N2O

No Bullshit Sane Framework for Wild Web

N2O: No Bullshit
Sane Framework
For Wild Web

SECOND EDITION

Book Design and Illustrations by Maxim Sokhatsky
Author Maxim Sokhatsky

Editors: Anton Logvinenko
Vladimir Kirillov
Viktor Sovietov
Dmitriy Sukhomlynov

Publisher imprint:
Toliman LLC
251 Harvard st. suite 11, Brookline, MA 02446
1.617.274.0635

Printed in Ukraine

Order a copy with worldwide delivery:
https://balovstvo.me/n2o

ISBN — 978-1-62540-038-3

Contents

1 N2O: Web Framework **9**
- 1.1 Wide Coverage . . . 9
- 1.2 Rich and Lightweight Applications . . . 11
- 1.3 JSON and BERT . . . 12
- 1.4 DSL and Templates . . . 12

2 Setup **15**
- 2.1 Prerequisites . . . 15
- 2.2 Kickstart Bootstrap . . . 15
- 2.3 Application Template . . . 16
- 2.4 Companion Dependencies . . . 17
- 2.5 Configuration . . . 18

3 Erlang Processes **21**
- 3.1 Reduced Latency . . . 21
- 3.2 Page Serving Process . . . 22
- 3.3 Transition Process . . . 22
- 3.4 Events Process . . . 23
- 3.5 Async Processes . . . 23
- 3.6 SPA Mode . . . 24

4 Endpoints **25**
- 4.1 HTML Pages over HTTP . . . 25
- 4.2 JavaScript Events over WebSocket . . . 26
- 4.3 HTTP API over REST . . . 27

5 Handlers **28**
- 5.1 Query . . . 28
- 5.2 Session . . . 28
- 5.3 Router . . . 29

6 Protocols **30**
- 6.1 HEART . . . 35
- 6.2 NITRO . . . 37
- 6.3 SPA . . . 39
- 6.4 BIN . . . 40

7 JavaScript Compiler **41**
7.1 Compilation and Macros 41
7.2 Erlang Macro Functions 41
7.3 JavaScript File Compilation 42
7.4 Mapping Erlang/OTP to JavaScript/OTP 43

8 API **44**
8.1 Update DOM **wf:update** 44
8.2 Wire JavaScript **wf:wire** 46
8.3 Message Bus **wf:reg** and **wf:send** 48
8.4 Async Processes **wf:async** and **wf:flush** 49
8.5 Parse URL and Context parameters **wf:q** and **wf:qp** . . 51
8.6 Render **wf:render** or **nitro:render** 52
8.7 Redirects **wf:redirect** 54
8.8 Session Information **wf:session** 54
8.9 Bridge information **wf:header** and **wf:cookie** 55

9 Elements **56**
9.1 Static Elements: HTML 56
9.2 Active Elements: HTML and JavaScript 57
9.3 Base Element 58
9.4 DTL Template **#dtl** 59
9.5 Button **#button** 60
9.6 Link **#dropdown** 60
9.7 Link **#link** 62
9.8 Text Editor **#textarea** 62

10 Actions **63**
10.1 JavaScript DSL **#jq** 63
10.2 Page Events **#event** 64
10.3 API Events **#api** 64
10.4 Message Box **#alert** 65
10.5 Confirmation Box **#confirm** 65

11 UTF-8 **66**
11.1 Erlang 66
11.2 JavaScript 66

12 MAD: Build and Packaging Tool **67**
12.1 History 67
12.2 Introduction 68
12.3 Single-File Bundling 68
12.4 Templates 69
12.5 Deploy 69
12.6 OTP Compliant 70
12.7 Apps Ordering 70

13 KVS: Abstract Erlang Database **71**
13.1 Polymorphic Records 71
13.2 Iterators 72
13.3 Containers 73
13.4 Extending Schema 73
13.5 KVS API 74
13.6 Service 74
13.7 Schema Change 74
13.8 Meta Info 74
13.9 Chain Ops 75
13.10 Raw Ops 75
13.11 Read Ops 75
13.12 Import/Export 76

14 Afterword **77**

To Mary and all sentient beings.

1 N2O: Web Framework

N2O was started as the first Erlang Web Framework that uses WebSocket 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.

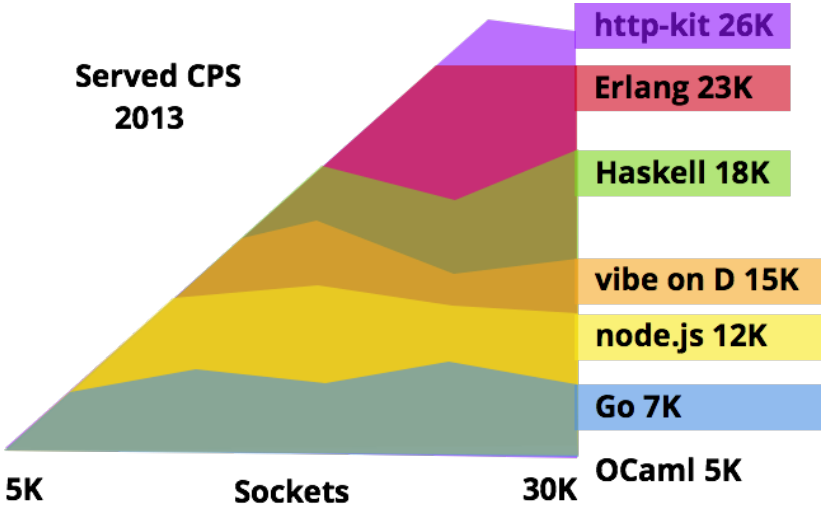

Figure 1: Web-Servers raw performance grand congregation

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. N2O also support this approach by proving bindings to DTL and ET template engines.

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

```
-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. Moreover this application is ready to run under multiplatform LING Erlang virtual machine.

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 version 18 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 Application Template

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: sample.erl

```
-module(sample).
-behaviour(supervisor).
-behaviour(application).
-export([init/1, start/2, stop/1, main/1]).

main(A)     -> mad:main(A).
start(_,_) -> supervisor:start_link({local,review},review,[]).
stop(_)     -> ok.
init([])    -> case cowboy:start_http(http,3,port(),env()) of
                    {ok, _}    -> ok;
                    {error,_} -> halt(abort,[]) end, sup().

sup()     -> { ok, { { one_for_one, 5, 100 }, [] } }.
port()    -> [ { port, wf:config(n2o,port,8000)  } ].
env()     -> [ { env, [ { dispatch, points() } ] } ].
static() ->   { dir, "apps/sample/priv/static", mime() }.
n2o()     ->   { dir, "deps/n2o/priv",           mime() }.
mime()    -> [ { mimetypes, cow_mimetypes, all   } ].
points() -> cowboy_router:compile([{'_', [

             { "/static/[...]",       n2o_static,  static()},
             { "/n2o/[...]",          n2o_static,  n2o()},
             { "/ws/[...]",           n2o_stream,  []},
             { '_',                   n2o_cowboy,  []} ]}]).
```

While Listing 2 is a little bit criptic we want to say that N2O intentionally not introduced here any syntax sugar. For any Erlang application you need to create application and supervisor behavior modules which we combined in the same Erlang file for simplicity.

Cowboy routing rules also leaved as is. We'd better to leave our efforts for making N2O protocol and N2O internals simplier. Here we can't fix a much. Just use this as template for bootstrapping N2O based applications.

2.4 Companion Dependencies

For raw N2O use with BERT message formatter you need only one N2O dependecy, but if you want to use DTL templates, JSON message formatter, SHEN JavaScript Compiler or NITRO Nitrogen DSL you can plug all of them separately.

```
https://github.com/synrc/n2o         2.9
https://github.com/synrc/nitro       2.9
https://github.com/synrc/extra       2.9
https://github.com/synrc/kvs         2.9
https://github.com/synrc/active      2.9
https://github.com/synrc/shen        1.5
https://github.com/synrc/rest        1.5
https://github.com/spawnproc/bpe     1.5
https://github.com/spawnproc/forms   1.5
```

2.5 Configuration

Listing 3: sys.config

```
[{n2o, [{port,8000},
        {app,review},
        {route,routes},
        {mq,n2o_mq},
        {formatter,bert},
        {log_modules,config},
        {log_level,config},
        {log_backend,n2o_log},
        {session,n2o_session},
        {origin,<<"*">>},
        {bridge,n2o_cowboy},
        {pickler,n2o_pickle},
        {erroring,n2o_error}]},
 {kvs, [{dba,store_mnesia},
        {schema, [kvs_user,
                  kvs_acl,
                  kvs_feed,
                  kvs_subscription ]} ]} ].
```

Ports

N2O uses two ports for WebSocket and HTTP connections.

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

If you use server pages mode N2O will render HTML with nessesary ports values. For single page application mode you should redefine these ports inside the template:

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

Application

In **app** setting you should place the name of your OTP application that will be treated by N2O and NITRO as a source for templates and other static data with **code:priv_dir**.

Routes

Setting **route** is a place for the name of Erlang module where resides mappings from URL to page modules.

Logging

N2O supports logging API and you can plug different logging module. It ships with **n2o_io** and **n2o_log** modules which you can set in the **log_backend** option. This is how logging looks like in N2O:

```
wf:info(index,"Message: ~p",[Message]),
```

First argument is a module from which function is being called. By using this N2O can filter all log messages with special filter settled with **log_modules** option. It says in which Erlang module function **log_modules/0** exists that returns allowed Erlang modules to log. Option **log_level** which specified in a similar way, it specifies the module with function **log_level/0** that could return one of **none**, **error**, **warning** or **info** atom values which means different error log levels.

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

log_level() -> info.
log_modules() -> [ login, index ].
```

Message Queue

In **mq** settings you should place the name of Erlang module which supports message queue API. By default N2O provides **n2o_mq** module.

Formatter

With **formatter** option you may set the WebSocket channel termination formatter such as **bert** or **json**.

Minimal Page

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

Listing 4: index.erl

```
-module(index).
-compile(export_all).
-include_lib("nitro/include/nitro.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 WebSocket initialization is complete. It takes three steps and three Erlang processes for doing that.

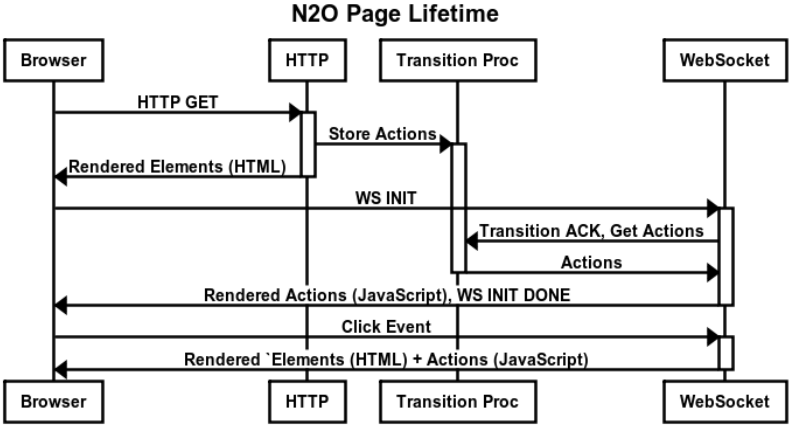

Figure 2: Page Lifetime

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. During page initialization function Module:**main/0** is called. This function should return raw HTML or NITRO elements that could be rendered into raw HTML. All created on the way JavaScript actions are stored in the transition process.

```
main() -> #dtl { file     = "login",
                 app      = review,
                 bindings = [ { body,
                      #button { id       = send,
                                postback = chat } } ] }.
```

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.

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

Transition process is only applicable to dynamically rendered pages served by **n2o_document** endpoint. You never deal with it manually.

3.4 Events Process

After that all client/server communication is performed over WebSocket 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.

```
event(init) -> wf:reg(room);
event(chat) -> #insert_top(history,#span{body="message"}),
               wf:flush(room).
```

This code will register all WebSocket processes under the same topic in pubsub and broadcast history changing to all registered process in the system under the same topic using **#flush** NITRO protocol message.

During page initialization before Module:**event(init)**, Module:**main/0** is called to render initial postbacks for elements. So you can share the same code to use SPA or DSL/DTL approach.

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.

```
event(init) -> wf:reg(room),
               wf:async("looper", fun async/1);

async(init) -> ok;
aynsc(Chat) -> io:format("Chat: ~p~n",[Chat]).
```

3.6 SPA Mode

In SPA mode N2O can serve no HTML at all. N2O elements are bound during initialization 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 in production.

```
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/[...]",           n2o_stream,  []},
{'_',                   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_stream** is a default JavaScript event handler based on Bullet library created by Loïc Hoguin, optimized and refined.

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 5: 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) -> #cx{
    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).
```

Listing 6: wf:fold/3

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,n2o_route}]}
```

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

```
-module(n2o_route).
-include_lib("n2o/include/wf.hrl").
-export(?ROUTING_API).

finish(S, Cx) -> {ok, S, Cx}.
init(S, Cx)   -> P = wf:path(Cx#context.req),
                 M = prefix(Path),
                 {ok, S, Cx#cx{path=P,module=M}}.

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

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

6 Protocols

N2O is more that just web framework or even application server. It also has protocol specification that covers broad range of application domains. In this chapter we go deep inside network capabilities of N2O communications. N2O protocol also has an ASN.1 formal description, however here we will speak on it freely. Here is the landscape of N2O protocols stack.

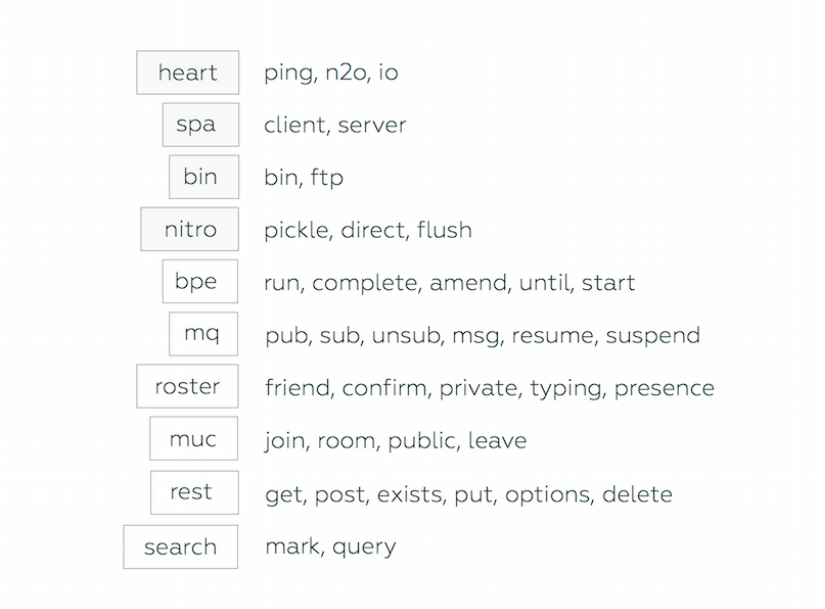

Figure 3: Protocols Stack

You may find it similar to XML-based XMPP, binary COM/CORBA, JSON-based WAMP, Apache Camel or Microsoft WCF communication foundations. We took best from all and put into one protocols stack for web, social and enterprise domains providing stable and mature implementation for Erlang in a form of N2O application server.

Cross Language Compatibility

N2O application server implemented to support N2O protocol definition in Erlang which is widely used in enterprise applications. Experimental implementation in Haskell **n2o.hs** exists which supports only core **heart** protocol along with **bert** formatter. We will show you how N2O clients are compatible across different server implementations in different languages.

Web Protocols: nitro, spa, bin

N2O protocols stack provides definition for several unoverlapped protocol layers. N2O application server implementaion of N2O protocol specification supports four protocol layers from this stack for WebSocket and IoT applications: **heart**, **nitro**, **spa** and **bin** protocols. HEART protocol is designed for reliable managed connections and stream channel initialization. The domain of NITRO protocol is HTML5 client/server interoperability, HTML events and JavaScript delivery. SPA protocol dedicated for games and static page applications that involves no HTML, such as SVG based games or non-gui IoT applications. And finally binary file transfer protocol for images and gigabyte file uploads and downloads. All these protocols transfers coexist in the same multi-channel stream.

Social Protocols: roster, muc, search

For social connectivity one may need to use **synrc/roster** instant messaging server that supports **roster** protocol with variation for enabling public rooms **muc** or full-text **search** facilities.

Enterprise Protocols: bpe, mq, rest

There is no single system shipped to support all of N2O protocols but it could exist theoretically. For other protocols implementation you may refer to other products like **spawnproc/bpe**, **synrc/rest** or **synrc/mq**.

Channel Termination Formatters

N2O protocol is formatter agnostic and it doesn't strict you to use a particular encoder/decoder. Application developers could choose their own formatter per protocol.

```
1. BERT : {io,"fire();",1}
2. WAMP : [io,"fire();",1]
3. JSON : {name:io,eval:"fire();",data:1}
4. TEXT : IO \xFF fire(); \xFF 1\n
5. XML  : <io><eval>fire();</eval><data>1</data></io>
```

E.g. N2O uses TEXT formatting for "PING" and "N2O," protocol messages, across versions N2O used to have IO message formatted with JSON and BERT both. All other protocol messages were BERT from origin. Make sure formatters set for client and server is compatible.

```
> application:set_env(n2o,formatter,bert).
```

Note that you may include to support more that one protocol on the client. At server side you can change formatter on the fly without breaking the channel stream. Each message during data stream could be formatted using only one protocol at a time. If you want to pass each message through more that one formatter you should write an echo protocol.

```
<script src='/n2o/protocols/bert.js'></script>
<script src='/n2o/protocols/client.js'></script>
<script>protos = [ $bert, $client ]; N2O_start();</script>
```

Protocol Loop

After message arrives to endpoint and handlers chain is being initializes, message then comes to protocol stack. N2O selects appropropriative protocol module and handle the message. After than message is being formatted and replied back to stream channel. Note that protocol loop is applicable only to WebSocket stream channel endpoint.

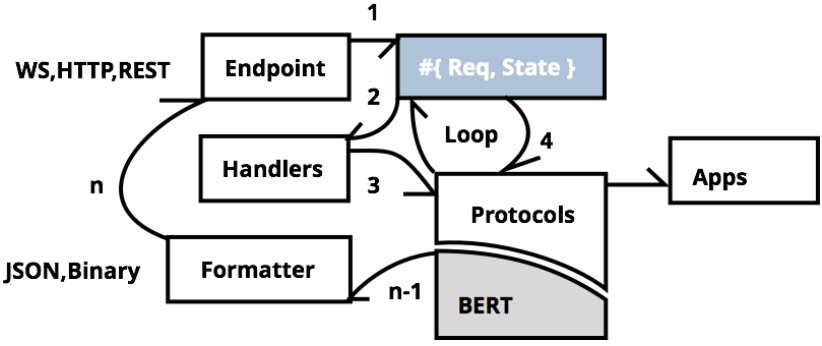

Figure 4: Messaging Pipeline

Here is pseudocode how message travels for each protocol until some of them handle the message. Note tnat this logic is subject to change.

```
reply(M,R,S)                -> {reply,M,R,S}.
nop(R,S)                    -> {reply,<<>>,R,S}.
push(_,R,S,[],_Acc)         -> nop(R,S);
push(M,R,S,[H|T],Acc)       ->
    case H:info(M,R,S) of
          {unknown,_,_,_} -> push(M,R,S,T,Acc);
         {reply,M1,R1,S1} -> reply(M1,R1,S1);
                       A -> push(M,R,S,T,[A|Acc]) end.
```

Listing 7: Top-level protocol loop in n2o_proto

Enabling Protocols

You may set up protocol from sys.config file, enabling or disabling some of them on the fly.

```
protocols() ->
   wf:config(n2o,protocols,[ n2o_heart,
                             n2o_nitrogen,
                             n2o_client,
                             n2o_file  ]).
```

For example in Skyline (DSL) application you use only **nitro** protocol:

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

And in Games (SPA) application you need only **spa** protocol:

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

6.1 HEART

HEART 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.

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

The **heart** protocol defined client originated messages N2O, PING and server originated messages PONG, IO and NOP. IO message contains EVAL that contains UTF-8 JavaScript string and DATA reply contains any binary string, including BERT encoded data. ”PING” and ”N2O,” are defined as text 4-bytes messages and second could be followed by any text string. NOP is 0-byte acknowledging packet. This is heart essence protocol which is enough for any rpc and code transfering interface. Normally heart protocol is not for active client usage but for supporting active connection with notifications and possibly DOM updates.

Session Initialization

After page load you should start N2O session in JavaScript with configured formatters and starting function that will start message loop on the client:

```
var transition = {pid: '', host: 'localhost', port:'8000'};
protos = [ $bert, $client ];
N2O_start();
```

Id **pid** field is not set in **transition** variable then you will request new session otherwise you may put here information from previously settled cookies for attaching to existing session. This **pid** disregarding set or empty will be bypassed as a parameter to N2O init marker.

You can manually invoke session initialization inside existing session:

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

In response on successfull WebSocket connection and enabled **heart** protocol on the server you will recieve the IO message event. IO events are containers for function and data which can be used as parameters. There is no predefined semantic to IO message. Second element of a tuple will be directly evaluated in WebBrowser. IO events are normally not constructed on client. Heartbeat protocol PING request returns PONG or empty message NOP binary response; N2O messages returns IO messages with .

```
 issue TEXT N2O expect IO
N2O is TEXT "N2O," ++ PID
PID is TEXT "" or any
 IO is BERT {io,_,_} or
       JSON {name:'io',eval,data}
```

Online Presence

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

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

```
 issue TEXT PING expect PONG
PONG is TEXT "PONG" or ""
```

6.2 NITRO

NITRO protocol consist of three protocol messages: **pickle**, **flush** and **direct**. 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.
Here is definition to NITRO protocol in expect language:

```
    issue BERT PICKLE expect IO
    issue BERT DIRECT expect IO
    issue BERT FLUSH  expect IO
PICKLE is BERT {pickle,_,_,_,_}
DIRECT is BERT {direct,_}
 FLUSH is BERT {flush,_}
```

Usually **pickle** events are being sent generated from server during rendering of **nitro** elements. To see how it looks like you can see inside IO messages returned from N2O initialization. There you can find something like this:

```
ws.send(enc(tuple(atom('pickle'),
    bin('loginButton'),
    bin('g2gCaAVkAAJldmQABGF1dGhkAAVsb2dpbmsAC2xvZ2lu'
        'QnV0dG9uZAAFZXZlbnRoA2IAAAWiYgAA72ViAA8kIQ=='),
    [ tuple(tuple(utf8_toByteArray('loginButton'),
                  bin('detail')),[]),
      tuple(atom('user'),querySource('user')),
      tuple(atom('pass'),querySource('pass'))])));
```

Invocation of **pickle** messages is binded to DOM elements using **source** and **postback** information from nitro elements.

```
#button { id=loginButton,
          body="Login",
          postback=login,
          source=[user,pass] } ].
```

Only fields listed in **source** will be included in **pickle** message on invocation. Information about module and event arguments (postback) is sent encrypted or pickled. So it would be hard to know the internal structure of server codebase for potential hacker. On the server you will recieve following structure:

```
{pickle,<<"loginButton">>,
        <<"g2gCaAVkAAJldmQABGF1dGhkAAVsb2dpbmsAC2xvZ2lu"
          "QnV0dG9uZAAFZXZlbnRoA2IAAAWiYgAA72ViAA8kIQ==">>,
       [{{"loginButton",<<"detail">>},[]},
        {user,[]},
        {pass,"z"}]}
```

You can depickle **#ev** event with **wf:depickle** API:

```
> wf:depickle(<<"g1AAAAA6eJzLYMpgTWFgSi1LYWDNyU/PzIPR2Qh+"
                "allqXkkGcxIDA+siIHEvKomB5cBKAN+JEQ4=">>).

#ev { module  = Module  = auth,
      msg     = Message = login,
      name    = event,
      trigger = "loginButton" }
```

Information for **#ev** event is directly passed to page module as **Module:event(Message)** . Information from sources **user** and **pass** could be retrieved with **wf:q** API:

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

event(login) ->
    io:format(lists:concat([":user:",wf:q(user),
                            ":pass:",wf:q(pass)])).
```

This is Nitrogen-based messaging model. Nitrogen WebSocket processes receive also flush and delivery protocol messages, but originated from server, which is internal NITRO protocol messages. All client requests originate IO message as a response.

6.3 SPA

If you are creating SVG based game you don't need HTML5 nitro elements at all. Instead you need simple and clean JavaScript based protocol for updating DOM SVG elements but based on **shen** generated or manual JavaScript code sent from server. Thus you need still IO messages as a reply but originating massesage shouldn't rely in **nitro** at all. For that purposes ingeneral and for **synrc/games** sample in particular we created SPA protocol layer. SPA protocol consist of CLIENT originated message and SERVER message that could be originated both from client and server. All messages expects IO as a response.

```
    issue BERT CLIENT expect IO
    issue BERT SERVER expect IO
SERVER is BERT {server,_}
CLIENT is BERT {client,_}
```

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 during page init:

```
event(init) -> io:format("Pid: ~p",[self()]);
```

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

6.4 BIN

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'),bin('request'))));
```

Ensure you have defined **#bin** handler and page you are asking is visible by router:

```
event(#bin{data=Data}) ->
    wf:info(?MODULE,"Binary Delivered ~p~n",[Data]),
    #bin{data = "SERVER v1"};
```

Having enabled all loggin in module **n2o_file**, **index** and **wf_convert** you will see:

```
n2o_file:BIN Message: {bin,<<"request">>}
index:Binary Delivered <<"request">>
wf_convert:BERT {bin,_}: "SERVER v1"
```

In JavaScript when you enable 'debug=true' you can see:

```
> {"t":104,"v":[{"t":100,"v":"bin"},
                {"t":107,"v":"SERVER v1"}]}
```

Or by adding handling for BIN protocol:

```
> $file.do = function (x)
    { console.log('BIN received: ' + x.v[1].v); }
> ws.send(enc(tuple(atom('bin'),bin('request'))));
> BIN received: SERVER v1
```

The formal description of BIN is simple relay:

issue BERT {bin,_} expect {bin,_}

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.

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,type=click}),
  []

> rp(iolist_to_binary(wf:render(wf:actions()))).
  <<"{var x=qi('element_id'); x && x.addEventListener('cl
  ick',function (event){{ if (validateSources([])) ws.sen
  d(enc(tuple(atom('pickle'),bin('element_id'),bin('g2gCa
  AVkAAJldmQABWluZGV4ZAADZXZ0awAKZWxlbWVudF9pZGQABWV2ZW50
  aANiAAAFoWIAB8kuYgAOvJA='),[tuple(tuple(utf8_toByteArra
  y('element_id'),bin('detail')),event.detail)]));else c
  onsole.log('Validation Error'); }});};">>
```

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/2** creates Erlang process, which communicate with the primary page process by sending messages. **wf:flush/0** should be called to redirect all updates and wire actions back to the page process from its async counterpart. But function **wf:flush/1** has completly another meaning, it uses pubsub to deliver a rendered actions in async worker to any process, previously registered with **wf:reg/1**, by its topic. Usually you send messages to async processes over N2O message bus **wf:send/2** which is similar to how **wf:flush/1** works. But you can use also **n2o_async:send/2** selectively to async worker what reminds **wf:flush/0**. In following example different variants are gives, both incrementing counter by 2. Also notice the async process initialization throung **init** message. It is not nessesary to include init clause to async looper.

```
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.

So let's mimic **session_id** and **#cx** in the shell:

```
> put(session_id,<<"d43adcc79dd64393a1eb559227a2d3fd">>).
  undefined

> wf:context(wf:init_context(undefined)).
  {cx,[{query,n2o_query},
       {session,n2o_session},
       {route,routes}],
       [],[],index,undefined,[],
       undefined,[],undefined,[]}

> wf:async("ho!",
     fun(X) -> io:format("Received: ~p~n",[X]) end).
  index:Received: init
  {<0.507.0>,{async,
    {"ho!",<<"d43adcc79dd64393a1eb559227a2d3fd">>}}}

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

Async workers suppors both sync and async messages, you may use **gen_server** for calling by pid, **n2o_async** for named or even built-in erlang way of sending messages. All types of handlilng like info, cast and call are supported.

```
> pid(0,507,0) ! "hey".
  Received: "hey"
  ok

> n2o_async:send("ho!","hola").
  Received: "hola"
  ok

> gen_server:call(pid(0,507,0),"sync").
  Received: "sync"
  ok
```

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:init_context(Req),
NewCtx = wf: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: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'); }});};">>
```

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)]).
```

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 8: 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}).
```

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[3] 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[4] emerged. Under the hood active is a client subscriber of fs[5] 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[6].

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[7], 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 9: Example of building N2O sample

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

```
                                Hot
make (erlang.mk)                2.588s
mad compile                     2.521s
```

Listing 10: Example of building Cowboy

12.2 Introduction

We were trying to make something minimalistic that fits out Web Stack[8]. 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

[8] 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[9].

```
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.

[9] 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
```

Listing 11: Solution

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

Listing 12: OTP Application

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[10].

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[11].

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).
```

[10] https://github.com/synrc/feeds
[11] 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[12], KVS[13], and MAD[14] 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.

[12] `https://synrc.com/apps/n2o`
[13] `https://synrc.com/apps/kvs`
[14] `https://synrc.com/apps/mad`