Erlang Web Frameworks

Steve Vinoski
Architect, Basho Technologies
Cambridge, MA USA
vinoski@ieee.org
@stevevinoski
http://steve.vinoski.net/
https://github.com/vinoski/

Agenda

- Intro to Erlang
- Mochiweb
- Webmachine
- Yaws
- Nitrogen
- Chicago Boss

Agenda

- Try to hit highlights of each framework, but can't cover everything
- Won't address client-side UI
- Won't address security
- Won't address performance or benchmarking — too app-specific
- Won't be specifically addressing concurrency

basho

Tutorial Code

- Everyone downloaded

  http://cufp.org/sites/all/files/uploads/erlang-web.tar.gz

  as directed on the tutorial website, right?

- If not please get it now (I also have it on a memory stick)

Tutorial Code

- I have some pre-written code for you to use, divided by framework
- So, another file to fetch:

  https://dl.dropbox.com/u/10363968/erlang-web-examples.tar.gz

- Also on memory stick

Tutorial Code

- At various times during the tutorial you'll be copying code from the code-examples directory into areas of the framework we're working on

Machines

- If you don't have the code on your own machine, you can log into some provided by the conference
- ssh cufpXX@tutorial2.icfp

  where "XX" is 01-15, password is cufp2012

Erlang Intro

Erlang is our solution to three problems regarding the development of highly concurrent, distributed "soft real-time systems":

- *To be able to develop the software quickly and efficiently*

- *To have systems that are tolerant of software errors and hardware failures*

- *To be able to update the software on the fly, that is, without stopping execution*

— Mike Williams, co-inventor of Erlang

(quote from "Erlang Programming" by F. Cesarini and S. Thompson)

basho

Erlang Origins

- Mid-80's, Ericsson Computer Science Laboratories (CSL)
- Joe Armstrong began investigating languages for programming next-generation telco equipment
- Erlang initially implemented in Prolog with influence and ideas from ML, Ada, Smalltalk, other languages

Erlang History

- Robert Virding and Mike Williams joined Joe to continue Erlang development
- 1991: Claes “Klacke” Wikström added distribution to Erlang
- 1993: first commercial version of Erlang
- 1996: Open Telecom Platform (OTP)
- 1998: Erlang goes open source, available from www.erlang.org

Erlang for Web Development

- What makes Erlang good for telecom also makes it good for Web systems
  - excellent fault tolerance, failover, scalability
  - exceptional concurrency support
  - application model supports easily running independently-developed apps in the same VM
  - excellent integration facilities
  - data types work well with Web languages HTML, JSON, XML, etc.
  - HTTP packet parsing is built in
  - Comes with Web server and Web client libraries

Erlang Basics

Code Comments

- Comments start with % and go to end of line
- No special delimiters for block comments
- Best practice (emacs erlang-mode enforced)
  - Use %%% to start comments in left column
  - Use %% to start comments at current indentation level
  - Use % comments for end-of-line comments

Hello World

```
-module(hello).
```

Hello World

```
-module(hello).
-export([world/0]).
```

Hello World

```
-module(hello).
-export([world/0]).

world() ->
```

Hello World

```
-module(hello).
-export([world/0]).

world() ->
    io:format("Hello, World!~n").
```

Function Calls

- Calls to functions in other modules:

  hello:world(),
  module_name:function_name(Arg1, Arg2),

- **Calls to local functions:**

  function_name(Arg1, Arg2),

Variables

- Immutable (bind once, no rebind)
- Start with a capital letter, e.g:

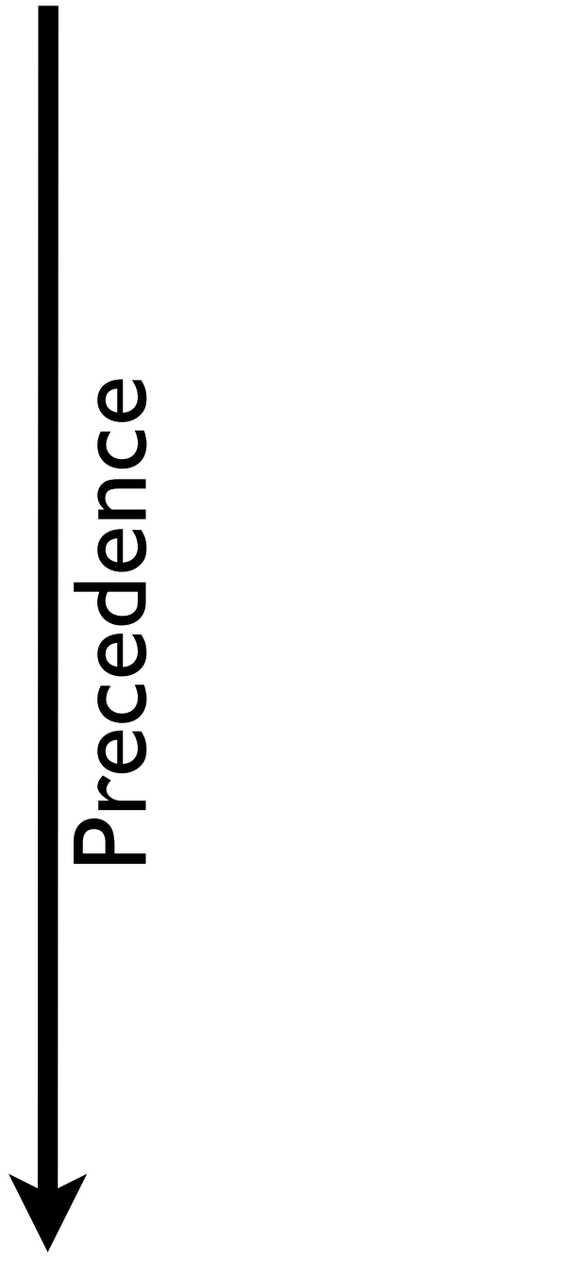
    Variable = 3.

Operators

- **arithmetic**: *, /, +, -, div, rem, unary + and -
- **comparison**: ==, /=, =<, <, >=, >, =:=, =/=
- **logical**: not, and, or, xor, andalso, orelse
- **bitwise**: bnot, band, bor, bxor, bsr, bsl
- **list addition and subtraction**: ++, --

Operator Precedence

- : (as in Module:Fun(...))
- # (for record access)
- (unary) +, (unary) -, bnot, not
- /, *, div, rem, band, and
- +, -, bor, bxor, bsl, bsr, or, xor
- ++, - - (right associative)
- ==, /=, =<, <, >=, >, =:=, =/=
- andalso
- orelse



Types

- atom: a named non-numeric constant
  - generally starts with lowercase letter
  - can also surround with single quotes, e.g., 'EXIT' is an atom
- number: floating point or integer, integers are arbitrary precision
- list: variable-length group of terms

  [numbers, [1, 2, 3, 4]]

Types

- string: just a list of characters

"string" = [$s, $t, $r, $i, $n, $g]

"string" = [115, 116, 114, 105, 110, 103]

- tuple: fixed-size group of terms

{this, "is", A, [$t, $u, $p, $l, $e]}

Records

- record: basically a tuple with named elements
- but unlike tuples records allow direct field access

```
-record(record_name, {
                      field1,
                      field2,
                      field3}).
```

Records

- Accessing a field:

  Var = RecordVar#rec_name.field2,

- Setting a field:

  NewRec = OldRec#rec_name{field3 = foo},

  - OldRec **is immutable, so** NewRec **is a (shallow) copy**

Pattern Matching

- Assignment is actually matching (but not regular expression matching)
- If left-hand side of assignment (LHS) is:
  - unbound: bind it to the value of the right-hand side (RHS)
  - bound and same as RHS: match
  - bound but not same as RHS: no match, badmatch exception

Functions ("funs")

- Exported or local
- 0-255 arguments
- Can be "overloaded" by varying function arity (i.e., number of arguments)
- Return value is whatever the last statement executed returns

Hello World, Again

```
world() ->
    io:format("Hello, World!~n"). % returns ok
```

- **The** io:format/1 **fun returns the atom** ok**, so** world/0 **returns** ok

Multiple Fun Clauses

- Funs of same name with different arities
- Funs of same name, same arity, but different argument matching
- Semicolon separates same arity clauses
- Period separates different arity clauses

Multiple Fun Clauses

```
add(List) ->
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).          % call arity 2 version
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).                % call arity 2 version
add([H | T], Total) ->           % H is head, T is tail
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).                % call arity 2 version
add([H | T], Total) ->           % H is head, T is tail
    add(T, Total + H);           % tail recursion
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).                % call arity 2 version
add([H | T], Total) ->           % H is head, T is tail
    add(T, Total + H);           % tail recursion
add([], Total) ->                % match empty list
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).                % call arity 2 version
add([H | T],Total) ->            % H is head,T is tail
    add(T,Total + H);            % tail recursion
add([],Total) ->                 % match empty list
    Total.                       % end recursion
```

Multiple Fun Clauses

```
add(List) ->
    add(List, 0).                % call arity 2 version
add([H | T], Total) ->           % H is head, T is tail
    add(T, Total + H);           % tail recursion
add([], Total) ->                % match empty list
    Total.                       % end recursion
```

First-Class Funs

- Funs can be assigned to variables
- Funs can be passed as arguments
- Funs can be anonymous

```
Fun = fun(X,Y) -> [{X-1,Y-1}, {Y+1,X+1}] end,
lists:unzip(Fun(3,7)).
```

- Returns {[2,8],[6,4]}

Binaries

- binary: basically, a memory buffer

  <<“string”, 16#BEEF:16/little>>

- Stored more efficiently than lists and tuples
- Large binaries (> 64 bytes) are reference counted, not copied

Matching/Binding

- Binding variables via matching can apply to multiple variables at once

```
{{Year, Month, Day}, {Hour, Min, Sec}} =
    calendar:local_time().
```

- **Result:** {{2011,7,27},{10,32,47}}
  - Year=2011, Month=7, Day=27
    Hour=10, Min=32, Sec=47

Binary Matching

- Matching a TCP header (from Cesarini & Thompson “Erlang Programming”)

```
<<SourcePort:16, DestinationPort:16,
  SequenceNumber:32, AckNumber:32,
  DataOffset:4, _Reserved:4, Flags:8,
 WindowSize:16, Checksum:16,
  UrgentPointer:16, Payload/binary>> = TcpBuf.
```

Control Constructs

- Guards
- Conditionals: case and if
- try - catch - after

Guards

- Augment pattern matching with simple comparisons and tests
- Must be free of side effects
- Can be grouped with “;” (for “or”) or “,” (for “and”)

```
add(L) when is_list(L) ->
```

Case

```
case Expression of
    Pattern1 [when Guard1] -> Expression(s);
    Pattern2 [when Guard2] -> Expression(s);
    ...
end
```

- Very common in Erlang code
- Use _ (underscore) as a match-all pattern
- Returns value of last expression evaluated

If

```
if
    Guard1 -> Expression(s);
    Guard2 -> Expression(s);
    Guard3 -> Expression(s);
    ...
end
```

- Not very common in Erlang code
- Use true as a catch-all clause
- Returns value of last expression evaluated

try-catch-after

```
try Expression(s) of
    Pattern1 [when Guard1] -> Expression(s);
    Pattern2 [when Guard2] -> Expression(s);
    ...
catch
    ExceptionTypeA [when GuardA] -> Expr(s);
    ExceptionTypeB [when GuardB] -> Expr(s);
    ...
after
    Expression(s)
end
```

try-catch-after

- Exception type is [Class]:ExceptionPattern
- _:_ is match-all exception type
- "of" can be left out
  - result is value of tried expression(s)
- "catch" part can be left out
  - any exception gets thrown to caller
- "after" part can be left out

Exception Example

```
try
    I = 2
catch
    Class:Exc ->
        io:format("class: ~p, exception: ~p~n",
                  [Class, Exc])
after
    io:format("example done~n")
end.
```

- **Executing this results in:**

```
class: error, exception: {badmatch,2}
example done
```

Error Handling

- Erlang philosophy: “let it crash”
- Code should deal with errors it knows how to deal with, and let other code handle the rest
- Idiomatic Erlang code doesn’t contain much error handling code (unlike, say, Java or C++)

Erlang Shell

- Erlang includes a very useful read-eval-print loop (REPL)
- Just run erl at the command line

```
$ erl
Erlang R14B04 (erts-5.8.5) [source] [smp:8:8] [rq:8]
[async-threads:0] [kernel-poll:false]

Eshell V5.8.5  (abort with ^G)
1>
```

Exiting the Shell

- One way is q().
  - terminates everything
- Another way is ctrl-g followed by q
  - this is preferable, since it can exit a shell on a remote VM without stopping the VM
- Ctrl-d doesn't work here

Simple Operations

```
1> 2+3.
5
2> X = 1.
1
3> X = 2.
** exception error: no match of right hand side value 2
4> [H|T] = [1,2,3,4].
[1,2,3,4].
5> {H,T}.
{1,[2,3,4]}
```

Managing Variables

```
1> X = 1.
1
2> f(X).
ok
3> X.
* 1: variable 'X' is unbound
4> f().
%% now all variables are unbound
```

Calling a Function

```
1> {{Year,Mon,Day},{Hr,Min,Sec}} =
       calendar:local_time().
{{2011,10,20},{22,20,6}}
2> {Day,Hr}.
{20,22}
```

Anonymous Fun

```
1> F = fun(Item) -> Item =:= foo end.
#Fun<erl_eval.6.80247286>
2> lists:map(F, [foo, bar, baz, foo]).
[true, false, false, true]
```

Displaying a Full Term

- The shell abbreviates large terms to avoid the pain of lengthy scrolling

1> List = lists:seq(1, 35).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
26,27,28,29|...]

- Print the entire term with rp()

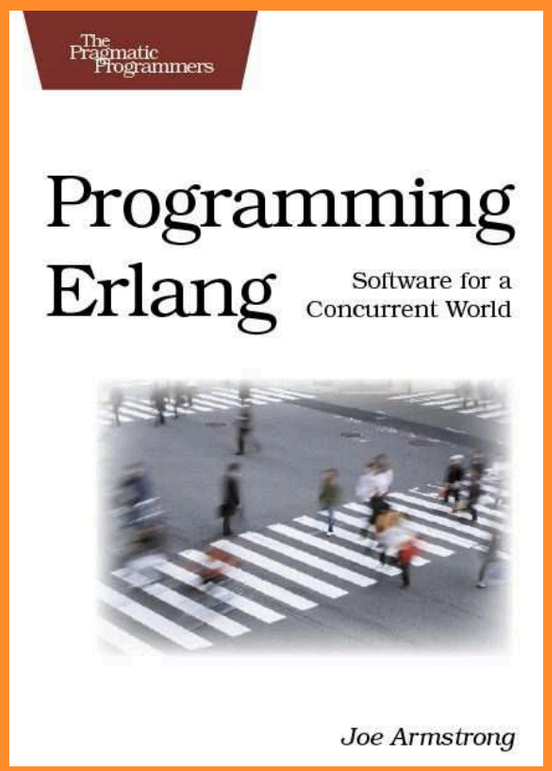
2> rp(List).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,2
0,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35]
ok

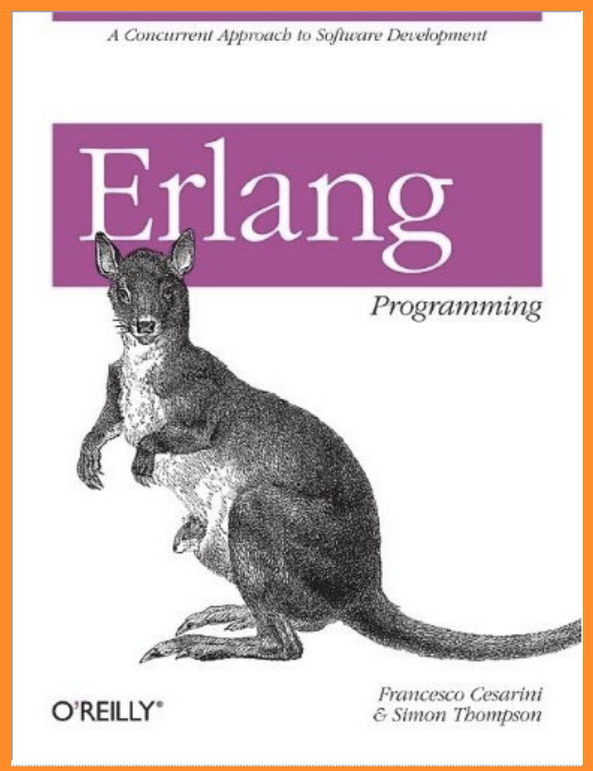
Job Control

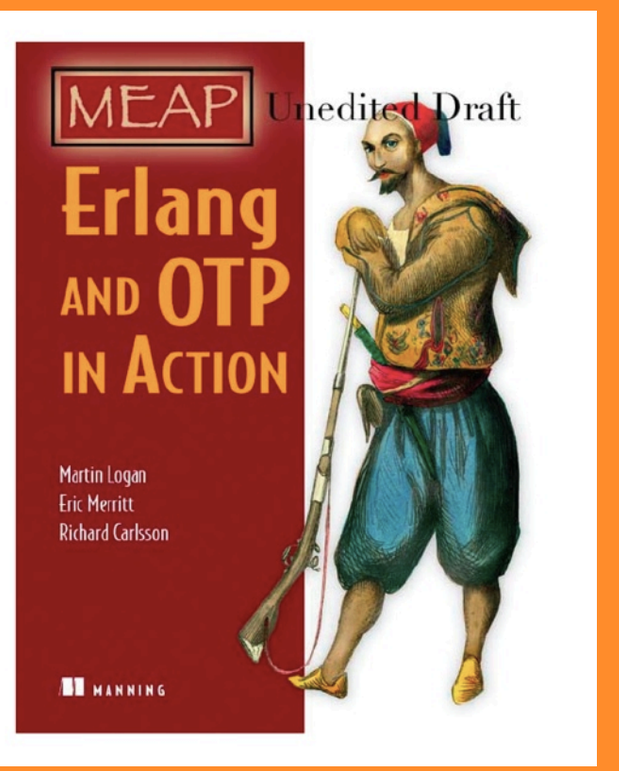
- Ctrl-g gets you to the job control menu
- Start, stop, and switch shells

```
User switch command
 --> h
 c [nn]                - connect to job
 i [nn]                - interrupt job
 k [nn]                - kill job
 j                     - list all jobs
 s [shell]             - start local shell
 r [node [shell]]      - start remote shell
 q                     - quit erlang
 ? | h                 - this message
```

For More Info







Also: http://learnyousomeerlang.com/

An Erlang Web Server

- Seems like lots of newcomers to Erlang learn the language by writing a web server
- Seems to result in a lot of unneeded web servers being open sourced
- Let's write SWS ("simple web server") to see how easy it is

An Erlang Web Server

- See file sws.erl in the erlang_examples directory

Start SWS

```
-module(sws).
-export([start/1, start/2]).
```

Start SWS

```
-module(sws).
-export([start/1, start/2]).

start(Handler) ->
    start(Handler, 12345).
```

Start SWS

```
-module(sws).
-export([start/1, start/2]).

start(Handler) ->
    start(Handler, 12345).
start(Handler, Port) ->
    {ok, LS} = gen_tcp:listen(Port, [{reuseaddr,true},binary,
                                     {backlog,1024}]),
```

Start SWS

```
-module(sws).
-export([start/1, start/2]).

start(Handler) ->
    start(Handler, 12345).
start(Handler, Port) ->
    {ok, LS} = gen_tcp:listen(Port, [{reuseaddr,true},binary,
                                     {backlog,1024}]),
    spawn(fun() -> accept(LS, Handler) end),
```

Start SWS

```
-module(sws).
-export([start/1, start/2]).

start(Handler) ->
    start(Handler, 12345).
start(Handler, Port) ->
    {ok, LS} = gen_tcp:listen(Port, [{reuseaddr,true},binary,
                                     {backlog,1024}]),
    spawn(fun() -> accept(LS, Handler) end),
    receive stop -> gen_tcp:close(LS) end.
```

basho

Start SWS

```
-module(sws).
-export([start/1, start/2]).

start(Handler) ->
    start(Handler, 12345).
start(Handler, Port) ->
    {ok, LS} = gen_tcp:listen(Port, [{reuseaddr,true},binary,
                                     {backlog,1024}]),
    spawn(fun() -> accept(LS, Handler) end),
    receive stop -> gen_tcp:close(LS) end.
```

basho

Accept Connections

```
accept(LS, Handler) ->
    {ok, S} = gen_tcp:accept(LS),
```

Accept Connections

```
accept(LS, Handler) ->
    {ok, S} = gen_tcp:accept(LS),
    ok = inet:setopts(S, [{packet, http}]),
```

basho

Accept Connections

```
accept(LS, Handler) ->
    {ok, S} = gen_tcp:accept(LS),
    ok = inet:setopts(S, [{packet, http}]),
    spawn(fun() -> accept(LS, Handler) end),
```

Accept Connections

```
accept(LS, Handler) ->
    {ok, S} = gen_tcp:accept(LS),
    ok = inet:setopts(S, [{packet, http}]),
    spawn(fun() -> accept(LS, Handler) end),
    serve(S, Handler, [{headers, []}]).
```

Accept Connections

```
accept(LS, Handler) ->
    {ok, S} = gen_tcp:accept(LS),
    ok = inet:setopts(S, [{packet, http_bin}]),
    spawn(fun() -> accept(LS, Handler) end),
    serve(S, Handler, [{headers, []}]).
```

basho

Serving Connections

```
serve(S, Handler, Req) ->
    ok = inet:setopts(S, [{active, once}]),
    HttpMsg = receive
                  {http, S, Msg} -> Msg;
                  _ -> gen_tcp:close(S)
              end,
    case HttpMsg of
        ...
    end.
```

HTTP Request Line

```
{http_request, M, {abs_path, Uri}, Vsn} ->
```

HTTP Request Line

```
{http_request, M, {abs_path, Uri}, Vsn} ->
    NReq = [{method, M},{uri, Uri},{version, Vsn}|Req],
```

HTTP Request Line

```
{http_request, M, {abs_path, Uri}, Vsn} ->
    NReq = [{method, M},{uri, Uri},{version, Vsn}|Req],
    serve(S, Handler, NReq);
```

HTTP Request Line

```
{http_request, M, {abs_path, Uri}, Vsn} ->
    NReq = [{method, M}, {uri, Uri}, {version, Vsn}|Req],
    serve(S, Handler, NReq);
```

HTTP Headers

```
{http_header, _, Hdr, _, Val} ->
    {headers, Hdrs} = lists:keyfind(headers, 1, Req),
    serve(S, Handler, lists:keystore(headers, 1, Req,
                                     {headers, [{Hdr,Val}|Hdrs]})));
```

basho

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
```

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
```

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
ok = gen_tcp:send(S, [
```

basho

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
ok = gen_tcp:send(S, [
    "HTTP/1.0 ", integer_to_list(Status), "\r\n",
```

basho

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
    ok = gen_tcp:send(S, [
      "HTTP/1.0 ", integer_to_list(Status), "\r\n",
      [[H, ": ", V, "\r\n"] || {H,V} <- Hdrs],
```

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
    ok = gen_tcp:send(S, [
        "HTTP/1.0 ", integer_to_list(Status), "\r\n",
        [[H, ": ", V, "\r\n"] || {H,V} <- Hdrs],
        "\r\n", Resp]),
```

basho

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
    ok = gen_tcp:send(S, [
        "HTTP/1.0 ", integer_to_list(Status), "\r\n",
        [[H, ": ", V, "\r\n"] || {H,V} <- Hdrs],
        "\r\n", Resp]),
    gen_tcp:close(S);
```

Dispatch Request

```
http_eoh ->
    ok = inet:setopts(S, [{packet, raw}]),
    {Status, Hdrs, Resp} = try Handler(S, Req)
                           catch _:_ -> {500, [], <<>>} end,
    ok = gen_tcp:send(S, [
        "HTTP/1.0 ", integer_to_list(Status), "\r\n",
        [[H, ": ", V, "\r\n"] || {H,V} <- Hdrs],
        "\r\n", Resp]),
    gen_tcp:close(S);
```

37 Readable LOC

Try It

```
$ erlc sws.erl
$ erl
Erlang R15B02 (erts-5.9.2) ...

Eshell V5.9.2  (abort with ^G)
1> Fun = fun(S, Req) -> ... end.
2> P = spawn(sws, start, [Fun]).
```

...use curl or browser to send requests...

```
3> P ! stop.
```

basho


basho

Mochiweb

Quick Glance

- Mochiweb is a popular bare-bones web server created by Bob Ippolito
- Originally created for serving ads for Mochi Media, Bob's startup
- Typically used to implement web services
  - supports full access to HTTP
  - great JSON support
  - no templates, etc.

Installation

make

- This builds Mochiweb itself
- To generate an application:

make app PROJECT=<project name>

Run the Server

- Let's say the project name is cufp_mw:

```
make app PROJECT=cufp_mw
cd ../cufp_mw
make
./start-dev.sh
```

- This starts a development server with an Erlang console
- Mochiweb will automatically reload compiled files

basho

Project Contents

$ ls -1F

Makefile	
deps/	# rebar dependencies
ebin/	# compiled Erlang
priv/	# static web files
rebar*	# rebar executable
rebar.config	
src/	# source files
start-dev.sh*	# server start script

basho

Document Root

- Access

  http::/localhost:8080

- By default this serves the file priv/www/index.html

Startup Code

- See src/cufp_mw_web.erl

Startup Code

- See src/cufp_mw_web.erl

```
start(Options) ->
    {DocRoot, Options1} =
        get_option(docroot, Options),
```

Startup Code

- See src/cufp_mw_web.erl

```
start(Options) ->
    {DocRoot, Options1} =
        get_option(docroot, Options),
Loop = fun(Req) ->
           ?MODULE:loop(Req, DocRoot)
       end,
```

Startup Code

- See src/cufp_mw_web.erl

```
start(Options) ->
    {DocRoot, Options1} =
        get_option(docroot, Options),
Loop = fun(Req) ->
            ?MODULE:loop(Req, DocRoot)
        end,
mochiweb_http:start([{name, ?MODULE},
        {loop, Loop} | Options1]).
```

Startup Code

- See src/cufp_mw_web.erl

```
start(Options) ->
    {DocRoot, Options1} =
        get_option(docroot, Options),
Loop = fun(Req) ->
               ?MODULE:loop(Req, DocRoot)
           end,
mochiweb_http:start([{name, ?MODULE},
          {loop, Loop} | Options1]).
```

Default Request Loop

loop(Req, DocRoot) ->

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
```

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
```

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
            Method when Method =:= 'GET'; Method =:= 'HEAD' ->
                case Path of
                    _ ->
                        Req:serve_file(Path, DocRoot)
end;
```

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
            Method when Method =:= 'GET'; Method =:= 'HEAD' ->
                case Path of
                    _ ->
                        Req:serve_file(Path, DocRoot)
                end;
            'POST' ->
                case Path of
                    _ ->
                        Req:not_found()
                end;
```

basho

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
            Method when Method =:= 'GET'; Method =:= 'HEAD' ->
                case Path of
                    _ ->
                        Req:serve_file(Path, DocRoot)
                end;
            'POST' ->
                case Path of
                    _ ->
                        Req:not_found()
                end;
            _ ->
                Req:respond({501, [], []})
        end
...
```

basho

Default Request Loop

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
            Method when Method =:= 'GET'; Method =:= 'HEAD' ->
                case Path of
                    _ ->
                        Req:serve_file(Path, DocRoot)
                end;
            'POST' ->
                case Path of
                    _ ->
                        Req:not_found()
                end;
            _ ->
                Req:respond({501, [], []})
        end
...
```

Handling a New Path

- You can modify the cufp_mw_web:loop/2 function to add handling for specific paths
- Let's add a hello_world path

hello_world Path

```
case Req:get(method) of
    Method when Method =:= 'GET'; Method =:= 'HEAD' ->
        case Path of
            _ ->
                Req:serve_file(Path, DocRoot)
        end;
```

hello_world Path

```
case Req:get(method) of
    Method when Method =:= 'GET'; Method =:= 'HEAD' ->
        case Path of
            "hello_world" ->
                Req:respond({200,
                             [{"Content-Type", "text/plain"}],
                             "Hello, World!"});
            _ ->
                Req:serve_file(Path, DocRoot)
    end;
```

hello_world Access

- Copy cufp_mw_web1.erl to cufp_mw/src/cufp_mw_web.erl and recompile

- Access

  http://localhost:8080/hello_world

- You'll see the "Hello, World!" message

Request API

- This API gives you access to everything arriving in the client HTTP request
- It also allows you to form the HTTP response

Examples

```
loop(Req, DocRoot) ->
    "/" ++ Path = Req:get(path),
    try
        case Req:get(method) of
            Method when Method =:= 'GET'; Method =:= 'HEAD' ->
                case Path of
                    _ ->
                        Req:serve_file(Path, DocRoot)
                end;
            'POST' ->
                case Path of
                    _ ->
                        Req:not_found()
                end;
            _ ->
                Req:respond({501, [], []})
        end
...
```

Req:get/1

- Req:get/1 lets you retrieve fields of the request
- Fields are:
  - socket
  - protocol scheme
  - method
  - raw path
  - protocol version
  - headers
  - peer
  - path
  - body length
  - range

Headers and Cookies

- Req:get_header_value/1 gets the value of the specified header
- Req:get_cookie_value/1 gets the value of the specified cookie
- Both return the atom 'undefined' if requested item not present

Query Strings

- Req:parse_qs/0 parses any query string portion of the URL and returns a property list
- Use the Erlang lists module "key*" functions or the proplists module to handle the result

Query Strings

- Previously we added this code:

```
case Req:get(method) of
    Method when Method =:= 'GET'; Method =:= 'HEAD' ->
        case Path of
    "hello_world" ->
        Req:respond({200,
                     [{"Content-Type", "text/plain"}],
                     "Hello, World!"});
    _ ->
        Req:serve_file(Path, DocRoot)
end;
```

basho

Query Strings

- Let's add query strings:

```
"hello_world" ->
    Q = Req:parse_qs(),
    Msg = proplists:get_value(
            "message", Q, "Hello, World!"),
    Req:respond({200,
            [{"Content-Type", "text/plain"}],
            Msg});
```

Query Strings

- Copy cufp_mw_web2.erl to cufp_mw/src/cufp_mw_web.erl and recompile
- With this new code deployed, access

  http://localhost:8080/hello_world?message=Hello%20CUFP

  to see the message change

POSTs

- POST requests have bodies that you must retrieve
- Bodies can have different content types
- Req:parse_post() will handle bodies of MIME type

  application/x-www-form-urlencoded

POSTs

- For GET on hello_world, let's return some HTML
- Copy cufp_mw_web3.erl to cufp_mw/src/ cufp_mw_web.erl to change the "hello" handler to:

```
Req:serve_file("message.html", DocRoot);
```

- This presents a field you can fill in with a message
- If you fill in a message and hit the Submit button, what do you get? Why?

POSTs

- We need to handle a POST to /hello_world. In the POST section of the code, add:

```
case Path of
    "hello_world" ->
        Post = Req:parse_post(),
        Msg = proplists:get_value(
            "message", Post, "Hello, World!"),
        Req:respond({200,
                     [{"Content-Type", "text/plain"}],
                     Msg});
```

- You can just copy cufp_mw_web4.erl to cufp_mw/src/cufp_mw_web.erl and recompile

File Upload

- File upload is essentially a POST
- Data has the MIME type "multipart/form-data"
- The module mochiweb_multipart helps handle this
- Copy cufp_mw_web5.erl to cufp_mw/src/cufp_mw_web.erl and recompile

GET Handler

```
"upload" ->
    Req:serve_file("upload.html", DocRoot);
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->



    end),
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->
            Nm = filename:join("/tmp", Nm0),


        end),
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->
            Nm = filename:join("/tmp", Nm0),
            {ok, F} = file:open(Nm, [raw, write]),

        end),
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->
            Nm = filename:join("/tmp", Nm0),
            {ok, F} = file:open(Nm, [raw, write]),
            chunk_handler(Nm, Type, F)
    end),
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->
            Nm = filename:join("/tmp", Nm0),
            {ok, F} = file:open(Nm, [raw, write]),
            chunk_handler(Nm, Type, F)
    end),
Req:ok({"text/plain", "Upload OK"});
```

POST Handler

```
"upload" ->
    mochiweb_multipart:parse_form(
        Req,
        fun(Nm0, Type) ->
            Nm = filename:join("/tmp", Nm0),
            {ok, F} = file:open(Nm, [raw, write]),
            chunk_handler(Nm, Type, F)
    end),
Req:ok({"text/plain", "Upload OK"});
```

chunk_handler/3

```
chunk_handler(Filename, ContentType, File) ->
```

chunk_handler/3

```
chunk_handler(Filename, ContentType, File) ->
    fun(Chunk) ->
```

end.

chunk_handler/3

```
chunk_handler(Filename, ContentType, File) ->
    fun(Chunk) ->
            case Chunk of
                eof ->
                    ok = file:close(File),
                    {Filename, ContentType};



            end
    end.
```

chunk_handler/3

```
chunk_handler(Filename, ContentType, File) ->
    fun(Chunk) ->
            case Chunk of
                eof ->
                    ok = file:close(File),
                    {Filename, ContentType};
                _ ->
                    ok = file:write(File, Chunk),
                    chunk_handler(Filename, ContentType, File)
            end
    end.
```

chunk_handler/3

```
chunk_handler(Filename, ContentType, File) ->
    fun(Chunk) ->
            case Chunk of
                eof ->
                    ok = file:close(File),
                    {Filename, ContentType};
                _ ->
                    ok = file:write(File, Chunk),
                    chunk_handler(Filename, ContentType, File)
            end
    end.
```

POST Handler Caveats

- We're not really handling errors
- This doesn't allow multiple users to upload the same file at the same time
  - write to temporary files to allow that

And More

- Receiving and sending directly from/to socket
- JSON encoding and decoding
- UTF-8 handling
- Unfortunately not well documented
- See https://github.com/mochi/mochiweb


basho

Webmachine

Quick Glance

- WebMachine essentially models HTTP — it's "shaped like the Web"
- Originally inspired by Alan Dean's "HTTP header status diagram" (we'll see this later)
- Helps build web applications that conform to and take advantage of HTTP
- Runs on top of Mochiweb

Installation

make

- This builds WebMachine itself
- To generate an application:

```
scripts/new_webmachine.sh \
    <app-name> [destination-dir]
```

Run the Server

- Let's say the project name is cufp_wm:

```
scripts/new_webmachine.sh cufp_wm ..
cd ../cufp_wm
make
./start.sh
```

- This starts a development server with an Erlang console, with Mochiweb's auto-reloading of recompiled modules in effect

Project Contents

$ ls -1F

Makefile	
README	# helpful instructions
deps/	# rebar dependencies
ebin/	# compiled Erlang
priv/	# static web files
rebar*	# rebar executable
rebar.config	
src/	# source files
start.sh*	# server start script

basho

Access the Server

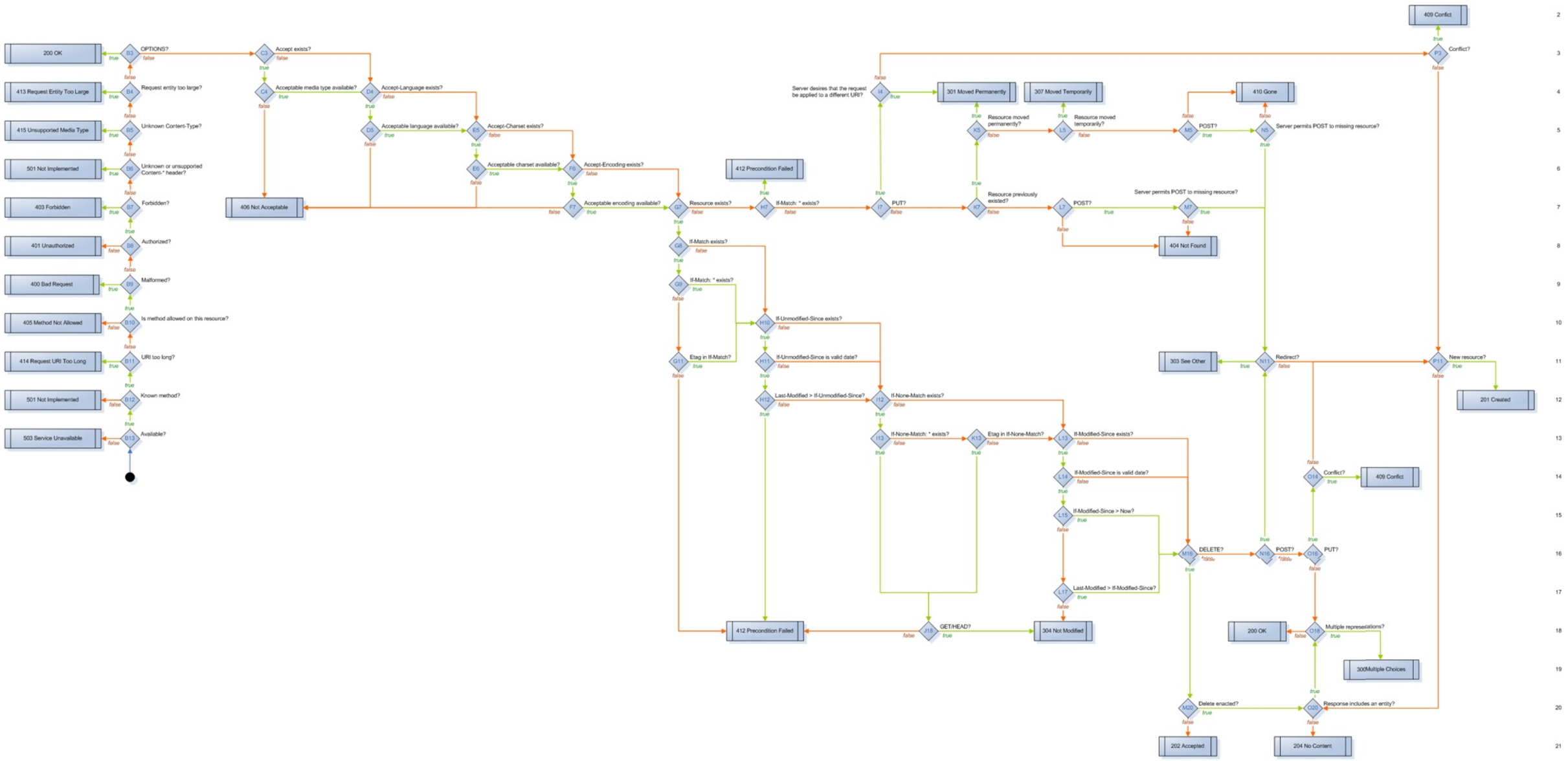
- Start the server and access

  http://localhost:8000

- You'll see a "Hello, new world" message

Fundamentals

- Dispatcher: this is how WebMachine decides how to dispatch requests
  - you supply dispatch rules for your applications
- Resources: you write these to represent your web resources
- Decision Core: basically, the HTTP flowchart



HTTP is complicated.

(see http://webmachine.basho.com/Webmachine-Diagram.html or file www/images/http-headers-status-v3.png

Dispatch Map

- Take a look at file

  priv/dispatch.conf

- Generating a project creates this file with a single dispatching rule:

  {[], cufp_wm_resource, []}.

Dispatch Map Entry

{<pathspec>, <resource_module>, <args>}

- <pathspec> matches a URL path
- <resource_module> handles the request
- <args> are passed to <resource_module:init>/1 function before request dispatch

Pathspecs

- URL paths are separated at "/" characters and then matched against the pathspec of each map entry
- A pathspec is a list of pathterms
- A pathterm is one of
  - string
  - atom
  - '*' (an asterisk as an Erlang atom)

Pathterms

- A string pathterm specifies a literal URL path segment
- An atom pathterm matches a single path segment
- The '*' matches zero or more path segments
- The '*' can appear only at the end of a pathspec

wrq Module

- The wrq module provides access to request and response data, including dispatch information
- wrq:path/1 returns the URL path
- wrq:disp_path/1 returns any path matching '*'
- wrq:path_info/1 returns a dictionary of atoms and their matches
- wrq:path_tokens/1 returns disp_path separated by "/"

URL Dispatching = Pattern Matching

```
{["a"],some_resource,[]}
```

`http://myhost/a` ⟶ match!

`any other URL` ⟶ no match

If no patterns match, then `404 Not Found`.

URL Dispatching = Pattern Matching

```
{["a"],some_resource,[]}
```

/a

[ ]	wrq:disp_path
"/a"	wrq:path
[ ]	wrq:path_info
[ ]	wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a"],some_resource,[]}
```

/a

[ ]	wrq:disp_path
"/a"	wrq:path
[ ]	wrq:path_info
[ ]	wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a"                ,some_resource,[]}
```

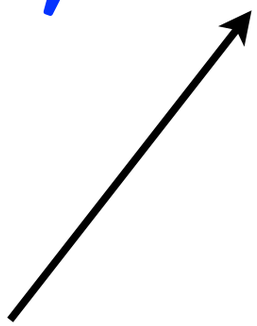
/a

[ ]	wrq:disp_path
"/a"	wrq:path
[ ]	wrq:path_info
[ ]	wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a", '*'],some_resource,[]}
```

/a



(binds the remaining path)

[ ]	wrq:disp_path
"/a"	wrq:path
[ ]	wrq:path_info
[ ]	wrq:path_tokens

basho

URL Dispatching = Pattern Matching

```
{["a", '*'],some_resource,[]}
```

/a/b/c

"b/c"	wrq:disp_path
"/a/b/c"	wrq:path
[]	wrq:path_info
["b", "c"]	wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a", foo],some_resource,[]}
```

/a/b/c ⟶ 404

(name-binds a path segment)

"b/c"

"/a/b/c"

[]

["b", "c"]

wrq:disp_path

wrq:path

wrq:path_info

wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a", foo],some_resource,[]}
```

/a/b

```
                []          wrq:disp_path
          "/a/b"            wrq:path
[{foo, “b”}]                wrq:path_info
                []          wrq:path_tokens
```

URL Dispatching = Pattern Matching

```
{["a", foo                some_resource,[]}
```

/a/b

```
                  []
           "/a/b"
  [{foo, “b”}]
                  []
```

wrq:disp_path
wrq:path
wrq:path_info
wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a", foo, '*'],some_resource,[]}
```

/a/b

```
                []          wrq:disp_path
          "/a/b"            wrq:path
[{foo, “b”}]                wrq:path_info
                []          wrq:path_tokens
```

URL Dispatching = Pattern Matching

```
{["a", foo, '*'],some_resource,[]}
```

/a/b/c/d

"c/d"	wrq:disp_path
"/a/b/c/d"	wrq:path
[{foo, "b"}]	wrq:path_info
["c","d"]	wrq:path_tokens

URL Dispatching = Pattern Matching

```
{["a", foo, '*'],some_resource,[]}
```

/a/b/c/d

"c/d"	wrq:disp_path
"/a/b/c/d"	wrq:path
[{foo, "b"}]	wrq:path_info
["c","d"]	wrq:path_tokens

Query Strings

/a/b/c/d?user=steve&id=123

- Access query string values using

  wrq:get_qs_value(QS, ReqData)

- wrq:get_qs_value("user", ReqData) would return "steve"

Resources

- Take a look at the file src/cufp_wm_resource.erl
- It contains an init/1 function and a single resource function

cufp_wm_resource.erl

```
-module(cufp_wm_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").
```

cufp_wm_resource.erl

```
-module(cufp_wm_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.
```

cufp_wm_resource.erl

```
-module(cufp_wm_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

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

basho

cufp_wm_resource.erl

```
-module(cufp_wm_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

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

init/1 and Context

- init/1 returns Context, which is app-specific request state
- WM doesn't touch the Context object, just passes it along through resource functions
- It's the atom 'undefined' for cufp_wm_resource.erl

to_html/2

- WM Decision Core examines the request Accept header when you request the cufp_wm "/" resource
- Determines the client can take an HTML representation as a response
- Calls to_html/2 on the target resource module to get the HTML response

to_html/2

to_html/2

- But if the client requests something other than HTML?

  $ curl -D /dev/tty -H 'Accept: text/plain' \
  http://localhost:8000

  HTTP/1.1 **406 Not Acceptable**
  Server: MochiWeb/1.1 WebMachine/1.9.0
  Date: Mon, 10 Sep 2012 00:24:38 GMT
  Content-Length: 0

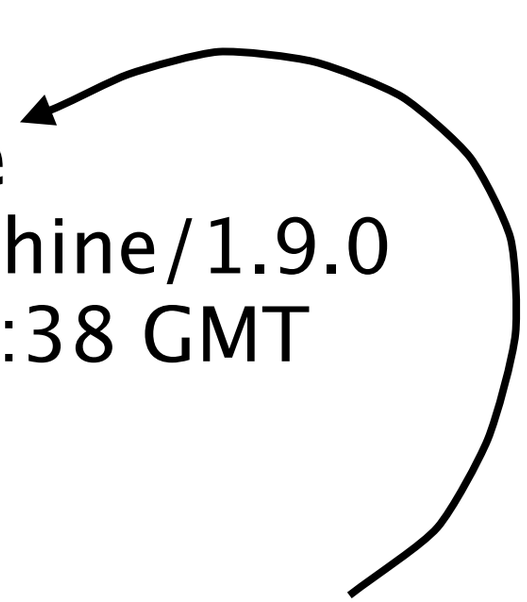
to_html/2

- But if the client requests something other than HTML?

  $ curl -D /dev/tty -H 'Accept: text/plain' \
  http://localhost:8000

  HTTP/1.1 **406 Not Acceptable**
  Server: MochiWeb/1.1 WebMachine/1.9.0
  Date: Mon, 10 Sep 2012 00:24:38 GMT
  Content-Length: 0

- No acceptable representation available!



Content Types Provided

- Resources define what content types they can provide with the

  content_types_provided/2

  resource function

- This defaults to "text/html" mapping to function to_html/2

Allowing text/plain

- Add content_types_provided/2 to cufp_wm_resource:

```
content_types_provided(ReqData, State) ->
    {[{"text/html", to_html},
      {"text/plain", to_text}],
     ReqData, State}.
```

- Copy cufp_wm_resource1.erl to cufp_wm/src/cufp_wm_resource.erl and recompile

Allowing text/plain

- Now add the to_text/2 resource function:

```
to_text(ReqData, State) ->
    {"Hello, new world", ReqData, State}.
```

Now It Works

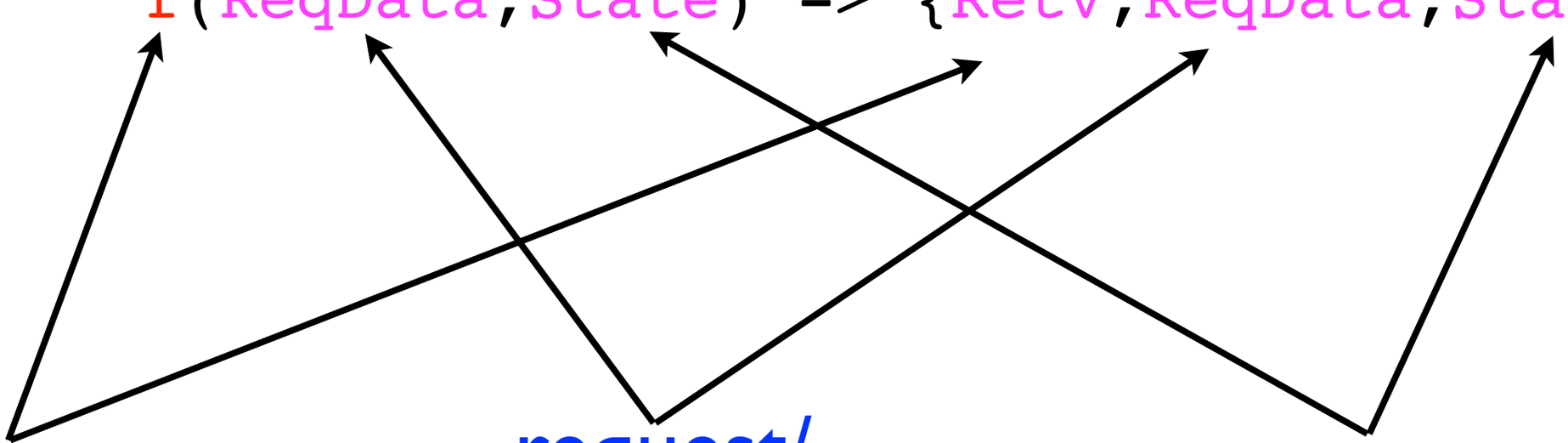
```
$ curl -D /dev/tty -H 'Accept: text/plain' \
http://localhost:8000

HTTP/1.1 200 OK
Vary: Accept
Server: MochiWeb/1.1 WebMachine/1.9.0
Date: Mon, 10 Sep 2012 00:44:48 GMT
Content-Type: text/plain
Content-Length: 16

Hello, new world
```

Resource Functions

- WM provides over 30 resource functions
- All have the same form:

```
f(ReqData,State) -> {RetV,ReqData,State}.
```

function behavior + request/response data + request state



Resource Functions

```
        to_html(ReqData,State) -> {Body,ReqData,State}.
  generate_etag(ReqData,State) -> {ETag,ReqData,State}.
  last_modified(ReqData,State) -> {Time,ReqData,State}.
resource_exists(ReqData,State) -> {bool,ReqData,State}.
  is_authorized(ReqData,State) -> {bool,ReqData,State}.
            ...
```

- WM's resource functions all have reasonable defaults
- App resources supply resource functions wherever they need to supply non-default behavior

basho

Request and Response Bodies

- The wrq module also provides functions for dealing with request and response bodies
- Apps can deal with bodies as
  - binaries, or
  - streamed chunks of data

wrq:req_body/1

- wrq:req_body/1 returns incoming request body as binary

wrq:stream_req_body/2

- wrq:stream_req_body/2 returns incoming request body as chunks of data in {binary, Next} pairs
- if Next == done, end
- otherwise, Next is a fun/0, so call it to get the next chunk

wrq:set_resp_body/2

- wrq:set_resp_body/2 sets the response body.
- It can take an iolist as a whole body
- Or a {stream, Next} pair where Next is either
  - {iolist, fun/0}, where fun/0 returns another Next
  - {iolist, done} to signal last chunk

Body Example

- Copy cufp_wm_resource2.erl to cufp_wm/src/cufp_wm_resource.erl and recompile
- First, the top of the file is unsurprising:

```
-module(cufp_wm_resource).
-export([init/1, allowed_methods/2, process_post/2]).
-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.
```

basho

Allowing POST

- Next, we have to allow the POST HTTP method with this resource function:

```
allowed_methods(ReqData, State) ->
    {['POST'], ReqData, State}.
```

Processing POST

- POSTs are handled by the process_post/2 resource function

Processing POST

```
process_post(ReqData, State) ->
```

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
```

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
    Body = get_streamed_body(Body0, []),
```

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
    Body = get_streamed_body(Body0, []),
    RespBody = send_streamed_body(Body,4),
```

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
    Body = get_streamed_body(Body0, []),
    RespBody = send_streamed_body(Body,4),
    NewReqData = wrq:set_resp_body(
      {stream, RespBody}, ReqData),
```

basho

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
    Body = get_streamed_body(Body0, []),
    RespBody = send_streamed_body(Body,4),
    NewReqData = wrq:set_resp_body(
      {stream, RespBody}, ReqData),
    {true, NewReqData, State}.
```

basho

Processing POST

```
process_post(ReqData, State) ->
    Body0 = wrq:stream_req_body(ReqData, 3),
    Body = get_streamed_body(Body0, []),
    RespBody = send_streamed_body(Body,4),
    NewReqData = wrq:set_resp_body(
      {stream, RespBody}, ReqData),
    {true, NewReqData, State}.
```

basho

get_streamed_body/2

```
get_streamed_body({Hunk,done},Acc) ->
    io:format("RECEIVED ~p~n",[Hunk]),
    iolist_to_binary(lists:reverse([Hunk|Acc]));
```

get_streamed_body/2

```
get_streamed_body({Hunk,done},Acc) ->
    io:format("RECEIVED ~p~n",[Hunk]),
    iolist_to_binary(lists:reverse([Hunk|Acc]));

get_streamed_body({Hunk,Next},Acc) ->
    io:format("RECEIVED ~p~n",[Hunk]),
    get_streamed_body(Next(),[Hunk|Acc]).
```

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
```

end.

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
        <<Hunk:Max/binary, Rest/binary>> ->
            io:format("SENT ~p~n",[Hunk]),
```

end.

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
        <<Hunk:Max/binary, Rest/binary>> ->
            io:format("SENT ~p~n",[Hunk]),
            F = fun() ->
                    send_streamed_body(Rest,Max)
                end,

end.
```

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
        <<Hunk:Max/binary, Rest/binary>> ->
            io:format("SENT ~p~n",[Hunk]),
            F = fun() ->
                    send_streamed_body(Rest,Max)
                end,
            {Hunk, F};




    end.
```

basho

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
        <<Hunk:Max/binary, Rest/binary>> ->
            io:format("SENT ~p~n",[Hunk]),
            F = fun() ->
                    send_streamed_body(Rest,Max)
                end,
            {Hunk, F};
        _ ->
            io:format("SENT ~p~n",[Body]),
            {Body, done}
    end.
```

basho

send_streamed_body/2

```
send_streamed_body(Body, Max) ->
    case Body of
        <<Hunk:Max/binary, Rest/binary>> ->
            io:format("SENT ~p~n",[Hunk]),
            F = fun() ->
                    send_streamed_body(Rest,Max)
                end,
            {Hunk, F};
        _ ->
            io:format("SENT ~p~n",[Body]),
            {Body, done}
    end.
```

basho

Try It

```
$ curl -D /dev/tty -d 1234567890 \
http://localhost:8000/

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Server: MochiWeb/1.1 WebMachine/1.9.0
Date: Mon, 10 Sep 2012 02:25:24 GMT
Content-Type: text/html

1234567890
```

Printed in the Console

```
RECEIVED <<"123">>
RECEIVED <<"456">>
RECEIVED <<"789">>
RECEIVED <<"0">>
SENT <<"1234">>
SENT <<"5678">>
SENT <<"90">>
```

WebMachine Trace

- Go back to src/cufp_wm_resource.erl and change init/1 to:

```
init([]) -> {{trace, "/tmp"}, undefined}.
```

- Or copy cufp_wm_resource3.erl to cufp_wm/src/cufp_wm_resource.erl and recompile

WebMachine trace

- Go to the WM console and type:

  wmtrace_resource:add_dispatch_rule("wmtrace", "/tmp").

- Now again access

  http://localhost:8000

WebMachine Trace

- Now visit

  http://localhost:8000/wmtrace

- It shows the path through the HTTP flowchart your request took
- Mousing over certain coordinates in the path show which resource functions were called at that point

More Examples

- See the Wriaki example for a bigger WebMachine example
  - https://github.com/basho/wriaki
  - http://steve.vinoski.net/blog/internet-computing-columns/#2012-1
- Also take a look at Riak, which uses WebMachine for its HTTP interface
  - https://github.com/basho/riak

Summary

- WebMachine is a framework shaped like HTTP
- Excellent for building Web services conforming to HTTP
- Helpful for RESTful Web services
- Very different from typical web frameworks: no templates, not focused on client UI


basho

Yaws

Quick Glance

- Yaws is a full-feature web server, offering quite a few features
- Can be used embedded or stand-alone
- It's an 11 year old open source project created by Claes "Klacke" Wikström
- Still under active development/maintenance
- Klacke still works on it, I became a committer in 2008, and Christopher Faulet became a committer in 2012

Installation

rebar compile

- This builds a self-contained server
- Yaws also supports autoconf/configure/make/make install, but we'll use rebar for this tutorial

Run the Server

bin/yaws -i

- Now access http://localhost:8000. You'll be prompted for user/passwd, which is user "foo" and passwd "bar"
- This also gives you an Erlang console

Project Layout

```
$ ls
ChangeLog                   debian                      rebar.config.script
LICENSE                     deps                        rel
Makefile                    doc                         scripts
README                      ebin                        src
README.pkg-config           etc                         ssl
README.win32-cygwin         examples                    test
_build.cfg                  include                     two-mode-mode.el
applications                include.mk.in               var
bin                         install-sh                  vsn.mk
c_src                       known_dialyzer_warnings     win32
config.guess                man                         www
config.sub                  munin                       yaws.pc.in
configure.in                priv                        yaws.rel.src
contrib                     rebar.config                yaws_logs
```

basho

Features

- Yaws has lots of features...

- HTTP 1.1
- URL/#arg rewriting
- appmods
- SSL support
- cookie/session support
- munin stats
- CGI and FCGI
- forward and reverse proxies
- file upload
- WebDAV
- small file caching
- SOAP support
- haXe support
- ehtml and exhtml
- virtual directories
- configurable deflate
- ACLs
- precompressed static files
- configurable MIME types
- .yaws pages
- virtual servers
- yapps
- JSON and JSON-RPC 2.0
- Server-Sent Events
- WebSocket
- GET/POST chunked transfer
- streaming
- multipart/mime support
- file descriptor server (fdsrv)
- server-side includes
- heart integration
- both make and rebar builds
- man pages
- LaTex/PDF documentation

Features

- Covering them all would be a full-day tutorial
- We'll try to hit a few of the highlights

Configuration

- For a stand-alone Yaws, server config comes from the yaws.conf file
  - see etc/yaws/yaws.conf in the tutorial code
- For an embedded Yaws, your app provides configuration via proplists

Global Config

- Settings that affect the whole Yaws server
- Erlang runtime paths
- Include file paths for code compiled on the fly
- Internal file cache settings
- Etc.

Virtual Servers

- A single Yaws server can host multiple virtual servers
- For example, running Yaws as directed earlier runs 3 virtual servers

Server Config

- Settings on a per-virtual-server basis
- IP address and port for listening
- Document root
- Dispatch rules
- Authorization settings
- SSL settings
- Etc.

Edit the Config

- Let's change the yaws.conf to get rid of the user/passwd on the localhost server
- Edit etc/yaws/yaws.conf, find the server localhost block, and remove the <auth> ... </auth> block
- Also change the docroot to be the same as the docroot of the server block above the localhost, should end in yaws/www
- Then Ctrl-g q <enter> in the console to quit Yaws, then restart it

Regular File Serving

- Yaws serves files from each virtual server's document root
- Extensive support for matching file suffixes to MIME types
- Yaws was the first Erlang web server to use OS sendfile for fast efficient file serving
  - the file:sendfile capability introduced in Erlang/OTP R15 is based on the Yaws sendfile driver (and Yaws uses it by default when run on R15)

out/1 Function

- A number of Yaws features are based on modules providing an out/1 function
- Yaws calls this when appropriate, passing an #arg record
- The #arg holds everything known about the current request

.yaws Pages

- You can mix Erlang into your HTML in a .yaws page

```
<html>
  <erl>
    out(_) ->
        %% return HTML here.
  </erl>
</html>
```

- Yaws replaces the <erl>...</erl> with the result of the out/1 function

html.yaws

```
<erl>
out(_) ->
    {html,
     "<p>Hello from html</p>"}.
</erl>
```

- You can have multiple <erl> blocks in the same .yaws file
- Each block has its own out/1 function
- Each block can contain other Erlang functions as well (helper functions)

HTML Support

- {html, iolist} lets you return a deep list of HTML made of characters, strings, binaries or nested lists thereof
- {ehtml, ErlangTerms} lets you define HTML using Erlang terms
  - easier to read
  - compiler checked
  - no forgetting closing tags

ehtml.yaws

```
<erl>
out(_) ->
    {ehtml, [{p, [], "Hello from ehtml"}]}.
</erl>
```

Application Modules (appmods)

- An "appmod" is an app-specific module that
- exports an out/1 function
- is registered in config to serve a particular URL path

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").
```

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").

out(#arg{client_ip_port={IP, _}}) ->
```

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").

out(#arg{client_ip_port={IP, _}}) ->
    IPStr = inet_parse:ntoa(IP),
```

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").

out(#arg{client_ip_port={IP, _}}) ->
    IPStr = inet_parse:ntoa(IP),
    Json = {struct, [{"clientIP", IPStr}]},
```

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").

out(#arg{client_ip_port={IP, _}}) ->
    IPStr = inet_parse:ntoa(IP),
    Json = {struct, [{"clientIP", IPStr}]},
    [{status, 200},
     {content, "application/json", json2:encode(Json)}].
```

Example: ex1.erl

```
-module(ex1).
-export([out/1]).
-include_lib("yaws_api.hrl").
```

```
out(#arg{client_ip_port={IP, _}}) ->
    IPStr = inet_parse:ntoa(IP),
    Json = {struct, [{"clientIP", IPStr}]},
    [{status, 200},
     {content, "application/json", json2:encode(Json)}].
```

Run ex1

- Compile ex1:

  erlc -I../yaws/include -o ../yaws/ebin ex1.erl

- In etc/yaws/yaws.conf add ex1 as an appmod. Add

  appmods = <ex1, ex1>

  to the localhost server config

- Then run "bin/yaws --hup" in a separate shell to force Yaws to reload its config

Access ex1

- Access

http://localhost:8000/ex1

and you'll see the JSON response

Serving "/"

- You can register an appmod to handle the URL path "/"
- This means it handles all requests sent to that server
- You can also exclude subpaths, e.g. for static file handling

```
appmods = </, myappmod exclude_paths icons js static>
```

Streaming

- Sometimes you have to deliver content in chunks
  - avoid reading a huge file into memory all at once to form a response body
  - or maybe you don't have the whole content yet, e.g. live video
  - or maybe you're using long-polling (Comet)

Streaming

- Yaws supports easy streaming of content
  - either using chunked transfer encoding
  - or regular non-chunked transfer

Chunked Transfer

- out/1 returns

  {streamcontent, MimeType, FirstChunk}

- Yaws then expects an app process to send chunks to it
- It encodes these for chunked transfer and sends them to the client

stream I .yaws

- A regular .yaws file
- The out/1 fun will return streamcontent
- Loop over a binary, delivering it in chunks

stream l .yaws

<erl>

</erl>

stream I .yaws

```
<erl>
out(A) ->
    YawsPid = A#arg.pid,

</erl>
```

stream l .yaws

```
<erl>
out(A) ->
    YawsPid = A#arg.pid,
    spawn(fun() ->

          end),

</erl>
```

stream l .yaws

```
<erl>
out(A) ->
    YawsPid = A#arg.pid,
    spawn(fun() ->
                B = <<"0123456789>>,
                Bin = list_to_binary([B,B,B,B]),
                loop(YawsPid, Bin, 10)
          end),


</erl>
```

stream l .yaws

```
<erl>
out(A) ->
    YawsPid = A#arg.pid,
    spawn(fun() ->
                B = <<"0123456789>>,
                Bin = list_to_binary([B,B,B,B]),
                loop(YawsPid, Bin, 10)
          end),
    {streamcontent,
     "application/octet-stream", <<"ABCDEF">>}.
</erl>
```

stream l .yaws

```
<erl>
out(A) ->
    YawsPid = A#arg.pid,
    spawn(fun() ->
                B = <<"0123456789>>,
                Bin = list_to_binary([B,B,B,B]),
                loop(YawsPid, Bin, 10)
          end),
    {streamcontent,
     "application/octet-stream", <<"ABCDEF">>}.
</erl>
```

stream I .yaws

```
loop(YawsPid, <<>>, _) ->
    yaws_api:stream_chunk_end(YawsPid);
```

stream I .yaws

```
loop(YawsPid, <<>>, _) ->
    yaws_api:stream_chunk_end(YawsPid);
loop(YawsPid, Bin, ChunkSize) ->
    case Bin of
```

end.

stream l .yaws

```
loop(YawsPid, <<<>>>, _) ->
    yaws_api:stream_chunk_end(YawsPid);
loop(YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>>> ->
            yaws_api:stream_chunk_deliver(YawsPid, Chunk),
            loop(YawsPid, Rest, ChunkSize);


end.
```

stream | .yaws

```
loop(YawsPid, <<<>>>, _) ->
    yaws_api:stream_chunk_end(YawsPid);
loop(YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>> ->
            yaws_api:stream_chunk_deliver(YawsPid, Chunk),
            loop(YawsPid, Rest, ChunkSize);
        _ ->
            yaws_api:stream_chunk_deliver(YawsPid, Bin),
            loop(YawsPid, <<>>, ChunkSize)
    end.
```

stream l .yaws

```
loop(YawsPid, <<>>, _) ->
    yaws_api:stream_chunk_end(YawsPid);
loop(YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>> ->
            yaws_api:stream_chunk_deliver(YawsPid, Chunk),
            loop(YawsPid, Rest, ChunkSize);
        _ ->
            yaws_api:stream_chunk_deliver(YawsPid, Bin),
            loop(YawsPid, <<>>, ChunkSize)
    end.
```

basho

Try It With Netcat

- We'll use the netcat client (/usr/bin/nc) so we can see the chunked transfer
- First, copy stream1.yaws to your docroot directory

Try It With Netcat

```
$ nc localhost 8000 < stream1.get
HTTP/1.1 200 OK
Server: Yaws 1.94
Date: Tue, 11 Sep 2012 15:47:32 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked

6
ABCDEF
12
012345678900123456
12
789001234567890012
8
34567890
0
```

Non-Chunked Streaming

- Similar to chunked streaming, except different Yaws API calls
- Streaming process waits for a message from Yaws to proceed
- To avoid chunked transfer, always need to remember to either include a Content-Length header, or erase the Transfer-Encoding header

stream2.yaws

<erl>

</erl>

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
```

```
</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
              receive
                  end
              end),

</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
              receive
                  {ok,YawsPid} ->
                      First = <<"ABCDEF">>,
                      B = <<"01234567890">>,
                      Bin = list_to_binary([First,B,B,B,B]),
                      loop(Sock, YawsPid, Bin, 18);


              end
          end),


</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
                receive
                    {ok,YawsPid} ->
                        First = <<"ABCDEF">>,
                        B = <<"01234567890">>,
                        Bin = list_to_binary([First,B,B,B,B]),
                        loop(Sock, YawsPid, Bin, 18);
                {discard,YawsPid} ->
                    yaws_api:stream_process_end(Sock,YawsPid)
                end
            end),


</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
              receive
                  {ok,YawsPid} ->
                      First = <<"ABCDEF">>,
                      B = <<"01234567890">>,
                      Bin = list_to_binary([First,B,B,B,B]),
                      loop(Sock, YawsPid, Bin, 18);
                  {discard,YawsPid} ->
                      yaws_api:stream_process_end(Sock,YawsPid)
              end
          end),
    [{header, {transfer_encoding, erase}},

</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
              receive
                  {ok,YawsPid} ->
                      First = <<"ABCDEF">>,
                      B = <<"01234567890">>,
                      Bin = list_to_binary([First,B,B,B,B]),
                      loop(Sock, YawsPid, Bin, 18);
                  {discard,YawsPid} ->
                      yaws_api:stream_process_end(Sock,YawsPid)
              end
          end),
     [{header, {transfer_encoding, erase}},
      {streamcontent_from_pid, "application/octet-stream", Pid}].
</erl>
```

stream2.yaws

```
<erl>
out(A) ->
    Sock = A#arg.clisock,
    Pid = spawn(fun() ->
              receive
                  {ok,YawsPid} ->
                      First = <<"ABCDEF">>,
                      B = <<"01234567890">>,
                      Bin = list_to_binary([First,B,B,B,B]),
                      loop(Sock, YawsPid, Bin, 18);
                 {discard,YawsPid} ->
                      yaws_api:stream_process_end(Sock,YawsPid)
              end
          end),
     [{header, {transfer_encoding, erase}},
      {streamcontent_from_pid, "application/octet-stream", Pid}].
</erl>
```

stream2.yaws

```
loop(Sock, YawsPid, <<<>>>, _) ->
    yaws_api:stream_process_end(Sock, YawsPid);
```

stream2.yaws

```
loop(Sock, YawsPid, <<<>>>, _) ->
    yaws_api:stream_process_end(Sock, YawsPid);
loop(Sock, YawsPid, Bin, ChunkSize) ->
    case Bin of
```

end.

stream2.yaws

```
loop(Sock, YawsPid, <<<>>>, _) ->
    yaws_api:stream_process_end(Sock, YawsPid);
loop(Sock, YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>> ->
            yaws_api:stream_process_deliver(Sock, Chunk),
            loop(Sock, YawsPid, Rest, ChunkSize);


end.
```

basho

stream2.yaws

```
loop(Sock, YawsPid, <<>>, _) ->
    yaws_api:stream_process_end(Sock, YawsPid);
loop(Sock, YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>> ->
            yaws_api:stream_process_deliver(Sock, Chunk),
            loop(Sock, YawsPid, Rest, ChunkSize);
        _ ->
            yaws_api:stream_process_deliver(Sock, Bin),
            loop(Sock, YawsPid, <<>>, ChunkSize)
end.
```

stream2.yaws

```
loop(Sock, YawsPid, <<>>, _) ->
    yaws_api:stream_process_end(Sock, YawsPid);
loop(Sock, YawsPid, Bin, ChunkSize) ->
    case Bin of
        <<Chunk:ChunkSize/binary, Rest/binary>> ->
            yaws_api:stream_process_deliver(Sock, Chunk),
            loop(Sock, YawsPid, Rest, ChunkSize);
        _ ->
            yaws_api:stream_process_deliver(Sock, Bin),
            loop(Sock, YawsPid, <<>>, ChunkSize)
    end.
```

Try It With Netcat

- First, copy stream2.yaws to your docroot directory

```
$ nc localhost 8000 < stream2.get
HTTP/1.1 200 OK
Connection: close
Server: Yaws 1.94
Date: Tue, 11 Sep 2012 16:04:08 GMT
Content-Type: application/octet-stream

ABCDEF0123456789001234567890012345678900123456789
```

Socket Access

- Streaming processes can also write directly to the Yaws socket if they wish
- Yaws yields control of socket to streamcontent_from_pid processes
- Yaws regains socket controls when you call yaws_api:stream_process_end/2

Long Polling

- HTTP is a client-server request-response protocol
- "Long polling" or "Comet" is a way of simulating events sent from server to client

Long Polling

- Client sends request
- Server just holds the connection open until it has something — an event — to send
- It eventually replies with the event
- If client wants more events, it sends another request

basho

Long Polling

- streamcontent_from_pid makes long polling easy
- Just have your streaming process wait until it has an event before sending anything back to Yaws

Server-Sent Events

- Server-Sent Events (SSE) is a W3C working draft for standardizing how servers send events to clients
- See http://www.w3.org/TR/eventsource/
- Supported by Chrome, Safari, Firefox, Opera

SSE Example

- Take a look at the file

  yaws/examples/src/server_sent_events.erl

- This uses SSE to keep a clock ticking on the client
- Visit http://yaws.hyber.org/server_sent_events.html to see it in action

SSE Example Details

- Built on top of streamcontent_from_pid
- Client must send a GET with Accept header set to "text/event-stream"
- Module uses a gen_server to keep track of the client socket, the Yaws pid, and a timer

SSE Example Details

- The yaws_sse:headers/1 call (line 37) returns the following to Yaws:

```
[{status, 200},
 {header, {"Cache-Control", "no-cache"}},
 {header, {connection, "close"}},
 {header, {transfer_encoding, erase}},
 {streamcontent_from_pid, "text/event-stream", SSEPid}].
```

basho

SSE Example Details

- Every time the timer ticks (every second), the gen_server:
  - gets a local server date string and formats it SSE data via yaws_sse:data/1 (line 63)
  - sends that string as an event on the client socket using yaws_sse:send_events/2 (line 64)

SSE Example Details

- On the client, a little chunk of JavaScript handles the updates
- See file yaws/www/server_sent_events.html

```
<script type="text/javascript">
    var es = new EventSource('/sse');
    es.onmessage = function(event) {
        var h2 = document.querySelector('#date-time');
        h2.innerHTML = event.data;
    }
</script>
```

basho

WebSocket

- The WebSocket Protocol is RFC 6455
- The HTTP connection is "upgraded" to WebSocket using Upgrade HTTP headers
- WebSocket has a bit of framing but is essentially a bidirectional TCP connection

Yaws WebSocket Support

- An out/1 function indicates it wants WebSocket by returning

  {websocket, CallbackModule, Options}

- Once the connection is upgraded, Yaws calls handle_message/1 or handle_message/2 on the callback module

WebSocket Example

- Point your browser at

  http://localhost:8000/websockets_example.yaws

- Then, see file

  yaws/examples/src/basic_echo_callback.erl

Basic Echo

- This file supplies a number of handle_message/1 clauses
- Uses pattern matching to deal with different "commands" sent from the browser UI

Basic Echo

```
handle_message({text, <<"bye">>}) ->
    io:format("User said bye.~n", []),
    {close, normal};
```

Basic Echo

```
handle_message({text, <<"something">>}) ->
    io:format("Some action without a reply~n", []),
    noreply;
```

Basic Echo

```
handle_message({text, <<"say hi later">>}) ->
    io:format("saying hi in 3s.~n", []),
    timer:apply_after(3000, ?MODULE, say_hi, [self()]),
    {reply, {text, <<"I'll say hi in a bit...">>}};
```

Basic Echo

```
say_hi(Pid) ->
    io:format("asynchronous greeting~n", []),
    yaws_api:websocket_send(Pid, {text, <<"hi there!">>}).
```

Basic Echo

```
handle_message({text, Message}) ->
    io:format("basic echo handler got ~p~n", [Message]),
    {reply, {text, <<Message/binary>>}};
```

Basic Echo

```
handle_message({binary, Message}) ->
    {reply, {binary, Message}};
```

Yaws Summary

- Yaws supplies a lot of features
- Known for great reliability, stability, and performance
- All features were driven by actual production usage
- Even though it's 11 years old, we continue to actively develop and maintain Yaws

basho

For More Info

- http://yaws.hyber.org
- http://github.com/klacke/yaws
- Yaws mailing list: https://lists.sourceforge.net/lists/listinfo/erlyaws-list
- http://steve.vinoski.net/blog/internet-computing-columns/#2011-4


basho

Nitrogen

Quick Glance

- Nitrogen runs on multiple web servers: Mochiweb, Yaws, inets, Webmachine, Cowboy
- Event-driven ala AJAX
- Builds on top of jQuery
- Originally created by Rusty Klophaus

Installation

make <web server target>

- Web server targets are rel_yaws, rel_mochiweb, rel_webmachine, rel_inets, rel_cowboy
- Generates a self-contained Nitrogen system under rel/nitrogen

Run the Server

```
cd rel/nitrogen
bin/nitrogen console
```

- Starts Nitrogen listening with an interactive Erlang console
- Try http://localhost:8000

Release Contents

$ ls -1 rel/nitrogen

BuildInfo.txt	# version information
Makefile	
bin	# runtime tools
erts-5.9.1	# erlang runtime
etc	# config files
lib	# erlang applications
log	# logfiles
rebar	
rebar.config	
releases	# erlang release files
site	# website pages

Site Contents

$ ls -1a rel/nitrogen/site

.prototypes	# page, action, and
	# element prototypes
ebin	# compiled modules
include	# include files
src	# module sources
static	# static web pages
templates	# config files

Nitrogen Pages

- A page corresponds to an Erlang module
- Supplies a main/0 function to provide a page template
- Typically also supplies other functions to populate the template and handle events

Request Routing

- Any URL ending with an extension is assumed to be a static file
- Any request to "/" is handled by page index.erl
- Otherwise, request is routed by matching URL path to module name

Request Routing

- Change "/" in URL path to underscores
- Look for matching module name
- For example:

  /products/12345 => products_12345

Longest Match

- If no exact match is found, Nitrogen looks for the longest match
- For example:

  /products/12345 => products

- Allows one module to handle a "family" of web resources

Generating a Page

bin/dev page foo
make
bin/dev compile

- Creates page in site/src/foo.erl
- Template for page creation is under site/.prototypes/page.erl
- "make" and "bin/dev compile" compile and load the new foo page into the running Nitrogen server

Page Sync

- In the Nitrogen server console, type sync:go().

- This tells Nitrogen to watch for changed sources. It will recompile and reload any changes.

main/0

- Open site/src/foo.erl

```
main() ->
  #template {
    file="./site/templates/bare.html"
  }.
```

Templates

- Page template with callouts to page modules
- Page module callout functions fill in the spots in the template they're called from
- Templates live under site/templates, you can add your own
- main/0 supplies the page template
- Let's open site/templates/bare.html

Callouts

[[[page:body()]]]

- "page" in this context always refers to the current module
- This calls out to body/0 function in the current module
- Return value of body/0 replaces [[[...]]] in the template
- Can also call out to other modules M:F(Args)

body/0

- Returns one or more Nitrogen elements
- Nitrogen renders these into HTML and JavaScript

Nitrogen Elements

- Elements are instances of Erlang records
- Look at site/src/foo.erl again

```
body() ->
    [
        #panel { style="margin: 50px 100px;", body=[
            #span { text="Hello from foo.erl!" },
            #p{},
            #button { text="Click me!", postback=click },
            #p{},
            #panel { id=placeholder }
        ]}
    ].
```

basho

Nitrogen Elements

- Elements correspond to HTML elements
- Element fields correspond to HTML attributes
- Elements allow you to define pages in Erlang — more readable, and with compiler checking

Element Attributes

- id
  - the id of the element
  - Erlang atom
  - used to refer to the element elsewhere

Element Attributes

- actions
  - an action or list of actions
  - we'll cover actions next

Element Attributes

- show_if
  - if true, the element is rendered

Element Attributes

- class
  - CSS class of the element
    - can be atom or string
- style
  - inline CSS style string

Element Attributes

- Individual element types can also have their own element-specific attributes

Element Types

- Layout (e.g. panel, span, table)
- HTML (e.g. heading, paragraph, link)
- Forms (e.g. button, textarea)
- Other (e.g. draggable, droppable)

Element Details

- See http://nitrogenproject.com/doc/elements.html for element details

Nitrogen Actions

- Actions ultimately render to JavaScript
- Nitrogen uses the jQuery library underneath to implement some actions (see http://jquery.com)

Action Example

- Copy Nitrogen example foo1.erl to site/src/foo.erl, then recompile
- We've changed body/0 to:

```
body() ->
    [
     #button {
        text="Submit",
        actions=#effect {
            effect=shake,
            options=[{times,3}]
        } }
    ].
```

Try It

- Visit http://localhost:8000/foo
- You should see a shaking button

Action Types

- Page manipulation (e.g. event)
- Effects (e.g. show, fade, toggle)
- Feedback (e.g., alert)

Action Attributes

- module
- set automatically
- module that contains logic of the action

Action Attributes

- trigger
  - id of the element that triggers this action
- target
  - id of the element affected by this action

Action Attributes

- actions
  - a list of actions nested within this action
- show_if
  - if true, this action is rendered

Action Attributes

- Actions can also have action-specific attributes

Action Details

- See http://nitrogenproject.com/doc/actions.html for action details

Action Wiring

- The wf:wire functions allow easy setting of actions
- Easier to read than setting in "actions" attribute

wf:wire/N

- wf:wire/1 sets an action for the page
- wf:wire/2 sets an action for a specific target, with page as trigger
- wf:wire/3 sets an action for a specific target with a specific trigger

Wiring Example

```
body() ->
    Effect = #effect {
        effect=shake,
        options=[{times,3}] },
```

Wiring Example

```
body() ->
    Effect = #effect {
          effect=shake,
          options=[{times,3}] },
    wf:wire(button1, Effect),
```

Wiring Example

```
body() ->
    Effect = #effect {
          effect=shake,
          options=[{times,3}] },
    wf:wire(button1, Effect),
    [
     #button { id=button1,
               text="Submit" }
    ].
```

Wiring Example

```
body() ->
    Effect = #effect {
          effect=shake,
          options=[{times,3}] },
    wf:wire(button1, Effect),
    [
     #button { id=button1,
                    text="Submit" }
    ].
```

Wiring Example

- Copy foo2.erl to site/src/foo.erl, recompile and try it

http://localhost:8000/foo

Click to Shake

- Previous example shakes the button on page load
- What if we want to shake the button by clicking it?

Click to Shake

- We'll start with foo2.erl below, and edit
- Or, just copy foo3.erl to site/src/foo.erl and recompile

```
body() ->
    Effect = #effect {
        effect=shake,
        options=[{times,3}] },
    wf:wire(button1, Effect),
    [
     #button { id=button1,
               text="Submit" }
    ].
```

basho

Click to Shake

```
body() ->
    Shake = #effect {
          effect=shake,
          options=[{times,3}] },
    Effect = #event { type=click,
                      actions=Shake },
    wf:wire(button1, Effect),
    [
     #button { id=button1,
               text="Shake It" }
    ].
```

Trigger and Target

```
body() ->
    Shake = #effect {
          effect=shake,
          options=[{times,3}] },
    Effect = #event { type=click,
                      actions=Shake },
    wf:wire(button1, lbl1, Effect),
    [
     #label { id=lbl1, text="Shake Me!" },
     #button { id=button1,
               text="Shake It" }
    ].
```

basho

Trigger and Target

```
body() ->
    Shake = #effect {
          effect=shake,
          options=[{times,3}] },
    Effect = #event { type=click,
                      actions=Shake },
    wf:wire(button1, lbl1, Effect),
    [
     #label { id=lbl1, text="Shake Me!" },
     #button { id=button1,
               text="Shake It" }
    ].
```

basho

The Original foo.erl

```
body() ->
    [
        #panel { style="margin: 50px 100px;", body=[
            #span { text="Hello from foo.erl!" },
            #p{},
            #button { text="Click me!",
                      postback=click },
            #p{},
            #panel { id=placeholder }
        ]}
    ].
```

basho

The Original foo.erl

```
body() ->
    [
        #panel { style="margin: 50px 100px;", body=[
            #span { text="Hello from foo.erl!" },
            #p{},
            #button { text="Click me!",
                      postback=click },
            #p{},
            #panel { id=placeholder }
        ]}
    ].
```

basho

Postbacks

- Actions are handled on the client
- Postbacks are handled on the server, by Erlang event/1 functions in the page module
- Arguments to event/1 can be any Erlang term
- You can have multiple event/1 clauses to handle different terms to distinguish different events

Postback Handler

```
body() ->
    #button { text="Submit",
              postback=click }.

event(click) ->
    ?PRINT({click, calendar:local_time()}).
```

Try It

- Copy foo4.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Click the button and look at your Nitrogen console

Postback ?PRINT Output

```
=INFO REPORT===== 1-Aug-2012::20:46:34 ===
DEBUG: foo:24
"{ click , calendar : local_time ( ) }"
  {click,{{2012,8,1},{20,46,34}}}
```

Multiple Handlers

```
body() ->
    [#button { text="Button 1",
               postback=click1 },
     #button { text="Button 2",
               postback=click2 }].

event(click1) ->
    ?PRINT({click1, calendar:local_time()});
event(click2) ->
    ?PRINT({click2, calendar:local_time()}).
```

basho

Try It

- Copy foo5.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Click the buttons, check the console

Read From Elements

```
body() ->
    [ #p { text="Type something here:" },
      #textbox { id=tb },
      #button { text="Submit",
                postback=click }].

event(click) ->
    TextYouTyped = wf:q(tb),
    ?PRINT({click, TextYouTyped}).
```

Read From Elements

```
body() ->
    [ #p { text="Type something here:" },
      #textbox { id=tb },
      #button { text="Submit",
                postback=click }].

event(click) ->
    TextYouTyped = wf:q(tb),
    ?PRINT({click, TextYouTyped}).
```

Try It

- Copy foo6.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Type some text then submit, and then check the console

Updating Elements

```
body() ->
    [ #textbox { id=tb, text="1" },
      #button { text="Increment",
                postback=click }].
```

Updating Elements

```
body() ->
    [ #textbox { id=tb, text="1" },
      #button { text="Increment",
                postback=click }].

event(click) ->
    Count = list_to_integer(wf:q(tb)),
```

Updating Elements

```
body() ->
    [ #textbox { id=tb, text="1" },
      #button { text="Increment",
                postback=click }].

event(click) ->
    Count = list_to_integer(wf:q(tb)),
    NewCount = integer_to_list(Count+1),
```

basho

Updating Elements

```
body() ->
    [ #textbox { id=tb, text="1" },
      #button { text="Increment",
                postback=click }].

event(click) ->
    Count = list_to_integer(wf:q(tb)),
    NewCount = integer_to_list(Count+1),
    wf:set(tb, NewCount).
```

Updating Elements

```
body() ->
    [ #textbox { id=tb, text="1" },
      #button { text="Increment",
                postback=click }].

event(click) ->
    Count = list_to_integer(wf:q(tb)),
    NewCount = integer_to_list(Count+1),
    wf:set(tb, NewCount).
```

Try It

- Copy foo7.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Click the button and watch the updates

Update Functions

- wf:update/2
- wf:replace/2
- wf:remove/1
- wf:insert_top/2
- wf:insert_bottom/2
- wf:set/2
- wf:flash/1 (flash a message into #flash element)

Update Efficiency

- Nitrogen uses AJAX techniques to update individual page elements

Validation

- In previous update example, what if textbox is changed to contain something other than a number?
- Page module event handler crashes
  - Try changing the number to a word, click the button, then check the console
- We should validate the textbox contents

Validator

```
body() ->
    wf:wire(btn, tb, #validate {
        validators=[ #is_integer { text="not an integer!" } ]}),
    [ #textbox { id=tb, text="1" },
      #button { id=btn, text="Increment",
                postback=click }].

event(click) ->
    Count = list_to_integer(wf:q(tb)),
    NewCount = integer_to_list(Count+1),
    wf:set(tb, NewCount).
```

Try It

- Copy foo8.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Change the number to a word, then click the button

Validators

- Required, Min Length, Max Length
- Is Email, Is Integer, Confirm Passwords Match
- Custom client-side validators (written in JavaScript)
- Custom server-side validators (written in Erlang)
- See http://nitrogenproject.com/doc/validators.html for details

Long-polling, a.k.a. Comet

- Long-polling allows a server to simulate pushing messages to the client
- Client sends a request, server receives it
  - if there's something to reply with, the server replies
  - if nothing yet, the server just sits on the request and delays the reply

Nitrogen Comet

- wf:comet/1 spawns a function
- Spawned function can update elements on the page whenever it wants to, as long as the client stays on the page

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].

update_tb(Val) ->
    timer:sleep(2000),
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].
```

```
update_tb(Val) ->
    timer:sleep(2000),
    NewVal = Val+1,
    wf:set(tb, integer_to_list(NewVal)),
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].
```

```
update_tb(Val) ->
    timer:sleep(2000),
    NewVal = Val+1,
    wf:set(tb, integer_to_list(NewVal)),
    wf:flush(),
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].
```

```
update_tb(Val) ->
    timer:sleep(2000),
    NewVal = Val+1,
    wf:set(tb, integer_to_list(NewVal)),
    wf:flush(),
    update_tb(NewVal).
```

Updating with Comet

```
body() ->
    Val = 1,
    wf:comet(fun() -> update_tb(Val) end),
    [ #textbox { id=tb, text=integer_to_list(Val) } ].

update_tb(Val) ->
    timer:sleep(2000),
    NewVal = Val+1,
    wf:set(tb, integer_to_list(NewVal)),
    wf:flush(),
    update_tb(NewVal).
```

Try It

- Copy foo9.erl to site/src/foo.erl and recompile
- Access http://localhost:8000/foo
- Watch the page auto-update

And Much More

- Page and session state
- Comet pools
- Authentication and authorization
- Custom validation
- Custom elements and actions
- Handlers
- Demos, see http://nitrogenproject.com/demos
- And more!
- See http://nitrogenproject.com


basho

Chicago Boss

Quick Glance

- Inspired by Ruby on Rails
- Model-View-Controller (MVC)
- Tie MVC apps to different URL spaces, host them all in the same server
- Full-stack framework
- Lots of code generation
- Created by Evan Miller

basho

Installation

make

- This builds Chicago Boss itself
- To generate an application:

make app PROJECT=<project name>

Run the Server

- Let's say the project name is cufp:

```
make app PROJECT=cufp
cd ../cufp
./init-dev.sh
```

- This starts a development server listening on port 8001, with console and code auto-recompile

Project Contents

$ ls -1F

boss.config	# CB config file
ebin/	
include/	
init-dev.sh	# start dev server
init.sh	# start production server
log/	
priv/	# various server files
rebar	
rebar.cmd	
rebar.config	
src/	# source files
start-server.bat	

basho

Project Contents

```
$ cd src
$ ls -1F
```

controller/	# controller sources
cufp.app.src	# source for app file
lib/	
mail/	
model/	# model sources
test/	
view/	# view sources
websocket/	

Controllers and Actions

- CB dispatches incoming requests to a controller
- A controller groups actions

Request Routing

- URL http://host/<C>/<A> is handled by
  - controller C
  - (which provides)
  - action A

Request Routing

- For example, controller module for http://host/ex1/hello_world is src/controller/cufp_ex1_controller.erl
- And hello_world is an action in that module
- Additional routes can be added to priv/cufp.routes file

Controller Module

```
-module(cufp_ex1_controller, [Req]).
-export([hello_world/2]).

hello_world('GET', []) ->
    {output, "Hello, World!"}.
```

Try It

- Copy cufp_ex1_controller.erl to src/controller
- CB should notice the new module and automatically compile & load it
- Access http://localhost:8001/ex1/hello_world

Controller Module

```
-module(cufp_ex1_controller, [Req]).
-export([hello_world/2]).

hello_world('GET', []) ->
    {output, "Hello, World!"}.
```

- It's a parameterized module, so Req is available to each function
- Req is a request object, provides access to query params, post data, HTTP headers, cookies, etc.

Action Arguments

- First argument: HTTP method
- Second arg: list of any URL path segments following the action name, split on "/"

Action Arguments

```
-module(cufp_ex2_controller, [Req]).
-export([hello_world/2]).

hello_world('GET', Path) ->
    Result = "Hello, world: " ++
        string:join(Path, ", "),
    {output, Result}.
```

Action Arguments

- Copy cufp_ex2_controller.erl to src/controller
- Accessing

  http://localhost:8001/ex2/hello_world/other/path/segments

- Yields

  Hello, world: other, path, segments

Action Results

- {output, String} — returns HTML
- {json, Proplist} — returns JSON
- {ok, Proplist} — passes Proplist to a template
- {redirect, URL} — redirects to new location
- {stream, GeneratorFunction, Acc} — streams response to client using chunked transfer
- not_found — returns HTTP status 404 Not Found
- and more, see http://www.chicagoboss.org/api-controller.html
- (a proplist or "property list" is a list of {key, value} tuples)

Models

- A model abstracts application data
- CB provides BossRecords and BossDB to manage and model persistent data

BossRecords

- Like Ruby on Rails ActiveRecord
- Specially-compiled parameterized modules

BossRecord Example

- Copy song1.erl to src/model/song.erl
- It has the following code:

```
-module(song, [Id, Title, Artist,
               Album, Year, Genre]).
-compile(export_all).
```

- That's all. No functions! But note all the module parameters

Song Module Params

- Id: all BossRecords must have an Id parameter. It must be a string
- Title: song title (string)
- Artist: song artist (string)
- Album: name of album song is from (string)
- Year: the year the song was released (string, could alternatively be an integer)
- Genre: type of music the song represents (string)

basho

BossRecord Example

- Now, either "./rebar compile" or refresh the ex2/hello_world browser page
- Either action will compile the song.erl model, and the CB server will load it

BossRecord Example

- Next, switch to the CB console
- At the console prompt, enter the command: m(song).
- You'll see the song module exports 20 functions — where'd they come from?
- All but two of these are generated by CB, many based on the module parameters

Creating Song Instances

- We'll create a bunch of song instances from a JSON music library file
- Let's look at musiclib.erl

musiclib.erl

```
-module(musiclib).
-export([load/0]).

load() ->
    {ok, Bin} = file:read_file("music-library.js"),
    {array, Songs} = mochijson:decode(Bin),
    load_songs(Songs).
```

- Read JSON file into binary
- Decode JSON binary
- Load the song data

basho

load_songs/1

```
load_songs([{struct, Attrs}|Songs]) ->
  FoldFun =
    fun(Key, Acc) ->
      {value, {Key, Value}} =
        lists:keysearch(Key, 1, Attrs),
      [Value | Acc]
    end,
```

- The fold fun just pulls attributes from each song item into a list

load_songs/1

```
ParamKeys = ["genre", "year",
      "album", "artist", "title", "id"],
Params = lists:foldl(FoldFun,  [],
                     ParamKeys),
```

- ParamKeys is list of song module parameters, in reverse order
- The fold pulls values for those params out of each song item

load_songs/1

```
Song = apply(song, new, Params),
Song:save(),
load_songs(Songs);
```

- The apply/3 call calls song:new/6 with those params to create a new song
- Song:save() saves the new Song in BossDB
- call load_songs/1 recursively with tail of song list

load_songs/1

```
load_songs([]) ->
    ok.
```

- When song list is empty, end recursion

Loading the Songs

- At the CB console prompt:

```
> pwd().
/some/path/to/cufp
ok
```

- Copy musiclib.erl and music-library.js to that location

Loading the Songs

- Next, at the CB console prompt:

```
> c(musiclib).
{ok, musiclib}
> musiclib:load().
ok
> boss_db:count(song).
105
```

boss_db

- You can use the boss_db module to work with your data. For example:

```
> Song = boss_db:find("7479").
{song,"7479","D Minor Blues",
      "Derek Trucks Band",
      "The Derek Trucks Band",[],"Blues"}
> Song:artist().
"Derek Trucks Band"
```

boss_db Queries

```
> Steve = boss_db:find(song, [{artist,
                matches, "^Steve"}]).
[{song,"8948",
"Woman of 1,000 Years","Steve Stevens",
    "Atomic Playboys","1989","Rock"}]
```

- A variety of queries are possible, see http://www.chicagoboss.org/api-db.html

CB Templates

- Define how pages look on the client
- Based on the Django framework's template language, see

  https://docs.djangoproject.com/en/dev/topics/templates/

- Templates live under

  src/view/<controller-name>

Songs Controller

- Lives in

  src/controller/cufp_songs_controller.erl

- Copy cufp_songs_controller1.erl to src/controller/cufp_songs_controller.erl

Songs Controller

```
-module(cufp_songs_controller, [Req]).
-export([list/2]).
```

Songs Controller

```
-module(cufp_songs_controller, [Req]).
-export([list/2]).

list('GET', []) ->
    Songs = boss_db:find(song, []),
```

Songs Controller

```
-module(cufp_songs_controller, [Req]).
-export([list/2]).

list('GET', []) ->
    Songs = boss_db:find(song, []),
    {ok, [{songs, Songs}]}.
```

Songs Controller

```
-module(cufp_songs_controller, [Req]).
-export([list/2]).

list('GET', []) ->
    Songs = boss_db:find(song, []),
    {ok, [{songs, Songs}]}.
```

Song List Template

- Lives in

  src/view/songs/list.html

Song List Template

```
<html>
  <head>
    <title>
     {% block title %}
     Song List
     {% endblock %}
    </title>
  </head>
```

Song List Template

```
<body>
    {% block body %}
        {% if songs %}
```

Song List Template

```
<body>
    {% block body %}
        {% if songs %}
        <ul>
    {% for song in songs %}
    <li>{{ song.title }}</li>
    {% endfor %}
</ul>
```

Song List Template

```
<body>
    {% block body %}
        {% if songs %}
        <ul>
    {% for song in songs %}
    <li>{{ song.title }}</li>
    {% endfor %}
</ul>
```

Song List Template

```
{% else %}
<p>No songs!</p>
{% endif %}
{% endblock %}
</body>
```

Song List

- mkdir src/view/songs
- Copy list1.html to src/view/songs/list.html
- Access

http://localhost:8001/songs/list

- With more template work, you could show all attributes of each song
- And add pagination, etc.

Adding Songs

- Adding a song requires
  - presenting a form to the user
  - handling POSTed form data in the controller
  - validating the data
  - storing the new song

Presenting a Form

- Could write a whole new template
- Or just extend the list template
- First, we need a "create" link

Add a Create Link

- Change src/view/songs/list.html or copy list2.html to src/view/songs/list.html:

```
<body>
    {% block body %}
    <p>
<a href="{% url action='create' %}">
Add a New Song
</a></p>
    {% if songs %}
```

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}
```

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
```

```
{% endblock %}
```

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">



</form>

{% endblock %}
```

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">
<label for="title">Title:</label>
<input type="text" id="title" name="title"><br/>


</form>

{% endblock %}
```

basho

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">
<label for="title">Title:</label>
<input type="text" id="title" name="title"><br/>
<!-- ...same for other fields... -->

</form>

{% endblock %}
```

basho

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">
<label for="title">Title:</label>
<input type="text" id="title" name="title"><br/>
<!-- ...same for other fields... -->
<input type="submit">
</form>

{% endblock %}
```

basho

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">
<label for="title">Title:</label>
<input type="text" id="title" name="title"><br/>
<!-- ...same for other fields... -->
<input type="submit">
</form>
<p><a href="{% url action='list' %}">View Song List</a></p>
{% endblock %}
```

basho

create.html

```
{% extends "songs/list.html" %}
{% block title %}Add a Song{% endblock %}

{% block body %}
<p>Add a song:</p>
<form method="post">
<label for="title">Title:</label>
<input type="text" id="title" name="title"><br/>
<!-- ...same for other fields... -->
<input type="submit">
</form>
<p><a href="{% url action='list' %}">View Song List</a></p>
{% endblock %}
```

create.html

- Copy create1.html to src/view/songs/create.html

Add create Action

```
create('POST', []) ->
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
          end,
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
          end,
    Args = lists:foldl(Fun,  [], Params),
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
        end,
    Args = lists:foldl(Fun,  [], Params),
    Song = apply(song, new, [id|Args]),
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
        end,
    Args = lists:foldl(Fun,  [], Params),
    Song = apply(song, new, [id|Args]),
    {ok, _} = Song:save(),
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
        end,
    Args = lists:foldl(Fun,  [], Params),
    Song = apply(song, new, [id|Args]),
    {ok, _} = Song:save(),
    {redirect, [{action, "list"}]};
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
        end,
    Args = lists:foldl(Fun,  [], Params),
    Song = apply(song, new, [id|Args]),
    {ok, _} = Song:save(),
    {redirect, [{action, "list"}]};
```

Add create Action

```
create('POST', []) ->
    Params = ["genre", "year", "album", "artist", "title"],
    Fun = fun(Param, Acc) ->
                [Req:post_param(Param)|Acc]
        end,
    Args = lists:foldl(Fun,  [], Params),
    Song = apply(song, new, [id|Args]),
    {ok, _} = Song:save(),
    {redirect, [{action, "list"}]};
create('GET', []) ->
    ok.
```

Add a Song

- Copy cufp_songs_controller2.erl to src/controller/cufp_songs_controller.erl
- Access

  http://localhost:8001/songs/create
- Or visit

  http://localhost:8001/songs/list

  and click the "Add" link at the top

Check the New Song

- In the CB console:

  boss_db:find(song, [{title, equals, X}]).

  where X is the title you entered

- Note the song ID, which was generated by CB because we passed the atom 'id' for the song module Id parameter

Validation

- What if we want to make sure the user doesn't leave any fields empty?
- Or make other checks on the fields?
- Answer: add validation tests to the src/model/song.erl model

validation_tests/0

- Optional function, returns a list of {TestFunction, Message}

  tuples to validate the BossRecord

- Any test function failures prevents saving the item

validation_tests/0

```
validation_tests() ->
    [{fun() -> length(Title) > 0 end,
      "Title must not be empty"},
     {fun() -> length(Artist)  > 0 end,
      "Artist must not be empty"},
    {fun() -> length(Album) > 0 end,
     "Album must not be empty"},
    {fun() -> length(Year) > 0 end,
     "Year must not be empty"},
    {fun() -> length(Genre) > 0 end,
     "Genre must not be empty"}].
```

Controller Validation

- We have to change this code, which doesn't handle errors:

```
{ok, _} = Song:save(),
```

Controller Validation

```
case Song:save() of

end;
```

Controller Validation

```
case Song:save() of
    {ok, _} ->
        {redirect, [{action, "list"}]};


end;
```

Controller Validation

```
case Song:save() of
    {ok, _} ->
        {redirect, [{action, "list"}]};
    {error, Errors} ->
        {ok, [{errors, Errors}, {song, Song}]}
end;
```

basho

Controller Validation

```
case Song:save() of
    {ok, _} ->
        {redirect, [{action, "list"}]};
    {error, Errors} ->
        {ok, [{errors, Errors}, {song, Song}]}
end;
```

basho

Enhance the Create Template

- Display errors
- Reuse existing fields so the user doesn't have to retype everything

Display the Errors

```
{% block body %}
 {%if errors %}
  <p>You must fill in all fields!</p>
  <ul>
  {% for err in errors %}
   <li>{{ err }}</li>
  {% endfor %}
  </ul>
{% endif %}
```

Avoid Field Retyping

```
<input type="text" id="title" name="title"
{% if song %} value="{{ song.title }}"
{% endif %}
><br/>
```

basho

Try It

- Copy create2.html to src/view/songs/create.html
- Copy song2.erl to src/model/song.erl
- Copy cufp_songs_controller3.erl to src/controller/cufp_songs_controller.erl

Try It

- Now, refresh you browser and navigate to the Add Song page
- Try hitting "Submit" without filling in any fields
- You should see errors listed

Try It

- Try filling in some fields but not all and hitting "Submit" again
- It should show you some errors but preserve the fields you've already entered

Could Also Add

- Show details of a specific song (hint: make title a link)
- Edit song details
- A search menu
- Delete a song

And Much More

- BossNews — events and notifications, letting you watch data changes and act
- BossMQ — channel-based messaging, supports Comet
- WebSockets
- Sessions
- Sending and receiving email
- Test framework for testing your web apps
- See http://www.chicagoboss.org

The End.