Building RESTful Web Services with Erlang and Yaws

Steve Vinoski
Member of Technical Staff
Verivue, Inc., Westford, MA USA
http://steve.vinoski.net/
QCon San Francisco
20 November 2008
Erlang

- Functional programming language created in 1986 at Ericsson
- Focuses on long-running, concurrent, distributed, highly reliable systems
- Small language that enables big productivity
Yaws

- “Yet Another Web Server” implemented starting in early 2002 by Claes “Klacke” Wikström, long-time Erlang expert

- Perhaps best known outside the Erlang community for the “Apache vs. Yaws” graphs

- Excellent for serving dynamic content

- Can run stand-alone or embedded within a larger Erlang application

- [http://yaws.hyber.org/](http://yaws.hyber.org/)
Apache vs. Yaws

- Yaws (in red) vs. Apache (green and blue)
- X axis: number of connections
- Y axis: throughput (kB/sec)

Find details of the experiment at http://www.sics.se/~joe/apachevsyaws.html
Topics

- Trying to cover Erlang, Yaws, and REST in depth in an hour doesn’t work (I’ve tried)
- Instead:
  - explain general Yaws capabilities
  - cover several areas to focus on when building RESTful web services
  - describe how to implement each of those areas using Yaws and Erlang
One way is to embed Erlang code in `<erl> ... </erl>` tags in your HTML:

```html
<html>
  <body>
    <p>
      <erl>
        out(Arg) ->
          {html, "Hello, World!"}.
      </erl>
    </p>
  </body>
</html>
```

Place this into a “.yaws” file and Yaws calls “out” which generates HTML to replace `<erl> ... </erl>`
Yaws calls application “out” functions in various contexts to produce dynamic content.

- written as “out/1” in Erlang notation, since “out” takes 1 argument

- The argument to “out” is an “arg” record
  - supplies access to all details of the incoming request — URI, methods, HTTP headers, etc.

- Depending on the calling context, out/1 returns either part or all of the response
Ehtml

- Returning HTML-formatted strings from out/1 is painful
- Embedded tags can get messy
- Yaws provides ehtml as a better alternative
  - Essentially HTML in Erlang syntax
  - Tuple consisting of the atom `ehtml` and a list of HTML elements
Ehtml Example

- `{ehtml, list-of-tags}`

- `list-of-tags:
  [{html, [], [{body, [], [{p, [], "Hello, World!"}]}]}]`

- Rewrite the previous `<erl> ... </erl>` example:

```erl
<erl>
out(Arg) ->
  {ehtml,
   [{html, [], [{body, [], [{p, [], "Hello, World!"}]}]}]}
</erl>
```
A Yaws appmod ("application module") is an Erlang module that:

- exports an out/1 function
- is tied into one or more URI paths

When it encounters a path element with an associated appmod, Yaws calls the appmod out/1 function to process the rest of the URI

Appmods are specified in the Yaws config file
Appmod Example

First set the appmod configuration in yaws.conf:

```
<server test>
    port = 8000
    listen = 127.0.0.1
    docroot = /usr/local/var/yaws/www
    appmods = <foo, foo>
</server>
```
-module(foo).
-export([out/1]).
-include("yaws_api.hrl").

out(Arg) ->
  {ehtml,
   [{'html', []},
    [{'body', []},
     [{'h1', []}, "Appmod Data"],
      {'p', []},
       yaws_api:f("appmoddata = ~s", [Arg#arg.appmoddata])],
      {'p', []},
       yaws_api:f("appmod_prepath = ~s", [Arg#arg.appmod_prepath])]}].
Invoking appmod foo

- Results of running curl http://localhost:8000/tmp/foo/bar/baz/

  ```html
  <html>
  <body>
  <h1>Appmod Data</h1>
  <p>appmoddata = bar/baz/</p>
  <p>appmod_prepath = /tmp/</p>
  </body>
  </html>
  ```

- Appmod prepath is /tmp/, appmod data is bar/baz/

- Could also access the rest of Arg to get query
Yapps

- Yapps — “yaws applications”

- Makes use of full Erlang/OTP application design principles for supervision, auto-restart, etc.

  - Yapps reside in the same Erlang VM instance with the Yaws application

- Yapps are tied to URIs like appmods, and they also have appmods under them

  - appmod: just a module

  - yapp: application comprising multiple modules, some of which are appmods
Yapp Framework

- The Yapp application itself is an optional framework under Yaws which manages user yapps.

- By default it persistently stores registrations for user yapps in mnesia (Erlang’s distributed fault-tolerant datastore).

- Easy to replace the mnesia default (e.g., I use an in-memory registry with bootstrapped yapps).

- For details on installing and using yapps, see http://yaws.hyber.org/yapp_intro.yaws
Focus Areas for RESTful Services

- Resources and identifiers
- Representations and media types
- Hypermedia and linking
- HTTP Methods
- Conditional GET
Dealing with URIs

- Some advise spending time designing “nice” URIs, some argue against it

- Arguments against say it doesn’t matter because with proper use of hypermedia, clients don’t care

- But I argue for good URI design because it affects your server implementation

- We’ve seen how appmods and yapps allow us to take over URI processing
Sidebar: Erlang Pattern Matching

- Erlang allows you to overload functions based on matching function arity and argument values.

- For example, in raising a value \( N \) to a power \( M \), we end the recursion with a version of the \( \text{pow}/3 \) function for which \( M == 0 \):

\[
\text{pow}(N, M) \rightarrow \text{pow}(N, M, 1).
\]
\[
\text{pow}(_N, 0, \text{Total}) \rightarrow \text{Total};
\]
\[
\text{pow}(N, M, \text{Total}) \rightarrow \text{pow}(N, M-1, \text{Total}*N).
\]
Handling URIs with Pattern Matching

Consider this out/1 function:

```erlang
out(Arg) ->
    Uri = yaws_api:request_url(Arg),
    Uri_path = Uri#url.path,
    Path = string:tokens(Uri_path, "/"),
    out(Arg, Uri, Path).
```

- Breaks the target URI path into a list of path elements
- Invokes a different function, out/3, with more detail and returns its result
- pass Arg and Uri for further access in called function
Handling URIs with Pattern Matching

out/3 might look like this:

\[\begin{align*}
&\text{out}(\text{Arg}, \text{Uri}, ["order"]) \rightarrow \\
&\quad \text{%% handles path order/;} \\
&\text{out}(\text{Arg}, \text{Uri}, ["order", \text{Order_id}]) \rightarrow \\
&\quad \text{%% handles path order/{order_id}/;} \\
&\text{out}(\text{Arg}, \text{Uri}, ["customer", \text{Cust_id}]) \rightarrow \\
&\quad \text{%% handles path customer/{cust_id}/} \\
&\text{out}(\text{Arg}, \text{Uri}, \_\_\_\_\_\_\_) \rightarrow \\
&\quad \{\text{status, -404}\}. \\
\end{align*}\]

Pattern-matching the URI path list lets us dispatch to specific handlers for each URI path.
Designing URIs

- So, yes, I would argue that you do want to design your URIs well if possible.
- Doing so allows you to make use of Erlang’s pattern-matching feature to assist with URI processing and dispatching.
- Can be combined with appmods as necessary to split processing and dispatching across different modules.
Focus Areas for RESTful Services

- Resources and identifiers
- Representations and media types
- Hypermedia and linking
- HTTP Methods
- Conditional GET
Representations and Media Types

- Each resource can have one or more representations
- Representation types are indicated by MIME types in the Content-type HTTP header
- Clients can negotiate content types by sending preferred types in Accept headers
  - preferences can be indicated using quality ("q") parameters
Example Accept Headers

- Safari 3.2:
  text/xml, application/xml, application/xhtml+xml, text/html; q=0.9, text/plain; q=0.8, image/png, */*; q=0.5

- Firefox 3.0.4:
  text/html, application/xhtml+xml, application/xml; q=0.9, */*; q=0.8

- IE 7.0.5730.13:
  image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*

- curl: */*
Service Clients and Accept Headers*

- As we can see, browser Accept headers tend to be long strings that contain so many options they’re almost meaningless.

- Web service client Accept headers are not like this:
  - they tend to either ask for exactly what they want...
  - ...or they don’t send an Accept header at all.

* this slide was added after the QCon presentation based on feedback from Mark Nottingham.
Parsening That Mess

- If your service handles both browser clients and service clients, you have to handle Accept

- A few years ago Joe Gregorio wrote mimeparse in Python to parse these header values

- I recently ported it to Erlang, available here:

- Also available in Ruby and PHP, same location
Using mimeparse

- For each resource, decide which MIME type(s) you want to support

- Pass a list of those types and the Accept header to `mimeparse:best_match/2`:
  
  ```
  Want = ["application/json", "text/html"],
  Accept = (A#arg.headers)#headers.accept,
  Best = mimeparse:best_match(Want, Accept)
  ```

- Handles quality parameters, etc.

- Returns empty list if no match
Dealing with MIME Types

- For the requested resource, determine the representation type the client wants
  - if there’s no Accept header then choose a default
  - if there’s an Accept header but no match with what you support, return HTTP status 406 (“Not Acceptable”)
- Use pattern matching again to dispatch to the right handler
MIME Type Dispatching

- Change our out/3 function to out/4, adding the MIME type:

\[
\text{out}(\text{Arg, Uri, "text/html", ["order"]}) \rightarrow \\
\text{%% handles HTML repr for path order/;}
\]

\[
\text{out}(\text{Arg, Uri, "application/atom+xml", ["order"]}) \rightarrow \\
\text{%% handles Atom repr for path order/;}
\]

\[
\text{out}(\text{Arg, Uri, Other, ["order"]}) \rightarrow \\
\text{\{status, -406\}.}
\]
Handling Common Representations

- Various packages allow you to natively handle common service resource representations in Erlang

  - JSON:
    - Yaws supplies a json module
    - Mochiweb (another Erlang web framework) supplies mochijson and mochijson2

  - XML:
    - xmerl, part of the Erlang system
    - erlsom, more modern and faster than xmerl
Returning Content

- To return content from your service, just return a “content” tuple from your out/1 function:

  ```erlang
  out(Arg, Uri, "application/json", Path) ->
  Json = {struct, [{name, "Steve Vinoski"},
                    {company, "Verivue"}]},
  Data = json:encode(Json),
  {content, "application/json", Data}.
  
- Sets the Content-type HTTP header to the MIME type you supply as the second tuple element
Supporting Multiple Representations

- Resources can have multiple representations
- Return the appropriate content type from each of your out/4 functions for that resource
- But set the Vary header to alert intermediaries of how the representation varies
- You can return HTTP status, headers, and content all at once like this:

```erlang
out(Arg, Uri, "application/json", Path) ->
  Json = {struct, [{name, "Steve Vinoski"},
    {company, "Verivue"}]},
  Data = json:encode(Json),
  [{status, 200},
    {header, {"Vary", "Accept"}},
    {content, "application/json", Data}].
```
Representations and Hypermedia

- A critical REST constraint is “hypermedia as the engine of application state” (HATEOAS)
  - Representations provide URIs to further resources to drive clients through their application state
  - This works only if the client understands that something in the representation is a URI
  - Common repr types like application/xml and application/json alone do not support HATEOAS!
  - XLink helps XML, and JSON making progress, see http://json-schema.org/
Focus Areas for RESTful Services

- Resources and identifiers
- Representations and media types
- Hypermedia and linking
- HTTP Methods
- Conditional GET
Handling the HTTP Method

- For each resource, decide which HTTP methods it supports
  - GET, PUT, POST, DELETE, OPTIONS, HEAD

- You get the method for a given request from the http_request record via the Arg record:
  
  Method = (Arg#arg.req)#http_request.method

- If a client invokes an unsupported method on a resource, return HTTP status 405 (“Method Not Allowed”)
You guessed it: more pattern matching

Change our out/4 function to out/5, adding the HTTP method:

```erlang
out(Arg, Uri, 'GET', "text/html", ["order"] ) ->
  %% handles GET HTML repr for order/;

out(Arg, Uri, 'POST', "text/html", ["order"] ) ->
  %% handles POST HTML repr for order/;
```
Retrieving Query and POST Data

- `yaws_api:parse_post(Arg)` returns a property list of name, value POST data pairs
- `yaws_api:postvar(Arg, Name)` looks up Name in POST data
- `yaws_api:parse_query(Arg)` returns a property list of name, value query string pairs
- `yaws_api:queryvar(Arg, Name)` looks up Name in the query string
Focus Areas for RESTful Services

- Resources and identifiers
- Representations and media types
- Hypermedia and linking
- HTTP Methods
- Conditional GET
Conditional GET

- Conditional GET and caching are critical to web scalability

- Read Mark Nottingham’s excellent “Caching Tutorial for Web Authors and Webmasters” for details (http://www.mnot.net/cache_docs/)

- Read Richardson’s and Ruby’s RESTful Web Services to learn about conditional GET
Conditional GET

Return Headers

Outgoing: set HTTP Etag and Last-modified headers

- Etag is a hash-like string that uniquely identifies a representation
- Last-modified is the date string of the resource’s most recent modification

Set these like any other header, using a header tuple as part of your out/5 return value:

```erlang
[{header, {"Etag", Etag_value}},
 {header, {"Last-modified", Last_mod_val}}]
```
Conditional GET
Incoming Headers

- To perform a conditional GET, client will send:
  - Last-modified value back in the If-modified-since header
  - Etag value back in the If-none-match header
  - or both, but Etag takes precedence

- Your code needs to look for these and handle them appropriately
Conditional GET
Incoming Headers

- For incoming Etag values, if one matches the requested representation’s Etag...
- ...or for incoming modification dates, if the resource hasn’t changed since that date...
- ...then your service should return status 304 ("Not Modified")
- This avoids creating potentially expensive-to-create representations and avoids returning potentially large representations
Development Concerns

- Yaws is *very* stable and robust
  - uses Erlang/OTP supervision and monitoring capabilities, and can auto-restart if any problems arise
- Provides interactive mode with debug output for tracking down issues with your code
- Full power of Erlang/OTP under it, so you can load new code on the fly for your yapps and appmods
Yaws Community

- Documentation and downloads available at http://yaws.hyber.org/
- Code is on sourceforge: http://sourceforge.net/projects/erlyaws
- Find the erlyaws mailing list there as well
- Since code is very stable, doesn’t change much
  - I recently added better support for the HTTP OPTIONS method
  - Current projects Klacke and I are working on: adding sendfile linked-in driver support, and general testing
But Wait, There’s More

- but not today :-)
- Read the Yaws documentation, lots there to discover
- Any final questions?