JSP technology -- friend or foe

An old Java technology hand and new Enhydra partisan, the author urges developers to consider alternatives to JavaServer Pages (JSP) servlets when choosing an approach to coding Web applications. JSP technology, part of Sun's J2EE platform and programming model, serves as a solution to the common dilemma of how to turn drab content into a visually appealing presentation layer. The fact is, Web developers aren't uniformly happy with JSP technology. Since many variations on the Sun technology are now available, you can choose from a number of presentation technologies. This article takes an in-depth look at JSP coding and explores some attractive alternatives.

Presentation technology was designed to transform plain ol' raw Web content into content wrapped in an attractive presentation layer. JavaServer Pages (JSP) technology, Sun's presentation model and part of the J2EE platform, has received significant attention. There are both advantages and disadvantages to using JSP technology, and Web developers should be aware of the good and the bad -- and know that they don't have to be limited to this single technology. In fact, these days a number of presentation technologies are available. This article begins by defining the problems presentation technologies were designed to solve. It then examines the specific strengths and weaknesses of the JSP model. Finally, it introduces some viable alternatives to Sun's presentation technology.

The premise

Today, a decade beyond those fledgling Windows applications, we are still dealing with this huge shift in the presentation paradigm. The woeful Visual Basic and C programmers who remain now find themselves working either on back-end systems or Windows-only applications, or they have added a Web-capable language such as the Java language to their toolbox. An application that doesn't support at least three of four ML-isms -- such as HTML, XML, and WML -- is considered shabby,not an outright failure. And, of course, that means we all care very deeply about the ability to easily develop a Web presentation layer.

As it turns out, using the new Internet, and all the languages we have at our disposal -- Java, C, Perl, Pascal, and Ada, among others -- hasn't been as easy as we might have hoped. A number of issues creep up when it comes to taking the programming languages everyone used

for back-end systems and leveraging them to generate markup language suitable for a client. With the arrival of more options on the browser (DHTML and JavaScript coding, for example), the increase in graphic artist talent in the Web domain, and tools that could create complex interfaces using standard HTML, the demand for fancy user interfaces has grown faster than our ability to develop these front ends to our applications. And this has given rise to presentation technology.

Presentation technology was designed to perform a single task: convert content, namely data without display details, into presentation -- meaning the various user interfaces you see on your phone, PalmPilot, or Web browser. What are the problems that these presentation technologies claimed to solve? Let's take a look.

A bit of history

Before diving into an explanation of presentation technology, it's helpful to fill in some details on the situation that led to the birth of the technology. Just 10 short years ago, the term thin client was a novelty. We still lived in a world of desktop applications, powered by wimpy 286 microprocessors with 14-inch monitors that we squinted at. Boy, have times changed! Now my desktop does nothing but power a Web browser, while servers from Sun, IBM, HP, Compaq, and the rest churn out computations, business logic, and content. And that little monitor? Replaced by flat-screen, plasma, whopping 21- and 25-inch beauties. Why? So we can see the intricate and complex HTML displays that serve as a front-end to these powerful applications. No longer does a clunky interface suffice; now we expect flashy graphics, moving images, color-coordinated presentations that would look good in any room in the house, and speedy rendering to boot.

The promise of JSP technology

Now, on to the specifics of JSP coding. The promise of JSP technology is to supply the designer and developer the only presentation technology they will ever need. JSP technology is part of the J2EE platform, which is the strongest show of support Sun can give one of its Java products. To give you an idea of how prevalent this solution is, try running a search on 'JSP' at amazon.com; you'll find more books devoted to JSP technology than about almost any

other single Java API. Before I dive into the specific problems that JSP technology presents, you need a clear understanding of what it claims to do.

Segregation vs. integration

The primary purpose of presentation technology is to allow a separation between content and presentation. In other words, business logic units (presumably in some programming language like C or Java) don't have to generate data in a presentation-specific manner. Data, or content, is returned raw, without formatting. The presentation technology then applies formatting, or presentation, to this content. The result is an amalgam of data surrounded by and intertwined with graphics, formatting, colors, and logos.

Work vs. rework

Besides the separation of content and presentation, another measure of a presentation technology's usefulness is the amount of rework that it eliminates. The divergence of presentation and content enforces a divergence in the roles of those developing the content. A programmer can focus on the raw content presented in the examples above, and a graphic artist or webmaster can attend to the presentation. A slight overlap of roles remains, however, in the process of taking the presentation -- or markup -- designed by the artist and applying it to the content the programmer's code delivers.

In the simplest case, the artist supplies the markup, and the developer provides code and also plugs the markup into the presentation technology. The application is "started up," and the content magically becomes a user interface. Of course, as we all know, development rarely ends there. Next come revisions and changes to the interface and new business rules that must be coded. This is where the true test of the presentation technology's flexibility comes into play. While it is usually simple to update the raw content being fed into the presentation layer, rarely can the graphic artists easily edit their original work. Changes to the presentation layer are common (we've all been victim to marketing departments changing this or that). So now a problem arises: what do the designers change to tweak their work? The original markup language page they gave to the developer? Probably not, as that page has most likely had custom tags or code inserted (JSP pages, template engines), converted to a Java servlet, or changed into something totally unrecognizable.

Often the designer must rework the original page and resubmit this page to the developer. Then the developer has to reconvert this page to the specific format needed for use in the presentation technology. Alternatively, the designer has to learn a scripting language or at least know that which areas of the page's source code from the developer are off limits. Of course, this is an error-prone, dangerous way to operate. Once you've determined that a presentation technology allows a clean split between content and presentation, you should try to ensure that a minimum amount of rework is necessary in order to make presentation changes.

Content vs. presentation

Above all, JSP technology is about separating content from presentation, foremost in Sun's published set of goals for JSP pages. This mixing of hard-coded content with runtime variables presented a horrible burden on servlet developers. It also made making even minor changes to the presentation layer difficult for the developer.

JSP technology addresses this situation by allowing normal HTML pages (and later, WML or other markup language pages) to be compiled at runtime into a Java servlet, essentially mimicking the out.println() paradigm, without requiring the developer to write this code. And it allows you to insert variables into the page that are not interpreted until runtime.

Code vs. markup

Second on the JSP technology's list of features is something that might raise a bit of concern. JSP coding lets you insert Java code directly into a page of markup. To understand why this decision was made, recall that when the JSP specification was being developed, Sun's competition from Microsoft was at an all-time high, primarily due to the success of Microsoft Active Server Pages (ASP). The similarity of the name JavaServer Pages to Active Server Pages was not merely coincidental. And the ability to mimic many of ASP's features was also intentional. So JSP authors were given the option to add Java code into their markup.

Designer vs. developer

A final (and admirable) goal of JSP technology worth mentioning is that it seeks to establish clearly defined roles in the application development process. By ostensibly breaking content from presentation, JSP technology creates a clearer distinction between the designer

and developer. The designer creates markup, using only standard HTML, WML, or whatever language is appropriate, and the developer writes code. Of course, many designers today have learned JavaScript, so it should come as no surprise that many of these same designers have begun to learn JSP coding. Often, instead of just doing pure markup, they encode a complete JSP page and hand it over to the developer. Then the usual tweaking takes place, and the developer puts the JSP page into place as a front-end for some portion of the overall application. The key, though, is that many designers do not learn JSP coding, so it must also be workable in that environment.

The problems

I've spelled out what a good presentation technology should provide, as well as the specific problems that JSP technology seeks to address. Now, I'm ready to cut to the chase: JSP technology, while built on good ideas, presents quite a few problems. Before you choose to use JSP coding in your applications (which you might still do), you should at least be aware of possible pitfalls.

You should also be aware of a facet of the J2EE programming platform that is often ignored: just because an API comes with the platform doesn't mean you have to use it. As silly as this sounds, many developers are struggling with the JSP, or EJB, or JMS APIs, thinking if they don't use these APIs, their applications somehow won't really be "J2EE applications." In fact, the platform boasts more APIs than most applications need. If you have problems with or doubts about JSP technology, you don't have to use it! Take a close look at both the positives and the negatives before choosing to use JSP technology in your applications. Let's take a look at some of the negatives.

Portability vs. language lock-in

JSP technology locks you into a specific language. This point shouldn't be given too much weight. Java technology for enterprise applications (in my opinion, at least) is the only language choice. And there are no language-independent solutions in this space anyway. Of course, at this stage of the game, I'm disregarding the Microsoft .NET platform for the smoke and mirrors it is. Only time will tell whether that platform will develop into one that is truly language-independent. (I'm more than a bit dubious.)

Still, choosing JSP technology forces you to use the Java language, at least for presentation and content. While CORBA can be used for business logic, JSP coding does necessitate some familiarity with servlets as well as the core Java language. Since many developers come to JSP coding through the J2EE platform, this doesn't usually present a problem.

Mingling vs. independence

Throughout this article, I've come back to the idea of separating content from presentation. You're probably pretty sick of hearing about this, so now's the time to determine whether or not JSP actually accomplishes this goal. As I've already discussed, JSP claims to have been designed for this separation purpose, and therefore we should assume it achieves its objectives, right? Not necessarily.

Single-processing vs. multi-tasking

Ideally, as discussed above, a designer ought to be able to perform a single process, working purely on graphic design, and a developer should be able to focus purely on coding. So the designer should be able to work on a page after it has been converted to an application-suitable format. In the case of a JSP page, that would be after JavaBeans have been imported, inline coding has been inserted, and custom tag libraries have been added to the page. The problem is that some designers use HTML editors, such as HoTMetaL, Macromedia Dreamweaver, or FrontPage, that do not recognize code scriptlets or tag libraries, which means the designer effectively receives only a partial page. Imagine the difficulties when tag libraries or code fragments generate rows of a table, or other formatting details for the page. Designers using the incompatible HTML editors can't see what those elements look like. When designers can't easily revise pages after developers finish coding them, instead of clarifying distinct roles, JSP coding can cause them to merge: a developer must multitask, becoming developer, designer, and more.

Unconvinced about the importance of this feature? Then download the J2EE Reference Implementation and load one of the included JSP pages into a WYSIWYG HTML editor, such as Dreamweaver. The page immediately fills with yellow areas letting you know about

all the "illegal" markup contained within the page. Of course, the yellow results from the JSP tags and code, rather than any real error in the page.

To date, no JSP-capable WYSIWYG editors exist, and I have not heard of any efforts to build one. While template engines have this same problem, many Java-based solutions, such as my favorite, Enhydra, allow you to supply the markup page as input to the presentation technology. In this case, the designer can make changes as often as needed and resupply the markup page. Running the engine or compiler for the presentation technology converts it to the proper format, and no code changes have to be made (in the typical case). The result is the desired one: designers remain designers, and developers remain developers.

So, be wary of the promise of JSP technology as compared to the reality of what it delivers. In practice, to function in a JSP technology-driven environment you must either have your developers handle a large portion of the markup or have designers learn at least some JSP coding.