Erlang Gives You Superpowers

Jack Moffitt
jack@metajack.im
@metajack
http://metajack.im

The Erlang Powers

(no alien birth, spider bite, radiation, or family misfortune required)

Functional language

Closures

Pattern matching

Binary manipulation

Many processes

Message passing

Distributed

Crash early

Native compiling, ports, NIFs, etc

Documentation

The Erlang Ecosystem

(everyone needs superfriends and sidekicks)

Best in class applications:

ejabberd, Riak, CouchDB, RabbitMQ

Amazing libraries:

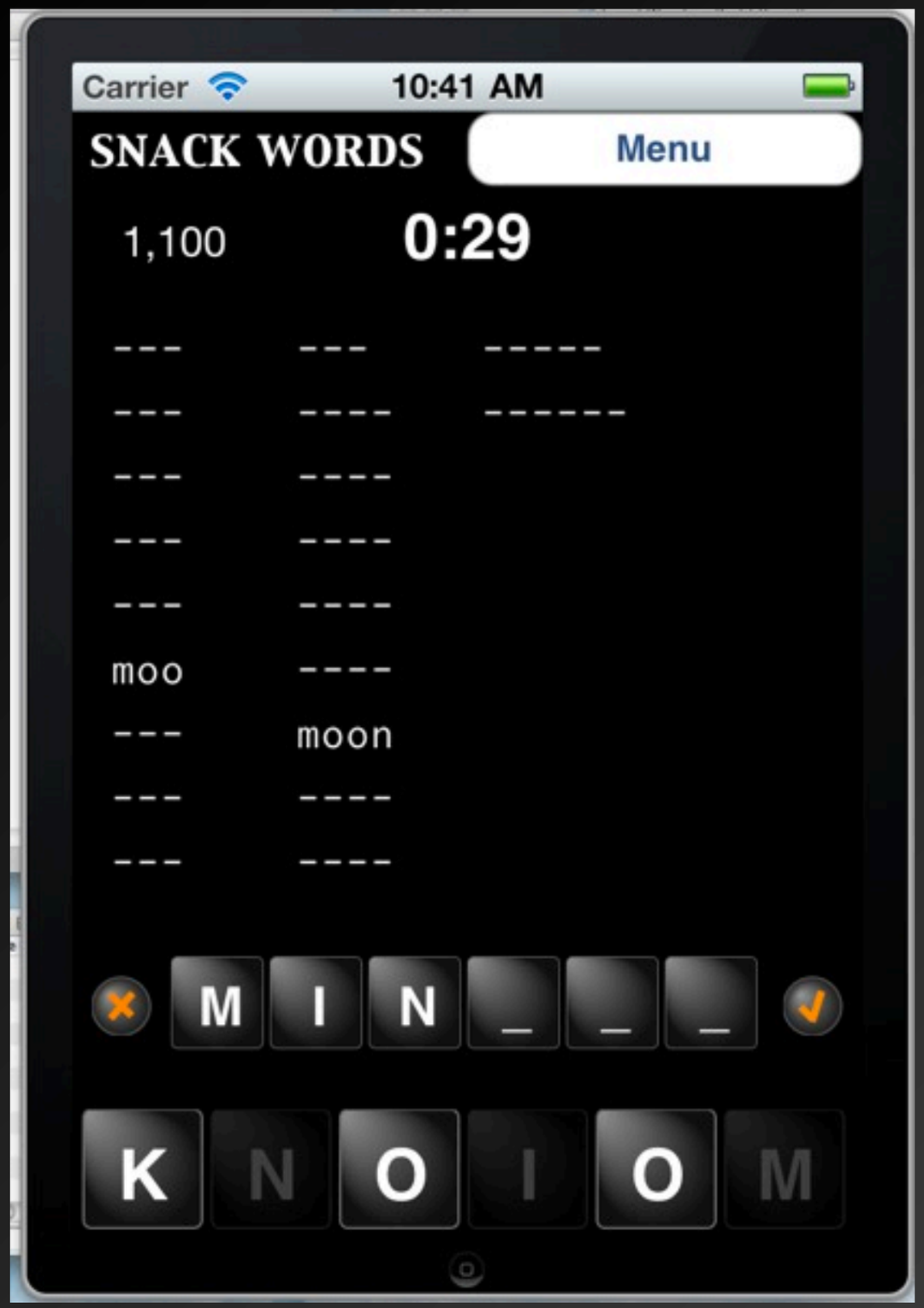
OTP, ibrowse, webmachine, mochiweb, gproc

Awesome tooling:

Dialyzer, rebar, QuickCheck, Distel

A Typical Superproject

(please use superpowers for good only)



Snack Words is an

iPhone game.

It needs a
real-time
game server.

It needs a **protocol**.

It should be

simple.

Hopefully, it will need to **scale.**

Protocol

Length prefixed JSON packets

gen_tcp makes these protocols trivial:

{packet, N} for N=0,1,2,4

gen_tcp:send and gen_tcp:recv take care of the framing

gen_tcp sends a message when packets arrive

jsonerl:encode()

jsonerl:decode()

```
{“cmd”: ”auth”,
 “device_id”: “XXXX”}
```

```
{“cmd”: “play”}
```

Game Logic

Clients are just a set of states and transitions.

This is what gen_fsm is made for.

States:

starting
authing
idle
waiting
playing

Games are code that
tracks time,
keeps score,
and communicates with players.

This sounds like a gen_server.

Receives

`{submit_word, Word}`

RPC calls from clients.

Handles tick messages from the timer.

Asynchronously notifies clients about events.

Events:

`{game_start, ...}`

`{clock, N}`

`{word_found, ...}`

`{game_end, ...}`

Server Design

The game server is an OTP application.

A listener process waits for connections.

A new client process is started for each connection.

The application supervisor monitors:

the listener,
the client supervisor,
and the game supervisor,
and the waiting pool.

Each client is a process
monitored by the client supervisor.

Each game is a process
monitored by the game supervisor.

1000 players playing 500 games means
1505 processes (1 listener and 4 supervisors).

This is not a big deal in Erlang.

Persistent state is minimal
and kept in Mnesia.

Data access is modularized
and easily replaceable.

Code

```
snackwords_listener.erl (gen_server)

handle_info({inet_async, ...}, State) ->
    ...
    {ok, Pid} =
        snackwords_client_sup:start_client(),
    ok = gen_tcp:controlling_process(Socket,
                                     Pid),
    snackwords_client_fsm:set_socket(Pid,
                                     Socket),
    ...
```

```
snackwords_client_fsm (gen_fsm)

starting({socket, Socket}, State) ->
    ok = inet:setopts(Socket,
                      [{active, once}]),
{next_state, authing,
 State#client_state{socket=Socket}}.
```

```
snackwords_client_fsm (gsm_fsm)

handle_info({tcp, Socket, Data}, StateName, State) ->
    %% set the socket so we can receive another data packet
    ok = inet:setopts(Socket, [{active, once}]),

    %% verify that Data is a JSON object
    case catch jsonerl:decode(Data) of
        {'EXIT', _} ->
            {stop, invalid_json, State};
        Json when is_tuple(Json) ->
            ?MODULE:StateName({data, tuple_to_list(Json)}, State);
        _ ->
            {stop, invalid_json, State}
    end;
handle_info({tcp_closed, _Socket}, _StateName, State) ->
    {stop, normal, State};
```

```
snackwords_client_fsm.erl (gen_fsm)

idle({data, Json}, State) ->
    ...

playing({data, Json}, State) ->
    ...
```

snackwords_game.erl (gen_server)

```
handle_call({submit_word, Word}, {Player, _},
            State) ->
    ...

handle_info(tick, State) ->
    ...
```

snackwords_db_mnesia.erl

```
-export([start/0,
         stop/1,
         get_player/3,
         create_player/2,
         create_player/4,
         save_player/2,
         authenticate/2,
         get_words/2]).
```

The code is tiny:

snackwords_listener.erl - 89 lines
snackwords_game.erl - 176 lines
snackwords_client_fsm.erl - 187 lines
snackwords_db_mnesia.erl - 211 lines
(db code contains all fixtures, too)

I expect final code to be under 2k lines.

Test code is ~400 lines currently.

Future Proofing

Distributed version of the server is pretty simple.

Each node runs the application under a load balancer.

The client and game subsystems need no changes!

Only need to worry about persistence and the waiting pool.

Database code is abstracted.

Mnesia could be easily be replaced with PostgreSQL or Riak.

The waiting pool needs to be globally known.

Or does it?

Conclusions:

It’s extremely simple and small.

The scaled up version is only a bit bigger.

I don’t have to work hard now;
I won’t have to work hard later.

jack@metajack.im
http://metajack.im
@metajack

http://snackwords.com