Let’s make release upgrades great again!

Who am i?

- Luis Rascão
- Work @ Miniclip
- Erlang’Ing since 2014

Miniclip (paid this trip for me)

- Mobile games company
  - Started out with mostly Flash games
  - Now focused on mobile
- Has some big hits in the stores
  - 8 Ball Pool
  - Soccer Stars
  - Agar.io
- I’m the tech lead of the 8 Ball Pool server

8 Ball Pool

- Miniclip's biggest hit
- 18 million daily active users
- 750K peak concurrent users
- 26 machine cluster just for the game servers
- Mostly Erlang

Bugs amirite?

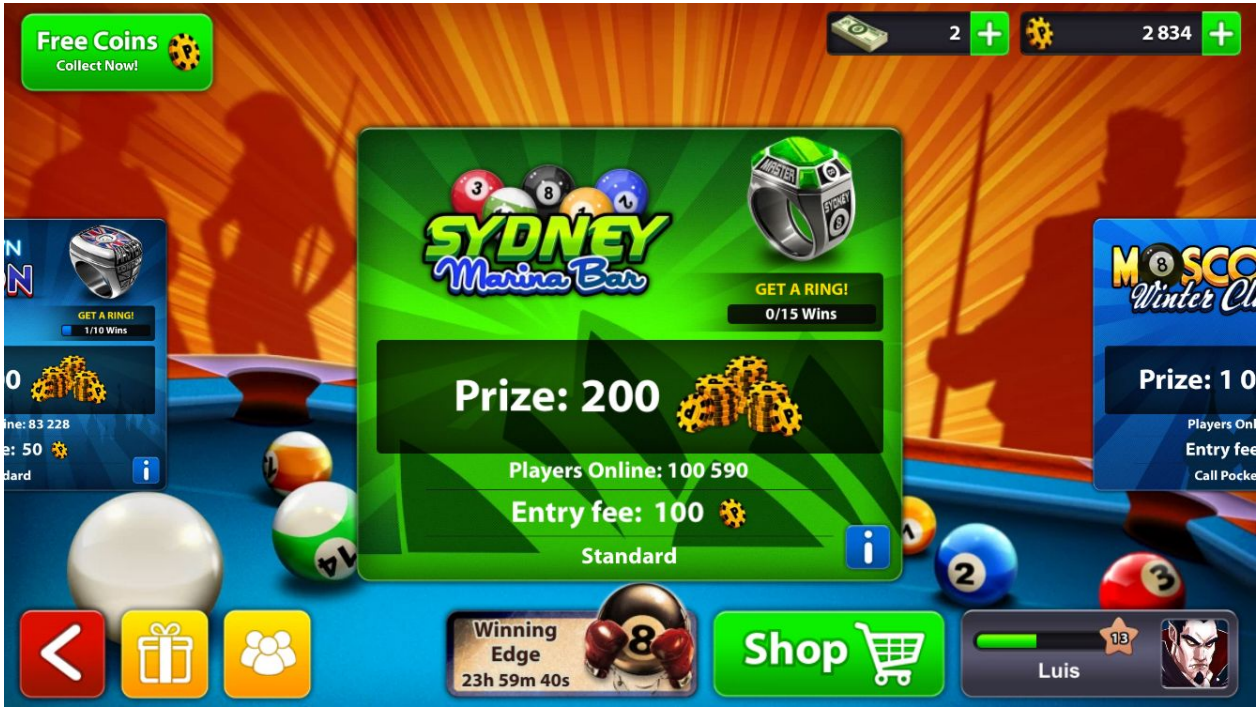
“Debugging is like being the detective in a crime movie where you are also the murderer.”

Filipe Fortes @fortes

There is no escaping them

- If you write software you'll also write bugs
- Corollary: if you don't have bugs it's because you're not writing software

8 Ball Pool


Free Coins
Collect Now!
2
2 834
SYDNEY
Marina Bar
GET A RING!
0/15 Wins
Prize: 200
Players Online: 100 590
Entry fee: 100
Standard
GET A RING!
1/10 Wins
Winning Edge
23h 59m 40s
Shop
Luis
Prize: 1 0
Players On
Entry fee
Call Pocke

What do you do?

- Stop the servers, fix the bug, start them again
  - Involves downtime
  - Money is lost

What do you do?

- Blue/Green deployment
  - Assuming a load balancer in front of your servers
    - Blue and Green are identical
  - Direct traffic to the canary node running the fix
  - Ensure the canary node is running as expected
  - Works fine
    - If you have persistent connections you'll have to wait for clients to disconnect from the Blue stack
    - You'll probably want to automate this (or have someone else do it)

Hot code loading

- Erlang/OTP offers us a way of updating running code without any downtime
- A lot simpler than Blue/Green deployments (of course it depends on the fix)

How? (the simple approach)

- Build the .beam file containing your fix
- Overwrite the buggy one on disk
- Attach to the running node
  - `l(Module)`
- Done

How? (the simple approach)

- Messy business
  - If there's more than one module you better be damn sure of the order in which you load them
  - Your app says it's running version x.y.z but that's not true anymore
  - If you changed something in your sys.config you should also set it at the console and update the file on disk

How? (the simple approach)

It’s not all bad

- You get to apply your changes incrementally and check for errors on each step

Always remember though

- Bug fixes can have bugs themselves, most of the times they’re nastier than the original ones

Hot code loading

- Two code pointers are kept per module
  - Current - points to the currently running code
  - Old - starts out as nil
- When you load a new version of the module
  - Current - points to the new code
  - Old - points to the old code
- All fully qualified function calls go to the current version

Hot code loading

- Take a peek at `lib/stdlib/src/c.erl`

```
l(Mod) ->
    code:purge(Mod),
    code:load_file(Mod).
```

- `code:purge(Mod)` is a brutal purge
- There's also `code:soft_purge(Mod)`

Purging

- Soft
  - Checks all processes, if any are running old code (with `erlang:check_process_code(Pid, Mod)`) fails the call
- Brutal
  - Checks all processes, kills any that are running old code

Stickiness

- You can declare a directory in the search path as “sticky”
  - `code:stick_dir(Dir)`
  - `code:unstick_dir(Dir)`
- `code:load_file(Module)` will fail if Module is on a sticky dir

Release upgrades

- The structured way of changing code with no downtime
- Much more than just loading modules at the shell
  - `gen_*:code_change/3`
  - Starts new applications
  - Reloads sys.config and informs applications of changes made to it
  - Upgrades and downgrades applications

Release upgrades (downsides)

- Really complicated
  - LYSE describes release upgrades as the “9th circle of Erl”
  - Fred does a walkthrough on the relup chapter with all the manual steps
    - Most people probably skip it (I know I did)

Release upgrade workflow

- It begins by writing the application upgrade file (i.e. appup)
- The application upgrade is translated to a release upgrade file (i.e. relup)
- With the relup + new code you're ready to apply the release upgrade on a running instance

Application upgrade

- A file containing sets of instructions that define how an application is upgraded or downgraded
- It's a single term of the format

```
{Vsn, [{UpFromVsn, Instructions}, ...],
      [{DownToVsn, Instructions}, ...]}.
```

- Say you're upgrading from 1.9 to 2.0

```
{"2.0", [{"1.9", Instructions}, ...],
        [{"1.9", Instructions}, ...]}.
```

Application upgrade

- Several instructions available to you
  - load_module
    - {load_module, Mod, PrePurge, PostPurge, DepMods}
    - PrePurge, PostPurge can be either brutal or soft
    - DepMods is a list of modules that should be loaded before this one
  - add_module / delete_module
    - Both take DepMods
  - add_application / remove_application / restart_application

Application upgrade

- update
  - Synchronized update of processes running the module to upgrade
  - Obtains all supervised processes (ie. recursively searching down from the main supervisor)
  - Iterates through all of them asking the ones that use the module to suspend themselves
  - gen_* processes all have the code_change/3,4 callback
    - Module:code_change(OldVsn, State, Extra)
    - Allows you to migrate state structure. When this method is called you get the old state (while running new code) and return new state
    - The Extra argument is additional data fed directly from the appup for custom processing

Application upgrade

- {update, Mod, ModType, Timeout, Change, PrePurge, PostPurge, DepMods}
  - ModType either `static` or `dynamic`
  - Change is either `soft` or `{advanced, Extra}`
    - that's where the Extra argument in code_change comes from
  - Timeout is the time allowed to wait for the suspend request

Configuration changes

- After the upgrade, the application controller compares the old and new configuration parameters for all applications
- It then calls `config_change/3` for all applications specified in the `mod` key of the .app file
- `Module:config_change(Changed, New, Removed)`

Doing it by hand (short version)

- Write the .appup file
- Give it to systools and ask it to create a relup file
  - `systools:make_relup/3,4`
  - `systools:make_tar/1,2`
- Unpacking and upgrading the release
  - `release_handler:unpack_release/1`
  - `release_handler:which_releases/0,1`
  - `release_handler:install_release/1,2`
  - `release_handler:make_permanent/1`

Automation (using rebar3)

- There are some plugins already that take away some of this manual work
  - erlup (https://github.com/soranoba/erlup)
  - relflow (https://github.com/RJ/relflow)
  - rebar3_appup_plugin (https://github.com/lrascao/rebar3_appup_plugin)
- I'll be talking about rebar3_appup_plugin (which is the one I wrote)

Automation (using rebar3_appup_plugin)

- Appup generation
- Module dependencies
- Appup.src compilation
- Automatic state record migration

Appup generation

- Generate two release versions
  - The one you're in and the one you want to upgrade to
- `rebar3 appup generate`
  - Compares two releases to find out what was changed and generate appropriate instructions
  - add/delete/load/upgrade modules
  - Check for changes in supervisor spec, generate instructions to start/stop children

Module dependencies

- Use xref to determine dependencies of each new/changed module
  - Only static dependencies are caught (ie. `Module:Function`)
  - No support for dynamic calls (made through `erlang:apply` for example)

Appup.src compilation

- Most of the times you'll want to add your .appup file to source control
- Follow the same principle as .app.src
  - save your .appup as an .appup.src file alongside your other source files
- The plugin will pick it up, validate, template and evaluate it

Appup.src compilation

- Templating
  - Using mustache template variables
    - {{vsn}} - current version of the release
- Evaluation
  - The whole file is evaluated as if it were an escript file
  - `STATE` (rebar3's state) variable is available
  - Methods from both your release and rebar3 are available to you
- Validation
  - Enforces a valid .appup format at the end

Appup.src compilation

```
%% find our app info in rebar's STATE
AppInfo = rebar3_appup_utils:find_app_info(<<"relapp">>, STATE),
"{{vsn}}" = rebar_app_info:original_vsn(AppInfo),
{"{{vsn}}",
    [
     {<<".*">>, [{restart_application, relapp}]}
    ],
    [
     {<<".*">>, [{restart_application, relapp}]}
    ]
}.
```

State migration

- Processes keep state, usually through a record
  - If you need to change the record structure it's going to be tricky
- Ways to do it
  - Manually, with a bunch of `erlang:setelement/2` (it will hurt your eyes)
  - Ulf Wiger's `exprecs` parse transform in the `parse_trans` project
    - Generates a `'#convert'` method for every record
    - You declare both versions of the record in the module and convert them on `code_change`
  - Using the plugin's code injection facility

State migration (through code injection)

- Upon request, the plugin will inject code into the `code_change` method that takes care of the conversion between record versions
- You need to declare the name of the record that holds the state
  - `-state_record(some_record).`

State migration (through code injection)

- Upon request the plugin will
  - Get the abstract code of the new version beam
  - Get the record definitions of the current/previous versions
  - Inject the old record definition and code that ports it to the new one into the new version abstract code
  - Overwrite the beam file
- Needs `debug_info` to be on
- When the release upgrade happens and `code_change` gets called the `State` is already the new one
- The old state is still kept in the `Extra` argument as a tuple
  - `{old_state, OldState}`

Demo

Thanks!

luis.rascao@gmail.com
http://lrascao.github.io
https://github.com/lrascao