Build web application with Nova

Contents

Nova

How to read this

Part 1

Here we will discuss foundation of Nova, how to use it as a view-controller. This section will be larger abd be the foundation of the rest of the book. I will describe not only Nova but also Erlang.

Part 2

In this section we will show how to use Nova as a api server withour views. With routing and plugins.

Part 3

In this section we will show how to use js frameworks and static pages with Nova.

Pre req

You will need erlang 22+ installed and rebar3.

Installing Nova

Add rebar3_nova to ~/.config/rebar3/rebar.config

```
{project_plugins, [rebar3_nova]}
```

Background

Erlang web frameworks

Both me and Niclas have used many different web frameworks in Erlang. We started to code Erlang around 2009 and at that time we had frameworks like Nitrogen, Erlang Web and Zotonic. Some years later Chicago Boss was introduced that tried to solve many things. But one of the things with Chicago Boss was that it didn't really follow OTP standard when it came to release handling.

In 2011 Loïc Hoguin released Cowboy, an Erlang web server. It was very fast and stable web server that got popular in the Erlang community.

Many Erlang companies started to use Cowboy and everytime me or Niclas came in contact with it we needed to start writing all the handlers, or work with a wrapper around.

I started my Erlang coding with a project course at Uppsala University where we did a e-commerce system written with Nitrogen and Riak. This was late 2009, riak was a new and untested database also was Nitrogen something new.

From this both me and Niclas have helped companies to launch web applications with Erlang or as employees develop these kind of systems.

The beginning

In a hotel room in Gratz we started to look at some of our projects we have started during all the years. Niclas had earlier started a framework called Burbweb, this would later be the start of Nova. Some years before that we had worked on a ORM library in Erlang. We got an idea that we should try to make a product of this and really try to aim for a new good framework that people can use.

We started to look into how we can help users to get started easy and also try to make easy release builds.

So in 2019 we released Nova that we now have worked on and try to make it better.

Zaark was a company that both me and Niclas worked at and later started to use Nova in production. Mostly as a api server without views. From the feedback using Nova we improved it and felt we are doing something great.

2020 we had our first presentation about Nova at code beam.

Architecture & Design

Architecture

Nova is built on top of Cowboy that is a web server written in Erlang. One of the reason was that it is a well tested web server, that have great features.

In the beginning we used Cowboy routing but later we did see that we wanted to handle appliction modilarity. That is for us that you can include other Nova applications into your application.

Design

One of the things that we wanted with Nova is that it should make it easier to start a new web application projet in Erlang. Without spending much time on boilerplate Cowboy for your project, or write your own wrapper around it.

The other thing is that what is core features in Nova should be small and we should not use that many dependencies. When we discuss new things we usually ask if this is something that extends the core Nova or if it is something that builds on top of Nova.

This is also one of thea reason for plugin system, that helps us to leave to other to write how they want to handle requests. Instead of building everything in Nova.

One of the design philosopies we have is that is should be using as much as we can standard OTP utilites and release tools. In this case we use rebar3 that uses relx, we see that more and more it is getting to be the standard. (While writing this OTP will make rebar3 as a part of OTP.)

Part 1 - View-Controller

In this section I will show how you can use Nova as a view-controller framework. We will use Nova views that uses erlydtl (Djangot templating).

In the end of this section we will have a web application where you can signup, login and list users and edit a user.

Getting Started

First we will generate a new Nova app that we will work with.

You will see that the template setup the directory and configurations to get first app running.

This will start user_management application with a shell. Now open a browser and go to `http://localhost:8080` you should see a page with the text `Nova is running!`

Erlang

```
rebar3 new nova user_management
```

```
===> Writing user_management/config/dev_sys.config
===> Writing user_management/config/prod_sys.config
===> Writing user_management/priv/user_management.routes.erl
===> Writing user_management/src/user_management.app.src
===> Writing user_management/src/user_management_app.erl
===> Writing user_management/src/user_management_sup.erl
===> Writing
 ↪  user_management/src/controllers/user_management_main_controller.erl
===> Writing user_management/rebar.config
===> Writing user_management/config/vm.args
===> Writing user_management/src/views/user_management_main.dtl
```

When this is installed you can run:

```
rebar3 shell
```

LFE

```
rebar3 new nova_lfe user_management
```

```
===> Writing user_management/config/sys.config
===> Writing user_management/priv/user_management.routes.lfe
===> Writing user_management/src/user_management.app.src
===> Writing user_management/src/app.lfe
===> Writing user_management/src/sup.lfe
===> Writing
 ↪  user_management/src/controllers/user_management_main_controller.lfe
===> Writing user_management/rebar.config
===> Writing user_management/config/vm.args
===> Writing user_management/src/views/user_management_main.dtl
```

When this is installed you can run:

```
rebar3 compile
rebar3 lfe repl
```

What did we get?

Nova app have some configuration files, route file, controllers and views.

First the sys.config, this handle the user_management application.

```
%% -*- mode: erlang;erlang-indent-level: 4;indent-tabs-mode: nil
 ↪  -*-

[
 {kernel, [
           {logger_level, debug}
          ]},
 {nova, [
         {cowboy_configuration, #{
                                  port => 8080
                                 }},
         {dev_mode, true},
         {bootstrap_application, my_first_nova}, %% Bootstraps
          ↪  the application
         {plugins, [
                    ]}
```

```
        ]}
  %% Please change your app.src-file instead if you intend to add
   ↪  app-specific configurations
].
```

What do we have here? Well kernel section is for logging with logger, here we can later add different formatters or log handlers, this is basic Erlang configurations.

Nova section handles the nova application and when we start Nova in our app my_first_nova it knows what port and plugins it should use. Also what is the main app, the bootstrap_application section.

Plugins is a new feature that we are working with, but it is a way to have modules do som pre/post handling on your request.

Example: > You want to get a user,for each request you want to have a correlation id. This can then be generated in a pre plugin that adds it to the header and pass it on. Then it can be used in the controller to keep track of what goes on.

Then we have the route file my_first_nova.routes.erl in this you will specify all endpoints or static assets that you want to expose.

Erlang

```
#{prefix => "",
  security => false,
  routes => [
             {"/",{ my_first_nova_main_controller, index},
              ↪  #{methods => [get]}}
            ],
  statics => [
              {"/assets/[...]", "assets"}
             ]
}.
```

LFE

```
#m(prefix ""
   security false
   routes (#("/"
             #(user_management_main_controller index)
             #M(methods (get)))))
```

Prefix is if you want to have something before the routes. Say "v1" then you can add it to prefix.

Security is if you want any way to auth the request. Change security to {Module, Function} instead of false if you want to have a auth module.

Routes is a tuple {PATH, { MODULE, FUNCTION }, OPTIONS}. If we break down the file above it will go to `http://localhost:8080/` and it will only allow method GET. When someone is doing a GET agains this it will use module my_first_nova_main_controller with function index.

This is the configurations that is for Nova.

Controllers & Views

Controllers and Views are the VC part of the framework. Nova haven't got the model part yet for data but it have been discussed.

In src we have a views directory that will contain all your views that will also have a controller with the same name in the directory controllers.

When we configure routing we will poing the module and function to the controller that will do some logic and then return {ok, Proplist}. When controller return ok tuple Nova will understand that it should render the view file and return it.

Controllers

This is a basic controller that will return back a proplist back to the view. In this case it will return with a message that the view will populate.

What do we have in this module?

Erlang

```erlang
%% This is the module declaration
-module(user_management_main_controller).
%% What functions that are exported from this module.
-export([
         index/1
        ]).
%% Function header.
index(_) ->
    %% Returns a tuple with ok and a proplists (Key-Value list)
    {ok, [{message, "Nova is running!"}]}.
```

LFE

```
(defmodule user_management_main_controller
  (export
   (index 1)))

(include-lib "logjam/include/logjam.hrl")

(defun index
  ;;
  ;; GET Handler
  ;;
  ((`#m(req #m(method #"GET")))
    `#(ok (#(message "nova is running!")))))
```

An Erlang module is structured first to have the module declaration to say that it is a module this is done with the first line `-module(Module)`. Comments are done with `%`, good rule is to use %% when comment on a line and % if you comment after code.

```
%% This is a comment
my_function() ->
    ok. % Comment after code
```

After this we have the export that will export functions outside the module so other can modules can use it or you can use it from a shell.

Then we have the index functions that in this take one argument.

If you change the message it will also be changed on the site.

Views

Each view have a controller for logic, the view is a Django template file.

Our view my_first_nova_main_view.dtl:

```
<html>
<body>
<h1>{{message}}</h1>
</body>
</html>
```

From the controller we will get `Nova is running!`. That we will template into `{{message}}` that will show it on the homepage. What will happen here is that `{{ message }}` will be changed to the text that we get from our index function

in section about the controller above. If we change the text it will also be added to the page.

Adding auth and different views

In previous section we created a Nova application.

Now we will add a small login form and try to auth and if we pass it will show a view with message "Welcome Daniel!".

The structure we have in applications using nova is that in src/ we usually have modules that is used by our application. In the directory src/controllers we will have erlang modules that will be used to handle requests. In the directory src/views we have the .dtl files for our endpoints. The names need to match so MY_VIEW.dtl should match MY_VIEW_controller.erl.

Security is handling in our routing file. It looks like this.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_main_controller, index},
   ↪  #{methods => [get]}}]
 }.
```

Nova has the possibiltity to have different routes depending on what you want to achieve. So for now we want to add a setting for endpoints that will use a security module.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_main_controller, index},
   ↪  #{methods => [get]}}]
 }.
#{prefix => "",
  security => {my_first_nova_auth, auth},
  routes => [
         {"/login", { my_first_nova_login_controller, index},
          ↪  #{methods => [post]}}
          ]
}.
```

When we add the input form it will submit username and password on the endpoint `/login`. If it pass our auth module my_first_nova_auth it will show my_first_nova_login.dtl and also my_first_nova_login_controller that will have the logic.

First we change the view for our main page my_first_nova_main.dtl.

```
<html>
<body>
  <div>
    <form action="/login" method="post" id="nameform">
      <label for="username">userame:</label>
      <input type="text" id="username" name="username"><br>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password"><br>
      <input type="submit" value="submit">
    </form>
  </div>
</body>
</html>
```

We have added an input form now that have username and password. When we click on the submit button it will trigger the auth module.

my_first_nova_auth.erl, we specified in our routing file what the module should be called and the function that will be used.

```
-module(my_first_nova_auth).

-export([auth/1]).

auth(Req) ->
    {ok, Data, Req1} = cowboy_req:read_body(Req),
    {Username, _} = ParsedData = parse_input(Data),
    case ParsedData of
          {<<"daniel">>, <<"test">>} ->
              {true, #{<<"username">> => Username,
                         <<"authed">> => true}};
          _ ->
              {true, #{<<"authed">> => false}}
    end.

parse_input(Data) ->
    [_, Username, _, Password] = bstring:tokens(Data, <<"=&">>),
    {Username, Password}.
```

Input will send data as a string so we will need to parse it to get the data. Nova uses cowboy as a web server and it have many functionality how to get headers or body. In this module we use `cowboy_req:read_body/1`.

Data here will be a binary string `<<"username=USERNAME&password=PASSWORD">>`.

bstring is a module that works in the same way as strings in erlang OTP library. Nova have some libraries inbuilt like jhn_stdlib.

When we return a tuple with {true, map()}, the map will be passed to the controller as a second argument. If we hada rest api or want to send back a 401 to the one that did the request we return false in our auth module. But in this case we wan't to redirect back to `/` if you pass in wrong credentials. If we did enter correct password and username we are going to show `Welcome USERNAME!`.

We need to create the controller now and the view for this, `my_first_nova_login_controller.erl` in `src/controllers/`.

Erlang

```
-module(user_management_login_controller).

-export([index/1]).

index(#{<<"authed">> := true,
          <<"username">> := Username}) ->
    {ok, [{message, <<"Welcome ", Username/binary, "!">>}]};
index(Req, #{<<"authed">> := false}) ->
    {redirect, "/"}.
```

LFE

```

```

In the controller we have a index function that take two arguments. First one is a cowboy req the other is nova state.

And then the view `my_first_nova_login.dtl` should be created in `src/views/`

```
<html>
<body>
<h1> {{ message }} </h1>
</body>
</html>
```

What will happen here is that if we have the correct username and password the auth module will pass on the map we are giving in it. If `authed`is false it will redirect back to `/` if it returned true we should print the Username.

In this case we didn't use any database or so to have users, just to show how things works. If you want to see this with what you enter in the username we can change the auth module to.

```
-module(my_first_nova_auth).

-export([auth/1]).

auth(Req) ->
    {ok, Data, Req1} = cowboy_req:read_body(Req),
    case parse_input(Data) of
    {Username, <<"test">>} ->
        {true, #{<<"username">> => Username,
             <<"authed">> => true}};
    _ ->
        {true, #{<<"authed">> => false}}
    end.

parse_input(Data) ->
    [_, Username, _, Password] = bstring:tokens(Data, <<"=&">>),
    {Username, Password}.
```

In this case Username will be passed on to the controller that will print it on the page.

Routing

Here I will write about routing in Nova and how it works.

When we generate a new nova app we will have a routing file in the directory priv/.

File is named MYAPP.routes.erl in that file we have a erlang map that will specify and configurate what rules is for a endpoint.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_login_controller, login},
   ↪  #{methods => [get]}}]
 }.
```

What does this routing configuration say us? Prefix is what we will match against first, in this way you can create different routing configures depending on prefix.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_login_controller, login},
   ↪  #{methods => [post]}}]
```

```
 }.

#{prefix => "/user/:userid",
  security => {my_first_nova_auth, auth},
  routes => [
         {"/", { my_first_nova_user_controller, get_user},
          ↪  #{methods => [get]}},
         {"/pet", {my_first_nova_user_controller, get_user_pet},
          ↪  #{methods => [get]}}
           ]
}.
```

First map we have a if you go against http://BASEURL:PORT/, we have said we will not have any authentication module on this endpoint. When we reach this endpoint we will call my_first_nova_login_controller:login if method is post.

In the second map we have a prefix that will check that every request with path http://BASEURL:PORT/user/:userid will use this configuration. Here we say that we want to use a auth module, so Nova will use my_first_nova_auth:auth to authenticate the request. If it is ok it will use the module my_first_nova_user_controller:get_user if the request was a get. In this way you can group endpoints also have different auth modules depending on endpoint.

In Nova we can also have explicit routing with that is that an endpoint will go direct to a module and function.

```
#{prefix => "",
  security => false,
  routes => [{"/user/:userid", { my_first_nova_user_controller,
   ↪  get_user}, #{methods => [get]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  update_user}, #{methods => [put]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  delete_user}, #{methods => [delete]}}]
 }.
```

This routing configuration will allow us to decide on method what we want to do at an endpoint. Novas routing is little different from cowboys routing. That is we can have multiple paths that are the same but methods are different.

One of the reasons we choosed to do this way was that we didn't want to match against methods in our controllers. It would be nice if we knew what we wanted to do in the controller depending on endpoint and method.

The routing file in the end will be a map on what paths goes to what module and functions.

Static content

In routing we can also add endpoints for static content.

```
#{prefix => "",
  security => false,
  routes => [{"/user/:userid", { my_first_nova_user_controller,
   ↪  get_user}, #{methods => [get]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  update_user}, #{methods => [put]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  delete_user}, #{methods => [delete]}}],
  statics => [
             {"/www/admin", "assets/admin.html"}
            ]
 }.
```

In this routing configuration we have added an endpoint to `/www/admin` that points to a file called `admin.html`. When someone goes to `/www/admin`Nova will serve `admin.html`.

To serve these files they need to be in priv, but in any directory structure as you want.

Part 2 - API server

Testing

We want to build a pet store that have a api that handles pets in different ways.

Before we start to write Nova api for this it is good to start creating tests so we are sure that things are working when we build the api.

We will use Common test for this and to test with that it is much easier if we use two terminals. One that have the node running and the other that we run our tests in.

In first terminal:

```
rebar3 shell
```

This will start Nova and you will see a console when it is done.

In the second terminal we will run common tests with command:

```
rebar3 ct
```

Test Suite

First we will create a new directory called `test` standing in root directory in you app.

In `./test` you craete a file called `pets_SUITE.erl` this will be our common test file that will keep our tests. What we want to test is to add a pet, get that pet, change name on pet and then remove the pet. Nova have a http client that can be used in tests (also in the code if want the node to do http requests).

This is a basic common test file:

```
-module(pets_SUITE).

-compile(export_all).

%% Includes
-include_lib("common_test/include/ct.hrl").

suite() ->
    [{timetrap,{seconds,30}}].

init_per_suite(Config) ->
    Config.

end_per_suite(Config) ->
    ok.

groups() -> [].

all() ->
    [].
```

What we will use here is init_per_suite and end_per_suite. Common test have more functionality that you can add, like init_per_group, end_per_group, init_per_testcase and end_per_testcase. All of this will initiate something before running the suite, group or testcase.

In init_per_suite I will have initial thing that will add the pet. The ID we get back I will store in the Config so the testcases that are in the Suite can use it. In end_per_suite I will remove the pet so we clean the data, end_per_suite will also be called if tests fails. This will remove test data that we add.

If we think that our app have a api that will add pets with a name with a post. `localhost:8080/pet` it will take a json `{"name":PETNAME}`. In our init_per_suite we will make a http request to our server and see if we get back a json with `{"name":PETNAME, "id":ID}` and status code 201.

```
-define(OPTS, #{close => true,
                headers => #{'Content-Type' =>
                 ↪  <<"application/json">>}}).
-define(BASE_URL, <<"http://localhost:8080/pet">>).

init_per_suite(Config) ->
    Json = json:encode(#{<<"name">> => <<"Hades">>}, [maps,
     ↪  binary]),
    case shttpc:post(?BASE_URL, Json, ?OPTS) of
        #{status := {201, _}, body := Body} ->
            #{<<"id">> := Id} = json:decode(Body, [maps]),
            [{id, Id}, {name, <<"Hades">>} | Config]
    end.
```

First we declare two macros this is values that will be constant during the tests, like http options and the base_url.

To define a macro you type `-define(MACRONAME, MACROVALUE)`, when we later want to user a macro we use `?MACRONAME`.

After we defined the macros we start coding init_per_suite. We encode a Erlang map `#{<<"name">> => <<"Hades">>}` into a json. After this we will make the post to our server.

```
case shttpc:post(?BASE_URL, Json, ?OPTS) of
        #{status := {201, _}, body := Body} ->
            #{<<"id">> := Id} = json:decode(Body, [maps]),
            [{id, Id}, {name, <<"Hades">>} | Config]
end.
```

This code… ## Plugins ##

Plugins in Nova is modules that have a behaviour. These behaviours will be a part of the pipeline flow of a request.

```
-module(nova_correlation_plugin).
-behaviour(nova_plugin).

-export([pre_http_request/2,
         post_http_request/2,
         plugin_info/0]).

pre_http_request(#{req := Req} = NovaState, _) ->
    UUID = uuid:uuid_to_string(uuid:get_v4()),
    Req1 = cowboy_req:set_resp_header(<<"x-correlation-id">>,
     ↪  UUID, Req),
    NewState = maps:put(req, Req1, NovaState),
```

```
    {ok, NewState}.

post_http_request(NovaState, _) ->
    {ok, NovaState}.

plugin_info() ->
    {<<"nova_cors_plugin">>, <<"0.1.0">>, <<"">>, <<"Add CORS
     ↪  headers to request">>, []}.
```

This is an example plugin that will add a correlation id to all your request.

Pipeline

We can look at the flow of a request as a pipeline that will go into different plugins before and after it have done controller code.

Plugins can be used on both http and websockets. The example above show two funcitons that the plugin have, pre_http_request (things that is before the controller) and post_http_request (things that is after the controller).

In the routing module we did show that you can use a security module to authenticate endpoints.

In Nova we have a security plugin that will check if security is set or not.

Nova Security Plugin

Prioritization & Configuration

We want to say in what order we want plugins to be run. This we will do in sys.config where we also can add optionsal settings to plugin.

```
{nova, [
        {cowboy_configuration, #{
                                  port => 8190
                                 }},
        {dev_mode, true},
        {bootstrap_application, MYAPP}, %% Bootstraps the
         ↪  application
        %% Plugins is written on form {RequestType, Module,
         ↪  Options, Priority}
        %% Priority is that the lowest number is executed first
        {plugins, [
                   {pre_http_request, nova_security_plugin, #{},
                    ↪  2},
                   {pre_http_request, nova_cors_plugin, #{}, 0},
                   {pre_http_request, nova_request_plugin,
                    ↪  #{decode_json_body => true,
```

```
                                                                    ↪  parse_bindings
                                                                    ↪  =>
                                                                    ↪  true,

                                                                    ↪  parse_qs
                                                                    ↪  =>
                                                                    ↪  true},
                                                                    ↪  10},
                    {pre_ws_upgrade, nova_security_plugin, #{},
                     ↪  20}
                   ]}
        ]},
```

Here we see that we have configured three plugins for http and one for websocket. For http we have the last number in the tuple to say what order to run things. Lowest number runs first.

In this case: nova_cors_plugin with prio at 0 nova_security_plugin with prio at 2 nova_request_plugin with prio at 10

Websocket only have security plugin so it will only run that.

In the nova_request_plugin we have set some values:

```
{pre_http_request, nova_request_plugin, #{decode_json_body =>
 ↪  true,

                                                                    ↪  parse_bindings
                                                                    ↪  =>
                                                                    ↪  true,

                                                                    ↪  parse_qs
                                                                    ↪  =>
                                                                    ↪  true},
                                                                    ↪  10}
```

What will request plugin do? What it will do is that it will move bindings from cowboy request to nova state. That the state that will be in the function header will look like this:

```
#{req => CowboyReq,
  bindings => Bindings} (Same as in CowboyReq, but you don't need
   ↪  to think of Req)
```

Then we have decode_json_body this will if we get a body in the request and it it content-type: application/json decode the body and move it to Nova state.

```
#{req => CowboyReq,
  bindings => Bindings,
  json => JSONMap} (Keys in JSONMap will be binary)
```

The last part of the request plugin will add QS to Nova state if QS is used.

```
#{req => CowboyReq,
  bindings => Bindings,
  json => JSONMap,
  qs => QS}
```

Because we have this in a plugin this will happen for all request that is used for this Nova application. So we don't need to handle json decode for all of our controllers.

Plugins is a way to remove things that you would need to do in all you requests to a place where it is handled by the system for you.

Writing plugins

It is possible to write your own plugins. This is a small example of how you can do it. Nova has some templates for this. Getting a plugin templated for you use this command:

```
rebar3 new nova_plugin pluginname
```

This will generate a new plugin for you:

```
-module(pluginname).
-behaviour(nova_plugin).

-include_lib("nova/include/nova.hrl").

-export([
         pre_request/2,
         post_request/2,
         plugin_info/0
        ]).

%%--------------------------------------------------------------------
%% @doc
%% Pre-request callback
%% @end
%%--------------------------------------------------------------------
-spec pre_request(State :: nova_http_handler:nova_http_state(),
 ↪  Options :: map()) ->
                         {ok, State0 ::
                          ↪  nova_http_handler:nova_http_state()}
                          ↪  |
```

```
                        {stop, State0 ::
                         ↪  nova_http_handler:nova_http_state()}
                         ↪  |
                        {error, Reason :: term()}.
pre_request(State, _Options) ->
    {ok, State}.


%%--------------------------------------------------------------------
%% @doc
%% Post-request callback
%% @end
%%--------------------------------------------------------------------
-spec post_request(State :: nova_http_handler:nova_http_state(),
 ↪  Options :: map()) ->
                          {ok, State0 ::
                           ↪  nova_http_handler:nova_http_state()}
                           ↪  |
                          {stop, State0 ::
                           ↪  nova_http_handler:nova_http_state()}
                           ↪  |
                          {error, Reason :: term()}.
post_request(State, _Options) ->
    {ok, State}.


%%--------------------------------------------------------------------
%% @doc
%% nova_plugin callback. Returns information about the plugin.
%% @end
%%--------------------------------------------------------------------
-spec plugin_info() -> {Title :: binary(), Version :: binary(),
 ↪  Author :: binary(), Description :: binary(),
                        [{Key :: atom(), OptionDescription ::
                         ↪  atom()}]}.
plugin_info() ->
    {<<"pluginname plugin">>,
     <<"0.0.1">>,
     <<"User <user@email.com">>,
     <<"Descriptive text">>,
     []}. %% Options is specified as {Key, Description}
```

Part 3 - Using JS framework

Routing

Here I will write about routing in Nova and how it works.

When we generate a new nova app we will have a routing file in the directory priv/.

File is named MYAPP.routes.erl in that file we have a erlang map that will specify and configurate what rules is for a endpoint.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_login_controller, login},
   ↪  #{methods => [get]}}]
 }.
```

What does this routing configuration say us? Prefix is what we will match against first, in this way you can create different routing configures depending on prefix.

```
#{prefix => "",
  security => false,
  routes => [{"/", { my_first_nova_login_controller, login},
   ↪  #{methods => [post]}}]
 }.

#{prefix => "/user/:userid",
  security => {my_first_nova_auth, auth},
  routes => [
         {"/", { my_first_nova_user_controller, get_user},
          ↪  #{methods => [get]}},
         {"/pet", {my_first_nova_user_controller, get_user_pet},
          ↪  #{methods => [get]}}
           ]
}.
```

First map we have a if you go against http://BASEURL:PORT/, we have said we will not have any authentication module on this endpoint. When we reach this endpoint we will call my_first_nova_login_controller:login if method is post.

In the second map we have a prefix that will check that every request with path http://BASEURL:PORT/user/:userid will use this configuration. Here we say that we want to use a auth module, so Nova will use my_first_nova_auth:auth to authenticate the request. If it is ok it will use the module my_first_nova_user_controller:get_user if the request was a get. In

this way you can group endpoints also have different auth modules depending on endpoint.

In Nova we can also have explicit routing with that is that an endpoint will go direct to a module and function.

```
#{prefix => "",
  security => false,
  routes => [{"/user/:userid", { my_first_nova_user_controller,
   ↪  get_user}, #{methods => [get]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  update_user}, #{methods => [put]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  delete_user}, #{methods => [delete]}}]
 }.
```

This routing configuration will allow us to decide on method what we want to do at an endpoint. Novas routing is little different from cowboys routing. That is we can have multiple paths that are the same but methods are different.

One of the reasons we choosed to do this way was that we didn't want to match against methods in our controllers. It would be nice if we knew what we wanted to do in the controller depending on endpoint and method.

The routing file in the end will be a map on what paths goes to what module and functions.

Static content

In routing we can also add endpoints for static content.

```
#{prefix => "",
  security => false,
  routes => [{"/user/:userid", { my_first_nova_user_controller,
   ↪  get_user}, #{methods => [get]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  update_user}, #{methods => [put]}},
             {"/user/:userid", { my_first_nova_user_controller,
              ↪  delete_user}, #{methods => [delete]}}],
  statics => [
             {"/www/admin", "assets/admin.html"}
            ]
 }.
```

In this routing configuration we have added an endpoint to `/www/admin` that points to a file called `admin.html`. When someone goes to `/www/admin`Nova will serve `admin.html`.

To serve these files they need to be in priv, but in any directory structure as you want.

Erlang

Erlang books

Learn you some erlang

Adopting Erlang

Erlang in anger

Recursive functions

This is me trying to describe recursive functions in Erlang. How you can write them and how it works.

Erlang uses recursive functions often because in other languages you have for, while and other loops. In Erlang we do it with recursive functions.

Example:

```
-module(recursive).

-export([sum/1]).

sum([]) ->
    0;
sum([Head | Tail]) ->
   Head + sum(Tail).
```

Here we do a sum function that will take a list of integers and summarize them.

```
15> recursive:sum([1,2,3]).
6
```

What happen when we call our function?

Thing in Erlang works that we go from the top to the bottom of our code.

recursive:sum([1,2,3]). When we call this function with argument [1,2,3] this will happen.

```
sum([]) ->
   0;
```

```
sum([1 | [2,3]]) ->
   1 + sum([2,3]).
```

First we will hit the first function clause and because our list contain three elements it will go on to the next function clause. Here the head of the list will be 1 and the tail will be list with two elements [2,3].

The recursive call that we call our function sum again but now with the tail will give us.

```
sum([]) ->
   0;
sum([2 | [3]]) ->
   2 + sum([3]).
```

We also do that with the last element in the list.

```
sum([]) ->
   0;
sum([3 | []]) ->
   3 + sum([]).
```

Last call here will hit the base case and return a 0.

This will calculate $3 + 2 + 1 + 0$ (base case) $= 6$.

You can also do this with tail recursive so you pass the result of the recursive to the next.

```
sum([], Sum) ->
    Sum;
sum([Head | Tail], Sum) ->
    sum(Tail, Sum + Head).
```

This is our function with tail recursive. What will happen here is very similar to what we did above. Different now is that we to the addition in the recursive step and send the result to the next recursive. When we hit base case we will return the result.

```
16> c(recursive).
{ok,recursive}
17> recursive:sum([1,2,3], 0).
6
```

We have now created a function called sum that now have aritry 2. It will take the list with integers and the start value.

```
sum([], Sum) ->
   Sum;
```

```
sum([1 | [2,3]], 0) ->
   sum([2,3], 0 + 1).
```

```
sum([], Sum) ->
   Sum;
sum([2 | [3]], 1) ->
   sum([3], 1 + 2).
```

```
sum([], Sum) ->
   Sum;
sum([3 | []], 3) ->
   sum([], 3 + 3).
```

```
sum([], 6) ->
   6;
sum([Head | Tail], Sum) ->
   sum(Tail, Sum + Head).
```

When we hit the base case and the list is empty we will return the variable Sum that is in this case 6.

Pattern matching

In last section we did some recursive functions. Now we will try to do some pattern matching and later combine it with recursive functions.

When it comes to Erlang we execute from right to left that makes it easier to understand the error message.

```
2> Variable = 1.
1
3> Variable = 1.
1
4> 1 = Variable.
1
5> a = Variable.
** exception error: no match of right hand side value 1
```

This is a Erlang shell that we bind Variable to 1 and then pattern match.

At 3>, we try to pattern match 1 to Variable, if Variable haven't been set before it will be 1. But now it is set at 2>. At 4>, we try to match Variable to 1, and if it is ok the value is returned. At 5> we have a mismatch, this is where we match things from right to left. So on the right hand side we try to match 1 to a. That fails because of a no match.

One other example we can do is fibonacci. Fibonacci works that it do addition on the last number and the number before. 1, 1, 2, 3, 5, 8, 13.

We implement this in Erlang using recursion and pattern matching.

```
-module(fibonacci).

-export([fib/1]).

fib(0) ->
    0;
fib(1) ->
    1;
fib(N) ->
    fib(N-1) + fib(N-2).
```

What does our code do?

We have two base case here, if N is 0 and if N is 1. In the last function clause we call fib function with N-1 and N-2.

maps, records, tuples

Say that we want to do different things depending on a value in a map.

We get a list with maps that looks something like:

```
[#{type => sms, text => <<"hi">>}, #{type => <<"email">>, text =>
↪  <<"hi">>}]
```

We want to sum the amount of each type in our list.

```
-module(pattern).

-export([sum_by_type/1,
         generate_list/0]).

sum_by_type(List) ->
    sum_by_type_aux(List, []).

sum_by_type_aux([], Acc) ->
    Acc;
sum_by_type_aux([#{type := Type} | Tail], Acc) ->
    case get_value(Acc, Type) of
        undefined ->
            sum_by_type_aux(Tail, [{Type, 1} | Acc]);
        Value ->
            NewAcc = remove_value(Acc, Type),
            sum_by_type_aux(Tail, [{Type, Value + 1} | NewAcc])
```

```
    end.

get_value([], _) ->
    undefined;
get_value([{Type, Value}|_], Type) ->
    Value;
get_value([_ | Tail], Type) ->
    get_value(Tail, Type).

remove_value(List, Type) ->
    remove_value_aux(List, Type, []).

remove_value_aux([], _, Acc) ->
    Acc;
remove_value_aux([{Type, _} | Tail], Type, Acc) ->
    remove_value_aux(Tail, Type, Acc);
remove_value_aux([Head | Tail], Type, Acc) ->
    remove_value_aux(Tail, Type, [Head | Acc]).

generate_list() ->
    [#{type => sms, text => <<"hi">>},
     #{type => email, text => <<"hi">>},
     #{type => sms, text => <<"hi">>},
     #{type => email, text => <<"hi">>},
     #{type => sms, text => <<"hi">>},
     #{type => sms, text => <<"hi">>},
     #{type => sms, text => <<"hi">>}].
```

Here is the code that will handle the sum and sort things. We create one function that will be the exported and that function will call a helper function with two arguments. First is the input we get, the other argument is the accumulator that we will return.

Helper function will match first element in the list and then check if we have already summed it. I have created my own functions now to get values and remove values to more show how pattern matching and recursion works. OTP library have proplists module or other modules to do this.

```
4> List = pattern:generate_list().
[#{text => <<"hi">>,type => sms},
 #{text => <<"hi">>,type => email},
 #{text => <<"hi">>,type => sms},
 #{text => <<"hi">>,type => email},
 #{text => <<"hi">>,type => sms},
 #{text => <<"hi">>,type => sms},
 #{text => <<"hi">>,type => sms}]
```

```
5> pattern:sum_by_type(List).
[{sms,5},{email,2}]
```

Small examples on how to use lists, tuples and maps.

Releases

Supervisors

Beam languages

LFE

It is possible to start an LFE web application with Nova.

```
rebar3 new nova_lfe mylfeapp
```

This will follow much of the getting started pages in Nova section.