From Zero to Emonk

The Power of NIFs

Paul J. Davis
http://davispj.com

This Talk

- Emonk - What is it?
- NIFs - Touring the API by example.
- Putting it together
- http://github.com/davisp/zero-to-emonk
- Slides are in the download section

Emonk - What is it?

- JavaScript (via SpiderMonkey) bindings for Erlang
- Originally based on erlang_js by Basho
- Re-written using NIFs
- Provides JavaScript for iOS port of Apache CouchDB

Emonk - Erlang API

```
Eshell V5.8.2  (abort with ^G)
1> {ok, Ctx} = emonk:create_ctx().
{ok,<<>>}
2> emonk:eval(Ctx, <<"var f = 2; f *= 4.5; f;">>).
{ok,9}
3> emonk:eval(Ctx, <<"f;">>).
{ok,9}
4> emonk:eval(Ctx, <<"f = {foo: 2.5};">>).
{ok,{[{<<"foo">>,2.5}]}}
5> emonk:eval(Ctx, <<"f = function(x) {return x;};">>).
{ok,undefined}
6> emonk:call(Ctx, <<"f">>, [[2, 4.3, null, false, []]]).
{ok,[2,4.3,null,false,[]]}
```

Your First NIF

```c
#include "erl_nif.h"

static ERL_NIF_TERM
say_hi(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    const char* mesg = "Hello, World!";
    return enif_make_string(env, mesg, ERL_NIF_LATIN1);
}

static ErlNifFunc funcs[] =
{
    {"say_hi", 0, say_hi}
};

ERL_NIF_INIT(first_nif, funcs, NULL, NULL, NULL, NULL);
```

```erlang
-module(first_nif).
-export([say_hi/0]).

-on_load(init/0).

say_hi() ->
    not_loaded(?LINE).

init() ->
    PrivDir = code:priv_dir(?MODULE),
    SoName = filename:join(PrivDir, ?MODULE),
    erlang:load_nif(SoName, 0).

not_loaded(Line) ->
    exit({not_loaded, [{module, ?MODULE}, {line, Line}]}).
```

```
Eshell V5.8.2  (abort with ^G)
1> first_nif:say_hi().
"Hello, World!"
```

Library Lifetime

We are beholden to dlopen(3) and friends

```erlang
-module(modname).
-export([myfun/0]).
-on_load(init/0).

init() ->
    erlang:load_nif("path/to/modname.so", any_term).

% When the NIF loads, it replaces this
% definition.
myfun() ->
    throw({error, not_loaded}).
```

```c
// Called in response to erlang:load_nif/2 when loading
// this NIF.
//
// Generally used as a place to create some sort of
// global scope that is then stored in priv.
int load(ErlNifEnv* env, void** priv, ERL_NIF_TERM info);

// When someone calls erlang:load_nif/2 and the NIF
// has already been loaded. Can be used to change
// state in priv.
int reload(ErlNifEnv* env, void** priv, ERL_NIF_TERM info);

// This has to do with code reloading somehow. I have
// no idea how to trigger it.
int upgrade(ErlNifEnv* env, void** priv,
        void** oldpriv, ERL_NIF_TERM info);

// Called when this NIF is being unloaded from the
// Erlang VM.
void unload(ErlNifEnv* env, void* priv);
```

Marshaling

```
// Unmarshaling
int
enif_get_TYPE(ElrNifEnv* env, ERL_NIF_TERM term, CTYPE* val);

// Marshaling
ERL_NIF_TERM
enif_make_TYPE(ErlNifEnv* env, CTYPE value);
```

```
const ERL_NIF_TERM
double(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    int val;
    if(argc != 1)
        return enif_make_badarg(env);
    if(!enif_get_int(env, argv[0], &val))
        return enif_make_badarg(env);
    return enif_make_int(env, val * 2);
}
```

Testing Complex Types

```
int enif_is_atom(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_binary(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_fun(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_pid(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_port(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_ref(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_tuple(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_list(ErlNifEnv* env, ERL_NIF_TERM term)
int enif_is_empty_list(ErlNifEnv* env, ERL_NIF_TERM term)

int enif_compare(ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)
int enif_is_identical(ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)
```

Maintaining State

- NIF's can store a private pointer that will be available to each NIF function
- When writing to shared state (static vars, private data) you MUST use locks.
- Yes, as in mutexes.

```c
#include <assert.h>
#include "erl_nif.h"

typedef struct {
    ErlNifMutex*    lock;
    int             global;
} state;

static int
load(ErlNifEnv* env, void** priv, ERL_NIF_TERM info)
{
    state* st = enif_alloc(sizeof(state));
    if(st == NULL) return 1;
    st->lock = enif_mutex_create("lock");
    if(st->lock == NULL) {
        enif_free(st);
        return 1;
    }
    if(!enif_get_int(env, info, &(st->global)))
        st->global = 0;
    *priv = (void*) st;
    return 0;
}
```

```c
static int
reload(ErlNifEnv* env, void** priv, ERL_NIF_TERM info)
{
    state* st = (state*) (*priv);

    if(!enif_get_int(env, info, &(st->global)))
        st->global = 0;

    return 0;
}

static void
unload(ErlNifEnv* env, void* priv)
{
    if(priv != NULL) enif_free(priv);
}
```

```c
static ERL_NIF_TERM
curr(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    state* st = (state*) enif_priv_data(env);
    int ret;

    enif_mutex_lock(st->lock);
    ret = st->global;
    enif_mutex_unlock(st->lock);

    return enif_make_int(env, ret);
}
```

```
static ERL_NIF_TERM
incr(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    state* st = (state*) enif_priv_data(env);
    int ret;

    enif_mutex_lock(st->lock);
    ret = ++st->global;
    enif_mutex_unlock(st->lock);

    return enif_make_int(env, ret);
}
```

```
static ERL_NIF_TERM
decr(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    state* st = (state*) enif_priv_data(env);
    int ret;

    enif_mutex_lock(st->lock);
    ret = --st->global;
    enif_mutex_unlock(st->lock);

    return enif_make_int(env, ret);
}
```

```c
static ERL_NIF_TERM
swap(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    state* st = (state*) enif_priv_data(env);
    int val;
    int ret;

    if(argc != 1)
        return enif_make_badarg(env);
    if(!enif_get_int(env, argv[0], &val))
        return enif_make_badarg(env);

    enif_mutex_lock(st->lock);
    ret = st->global;
    st->global = val;
    enif_mutex_unlock(st->lock);

    return enif_make_int(env, ret);
}
```

```
static ErlNifFunc funcs[] =
{
    {"curr", 0, curr},
    {"incr", 0, incr},
    {"decr", 0, decr},
    {"swap", 1, swap}
};

ERL_NIF_INIT(nifstate, funcs, &load, &reload, NULL, &unload);
```

```
Eshell V5.8.2  (abort with ^G)
1> 0 = nifstate:curr().
0
2> 1 = nifstate:incr().
1
3> 2 = nifstate:incr().
2
4> 1 = nifstate:decr().
1
5> 1 = nifstate:swap(12345).
1
6> 12345 = nifstate:curr().
12345
7> nifstate:reload(10).
ok
8> 11 = nifstate:incr().
11
```

Resource Types

- Used to safely pass C pointers through Erlang on the same node.
- Ref-counted by Erlang similar to binaries
- Each type has a custom destructor called when ref-count hits zero.
- Used to managed complex C types

MD5 API

```
Ctx = resources:init().

ok = resources:update(Ctx, <<“foobar”>>)

Md5 = resources:hex(Ctx)
```

```c
#include <assert.h>
#include <openssl/md5.h>
#include "erl_nif.h"

typedef struct {
    MD5_CTX      md5;
    int          finalized;
} md5ctx;

void
md5_dtor(ErlNifEnv* env, void* obj)
{
    md5ctx* ctx = (md5ctx*) obj;
    unsigned char hash[16];
    if(!ctx->finalized)
        MD5_Final(hash, &(ctx->md5));
}
```

```c
ErlNifResourceType* md5_type;

static int
load(ErlNifEnv* env, void** priv, ERL_NIF_TERM info)
{
    int flags = ERL_NIF_RT_CREATE | ERL_NIF_RT_TAKEOVER;
    md5_type = enif_open_resource_type(
            env, NULL, "md5", md5_dtor, flags, NULL
        );
    if(md5_type == NULL) return 1;
    return 0;
}
```

```
static ERL_NIF_TERM
init(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ERL_NIF_TERM ret;

    md5ctx* ctx = enif_alloc_resource(md5_type, sizeof(md5ctx));

    if(!MD5_Init(&(ctx->md5))) {
        ctx->finalized = 1;
        enif_release_resource(ctx);
        return make_atom(env, "init_error");
    }

    ctx->finalized = 0;

    ret = enif_make_resource(env, ctx);

    // Release our reference, ret now has ownership.
    enif_release_resource(ctx);

    return ret;
}
```

```c
static ERL_NIF_TERM
update(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    md5ctx* ctx;
    ErlNifBinary bin;
    if(argc != 2)
        return enif_make_badarg(env);

    if(!enif_get_resource(env, argv[0], md5_type, (void**) &ctx))
        return enif_make_badarg(env);

    if(!enif_inspect_binary(env, argv[1], &bin))
        return enif_make_badarg(env);
    if(ctx->finalized)
        return make_atom(env, "already_finalized");
    if(!MD5_Update(&(ctx->md5), bin.data, bin.size))
        return make_atom(env, "update_error");
    return make_atom(env, "ok");
}
```

```c
static ERL_NIF_TERM
hex(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    md5ctx* ctx;
    ERL_NIF_TERM ret;
    unsigned char* hash = enif_make_new_binary(env, 16, &ret);

    if(argc != 1)
        return enif_make_badarg(env);
    if(!enif_get_resource(env, argv[0], md5_type, (void**) &ctx))
        return enif_make_badarg(env);

    if(!MD5_Final(hash, &(ctx->md5)))
        return make_atom(env, "finalization_error");
    ctx->finalized = 1;

    return ret;
}
```

```c
static ErlNifFunc funcs[] =
{
    {"init", 0, init},
    {"update", 2, update},
    {"hex", 1, hex}
};

ERL_NIF_INIT(resources, funcs, &load, NULL, NULL, NULL);
```

```
Eshell V5.8.2  (abort with ^G)
1> Ctx1 = resources:init().
<<>>
2> resources:update(Ctx1, <<"foobar">>).
ok
3> resources:hex(Ctx1).
<<56,88,246,34,48,172,60,145,95,48,12,102,67,18,198,63>>
4> Ctx2 = resources:init().
<<>>
5> resources:update(Ctx2, <<"foo">>).
ok
6> resources:update(Ctx2, <<"bar">>).
ok
7> resources:hex(Ctx2).
<<56,88,246,34,48,172,60,145,95,48,12,102,67,18,198,63>>
8> crypto:start().
ok
9> crypto:md5(<<"foobar">>).
<<56,88,246,34,48,172,60,145,95,48,12,102,67,18,198,63>>
```

Process Independent Environments

- Used to persist terms beyond the scope of a NIF function.
  - Sending terms from threads
  - Storing terms for later use
- Need to be tracked just as any other C pointer

Term Pointers (Sorta)

```
Ptr = termptr:wrap(Term).
```

```
Term = termptr:unwrap(Ptr).
```

```c
#include <assert.h>
#include <stdio.h>

#include "erl_nif.h"

typedef struct {
    ErlNifEnv*      env;
    ERL_NIF_TERM    data;
} termptr;

void
termptr_dtor(ErlNifEnv* env, void* obj)
{
    termptr* ptr = (termptr*) obj;
    assert(ptr->env != NULL && "Invalid termptr.");
    enif_free_env(ptr->env);
}
```

```c
ErlNifResourceType* ptr_type;

static int
load(ErlNifEnv* env, void** priv, ERL_NIF_TERM info)
{
    int flags = ERL_NIF_RT_CREATE | ERL_NIF_RT_TAKEOVER;
    ptr_type = enif_open_resource_type(
            env, NULL, "termptr", termptr_dtor, flags, NULL
        );
    if(ptr_type == NULL) return 1;
    return 0;
}
```

```c
static ERL_NIF_TERM
wrap(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    termptr* ptr = (termptr*) enif_alloc_resource(
            ptr_type, sizeof(termptr)
        );
    ERL_NIF_TERM ret;

    ptr->env = enif_alloc_env();

    if(argc != 1) {
        enif_release_resource(ptr);
        return enif_make_badarg(env);
    }

    ptr->data = enif_make_copy(ptr->env, argv[0]);

    ret = enif_make_resource(env, ptr);
    enif_release_resource(ptr);
    return ret;
}
```

```c
static ERL_NIF_TERM
unwrap(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    termptr* ptr;

    if(argc != 1)
        return enif_make_badarg(env);
    if(!enif_get_resource(env, argv[0], ptr_type, (void**) &ptr))
        return enif_make_badarg(env);

    return enif_make_copy(env, ptr->data);
}
```

```c
static ErlNifFunc funcs[] =
{
    {"wrap", 1, wrap},
    {"unwrap", 1, unwrap}
};

ERL_NIF_INIT(termptr, funcs, &load, NULL, NULL, NULL);
```

```
Eshell V5.8.2  (abort with ^G)
1> Ptr = termptr:wrap([{foo, bar}, 1.34, self(), make_ref()]).
<<>>
2> termptr:unwrap(Ptr).
[{foo,bar},1.34,<0.31.0>,#Ref<0.0.0.29>]
3> OtherFun = fun() ->
3>     receive
3>         Ptr1 ->
3>             io:format("~p~n", [termptr:unwrap(Ptr1)])
3>     end
3> end.
#Fun<erl_eval.20.67289768>
4> Pid = spawn(OtherFun).
<0.37.0>
5> Pid ! Ptr.
[{foo,bar},1.34,<0.31.0>,#Ref<0.0.0.29>]
<<>>
```

Sending Terms

- Limited to Pid's on same node
- Basic steps

1. Create empty environment
2. Copy terms to it
3. Send message
4. Destroy or clear environment

```c
#include "erl_nif.h"

static ERL_NIF_TERM
send(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifEnv* cp_env = enif_alloc_env();
    ErlNifPid dst;
    ERL_NIF_TERM msg, ret;

    if(argc == 1) {
        if(!enif_self(env, &dst)) {
            ret = enif_make_badarg(env);
            goto done;
        }
        msg = argv[0];
    } else if(argc == 2) {
        if(!enif_get_local_pid(env, argv[0], &dst)) {
            ret = enif_make_badarg(env);
            goto done;
        }
        msg = argv[1];
    } else {
        ret = enif_make_badarg(env);
        goto done;
    }
```

```
    msg = enif_make_copy(cp_env, msg);

    if(!enif_send(env, &dst, cp_env, msg)) {
        ret = enif_make_badarg(env);
        goto done;
    }

    ret = enif_make_atom(env, "ok");

done:
    enif_free_env(cp_env);
    return ret;
}

static ErlNifFunc funcs[] =
{
    {"send", 1, send},
    {"send", 2, send}
};

ERL_NIF_INIT(termsend, funcs, NULL, NULL, NULL, NULL);
```

Threads!

- Yes threads.
- Fairly standard threading API, will be familiar if you know pthreads
- But first, why threads?

Lessons in Sharing

- NIF functions are called from the Erlang scheduler thread
- The scheduler (written in C) calls the NIF C function directly
- You can't pause a function call in C (without voodoo).
- Thus, NIFs need to return quickly

```c
#include <unistd.h>
#include "erl_nif.h"

static ERL_NIF_TERM
nif_sleep(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    unsigned int num;
    if(argc != 1)
        return enif_make_badarg(env);
    if(!enif_get_uint(env, argv[0], &num))
        return enif_make_badarg(env);

    sleep(num);

    return enif_make_int(env, 0);
}

static ErlNifFunc funcs[] =
{
    {"sleep", 1, nif_sleep}
};

ERL_NIF_INIT(badnif, funcs, NULL, NULL, NULL, NULL);
```

```
#!/usr/bin/env escript

tick() ->
    {_, Secs, _} = now(),
    tick(Secs).

tick(Secs) ->
    {_, S, _} = now(),
    io:format("~~tick~~  ~p~n", [S-Secs]),
    timer:sleep(1000),
    tick(Secs).

bad_sleep(0) ->
    ok;
bad_sleep(N) when N > 0 ->
    spawn(fun() -> badnif:sleep(5) end),
    bad_sleep(N-1).

main([]) ->
    spawn(fun() -> tick() end),
    bad_sleep(N),
    timer:sleep(20000).
```

N=1

```
$ ./run badnif
~tick~  0
~tick~  1
~tick~  2
~tick~  3
~tick~  4
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
~tick~  10
~tick~  11
~tick~  12
~tick~  13
~tick~  14
~tick~  15
~tick~  16
~tick~  17
~tick~  18
~tick~  19
```

N=2

```
$ ./run badnif
~tick~  0
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
~tick~  10
~tick~  11
~tick~  12
~tick~  13
~tick~  14
~tick~  15
~tick~  16
~tick~  17
~tick~  18
~tick~  19
```

N=4

```
$ ./run badnif
~tick~  0
~tick~  10
~tick~  11
~tick~  12
~tick~  13
~tick~  14
~tick~  15
~tick~  16
~tick~  17
~tick~  18
~tick~  19
```

N=8

```
$ ./run badnif
~tick~  0
~tick~  20
```

Schedulers: 2 (all tests)

Threads!

- Long blocking calls need to be dispatched to threads
- My rule of thumb: If execution time is not a pure function of the NIF arguments, it needs threads.
- Its easiest to keep things simple

```c
#include <unistd.h>
#include "erl_nif.h"

typedef struct {
    ErlNifTid           tid;
    ErlNifThreadOpts*   opts;
    ErlNifPid           dst;
    ErlNifEnv*          env;
    ERL_NIF_TERM        msg;
    unsigned int        delay;
} state;

void
state_dtor(ErlNifEnv* env, void* obj)
{
    state* st = (state*) obj;
    void* ret;
    enif_thread_join(st->tid, &ret);
    enif_thread_opts_destroy(st->opts);
    enif_free_env(st->env);
}
```

```c
ErlNifResourceType* st_type;

static int
load(ErlNifEnv* env, void** priv, ERL_NIF_TERM info)
{
    int flags = ERL_NIF_RT_CREATE | ERL_NIF_RT_TAKEOVER;
    st_type = enif_open_resource_type(
            env, NULL, "state", state_dtor, flags, NULL
        );
    if(st_type == NULL) return 1;
    return 0;
}
```

```c
void*
state_run(void* obj)
{
    state* st = (state*) obj;
    enif_keep_resource(st);

    sleep(st->delay);

    enif_send(NULL, &(st->dst), st->env, st->msg);

    enif_release_resource(st);

    return NULL;
}
```

```
static ERL_NIF_TERM
delayed_send(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    state* st;
    int status;
    ERL_NIF_TERM ret;

    st = enif_alloc_resource(st_type, sizeof(state));
    st->opts = enif_thread_opts_create("state");
    st->env = enif_alloc_env();

    if(argc != 3) {
        ret = enif_make_badarg(env);
        goto done;
    } else if(!enif_get_local_pid(env, argv[0], &(st->dst))) {
        ret = enif_make_badarg(env);
        goto done;
    } else if(!enif_get_uint(env, argv[2], &(st->delay))) {
        ret = enif_make_badarg(env);
        goto done;
    }

    st->msg = enif_make_copy(st->env, argv[1]);
```

```
    status = enif_thread_create(
            "state", &(st->tid), state_run, st, st->opts
        );
    if(status != 0) {
        ret = enif_make_badarg(env);
        goto done;
    }

    ret = enif_make_resource(env, st);

done:
    enif_release_resource(st);
    return ret;
}

static ErlNifFunc funcs[] =
{
    {"delayed_send", 3, delayed_send}
};

ERL_NIF_INIT(goodnif, funcs, &load, NULL, NULL, NULL);
```

```erlang
tick() ->
    {_, Secs, _} = now(),
    tick(Secs).

tick(Secs) ->
    {_, S, _} = now(),
    io:format("~~tick~~  ~p~n", [S-Secs]),
    timer:sleep(1000),
    tick(Secs).

sink() ->
    receive Term -> io:format("Received: ~p~n", [Term]) end,
    sink().

good_sleep(_Dst, 0) ->
    ok;
good_sleep(Dst, N) when N > 0 ->
    spawn(fun() -> goodnif:delayed_send(Dst, N, 4) end),
    good_sleep(Dst, N-1).

main(_) ->
    spawn(fun() -> tick() end),
    good_sleep(spawn(fun() -> sink() end), N),
    timer:sleep(10000).
```

N=1

```
$ ./run goodnif
~tick~  0
~tick~  1
~tick~  2
~tick~  3
Received: 1
~tick~  4
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
```

N=2

```
$ ./run goodnif
~tick~  0
~tick~  1
~tick~  2
~tick~  3
Received: 1
Received: 2
~tick~  4
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
```

N=4

```
$ ./run goodnif
~tick~  0
~tick~  1
~tick~  2
~tick~  3
Received: 4
Received: 1
Received: 3
Received: 2
~tick~  4
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
```

N=8

```
$ ./run goodnif
~tick~  0
~tick~  1
~tick~  2
~tick~  3
Received: 1
Received: 2
Received: 3
Received: 6
Received: 4
Received: 7
Received: 8
Received: 5
~tick~  4
~tick~  5
~tick~  6
~tick~  7
~tick~  8
~tick~  9
```

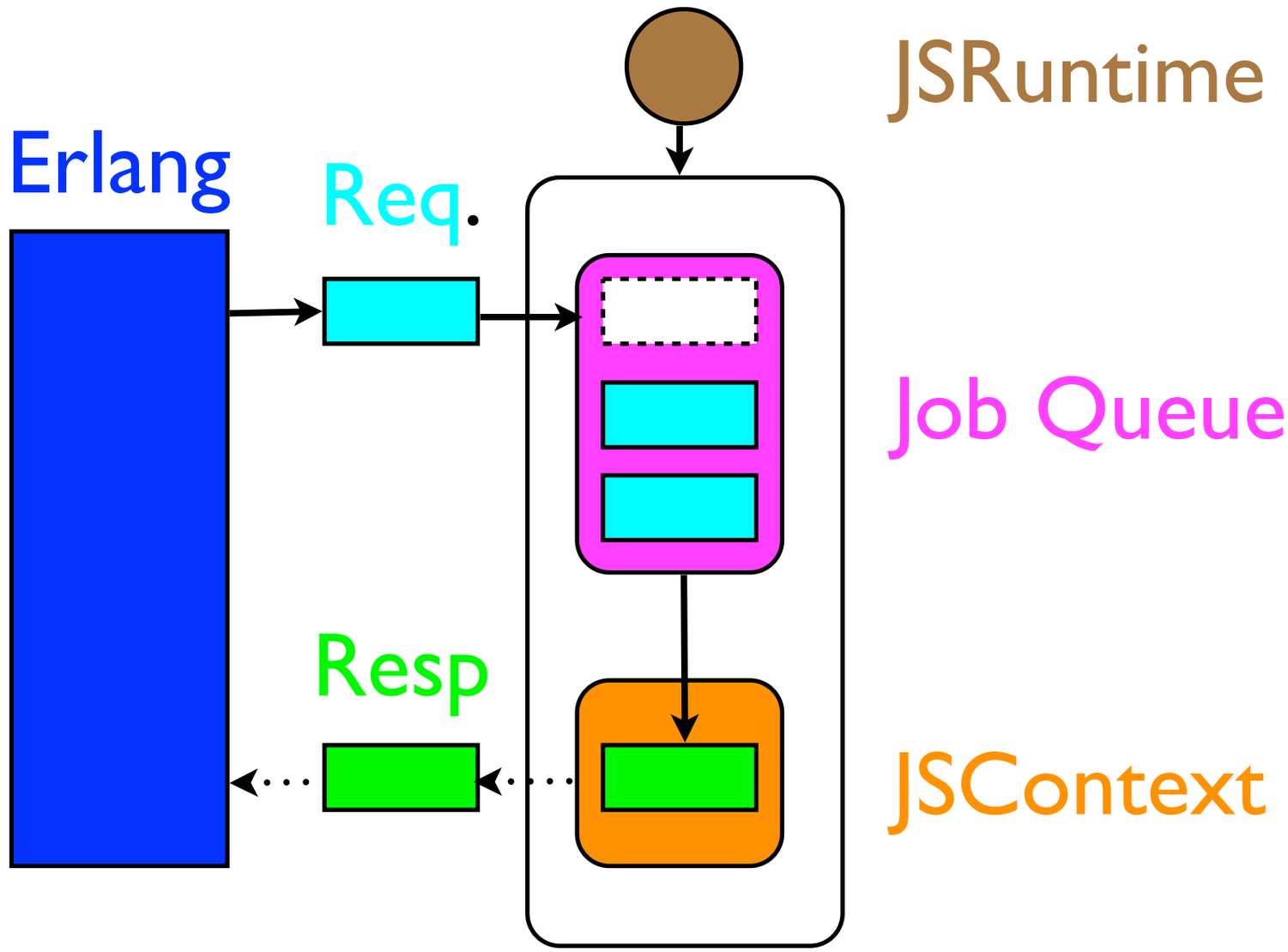
Schedulers: 2 (all tests)

SpiderMonkey API

- JSRuntime - Generally process global
- JSContext - An execution context, thread sensitive
- jsval - Opaque value for JS types

Emonk Outline

- JSRuntime - NIF priv_data member
- JSContext - ErlNifResourceType with a thread
- jsval - ERL_NIF_TERM



VM Resource

```
eval(Ctx, Script) ->
    eval(Ctx, Script, ?TIMEOUT).

eval(Ctx, Script, Timeout) ->
    Ref = make_ref(),
    ok = eval(Ctx, Ref, self(), Script),
    receive
        {Ref, Resp} ->
            Resp;
        Other ->
            throw(Other)
    after Timeout ->
        throw({error, timeout, Ref})
    end.

eval(_Ctx, _Ref, _Dest, _Script) ->
    not_loaded(?LINE).
```

```
struct state_t
{
    ErlNifResourceType*     res_type;
    JSRuntime*              runtime;
};

typedef struct state_t* state_ptr;
```

```c
struct vm_t {
    ErlNifTid           tid;
    ErlNifThreadOpts*   opts;
    JSRuntime*          runtime;
    queue_ptr           jobs;
    job_ptr             curr_job;
};

typedef struct vm_t* vm_ptr;

void
vm_destroy(ErlNifEnv* env, void* obj) {
    vm_ptr vm = (vm_ptr) obj;
    job_ptr job = job_create();
    void* resp;

    job->type = job_close;
    queue_push(vm->jobs, job);
    enif_thread_join(vm->tid, &resp);
    queue_destroy(vm->jobs);

    enif_thread_opts_destroy(vm->opts);
}
```

```c
static int
load(ErlNifEnv* env, void** priv, ENTERM load_info)
{
    ErlNifResourceType* res;
    state_ptr state = (state_ptr) enif_alloc(sizeof(struct state_t));
    const char* name = "Context";
    int flags = ERL_NIF_RT_CREATE | ERL_NIF_RT_TAKEOVER;

    state->res_type = enif_open_resource_type(
            env, NULL, name, vm_destroy, flags, NULL
        );

    if(state->res_type == NULL) {
        enif_free(state);
        return 1;
    }

    state->runtime = init_js_runtime();

    *priv = (void*) state;
    return 0;
}
```

```c
static ENTERM
create_ctx(ErlNifEnv* env, int argc, CENTERM argv[])
{
    state_ptr state = (state_ptr) enif_priv_data(env);
    unsigned int stack_size;
    vm_ptr vm;
    ENTERM ret;

    if(argc != 1 || !enif_get_uint(env, argv[0], &stack_size))
        return enif_make_badarg(env);

    vm = vm_init(
            state->res_type, state->runtime, (size_t) stack_size
        );
    if(vm == NULL)
        return util_mk_error(env, "vm_init_failed");

    ret = enif_make_resource(env, vm);
    enif_release_resource(vm);

    return util_mk_ok(env, ret);
}
```

```
vm_ptr
vm_init(ErlNifResourceType* res_type, JSRuntime* runtime, size_t size)
{
    int status;
    vm_ptr vm = (vm_ptr) enif_alloc_resource(
            res_type, sizeof(struct vm_t)
        );

    vm->runtime = runtime;
    vm->curr_job = NULL;
    vm->stack_size = size;

    vm->jobs = queue_create();

    vm->opts = enif_thread_opts_create("vm_thread_opts");
    status = enif_thread_create("", &vm->tid, vm_run, vm, vm->opts)
    if(status != 0) goto error;

    return vm;

error:
    enif_release_resource(vm);
    return NULL;
}
```

```
static ENTERM
eval(ErlNifEnv* env, int argc, CENTERM argv[])
{
    state_ptr state = (state_ptr) enif_priv_data(env);
    vm_ptr vm;
    ENPID pid;
    ENBINARY bin;

    if(argc != 4) return enif_make_badarg(env);

    if(!enif_get_resource(
            env, argv[0], state->res_type, (void**) &vm
        ))
        return enif_make_badarg(env);
    if(!enif_is_ref(env, argv[1]))
        return util_mk_error(env, "invalid_ref");
    if(!enif_get_local_pid(env, argv[2], &pid))
        return util_mk_error(env, "invalid_pid");
    if(!enif_inspect_binary(env, argv[3], &bin))
        return util_mk_error(env, "invalid_script");

    if(!vm_add_eval(vm, argv[1], pid, bin))
        return util_mk_error(env, "error_creating_job");

    return util_mk_atom(env, "ok");
}
```

```c
int
vm_add_eval(vm_ptr vm, ENTERM ref, ENPID pid, ENBINARY bin)
{
    job_ptr job = job_create();

    job->env = enif_alloc_env();
    job->type = job_eval;
    job->ref = enif_make_copy(job->env, ref);
    job->pid = pid;

    if(!enif_alloc_binary(bin.size, &(job->script))) goto error;
    memcpy(job->script.data, bin.data, bin.size);

    if(!queue_push(vm->jobs, job)) goto error;

    return 1;

error:
    if(job != NULL) job_destroy(job);
    return 0;
}
```

```c
void*
vm_run(void* arg)
{
    vm_ptr vm = (vm_ptr) arg;
    JSContext* cx;
    job_ptr job;
    ENTERM resp;
    cx = init_js_context();
    if(cx == NULL) goto done;
    while(1)
    {
        job = queue_pop(vm->jobs);
        if(job->type == job_close) {job_destroy(job); break;}
        if(job->type == job_eval) {
            resp = vm_eval(cx, gl, job);
        } else if(job->type == job_call) {
            resp = vm_call(cx, gl, job);
        }
        enif_send(NULL, &(job->pid), job->env, resp);
        job_destroy(job);
    }
done:
    if(cx != NULL) JS_DestroyContext(cx);
    return NULL;
}
```

```
ENTERM vm_eval(JSContext* cx, JSObject* gl, job_ptr job)
{
    ENTERM resp;
    const char* script = (const char*) job->script.data;
    size_t length = job->script.size;
    jsval rval;
    int cnt, i;

    for(i = 0, cnt = 0; i < length; i++) {
        if(script[i] == '\n') cnt += 1;
    }

    if(!JS_EvaluateScript(cx, gl, script, length, "", cnt, &rval)){
        if(job->error != 0)
            resp = vm_mk_error(job->env, job->error);
        else
            resp = vm_mk_error(
                    job->env, util_mk_atom(job->env, "unknown")
                );
    } else {
        resp = vm_mk_ok(job->env, to_erl(job->env, cx, rval));
    }

    return enif_make_tuple2(job->env, job->ref, resp);
}
```

Questions?