Building RESTful Services with Erlang and Yaws

Steve Vinoski
Member of Technical Staff
Verivue
Westford, MA USA
http://steve.vinoski.net/
vinoski@ieee.org
Why Yaws? (“Yet Another Web Server”)

Here we see Yaws handling 80000 concurrent connections, but Apache dying at 4000
Yaws is Written in Erlang

- Erlang began life in 1986 for developing highly reliable distributed concurrent systems
- Developed at Ericsson for telecom equipment
- Open sourced in 1998, it’s been used to develop systems with guaranteed nine nines reliability (31.5ms downtime per year)
- Under active development, version R12B-2 came out in April 2008
Enabling highly reliable systems is a primary goal for Erlang (designed for “concurrent programs that run forever” — Joe Armstrong, creator of Erlang)

- It encourages designs that accept that failure will occur and must be dealt with
  - Processes can be arranged in distributed supervision trees, supervisors watch and restart failed processes
- Code can be loaded into running systems
- The Open Telecom Platform (OTP) libraries provide common application behaviors supporting reliability
Message Passing

- Erlang avoids shared state, uses message passing instead
  - the type of message passing originally intended for OO languages
  - very fast, asynchronous, same host or across hosts
- Erlang variables cannot be re-assigned; they’re bound once and that’s it, to avoid mutable state
- No explicit code for concurrency guards, locks, synchronization etc. required
Pattern Matching

- In Erlang, $X = 3$ is a pattern matching operation
  - if $X$ is unbound, it's bound to the value 3
  - if $X$ is already bound, it's an error unless it's bound to the value 3
- avoids mutable state and the need to guard it
- Pattern matching is a significant and important feature of Erlang
  - used for assignment, checking for values, receiving messages, function selection
Yet Another Web Server (Yaws)

- Yaws was written and is maintained by Claes "Klacke" Wikström, and is open source available from http://yaws.hyber.org/
- Written in Erlang as an OTP application
- Takes advantage of Erlang's concurrency and distribution capabilities to provide significant scalability
- Testimonials often state that Yaws handles on a single host loads that other web servers need multiple hosts to handle
We don’t have enough time for an Erlang tutorial

Get Joe Armstrong's book *Programming Erlang*

- very readable
- both an introduction and a language reference

Lots of information at http://www.erlang.org/

*erlang-questions* mailing list (available from above link)
Restful Design

- Name your resources with URIs
- For each resource, decide:
  - what each HTTP method does and what status codes it returns
  - what media types to support
  - how each representation of the resource guides the client through its application state
  - how to handle conditional GET (etags, last-modified)
REST Basics

- REST is an architectural style that targets large-scale distributed hypermedia systems
- It imposes certain constraints to achieve desirable properties for such systems
Desired System Properties

- Performance, scalability, portability
- Simplicity: simple systems are easier to build, maintain, more likely to operate correctly
- Visibility: monitoring, mediation
- Modifiability: ease of changing, evolving, extending, configuring, and reusing the system
- Reliability: handling failure and partial failure, and allowing for load balancing, failover, redundancy
Constraints Induce Desired Properties

- REST intentionally places constraints on the system to induce these properties
- In general, software architecture is about
  - imposing constraints and
  - choosing from the resulting trade-offs in order to achieve desired properties
REST Constraints

- Client-Server
- Statelessness
- Caching
- Layered System
- Uniform Interface
- Code-on-demand
Uniform Interface Constraint

- What: all servers present the same general interface to clients
  - In HTTP, this interface comprises the protocol’s verbs: GET, PUT, POST, DELETE
- Why: important for implementation hiding, visibility of interactions, intermediaries, scalability
- This constraint induces several more constraints, described later
## HTTP Verbs are Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Idempotent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieve resource state representation</td>
<td>Yes (no side effects)</td>
</tr>
<tr>
<td>PUT</td>
<td>Provide resource state representation</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>Create or extend a resource</td>
<td>No</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Uniform Interface Benefits

- Enables visibility into interactions
  - including caching, monitoring, mediation applicable across all resources
- Provides strong implementation hiding, independent evolvability
- Simplified overall architecture
Uniform Interface Sub-Constraints

- Resource identification via URIs
- Resource manipulation through the exchange of resource state representations
- Self-describing messages with potentially multiple representation formats
- Hypermedia as the engine of application state (a.k.a. HATEOAS, or hypermedia constraint)
Representations

- Method payloads are representations of resource state
  - hence the name “Representational State Transfer”
- REST separates methods and data formats
- Fixed set of methods, many standardized data formats, multiple formats possible per method per resource
Media Types

- Representation formats are identified using media (MIME) types
- These types are standardized/registered through the IANA (http://www.iana.org/assignments/media-types/)
- Allows reusable libraries and tools in a variety of programming languages to handle various MIME types
Hypermedia Constraint

- Resources keep resource state, clients keep application state
- Resources provide URIs in their state to guide clients through the application state
- Clients need “know” only a single URI to enter an application, can get other needed URIs from resource representations
For Example

- Consider a bug-tracking system
- HTML representations for interactive viewing, additions, modifications
- Excel or CSV representations for statistical tracking by importing into other tools
- XML (e.g., AtomPub) or JSON to allow integration with other tools
- Atom feeds for watching bug activity
- Existing clients that understand these formats can easily adapt to use them — serendipity
RESTful Design With Yaws

- Design URIs for your resources
- For each resource, decide:
  - what each HTTP method does and what status codes it returns
  - what media types to support
  - how each representation of the resource guides the client through its application state
  - how to handle conditional GET (etags, last-modified)
URI Design

- URIs must be designed from an application perspective, not from a server perspective.
  - In the old days, URIs corresponded to file pathnames on the web server.
  - That’s still possible, but RESTful services often don’t deal with files at all.
- URIs name the resources the client will access and manipulate.
- URIs collectively form an application state space the client can navigate.
URI Examples

- Bugs for project "Phoenix" might be found here:
  http://example.com/projects/Phoenix/bugs/
- The specific bug numbered 12345 might be here:
  http://example.com/projects/Phoenix/bugs/12345/
- Bugs for user jsmith might be here:
  http://example.com/projects/Phoenix/users/jsmith/bugs/
Representation Generation

- Yaws provides three ways for your code to generate resource representations:
  - .yaws pages
  - application modules (appmods)
  - Yaws applications (yapps)
.yaws Pages

- Enclose a function named “out” taking an Arg (HTTP request argument) within `<erl></erl>` tags in a file
  - in Erlang we refer to this function as `out/1`
  - function named “out” with arity 1 (i.e., 1 argument)
- Give the file a “.yaws” extension
- When the file is requested Yaws executes the `out/1` function and replaces `<erl>...<erl>` with the output of the function
- Mainly useful for relatively static content
- URIs are controlled by where .yaws file is placed relative to the document root
Application Modules (appmods)

- Lets application code take control of URIs
- An Erlang module exporting an out/1 function is configured in the Yaws config file to correspond to a URI path element
- When Yaws sees a request for that path element, it calls the out/1 function passing the HTTP request details, then returns the result of the function
- Such URIs need not correspond to file system artifacts
Yaws Applications (yapps)

- Similar to appmods, but a yapp is a full Erlang/OTP application
- This means it can run an init function, can have state, can support on-the-fly code changes, can be controlled by a supervisor, etc.
- Useful for talking to back-end services, e.g. maintaining connections to the back-end
HTTP Request Details (#arg)

- All out/1 functions receive an #arg record (basically a tuple) containing details of the HTTP request for which they’re being invoked.
- #arg provides details such as HTTP headers and various forms of URI path information.
- For example, to get the request URI:

```erlang
out(Arg) ->
    Uri = yaws_api:request_url(Arg),
```
URI-based Dispatching

- Use Erlang’s pattern matching to dispatch to the right function to handle a given URI

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

out(Arg, ["projects", "Phoenix", "bugs"]) ->
    % handle the bugs URI here;

out(Arg, ["projects", "Phoenix", "bugs", Bug]) ->
    % handle bug number "Bug" here.
```
Handling HTTP Methods

- Same pattern matching approach can be used to dispatch on HTTP method

```erlang
out(Arg) ->
  % get Uri and Path as in previous example
  Method = (Arg#arg.req)#http_request.method,
  out(Arg, Method, Path).
out(Arg, 'GET', ['projects', 'Phoenix', 'bugs']) ->
  % return representation of bug list;
out(Arg, 'POST', ['projects', 'Phoenix', 'bugs']) ->
  % add new bug to the list;
out(Arg, Method, ['projects', 'Phoenix', 'bugs']) ->
  [{status, 405}]; % other methods not allowed
```
Same Again for MIME Types

- Representation the client wants is in the Accept header

```erlang
out(Arg) ->
    % get Uri, Path, Method as in previous examples
    Accept_hdr = (Arg#arg.headers)#headers.accept,
    out(Arg, Method, Accept_hdr, Path).
out(Arg, 'GET', "text/html", ["projects", "Phoenix", "bugs"]) ->
    % return HTML representation of bug list;
out(Arg, 'GET', "application/xml", ["projects", "Phoenix", "bugs"]) ->
    % return XML representation of bug list;
out(Arg, 'GET', Accept, ["projects", "Phoenix", "bugs"]) ->
    [{status, 406}]; % other representations not acceptable
```
Conditional GET Support

- Whenever possible, design your RESTful service to return Last-modified and/or Etag HTTP headers.
- Allows clients to cache and do conditional GETs based on whether the resource has changed since they last retrieved it.
- If no change, server returns status 304 with no payload — big scalability win.
- Can be tricky to design this so that computing Etags has reasonable cost.
For More Information

- **RESTful Web Services** teaches you everything you need to know about developing using the REST style
- My InfoQ article “RESTful Services with Erlang and Yaws” ([http://www.infoq.com/articles/vinoski-erlang-rest](http://www.infoq.com/articles/vinoski-erlang-rest))
- My “Toward Integration” columns in IEEE Internet Computing (all available from [http://steve.vinoski.net/](http://steve.vinoski.net/))
- yaws.hyber.org and erlang.org