RC 21628 (97505) 20DEC1999 **Computer Science/Mathematics**

Research Report

Developing Highly-Responsive User Interfaces with DHTML and Servlets

Katherine Betz, Avraham Leff, James T. Rayfield

IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Developing Highly-Responsive User Interfaces with DHTML and Servlets

Katherine Betz, Avraham Leff, James T. Rayfield
IBM T. J. Watson Research Center
{kbetz,avraham,jtray}@us.ibm.com

Abstract

Due to communication overhead and latency, Web-based user interfaces that rely exclusively on the server to refresh client screens suffer from poor performance. In this paper we show that implementation of the classic Model-View-Controller architecture on the client enables the client to refresh the screen in certain cases, and thus improves performance. This "dual-MVC" approach is discussed in the context of a sample Web-based application.

1. Introduction

When implementing Web-based user interfaces, it is tempting to rely exclusively on the server to refresh the client's screens. In this approach, whenever the client edits the screen (e.g., by clicking on a button), the request is forwarded to the server which:

1. Based on the client's input, modifies the state of the application's business objects.

2. Constructs an HTML representation of the relevant set of business objects.

3. Refreshes the client's screen by writing the HTML to the browser.

In terms of the classic model-view-controller (or *MVC*) architecture **[1][2]**: the

- **Model** (the set of business objects) resides entirely on the server.
- **View** resides on the client's Web browser, and the view-generating logic resides on the server (the code that generates the HTML).
- **Controller** resides partially on the client (the code that detects that the button was clicked and submits the HTTP request to the server), but mostly resides on the server (the code that receives the client's HTTP request and invokes the appropriate methods so as to update the state of the model's business objects).

In this server-centric approach, the main role of the client is to be a View of the server-side Model. Figure 1 sketches the architecture of this implementation approach. In this paper we introduce an alternative architecture, termed "dual-MVC", that provides better performance than the server-centric approach in certain cases.

2. Motivating Example

To help understand the performance issues addressed in this paper, we introduce a simplified example of a Web-based user interface and application. This example will be used to point out where traditional implementation approaches overlook potential performance improvements and how our approach of client-side screen refreshes can improve the response time of Web-based user interfaces.

Picture a "Help Desk" application in which help desk consultants use a Web interface to record and track reported problems. The consultant, in other words, is sitting in front of the Web-based user interface, and is using that interface to communicate with the Help Desk server. When a customer calls to report a problem:

1. The consultant, looking at the panel shown in Figure 2 asks the customer to specify an id so that the problem report can be associated with the right person; the id is entered in the appropriate entry field.

2. The client application validates the id by submitting the form to the server which queries a database to get information about the customer.

3. If the query is successful, the server sends the customer's *name* and *phone number* back to the client;

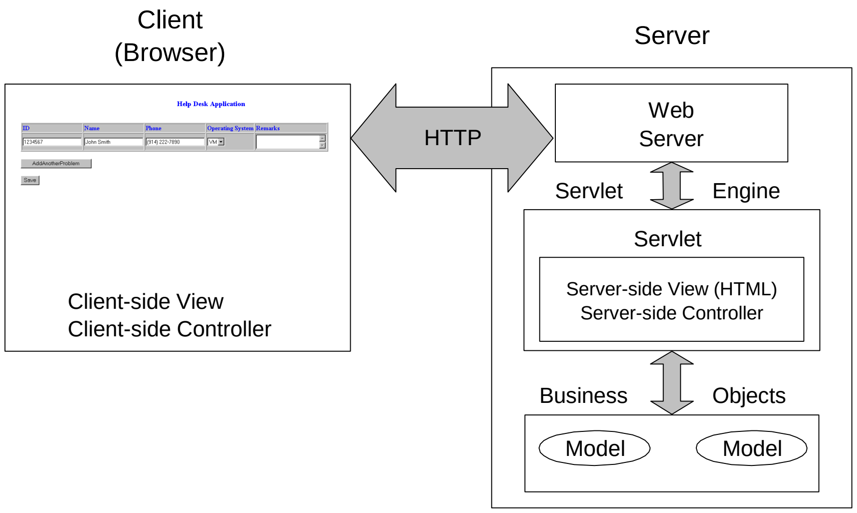


FIGURE 1. Model-View-Controller for "server-centric" user interface implementation

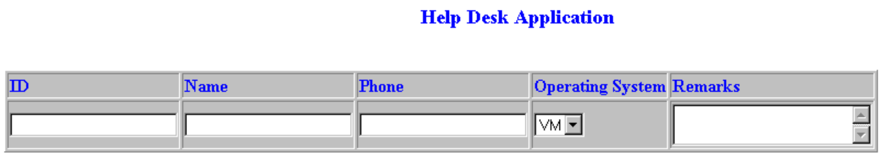
the consultant's screen is refreshed to look like the top panel of Figure 3.

4. The consultant asks the customer which operating system she is using. The server then refreshes the consultant's screen with a system-specific panel that must be filled in. If the customer is using Windows NT, the consultant selects this option from the pull-down list. The screen is refreshed, and now looks like Figure 3. The consultant queries the customer to fill in the *version*, *platform*, and *hostname* entries.

5. When the *hostname* is filled in, the client asks the server to return the IP address associated with the hostname.

6. When the consultant has sufficient information, he presses the *Save* button, and the problem report is transmitted to the server.

7. A client may have more than one problem to report. In such cases the consultant presses the *Add Another Problem* button, and the server refreshes the screen with an additional row in which the *ID*, *Name*, and *Phone* entries are already filled in.



Help Desk Application

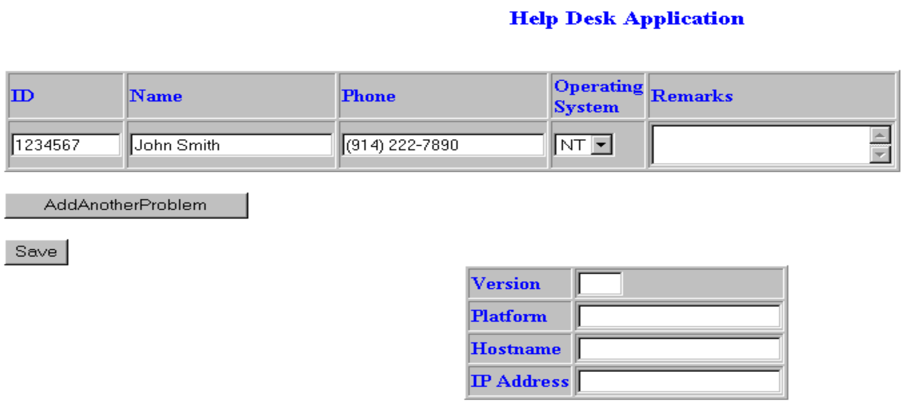
ID	Name	Phone	Operating System	Remarks
			VM	

FIGURE 2. Initial panel of Help-Desk application

3. The Problem

The "server-centric" approach described in Section 1 has the advantage of simplicity as it almost completely factors out the role of the client: developers are able to focus on the server part of the application. The problem with this approach is that the server must *always* be involved *whenever* the client's screen needs to be refreshed, and the resulting communication overhead and latency degrades response time. Entering *n* problem reports requires $3n + 1$ round-trip interactions between the web-browser and the web server.

In our sample application, two steps do not intrinsically require server involvement. Step **# 4** (where the screen is refreshed to show a system-specific configuration panel) and step **# 7** (where a new problem report row is added to the panel) require only information that is already available at the client. In such situations,


Help Desk Application

ID	Name	Phone	Operating System	Remarks
1234567	John Smith	(914) 222-7890	NT	

Version	
Platform	
Hostname	
IP Address	

FIGURE 3. Consultant's panel after customer has specified an NT-related problem

response time is improved by not involving the server in the screen refresh process.

Note that we do not claim that *all* screen refreshes can be done solely by the client. In our sample application, steps **# 2** (mapping the customer id to customer name and phone number), and **# 5** (mapping the hostname to the IP address) do require the server to refresh the screen since this information is maintained only on the server.

The point here is that many user interface applications contain screen refresh points in which the server need not be involved, and that performance can be improved if the client is able to do the screen refresh itself. In addition, the "thicker" the client -- e.g., if a small "host" database were maintained on the consultant's machine -- the more opportunities exist to exploit client screen refreshes.

3.1. A Partial Solution

The performance problems associated with the server-centric approach are well known, and are often addressed with a limited form of client-side processing. For example, syntactic validation of user input is done on the client to avoid the round-trip delay incurred by server-side validation. Scripting languages such as JavaScript **[5][6]** are good tools for such client-side processing.

This "mixed" (i.e., "server plus JavaScript") approach, however, ignores the type of optimization discussed above, since situations in which user input triggers a screen refresh are handled by asking the server to refresh the screen. Client-side refreshing of the screen requires that some subset of the server-side Model exist on the client as well. Without a Model, even if there *is* knowledge of how to refresh a screen, there is no version of the current state that *can* be refreshed.

3.2. DHTML *versus* Applets?

Because applets run on the client, and therefore can perform client-side screen refreshes, they offer another solution to the performance issues raised in this paper. Also, because they offer the programmer better control of user interface factors such as color and placement, applets might seem to be the better technology for Web-based user interface applications. However, we feel that the "browser incompatibility problem" imposes a considerable disadvantage for applets. Because clients in the real world use different browsers, each supporting different versions of the JDK, deploying an large-scale application based on applets is problematic. Admittedly, there are considerable differences between DHTML versions; however, as long as the application does not exploit advanced DHTML features, most browsers *will* support DHTML in common.

4. Our Approach

Our approach to implementing Web-based user interfaces exploits the potential performance improvement in which screen refreshes are performed on the client side whenever possible. This requires that the client side role of the application be enhanced relative to its roles in the server-centric or mixed approaches. Specifically, a Model corresponding to the relevant subset of server-side business objects must be maintained on the client; the state of this Model is typically more current than the state of the server-side Model. Since this approach requires that the client also maintain a Model-View-Controller of its own -- and not rely on the server-side Model -- we term it the *dual MVC* approach. If transient session information (e.g. name, ID) are contained in the client-side model, the display of Figure 3 can be generated without an interaction with the server. Entering *n* problem reports requires *n + 2* round-trip interactions under the Dual-MVC approach (instead of *3n + 1*).

Implementation of the user interface is done by defining Controllers -- that drive updates to the application's business objects -- using the *document object model* (or *DOM*) **[3][4]**, and declaring them in dynamic HTML (or *DHTML*) **[4]**. (Of course, XML**[10]**/ XSL**[11]** can be used to declare the DOM instead of DHTML.) The business objects can be of any type: in our current work Java clients access Enterprise Java Beans running on the server. The client communicates with the server *via* HTTP and server-side servlets **[7]** that implement the server-side Controllers and access the application's business objects. When the Model is refreshed, other servlets dynamically generate DHTML corresponding to the new View, and send the DHTML back to the client.

At a high-level, the implementation of the user interface is simply partitioned into server-resident and client-resident portions, with both portions containing Views and Controllers of the same Model. While not required by the dual-MVC approach, we find it useful to partition each client screen into two frames: an invisible frame that serves as a stable "anchor point" for the client-side portion of the implementation, and a visible frame with which the user actually interacts (see Figure 4). This is necessary to enable the DHTML running on the client to contain the code to write (in addition to the current frame) all possible subsequent frames that may need to be drawn in response to user input. Although HTML 4 features such as style sheets and layers provide hooks for "refresh in place", writing the DHTML to generate all possible frames is a tedious process. We therefore factor out the code generation function, and implement it in the invisible frame which is never rewritten on the client. This allows the visible frame to respond to user input by sending it to the invisible frame for processing.

4.1. Anchor Frame

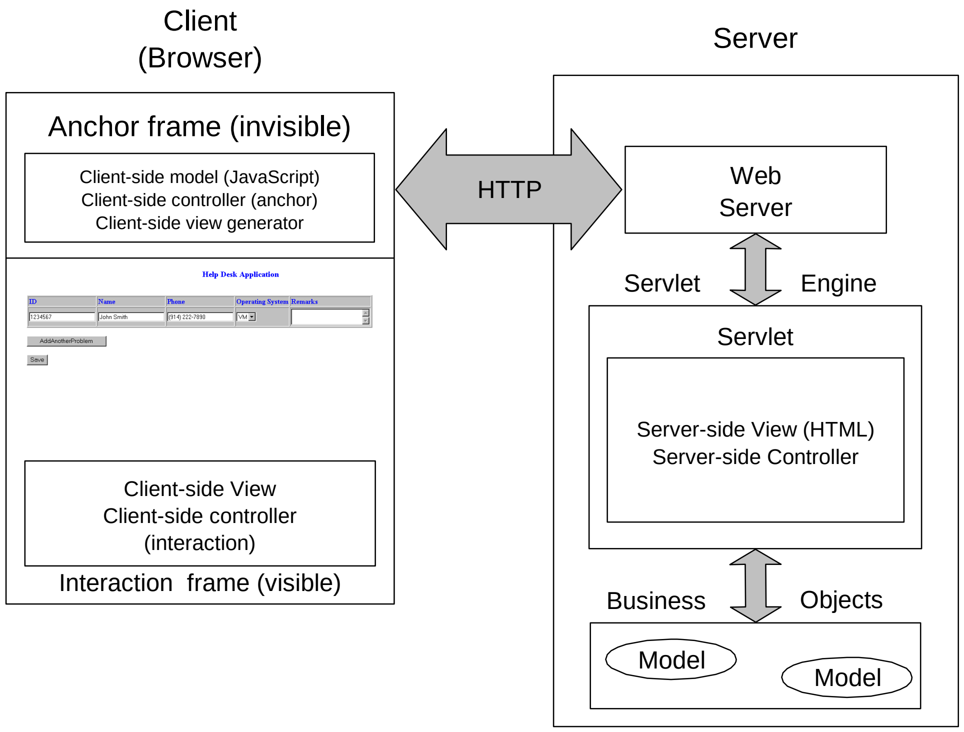
As we explained above, an anchor frame is needed to provide a stable point within the application from which the visible (interaction) frame (Section 4.2) can be refreshed "in place". Because it is invisible, the anchor frame does not declare its Model using DHTML. Instead, the Model is declared using JavaScript objects, and typically makes use of JavaScript variables to hold the relevant set of business objects.

Despite its invisibility, the anchor frame *has* a View: namely, it generates DHTML source that represents its Model, and then writes the DHTML to the interaction frame. Whenever a user inputs a change that does not require server processing, the interaction frame invokes the anchor frame's Controller (JavaScript code) which examines the user's input and updates the state of the anchor frame's Model.

4.2. Interaction Frame

The structure of the interaction frame is similar to that of the client screen under the "mixed" approach. That is, the frame is supplied with DHTML input from the anchor frame (the anchor frame's View). Because the DHTML contains JavaScript the browser engine is able to interpret and render this View on the fly. User inputs (e.g., button clicks) are processed by the interaction frame's JavaScript, which updates the anchor frame's Model by modifying its JavaScript variables. If the server does not need to be involved in the refresh, the screen refresh is performed when the anchor frame updates its View to reflect the changed state of its Model, and then writes the new View to the interaction frame.

If the server must be involved (e.g., the application must perform a database query), the anchor frame sends an HTTP request to the server; the server runs its Controller, updates its Model, updates its View and returns it



ID	Name	Phone	Operating System	Remarks
1234567	John Smith	(914) 222-7890	VM	

FIGURE 4. Dual-MVC architecture for "partitioned approach" user interface implementation

to the anchor frame. This View is then mapped to an updated anchor frame Model, thus closing the loop.

4.3. Using the Dual-MVC Approach to Develop an Application

In this section we describe how we implemented the help-desk application of Section 2 under the dual-MVC approach.

4.3.1. Defining the Server-Side and Client-Side Models. Figure 5 shows the Help-Desk application's object model.

A persistent version of this model is implemented on the server as a set of Enterprise Java Beans. The Model's objects are grouped on a per-session (i.e., single consultant/customer interaction) basis for convenient access *via* the HttpSession associated with a servlet's HttpServletRequest object.

On the client side, the model is maintained in a set of JavaScript variables:

- A **customer** Object (with *ID*, *name*, and *phone* properties) maintains customer state.
- A **problemReports** Array, in which each Array element's properties maintains the state associated with a single problem report.

4.3.2. Client-Side View Generation. As explained in Section 4.1, the anchor frame contains DHTML code that generates the View. In our implementation, we coded JavaScript functions such as *writeBlankFrame*, *writeOSSpecificFrame*, and *writeHelpDeskFrame* that emit HTML (*via* document.write commands) as a function of the state of the client-side Model. For example, if the Model does not yet contain a problem report, only HTML corresponding to Figure 2 is generated. If the Model's "current" problem is an NT problem report, the JavaScript function generate HTML corresponding to Figure 3.

4.3.3. Client-Side Updates to Client-Side Model.

Conceptually, this part of the application is fundamental to the dual-MVC approach because it enables the

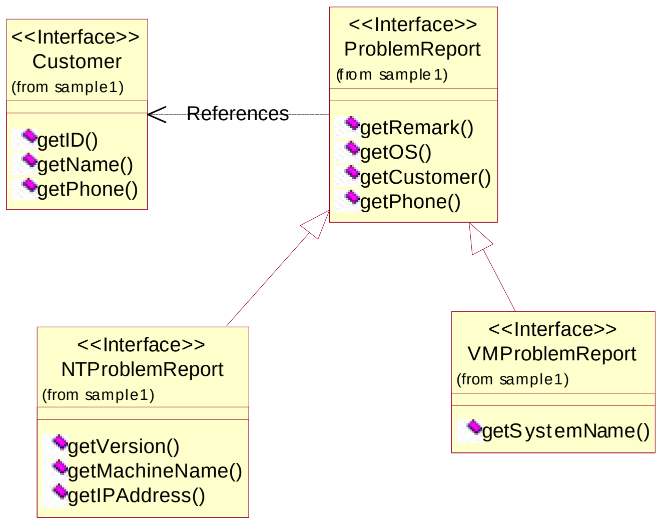


FIGURE 5. Server-side Object Model for Help-Desk application

screen to be refreshed without server involvement. In terms of implementation, however, this task is straightforward: DHTML code changes the state of the relevant JavaScript variables and then invokes the JavaScript "screen rewrite" functions (Section 4.3.2) to refresh the screen.

4.3.4. Client-Side Updates to Server-Side Model.

Clients modify the server-side model through a set of Controllers implemented as servlets:

- **ProcessHostNameServlet:** invoked when a customer supplies the name of the machine associated with the problem report.
- **ProcessIDServlet**: invoked when a customer supplies her ID.
- **ProcessSaveServlet:** invoked when the consultant saves a problem report.

(Servlet invocation typically causes the server-side to refresh the View so that the client is presented with a screen update. This process is described in Section 4.3.5.) These controllers are invoked *via* HTML *onChange* or *onClick* directives associated with the relevant input fields. This occurs because input field modification is associated with a JavaScript method which:

1. Extracts the relevant parts of the client-side Model from the JavaScript variables and from the Form's input fields.

2. Loads the Model's state into the hidden fields of a JavaScript *Form* object.

3. Invokes the Form object's *submit* method. This triggers the Form's *Action* attribute: namely, the servlet that will process the client's information.

This set of steps expands, in Step **# 1** of the help-desk example (when the consultant enters the customer's id), as follows:

1. The HTML for an empty Customer field specifies that the anchor frame's *getNamePhone()* function is invoked when the input field is changed.

2. The JavaScript *customer.ID* property is set to the value of the input field.

3. The "process ID" Form's *ID.value* is set to *customer.ID*.

4. The "process ID" Form is submitted.

5. The "process ID" Form's *Action* specifies the *ProcessIDServlet*, passing the *ID* as a hidden field.

The Controller servlets update the server-side Model. In the case of *ProcessIDServlet*, a *findByPrimaryKey(ID)* operation is invoked on the Customer Home to locate the Customer object. If no such object is found, the server throws an exception. Otherwise, the server-side Model loads the returned Customer object into the set of state associated with the consultant's current session. The server now has to generate an updated View that reflects the Customer's *Name* and *Phone*.

4.3.5. Server-Side View-Generation. In this task, the server has to generate (and return to the client browser) DHTML consisting of:

- the JavaScript code that declares the variables that provide the client-side View (Section 4.3.1), and the code that sets these variables to the state of the corresponding server-side Model.
- the JavaScript functions that generate the client-side View (Section 4.3.2).
- the JavaScript functions that provide client-side Controller function (Section 4.3.4).

Our implementation uses Java Server Pages technology **[9]**. JSPs enable us to separate the static content of the page (the JavaScript functions) from the dynamic content (the code that sets the state of the JavaScript variables that define the client-side Model. The dynamic portion is generated through JSP “scriptlets” that:

- set the JavaScript *customer* variable’s properties to those of the EJB *Customer* object.
- less trivially, generate and load a JavaScript *problemReport* Array from the Enumeration returned by a *findByCustomer* query on the EJB *NTProblemReport* Home. Problem reports for other platforms (e.g., *VMProblemReport*) are similarly accessed *via* queries to the appropriate Home and loaded into the client-side Model’s JavaScript variables.

In our simple application, a single JSP suffices to generate all of the Views required by the help-desk application. By using this JSP, the servlets discussed in Section 4.3.3 do not directly generate and return DHTML to the client. Instead, after updating the server-side Model, the servlets:

- Create a RequestDispatcher, and pass it the location of the JSP.
- Forward the processing of the client’s request to the RequestDispatcher.

4.3.6. Priming the Pump. How does this complex cycle of dual-MVC interaction get started? The consultant connects to the server by pointing her browser at *FramePage.html*; this file contains vanilla HTML to draw Figure 2 and *also* loads *startServlet* in an invisible frame. By entering data into the form, the consultant initiates *startServlet* processing. This creates a new, empty, server-side Model that will be associated with the consultant’s session, and then forwards the request to the JSP which results in the server generating the initial View (Section 4.3.5).

5. Future Work

When an application’s characteristics offer the opportunity for client-side screen refreshes, Web-based user interface implementations can use the Dual-MVC approach described in this paper to improve response time. An important issue that we are currently exploring relates to *development environments* for such implementations. Ideally, the development environment will use a single representation to store information about an application’s set of Models, Views, and Controllers in order that deployment decisions -- server-side *versus* client-side -- can be deferred as much as possible. A “canonical form”, in other words, coupled with emitter tools would remove the need to have separate implementations for the dual MVC representations, and would enable far more flexible deployment.

6. References

1. Glenn E. Krasner and Stephen T. Pope, *A cookbook for using the model view controller interface paradigm in Smalltalk-80*, Journal of Object Oriented Programming, 1(3):26-49, August/September 1988.

2. A. Goldberg, **Smalltalk 80: The Interactive Programming Environment,** Addison Wesley, 1984.

3. HTML Document Object Model `http://www.w3c.org/DOM/`

4. Danny Goodman, **Dynamic HTML, The Definitive Reference**, OReilly, 1998.

5. Standard ECMA-262, ECMAScript Language Specification `http://www.ecma.ch/stand/ecma-262.htm`

6. David Flanagan, **JavaScript, The Definitive Guide,** 3rd edition, OReilly, 1998

7. Java Servlet API `http://java.sun.com/products/servlet/index.html`

8. Enterprise JavaBean Specification Version 1.0, `http://java.sun.com/products/ejb/docs.html`

9. Java Server Pages, `http://java.sun.com/products/jsp/`

10. W3C XML Specification DTD, available at http://www.w3.org/XML/1998/06/xmlspec-report-19990205.htm

11. XSL Transformations (XSLT), Version 1.0, W3C Working Draft 13 August 1999, available from http://www.w3.org/1999/08/WD-xslt-19990813