MODERN ONLINE RADIO WITH LIQUIDSOAP

TONY MILLER

Modern Online Radio with Liquidsoap

Tony Miller

This book is for sale at http://leanpub.com/modernonlineradiowithliquidsoap

This version was published on 2015-04-21

Contents

3.0 Digging into the basic example **1**
Three sources to gradually fallback 1
Static playlists 2
Live dj input 2
fallibility, preparing for failure 2
Cascading fallbacks 2
outputs 3
Saving recordings to files 3

4.0 The liquidsoap programming language **4**
types 4
about mutability, no changing variables 4
no unused variables 5
static typing 5
function notation in the API docs 5
defining functions 6
refs 7
converting types 7

6.0 dj authentication **8**
source client 8
Using input.harbor's auth parameter 8
other methods 10

3.0 Digging into the basic example

In this chapter we're going to dig a bit deeper into the basic example from the previous chapter and explain some things.

```
#!/usr/local/bin/liquidsoap

set("log.file",true)
set("log.file.path","./log/liquidsoap.log")
set("log.stdout",true)
set("log.level",3)

set("harbor.bind_addr","0.0.0.0")
```

This is mostly just some basic boilerplate. We output logging to stderr and also to a file. The log level is set to 3, the highest logging level. We also set the harbor.bind_addr to localhost.

Three sources to gradually fallback

What you are usually going to do in liquidsoap is set up one or more sources. These can be playlists, live inputs, or single files. Here we are going to set up 3 sources, then tell liquidsoap which one to play based on priority with the fallback function.

```
backup_playlist = playlist("./playlist.txt",conservative=true,mode="normal",relo\
ad_mode="watch")
output.dummy(fallible=true,backup_playlist)

live_dj = input.harbor("live",port=9000)

on_fail = single("./technical_difficulties.wav")

source = fallback(track_sensitive=false,
                  [live_dj,backup_playlist,on_fail])
```

Here we have a playlist, live dj input, and a single track. The second argument to fallback is an array of sources in descending priority. In this example, live_dj will be played if available, if not, the playlist is played. If the playlist becomes unavailable, the on_fail source will be played, which is just a single file.

Static playlists

We can use the playlist function to create a playlist. Playlists can be a text file listing of local or remote files, or you can simply pass a directory path and files from that directory will be played.

Here we create one from a static txt file. The playlist file simply looks like this:

```
mp3s/I_Cactus_-_01_-_yellow_cactus.mp3
mp3s/I_Cactus_-_02_-_chartreuse_cactus.mp3
mp3s/I_Cactus_-_03_-_green_cactus.mp3
mp3s/I_Cactus_-_04_-_bamboo_cactus.mp3
```

I'm using some mp3s that are released under a Creative Commons(CC) license in this tutorial. You can change this playlist to whatever you want, using relative or absolute paths to your music files in the playlist file.

Live dj input

The second source is a live input from the harbor. It is given a name "live" and it will listen on port 9000. You can connect to this just like you connect to icecast/shoutcast from a source client normally.

fallibility, preparing for failure

Sources in liquidsoap have a concept of 'fallibility'. Liquidsoap wants you to have a robust system and therefore won't allow you to use a fallible source. Try changing the filename passed to `single` to one that doesn't exist and run the script again. You'll get an error message:

```
that source is fallible
```

Likewise, if you try to use just a playlist or just a harbor source, you'll get the same message. You need to provide some sort of fallback if you want an infallible source.

There is a way around this by using the `fallible=true` option.

You can also force sources to be safe by using the `mksafe` function. This function will output silence when the source fails.

Cascading fallbacks

We have set up the fallback system in order of priority.

live dj -> playlist -> single emergency file

The live dj input always takes priority, but if its not available, we have the playlist of static files. If for some reason the playlist fails, we fall back to the single file.

outputs

Then we simply need to output our sources. Here we output an ogg and mp3 stream to a local icecast server.

```
output.icecast(%vorbis,id="icecast",
               mount="myradio.ogg",
               host="localhost", password="hackme",
               icy_metadata="true",description="cool radio",
               url="http://myradio.fm",
               source)
output.icecast(%mp3,id="icecast",
               mount="myradio.mp3",
               host="localhost", password="hackme",
               icy_metadata="true",description="cool radio",
               url="http://myradio.fm",
               source)
```

Saving recordings to files

```
# dump live_dj recordings to a file
time_stamp = '%m-%d-%Y, %H:%M:%S'
output.file(%mp3, "./live_dj_#{time_stamp}.mp3", live_dj,fallible=true)
```

The output function can also output to a file. It would be nice to save the live input recordings to disk. We can use time format modifiers to help us sort through the recordings later.

4.0 The liquidsoap programming language

In this chapter, I'll go into a bit more detail about how the liquidsoap language.

Liquidsoap is a *functional* language, which you may find confusing if you have not programmed much before. In functional languages you can't do things you might be used to doing in other languages like mutating values, changing state, etc.

types

The basic types in liquidsoap are integers, floats, strings and booleans.

- integers `42`
- floats `3.33`
- strings `"foo"`
- booleans `true` or `false`

You probably are quite familar with these if you have programmed before. There are other types specific to liquidosoap, these are

- source (produces a stream)
- request (something that a stream will play, like a file)
- encoding format (for different audio/video encodings)

and a few others.

about mutability, no changing variables

Since liquidsoap is a functional language, there are no variables, only definitions. So in this example, I'm not really changing the value of the `source` variable, I'm simply creating new ones and using the same name.

```
source = fallback(track_sensitive=false,
                  [live_dj,backup_playlist,on_fail])
source = on_metadata(pub_metadata, source)

source = server.rms(source)
```

no unused variables

You aren't allowed to declare a variable and then not use it, liquidsoap will complain:

```
At line 6, character 15: The variable some_variable defined here is not used
  anywhere in its scope. Use ignore(...) instead of some_variable = ... if
  you meant to not use it. Otherwise, this may be a typo or a sign that
  your script does not do what you intend.
```

static typing

Liquidsoap is statically typed, but the types are inferred. This means you won't have to write types like in C or Java.

function notation in the API docs

The function notation used in the liquidsoap api documentation[1] may be a bit daunting at first. For example here is the documentation for `input.harbor`:

```
(?id:string,?auth:((string,string)->bool),?buffer:float,
 ?debug:bool,?dumpfile:string,?icy:bool,
 ?icy_metadata_charset:string,?logfile:string,?max:float,
 ?metadata_charset:string,
 ?on_connect:(([(string*string)])->unit),
 ?on_disconnect:(()->unit),?password:string,?port:int,
 ?timeout:float,?user:string,string)->source('a)
```

All the question marks are simply optional named arguments and usually described in the function's documentation. Required arguments are marked with a ~ instead of a question mark.

`?id:string`

This is a optional parameter called `id` that is a type `string`.

[1] http://savonet.sourceforge.net/doc-svn/reference.html

`?on_disconnect:(()->unit)`

This is an optional parameter called on_disconnect that takes a function that is of type `()->unit`, which is a function that takes no arguments and returns `unit`. What's a `unit`, you ask? From the wikipedia page[2]

```
In the area of mathematical logic and computer science known as type theory, a
unit type is a type that allows only one value (and thus can hold no
information).
```

In simple terms, a unit is returned when you don't care about the return value. In other languages, you could simply not return anything. However, in functional languages, all functions *must* return a value, so we denote the return value as `unit` when we don't care about the return value.

`[t]` are lists and `(t*t)` are 'pairs' (or 'tuples' if you come from another language like Python).

`([(string*string)])->unit`

This is a function that takes a list of `(string*string)` (such as `("a","b")`) tuples and returns a unit.

Finally you can see what this function returns by looking at the arrow(`->`) at the end.

`->source('a)`

This means the function returns a source type.

defining functions

We can define a function with `def` and `end` to mark the end of a function.

```
  def function_name(arguments) =
    function_body
  end
```

like so:

[2] http://en.wikipedia.org/wiki/Unit_type

```
def get_user(user,password) =
  if user == "source" then
    x = string.split(separator=';',password)
    list.nth(x,0)
  else
    user
  end
end
```

The return type of the function is simply the last line of the body of the function. So in the case of the function above, it would be string.

refs

Liquidsoap is a functional language, however it is not a *pure* functional language. You can use mutatable variables if you want to. Liquidsoap borrows a concept from ocaml known as `ref`:

```
title = ref ""
current_dj_name = ref ""
```

converting types

You are able to convert between types, for example to convert a string to a boolean, you can use `bool_of_string`, for a integer from a string, you can use `int_of_string`. Search the documentation for the `*_of_*` function you are looking for. Here is what is available:

```
bool_of_float
bool_of_int
dB_of_lin
float_of_int
int_of_float
lin_of_dB
bool_of_string
float_of_string
int_of_string
string_of_metadata
```

Notice there are also functions for special liquidsoap types like dB, lin, and metadata that you might not see in other programming languages.

Check out the official liquidsoap language reference[3] for more.

[3] http://savonet.sourceforge.net/doc-svn/language.html

6.0 dj authentication

You will likely want to authenticate source clients for your stream, whether its a single username/-password or different username/password's per DJ.

source client

What's a source client? Simply whatever software your DJ uses to broadcast to icecast/shoutcast. `input.harbor` can take input from these clients. Here are some example source clients.

- Broadcast Using This Tool (win, mac)
- Ladiocast (mac)
- shoutvst (vst plugin)
- Traktor

Using input.harbor's auth parameter

You can supply a function as an argument to input.harbor's `auth` parameter.

```
1 live_dj = input.harbor("myradio",port=9000,auth=dj_auth,on_disconnect=on_disconn\
2 ect)
```

Its not mentioned in the API documentation, but this function is supposed to return true or false. The function will be passed the username and password as arguements.

On my station, I store my users in a database with passwords. They can sign up on a website. I call an external script to determine if they are authenticated or not.

Just as an example, here is what I use, its a ruby script that uses the json api on my rails app running on heroku to log in. It simply returns true or false based on the result of the login.

```ruby
#!/usr/bin/env ruby

Bundler.require

require 'httparty'

username = ARGV[0]
password = ARGV[1]

opts = { body: { :user => {"login" => username, "password" => password} } }

resp = HTTParty.post("http://www.datafruits.fm/login.json", opts)
if resp["success"] == true
  puts true
else
  puts false
end
```

Define a function in liquid soap that calls this external script, and you can pass that function as the auth parameter to input.harbor

```
def dj_auth(user,password) =
  u = get_user(user,password)
  p = get_password(user,password)
  #get the output of the script
  ret = get_process_lines("bundle exec ./dj_auth.rb #{u} #{p}")
  ret = list.hd(ret)
  #return true to let the client transmit data, or false to tell harbor to decli\
ne
  if ret == "true" then
    true
  else
    false
  end
end
```

Keep in mind, in some source clients there is stupidly no option to set the source password. If your djs are using said clients, you have a couple of options. You could use a hack where you simply enter the password in the same field separated by some character and parse that. I use this method, I instruct DJs to enter their username and password separated by a semicolon if their source client has no option to set the username.

```
def get_user(user,password) =
  if user == "source" then
    x = string.split(separator=';',password)
    list.nth(x,0)
  else
    user
  end
end

def get_password(user,password) =
  if user == "source" then
    x = string.split(separator=';',password)
    list.nth(x,1)
  else
    password
  end
end
```

This method is also backwards compatible with source clients that *do* support entering a password. You simply check if the username sent was ‘source’ and then split the password string at ‘;’.

other methods

You can supply any function at all to input.harbor’s auth parameter, so you can basically perform authentication anyway you choose.