MAD

Small and Fast Build Tool for Erlang Apps

Contents

1 MAD: Erlang Containers

1.1 Purpose

We were trying to make something minimalistic that fits out application stack[1]. The main idea of mad is to provide clean and simple rebar-like fast dependency manager that is able to build several types of packages and provides interface of containered deployments to virtualiezed environments.

1.2 Several Types of Packaging

The key feature of mad is ability to create single-file bundled web sites. This target escript is ready to run on Windows, Linux and Mac.

1.3 Deployment Options

As a deploy tool mad is also supposed to launch, start, stop and manage containers, locally or remote. You can make containers from different type of packages, like making runc container with beam release.

1.4 OTP Compliant

Mad supports ERTS boot files generation with systools and erlang application format used by OTP. This is the main format of application repository. Also boot files are suported on both LING and BEAM.

1.5 Tiny Size

And the good part:

	Sources	Binary
mad	**967 LOC**	**52 KB**
rebar	**7717 LOC**	**181 KB**

[1] https://github.com/synrc

1.6 History

We came to conclusion that no matter how perfect your libraries are, the comfort and ease come mostly from developing tools. Everything got started when Vladimir Kirillov[2] decided to replace Rusty's sync beam reloader. As you know sync uses filesystem polling which is neither energy-efficient nor elegant. Also sync is only able to recompile separate modules while common use-case in N2O is to recompile DTL templates and LESS/SCSS stylesheets. That is why we need to recompile the whole project. That's the story how active[3] emerged. Under the hood active is a client subscriber of fs[4] library, native filesystem listener for Linux, Windows and Mac.

De-facto standard in Erlang world is rebar. We love rebar interface despite its implementation. First we plugged rebar into active and then decided to drop its support, it was slow, especially in cold recompilation. It was designed to be a stand-alone tool, so it has some glitches while using as embedded library. Later we switched to Makefile-based build tool otp.mk[5].

The idea to build rebar replacement was up in the air for a long time. The best minimal approach was picked up by Sina Samavati[6], who implemented the first prototype called 'mad'. Initially mad was able to compile DTL templates, YECC files, escript (like bundled in gproc), also it had support for caching with side-effects. In a month I forked mad and took over the development under the same name.

Listing 1: Example of building N2O sample

```
                              Cold      Hot
rebar get-deps compile        53.156s   4.714s
mad deps compile              54.097s   0.899s
```

[2] https://github.com/proger
[3] https://github.com/synrc/active
[4] https://github.com/synrc/fs
[5] https://github.com/synrc/otp.mk
[6] https://github.com/s1n4

Listing 2: Example of building Cowboy

```
                                Hot
make (erlang.mk)                2.588s
mad compile                     2.521s
```

2 Setup

2.1 Installing Binary

Fresh version of mad included as a binary in its primary github repository:

```
# curl -fsSL https://raw.github.com/synrc/mad/master/mad > mad \
          && chmod +x mad \
          && sudo cp /usr/local/bin
```

Or you may want to add mad to your PATH.

2.2 Compiling Sources

If you want you can compile mad by yourself:

```
# git clone http://github.com/synrc/mad \
         && cd mad \
         && make
```

Note that mad uses mad to build mad. It's mad.

2.3 Creating a sample N2O project

mad also comes with N2O templates. So you can bootstrap a N2O-based site just having a single copy of mad binary.

```
# mad app sample
# cd sample
# mad deps compile release sample
```

After that you can just run escript web_app under Windows, Linux and Mac and open http://localhost:8000[7].

[7] http://localhost:8000

```
C:\> escript sample
Applications: [kernel,stdlib,crypto,cowlib,ranch,
               cowboy,compiler,syntax_tools,
               erlydtl,gproc,xmerl,n2o,sample,
               fs,active,mad,sh]
Configuration: [{n2o,[{port,8000},
                      {route,routes}]},
                 {kvs,[{dba,store_mnesia},
                      {schema,[kvs_user,
                               kvs_acl,
                               kvs_feed,
                               kvs_subscription]}]}]
Erlang/OTP 17 [erts-6.0] [64-bit] [smp:4:4]
              [async-threads:10] [kernel-poll:false]

Eshell V6.0  (abort with ^G)
1>
```

3 Configuration File

3.1 rebar.config

mad uses **rebar.config** filename to load mad configuration. Despite mad is no fully rebar compatible (e.g. it can't uses rebar plugins, ports compilation is rather different, etc), it uses its name to achive certail level of compatibility.

3.2 deps

deps is the core option of mad. It says which OTP applications shold be used and where they could be found. Yoy may also specify versions. Here is simpliest example:

Listing 3: deps Option

```
{deps, [
        {kvs,   ".*", {git,"git://github.com/synrc/kvs"}},
        {forms, ".*", {git,"git://github.com/spawnproc/forms"}}
]}.
```

3.3 deps_dir

To specify where deps should be stored after fetching inside your application you use deps_dir option:

Listing 4: deps_dir Option

```
{deps_dir, "deps"}.
```

3.4 sub_dirs

If your application consist of more than one src directory, you may specify all of the sub-applications. Each sub-application should be valid OTP application with its own rebar.config configuration file.

```
{sub_dirs,["apps"]}.
```

3.5 lib_dirs

To use include directive across your sub-applications you should specify the **lib_dirs** directories which will be settled as include directories during compilation.

```
{lib_dirs,["apps"]}.
```

E.g. you have my_app and my_server applications inside apps directory and you including HRL file from my_server application from ap_app application:

```
-module(my_app).
-include_lib("my_server/include/my_server.hrl").
```

4 Commands

Synrc mad has a simple interface as follows:

```
MAD Container Tool version b547fa

    invoke = mad params
    params = [] | command [ options  ] params
   command = app     | deps  | clean | compile | up
           | release [ beam  | ling  | script  | runc ]
           | deploy  | start | stop  | attach  | sh
```

It seems to us more natural, you can specify random commands set with different specifiers (options).

4.1 deps, dep

In rebar-like managers we are selecting deps from rebar.config:

```
{sub_dirs,["apps"]}.
{deps_dir,"deps"}.
{deps, [active,{nitro,"2.9"},{n2o,"2.9"}]}.
```

The search sequence for dependecies is follows. First mad will try to reach global package repository at http://synrc.com/apps/index.txt[8], this address is configurable. No application server is required for mad package management, only static files with OTP application format.

```
{application,bpe,
    [{description,"BPE SRC Business Process Engine"},
     {vsn,"1.9"},
     {registered,[]},
     {applications,[kernel,stdlib,kvs,n2o]},
     {dependencies,[kernel,stdlib,fs,ranch,crypto,mnesia,
                    gproc,cowlib,kvs,cowboy,n2o,active,
                    jsone,mad,nitro,sh,bpe]},
     {mod,{bpe_app,[]}},
```

[8] http://synrc.com/apps/index.txt

```
{env,[]},
{modules,[bpe,bpe_app,bpe_date,bpe_event,bpe_metainfo,bpe_proc,
          bpe_sup,bpe_task,default_railing,log_allow,routes,
          sampleproc,sampleproc_process]}]}.
```

If no file is found or server is unavailable then application registry will be taken from mad built-in index.txt. If no luck then the name of application, e.g. "spawnproc/rete" will be interpreted as github repository address.

```
$ mad dep active n2o kvs ling "spawnproc/rete"
```

4.2 compile, com

Performs compilation of all known compilations backends in complilation profile of mad:

```
app     app.src erlang templating
dtl     DTL compiler
erl     BEAM compiler
c/c++     for gcc cland and other native compilation
script     .script file used in projects like gproc
yrl/xrl     DSL language parser compilers
upl     UPL compiler
```

4.3 release, rel, bundle, bun

Taking all dependecies and resolve boot sequence according to dependecy order. Storing this value in .applist. If release type is not defined (**beam** in following example), then **script** release will be taken as a default.

```
$ mad release beam sample
Ordered: [kernel,stdlib,fs,ranch,crypto,compiler,syntax_tools,
          gproc,cowlib,cowboy,n2o,sample,active,erlydtl,jsone,
```

```
          mad,nitro,sh]
*WARNING* : Missing application sasl. Can not upgrade with this release
sample.boot: ok
OK:  "sample"

$ mad rel mad
Ordered: [kernel,stdlib,inets,sh,mad]
OK:  "mad"
```

MAD supports several releasing backends:

```
script     script bundles, like mad itself
beam     ERTS releases with systools
ling     LING portable unikernels
runc     Docker-compatible containers
```

4.4 sh, repl, rep

Start REPL shell session.

5 Dependencies

5.1 OTP Compliant

mad supports app files inside ebin, priv static folder and c$_{s}rcportsdirectoriespoir$

Listing 5: Solution

```
+-- apps
+-- deps
+-- rebar.config
+-- sys.config
```

Listing 6: OTP Application

```
+-- deps
+-- ebin
+-- include
+-- priv
+-- src
+-- rebar.config
```

5.2 Application Depot

As you may know you can create OTP releases systools from sasl application. mad currently creates releases with systools, but manually manages binary access: from local erlang or public applcation depot.

To bundle binary BEAM or LING along with synrc spplications MAD can use global repository statically avaliable at Github Pages:

```
$ curl -X GET http://synrc.com/apps/index.txt
[ {bin,[beam],     ["7.0.3"]},
  {lib,[active],   ["0.9"]},
  {lib,[compiler], ["6.0"]},
  {lib,[cowboy],   ["1.0.1"]},
  {lib,[cowlib],   ["1.0.0"]},
```

```
{lib,[crypto],   ["3.6"]},
{lib,[erlydtl],  ["0.8.0"]},
{lib,[fs],       ["1.9"]},
{lib,[gproc],    ["0.3"]},
{lib,[jsone],    ["v0.3.3"]},
{lib,[kernel],   ["4.0"]},
{lib,[stdlib],   ["2.5"]},
{lib,[kvs],      ["2.9"]},
{lib,[mad],      ["2.9"]},
{lib,[mnesia],   ["4.13"]},
{lib,[n2o],      ["2.10"]},
{lib,[nitro],    ["0.9"]},
{lib,[ranch],    ["1.0.0"]},
{lib,[rest],     ["2.9"]},
{lib,[review],   ["2.9"]},
{lib,[sh],       ["1.9"]},
{lib,[syntax_tools], ["1.7"]} ].
```

6 Container Bundles

6.1 ESCRIPT Bundles

The key feature of mad is ability to create single-file bundled web sites. Thus making dream to boot simpler than node.js come true. This bundle target is ready to run on Windows, Linux and Mac.

To make this possible we implemented a zip filesytem inside escript. mad packages priv directories along with ebin and configs. You can redefine each file in zip fs inside target escript by creation the copy with same path locally near escript. After launch all files are copied to ETS. N2O also comes with custom cowboy static handler that is able to read static files from this cached ETS filesystem. Also bundle are compatible with active online realoading and recompilation.

E.g. you main create a single file site with:

```
# mad bundle app_name
```

app_name shoul be the same as a valid Erlang module, with app_-module:main/1 function defined, which will boot up the bundle. This function could be like that:

```
-module(app_name).
main(Params) -> mad_repl:sh(Params).
```

6.2 BEAM ERTS Releases

As you may know you can create OTP releases with reltool (rebar generate) or systools (relx). mad creates releases boot script with systools and pack tra by itself.

```
# mad release beam sample
```

6.3 LING Unikernels

Sample rebar.config for your application you want to go unikernel:

```
{deps_dir,"deps"}.
{deps, [{ling,"master"},{sh,"1.9"}]}.
```

Now you should build LING/posix:

```
$ mad dep
$ cd deps/ling
$ ARCH=posix make
```

Now pack vmling.o, your OTP apps and rest static to single-file LING bundle with VM inside.

```
$ mad release ling mad
Ling Params: []
ARCH: posix_x86
Bundle Name: mad
System: [compiler,syntax_tools,sasl,tools,mnesia,reltool,xmerl,crypto,k
         stdlib,wx,webtool,ssl,runtime_tools,public_key,observer,inets,
         et,eunit,hipe,os_mon]
Apps: [kernel,stdlib,sh,mad]
Overlay: ["crypto.beam","9p.beam","9p_auth.beam","9p_info.beam",
          "9p_mounter.beam","9p_server.beam","9p_tcp.beam","9p_zero.bea
          "disk.beam","disk_server.beam","embedded_export.beam",
          "goo_export.beam","goofs.beam","hipe_unified_loader.beam",
          "inet_config.beam","kernel.beam","ling_bifs.beam","ling_code.
          "ling_disasm.beam","ling_iops.beam","ling_iopvars.beam",
          "ling_lib.beam","net_vif.beam","os.beam","prim_file.beam",
          "user_drv.beam","os_mon.beam","dets.beam","filename.beam",
          "maps.beam","unicode.beam","zlib.beam"]
Bucks: [{boot,"/boot",2},
        {os_mon,"/erlang/lib/os_mon/ebin",1},
        {crypto,"/erlang/lib/crypto/ebin",1},
        {kernel,"/erlang/lib/kernel/ebin",90},
        {stdlib,"/erlang/lib/stdlib/ebin",85},
        {sh,"/erlang/lib/sh/ebin",6},
        {mad,"/erlang/lib/mad/ebin",43}]
Initializing EMBED.FS:
Mount View:
```

```
 /boot /boot
/erlang/lib/os_mon/ebin /os_mon
/erlang/lib/crypto/ebin /crypto
/erlang/lib/kernel/ebin /kernel
/erlang/lib/stdlib/ebin /stdlib
/erlang/lib/sh/ebin /sh
/erlang/lib/mad/ebin /mad
Creating EMBED.FS C file: ...ok
Compilation of Filesystem object: ...ok
Linking Image: ok
```

Run it:

```
$ rlwrap ./image.img
Erlang [ling-0.5]

Eshell V6.3  (abort with ^G)
1> application:which_applications().
[{mad,"MAD VXZ Build Tool","2.2"},
 {sh,"VXZ SH Executor","0.9"},
 {stdlib,"ERTS  CXC 138 10","2.2"},
 {kernel,"ERTS  CXC 138 10","3.0.3"}]
```

6.4 Docker-compatible RUNC Containers

Creating runc-complatible container is simple:

```
# mad release runc sample
```

6.5 Makefiles with OTP.MK

OTP.MK is a tiny 50 lines Makefile that allows to start your set of application using run_erl and to_erl tools from OTP distribution. We use that way in poduction. This is the best option also in development mode because all directory structure is open and mutable, so you can reload modified files and perform recompilation on the fly.

It uses the original code to fast resolve dependencies into the right boot sequence to start. If you want more powerful Makefile-based erlang package management you may take a look onto ERLANG.MK by Nine Nines.

```
# make console
```