

Erlang For The Practical Man

Jarosław Rzeszółtko, sztywny@gmail.com, May 2007

1: General overview

Functional programming languages are often underestimated, as they seem to be not suited for the “real world”, being thought of as designed by scientists who spend their time mainly developing theories, not practical software. What makes Erlang stand out from the crowd is the fact it was designed by the industry and for the industry. Ericsson, where the language originated, has a telecom switch with more than 2 million lines of Erlang code running it that has a downtime of a few minutes per year - now this is reliability!

Telecom switches don't make programmers too excited those days through, so we will try to do something more related to your everyday hacking. In this tutorial I will show you how to write a standalone daemon, that checks your email every x minutes, and makes the results available as an RSS feed. If you want to centralize all your notifications around RSS this may even be useful a bit and even if not, at least it makes a good example of some of Erlang strengths. I try to explain everything in as much detail as possible, but be warned - we won't be writing any “hello worlds” here, we will just rapidly introduce a lot of awkward stuff as it becomes necessary, so if you don't have a solid programming background you may have serious trouble. Either way, you will probably want to consult [Programming Erlang](#) and the [free, official Erlang docs](#) if you want to do anything serious in Erlang. Also see [my previous post](#) for an overview of general high level Erlang concepts.

Let's start with an overview of the parts that we will need. One component will have to connect to a POP3 server once in a while, grab the list of messages (or message titles) and make a nice list out of it. We don't want it to interfere in any way with accessing the feed, so it will be a separate process. We'll also need a function that given a simple list of messages, returns a well formed RSS feed created from it. When we have the feed, we want to have a simple HTTP server running that will simply send the feed to anyone who will make a GET. Of course there should be a possibility to update the feed the server is sending, when new mail arrives or simply when we check it the next time. As we said before, serving the feed should be independent from checking email, so it will be another process - and as we allow multiple connections to the server at the same time (maybe you will want to show you email to the world one day, who knows?), it will spawn a new request handling process for each accepted TCP connection, so multiple requests will be handled concurrently. Thinking in terms of processes and concurrent activities in the system is the core of Erlang programming paradigm and is the essence of learning the language. The cool part is that in Erlang modeling software as multiple processes cooperating finally becomes a natural, seamless abstraction and you don't have to worry all the time about low-level stuff like deadlocks.

2: Making an RSS

The data structures we will use are rather basic - we represent messages as tuples and store them in a list. A tuple simply groups a bunch of variables into a single entity. Our program will use tuples to represent messages - the first field will be the “message” symbol used to emphasize the meaning of the gathered data, then goes the date of arrival, sender, subject and body. Here are examples of our messages using Erlang syntax:

```
{message, “12:00 01-10-1987”, “Foo Bar”, “Hello Bar!”, “Nice to meet you”}
```

```
{message, “16:32 05-05-2006”, “whoever”, “Some subject”, “whatever”}
```

You can get to individual tuple fields using the element function:

```
1> element(2, {abc, def, ghi}).  
def
```

We want to store at least the 10 most recent messages, so we have to group them somehow. Erlang has a powerful list abstraction in the spirit of Lisp or Scheme, so it should suit our needs. A list in Erlang syntax looks like that:

```
[{message, "12:00 01-10-1987", "Foo Bar", "Hello Bar!", "Nice to meet you"},
 {message, "16:32 05-05-2006", "whoever", "Some subject", "whatever"}]
```

From the components we outlined, creating the feed from a supplied list of messages seems to be the simplest task to do. Just put the various message field into corresponding XML fields and wrap it with some more XML on the beginning and on the end.

```
-module(rss_wrap).
-compile(export_all).

-define(prologue(Date),
    "<?xml version=\"1.0\"?>
    <rss version=\"2.0\">
    <channel>
    <title>GMailRSS</title>
    <description>Your mailbox in RSS using Erlang.</description>
    <language>en-us</language>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>RssGmail</generator>" ++
    "<pubDate>" ++ Date ++ "</pubDate>").

-define(epilogue,
    "</channel>
    </rss>").

escape_from(From) ->
    Left = element(2, regexp:gsub(From, "<", "(")),
    Right = element(2, regexp:gsub(Left, ">", ")")),
    Right.

rss_wrap(MsgList) ->
    case MsgList of
    [] ->
        CurrTime = erlang:universaltime(),
        Date = httpd_util:rfc1123_date(CurrTime);
    List ->
        Date = element(2, lists:last(MsgList)),
    end,

    ?prologue(Date) ++ rss_wrap_list(MsgList) ++ ?epilogue.

rss_wrap_list(MsgList) ->
    WrapItem = fun(Item, Result) -> Result ++ rss_wrap_item(Item) end,
    lists:foldl(WrapItem, "", MsgList).

rss_wrap_item({message, Date, From, Subject, Content}) ->
    "<item>" ++
    "<title>" ++ Subject ++ "</title>" ++
    "<description>" ++
    "From: " ++ escape_from(From) ++
    Content ++
    "</description>" ++
    "<pubDate>" ++ Date ++ "</pubDate>" ++
    "</item>".
```

The “-module” declaration gives you a namespace for the functions you will define in the file and it must be the same as the filename in which the code is stored (but without the “.erl”). “-compile(export_all)” - tells the Erlang compiler to export all function automatically, so that they become globally available. In production you would like to export each of them manually, but while you are developing something it’s convenient not too change the exports everytime you add a function. “-define” lets you define constants, which you can later refer to with a preceding quotation mark. You can also use it to write something similar to C macros - like in the prologue definition. “++” is string concatenation, so the prologue will contain the beginning of the feed’s with the publication date embedded somewhere inside.

The main function “rss_wrap” checks if the message list is empty and either sets the date to the current time using the matching RFC’s syntax, or to the arrival date of the last message in the list. Later it returns the result - the concatenation of the XML “prologue” with the date embedded, the result of the rss_wrap_list function call and the XML “epilogue”. Note that consecutive statements in functions are separated by commas, and that each definition ends with a dot. We will cover the details of the “case” construction later on.

The rss_wrap_list function shows one of the most common used idioms in any serious programming language. What foldl does (it is also present in Lisp, and in Ruby as “inject”) is accumulating all the elements of a list into a single variable in a way defined by the user. In Erlang you first define a fun (which is simply a function with no name bound to it, that’s why we have to assign it to something or pass to a function immediately) taking two parameters, one representing the element of the list that is currently being processed and the second one being the accumulator. The fun has to return the new value of the accumulator after processing one element. We give that fun to foldl, together with the initial value of the accumulator and with the actual list. If you have trouble understanding it look at a simpler example first:

```
-module(test123).  
-compile(export_all).  
  
foldl_test() ->  
    ProcessedList = [1, 2, 3, 4, 5],  
    CollectFun = fun(ProcessedItem, Accumulator) -> Accumulator + ProcessedItem  
end,  
    lists:foldl(CollectFun, 0, ProcessedList).
```

We start the foldl with the initial accumulator value of 0. The first item of the array is 1 and the fun returns the accumulator summed with the item, so here it returns 1, which becomes the new value of the accumulator. Now the fun is called for the second item, which is 2, summing it up with the accumulator we get 3, which becomes the new accumulator. And so it goes until the end of the list - finally foldl returns the sum of all elements in the array.

In rss_wrap_list we do precisely the same thing, but we’re using a string accumulator and the rss_wrap_item function which wraps each message tuple in XML conforming to the RSS 2.0 standard. We also need to escape the “<” and “>” characters the From header may contain, so we introduce the escape_from function doing precisely that. We use “element” in it, because the gsub function returns a tuple with several fields, not only the string after the substitution.

One more thing: in the above examples we’re using assignment - keep in mind that in Erlang all “variables” are immutable - once you define something, you can’t change its value. This may sound bad to you, but this is also one of the reasons concurrency in Erlang is easy.

Now we have one component ready. We can save it in a file named “rss_wrap.erl” and test it in the Erlang shell:

```
c(rss_wrap).  
rss_wrap:rss_wrap([{}], {}).
```

3: Fighting with POP3 / SSL

Now time for the tough part. We need to connect to the POP3 server and create the list we will wrap in the XML later on. This is about writing practical stuff, so we will use GMail’s POP3 as a testbed. This doesn’t make things easier, as Google requires you to use ssl. Could it be so hard with Erlang excellent libraries, will it be so hard? Lets see...

First of all, if you happen to not know anything about the POP3 protocol, consult the [suitable RFC](#). RFCs aren’t an easy read (well, [sometimes](#) they are), but hey, studying them will make you feel like a real hacker and may even turn you into one someday (if you really want it). Basically it’s a simple dialogue:

Client: [opens the socket]

Server: +OK I'm one cool pop3 server [Introduces himself, waits for authorization]

Client: USER foobar

Server: +OK Give me the password

Client: PASS whatever

Server: +OK [Gets into transaction mode]

When in transaction mode, you can send any number of commands you want, until a final QUIT. You can actually have this kind of conversation with your POP3 server using the fabulous netcat tool. Sadly, the standard netcat doesn't support SSL, so for gmail you will have to download something more fancy like socat or ncat. Or you could write a short app for chatting with remote hosts through SSL...

Anyway, let's see how the basic communication works. We will open a SSL socket, receive the initial POP3 greeting line, display it, close the socket and exit:

```
-module(pop3).
-compile(export_all).

fetch() ->
    application:start(ssl),
    {ok, SecureSock} = ssl:connect("pop.gmail.com", 995,
                                [{mode, list}, {packet, 0}, {active,
false}],),
    {ok, Response} = ssl:recv(SecureSock, 0),
    io:format("~s~n", [Response]),
    ssl:close(SecureSock).
```

What do those fancy statements looking like assignments with tuples mean? Well, this is pattern matching - yet another feature of many functional languages that didn't yet made it into the mainstream. If you look at the [docs for the ssl module](#) you will see that the ssl:connect function either returns a tuple where the first element is the symbol "ok", and the second the socket you just opened, or accordingly the "error" symbol and the cause of the error, also as symbol ie. "timeout". If everything goes good than the leftside "ok" will get matched with the rightside "ok" doing precisely nothing, and the leftside unbound variable "SecureSock" will get matched with the returned socket - assigning the socket to the variable. But if an error occurs, than "ok", will be matched with "error", the match will not succeed and you will get a runtime error from Erlang. Trust me, you really don't want to see an Erlang runtime error - they are mostly incomprehensible without the use of the Erlang debugger, which can at least show you the line the error occurred. You can avoid seeing them with writing your code in a fault-tolerant way - we could do error handling by matching the result of the call against more than one pattern, but this article is already getting long... Also, we should have been using pattern matching instead of the "element" function before - this way we make sure the tuple is of the format we're expecting.

Going back to pop3, the general pattern of communication will be not too surprisingly Request -> Response, so maybe we can capture a higher level abstraction... Actually, we also don't have a guarantee that the response will come in one portion (hence one ssl:recv), so we need to pull data out of the socket until we reach a CR/LF at the end:

```
request(Socket, Command) ->
    io:format("Request: ~s~n", [Response]),
    ssl:send(Socket, Command ++ "\n").

retrieve_line(Socket, Data) ->
    Length = length(Data),
    validP = (Length > 2) andalso (string:substr(Data, Length - 1, 2) ==
"\r\n"),
    if
        validP == true ->
            Data;
        true ->
            {ok, NewData} = ssl:recv(Socket, 0),
            retrieve_line(Socket, Data ++ NewData)
    end.
```

```

response_line(Socket) ->
    Response = retrieve_line(Socket, []),
    io:format("Response: ~s~n", [Response]),
    string:substr(Response, 1, string:len(Response) - 2).

```

The request method simply sends whatever you want through the socket, only adding a newline so that you don't have to type it every time.

Getting the response is a bit more complicated. Before checking the string ending for the CR/LF we need to check if the string is long enough - if the indices in `string:substr` are out of string bounds we will get a runtime error from Erlang. We do it using "andalso" - if the length is less than or equal to two the second statement will not be evaluated. Afterwards we actually check for the CR/LF, and if it is found, we return the data to the caller function. Otherwise, we try to get yet more data from the socket and we call the function we're actually in - note that nothing gets returned to the caller until the recursion ends. Also, the if statement works different a bit than we're used to - it does pattern matching in fact, that's why use "true" instead of "else", because "true" always matches.

`Receive_line` may look inconspicuous, but it illustrates one of the important "patterns" in Erlang - the way you deal with things that change over time. Normally you would use variables for that, but as we already said they are immutable. So in Erlang you use function parameters and recursive calls to deal with that. Actually, to represent a process, one also uses something that looks like a recursive function call running for very long.

Stepping back to our program, the `response_rn` method strips the final newline from the response and outputs it to the standard output - it's mainly a convenience wrapper. Notice that things in Erlang are numbered from one, not from zero like in C for example.

Now we can have the conversation we talked about:

```

fetch() ->
    application:start(ssl),
    {ok, SecureSock} = ssl:connect("pop.gmail.com", 995,
                                [{mode, list}, {packet, 0}, {active,
false}]),
    response(SecureSock),

    request(SecureSock, "USER foobar"),
    response(SecureSock),
    request(SecureSock, "PASS hello"),
    response(SecureSock),

    ssl:close(SecureSock).

```

Just remember to replace foobar with your gmail user name and hello with you password. Here comes the hardest part - we need to get the emails out of the POP3 and turn them into a nice list in the format we introduced on the very beginning. This sounds like pain to me. We will also need to take a look at another RFC, describing the format of the message returned by the server.

Generally, after authenticating we need to do a STAT. This will give us the number of messages in the mailbox. Than we need to make a RETR for each consecutive message number.

```

-module(pop3).
-compile(export_all).

```

```

response(Socket) ->
    {ok, Response} = ssl:recv(Socket, 0),
    io:format("Response: ~s", [Response]),
    Response.

```

```

request(Socket, Command) ->
    io:format("Request: ~s~n", [Command]),
    ssl:send(Socket, Command ++ "\n").

```

```

authenticate(Socket, Login, Pass) ->
    request(Socket, "USER " ++ Login),
    response(Socket),
    request(Socket, "PASS " ++ Pass),
    response(Socket).

do_stat(Socket) ->
    request(Socket, "STAT"),
    {ok, Tokens} = regexp:split(response(Socket), " "),
    {list_to_integer(lists:nth(2, Tokens)),
     list_to_integer(lists:nth(3, Tokens))}.

fetch() ->
    application:start(ssl),
    {ok, SecureSock} = ssl:connect("pop.gmail.com", 995,
                                  [{mode, list}, {packet, 0}, {active,
false}]),
    response(SecureSock),
    authenticate(SecureSock, "foobar", "barfoo"),
    {MessageCount, _} = stat(SecureSock),
    ssl:close(SecureSock).

```

I extracted the authentication into a separate method, and added the "stat" function. It makes a STAT request, which in response gives you something like this:

```
+OK 10 12345 Anything can be here
```

Where 10 is the number of messages and 12345 size of all of them. So what the stat function does is to split the response on spaces, extract the two fields we may care about, and convert them to integers. Strings in Erlang are just a list of integers (unfortunately), so the function list_to_integer is also used for converting strings to integer. The function nth from the lists module simply extracts the nth element from a list.

Now we need to make a "RETR" request for every message. It takes a number as an argument, where valid numbers go from one to the number STAT gave us. So "normally" we would probably use a for loop for this, but in Erlang iteration is often done in a different manner, take a look at the following code snippet:

```

retrieve_messages(Socket, From, To) when From < To ->
    [retr(Socket, From) | retrieve_messages(Socket, From+1, To)];

retrieve_messages(Socket, To, To) ->
    [retr(Socket, To)].

```

This may look like a nonsense, but you must know that Erlang does also pattern matching on function arguments. Thus, the second definition of "retrieve_messages" will only be called when the second and third argument is exactly the same, otherwise the first definition will be used. Syntactically, notice that multiple "versions" of the same function are separated with semicolons, the only exception being the last definition, which you end with a dot. Also, if you define a "retrieve_messages" with two arguments, it will be a completely different function.

If you've written functional programs before, you probably already know what the code above does, otherwise it may be a little bit hard to wrap your mind around this for the first time. What the first definition says is that the result of calling it is a list built from calling retr and from calling retrieve_messages with the same arguments, except the From parameter being increased by one. "|" is the basic list building thing, like "cons" from Lisp. This obviously is recursive, so the second call to retrieve_messages may also expand further. Let's say we call retrieve_messages(SomeSocket, 1, 3). It will go like this:

retrieve_messages(SomeSocket, 1, 3) returns:

```
[retr(Socket, 1) | retrieve_messages(Socket, 2, 3)]
```

But `retrieve_messages(Socket, 2, 3)` returns:

```
[retr(Socket, 2) | retrieve_messages(Socket, 3, 3)]
```

So the first call gets further expanded to:

```
[retr(Socket, 1), retr(Socket, 2) | retrieve_messages(Socket, 3, 3)]
```

`retrieve_messages(Socket, 3, 3)` matches the second definition, so the recursion ends, because we get "[retr(Socket, 3)]" from it. So the final expansion looks like this:

```
[retr(Socket, 1), retr(Socket, 2) | [retr(Socket, 3)]]
```

Which is evaluated simply to:

```
[retr(Socket, 1), retr(Socket, 2), retr(Socket, 3)]
```

Well, nobody said that this is going to be easy... Doing the actual RETR isn't all that fun either. More often than not, the message will be send in portions, so our response method isn't exactly valid anymore. As we can read in the RFC, a multiline response is terminated by the "\r\n.\r\n" string. So we can write another response function:

```
retrieve_multiline(Socket, Data) ->
  Length = string:len(Data),
  Ending = string:substr(Data, Length - 4, 5),
  ValidP = (Length > 5) andalso (Ending == "\r\n.\r\n"),
  if
    ValidP == true ->
      Data;
    true ->
      {ok, NewData} = ssl:recv(Socket, 0),
      retrieve_multiline(Socket, Data ++ NewData)
  end.

response_multiline(Socket) ->
  Response = retrieve_multiline(Socket, []),
  io:format("Response: ~s~n", [Response]),
  string:substr(Response, 1, string:len(Response) - 5).
```

But that is very repetitive. Let's rewrite both response functions in a more generic way:

```
retrieve(Socket, Data, Ending) ->
  DataLen = length(Data),
  EndingLen = length(Ending),
  ValidP = (DataLen > EndingLen) andalso
    (string:substr(Data, DataLen - EndingLen + 1, EndingLen) ==
Ending),
  if
    ValidP == true ->
      Data;
    true ->
      {ok, NewData} = ssl:recv(Socket, 0),
      retrieve(Socket, Data ++ NewData, Ending)
  end.

response(Socket, Ending) ->
  Response = retrieve(Socket, [], Ending),
  io:format("Response: ~s~n", [Response]),
  string:substr(Response, 1, string:len(Response) - length(Ending)).
```

```

response_line(Socket) ->
  response(Socket, "\r\n").

response_multiline(Socket) ->
  response(Socket, "\r\n\r\n").

```

Now we can retrieve every message in the mailbox:

```

fetch() ->
  application:start(ssl),
  {ok, SecureSock} = ssl:connect("pop.gmail.com", 995,
                                [{mode, list}, {packet, 0}, {active,
false}]),
  response(SecureSock),
  authenticate(SecureSock, "foobar", "barfoo"),
  {MessageCount, _} = stat(SecureSock),
  if MessageCount > 0 ->
    Messages = retrieve_messages(SecureSock, 1, MessageCount),
    true ->
      io:format("No new messages~n"),
      Messages = []
  end,
  ssl:close(SecureSock),
  Messages.

```

The fetch function should return the list of messages at the end; I showed it at the very beginning. So the last thing we need to do is to write something that will make a tuple of the format we want out of a plain text message. When we have this function, we will map it over the Messages list and our POP3 mission will finally be finished. Parsing the message has not much to do with pop3, so we will create another module:

```

-module(message).
-compile(export_all).

get_header(Headers, HeaderName) ->
  FindHeader = fun(Item) -> lists:prefix(HeaderName, Item) end,
  Header = lists:nth(1, lists:filter(FindHeader, Headers)),
  element(2, regexp:sub(Header, HeaderName, "")).

parse(Message) ->
  {_, Boundary, _} = regexp:first_match(Message, "\r\n\r\n"),
  Header = string:substr(Message, 1, Boundary),
  Body = string:substr(Message, Boundary + 4, length(Message) - Boundary),
  {ok, Headers} = regexp:split(Header, "\r\n"),
  {message, get_header(Headers, "Date: "), get_header(Headers, "From: "),
  get_header(Headers, "Subject: "), Body}.

```

Lets start the analysis with the function parse. First we need to split the header part of the message from the body - they are separated by an empty line, or two CR/LFs if you wish. The function regexp:first_match return a tuple of the format "{match, Start, Length}", but we only care about that the middle field - thats why we use "_". Its simply a way to say that you know about the presence of other fields, but you don't want them to be bound to anything. We could also say:

```
Boundary = element(2, regexp:first_match(Message, "\r\n\r\n"))
```

because the element function simply extracts the nth element of the tuple. Than we actually split the string using the found Boundary. The extracted header gets splitted one more time, so that each line (and each header field at the same time) is an element in a list. The final expression builds the tuple representing the message using the function get_header.

In get_header we use another powerful function operating on lists - filter. It takes a fun returning a boolean value as an argument and returns only those elements of the list, which when supplied to the fun give "true" as a result.

In the definition of the fun we once again take advantage of the misery of Erlang string as a list approach - thats why we can use lists:prefix to check if the string starts with the header name we supply to get_header as an argument.

We actually only care about the first occurrence of the header, so we make it a variable instead of a list using lists:nth. We don't want to have the header name in our final tuple, so in the last step we remove it and return only the "body" of the header. Now we only have to modify the fetch function from the pop3 module to apply message:parse to each message:

```
fetch() ->
    application:start(ssl),
    {ok, SecureSock} = ssl:connect("pop.gmail.com", 995,
                                  [{mode, list}, {packet, 0}, {active,
false}]),
    response_line(SecureSock),
    authenticate(SecureSock, "foobar", "barfoo"),
    {MessageCount, _} = do_stat(SecureSock),
    if MessageCount > 0 ->
        Messages = retrieve_messages(SecureSock, 1, MessageCount),
        MsgProc = fun(Item) -> message:parse(Item) end,
        ProcessedMessages = lists:map(MsgProc, Messages);
    true ->
        ProcessedMessages = []
    end,
    ssl:close(SecureSock),
    ProcessedMessages.
```

Voila - after all the struggles we have our messages ready for processing. We can call something like rss_wrap:rss_wrap(pop3:fetch()) to get a nice feed representing our mailbox content.

4: Serving the thing through HTTP

Now we need a simple web server to make our feed available in the browser. We could use Erlangs built-in inets module for doing HTTP, but since we don't need anything fancy here we can as well implement it on our own, as it will be an excellent opportunity to introduce more language features - what we did above was all sequential stuff, now we will get to the part that makes people curious about Erlang - concurrency.

The basic skeleton of a server operating through TCP looks like this:

```
-module(http).
-compile(export_all).

start() ->
    spawn(http, server, []).

server() ->
    {ok, ListenSocket} = gen_tcp:listen(80, [list, {packet, 0}, {active,
false}]),
    listen(ListenSocket).

listen(ListenSocket) ->
    case gen_tcp:accept(ListenSocket) of
        {ok, Socket} -> ->
            spawn(http, retrieve, [Socket, []]),
            listen(ListenSocket);
        _ ->
            listen(ListenSocket)
    end.

retrieve(Socket, Data) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Packet} ->
            %% Do something with the retrieved data
            retrieve(Socket, Data ++ Packet);
        {error, closed} ->
            gen_tcp:close(Socket)
```

end.

Several new constructions appear in this example. The most important thing here is the spawn function - it runs a function as a separate process. This means that you don't receive any return value immediately like when normally calling function - instead computing the function starts, but at the same time the control flow is passed back to the caller function. Spawn in the most commonly used form takes three parameters - the module name, the function name, and a list of arguments to be supplied to the function. It returns a pid - a unique identifier of the process in the Erlang system, which can be used later mainly for passing messages. Knowing that, understanding start() shouldn't be hard - it simply starts the http server (represented by the server() function) without blocking any other activities going on in the system and returning the control flow to the caller function as soon as the server process starts.

The spawned process sets up a socket for accepting incoming connections. The socket is passed to the listen function. Listen() makes use of the case instruction, used very often in Erlang. It basically does pattern matching on the result of the expression you supply it - here it checks for an incoming connection, if there is one pending it spawns a separate process to handle the request and goes back to listening for new connections. If there are no connection attempts, it just waits. Notice another use of recursive function definitions - here they are used to make a continuously running process.

Finally, the retrieve function pulls data out of the supplied socket and responds somehow to the recognized request. It uses its second argument to accumulate the received data over time.

Now we need to extend the general scheme to use HTTP. We will simplify it as much as possible - we will only respond to one kind of request, to a "GET /". For now, we will also respond with some hardcoded text, not with content that can be dynamically changed:

```
-module(http).  
  
-compile(export_all).  
  
-define(ok_200, "HTTP 1.1 200 OK\r\n").  
-define(content_type,  
    "Content-Type: text/plain\r\n").  
-define(content_length(Length),  
    "Content-Length: " ++ integer_to_list(Length) ++ "\r\n").  
  
start() ->  
    spawn(http, server, []).  
  
server() ->  
    {ok, ListenSocket} = gen_tcp:listen(80, [list, {packet, 0}, {active,  
false}]),  
    listen(ListenSocket).  
  
listen(ListenSocket) ->  
    case gen_tcp:accept(ListenSocket) of  
        {ok, Socket} ->  
            spawn(http, retrieve, [Socket, []]),  
            listen(ListenSocket);  
        _ ->  
            listen(ListenSocket)  
    end.  
  
retrieve(Socket, Data) ->  
    case gen_tcp:recv(Socket, 0) of  
        {ok, Packet} ->  
            Request = Data ++ Packet,  
            process_request(Socket, Request),  
            retrieve(Socket, Request);  
        {error, closed} ->  
            gen_tcp:close(Socket)  
    end.  
  
process_request(Socket, "GET / " ++ _) ->  
    Text = "Hello, HTTP!",
```

```

Response = ?ok_200 ++
           ?content_type ++
           ?content_length(length(Text)) ++ "\r\n" ++
           Text,
gen_tcp:send(Socket, Response),
gen_tcp:close(Socket).

```

```

process_request(Socket, _) ->
0.

```

The function `process_request` does something only when the request begins with a "GET /" - the "++" in patterns is splitting the string, not concatenating it. In case of a GET, we compose a simple response consisting of the HTTP headers and some hardcoded text. Then we send the response through the socket and close it - remember that in HTTP each request is completely independent. If the request is not a GET, we just return 0 and do nothing else with it. We can start in the Erlang shell

```
c(http).
```

```
http:start().
```

Note that Erlang lets only privileged users grab a port, so you may have to run it with `sudo`.

5: Putting the pieces together

Until now all the modules we developed were rather separated. Now we need to glue the pieces together, so the processes we will be running can communicate.

Communication between processes in Erlang is done mainly by using two primitives - the "!" operator and the "receive" block. "!" sends a message to a pid. A message can be quite any valid Erlang data type. We use it like this:

```

SomePid ! do_something
SomePid ! {this, is, a, tuple}
SomePid ! [1, 2, 3, 4, 5]

```

The only requirement is that the process represented by `SomePid` has the matching receive that will do something with the message. As many other constructions in Erlang, it uses pattern matching to recognize various kinds of messages, ie.:

```

receive
  {something, Data} ->
    io:format("~s~n", [Data]);
  {anotherthing, AnotherData} ->
    process(AnotherData)
end.

```

Before we use it in practice, lets write a separate simple example to illustrate message passing:

```

-module(test).
-compile(export_all).

server() ->
  receive
    Data ->
      io:format("~w~n", Data),
      server()
  end.

```

This tiny function listens for any kind of message. When it receives one, it simply outputs it using standard Erlang syntax. You can now send something to it from the shell:

```
SomePid = spawn(test, server, []).
```

```

SomePid ! {something, foobar}.
SomePid ! [1, 2, 3, 4, 5].
SomePid ! "whatever".

```

Going back to our main program, we need messages to pass the feed to the HTTP server. What we will try do is making the main function server accept two kinds of messages - one for setting the server response, and another for returning it. Actually "returning" is a tricky word here - a process wanting to get the response will have to supply his Pid and we sent the result back using messages. Look at the final version of the module:

```

-module(http).
-compile(export_all).

-define(ok_200, "HTTP 1.1 200 OK\r\n").
-define(content_type, "Content-Type: text/plain\r\n").
-define(content_length(Length),
    "Content-Length: " ++ integer_to_list(Length) ++ "\r\n").

control_server(Response) ->
    receive
        {set, response, NewResponse} ->
            control_server(NewResponse);
        {get, response, Pid} ->
            Pid ! {response, Response};
        _ ->
            control_server(Response)
    end.

server() ->
    ControlServerPid = spawn(http, control_server, [""]),
    register(control_server, ControlServerPid),
    spawn(http, initialize, []),
    ControlServerPid.

initialize() ->
    case gen_tcp:listen(80, [list, {packet, 0}, {active, false}]) of
        {ok, ListenSocket} ->
            listen(ListenSocket);
        _ ->
            stop
    end.

listen(ListenSocket) ->
    case gen_tcp:accept(ListenSocket) of
        {ok, Socket} ->
            spawn(http, retrieve, [Socket, []]),
            listen(ListenSocket);
        _ ->
            listen(ListenSocket)
    end.

retrieve(Socket, Request) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Packet} ->
            process_request(Socket, Request ++ Packet),
            retrieve(Socket, Request ++ Packet);
        {error, closed} ->
            gen_tcp:close(Socket)
    end.

process_request(Socket, "GET / " ++ _) ->
    control_server ! {get, response, self()},
    receive
        {response, Text} ->
            Response = ?ok_200 ++
                ?content_type ++
                ?content_length(length(Text)) ++ "\r\n" ++
                Text,

```

```

        gen_tcp:send(Socket, Response),
        gen_tcp:close(Socket)
    end;
process_request(Socket, Request) ->
    0.

```

First we introduce a “control server”. This process will serve as a buffer between the outside world and the HTTP server. The function starting the server (`http:server()`) returns the Pid of the control server to the caller process, so the user of the http module can send messages to the control server, altering the behaviour of the http server. But since we don’t have explicit state in Erlang, all the “variables” are stored as function parameters. We have only one dynamic thing here, but if there was more of them, we would probably want to use a property list for storing various settings of the server. `server()` does also one more interesting thing - it registers the control server process, so you can refer to it by a symbolic name instead of the pid and you don’t have to pass its pid to every function.

The reason we want the control server to be available everywhere is that the only way to check a setting (or in our case the desired server response) is to send a message to it. This is exactly what we do in the `process_request` function, body of which also explains why we called the process a “control_server”. We send a message to it that we want to get the response and we also supply the pid the control server should respond to, by using the function `self()` returning the pid of the current process. Then we wait for the response from the control server and when it arrives we’re ready to actually send the response through the socket.

Now we can finally put the pieces together using this little beauty:

```

-module(gmailrss).
-compile(export_all).

main() ->
    ServerPid = http:server(),
    spawn(gmailrss, loop, [ServerPid]).

loop(ServerPid) ->
    Messages = pop3:fetch(),
    Rss = rss_wrap:rss_wrap(Messages),
    ServerPid ! {set, response, Rss},
    timer:sleep(300000),
    loop(ServerPid).

```

Just run the Erlang shell and call:

```
gmailrss:main().
```

Then you can fire up your RSS reader, add “`http://localhost/`” as the feeds address and enjoy your email in it! The time value for the sleep function is measured in milliseconds, so it will get checked every 5 minutes.

6: Conclusions

Wow, that turned out really long and comprehensive, considering it was first meant to be a short blog post. You can grab the source code of the finished code [here](#). We still skipped a lot of subjects ie. we totally ignored any errors that might occur, to make explaining things easier. We didn’t care too much about message encoding and stuff like that either. There is also obviously much more cool stuff in Erlang, but that’s what books and docs are for. I wonder if someone made it this far at all, but nevertheless...

Happy hacking!