RC24091 (W0610-156) October 27, 2006
Computer Science

IBM Research Report

WebRB: A Different Way to Write Web-Applications

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

WebRB: A Different Way To Write Web-Applications

Avraham Leff James T. Rayfield

Abstract

Relational Blocks (RBLOCKS) is a visual language that enables developers to assemble enterprise applications without adding imperative code. *WebRB* is an implementation of RBLOCKS for a web-application environment. Its visual editor runs in a standard browser, and its assembled applications execute, as well, in a standard browser. The *WebRB* server is responsible for storing applications and their data on behalf of clients, and instantiating them as they are needed in client browsers. The server instantiates an application by creating the web pages that run in the client browser and by providing the required inter-page navigation and associated data-flows.

Index Terms

relational blocks, web relational blocks, programming web applications, visual programming languages, declarative programming, relational algebra, relational model, software as a service

I. Introduction

Relational Blocks (RBLOCKS) [1] is a visual language that allows developers to assemble enterprise applications without adding any imperative code. *WebRB* is a client/server implementation of RBLOCKS for web-applications in which the *WebRB* server provides the following services to standard web browser clients:

- The server provides a visual editor that allows developers to assemble web-applications in a standard browser. Each web page is constructed by "dragging & dropping" prototype widget, algebra, and model blocks from a palette, and wiring them together. Developers use the editor to specify inter-page navigation and data flows.
- The server persistently stores a developer's *WebRB* page designs for later use.
- The server validates *WebRB* page designs and reports errors to the visual editor.
- The server instantiates the *WebRB* application as a web-application that executes in a standard browser.

Typically, applications that require much user interaction – such as program editors – reside on the client. The assumption is that having them reside on the server will result in unacceptable application latency. *WebRB* shows that a powerful visual editor *can* reside on the server and still provide a good user experience. We therefore prefer to use the "Software as a Service" model (SaaS [2]) so that *WebRB* is deployed, managed, and updated to servers rather than to end-user computers. Having our application reside on a server makes maintenance much simpler, since bug-fixes and improvements are installed only on the server, and are immediately available to all clients. Another advantage is that collaboration between multiple developers is facilitated since the server page-designs can be shared. Because the *WebRB* editor runs in a standard browser, it is especially suited to benefit from some of the advantages of the SaaS model. First, no installation is required, in contrast to "rich" editors such as those based on GEF [3] and Eclipse [4]. People prefer not to have to install yet-another-application which might break their computer, be incompatible with existing applications, or require patches. Second, browsers are so ubiquitous that it's

IBM T.J. Watson Research Center email: avraham@ibm.com
IBM T.J. Watson Research Center email: jtray@ibm.com

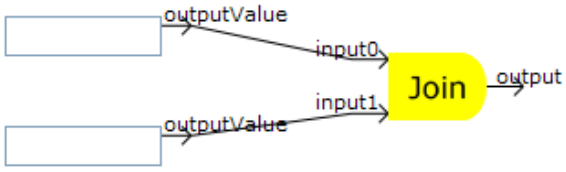

Fig. 1. Blocks and Wires

hard to find yourself using a computer without a browser anymore. Users can access the *WebRB* service on their "home" computer, or even when using somebody else's computer.

WebRB thus exhibits three interesting features. First, as an implementation of RBLOCKS, developer productivity can increase because of the language features discussed in [1]. Second, the visual editor's feature-set and responsiveness resemble "rich client" behavior (e.g., those developed using the *Graphical Editing Framework* [3]) — even though it executes in a standard browser. Third, *WebRB* applications are deployed as standard web-applications, that execute in a standard browser, without developers adding any client-side or server-side imperative code.

We discuss these features in this paper. After a summary of the RBLOCKS visual language, we discuss the visual editor in Section II, and the runtime environment in Section III. We offer some conclusions about *WebRB*'s use of web browsers in Section IV.

A. Relational Blocks Language

RBLOCKS is a visual language whose programs consist of blocks (of various types) that are interconnected to form a relational "circuit". The circuit is relational because the language uses only the relation data-type, and because data transformations are described using relational algebra. (See [5], which provides a good introduction to a type system based on the relational model.)

Figure 1 shows three blocks (two text widgets and a JOIN block) connected so that the JOIN block's output is a relation consisting of a two-attribute tuple. (We prefer to avoid the "persistent database" connotations associated with SQL terminology: e.g., we refer to *attributes*, rather than *columns*). Each of the text widgets have an *output* pin ("outputValue"); the JOIN block has two *input* pins ("input0" and "input1"), and one output pin ("output"). An input pin implies that the block can receive relation-valued input from other blocks. An output pin implies that the block can transmit relation-valued output to other blocks. Wires specify relational data-flow between two blocks by connecting input pins to output pins.

All "primitive" blocks fall into one of three categories: *model*, *widgets*, and *algebra*. These correspond, respectively, to the well-known Model/View/Controller paradigm. Model blocks are used to access and update the state of the application. Model-block updates occur when a "GUI event" occurs; for example, when a user clicks on a "submit" button. Broadly speaking, model blocks may be backed by either persistent or transient storage; this is an application design issue and the RBLOCKS model API is the same for both. A model block's output is the current state of the relation; its input is the desired next state of the relation.

In contrast, Algebra blocks in principle will update their outputs whenever their input value(s) change (although they are not necessarily implemented this way). Thus, a Join block's output is always the relational join of it's two inputs. Algebra blocks are used to declaratively specify an application's controller logic. RBLOCKS uses a formulation of the relational algebra based on *Relational A* ([5], Appendix A). Relational algebra is in many ways a perfect match for a relational model because it provides a declarative description of (1) what data should be extracted from the model and (2) how that data should be manipulated [6]. Also, relational algebra operations are reasonably simple in isolation, small in number, and can

be easily composed to form more complex operations.

Widget-block outputs are always based on the current contents of the widget; e.g. a text-input widget always provides its current contents on its output pin. However, non-interactive widgets (e.g. text, tables) are only updated when a GUI event occurs. Widget blocks are rendered at runtime as an HTML GUI widget, but also implement the relational API. Program-writeable widgets have relation inputs (e.g., a single-tuple input to a label block with *text* and *font* attributes). Program-readable widgets have relation outputs (e.g., a single-tuple output from a slider block containing a "value" attribute). More complicated widgets such as tables and lists are multi-tuple relations.

Because all blocks have the same visual representation and semantics, RBLOCKS does not place any constraints on which blocks can talk to other blocks. For example, an algebra block's inputs can include the current state of model blocks or the current values of the readable widgets. Similarly, an algebra block's output can be wired to a model block or to a writeable widget block.

A developer assembles instances of RBLOCKS primitive blocks to produce *pages* with higher-level function. Such pages can be embedded in another page in a hierarchical approach to application construction. Pages have exactly the same relational API and visual representation as primitive blocks.

II. WEBRB EDITOR

The *WebRB* editor runs in a standard Mozilla Firefox browser and is used to visually construct RBLOCKS programs. Because RBLOCKS is a visual language, there is a good fit between the editor and the programs that it constructs. Readers who wish to experience *WebRB* "live" can currently access it as a free alphaWorks service [7]. Figure 2 shows that the *WebRB* editor is comprised of several frames. We focus here on the "palette" (which contains the set of available primitive blocks); the "page editor" (used to edit a single web-page); and the "property editor" (used to customize a primitive block instance).

As discussed in Section I, the *WebRB* editor is a "service" in the sense that no code is down-loaded and installed on a developer's computer. Instead, a developer's *WebRB* page designs are stored on a server, and loaded into the developer's browser as needed. Once we decided that the *WebRB* editor should be a "service", we faced the challenge of making it responsive and "feature-rich" despite the fact that no software is installed on the developer's computer. For example, developers can "drag and drop" blocks from the palette to precise positions on the page editor. At runtime, corresponding GUI widgets are rendered at precisely those positions on the web-page. This is much easier than using imperative code to draw and position the widgets. A wire is drawn by "left-clicking" near an given pin, dragging the mouse to the other pin while holding the left mouse-button down, and then releasing the mouse button. Wires can be deleted by clicking (to select the wire) and hitting the "delete" key; blocks (and all of their wires) are deleted by clicking (to select the block) and hitting the "delete" key. Also, whenever a developer selects a block, a block-specific editor is loaded into the property editor frame, and populated with that block's state so it can be customized.

We solved this challenge by structuring the life-cycle of a page-design so that the browser communicates infrequently with the server. Communication is required for page life-cycle operations (create, load, save, and delete a page design), page validation, and for some property-editor functions (such as loading page- and table-selector widgets). Otherwise, we use "AJAX" [8] technologies such as DHTML, JavaScript, and XmlHttpRequest to enable the editor to manipulate a page-design entirely within the browser. Every primitive block is implemented as a JavaScript class in which the block itself is drawn with HTML. Wires are drawn using SVG [9], since HTML does not have enough functionality to draw diagonal lines. Wires are made easily "selectable" by using SVG to trace a transparent outline that is much thicker than the visible outline itself. Once a block is selected, we use a combination of CSS absolute positioning and mouse events to implement "dragging". A block's JavaScript code uses the DOM events API to track

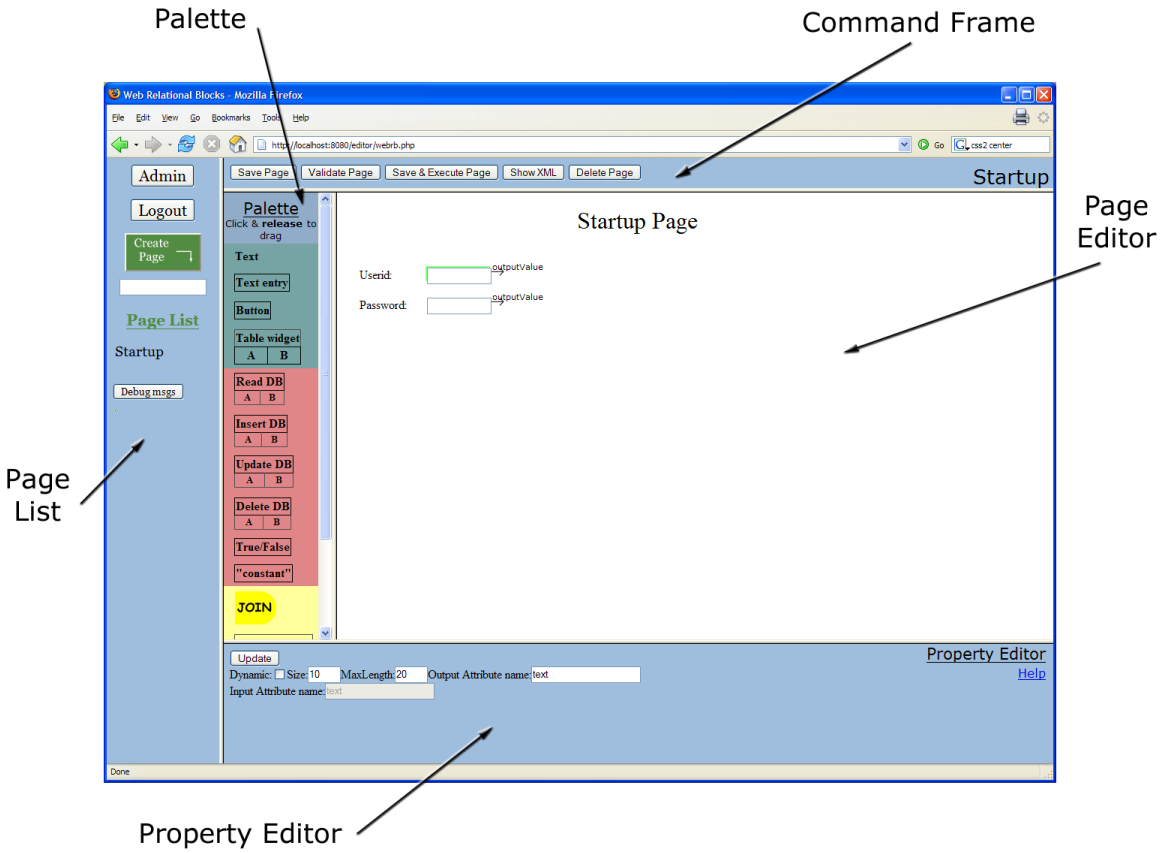

Fig. 2. The *WebRB* Editor

events such as “mouseup” and “mousemove” to determine where a block has been dragged to. Methods such as Firefox’s `getComputedStyle` return the computed CSS for rendered HTML. We can thus calculate a block’s height and width, and determine where to draw pins and selection outlines.

Page-design communication between the *WebRB* editor and the server (e.g., a “save” or “load” operation), is implemented with a framework based on XML DOM trees [10]. Each block type has a `getState()` method that returns a DOM tree that contains all state (e.g., screen position and property settings) needed to subsequently hydrate the block (*via* its JavaScript constructor method). Wires also have a `getState()` method that returns a DOM tree indicating the endpoints (block and pin). We use the Mozilla browser’s XMLSerializer API to serialize an entire page-design as an XML document that contains the state of all the page’s DOM trees. Finally, we invoke the `XmlHttpRequest` API to transmit the XML document to the server, where it is persistently stored in a database table. Loading a page-design from the server into the *WebRB* editor is the mirror image of this process.

III. WebRB Runtime

The *WebRB* runtime is responsible for deploying the page-designs discussed in Section II as web-applications that execute in a standard browser. We explain in this Section how our runtime can do this without developers adding any code on either the client or the server. We also explain how individual page-designs are assembled as a web-application.

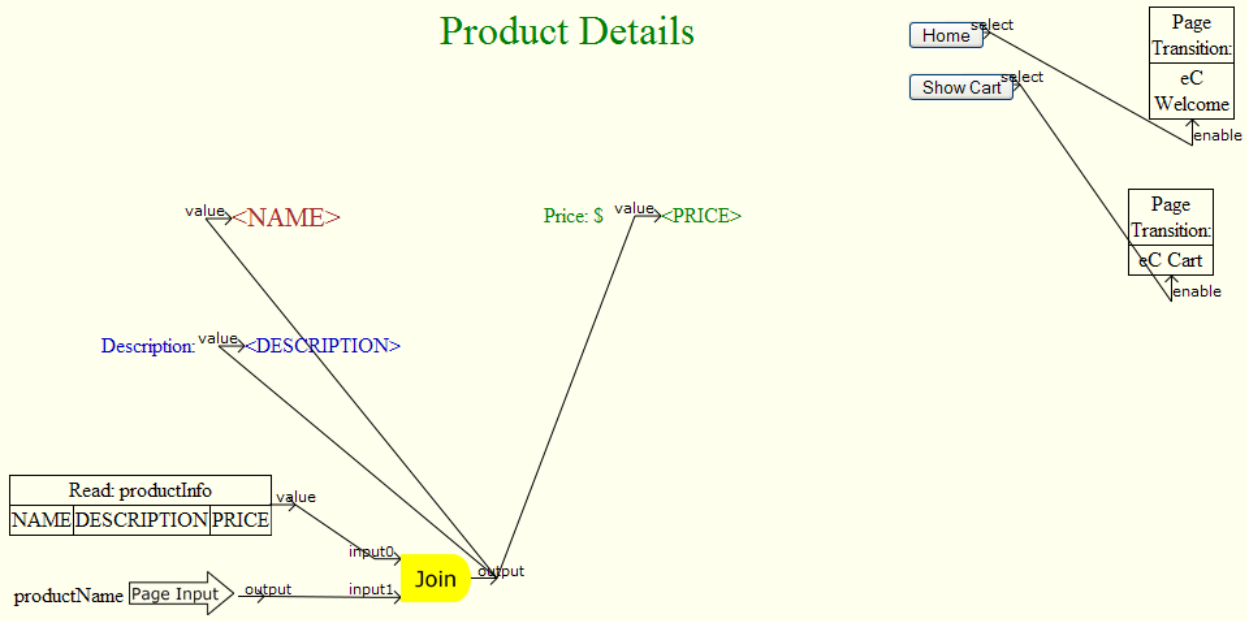

Fig. 3. Example: an eCommerce "Details" WebRB Page Design

The server does not have to directly manipulate the RBLOCKS visual language because page-designs are stored in their serialized XML representation. The XML representation serves as a canonical form of a given block: it's instantiated as an PHP object on the server, and as a JavaScript object in the browser. The *WebRB* runtime renders a *WebRB* page by creating server-side versions of each RBLOCKS "block". Model blocks are implemented as a wrapping of connections to database tables; algebra blocks are implemented to provide the required algebra function; and widget blocks are implemented so that their `getHTML()` method produces the corresponding HTML that can be rendered in a web-browser. Block inter-connection is implemented by driving the specified data-flow from output pins to their input pins, and so on, using a recursive process.

The initial page of an application is displayed when the client's browser issues an HTTP GET request: the server's *WebRB* runtime calls `getHTML()` to create the initial page, and returns the web-page to the browser. During initial-page evaluation, widgets with output pins will return empty strings. Once launched, the server "forgets" about the page until the user interacts with the web-page, causing an HTTP POST of the page and its data to the server. The *WebRB* runtime uses the POST data to instantiate the corresponding server-side version of that web-page, supplies the user-supplied data (e.g., text-field input) to the widget blocks, and evaluates the resulting relational "circuit". The runtime must then determine which web-page should be instantiated next. Consider, for example, Figure 3 which is a screen-shot of a "Details" page-design from an eCommerce web-application. (This is taken from the eCommerce example that can be imported by users of the *WebRB* alphaWorks service [7].) This page gives details about a specific product (NAME, DESCRIPTION, and PRICE). It also shows how developers use "page-transition" blocks (upper-right) to specify what web-page should be instantiated after a user interacts with the "product details" page. The "Home" and "Show Cart" navigation buttons are wired to the corresponding eCommerce pages: only the button that the user clicked will produce the TRUE relation output – thus enabling the page transition – the other button will produce a FALSE output. In general, exactly one page-transition block must be enabled for any given evaluation of a page-design. Anything else signifies a logical error in the design, and is detected and reported by the runtime.

One final step is done by the *WebRB* runtime before instantiating the application's next web-page. Note the use of a "page input" block in the lower-left of Figure 3. The developer has specified that a "product name" input must be provided before this page can be instantiated. (The value of this input is JOIN'd with the product information database table to create the HTML labels that display the product details.) Any page-design that specifies a transition to this "Details" page (via a "page-transition" block) will have to also include a wire to the corresponding input pin of the "Details" page. Figure 4 shows how this works in practice. The Figure shows screen-shots of two instantiated web-pages. The first is a "Search Results" page that lets a user type a product query string. In this screen-shot, the user previously queried for "staplers", and the eCommerce application returned a populated web-page that allows the user to drill-down to get more details about the product. The second screen-shot shows the web-page that the *WebRB* runtime returns when the user clicks on the "details" button. The "details" button in the "Search Results" page-design is wired to the "enable" pin of the "Details" page-transition block. In addition, the *table widget* block feeds the value of that row's NAME attribute to the "product name" input pin of the "Details" page-transition block. Therefore, after evaluating the HTTP POST of the "Search Results" page (and determining that the "Details" page is to be instantiated next), the *WebRB* runtime flows the NAME value into the "Details" page as it is instantiated. The result is shown in the second screen-shot of Figure 4.

IV. Conclusion

WebRB is a different way to write web-applications. Rather than using a mix of imperative code (e.g., PHP and JavaScript) on both the client and server, a single visual language is used which requires no additional code from the developer. The *WebRB* visual editor runs in a standard browser, and *WebRB* applications execute as web-applications that run in a standard browser.

We note in closing, that although browsers provide a more restricted user-interaction model than so-called "rich client" editors, this is in some ways a benefit. Developers already understand how to interact with a browser, and therefore already (mostly) know how to use *WebRB*. Also, because *WebRB* applications are themselves browser-based, we can immediately show the developer the results of any changes. There is a one-to-one correspondence between a design-page and a web-page: when a developers edits a *WebRB* page, she can immediately see the effects of that change in the visual editor, and also execute the page immediately if desired.

We focused in this paper on the way that *WebRB* is implemented as a "software service" in which the server loads the visual editor into a user's web browser, and page designs and application state are stored on the server. The software service model has well-known advantages, but required us to make the editor responsive and feature-rich while executing in a standard browser. We also explained how applications written in the *WebRB* visual language are translated in a straight-forward way to execute as a web-applications. Readers can experience *WebRB* as a live service [7].

References

[1] Avraham Leff and James T. Rayfield. Relational blocks: Declarative visual assembly of enterprise applications. http://domino.research.ibm.com/library/cyberdig.nsf/, 2006. RC24014.

[2] David Greschler and Tim Mangan. Networking Lessons in Delivering 'Software as a Service' - Part I. *International Journal of Network Management*, 12(5):317–321, September/October 2002.

[3] Graphical Editing Framework. http://www.eclipse.org/gef, 2006.

[4] Eclipse Project. http://www.eclipse.org/eclipse, 2006.

[5] C.J. Date and H. Darwen. *Databases, Types and the Relational Model (3rd Edition)*. Addison-Wesley, Boston, MA, 2006.

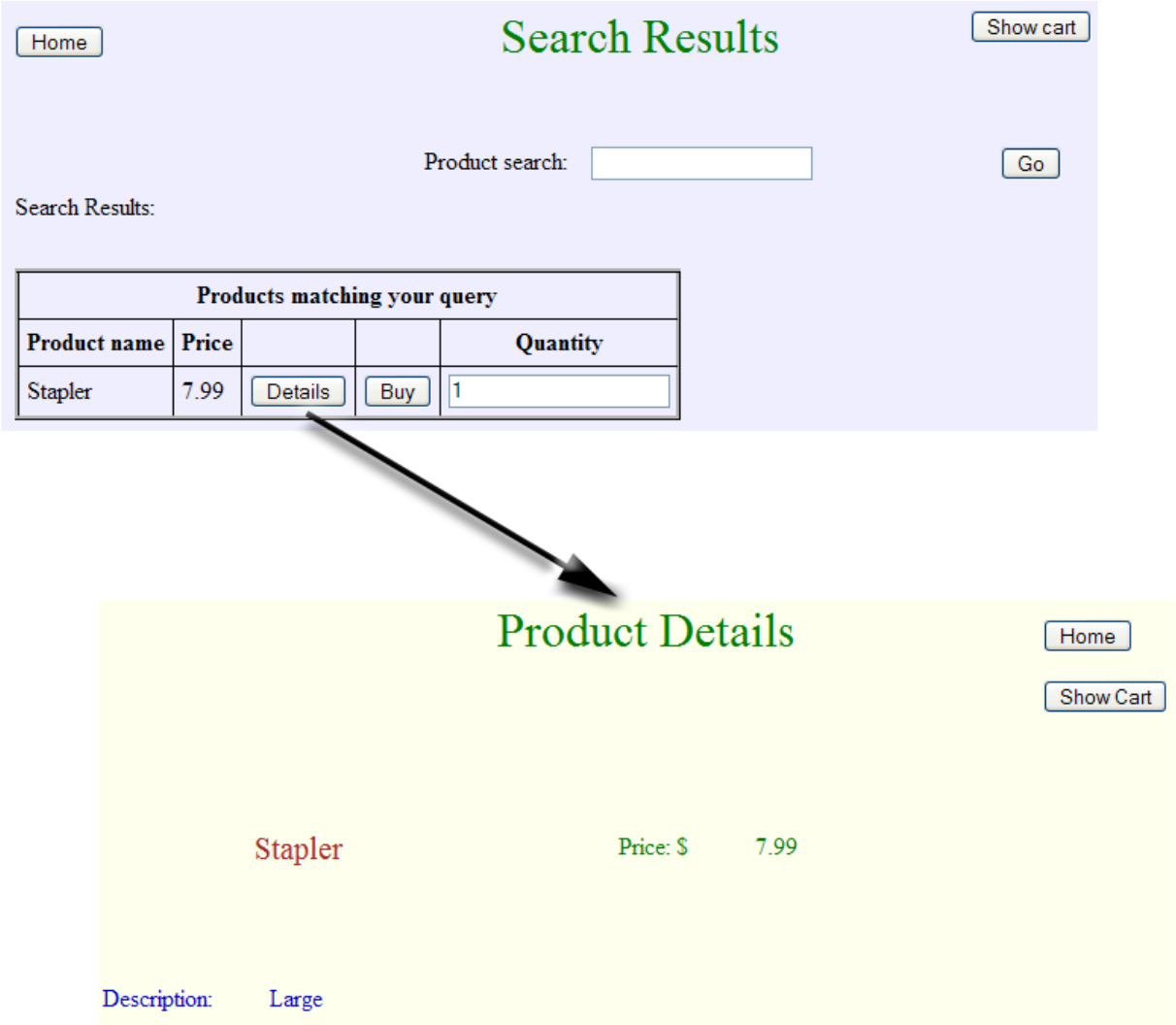

Fig. 4. WebRB Pages Executing In the Browser

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
[7] alphaWorks Services: Web Relational Blocks. http://services.alphaworks.ibm.com/webrb/, 2006.
[8] Justin Gehtland, Dion Almaer, and Ben Galbraith. *Pragmatic Ajax: A Web 2.0 Primer*. Pragmatic Bookshelf, 2006.
[9] Scalable Vector Graphics (SVG) 1.1 Specification. www.w3.org/TR/SVG/, 2006.
[10] Document object model (DOM). http://www.w3.org/TR/DOM-Level-2-Core/, 2006.