The Flamingo Route

Assignment 4

Kai Arne S. Myklebust, Silvan Adrian

Handed in: October 10, 2018

Contents

1 Solution

1.1 Files

All Files are situated in the **src/** folder:

- **flamingo.erl** The flamingo server implmentation
- **greetings.erl** The greetings module implementation

- **hello.erl** The hello module implementation
- **mood.erl** Mood module implementation
- **counter.erl** Counter module implementation
- **test_xxx.erl** Tests for each module

1.2 Running the programm

Out of convenience we used a Emakefile which compiles all the erlang files in one go then rather compile each file on it's own. This can be done by using the erlang shell and run:

```
1 make:all([load]).
```

1.3 Running the tests

The tests can be run with eunit, we included tests for each module in a own file. Example running tests for flamingo:

```
1 eunit:test(test_flamingo, [verbose]).
```

2 Implementation

2.1 Assumptions

We assumed that empty Paths shouldn't be inputed into a routing group, therefore they get ignored. Also we decided that duplicated paths get removed from existing routing groups and get added as a new routing group. In case the path of a routing group is empty the routing group gets removed from the routing group list.

2.2 Our implementation

We chose to implement the routing groups as a list, which contains triplets consisting of List of Paths, Function and State. When a new route gets registered we traverse the routing group list and check every single path in each routing group to either update it or insert it as a new routing group.
For prefix handling we also traverse each routing group to search for the longest

matching prefix in each group, from there we take the longest out of the longest matching ones.

Sadly we weren't able to solve the issue of functions of each routing group only running consecutively.

2.3 Possible improvements

We started off doing it with a map, but ended up using a list since it was easier to handle. For big routing groups a map might be faster due to easier lookups and handling of the data. Also traversing the lists is not very nice due to lists in lists which ends up to be not easily readable.

For letting the function in a routing group not run in a concurrent fashion we would use spawn_link to keep track if a process is still running or not.

3 Assessment

3.1 Scope of Test Cases

We tested edge cases as well as happy cases, which we came up with. The only thing we weren't able to test is some try/catches where we weren't able to produce an error case. Also thanks to our test cases we were able to find some bugs, for example the empty Path and empty Path list.

3.2 Correctness

We are quite happy with the result, since our test cases and the OnlineTA didn't find any more bugs or errors. So we come to the conclusion that our code is relatively stable and useable.

3.3 Code Quality

Our code is well structured and commented where needed, on the other hand our loop seems to be a little bloated which might not seem that nice. But we weren't able to find a much better solution for it, so we are quite happy with it. We followed closely the requirements specified in the assignment, which we wrote tests for. Therefore we think that the functionality matches the requirements.

A Code Listing

```
-module(flamingo).

-export([new/1, request/4, route/4, drop_group/2]).

new(Global) ->
  try ({ok, spawn(fun() -> loop(Global, []) end)})
  catch
    _:Error -> {error, Error}
  end.

request(Flamingo, Request, From, Ref) ->
  Flamingo ! {From, request, Request, Ref}.

route(Flamingo, Path, Fun, Arg) ->
  Flamingo ! {self(), routes, Path, Fun, Arg},
  receive
    {Status, Content} -> {Status, Content}
  end.

drop_group(_Flamingo, _Id) ->
  not_implemented.

loop(Global, RouteGroups) ->
  receive
    % requests
    {From, request, {Path, Request}, Ref} ->
      % get the matching route, with action(Fun) and state(Arg)
      {MatchedRoute, Fun, Arg} = getMatchingRoute(Path,
      ↪  RouteGroups, {"", none, none}),
      case MatchedRoute of
        "" -> From ! {Ref, {404, "No matching route found"}};
        _ ->
        % try to run the action
        try Fun({MatchedRoute, Request}, Global, Arg) of
          R ->
            case R of
              % if the action returns new_state we update the
              ↪  local state
              {new_state, Content, NewState} ->
                NewRouteGroups = updateState(RouteGroups,
                ↪  MatchedRoute, NewState),
```

```
39                  From ! {Ref, {200, Content}}, % everything worked
                     ↪  fine so 200
40                  loop(Global, NewRouteGroups);
41                {no_change, Content} ->
42                  From ! {Ref, {200, Content}}, % everything worked
                     ↪  fine so 200
43                  loop(Global, RouteGroups)
44              end
45          catch
46            error:_ ->
47              From ! {Ref, {500, "error in action"}},
48              loop(Global, RouteGroups)
49          end
50        end,
51        loop(Global, RouteGroups);
52      % new routes, need to update routing groups
53      {From, routes, Path, Fun, Arg} ->
54        case Path of
55          [] ->
56            From ! {error, "No Path given"},
57            loop(Global, RouteGroups);
58          [""] ->
59            From ! {error, "Empty Path given"},
60            loop(Global, RouteGroups);
61          _ -> try updateRouteGroups(Path, Fun, Arg, RouteGroups)
               ↪  of
62                NewRoutes ->
63                  From ! {ok, make_ref()},
64                  loop(Global, NewRoutes)
65                catch
66                   _:Reason ->
67                    From ! {error, Reason},
68                    loop(Global, RouteGroups)
69            end
70        end
71    end.
72
73  % update local state if an action returned a state
74  updateState([{Path, Fun, Arg} | GroupTail], MatchedRoute,
    ↪  NewState) ->
75    case lists:member(MatchedRoute, Path) of
76      true -> [{Path, Fun, NewState} | GroupTail];
77      false -> [{Path, Fun, Arg} | updateState(GroupTail,
         ↪  MatchedRoute, NewState)]
```

```
78    end.
79
80  % adds the new Path, Action and arguments to the routing group
81  updateRouteGroups(Path, Fun, Arg, OldGroup) ->
82    [{Path, Fun, Arg} | updateOldGroup(Path, OldGroup)].
83
84  % removes older routings with the same path as the new one
85  % generates a new list
86  updateOldGroup(_, []) -> [];
87  updateOldGroup(NewPath, [{Path, Fun, Arg} | GroupTail]) ->
88    NewPathGroup = updateOldGroupPaths(NewPath, Path),
89    % if a group has no path associated anymore we delete the whole
      ↪  group
90    case NewPathGroup == [] of
91      true -> updateOldGroup(NewPath, GroupTail);
92      false -> [{NewPathGroup, Fun, Arg} | updateOldGroup(NewPath,
        ↪  GroupTail)]
93    end.
94
95  % skips where the old path is the same as in the new path
96  % generates a new list with all old paths
97  updateOldGroupPaths(_, []) -> [];
98  updateOldGroupPaths(NewPath, [OldPath | OldPathTail]) ->
99    case lists:member(OldPath, NewPath) of
100     true -> updateOldGroupPaths(NewPath, OldPathTail);
101     false -> [OldPath | updateOldGroupPaths(NewPath,
        ↪  OldPathTail)]
102   end.
103
104 % gets the matching route by calling for the prefix
105 % and if it is longer than the one we had from before it updates
106 getMatchingRoute(_Path, [], MatchedRoute) -> MatchedRoute;
107 getMatchingRoute(Path, [Routes | RestRoutingGroup ],
    ↪  {MatchedRoute, Fun, Arg}) ->
108     {MatchedRouteNew, FunNew, ArgNew} = matchPrefix(Path,
        ↪  Routes),
109     case length(MatchedRouteNew) > length(MatchedRoute) of
110       true -> getMatchingRoute(Path, RestRoutingGroup,
          ↪  {MatchedRouteNew, FunNew, ArgNew});
111       false -> getMatchingRoute(Path, RestRoutingGroup,
          ↪  {MatchedRoute, Fun, Arg})
112     end.
113
114 % gets the longest prefix from this routing group
```

```
115 % if no prefixes it returns the empty string
116 matchPrefix(Path, {Routes, Fun, Arg}) ->
117   MatchedPrefixes = lists:filter(fun(Route) -> string:left(Path,
      ↪  length(Route)) == Route end,Routes),
118   case MatchedPrefixes of
119     [] -> {"", Fun, Arg};
120     _ -> {lists:max(MatchedPrefixes), Fun, Arg}
121   end.
```

```
1  -module(counter).
2  -export([server/0,try_it_inc/2,try_it_dec/2]).
3
4  getNum(Arg) ->
5    % check if x is in one of the tuples
6    case lists:keyfind("x", 1, Arg) of
7      false -> 1;
8      {"x", SNumber} ->
9        case string:to_integer(SNumber) of
10         {error, _} -> 1;
11         {Int, _} ->
12           case Int > 0 of
13             false -> 1;
14             true -> Int
15           end
16       end
17   end.
18
19 counter({Path,Arg}, _, State) ->
20   case Path of
21     "/inc_with" ->
22       NewState = State + getNum(Arg),
23       Content = integer_to_list(NewState),
24       {new_state, Content, NewState};
25     "/dec_with" ->
26       NewState = State - getNum(Arg),
27       Content = integer_to_list(NewState),
28       {new_state, Content, NewState}
29   end.
30
31 server() ->
32   {ok, F} = flamingo:new("Mood Module"),
33   flamingo:route(F, ["/inc_with", "/dec_with"], fun counter/3,
      ↪  0),
```

```
  F.

try_it_inc(Server, Arg) ->
  Me = self(),
  Ref = make_ref(),
  flamingo:request(Server, {"/inc_with", Arg}, Me, Ref),
  receive
    {Ref, Reply} -> Reply
  after 5000 ->
    erlang:error(timeout)
  end.

try_it_dec(Server, Arg) ->
  Me = self(),
  Ref = make_ref(),
  flamingo:request(Server, {"/dec_with", Arg}, Me, Ref),
  receive
    {Ref, Reply} -> Reply
  after 5000 ->
    erlang:error(timeout)
  end.
```

```
-module(mood).
-export([server/0,try_it_mood/1,try_it_moo/1]).

mood({Path,_}, _, State) ->
  case Path of
    "/mood" ->
      case State of
        1 -> {no_change, "Happy!"};
        _ -> {no_change, "Sad"}
      end;
    "/moo" ->
      {new_state, "That's funny", 1}
  end.

server() ->
  {ok, F} = flamingo:new("Mood Module"),
  flamingo:route(F, ["/mood", "/moo"], fun mood/3, 0),
  F.

try_it_mood(Server) ->
  Me = self(),
```

```
  Ref = make_ref(),
  flamingo:request(Server, {"/mood", []}, Me, Ref),
  receive
    {Ref, Reply} -> Reply
  after 5000 ->
    erlang:error(timeout)
  end.

try_it_moo(Server) ->
  Me = self(),
  Ref = make_ref(),
  flamingo:request(Server, {"/moo", []}, Me, Ref),
  receive
    {Ref, Reply} -> Reply
  after 5000 ->
    erlang:error(timeout)
  end.
```

```
-module(hello).
-export([server/0,try_it_hello/1,try_it_goodbye/1]).

hello({_, _}, _, _) ->
  {no_change, "Hello my friend"}.
goodbye({_, _}, _, _) ->
  {no_change, "Sad to see you go."}.

server() ->
  {ok, F} = flamingo:new("Hello Module"),
  flamingo:route(F, ["/hello"], fun hello/3, none),
  flamingo:route(F, ["/goodbye"], fun goodbye/3, none),
  F.

try_it_hello(Server) ->
  Me = self(),
  Ref = make_ref(),
  flamingo:request(Server, {"/hello", []}, Me, Ref),
  receive
    {Ref, Reply} -> Reply
  end.

try_it_goodbye(Server) ->
  Me = self(),
  Ref = make_ref(),
```

```
26    flamingo:request(Server, {"/goodbye", []}, Me, Ref),
27    receive
28      {Ref, Reply} -> Reply
29    end.
```