N2O

No Bullshit
Sane Framework
for Wild Web

N2O: No Bullshit
Sane Framework
For Wild Web.

FIRST EDITION

Book Design and Illustrations by Maxim Sokhatsky
Author Maxim Sokhatsky

Editors: Anton Logvinenko
Vladimir Kirillov
Viktor Sovietov
Dmitriy Sukhomlynov

Contents

To Mary and all sentient beings.

1 N2O: Web Framework

N2O was started as the first Erlang Web Framework that uses Web-Socket protocol only. We saved great compatibility with Nitrogen and added many improvements, such as binary page construction, binary data transfer, minimized process spawns, transmission of all events over the WebSocket and work within Cowboy processes. N2O renders pages several times faster than Nitrogen.

1.1 Wide Coverage

N2O is unusual in that it solves problems in different web development domains and stays small and concise at the same time. Started as a Nitrogen concept of server-side framework it can also build offline client-side applications using the same source code. This became possible with powerful Erlang JavaScript Parse Transform which enables running Erlang on JavaScript platform and brings in Erlang and JavaScript interoperability. You can use Elixir, LFE and Joxa languages for backend development as well.

N2O supports DSL and HTML templates. It lets you build JavaScript control elements in Erlang and perform inline rendering with DSL using the same code base for both client and server-side. How to use N2O is up to you. You can build mobile applications using server-side rendering for both HTML and JavaScript thus reducing CPU cycles and saving the battery of a mobile device. Or you can create rich offline desktop applications using Erlang JavaScript compiler.

Why Erlang in Web?

We have benchmarked all the existing modern web frameworks that were built using functional languages and Cowboy was still the winner. The chart below shows raw HTTP performance of functional and C-based languages with concurrent primitives (Go, D and Rust) on a VAIO Z notebook with i7640M processor.

Figure 1: Web-Servers raw performance grand congregation

Erlang was built for low latency streaming of binary data in telecom systems. It's fundamental design goal included high manageability, scalability and extreme concurrency. Thinking of WebSocket channels as binary telecom streams and web pages as user binary sessions helps to get an understanding reasons behind choosing Erlang over other alternatives for web development.

Using Erlang for web allows you to unleash the full power of telecom systems for building web-scale, event-driven, message-passing,

NoSQL, asynchronous, non-blocking, reliable, highly-available, performant, secure, real-time, distributed applications. See Erlang: The Movie II.

N2O outperforms full Nitrogen stack with only 2X raw HTTP Cowboy performance downgrade thus upgrading rendering performance several times compared to any other functional web framework. And sure it's faster than raw HTTP performance of Node.js.

1.2 Rich and Lightweight Applications

There are two approaches for designing client/server communication. The first one is called 'data-on-wire'. With this approach only JSON, XML or binary data are transferred over RPC and REST channels. All HTML rendering is performed on the client-side. This is the most suitable approach for building desktop applications. Examples include React, Meteor and ClojureScript. This approach can also be used for building mobile clients.

Another approach is sending pre-rendered parts of pages and JS and then replacing HTML and executing JavaScript on the client-side. This approach is better suited for mobile web development since the client doesn't have much resources.

With N2O you can create both types of applications. You can use N2O REST framework for desktop applications based on Cowboy REST API along with DTL templates for initial HTML rendering for mobile applications. You can also use Nitrogen DSL-based approach for modeling parts of pages as widgets and control elements, thanks to Nitrogen rich collection of elements provided by Nitrogen community.

In cases when your system is built around Erlang infrastructure, N2O is the best choice for fast web prototyping, bringing simplicity of use and clean codebase. Despite HTML being transfered over the wire, you still have access to all your Erlang services directly.

You can also create offline applications using Erlang JavaScript compiler just the way you would use ClojureScript, Scala.js, Elm, WebSharper or any other similar tool. N2O includes: REST micro frameworks, server-side and client-side rendering engines, WebSocket events streaming, JavaScript generation and JavaScript macro system along with **AVZ** authorization library (Facebook, Google, Twitter, Github, Microsoft), key-value storages access library **KVS** and **MQS** Message Bus client library (gproc, emqttd).

1.3 JSON and BERT

N2O uses JSON and BERT. All messages passed over WebSockets are encoded in native Erlang External Term Format. It is easy to parse it in JavaScript with **dec(msg)** and it helps to avoid complexity on the server-side. Please refer to http://bert-rpc.org[1] for detailed information.

1.4 DSL and Templates

We like Nitrogen for the simple and elegant way it constructs typed HTML with internal DSL. This is analogous to Scala Lift, OCaml Ocsigen and Haskell Blaze approach. It lets you develop reusable control elements and components in the host language.

Template-based approach (Yesod, ASP, PHP, JSP, Rails, Yaws and ChicagoBoss) requires developers to deal with raw HTML. It allows defining pages in terms of top-level controls, placeholders and panels. Thus N2O combines both approaches.

The main N2O advantage is its sutability for large-scale projects without sacrificing simplicity and comfort of prototyping solutions in fast and dynamic manner. Below is an example of complete Web Chat implementation using WebSockets that shows how Templates, DSL and asynchronous inter-process communication work in N2O.

---

[1]`http://bert-rpc.org`

Listing 1: chat.erl

```erlang
-module(chat).
-include_lib("nitro/include/nitro.hrl").
-compile(export_all).

main() ->
   #dtl { file    = "login",
          app     = review,
          bindings = [ { body, body() } ] }.

body() ->
  [ #span    { id=title,       body="Your nickname: " },
    #textbox { id=user,        body="Anonymous" },
    #panel   { id=history },
    #textbox { id=message },
    #button  { id=send,        source=[user,message],
                               body="Send",
                               postback=chat } ].

event(init) -> wf:reg(room), wf:async("looper",fun loop/1);
event(chat) -> User    = wf:q(user),
               Message = wf:q(message),
               n2o_async:send("looper",{chat,User,Message}).

loop({chat,User,Message}) ->
   Terms = #panel { body = [
           #span  { body = User }, ": ",
           #span  { body = Message } ]},
   wf:insert_bottom(history, Terms),
   wf:flush(room).
```

Just try to build the similar functionality with your favorite language/framework and feel the difference! Here are one message bus, one async **gen_server** worker under supervision, NITRO DSL, DTL template, WebSockets, HTML and JavaScript generation in a simple file that you can put in your N2O application directory tree without restart and manual compilation. Also you can create single-file bundle which is able to run in Windows, Linux and Mac.

Changes from Nitrogen

We took a liberty to break some compatibility with the original Nitrogen framework, mostly because we wanted to have a clean codebase and achieve better performance. However, it's still possible to port Nitrogen web sites to N2O quite easily. E.g., N2O returns id and class semantics of HTML and not **html_id**. We simplified HTML rendering without using **html_encode** which should be handled by application layer.

Nitrogen.js, originally created by Rusty Klophaus, was removed because of the pure WebSocket nature of N2O which doesn't require jQuery on the client-side anymore. In terms of lines of code we have impressive showing. New **xhr.js** 25 LOC and **bullet.js** 18 LOC was added as the replacement, also **nitrogen.js** takes only 45 LOC. UTF-8 **utf8.js** 38 LOC could be plugged separately only when you're using **bert.js** 200 LOC formatter. **n2o.js** protocol handler is about 20 LOC.

We also removed **simple_bridge** and optimized N2O on each level to unlock maximum performance and simplicity. We hope you will enjoy using N2O. We are fully convinced it is the most efficient way to build Web applications in Erlang.

Original Nitrogen was already tested in production under high load and we decided to remove **nprocreg** process registry along with **action_comet** heavy process creation. N2O creates a single process for an async WebSocket handler, all operations are handled within Cowboy processes.

Also, we introduced new levels of abstraction. You can extend the set of available protocols (Nitrogen, Heartbeat, Binary), change protocol formatters to BERT, JSON or MessagePack, inject your code on almost any level. The code structure is clean and Nitrogen compatibility layer NITRO is fully detachable from N2O and lives in a separate **synrc/nitro** application.

2 Setup

2.1 Prerequisites

To run N2O websites you need to install Erlang R18 or higher. N2O works on Windows, Mac and Linux.

2.2 Kickstart Bootstrap

To try N2O you only need to fetch it from Github and build. We don't use fancy scripts, so building process is OTP compatible: bootstrap site is bundled as an Erlang release.

```
$ git clone git://github.com/synrc/n2o
$ cd n2o/samples
$ ./mad deps compile plan repl
```

Now you can try: http://localhost:8000[2].
On Linux you should do at first:

```
$ sudo apt-get install inotify-tools
```

---

[2]http://localhost:8000

2.3 Manual Setup

If you want to start using N2O inside your application, you can use Cowboy dispatch parameter for passing HTTP, REST, WebSocket and Static N2O endpoints:

Listing 2: web_sup.erl

```
cowboy:start_http(http, 100,
    [{port, wf:config(n2o,port)}],
    [{env, [{dispatch, dispatch_rules()}]}]),

mime() -> [{mimetypes,cow_mimetypes,all}].

dispatch_rules() ->
  cowboy_router:compile(
    [{'_', [
    {"/css/[...]", n2o_dynalo, {dir, "apps/hello/priv", mime()}},
    {"/n2o/[...]", n2o_dynalo, {dir, "deps/n2o/priv", mime()}},
    {"/rest/:resource", rest_cowboy, []},
    {"/rest/:resource/:id", rest_cowboy, []},
    {"/ws/[...]", bullet_handler, [{handler, n2o_bullet}]},
    {'_', n2o_cowboy, []}
    ]}]).
```

N2O uses two ports for SSL and non-SSL connections:

```
wf:config(n2o,port,443)
```

You can also redefine these ports in bullet connection inside template:

```
<script>
    var transition = {pid: '', host: 'localhost', port: '443' };
</script>
```

And then add a minimal **index.erl** page:

Listing 3: index.erl

```
-module(index).
-compile(export_all).
-include_lib("n2o/include/wf.hrl").

main() -> #span{body="Hello"}.
```

3 Erlang Processes

3.1 Reduced Latency

The secret to reducing latency is simple. We try to deliver rendered HTML as soon as possible and render JavaScript only when Web-Socket initialization is complete. It takes three steps and three Erlang processes for doing that.
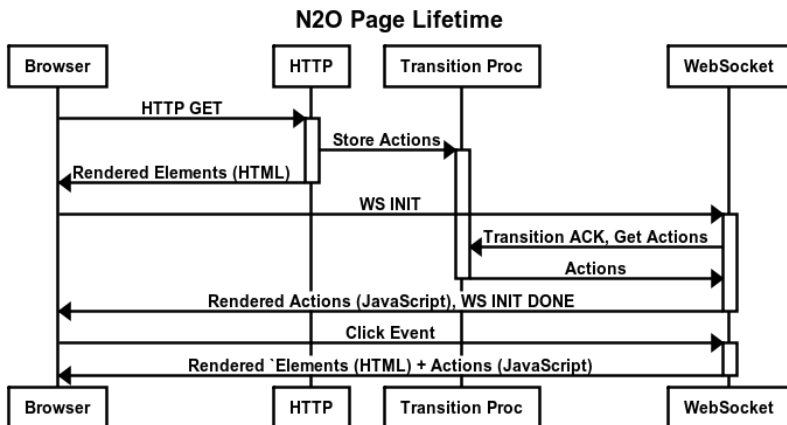
Figure 2: Page Lifetime

N2O request lifetime begins with the start of HTTP process serving the first HTML page. After that it dies and spawns Transition process. Then the browser initiates WebSocket connections to the similar URL endpoint. N2O creates persistent WebSocket process and the Transition process dies.

Your page could also spawn processes with **wf:async**. These are persistent processes that act like regular Erlang processes. This is a usual approach to organize non-blocking UI for file uploads and other time consuming operations.

3.2 Page Serving Process

This processes are applicable only to the case when you serving not static HTML, but dynamically rendered pages with NITRO, DTL or ET template engines. The very first HTTP handler only renders HTML. All created on the way JavaScript actions are stored in the transition process.

```
transition(Actions) ->
    receive {'N2O',Pid} -> Pid ! Actions end.
```

HTTP handler will die immediately after returning HTML. Transition process stores actions and waits for a request from a WebSocket handler.

3.3 Transition Process

Right after receiving HTML the browser initiates WebSocket connection thus starting WebSocket handler on the server. After responding with JavaScript actions the Transition process dies and the only process left running is WebSocket handler. At this point initialization phase is complete.

3.4 Events Process

After that all client/server communication is performed over Web-Socket channel. All events coming from the browser are handled by N2O, which renders elements to HTML and actions to JavaScript. Each user at any time has only one WebSocket process per connection.

3.5 Async Processes

These are user processes that were created with **wf:async** invocation. This processes was very useful to organize persiste stateful connection for legacy async technology like COMET for XHR channel. If you have problem with proxying WebSocket stream you can easily use XHR fallback that is provided by **xhr.js** N2O companion library. Async

processes are optional and only needed when you have a UI event taking too much time to be processed, like gigabyte file uploads. You can create multiple async processes per user. Starting from N2O 2.9 all async processes are being created as **gen_server** under **n2o_sup** supervision tree and new .

3.6 SPA Mode

In SPA mode your N2O can serve no HTML at all. N2O elements are bound during INIT handshake and thus can be used regularly as in DSL mode. In the example provided in n2o/samples you can find two different front end to the same **review** application which consist of two page modules **index** and **login**. You can access this application involving no HTML rendering by using static file serving that could be switched to direct nginx serving.

```
open http://localhost:8000/static/app/login.htm
```

Or you can see DTL rendered HTML pages which resides at following address:

```
open http://localhost:8000/login.htm
```

4 Endpoints

N2O Erlang Processes are instantiated and run by Web Server. Depending on Web Server endpoint bindings you can specify module for HTTP requests handling.

N2O comes with three endpoint handlers for each Web Server supported. However you are not required to use any of these. You can implement your own endpoint handlers, e.g. for using with Meteor.js or Angular.js and providing Erlang back-end event streaming from server-side. Here is an example of using HTTP, WebSocket and REST endpoint handlers with Cowboy Web Server.

```
{"/rest/:resource", rest_cowboy, []},
{"/rest/:resource/:id", rest_cowboy, []},
{"/ws/[...]", bullet_handler, [{handler, n2o_bullet}]},
{'_', n2o_cowboy, []}
```

4.1 HTML Pages over HTTP

This handler is used for serving initial dynamic HTML page. In case you are serving static HTML content this handler is not included into the running stack. **n2o_cowboy** is a default HTML page handler.

On initial page load **n2o_document:run** of page document endpoint is started. During its execution **wf_render:render** proceeds by calling **Module:main** selected by the routing handler.

4.2 JavaScript Events over WebSocket

JavaScript handler shares the same router information as the HTML handler because during its initial phase the same chain of N2O handlers is called.

This handler knows how to deal with XHR and WebSocket requests. **n2o_bullet** is a default JavaScript event handler based on Bullet library created by Loïc Hoguin.

You can send several types of events directly from JavaScript using various protocols. E.g. you man need to use client protocol:

```
JavaScript> ws.send(enc(tuple(atom('client'),
                   tuple(atom('phone_auth'),"+380..."))));
```

And catch this event at Erlang side:

```
event({client,{phone_auth,Phone}}) ->
    io:format("Phone: ~p~n",[Phone]).
```

You can also send direct messages to event/1, but use it carefully because it may violate security rules.

```
> ws.send(enc(tuple(atom('direct'),atom('init'))));
```

With catching at Erlang side:

```
event(init) ->
    io:format("Init called~n").
```

4.3 HTTP API over REST

REST handler's request context initialization differs for the one used by HTML and JavaScript handlers. N2O handler chains are not applied to REST requests. **rest_cowboy** is a default REST handler.

```
{"/rest/:resource", rest_cowboy, []},
{"/rest/:resource/:id", rest_cowboy, []},


-module(users).
-behaviour(rest).
-compile({parse_transform, rest}).
-include("users.hrl").
-export(?REST_API).
-rest_record(user).

init() -> ets:new(users,
                 [public, named_table, {keypos, #user.id}]).

populate(Users) -> ets:insert(users, Users).
exists(Id) -> ets:member(users, wf:to_list(Id)).
get() -> ets:tab2list(users).
get(Id) -> [User] = ets:lookup(users, wf:to_list(Id)), User.
delete(Id) -> ets:delete(users, wf:to_list(Id)).
post(#user{} = User) -> ets:insert(users, User);
post(Data) -> post(from_json(Data, #user{})).
```

Listing 4: users.erl

To add users to in-memory storage perform POST requests:

```
curl -i -X POST -d "id=vlad" localhost:8000/rest/users
curl -i -X POST -d "id=doxtop" localhost:8000/rest/users
curl -i -X GET localhost:8000/rest/users
curl -i -X PUT -d "id=5HT" localhost:8000/rest/users/vlad
curl -i -X GET localhost:8000/rest/users/5HT
curl -i -X DELETE localhost:8000/rest/users/5HT
```

5 Handlers

HTML and JavaScript Web Server HTTP handlers share the same system of context initialization.

```
init_context(Req) -> #context{
    actions=[], module=index, path=[],
    req=Req, params=[], session=undefined,
    handlers= [ {'query', wf:config('query', n2o_query)},
                {session, wf:config(session, n2o_session)},
                {route,   wf:config(route,   n2o_route)} ]}.
```

Chain of three N2O handlers that are always called on each HTTP request. You can redefine any of them or plug your own additional handler in the chain to transform web server requests.

```
fold(Fun,Handlers,Ctx) ->
    lists:foldl(fun({_,Module},Ctx1) ->
        {ok,_,NewCtx} = Module:Fun([],Ctx1),
        NewCtx end,Ctx,Handlers).
```
<center>Listing 5: wf_core:fold/3</center>

5.1 Query

Query Handler parses URL query and HTTP form information from HTTP request.

5.2 Session

Session Handler manages key-value in-memory database ETS table.

5.3 Router

You can specify routing table with application config:

```
{n2o, [{route,custom_router}]}
```

Remember that routing handler should be kept very simple because it influences overall initial page load latency and HTTP capacity.

```erlang
-module (n2o_route).
-author('Maxim Sokhatsky').
-include_lib("n2o/include/wf.hrl").
-export(?ROUTING_API).

finish(State, Ctx) -> {ok, State, Ctx}.
init(State, Ctx) ->
    Path = wf:path(Ctx#context.req),
    Module = route_prefix(Path),
    {ok, State, Ctx#context{path=Path,module=Module}}.

route_prefix(<<"/ws/",P/binary>>) -> route(P);
route_prefix(<<"/",P/binary>>) -> route(P);
route_prefix(P) -> route(P).

route(<<>>)             -> index;
route(<<"index">>)      -> index;
route(<<"login">>)      -> login;
route(<<"favicon.ico">>) -> static_file;
route(_) -> index.
```

Listing 6: n2o_route.erl

6 Protocols

We made it to make you happy. You can plug any binary formatting protocol like BERT, BED, MessagePack or even textual ones like TXT, JSON or custom binary protocols like MQTT, AMQP or legacy XMPP. Nitrogen-support here is just three protocol messages: **pickle**, **flush** and **delivery**. N2O also provides basic heartbeat protocol that can be formatted at your whim. Currently N2O supports 4 protocols: Heartbeat, Nitrogen, Client/Server, Binary.

```
protocols() ->
    wf:config(n2o,protocols,[ n2o_heart,
                              n2o_nitrogen,
                              n2o_client,
                              n2o_binary  ]).
```
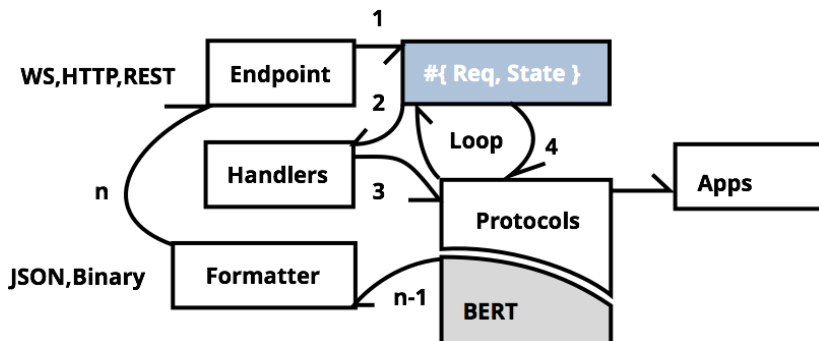
Figure 3: Messaging Pipeline

For example in Skyline (DSL) application you use only nitrogen and binary:

```
> wf:config(n2o,protocols).
[n2o_heart,n2o_nitrogen,n2o_binary]
```

And in Games (SPA) application you need only client/server protocol:

```
> wf:config(n2o,protocols).
[n2o_heart,n2o_client]
```

6.1 Heartbeat

Heartbeat protocol is essential WebSocket application level protocol
for PING and N2O initialization. It pings every 4-5 seconds from
client-side to server thus allowing to determine client online presence.
On reconnection or initial connect client sends N2O init marker telling
to server to reinitialize the context.

```
ws.send('PING');
ws.send('N2O,');
```

You can try manually send these messages in web console to see
whats happening, also you can enable logging the heartbeat protocol
by including its module in log_modules:

```
log_modules() -> [n2o_heart].
```

Heartbeat protocol PING request returns empty message NOP
binary response; N2O messages returns JSON with EVAL fields of
rendered actions.

6.2 Nitrogen Compatibility Layer

Pickled messages are used if you send messages over unencrypted
channel and want to hide the content of the message, that was
generated on server. You can use BASE64 pickling mechanisms with
optional AES/RIPEMD160 encrypting.

```
ws.send(enc(tuple(atom('pickle'),
    binary('ddtake'),
    binary('g2gCaAVkAAJldmQABWluZGV4ZAAEdGFzWsABH'+
        Rha2VkAAVldmVudGgDYgAABXViAAQKXmIAC3cK'),
    [tuple(atom('ddtake'),'0')])));
```

Where Base64 represents the N2O EVENT:

```
#ev{module=index,msg=take,trigger="take",name=event}
```

This is Nitrogen-based messaging model. Nitrogen WebSocket processes receive also flush and delivery protocol messages, but originated from server. These are internal Nitrogen protocol messages. This request will return JSON with EVAL field only.

6.3 Client/Server

Client messages usually originated at client and represent the Client API Requests:

```
ws.send(enc(tuple(
    atom('client'),
    tuple(atom('join_game'),1000001))));
```

Server messages are usually being sent to client originated on the server by sending **info** notifications directly to Web Socket process:

```
> WebSocketPid ! {server, Message}
```

You can obtain this Pid like here:

```
event(init) -> wf:info(?MODULE, "Web Socket Pid: ~p",[self()]);
```

You can also send server messages from client relays and vice versa. But it is up to your application and client/server handlers how to handle these messages.

```
ws.send(enc(tuple(
    atom('server'),
    tuple(atom('attach'),1000001))));
```

NOTE: client/server request may return JSON with EVAL and DATA fields.

JSON enveloped EVAL and DATA

Each message from Web Socket channel to Client is encoded as JSON object. N2O.js[3] is used to decode WebSocket binary messages from JSON container.

```
{ "eval": "ws.send("Send Back This String");",
  "data": [ 131,104,2,100,0,7,109,101,115,115,
            97,103,101,107,0,5,72,101,108,108,111 ] }
```

EVAL values are evaluated immediately and DATA values are passed to handle_web_socket(data) function if it exists.

```
function handle_web_socket(body) { console.log(body); }
```

JSON enveloped BERT

Usually in DATA come BERT messages (Binary Erlang Term Format). BERT.js[4] is used to decode application protocol message.

```
function handle_web_socket(body) {
    console.log(String(dec(body))); }

E> Received: {message,"Hello"}
```

6.4 Binary

When you need raw binary Blob on client-side, for images or other raw data, you can ask server like this:

```
ws.send(enc(tuple(
    atom('bin'),
    binary('API Request'));
```

---

[3]https://github.com/5HT/n2o/blob/master/priv/n2o.js
[4]https://github.com/5HT/n2o/blob/master/priv/bert.js

And handle also in binary clause:

```
event({binary,Message}) ->
    wf:info(?MODULE, "This API will return Raw Binary", []),
    <<84,0,0,0,108>>;
```

NOTE: if event returns not the binary, client will receive BERT encoded message.

BERT

Erlang RPC protocol interconnection with JavaScript nodes should be transferred as BERT answers.

```
function handle_web_socket(body) {
    console.log(String(dec(body))); }
```

RAW Binary

Raw images for fastest possible speed should be transferred as binary answers.

```
function handle_web_socket_blob(body) { }

E> Unknown Raw Binary Received: [72,101,108,108,111]
```

6.5 Rails-like State Protocol

N2O ships also with state protocol that requires from developer to specify explicitly the return values from events. It also allows you to use Rails-like semantic for backends:

```
Controller:Action(Event,Context).
```

It allows you also to name your events with any name like in ChicagoBoss.

7 JavaScript Compiler

7.1 Compilation and Macros

Erlang JavaScript/OTP Parse Transform has two modes defined by
jsmacro and **js** Erlang module attributes. The first mode precompiles
Erlang module functions into JavaScript strings. The second one
exports Erlang functions into a separate JavaScript file ready to run
in the browser or Node.js.

Sample usage of **jsmacro** and **js**:

```
-module(sample).
-compile({parse_transform, shen}).
-jsmacro([tabshow/0,doc_ready/1,event/3]).
-js(doc_ready/1).
```

7.2 Erlang Macro Functions

Macro functions are useful for using N2O as a server-side framework.
Functions get rewritten during Erlang compilation into a JavaScript
format string ready for embedding. Here is an example from N2O
pages:

```
tabshow() ->
    X = jq("a[data-toggle=tab]"),
    X:on("show",
        fun(E) -> T = jq(E:at("target")),
        tabshow(T:attr("href")) end).

doc_ready(E) ->
    D = jq(document),
    D:ready(fun() ->
        T = jq("a[href=\"#" ++ E ++ "\"]"),
        T:tab("show") end).

event(A,B,C) ->
    ws:send('Bert':encodebuf(
        [{source,'Bert':binary(A)}, {x,C},
         {pickle,'Bert':binary(B)}, {linked,C}])).

main() ->
```

```
        Script1 = tabshow(),
        Script2 = event(1, 2, 3),
        Script3 = doc_ready(wf:js_list("tab")),
        io:format("tabshow/0:~n~s~nevent/3:~n~s~ndoc_ready/1:~n~s~n",
            [Script1,Script2,Script3]).
```

Perform compilation and run tests:

```
$ erlc sample.erl
$ erl
> sample:main().
```

You'll get the following output:

```
tabshow/0:
    var x = $('a[data-toggle=tab]');
    x.on('show',function(e) {
        var t = $(e['target']);
        return tabshow(t.attr('href'));
    });

event/3:
    ws.send(Bert.encodebuf({source:Bert.binary(1),
                            x:3,
                            pickle:Bert.binary(2),
                            linked:3}));

doc_ready/1:
var d = $(document);
d.ready(function() {
    var t = $('a[href="#' + 'tab' + '"]');
    return t.tab('show');
});
```

As you see, no source-map needed.

7.3 JavaScript File Compilation

Export Erlang function to JavaScript file with **-js([sample/0,fun_-args/2])**. You could include functions for both **macro** and **js** definitions.

31

7.4 Mapping Erlang/OTP to JavaScript/OTP

Following OTP libraries are partially supported in Erlang JavaScript Parse Transform: **lists**, **proplists**, **queue**, **string**.

Example 1

```
S = lists:map(fun(X) -> X * X end,[1,2,3,4]),
```

transforms to:

```
s = [1,2,3,4].map(function(x) {
    return x * x;
});
```

Example 2

```
M = lists:foldl(fun(X, Acc) -> Acc + X end,0,[1,2,3,4]),
```

transforms to:

```
m = [1,2,3,4].reduce(function(x,acc) {
    return acc + x;
},0);
```

8 API

8.1 Update DOM wf:update

You can update part of the page or DOM element with a given element or even raw HTML. N2O comes with NITRO template engine based on Erlang records syntax and optimized to be as fast as DTL or EEX template engines. You may use them with **#dtl** and **#eex** template NITRO elements. N2O Review application provides a sample how to use DTL templates. For using Nitrogen like DSL first you should include **nitro** application to your rebar.config

```
{nitro,".*",{git,"git://github.com/synrc/nitro",{tag,"2.9"}}},
```

And also plug it in headers to your erlang page module:

```
-include("nitro/include/nitro.hrl").
```

Here is an example of simple **#span** NITRO element with an HTML counterpart.

```
wf:update(history,[#span{body="Hello"}]).
```

It generates DOM update script and sends it to WebSocket channel for evaluation:

```
document.querySelector('#history')
        .outerHTML = '<span>Hello</span>';
```

Companions are also provided for updating head and tail of the elements list: **wf:insert_top/2** and **wf:insert_bottom/2**. These are translated to appropriate JavaScript methods **insertBefore** and **appendChild** during rendering.

```
wf:insert_top(history,
    #panel{id=banner, body= [
        #span{ id=text,
               body = wf:f("User ~s logged in.",[wf:user()]) },
        #button{id=logout, body="Logout", postback=logout },
        #br{} ]}),
```

Remember to envelop all elements in common root element before inserts.

For relative updates use **wf:insert_before/2** and **wf:insert_after/2**. To remove an element use **wf:remove/2**.

Element Naming   You can specify element's id with Erlang atoms, lists or binaries. During rendering the value will be converted with **wf:to_list**. Conversion will be consistent only if you use atoms. Otherwise you need to care about illegal symbols for element accessors.

During page updates you can create additional elements with runtime generated event handlers, perform HTML rendering for template elements or even use distributed map/reduce to calculate view. You have to be aware that heavy operations will consume more power in the browser, but you can save it by rendering HTML on server-side. All DOM updates API works both using JavaScript/OTP and server pages.

List of elements you can use is given in **Chapter 9**. You can also create your own elements with a custom render function. If you want to see how custom element are being implemented you may refer to **synrc/extra** packages where some useful controls may be found like file uploader, calendar, autocompletion textboxlist and HTML editor.

8.2 Wire JavaScript wf:wire

Just like HTML is generated from Elements, Actions are rendered into JavaScript to handle events raised in the browser. Actions are always transformed into JavaScript and sent through WebSockets pipe.

Direct Wiring

There are two types of actions. First class are direct JavaScript strings provided directly as Erlang lists or via JavaScript/OTP transformations.

```
wf:wire("window.location='http://synrc.com'").
```

Actions Render

Second class actions are in fact Erlang records rendered during page load, server events or client events.

```
wf:wire(#alert{text="Hello!"}).
```

However basic N2O actions that are part of N2O API, **wf:update** and **wf:redirect**, are implemented as Erlang records as given in the example. If you need deferred rendering of JavaScript, you can use Erlang records instead of direct wiring with Erlang lists or JavaScript/OTP.

Any action, wired with **wf:wire**, is enveloped in **#wire**{**actions=[]**}, which is also an action capable of polymorphic rendering of custom or built-in actions, specified in the list. Following nested action embedding is also valid:

```
wf:wire(#wire{actions=[#alert{text="N2O"}]}).
```

You mau try to see how internally wiring is working:

```
> wf:actions().
  []

> wf:wire(#alert{text="N2O"}).
  [#wire{ancestor = action,trigger = undefined,
         target = undefined,module = action_wire,
         actions = #alert{ancestor = action,
                          trigger = undefined,
                          target = undefined,
                          module = action_alert,
                          actions = undefined,
                          source = [], text = "N2O"},
          source = []}]

> iolist_to_binary(wf:render(wf:actions())).
  <<"alert(\"N2O\");">>
```

Consider wiring **#event** if you want to add listener to existed element on page:

```
> wf:wire(#event{target=btn,postback=evt}).
  []

> rp(iolist_to_binary(wf:render(wf:actions()))).
  <<"{var x=qi('btn'); x && x.addEventListener('default',
  function (event){{ if (validateSources([])) ws.send(enc
  (tuple(atom('pickle'),bin('btn'),bin('g2gCaAVkAAJldmQAB
  WluZGV4ZAADZXZ0awADYnRuZAAFZXZlbnRoA2IAAAWhYgAG8wViAAHk
  tw=='),[tuple(tuple(utf8_toByteArray('btn'),bin('detail
  ')),event.detail)]))));else console.log('Validation Erro
  r'); }});};">>
```

8.3 Message Bus wf:reg and wf:send

N2O uses **gproc** process registry for managing async processes pools. It is used as a PubSub message bus for N2O communications. You can associate a process with the pool with **wf:reg** and send a message to the pool with **wf:send**.

```
loop() ->
    receive M ->
        wf:info(?MODULE, "P: ˜p, M: ˜p",[self(),M]) end, loop().
```

Now you can test it

```
> spawn(fun() -> wf:reg(topic), loop() end).
> spawn(fun() -> wf:reg(topic), loop() end).
> wf:send(topic,"Hello").
```

It should print in REPL something like:

```
> [info] P: <0.2012.0>, M: "Hello"
> [info] P: <0.2015.0>, M: "Hello"
```

Custom Registrator You may want to replace built-in **gproc** based PubSub registrator with something more robust like MQTT and AMQP or something more internal like **pg2**. All you need is to implement following API:

```
-module(mqtt_mq).
-compile(export_all).

send(Topic, Message) -> mqtt:publish(Topic, Message).
reg(Topic)           -> mqtt:subscribe(Topic, Message).
reg(Topic,Tag)       -> mqtt:subscribe(Topic, Tag, Message).
unreg(Topic)         -> mqtt:unsubscribe(Topic).
```

And set it in runtime:

```
> application:set_env(n2o,mq,mqtt_mq).
```

8.4 Async Processes wf:async and wf:flush

Function **wf:async** creates Erlang process, which communicate with the primary page process by sending messages. **wf:flush** should be called to redirect all updates and wire actions back to the page process from its async counterpart. Usually you send messages to async processes over N2O message bus **wf:send**, but you can use also **n2o_async:send**. In following example different variants are gives, both incrementing counter by 2. Also notice the async process initialization throung **init** message.

```
body()       -> [ #span  { id=display, body="0"},
                  #button { id=send, body="Inc",
                            postback=inc} ].

event(init) -> wf:async("counter",fun loop/1);
event(inc)  -> wf:send(counter,up),
               n2o_async:send("counter",up).

loop(init)  -> wf:reg(counter), put(counter,0);
loop(up)    -> C = get(counter) + 1,
               put(counter,C),
               wf:update(display,
                   #span{id=display,body=wf:to_binary(C)}),
               wf:flush().
```

Process Naming The name of async process is globally unique. There are two versions, **wf:async/1** and **wf:async/2**. In the given example the name of async process is specified as "counter", otherwise, if the first parameter was not specified, the default name "looper" will be used. Internally each async process includes custom key which is settled by default to session id.

```
> supervisor:which_children(n2o_sup).
  [{{async,{"counter",<<"d43adcc79dd64393a1eb559227a2d3fd">>}},
    <0.11564.0>,worker,
    [n2o_async]}]
```

8.5 Parse URL and Context parameters wf:q and wf:qp

These are used to extract URL parameters or read from the process context. **wf:q** extracts variables from the context stored by controls postbacks. **wf:qp** extracts variables from URL params provieded by cowboy bridge. **wf:qc** extracts variables from **#cx.params** context parsed with custom query handler during endpoint initialization usually performed inside N2O with something like.

```
Ctx = wf_context:init_context(Req),
NewCtx = wf_context:fold(init,Ctx#cx.handlers,Ctx),
wf:context(NewCtx),
```

8.6 Render wf:render or nitro:render

Render elements or actions with common render. Rendering is usually done automatically inside N2O, when you use DOM or Wiring API, but sometime you may need manual render, e.g. in static site generators and other NITRO applications which couldn't be even dependent from N2O. For that purposes you may use NITRO API

```
> nitro:render(#button{id=id,postback=signal}).
  <<"<button id=\"id\" type=\"button\"></button>">>
```

This is simple sample you may use in static site generators, but in N2O context you also may need to manual render JavaScript actions produced during HTML rendering. First of all you should know that process in which you want to render should be initialized with N2O **#cx** context. Here is example of JavaScript produced during previous **#button** rendering:

```
> wf:context(wf_context:init_context([])).
  undefined

> rp(iolist_to_binary(nitro:render(wf:actions()))).
  <<"{var x=qi('id'); x && x.addEventListener('click',
  function (event){{ if (validateSources([])) ws.send(
  enc(tuple(atom('pickle'),bin('id'),bin('g2gCaAVkAAJl
  dmQABWluZGV4ZAAGc2lnbmFsawACaWRkAAVldmVudGGDYgAABaFi
  AAbo0GIACnB4'),[tuple(tuple(utf8_toByteArray('id'),b
  in('detail')),event.detail)])));else console.log('Va
  lidation Error'); }});};"">>
```

40

Here is another more complex example of menu rendering using NITRO DSL:

```
menu(Files,Author) ->
   #panel{id=navcontainer,body=[#ul{id=nav,body=[

   #li{body=[#link{href="#",body="Navigation"},#ul{body=[
             #li{body=#link{href="/1.htm",body="Root"}},
             #li{body=#link{href="../1.htm",body="Parent"}},
             #li{body=#link{href="1.htm",body="This"}}]}]},

   #li{body=[#link{href="#",body="Download"},#ul{body=[
             #li{body=#link{href=F,body=F}}|| F <- Files ] }]},

   #li{body=[#link{href="#",body="Translations"},#ul{body=[
             #li{body=#link{href="#",body=Author}}]}]}]}]}.
```

```
> rp(iolist_to_binary(wf:render(menu(["1","2"],"5HT")))).
<<"<div id=\"navcontainer\"><ul id=\"nav\"><li>
<a href=\"#\">Navigation</a><ul><li><a href=\"/
1.htm\">Root</a></li><li><a href=\"../1.htm\">P
arent</a></li><li><a href=\"1.htm\">This</a></l
i></ul></li><li><a href=\"#\">Download</a><ul><
li><a href=\"1\">1</a></li><li><a href=\"2\">2<
/a></li></ul></li><li><a href=\"#\">Translation
s</a><ul><li><a href=\"#\">5HT</a></li></ul></l
i></ul></div>">>
```

Also notice some helpful functions to preprocess HTML and JavaScript escaping to avois XSS attacks:

```
> wf:html_encode(wf:js_escape("alert('N2O');")).
"alert(\\&#39;N2O\\&#39;);"
```

8.7 Redirects wf:redirect

Redirects are implemented not with HTTP headers, but with JavaScript action modifying **window.location**. This saves login context information which is sent in the first packet upon establishing a WebSocket connection.

8.8 Session Information wf:session

Store any session information in ETS tables. Use **wf:user**, **wf:role** for login and authorization. Consult **AVZ** library documentation.

8.9 Bridge information wf:header and wf:cookie

You can read and issue cookie and headers information using internal Web-Server routines. You can also read peer IP with **wf:peer**. Usually you do Bridge operations inside handlers or endpoints.

```
wf:cookies_req(?REQ),
wf:cookie_req(Name,Value,Path,TTL,Req)
```

You can set cookies for the page using public cookies API during initial page rendering.

```
body() -> wf:cookie("user","Joe"), [].
```

You should use wiring inside WebSocket events:

```
event(_) ->
    wf:wire(wf:f("document.cookie='~s=~s'",["user","Joe"])).
```

9 Elements

With N2O you don't need to use HTML at all. Instead you define your page in the form of Erlang records so that the page is type checked at the compile time. This is a classic CGI approach for compiled pages and it gives us all the benefits of compile time error checking and provides DSL for client and server-side rendering.

Nitrogen elements, by their nature, are UI control primitives that can be used to construct Nitrogen pages with Erlang internal DSL. They are compiled into HTML and JavaScript. Behavior of all elements is controlled on server-side and all the communication between browser and server-side is performed over WebSocket channels. Hence there is no need to use POST requests or HTML forms.

9.1 Static Elements: HTML

The core set of HTML elements includes br, headings, links, tables, lists and image tags. Static elements are transformed into HTML during rendering.

Static elements could also be used as placeholders for other HTML elements. Usually "static" means elements that don't use postback parameter:

```
#textbox { id=userName, body= <<"Anonymous">> },
#panel { id=chatHistory, class=chat_history }
```

This will produce the following HTML code:

```
<input value="Anonymous" id="userName" type="text"/>
<div id="chatHistory" class="chat_history"></div>
```

9.2 Active Elements: HTML and JavaScript

There are form elements that provide information for the server and gather user input: button, radio and check buttons, text box area and password box. Form elements usually allow to assign an Erlang postback handler to specify action behavior. These elements are compiled into HTML and JavaScript. For example, during rendering, some Actions are converted to JavaScript and sent to be executed in the browser. Element definition specifies the list of **source** elements that provide data for event's callback.

```
{ok,Pid} = wf:async(fun() -> chat_loop() end),
#button { id=sendButton, body= <<"Send">>,
          postback={chat,Pid}, source=[userName,message] }.
```

This will produce the following HTML:

```
<input value="Chat" id="sendButton" type="button"/>
```

and JavaScript code:

```
$('#sendButton').bind('click',function anonymous(event) {
    ws.send(Bert.encodebuf({
        source: Bert.binary('sendButton'),
        pickle: Bert.binary('g1AAAINQAAAAdX...'),
        linked: [
            Bert.tuple(Bert.atom('userName'),
            utf8.toByteArray($('#userName').val())),
            Bert.tuple(Bert.atom('message'),
            utf8.toByteArray($('#message').val()))] })); });
```

If postback action is specified then the page module must include a callback to handle postback info:

```
event({chat,Pid}) ->
    wf:info(?MODULE, "User ~p Msg ~p",
            [wf:q(userName),wf:q(message)]).
```

45

9.3 Base Element

Each HTML element in N2O DSL has record compatibility with the base element.

```
#element { ancestor=element,
           module,
           id,
           actions,
           class=[],
           style=[],
           source=[],
           data_fields=[],
           aria_states=[],
           body,
           role,
           tabindex,
           show_if=true,
           html_tag=Tag,
           title }.
```

Here **module** is an Erlang module that contains a render function. Data and Aria HTML custom fields are common attributes for all elements. In case element name doesn't correspond to HTML tag, **html_tag** field provided. **body** field is used as element contents for all elements.

Most HTML elements are defined as basic elements. You can even choose element's name different from its original HTML tag name:

```
-record(h6,    ?DEFAULT_BASE).
-record(tbody, ?DEFAULT_BASE).
-record(panel, ?DEFAULT_BASE_TAG(<<"div">>)).
-record('div', ?DEFAULT_BASE_TAG(<<"div">>)).
```

9.4 DTL Template #dtl

DTL stands for Django Template Language. A DTL element lets to construct HTML snippet from template with given placeholders for further substitution. Fields contain substitution bindings proplist, filename and templates folder.

```
-record(dtl, {?ELEMENT_BASE(element_dtl),
        file="index",
        bindings=[],
        app=web,
        folder="priv/templates",
        ext="html",
        bind_script=true }).
```

Consider we have **prod.dtl** file in **priv/templates** folder with two placeholders {{title}}, {{body}} and default placeholder for JavaScript {{script}}. All placeholders except {{script}} should be specified in #dtl element. Here is an example of how to use it:

```
body() -> "HTML Body".
main() ->
  [ #dtl { file="prod", ext="dtl",
           bindings=[{title,<<"Title">>},{body,body()}]} ].
```

You can use templates not only for pages, but for controls as well. Let's say we want to use DTL iterators for constructing list elements:

```
{% for i in items %} <a href="{{i.url}}">{{i.name}}</a><br>
        {% empty %} <span>No items available :-(</span>
        {% endfor %}
```

Listing 7: table.html

Here is an example of how to pass variables to the DTL template we've just defined:

```
#dtl{file="table", bind_script=false, bindings=[{items,
  [ {[{name, "Apple"},      {url, "http://apple.com"}]},
    {[{name, "Google"},     {url, "http://google.com"}]},
    {[{name, "Microsoft"}, {url, "http://microsoft.com"}]} ]}]}.
```

bind_script should be set to true for page templates. When control elements are rendered from DTL, bind_script should be set to false.

9.5 Button #button

```
-record(button, {?ELEMENT_BASE(element_button),
        type= <<"button">>,
        name,
        value,
        postback,
        delegate,
        disabled}).
```

Sample:

```
#button { id=sendButton, body= <<"Send">>,
        postback={chat,Pid}, source=[userName,message] }.
```

9.6 Link #dropdown

```
-record(dropdown, {?ELEMENT_BASE(element_dropdown),
        options,
        postback,
        delegate,
        value,
        multiple=false,
        disabled=false,
        name}).

-record(option, {?ELEMENT_BASE(element_select),
        label,
        value,
        selected=false,
        disabled}).
```

48

Sample:

```
#dropdown { id=drop,
            value="2",
            postback=combo,
            source=[drop], options=[
    #option { label= <<"Microsoft">>, value= <<"Windows">> },
    #option { label= <<"Google">>,    value= <<"Android">> },
    #option { label= <<"Apple">>,     value= <<"Mac">> }
]},
```

9.7 Link #link

```
-record(link, {?ELEMENT_BASE(element_link),
        target,
        url="javascript:void(0);",
        postback,
        delegate,
        name}).
```

9.8 Text Editor #textarea

```
-record(textarea, {?ELEMENT_BASE(element_textarea),
        placeholder,
        name,
        cols,
        rows,
        value}).
```

10 Actions

#action is the basic record for all actions. It means that each action has **#action** as its ancestor.

```
#action { ancestor,
          target,
          module,
          actions,
          source=[] }.
```

target specifies an element where this action will arise.

10.1 JavaScript DSL #jq

JavaScript query selector action mimics JavaScript calls and assignments. Specific action may be performed depending on filling**property** or **method** fields.

```
-record(jq, {?ACTION_BASE(action_jq),
         property,
         method,
         args=[],
         right }).
```

Here is an example of method calls:

```
wf:wire(#jq{target=n2ostatus,method=[show,select]}).
```

unfolded to calls:

```
document.querySelector('#n2ostatus').show();
document.querySelector('#n2ostatus').select();
```

And here is example of property chained assignments:

```
wf:wire(#jq{target=history,property=scrollTop,
    right=#jq{target=history,property=scrollHeight}}).
```

which transforms to:

```
document.querySelector('#history').scrollTop =
    document.querySelector('#history').scrollHeight;
```

Part of N2O API is implemented using #jq actions (updates and redirect). This action is introduced as transitional in order to move from Nitrogen DSL to using pure JavaScript transformations.

Event Actions

Objects passed over WebSockets channel from server to client are called **actions**. Objects passed over the same channel from client to server are called **events**. However events themselves are bound to HTML elements with **addEventListener** and in order to perform these bindings, actions should be sent first. Such actions are called **event actions**. There are three types of event actions.

10.2 Page Events #event

Page events are regular events routed to the calling module. Postback field is used as the main routing argument for **event** module function. By providing **source** elements list you specify HTML controls values sent to the server and accessed with **wf:q** accessor from the page context. Page events are normally generated by active elements like **#button**, **#link**, **#textbox**, **#dropdown**, **#select**, **#radio** and others elements contain postback field.

Control events are used to solve the need of element writers. When you develop your own control elements, you usually want events to be routed not to page but to element module. Control events were introduced for this purpose.

10.3 API Events #api

When you need to call Erlang function from JavaScript directly you should use API events. API events are routed to page module with **api_event/3** function. API events were used in **AVZ** authorization

library. Here is an example of how JSON login could be implemented using **api_event**:

```
api_event(appLogin, Args, Term) ->
    Struct = n2o_json:decode(Args),
    wf:info(?MODULE, "Granted Access"),
    wf:redirect("/account").
```

And from JavaScript you call it like this:

```
document.appLogin(JSON.stringify(response));
```

All API events are bound to root of the HTML document.

10.4 Message Box #alert

Message box **alert** is a very simple dialog that could be used for client debugging. You can use **console.log** along with alerts.

```
event({debug,Var}) ->
    wf:wire(#alert{text="Debug: " ++ wf:to_list(Var)}),
```

10.5 Confirmation Box #confirm

You can use confirmation boxes for simple approval with JavaScript **confirm** dialogs. You should extend this action in order to build custom dialogs. Confirmation box is just an example of how to organize this type of logic.

```
event(confirm) ->
    wf:wire(#confirm{text="Are you happy?",postback=continue}),

event(continue) -> wf:info(?MODULE, "Yes, you're right!", []);
```

11 UTF-8

11.1 Erlang

The main thing you should know about Erlang unicode is that

```
unicode:characters_to_binary("Uni") == <<"Uni"/utf8>>.
```

I.e. in N2O DSL you should use:

```
#button{body= <<"Unicode Name"/utf8>>}
```

11.2 JavaScript

Whenever you want to send to server the value from DOM element you should use utf8_toByteArray.

```
> utf8_toByteArray(document.getElementById('phone').value);
```

However we created shortcut for that purposes which knows about radio, fieldset and other types of DOM nodes. So you should use just:

```
> querySource('phone');
```

querySource JavaScript function ships in nitrogen.js which is part of N2O JavaScript library.

Whenever you get unicode data from server you should prepare it before place in DOM with utf8_decode:

```
> console.log(utf8_decode(receivedMessage));
```

12 MAD: Build and Packaging Tool

12.1 History

We came to conclusion that no matter how perfect your libraries are, the comfort and ease come mostly from development tools. Everything got started when Vladimir Kirillov[5] 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[6] emerged. Under the hood active is a client subscriber of fs[7] 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. Rebar 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[8].

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[9], who implemented the first prototype called 'mad'. Initially mad was able to compile DTL templates, YECC files, escript (like bundled in gproc), and it also had support for caching with side-effects.

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

Listing 8: Example of building N2O sample

---

[5]https://github.com/proger
[6]https://github.com/synrc/active
[7]https://github.com/synrc/fs
[8]https://github.com/synrc/otp.mk
[9]https://github.com/s1n4

```
                                   Hot
    make (erlang.mk)               2.588s
    mad compile                    2.521s
```
Listing 9: Example of building Cowboy

12.2 Introduction

We were trying to make something minimalistic that fits out Web Stack[10]. Besides we wanted to use our knowledge of other build tools like lein, sbt etc. Also for sure we tried sinan, ebt, Makefile-based scripts.

Synrc mad has a simple interface as follows:

```
BNF:
    invoke := mad params
    params := [] | run params
       run := command [ options ]
   command := app | lib | deps | compile | bundle
              start | stop | repl
```

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

12.3 Single-File Bundling

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 target escript is ready for 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 creating the copy with the same path locally near escript. After launch all files are copied to ETS. N2O also comes with custom cowboy static handler that is able

---

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

to read static files from this cached ETS filesystem. Also bundles are compatible with active online realoading and recompilation.

12.4 Templates

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

```
# mad app sample
# cd sample
# mad deps compile plan bundle web\_app
```

After that you can just run escript web_app under Windows, Linux and Mac and open http://localhost:8000[11].

```
C:\> escript web_app
Applications: [kernel,stdlib,crypto,cowlib,ranch,cowboy,compiler,
               syntax_tools,erlydtl,gproc,xmerl,n2o,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]

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

12.5 Deploy

mad is also supposed to be a deploy tool with ability to deploy not only to our resources like Erlang on Xen, Voxoz (LXC/Xen) but also to Heroku and others.

---

[11]http://localhost:8000

12.6 OTP Compliant

mad supports rebar umbrella project structure. Specifically two kinds of directory layouts:

```
apps
deps
rebar.config
sys.config
```

<div align="center">Listing 10: Solution</div>

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

<div align="center">Listing 11: OTP Application</div>

12.7 Apps Ordering

As you may know, you can create OTP releases with reltool (rebar generate) or systools (relx). mad currently creates releases with relx but is going to do it independently soon. Now it can only order applications.

```
# mad plan
Ordered: [kernel,stdlib,mnesia,kvs,crypto,cowlib,ranch,
          cowboy,compiler,syntax_tools,erlydtl,gproc,
          xmerl,n2o,n2o_sample,fs,active,mad,rest,sh]
```

And the good part about mad is it's size:

```
            Sources         Binary
mad         567 LOC         39 KB
rebar       7717 LOC        181 KB
```

13 KVS: Abstract Erlang Database

KVS is an Erlang abstraction over various native Erlang key-value databases, like Mnesia. Its meta-schema includes only concept of iterators (persisted linked lists) that are locked or guarded by containers (list head pointers). All write operations to the list are serialized using a single Erlang process to provide sequential consistency. The application which starts Erlang processes per container called feeds[12].

The best use-case for KVS and key-value storages is to store operational data. This data should be later fed to SQL data warehouses for analysis. Operational data stores should be scalable, secure, fault-tolerant and available. That is why we store work-in-progress data in key-value storages.

KVS also supports queries that require secondary indexes, which are not supported by all backends. Currently KVS includes following storage backends: Mnesia, Riak and KAI[13].

13.1 Polymorphic Records

Any data in KVS is represented by regular Erlang records. The first element of the tuple as usual indicates the name of bucket. The second element usually corresponds to the index key field.

```
            Rec = {user,"maxim@synrc.com",[]}.

RecordName = element(1, Rec).
        Id = element(2, Rec).
```

---

[12]https://github.com/synrc/feeds
[13]https://github.com/synrc/kai

13.2 Iterators

Iterator is a sequence of fields used as interface for all tables represented as doubly-linked lists. It defines id, next, prev, feed_id fields. This fields should be at the beginning of user's record, because KVS core is accessing relative position of the field (like #iterator.next) with setelement/element BIF, e.g.

```
setelement(#iterator.next, Record, NewValue).
```

Iterator is a sequence of fields used as interface for all tables represented as doubly-linked lists. It defines id, next, prev, feed_id fields and should be in the begin of the record's memory footprint because KVS core is accessing relative position of the field (like #iterator.next) with setelement/element BIF.

All records could be chained into the double-linked lists in the database. So you can inherit from the ITERATOR record just like that:

```
-record(access, {?ITERATOR(acl),
    entry_id,
    acl_id,
    accessor,
    action}).

#iterator { record_name,
            id,
            version,
            container,
            feed_id,
            prev,
            next,
            feeds,
            guard }
```

This means your table will support add/remove linked list operations to lists.

```
1> kvs:add(#user{id="mes@ua.fm"}).
2> kvs:add(#user{id="dox@ua.fm"}).
```

Read the chain (undefined means all)

```
3> kvs:entries(kvs:get(feed, user), user, undefined).
[#user{id="mes@ua.fm"},#user{id="dox@ua.fm"}]
```

or just

```
4> kvs:entries(user).
[#user{id="mes@ua.fm"},#user{id="dox@ua.fm"}]
```

Read flat values by all keys from table:

```
4> kvs:all(user).
[#user{id="mes@ua.fm"},#user{id="dox@ua.fm"}]
```

13.3 Containers

If you are using iterators records this automatically means you are
using containers. Containers are just boxes for storing top/heads of
the linked lists. Here is layout of containers:

```
#container { record_name,
             id,
             top,
             entries_count }
```

13.4 Extending Schema

Usually you only need to specify custom Mnesia indexes and tables
tuning. Riak and KAI backends don't need it. Group your table into
table packages represented as modules with handle_notice API.

```
-module(kvs_feed).
-inclue_lib("kvs/include/kvs.hrl").

metainfo() ->
    #schema{name=kvs,tables=[

        #table{ name = feed, container = true,
                fields = record_info(fields,feed)},

        #table{ name = entry, container = feed,
                fields = record_info(fields,entry),
                keys = [feed_id,entry_id,from] },

        #table{ name = comment, container = feed,
                fields = record_info(fields,comment),
                keys = [entry_id,author_id] } ]}.
```

And plug it into schema sys.config:

```
{kvs, {schema,[kvs_user,kvs_acl,kvs_feed,kvs_subscription]}},
```

After run you can create schema on local node with:

```
1> kvs:join().
```

It will create your custom schema.

13.5 KVS API

13.6 Service

System functions for start and stop service:

```
-spec start() -> ok | {error,any()}.
-spec stop() -> stopped.
```

13.7 Schema Change

This API allows you to create, initialize and destroy the database schema. Depending on database the format and/or feature set may differ. **join/1** function is used to initialize database, replicated from remote node along with its schema.

```
-spec destroy() -> ok.
-spec join() -> ok | {error,any()}.
-spec join(string()) -> [{atom(),any()}].
-spec init(atom(), atom()) -> list(#table{}).
```

13.8 Meta Info

This API allows you to build forms from table metainfo. You can also use this API for metainfo introspection.

```
-spec modules() -> list(atom()).
-spec containers() -> list(tuple(atom(),list(atom()))).
-spec tables() -> list(#table{}).
-spec table(atom()) -> #table{}.
-spec version() -> {version,string()}.
```

13.9 Chain Ops

This API allows you to modify the data, chained lists. You can use
create/1 to create the container. You can add and remove nodes from
lists.

```
-spec create(atom()) -> integer().
-spec remove(tuple()) -> ok | {error,any()}.
-spec remove(atom(), any()) -> ok | {error,any()}.
-spec add(tuple()) -> {ok,tuple()} |
                      {error,exist} |
                      {error,no_container}.
```

13.10 Raw Ops

These functions will patch the Erlang record inside database.

```
-spec put(tuple()) -> ok | {error,any()}.
-spec delete(atom(), any()) -> ok | {error,any()}.
```

13.11 Read Ops

Allows you to read the Value by Key and list records with given
secondary indexes. **get/3** API is used to specify default value.

```
-spec index(atom(), any(), any()) -> list(tuple()).
-spec get(atom(),any(), any()) -> {ok,any()}.
-spec get(atom(), any()) -> {ok,any()} |
```

```
                              {error,duplicated} |
                              {error,not_found}.
```

13.12   Import/Export

You can use this API to store all database in a single file when it is
possible. It's ok for development but not very good for production
API.

```
-spec load_db(string()) -> list(ok | {error,any()}).
-spec save_db(string()) -> ok | {error,any()}.
```

14 Afterword

Hope you find N2O[14], KVS[15], and MAD[16] stack small and concise, because it was the main goal during development. We stay with minimal viable functionality criteria.

N2O is free from unnecessary layers and code calls as much as possible. At the same time it covers all your needs to build flexible web messaging relays using rich stack of protocols.

Minimalistic criteria allows you to see the system's most general representation, which drives you to describe efficiently. You could be more productive by focusing on core. Erlang N2O and companion libraries altogether make your life managing web applications easy without efforts due to its naturally compact and simple design, and absence of code bloat.

You can see that parse_transform is very useful, especially in JavaScript protocol generation (SHEN) and REST record-to-proplist generators. So having quote/unquote in language would be very useful. Fast and small Erlang Lisp (LOL) is expecting compiler is this field as universal Lisp-based macro system.

All apps in stack operate on its own DSL records-based language: N2O — #action/#element; KVS — #iterator/#container. This language is accessible directly from Erlang-based languages: Joxa, Elixir, Erlang, Lol.

We hope that this book will guide you in the wild world of Erlang web development and you will be enlightened by its minimalistic power.

---

[14]https://synrc.com/apps/n2o
[15]https://synrc.com/apps/kvs
[16]https://synrc.com/apps/mad