Design and Architecture of the Brazil Web Application Framework

Stephen A. Uhler

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, California 94303

Introduction

Web servers and their more impressively named cousins, "Web Application Frameworks", constitute the single most important component of the network content delivery system we know as "The Web". The first Web Servers started to appear in 1994 on UNIX® systems on the Internet. The design of those early systems reflects their UNIX heritage. URLs (Uniform Resource Locators) are equivalent to UNIX file names. Each URL, when requested by a client program, typically a "Web Browser", is mapped to the UNIX file of the same name, wrapped in HTTP (HyperText Transport Protocol), and delivered to the client. In those cases where content cannot be represented as a static file and needs to be dynamically generated, the URL names the program that is run to generate the content. This capability, known as CGI (Common Gateway Interface), stems from the traditional UNIX practice of making everything look like a file. Thus each URL represents a file that either contains the content, or contains the program that is used to generate the content.

Background

Over the next five years the WEB saw explosive growth, and the architecture of the original Web Servers, though simple and elegant, was beginning to strain. Static content was still delivered effectively by mapping URLs into files, but dynamic content was becoming problematic. The notion of programs as files, as well as the mechanisms for identifying, launching, managing, and communicating with CGI programs is very specific to the UNIX operating system, which makes porting web servers and their corresponding content to non UNIX systems difficult. In addition, as content management techniques required more of the content to be generated dynamically, even if simply to paste together several static files in response to a single URL, the CGI programs rapidly became the bottleneck. Each dynamic page requires a separate program to be launched and executed by the operating system, only to be terminated each time a request is completed. In addition, the communication between the Web Server and the CGI program is very limited.

Only the URL and its corresponding HTTP envelope information is made available to the CGI program, which can only return content; the ability to pass meta-information back to the server is almost nonexistent.

The next state in the evolution of Web servers focused on eliminating the CGI bottleneck, specifically the program creation and execution step required for each URL requested. Generally, three different approaches have been taken: keeping the basic CGI interface, only making it faster; building web server specific APIs, often by requiring the dynamic code generating portions to be bound into the same process as the web server; or defining language-specific APIs whose implementations don't require the overhead implied by the CGI model.

The *FastCgi* interface tries to improve the performance of the CGI specification by eliminating the process creation and execution step at every request, yet maintaining backward compatibility wherever possible. The *FastCgi* inter-face, rep-resented by the file that maps from the URL, is created and started once when the web server starts. Multiple requests for the same URL are sent to the same *FastCgi* process by defining a request packet protocol than can accommodate multiple requests and responses for each *FastCgi* process. *FastCgi* has the advantage of preserving a separate execution context for dynamic content generation, while eliminating the bulk of the process creation overhead of traditional CGI programs. Consequently *FastCgi* pro-grams are easily ported to work with many different web servers.

The second approach to eliminating the CGI bottleneck is to move the dynamic content generation into the same execution context as the server, by expressing dynamic content generation in terms of APIs that are specific to a particular web server. This approach eliminates the process creation and execution overhead of CGI programs entirely, but at the expense of close coupling to a particular web server. Most major web servers provide such API definitions. However, dynamic content generation using these APIs is rarely portable to a different server. In addition, by having the dynamic content generation in the same execution context as the server, a bug in a dynamic content generation module can negatively impact the entire web server, including URL requests that have nothing to do with the bug-containing module.

The third approach used to eliminate the CGI bottleneck is to create a set of language-specific APIs that can be logically bound into the execution context of the web server, yet be defined in a web server independent way. Servlets are the leading example of this approach. A servlet is a Java™ program-ming language that conforms to a defined set of Java APIs, which can be (and have been) implemented to provide dynamic content for many different web servers. Thus servlets combine the advantages of *FastCgi* -- portability to

different web servers -- with the close coupling of server specific extensions.

The issues

Although all three approaches reduce the performance problems associated with the CGI interface, they still fundamentally retain the notion of a one-to-one mapping between URLs and files, where URLs are unrelated to each other.

As the web has grown, the notion that every URL request and its associated file is independent of any other request has become a serious architectural roadblock. It is now common for a single Web "form" to be spread over multiple pages (URLs), or for a single user to have unique state associated with a sequence of requests that may span days or even years. Finally, as the sheer volume of content on the web has mushroomed, it is no longer appropriate to assume, as is implicit in the CGI one-file- per URL model, that the content resides on the server machine at all. The software architecture that was designed to deliver individual pages in response to URL requests is now being used to build sophisticated applications, whose content happens to be wrapped in HTTP. Somewhere in the switch from delivering static files as URLs to creating full-blown applications, web servers became web application development frameworks.

As the need for more sophisticated features has grown, so too have the capabilities of the web servers used to implement them. However, they are still based on the original one file per request architecture that was seemingly elegant in the old days, but now just gets in the way. To support these added capabilities, the size and complexity of the APIs has grown. The descendants of the CGI architecture are stressed to provide functionality that isn't a good fit for their designs. As an example, a recent Servlet API (2.0) needs over two dozen classes and almost ten times that many methods to describe its interface.

To be fair, the entire reason for the explosion of interface complexity isn't totally due to the complexity of the interactions required by implementors of the interface. As web servers have become web application frameworks, the notion that the same pile of content can be delivered by any server has persisted. Somehow the "content" is viewed as separable from the server used to deliver it. Consequently, every new web server that arrives on the scene feels obliged to incorporate every nuance and capability of every previously deployed server, to insure that pre-existing content can be delivered with the new software without change. This "feature-bloat" adds significantly to the size of the system, while providing only a small increase in capability.

A new vision for the web

As the web matures, we see a transition away from the current client-server paradigm, where browsers are clicking at particular web sites, whose servers deliver all the content on that site. Instead, a more distributed model will

emerge. In this new model, both the traditional browsers and servers will still exist, but the content received by the client for a particular page is likely to have been retrieved from several traditional back-end servers, and modified to suit the requirements and desires of the particular client. This is akin to a traditional *workflow* business model, where the content passes through various stages, and is transformed at each stage.

Early versions of these intermediate stages, we'll call them *meta-servers*, are already starting to appear on the web. Some of the *meta-servers* are called "portals", and others are known as "content aggregators". In our view, portals and content-aggregators are one in the same. Its looks like a portal when viewed from the perspective of the client, and a content aggregator from the perspective of the traditional server (we'll dub *content-server*).

As these *meta-servers* begin to play a more prominent role in the infra-structure, they will have a profound impact on the way in which traditional content-servers are constructed. No longer will the content-server produce both the content and its presentation (look and feel). Meta-servers will transform the content after it leaves the content-server, allowing content-servers to be simpler. Today's content-servers not only provide the content, but manage the presentation, user preferences, and browser differences as well. In the future, content-servers can be simpler, providing just the content. The integration with other content, as well as the shaping of the look and feel for a particular browser will be added in stages by various *meta-servers* as the content flows toward the ultimate consumer.

Many types of content that are not traditionally located on a web server will become available. This new content, not able to stand on its own in the traditional web world, will be consumed by *meta-servers* which will integrate it with information from other content and meta-servers. Devices, sensors, and actuators will be accessible over the web, and will have their information integrated into the web fabric created by the network of content-servers, devices, and *meta-servers.*

Brazil

Brazil is a new architecture and sample implementation for building both content-servers and *meta-servers.* In the content-server context it permits the attachment of simple devices to the web with the barest of capabilities, squeezing into the tiniest places - a *micro-server*. In the *meta-server* context, it provides rich and flexible mechanisms for syn-thesizing, transforming, and integrating content: content retrieved both from traditional content-servers as well as the new breed of *micro-servers.* Finally, the architecture provides capabilities to integrate with traditional N-tier ap-plications, providing the bridge between the current client-server web into the future.

To achieve this two part goal, a two part strategy is taken. The existing notion of mapping URLs to UNIX files is abandoned. Modern URLs are too fluid to have a fixed binding to underlying files. Indeed, many small devices have no notion of file systems at all. Specific mechanisms used to implement existing content-server capabilities are discarded. For example, most traditional content-servers use `.htaccess` files to provide password protection for content. The file based nature of the `.htaccess` mechanism is inappropriate for the Brazil architecture, so `.htaccess` support is not built-in. Password protected URLs are still available, albeit via a different mechanism.

The second part of the strategy is based on defining a series of abstract capabilities for Brazil that support the entire range of applications, from the tiniest *micro-server* to a more traditional *content-server* to a sophisticated *meta-server*. This determines an architecture that starts with the small core and simple interface for adding functionality required for a *micro-server* implementation, and adds to it a set of composable, interchangeable modules that can operate together in a scalable way. With modules for manipulating traditional file-based content, the traditional content-server capability can be obtained. By adding modules that can string together arbitrary relationships between users and pages, and combining them with modules that can obtain and manipulate foreign content, sophisticated *meta-servers* are possible.

The Brazil architecture

Four key components and the interrelationships between them define the Brazil architecture. These components are described in the context of the prototype implementation, written in the Java programming language. Java objects represent two of the components, called `Server` and `Request`. The third component, a Java interface definition called a `Handler`, is the mechanism by which functionality is added into Brazil. The final Brazil component is the data structure for managing the information flow between the other parts, called the Brazil *properties*, named after the Java base class used in the prototype implementation. The *properties* are the name/value pairs that represent the current state of a URL request, along with methods for managing both the lexical and temporal scope of the data.

Building *micro-servers* with Brazil

As content management capabilities are shifted from traditional web servers to meta-servers, the traditional web server can focus entirely on the content it needs to deliver. At the extreme, it becomes a micro-server, delivering domain specific content in a bare bones way. These smaller, simpler servers can now be attached to sources of content that previously would be considered too small or unimportant to justify their own web servers. Examples include a digital thermometer whose content consists of the temperature of something, or a light switch, whose content is either *on* or *off*.

Although a web server whose content consists entirely of "on" or "off" might not make it into the website "top ten" list, when used in conjunction with meta-servers that can aggregate content from this and hundreds or millions of other similar servers, the content suddenly becomes quite interesting.

We use the term UPI, which stands for URL Programming Interface, to talk about the capabilities of these micro-servers. A UPI is just like an API, or Application Programming Interface, traditionally described in terms of specific programming language bindings, only UPI's are described in terms of URLs. Taken in this light, a URL no longer represents a file, instead it represents a set of programmable interfaces or remote procedure calls, that happen to be accessible via HTTP.

Using Brazil as a micro-server becomes defining a UPI for the desired functionality, using the built-in HTTP protocol stack as the transport mechanism, and writing the code to adapt the existing applications functionality to the UPI.

The `Server` object is the simplest of the four Brazil components. It represents the information relevant for the life of the Brazil server. This includes the port number the server is contacted on, the name of the `handler` (described below) that will turn a URL request into content, and an initial set of `properties`, used by the `handler` (or `handlers`) to satisfy an HTTP request. A Brazil application may have one or more active `Servers`, which usually operate independently.

When a URL request arrives at the `server`, it creates a `Request` object. The `Request` contains all of the information that pertains to client's URL request as well as methods that encapsulate the HTTP protocol. Then the `Server` arranges for all information pertinent to this URL request to be added to the `properties` object. Finally, the `handler` is called to produce the content.

A `handler` is the interface that defines how URLs get mapped into content. It consists of two methods, *`init`* and *`respond`*. When the `Server` starts, it creates an instance of the handler, and calls its `init` method, providing it with a reference to the `Server` object. Each time a `Request` object is created, in response to an HTTP request, the `respond` method is called, and supplied the `Request` object as a reference. The Handler examines the request, and by using the methods in the `Request` object, formulates an HTTP response. Once the request has been satisfied, the `Request` object is discarded.

If any parameters are required to configure the handler, they are placed in the "properties" when the server is started. The handler can find its configuration information either in the `Server` object passed to the `init` method, or in the `Request` object provided with each request.

The setup described so far is ideal for "micro-server" applications. The `Server` and `Request` objects provide the framework for encapsulating and managing HTTP requests, and the handler maps the URLs onto device specific functionality. There is little unused infrastructure, and implementations can be made quite small. Configuration information required by the `handler` is provided to the `server` at startup time, and passed to the `handler` when its methods are called.

Building *meta-servers* with Brazil

The creation of *meta-servers* , that operate both as portals and content aggregators, use the same framework, and the identical interfaces as the micro-server. However, instead of building a system from a single handler that would need to be modified or rewritten for each new *meta-server* application, the meta-server is constructed as a cooperating collection of handlers, whose arrangement and configuration can be modified to provide a wide range of capabilities.

Because the `handler` interface is so small, it is easy to create a handler that functions both as a consumer of the handler interface, as in the micro-server example above, and as a provider of the handler interface. This insight lets us create a handler that calls other handlers, permitting multiple handlers to participate in the processing of each HTTP request. By combining these "interior node" handlers with the simple, or "leaf" handlers, a directed graph of handlers can be created. This permits the construction of *meta-servers* by combining small bits of reusable functionality together.

A simple yet powerful use of "interior node" handlers in Brazil can be illustrated by the Brazil `ChainHandler`, which *chains* together a list of other handlers (possibly including other `ChainHandlers`), forming the basic mechanism for creating handler trees.

As indicated above, the data used to configure the handler is placed into the "properties" when the server is started. As long as there is only one handler, this scheme works fine. However, when multiple handlers are used in the same server, configuration collisions can result either from different handlers choosing the same name for a configuration parameter, or the same handler instantiated multiple times with different configurations.

To overcome this limitation, configuration properties for handlers are statically scoped within the `properties` to allow handlers that use the same configuration property names to have different values. Each "interior node" handler is responsible for creating the set of handlers that use it as the containing side of the `handler` interface. When each of the handlers is created, it is assigned a name which it uses to identify its configuration parameters, thus avoiding any possibility of name collisions.

For a handler used in the "micro-server", any `request` that is not dealt with

explicitly results in the server sending the requester an HTTP `Not Found` response. In the *meta-server* case, where multiple handlers have the opportunity to examine and respond to an HTTP request, a handler may alter the state of the current HTTP request without providing content to the requester. This alteration can take the form of modifying the configuration parameters of other handlers by changing the appropriate values in the "properties" object.

Because the "properties" is a stack, and handlers typically retrieve their configuration properties from the top of the stack, the duration over which a handler's configuration is altered is controllable by manipulating the "properties" stack. In the common case, changes one handler makes to another's properties will only be in effect for the duration of the current request.

A simple *meta-server* example

A common feature of many web servers is to allow users on a timesharing system to have their own private directory of files that are delivered as URLs. URLs that begin with `/~joe` would be delivered from *joe's* private directory of files instead of the main server directory. While most servers have this special capability built-in, the same effect is easy to provide in Brazil with a pair of `handlers` that co-operate with each other. The `FileHandler` is used to convert URLs into UNIX file names, and deliver the content of the files to the client. It is configured with a *document root*, the directory in the file system that acts as the root of the URL space.

To manage user's files, a "user-file" Handler is run *before* the `FileHandler`. If the URL starts with /~ the handler modifies the request by changing the URL by removing the user name portion, and setting the `FileHandler's` document root parameter in the "properties" to the proper user's home directory.

When the `FileHandler` gets the request, it delivers the proper file from the user's directory, based on the new configuration parameters placed in the "properties". When the next request comes in, the new configuration information will have been popped from the properties stack, and be unaffected by the previous modification. The same file handling code is reused in a different context.

Just as the "user-file" handler permits reuse of existing capabilities, by changing the "properties" to reconfigure the server "on-the-fly" in response to a particular request, other "handlers" use the same technique to provide password protection, session management, URL mapping, and a host of other services.

Important components

Just having a mechanism for composing handlers is not sufficient for creating a full featured meta-server. That requires many handlers, each performing a different task, but working together to

create a powerful content manipulation environment. In this section we'll visit some of the more important handlers and describe how they work together to create the Brazil *meta-servers* environment.

The first class of handlers, of which the `FileHandler` described above is an example, can be used together to provide the functionality of a traditional web server, including delivering files, running CGI scripts, providing password protected pages, and interfacing to other, non HTTP protocols such as LDAP or JDBC. Sometimes the meta-server needs to act as a traditional web server too.

The next class of handlers performs the content aggregation capability required by a meta-server. These handlers act as HTTP clients and retrieve content from a different server. The core of this capability is a fast proxy that implements the client side of the HTTP protocol. The `ProxyHandler` causes entire web sites to appear as if the content were stored locally in files. As each URL is retrieved from a content-server, the contents are examined, and every URL that points back to the content-server is rewritten so as to appear locally. When used in conjunction with the `FileHandler`, the `ProxyHandler` pro-vides the illusion of a single UNIX filesystem, where arbitrary sub-directories are actually retrieved dynamically from other servers. This capability, called "web mount", provides an analogous set of semantics as the "filesystem mount" facility does for ordinary files in the UNIX operating system.

Another interesting content aggregation handler is designed for use with micro-servers (or traditional web servers), whose simple content needs to be integrated with additional data in order to be presented to the user in a meaningful way. Content retrieved from this handler is converted into a set of name/value pairs and placed into the "properties" for further processing. The property values are then used by other handlers to formulate the final response. The content may be extracted from other web sites synchronously each time a request arrives, or in the background, updated on a periodic basis. Using the background method, the client doesn't need to wait for the data to come from the other server; the most recently obtained values are used instead. An interesting variation on the use of this handler is the ability to provide micro-servers that dynamically affect the operation of the main server, by returning values that represent configuration parameters of one or more handlers in the main server.

The third category of handlers is used to manipulate content once it has been obtained. These handlers come in two flavors, content extraction and content integration. The content extraction handlers use the HTML and XML processing capabilities provided by Brazil to analyze and decompose content, and convert it into name/value

pairs that are stored in the "properties". It doesn't matter whether the content was obtained locally, from a file, or remotely from a remote content-server.

The content integration handlers also use the HTML and XML processing capabilities, but this time to insert the previously extracted content into XML templates for final delivery to the requester. The "properties" are used as the rendezvous location, not only to characterize both the HTTP request and the server configuration, but to hold extracted content as well. Handlers can access and manipulate all three kinds of data in a uniform manner.

Handlers in the final category, unlike the other handlers described so far, don't participate directly in generating or manipulating requests or responses. Instead, they are used to insert alternate implementations for key data structures used by the server. For example, there is an implementation of "properties" that may be installed, with a handler, that causes portions of the name/value pairs to be stored and retrieved by a database, providing horizontal scalability and persistence for demanding applications.

Simply by rearranging handlers, and changing the way they interact with each other, a wide variety of web services can be created, often without the need to create new handlers. The following two examples are typical of services that are easily crafted using the current Brazil implementation. The first Brazil micro-server defined a UPI for accessing smartcards. We were able to add smartcard based identity, authentication, and payment to several existing web-sites, with only minor changes to the the existing sites. As is often the case with Brazil applications, we were able to reuse the smartcard UPI in a totally different context: combining it with the Brazil public key Certificate Authority handler to enable smartcard-authenticated delivery of public key certificates.

We built a micro-server that extracts real-time sensor data from home appliances, and a corresponding meta-server that inserts the sensor data into pages of an existing website, while not requiring a single modification to the original web site. From the perspective of the user, the appliance sensor data appears to be seamlessly integrated into the original web site.

Summary

By using a simple interface, in conjunction with powerful, reusable components, the Brazil system is able to deliver a wide range of flexible web solutions, ranging from tiny micro-servers, to traditional web capabilities to fully functional meta-servers that provide sophisticated portal and content aggregation capabilities.

References

http://www.experimentalstuff.com/
A web site that uses, describes, and supplies downloads of the current
Brazil prototype.