Building applications with

Talk summary:

- 1) where did it come from
- 2) what’s it good for
- 3) what’s it look like
- 4) OTP

Named after Agner Krarup Erlang, Danish queuing theory pioneer and telephony god whose work led to the creation of the Erlang unit of telephonic load
http://en.wikipedia.org/wiki/Agner_Krarup_Erlang
http://en.wikipedia.org/wiki/Erlang_(unit)

Invented at ericsson in 1986, when they were making phones like this.
Based and initially written in prolog
Designed to help manage telephone switching at exchanges, highly concurrent systems that had to be fault tolerant

Open sourced in 1998 when they were making phones like this
Now widely deployed at scale

- Facebook chat
- WhatsApp
- rabbitmq
- ejabberd
- couchdb
- Lots of messaging and voip platforms

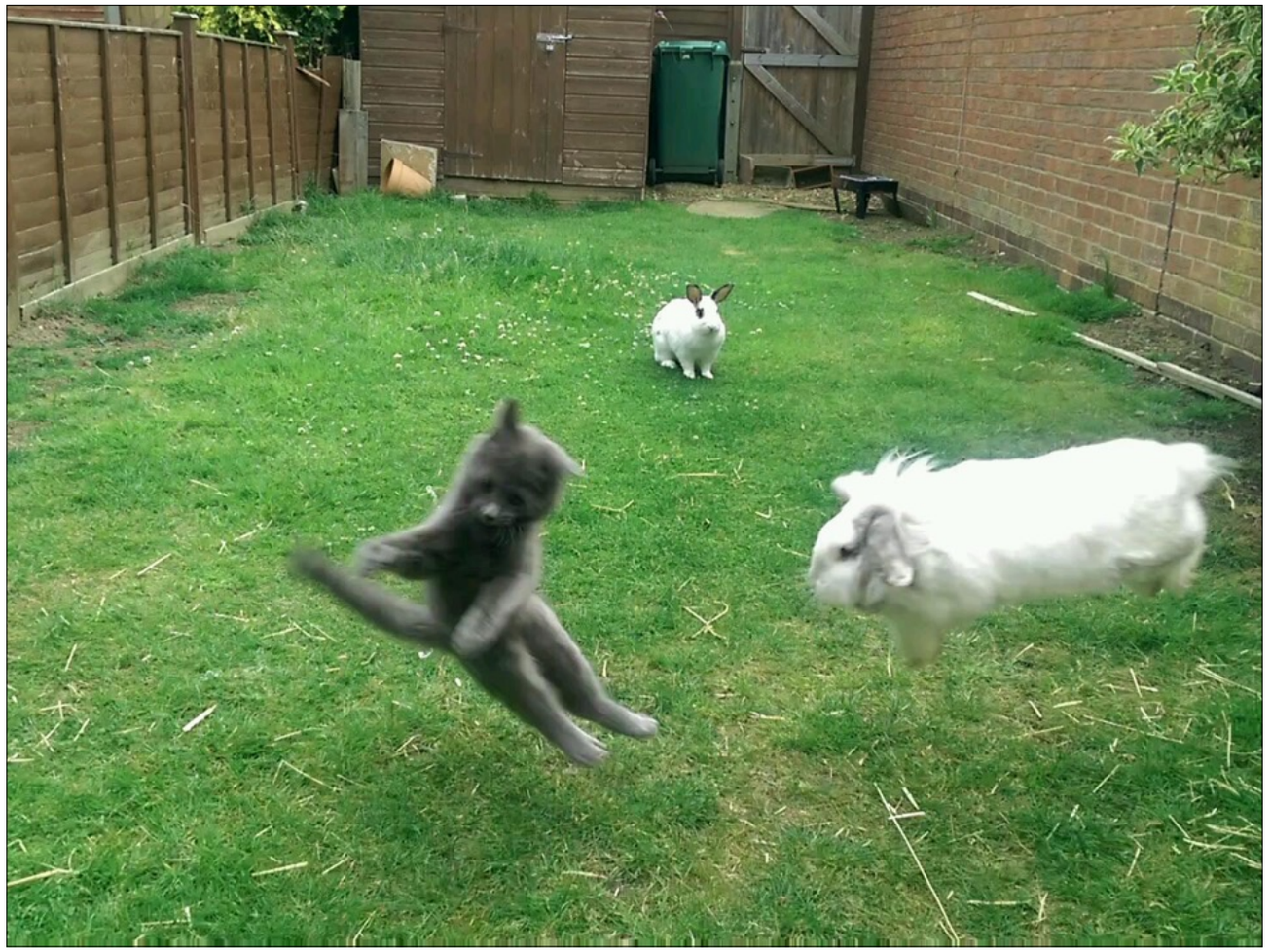
IRCCloud.com

An IRC client without the baggage

My company, irccloud.com is an integrated IRC bouncer and client
We use erlang's concurrency and hot code-reload features to keep long running IRC connections open for our users.

Erlang: The Movie
https://www.youtube.com/watch?v=xrljfIjssLE
an example of Erlang in action, made by the creators, for internal publicity at Ericsson
- Joe Armstrong, Robert Virding, Mike Williams

The short version: https://www.youtube.com/watch?v=Ue8qDY-4-ZA



- what problems does it solve?
http://imgur.com/n985PW7

- as mentioned, it's useful for building concurrent systems, such as telephony or messaging apps
- also designed for distribution: code is portable, thanks to the erlang virtual machine, built-in support for synchronising across multiple nodes
- often touted as good for "soft-realtime" problems. but that doesn't really mean anything that i can discern
- basically it's well suited for managing data flow in multi-actor systems
- NOT that well-suited for heavyweight blocking computation e.g. image processing

Erlang’s concurrency model is probably the most interesting aspect to it

- one of the key concepts in erlang is its light weight processes
- erlang code runs in a virtual machine called the BEAM (Björn's Erlang Abstract Machine)
- a single OS process
- provides an abstraction layer for passing messages to internal processes
- workload in the BEAM is managed with a run queue
- when a message is sent, the receiving process is put on the queue
- the VM manages context switching to get that work done
- each process has its own message queue called a mailbox
- language features built around this concept

https://www.flickr.com/photos/alpat/5007011905/

Basic primitives

```
97                  % integer
1.23                % float
hello, true, false  % atom
```

- very minimal basic primitive set
- integers, floats, atoms
- atoms are named constants
- no bool; true, false are just atoms

Funky primitives

```
% anonymous and named function
#Fun<mod.0.0>     fun(X) -> X+1 end
#Fun<foo.bar.1>   fun foo:bar/1

% process
<0.1.0>           spawn(Fun)  pid(0,1,0)

% port
#Port<0.0>        open_port()

% reference
#Ref<0.0.0.1>     make_ref()
```

- but that's not the whole story, erlang has a few extras up its sleeve
- first column is how they look pretty printed in an erlang shell
- second column is how you make them
- first class functions of course, named and anonymous
- first class processes, identified by a process id, or "pid"
- ports are how you communicate outside the erlang vm, so TCP sockets, shelling out to other programs, etc
- make_ref gives you a globally unique reference that can be used as identifiers
    - in practice we have integer ids from the database and pids so refs are rarely used

Compound data types

```
[97,98,99]       "abc"                  % list
<<97,98,99>>  <<"abc">>                 % binary
{1,2,3}         {"a", "b", 3}           % tuple
#rec{key=value, key2="value2"}          % record
#{key => value,"key2" => "value2"}      % map

% records are just compiler sugar on tuples
{rec, value, value2}
% key names are atoms, set in their definition
-record(rec, {key, key2}).
```

- some compound data types, some more familiar than others
- no strings. lists of unicode code point integers
- binaries are sequences of 8-bit bytes
- maps new in erlang 17, allow any term as key or value
    - erlang term is any of the primitive or compound types we've seen
    - still not that widespread because they're so new
    - a lot of erlang code still uses records
- records are a compiler hack for key value data, keys must be atoms

```
97
1.23
hello           true  false
[97,98,99]      "abc"
<<97,98,99>>    <<"abc">>
{1,2,3}         #rec{key=value}
#{key => value}
#Fun<mod.0.0>
<0.1.0>
#Port<0.0>
#Ref<0.0.0.1>
```

hello.erl

```erl
% usage: Pid = hello:start().

-module(hello).

-export([start/0]).

start() ->
  Pid = spawn(fun loop/0),
  % Use ! to send messages to a process
  Pid ! hello,
  Pid ! {hello, defshef16},
  Pid.
% ...
```

- first class functions
- first class processes
- pattern matching
- ; , . used as separators
- value of last expression is return value
- also: single assignment

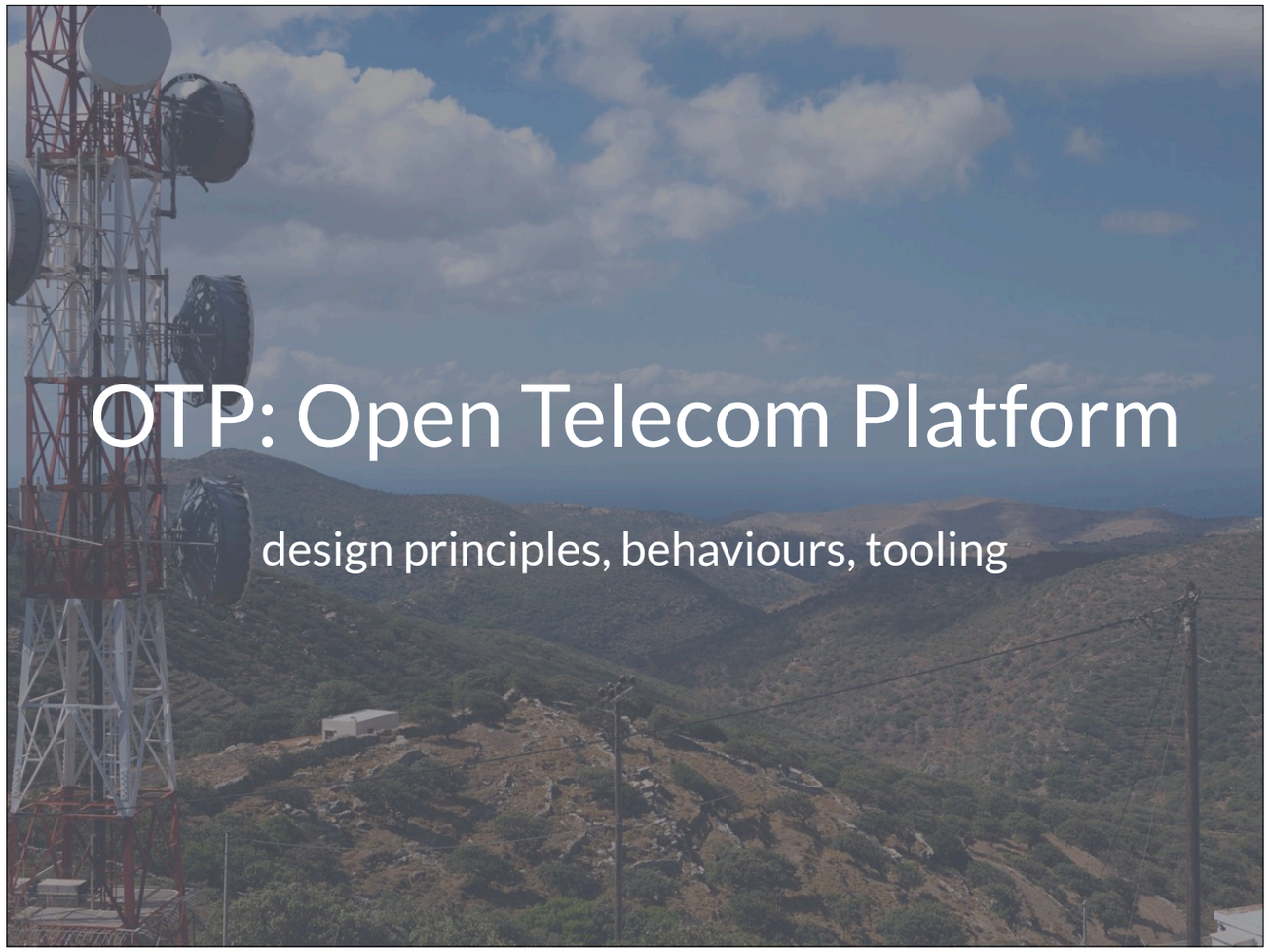
https://github.com/jwheare/defshef16/tree/master/talk/hello.erl

```erlang
% ...
loop() ->
  receive
    hello ->
      % say hi
      io:format("Hello world!~n"),
      loop();
    {hello, Name} ->
      % say hi to someone
      io:format("Hello ~s!~n", [Name]),
      loop();
    Unrecognised ->
      % dunno
      io:format("huh? what's ~s?~n", [Unrecognised]),
      loop()
  end.
```

- first class functions
- first class processes
- pattern matching
- ; , . used as separators
- value of last expression is return value
- also: single assignment

https://github.com/jwheare/defshef16/tree/master/talk/hello.erl

https://www.flickr.com/photos/jwheare/2050652341



http://www.erlang.org/doc/design_principles/des_princ.html
OTP is sort of like Erlang's standard library, helps you structure applications

https://www.flickr.com/photos/jwheare/15463744215

- behaviours are the building blocks of otp apps
- the most commonly used behaviour is the gen_server
- lets you store state in the process and pass in messages to act on that state
- most workers in an otp app will be `gen_servers`
- there are other behaviours such as `gen_event` for event handlers and `gen_fsm` for finite state machines
  - but they're far less useful and common
  - you can mostly replicate them with a `gen_server` anyway

https://www.flickr.com/photos/jwheare/85417794

- supervisors enable error recovery
- one of the main design principles of erlang is "let it crash" and that's mainly thanks to the supervisor.
- supervisors manage child processes, starting and restarting them when they exit unexpectedly
- child processes can be other supervisors, or workers
- this lets you built a supervision tree to keep individual parts of your app isolated or coupled as needed

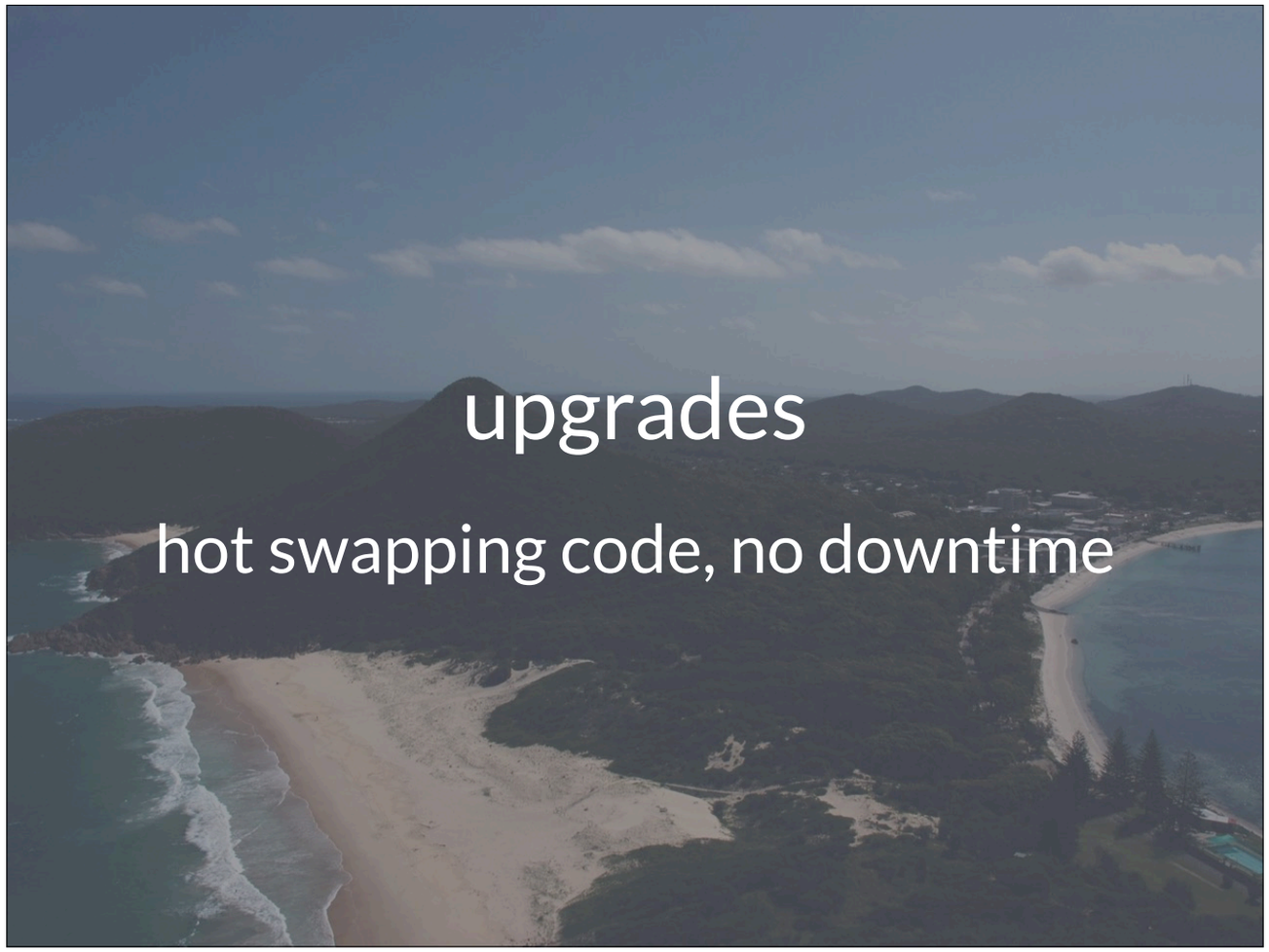
https://www.flickr.com/photos/jwheare/1082644614

- `application` is a behaviour defining the top level container for, er, an application. a logical grouping of functionality
- apps can include other apps or operate independently of each other.

https://www.flickr.com/photos/jwheare/70712274

- releases are a packaged collection of apps that get run when you start up your system
- releases define the environment and boot parameters on the node
- provide a packaged binary that can start and stop the node and let you attach to the running process from a shell
- everything you need to make releases is included with erlang, but we'll be using a 3rd party tool called rebar to make it easier
http://www.rebar3.org

https://www.flickr.com/photos/jwheare/12015403786



- finally, what good is a long running, fault tolerant system if we can't upgrade it.
- i'll be demonstrating how to upgrade a release without restarts

https://www.flickr.com/photos/jwheare/1082623754

demo

github.com/jwheare/defshef16

https://github.com/jwheare/defshef16

multiple function heads and guards

```
% functions with the same arity can be defined
% with multiple function heads

is_positive(X) when X > 0 ->
  true;
is_positive(X) ->
  false.
```

bonus slides

conditionals

`if` expressions exist, but they're weird and not very useful

```
if
  Condition ->
    do_something();
  true ->
    do_something_else()
end.

% conditions tend to be written with case expressions instead
case Condition of
  true ->
    do_something();
  false ->
    do_something_else()
end.
```

bonus slides

looping

loops are all about recursion, the `lists` module is our friend

```
lists:foreach(fun(X) ->
  io:format("Number: ~B!~n", [X])
end, [1,2,3]).

% sum elements in a list, [1,2,3] -> 6
Sum = lists:foldl(fun(X, Acc) ->
  Acc + X
end, 0, [1,2,3]).

% list comprehensions, increment each item
[X+1 || X <- [1,2,3]].
```

bonus slides

heads or tails

lists can be used as cons cells, with a head and a tail

```
List = [1,2,3]
Head = hd(List) % 1
Tail = tl(List) % [2,3]

% pipe: |, is the cons operator, used to construct lists
[Head | Tail]        % [1,2,3]
% like a russian doll
[1 | [2 | [3 | []]]] % [1,2,3]

% or for pattern matching
[First | Rest] = [1,2,3]
```

bonus slides