Beginning Apache Struts

From Novice to Professional

Arnold Doray

Apress®

Beginning Apache Struts: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-604-3

ISBN-10 (pbk): 1-59059-604-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Kunal Mittal
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Lori Bring
Indexer: Valerie Perry
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 5

The MVC Design Pattern

Servlet technology is the primary base on which you can create web applications with Java. JavaServer Pages (JSP) technology is based on servlet technology but extends it by allowing HTML content to be created easily.

Note that JSP technology doesn't *replace* servlet technology, but rather *builds on* it, to address some of its deficiencies. Struts follows this pattern, and builds on servlet and JSP technologies to address their shortcomings. These shortcomings are in two broad areas:

- **Creating a "separation of concerns"**: Pieces of code that do different things are bundled separately and given standardized ways of communicating between themselves.

- **An infrastructure for webapps**: This includes things like validating user input, handling and reporting errors, and managing flow control.

If these shortcomings are addressed, the very practical problems of building webapps (robustness, maintainability, localization, etc.) become easier to tackle.

One tool that can be used to make some headway in resolving these issues is an organizing principle called the **M**odel-**V**iew-**C**ontroller (MVC) design pattern.

WHAT ARE DESIGN PATTERNS?

Design patterns are essentially recipes for solving generic coding problems. They are not algorithms but rather *organizational principles* for software.

The idea of using "design patterns" to solve common software engineering problems first came to prominence with the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1995), by, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The authors credit an earlier work, *A Pattern Language for Building and Construction* (Oxford University Press, 1977) by the architect Christopher Alexander as the inspiration for the idea of design patterns for software.

The MVC design pattern, which is the subject of this chapter, was first used in an AI language called Smalltalk in the 1980s to solve a problem similar to the one we're discussing here for webapps. A variation on the pure MVC pattern is also used in Swing, Java's platform-independent GUI toolkit.

The MVC design pattern calls for a separation of code by their function:

- **Model**: Code to control data access and persistence
- **View**: Code to handle how data is presented to the user
- **Controller**: Code to handle data flow and transformation between Model and View

Figure 5-1 illustrates MVC.

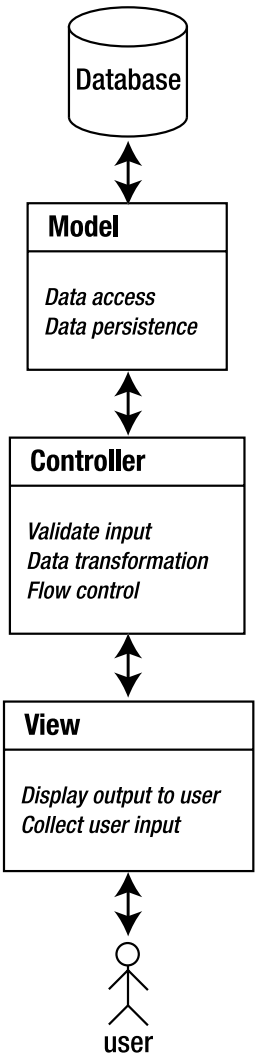

Figure 5-1. *The MVC design pattern*

In addition to this separation, there are a couple of implicit constraints that the Model, View, and Controller code follow:

- The View gets data from the Model only through the Controller.
- The Controller communicates with the View and Model through well-defined, preferably standardized ways. For example, embedding SQL code directly in your Controller code violates this principle of standardized communication. Using Model classes that expose functions for data access would be an example of standardized communication between the Controller and the Model.

It may not be immediately obvious to you how this solves anything, but I hope you'll come to appreciate the power of the MVC approach as you progress. For now, it is sufficient for you to accept that applying the MVC design pattern helps us achieve the larger goal of separation of concerns, which greatly simplifies the building of web applications.

Our discussion so far has been quite general for a purpose. *Any* web application framework will use the MVC principle, so it is to your advantage to understand it apart from Struts' implementation of it.

Later, we'll discuss how Struts implements the MVC design pattern. But before we do, I'll give you an example of how to apply the MVC design pattern.

The Registration Webapp

Many web applications have a "user registration" facility. I'm sure you've seen this before. Some online news sites, for example, have the annoying requirement that you "subscribe" before you can view their content.

I'll call this facility the "Registration Webapp." What I'll do is specify the requirements of this webapp, and for each requirement, I'll identify the types of code (Model, View, or Controller) that might be needed.

Requirement 1

The user is asked to specify a user ID (userid) and a password, as well as a password confirmation, as shown in Figure 5-2.

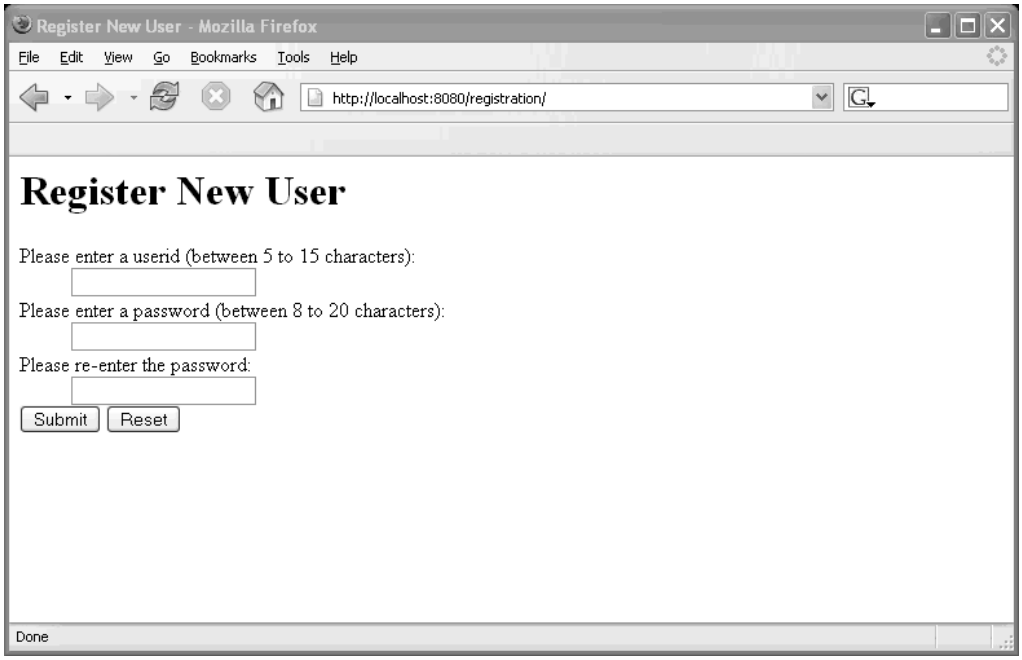

Figure 5-2. *The Registration webapp main page*

Discussion

Clearly, you need View code to create the form containing the user ID and password fields. The output of the View code would be HTML. The View code itself might be this displayed HTML, or a JSP that is transformed into HTML by the servlet container.

Requirement 2

The form data is validated to check for a malformed user ID or password. The form is also checked to ensure that the same password was entered both times and that it has the right length.

Discussion

You will need Controller code to do the validation. One important point to note is that these checks can be done without referring to a database. This means that you could validate either on the client (the web browser), perhaps using JavaScript, or use Java classes residing on the server.

The first option would bend the MVC rule of strict separation between View and Controller because the JavaScript would have to be placed in the HTML or JSP code in Requirement 1. Even if you factored out the JavaScript validators into a separate `.js` file, you'd still need Controller code on the HTML/JSP View code to transfer the form values to the JavaScript validators. Don't get me wrong; I'm *not* saying that client-side JavaScript is an evil that crawls out from the bowels of Hell. I'm simply saying that *hand-coded* client-side JavaScript to perform validation violates MVC.

Violations of MVC, *no matter how well justified*, come at the cost of reduced maintainability.

Fortunately, it is possible to have your cake and eat it too. Instead of using hand-coded client-side JavaScript, Struts gives you the option of *autogenerating* client-side JavaScript. This would greatly reduce Controller/View overlap. You'll see this in action when we cover the Validator framework in Chapter 15.

Or, we could simply cut the Gordian knot and do all validations on the server. Unfortunately, this isn't always an option, especially if the client-server connection is slow.

Requirement 3

The data is further checked to ensure that there are no other users with that user ID.

Discussion

As in Requirement 2, we obviously need Controller code to run the validation. Unlike Requirement 2, a client-side validation is less feasible, because we might need to check with a database in order to find out if the given user ID exists.

This also implies we need Model code to read user IDs off the database. The Model code has a function like this:

```
public boolean exists(String userid)
```

to run the user ID check. Our Controller code would use this function to run the validation.

Requirement 4

If at any stage there is an error, the user is presented with the same start page, with an appropriate error message (see Figure 5-3). Otherwise, the user's user ID and password are stored.

Figure 5-3. *Signaling errors to the user*

Discussion

You'll need a Controller to decide what to do if there's an error: display the error page or save the data?

You'll also need a View to display the error message. Here's where enforcing strict MVC separation makes things complicated. For example, do you need to duplicate View code from Requirement 1 to redisplay the form data? How about re-populating the previously keyed-in form data? In Figure 5-3, the user ID field isn't blank—it contains the previously keyed-in user ID.

In fact, we could handle errors from Requirement 2 if we threw out MVC and mixed Controller and View code, as I described in the discussion to Requirement 2. Although this still wouldn't solve the same display issues arising from Requirement 3, we might be able to hack a solution by merging Model and Controller code with our View code by embedding scriptlets in our JSP View code.

These complications illustrate the need for a web application framework like Struts. You need Struts to help you enforce MVC. Without Struts (or a web application framework), it is probably impossible to cleanly enforce MVC separation on your webapp code.

Note that I'm only saying that Struts (or a webapp framework) makes MVC *possible*. Don't take this to mean that Struts makes bad code impossible! You can cheerfully violate MVC even if you're using Struts.

Lastly, you'd also need Model code to save the user ID and password pair. Listing 5-1 outlines how the Java class for your Model code would look.

Listing 5-1. *Skeleton for the Registration Webapp Model*

```
public class User{

   protected String _userId = null;
   protected String _password = null;

   /* Bean-like getters and setters */
   public String getUserId(){
      return _userId;
   }

   public String setUserId(String userId){
      _userId = userId;
   }

   public String getPassword(){
      return _password;
   }

   public String setPassword(String password){
      _password = password;
   }

   /**
   * Checks if the userid exists.
   */
   public static boolean exists(String userid){
      //implementation omitted
   }

   /**
   * Saves the _userid and _password pair
   */
   public void save() throws Exception{
      //implementation omitted
   }
}
```

Of course, in real applications, you'd have more than one class for your Model code.

Requirement 5

If the registration is successful, the new user gets a "You are registered" web page (see Figure 5-4).

Figure 5-4. *Successful registration*

Discussion

You'll need View code to display the page. Plain ol' HTML would do, unless you're embedding this message within a larger page, as is often the case in real life. Like Requirement 4, this latter possibility also complicates strict MVC enforcement.

But again, Struts comes to your rescue. As you'll see in Part 2 of this book, Struts has an extension ("plug-in" in Struts-speak) to give you just this functionality.

I hope this simple example gives you a good idea how to classify your code with the MVC worldview. I also hope it gives you an idea of why you'd need a framework to help you actually enforce MVC. To give you a little more practice, try the following quiz.

Lab 5: MVC Quiz

LILLDEP (Little Data Entry Program) is a simple data entry program for storing and managing a database of contacts. It's a common type of web application used in many SMEs (**S**mall **M**edium **E**nterprises). Figure 5-5 shows the basic LILLDEP main page.

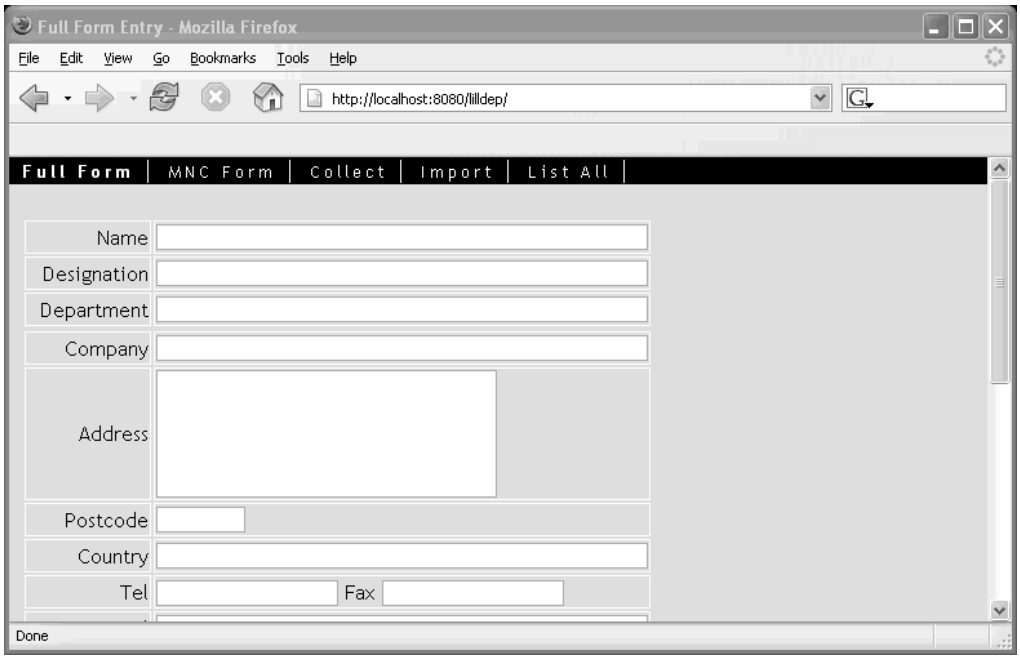

Figure 5-5. *The LILLDEP main page*

Have a look at the three requirements that follow and try to identify the Model-View-Controller parts that might be relevant. Imagine a possible implementation for each, as I did in the previous section, and identify where you might need help to enforce strict MVC code separation.

- **Requirement 1**: A page to collect information about a contact
- **Requirement 2**: Code to check contact details (postal code, email, or required fields) after submission and display errors
- **Requirement 3**: Code to store and retrieve data into a database

Which Comes First?

When you develop your webapp with the MVC design pattern, which portion (Model/View/Controller) would you specify and design first? Why?

Close the book and give yourself a few minutes to answer this before reading the following discussion.

There's no "right" answer to this one, but most people start with either View code or Model code. Rarely would you start with a design for the Controller since it depends on View and Model components.

My personal preference is to specify and design Model code first. This means defining at an early stage *what* data to store and *how* to expose data with data access methods (like the `exists()` method in Listing 5-1) to other code. Here are my reasons for holding this position:

- Without a clear specification of *what* data to store, you can't design the rest of the application.

- Without a specification of *how* to access data, a clear separation between Model and Controller is difficult to achieve. For example, if you're using a relational database, you might end up with SQL code in the Controller. This violates the MVC principle of well-defined interfaces.

Many people (especially new developers) prefer to design View code first, possibly because this helps them understand the application's inputs and control flow. In this approach, View comes first, then Controller, and Model last. Though it's perfectly valid, I believe this approach is prone to certain pitfalls:

- **Redundant information** being stored in the database. Like the proverbial three blind men describing the elephant, using View code to drive Model design is bad because you lack a global view of the application's data requirements. This is especially so in complex applications, where you could store the same information twice, in two different tables.

- **Poorly organized database tables**, because the Model code and data organization is designed to conform to View requirements, which may not reflect the natural relationships between Model elements.

- **Less future-proof code,** because a change in the View code may necessitate a revamp of Model code and the database design. Essentially, Model code is too strongly coupled to View code.

All these stem from the fact that those who take the "View first" approach use View code or mockups to indirectly educate them about Model requirements.

Having said that, there are many situations where a "Model first" approach may *seem* impractical. For example, in many corporate IT departments, developers don't have a full understanding of the problem domain. Instead, they have to rely on and frequently communicate with non-IT personnel like managers or admin staff ("domain experts") in order to understand the system's requirements.

In this scenario, the application is naturally View driven because mockups are used as a communication tool between developers and domain experts. My advice to those in such a situation is to *incrementally* build a picture of the Model code and data immediately after each discussion with domain experts. Don't leave Model design to the last.

Whichever way you choose, Model-first or View-first, I believe it's important to understand why you do things the way you do, simply because this helps you become a better programmer.

In the following section, I'll describe how Struts fits into the MVC design pattern.

Struts and MVC

In Struts, **Model** code consists of plain old Java objects (POJOs).

In other words, Struts places no restrictions on how you code the Model portion of your webapp. You should encapsulate data access/persistence into Model classes, but Struts does not force you to do this. Remember, Struts is here to help you achieve MVC nirvana as easily as possible. Apparently, you don't really need help to separate Model from View and Controller.

View code consists of JSPs and a set of custom tags supplied by Struts. These custom tags enable you to separate View from Controller, because they address all the issues I raised earlier when I tried to apply MVC to the Registration webapp.

With Struts, **Controller** code falls into three broad categories:

- **Simple validations** are performed by your subclasses of the Struts base class called `ActionForm`. Checks for password length or email address format are examples of this. Later in this book, you'll see how Struts greatly simplifies validation with the Validator framework.

- **Complex validations and business logic** are done in your subclasses of the Struts base class called `Action`. The check for the duplicate user ID for the Registration webapp is an example of complex validation. An example of business logic is calculating the total amount due after a purchase in a shopping cart webapp.

- **Flow control** is also decided by your `Action` subclasses, restricted to paths declared in the Struts configuration file called `struts-config.xml`.

You'll learn more about `ActionForm`, `Action`, and `struts-config.xml` as this book progresses. At this stage, you should simply understand how MVC maps into Struts. Figure 5-6 depicts this mapping.

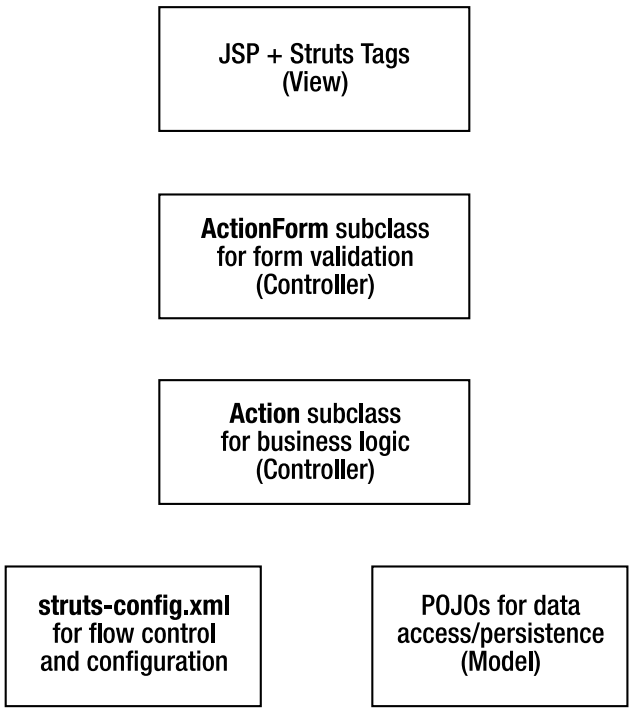

Figure 5-6. *How MVC maps into Struts*

Lifecycle of a Struts Request

The three categories of processing (simple/complex validation, business logic, and flow control) come into play when an incoming HTTP request is generated by a user submitting form data.

With servlet technology all such processing is done by servlets. As I mentioned in Chapter 2, Servlets are just Java objects, *usually* your subclasses of `HttpServlet`.

Even JSP pages are converted at runtime into servlet classes—actual `.java` source files. These autogenerated Java source files are later compiled.

Since Struts is built on servlet technology, it is no exception to this rule. All submissions of form data to a Struts application are intercepted by a *single* "master" servlet, an instance of `ActionServlet`, which is a part of Struts.

This master servlet is responsible for delegating the actual work of your application to your subclasses of `ActionForm` and `Action` (see Figure 5-6).

USUALLY?

Servlet technology is quite generic, meaning it is not at all tied to processing HTTP requests, although this is by far the most common use.

This is because servlets solve a number of *generic* problems that stem from client-server communication, such as object pooling and session management. Your servlet classes could implement the basic `Servlet` interface from scratch, or subclass the `GenericServlet` base class. You'd do this if you wanted to take advantage of servlet technology's solutions to these generic problems.

If you're primarily concerned with processing and displaying web pages, `HttpServlet` is the base class you'd subclass.

■ **Note** In addition to this, `ActionServlet` and its helper classed are responsible for many other behind-the-scenes things like pooling `Action` subclasses for efficiency, reading configuration data, or initializing Struts extensions called plug-ins. These activities are invisible to Struts developers, and are one good reason why Struts is so easy to use. You don't have to know how the clock works to tell the time!

The incoming form data is processed in stages:

- **Data transfer**: Form data is transferred to your `ActionForm` subclass.
- **Simple validation**: The form data in this subclass is passed through simple validation. If simple validation fails, then the user is automatically presented with the form with the error messages detailing the errors. This is a *typical* scenario, but the details would depend on how you configured Struts. We'll go through the mechanics of simple validation in Chapter 6 and tackle configuration in Chapter 9.
- **Further processing**: Form data is next sent to your `Action` subclass for complex validation and the processing of business logic. We'll discuss this in Chapter 7.

Your `Action` subclass also specifies the "next" page to be displayed, and this ends the processing of a single request.

Frameworks for the Model

In a previous section, I sang the joys of designing the Model portions of your webapp first, so Struts' lack in this area might come as a bit of a letdown.

One reason for this glaring deficiency in Struts is because there are already several frameworks to help you with data access and persistence. These go under the monikers "persistence framework/layer" or "object/relational (O/R) persistence service." The reason these frameworks exist is to persist data, and do it *well*. You may use any one of these in conjunction with Struts. So you see, it isn't a deficiency at all but just good design.

There are two different approaches to data persistence:

- **Persist the classes**: This approach lets you store and retrieve any Java object to/from a relational database. You are provided with a set of classes, which you must use to persist and retrieve your own Java classes. An example of this approach is Hibernate, from Apache.

- **Classes that persist**: In this approach, you specify a description for your database tables and their interrelationships. The persistence framework *autogenerates* Java classes (source code), which have mappings to your database tables. These autogenerated classes have `save()`, `select()`, and `delete()` functions to allow persistence and retrieval of data objects. A prime example of this approach is Torque, also from Apache.

The "persist the classes" approach is by far the more flexible of the two, since you could potentially persist any Java class, including legacy ones, or third-party classes. The "classes that persist" approach, on the other hand, lacks this capability but produces a cleaner separation between Model and Controller.

I should warn you that while this comparison between approaches is valid, you can't use it to choose between products like Hibernate and Torque. Hibernate, for example, provides many features beyond what I've described here. These extra features are what you'd want to consider when choosing a persistence framework.

In Appendix A, I give you simple examples of how to use Hibernate and Torque, and also (shameless plug) a simple, open source, "classes that persist" framework called Lisptorq, developed by the company I work for.

Useful Links

- Hibernate: `www.hibernate.org/`
- Torque: `http://db.apache.org/torque/`
- Lisptorq: `www.thinksquared.net/dev/lisptorq/`

Summary

- The Model-View-Controller (MVC) design pattern brings many benefits to the webapp that implements it.
- Implementing MVC is difficult, which is why you need a web application framework like Struts.
- Struts only places restrictions on the View and Controller. You are free to implement the Model portion in any way you wish.
- Two very popular persistence frameworks are Hibernate and Torque.