Managing State for Ajax-Driven Web Components

John Ousterhout and Eric Stratmann
Department of Computer Science
Stanford University
`{ouster,estrat}@cs.stanford.edu`

Abstract
Ajax-driven Web applications require state to be maintained across a series of server requests related to a single Web page. This conflicts with the stateless approach used in most Web servers and makes it difficult to create modular components that use Ajax. We implemented and evaluated two approaches to managing component state: one, called *reminders*, stores the state on the browser, and another, called *page properties*, stores the state on the server. Both of these approaches enable modular Ajax-driven components but they both introduce overhead for managing the state; in addition the reminder approach creates security issues and the page property approach introduces storage reclamation problems. Because of the subtlety and severity of the security issues with the reminder approach, we argue that it is better to store Ajax state on the server.

1 Introduction

Ajax (shorthand for "Asynchronous Javascript And XML") is a mechanism that allows Javascript code running in a Web browser to communicate with a Web server without replacing the page that is currently displayed [6]. Ajax first became available in 1999 when Microsoft introduced the XMLHTTP object in Internet Explorer version 5, and it is now supported by all Web browsers. In recent years more and more Web applications have begun using Ajax because it permits incremental and fine-grained updates to Web pages, resulting in a more interactive user experience. Notable examples of Ajax are Google Maps and the auto-completion menus that appear in many search engines.

Unfortunately, Ajax requests conflict with the stateless approach to application development that is normally used in Web servers. In order to handle an Ajax request, the server often needs access to state information that was available when the original page was rendered but discarded upon completion of that request. Current applications and frameworks use *ad hoc* approaches to reconstruct the state during Ajax requests, resulting in code that is neither modular nor scalable.

We set out to devise a systematic approach for managing Ajax state, hoping to enable simpler and more modular code for Ajax-driven applications. We implemented and evaluated two alternative mechanisms. The first approach, which we call *reminders*, stores the state information on the browser with the page and returns the information to the server in subsequent Ajax requests. The second approach, which we call *page properties*, stores the state information on the server as part of the session. Both of these approaches allow the creation of reusable components that encapsulate their Ajax interactions, so that Web pages can use the components without being aware of or participating in the Ajax interactions. Although each approach has disadvantages, we believe that the page property mechanism is the better of the two because it scales better and has fewer security issues.

The rest of this paper is organized as follows. Section 2 introduces the Ajax mechanism and its benefits. Section 3 describes our modularity goal and presents an example component that is used in the rest of the paper. Section 4 describes the problems with managing Ajax state, and how they impact the structure of applications. Sections 5 and 6 introduce the reminder and page property mechanisms, and Section 7 compares them. Section 8 presents examples of Ajax-driven components using these mechanisms, Section 9 describes related work, and Section 10 concludes.

2 Ajax Background

Ajax allows a Web page to communicate with its originating Web server as shown in Figure 1. An Ajax-driven page is initially rendered using the normal mechanism where the browser issues an HTTP request

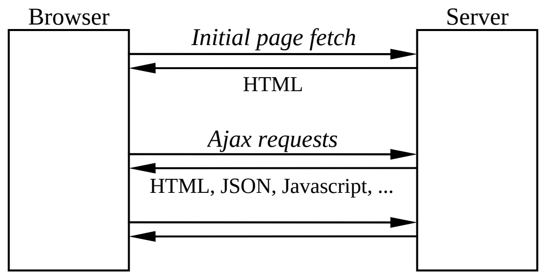


Figure 1. After the initial rendering of a Web page, Ajax requests can be issued to retrieve additional data from the server, which can be used to make incremental modifications to the page displayed in the browser.

to the server and the server responds with HTML for the page contents. Once the page has been loaded, Javascript event handlers running in that page can issue Ajax requests. Each Ajax request generates another HTTP request back to the server that rendered the original page. The response to the Ajax request is passed to another Javascript event handler, which can use the information however it pleases.

Ajax responses can contain information in any format, but in practice the response payload usually consists of one of three things:

- An HTML snippet, which the Javascript event handler assigns to the `innerHTML` property of a page element in order to replace its contents.
- Structured data in a format such as JSON [4], which the Javascript event handler uses to update the page by manipulating the DOM.
- Javascript code, which is evaluated in the browser (this form is general enough to emulate either of the other forms, since the Javascript can include literals containing HTML or any other kind of data).

The power of Ajax stems from the fact that the response is passed to a Javascript event handler rather than replacing the entire page. This allows Web pages to be updated in an incremental and fine-grained fashion using new information from the server, resulting in a more interactive user experience. One popular example is Google Maps, which allows a map to be dragged with the mouse. As the map is dragged, Ajax requests fetch additional images that extend the map's coverage, creating the illusion of a map that extends infinitely in all directions. Another example is an auto-completion menu that appears underneath the text entry for a search engine. As the user types a search term the auto-completion menu updates itself using Ajax requests to display popular completions of the search term the user is typing.

Although the term “Ajax” typically refers to a specific mechanism based on Javascript `XMLHttpRequest` objects, there are several other ways to achieve the same effect in modern browsers. One alternative is to create a new `<script>` element in the document, which causes Javascript to be fetched and executed. Another approach is to post a form, using the `target` attribute to direct the results to an invisible frame; the results can contain Javascript code that updates the main page. In this paper we will use the term “Ajax” broadly to refer to any mechanism that allows an existing page to interact with a server and update itself incrementally.

3 Encapsulation Goal

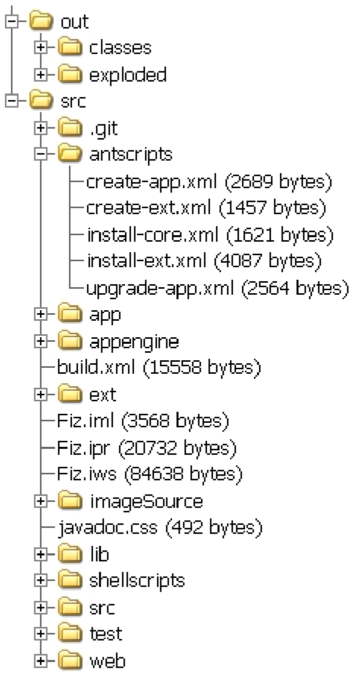
The basic Ajax mechanism is quite simple and flexible, but it is difficult to incorporate cleanly into Web application frameworks. Our work with Ajax occurred in the context of Fiz [10], an experimental server-side Web application framework under development at Stanford University. The goal for Fiz is to raise the level of programming for Web applications by encouraging a component-based approach, where developers create applications by assembling pre-existing components. Each component manages a portion of the Web page, such as:

- A form field that displays a calendar from which a user can select a particular date.
- A general-purpose table that can be sorted based on the values of one or more column(s).
- A catalog display tailored to the interests of the current user.
- A shopping cart.

Ideally, a component-based approach should simplify development by encouraging reusability and by hiding inside the components many of the complexities that developers must manage explicitly today, such as the quirks of HTML, Ajax requests, and a variety of security issues.

For a component framework to succeed it must have several properties, one of the most important of which is *encapsulation*: it must be possible to use a component without understanding the details of its implementation, and it must be possible to modify a component without modifying all of the applications that use the component. For example, consider a large Web site with complex pages, such as Amazon. Teams of developers manage different components that are used on various Web pages, such as sponsored advertisements, user-directed catalog listings, and search bars. Each team should be able to modify and improve its own components (e.g., by adding Ajax interactions) without requiring changes in the pages that use those components. Thus, one of our goals for Fiz is that a component should be able to use Ajax requests in its implementation without those requests being visible outside the component.

In this paper we will use the TreeSection component from Fiz to illustrate the problems with Ajax components and the potential solutions. TreeSection is a class that provides a general-purpose mechanism for browsing hierarchical data as shown in Figure 2(a). It displays hierarchically-organized data using icons and indentation; users can click on icons to expand or hide subtrees. In order to support the display of large structures, a TreeSection does not download the entire tree



```
out
  classes
  exploded
src
  .git
  antscripts
    create-app.xml (2689 bytes)
    create-ext.xml (1457 bytes)
    install-core.xml (1621 bytes)
    install-ext.xml (4087 bytes)
    upgrade-app.xml (2564 bytes)
  app
  appengine
  build.xml (15558 bytes)
  ext
  Fiz.iml (3568 bytes)
  Fiz.ipr (20732 bytes)
  Fiz.iws (84638 bytes)
  imageSource
  javadoc.css (492 bytes)
  lib
  shellscripts
  src
  test
  web
```

(a)

```
new TreeSection("FS.filesInDir",
                "code/Fiz");
```

(b)

Figure 2. The Fiz TreeSection component displays hierarchical information using nested indentation and allows the structure to be browsed by clicking on + and - icons: (a) the appearance of a TreeSection that displays the contents of a directory; (b) Java code to construct the TreeSection as part of a Web page.

to the browser. Instead, it initially displays only the top level of the tree; Ajax requests are used to fill in the contents of subtrees incrementally when they are expanded.

The TreeSection class automatically handles a variety of issues, such as the tree layout, Javascript event handlers to allow interactive expansion and collapsing, and the Ajax-based mechanism for filling in the tree structure on demand. It also provides options for customizing the display with different icons, node formats, and graphical effects.

In order to maximize its range of use, the TreeSection does not manage the data that it displays. Instead, whenever it needs information about the contents of the tree it invokes an external *data source*. The data source is passed the name of a node and returns information about the children of the node. When a TreeSection is constructed it is provided with the name of the data source method (`FS.filesInDir` in Figure 2(b)), along with the name of the root node of the tree (`code/Fiz` in Figure 2(b)). In the example of Figure 2 the data source reads information from the file system, but different data sources can be used to browse different structures. The TreeSection invokes the data source once to display the top level of the tree during the generation of the original Web page, then again during Ajax requests to expand nodes.

The challenge we will address in the rest of this paper is how to manage the state of components such as TreeSection in a way that is convenient for developers and preserves the encapsulation property.

4 The Ajax State Problem

Using Ajax today tends to result in complex, non-modular application structures. We will illustrate this problem for servers based on the model-view-controller (MVC) pattern [11,12]. MVC is becoming increasingly popular because it provides a clean decomposition of application functionality and is supported in almost all Web development frameworks. Similar problems with Ajax state arise for Web servers not based on MVC.

When an HTTP request arrives at a Web server based on MVC it is dispatched by the application framework to a particular method in a particular controller class, based on the URL in the request, as shown in Figure 3(a). The controller method collects data for the page from model classes and then invokes one or more view classes to render the page's HTML. If the page subsequently makes an Ajax request, the request is dispatched to another method in the same controller. The Ajax service method invokes model classes to collect data and then view classes to format a response.

Unfortunately, with this approach the controller for a page must be involved in every Ajax request emanating from the page, which breaks the application's modularity. If one of the views used in a page introduces new Ajax requests, every controller using that view must be modified to mediate those requests. As a result, it is not possible to create reusable components that encapsulate Ajax, and Ajax-driven applications tend to have complex and brittle structures.

The first step in solving this problem is to bypass the controller when handling an Ajax request and dispatch directly to the class that implements the component, as shown in Figure 3(b). Virtually all frameworks have dispatchers that can be customized to implement this behavior.

Dispatching directly to the component creates two additional issues. First, the controller is no longer present to collect data for the component, so the component must

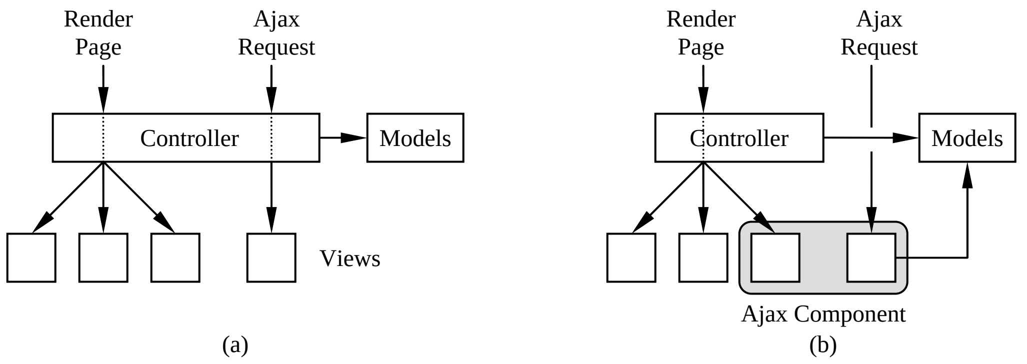


Figure 3. (a) The structure of a typical Web application today, where all HTTP requests for a page (including Ajax requests) are mediated by the page's controller class; (b) A component-oriented implementation in which all aspects of the Ajax-driven element, including both its original rendering and subsequent Ajax requests, are encapsulated in a reusable component. In (b) Ajax requests are dispatched directly to the component, bypassing the controller, and the component fetches its own data from model classes

invoke models itself to gather any data it needs. This is not difficult for the component to do, but it goes against the traditional structure for Web applications, where views receive all of their data from controllers.

The second problem created by direct dispatching relates to the state for the Ajax request. In order to handle the Ajax request, the component needs access to configuration information that was generated during the original rendering of the page. In the TreeSection example of Figure 2, the component needs the name of the data source method in order to fetch the contents of the node being expanded. This information was available at the time the component generated HTML for the original page, but the stateless nature of most Web servers causes data like this to be discarded at the end of each HTTP request; thus Ajax requests begin processing with a clean slate.

One of the advantages of dispatching Ajax requests through the controller is that it can regenerate state such as the name of the data source method. This information is known to the controller, whose code is page-specific, but not to the component, which must support many different pages with different data sources. If Ajax requests are dispatched directly to the component without passing through the controller, then there must be some other mechanism to provide the required state to the component.

Solutions to the state management problems fall into two classes: those that store state on the browser and those that store state on the server. We implemented one solution from each class in Fiz and compared them. The next section describes our browser-based approach, which we call *reminders*; the server-based approach, which we call *page properties*, is described in the following section.

5 Reminders

Our first attempt at managing Ajax state was to store the state in the browser so the server can remain stateless. When an Ajax-driven component renders its portion of the initial page it can specify information called *reminders* that it will need later when processing Ajax requests. This information is transmitted to the browser along with the initial page and stored in the browser using Javascript objects (see Figure 4). Later, when Ajax requests are issued, relevant reminders are automatically included with each Ajax request. The reminders are unpacked on the server and made available to the Ajax handler. An Ajax handler can create additional reminders and/or modify existing reminders; this

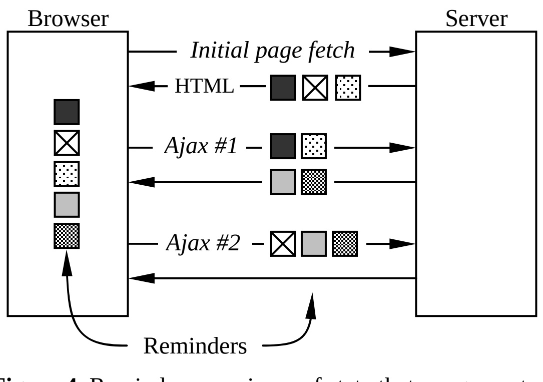
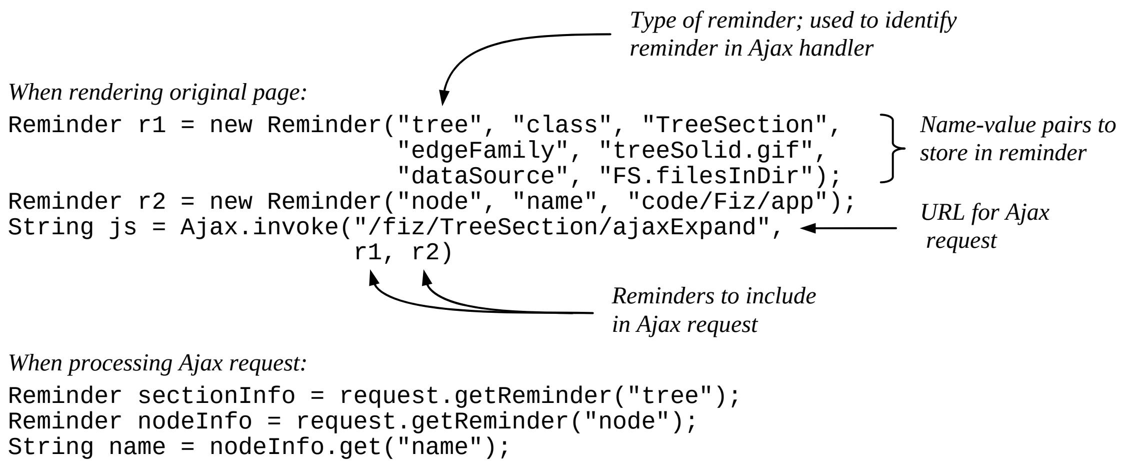


Figure 4. Reminders are pieces of state that are generated by the server, stored in the browser, and returned in later Ajax requests. Additional reminders can be generated while processing Ajax requests.


Type of reminder; used to identify reminder in Ajax handler
Name-value pairs to store in reminder
URL for Ajax request
Reminders to include in Ajax request

When rendering original page:

```
Reminder r1 = new Reminder("tree", "class", "TreeSection",
                           "edgeFamily", "treeSolid.gif",
                           "dataSource", "FS.filesInDir");
Reminder r2 = new Reminder("node", "name", "code/Fiz/app");
String js = Ajax.invoke("/fiz/TreeSection/ajaxExpand",
                        r1, r2)
```

When processing Ajax request:

```
Reminder sectionInfo = request.getReminder("tree");
Reminder nodeInfo = request.getReminder("node");
String name = nodeInfo.get("name");
```

Figure 5. Examples of the APIs for creating and using reminders in the Fiz TreeSection.

information is returned to the browser along with the Ajax response, and will be made available in future Ajax requests. Reminders are similar to the View State mechanism provided by Microsoft's ASP.NET framework; see Section 9 for a comparison.

Figure 5 illustrates the APIs provided by Fiz for managing reminders. Each Reminder object consists of a type and a collection of name-value pairs. The type is used later by Ajax request handlers to select individual reminders among several that may be included with each request. For example, the TreeSection creates one reminder of type `tree` containing overall information about the tree, such as its data source and information needed to format HTML for the tree. It also creates one reminder of type `node` for each expandable node in the tree, which contains information about that particular node. When the user clicks on a node to expand it, the resulting Ajax request includes two reminders: the overall `tree` reminder for the tree, plus the `node` reminder for the particular node that was clicked.

Fiz automatically serializes Reminder objects as Javascript strings and transmits them to the browser. Fiz also provides a helper method `Ajax.invoke`, for use in generating Ajax requests. `Ajax.invoke` will create a Javascript statement that invokes an Ajax request for a given URL and includes the data for one or more reminders. The result of `Ajax.invoke` can be incorporated into the page's HTML; for example, TreeSection calls `Ajax.invoke` once for each expandable node and uses the result as the value of an `onclick` attribute for the HTML element displaying the + icon.

When an Ajax request arrives at the Web server, Fiz dispatches it directly to a method in the TreeSection class. Fiz automatically deserializes any reminders attached to the incoming request and makes them available to the request handler via the `getReminder` method. In the TreeSection example the request handler collects information about the node being expanded (by calling the data source for the tree), generates HTML to represent the node's contents, and returns the HTML to the browser, where it is added to the existing page. If the node's contents include expandable sub-nodes, an additional `node` reminder is created for each of those sub-nodes and included with the Ajax response.

5.1 Evaluation of reminders

The reminder mechanism makes it possible to encapsulate Ajax interactions within components, and it does so without storing any additional information on the server. Ajax interactions are not affected by server crashes and reboots, since their state is in the browser.

Reminders have two disadvantages. First, they introduce additional overhead for transmitting reminder data to the browser and returning it back to the server. In order to minimize this overhead we chose a granular approach with multiple reminders per page: each Ajax request includes only the reminders needed for that request. In our experience implementing Ajax components we have not yet needed reminders with more than a few dozen bytes of data, so the overhead has not been a problem.

The second disadvantage of reminders is that they introduce security issues. The data stored in reminders represents internal state of the Web server and thus may need to be protected from hostile clients. For example, the reminders for the TreeSection include the name of the data source method and the name of the directory represented by each node. If a client modifies reminders it could potentially invoke any method in the server and/or view the contents of any directory.

Fiz uses message authentication codes (MACs) to ensure the integrity of reminders. Each reminder includes a SHA-256 MAC computed from the contents of the reminder using a secret key. There is one secret key for each session, which is stored in the session and used for all reminders associated with that session. When the reminder is returned in an Ajax request the MAC is verified to ensure that the reminder has not been modified by the client.

MACs prevent clients from modifying reminders, but they don't prevent clients from reading the contents of reminders. This could expose the server to a variety of attacks, depending on the content of reminders. For example, if passwords or secret keys were stored in reminders then hostile clients could extract them. It is unlikely that an application would need to include such information in reminders, but even information that is not obviously sensitive (such as the name of the data source method for the TreeSection) exposes the internal structure of the server, which could enable attackers to identify other security vulnerabilities. Unfortunately, it is difficult to predict the consequences of exposing internal server information. In order to guarantee the privacy of reminders they must be encrypted before computing the MAC. Fiz does not currently perform this encryption.

The granular nature of reminders also compromises security by enabling mix-and-match replay attacks. For example, in the TreeSection each Ajax request includes two separate reminders. The first reminder contains overall information about the tree, such as the name of the data source method. The second reminder contains information about a particular node being expanded, including the pathname for the node's directory. A hostile client could synthesize AjaxRequests using the tree reminder for one tree and the node reminder for another; this might allow the client to access information in ways that the server would not normally allow.

One solution to the replay problem is to combine all of the reminders for each page into a single structure as is done by View State in ASP.NET. This would prevent mix-and-match attacks but would increase the mechanism's overhead since all of the reminders for the page would need to be included in every request. Another approach is to limit each Ajax request to a single reminder. Each component would need to aggregate all of the information it needs into one reminder; for example, the TreeSection would duplicate the information about the tree in each node reminder. This approach would make it difficult to manage mutable state: if, for example, some overall information about the tree were modified during an Ajax request then every node reminder would need to be updated. Yet another approach is to use unique identifiers to link related reminders. For example, the tree reminder for each TreeSection might contain a unique identifier, and the server might require that each node reminder contains the same unique identifier as its tree reminder. This would prevent a node reminder from being used with a different tree reminder, but it adds to the complexity of the mechanism.

As we gained more experience with reminders we became concerned that it would be difficult to use them in a safe and efficient fashion, and that these problems will increase as Ajax usage becomes more pervasive and sophisticated. If the framework handles all of the security issues automatically it will require a heavyweight approach such as aggregating all state into a single reminder that is both encrypted and MAC-protected. However, this would probably not provide acceptable performance for complex pages with many Ajax-driven components. On the other hand, if developers are given more granular control over the mechanism they could probably achieve better performance, but at a high risk for security vulnerabilities. Even highly skilled developers are unlikely to recognize all of the potential loopholes, particularly when working under pressure to bring new features to market. For example, when asked to create an alternative implementation of TreeSection in Ruby on Rails for comparison with Fiz, a Stanford undergraduate with experience developing Web applications did not recognize that the node names need to be protected from tampering. The prevalence of SQL injection attacks [1] also indicates how difficult it is for Web developers to recognize the security implications of their actions.

6 Page Properties

Because of the problems with the reminder mechanism we decided to implement a different approach where component state is kept in the server instead of the browser. This eliminated the security and overhead issues with reminders but introduced a different problem related to garbage collection.

The server-based approach is called *page properties*. A page property consists of a name-value pair that is accessible throughout the lifetime of a particular Web

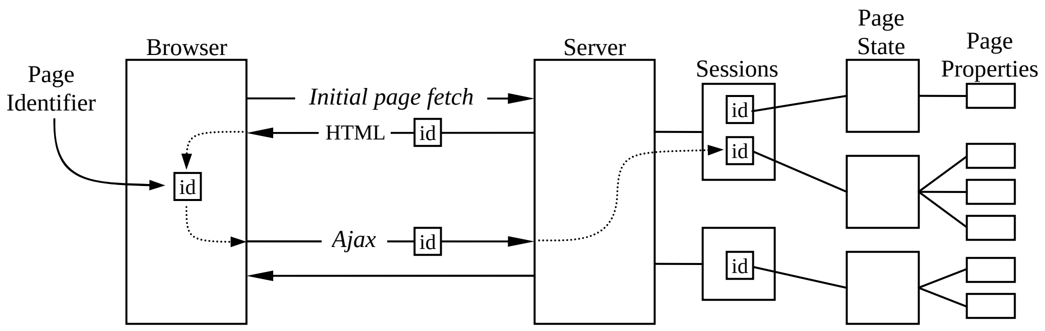


Figure 6. Fiz stores page properties on the server as part of the session. Each Web page is assigned a unique identifier, which is included in the page and returned to the server as part of each Ajax request; this allows the server to locate the properties for the page.

page. The name must be unique within the page and the value may be any serializable Java object. Page properties may be created, examined, and updated at any time using a simple API consisting of `getPageProperty` and `setPageProperty` methods. For example, the TreeSection creates a page property for each tree when it renders the top level of the tree during initial page display. The page property contains overall information about the tree, such as the name of the data source method, plus information about each node that has been rendered in the tree.

When an Ajax request arrives to expand a TreeSection node, it is dispatched directly to the TreeSection class just as in the reminder approach. The Ajax request includes an identifier for a particular tree instance (in case there are several trees in a single page) and an identifier for the node that is being expanded. The Ajax handler in TreeSection retrieves the page property for the tree instance, looks up the node identifier in the page property object, and uses that information to retrieve information about the children of the expanded node; this information is used to generate HTML to return to the browser, and also to augment the page property object with information about children of the expanded node.

The names of page properties are only unique within a page, so Fiz associates a unique *page identifier* with each distinct Web page and uses it to separate the page properties for different pages. A page identifier is assigned during the initial rendering of each page and is stored in the page using a Javascript variable (see Figure 6). Subsequent Ajax requests and form posts coming from that page automatically include the page identifier as an argument. Operations on page properties apply to the properties associated with the current page.

Fiz stores page properties using the session mechanism: all of the properties for each page are collected into a PageState object, and each session can contain multiple PageState objects, indexed by their page identifiers (see Figure 6). Storing page properties in the session ensures that they are preserved across the various requests associated with a page, even though the individual requests are implemented in a stateless fashion. Page properties have the same level of durability as other session information.

6.1 Evaluation of page properties

Page properties avoid the issues that concerned us with reminders: the only information sent to the browser is the page identifier, so page properties reduce the overhead of transmitting data back and forth across the network. Page properties also avoid the security issues associated with reminders, since state information never leaves the server. The use of sessions to store page properties ensures isolation between different users and sessions.

However, page properties introduce new issues of their own. First, in order for page properties to survive server crashes they must be written to stable storage after each request along with the rest of the session data. If page properties contain large amounts of information then they could still result in substantial overhead (e.g. for a TreeSection displaying hundreds of expandable nodes there could be several kilobytes of page properties). However, the overhead for saving page properties on the server is likely to be less than the overhead for transmitting reminders back and forth over the Internet to browsers.

Fiz currently stores page properties using the standard session facilities provided by the underlying Java servlets framework. However, it may ultimately be better to implement a separate storage mechanism for page properties that is optimized for their access patterns. For example, many session implementations read and write the entire session monolithically; however, a given request will only use the page properties for its particular page, so it may be inefficient to read and write all of the properties for other pages at the same time. In addition, the standard session mechanisms for reflecting session data across a server pool may not be ideal for page properties.

A second, and more significant, problem concerns the garbage collection of page properties: when is it safe to delete old page properties? Unfortunately the lifetime of a Web page is not well-defined: the server is not notified when the user switches to a different page; even if it were notified, the user can return to an old page at any time by clicking the "Back" browser button. There is no limit on how far back a user can return. If the server deletes the page properties for a page and the user then returns to that page, Ajax requests from the page will not function correctly.

To be totally safe, page properties must be retained for the lifetime of the session. However, this would bloat the size of session data and result in high overheads for reading and writing sessions (most frameworks read and write all of the data for a session monolithically, so all page properties for all pages will be read and written during each request).

For the Fiz implementation of page properties we have chosen to limit the number of pages in each session for which properties are retained. If the number of PageState objects for a session exceeds the limit, the least recently used PageState for that session is discarded. If a user invokes an Ajax operation on a page whose properties have been deleted, Fiz will not find the PageState object corresponding to the page identifier in the request. Fiz then generates an Ajax response that displays a warning message in the browser indicating that the page state is stale and suggesting that the user refresh the page. If the user refreshes the page a fresh page identifier will be allocated and Ajax operations will work once again; however, the act of refreshing the page will reset the page display (in the case of the TreeSection the tree will revert to its original display showing only the top-level nodes). We call this situation a *broken page*. Broken pages will be annoying for users so it is important that they not occur very frequently (of course, users will not notice that a page is broken unless they invoke an Ajax operation that requires page properties).

The frequency of broken pages can be reduced by retaining more PageState objects for each session, but this will increase the overhead for storing page properties.

In order to estimate the frequency of broken pages with LRU replacement, we ran a trace-driven simulation experiment. We wrote a Firefox add-on that records all operations that change the current Web page being displayed and transmits that information to a central server at regular intervals. The information logged includes new pages, "back" and "forward" history operations, redisplays, and tab switches. We used the add-on to collect data from about thirty people (mostly Stanford students and faculty) over a period of two months (approximately 200,000 page views in total). We then used the data to drive two simulations of the page property

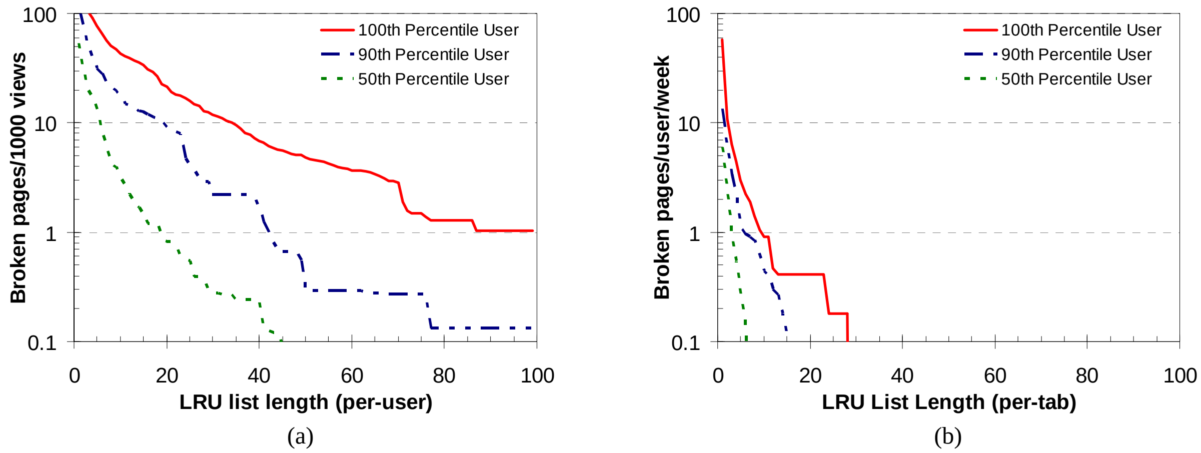


Figure 7. A trace-driven simulation of LRU lists for page properties assuming a single LRU list for each user's interaction with each server host (a) and separate LRU lists for each tab (b). The top curve in each figure shows the rate of broken pages for the worst-case user for each LRU list size, and the bottom curve shows behavior for the median user. 1000 page views represents roughly one week's worth of activity for a typical user.

mechanism.

In the first simulation (Figure 7 (a)) we assumed one LRU list of page properties for each session (a particular user accessing a particular server host). The figure shows the rate of broken pages as a function of LRU list length, both for "typical" users and for more pathological users. It assumes that every page uses Ajax and requires page properties. For the trace data we collected, an LRU limit of 50 pages per session results in less than one broken page per thousand page views for most users. The actual frequency of broken pages today would be less than suggested by Figure 7, since many pages do not use Ajax, but if Ajax usage increases in the future, as we expect, then the frequency of broken pages could approach that of Figure 7.

The primary reason for broken pages in the simulations is switches between tabs. For example, if a user opens an Ajax-driven page in one tab, then opens a second tab on the same application and visits several Ajax-driven pages, these pages may flush the page properties for the first tab, since all tabs share the same session. If the user switches back to the first tab its page will be broken.

Figure 7(b) shows the frequency of broken pages if a separate LRU list is maintained for each tab in each session; in this scenario LRU lists with 10 entries would eliminate almost all broken pages. Per-tab LRU lists will result in more pages cached than per-session LRU lists of the same size, since there can be multiple tabs in a session. In our trace data there were about 2 tabs per session on average; per-tab LRU lists used roughly the same memory as per-session LRU lists 2.5-3x as long. Overall, per-tab LRU lists would result in fewer broken pages with less memory utilization than per-user LRU lists, and they also improve the worst-case behavior.

Unfortunately, today's browsers do not provide any identifying information for the window or tab responsible for a given HTTP request: all tabs and windows participate indistinguishably in a single session. Such information would be easy for a browser to provide in a backwards-compatible fashion: it could consist of an HTTP header that uniquely identifies the window or tab for the request; ideally it would also include information indicating when tabs have been closed. Window/tab information also has other uses: for example, it would enable applications to implement sub-sessions for each tab or window so that the interaction stream for each tab/window can be handled independently while still providing shared state among all of the tabs and windows. Without this information, some existing Web applications behave poorly when a single user has multiple tabs open on the same application, because interactions on the different tabs get confused.

In the absence of tab identifiers, and assuming that Ajax becomes pervasive, so that virtually all Web pages need Ajax state, we conclude that servers would need to retain state for about 50 pages per session in order to reduce the frequency of broken pages to an acceptable level. In our current uses of page properties the amount of state per Ajax component is typically only a few tens of bytes (see Section 8), so storing state for dozens of pages would not create a large burden for servers.

It would also be useful to add priorities to the page property mechanism; the state for higher priority pages would be retained in preference to that for lower priority pages. It is particularly annoying for users to lose partially entered form data, so a priority mechanism could be used to preserve the state for pages containing unsubmitted forms. Once the form has been submitted successfully, the priority of its data could be reduced to allow reclamation.

7 Comparisons

After implementing and using both page properties and reminders, our conclusion is that the page property approach is the better of the two. Both mechanisms introduce overhead to transmit or store state, but the overheads for page properties are likely to be lower, since the state can be stored locally on the server without transmitting it over the Internet. The reminder approach has unique problems related to security, and the page property approach has unique problems related to garbage collection. However, we believe that the garbage collection issues for page properties are manageable and that the subtle security loopholes that can occur in the reminder mechanism will cause more catastrophic problems.

Ideally, the state management mechanism should scale up gracefully as Web applications make more intensive use of Ajax interactions and develop more complex state. We believe that page properties are likely to handle such scaling better than reminders. Consider a Web page with a collection of related Ajax-driven components. It is possible that multiple Ajax requests might be issued from different components simultaneously; for example, one component might be refreshing itself periodically based on a timer, while another component issues an Ajax request because of a user interaction. If the components are related then they may also share state (reminders or page properties). With the reminder approach each request will receive a separate copy of the relevant reminders and there is no obvious way for the concurrent requests to serialize updates to their reminders. With the page property approach the concurrent requests will access the same page properties, so they can synchronize and serialize their updates to those properties using standard mechanisms for manipulating concurrent data structures.

In both the reminder and page property mechanisms the state must be serializable. For reminders the state must be serialized so it can be transmitted to and from the browser; for page properties the state must be serialized to save it as part of the session.

8 Component Examples in Fiz

We have implemented three components in Fiz that take advantage of the page property mechanism; they illustrate the kinds of state that must be managed for Ajax requests.

The first component is the TreeSection that has already been described. The state for this component divides into two parts. The first part consists of overall state for the entire tree; it includes the name of the data source method that supplies data about the contents of the tree, the `id` attribute for the HTML element containing the tree, and four other string values containing parameters that determine how tree nodes are rendered into HTML. The second part of the state for a TreeSection consists of information for each node that has been displayed so far; it includes the `id` attribute for the node's HTML element and an internal name for the node, which is passed to the data source in order to expand that node.

The second component is an auto-complete form element. The component renders a normal `<input type="text">` form element in the page but attaches event handlers to it. As the user types text into the element, Ajax requests are issued back to the server with the partial text in the form element. The server computes the most likely completions based on the partial text and returns them back to the browser where they are displayed in a menu underneath the form element. The auto-complete component handles relevant mouse and keyboard events, issues Ajax requests, displays and undisplays the menu of possible completions, and allows the user to select completions from the menu. However, the auto-complete component does not contain code to compute the completions since this would restrict its reusability. Instead, it calls out to an external method to compute the completions during Ajax requests. The name of this method is provided as a parameter during the rendering of the original page, and it is saved in a page property along with the `id` attribute of the HTML form element. The auto-complete component is general-purpose and reusable: there can be multiple auto-complete form elements on the same page, and each auto-complete element can use a different mechanism to compute completions.

The third usage of page properties in Fiz is for form validation. When a form is initially rendered in Fiz, one or more validators can be associated with each field in the form. Information about these validators is saved in a page property, including the name of a method that will perform the validation, one or more parameters to pass to that method (for example, the `validateRange` method takes parameters specifying the end points of the valid range), and additional information used to customize the HTML formatting of error messages when validation fails. When the form is posted Fiz uses the information in the page property to validate the information in the form; if any validations fail then Fiz automatically returns error messages for display in the browser. In this case the browser-server communication mechanism is a form post rather than an Ajax request, but it uses the same page property mechanism. The form validation mechanism also allows forms to be validated dynamically as the user fills them in, using Ajax requests to request validation.

9 Related Work

9.1 View State

Most existing Web frameworks provide little or no support for managing the state of Ajax requests. One exception is Microsoft's ASP.NET framework, which includes a mechanism called View State that is similar to the reminder mechanism in Fiz. View State is used extensively by components in ASP.NET [5,8]. The View State mechanism provides each control with a ViewState property in which it can store key-value pairs. ASP.NET automatically packages all of the ViewState properties for all components into a single string, which is sent to the browser as part of the HTML for a page. Any "postbacks" for that page (which include both Ajax requests and form posts) include the View State, which is deserialized and made available to all of the components that handle the postback. Components can modify their View State while handling the request, and a new version of the View State is returned to the browser as part of the response.

The primary difference between View State and reminders is that View State is monolithic: all of the state for the entire page is transmitted in every interaction. In contrast, reminders are more granular: each request includes only the reminders needed for that request. The View State approach avoids the complexity of determining what information is needed for each request, but it results in larger data transfers: every request and every response contains a complete copy of the View State for the entire page. Many of the built-in ASP.NET components make heavy use of View State, so it is not unusual for a page to have 10’s of Kbytes of View State [8]. Complaints about the size of View State are common, and numerous techniques have been discussed for reducing its size, such as selectively disabling View State for some components (which may impact their behavior).

The security properties of View State are similar to those for reminders. View State is base-64 encoded to make it difficult to read, and by default a MAC is attached and checked to prevent tampering. However, the MAC is not session-specific so applications must attach an additional “salt” to prevent replay attacks between sessions; reminders eliminate this problem by using a different MAC for each session. View State also supports encryption, but by default it is disabled.

It is possible for applications to modify the View State mechanism so that View State is stored on local disk instead of being transmitted back and forth to the browser. This provides a solution somewhat like page properties. However, the approach of storing View State on disk does not appear to be heavily used or well supported. For example, the problem of garbage collecting old View State is left up to individual applications.

9.2 Query Values

A simpler alternative than either reminders or page properties is to use URL query values: if a component needs state information to handle an Ajax request, it can serialize that state when rendering the original page and incorporate it into the URL for the Ajax request as a query value; when the request is received, the component can read the query value and deserialize its state. This approach works well for simple state values where security is not an issue, and it is probably the most common approach used for Ajax requests today. However, it leaves serialization and deserialization up to the application and does not handle any of the security issues associated with Ajax state. In addition, the query-value approach does not easily accommodate changes to the state, since this would require modifying the URLs in the browser. The reminder mechanism handles all of these issues automatically. If the Ajax state becomes large, then query values will have overhead problems similar to those of reminders and View State.

9.3 Alternative application architectures

The issues in managing Ajax state arise because the functionality of a Web application is split between the server and the browser. In most of today's popular Web development frameworks this split must be managed explicitly by application developers. However, there exist alternative architectures for Web applications that change this split and the associated state management issues.

One approach is to change the application structure so that it is driven by Javascript in the browser. In this approach the server does not generate HTML; its sole purpose is to provide data to the browser. The application exists entirely as Javascript running in the browser. The initial page fetch for the application returns an empty HTML page, plus `<script>` elements to download Javascript. The Javascript code makes Ajax requests to the server to fetch any data needed for the page; the server returns the raw data, then Javascript code updates the page. As the user interacts with the page additional Ajax requests are made, which return additional data that is formatted into HTML by Javascript or used to modify the DOM. Google’s Gmail is one example of such an application.

In a Javascript-driven application there is no need for the server to maintain state between Ajax requests: all of the interesting state is maintained in Javascript structures in the browser. For example, if a Javascript-driven application contains a component like the TreeSection, parameters for the TreeSection such as the data source method and HTML formatting information are maintained in Javascript variables on the browser. The server can process each incoming request independently and there is no state carried over from one request to another. There are still security issues with this approach, but they are simpler and more obvious: the server treats each request as potentially hostile and validates all arguments included with the request.

The biggest disadvantage of Javascript-driven applications is the overhead of downloading all of the Javascript code for the application; for complex applications the Javascript code could be quite large [9]. This approach also requires the entire application to be written in Javascript, whereas the traditional server-oriented approach permits a broader selection of languages and frameworks. There exist a few frameworks, such as Google’s GWT [7], where application code can be written in other languages (Java in the case of GWT) and the framework automatically translates the code to Javascript.

Javascript-driven applications also have the disadvantage of exposing the application's intellectual property, since essentially the entire application resides on the browser where it can be examined. To reduce this exposure some applications use Javascript obfuscators that translate the Javascript code into a form that minimizes its readability. These obfuscators can also compact the code to reduce the download overheads.

Another alternative architecture is one where the partitioning of functionality between server and browser is handled automatically. The developer writes Web application code without concern for where it will execute, and the framework automatically decides how to partition the code between server and browser. In this approach the framework handles all of the issues of state management, including the related security issues. Automatic partitioning has been implemented in several experimental systems, such as Swift [2]. Although developers can use these systems without worrying about Ajax state issues, the framework implementers will still have to face the issues addressed by this paper.

9.4 Dynamic application state

The Ajax state discussed in this paper consists of information used to manage a Web user interface, such as information about the source(s) of data and how to render that data in the Web page. However, Ajax is often used to address another state management problem, namely the issue of dynamic application state. If the state underlying an application, such as a database or a system being monitored, changes while a Web page is being displayed, Ajax can be used to reflect those changes immediately on the Web page. A variety of mechanisms have been implemented for “pushing” changes from Web servers to browsers, such as Comet [3,13]. The issue of managing dynamic application state is orthogonal to that of managing Ajax state: for example, the techniques described in this paper could be used in a Comet-based application to keep track of the data sources that are changing dynamically.

10 Conclusion

Managing the state of Web applications has always been complex because the application state is split between server and browser. The introduction of Ajax requests requires additional state to maintain continuity across requests related to a single page, yet the stateless nature of most servers makes it difficult to maintain this state. Furthermore, if Ajax-driven interactions are to be implemented by reusable components then even more state is needed to maintain the modularity of the system.

This paper has explored two possible approaches to maintaining Ajax state, one that stores the state on the browser (reminders) and one that stores state on the server (page properties). Although both approaches meet the basic needs of Ajax-driven components, each of them has significant drawbacks. The browser-based approach introduces overheads for shipping state between browser and server, and it creates potential security loopholes by allowing sensitive server state to be stored in the browser. The server-based approach introduces overheads for saving state as part of sessions, and it has garbage-collection issues that can result in the loss of state needed to handle Ajax requests if the user returns to old pages. Based on our experiences we believe that the disadvantages of the server-based approach are preferable to those of the browser-based approach.

In the future we expect to see the use of Ajax increase, and we expect to see pages with more, and more complex, Ajax components. As a result, the issues of managing Web application state will probably become even more challenging in the future.

11 Acknowledgments

Jeff Hammerbacher and the anonymous referees made many useful comments that improved the presentation of the paper. This work was supported in part by a grant from the Google Research Awards program.

12 References

[1] Anley, Chris, *Advanced SQL Injection in SQL Server Applications* (http://www.nextgenss.com/papers/advanced_sql_injection.pdf).

[2] Chong, S., Liu, J., Myers, A., Qi, X., Zheng, L., and Zheng, X., “Secure Web Applications via Automatic Partitioning,” *Proc. 21st ACM Symposium on Operating System Principles*, October 2007, pp. 31-44.

[3] Cometd project home page (http://cometd.org/).

[4] Crockford, D., *The application/json Media Type for JavaScript Object Notation (JSON)*, IETF RFC 4627, http://tools.ietf.org/html/rfc4627, July 2006.

[5] Esposito, D., “The ASP.NET View State,” *MSDN Magazine*, February 2003,

[6] Garrett, Jesse James, *Ajax: a New Approach to Web Applications*, http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[7] Google Web Toolkit home page (http://code.google.com/webtoolkit/).

[8] Mitchell, S., *Understanding ASP.NET View State*, http://msdn.microsoft.com/en-us/library/ms972976.aspx, May 2004.

[9] Optimize a GWT Application (http://code.google.com/webtoolkit/doc/latest/DevGuideOptimizing.html).

[10] Ousterhout, J., *Fiz: A Component Framework for Web Applications*, Stanford CSD technical report, http://www.stanford.edu/~ouster/cgi-bin/papers/fiz.pdf, January 2009.

[11] Reenskaug, Trygve, *Models-Views-Controllers*, Xerox PARC technical notes, December, 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf).

[12] Reenskaug, Trygve, *Thing-Model-View*-Editor, Xerox PARC technical note, May 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf).

[13] Russell, Alex, *Comet: Low Latency Data for the Browser*, March 2006 (http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/).