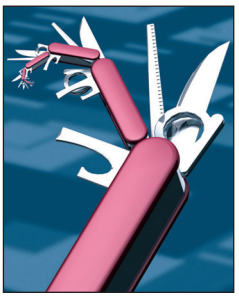
Developing RESTful Web Services with Webmachine



Justin Sheehy • *Basho Technologies*
Steve Vinoski • *Verivue*

In the November/December 2008 issue, as part of his old "Toward Integration" column, Steve Vinoski published his "RESTful Web Services Development Checklist."[1] There, he covered several areas and issues to which developers of RESTful (Representational State Transfer) Web services should pay close attention. His checklist covered the following areas of concern:

- identifiers, resources, and applications;
- representations and media types;
- HTTP methods; and
- conditional `GET`.

In this column, we use this checklist to evaluate Webmachine (see the Web extra at www.computer.org/cms/Computer.org/dl/mags/ic/2010/01/extras/mic2010020089s.pdf), a RESTful Web services development framework implemented in Erlang. In "Build Your Next Web Application with Erlang," Dave Bryson and Vinoski provided a high-level overview of Webmachine.[2] Here, we'll take a more thorough look at the abstractions and mechanisms Webmachine supplies to RESTful Web services developers.

Not Your Typical Framework

Web services frameworks typically fall into one of the following broad categories:

- Many frameworks lean heavily toward three-tier applications and focus almost entirely on backend database integration and object-relational data mappings.
- Some cater to users of a particular programming language by hiding the Web behind language-specific constructs.
- Others provide low-level access to HTTP requests and responses but don't provide much in the way of abstractions, models, or rules.

Typically, only the last of these approaches is in any way helpful to developers of RESTful Web services because, for a given request, they can access all the HTTP headers and method names, the full target URI, and any request body. The approach also allows full control over response headers and bodies. However, such access is usually quite raw and can thus be difficult to use. The first two approaches often work against RESTful services developers because they provide abstractions that hide Web details in an attempt to make things easier for the average developer. In doing so, they hide the very details required for implementing RESTful Web services.

Webmachine doesn't fit into any of these categories. Instead, it focuses on systematically applying standard HTTP semantics to Web application resources. The fundamentals of Webmachine were originally inspired by the seminal "HTTP headers status diagram" published in January 2007 by Alan Dean, currently the CTO of MoveMe.com (www.moveme.com). This decision flowchart's current version (see http://webmachine.basho.com/diagram.html), which the Webmachine team helped to augment and improve beyond the original, shows how a Web server or application can examine and analyze the headers of an incoming HTTP request to know how to respond to it appropriately. It accurately and succinctly codifies much of the HTTP 1.1 specification's (RFC 2616) prose. Because of the flowchart's diagrammatic nature, most developers find that not only is it straightforward to follow, but it also makes it easier to understand the HTTP specification's details.

Getting Started with Webmachine

Thanks to some handy helper scripts, getting your first Webmachine application running is trivial. Assuming you've installed Erlang, and you've already downloaded Webmachine following the instructions available at http://webmachine.basho.com/docs.html, the following commands create a new skeleton application for you under the directory `/tmp/skel` and execute the application:

1089-7801/10/$26.00 © 2010 IEEE

```
./scripts/new_webmachine.erl \
   skel /tmp
cd /tmp/skel
make
./start.sh
```

Once the application executes, point your browser to http://localhost:8000/ to display the simple message "Hello, new world." This indicates that the Webmachine Web server is running and can direct requests for the "/" resource to the generated skeleton code. The specific part of the generated code that provides the response message is

```
to_html(ReqData, State) ->
    {"<html><body>" ++
     "Hello, new world" ++
     "</body></html>",
     ReqData, State}.
```

Even if you're not an Erlang programmer, it's not too hard to figure out the purpose of this function. From the incoming HTTP request headers — specifically, the `Accept` header — Webmachine determines that the client is requesting an HTML representation of the "/" resource. To obtain that representation, Webmachine invokes the `to_html` function of the code implementing that resource, then returns the resulting representation to the client.

Alternatively, we could send a request with an `Accept` header for some media type other than HTML. Here, we send such a request using `curl`:

```
$ curl -D /dev/tty -H 'Accept:
   image/jpeg'
   http://localhost:8000/
HTTP/1.1 406 Not Acceptable
Server: MochiWeb/1.1
   WebMachine/1.5.2
Date: Thu, 21 Jan 2010
   04:56:47 GMT
Content-Length: 0
```

This time, we receive HTTP response code "406 Not Acceptable," which means the server was unable to supply a representation of the requested resource fulfilling the requested media type. In this case, Webmachine checked the `Accept` header against the resource's media types, found no match, and returned the error response.

This example focuses solely on processing just the HTTP `Accept` header and so glosses over several other useful details. Plus, it uses simplified generated code in place of the code a developer would normally have to write for a real application. Even so, this simple example illustrates Webmachine's fundamentals: it applies the rules of HTTP 1.1 to each incoming request as the basis for working with your Web application to produce the most suitable response for each request.

As we work through the RESTful Web services development checklist and evaluate Webmachine against it, it's important to keep in mind that the decision-flow diagram to which we referred earlier isn't just a novelty, but rather shows the explicit codepath Webmachine takes to process requests and handle resources, representations, methods, status codes, and other RESTful Web service concerns.

Identifiers, Resources, and Applications

RESTful Web service applications use URIs to identify their resources. To their clients, these URIs are opaque identifiers, but to the service applications themselves, URIs act similarly to keys the services can use to associate implementation artifacts with the respective Web resources they implement. For example, a Web service dealing with orders might create URIs with paths of the form `/orders/<order ID>`, in which "`<order ID>`" is some sort of identifier that lets the service find specific order details in a database. When a Web service receives an HTTP request, it typically breaks the URI for the target resource into its components and uses path elements to help find the right function or object to which it can dispatch the request.

Webmachine provides a straightforward yet powerful approach for service applications to specify URI-based dispatching. Applications provide a dispatch map, which is a list of 3-tuples. Such a 3-tuple might appear as follows:

```
{["portal", "web", "internet",
   "home"],
  internet_computing_resource,
  []}
```

Each 3-tuple consists of the following items:

- *A pathspec*. This is a list of URI path components called *pathterms* split along the '/' characters in the URI. For example, given the URI path `/portal/web/internet/home`, its equivalent pathspec would consist of the pathterms "`portal`," "`web`," "`internet`," and "`home`," as shown earlier. Pathterms could be strings, as shown here, or Erlang atoms, including the atom `'*'`, which serves as a wildcard.
- *A Webmachine resource*. This is an Erlang atom identifying a module. The module exports functions that inform Webmachine about how to construct certain HTTP headers for the resource, what content types the resource supports, and provide other information pertinent to the resource this module represents.
- *A list of arguments*. Before Webmachine asks a resource module to handle a request, it invokes its `init` function and passes to it this third element of the 3-tuple. This list can be empty.

When Webmachine receives a cli-

ent's request, it iterates through the dispatch map and dispatches the request to the first resource whose pathspec matches the request URI. The resource pathspec's string pathterms must match literally, whereas atom pathterms match any single URI path component. The special `'*'` atom matches any number of path components but only at the URI's tail. Webmachine passes all atom matches into the chosen resource as part of the dispatch and includes any query-string data from the URI as well.

Webmachine's approach to URI dispatching is both succinct and highly flexible. Applications can easily use a single wildcard atom to have all URIs dispatched to a single resource implementation, can choose to specify a fixed set of pathspecs and associated resources using only string pathterms to enforce exact matches, or can take the middle ground and use a mixture of atoms and strings. They can also order the pathspecs in their dispatch map so that certain resources will match before others. The end result is that Webmachine lets developers define dispatching rules briefly and precisely.

Representations and Media Types

Exchanging representations of resource states between server and client is a fundamental REST tenet. Within a Web framework, resources are represented in terms of a programming language, typically the one used to implement the framework itself or as data within some form of database. To support RESTful services, frameworks must, at a minimum, enable resource implementations to convert their state to the media types each client declares to be acceptable.

To tell Webmachine what content types it supports, a resource module implements a function named `content_types_provided`. Webmachine expects this function to return a list of pairs (a "property list" in Erlang terminology) in which each pair consists of a media type name and a resource module function to handle that media type. We provided an example of a media type handler function earlier when we described the `to_html` function in the generated Webmachine demo code.

A resource module's `content_types_provided` function lets Webmachine handle HTTP content negotiation properly. If a client provides an `Accept` header in a request that contains at least one of the media types the target resource module's `content_types_provided` function provides, Webmachine will invoke the resource's media type handler function corresponding to that media type to let the resource provide a representation of its state in a form acceptable to the client.

Clients can specify a number of media types in an `Accept` header and can also supply varying *q* values in the header to specify which types it prefers over others. Webmachine implements the HTTP rules for handling multitype `Accept` headers, negotiating between what the client requests and what the resource provides and thereby ensuring the content type returned to the client is the best possible match. If no content-type overlap exists between client and resource, Webmachine returns a "406 Not Acceptable" HTTP status code (see the HTTP decision flowchart, coordinates C7). We saw an example of this earlier when we asked our demo resource for a JPEG representation, and the resource itself provided only the HTML content type. Resources declare the media types they support, and Webmachine handles the rest.

HTTP Methods

`GET`, `PUT`, `POST`, and `DELETE` are HTTP's four basic methods, and `HEAD` and `OPTIONS` can be useful as well. Simple stand-alone static resources, such as files and images, normally support only `GET` and `HEAD`, but RESTful services usually involve multiple resources and so use all the core HTTP methods.

Webmachine makes it trivial for a resource module to indicate the HTTP methods it supports: the module need only export an `allowed_methods` function that returns a list of supported method names. If a client sends a request to a resource with a method not on the list, Webmachine returns the HTTP status code "405 Method Not Allowed" (see the HTTP decision flowchart, coordinates B10).

The core element of different behaviors is the resource, not the method, so the various resource functions that Webmachine invokes aren't generally separated by method.

If a client wants to find out what methods a resource supports, it can send an `OPTIONS` request. If a resource supports the `OPTIONS` method, Webmachine expects it to export an `options` function that returns a list of response headers. An `OPTIONS` response normally includes an `Allow` header to indicate the methods a resource supports, so developers should make sure their `options` function's return values always include that header set to an appropriate value.

Due to the general nature of the

HTTP POST method, it often requires special attention within a Web services framework. Some resources treat it as a way to create new resources, whereas others treat it as a general request-processing method. A resource that supports POST as a resource-creation method implements the post_is_create function to return a true value; this causes Webmachine to invoke the resource's create_path function to supply the new URI for the new resource (see the HTTP decision flowchart, coordinates P11). On the other hand, if post_is_create returns a false value, which is the default, Webmachine invokes the resource's process_post function to take care of the POST request. Similarly, the delete_resource and delete_completed functions let resources control whether they support the DELETE method and, if so, how deletions are carried out.

Note that Webmachine doesn't require developers to write a function for each supported HTTP method. The core element of different behaviors is the resource, not the method, so the various resource functions that Webmachine invokes aren't generally separated by method. This approach significantly helps developers focus on resources and representations rather than forcing them to treat HTTP methods as being the primary contract between the framework and resource implementations. Indeed, if you follow the decision flow diagram from its entry point at coordinates B14 to the various end states, you find that most HTTP method consideration occurs toward the ends of the various paths.

Conditional GET

A significant portion of the Webmachine decision-flow diagram is dedicated to handling HTTP headers related to conditional requests, which are critical to the general scalability of the Web. Important RESTful properties and constraints such as visibility, statelessness, and self-describing messages let intermediaries accurately cache and serve responses they obtain from origin servers, whereas HTTP conditional headers enable clients and servers to control how and when intermediate responses are cached, validated, and served.

Etag and If-None-Match are examples of conditional HTTP headers. A resource can set a value into an Etag header in a response, for which the value is a hash or other compact representation of the resource's current state. A client receiving such a response can save the Etag header value and later set it into an If-None-Match header in its next request to the same resource; if that value still matches the resource state, the resource can return the HTTP status "304 Not Modified" to indicate that it hasn't changed in the interim (see the HTTP decision flowchart, coordinates L18).

To facilitate conditional requests, resources can provide several functions. The last_modified function lets a resource control the HTTP Last-Modified response header date setting — for example, this is useful with the conditional If-Modified-Since request header. Similarly, the expires function lets a resource control the Expires response header date setting, which allows intermediaries to know how long they can cache a representation the resource returns before it's considered stale and in need of revalidation. The generate_etag header lets a resource calculate a value for storage in the Etag response header.

Webmachine is aptly named, given how it accurately and usefully codifies HTTP rules. That feature alone is valuable because it saves RESTful Web services developers from having to memorize all the details of the HTTP 1.1 specification, which isn't a trivial task. But it also provides a framework interface for resource implementations that's both rich and minimal, making the developer provide only those functions required for a given resource when the sensible defaults won't work. It easily addresses all the items of the RESTful Web services development checklist.

Developers who already understand REST and HTTP will find Webmachine intuitive, whereas those who are still learning REST or HTTP will come to understand them much more thoroughly and correctly with Webmachine's guidance. The Webmachine decision-flow diagram is often the only documentation developers need to successfully implement a resource. Finally, the fact that the framework is implemented in Erlang, a language with a simplicity that belies its unmatched support for highly reliable and scalable systems, makes Webmachine appealing and advantageous.

References

1. S. Vinoski, "RESTful Web Services Development Checklist," *IEEE Internet Computing*, vol. 8, no. 6, 2004, pp. 94–95.
2. D. Bryson and S. Vinoski, "Build Your Next Web Application with Erlang," *IEEE Internet Computing*, vol. 13, no. 4, 2009, pp. 93–96.

Justin Sheehy is the CTO of Basho Technologies, the company behind Webmachine and Riak. His research interests include resilient systems and applying programming language implementation techniques to unusual problems. He performed both undergraduate and graduate studies in computer science at Northeastern University. Contact him at justin@basho.com.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Massachusetts. He's a senior member of the IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and reach him at vinoski@ieee.org.