's take a look at the code for the project list page which uses the `ProjectListPage.java` class and the `ProjectListPage.html` markup. We want to fetch the list of projects, and then bind them to some table markup in the page. Let's start with the markup since it will help make the page code clearer.

Example 5.7. `ProjectListPage.html`

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:wicket="http://wicket.apache.org/">
<body>
<wicket:extend>
  <table class="dataTable">
    <tr>
      <th>ID</th>
      <th>Title</th>
      <th>Link</th>
    </tr>
    <tr wicket:id="listView">
      <td><span wicket:id="id">[ID]</span></td>
      <td><span wicket:id="title">[Title]</span></td>
      <td><a href="#" wicket:id="viewLink">view</a></td>
    </tr>
  </table>

</wicket:extend>

</body>
</html>
```

This is the markup for the project view page. It contains a HTML table with a header row, a row with a `listView` wicket id on it, and some columns. The columns contain either spans or a link with a wicket id in it. Now let us look at the page code for the project listing page.

Example 5.8. ProjectListPage.java

```
public class ProjectListPage extends BasePage {

    public ProjectListPage() {
        super();
        initPage();
    }

    public void initPage() {

        LoadableDetachableModel model = new LoadableDetachableModel() {

            @Override
            protected Object load() {
                ProjectDao dao = new ProjectDao();
                dao.setEntityManager(EMF.createEntityManager());
                return dao.listProjects();
            }
        };

        ListView listView = new ListView("listView", model) {

            @Override
            protected void populateItem(ListItem item) {

                //get the project for this row
                Project project = (Project) item.getModelObject();

                //add the components for this row
                item.add(new Label("id", project.getId().toString()));
                item.add(new Label("title", project.getTitle()));

                //create a link for the project view page
                PageParameters params = new PageParameters();
                params.add("projectId", project.getId().toString());
                item.add(new BookmarkablePageLink("viewLink",
                    ProjectViewPage.class, params));
            }
        };

        add(listView);
    }
}
```

This code has a couple of new things going on. For the model, we used a `LoadableDetachableModel` which acts as a proxy and only loads the actual data when needed by calling the `load` method. This data is detached (set to null) when the `detach()` method is called which means that we only keep our data

around as long as we need it and it is not stored with the page object, the Model proxy is stored instead which saves server session space. Once we have our model, we bind it to a `ListView` component which provides a means of iterating over a collection of items. For each row, the `populateItem` method is called and components can be added to that row.

We also created our first link in Wicket. I found linking was a little cumbersome at first given that there is now obvious way to do it. In the HTML, we create a link with an empty `href` attribute and we give it a wicket id. In the java code, we create a link component with the same component Id, a Wicket `WebPage` class, and we add the optional parameters to pass the `projectId` for that row. If you execute the application, you will see that the link generated is `projects?wicket:bookmarkablePage=:org.wicketracker.web.pages.projects.ProjectViewPage&projectId=...`. Kind of ugly huh? Thankfully, we can beautify it by mounting the page as one that can be bookmarked. In our main `WebApplication` class, we can add the following code to make our URLs more user friendly.

Example 5.9. `WicketTrackerApplication.java`

```
public class WicketTrackerApplication extends WebApplication {

    @Override
    public Class getHomePage() {
        return ProjectListPage.class;
    }

    @Override
    protected void init() {
        super.init();
        mountBookmarkablePage("/projects", ProjectListPage.class);
        mountBookmarkablePage("/projectView", ProjectViewPage.class);
        mountBookmarkablePage("/issueView", IssueViewPage.class);
    }
}
```

Now when we look at our URLs, we will get something like `http://localhost:8080/WicketIssue/projectView/projectId/3/`.

5.2. Listing Issues

Our project view page needs to have a list of the issues for that project which entails grabbing the list of issues for the project and putting them in a table. We will use a Wicket panel which are similar to facelets in that they have markup that can be re-used in several pages. Unlike facelets, there is code associated with the panel as there is with most other Wicket elements. The panel is built the same way pages are, by using markup and then code behind it binding the markup to server side components. One key difference is that typically the model is passed into the panel constructor. Usually the data displayed in a panel depends on the context of the parent page in which it is displayed. Consider a panel to edit an address where the address to be edited depends wholly on the page it is contained in (person address, company address etc). In such cases, the panel itself is not responsible for loading the data, instead it relies on the containing page to pass it the data. Of course, this is not always the case, and the required data may be generated in the panel as the situation demands (i.e. a panel that lets you search for items). In our simple case, we want to

pass in a list of Issues as the model into the constructor. However, good design with Wickets flexibility says we can create constructors (or static methods) that take a `projectId` and create a model for the issues for that project and pass that on to the proper panel constructor. We can also write something similar to promote type safety so this panel is only ever passed a list of `Issue` objects which are then wrapped in a model instance.

Example 5.10. `IssueListPanel.html`

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:wicket="http://wicket.apache.org/">
<body>
<wicket:panel>

  <table class="dataTable">

    <tr>
      <th>ID</th>
      <th>Title</th>
      <th>Status</th>
      <th></th>
    </tr>

    <tr wicket:id="issueList">
      <td><span wicket:id="id">[ID]</span></td>
      <td><span wicket:id="title">[Title]</span></td>
      <td><span wicket:id="status">[Status]</span></td>
      <td><a href="#" wicket:id="linkViewIssue">view</a><br/>
      </td>
    </tr>

  </table>

</wicket:panel>
</body>
</html>
```

Example 5.11. IssueListPanel.java

```
package org.wicketracker.web.pages.issues;

import org.apache.wicket.PageParameters;
import org.apache.wicket.markup.html.basic.Label;
import org.apache.wicket.markup.html.link.BookmarkablePageLink;
import org.apache.wicket.markup.html.list.ListItem;
import org.apache.wicket.markup.html.list.ListView;
import org.apache.wicket.markup.html.panel.Panel;
import org.apache.wicket.model.IModel;
import org.wicketracker.model.model.Issue;

public class IssueListPanel extends Panel {

    public IssueListPanel(String id, IModel model) {
        super(id, model);
        bindList();
    }

    public void bindList() {
        ListView listView = new ListView("issueList",getModel()) {

            @Override
            protected void populateItem(ListItem item) {
                Issue issue = (Issue) item.getModelObject();
                item.add(new Label("id",issue.getId().toString()));
                item.add(new Label("title",issue.getTitle()));
                item.add(new Label("status",issue.getStatus().getTitle()));

                PageParameters params = new PageParameters();
                params.add("issueId", issue.getId().toString());
                item.add(new BookmarkablePageLink("linkViewIssue",IssueViewPage.class,params))
            }

        };
        add(listView);
    }
}
```

The concept should be fairly simple. The constructor for the Panel takes a model that we then bind to our `ListView` called `issueList`. This `issueList` is referenced in the markup in the table row. In the `ListView` anonymous class, we implemented the `populateItem` method to bind the data for that row to the issue object for that row. We don't see any code to actually fetch issues because we expect the parent container to do that for us since all this panel knows is that it expects a list of `Issue` objects. However, this code doesn't enforce that rule, you could send in a list of any type wrapped in a model and it would accept it. However, Wicket allows you to enforce type safety by creating methods to accept typed parameters.

There is a small gotcha here in that you need to think ahead about the interface to these panels. For example, if the panel relied on a compound property model in the view, is it up to the panel owner (the page) to wrap the data in that model, or should the panel receive the raw objects and then wrap them in a property model? It seems that the best strategy is to use the latter since then the panel gets to create the exact model it needs, and the caller is presented with a type safe interface where it passes in (a list of) typed objects. However, this leads to the problem of scoping the model, since you are passing raw objects, how is the panel to know whether it is detachable, or whether it is to be serialized in the page? To get around this, you could pass in an `IModel` instance that will be wrapped by the panel in a compound property model, but then we lose type safety. Current versions of Wicket (1.3.6) don't have support for generics which is a shame since genericized models would be a great help in cases like this. Regardless, code needs to be documented to describe how to interface with these components.

This issue list panel can be reused in several places where we could pass in different lists of issues. Here we will use it in the project view page to show the list of issues for the project. To do this, we need to add the panel placeholder to the project view page and add the panel and pass it a model in the page code.

Example 5.12. Additional markup in `ProjectViewPage.html` for the issues list panel.

```
<div wicket:id="issues"/>
```

Example 5.13. Additional code in `ProjectViewPage.java` for the issues list panel in `initPage()`

```
IModel issuesModel = new LoadableDetachableModel() {  
  
    @Override  
    protected Object load() {  
        ProjectDao dao = new ProjectDao();  
        dao.setEntityManager(EMF.createEntityManager());  
        return dao.findIssuesForProject(id);  
    }  
};  
add(new IssueListPanel("issues", issuesModel));
```

That's it, pretty simple. This code also handles the issue of state management so the list is not saved with the page since we are using a detachable model.

5.3. Editing Projects

To edit a project, we go through the same steps as before creating an HTML page and a java class with the same name, and putting our markup in the HTML file and binding it to components in the java class. This time, we will be creating text editors and submit button components. In this section, we will use a `Form` component to handle the form submission. We will create an inner class to define the form which

will contain the edit controls and handle the submission. In the form submission, we will save the updates to the `Project` instance.

Example 5.14. `ProjectEditPage.html`

```
<html>
<body>
<wicket:extend>
  <form wicket:id="form">
    <table>
      <tr>
        <td class="label">ID</td>
        <td><span wicket:id="id">{id}</span></td>
      </tr>

      <tr>
        <td class="label">Title</td>
        <td><input wicket:id="title" /></td>
      </tr>
    </table>
    <input type="submit" value="Save" wicket:id="saveButton"/>
    <input type="submit" value="Cancel" wicket:id="cancelButton"/>
  </form>

</wicket:extend>
</body>
</html>
```

Example 5.15. ProjectEditPage.java

```
public class ProjectEditPage extends BasePage {

    private final Long id;
    private final EntityManager entityManager;
    private final Project project;

    public ProjectEditPage(PageParameters parameters) {
        id = ParamUtils.getLongObject(parameters, "projectId");
        entityManager = EMF.createEntityManager();
        if (id == null) {
            project = new Project();
        } else {
            project = entityManager.find(Project.class, id);
        }
        initPage();
    }

    public void initPage() {
        IModel propModel = new CompoundPropertyModel(project);
        add(new ProjectEditForm("form", propModel));
    }
}
```

We are keeping hold of the instance of the `Project` object and the `EntityManager` instance in the page object. This makes it easier to handle persisting the entity back to the same entity manager without attachment issues. This is a questionable practice since this page will probably be serialized and it would include the entity manager which may or may not be serializable. There may also be issues in a replicated environment where replicating the entity manager may have unintended consequences. We use a `CompoundPropertyModel` assigned to the form to set the values of the individual editors (`Id` and `Title`). The form component is set up server side using an inner class called `ProjectEditForm`.

Example 5.16. ProjectEditForm inner class in the ProjectEditPage class.

```
class ProjectEditForm extends Form {

    public ProjectEditForm(String id, IModel model) {
        super(id, model);
        add(new Label("id"));
        add(new TextField("title"));
        Button saveButton = new Button("saveButton");
        Button cancelButton = new Button("cancelButton") {
            @Override
            public void onSubmit() {
                setResponsePage(ProjectListPage.class);
            }
        };
        cancelButton.setDefaultFormProcessing(false);

        add(cancelButton);
        add(saveButton);
    }

    @Override
    protected void onSubmit() {
        super.onSubmit();

        //get the project from the model
        Project project = (Project) getModelObject();
        ProjectDao dao = new ProjectDao();
        dao.setEntityManager(entityManager);
        dao.saveProject(project);

        //redirect to the project view page
        PageParameters params = new PageParameters();
        params.add("projectId", project.getId().toString());
        setResponsePage(ProjectViewPage.class, params);
    }
}
```

Another way we could have done this is by not using an inner class and reproducing this code in the main page class, and adding the components to a `Form` instance we created. We can demonstrate this in the `IssueEditPage.java` class.

For the issue edit page, our markup is pretty much the same as you would expect, text editors placed inside a form with a save and cancel button. The HTML hierarchy follows the same structure as the correlating components in the java class. The page at the top with a child form, and the id, title etc.. as child components.

Example 5.17. IssueEditPage.html

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:wicket="http://wicket.apache.org/">
<body>
<wicket:extend>
  <form wicket:id="form">
    <table>
      <tr>
        <td class="label">ID</td>
        <td><span wicket:id="id">{id}</span></td>
      </tr>

      <tr>
        <td class="label">Title</td>
        <td><input wicket:id="title" value="{title}" /></td>
      </tr>

      <tr>
        <td class="label">Description</td>
        <td><input wicket:id="description" value="{description}" /></td>
      </tr>

      <tr>
        <td class="label">Project</td>
        <td><span wicket:id="project.title">Project.title</span></td>
      </tr>

      <tr>
        <td class="label">Status</td>
        <td><input wicket:id="status.title" value="{status}" /></td>
      </tr>
    </table>
    <input type="submit" value="Save" wicket:id="saveButton" />
    <input type="submit" value="Cancel" wicket:id="cancelButton" />
    <a href="projects">Back To Projects</a>
  </form>
</wicket:extend>
</body>
</html>
```

For now, I am leaving the project and status drop down selections as text fields. We'll look at turning them into drop down boxes later. Here we are creating the form and attaching the child components within our page instance as opposed to using an inner class.

```
//create the model around the issue object
IModel model = new CompoundPropertyModel(issue);
                                The Wicket Factor
```

```
//create our form instance
Form form = new Form("form",model) {
Example 5.18: IssueEditPage.java
    protected void onSubmit() {
        IssueDao dao = new IssueDao();
        dao.setEntityManager(entityManager);
        dao.save(issue);
        setResponsePage(IssueViewPage.class,new PageParameters("issueId="+issue.getId())
    }
};

//add the form components
form.add(new TextField("title"));
form.add(new TextField("description"));
form.add(new Label("id"));
form.add(new Label("project.title"));
form.add(new TextField("status.title"));

//create the form buttons
Button saveButton = new Button("saveButton");

//when cancelling, we check to see if it was a new issue
//if so, we go to the project page
//otherwise, we go to the issue view page
Button cancelButton = new Button("cancelButton") {
    @Override
    public void onSubmit() {
        super.onSubmit();
        PageParameters params = new PageParameters();

        if (issue.getId() == null) {
            //goto project view since this issue doesn't exist
            params.add("projectId", projectId.toString());
            setResponsePage(ProjectViewPage.class,params);
        } else {
            //goto issue view page
            params.add("issueId", issue.getId().toString());
            setResponsePage(IssueViewPage.class,params);
        }
    }
};

//we don't want to handle the submit on clicking the button
cancelButton.setDefaultFormProcessing(false);

//add the buttons to the form
form.add(saveButton);
form.add(cancelButton);

//add the form
add(form);
}
```

That's a lot of code, but it actually does quite a bit. We perform some real-world validation on the input parameters and if it fails we immediately jump to another page and pass an error message. If the user cancels the editing, we jump to either the issue view page if this was an existing Issue, or to the project page if this was a new issue (and therefore there is nothing to view). Compared to the XML based navigation of JSF this appears nice and tidy as well as being flexible enough to handle complex cases with java code. Furthermore, we didn't have to jump to different files to handle the different pieces of the process. If we get to a point where a page needs multiple navigation rules depending on how it is used, we can implement some additional code later, even in a separate class if needs be and refactor our existing code. Also, rather than create a new inner class, we created our own `Form` instance, bound it to the model and added components to it in the page itself. The `defaultFormProcessing` on the button is similar to the `immediate` attribute on JSF buttons. When this is turned off, Wicket skips the conversion, validation and model updating stages of the request process.

5.4. Enhancing The Application

So far, we have handled the basics of a web application. Loading, displaying, editing and saving data. The Seam vs Spring Web Flow articles also tackle more complex issues such as stateful navigation. The main test for this was the ability to edit an object (like the issue), go to another page to select a project and then come back to the issue editor with the original issue changes intact and the new project selected. The goal was to see if the frameworks would provide us with high level functionality without forcing us to develop unscalable and complex low level solutions.

5.4.1. Entity Drop Downs

First though, let's start by creating drop downs from entity objects and bind them to an object property and we'll see if Wicket handles the conversion and update easily and correctly. We'll start with fetching the list of status values in the `IssueEditPage` .

Example 5.19. Adding issue status drop downs IssueEditPage.java .

```

public class IssueEditPage extends BasePage {

    ...
    ...
    private final List<IssueStatus> statuses;

    public IssueEditPage(PageParameters parameters) {
        ...
        ...
        //load the list of issue status values
        IssueDao dao = new IssueDao();
        dao.setEntityManager(entityManager);
        statuses = dao.listStatuses();

        initPage();
    }

    public void initPage() {
        ...
        ...
        DropDownChoice statusSelect = new DropDownChoice("status", statuses);
        form.add(statusSelect);
        ...
        ...
    }
}

```

We removed the original component for `status.title` and replaced it with the above drop down choice. We also changed the HTML markup to use a select.

Example 5.20. HTML to use a select instead of text in IssueEditPage.html

```
<select wicket:id="status" value="{status}" />
```

Because we are using the compound model in the form which references the issue, we need to refer to the `status` property of the `Issue` object. This code will give us the ability to show the issue state in a drop down and when we save it, write the value back to the issue's status property because we are using a `ComponentPropertyModel`.

Figure 5.1. Drop downs using Wicket

As you can see, it works, almost. We need to display the correct text in the drop down as we currently display the `toString()` value. Looking at the options for creating a `DropDownChoice`, we can also pass in an `IChoiceRenderer`. We add that as an anonymous class to the constructor. Obviously, the

better solution is to creating a class that implements this interface for this type so it can be reused anywhere we need to display the issue status information.

Example 5.21. Adding the `IChoiceRenderer` to `IssueEditPage.java`

```
DropDownChoice statusSelect = new DropDownChoice("status", statuses, new IChoiceRe

    public Object getDisplayValue(Object object) {
        return ((IssueStatus)object).getTitle();
    }

    public String getIdValue(Object object, int index) {
        return ((IssueStatus)object).getId().toString();
    }

});
```

Figure 5.2. Wicket drop downs with properly titled choices

That was pretty painless and avoided the problems of having to wrap the objects in a `DataModel` as JSF does. When we change values, and save the issue, the correct value is put into the status property thanks to the `CompoundPropertyModel`.

5.4.2. Selecting Projects

Now let's see how we can navigate from the issue edit page, select a project and go back to the issue edit page to resume editing the issue. First let's consider how we do this since we are completely on our own on how we implement it. We want to navigate to the project page and when we select a project, put it into the `project` property on the issue we are editing and return to the issue edit page. This is all while keeping any changes we made to the issue before we navigated away to the project page and back. Let's look at the project list page first to see how this might be done. We want to add a column containing a select link in the list of projects in the project list page to select that project. In the backing code, we add this link component and give it an on click handler which calls a method to handle the selection. The `handleSelect` method is empty and will be filled in by the calling page since only that page knows what to do when you select an entity.

Example 5.22. Adding the select link in the ProjectListPage.java .

```
public void initPage() {

    LoadableDetachableModel model = new LoadableDetachableModel() {
        ...
    };

    ListView listView = new ListView("listView", model) {
        @Override
        protected void populateItem(ListItem item) {
            final Project project = (Project) item.getModelObject();

            item.add(new Label("id", project.getId().toString()));
            item.add(new Label("title", project.getTitle()));
            ...
            ...
            //Adding the select link here
            item.add(new Link("select") {
                @Override
                public void onClick() {
                    log.debug("Submit clicked on project");
                    handleSelect(project);
                }
            });
        }
    };
    add(listView);
}
```

In our ProjectListPage.html page we add the link into a new column in the table showing the list of projects.

Example 5.23. ProjectListPage.html

```

<table class="dataTable">
  <tr>
    <th>ID</th>
    <th>Title</th>
    <th>Link</th>
    <th>Select</th>
  </tr>
  <tr wicket:id="listView">
    <td><span wicket:id="id">[ID]</span></td>
    <td><span wicket:id="title">[Title]</span></td>
    <td>
      <a href="#" wicket:id="viewLink">view</a>
      <a href="#" wicket:id="editLink">edit</a>
    </td>
    <td>
      <!-- New Select Link Goes Here -->
      <td><a href="#" wicket:id="select">Select</a></td>
    </td>
  </tr>
</table>

```

Going back to our issue editor page, we add a button next to our project display text to invoke the project selection.

Example 5.24. Adding the project change button to our IssueEditPage.html .

```

<tr>
  <td class="label">Project</td>
  <td>
    <span wicket:id="project.title" >{project}</span>
    <input type="submit" wicket:id="select" value="Change"/>
  </td>
</tr>

```

We then add the corresponding "Change Project" button component in the `IssueEditPage.java` code. In the `onSubmit()` method of the button we set the response to point to an instance of the project list page. We pass an instance that we create here as an anonymous class because we want to override the `handleSelect` method in the `ProjectListPage` class to set the `project` property on the issue entity in this page. After assigning the project, we set the response page to this instance of the `IssueEditPage` class. This causes us to end up back on the original page that we called the project list page from with our original changes to the issue object. To spice things up a bit I put in some validation so you couldn't select the project with an id of 3 just to see how easy it is to validate that selection and stay on that same page while passing a message back to the user.

Example 5.25. Adding the project selection button in `ProjectEditPage.java`.

```

Button changeProject = new Button("btnSelectProject") {

    @Override
    public void onSubmit() {
        setResponsePage(new ProjectListPage() {
            @Override
            public void handleSelect(Project entity) {
                if (entity.getId() == 3) {
                    getSession().error(
                        "cannot select project with Id 3");
                } else {
                    issue.setProject(entity);
                    setResponsePage(ProjectEditPage.this);
                }
            }
        });
    }
};

changeProject.setDefaultFormProcessing(false);
form.add(changeProject);
add(form);

```

You may have noticed is that the select link is always displayed on the project list. In the Seam and Spring JSF versions, we always hid it if there was no selection possible. The reason I have not done so here is because there are ways to do it by componentizing your table and columns which I didn't get in to here. However, the solution is relatively trivial, but requires replacing the table with a `DataTable` component and defining your columns in code. The `DataTable` is part of the wicket extensions.

This solution for handling more complex navigation works, and it works really well. It doesn't involve a mess of HTML or XML, just some Java code. Granted, a rigid implementation like this can soon get complex if you need something more sophisticated, however, Wicket makes it easy to knock up a more generic version in a small amount of time. This mechanism is re-usable, and the project list page knows nothing about the calling page which is how we want it. One subtle gem here is the fact that unlike the managed environments of Seam and Spring, you can just create a new anonymous class and override methods to change functionality. This isn't possible with Seam and Spring because they are managed environments and you never create your own components, therefore such dynamic interactions can be harder to implement leaving you to hard wire the relationships between the issue editor and the project selection.

5.5. Summing Up Wicket

Wicket is a really good framework to use, it lays out a minimalist set of ground rules and gives you a great environment to work in. The first important aspect is the ability for Wicket to hold page state which enables you to hold objects between requests. This is a fundamental piece of both Seam and SWF, but Wicket offers this in a simplified fashion where only the page is held as opposed to whatever conversational objects you invoke as part of the flow, as well as incidental ones created by the framework (just look at the list of objects in a conversion in the `seam.debug` page).

Wicket HTML markup consists of markers into which Wicket will place the component HTML. This makes editing the view straightforward and doesn't push too much decision making and certainly no business logic into the view. On the back end, in the page object we add the components, couple them to models and set property values on the components that alters the final HTML. None of this is controlled from the view HTML, it is all done on the java server side. This server side only approach makes component instantiation easier and more natural but can be verbose. You gain the benefit of IDE features which apply to strongly typed Java code such as static typing, code completion, refactoring as well as renaming. To access components in JSF you define a component in a web page, and it can optionally have a binding to a bean property which will hold the JSF component reference on the server side. Wicket just holds the components as a plain old java variable (pojv?) which is natural and efficient without the need to lookup EL component bindings, but, this comes at the expense of having to explicitly declare each component declaration, creation and binding in code. It has been claimed that you can see Wicket markup by opening up the HTML file which is true but since you should be using templating and probably runtime specific stylesheets (i.e, theming) it would be hard to get a good idea of what the page looks like.

Other frameworks (I'm mostly considering Seam and SWF here) have far more rules to obey and they are somewhat more contextual in that they only apply under certain circumstances. They do deliver far more in terms of features, but following the 80/20 principle, 20% of the features will be used 80% of the time, and simplified versions of that can be written in Wicket. I'm not suggesting that you need to consider implementing Drools or full conversion support in Wicket. You could however code individual pages to share objects across pages and requests which is mostly what conversations are used for. However, Wicket does provide built in support for providing pagination components and handling it on the back end in the model and these do work well.

I found Wicket to be fairly simple to use, and rarely had to go look at any reference documentation once the underlying principles were understood. I had a little extra work to look up info related to links, templating and displaying data tables, but beyond that, most things could be found using code completion to give me an idea of what I was looking for.

I have a couple of solid concerns and one 'philosophical' concern which is that Wicket really lets you 'explore the studio space', and some people will take this and build huge monuments to good coding practices and OO design. Other developers will take this and create spaghetti code. Regardless, Wicket gives you free will to do with your code what you will, and when you start using it, you'll start spotting all sorts of places where you can invoke the DRY principle (Don't Repeat Yourself) and reduce code repetition. The inverse is that the project might become a victim of Astronaut Architects or code freeze as you try and define the perfect architecture on top of Wicket. Another aspect of this is that any discussion of best practices for Wicket cannot exclude a discussion of best practices for OO code such as using anonymous versus inner versus top level classes, composition over inheritance, and use of GoF design patterns. This is a huge (endless?) discussion which could detract from actually getting things done. The Seam and SWF frameworks expect and direct the boilerplate code to be handled a certain way which as long as it works well is a benefit. Yes, I'm saying that too much freedom can be a bad thing.

Another problem is the potential for abusing the session which is a noted concern by critics of Wicket. Common sense should be applied so you don't hold unnecessary data in the page that ends up being serialized in the session. Using detachable models can help by reloading the data each time the page is rendered. However, this same problem will exist in all stateful frameworks, and caution is required by any framework that handles state in the session. There is also some chatter of problems when holding references to other pages and handling circular references to pages upon serialization. Passing page references around may make stateful coding easier, but it seems that it too could come back to haunt you. Seam and SWF store the state in conversations which time out and are destroyed. While this allows any number of conversations, because they are managed, all the pieces of the conversation time out and are destroyed together and it can help control session sizes.

While the current version of Wicket (1.3.6) doesn't use generics, version 1.4 which is now in release candidate 6 at the time of writing does use generics. However, one problem is the incompatibility between

applications using Wicket 1.3.x working with Wicket 1.4. On this basis, you would need to think hard on whether to start with the current stable release even if it is incompatible with the next version which is currently only a release candidate.

JSF , Seam and SWF all depend on EL expressions (or variants of) which can get expensive when you have an expression with multiple elements that need evaluating especially since JSF could repeat the evaluation more than once in a page. In Wicket, you are typically binding the actual object to a model which uses reflection on the object to obtain property values for reading and writing. This is far more efficient since it doesn't have to repeatedly lookup the initial object in a massive dictionary of available objects several times a page.

Wicket appears to be much faster than the JSF based frameworks. No doubt this is partly because of the expense of JSF and EL as well as the bundling of all the other features in the frameworks. Another 'feature' if you want to see it that way is that it can be used with Google App Engine for Java unlike JSF (and therefore Seam and Spring Faces). Web Beans also comes with an example that uses Wicket and GAE, or you can also use Spring with GAE.

Wicket does lack some of the functionality and flexibility of Seam and SWF . Seam offers JSF based email and pdf and excel documents from JSF documents by using the already defined data components letting you do more with what you have already built. Both SWF and Seam come with Dependency Injection capabilities whereas Wicket does not although there are add-ons for Spring, Google Guice, injecting EJBs and even Web Beans. Wicket has a disconnect with injection frameworks since it is an unmanaged and (semi-?)stateful framework. The fact that page instances can be serialized means we need special handling for references to managed beans so we don't serialize and replicate them. However, there is a very capable work around with the spring annotations extensions. Wicket also comes AJAX ready with some easy to use AJAX controls and an AJAX debugger so you can see what messages are going on behind the scenes (a very nice idea).

One important note is that by using POJOs in an unmanaged environment we can very easily subclass web pages or create anonymous class instances with different handlers for different events. We can't just create a new anonymous Spring Bean or Seam component out of thin air and augment it with new features. This is a huge plus for Wicket since this flexibility lets you extend classes in ways such that the base class remains ignorant of the context it is being used. For example, a search and select popup panel should know nothing about the recipient of the selection (the parent page). To get such features in Seam or Spring, you need to rely on the workflow engine to act as the middle man between the different pieces.

Overall, Wicket is a great framework to use, and I hope it really receives the success it deserves as a great lightweight component framework.

5.5.1. Choosing Between Them

As I was thinking about this, I started comparing the frameworks in terms of different languages :

- Wicket = Assembler - Small, powerful, but has lengthy code that quickly gets messy if you aren't able to organize OO code well. Little stands between you and the bare metal and this gives you a lot of control. To get more complex things done, you need to build on top of it to create a more advanced framework.
- Spring Web Flow = C++ - It's fast, it's popular, (Spring at least is already in widespread use), but some pieces don't fit well with others at times. A robust but no frills development experience.
- Seam = Delphi - Almost as capable as C++, and covers a broader range of problems well. It's not as popular, but very easy to use with excellent tooling and all round integration. Solves a number of problems inherent in the other solutions. Mildly slower than C++ (at least early on, not so much later).

They all have different appeals and benefits from different perspectives. If you are looking at frameworks in terms of career, looking at job listings, JSF is leading most web frameworks aside from Struts. Seam may

be slow in picking up momentum, but it is ahead of Wicket and SWF. SWF is helped by massive Spring adoption while Seam is almost alien to existing EJB or POJO based applications. As far as popularity goes, Spring Faces is probably the better option with the JSF/Spring/Spring Web Flow combination looking good on a resume. This is interesting, because while I think this is a great option, it has a couple of fairly deep flaws that the others don't which would put them ahead of SWF.

If I were looking at it from the perspective of a single person startup for a small project (which I'm also considering), then this introduces a few more issues. If you want to keep hosting cheap, perhaps even running on something like Google App Engine, then JSF is problematic as not only does 1.2 not run on GAE, but it would probably chew up more resources (on any host) than you would like. For me, Wicket would be the choice there as it is more lightweight and session size is less of a concern given that you have a little more control. Obviously, depending on your needs, Wicket could be a winner in larger projects and isn't limited to smaller projects. With a larger team and a more complex project, more groundwork is needed to standardize the coding practices so you don't have one too many differing coding styles. JSF and Seam/SWF pretty much direct you down that path with one most obvious way of doing most things. Wicket is also useful where alternatives might be overkill such as a shopping cart type app with little state management and user CRUD operations. I really like the Wicket programming model, I just don't think it is applicable in all cases (just like assembly language isn't).

Lastly, I think Seam leads in terms of features and productivity in a cohesive package. I can start a new project and easily code CRUD pages in a matter of minutes with just a few lines of code (mostly XHTML). I can then extend that to provide emails, pdfs and excel spreadsheets, ajax, Seam remoting, and web services using the same backing beans. I don't have to worry about state management, OSiV, Lazy Initializations, I can introduce workflow any time I want and I don't have to worry about pages where I don't want to use workflow and having to come up with an alternate method of getting to my data. This is all wrapped up in a nice cool IDE that is so good I use it with Wicket and Spring Web Flow development. While it has a number of blemishes, most of them can be worked around, and while it has a steeper learning curve, it's not impossible to learn, and once you get it, the productivity can be quite a pay off. Interestingly enough, I have trained people to use Seam whom I cannot imagine would be as quick to pick up the nuances of good OO design in order to work with Wicket.

Lastly, I think it will be interesting to see how the choice of stateful and semi-stateful web frameworks will grow. There is a beta version of Conversations for Tapestry [<http://docs.codehaus.org/display/TRAILS/Conversations+in+Trails>] , and Web Beans (JSR 299) [<http://jcp.org/en/jsr/detail?id=299>] will provide a common framework for handling conversation scoped beans as well as a standard for (conversational) dependency injection. It will be interesting to see what new solutions arise from these and how they differ and improve on these pioneering stateful frameworks.