Elixir console application with JSON parsing. Let's print to console!

Wikipedia search with HTTPoison, Poison and Escript

Stephan Bakkelund Valois Follow
Sep 24, 2016 · 10 min read

Hello friends.
I've been playing a bit around with JSON parsing, and extracting information from the received map in Elixir. Figured I'd write about it, and I also thought we could write a very simple console based wikipedia search application together. We are going to use a couple of hex packages in this application:

HTTPoison

"HTTP client for Elixir, based on HTTPotion"
We need a HTTP client to connect with Wikipedia's web API.

Poison

"An incredibly fast, pure Elixir JSON library"
Once we collect JSON data from Wikipedia's web API, we need to parse it.

Escript

"Builds an executable that can be invoked from the command line. An escript can run on any machine that has Erlang installed and by default does not require Elixir to be installed, as Elixir is embedded as part of the escript."
So that we can run our application as a regular command line application.

Alright, so let's jump into it!

As usual, we'll start of by creating a new project. I will call my application "Wiki Dream":

```
~/$ mix new wiki_dream
* creating README.md
* creating .gitignore
* creating mix.exs
```

We'll cd into our application, and add our dependencies to our mix.exs file. Since HTTPoison runs on a different process, we need to add it to our application function, so that it gets started up when we launch our application:

```
# mix.exs
......

def application do
  [applications: [:logger, :httpoison]]
end


defp deps do
  [{:httpoison, "~> 0.9.0"},
  {:poison, "~> 2.0"}]
end
```

And we'll fetch and install our added dependencies:

```
~/wiki_dream$ mix deps.get
Running dependency resolution
Dependency resolution completed
certifi: 0.4.0
hackney: 1.6.1
......
```

Now, let's also set up our folder structure like this:

```
wiki_dream/
- lib/
  — wiki_dream/
```

```
      – cli.ex
    – wiki_dream.ex
```

If you look at the structure above, you'll also notice that I added a new file, cli.ex. Perfect, all set up. Let's get to work.

How our application is going to run:

- Process given argument from our console
- Fetch data from Wikipedia
- Parse fetched data
- Extract the information we want from the data
- Print out the data to our console

Process given argument from our console

Our cli.ex file is going to handle the inputs from our console. If you didn't know already, CLI stands for "Command Line Interface", and is kind of the front-end of console based applications. Since we are building an executable application with escript, our first function is going to be main/1. This is required by escript:

```
# lib/wiki_dream/cli.ex


defmodule WikiDream.CLI do
  def main(args) do
    parse_args(args)
    |> process
  end
end
```

We also add a function, parse_args/1, that we haven't defined yet. This function will call the built in OptionParser module, which contains functions to parse command line options. The process/1 function will contain our applications different steps:

```
# lib/wiki_dream/cli.ex
......

def parse_args(args) do
  parse = OptionParser.parse(args, switches: [help:
:boolean],
                             aliases: [h: :help])


  case parse do
    {[help: true], _, _}
      -> :help
    {_, [search_term], _}
      -> {search_term}
  end
end
```

OptionParser will take whatever argument we give it, and put it into our parse variable. It will also look out for a help or -h flag.
We'll add a case structure, which returns the atom :help, if a help flag is given, or returns our search term as a one-element tuple. Let's see if we can get our console to print something. We'll write a very simple process function just to see if it works:

```
# lib/wiki_dream/cli.ex
......

def process({search_term}) do
  IO.inspect search_term
end
```

Now, to run our application as an executable, we need to build it with escript. To do that, we need to add it to our mix.exs file, and give it our main module (the module where our main/1 function is):

```
# mix.exs
......


def project do
  [app: :wiki_dream,
  version: “0.1.0”,
  elixir: “~> 1.3”,
  build_embedded: Mix.env == :prod,
  start_permanent: Mix.env == :prod,
  escript: [main_module: WikiDream.CLI], #Added escript
  deps: deps()]
end

......
```

In our console, we'll use mix to build it for us:

```
~/wiki_dream$ mix escript.build
===> Compiling idna
===> Compiling mimerl
......
Generated httpoison app
==> wiki_dream
Compiling 2 files (.ex)
Generated wiki_dream app
Generated escript wiki_dream with MIX_ENV=dev
```

Let's try and run it, and give it an argument:

```
~/wiki_dream$ ./wiki_dream hello
“hello”
```

Sweet! it works! Now, what happens if we give it the help flag?

```
~/wiki_dream$ ./wiki_dream —-help
** (FunctionClauseError) no function clause matching in
WikiDream.CLI.process/1
(wiki_dream) lib/wiki_dream/cli.ex:18:
WikiDream.CLI.process(:help)
(elixir) lib/kernel/cli.ex:76: anonymous fn/3 in
Kernel.CLI.exec_fun/2
```

It blows up. As expected, since we haven't written any functions to handle it yet. Let's do that next:

```
# lib/wiki_dream/cli.ex
......

def process(:help) do
  IO.puts “””
  Wiki Dream
  — — — — —
  usage: wiki_dream <search_term>
  example: wiki_dream lion
  “””
end
```

We'll build our executable again, and see if it works:

```
~/wiki_dream$ ./wiki_dream --help
Wiki Dream
— — — — —
usage: wiki_dream <search_term>
example: wiki_dream lion
```

Awesome! We've got the console argument parsing working!

Fetch data from Wikipedia

Now we're going to put HTTPoison to use. We need to access Wikipedia's API and fetch some JSON data. If we're giving "elixir" as an argument, we'll receive the JSON data from Wikipedia's elixir article. Now, Wikipedia won't allow scraping articles from their webpage, so getting information from https://en.wikipedia.org/wiki/Elixir won't work. However, if go through their API,

https://en.wikipedia.org/w/api.php?format=json&action=query&prop=extracts&exintro=&explaintext=&titles=elixir

Wikipedia will give us a JSON file with everything we need. We'll create a new file and module, json_fetch.ex, which will handle fetching and parsing the JSON received. First, we'll remove IO.inspect from process/1 and make it call a fetch/1 function which we haven't written yet:

```
# lib/wiki_dream/cli.ex
......

def process({search_term}) do
  WikiDream.JSONFetch.fetch(search_term)
end
......
```

And in our freshly new fetch_json.ex file:

```
# lib/wiki_dream/fetch_json.ex

defmodule WikiDream.JSONFetch do
  def fetch(search_term) do
    wiki_url(search_term)
    |> HTTPoison.get
    |> IO.inspect
  end

  defp wiki_url(search_term) do
    “https://en.wikipedia.org/w/api.php?
format=json&action=query&prop=extracts&exintro=&explaintext
=&titles= #{search_term}"
  end
end
```

To make our fetch/1 function less noisy, I've added a private function with the Wikipedia API URL. Notice the string interpolation in the URL.

We then pipe our URL with the search term through HTTPoison. I've added an IO.inspect so that we can see—in our terminal—the output. Build a new escript and fire up our application:

```
~/wiki_dream$ ./wiki_dream elixir
{:ok, %HTTPoison.Response{body:
“{\”batchcomplete\”:\”\”,\”query\”:{\”normalized\”:
[{\”from\”:\”elixir\”,\”to\”:\”Elixir\”}],\”pages\”:
{\”457424\”:
{\”pageid\”:457424,\”ns\”:0,\”title\”:\”Elixir\”,\”extract\
”:\”An elixir (from Arabic:
\\u0627\\u0644\\u0625\\u06...............
```

Nice, we received noise! Well, if you look at the output, you'll discover a tuple, that returned :ok (Which is good!), and a whole bunch of other data. (Body and header). If we look in the body, we can see the string from the Wikipedia article on Elixir (a liquid, not our language).

Parse fetched data

We are going to use the second dependency we installed, Poison, to handle and extract the body of our fetched data:

```
# lib/wiki_dream/json_fetch.ex
......

def handle_json({:ok, %{status_code: 200, body: body}}) do
  {:ok, Poison.Parser.parse!(body)}
end
def handle_json({_, %{status_code: _, body: body}}) do
  IO.puts "Something went wrong. Please check your internet
           connection"
end


......
```

So, we have two handle_json/1 functions, one that handles a tuple with the atom :ok, and a map with status_code: 200, and another one which handles everything else. Our first handle_json/1 function will return a tuple with :ok, and a map with the body. The other one will return our error.

Let's add our handle_json/1 to our fetch/1 function:

```
# lib/wiki_dream/json_fetch.ex


def fetch(search_term) do
  wiki_url(search_term)
  |> HTTPoison.get
  |> handle_json
  |> IO.inspect
end
```

```
......
```

Build the escript, and run the application:

```
~/wiki_dream$ ./wiki_dream elixir
{:ok,
%{“batchcomplete” => “”,
“query” => %{“normalized” => [%{“from” => “elixir”, “to” =>
“Elixir”}],
“pages” => %{“457424” => %{“extract” => “An elixir (from
Arabic: الإكسير – al-’iksīr) is a clear, sweet-flavored
liquid used for medicinal purposes, to be taken orally and
intended to cure one’s illness. When used as a
pharmaceutical preparation, an elixir contains at least one
active ingredient designed to be taken orally.”,“ns” => 0,
“pageid” => 457424, “title” => “Elixir”}}}}}
```

Sweet. Everything went as expected. We received a tuple with :ok and a map with the body. Now, go ahead and delete the line in fetch/1, IO.inspect, as we don’t need it anymore.

Extract the information we want from the data

Now, if you take a look at the received data, you’ll notice that the information we are looking for is the value of the key “extract”. We need to make our way through this map and fetch “extract”s value.

We’ll start of by creating a new file, extract_map.ex, which will contain the function we’ll use as a digging tool:

```
# lib/wiki_dream/extract_map.ex
```

```
defmodule WikiDream.ExtractMap do
  def extract_from_body(map) do
    {:ok, body} = map
    IO.inspect body
  end
end
```

Here, we’re making some good use of pattern matching. As mentioned before, our data is now a tuple with an :ok atom, and a map. We pattern match on both the tuple, and the body, and store the body in a variable named body. We’ll add an IO.inspect to see the data output.
We also need to add this function to our process/1 function:

```
# lib/wiki_dream/cli.ex
......

def process({search_term}) do
  WikiDream.JSONFetch.fetch(search_term)
  |> WikiDream.ExtractMap.extract_from_body
end

......
```

We’ll build our app and run it:

```
~/wiki_dream$ ./wiki_dream elixir
%{"batchcomplete" => "",
"query" => %{"normalized" => [%{"from" => "elixir", "to" =>
"Elixir"}],
"pages" => %{"457424" => %{"extract" => "An elixir (from
Arabic: الإكسير - al-'iksīr) is a clear, sweet-flavored
liquid used for medicinal purposes, to be taken orally and
intended to cure one's illness. When used as a
pharmaceutical preparation, an elixir contains at least one
active ingredient designed to be taken orally.", "ns" => 0,
"pageid" => 457424, "title" => "Elixir"}}}}
```

Cool. The tuple is gone, since we asked to only inspect the map inside it. Now, if you run the application with different arguments, you’ll notice that every key except the page number (string of integers) is static, fixed, don’t change. We can use a built in Elixir function to do some heavy lifting for us, the get_in/2.

> ***get_in(data, keys)***
> Gets a value from a nested structure

/ Elixir documentation

We need to dig our way into "query", "pages", "page number" and "extract". The first two keys should be easy since they never change. Let's give it a shot:

```
# lib/wiki_dream/extract_map.ex
......


def extract_from_body(map) do
  {:ok, body} = map


  extract_article = get_in(body, ["query"])
  |> get_in(["pages"])
  IO.inspect extract_article
end


......
```

So we get the the value from the "query" key, which gives us access to the "pages" key. Let's see what's inside the "pages" key:

```
~/wiki_dream$ ./wiki_dream elixir
%{“457424” => %{“extract” => “An elixir (from Arabic:
الإكسير – al-’iksīr) is a clear, sweet-flavored liquid used
for medicinal purposes, to be taken orally and intended to
cure one’s illness. When used as a pharmaceutical
preparation, an elixir contains at least one active
ingredient designed to be taken orally.”,
“ns” => 0, “pageid” => 457424, “title” => “Elixir”}}
```

Nice going. We dug our way to the page number. We are getting closer and closer to the "extract" key. Now, our next objective is slightly more tricky than the previous ones. The key name will change depending on the article's page number. However, the key name will always be an integer, so we can write a function for that.

Enum.find(enumerable, default \\ nil, fun)
Returns the first item for which fun returns a truthy value. If no such

> item is found, returns default
> / Elixir documentation

We'll use Enum.find to return the first key which is an integer. Since there's only one key, and that key is an integer, we should be able to get one step closer:

```
# lib/wiki_dream/extract_map.ex


def extract_from_body(map) do
  {:ok, body} = map

  extract_article = get_in(body, ["query"])
  |> get_in(["pages"])
  |> Enum.find(fn {key, _value} ->
       case Integer.parse(key) do
         :error -> false
         _ -> key
       end
     end)
  |> IO.inspect
end
```

> ***Integer.parse(binary, base \\ 10)***
> Parses a text representation of an integer
> / Elixir documentation

So we use Integer.parse to find a key with a text representation of an integer. If it exists, we return the key, if not, we return false.

If we run our program, we will receive a tuple with the page number as the first element, and the rest of the map as the second element. We don't really care about the page number any longer, so we'll do some more pattern matching to get the "extract" key. We'll then use Map.fetch!/2 to fetch the value from the "extract" key:

> ***Map.fetch!(map, key)***
> Fetches the value for specific key
> / Elixir documentation

```
# lib/wiki_dream/extract_map.ex

def extract_from_body(map) do
  {:ok, body} = map


  extract_article = get_in(body, [“query”])
  |> get_in([“pages”])
  |> Enum.find(fn {key, _value} ->
       case Integer.parse(key) do
         :error -> false
         _ -> key
       end
     end)

  {_, extract_article_content} = extract_article
  Map.fetch!(extract_article_content, “extract”)
end
```

Print out the data to our console

We have come a long way since we started, my friend. We are now searching for data, receiving data, parsing data, and extracting information from the data. Now, we need to print the data out to the console. Let's start with an IO.inspect to see what we got:

```
# lib/wiki_dream/cli.ex
......


def process({search_term}) do
  WikiDream.JSONFetch.fetch(search_term)
  |> WikiDream.ExtractMap.extract_from_body
  |> IO.inspect
end

......
```

Build and run:

```
~/wiki_dream$ ./wiki_dream elixir
“An elixir (from Arabic: الإكسير — al-’iksīr) is a clear,
sweet-flavored liquid used for medicinal purposes, to be
taken orally and intended to cure one’s illness. When used
as a pharmaceutical preparation, an elixir contains at
least one active ingredient designed to be taken orally.”
```

We are outputting a string of the article content. Which is what we wanted! Since it’s a string, we don’t really need to use IO.inspect anymore. We can use IO.puts to output the string. Let’s also put on some very light string formatting, a line break at every period:

> ***String.replace(subject, pattern, replacement, options** \\ [])*
> Returns a new string created by replacing occurrences of pattern in subject with replacement
> / Elixir documentation

```
# lib/wiki_dream/cli.ex
......

def process({search_term}) do
  WikiDream.JSONFetch.fetch(search_term)
  |> WikiDream.ExtractMap.extract_from_body
  |> string_format
end


......

def string_format(string) do
  String.replace(string, “. “, “. \n”)
  |> IO.puts
end


......
```

Build and run:

```
~/wiki_dream$ ./wiki_dream elixir
An elixir (from Arabic: الإكسير – al-’iksīr) is a clear,
sweet-flavored liquid used for medicinal purposes, to be
taken orally and intended to cure one’s illness.
When used as a pharmaceutical preparation, an elixir
```

```
  contains at least one active ingredient designed to be
  taken orally.
```

And that's it. A pretty useless JSON-parsing, map-digging CLI application!

There is of course other string formatting that could be added to make the application look better, however that's beyond the scope of this article.

There are probably easier, or different ways of doing this. However, this is the way I found worked out well for me. I also think it clearly shows how functional programming works. where different functions modifies the data, one by one, until you have the desired result. We didn't change the original data we started out with, we just took it, copied it, modified it along the way, and ended up with something completely different.

The cool thing about an Escript executable, is that the entire application with all it's dependencies and codes are within this single file. If you move it outside the application folder and run it, it still works. You can even run it on a different computer, as long as the computer has Erlang installed.

That's all for now.

Until next time
Stephan Bakkelund Valois