Extracted from:

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
and Fred Daoud

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
Fred Daoud

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93778-563-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2014

Day 2: Building Apps

Yesterday we saw most of the pieces we'll need to build a link shortener, and today we'll put those pieces together to build the first version of it, called Petite.

Today we'll also look at basic front-end tasks with Webmachine and related libraries as we build a web UI for Petite. You'll see how to integrate HTML templating into a Webmachine resource with mustache.erl,[3] a Mustache template implementation for Erlang.

Webmachine makes it easy to handle different types of incoming data to support both human and API use of the same resource. You'll discover that representations of incoming data and the resource itself are important parts of Webmachine.

Shortening Links

We'll apply what we learned yesterday to build the first iteration of Petite, our link shortener. This first version will be able to shorten links and redirect incoming visitors to the corresponding real URLs.

First, create a new Webmachine project called `petite`. Starting from this shell, we'll keep expanding Petite as we go along.

Compression and Storage

Before we can write our Webmachine resource for Petite's shortening API, we must have (1) a way to shorten the link and (2) a lookup table that associates shortened codes with their corresponding URLs. Erlang contains built-in tools to help with both of these problems.

If you want to make a string of digits shorter, one easy trick is to write it in a larger numeric base. For example, `10000000` in binary becomes `128` in decimal and `3K` in base 36. Using this, we can attach a number to each real URL, incrementing the number each time. The short code returned can just be the number represented in a high numeric base so that it's as compact as possible. Erlang can convert to different numeric bases up to base 36 with `integer_to_list(Number, Base)`.

Storing the lookup table can be done a number of ways, but the easiest is to use an ETS (short for Erlang Term Storage) table. An ETS table is an in-memory key value store that is built into the Erlang standard library. You

3. https://github.com/mojombo/mustache.erl

can store arbitrary Erlang tuples in it, and the first element becomes the key and the whole tuple is the value.

Putting these two pieces together, we can write a gen_server module, which is a self-contained Erlang service that produces codes given URLs and returns URLs given codes. There's not enough room to fully explain gen_servers here, which are part of Erlang's standard library; if you're interested, see *Erlang Programming [CT09]* by Francesco Cesarini and Simon Thompson or Joe Armstrong's book, *Programming Erlang [Arm13]*. Let's just look at the important bits:

webmachine/petite/day1/petite/src/petite_url_srv.erl
```
-module(petite_url_srv).

%% public API
-export([start_link/0,
         get_url/1,
         put_url/1]).

-behaviour(gen_server).
-export([init/1,
         terminate/2,
         code_change/3,
         handle_call/3,
         handle_cast/2,
         handle_info/2]).

-define(SERVER, ?MODULE).
-define(TAB, petite_urls).

-record(st, {next}).

%% public API implementation

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

1 get_url(Id) ->
    gen_server:call(?SERVER, {get_url, Id}).

put_url(Url) ->
    gen_server:call(?SERVER, {put_url, Url}).

%% gen_server implementation

2 init(_) ->
    ets:new(?TAB, [set, named_table, protected]),
    {ok, #st{next=0}}.
```

```
terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

handle_call({get_url, Id}, _From, State) ->
    Reply = case ets:lookup(?TAB, Id) of
                [] ->
                    {error, not_found};
                [{Id, Url}] ->
                    {ok, Url}
            end,
    {reply, Reply, State};

handle_call({put_url, Url}, _From, State = #st{next=N}) ->
    Id = b36_encode(N),
    ets:insert(?TAB, {Id, Url}),
    {reply, {ok, Id}, State#st{next=N+1}};

handle_call(_Request, _From, State) ->
    {stop, unknown_call, State}.

handle_cast(_Request, State) ->
    {stop, unknown_cast, State}.

handle_info(_Info, State) ->
    {stop, unknown_info, State}.

%% internal functions

b36_encode(N) ->
   integer_to_list(N, 36).
```

1. The public API of this module simply delegates to the server process. This is common for gen_server implementations, since the server itself may later change the internal message formats.

2. We initialize the server by creating a new ETS table to store the codes and their corresponding URLs, and we start the counter at 0.

   Webmachine threads the State variable returned from a resource's init function through all the resource functions, and gen_server does the same. You can see where Webmachine got the idea.

3. Retrieving a URL is a simple matter of looking it up in the ETS table.

4. Putting a URL into the server creates a code and then inserts a corresponding entry. Notice that it increments the counter in the returned state.

5. Creating a code is as easy as integer_to_list, at least if you don't need anything higher than base 36.

gen_servers are usually attached to a supervisor process that ensures they keep running and restarts them when they crash. In order to use petite_url_srv, we must add it to Petite's main supervisor, petite_sup. The following highlighted lines inside the init function show the modifications that are needed:

webmachine/petite/day1/petite/src/petite_sup.erl

```
  Web = {webmachine_mochiweb,
         {webmachine_mochiweb, start, [WebConfig]},
         permanent, 5000, worker, [mochiweb_socket_server]},
➤ UrlServer = {petite_url_srv,
➤              {petite_url_srv, start_link, []},
➤              permanent, 5000, worker, []},
➤ Processes = [Web, UrlServer],
  {ok, { {one_for_one, 10, 10}, Processes} }.
```

Compile and start Petite, and let's play with our new service at the Erlang shell. Note that if you don't see the 1> prompt after you start the app, hit Enter to make one appear.

```
$ ./start.sh
«omitted output»
=PROGRESS REPORT==== 15-May-2013::21:10:14 ===
         application: petite
          started_at: nonode@nohost
1> whereis(petite_url_srv).
<0.92.0>
2> petite_url_srv:put_url("https://pragprog.com/").
{ok,"0"}
3> petite_url_srv:put_url("https://github.com/basho/webmachine").
{ok,"1"}
4> petite_url_srv:get_url("1").
{ok,"https://github.com/basho/webmachine"}
5> petite_url_srv:get_url("3K").
{error,not_found}
```

Our service works, and it is ready for use by any Webmachine resources we create.

Redirection

You've already learned how to create Webmachine resources and how to use resource functions like resource_exists and moved_permanently to redirect HTTP requests. You also saw how to bind path tokens to atoms during dispatch and retrieve them with wrq:path_info. All that remains is to combine these with petite_url_srv, and Petite can shorten links.

First, create a rule in priv/dispatch.conf for your new resource:

webmachine/petite/day1/petite/priv/dispatch.conf
```
{[code], petite_fetch_resource, []}.
```

Then create the petite_fetch_resource module. Try modifying the redirection example in *Working with Resource Functions, on page ?*, to use petite_url_srv before peeking at the following implementation:

webmachine/petite/day1/petite/src/petite_fetch_resource.erl
```
-module(petite_fetch_resource).
-export([init/1,
         to_html/2,
         resource_exists/2,
         previously_existed/2,
         moved_permanently/2]).

-include_lib("webmachine/include/webmachine.hrl").

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

to_html(ReqData, State) ->
    {"", ReqData, State}.

resource_exists(ReqData, State) ->
    {false, ReqData, State}.

previously_existed(ReqData, State) ->
    Code = wrq:path_info(code, ReqData),
    case petite_url_srv:get_url(Code) of
        {ok, Url} ->
            {true, ReqData, Url};
        {error, not_found} ->
            {false, ReqData, State}
    end.

moved_permanently(ReqData, State) ->
    {{true, State}, ReqData, State}.
```

Recompile Petite and add some links at the Erlang shell as before. Once it has shortened a few links, you can test the resource:

```
$ curl -i http://localhost:8000/1
HTTP/1.1 301 Moved Permanently
Server: MochiWeb/1.1 WebMachine/1.9.2
Location: https://github.com/basho/webmachine
Date: Thu, 16 May 2013 03:29:09 GMT
Content-Type: text/html
Content-Length: 0
```

Our link shortener is working but is still short a few features. We need an HTTP API to shorten new links. For that, we'll create a new resource, petite_shorten_resource.

Shortening API

The API to shorten a link is simple. HTTP POST requests will include form data with a url field set to the link to shorten. Petite will return the shortened link as text in the response.

Let's think about the first questions Webmachine asks our resource and how our resource should answer them. First, we'll need to answer allowed_methods by indicating support for HTTP POST. Next, since our response will be text, the resource must respond appropriately to content_types_provided and provide to_text. Webmachine requires we provide a body-generating function even in the case of POSTs where it's not strictly needed.

So far, these are resource functions that you've seen before when processing HTTP GET requests. For HTTP POST requests, Webmachine first calls post_is_create to determine if this request creates a new resource. If the answer is false, the Webmachine state machine delegates processing to process_post. If the answer is true, Webmachine follows a path of inquiry in the state machine that we don't have room to cover in this chapter. Since Petite is not creating new resources in this API call, it will follow the former path.

process_post must parse the form data in the request, shorten the link, and then generate a suitable response. Let's look at how this is done:

webmachine/petite/day1/petite/src/petite_shorten_resource.erl

```
-module(petite_shorten_resource).
-export([init/1,
         allowed_methods/2,
         process_post/2,
         content_types_provided/2,
         to_text/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) ->
    {ok, undefined}.
allowed_methods(ReqData, State) ->
    {['POST'], ReqData, State}.

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

process_post(ReqData, State) ->
    Host = wrq:get_req_header("host", ReqData),
    Params = mochiweb_util:parse_qs(wrq:req_body(ReqData)),
    Url = proplists:get_value("url", Params),
    {ok, Code} = petite_url_srv:put_url(Url),
    Shortened = "http://" ++ Host ++ "/" ++ Code ++ "\n",
    {true, wrq:set_resp_body(Shortened, ReqData), State}.

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

wrq:get_req_header returns the value of a request header. Here it's used to retrieve the host and port the client is connected to so that we can use that information to build the final shortened link.

mochiweb_util:parse_qs is a function that parses query strings or form data. This is provided by MochiWeb, which is the HTTP processing library that Webmachine—and most other Erlang web libraries—are built on top of. We have provided it wrq:req_body(ReqData) as input, which is the extracted body of the incoming request.

mochiweb_util:parse_qs returns an Erlang property list, and process_post grabs the url property, sends it to the internal shortening service, and then builds a new, shortened link.

Finally, wrq:set_resp_body is used to set the body of the response to the shortened link. Since data in Erlang is immutable, wrq:set_resp_body returns an altered version of the ReqData structure that is passed along. Returning true from process_post indicates successful processing.

Petite will also need a new dispatch rule for this resource. Put this rule before the petite_fetch_resource rule:

webmachine/petite/day1/petite/priv/dispatch.conf

```
{["shorten"], petite_shorten_resource, []}.
```

You can now rebuild Petite and test it out:

```
$ curl -i -X POST http://localhost:8000/shorten \
> --data 'url=https%3A%2F%2Fpragprog.com%2F'
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.9.2
Date: Fri, 17 May 2013 04:56:25 GMT
Content-Type: text/plain
Content-Length: 24

http://localhost:8000/0

$ curl -i http://localhost:8000/0
HTTP/1.1 301 Moved Permanently
Server: MochiWeb/1.1 WebMachine/1.9.2
```

```
Location: https://pragprog.com/
Date: Fri, 17 May 2013 04:57:03 GMT
Content-Type: text/html
Content-Length: 0
```

Petite can now shorten long links and redirect shortened links to their original URLs. Developers have written link shorteners in many languages and frameworks, but it's hard to imagine a simpler implementation than this Webmachine version. By modeling HTTP as a state machine and separating decision logic from simple answers, Webmachine has made dealing with redirection almost as simple as "Hello, World."

Even with just this basic functionality, it could serve as an internal shortening service for your own web applications. Of course, you'd probably want to persist the lookup table in a production version.

While Petite is working, it doesn't yet have any front end for human users of the service. Let's look at how Webmachine handles front-end tasks so we can remedy this situation.