Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real Time Programming

Laboratory work #0

Functional Programming / The Actor Model

Author:
Popa Eugeniu
std. gr. FAF-202

Supervisor:
Alexandru Osadcenco

Chișinău 2023

1 P0W1

Minimal Task Follow an installation guide to install the language / development environment of your choice.

I have downloaded the Elixir installer from the official website and installed it on my computer. I have used Visual Studio Code as a development tool, with "ElixirLS: Elixir support and debugger" extension.

Minimal Task Write a script that would print the message "Hello PTR" on the screen. Execute it.

```
def hello_ptr do
message = "Hello PTR"
IO.puts(message)
message
end
```

Main Task Initialize a VCS repository for your project. Push your project to a remote repo.

I have created a repository on GitHub.

Bonus Task Write a comprehensive readme for your repository.

The readme is in the repository.

Bonus Task Create a unit test for your project. Execute it.

```
use ExUnit.Case
doctest Week1

test "message is Hello PTR" do
assert Week1.hello_ptr() == "Hello PTR"
end
```

2 P0W2

Minimal Task Write a function that determines whether an input integer is prime.

```
def is_prime?(n) when n <= 1, do: :false
def is_prime?(n) when n in [2, 3], do: :true
def is_prime?(n) do
floored_sqrt = round(Float.floor(:math.sqrt(n)))
!Enum.any?(2..floored_sqrt, fn number -> rem(n, number) == 0 end)
end
```

The function takes an integer n as input and returns true if n is a prime number, and false otherwise.

Let's break down each line of the function:

```
def is_prime?(n) when n <= 1, do: :false
```

This line uses pattern matching to handle the cases where n is less than or equal to 1. In those cases, the function immediately returns :false, indicating that the number is not prime.

```
def is_prime?(n) when n in [2, 3], do: :true
```

This line handles the special cases where n is 2 or 3. In those cases, the function immediately returns :true, indicating that the number is prime.

```
def is_prime?(n) do
  floored_sqrt = round(Float.floor(:math.sqrt(n)))
  !Enum.any?(2..floored_sqrt, fn number -> rem(n, number) == 0 end)
end
```

This line handles all other cases where n is greater than 3. The function calculates the square root of n using the :math.sqrt/1 function and rounds it down to the nearest integer using Float.floor/1 and round/1.

Then, the function uses the !Enum.any?/2 function to check if any numbers between 2 and floored_sqrt evenly divide n. If any number does divide n evenly, the function returns false, indicating that n is not prime. If no number divides n evenly, the function returns true, indicating that n is prime.

In summary, this implementation checks if a number is prime by handling the special cases where the number is less than or equal to 1, 2, or 3. For all other cases, it checks if any numbers between 2 and the square root of the number divide the number evenly.

Minimal Task Write a function to calculate the area of a cylinder, given it's height and radius.

```
def cylinder_area(height, radius) do
area = 2 * :math.pi * radius * (height + radius)
Float.ceil(area, 4)
end
```

The function body starts with the assignment of a variable called area. This variable represents the surface area of the cylinder and is calculated using the formula for the lateral area of a cylinder: $2\pi rh$, where r is the radius and h is the height. The formula has been modified to account for the two circular bases of the cylinder, which have an area of πr^2 each, hence the $(2\pi rh + 2\pi r^2)$ term. The :math.pi atom is a constant value that represents the mathematical constant π (pi) in Elixir. The * operator is used for multiplication.
The Float.ceil function is called on the area variable. This function takes two arguments: a floating-point number and an integer that represents the number of decimal places to round up to. In this case, the area is rounded up to 4 decimal places.
The end keyword is used to mark the end of the function definition.
Overall, this function calculates the surface area of a cylinder and rounds the result to 4 decimal places.

Minimal Task Write a function to reverse a list.

```
def reverse(list) do
do_reverse(list, [])
end
```

This function uses recursion to reverse the order of elements in a given list.
The first function clause, def reverse(list) do ... end, simply calls the private do_reverse function with the input list and an empty list [] to store the reversed list.
The second and third function clauses define the private do_reverse function using Elixir's defp keyword, which indicates that the function can only be called within the same module.
The do_reverse function takes two arguments: the input list [h|t], which is pattern-matched into the head h and tail t, and an accumulator reversed that is initially an empty list [].

The first function clause, defp do_reverse([h|t], reversed) do ... end, uses recursion to reverse the order of elements in the input list. It takes the first element h from the input list and appends it to the front of the reversed list. It then calls itself with the remaining list t and the updated reversed list. This process continues recursively until the entire input list has been processed.
The second function clause, defp do_reverse([], reversed) do ... end, simply returns the reversed list once the input list has been fully processed.

Minimal Task Write a function to calculate the sum of unique elements in a list.

```
def unique_sum(list) do
unique_list = Enum.uniq(list)
Enum.sum(unique_list)
end
```

The function takes a list of numbers as input, removes all duplicates from the list, and returns the sum of the remaining unique numbers in the list.
Here's a breakdown of the function:
def unique_sum(list) do: This is a function definition in Elixir. It defines a function called unique_sum that takes one parameter called list.
unique_list = Enum.uniq(list): This line of code creates a new list called unique_list by removing all duplicates from the original list using the Enum.uniq/1 function from the Elixir standard library. This function takes a list as input and returns a new list with all duplicate elements removed.
Enum.sum(unique_list): This line of code calculates the sum of all the elements in the unique_list using the Enum.sum/1 function from the Elixir standard library. This function takes a list of numbers as input and returns the sum of all the numbers in the list.
The function returns the result of the previous line, which is the sum of all unique elements in the original list.
In summary, this function takes a list of numbers, removes all duplicates, and returns the sum of the remaining unique numbers in the list.

Minimal Task Write a function that extracts a given number of randomly selected elements from a list.

```
def extract_random_number(list, n) do
shuffled_list = Enum.sort_by(list, fn _ -> :rand.uniform() end)
Enum.take(shuffled_list, n)
end
```

The function takes a list of numbers as input and an integer n that specifies how many random numbers to extract from the list.
Here's a breakdown of the function:
def extract_random_number(list, n) do: This is a function definition in Elixir. It defines a function called extract_random_number that takes two parameters: list and n.
shuffled_list = Enum.sort_by(list, fn _ -> :rand.uniform() end): This line of code creates a new list called shuffled_list by sorting the original list based on a random number generated for each element in the list using the :rand.uniform() function from the Erlang standard library. The :rand.uniform() function generates a random floating-point number between 0 and 1 for each element in the list, and the Enum.sort_by/2 function sorts the list based on these random numbers.
Enum.take(shuffled_list, n): This line of code extracts the first n elements from the shuffled_list using the Enum.take/2 function from the Elixir standard library. This function takes two arguments: a list and a number n, and returns a new list containing the first n elements of the original list.
The function returns the result of the previous line, which is a new list containing n random numbers extracted from the original list in a random order.

In summary, this function takes a list of numbers, shuffles the list based on randomly generated numbers, and extracts the first n elements from the shuffled list to create a new list containing n random numbers extracted from the original list in a random order.

Minimal Task Write a function that returns the first n elements of the Fibonacci sequence.

```
def first_fibonacci_elements(x) do
for n <- 0..x - 1, do: fib(n)
end

defp fib(0), do: 0
defp fib(1), do: 1
defp fib(n) when n > 1 do
fib(n - 1) + fib(n - 2)
end
```

Here's a breakdown of the function:
def first_fibonacci_elements(x) do: This is a function definition in Elixir. It defines a function called first_fibonacci_elements that takes one parameter x.
for n <- 0..x - 1, do: fib(n): This line of code uses a for comprehension to iterate over a range of numbers from 0 to x 1 (inclusive), and for each value of n, it calls the fib(n) function and returns the result as a list. The for comprehension in Elixir is similar to a loop in other programming languages, but it returns a list of values instead of executing statements.
The first_fibonacci_elements function returns the list of Fibonacci numbers generated by the for comprehension.
defp fib(0), do: 0: This is a private function definition in Elixir. It defines a function called fib that takes one parameter 0 and returns 0.
defp fib(1), do: 1: This is another private function definition that defines the Fibonacci sequence's first two numbers, 0 and 1.
defp fib(n) when n > 1 do fib(n - 1) + fib(n - 2) end: This is a recursive private function definition in Elixir. It defines a function called fib that takes one parameter n and calculates the nth Fibonacci number using recursion. If n is 0, the function returns 0. If n is 1, the function returns 1. Otherwise, it recursively calculates the nth Fibonacci number by adding the two previous Fibonacci numbers together (fib(n - 1) and fib(n - 2)).
In summary, the first_fibonacci_elements function generates the first x Fibonacci numbers using the fib function, which uses recursion to calculate each Fibonacci number. The for comprehension is used to iterate over a range of numbers and call the fib function for each number to generate the list of Fibonacci numbers.

Minimal Task Write a function that, given a dictionary, would translate a sentence. Words not found in the dictionary need not be translated.

```
def translate(dictionary, string) do
words = String.split(string, " ", [trim: true])
translated_words = Enum.map(words, fn word -> dictionary[String.to_atom(word)] ||
    word end)
Enum.join(translated_words, " ")
end
```

The function takes two parameters: a dictionary as a map and a string to translate.
Here's a breakdown of the function:
def translate(dictionary, string) do: This is a function definition in Elixir. It defines a function called translate that takes two parameters: dictionary and string.
words = String.split(string, " ", [trim: true]): This line of code splits the input string string into words using the String.split/3 function from the Elixir standard library. The function splits the

string on spaces and returns a list of words. The [trim: true] option removes any leading or trailing whitespace from each word.
translated_words = Enum.map(words, fn word > dictionary[String.to_atom(word)] || word end): This line of code translates each word in the words list by looking up its translation in the dictionary map. The Enum.map/2 function from the Elixir standard library is used to apply a translation function to each word in the list. The translation function looks up the atom version of the word in the dictionary map using the String.to_atom/1 function and returns the translation if it exists in the map, or the original word if it does not exist.
Enum.join(translated_words, " "): This line of code joins the translated words back together into a single string using the Enum.join/2 function from the Elixir standard library. The function takes a list of words and a separator string (in this case, a space character) and returns a string containing all the words joined by the separator.
The translate function returns the translated string.
In summary, this function takes a dictionary map containing translations for words and a string to translate. It splits the input string into words, translates each word using the dictionary map, and joins the translated words back together into a single string. The function returns the translated string.

Minimal Task Write a function that receives as input three digits and arranges them in an order that would create the smallest possible number. Numbers cannot start with a 0.

```
def smallest_number(a, b, c) do
numbers = [a, b, c]
numbers = Enum.sort(numbers)
first_non_zero = Enum.find(numbers, fn number -> number != 0 end)
numbers = List.delete(numbers, first_non_zero)
numbers = [first_non_zero | numbers]
smallest_num = Enum.join(numbers)
String.to_integer(smallest_num)
end
```

The function takes three parameters, a, b, and c, which represent three integers.
Here's a breakdown of the function:
numbers = [a, b, c]: This line creates a list of three integers.
numbers = Enum.sort(numbers): This line sorts the list of numbers in ascending order using the Enum.sort/1 function from the Elixir standard library.
first_non_zero = Enum.find(numbers, fn number > number != 0 end): This line finds the first non-zero number in the list using the Enum.find/2 function from the Elixir standard library. The fn number -> number != 0 end function is a predicate function that returns true for non-zero numbers and false for zero.
numbers = List.delete(numbers, first_non_zero): This line removes the first non-zero number from the list using the List.delete/2 function from the Elixir standard library.
numbers = [first_non_zero | numbers]: This line prepends the first non-zero number back to the list.
smallest_num = Enum.join(numbers): This line joins the list of numbers into a single string using the Enum.join/2 function from the Elixir standard library.
String.to_integer(smallest_num): This line converts the joined string back to an integer using the String.to_integer/1 function from the Elixir standard library.
In summary, this function takes three integers, sorts them in ascending order, moves the first non-zero number to the beginning of the list, joins the list into a single string, and converts the string back to an integer. The resulting integer is the smallest number that can be formed using the three input integers.

Minimal Task Write a function that would rotate a list n places to the left.

```
def rotate_left(list, number) do
to_rotate = Enum.drop(list, number)
remaining = Enum.take(list, number)
to_rotate ++ remaining
end
```

The function takes two parameters, list and number, where list is a list and number is the number of positions to rotate the list to the left.
Here's a breakdown of the function:
to_rotate = Enum.drop(list, number): This line creates a new list to_rotate that is a copy of the input list, with the first number elements dropped using the Enum.drop/2 function from the Elixir standard library. This creates a list that contains all the elements to be rotated to the left.
remaining = Enum.take(list, number): This line creates a new list remaining that is a copy of the first number elements of the input list using the Enum.take/2 function from the Elixir standard library. This creates a list that contains all the elements that should remain at the beginning of the rotated list.
to_rotate ++ remaining: This line concatenates the two lists to_rotate and remaining using the ++ operator. The result is a new list that contains the original list rotated number positions to the left.
The rotate_left function returns the rotated list.
In summary, this function takes a list and a number, and rotates the list number positions to the left. It does this by creating two new lists: to_rotate, which contains all the elements that should be rotated to the left, and remaining, which contains all the elements that should remain at the beginning of the rotated list. It then concatenates these two lists to create a new list that represents the rotated input list.

Minimal Task Write a function that lists all tuples a, b, c such that $a^2 + b^2 = c^2$ and a, b <= 20.

```
def list_right_angled_triangles do
max_a = 20
max_b = 20
max_c = trunc(:math.sqrt(max_a * max_a + max_b * max_b))
list = for a <- (1..max_a), b <- (1..max_b), c <- (1..max_c), do: {a, b, c}
Enum.filter(list, fn {a, b, c} -> check_p_theorem(a, b, c) end)
end

defp check_p_theorem(a, b, c) do
if c * c == a * a + b * b, do: true, else: false
end
```

The function returns a list of tuples representing all the right-angled triangles whose sides are integers and whose sides are no larger than max_a, max_b, and max_c, respectively.
Here's a breakdown of the function:
max_a = 20, max_b = 20, and max_c = trunc(:math.sqrt(max_a * max_a + max_b * max_b)): These lines set the maximum values for the sides of the right-angled triangles. The value of max_c is calculated using the Pythagorean theorem.
list = for a <- (1..max_a), b <- (1..max_b), c <- (1..max_c), do: a, b, c: This line generates a list of all possible combinations of a, b, and c, where each side is an integer and no larger than max_a, max_b, and max_c, respectively. Each combination is represented as a tuple.
Enum.filter(list, fn a, b, c -> check_p_theorem(a, b, c) end): This line filters the list of tuples to keep only those that satisfy the Pythagorean theorem. The Enum.filter/2 function from the Elixir

standard library is used to iterate over each tuple in the list and check if the tuple represents a right-angled triangle using the check_p_theorem/3 function.
The list_right_angled_triangles function returns the filtered list of tuples representing right-angled triangles.
The check_p_theorem/3 function is a private function that takes three integers a, b, and c as arguments and returns true if the Pythagorean theorem is satisfied (i.e., c * c == a * a + b * b) and false otherwise. It is used as a filter in the Enum.filter/2 function to keep only the right-angled triangles in the list.
In summary, this function generates a list of all possible combinations of the sides of right-angled triangles whose sides are integers and no larger than max_a, max_b, and max_c, respectively, and filters this list to keep only those that satisfy the Pythagorean theorem. The resulting list contains tuples representing all the right-angled triangles satisfying the given criteria.

Main Task Write a function that eliminates consecutive duplicates in a list.

```
def remove_consecutive_duplicates(list) do
Enum.dedup(list)
end
```

The function takes a list as input and removes consecutive duplicate elements from it. Here's a breakdown of the function:
Enum.dedup(list): This line uses the Enum.dedup/1 function from the Elixir standard library to remove consecutive duplicate elements from the list.
The remove_consecutive_duplicates function returns the deduplicated list.
In summary, this function removes consecutive duplicate elements from a list by using the Enum.dedup/1 function from the Elixir standard library.

3 P0W3

Minimal Task Create an actor that prints on the screen any message it receives.

```
def print_message do
receive do
  message -> IO.puts(message)
end
print_message()
end
```

The function listens for incoming messages using the receive construct, prints the message to the console using IO.puts/1, and then calls itself recursively to wait for the next message. Here's a breakdown of the function:
receive do: This line starts a message-receiving block.
message -> IO.puts(message): This line defines a pattern match that matches any message received, binds the received message to the variable message, and then prints the message to the console using IO.puts/1.
end: This line ends the message-receiving block.
print_message(): This line calls the print_message function recursively to wait for the next message.
In summary, this function listens for incoming messages and prints them to the console using IO.puts/1, and then calls itself recursively to wait for the next message. It can be used to create a simple message-processing loop in an Elixir program.

Minimal Task Create an actor that returns any message it receives, while modifying it.

```
def modify_message do
receive do
  message ->
    modified_message = change_message(message)
    IO.puts("Received: #{modified_message}")
end
modify_message()
end

def change_message(message) do
case message do
  int when is_integer(int) -> int + 1
  str when is_binary(str) -> String.downcase(str)
 _ -> "I don't know how to HANDLE this!"
end
end
```

These functions work together to receive messages, modify them, and print out the modified versions. Here's a breakdown of how these functions work:
change_message(message): This function takes a message as input and uses a case statement to match on the type of the message. If the message is an integer, it adds 1 to the value and returns the result. If the message is a string, it converts the string to lowercase and returns the result. If the message is of any other type, it returns the string "I don't know how to HANDLE this!".
modify_message(): This function listens for incoming messages using the receive construct. When a message is received, it calls the change_message function to modify the message and then prints out the modified message to the console using IO.puts/1. Finally, the function calls itself recursively to wait for the next message.
In summary, these functions work together to receive messages, modify them based on their type using the change_message function, and print out the modified versions. They can be used to create a simple message-processing loop in an Elixir program, where different types of messages can be handled in different ways based on their content.

Minimal Task Create a two actors, actor one "monitoring" the other. If the second actor stops, actor one gets notified via a message.

```
defmodule Week3MonitoringActor do
  def run_monitoring_actor do
    IO.puts("The monitoring actor has started.")
    spawn_monitor(Week3MonitoredActor, :run_monitored_actor, [])
    receive do
      {:DOWN, _ref, :process, _from_pid, reason} -> IO.puts("The monitoring actor has
     detected that the monitored actor has stopped. Exit reason: #{reason}.")
      Process.sleep(5000)
    end
    run_monitoring_actor()
  end
end

defmodule Week3MonitoredActor do
    def run_monitored_actor do
    IO.puts("The monitored actor has started.")
    Process.sleep(5000)
    IO.puts("The monitored actor has finished.")
    exit(:crash)
  end
end
```

These functions demonstrate how to use the Process module to monitor a separate process and detect when it stops running.
The Week3MonitoringActor module defines a function run_monitoring_actor that starts by printing a message to the console to indicate that the monitoring actor has started. It then uses the spawn_monitor function to create a new process that runs the Week3MonitoredActor module's run_monitored_actor function. The spawn_monitor function allows the monitoring actor to receive notifications when the monitored actor crashes or stops running.
After starting the monitored actor, the monitoring actor enters a receive block to wait for a message. When it receives a :DOWN, _ref, :process, _from_pid, reason message, it prints a message to the console indicating that the monitored actor has stopped running, along with the reason for its exit. The monitoring actor then waits for 5000 milliseconds (5 seconds) using the Process.sleep function, and then loops back to the beginning of the run_monitoring_actor function to start the process over again.
The Week3MonitoredActor module defines a function run_monitored_actor that prints a message to the console to indicate that the monitored actor has started. It then waits for 5000 milliseconds using the Process.sleep function and prints another message to indicate that it has finished. Finally, it calls the exit function to exit the process with an error condition (:crash).
Overall, these two modules demonstrate how to use Elixir's Process module to monitor other processes and respond to their exit conditions.

Minimal Task Create an actor which receives numbers and with each request prints out the current average.

```
def average(total, count) do
receive do
  number ->
    new_total = total + number
    new_count = count + 1
    average = new_total / new_count
    IO.puts("Current average is: #{average}")
    average(new_total, new_count)
end
end
```

This function calculates and prints out the average of a series of numbers received through the receive construct. Here's a breakdown of how this function works:
average(total, count): This function takes two arguments, total and count, which represent the running total and count of numbers received so far, respectively.
receive do ... end: This construct listens for incoming messages. When a message is received, it executes the code inside the block.
number -> ... end: This pattern matches on the received message and binds its value to the variable number.
new_total = total + number: This line adds the received number to the running total.
new_count = count + 1: This line increments the count of numbers received so far.
average = new_total / new_count: This line calculates the new average by dividing the running total by the count of numbers received so far.
IO.puts("Current average is: #average"): This line prints out the current average to the console.
average(new_total, new_count): This line calls the average function recursively, passing in the updated total and count as arguments. This causes the function to continue listening for new messages and updating the average accordingly.
In summary, this function listens for incoming numbers, updates a running total and count, calculates the current average, and prints it out to the console. This allows you to calculate the average of a series of numbers in real-time as they are received.

4 P0W4

Minimal Task Create a supervised pool of identical worker actors. The number of actors is static, given at initialization. Workers should be individually addressable. Worker actors should echo any message they receive. If an actor dies (by receiving a "kill" message), it should be restarted by the supervisor. Logging is welcome.

```
defmodule Week4WorkingActor do
  def run_working_actor do
    pid = spawn(Week4WorkingActor, :working_actor, [])
    name = for _ <- 1..8, into: "", do: <<Enum.random('abcdefghijklmnopqrstuvwxyz')>>
    Process.register(pid, String.to_atom(name))
    IO.puts("Worker with Id: (#{inspect(pid)}) and Name: (:#{name}) has been created"
    )
    pid
  end

  def working_actor do
    receive do
      :kill ->
        exit(:kill)
      message -> IO.puts("Worker with Id: (#{inspect(self())}) received a message: #{
    inspect(message)}")
    end
    working_actor()
  end
end

defmodule Week4SupervisedPool do
  def run_supervised_pool(n) do
    IO.puts("Supervisor has started")
    processes = Enum.map(1..n, fn _ -> Week4WorkingActor.run_working_actor() end)
    spawn(Week4SupervisedPool, :supervised_pool, [processes])
  end

  def supervised_pool(processes) do
    IO.puts("Workers monitored by the supervisor: #{inspect(processes)}")
    Enum.map(processes, fn process-> Process.monitor(process) end)

    receive do
      {:DOWN, _ref, :process, pid, :kill} -> IO.puts("Worker with Id: (#{inspect(pid)
    }) has finished. Creating and monitoring a new worker")
      new_pid = Week4WorkingActor.run_working_actor()
      processes = List.delete(processes, pid)
      processes = [new_pid] ++ processes
      supervised_pool(processes)
    end
  end
end
```

The Week4WorkingActor module contains two functions, run_working_actor and working_actor. run_working_actor function creates a new process and assigns it a random name composed of 8 lowercase letters using Enum.random and String.to_atom functions. It then registers the process with the given name using Process.register function, and finally returns the process ID.
The working_actor function is the behavior of the worker process. It waits to receive a message, and when it receives one, it prints out the message along with its own process ID. If the message is :kill, the process exits with :kill reason.
The Week4SupervisedPool module has a single function run_supervised_pool, which takes an

argument n and creates n worker processes using the Week4WorkingActor.run_working_actor function. It then spawns a new process to supervise these workers by calling the supervised_pool function.
The supervised_pool function receives messages about worker processes that have exited and creates a new worker process to replace it. It first monitors all the worker processes using Process.monitor, and then waits to receive a message. If a worker process has exited with the reason :kill, the function prints a message and creates a new worker process using Week4WorkingActor.run_working_actor function. It then replaces the old worker process with the new one and continues to monitor the workers by recursively calling supervised_pool with the updated list of worker processes.
In summary, the Week4WorkingActor module defines the behavior of worker processes, while the Week4SupervisedPool module creates and supervises a pool of worker processes. The supervisor process monitors the worker processes and replaces any that have exited with a new worker process.

5 P0W5

Minimal Task Write an application that would visit this link. Print out the HTTP response status code, response headers and response body.

Minimal Task Continue your previous application. 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.

Minimal Task Continue your previous application. Persist the list of quotes into a file. Encode the data into JSON format. Name the file quotes.json.

```elixir
defmodule Week5 do
  def request do
    import HTTPoison
    import Floki

    url = "https://quotes.toscrape.com/"
    response = get!(url)

    IO.puts("Status code: #{inspect(response.status_code)}")
    IO.puts("Headers: #{inspect(response.headers)}")
    IO.puts("Body : #{inspect(response.body)}")

    quotes = parse_document!(response.body)
             |> find(".quote")
             |> Enum.map(fn quote ->
                  text = quote |> find(".text") |> text()
                  author = quote |> find(".author") |> text()
                  tags = quote |> find(".tag") |> Enum.map(&text/1)
                  %{text: text, author: author, tags: tags}
                end)

    IO.puts("Found #{length(quotes)} quotes:")
    Enum.each(quotes, fn quote ->
      IO.puts("#{quote.text}\n - #{quote.author}\n Tags: #{inspect(quote.tags)}\n")
    end)

    save_quotes(quotes)
  end

  defp save_quotes(quotes) do
```

```
    import Jason
    json = encode!(quotes)
    File.write("quotes.json", json)
    IO.puts("Quotes saved in quotes.json")
  end
end
```

The function request uses two external modules: HTTPoison and Floki. The import statements at the beginning of the function allow the use of functions defined in these modules.
The function then sets a URL to scrape, sends an HTTP GET request to that URL, and prints out the response status code, headers, and body. It then uses Floki to parse the HTML document in the response body and extract quotes, authors, and tags.
Next, the function uses Jason to encode the extracted data as JSON and saves it to a file called "quotes.json". Finally, it prints a message to confirm that the quotes have been saved.
Overall, the request function sends an HTTP request to a website, extracts quotes from the HTML response using Floki, encodes the extracted data as JSON using Jason, and saves the JSON data to a file.

6 Conclusion

In conclusion, my experience implementing the task has highlighted the effectiveness of Functional Programming and the Actor Model in building fault-tolerant and highly available systems. Elixir, which is built on top of the Erlang Virtual Machine, offers a robust platform for developing distributed and concurrent applications. By using these tools, developers can create scalable systems that are easy to maintain, test, and understand. As the need for highly concurrent and distributed systems continues to grow, Functional Programming and the Actor Model offer a promising solution for building reliable and scalable systems.