Flamingo

Philip Lassen (vgh804), Nicolas Larsen (vgn209)

November 2018

1 Implementation

1.1 flamingo

The majority of the logic for the flamingo server was implemented in the loop function. The loop took 3 arguments. One was the global state of the Flamingo server. The second argument RouteMap is a map of prefixes to to Reference IDs. These Reference IDs are shared among routing groups. The third argument for the loop function is StateMap which maps the Reference IDs of the Routing groups to a pair. This pair is the associated action for the routing group and the associated local state for the routing group. We made this implementation choice because it allowed us just to change the value that the IDs for the routing group points to.

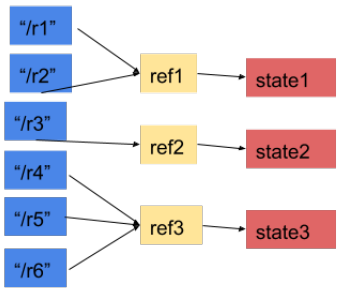

Figure 1: Routing Group Implementation

Figure 1 gives an illustration of how this implementation works. If we have three routing groups: {"/r1", "/r2"}, {"/r3"}, and {"/r4", "/r5", "/r6"}. Each of these groups points to their own unique reference id. In this case ref1, ref2, and ref3 respectively. And then each of these IDs further points to the local state, state1, state2 and state3 respectively for this example.

We created a helper function to make sure that both when two prefixes match a path that the longer prefix is matched. The helper function getRoute

takes a String and List and finds the first element in the list which is prefix for the string provided as an argument. In order for this to have the desired affect we get a list of keys of the routeMap and reverse it, and then pass it into this function. The reason we do this is because the key list is by default return in alphabetically sorted order and we aim to match the longest string that matches the path. Thus we reverse.

1.2 Supporting Module's

1.2.1 Hello

The Hello Module had a very simple implementation. The routes `"\hello"` and `"\goodbye"` have simple action's associated with them that simply print the desired output.

1.2.2 Mood

The prefixes `"\moo"` and `"\mood"` are registered to the same routing group which uses boolean to represent the local State. The local State is initialized to false. The associated action is a function which in the case of `"\moo"` prints "That is funny". If `"\moo"` has been called for the first time the local State is changed to true. If the path given as an argument is `"\mood"` then if the Local state is false we print "Sad" and otherwise we print "Happy!".

1.2.3 Counter

The prefixes `"inc_with"` and `"dec_with"` are registered to the same routing group which uses an integer to represent the local state. The local state is initialized to 0. The associated action is a function which in the case of `"\inc_with"` increments the local state by 1, except if one of the arguements to the action is a key of `"x"` with an integer value, then we increment the local state by the value. In the case of `"\dec_with"` the we do the exact same thing we do in `"\inc_with"` except we decrement instead of incrementing.

2 Testing

We passed all the provided tests by the online TA. For our own tests we used the EUnit framework. We tested the Flamingo Server and the modules we were meant to implement in the test file `test_server.erl`.

There are four separate functions that test a variety of different pieces of the flamingo server as well as the implementation of the 3 modules that we were asked to implement. We make sure to test that the three possible responses 200, 404, and 500 are successfully managed. We also test to make sure that the modules have the required functionality, Successful response, no matching route, and action failed respectively.

The tests showed that all the specifications were met and that responses were handled successfully.

The tests can be run by doing the following (provided you are in the src directory.

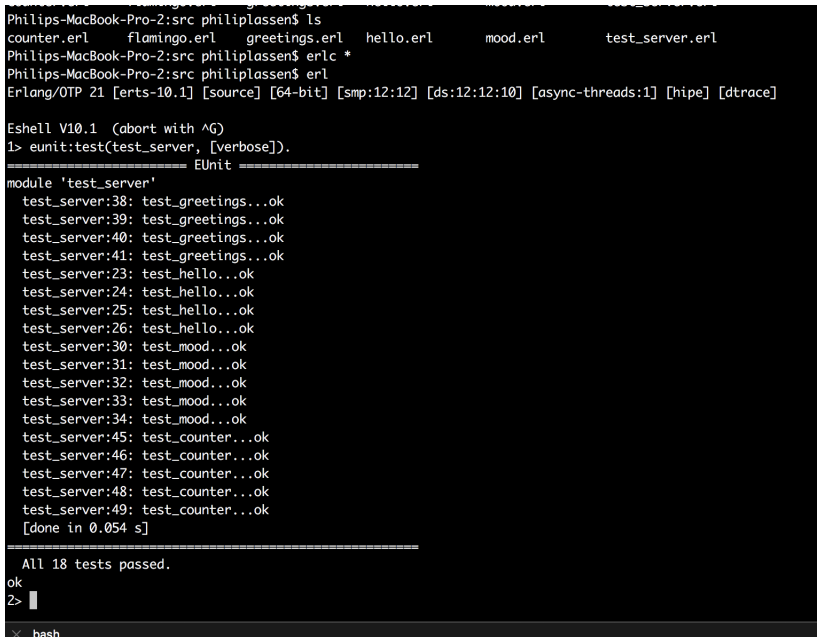

Figure 2: Running Tests

3 Assessment

We were able to implement the flamingo module and all the other modules in a way that worked with respect to the specification. So all in all we were very pleased with our work on this assignment. We felt that it may have been possible to handle the routing groups in a neater way instead of having to maintain two maps to keep track of routing groups and local state. However we could not come up with a better structure. It isn't an intuitive implementation, which can make the code difficult to read. But all in all the implementation was fairly concise.

4 Code

4.1 Flamingo

```
-module(flamingo).
-export([new/1, route/4, loop/3, getRoute/2, request/4, drop_group/2]).

%Create and return a new Server
new(_Global) -> try
                  {ok, spawn(flamingo, loop, [_Global, #{}, #{}])}
                catch _:_ ->
                  {error, "There was an issue"}
                end.

%Route a prefix group on the Flamingo Server with an Action and Initial Local State
route(Flamingo, Prefixes, Action, Init) ->
  Flamingo ! {self(), {route, Prefixes, Action, Init}},
  receive
    {ok, Ref} -> {ok, Ref}
  end.

loop(State, RouteMap, StateMap) ->
  receive
    {From, {route, Prefixes, Action, Init}} ->
      Ref = make_ref(),
      Pairs = [{P, Ref} || P <- Prefixes],
      Fun = fun(K, V, AccIn) -> maps:put(K, V, AccIn) end,
      NewRouteMap = maps:fold(Fun, RouteMap, maps:from_list(Pairs)),
      NewStateMap = maps:put(Ref, {Action, Init}, StateMap),
      From ! {ok, Ref},
      loop(State, NewRouteMap, NewStateMap);
    {From, {test}} ->
      From ! {results, RouteMap, StateMap},
      loop(State, RouteMap, StateMap);
    {request, {Path, Args}, From, Ref} ->
      case getRoute(Path, lists:reverse(maps:keys(RouteMap))) of
        {error, 404} -> From ! {Ref, {404, "There are no matching routes"}},
                        loop(State, RouteMap, StateMap);
        {ok, P} ->
          R = maps:get(P, RouteMap),
          {F, LocalState} = maps:get(R, StateMap),
          try
            Result = apply(F, [{P, Args}, State, LocalState]),
            case Result of
              {new_state, Content, NewState} ->
                NewStateMap = maps:put(R, {F, NewState}, StateMap),
```

```
                From ! {Ref, {200, Content}},
                loop(State, RouteMap, NewStateMap);
              {no_change, Content} ->
                From ! {Ref, {200, Content}},
                loop(State, RouteMap, StateMap);
              _ ->
                From ! {Ref, {500, "Action failed on the flamingo server"}},
                loop(State, RouteMap, StateMap)
            end
          catch _:_ ->
             From ! {Ref, {500, "Action failed on the flamingo server"}},
             loop(State, RouteMap, StateMap)
          end
      end
  end.

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

%Routes must be in reverse order"
getRoute(_, []) -> {error, 404};
getRoute(Path, [H | T]) ->
  case  string:substr(Path, 1, length(H)) of
    H -> {ok, H};
    _ -> getRoute(Path, T)
  end.
```

4.2 Hello

```
-module(hello).
-export([server/0]).
hello({_Path, _}, _, _) ->
    {no_change, "Hello my friend"}.
bye({_Path, _}, _, _) ->
    {no_change, "Sad to see you go."}.
server() ->
    {ok, F} = flamingo:new("Hi and Bye Server"),
    flamingo:route(F, ["/hello"], fun hello/3, none),
    flamingo:route(F, ["/goodbye"], fun bye/3, none),
    F.
```

4.3 Mood

```
-module(mood).
-export([server/0]).
mooHandler({_Path, _}, _, LocalState) ->
```

```
  case _Path of
    "/moo" -> Content = "That's funny",
              if
                LocalState -> {no_change, Content};
              true ->
                {new_state, Content, true}
              end;
    "/mood" -> if
                 LocalState -> {no_change, "Happy!"};
               true ->
                 {no_change, "Sad"}
               end
    end.
server() ->
    {ok, F} = flamingo:new("The Flamingo Server"),
    flamingo:route(F, ["/mood", "/moo"], fun mooHandler/3, false),
    F.
```

4.4 Counter

```
-module(counter).
-export([server/0]).
argParser([]) -> 1;
argParser([{"x", Val} | Remainder]) ->
                              %io:fwrite("matched with x"),
                              try
                                Arg = list_to_integer(Val),
                                if
                                  is_integer(Arg) and (Arg > 0) -> Arg;
                                true ->
                              %    io:fwrite("made it to the else branch"),
                                    argParser(Remainder)
                                end
                              catch error:badarg ->
                                argParser(Remainder)
                              end;

argParser([_ | Remainder]) -> argParser(Remainder).
countHandler({_Path, Array}, _, LocalState) ->
  Val = argParser(Array),
  case _Path of
    "/inc_with" -> {new_state, integer_to_list(Val + LocalState), Val + LocalState};
    "/dec_with" -> {new_state, integer_to_list(LocalState - Val), LocalState - Val}
  end.
server() ->
    {ok, F} = flamingo:new("The Counter Server"),
```

```
flamingo:route(F, ["/inc_with", "/dec_with"], fun countHandler/3, 0),
F.
```