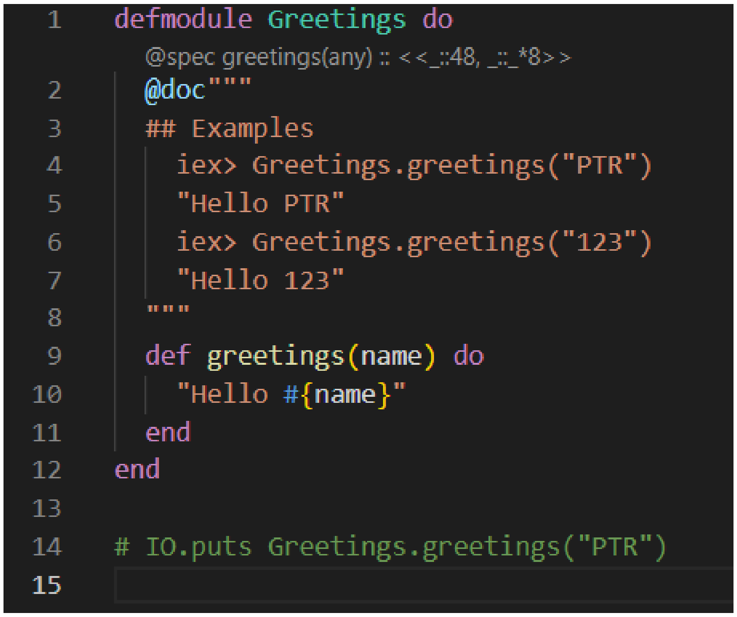
FAF.PTR16.1 -- Project 0

Performed by: Covtun Serghei, group FAF-203
Verified by: asist. univ. Alexandru Osadcenco

P0W1

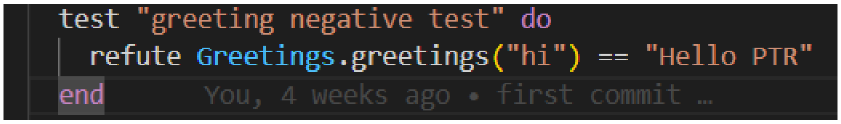
Task1: Write a script that would print the message "Hello PTR" on the screen.



```elixir
defmodule Greetings do
  @spec greetings(any) :: <<_::48, _::_*8>>
  @doc"""
  ## Examples
    iex> Greetings.greetings("PTR")
    "Hello PTR"
    iex> Greetings.greetings("123")
    "Hello 123"
  """
  def greetings(name) do
    "Hello #{name}"
  end
end

# IO.puts Greetings.greetings("PTR")

```

Function just return string `Hello` with `name` variable in it.

Task2: Create a unit test for your project.



```elixir
test "greeting negative test" do
  refute Greetings.greetings("hi") == "Hello PTR"
end        You, 4 weeks ago • first commit …
```

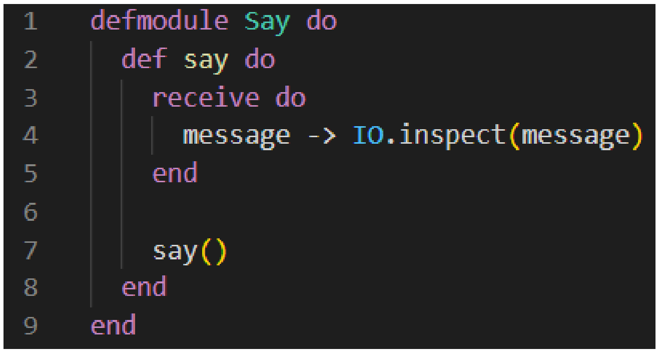
Following test checks if output of `func/1` greetings is equal to "Hello PTR"

P0W2

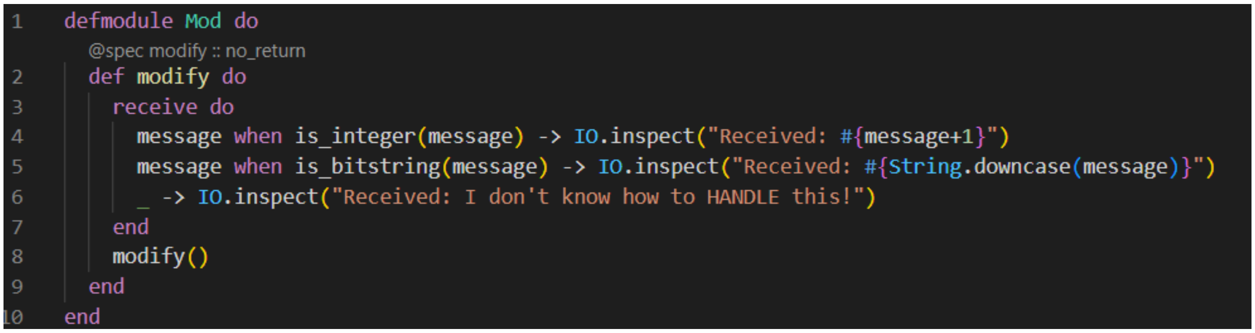
Task: The main tasks was to create some math functions and functions to perform actions on strings. The whole info about functions, explanations and examples could be found on the link:
https://github.com/siorkis/PTR/tree/main/lib/lab1/checkpoint_2

P0W3

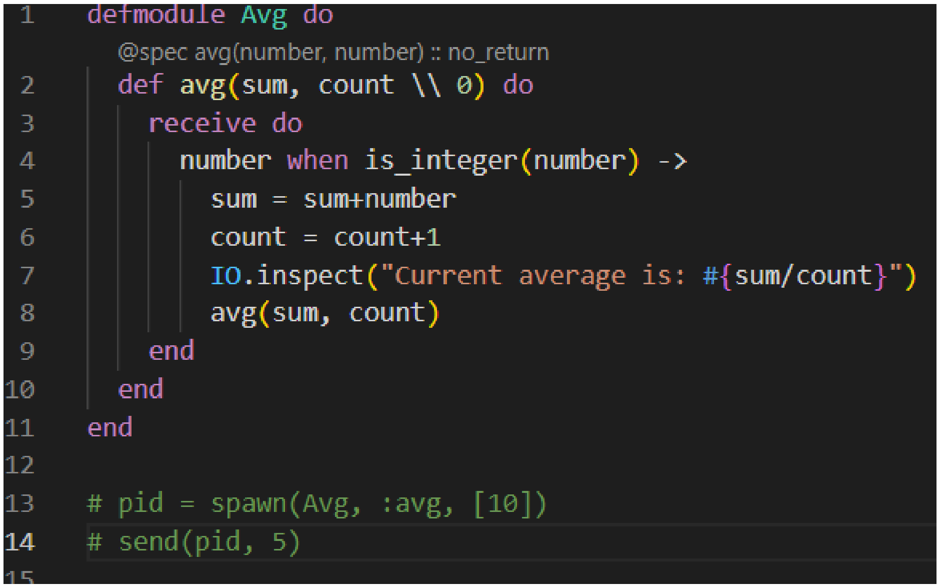
Task1: Minimal Task is creating several actors, one of them would print any message that it receives. Second one should monitor the others and if one stops, actor should be notified via message. The third one should receive numbers and print current average with each request.



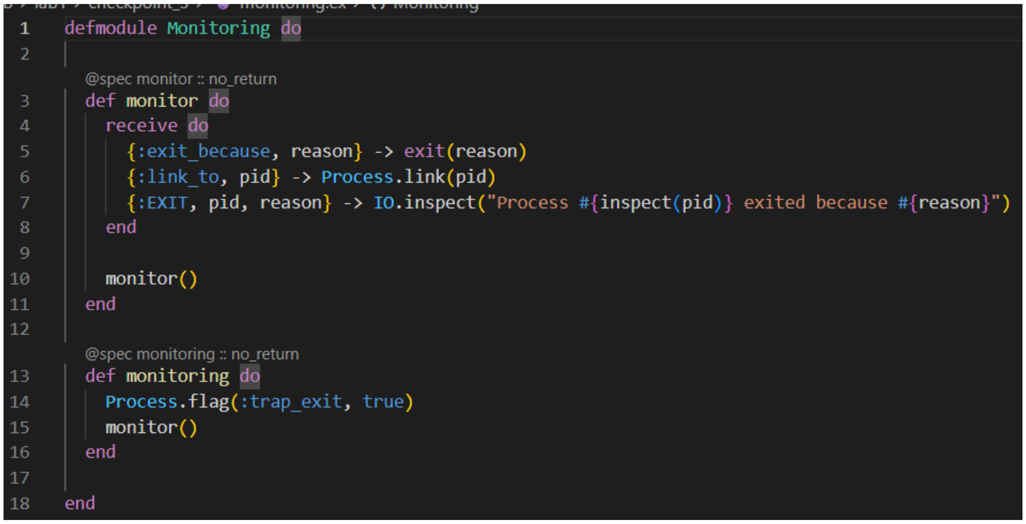
```
defmodule Say do
  def say do
    receive do
      message -> IO.inspect(message)
    end

    say()
  end
end
```



```
defmodule Mod do
  @spec modify :: no_return
  def modify do
    receive do
      message when is_integer(message) -> IO.inspect("Received: #{message+1}")
      message when is_bitstring(message) -> IO.inspect("Received: #{String.downcase(message)}")
      _ -> IO.inspect("Received: I don't know how to HANDLE this!")
    end
    modify()
  end
end
```



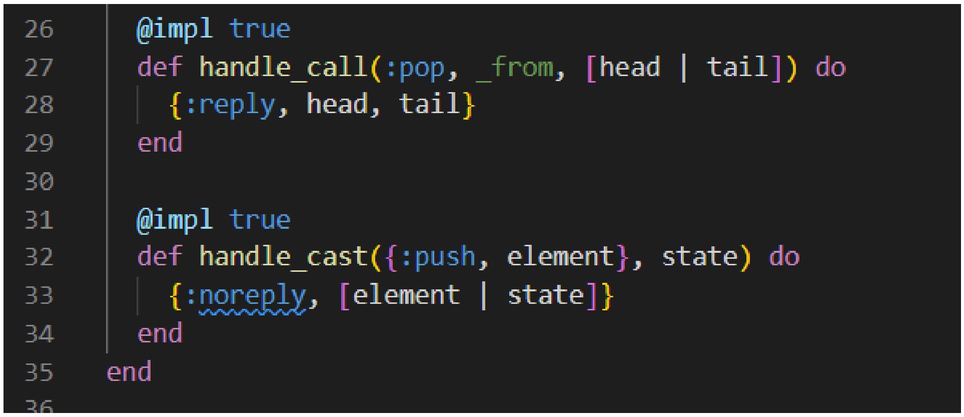
```
defmodule Avg do
  @spec avg(number, number) :: no_return
  def avg(sum, count \\ 0) do
    receive do
      number when is_integer(number) ->
        sum = sum+number
        count = count+1
        IO.inspect("Current average is: #{sum/count}")
        avg(sum, count)
    end
  end
end

# pid = spawn(Avg, :avg, [10])
# send(pid, 5)
```



```elixir
defmodule Monitoring do

  @spec monitor :: no_return
  def monitor do
    receive do
      {:exit_because, reason} -> exit(reason)
      {:link_to, pid} -> Process.link(pid)
      {:EXIT, pid, reason} -> IO.inspect("Process #{inspect(pid)} exited because #{reason}")
    end

    monitor()
  end

  @spec monitoring :: no_return
  def monitoring do
    Process.flag(:trap_exit, true)
    monitor()
  end

end
```

Module `Mod` returns modified message that receives (int = int+1 | String going to lowercase). Module `Avg` stores in local variable sum of all input numbers and count them, then returns average `sum/count`. Module `Monitoring` allows to create two actors and link one to another. When one process exited (`send(pid, {:exited_because, :bad_thing})`) then first actor will see it.

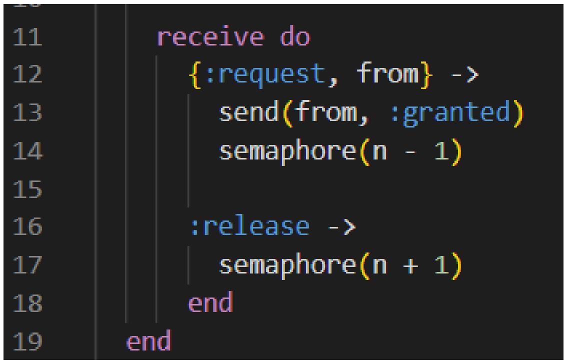
Task2: Main Task is to create actor with stack behavior and implement semaphore.



```elixir
  @impl true
  def handle_call(:pop, _from, [head | tail]) do
    {:reply, head, tail}
  end

  @impl true
  def handle_cast({:push, element}, state) do
    {:noreply, [element | state]}
  end
end
```

4

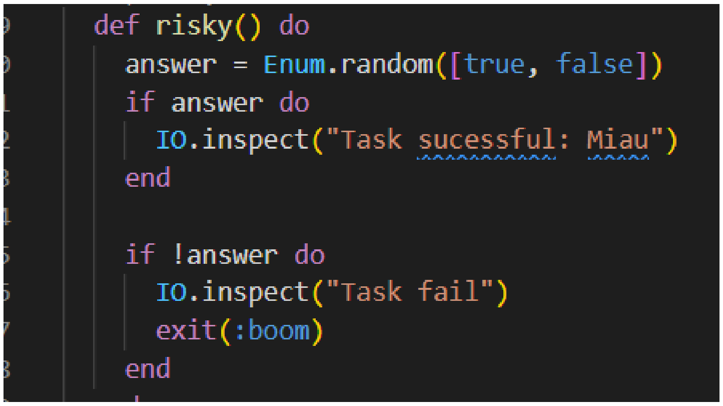
Here actor receives calls and either push element in the local storage list or pop them.



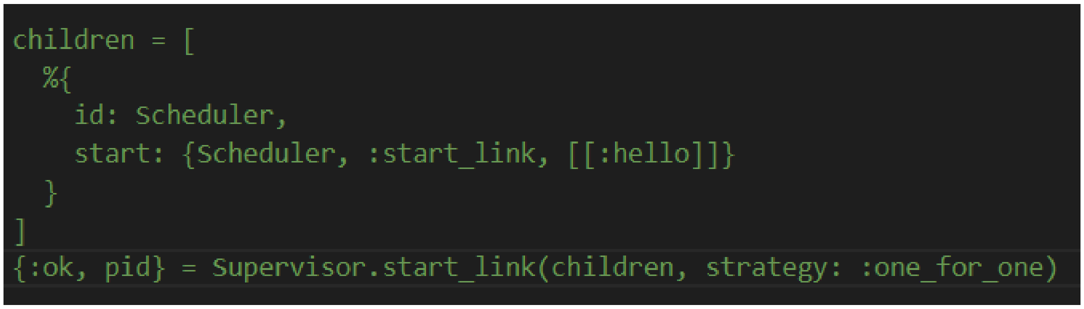
```
    receive do
      {:request, from} ->
        send(from, :granted)
        semaphore(n - 1)

      :release ->
        semaphore(n + 1)
      end
  end
```

I have implemented counter semaphore which receives at initialization number of maximum allowed processes, and by adding process goes down and up otherwise. Ex: We have 0 running processes: Sem=3; Then we have 1 running process: Sem=2; The we again have 0 running processes: Sem=3. So Semaphore allows to only N process to run at the same time.

Task3: Bonus Task is to create scheduler and actor which can crashes with 50% chance and scheduler should restart it in that case.



```
  def risky() do
    answer = Enum.random([true, false])
    if answer do
      IO.inspect("Task sucessful: Miau")
    end

    if !answer do
      IO.inspect("Task fail")
      exit(:boom)
    end
```

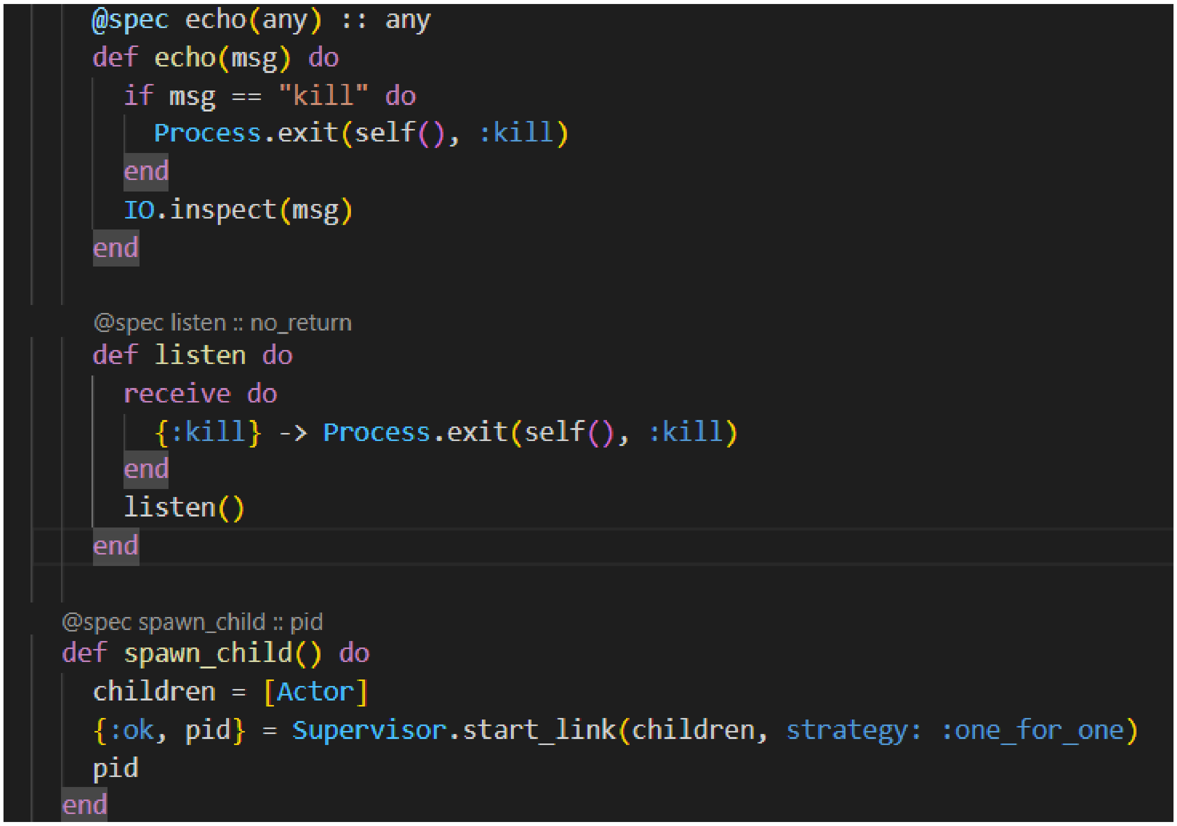


```
children = [
  %{
    id: Scheduler,
    start: {Scheduler, :start_link, [[:hello]]}
  }
]
{:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
```

We have random choice between true and false. In false case task exited and restarted by Supervisor.

P0W4

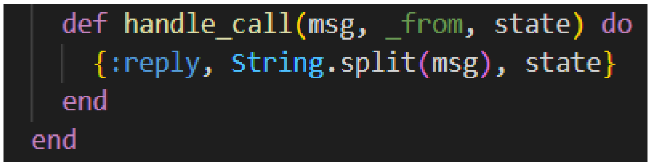
Tast1: Minimal Task is to create actor that echo any message that receives, bit in case of `kill` message it should be exited and restarted.



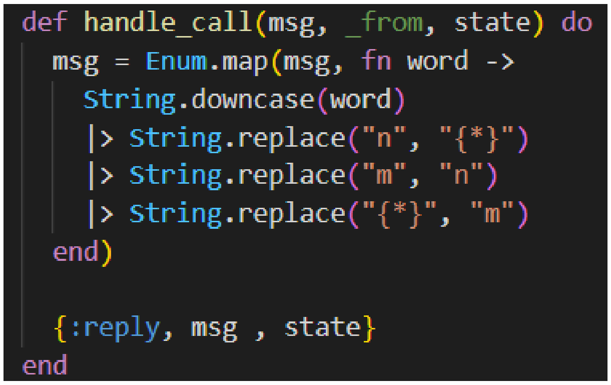
```
    @spec echo(any) :: any
    def echo(msg) do
      if msg == "kill" do
        Process.exit(self(), :kill)
      end
      IO.inspect(msg)
    end

    @spec listen :: no_return
    def listen do
      receive do
        {:kill} -> Process.exit(self(), :kill)
      end
      listen()
    end

  @spec spawn_child :: pid
  def spawn_child() do
    children = [Actor]
    {:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
    pid
  end
```

If actor receives “kill” it exit and restarts.

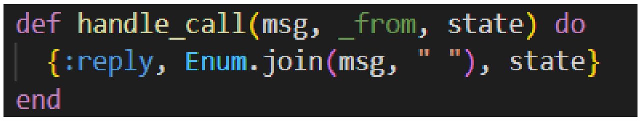
Task2: Create a supervised processing line to clean messy strings. The first worker in the line would split the string by any white spaces (similar to Python’s str.split method). The second actor will lowercase all words and swap all m’s and n’s (you nomster!). The third actor will join back the sentence with one space between words (similar to Python’s str.join method). Each worker will receive as input the previous actor’s output, the last actor printing the result on screen. If any of the workers die because it encounters an error, the whole processing line needs to be restarted. Logging is welcome



```elixir
  def handle_call(msg, _from, state) do
    {:reply, String.split(msg), state}
  end
end
```



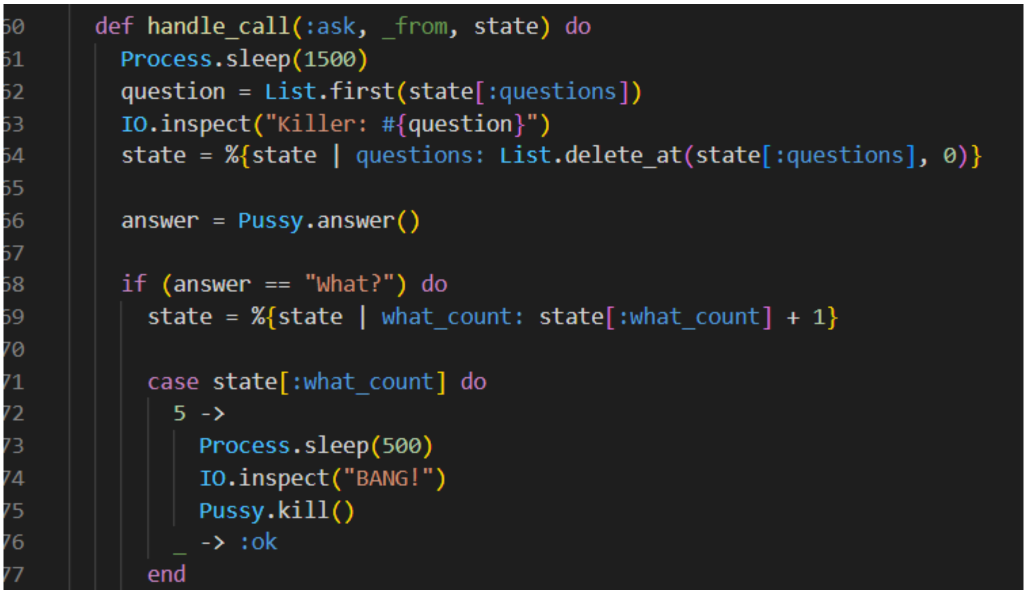
```elixir
def handle_call(msg, _from, state) do
  msg = Enum.map(msg, fn word ->
    String.downcase(word)
    |> String.replace("n", "{*}")
    |> String.replace("m", "n")
    |> String.replace("{*}", "m")
  end)

  {:reply, msg , state}
end
```



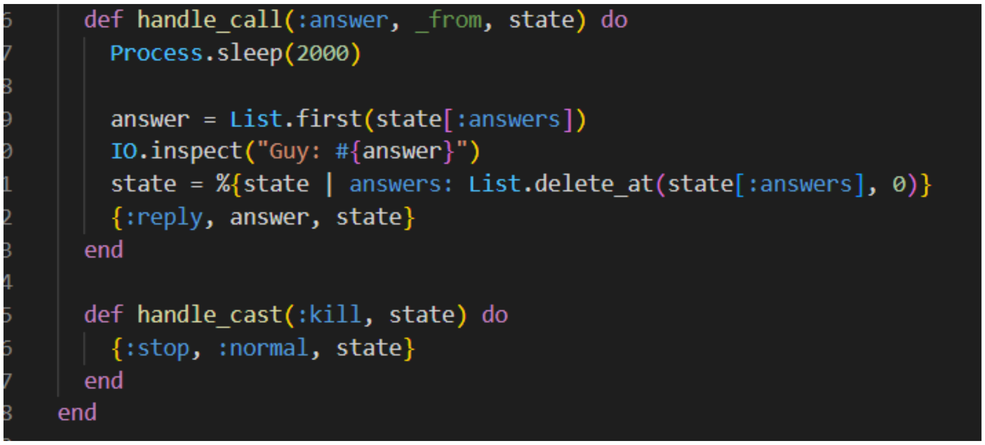
```elixir
def handle_call(msg, _from, state) do
  {:reply, Enum.join(msg, " "), state}
end
```

Those three methods performs build in functions like `split`, `replace`, `join`.

Task3: Write an application that, in the context of actor supervision. would mimic the exchange in that scene from the movie Pulp Fiction



```elixir
def handle_call(:ask, _from, state) do
  Process.sleep(1500)
  question = List.first(state[:questions])
  IO.inspect("Killer: #{question}")
  state = %{state | questions: List.delete_at(state[:questions], 0)}

  answer = Pussy.answer()

  if (answer == "What?") do
    state = %{state | what_count: state[:what_count] + 1}

    case state[:what_count] do
      5 ->
        Process.sleep(500)
        IO.inspect("BANG!")
        Pussy.kill()
      _ -> :ok
    end
```

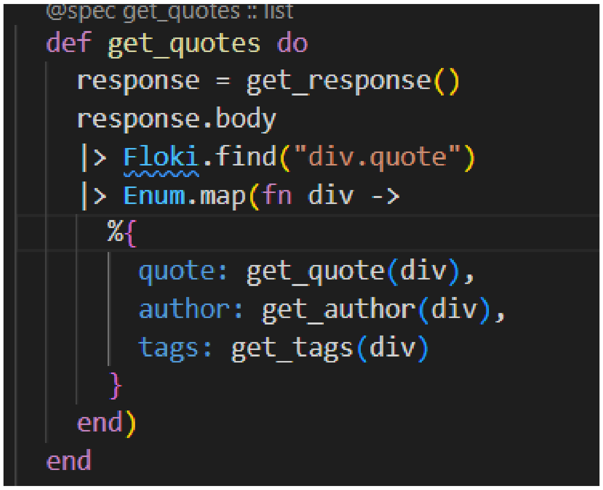


```
  def handle_call(:answer, _from, state) do
    Process.sleep(2000)

    answer = List.first(state[:answers])
    IO.inspect("Guy: #{answer}")
    state = %{state | answers: List.delete_at(state[:answers], 0)}
    {:reply, answer, state}
  end

  def handle_cast(:kill, state) do
    {:stop, :normal, state}
  end
end
```

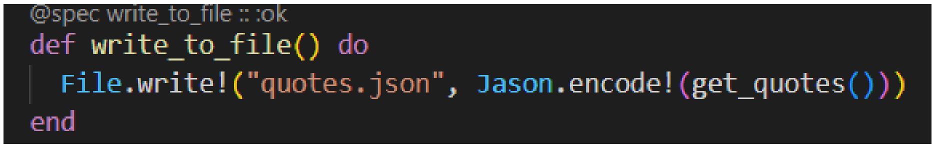
Module PulpFiction starts Killer module which starts Buyer module. Killer and Buyer have lists of questions and answers, which are pops to console one by one. Also Killer have counter of word `What?`, if it reaches 5 Killer kills actor Buyer.

P0W5

Tast1: Minimal Task is Write an application that would visit this link. Print out the HTTP response status code, response headers and response body. Extract all quotes from the HTTP response body. Collect the author of the quote, the quote text and tags. Save the data into a list of maps, each map representing a single quote. Persist the list of quotes into a file. Encode the data into JSON format. Name the file quotes.json.



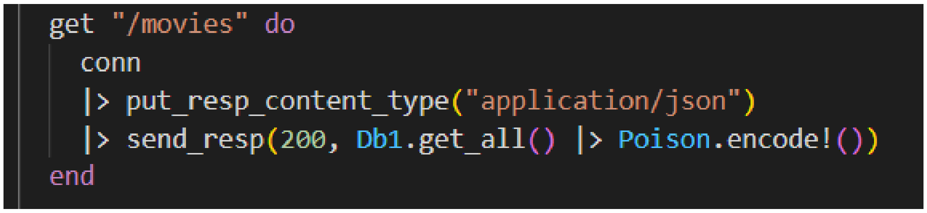
```
@spec get_quotes :: list
def get_quotes do
  response = get_response()
  response.body
  |> Floki.find("div.quote")
  |> Enum.map(fn div ->
    %{
      quote: get_quote(div),
      author: get_author(div),
      tags: get_tags(div)
    }
  end)
end
```



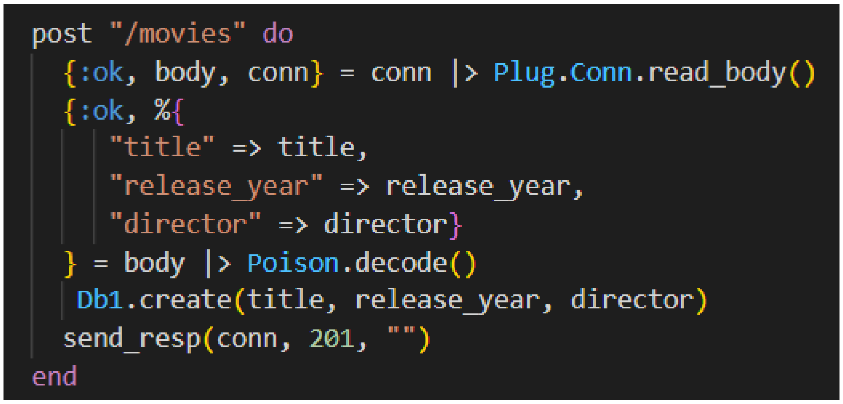
```
@spec write_to_file :: :ok
def write_to_file() do
  File.write!("quotes.json", Jason.encode!(get_quotes()))
end
```

Code above gets response from provided link and start to parse through parsing three in order to find required blocks in html page, by tag and its class. Also you could create new file `quotes.json` with all quotes in it.

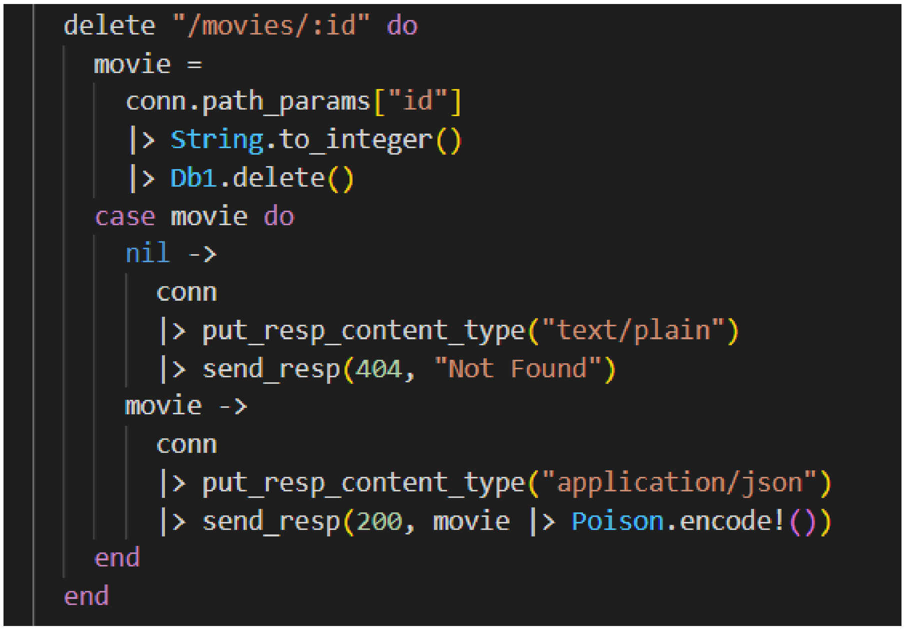
Tast1: Main Task is to write an application that would implement a Star Wars-themed RESTful API. The API should implement the basic HTTP methods.



```
get "/movies" do
  conn
  |> put_resp_content_type("application/json")
  |> send_resp(200, Db1.get_all() |> Poison.encode!())
end
```



```
post "/movies" do
  {:ok, body, conn} = conn |> Plug.Conn.read_body()
  {:ok, %{
      "title" => title,
      "release_year" => release_year,
      "director" => director}
  } = body |> Poison.decode()
   Db1.create(title, release_year, director)
  send_resp(conn, 201, "")
end
```



```
delete "/movies/:id" do
  movie =
    conn.path_params["id"]
    |> String.to_integer()
    |> Db1.delete()
  case movie do
    nil ->
      conn
      |> put_resp_content_type("text/plain")
      |> send_resp(404, "Not Found")
    movie ->
      conn
      |> put_resp_content_type("application/json")
      |> send_resp(200, movie |> Poison.encode!())
  end
end
```

Module DB1 represent local database with sort of basic CRUD operations. We can call them from router function which allows user interaction. POST method as well as PUT requires json format input with request, but others don’t need it. For example DELETE gets id from DB and delete that element if it cannot find such id it will returns message `Not found` otherwise success status code.

Conclusion.

During this laboratory work we learned the basics of elixir language, also we tried to write programs in functional way. Besides basic tasks we also tried to write actor/supervisor behavior and interact with http servers.

Bibliography.

https://elixir-lang.org/docs.html

https://hexdocs.pm/elixir/1.12.3/Kernel.html

https://elixirschool.com/en