

# Instant Update

## Making your data and spreadsheets Web viewable through MVC

Paul Sholtz  
New Architect Magazine  
January 2003

The Model-View-Controller (MVC) pattern is an established and well-understood software design method. It works well in the context of rich-client GUI applications that maintain persistent state. However, the Web is an entirely different medium than a rich client GUI, and porting MVC to the Web isn't straightforward.

The goal in MVC is to separate the application data (the model) from the way the data is rendered to the user (the view) and from the way in which the user controls the data (the controller). Dividing the concerns of the application in this way dramatically increases the system's modularity, and enhances the overall flexibility of the design. Such flexible design is particularly useful for e-commerce.

### Modularity

Modularity is a key requirement for any type of software application. Object-oriented programming (OOP) arose out of the desire to increase code reuse and make

software architectures more flexible. It was meant to promote better encapsulation of the data used by a program and of the methods that operate upon it. In principle, because objects only interact with other objects through well-defined interfaces, it should be relatively straightforward to swap out components so long as the new components implement the same interfaces as the old ones. The Web services trend takes this model to the next logical level by exposing only well-defined component interfaces to the public via the Internet. Service providers can swap out and upgrade the back-end functionality as needed.

While modularity and separation of concerns are important design issues for any software subsystem, they are particularly critical for user interface design. Consider, for example, spreadsheet programs. Even though the user is able to enter data by typing (or importing) information into the cells of the spreadsheet document, staring at pages of numbers is tedious. Although the spreadsheet cells accurately reflect the information content of the spreadsheet, this is a poor way to visualize the data contained in the document. Hidden patterns may be lying in the data, waiting to be discovered, but they are impossible to see without data representation tools that can provide users with a proper "view" of the data.

Moreover, the user might want to drill down and explore different views of the same spreadsheet document. For example, a simple pie chart might suffice for most of a

## Instant Update

user's data visualization needs, but perhaps the user also needs to plot a bar chart or scatter graph to better understand some other trend contained in the data. The user also could expect that the charts and graphs already generated would accurately reflect any changes made to the underlying data model (e.g., spreadsheet cells).

### Classic Separation of Concerns

Our spreadsheet example is perhaps the most common design pattern in current use: the MVC pattern. First introduced in the Smalltalk-80 interface by Trygve Reenskaug in the late 1970s, MVC is an elegant and simple approach to user-interface programming. It allows developers to seamlessly add, remove, or change views (e.g., new graphic components in a spreadsheet application) without affecting the functionality of any other part of the system.

Another way to think about MVC is in terms of input/output program flow. Controllers define the input interface for a program, the model defines data processing functionality, and the view defines the program output.

At a high level, MVC framework components can be described as:

#### Model

The model object encapsulates the raw data that defines the application's central structure. A model object has no inherent notion of how it will be rendered when presented

to the user, although it does contain public query methods that allow clients (e.g., view objects) to ascertain the model's current state. Model objects also contain mutator methods that allow clients (e.g., controller objects) to modify the model's current state.

Model objects must have some mechanisms to "register" view objects that are interested in communicating the current state of the application to a user. The model object must also be able to notify all registered views whenever a change has been made to the underlying data model.

In Java, the most natural way to implement an MVC architecture is to utilize the Observer/Observable infrastructure that comes as part of this system and provides much of the needed registration and notification framework necessary to make MVC happen. Objects that represent the data in an MVC system should subclass the `java.util.Observable` class. This allows one or more Observer objects to register themselves with the Observable (model) object. In an MVC system, the `java.util.Observer` interface would ordinarily be implemented by a view object.

When the data model changes (i.e., when an accessor method is invoked), the model object should itself invoke two methods found in the parent `java.util.Observable` class: `setChanged()`, which tells the Observable parent that the state of the data model has been updated, and `notifyObservers()`, which signals to any interested

## Instant Update

Observer objects that a state change has occurred in the underlying data model and that the view needs to be refreshed.

### **View**

The separation of presentation from the data model is an extremely important design principle. If there is a clean and solid separation between model and view in the design, it should be possible to develop entirely different kinds of user interfaces for the same model code. For example, you could transition between a rich GUI client, a Web browser interface, a remote API, and even a command line interface while still sticking to the same model code.

Even if an application just uses one form of user interface, it should still be able to support multiple views of the underlying data model simultaneously. For example, rich client interfaces ordinarily display several representations of the data model on screen at once (as in our spreadsheet example). If a user makes a change to the model from one of the views, that update should be propagated to all the other views simultaneously. An e-commerce Web site, on the other hand, might display different customer pages at different points in the sales process. Separation of presentation from the data model builds in the modularity that allows for this flexibility.

As I mentioned, the best way to implement MVC functionality in Java is to use the built-in Observer/Observable infrastructure. View objects should

implement the `java.util.Observer` interface, which allows them to register themselves with any subclass of `java.util.Observable` and also to receive notification when state changes occur in the `Observable` objects they are watching. The `Observer` interface consists of a single method, `update(Observable, Object)`, which is called when the `Observable` object invokes the `notifyObservers()` method. The first argument tells the `Observer` which one of the `Objects` is sending the signal, and the second argument includes any optional argument the `Observer` object might need to pass along.

### **Controller**

The separation of the view from the controller is less important than that of the presentation from the data model. In fact, in many applications, the view and the controller are the same object (e.g., cells in a spreadsheet program). Whether a view also functions as a controller depends upon whether the view is meant to be editable. In our spreadsheet example, the cells in a spreadsheet program provide an editable view of the data, while the graphs and charts don't.

Controller objects should have a reference to the model objects they are controlling. This lets them invoke mutator methods to update the state of the model.

In our spreadsheet example, the different graphs (pie charts, scatter graphs, bar graphs, and so on) represent various views into the data contained in the spreadsheet. These components let the user view data, but not edit or

## Instant Update

control it. The spreadsheet cells themselves function both as views and controllers. They provide the user with the view of data contained in a specific spreadsheet cell. They also let the user update or control the data model by entering new information. Finally, the application's data model is encapsulated in a self-contained document object. Although the user has views into the document (through MVC "view" objects) and can interact with and update the data by typing information into the cells (which are MVC "controller" objects), the user doesn't usually interact directly with the document object itself. Rather, the document object exposes interfaces through which the user can view and modify it.

I've prepared some sample source code that provides an example of implementing the MVC pattern within the context of a small "thermometer" applet. This example illustrates how to use the Observer pattern to enable MVC within the context of Java. You can see the source code, and download the applet.

### LISTING

```
<html>
<head><title>Hello World JSP</title></head>
<body bgcolor="#FFFFFF" text="#000000">

<!-- This first Java code block is a bit contrived. Ordinarily
```

one would not place a JSP tag in an HTML document just to produce more HTML markup. Rather, the purpose would be to call on some piece of back-end server functionality that is ordinarily not available in HTML alone. -->

```
<center>
<%
  out.println("<h2>Hello World!</h2>\n");
%>
</center>
```

<!-- This next example is less contrived. When this Java code is compiled into a servlet on the back end, it queries the HTTP request object for various attributes. This JSP page is then responsible for rendering the results in HTML.

```
-->
<b>Information about your HTTP Request:</b>
<ul>
<li>JSP Request Method: <%= request.getMethod() %>
<li>Request URI: <%= request.getRequestURI() %>
<li>Request Protocol: <%= request.getProtocol() %>
<li>Servlet Path: <%= request.getServletPath() %>
<li>Path Info: <%= request.getPathInfo() %>
<li>Path Translated: <%= request.getPathTranslated() %>
<li>Query String: <%= request.getQueryString() %>
<li>Content Length: <%= request.getContentLength() %>
<li>Content Type: <%= request.getContentType() %>
```

## Instant Update

```
<li>Server Name: <%= request.getServerName() %>
<li>Server Port: <%= request.getServerPort() %>
<li>Remote User: <%= request.getRemoteUser() %>
<li>Remote Address: <%= request.getRemoteAddr() %>
<li>Remote Host: <%= request.getRemoteHost() %>
<li>Authorization Scheme: <%= request.getAuthType() %>
<li>Locale: <%= request.getLocale() %>
<li>Browser: <%=request.getHeader("User-Agent") %>
</ul>

</body>
</html>
```

### Using an MVC Pattern Across the Web

How does a model like MVC scale up to the Web? There are key differences to consider. The Web is stateless, which complicates the ability of model objects to stay synchronized with all the views (HTML pages) that have been generated based on the data model. Also, the number of distributed views and controllers interacting with the data model can be orders of magnitude larger than in stand-alone, rich-client GUI applications. Separating presentation logic from the data model is just as important for Web applications as it is for any other type of software system, but bringing the MVC model to the Web presents unique challenges that require careful consideration and planning.

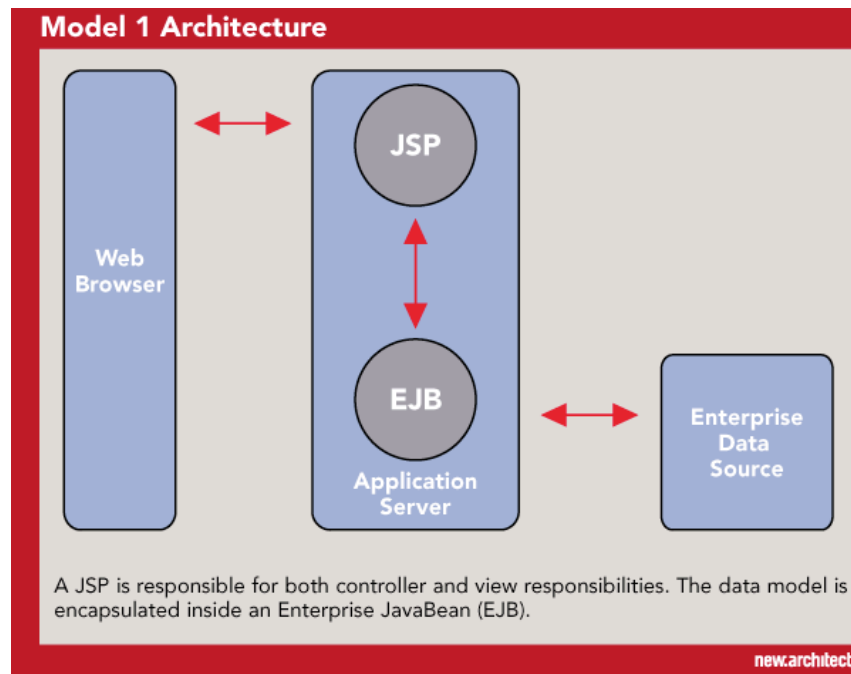
In many ways, effective separation of concerns is even more important for Web projects than it is for other types of software development. A Web application's UI is often developed by a team of graphic artists who have little or no knowledge of Java coding and syntax, rather than by a team of professional software engineers, as might be the case for a rich-client GUI. If the work is broken up along clearly defined interfaces so that each team can focus on its core competency, the project will be completed much more quickly. By effectively separating concerns early in the development cycle, project managers can ensure that programmers stay focused on programming, and designers on design.

Separation of concerns was the major driving force behind the development of JSP technology. Most HTML page content available on the Web is actually a combination of "static" template HTML markup along with dynamically generated information that is retrieved and/or updated anew with each and every HTTP request. JSP was devised as a means to separate the Web content (much of which could originate in back-end databases) from the way it's presented to the user (e.g., HTML code).

JSPs resemble HTML pages, except they contain embedded segments of Java code. They are a presentation-centric, declarative way of binding dynamic content with application logic. Because they run inside a Web server and are compiled into servlets prior to execution, JSPs are capable of invoking a range of

## Instant Update

server-side functionality, including transaction support, database access, state maintenance, security, distribution, and load balancing. However, system architects may want to carefully weigh the costs and benefits of invoking particular types of behavior—particularly database access—directly from a JSP page, rather than using some type of middle-tier abstraction layer.



Expansion of Diagram: A JSP is responsible for both controller and view responsibilities. The data model is encapsulated inside the Enterprise JavaBean (EJB).

JSPs are a more user-friendly way of managing presentation code than are servlets. Servlets are a programmatic tool and are best suited for low-level application functions that don't require frequent modification. Ultimately, however, JSPs are compiled into servlets and run on the Web server itself. The Java code embedded into the (original) JSP page is compiled "as is" into the resulting servlet. The HTML markup portion of the JSP page is compiled into a series of `out.println()` statements in the resulting servlet, saving developers the time of manually typing in mindless lines of `out.println()` statements and allowing graphic design teams to work directly on the template layout in HTML. See the Listing for an example HelloWorld.jsp page.

How does JSP technology relate to the MVC model? So far, it doesn't! What I've been describing is known in the JSP specification as a Model 1 architecture (see graphic). In a Model 1 system, the JSP page alone is responsible for processing the incoming request and replying to the client. The data model is usually represented by an entity Enterprise JavaBean (EJB), or possibly by a session EJB that is acting as a façade for an entity EJB. Model 1 succeeds in separating content from presentation, but can result in a significant amount of Java code being added to JSPs. This in turn can slow down a large project if the JSPs are being created and maintained by a team of graphic designers, as is often the case. Moreover, in Model 1, JSPs are statically linked to one another. This makes Web application functionality

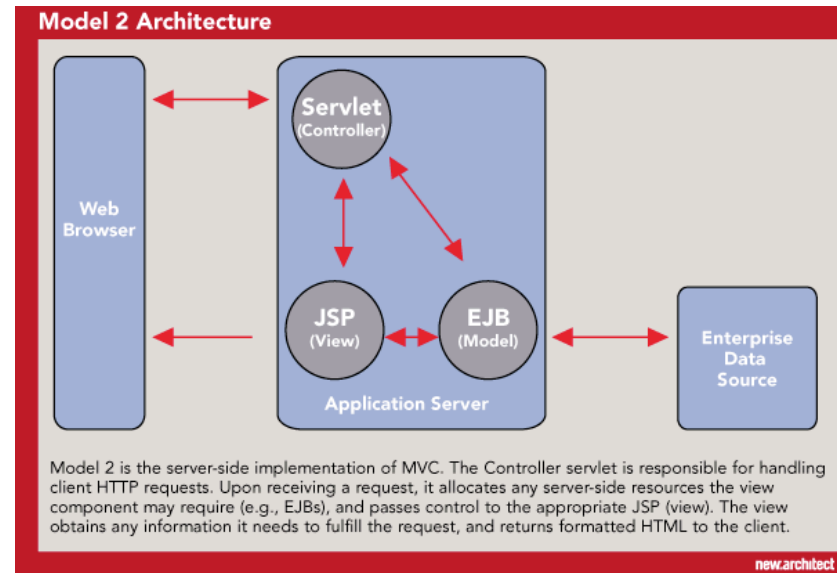
## Instant Update

less reusable, and can complicate adding and modifying client types, data views, and workflow. Each JSP page is also individually responsible for managing application state and verifying client authentication and security.

Model 2 (see graphic) is an enhancement to the Model 1 architecture that is designed to address some of these shortcomings. Model 2 is the server-side implementation of the MVC pattern we discussed earlier. Here there is a separation between the components responsible for presentation (JSPs) and the components responsible for processing HTTP requests (controlling servlets).

Servlets, acting as controllers, are responsible for creating any beans or other server-side resources that may be used by the presentation components. The servlets also decide (based upon user input extracted from the HTTP request), which presentation component gets the request. This lets the servlets act as a single entry point for the application, making the management of application state, security, and presentation more straightforward.

Explanation of Diagram: Model 2 is the server-side implementation of MVC. The Controller servlet is responsible for handling client HTTP requests. Upon receiving a request, it allocates any server-side resources the view component may require (e.g., EJBs), and passes control to the appropriate JSP (view). The view obtains any information it needs to fulfill the request, and returns formatted HTML to the client.



In the Model 2 pattern, there is no processing logic within the presentation components (JSPs) themselves. Instead, the JSPs simply perform "classic" view functionality by extracting dynamic information from whatever server-side resources are necessary (e.g., EJBs). JSPs then insert that dynamic content into the static templates defined by the JSP, and pass along the response to the user. By eliminating unnecessary processing logic from JSPs, this technique makes JSP creation and management easier for graphic design teams.

## Instant Update

### Model 2 Frameworks

A number of frameworks are available to help developers architect MVC-compliant interfaces for their Web applications. Perhaps the best known is Struts ([jakarta.apache.org/struts/](http://jakarta.apache.org/struts/)), an open source project that is part of the Apache Jakarta initiative. The Struts package provides a unified set of reusable components for building user interfaces that can easily be adapted for any Web-based connection (e.g., HTTP requests, WAP, or even standard socket-level applications). Struts ships with a controller servlet, custom JSP tag libraries, and some utility classes.

Struts has a relatively steep learning curve, but most programmers find it worth the effort. If, however, you're looking for something a little easier to manage so that you can get up and running a bit faster, Maverick might be the way to go. Maverick offers many of the same features as Struts, as well as some unique XSLT transformation features.

Different software architectures are applicable in different contexts and use cases. If you're rapidly prototyping a proof of a concept, chances are that a simple Model 1 pattern will be sufficient for your requirements. If, on the other hand, you are developing an application that you intend to roll into production and maintain for years, you should design with flexibility, extensibility, and modularity in mind. Software patterns like MVC and Model 2 provide a powerful design technique that will help ensure that

your code remains in use for years to come.

--

Contact Paul Sholtz:  
Phone: 917.438.7087  
E-mail: 646.489.5055