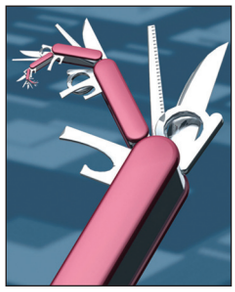

Wriaki, a Webmachine Application

Steve Vinoski • *Basho Technologies*

In the March/April 2010 issue of this column,[1] Justin Sheehy and I evaluated Webmachine, a unique Web toolkit, against a RESTful Web services development checklist. Webmachine is unique because it codifies the HTTP flow diagram originally created by Alan Dean, currently of Adaptavis, and later enhanced by Dean and Sheehy (see http://webmachine.basho.com/diagram.html). Webmachine systematically applies HTTP semantics to Web resources, unlike most Web frameworks, which tend to focus on how to make HTTP conveniently accessible from a particular programming language. Because of its strict conformance to how HTTP works, it's unsurprising that Sheehy and I found that Webmachine fulfilled the RESTful development checklist with ease.

In this column, I explore a Webmachine wiki application called Wriaki. I show that the wiki is pretty easy to implement given everything Webmachine handles on its behalf at the Web interface. Both Wriaki and Webmachine are implemented in Erlang, which is what this discussion focuses on, but Webmachine has also been ported to Python, Ruby, Clojure, and Agda, so the ideas presented here are applicable beyond Erlang.

Running Wriaki

Bryan Fink of Basho Technologies originally wrote Wriaki as a sample application for Riak, an open source key-value database also from Basho. You can get the source code for Wriaki from github.com by cloning git://github.com/basho/wriaki.git. As its documentation explains, Wriaki relies on several other packages:

- The Erlang programming language, which you can obtain from www.erlang.org.
- Riak, which you can get from http://downloads.basho.com/riak/ or directly from github.com by cloning git://github.com/basho/riak.git.
- Genshi, a Python library for generating Web output. You can obtain it by running "`easy_install Genshi`" in your shell or by visiting http://genshi.edgewall.org.
- Creoleparser, a Python library for converting Creole wiki markup to Web output. You can obtain it by running "`easy_install Creoleparser`" in your shell or by visiting http://code.google.com/p/creoleparser/. Creoleparser depends on Genshi.

As you might guess from the nature of these packages, Wriaki is largely an integration project. It relies on Webmachine to make its wiki pages act as proper Web resources, Creoleparser for wiki markup support, and Riak for persistent storage.

Once you've obtained and installed these packages, you can download and build Wriaki by running the following commands in your shell:

```
git clone\
  git://github.com/basho/wriaki.git
cd wriaki
make rel
```

These commands create an operational Wriaki wiki under a directory named `rel/wriaki`. By default, Wriaki's Web server listens on all network interfaces on port 8000, but you can change these and other configuration options in the file `rel/wriaki/etc/app.config`. The default configuration works without changes for most users.

To run Wriaki, first make sure Riak is running (see http://wiki.basho.com/Basic-Cluster-Setup.html)

Published by the IEEE Computer Society 1089-7801/12/$31.00 © 2012 IEEE

and then execute the following shell command:

```
./rel/wriaki/bin/wriaki start
```

This runs Wriaki in the background. If you're familiar with Erlang and would prefer to start Wriaki with an Erlang shell attached, substitute "`console`" for "`start`" in this shell command.

Resources and Dispatching

A wiki is, of course, a collection of versioned user-editable pages. The Wriaki Web interface comprises several Web resources that implement the wiki:

- *User resources* represent wiki users.
- *Articles* represent wiki pages.
- *Archives* represent individual versions of wiki pages.
- A *history object* contains an entire set of versions for a wiki page.
- *Sessions* track user logins.

These resources are tied into Webmachine through a dispatch map, which is an Erlang list consisting of one or more `{pathspec, resource, args}` tuples. When a request arrives, Webmachine consults its dispatch map and dispatches the request to the first resource whose pathspec matches the request URL. If it finds no match, Webmachine returns HTTP status code 404 indicating that the resource wasn't found.

A pathspec indicates the URL path for the resource; it's composed of a list of pathterms that are either strings, atoms, or the special atom `'*'`. Webmachine performs pathspec matching by splitting the URL path at its "/" characters to produce a list of path components. As Webmachine walks the dispatch map, it attempts to match each URL component to a pathterm in each pathspec. There are three pathterm types:

- A string pathterm indicates a fixed component of the URL path. For example, a Web resource at the URL path `/a/b` could have the pathspec `["a", "b"]`.
- An atom pathterm creates a named match for any path component. For example, a URL path `/a/b` could match the pathspec `["a", second]`, letting the resource implementation look up the "b" part of the path at runtime via the name `second`.
- A star pathterm (`'*'`), which can be used only as the last component of a pathspec, indicates a wildcard that can match one or more path components. For example, given the path `/a/b/c/d` and the pathspec `["a", "b", '*']`, the "a" and "b" match literally and the star matches the remaining components, making them programmatically accessible to the resource implementation.

```
{["wiki"],                redirect_resource,     "/wiki/Welcome"}.
{["wiki",'*'],            wiki_resource,         []}.
{[],                      redirect_resource,     "/wiki/Welcome"}.

{["user"],                login_form_resource,   []}.
{["user",name],           user_resource,         []}.
{["user",name,session],   session_resource,      []}.

{["static",'*'],          static_resource,       "www"}.
```

Figure 1. The dispatch map for Wriaki. Each line contains a pathspec, the name of a resource module that serves requests on URLs matching the pathspec, and a list of arguments passed to the resource module's `init/1` function.

Webmachine considers a whole pathspec to match a URL path when all pathterms in the pathspec are used and all URL path components are matched.

Besides the pathspec, a dispatch map tuple contains two other elements: the `resource` element, which indicates the name of the Erlang module implementing the resource, and the `args` element, which is a list of arguments that will be passed to the resource module's `init/1` function. (In Erlang, references to functions are normally written in the form "function/arity," where "function" is the name of the function and "arity" is a number indicating the number of arguments the function requires.) Figure 1 shows the dispatch map for Wriaki, found in the file `apps/wriaki/priv/dispatch.conf`.

The dispatch map shows that the Wriaki URL space is supported with six Erlang modules. Webmachine expects each resource module to supply functions it can invoke as appropriate as it guides incoming requests through its HTTP flow diagram. All resource functions have the same signature — each takes two arguments, request data and resource process state, and returns a 3-tuple consisting of a return value, response data, and resource process state. To return specific response data, a resource function augments a copy of its request data argument to include the response data, and returns the augmented request data as the second element of its returned 3-tuple. (A copy is required because Erlang enforces immutability of variables. Erlang ensures that such copies are inexpensive by sharing as much of the original value as possible with the new copy.) Similarly, if a resource function modifies the process state of a resource, it might note the new state in a new instance of its resource process state argument and return it as the third element of the return value. A resource function is also free to simply pass its request data and resource process state unmodified in its return value.

Because requiring every resource module to implement every possible resource function would make it tedious and error-prone for developers to implement resources, Webmachine supplies reasonable defaults for all resource functions. This means each resource module need supply only those functions it must specialize based on its resource's specific Web characteristics. In the next few sections, I examine some of Wriaki's resource modules and the resource functions they provide.

Redirect

Wriaki implements a redirect resource in the Erlang source file `apps/wriaki/src/redirect_resource.erl`. As the dispatch map shows, this resource is used for the paths / and `/wiki`, and each pathspec includes the string argument `"/wiki/Welcome"` to be passed to the `redirect_resource:init/1` function:

```
init(Target) ->
    {ok, Target}.
```

The `redirect_resource:init/1` function simply returns its single argument as the second element of a 2-tuple indicating successful resource initialization. Webmachine treats the second element of the tuple as the resource's process state, storing it on the resource's behalf and later passing it along to other resource functions it invokes as part of the same request handling process. In this case, `Target` is simply the string argument from the dispatch map. This value figures prominently in the resource's `moved_permanently/2` function:

```
moved_permanently(RD, Target) ->
    {{true, Target}, RD, Target}.
```

"Moved permanently" corresponds to HTTP status code 301, which in turn corresponds to location K5 in the Webmachine decision flow diagram. By supplying this resource function, the `redirect_resource` implementation redirects any incoming requests to `Target`, which, as explained earlier, is specified in the dispatch path as the URL path `/wiki/Welcome`. Together, these dispatch path entries and the `redirect_resource` implementation alias the paths / and `/wiki` to the path `/wiki/Welcome`. The end effect of matching either path in the dispatch map is a redirection of the request over to the `wiki_resource` module.

Wiki Resource

The `wiki_resource` module, found in the file `apps/wriaki/src/wiki_resource.erl`, implements Wriaki's wiki pages. Its `init/1` function returns a record instance for its process state:

```
init([]) ->
    {ok, Client} = wrc:connect(),
    {ok, #ctx{client = Client}}.
```

This function connects to the Riak database and stores the database handle into an instance of the `ctx` record defined locally within the module. In addition to the `client` field for the database connection, the `ctx` record has fields for tracking a user editing a wiki page, as well as version information for the page being viewed.

The `wiki_resource` module also implements an `allowed_methods` resource function:

```
allowed_methods(RD, Ctx) ->
   {['HEAD','GET','POST','PUT'],
     RD, Ctx}.
```

This function informs Webmachine that the wiki resource accepts four HTTP methods — `HEAD`, `GET`, `POST`, and `PUT` — indicating the read/write nature of the wiki pages. By default, a Webmachine resource is read-only, accepting only `GET` and `HEAD` requests. The resource need only declare what methods it supports; should a client invoke a method on a resource that isn't allowed, Webmachine automatically returns HTTP status 405, which means "method not allowed," to the client (see location B10 of the Webmachine HTTP flow diagram).

To store a new wiki page, the `wiki_resource` module supplies two functions: `content_types_accepted/2` and `accept_form/2`. The first function tells Webmachine what content types the resource is willing to accept, whereas the second is the function Webmachine invokes when content of the given type is sent to the resource:

```
content_types_accepted
  (RD, Ctx) ->
    MT = "application/x-www-
    form-urlencoded",
    {[{MT, accept_form}], RD,
    Ctx}.
```

The `content_types_accepted/2` function tells Webmachine that wiki pages accept content with the MIME type "application/x-www-form-urlencoded," which is the default content type for HTML forms. If content of this type is posted to a wiki page due to a user creating a new page or editing a page, the `content_types_accepted/2` function return value tells Webmachine to process the content by calling the `accept_form/2` function. Should a client try to send a content type the resource doesn't declare as an acceptable type, Webmachine automatically returns HTTP status 415, meaning "unsupported media type," to the client (see location B5 of the Webmachine HTTP flow diagram).

Figure 2 shows just how straightforward the `accept_form/2` function is. It extracts the article text from the `RD` request data argument, stores the article in the page's archive and history, and finally stores the article itself in Riak. The function returns `true` to inform Webmachine it successfully accepted the content.

Because a wiki page that can't be viewed isn't very useful, the

wiki_resource module also supplies a to_html/2 function. Webmachine's default version of the content_types_provided/2 resource function returns [{"text/html", to_html}], indicating that resources support the "text/html" MIME type via a to_html/2 function. Given that wiki_resource doesn't supply its own content_types_provided/2 function, Webmachine's default applies, but wiki_resource supplies its own to_html/2 function that can render wiki pages in HTML.

Users

A wiki can't exist unless it has users to create, update, and delete pages. The implementation of Wriaki's user resource primarily tracks user details and makes authorization decisions. The user resource is implemented in the apps/wriaki/src/user_resource.erl module. The user_resource module supplies init/1, allowed_methods/2, and content_types_accepted/2 functions that look very much like their counterparts described earlier for the wiki_resource module. It also supplies an accept_form/2 function, but it's implemented much differently than in wiki_resource (see Figure 3).

Here, accept_form/2 has two clauses (separated by the semicolon). The first is invoked when the user field of the user_resource ctx process state record (which is a different type than the wiki_resource ctx record described earlier) equals notfound, indicating that a new user is registering. The function creates a new user and stores it in the database, starts a new session, and passes this information to the second function clause. The second function clause, which Webmachine calls when users update their wiki registration information, parses the form data, updates the user's details in the database, and returns true to inform Webmachine that the user form was accepted successfully.

```
accept_form(RD, Ctx=#ctx{client=Client}) ->
    Article = article_from_rd(RD, Ctx),

    %% store archive version
    ok = wrc:put(Client, article:create_archive(Article)),

    %% update history
    ok = article_history:add_version(Client, Article),

    %% store object
    ok = wrc:put(Client, Article),

    {true, RD, Ctx}.
```

Figure 2. The accept_form/2 function for Wriaki's wiki resource. When a user finishes adding or updating a wiki page, this function extracts the submitted page content, stores it, and updates the history of the target wiki page.

```
accept_form(RD, Ctx=#ctx{user=notfound}) ->
    %% register new user
    User = wuser:create(username(RD), []),
    {AuthRD, Auth} = wriaki_auth:start_session(RD, User),
    accept_form(AuthRD, Ctx#ctx{user=User, auth=Auth});
accept_form(RD, Ctx=#ctx{user=User, client=C}) ->
    {ok, Client} = wrc:set_client_id(C, wobj:key(User)),
    ReqProps = mochiweb_util:parse_qs(wrq:req_body(RD)),
    ModUser = update_password(
                update_bio(
                  update_email(User, ReqProps),
                  ReqProps),
                ReqProps),
    ok = wrc:put(Client, ModUser),
    {true, RD, Ctx#ctx{client=Client, user=ModUser}}.
```

Figure 3. The accept_form/2 function for Wriaki's user resource. It handles registration of new users and updates to registered user information.

To verify authorization, user_resource also supplies an is_authorized/2 function, which checks to see if a user is viewing his or her own user page. If so, the user can edit the information on the page; if not, the wiki renders the user page for viewing only.

Simplicity

If you take the time to study the Wriaki implementation, you'll be struck by its elegance and simplicity, especially given that it implements a fully functional wiki. There are several reasons for its simplicity:

- As we've seen from the redirect_resource, wiki_resource, and user_resource Wriaki modules, implementing a Webmachine resource is just a matter of identifying the resource functions the resource must provide to implement its desired behavior. Because Webmachine encapsulates and handles HTTP, the Wriaki source code contains very little direct HTTP knowledge.
- The fact that Webmachine supplies defaults for all resource functions greatly simplifies resource modules, letting them implement only

those functions for which Webmachine's defaults don't apply. The fact that all resource functions have the same signature also makes them easy for developers to understand.

- Wriaki gets substantial mileage out of using the Riak database and other packages on which it depends. It encapsulates these facilities in easy-to-use modules that expose only those utilities required to implement the wiki. The bulk of the implementation of the wiki itself consists of integrating calls to these other packages as needed.
- By following Erlang's "let it crash" philosophy, Wriaki avoids numerous lines of defensive error-handling code. When you read the Wriaki source, it tells you very clearly what should happen, assuming everything works as expected. Should something go wrong, Erlang's process-supervision capabilities kick in, allowing only the Web request that caused the problem to die, without affecting any other concurrent requests. The fact that Wriaki is built over the reliable and highly available Riak database helps here as well.
- In general, Erlang code tends to be compact and easy to read because the language itself is relatively simple. It has comparatively few language elements, and its syntax, though not derived from the C family of languages, is straightforward and consistent.

As this exploration of Wriaki shows, Webmachine is unique, unlike any other HTTP toolkit you've used. It handles all the intricacies of HTTP on behalf of your application, such that your resources wind up being good Web citizens, and its collection of resource functions helps minimize the amount of resource implementation code you have to write. Whether you're an Erlang expert or are new to the language, you'll find that Webmachine can greatly ease the development of HTTP-savvy Web applications.

Acknowledgments

Thanks to my teammates Bryan Fink and Justin Sheehy of Basho Technologies for their reviews of a draft of this column.

Reference

1. J. Sheehy and S. Vinoski, "Developing RESTful Web Services with Webmachine," *IEEE Internet Computing*, vol. 14, no. 2, 2010, pp. 89–92.

Steve Vinoski is an architect at Basho Technologies in Cambridge, Mass. He's a senior member of IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and contact him at vinoski@ieee.org or on Twitter at @stevevinoski.