Rich Web Applications with Java and AJAX

June 2005

Stephen Maryka
CTO
ICEsoft Technologies Inc.

The Evolution of the Enterprise Application

Is evolution really circular? Or, maybe it's a never-ending spiral. This appears to be the case when we examine the history of the enterprise software application. In the early era of enterprise computing there were no trade-offs made with regard to an appropriate application model. There was only one option—a thin-client model leveraging centralized servers and delivering presentation to the user via a dumb terminal.

With the onset of the personal computer, the model changed, and power to the desktop had a strong evolutionary effect on the application model, moving it away from centralized computing toward a distributed model where individual computers hosted their own set of applications. This fat-client model pushed computation and presentation to the client. With advancement in graphics technologies, it became feasible to provide the user with a rich interactive presentation environment for applications. But these advancements did not come for free. Power to the desktop resulted in skyrocketing costs for deployment, management and maintenance of applications. Total cost of ownership for applications within the enterprise increased dramatically.

Next came the Internet era, into which the web application was born, and we spiraled back to a thin-client model. This time around, the application's presentation was delivered as markup (HTML specifically) and the web browser (the modern dumb terminal) rendered that markup to generate the user presentation for the application. For the enterprise, this new thin-client model reversed the total cost of ownership trend from the previous era, but again, not for free. The crude user interaction model for the web seriously impacted the richness that could be delivered to the application user, resulting in less effective applications and impeded the total cost of ownership gains that the thin-client model touts.

So what's next in the evolution of the enterprise application? Are we doomed to spiral back toward a thick-client model in order to achieve the application richness that we have come to expect? Microsoft would like to think so as they push forward with their vision of the Universe—namely Longhorn and Avalon. Replacing the web model entirely seems a bit far-fetched, even for Microsoft, so we ask ourselves, 'Can the existing web model be leveraged to deliver the superior presentation capabilities we demand while continuing to drive down the total cost of ownership for enterprise applications?' From the business perspective, can we create industrial strength enterprise applications that transform the user experience without increasing total cost of ownership? Must we spiral backward to the thick-client model to achieve richness, or can we leap forward within the existing thin-client model?

This paper explores this possibility further through the examination of current trends, dissection of the underlying problem, and the introduction of an approach that delivers a superior user experience to web-based enterprise applications, while minimizing the total cost of ownership of those applications by preserving the thin-client model.

State of the Art—A Proprietary Hodgepodge

The J2EE technology stack is a shining example of a standards-based approach that works. Look no further than the number of J2EE applications deployed today to validate its success—over 250,000 J2EE enterprise application deployments, and growing by more than 30,000 per year. The J2EE stack provides a solid foundation for both B2B and B2C applications, and a multitude of commercial and open source solutions exist for the development and deployment of these applications. Naturally, a very strong

development community has matured around J2EE, and the Java Community Process continues to drive new capabilities into the stack. One of the newest additions is JavaServer Faces (JSF), which finally delivers a comprehensive, component-based application model for client-presented web applications. So it would seem that J2EE delivers an industrial-strength, standards-based solution for the development of rich web applications today. Unfortunately, while JSF undoubtedly delivers a superior component-based framework for web applications, it is a server-side technology that relies on markup-based presentation and the existing web application model. As such, JSF is exposed to all the same limitations that seem to be driving us back toward the thick-client model.

It is apparent that while J2EE provides a solid technology foundation and has fostered a vibrant development community, it still struggles in the final mile when we leave the server environment and deliver content, and supposedly, the rich user experience to the browser. So how has industry compensated? Well, up until recently it has been through a variety of proprietary approaches that require proprietary markup formats and rely on applet or plugin-based support at the browser to create the desired richness. While these approaches can be effective, they don't typically mix well with existing web content and often require a proprietary development environment, which forces developers away from their core competencies of J2EE development. Last, these approaches begin to drive back up total cost of ownership as both Java Applets, and browser plugins introduce browser and operating system idiosyncrasies. At enterprise scale, the maintenance issues become significant quickly and can have a devastating impact on total cost of ownership. As a result, these proprietary approaches have been largely relegated to green field applications and have gained little, if any, traction in the enterprise.

AJAX – The JavaScript House of Cards

If we dismiss proprietary approaches, as most of the J2EE world has, what are we left with? Today, a tremendous amount of energy is being expended on the one standards-based, client-side mechanism available to us—namely, JavaScript. We all know JavaScript as the script-based Java mutation that has succeeded in delivering some basic rich features to web applications. It achieves this through manipulation of the web application's UI via the DOM representation of that UI in the browser. JavaScript also facilitates client-side response to user interaction with the application UI. Thus we get dynamic behavior such as buttons changing their presentation when we mouse over them. The user of JavaScript and HTML is often referred to Dynamic HTML or DHTLM.

Most recently, industry has pushed beyond basic DHTML and has begun to exploit a server communication mechanism in JavaScript called XMLHttpRequest. This activity has spawned a new generation of web applications based on an approach referred to as Asynchronous JavaScript and XML or AJAX. The evidence is clear when you look at sophisticated AJAX-based applications like Google's Gmail™; AJAX can deliver a rich user experience within the existing web application model. So, problem solved, right? Well, not exactly. Developing sophisticated AJAX-based applications is fraught with peril. To begin with, JavaScript is not Java. It is not an industrial-strength language with industrial-strength development tools, so writing, debugging, and maintaining AJAX applications is difficult. Also, JavaScript is not the core competency of all those J2EE developers out there, so now you need to recruit some of those scarce resources that are forging the AJAX path. Even with that expertise in hand, you are not out of the woods yet. What about the idiosyncratic nature of JavaScript implementations within different commercial browsers? There is nothing more rewarding than maintaining scads of browser-specific code

within your application—was that one application or four applications that you are trying to maintain? And what about those nasty JavaScript memory leaks? What used to be a short-lived problem (the life time of a single web page) is now a full-fledged issue as single page AJAX applications can hang around indefinitely chewing up resources. And don't forget about all those security issues. Not only is all your source code readily available from the browser's source view, but also your corporate data is traveling over the network in clear text XML documents. Financial institutions are not likely to be thrilled with those security features. The bottom line is, ad hoc AJAX-based applications will leave your total cost of ownership equation in tatters. There must be a way to harness the power of AJAX within a standards-based application model like JSF to deliver truly industrial-strength solutions with the rich features that enterprise applications are demanding.

The Root Problem With AJAX

So it seems that even though AJAX-based solutions are in their infancy, serious issues with the approach are mounting rapidly. Let's examine why that might be. Generally, the way we achieve increased richness in an AJAX application is to replicate a subset of the application data model and business logic in the client browser using JavaScript. We then pass that replicated data through the replicated business logic to cause some sort of intelligent manipulation of the application's presentation via the browser DOM to create a richer user experience. That last bit—manipulation of the presentation layer via the browser DOM—is the only bit that JavaScript was originally intended to do. The rest of it is pushing JavaScript outside its intended purpose, which typically spells trouble. Furthermore, the approach pretty much dismantles the Model View Controller architecture at the root of the web application model, forcing duplication of data and business logic between client and server. This duplication opens the door to divergent functionality between client and server, and compounds the maintenance issues for applications. Once again the root issue is escalating complexity, which yields skyrocketing total cost of ownership, and makes the AJAX approach questionable for enterprise applications.

Towards a Solution—Harnessing AJAX

So if we reject complex, JavaScript-based client-side application replication as an approach, can we still leverage AJAX mechanisms to achieve richness? Let's start by putting JavaScript back in its place, so we will relegate it to its primary functions of basic presentation layer manipulation via the browser DOM and event generation based on user interaction with the presentation. Next, we move the data model and business logic back where they belong, onto the server. J2EE provides an appropriate foundation to support this. Now we are left with the same old problem that we have always had. How do we bridge between the server-side application logic and the client-side presentation layer to deliver that rich user experience? The answer is alarmingly simple if we move the presentation layer to the server and invent a mechanism for intelligently replicating it back to the client browser.

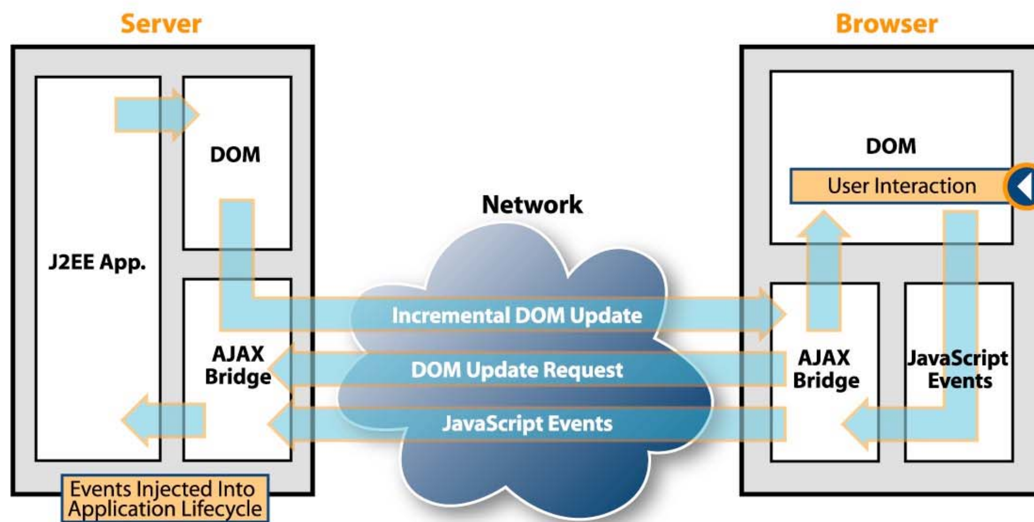
In the browser, the presentation layer is ultimately represented in a DOM, and it is that DOM that we move to the server for manipulation. When application state changes cause presentation changes, those changes manifest themselves in the server-side DOM. From here, incremental DOM updates are transmitted to the browser via a lightweight AJAX-based bridge where the changes are reconstructed and the browser DOM is modified appropriately. This mechanism results in superior presentation updates

because only minimal incremental changes are required, not full page refreshes as we have come to expect from web applications. Furthermore, the approach facilitates asynchronous presentation updates driven from the server as application state changes. And yes, the AJAX bridge overcomes most of the general concerns that we voiced earlier with regard to ad hoc AJAX. Specifically, the AJAX bridge:

- Is small and well contained, deals with browser idiosyncrasies and eliminates JavaScript memory leaks.
- Eliminates data and business logic replication at the browser, thus eliminating security issues within the application.

We have now addressed most of the issues related to improving the presentation element of the Application, but we need to address user interaction with that presentation. To achieve this we hook the JavaScript event model back into the application event model via the same AJAX bridge. This means that our application can react to user interaction with the presentation in real time and effect changes in the back end application data model. Ultimately, these state changes are reflected in the application's presentation, and the incremental presentation changes can be propagated back to the client for presentation to the user. We now have an architecture that facilitates rich web applications and leverages server-side, industrial-strength, standards-based J2EE technologies to achieve it. This basic architecture is illustrated in Figure 1 below.

Figure 1: Basic J2EE/AJAX Architecture



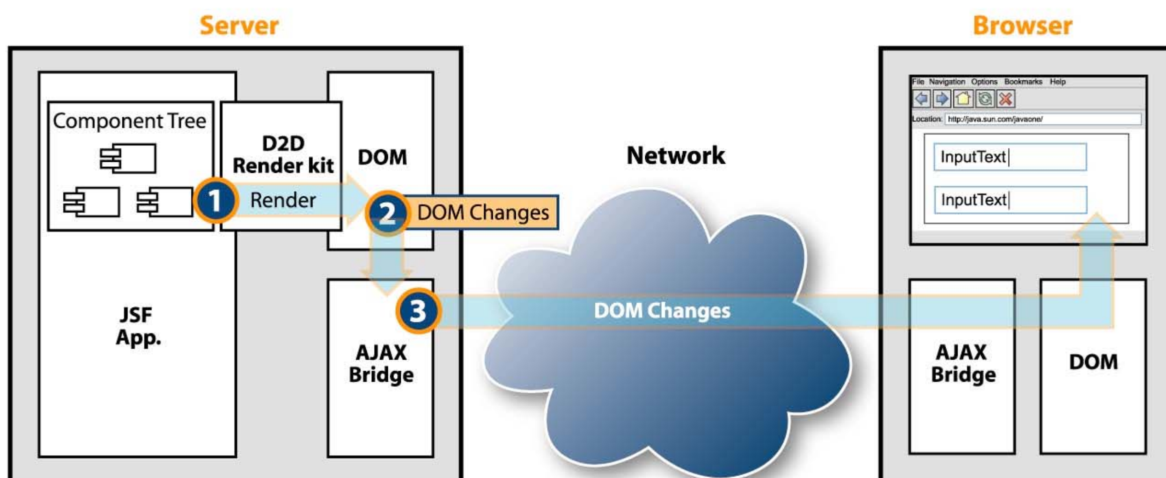
While this basic architecture has merit, there are a number of holes that need to be filled for an implementation, like causing a J2EE application to produce DOM as its presentation output. We now examine an approach that fills these holes and results in an effective implementation of the architecture. We start with JSF as the foundation.

JSF with Direct-To-DOM Rendering and Incremental Update

One of the profound concepts in JSF is the RenderKit architecture. It provides separation between JSF component behavior, and the markup that represents those components in the presentation. The RenderKit architecture facilitates plugging different RenderKits into the same application in support of different presentation environments; so you might render HTML to a desktop browser, and WML to a mobile phone, but maintain common server-side application logic. We leverage this RenderKit architecture in the implementation of the JSF/AJAX architecture illustrated above, and introduce a revolutionary technology called Direct-to-DOM rendering to achieve it.

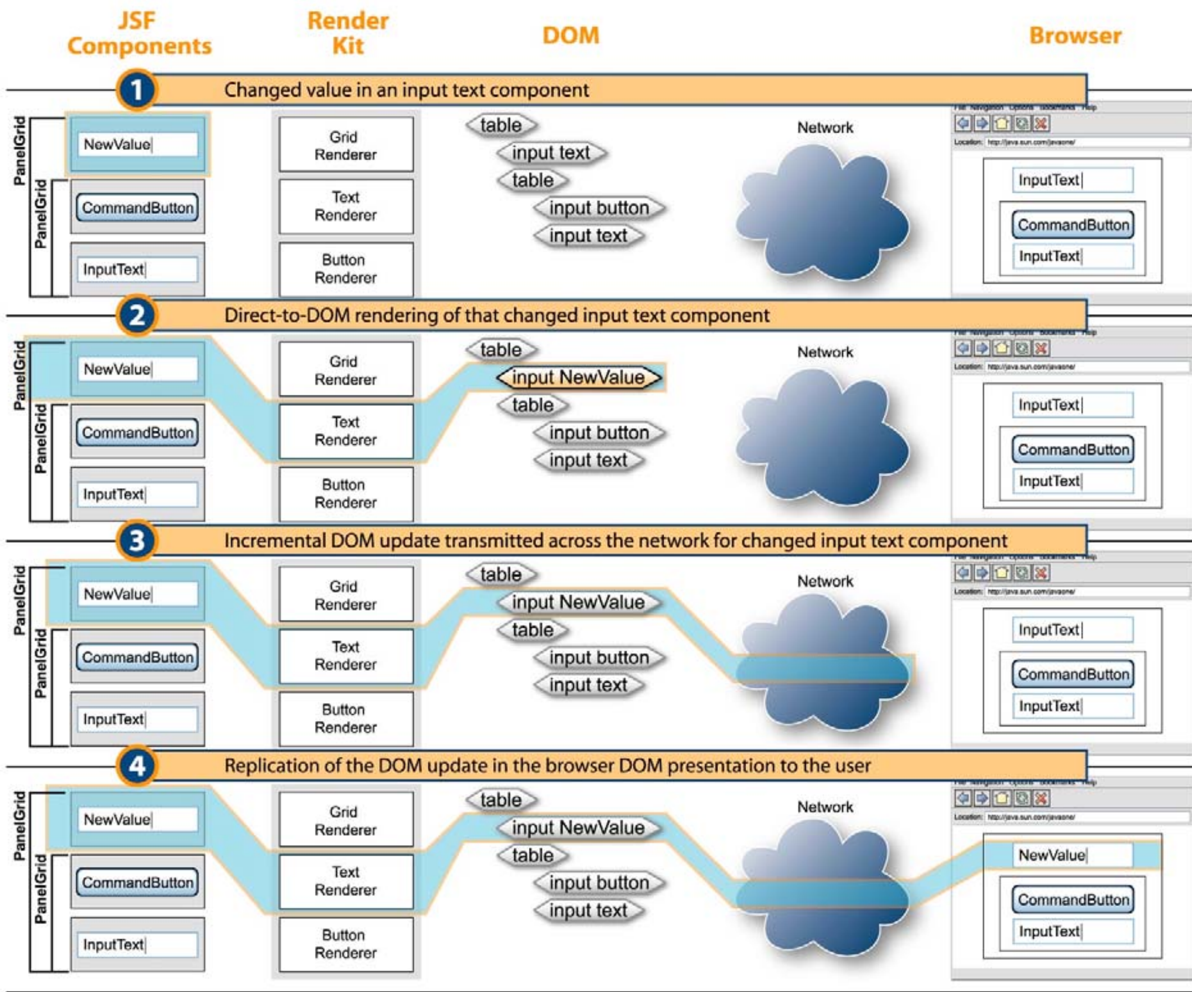
Direct-to-DOM rendering is just what it sounds like, the ability to render a JSF component tree directly into a W3C DOM data structure. During a standard JSF render pass, the component tree is traversed and each component renderer produces its output. Direct-to-DOM renderers produce their output into the server-side DOM. The DOM mutations that result are packaged up and delivered to the browser via the AJAX bridge and reassembled to create the presentation for the application. The basic process is illustrated in Figure 2 below.

Figure 2: Direct-to-DOM Rendering



So, if we equip the JSF framework with a Direct-To-DOM RenderKit, we have a mechanism for efficiently generating and maintaining a server-side DOM, and if we render only components that have changed, we minimize the number of DOM mutations that must be replicated at the client-side DOM. The following sequence of diagrams illustrates an incremental presentation layer update using Direct-to-DOM rendering.

Figure 3: Direct-to-DOM Incremental Update



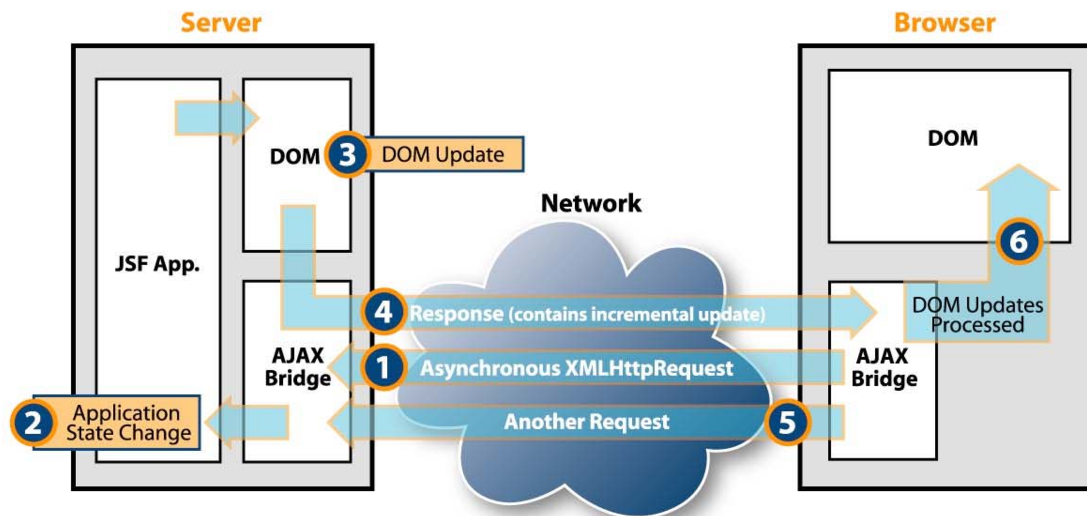
The combination of Direct-to-DOM rendering and incremental update completely change how we think about web application page design and opens the door to a myriad of rich, interactive features that are not shackled to the standard page refresh model.

Asynchronous Presentation Update

Now we not only want smooth incremental updates for presentation changes, but we want to be able to drive those changes to the client in an asynchronous fashion when the application state changes. To achieve this, we prime an ongoing update request loop from the client-side AJAX bridge with an asynchronous XMLHttpRequest for incremental presentation changes. This request gets fulfilled at the server when the next set of incremental DOM updates is prepared. The response is transmitted back to

the browser and a new request is issued prior to processing the DOM updates, thus facilitating ongoing asynchronous presentation updates, as illustrated in Figure 4 below.

Figure 4: AJAX-driven Asynchronous Presentation Updates



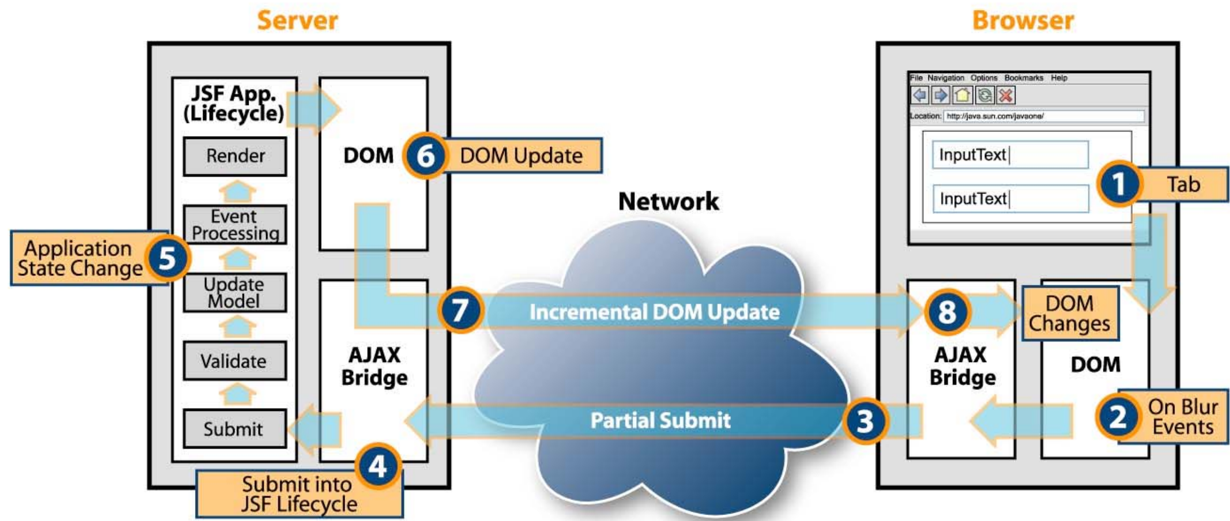
Within the backing application, logic trigger points can be identified that cause presentation updates and these trigger points can force a JSF render pass to generate the required changes to the presentation. For example, a simple clock bean could produce and consume events once per second and use those events to trigger an update of the clock presentation on a page. With Direct-to-DOM rendering the clock will tick smoothly even as the user interacts with the page.

Partial Submit—Intelligent Form Processing

While our basic architecture in Figure 1 illustrates a general-purpose event mechanism tying user events back in the application event model via AJAX, the JSF lifecycle is based on the standard HTTP Submit mechanism. So we look to leverage this mechanism to create a fine-grained user interaction model for our JSF application. In the normal JSF lifecycle, a submit causes input validation, model update, event processing, page navigation, and finally presentation rendering. Of course with Direct-to-DOM rendering, this process results in smooth and efficient presentation updates, but it does nothing on its own to enhance the form entry process. There is no client-side form validation, or intelligent client-side form-processing model available. Again, we shy away from duplicating this kind of logic in JavaScript on the client, and look to leverage server resources to achieve this fine-grained interaction with the user.
A mechanism called partial submit is introduced that allows us to connect JavaScript events back into the JSF application lifecycle through the standard submit mechanism. The partial submit causes a form submit, but restricts the validation process to controls that the user is interacting with. The rest of the JSF lifecycle runs normally, allowing the business logic to react to the user input through valueChangedEvents or other standard JSF mechanisms. Presentation changes may occur as a result, and these changes will be processed and delivered through the normal Direct-to-DOM rendering mechanism back to the client. The end result is a fine-grained user interaction model that facilitates intelligent form

processing for JSF applications. The application developer controls the level of granularity in a manner appropriate for the application. Interaction may be warranted on a per keystroke basis, between fields, or between sections of a form. Figure 5 illustrates a partial submit generated by an onBlur event resulting when the user tabs between fields in a form.

Figure 5: Partial Submit using onBlur

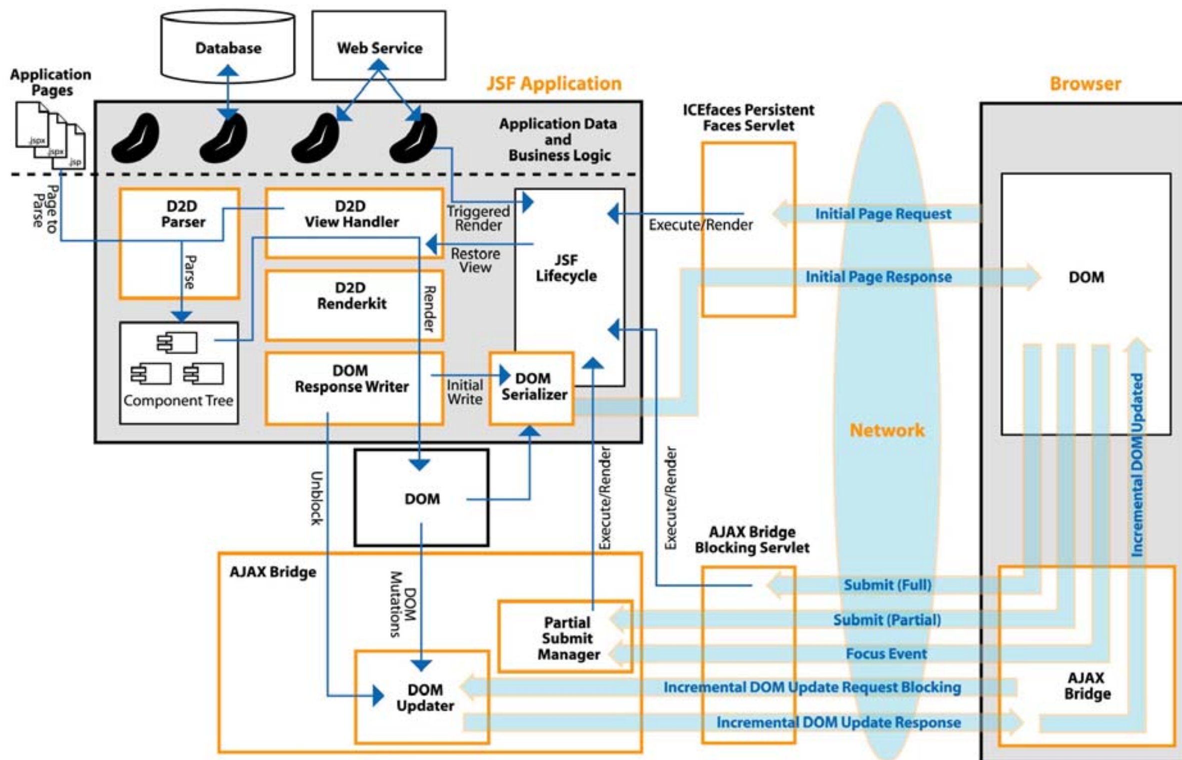


ICEfaces–Creating Rich Web Applications Today

So we have illustrated a number of key concepts that enable us to build rich JSF-based applications, but how can these concepts be leveraged today to create superior web applications? ICEfaces is the commercial embodiment of these concepts, and as such, provides a comprehensive development and runtime environment for them. ICEfaces arms you with an industrial-strength, JSF-compliant solution for efficiently building and deploying rich web applications in a pure thin-client model. There is no need for proprietary browser plugins, or proprietary API extensions, or brittle JavaScript-laden custom components. Take any JSF application that you have today and unleash its true potential by using ICEfaces and Direct-to-DOM rendering. Your applications will never look or act the same again once you apply your imagination.

Figures 6 below provides a high-level view of the ICEfaces architecture, and how it fits into JSF, but for all the details see the ICEfaces product documentation at http://www.icesoft.com/support/devguides.html.

Figure 6: ICEfaces Architecture



Evolution Revisited

So it would seem that the evolutionary spiral back to the thick-client model can be averted. We have shown how JSF and AJAX can be combined to produce an industrial-strength solution for constructing rich web applications in a cost effective manner. We have shown that key mechanisms in ICEfaces such as Direct-to-DOM rendering and partial submit can transform an ordinary JSF application into something extraordinary with minimal additional development effort. The benefits of using ICEfaces for developing your rich enterprise web applications are numerous:

- Create a superior user experience and produce more effective enterprise web applications.
- Stay J2EE and JSF standards compliant.
- Produce industrial-strength solutions in pure Java.
- Minimize your dependency on complex and brittle JavaScript.
- Leverage industrial-strength J2EE development environments and tools.
- Draw resources from the mature and vibrant J2EE development community.
- Minimize total cost of ownership for applications through a pure thin-client model.

Visit the ICEsoft web site at www.icesoft.com to learn more about ICEfaces and begin to realize these benefits today.

Copyright 2005 ICEsoft Technologies, Inc. All rights reserved.

No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of ICEsoft Technologies, Inc.

The content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by ICEsoft Technologies, Inc.

ICEsoft Technologies, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this document.

ICEbrowser and ICEreader are registered trademarks of ICEsoft Technologies, Inc. in Canada and the United States of America.

ICEpdf and ICEfaces are trademarks of ICEsoft Technologies, Inc.

Sun, Sun Microsystems, the Sun logo, Solaris, Java and The Network is The Computer are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and in other countries.

Google and Gmail are trademarks owned by Google, Inc.

All other trademarks mentioned herein are the property of their respective owners.

About ICEsoft

ICEsoft Technologies, Inc. is the leading provider of Java web application and web client technology for enterprise developers.

Our portfolio of "Industrial Strength" Java products includes ICEbrowser, the most widely distributed Java browser in world and ICEpdf, the leading Java PDF rendering technology.

ICEsoft's new breakthrough product, ICEfaces, with Direct-to-to DOM technology, allows J2EE developers to develop a new class of web applications. ICEfaces is a revolutionary development and runtime environment for developing rich J2EE web applications in a pure thin-client model.

ICEsoft Technologies, Inc.
Suite 300, 1717 10th St. NW
Calgary, Alberta, Canada
T2M 4S2

Toll Free: 1-877-263-3822 USA & Canada
Main: +001 (403) 663-3322
Fax: +001 (403) 663-3320

For additional information, please visit the ICEsoft website:
http://www.icesoft.com/

June 2005§