

Better, Faster, Lighter Java™



O'REILLY®

Bruce A. Tate & Justin Gebtland

Extending jPetStore

The previous chapter introduced the workhorse Simple Spider service with its console-based user interface and web service endpoint. In this chapter, we see how easy it is to add the Spider to an existing application, jPetStore. Some might argue the jPetStore already has a search tool; but that tool only searches the database of animals in the pet store, not all the pages on the site. Our customer needs to search the entire site; jPetStore has at least one page in the current version that isn't searchable at all (the Help page) and text describing the different animals that doesn't show up in a query.

We'll add the Spider to the jPetStore, paying careful attention to what we need to change in the code in order to enable the integration. In addition, we will replace the existing persistence layer with Hibernate. By carefully adhering to our core principles, our code will be reusable, and since the jPetStore is based on a lightweight framework (Spring), it doesn't make unreasonable demands on our code in order to incorporate the search capability or the new persistence layer. Coming and going, the inclusion will be simple and almost completely transparent.

A Brief Look at the Existing Search Feature

The search feature that comes with jPetStore takes one or more keywords separated by spaces and returns a list of animals with a name or category that includes the term. A search for "dog" turns up six results, while a search for "snake" nets one. However, a search for "venomless" gets no results, even though animal EST-11 is called the Venomless Rattlesnake. Even worse, none of the other pages (such as the Help page) shows up in the search at all; neither will any other pages you might add, unless they're an animal entry in the database.

The search feature has the following architecture (shown in Figure 10-1):

1. Any page of the jPetStore application may contain a search entry box with Search button.
2. Clicking the button fires a request (for `/shop/searchProducts.do`) passing the keywords along as part of the request.

3. *petstore-servlet.xml*, the configuration file for the MVC portion of the jPetStore Spring application, has the following definition:

```
<bean name="/shop/searchProducts.do"
class="org.springframework.samples.jpetsotre.web.spring.SearchProductsController">
  <property name="petStore"><ref bean="petStore"/></property>
</bean>
```

This creates a handler for the “/shop/searchProducts.do” request and maps it to an instance of *SearchProductsController*, passing along an instance of *petStoreImpl* called *petStore*.

4. *SearchProductsController* instantiates an instance of a class that implements the *ProductsDao* interface, asking it to search the database for the specified keywords.
5. *ProductsDao* queries the database and creates an instance of *Product* for each returned row.
6. *ProductDao* passes a *HashMap* containing all of the *Product* instances back to *SearchProductsController*.
7. *SearchProductsController* creates a new *ModelAndView* instance, passing in the name of the JSP page to display the results (*SearchProducts*) and the *HashMap* of values. The JSP page then renders the results using the *PagedListHolder* control (a list/table with built-in paging functionality).

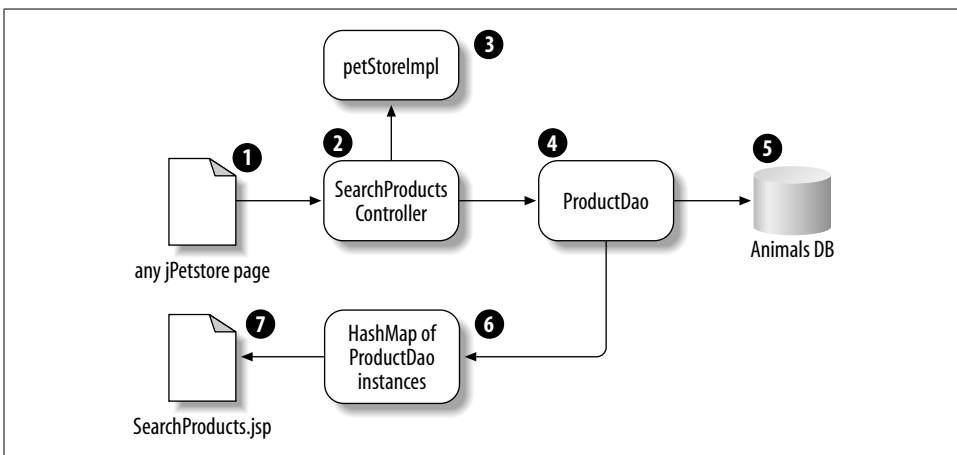


Figure 10-1. The original jPetStore search architecture

Only the *ProductsDao* knows how to interact with the underlying data. *Product* is a straightforward class with information about each product, and the view (*SearchProducts.jsp*) simply iterates through the returned results to create the output page.

Deciding on the Spider

We've identified how the current search feature works and its limitations: the search feature only searches products in the database, not the site as a whole, and even then it doesn't search all available data about the products. The results it returns are extremely limited—though well-formatted.

The Simple Spider is a crawler-based search feature instead of focusing on the database: it searches everywhere on the site, not just the products table, and it treats any textual information visible to users as part of the search domain. The Spider does have a major limitation—since it is based on a web crawler, it can only catalog pages linked to other pages on the site. If a page is only accessible via some server-side logic (for instance, selecting a product from a drop-down list and submitting the form to the server, which returns a client-side or server-side redirect), the crawler never reaches that page and it won't be part of the search.

With a problem like this, in which a feature of the application is too limited to be of much service to our users, we have to decide between refining the existing service or replacing it entirely. The limitation of the `jPetStore` search is partly due to the fundamental nature of the service (it searches the database, not the site). Refining it to accomplish the full-site search would be horribly inefficient. The Spider is the obvious solution, but we must consider what we are already dealing with (remember, you are what you eat). If `jPetStore` uses a lot of server-side logic to handle navigation, the Spider simply won't be able to provide a complete catalog. In this case, though, all the navigation on the site is handled client-side, so the Spider is a perfect fit for solving our problem and coexisting with our current application.

Extending `jPetStore`

We have decided that an existing service layer of the application is unsuited to our current needs. Additionally, we have decided that replacing the service with a new one is the appropriate solution. This situation is a perfect test of extension: how easy will it be to replace this service? Will it involve new code? Changes to existing code? Or just changes to our configuration services?

In order to replace the existing functionality with the Simple Spider, we need to change the output formatting a little (our returns will display full URLs instead of product instances), write a new controller that knows to launch the Simple Spider instead of the `ProductsDao` object, and change our mapping layer to point to the new controller. Finally, we'll use Spider's configuration service so Spider works better with the new web site.

Looking at these requirements, we can already see we'll need to write fewer than 100 lines of code and make only minor configuration changes in order to get this to work. It's a reasonable price to pay for the end result we want. Because `jPetStore`

and the Simple Spider were designed to allow for extension in the first place, they fit together well with minimal work.

Conversely, we could write much less code and in fact do almost no work at all if we chose to connect to the Spider through the existing web service interface rather than integrating it directly with the `jPetStore`. Since the web service interface already exists, it might be construed as a violation of the “do one thing, and do it well” principle to add another, seemingly useless interface. In this instance, though, the added cost of sending a bloated message (XML/SOAP) over a slow transport mechanism (HTTP) is too heavy, especially given the minimal amount of work it will take to get a faster, more efficient integration.

Replacing the Controller

First, let’s replace the `SearchProductsController`. Here’s the main method of that class:

```
public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
                                response) throws Exception {
    if (request.getParameter("search") != null) {
        String keyword = request.getParameter("keyword");
        if (keyword == null || keyword.length() == 0) {
            return new ModelAndView("Error", "message",
                "Please enter a keyword to search for,
                then press the search button.");
        }
        else {
            PagedListHolder productList = new PagedListHolder(
                this.petStore.searchProductList(keyword.toLowerCase()));
            productList.setPageSize(4);
            request.getSession().setAttribute(
                "SearchProductsController_productList", productList);
            return new ModelAndView("SearchProducts", "productList",
                productList);
        }
    }
    else {
        String page = request.getParameter("page");
        PagedListHolder productList = (PagedListHolder)
            request.getSession().getAttribute("SearchProductsController_productList");

        if ("next".equals(page)) {
            productList.nextPage();
        }
        else if ("previous".equals(page)) {
            productList.previousPage();
        }
        return new ModelAndView("SearchProducts", "productList", productList);
    }
}
```

The method returns a new instance of `ModelAndView` and Spring uses it to determine which JSP to load and how to wire data up to it. The method takes an `HttpServletRequest` and `HttpServletResponse` in order to interact directly with the HTTP messages.

The first thing the method does is make sure the user entered a search term. If not, it displays an error to the user; if so, it creates a `PagedListHolder` called `productList` with a maximum page size (number of rows per page) set to four. Finally, it calls the `petStore` instance's `searchProductList` method, which calls to `ProductsDao` and finally returns the `HashMap` of `Product` instances. The second clause is for when the user clicks the Next Page or Previous Page buttons on the paged list.

Rewrite or Replace?

The next question a conscientious programmer should ask is, does it make more sense to rewrite this class to make use of the Spider, or to write an entirely new controller? In order to answer that question, we need to consider three more-specific questions first:

1. Do we have access to the original source? Now that we have the `jPetStore` application, do we control the source, or is it all binary? If we don't control the source, we can short-circuit the rest of the decision. We can only replace the class; we can't rewrite it.
2. Will we ever need to use the original service again? Assuming we have the source and *can* rewrite the class, can we foresee ever needing to revert to or make use of the database-search functionality? For the sake of flexibility, we usually want to retain old functionality unchanged, which means we want to replace, not rewrite. However...
3. Does the current class implement an easily reused interface? If we are going to replace the class, how much work will we have to do to get the application to recognize and accept your new class? Think of this as an organ transplant; how much work and medication has to go into the host body to keep it from rejecting the new organ? Will our changes be localized around the new class or more systemic?

Here's the answer to these questions: yes, we have the source code; yes, we'll want access to retain the potential for using the old service; and yes, the controller implements a very simple interface. The controller only needs to implement a single method, `handleRequest`, which takes an `HttpServletRequest` and a `HttpServletResponse` and returns a `ModelAndView`. This means the `jPetStore` application doesn't need any systemic changes in order to use our new controller, as long as we support that interface.

Implementing the Interface

To replace this class, we're going to write our own controller class called `SearchPagesController`. It must implement the `Controller` interface, which defines our `handleRequest` method.

```
public class SearchPagesController implements Controller {
    ...
}
```

Here's our controller's `handleRequest` method:

```
public ModelAndView handleRequest(HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
    if (request.getParameter("search") != null) {
        String keyword = request.getParameter("keyword");
        if (keyword == null || keyword.length() == 0) {
            return new ModelAndView("Error", "message", "Please enter a
                keyword to search for, then press the search button.");
        }
        else {
            ConfigBean cfg = new ConfigBean();
            String indxpath = "";
            try
            {
                indxpath = cfg.getCurIndexPath();
            }
            catch(Exception ex)
            {
                return new ModelAndView("Error", "message",
                    "Could not find current index path.");
            }

            QueryBean qb = new QueryBean(indxpath, keyword, "contents");

            qb.execute();

            HashMap hits = new HashMap(qb.getResults().length);
            for(int i =0;i<qb.getResults().length;i++)
            {
                hits.put("hits", qb.getResults()[i]);
            }
            return new ModelAndView("SearchProducts", hits);
        }
    }
}
```

For our search functionality, we won't use the paged results. We simply list all the results on a single page; as a result, we don't have to deal with the `Next Page` and `Previous Page` code. Our controller again checks for null keywords and returns an error if it finds them empty. Otherwise, the service is used almost identically as the console application in the Chapter 9 was used. First, create an `instead` of `ConfigBean`

to find the most current index of the site, then create a `QueryBean` based on that index path. Finally, execute the query and put all the `HitBean` instances into a `HashMap` to return to the `View`.

The usage pattern is identical to that in the last chapter; the only difference is the format of our returned data. Instead of passing the native array of `HitBeans` back, the `ModelAndView` object requires a `HashMap`. It's easy enough to create the one from the other, and now we have an entirely new access point for the `Spider` application with almost no work.

There is one last detail we need to work out. The original `SearchProductsController` has a field called `petStore` of type `PetStoreFacade` that the Spring framework populates for it. In order to be a complete replacement for the original, our new controller needs to expose the same property and accessor methods, even though they aren't officially found on a standalone interface anywhere in the application. You will often find examples of this when you're extending or modifying an application.

```
private PetStoreFacade petStore;

public void setPetStore(PetStoreFacade petStore) {
    this.petStore = petStore;
}
```

Registering Our New Class with jPetStore

Finally, we alert `jPetStore` to the new controller's existence. If `jPetStore` is not coded for extensibility, we have to modify the application code in order to get it to work. For instance, if there are methods of `jPetStore` that create instances of `SearchProductsController` directly, we must change each of those lines to create a `SearchPagesController` instead.

It turns out, however, that `jPetStore` is quite ready for extensibility—partly because it is based on the Spring framework. In order to tell `jPetStore` about our new controller, we modify a single configuration file (*petstore-servlets.xml*). This file tells Spring what objects to create and how to wire them together to make a sensible application. Now, we just need to find the configuration setting used to launch the `SearchProductsController` and point it to our new `SearchPagesController` instead.

```
<bean name="/shop/searchProducts.do"
      class="org.springframework.samples.jpetsy.web.spring.SearchPagesController">
    <property name="petStore"><ref bean="petStore"/></property>
</bean>
```

We're telling the application to map requests for `"/shop/searchProducts.do"` to a new instance of `SearchPagesController`. At the same time, we tell it, provide the `SearchPagesController` with the current instance of `petStore` (in a property called `petStore`).

Principles in action

- Keep it simple: the controller logic is a simple invocation of Spider; the controller interface is *very* simple (one method)
- Choose the right tools: Spring and the Spider
- Do one thing and do it well: since the Spider is so well-encapsulated, it's easy to add to an existing service; the controller deals with invoking the Spider and the JSP only needs to display the results—MVC pattern well-demonstrated
- Strive for transparency: the site doesn't care how it is indexed; it can easily switch between data-driven and HTML-driven search technologies
- Allow for extension: we quickly expanded our search capabilities by adding a new tool with minimal code; the configuration abilities of jPetStore allow for no-code recognition of new service

The User Interface (JSP)

The user interface is fairly straightforward. Instead of just dumping our results to the console or creating an XML document of the results (as in the web service implementation from Chapter 9), this time we need to write a JSP that iterates over the results and displays them as hyperlinks in a table.

The original jPetStore search feature used a `PagedListHolder` for its results because it displayed the image associated with each returned product in the table. Since the images were arbitrary in size, jPetStore didn't want to display too many entries on a given page since it might result in a lot of vertical scrolling. Our results consist of a hyperlink to the returned URL and the relative rank of the given result; therefore, we'll use a simple table to display our results.

Again, we are faced with the rewrite-or-replace question. Just like last time, we have three questions to consider:

1. Do we have access to the original source? We must, since JSPs are just text files in both development and deployment mode.
2. Will we ever want to reuse the existing service? We do, but in this case, a JSP is so easy to recreate that it won't make much difference.
3. Does the current version implement some standard interface? Not as such, since JSPs are just mixes of static HTML and dynamic content.

Because of the rather trivial nature of the changes and because JSPs are easily edited in place (no compilation necessary), we'll just repurpose the existing `SearchProducts.jsp` file. This strategy saves us from having to change any more configuration settings:

```
<%@ include file="IncludeTop.jsp" %>

<table align="left" bgcolor="#008800" border="0" cellspacing="2" cellpadding="2">
<tr>
```

```

        <td bgcolor="#FFFF88">
        <a href="<c:url value="/shop/index.do"/>">
            <b><font color="BLACK" size="2"> &lt;&lt; Main Menu</font></b>
        </a>
        </td>
    </tr>
</table>
<table align="center" bgcolor="#008800" border="0" cellspacing="2"
    cellpadding="3">

    <tr bgcolor="#CCCCCC"> <td><b>URL</b></td> <td><b>Rank</b></td> </tr>
    <c:forEach var="page" items="${hits}">
        <tr bgcolor="#FFFF88">
            <td><a href="<c:out value="\${page.url}"/>">
                <c:out value="\${page.url}"/></a>
            </td>
            <td>
                <c:out value="\${page.score}"/>
            </td>
        </tr>
    </c:forEach>
</table>

<%@ include file="IncludeBottom.jsp" %>

```

The JSP files in the application have a standard header and footer defined in *IncludeTop.jsp* and *IncludeBottom.jsp*. All we have to do is render the results in between the include directives. Start by creating a JSP-style `forEach` loop, with an enumerator called `page` pointing at each member of the `HashMap` called “hits.” For each hit, we render a table row containing the URL (the value of which is both the text to display and the HREF to point it to) and the relative rank of the hit. JSP handles hooking up the variables and properties using reflection. However, when implementing this page, we come across the first (and only) reason to change some of the original Spider code.

Changes to the Original Code to Fit the JSP

JSP reflects on fields to hook up properties to `<out>` display tags instead of getters and setters. Unfortunately, our original implementation of `HitBean` marked all of its data private and only exposed getters and setters (normally, the appropriate strategy). Since we now have to have the fields exposed directly, we need to make a simple change to the Spider. The original class started with these declarations:

```

final String url;
final String title;
final float score;

```

It now has to become:

```

public final String url;
public final String title;
public final float score;

```

What if We Don't Have the Spider Source?

It is instructive to examine what happens when we aren't the original authors of either the application we are extending (jPetStore) or the service we are integrating (Simple Spider). If we don't have access to the source code of either project, we can still make the extension we've been working on. For the jPetStore, all we did was modify a configuration file and a JSP (which we always have the source for) and add a new class.

If we don't have access to the original source for the HitBean class, how can we make it work with the JSP? The answer is simple: write a wrapper class that exposes the correct properties (or just use the already exposed web service interface):

```
public class HitBeanWrapper {
    private HitBean _hitbean;
    public String url;
    public String title;
    public float score;

    public HitBeanWrapper(HitBean hitbean)
    {
        _hitbean = hitbean;
        url = hitbean.getUrl();
        title = hitbean.getTitle();
        score = hitbean.getScore();
    }
    public String getScoreAsString() {
        return _hitbean.getScoreAsString();
    }
}
```

This requires a change to the `handleRequest` method of the `SearchPagesController`, as well:

```
HashMap hits = new HashMap(qb.getResults().length);
for(int i =0;i<qb.getResults().length;i++)
{
    hits.put("hits", new HitBeanWrapper(qb.getResults()[i]));
}
return new ModelAndView("SearchProducts", hits);
```

That's it. We've edited the Spider all we need to in order to incorporate it into the jPetStore application.

Principles in Action

- Keep it simple: display the URL to result pages instead of complex rendering of product information; use simple table output instead of `PagedListHolder` (the need for it was gone)
- Choose the right tools: table, not `PagedListHolder`; JSP

- Do one thing, and do it well: JSP focuses on display of output, not search intricacies
- Strive for transparency: HitBean exposes simple data properties; use a wrapper for HitBean if the source is not available
- Allow for extension: none

Setting Up the Indexer

Now that the search service is integrated into the application, we'll configure the indexer to automatically update against the current version of the web site on a regular basis. If you recall from the previous chapter, both the console application and the web service have mechanisms that let you launch the indexer service instead of the search service. The question is, how should the indexer be integrated with jPetStore?

Embed in jPetStore or Launch Externally?

The first approach is to make the indexer part of the jPetStore application itself; in other words, to add code to jPetStore that invokes the indexer. jPetStore could invoke the indexer at the request of a user or on a schedule. Both methods have problems: if we expose a user interface for launching the indexer, we have to wrap it in some kind of secured section of the site for administrative users only. Currently, jPetStore has no such security built in. Building it just to wrap around the indexer seems like a major stretch—too much complexity, not enough payoff. Which means a manual access point is out.

The other option is to build a scheduler into the jPetStore application. Regardless of how the architecture, a scheduler would require the jPetStore application to be running for indexing to occur. Since jPetStore is a web- and container-based application, its lifecycle is entirely dependent on the external hosts. If the web server software is turned off for any reason, jPetStore shuts down as well. If the interval for the indexer falls in that window, the indexer doesn't run. In addition, writing scheduling code is completely outside of the problem domain for jPetStore, just as it was for the Simple Spider. The jPetStore application should do one thing: display animals in a web catalog.

We have no option but to invoke the indexer from some other location. A good strategy is to leverage an existing scheduler system: on Windows it's *schtasks* and on Linux it's *cron*. Let's implement the scheduled indexer on Windows.

Using the System Scheduler

For ease of use, we create a batch file for actually launching the service. We want to invoke the Java runtime to run our ConsoleSearch class's main method, passing in

the starting point for jPetStore. The command (and, therefore, the contents of our batch file) looks like this:

```
java c:\the\path\to\ConsoleSearch /i:http://localhost/jpetstore
```

We store that in a file called *jpetstoreIndexer.bat*. For simplicity's sake, we'll store it in *c:\commands*.

In order to schedule the indexer to run every night at 2:00 a.m., issue the following command (whiled logged in as a local administrator):

```
c>schtasks /create /tn "jpetstore Indexer" /tr:c:\commands\jpetstoreIndexer.bat  
/sc daily /st 02:00:00
```

The */tn* flag creates a unique name for the text; */tr* points to the actual command to invoke; */sc* is the time interval; and */st* is the specific time to launch the indexer on that interval.

Similarly, on Linux, edit the *crontab* file and launch the *cron daemon* to accomplish the same thing.

Smell the Roses

The beauty of this solution is that our application, the Simple Spider, has been repurposed to run in both a container-based environment (Spring) and a direct runtime environment (via the scheduler calling the Java runtime directly) without any extra code whatsoever. Because of its simple architecture and loosely coupled services, the Spider itself can operate just fine in both environments simultaneously. We didn't have to write a new access point or code a new UI or even make any configuration changes. Even better, we were able to take a single application from our first chapter and repurpose its internal services to two different endpoints without much work. It's good to step back every now and again and smell the roses, just to realize what a little forethought and adherence to simple principles gets you.

Principles in Action

- Keep it simple: use system-provided scheduler and existing console-based access point to application
- Choose the right tools: *schtasks*, *cron*, *ConsoleSearch*
- Do one thing, and do it well: neither Spider nor jPetStore worry about the scheduling of the indexer; the scheduler only worries about the index, not the rest of the functionality
- Strive for transparency: the scheduler knows nothing about the implementation details of the indexer or even where the results of the indexing will end up: it's all handled in configuration files
- Allow for extension: none

Making Use of the Configuration Service

If we jump straight in and start using the search as it's currently configured, we'll notice a problem. Our searches are returning lots of results—more than can be possible given the number of products in the database. In fact, a search for “dog” returns over 20 results, even though there are only 6 dogs in the database.

This is happening because of the brute-force nature of the crawling service. Without extra help, the crawler finds every link on every page and follows it, adding the results to the index. The problem is that in addition to the links that allow users to browse animals in the catalog, there are also links that allow users to add the animals to their shopping carts, links to let them remove those items from their carts, links to a sign-in page (which, by default in `jPetStore`, loads with real credentials stored in the text-boxes), and a live link for “Login,” which the crawler will happily follow—thus generating an entirely new set of links, with a session ID attached to them.

We need to make sure our crawler doesn't get suckered into following all the extraneous links and generate more results than are helpful for our users. In the first part of Chapter 9, we talked about the three major problems that turn up in a naïve approach to crawling a site:

Infinite loops

Once a link has been followed, the crawler must ignore it.

Off-site jumps

Since we are looking at `http://localhost/jpetstore`, we don't want links to external resources to be indexed: that would lead to indexing the entire Internet (or, at least, blowing up the application due to memory problems after hours of trying).

Pages that shouldn't be indexed

In this case, that's pages like the sign-in page, any page with a session ID attached to it, and so on.

Our crawler/indexer service handles the first two issues for us automatically. Let's go back and look at the code. The `IndexLinks` class has three collections it consults every time it considers a new link:

```
Set linksAlreadyFollowed = new HashSet();
HashSet linkPrefixesToFollow = new HashSet();
HashSet linkPrefixesToAvoid = new HashSet();
```

Every time a link is followed, it gets added to `linksAlreadyFollowed`. The crawler never revisits a link stored here. The other two collections are a list of link prefixes that are allowed and a list of the ones that are denied. When we call `IndexLinks.setInitialLink`, we add the root link to the `linkPrefixesToFollow` set:

```
linkPrefixesToFollow.add(new URL(initialLink));
```

`IndexLinks` also exposes a method, `initAvoidPrefixesFromSystemProperties`, which tells the `IndexLinks` bean to read the configured system properties in order to initialize the list:

```

public void initAvoidPrefixesFromSystemProperties() throws MalformedURLException {
    String avoidPrefixes = System.getProperty("com.relevance.ss.AvoidLinks");
    if (avoidPrefixes == null || avoidPrefixes.length() == 0) return;
    String[] prefixes = avoidPrefixes.split(" ");
    if (prefixes != null && prefixes.length != 0) {
        setAvoidPrefixes(prefixes);
    }
}

```

First, the logic for considering a link checks to make sure the new link matches one of the prefixes in `linkPrefixesToFollow`. For us, the only value stored there is `http://localhost/jpetstore`. If it is a subpage of that prefix, we make sure the link doesn't match one of the prefixes in `linkPrefixesToAvoid`.

A special side note: good code documentation is an important part of maintainability and flexibility. Notice the rather severe lack of comments in the code for the Simple Spider. On the other hand, it has rather lengthy method and type names (like `initAvoidPrefixesFromSystemProperties`), which make comments redundant, since they clearly describe the entity at hand. Good naming, not strict commenting discipline, is often the key to code readability.

All we need to do is populate the `linkPrefixesToAvoid` collection. `ConsoleSearch` already calls `initAvoidPrefixesFromSystemProperties` for us, so all we have to do is add the necessary values to the `com.relevance.ss.properties` file:

```

AvoidLinks=http://localhost:8080/jpetstore/shop/signonForm.do http://localhost:8080/
jpetstore/shop/viewCart.do http://localhost:8080/jpetstore/shop/searchProducts.do
http://localhost:8080/jpetstore/shop/viewCategory.do;jsessionid= http://localhost:
8080/jpetstore/shop/addItemToCart.do http://localhost:8080/jpetstore/shop/
removeItemFromCart.do

```

These prefixes represent, in order, the sign-on form of the application, any links that show the current user's cart, the results of another search, any pages that are the result of a successful logon, pages that add items to a users cart, and pages that remove items from a users cart.

Principles in Action

- Keep it simple: use existing Properties tools, not XML
- Choose the right tools: `java.util.Properties`
- Do one thing, and do it well: the service worries about following provided links; the configuration files worry about deciding what links can be followed
- Strive for transparency: the service doesn't know ahead of time what kinds of links will be acceptable; configuration files make that decision transparent to the service
- Allow for extension: expandable list of allowable link types

Adding Hibernate

jPetStore uses a relatively straightforward architecture for providing database access. There is an interface layer that provides functional mapping to the DAOs themselves without worrying about actual implementation details. The specific DAOs vary based on the backend database; we'll be examining the ones targeting HSQLDB (Hypersonic SQL).

Existing Architecture

Let's look at how the Product class is managed. Product is the domain object that represents one item in the catalog.

```
package org.springframework.samples.jpeteststore.domain;
import java.io.Serializable;
public class Product implements Serializable {

    private String productId;
    private String categoryId;
    private String name;
    private String description;

    public String getProductId() { return productId; }
    public void setProductId(String productId) { this.productId = productId.trim(); }

    public String getCategoryId() { return categoryId; }
    public void setCategoryId(String categoryId) { this.categoryId = categoryId; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description;}

    public String toString() {
        return getName();
    }
}
```

Its persistence is managed through an object that implements the ProductDao interface. A ProductDao must be able to load a specific product given its ID, or load a list of products either from a category or from a set of keywords.

```
public interface ProductDao {

    List getProductListByCategory(String categoryId) throws DataAccessException;
    List searchProductList(String keywords) throws DataAccessException;
    Product getProduct(String productId) throws DataAccessException;

}
```

There currently exists a class called `SqlMapProductDao` that looks up product information in Hypersonic SQL through SQL mapping files.

Hibernate Mappings for Existing Domain Objects

To replace this architecture with one based on Hibernate, we first have to create mapping files that define the relationship between the domain objects and the database. Looking again at `Product`, we'll create a mapping file called `Product.hbm.xml` which looks like:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping
    package="org.springframework.samples.jpeteststore.domain">

    <class name="Product" table="product">
        <id name="productId"
            column="productId"
            type="string">
            <generator class="native"/>
        </id>
        <property name="categoryId" column="category" type="string"/>
        <property name="name" column="name" type="string"/>
        <property name="description" column="description" type="string"/>
    </class>

</hibernate-mapping>
```

In the mapping file, we first identify the package and particular class (`org.springframework.samples.jpeteststore.domain.Product`) that we are mapping. We have to tell it what table to map to ("product", in this case) and then map the individual properties of the domain object to the columns in the table. This file needs to be saved somewhere on the class path; we'll create a new folder in the project structure called "hibernate" to hold our map files and our new DAOs.

Hibernate DAOs

The next step is to create a DAO that uses Hibernate as the persistence layer instead of the SQL mappings used in the original version. The new DAO needs to implement the `ProductDao` interface, just like the original DAO. However, the implementation of that interface will be totally different.

Here is the code for the new DAO:

```
public class HibernateProductDao implements ProductDao {
    SessionFactory factory;
    Configuration cfg;
```

```

public HibernateProductDao() {
    try {
        cfg = new Configuration().addClass(
            org.springframework.samples.jpeteststore.domain.Product.class);
        factory = cfg.buildSessionFactory();
    } catch (Exception ex) {
        System.out.println("Hibernate configuration failed: " + ex);
    }
}

public List getProductListByCategory(String categoryId)
    throws DataAccessException {
    List results = null;
    try {
        Session session = factory.openSession();
        results = session.find("from product where product.category = ?",
            categoryId, Hibernate.STRING);

        session.close();
    } catch (Exception ex) {
        System.out.println("Failed to connect to database:" + ex);
    }
    return results;
}

public List searchProductList(String keywords) throws DataAccessException {
    return null;
}

public Product getProduct(String productId) throws DataAccessException {
    Product p = null;
    try {
        Session session = factory.openSession();
        p = (Product)session.load(Product.class, productId);
        session.close();
    } catch (Exception ex) {
        System.out.println("failed to connect to database: " + ex);
        p = null;
    }

    return p;
}
}

```

First, we need a way to interact with Hibernate. As shown in Chapter 7, we need to create a Hibernate `SessionFactory` and use it to get a `Session` with which to interact with the database. The DAO's constructor instantiates a new Hibernate configuration, loading the mapping file from the class path based on the name of the class added to the configuration. Then, it gets the `SessionFactory` from the `Configuration`.

Each method uses the `SessionFactory` to open a new `Session` with the database. The `getProduct` method is the most straightforward; first, we get the `Session`. Then, we ask the session to load an instance of the `Product` class, given its `productId`. Note that

the result from the `session.load()` call is of type `Object`, which we have to cast to `Product`. Finally, we close the `Session`. Hibernate handles all the SQL commands, looking up the mapping files, matching the `productId` to the right column in the table, populating all the fields, everything.

The `getProductListByCategory()` method is less straightforward; it takes a `categoryId` and returns a `List` of all the products that match that category. In this case, we can't rely on the built-in SQL generation; we have to create our own query. Again, we first grab a `Session` from the `SessionFactory`, then use the `session.find()` method, which returns a `List` of `Objects`. `Find` takes three parameters in this case: the HSQL query (which contains a placeholder for a query parameter, marked with a "?"), the value to fill into the query parameter, and the type of that parameter.

As shown in Chapter 7, HSQL (Hibernate SQL) queries look a lot like regular SQL statements, except here we left off the "SELECT [values]" part of the statement, since Hibernate will fill those in for us based on the mapping. This method will now look up all the rows in the `Product` table where `categoryId` equals the value passed in to the method, and create one instance of `Product` for each row in the resultset. All the product instances are placed in a `List` and returned.

The final method of the DAO, `searchProductList`, would be a lot more complex, but luckily, we don't have to implement it. Since we have already replaced the original search functionality with the `Simple Spider`, this method will never be called now, so we simply return `null` (we have to do something, since the `ProductDao` interface still mandates its inclusion).

To finish out the new architecture, we just repeat these steps for each of the remaining five domain objects. Each gets a mapping file and an implementation of the appropriate DAO interface.

Changing the Application Configuration

In order to get the new DAOs working with `jPetStore`, we need to modify some configuration files. First, we'll need to create the global `hibernate.properties` file, which tells Hibernate which database to use and how to use it. `jPetStore` is currently configured to use a local instance of Hypersonic SQL, with a username of "sa" and a blank password (NEVER do this in a production environment). The `hibernate.properties` file looks like this:

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsq://localhost:9002
hibernate.connection.username = sa
hibernate.connection.password =
hibernate.dialect=net.sf.hibernate.dialect.HSQLDialect
hibernate.show_sql=true
```

This file should be saved in the project root file, next to the other global configuration files. Hibernate will look for it by name.

Next, open up `jPetStore`'s `dataAccessContext-*.xml` files (one is `dataAccessContext-jta.xml` and the other is `dataAccessContext-local.xml`). In each, there is a section that maps the DAOs for the project. Change each mapping to point to the new DAO, and eliminate the now unnecessary properties. For example, the original mapping for `ProductDao` was:

```
<bean id="productDao" class="org.springframework.samples.jpetestore.dao.ibatis.
    SqlMapProductDao">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="sqlMap"><ref local="sqlMap"/></property>
</bean>
```

This now becomes:

```
<bean id="productDao" class="org.springframework.samples.jpetestore.dao.hibernate.
    HibernateProductDao"/>
```

We can eliminate the properties because the Hibernate versions of the DAOs do not require any configuration information to be passed in by the controller; Hibernate manages those issues for us.

Once you have successfully changed all the DAO references, the last remaining piece is to include the necessary *jar* files in your class path. Hibernate requires the following *jars*: `hibernate2.jar`, `cglib2.jar`, `ehcache.jar`, `commons-collections.jar`, `dom4j.jar`, and `jta.jar` (all of which are included in the Hibernate download).

Spring's Built-In Hibernate Support

Now that you have seen the explicit way to do things, let's briefly take a look at the supporting infrastructure Spring provides for Hibernate. Spring, through its "inversion of control" architecture, can fully manage the creation of the `SessionFactory` for you. In addition, it provides a new class, `HibernateDaoSupport`, which allows your application-specific DAOs to reuse standard, template-derived calls for interacting with the `datasource`.

To set it up, you need to change your DAOs to extend `HibernateDaoSupport`. So, this:

```
public class HibernateProductDao implements ProductDao
```

becomes:

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao
```

Then add the following code to enable Spring to pass in a `SessionFactory`:

```
private SessionFactory sessionFactory;
public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
```

After adding this, your DAOs can use an object provided by `HibernateDaoSupport` called `HibernateTemplate`. This new class, accessed through the new `getHibernateTemplate()` method inherited from `HibernateDaoSupport`, exposes a

series of helper methods for interacting with the database, such as load, save, update, saveOrUpdate, get, and find. Our ProductDao becomes a lot simpler:

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao {
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public HibernateProductDao() {
    }

    public List getProductListByCategory(String categoryId) {
        return getHibernateTemplate().find("from product where product.category = ?",
            categoryId, Hibernate.STRING);
    }

    public List searchProductList(String keywords) throws DataAccessException {
        return null;
    }

    public Product getProduct(String ProductID) throws DataAccessException {
        return (Product) getHibernateTemplate().load(Product.class, productId);
    }
}
```

To configure all of this, you'll have to make some changes to your configuration files. You now have to add a property for the SessionFactory where you defined the ProductDao bean:

```
<bean id="productDao"
    class="org.springframework.samples.jpeteststore.dao.hibernate.HibernateProductDao">
    <property name="sessionFactory"/>
    <ref bean="mySessionFactory"/>
</bean>
```

Then add a definition of the mySessionFactory bean:

```
<bean id="mySessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
    <list>
        <value>product.hbm.xml</value>
    </list>
    <!-- etc. -->
    </property>
    <property name="hibernateProperties">
    <props>
    <prop key="hibernate.dialect">net.sf.hibernate.dialect.HSQLDialect</prop>
    </props>
    </property>
    <property name="dataSource">
```

```
<ref bean="dataSource"/>
</property>
</bean>
```

Add as many entries to the `mappingResources` property as you have *map* files, and make sure that the `dataSource` property refers to your already-configured `dataSource` bean. With these minimal changes, your DAOs become much more standardized and compacted, and Spring handles all your `SessionFactory` and `Session` implementation details for you. You are free to focus, yet again, on the problem at hand rather than the supporting framework.

That's it! Once again, we've managed to replace an entire swath of existing code without touching the original codebase itself. We have simply added new classes to the project and changed some configuration settings, and voila! Hibernate.

Principles in Action

- Keep it simple: domain objects remain unaware of persistence logic, Hibernate manages all configuration
- Choose the right tools: Hibernate
- Do one thing, and do it well: the domain model is focused on the business problem, the DAOs focus on data manipulation and are database-agnostic
- Strive for transparency: domain model is completely unaware of persistence layer
- Allow for extension: Spring configuration and IoC allow us to change persistence layers

Summary

Over the last two chapters, we have taken an initial customer's requirements for a generic, flexible web site search engine and refined them to meet our core principles. We then designed a simple, straightforward application that met those requirements—and then some—and made use of existing tools (however unorthodoxly) to accomplish a complex set of tasks. The result was a solution to the initial requirements that came in well below the \$18,000 that Google charges for its search appliance, even if we had billed the customer not only for design and development, but also for all the time spent researching the included open source tools and writing these two chapters! And, frankly, we aren't cheap. Simplicity really does have its rewards.

We learned how easy it is to integrate two applications designed with our core principles in mind. Since the `jPetStore` sample is built on a lightweight framework (Spring) and makes good use of the world's most common design pattern (MVC), it was child's play to introduce a replacement service for the limited one provided.

Since the Spider is well factored and provides flexibility through its configuration service, it is easy to adapt it for use in a new context, in a container-based environment, and with an entirely new user interface, using only three changed lines of code to the original application (and those lines only added a new scoping keyword). And since Hibernate is also built on these same principles, it was incredibly easy to swap it into the project in place of the existing persistence mechanism.

These examples demonstrate that Java doesn't have to be hard. In fact, if you focus on the principles outlined in this book, Java can be downright fun.