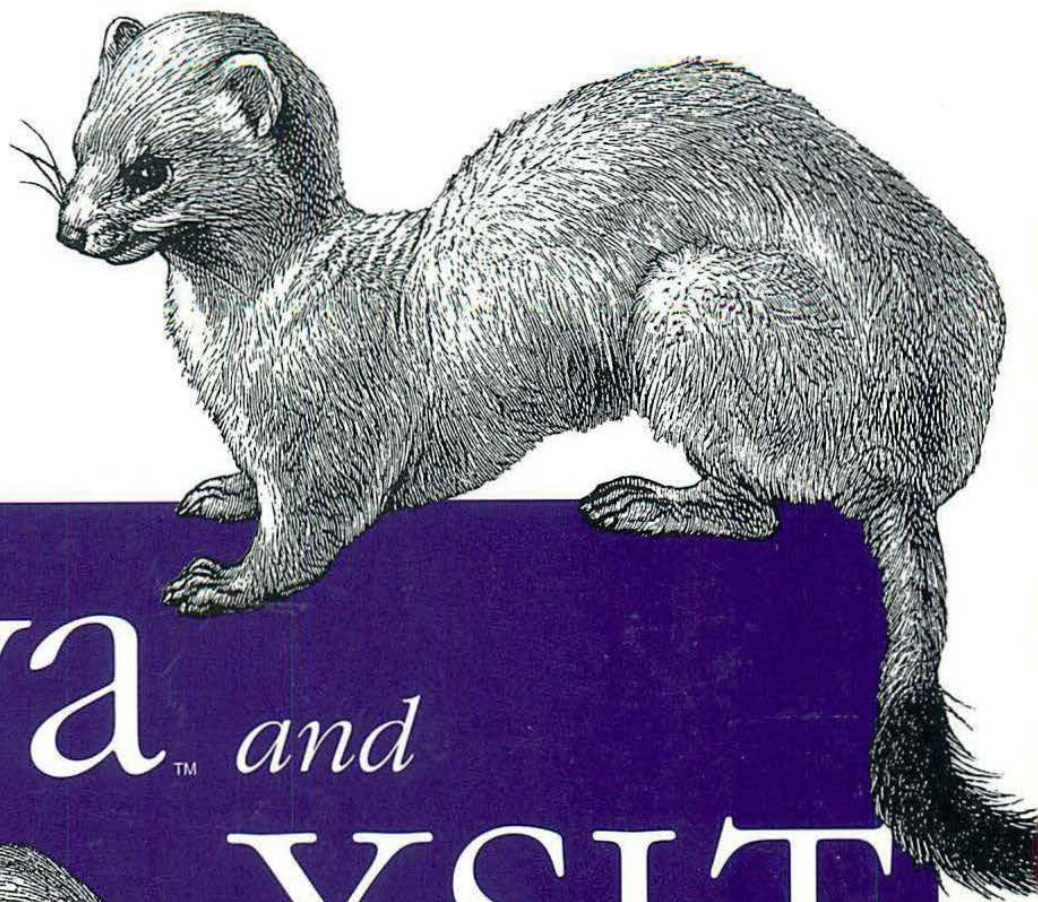
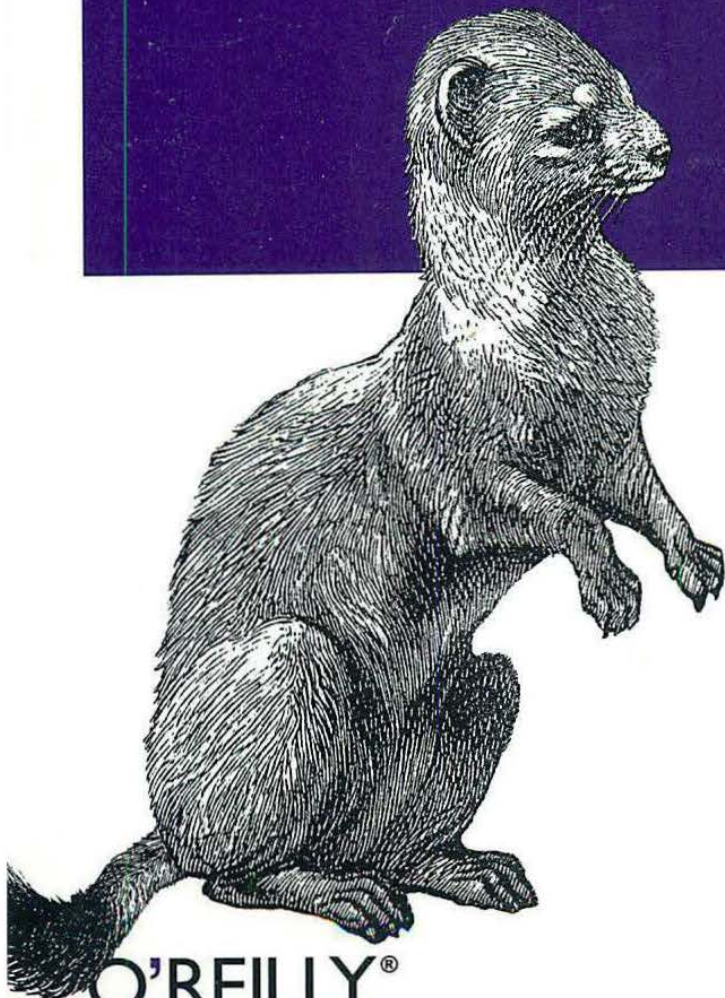




Java[™] *and* XSLT



O'REILLY[®]

Eric M. Burke

Java™ and XSLT

Other Java™ Titles from O'Reilly

Creating Effective JavaHelp™	Java™ in a Nutshell
Database Programming with JDBC™ and Java™	Java™ Internationalization
Developing JavaBeans™	Java™ Message Service
Enterprise JavaBeans™	Java™ Network Programming
Java™ 2D Graphics	Java™ Performance Tuning
Java™ & XML	Java™ Professional Library
Java™ Cryptography	Java™ Security
Java™ Distributed Computing	JavaServer™ Pages
Java™ Enterprise in a Nutshell	Java™ Servlet Programming
The Java™ Enterprise CD Bookshelf	Java™ Swing
Java™ Examples in a Nutshell	Java™ Threads
Java™ Foundation Classes in a Nutshell	Jini™ in a Nutshell
Java™ I/O	Learning Java™
	Java™ Cookbook

Java[™] and XSLT

Eric M. Burke

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Java™ and XSLT

by Eric M. Burke

Copyright © 2001 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Mike Loukides

Production Editor: Matt Hutchinson

Cover Designer: Ellie Volckhausen

Printing History:

September 2001: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. The association between the image of an ermine and Java and XSLT is a trademark of O'Reilly & Associates, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly & Associates, Inc., is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00143-6

[C]

enhanced error checking, database connection pooling, custom tag libraries, and other value-added features. As you add more framework-specific features, however, your flexibility to choose another framework or vendor quickly diminishes.

One typical framework is Turbine, which is one of many different frameworks supported by Apache. Turbine is a large framework with many value-added features including:

- Database connection pooling, integration with object to relational mapping tools, and relational database abstractions
- Integration with numerous template engines
- Role-based security and access control lists
- Web browser detection
- Integration with JavaMail

This is only a short list of Turbine's features. At its core, however, the compelling reason to use a framework like Turbine is the underlying object model. The fundamental approach of Turbine is to cleanly separate validation logic, the servlet itself, and page rendering into distinctly different modules. In fact, Turbine uses a single servlet, so your validation and rendering logic have to go elsewhere. The approach is to define helper classes called *actions*, which are responsible for validation of incoming requests. Once an action has validated the inbound request, other classes such as *Layout*, *Page*, and *Navigation* are responsible for rendering a view back to the browser.

When compared to a pure XML/XSLT approach, frameworks have the advantage of value-added features. If you remove all of the non-web features, such as database connection pooling and object-to-relational mapping tools, you will see that the underlying model-view-controller architecture is very easy to implement. You should be wary of any framework that provides too much non-web-related functionality because many of these features should be placed on the application server instead of the web server anyway. The remainder of this chapter is devoted to showing you how to structure a complex web application without committing yourself to a specific framework.

Model-View-Controller

Cleanly separating data and presentation logic is important. What exactly are the benefits? First and foremost, when data is completely isolated from the user interface, changes can be made to the visual appearance of an application without affecting the underlying data. This is particularly important in web applications that have to support multiple incompatible browsers or even WML, XHTML Basic,

or HTML. It is much harder to adapt to new user interface requirements when data and presentation are mixed.

Programming logic should also be separated from data and presentation logic. To a certain extent, programming logic must depend in part on both data and presentation. But you can generally isolate business logic, which depends on the data, and presentation logic, which depends on the user interface. Figure 4-4 illustrates these dependencies.

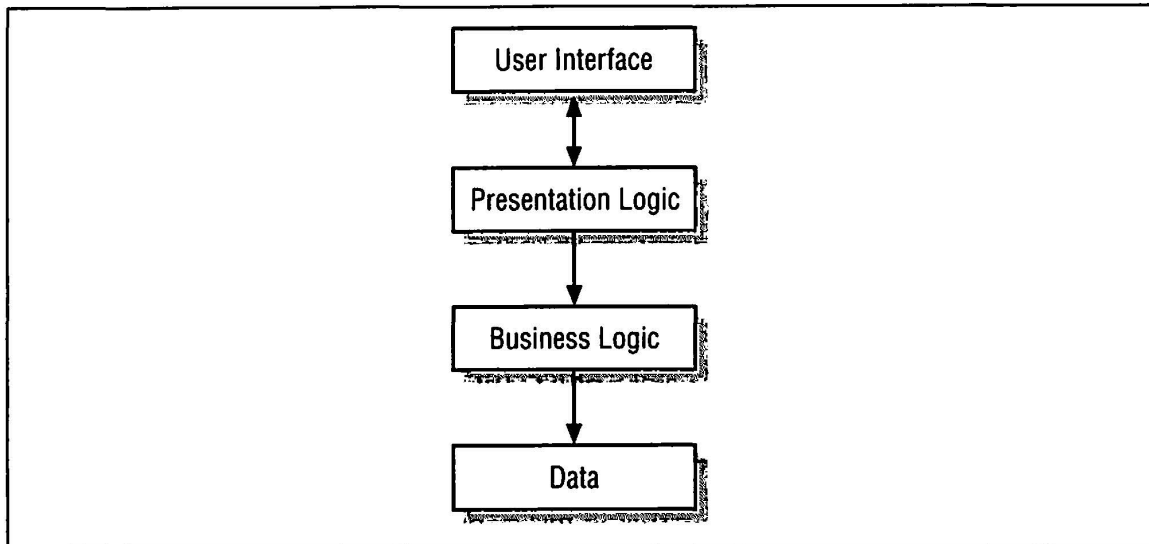


Figure 4-4. Dependencies

The arrows indicate dependencies. For example, if your underlying data changes, then the business logic will probably have to change. However, that does not always flow up and break your presentation logic. In general, if changes are sweeping, it is hard to avoid affecting upper layers, but minor changes can almost always be encapsulated. If the implementation of your business logic changes, however, there is no reason to change the underlying data. Likewise, you should be able to make changes to the presentation logic without breaking the business logic. Later in this chapter, we will see how Java, XML, and XSLT can be utilized to satisfy these dependencies.

The dominant pattern in scalable web sites is model-view-controller. The MVC pattern originated with Smalltalk-80 as a way to develop graphical user interfaces in an object-oriented way. The basics are simple. GUI components represent the view and are responsible for displaying visual information to the user. The model contains application data. The controller is responsible for coordinating between the model and the view. It intercepts events from the view components, queries the model for its current state, makes modifications to the model, and notifies the view of changes to the model. Figure 4-5 illustrates the interaction between these three components.

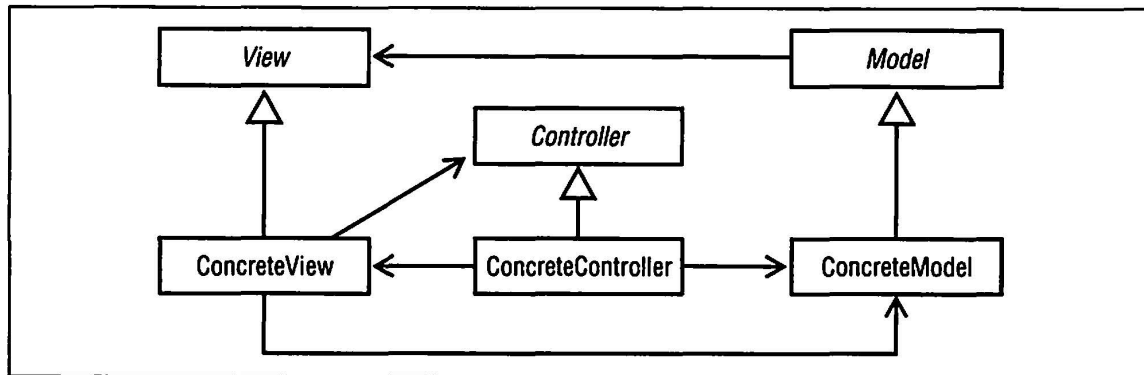


Figure 4-5. Model-view-controller

As shown, the `Model`, `View`, and `Controller` are either abstract classes or interfaces. The concrete classes are application-specific, and the open arrows indicate the direction of association between the various classes. For example, the abstract `Model` sends notifications only to the abstract `View`, but `ConcreteView` knows about its `ConcreteModel`. This makes sense when you consider how hard it would be to create a specific view, such as a customer editor panel, without knowledge of a specific data model like `Customer`. Since the `Model` only knows about `View` instances in an abstract way, however, it can send generic notifications when it changes, allowing new views to be attached later.

It is important to remember that this is just a pattern; specific implementations may vary somewhat and use different class names. One variation is to eliminate the explicit references from `ConcreteView` to `ConcreteModel` and from `Model` to `View`. In this approach, the `Controller` would take a more prevalent role. A common theme in Java is to remove the explicit controller using data models and view components that send notifications to event listeners. Although typically thought of in terms of GUI applications, the MVC architecture is not limited to this domain. For web applications, it is commonly used in:

- The hybrid servlet + JSP approach
- Most servlet frameworks
- The XSLT approach

In the hybrid approach, the servlet is the controller and the JSP is the view. It is assumed that the data will be retrieved from a database or Enterprise JavaBeans (EJB) components, which act as the model. A good framework may make the distinction between model, view, and controller more explicit. Instead of using the servlet as a controller, a common pattern is to use a single servlet that delegates work to helper classes that act as controllers. Each of these classes is equivalent to `ConcreteController` in Figure 4-5 and has knowledge of specific web pages and data.

Although originally intended for Smalltalk GUIs, MVC has always been one of the most frequently used patterns in all sorts of GUIs, from Motif to Java. On the web, MVC is also prevalent, although a few mechanics are slightly different. In a web environment, we are restricted to the HTTP protocol, which is *stateless*. With each click of a hyperlink, the browser must establish a new connection to the web server. Once the page has been delivered, the connection is broken. It is impossible for the server to initiate a conversation with the client, so the server merely waits until the next request arrives.

Implementing MVC in this stateless architecture results in looser coupling between the controller and the view. In a GUI environment, the controller immediately notifies the view of any changes to the underlying model. In a web environment, the controller must maintain state information as it waits for the browser to make another request. As each browser request arrives, it is the controller's job to validate the request and forward commands on to the model. The controller then sends the results back to the view.

This may all sound academic and vague at this point. The next few sections will present much more detailed diagrams that show exactly how MVC is implemented for an XSLT-driven web site.

XSLT Implementation

All of the approaches presented up to this point are, of course, building up to the XSLT approach. In many respects, the XSLT approach is simultaneously the most powerful and the easiest to understand. For a single web page, the XSLT approach is probably harder than a servlet or JSP to configure. Configuration of the XML parser and XSLT processor can be quite difficult, mostly due to CLASSPATH issues.* But as the complexity of a web application increases, the benefits of using XSLT become obvious. Figuring out how to tackle these complex web applications is the real goal of this chapter.

The XSLT approach maps fairly directly to the MVC pattern. The XML represents the model, the servlet represents the controller, and the XSLT produces HTML, which represents the view. The XSLT stylesheets may contain a minimal amount of logic, potentially blurring the line between view and controller. Figure 4-6 represents a conceptual view of how the XSLT approach maps to MVC.

One weakness common to every approach other than XSLT is the HTML-centric viewpoint. In every example presented thus far, it was assumed that we generated HTML. What happens when the requirement to support cellular phones arises? It

* This can be a frustrating experience when the servlet container comes with an older XML parser that uses DOM or SAX Version 1. Most XSLT processors require Version 2 parsers.