Ajax and REST, Part 1

Advantages of the Ajax/REST architectural style for immersive Web applications

Bill Higgins

October 02, 2006

The more that server-side Web applications become immersive by following rich-application models and delivering personalized content, the more their architectures violate Representational State Transfer (REST), the Web's architectural style. These violations can decrease application scalability and increase system complexity. By achieving harmony with REST, Ajax architecture lets immersive Web applications eliminate these negative effects and enjoy REST's desirable properties.

In just 15 years, the World Wide Web has grown from a researcher's experiment to one of the technological pillars of the modern world. Originally invented to let people easily publish and link to information, the Web has also grown into a viable platform for software applications. But as applications have become more immersive by using rich application models and generating personalized content, their architectures have increasingly violated Representational State Transfer (REST), the Web's architectural style. These violations tend to decrease application scalability and increase system complexity.

The emerging Ajax Web client architectural style lets immersive Web applications achieve harmony with the REST architectural style. They can enjoy REST's desirable properties while eliminating the undesirable properties experienced when an application violates REST's principles. This article explains how and why Ajax and REST succeed together for immersive Web applications.

REST: The architecture of the Web

Although the World Wide Web was built upon decades of related research, its effective birth date was December 1990, when Tim Berners-Lee completed working prototypes of the Web's major components: Unified Resource Identifiers (URI), HTTP, HTML, browser, and server. The explosive adoption of the Web exceeded all of its pioneers' hopes. In his famous thesis (see Related topics), Roy Fielding describes the mood of the time:
"Although elated by its success, the Internet developer community became concerned that the rapid growth in the Web's usage, along with some poor network characteristics of early HTTP, would quickly outpace the capacity of the Internet infrastructure and lead to a general collapse."

"An architecture has, by design, a set of properties that allow it to meet or exceed the system requirements. Ignorance of those properties may lead to later changes which violate the architecture, just as the replacement of a load-bearing wall with a large window frame may violate the structural stability of a building." -- Roy Fielding

Fielding and others reexamined the Web's architecture and its adequacy to support massive expansion and use. The tangible result of this reexamination consisted of updates to important standards such as URI and HTTP. An intangible but meaningful result of the reexamination was the identification of a new architectural style for hypermedia applications, which Fielding named Representational State Transfer (REST). Fielding asserted that components deployed on the Web that accepted and aligned with the REST's design constraints would enjoy the Web's desirable properties. He also warned that Web components that diverged from the REST principles would not enjoy these benefits.

In the early days, most Web sites and simple Web applications naturally aligned with REST's principles. But as Web applications became more immersive, Web application architectures diverged from REST's principles and suffered the consequences. The problems of immersive server-side Web architectures are a bit tricky to deconstruct because a decade of using this architectural style has fostered the belief that these problems are intrinsic to Web application architecture. They're not. Rather, they are specifically caused by the server-side Web application architectural style. To break through these prejudices, it's helpful to look back at the architectural progression that led to the current state of affairs. I'll explain why many assumptions we've accepted are no longer valid now that creating Ajax applications is commercially viable.

A brief history of Web applications

Berners-Lee created the Web as a means for researchers to share documents across wide distances and to create simple links between documents to speed the dissemination of knowledge and ideas. However, the URI standard's architectural characteristics quickly enabled the sharing of more than static documents.

Web sites serving static documents

The earliest content on the Web consisted of static HTML documents with links to other static documents, as illustrated in Figure 1:

Figure 1. A Web site serving static documents.
REST makes retrieving static documents incredibly efficient and scalable because they can be easily cached based on the URI and last-modified date. Soon developers would move beyond static documents and support serving dynamic content.

**Early dynamic Web applications**

Berners-Lee and others designed the URI standard to support universally unique identification of a resource while allowing its representation (HTML, text, and so on) to vary based on a negotiation between a Web client (typically a Web browser) and a Web server. Because URI differentiates between resource identification and the resource's underlying storage mechanism, Web developers could create programs that inspect a URI's syntax and generate a document on the fly, combining predefined UI elements and dynamically retrieved data, often from a relational database (see Figure 2). Though the documents are generated, their caching characteristics are identical to those of static files.

Figure 2. A Web site serving database records embedded in HTML template code

A simple example of these early applications is a university directory Web application. The application typically works like this:

1. A user enters a name (for example, Bill Higgins) in a Web form and clicks a Submit button. 
2. The form creates a URI based on the name entered and requests the content of this URI from the server (for example, GET http://psu.edu/Directory/Bill+Higgins). 
3. The server examines the URI and generates a Web page with the student's phone number and address. 
4. The server sends the generated page back to the user's browser.

An important property of this interaction is that it’s **idempotent**, meaning that the output would be identical for an identical request, unless the underlying resource had changed (for example, if Bill changed his phone number). This means that a browser or proxy server can cache the Bill Higgins document locally and, if the underlying resource hasn't changed, retrieve the resource from the local cache rather than the remote server. This approach improves user-perceived responsiveness and increases overall system efficiency and scalability. These early dynamic Web applications worked well and brought a vast amount of information to users' fingertips.

**Immersive Web applications**

The next generation of Web applications aimed to be highly immersive, providing personalized content and rich application models. Over the next 10 years, Web developers succeeded in creating these immersive applications. A quintessential example is the Amazon.com e-commerce site. As a user interacts with the Amazon Web application, it creates complex custom pages that
recommend particular products, show browsing history, and display the cost of items placed in the user's shopping cart.

**Immersive server-side applications and REST**

Immersive Web applications are certainly useful, but the server-side immersive Web application style is in fundamental disharmony with REST architectural principles. Specifically, it violates a key REST constraint and fails to take advantage of one of REST's most important advantages, thereby spawning a new set of problems.

**Violating the "stateless server" constraint**

REST's "client-stateless-server" constraint forbids session state on the server. Designing within this constraint promotes the system properties of visibility, reliability, and scalability. But immersive server-side Web applications wish to provide a great deal of personalization to a single user, so they must choose between two designs. The first is to send a massive amount of state information with each client request, so that each request is context-complete and the server can remain stateless. A second, ostensibly simpler, solution favored by application developers and middleware vendors alike is to send a simple user identity token and associate this token with a "user session" object on the server side (see Figure 3). The second design directly violates the client-stateless-server constraint. It certainly enables desirable user functionality (especially personalization), but it places tremendous strain on the architecture.

**Figure 3. An immersive server-side Web application containing large amounts of server-side session state**

Java™ Servlet's HttpSession API provides an example of this strain. HttpSession lets you associate session state with a particular user. This API seems deceptively simple to novice developers. Indeed, it appears that you can store any object in HttpSession and pull it out without ever coding any special lookup logic yourself. But as you start putting more objects in HttpSession, you start to notice that your application server uses more and more memory and processing resources. Soon you decide that you need to deploy your application in a clustered environment to help with growing resource needs. Then you realize that for HttpSession to work in a clustered environment, each object must implement Java's `Serializable` interface so that session data can be transmitted between servers in a clustered environment. Then you must decide whether or not your application server should persist session data in the case of a shutdown/restart cycle. Soon you begin to question whether violating the client-stateless-server constraint was such a good idea after all. (Actually, many developers are ignorant of this constraint.)
Making distributed caching impossible

A second severe consequence of the immersive server-side Web application is that it's practically impossible to take advantage of REST's first-class support for resource caching. To quote Fielding: "The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server."

You can look at an immersive Web application as almost a living entity, continually changing based on new input that the user provides, new input provided by others, and new back-end data. Because the server must generate each page based on multiple users' interactions with the application, it's practically impossible to generate the same document twice. As a consequence, a Web browser or proxy servers can't cache server resources.

Several solutions are available to deal with the problems of resources not being cacheable. One is to create server-side caches of fine-grained resources so that the server can build a coarse-grained page from preassembled parts rather than from basic elements (HTML and data). But the problem remains: every request results in nontrivial server processing, hurting scalability and, potentially, user-perceived response time.

Another consequence of not being able to serve cacheable resources is that extremely dynamic Web applications must explicitly forbid search engines and other "bots" from making requests, because each request involves expensive processing; in a RESTful application you'd serve up a resource once to each bot and send a simple "Not-modified" message on subsequent bot visits.

Client-side processing without Ajax

As more users access a Web application, the system requires more resources. You can ask the server to do more, but you'll need a bigger server or clustered servers (and server-side state doesn't play well in clustered environments). But if you distribute processing to clients, with each new user, you have a new PC to support part of the new load. And if you distribute session state to the client, then you have a stateless server -- a desirable property of a scalable Web application. It seems like a no-brainer, so why aren't all immersive Web applications designed this way? Before Ajax, the answer was simple: Because application state is destroyed each time the user visits a new Web page.

Each time you visit a Web page, you download a file or set of files containing content (data surrounded by structural information such as tables and lists) and styles affecting the appearance of the content (for example, red text). Within your Web browser, this information is viewed as an abstract set of document objects. Take this list for example:

- Ford
- BMW
- Toyota

Your browser considers this HTML to be an "unordered list" object containing three list elements; each list element contains text. The entire document can be viewed as a complex tree of interrelated objects. When you browse from one page to the next, your browser destroys the object tree for the current page and creates a new object tree for the next page.
But why centralize so much resource consumption on an overburdened server, when in theory you could distribute processing and memory needs to clients? The simple answer is that given traditional Web browser constraints, this wasn't feasible (see Client-side processing without Ajax). But the Ajax architectural style lets developers distribute processing and state requirements to the client. Read on to learn why immersive applications that choose an Ajax-style architecture can regain harmony with REST and enjoy its benefits.

**Ajax and REST**

As you saw earlier, traditional server-side Web applications combine presentation and dynamic data elements on the server and return fully formed HTML documents to the browser. Ajax applications are different in that the majority of their UI and application logic resides within the browser; the browser-based application code fetches new server data as necessary and weaves this data into the current page (see Related topics for Jesse James Garrett's seminal article on Ajax). The location of presentation and data binding might seem like an implementation detail, but this difference leads to completely different architectural styles.

**Enjoying a stateful Web client**

People often describe Ajax applications as Web pages that don't need to perform a full page refresh on every click. This description is accurate, but the underlying motivation is that full page refreshes are distracting and detract from an enjoyable, immersive user experience. Full page refreshes are even nastier from an architectural point of view in that they eliminate the option of storing application state in the client, which in turn results in design decisions that prevent applications from taking advantage of many of the Web's strongest architectural design points.

The fact that Ajax lets you interact with a server without a full refresh puts the option of a stateful client back on the table. This has profound implications for the architectural possibilities for dynamic immersive Web applications: Because application resource and data resource binding is shifted to the client side, these applications can enjoy the best of both worlds -- the dynamic, personalized user experience we expect of immersive Web applications and the simple, scalable architecture we expect from RESTful applications.

**Caching the Ajax engine**

Imagine that Amazon.com was completely reimplemented as a pure Ajax application -- a single Web page that fetches all of its data dynamically from the server. (Amazon might not want to do this for good business reasons, but that's a matter for another article.) Because a large amount of the UI and application logic now runs in the client rather than the server, the initial page load needs to download Amazon's Ajax "engine," as described by Garrett. This engine contains a bunch of application logic (implemented as JavaScript code) and some UI scaffolding that is populated later with business data fetched asynchronously from the server (see Figure 4):
An interesting characteristic of the Ajax engine is that although it contains a great deal of 
application logic and presentation framework elements, if designed properly, it contains no 
business data or personalized content. Application and presentation are frozen at deployment 
time. In a typical Web environment, application resources might change only every one to six 
months. This means that an Ajax engine that segregates application resources and data resources 
is highly cacheable.

The Dojo Toolkit is a good example (see Related topics). Dojo provides build-time tools to create 
a compressed JavaScript file containing all of your application's logic, presentation, and styles. 
Because it's ultimately just a file, a Web browser can cache it, meaning that the second time you 
visit a Dojo-enabled Web application, you're likely loading the Ajax engine from your browser's 
cache rather than from the server. Compare this to a highly immersive server-side Web application 
where every request results in significant server processing because your browser and network 
intermediaries can't cache the ever-changing resources.

Because the Ajax application engine is just a file, it's also proxyable. On a large corporate intranet, 
only a single employee might ever download a particular version of the application's Ajax engine, 
and everyone else just picks up a cached copy from the intranet gateway.

So with regard to application resources, a well-designed Ajax application engine aligns with REST 
principles and provides significant scalability advantages versus server-side Web applications.

Caching Ajax data

Okay, users browse to an Ajax Web site and load the Ajax application engine, preferably from the 
browser's cache or, if not, from a local proxy server. What about the business data? Because the 
application logic and state reside and execute on the browser, the application's interaction with 
the server becomes very different from that of a traditional Web application. Instead of fetching 
amalgamated pages of content, it simply needs to fetch business data.

Going back to the Amazon.com example, imagine that you click a link to view information about a 
book on design patterns. In Amazon.com's current application, the link click action sends a variety 
of information identifying the requested resource. It also sends various session-state artifacts that 
let the server create a new page that includes prior session state (such as recently viewed items), 
personalization information (such as "you purchased this book in 1999"), and the actual business 
resource itself. The application is very dynamic and highly personalized -- but not cacheable 
and not natural scalable (though as Amazon demonstrates, these architectural problems can be 
overcome with millions of dollars of infrastructure engineering). Now consider this action in the 
(imaginary) Ajax version of the application. No processing needs to occur with regard to "recently
viewed items." This is simply information already present in the page that won't go away when you click on a link. Two requests will likely be related to the design patterns book:

- /Books/0201633612 (where 0201633612 is the design pattern book's ISBN number)
- /PurchaseHistory/0201633612/bhiggins@us.ibm.com

The first hypothetical request returns information about the book (author, title, descriptions, and so on); it contains no user-specific data. The absence of user-specific data means that as more users request the same resource, it is likely that they'll retrieve cached versions of it from intermediate nodes on the Internet rather than from the originating server. This characteristic reduces server and overall network load. The second request, on the other hand, contains user-specific information (Bill Higgins' purchase history for this book). Because this data includes personalized information, only a single user should ever successfully retrieve and cache the data from this URI. Although this personalized data doesn't have the scalable characteristics of the nonpersonalized data, the important point is that this information is retrieved from a distinct URL and therefore has the positive characteristic that it doesn't interfere with the caching of other application and data resources that are cacheable.

**Ajax and robustness**

Another benefit of the Ajax architectural style is the ability to deal easily with server failure. As I mentioned before, server-side Web applications with immersive user experiences tend to hold large amounts of user session state on the server. If a server fails, the session state goes away and users experience odd browser behavior ("Why am I back to the home page? And where are the items in my shopping cart?"). In an Ajax application with a stateful client and stateless services, a server crash/restart can be completely transparent to the user because the server crash can't affect session state, which lives in the user's browser; a stateless service's behavior is idempotent and determined solely by the content of user requests.

**Promise and problems**

For the class of Web applications that I call immersive Web applications, well-designed Ajax/REST applications are far superior to traditional server-wide Web applications with regard to user experience, responsiveness, and scalability. However, an architectural style's run-time characteristics aren't the only determinant of success for a software project and Web application. There are some tough non-run-time problems with creating Ajax/REST applications, including determining how to adopt Ajax for your application (if at all), problems of large-scale JavaScript development, cultural issues, and packaging problems. In Part 2, I'll discuss different adoption options and organizational considerations to improve your chances for success with Ajax.

**Acknowledgements**

I'd like to thank my colleagues Chris Mitchell, Josh Staiger, Pat Mueller, Scott Rich, and Simon Archer their helpful technical feedback on this article. I'd also like to thank James Governor and Steve O'Grady of the Redmonk analyst firm, who originally challenged me to consider REST-style Web services.
Related topics

• "Architectural Styles and the Design of Network-based Software Architectures" (Roy Fielding, University of California - Irvine, 2000): Fielding's doctoral dissertation describes the architectural constraints of REST.


• *All Things Distributed: Werner Vogels' weblog on building scalable and robust distributed systems* (Werner Vogels, Amazon.com): Amazon's CTO probably knows much more about scalability than the author admittedly does, but Amazon's Web application happens to be one of the best known immersive server-side Web apps on the planet, so it makes a great example.

• **The Dojo Toolkit**: The Dojo Toolkit is an open source Ajax framework that provides higher-level programming abstractions and normalizes browser incompatibilities.