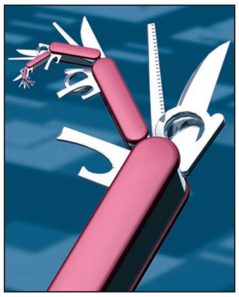
Build Your Next Web Application with Erlang



Dave Bryson
Steve Vinoski • *Verivue*

The Erlang programming language, together with its Open Telecom Platform (OTP) framework, is renowned for its exceptional concurrency and fault-tolerance capabilities. Developers are often initially attracted to Erlang just so they can try out these features, and experienced Erlang server application developers make heavy use of these and other Erlang traits. Successful Erlang server applications often must be Web-accessible, even if not initially designed that way, which means developers must integrate such services into the world of HTTP.

Fortunately, building a Web application in Erlang isn't necessarily difficult; it mostly requires a change in thinking. For a small investment in learning, your Web application can take advantage of all the exceptional features Erlang provides to non-Web applications: scalability, fault tolerance, concurrency, relatively painless distributed system capabilities, and live system upgrades.

The growing interest in applying Erlang/OTP to Web services and Web applications is driving the development of several interesting open source projects. In this column, we'll look at some of the more popular Erlang Web frameworks and Web servers.

Web Servers

One of the best known Erlang Web servers is Yaws (*yet another Web server*; http://yaws.hyber.org/), written and maintained by longtime Erlang expert Claes "Klacke" Wikström, currently of Tail-f Systems. Klacke has contributed significantly to Erlang over the years, devising and implementing several important features, including Erlang term storage (ets), Distributed Erlang, the eprof profiler, the Mnesia database, the Erlang bit syntax, and more.

Yaws is a general-purpose, open source Web server that supports a number of common use cases. At the simple end of the spectrum, a normal Yaws deployment can serve static files from disk without requiring any developer code or extensions. However, Yaws shines when it comes to serving dynamic content, and getting it to do that means writing some code. Here's an example of a simple handler that returns "hello from yaws" with a content type of "text/plain" and a 200 HTTP status code:

```
out(_Arg) ->
  {content,
   "text/plain",
   "hello from yaws"}.
```

Yaws allows `out()` functions like this one — normally referred to in Erlang parlance as `out/1`, where the "1" denotes the function's arity — to appear in several contexts. For example, developers can place `out/1` functions within `<erl>...</erl>` tags in an HTML representation. When serving such a representation, which it expects to find in a `.yaws` file, Yaws first invokes each `out/1` function and replaces its enclosing `<erl>` tags with the function's return value. The argument to the `out/1` function is an instance of a Yaws `arg` record, which encapsulates all information about the incoming HTTP request, including HTTP headers, the target URI, the HTTP method invoked by the client, and incoming request data (if applicable).

For applications that serve dynamic content other than HTML, Yaws also lets `out/1` functions appear in application modules, or *appmods*. An appmod — a regular Erlang module that exports an `out/1` function — lets an application handle requests for target URIs of its choosing by registering individual appmods onto base URIs. For example, if a developer registers an `orders` module onto the URI path `/orders`, Yaws will dispatch any request for any URI starting with /orders to the orders:out/1 function for processing.

Yaws also lets developers use appmods to integrate whole Erlang applications into the Web server via its Yaws applications or *yapps* functionality. An appmod executes within the context of Yaws, whereas a yapp encapsulates one or more appmods into an application separate from the Yaws application. Using multiple applications lets you manage them independently at runtime, which allows for separate deployment and for application-specific restart strategies and code upgrades.

Unfortunately, our column space doesn't permit a full description of Yaws, which has gained a good deal of flexibility and options for Web service development over its life span of seven years and counting. For more information, please see Steve's 2008 article "RESTful Services with Erlang and Yaws" (www.infoq.com/articles/vinoski-erlang-rest).

Mochiweb (http://code.google.com/p/mochiweb) is an open source, lightweight, and fast HTTP server developed by Mochi Media to drive many of their services. It provides a small and simple API that gives you complete control over how you handle the HTTP request and response. Here's an example of a very simple application that starts a server on port 8080 and returns "hello from mochiweb" with a 200 status code:

```
start() ->
  mochiweb_http:start(
   [{port,8080},
    {loop,{?MODULE,loop}}]).

loop(Req) ->
  Req:ok({"text/plain",
  <<"hello from mochiweb">>}).
```

Here, the start/0 function starts the server on port 8080 and tells Mochiweb to pass each request to the loop/1 function. The loop/1 function in turn calls the Mochiweb Req request module's ok() function, passing in "text/plain" as the content type, and "hello from mochiweb" as the response message. The Req:ok/1 function automatically sets the HTTP status for the response to 200.

Mochiweb is built around the OTP framework principles and comes with a script to generate an application structure along with some code for your new application. Some of the pregenerated code includes an Erlang supervisor to monitor the HTTP server and restart it if it fails, as well as an initial module to handle incoming requests. Of course, you're not bound to use this script or structure, so it's easy to embed Mochiweb into existing applications. If you're building a Representational State Transfer (REST) service and use JavaScript Object Notation (JSON) as the message exchange format, Mochiweb includes a module for encoding and decoding Erlang terms into JSON. Overall, Mochiweb is a fast, production-ready HTTP server you can use to quickly Web-enable your Erlang application.

Finally, Erlang comes with a built-in HTTP server called Inets that includes several pluggable modules you can use to extend the server. Developers configure modules via the Inets configuration file, and each incoming HTTP request passes through the modules in the order in which they appear until a module finally returns an HTTP response. Inets provides modules to handle CGI, SSL, user authentication, dynamic pages, URI aliasing, and more. You can also create your own modules by following the guidelines in the Inets Web Server API. Keeping with our running example, here's how you'd implement a server with Inets to return the message "hello from inets:"

```
start() ->
  % Set up the server
  inets:start(),
  inets:start(
   httpd,
   [{port,8001},
   {server_name,"httpd_test"},
   {bind_address,
    "localhost"},
   {server_root,"."},
   {document_root,"."},
   {modules, [?MODULE]}]).

do(_Info) ->
  {proceed,
   [{response,
   {200,
   [<<"hello from inets">>]}}
  ]}.
```

The start/0 function sets up the server and starts it on port 8001. The do/1 function is the required callback you need to implement for the Inets server API. Here, the `do/1` function returns a tuple with a 200 status code and the "hello from inets" response message.

Now that you have a server on which to deploy, let's look at some open source frameworks you can use to develop an Erlang Web application.

Web Frameworks

Erlyweb (http://erlyweb.org) is a popular open source framework built around the model view controller (MVC) pattern. It works with Yaws and provides many of the features you need to build a full-blown, database-driven Web application. You can be up and running quickly with Erlyweb using its built-in ability to generate the application directory structure along with some initial code for a controller and view. Like many MVC-based Web frameworks, you implement your application's flow logic in the controller module and feed the controller output to Erlyweb's built-in template language to help separate the HTML from the business logic.

By default, Erlyweb will automatically route a request to your application using the path elements in the URL. For example, a request sent to http://example.com/foo/bar/1/2/3 would make a call to the `foo` module with the function `bar` and pass the argument list `[1,2,3]` to it. You get all that for free without any configuration. It also provides plenty of hooks you can implement in the controller to alter the request and response as needed. For example, you can implement a "before" filter, often used for authentication.

Erlyweb includes a database-abstraction layer (Erlydb) that users of ActiveRecord, the Ruby on Rails Object-Relational Mapping (ORM) layer, will find familiar. It currently supports the MySQL and Postgres relational databases as well as Erlang's Mnesia database. Erlydb provides access to the database via a standard API with automatically generated functions such as `save`, `find_first`, `update`, and more. Erlyweb uses the metadata from the database along with your code to automatically map your models to the underlying database table. It also provides one-to-many and many-to-many relationships between models by simply declaring the relationships in the source code.

Another open source framework is Erlang Web (www.erlang-web.org), developed and maintained by Francesco Cesarini, a world-class Erlang expert, and his colleagues at Erlang Training and Consulting. Like Erlyweb, it also supports Yaws as well as the native Erlang Inets Web server.

Erlang Web uses an approach to building Web applications that should be familiar to users of Java's Servlet/Java Server Pages (JSP) technology: you construct an application using controllers and templates. The application's data flow is defined in a controller as an Erlang module with functions. You can perform preprocessing on a request by specifying one or more functions that should be executed before the targeted action. Such preprocessing functions might include, for example, checking for authentication and validating incoming request data. Erlang Web also provides a nice dispatcher that lets you customize request mapping to the actual controllers via a configuration file containing regular expressions.

The template engine in Erlang Web is based on XHTML and the Erlang `xmerl` library. It includes a built-in tag library, which lets you construct dynamic pages in parts and reuse chunks of XHTML across the application. You can also perform data validation by defining data structures in Erlang records that could also automatically generate input forms for your application. For persistent storage, Erlang Web has a database management system layer supporting the Mnesia database as well as beta support for the popular CouchDB, an innovative, document-oriented database (http://couchdb.apache.org/).

The growing interest in applying Erlang/OTP is driving the development of several interesting open source projects.

Webmachine (http://bitbucket.org/justin/webmachine) takes a very different and interesting approach to building RESTful Web applications using the Mochiweb server. In Webmachine, you build your application around resources and a set of predefined functions — resource and function are analogous to controller and action in Ruby on Rails.

When a request comes into Webmachine, it automatically flows through a built-in decision path that can examine the request to determine the next processing step. At each step in the flow, Webmachine provides a predefined *hook* — a resource function you can override to implement your application's logic. Each of the predefined resource functions have a sensible default return value that results in an appropriate HTTP status code, so you override only the functions your applications need. For example, when a request comes into Webmachine, it automatically checks the predefined `is_authorized` function. By default, the check returns `true`, and the request continues via the decision path. However, if you override the `is_authorized` function and return anything other than `true`, Webmachine will automatically respond with the

401 (unauthorized) HTTP status code.

Another nice feature of Webmachine is its built-in trace facility. When you enable debugging in your application, you can follow the request via a dynamically generated view you access from a Web browser. The trace provides a decision graph showing the request path via your application. You can further drill down into the trace to see details on the specific functions called and the decision made at that point.

If you prefer an event-driven approach to building Web applications, you'll want to look at Nitrogen (http://nitrogenproject.com/). Nitrogen comes with support for Asynchronous JavaScript and XML (Ajax) and Comet (long-polling HTTP requests) and provides built-in "tags" that make it easy to add Ajax effects and more with as little as one line of code. It also supports all the popular Erlang Web servers: Yaws, Mochiweb, and Inets.

With Nitrogen, each page in your application corresponds to an Erlang module using a simple naming convention. For example, a request to `/web/blog/list` would map to a module named `web_blog_list`. Within a module, you define a `main()` function as the entry point for the request along with one or more event functions for processing actions.

A Nitrogen application's building blocks are "elements" and "actions." An element is nothing more than an Erlang record you use to build the pieces of an HTML page, such as tables, links, `div` tags, and more. Behind the scenes, the Nitrogen engine transforms the Erlang records to HTML. Nitrogen contains a rich set of elements that cover most common needs. In addition, you're free to create custom elements by simply defining a new record and module for the tag.

You can bind an element to an action to build dynamic pages and use an action to capture an interaction with an element — link click, mouseover, and so on. Nitrogen wraps the JavaScript behind an action using the jQuery library (http://jquery.com/) and provides many common popular element effects, such as toggle, fade, animate, and more. To handle actions such as sending information to the server, you simply bind a page element to an Erlang function. Nitrogen will handle mapping the request to your function using the power of pattern matching.

Finally, BeepBeep (http://github.com/davebryson/beepbeep/tree/master) is a small framework still in its early developmental stages. It's designed to provide a "convention-over-configuration" approach to building Web applications similar to that of Ruby on Rails. If you follow the code structure layout and a few rules when building your application, BeepBeep will automatically map URI requests to your modules and templates. BeepBeep is built around the Mochiweb Web server and provides dynamic pages using the Django template language (http://www.djangoproject.com) from the world of Python.

These Erlang Web servers and frameworks cover a broad and useful portion of the Web development spectrum. The fact that Erlang lets developers express solid solutions quite compactly means that you can have a service or application up and running with these servers and frameworks in very little time and with minimal effort. Developers who make the effort to learn Erlang/OTP not only find its focus on pragmatism refreshing, but they're also rewarded with systems that maximize the benefits of multicore systems and concurrency, so they scale and perform well. The brevity of the code implementing these systems also makes them relatively easy to extend and maintain. Plus, they can exploit Erlang/OTP's hot code-loading features to stay up and running even as they're being upgraded.

As you can see, you have plenty of choices for building and deploying your next Web application using Erlang. Future columns will explore some of these servers and frameworks in more detail, starting with Webmachine, which will receive a much more thorough treatment in the next issue.

Dave Bryson is an experienced Web application developer. You can read Bryson's blog at http://weblog.miceda.org, follow his code at http://github.com/davebryson, and contact him at daveb@miceda.org.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Mass. 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 contact him at vinoski@ieee.org.

CISCO

Cisco Systems, Inc. is accepting resumes for the following position in:

Irvine, CA

Test Engineer

(Ref#: IRV2)

Build test equipment and test diagnostics for new products based on manufacturing designs.

Please mail resumes with reference number to Cisco Systems, Inc., Attn: J51W, 170 W. Tasman Drive, Mail Stop: SJC 5/1/4, San Jose, CA 95134. No phone calls please. Must be legally authorized to work in the U.S. without sponsorship. EOE.

www.cisco.com