Command Line Application with Elixir

Pierre Sugar
pierre@sugaryourcoffee.de

January 6, 2015

Abstract

The programming world gets parallel because of processors with multiple cores. Most of the programming languages aren't build for multiprocessing and hence cannot use the performance that comes with the multi core processors. Erlang is one of the most sophisticated language build especially for multiprocessing. But Erlang has some get to used rules and is not so much fun to program with. This is where Elixir comes into play. Elixir combines the advantages of Erlang with the elegance of Ruby.

This tutorial is introducing into Elixir by developing a command line interface based on a suggestion from Dave Thomas' Elixir book.

1 Installation

`Elixir` is run on the `Erlang` virtual machine. That is we need `Elixir` and `Erlang` installed. The most up to date installation instructions for `Elixir` can be found at `http://elixir-lang.org/install.html` and for `Erlang` at `https://www.erlang-solutions.com/downloads/download-erlang-otp`.

To install `Erlang` we issue following commands from the command line

```
\$ wget http://package.erlang-solutions.com/erlang-solutions_1.0_all.deb
\$ sudo dpkg -i erlang-solutions_1.0_all.deb
\$ wget http://package.erlang-solutions.com/ubuntu/erlang_solutions.asc
\$ sudo apt-key add erlang_solutions erlang_solutions.asc
\$ sudo apt-get update
\$ sudo apt-get install erlang
```

Next we install `Elixir`

```
\$ wget http://packages.erlang-solutions._1.0_all.deb
\$ sudo dpkg -i erlang-solutions_1.0_all.deb
\$ sudo apt-get update
\$ sudo apt-get install elixir
```

Now we are ready to go.

2 Define the Project

The `National Climatic Data Center` or short `NOAA` is providing web services to retrieve climatic data from cities of the American continent. We want to retrieve this data with our application and display it on the console.

2.1 Project outline

The program runs from the command line. We will provide a location and the program will print the wheather data.

- Provide location $ noaa ¡location¿
- Parse the input
- Fetch the data from NOAA
- Extract the wheather data
- Print the results

But before we jump into the application development we need to understand the web service from `NOAA` to know what we actually have to provide as parameters.

2.2 Understanding the NOAA Web Service

To use the web service from `NOAA` we need to request a token. With each web service request we have to provide that token.

To request a token go to `https://www.ncdc.noaa.gov/cdo-web/token` and enter and submit your e-mail address. You will receive a token send to the provided e-mail address. This token you allways have to provide when sending a web service request to `NOAA`.

Let's check out the web service with `curl`. Information about how to use the web service can be found at `https://www.ncdc.noaa.gov/cdo-web/webservices/v2`. The `-H` switch includes an extra header with the token to the HTTP `get` request.

To know wich locations are available to retrieve wheather data for we can retrieve the data with the `/locations` url.

```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
        http://www.ncdc.noaa.gov/cdo-web/api/v2/locations
{"results":[
 {"id":"CITY:AE000002",
  "name":"Ajman, AE",
  "datacoverage":0.6855,
  "mindate":"1944-03-01",
  "maxdate":"2014-12-29"},
 {"id":"CITY:AE000003",
  "name":"Dubai, AE",
  "datacoverage":0.6855,
  "mindate":"1944-03-01",
  "maxdate":"2014-12-29"},
  ...
 {"id":"CITY:AR000010",
  "name":"Mendoza, AR",
  "datacoverage":0.997,
  "mindate":"1959-10-01",
```

```
  "maxdate":"2014-12-29"},
 {"id":"CITY:AR000011",
  "name":"Neuquen, AR",
  "datacoverage":0.9814,
  "mindate":"1956-08-01",
  "maxdate":"2014-12-29"}],
 "metadata":{"resultset":{"limit":25,"count":38497,"offset":1}}}
```


As we can see the data is provided in a JSON format with `"results"` and `"metadata"` fields. But it seems there are loads of locations. That is our application also has to provided a way to list all available location data.

The data is provided in different datasets. Dependent the information we want to retrieve we have to know the datasets `datatypeid`. We can obtain the information with the `/datasets` url.

```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
        http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets
{"results":[
  {"uid":"gov.noaa.ncdc:C00040","id":"ANNUAL","name":"Annual Summaries",
   "datacoverage":1,"mindate":"1831-02-01","maxdate":"2014-07-01"},
  {"uid":"gov.noaa.ncdc:C00861","id":"GHCND","name":"Daily Summaries",
   "datacoverage":1,"mindate":"1763-01-01","maxdate":"2014-12-31"},
  {"uid":"gov.noaa.ncdc:C00841","id":"GHCNDMS","name":"Monthly Summaries",
   "datacoverage":1,"mindate":"1763-01-01","maxdate":"2014-11-01"},
  {"uid":"gov.noaa.ncdc:C00345","id":"NEXRAD2",
   "name":"Weather Radar (Level II)",
   "datacoverage":0.95,"mindate":"1991-06-05","maxdate":"2014-12-31"},
  {"uid":"gov.noaa.ncdc:C00708","id":"NEXRAD3",
   "name":"Weather Radar (Level III)",
   "datacoverage":0.95,"mindate":"1994-05-20","maxdate":"2014-12-28"},
  {"uid":"gov.noaa.ncdc:C00821","id":"NORMAL_ANN",
   "name":"Normals Annual/Seasonal",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-01-01"},
  {"uid":"gov.noaa.ncdc:C00823","id":"NORMAL_DLY","name":"Normals Daily",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-31"},
  {"uid":"gov.noaa.ncdc:C00824","id":"NORMAL_HLY","name":"Normals Hourly",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-31"},
  {"uid":"gov.noaa.ncdc:C00822","id":"NORMAL_MLY","name":"Normals Monthly",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-01"},
  {"uid":"gov.noaa.ncdc:C00505","id":"PRECIP_15",
   "name":"Precipitation 15 Minute",
   "datacoverage":0.25,"mindate":"1970-05-12","maxdate":"2013-07-01"},
  {"uid":"gov.noaa.ncdc:C00313","id":"PRECIP_HLY",
   "name":"Precipitation Hourly",
   "datacoverage":1,"mindate":"1900-01-01","maxdate":"2013-10-01"}],
 "metadata":{"resultset":{"limit":25,"count":11,"offset":1}}}
```

This information we have to additionally provide to the location information for that we want to retrieve data from. Hence we also have to provide a way to retrieve possible datasets with our application.

From the `location` information we can obtain the min and max date weather data is available for each country. From the `dataset` information we can obtain which datasets are available. Based on that information we call the `data` url to retrieve the weather data.

And finally to retrieve actual weather data we invoke the `/data` url with the `datasetid` GHCND and the `locationid` CITY:000019 for Munich.

```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
        http://www.ncdc.noaa.gov/cdo-web/api/v2/data?datasetid=GHCND&\
        locationid=CITY:GM000019&startdate=2014-10-01&enddate=2014-10-31
{"results":[
  {"station":"GHCND:GM000004199","value":14,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":193,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":120,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":3,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":191,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":104,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":195,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":105,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":184,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":73,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":182,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":89,"attributes":",,E,",
```

```
 "datatype":"TMIN","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
 "datatype":"PRCP","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
 "datatype":"SNWD","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":184,"attributes":",,E,",
 "datatype":"TMAX","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":67,"attributes":",,E,",
 "datatype":"TMIN","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
 "datatype":"PRCP","date":"2014-10-04T00:00:00"}],
"metadata":{"resultset":{"limit":25,"count":248,"offset":1}}}
```


Now we have all information we need to build our command line interface which will, from the current information state, provide these functions.

- List a specified count of available `datasets`
- List all or selected count of available `locations`
- Search for a specific city and print the `locationid`
- Get the wheather data for a specific location base on the `locationid`

The invocation variants are listed in the table 1 on page 5.

Table 1: Command line interface

Command	**Description**
noaa datasets -c 10	Lists 10 datasets
noaa datasets	Lists all available datasets
noaa locations -c 10	Lists 10 locations
noaa locations	List all available locations
noaa locations -s Munich	Searches of the Vancouver `locationid`
noaa data -l CITY:GM000019 -d GHCND -f 2014-10-01 -t 2014-10-31	Print weather data for Munich (-l) for the specified dataset (-d) between 1st (-f) and 31st (-t) of October

3 Create the Project

`Elixir` comes with the tool `Mix` to create and manage `Elixir` projects.

To create a project we issue the `Mix` command `new`. But first we change directory where we want to have created our project tree.

```
\$ cd ~/Learn/Elixir/noaa
\$ mix new noaa
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
```

```
* creating lib/noaa.ex
* creating test
* creating test/test_helper.exs
* creating test/noaa_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

  cd noaa
  mix test

Run 'mix help' for more commands.

\$
```

Now we have a fully implemented project tree and we can right away run our first test.

```
\$ cd noaa
\$ mix test
Compiled lib/noaa.ex
Generated noaa.app
.

Finished in 0.07 seconds (0.07s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 963329
\$
```

But before we actually start we want to set up version control for `noaa`.

```
\$ git init
\$ git add .
\$ git commit -am "Initial commit of noaa"
```

4 Implement the Command Line Interface

Let's try behaviour driven development (BDD) with `Elixir`. `Elixir` comes with a test environment called `EXUnit`. With BDD we start with tests first and then implement the code. So let's try that.

Listing 1: test/cli_test.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
  end
```

```
  test ":datasets returned with default count when only dataset is given" do
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
  end

end
```

This is a bare test file with only a description we want to test. Let's run it with

```
\$ mix test
** (CompileError) test/cli_test.exs:4: module Noaa.CLI is not loaded and could
not be found
    (stdlib) lists.erl:1352: :lists.mapfoldl/3
    (stdlib) lists.erl:1353: :lists.mapfoldl/3
```

No surprise so far as we don't have an implementation of `Noaa.CLI` yet. So the next step is to get it compiled. To do this we have to implement `Noaa.CLI` next. As per convention the main source code of an application is saved into `lib/project_name`. Our proeject's name is `noaa`, hence our source code directory is named `lib/noaa`. Also a each module is saved to a separate file and each module is name spaced with the project name. So our first module for the command line interface (CLI) is called `noaa.CLI`. Table 2 on page 8 shows a summary of the conventions of `Elixir` projects.

Listing 2: lib/noaa/cli.ex

```
defmodule Noaa.CLI do
```

Table 2: Conventions for Elixir Projects

Command	Description
`lib/noaa/`	Directory for main source code
`noaa.CLI`	Each module is name spaced with the project name
	Each module lives in an own file

```
@default_count 10

@moduledoc """
Handle the command line parsing and dispatching to the respective functions
that list the weather conditions of provided cities.
"""

def run(argv) do
  parse_args(argv)
end

@doc """
`argv` can be one of the following options.

* datasets  --count    COUNT
* locations --count    COUNT
* locations --search   CITY
* data      --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])
```

```
  end

end
```

With `cli.ex` in place we run the test again.

```
\$ mix test
Compiled lib/noaa/cli.ex
Generated noaa.app
test/cli_test.exs:4: warning: unused import Noaa.CLI
...........

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
11 tests, 0 failures

Randomized with seed 289389
```

This looks good our tests ran successfully without errors - but actually this is still boring. But now we get into the excitement. We now add our first actual test for `:help`.

Listing 3: test/cli_test1.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
```

```
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
  end

end
```

Note that we didn't implement the function in `lib/noaa/cli.ex`. This is why our test at that moment fails when we run `mix test`.

Listing 4: mix test

```
.....

  1) test :help returned by option parsing with -h and --help options (CliTest)
     test/cli_test.exs:6
     Assertion with == failed
     code: parse_args(["--help", "anything"]) == :help
     lhs:  {[help: true], ["anything"], []}
     rhs:  :help
     stacktrace:
       test/cli_test.exs:7

.....

Finished in 0.1 seconds (0.1s on load, 0.01s on tests)
11 tests, 1 failures

Randomized with seed 305907
```

To make the test pass we will implement the parser section for `:help`.

Listing 5: lib/noaa/cli2.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    parse_args(argv)
  end

  @doc """
  `argv` can be one of the following options.
```

```
* datasets  --count   COUNT
* locations --count   COUNT
* locations --search  CITY
* data      --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    { [ data: true,
        dataset: dataset,
        location: location,
        from: from,
        to: to ], _, _ }
      -> { :data, dataset, location, from, to }
    _ -> :help
  end
end
```

```
end
```

When we run the test again we will see all tests pass.

Listing 6: mix test

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

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
11 tests, 0 failures

Randomized with seed 962325
```

Now we implement the rest of the tests.

Listing 7: test/cli2_test

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
```

```
  end

end
```

When we run the tests they should all fail except the one for `:help`.

Listing 8: mix test

```
.

  1) test :datasets returned with default count when only dataset is given (Cli
     test/cli_test.exs:11
     ** (CaseClauseError) no case clause matching: {[datasets: true], [], []}
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:12

.

  2) test :data returned with -s, -d, -b and -e if all required data is given (
     test/cli_test.exs:35
     ** (CaseClauseError) no case clause matching: {[data: true, dataset: "ABCD
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:36



  3) test :datasets returned with count if dataset and count is given (CliTest)
     test/cli_test.exs:15
     ** (CaseClauseError) no case clause matching: {[datasets: true, count: 5],
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:16



  4) test :location returned with default count if only location is given (CliT
     test/cli_test.exs:20
     ** (CaseClauseError) no case clause matching: {[locations: true], [], []}
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:21



  5) test :location returned with city location and city is given (CliTest)
     test/cli_test.exs:29
     ** (CaseClauseError) no case clause matching: {[locations: true, location:
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
```

```
       test/cli_test.exs:30


  6) test :location returned with count if location and count is given (CliTest
     test/cli_test.exs:24
     ** (CaseClauseError) no case clause matching: {[locations: true, count: 8]
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:25



Finished in 0.3 seconds (0.3s on load, 0.05s on tests)
8 tests, 6 failures

Randomized with seed 569703
```

To make them pass we implement the rest of the command line interface in `lib/cli.ex`

Listing 9: lib/cli3.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
  * locations --search  CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
```

```
                                switches: [ help:       :boolean,
                                            datasets:   :boolean,
                                            locations:  :boolean,
                                            data:       :boolean,
                                            count:      :integer,
                                            search:     :string,
                                            dataset:    :string,
                                            location:   :string,
                                            from:       :string,
                                            to:         :string ],
                                aliases:  [ h:          :help,
                                            c:          :count,
                                            s:          :search,
                                            d:          :dataset,
                                            l:          :location,
                                            f:          :from,
                                            t:          :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, dataset, location, from, to }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
```

```
    noaa fetches weather data from NOAA

    usage: noaa command args

    noaa --datasets  [ --count [ count | #{@default_count} ] ]
    noaa --locations [ --count [ count | #{@default_count} ] ] | --search city ]
    noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                     --to YYYY-MM-DD
    """
    System.halt(0)
  end

  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

And we see them all pass. There is a specialty regarding the `data` command. This has more than one switch. Currently the switches have to be provided in a specific order. This is not user friendly. We should only make sure that all switches are provided. We write a test where we provide the switches in an arbitrary sequence.

Listing 10: test/cli_test3.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
    assert parse_args(["--datasets"]) == { :dataset, 10 }
  end
```

```
  test ":datasets returned with count if dataset and count is given" do
    assert parse_args(["--datasets", "--count", "5"]) == { :dataset, 5 }
    assert parse_args(["--datasets", "-c",      "5"]) == { :dataset, 5 }
  end

  test ":location returned with default count if only location is given" do
    assert parse_args(["--locations"]) == { :location, 10 }
  end

  test ":location returned with count if location and count is given" do
    assert parse_args(["--locations", "--count", "8"]) == { :location, 8 }
    assert parse_args(["--locations", "-c",      "8"]) == { :location, 8 }
  end

  test ":location returned with city location and city is given" do
    assert parse_args(["--locations",
                       "--location",
                       "Munich"]      ) == { :location, "Munich" }
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
    assert parse_args(["--data",
                       "--dataset",  "ABCD",
                       "--location", "Munich",
                       "--from",     "2014-12-24",
                       "--to",       "2015-01-01"]) == { :data,
                                                         "ABCD",
                                                         "Munich",
                                                         "2014-12-24",
                                                         "2015-01-01" }
  end

  test ":data returned with values provided in arbitrary sequence" do
    assert parse_args(["--data",
                       "--location", "Munich",
                       "--from",     "2014-12-24",
                       "--to",       "2015-01-01",
                       "--dataset",  "ABCD"]      ) == { :data,
                                                         "ABCD",
                                                         "Munich",
                                                         "2014-12-24",
                                                         "2015-01-01" }
  end

end
```

If we run the test we see it fail.

Listing 11: mix test

```
Compiled lib/noaa/cli.ex
```

```
Generated noaa.app
........

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
8 tests, 0 failures

Randomized with seed 126754
```

To make it pass we have to refactor our code where we do the parsing. We replace the `_` and the `:data` parts with a function `parse_remains` that is trying to match the command line arguments that haven't been matched by the case statements before.

Listing 12: lib/cli4.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
  * locations --search  CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
                                           count:     :integer,
                                           search:    :string,
                                           dataset:   :string,
                                           location:  :string,
```

```
                                        from:       :string,
                                        to:         :string ],
                                aliases:  [ h:          :help,
                                            c:          :count,
                                            s:          :search,
                                            d:          :dataset,
                                            l:          :location,
                                            f:          :from,
                                            t:          :to ])

    case parse do
      { [ help: true ], _, _ }
        -> :help
      { [ datasets: true, count: count ], _, _ }
        -> { :dataset, count }
      { [ datasets: true ], _, _ }
        -> { :dataset, @default_count }
      { [ locations: true, count: count ], _, _ }
        -> { :location, count }
      { [ locations: true ], _, _ }
        -> { :location, @default_count }
      { [ locations: true, location: location ], _, _ }
        -> { :location, location }
      _ -> parse_remains(parse)
    end
  end

  def parse_remains([ data: true,
                      dataset: dataset,
                      location: location,
                      from: from,
                      to: to ]), do: { :data, dataset, location, from, to }

  def parse_remains({ parse, _, _ }) do
    parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                           fn(x) -> List.keyfind(parse, x, 0) end))
  end

  def parse_remains(_), do: :help

  def process(:help) do
    IO.puts """
    noaa fetches weather data from NOAA

    usage: noaa command args

    noaa --datasets  [ --count [ count | #{@default_count} ] ]
    noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
    noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                     --to YYYY-MM-DD
```

```
    """
    System.halt(0)
  end

  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

If we run the test again we see it pass.

Listing 13: mix test

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

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
13 tests, 0 failures

Randomized with seed 773652
```

Next we fetch the data from `NOAA`.

5 Process the Parsed Command Line Data

The next step is to process the parsed command line data. So we add a `process` function to our `run` function in the `Noaa.CLI` module. But before we do that we write the test.

Listing 14: test/cli_process_test.exs

```
defmodule CliProcessTest do
  use ExUnit.Case
  import Noaa.CLI, only: [ process: 1 ]

  test "process :datasets to fetch datasets" do
    result = Noaa.CLI.process({:datasets, 10})

    assert result == "10 datasets"
  end
```

```
  test "process :locations to fetch locations" do
    result = Noaa.CLI.process({:locations, 10})

    assert result == "10 locations"
  end

  test "process :locations to search for a city" do
    result = Noaa.CLI.process({:locations, "Munich"})

    assert result == "Munich location"
  end

  test "process :data to fetch data for a specified city" do
    result = Noaa.CLI.process({:data,
                               ["GHCND",
                               "CITY:GM000019",
                               "2014-10-01",
                               "2015-01-01"]})

    assert result == "Data for specified city"
  end
end
```

When we run the test we see it fail. To make the test pass we implement the `process` function with that much functionality in the `Noaa.CLI` module to make it pass. It is not the final solution but it makes the test pass and it shows that the process functions are invoked.

Listing 15: lib/noaa/cli5.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
  * locations --search  CITY
```

```
* data       --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, dataset, location, from, to }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
```

```
                        fn(x) -> List.keyfind(parse, x, 0) end))
  end

  def parse_remains(_), do: :help

  def process(:help) do
    IO.puts """
    noaa fetches weather data from NOAA

    usage: noaa command args

    noaa --datasets  [ --count [ count | #{@default_count} ] ]
    noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
    noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                     --to YYYY-MM-DD
    """
    System.halt(0)
  end

  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

When we run the test again we see it pass. We cannot test the `process(:help)` function because it calls `System.halt(0)` and this will cancel the `mix test`. So we test if `:help` is processed by running our function.

Listing 16: Running the function with mix

```
noaa fetches weather data from NOAA

usage: noaa command args

noaa --datasets  [ --count [ count | 10 ] ]
noaa --locations [ --count [ count | 10 ] | --search city ]
noaa --data      --dataset dataset --location location --from YYYY-MM-DD
--to YYYY-MM-DD
```

We see the nicely formatted `:help` message displayed. We can also run the other commands. Let's do that.

Listing 17: Running :data with mix

```
Compiled lib/noaa/cli.ex
Generated noaa.app
```

Of course we don't see any output yet but at least we see not error message. Now let's move on to implement the functionality to fetch the weather data from `NOAA`.

`NOAA` is delivering data over a web service in the `JSON` format. To fetch and parse the data we will use external libraries from `http://hex.pm` which is similar to `https://rubygems.org` from where you can install external libraries to your proect.

5.1 Installing external Libraries

We will use (on advise of Dave Thomas) `HTTPoison` as the HTTP client library and `jsx` as the `JSON` library.

To install the libraries we have to add them to `mix.exs` in the `deps` section.

Listing 18: mix.exs

```
defmodule Noaa.Mixfile do
  use Mix.Project

  def project do
    [app: :noaa,
     version: "0.0.1",
     elixir: "~> 1.0",
     deps: deps]
  end

  # Configuration for the OTP application
  #
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    [
```

```
      { :httpoison, "~> 0.5.0" },
      { :jsx,       "~> 2.4.0" }
    ]
  end
end
```

`$ mix` is managing the dependencies for us. We can issue `$ mix deps` to list the dependencies and their status. To download the dependencies issue `$ mix deps.get`. To actually install the libraries issue `$ mix deps` again.

Actual package management is done by `Hex`. If it is not installed yet you will be asked whether to install `Hex` when issuing `$ mix deps`. With `$ mix local` you can list available `Hex` tasks.

Now do the installation. First we list the dependencies and their status and on the way install `Hex` if it not installed yet.

`$ mix deps`

Listing 19: mix deps

```
* jsx (Hex package)
  the dependency is not available, run `mix deps.get`
* httpoison (Hex package)
  the dependency is not available, run `mix deps.get`
```

Then downlaod the dependencies

`$ mix deps.get`

Listing 20: mix deps.get

```
Running dependency resolution
Unlocked:  httpoison, jsx
Dependency resolution completed successfully
  httpoison: v0.5.0
  idna: v1.0.1
  jsx: v2.4.0
  hackney: v0.14.3
* Getting httpoison (Hex package)
Checking package (https://s3.amazonaws.com/s3.hex.pm/tarballs/httpoison-0.5.0.t
Using locally cached package
Unpacked package tarball (/home/pierre/.hex/packages/httpoison-0.5.0.tar)
* Getting jsx (Hex package)
Checking package (https://s3.amazonaws.com/s3.hex.pm/tarballs/jsx-2.4.0.tar)
Using locally cached package
Unpacked package tarball (/home/pierre/.hex/packages/jsx-2.4.0.tar)
* Getting hackney (Hex package)
Checking package (https://s3.amazonaws.com/s3.hex.pm/tarballs/hackney-0.14.3.ta
Using locally cached package
Unpacked package tarball (/home/pierre/.hex/packages/hackney-0.14.3.tar)
* Getting idna (Hex package)
Checking package (https://s3.amazonaws.com/s3.hex.pm/tarballs/idna-1.0.1.tar)
Using locally cached package
Unpacked package tarball (/home/pierre/.hex/packages/idna-1.0.1.tar)
```

`$ mix deps`

Then finally install the libraries

Listing 21: mix deps

```
* idna (Hex package)
  locked at 1.0.1 (idna)
  the dependency build is outdated, please run `mix deps.compile`
* jsx (Hex package)
  locked at 2.4.0 (jsx)
  the dependency build is outdated, please run `mix deps.compile`
* hackney (Hex package)
  locked at 0.14.3 (hackney)
  the dependency build is outdated, please run `mix deps.compile`
* httpoison (Hex package)
  locked at 0.5.0 (httpoison)
  the dependency build is outdated, please run `mix deps.compile`
```

If you get the information that the packages are outdated (not compiled) then follow the hint and issue `mix deps.compile`, even though the next time your project will be compiled the dependencies also get compiled. But we will do that right away.

```
$ mix deps.compile
```

Listing 22: mix deps.compile

```
==> idna (compile)
Compiled src/idna.erl
Compiled src/punycode.erl
Compiled src/idna_unicode.erl
Compiled src/idna_ucs.erl
Compiled src/idna_unicode_data.erl
==> jsx
Compiled src/jsx.erl
Compiled src/jsx_to_term.erl
Compiled src/jsx_config.erl
Compiled src/jsx_decoder.erl
Compiled src/jsx_to_json.erl
Compiled src/jsx_encoder.erl
Compiled src/jsx_verify.erl
Compiled src/jsx_parser.erl
Generated jsx.app
==> hackney (compile)
Compiled src/hackney_connect/hackney_pool_handler.erl
Compiled src/hackney_connect/hackney_tcp_transport.erl
Compiled src/hackney_connect/hackney_ssl_transport.erl
Compiled src/hackney_connect/hackney_http_connect.erl
Compiled src/hackney_connect/hackney_connect.erl
Compiled src/hackney_connect/hackney_socks5.erl
Compiled src/hackney_connect/hackney_pool.erl
Compiled src/hackney_lib/hackney_cookie.erl
Compiled src/hackney_lib/hackney_date.erl
Compiled src/hackney_lib/hackney_headers.erl
Compiled src/hackney_lib/hackney_bstr.erl
Compiled src/hackney_lib/hackney_multipart.erl
```

```
Compiled src/hackney_lib/hackney_http.erl
Compiled src/hackney_lib/hackney_url.erl
Compiled src/hackney_client/hackney_stream.erl
Compiled src/hackney_client/hackney_manager.erl
Compiled src/hackney_client/hackney_request.erl
Compiled src/hackney_client/hackney_util.erl
Compiled src/hackney_client/hackney_response.erl
Compiled src/hackney_client/hackney_idna.erl
Compiled src/hackney_app/hackney_deps.erl
Compiled src/hackney_app/hackney_sup.erl
Compiled src/hackney_app/hackney_app.erl
Compiled src/hackney_client/hackney.erl
Compiled src/hackney_lib/hackney_mimetypes.erl
==> httpoison
Compiled lib/httpoison/base.ex
Compiled lib/httpoison.ex
Generated httpoison.app
```

If you look at your project tree you will see a new directory `deps`. There you will find all the dependencies you have just compiled.

Now we are good to go.

5.2 Fetching Data from NOAA

Except for the `:help` function we only have dummy implementations. We now want to implement `:datasets`, `:locations` and `:data`. We will host these functions in the `Noaa.Webservice` module.

We first write the test with a stub implementation of `Noaa.Webservice` in place.

Listing 23: lib/noaa/webservice.ex

```
defmodule Webservice do
  @token [ { "token", "xLCcTVopsdphuEvPyOyhkAbVlObmWQra" } ]

end
```

Listing 24: test/test_webservice.exs

```
defmodule Webservice do
  @token [ { "token", "xLCcTVopsdphuEvPyOyhkAbVlObmWQra" } ]

end
```

To get `HTTPoison` to work it has to be started as a separate application. Actually we don't have to do that manually we rather add `HTTPoison` to `mix.exs` in the `application` section.

Listing 25: mix.exs with HTTPoison

```
defmodule Noaa.Mixfile do
  use Mix.Project

  def project do
```

```
    [app: :noaa,
     version: "0.0.1",
     elixir: "~> 1.0",
     deps: deps]
  end

  # Configuration for the OTP application
  #
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger, :httpoison]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    [
      { :httpoison, "~> 0.5.0" },
      { :jsx,       "~> 2.4.0" }
    ]
  end
end
```

When we run the test it will fail. We then implement `Noaa.Webservice` and run the test again.

Listing 26: list/noaa/webservice.ex

```
defmodule Noaa.Webservice do
  @token [ { "token", "xLCcTVopsdphuEvPyOyhkAbVlObmWQra" } ]

  def fetch(url) do
    url
    |> HTTPoison.get(@token)
    |> handle_response
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 200,
                                                body: body}}) do
    { :ok, body }
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 404}}) do
    { :error, [{"message", "Page not found"}] }
```

```
  end

  def handle_response({:error, %HTTPoison.Error{reason: reason}}) do
    {:error, reason}
  end


end
```

And when we run the test now it will be executed without errors. Actually we are only checking for the `:ok` response from the web service and ignore the body. If we get the data right we will see when we do the printing of the results. What we test here that we successfully can obtain data over the web service. Actually this isn't a good idea to access the web service in tests. For one accesses are limited per day and two if we are running a lot of tests, what we actually should, it might put heavy load to the server. If we are done we can skip these tests. But for now (NOAA forgive me) we will use them.

Currently `Noaa:Webservice.fetch` is returning the raw data from `NOAA`. But what we want is the data in a way we can handle them more easily. This is the next step in our process chain. We are converting the data to an internal representation.

6 Converting the Response Data

The response from `NOAA` is in `JSON` format. To convert it to a data structure we use the previously installed `jsx` library. In the `Noaa.Webservice` we convert the data with `jsx`.

Listing 27: lib/noaa/webservice.ex

```
defmodule Noaa.Webservice do
  @token [ { "token", "xLCcTVopsdphuEvPyOyhkAbVlObmWQra" } ]

  def fetch(url) do
    url
    |> HTTPoison.get(@token)
    |> handle_response
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 200,
                                                body: body}}) do
    {:ok, :jsx.decode(body)}
  end

  def handle_response({:ok, %HTTPoison.Response{status_code: 404}}) do
    {:error, [{"message", "Page not found"}]}
  end

  def handle_response({:error, %HTTPoison.Error{reason: reason}}) do
    {:error, :jsx.decode(reason)}
  end
```

```
end
```

Our test should still pass, as we are not checking for the body, but only for the `:ok` response.

If we look the converted data it looks like in 28 on page 30.

Listing 28: JSON format of webservice response

```
{:ok,
[{"results",
  [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
    {"name", "Annual Summaries"}, {"datacoverage", 1},
    {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
 {"metadata",
  [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]}
```

We can use our application in `iex` by starting `iex` with the `$ mix -S mix` switch.

```
\$ iex -S mix
Erlang/OTP 17 [erts-6.3] [source] [64-bit] [smp:4:4] [async-threads:10]
[kernel-poll:false]
Compiled lib/noaa.ex
Compiled lib/noaa/webservice.ex
lib/noaa/cli.ex:105: warning: variable city is unused
Compiled lib/noaa/cli.ex
Generated noaa.app
Interactive Elixir (1.0.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
iex(1)> ds = Noaa.Webservice.fetch("http://www.ncdc.noaa.gov/cdo-web/api/v2/data
sets?limit=1")
{:ok,
 [{"results",
   [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
     {"name", "Annual Summaries"}, {"datacoverage", 1},
     {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
  {"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]}
```

We acutally want to work with the `"results"` and the `"metadata"`. We can destructure the `tuple` with a pattern match.

```
iex(2)> { _ , body } = ds
{:ok,
 [{"results",
   [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
     {"name", "Annual Summaries"}, {"datacoverage", 1},
     {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
  {"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]}
iex(3)> body
[{"results",
```

```
  [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
    {"name", "Annual Summaries"}, {"datacoverage", 1},
    {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
 {"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]
```

To access the `"results"` and the `"metadata"` with pattern matching or with `List.keyfind`. First we destructure the body with pattern matching.

```
iex(52)> [ results, metadata ] = body
[{"results",
  [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
    {"name", "Annual Summaries"}, {"datacoverage", 1},
    {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
 {"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]
iex(53)> results
{"results",
 [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
   {"name", "Annual Summaries"}, {"datacoverage", 1}, {"mindate", "1831-02-01"},
   {"maxdate", "2014-07-01"}]]}
iex(54)> metadata
{"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}
```

We do the same now with `List.keyfind`.

```
iex(4)> results = List.keyfind(body, "results", 0)
{"results",
 [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
   {"name", "Annual Summaries"}, {"datacoverage", 1}, {"mindate", "1831-02-01"},
   {"maxdate", "2014-07-01"}]]}
iex(5)> meta = List.keyfind(body, "metadata", 0)
{"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}
```

The pattern matching is a conciser. We can destructure the body into the `"results"` and `"metadata"` in one swoop as we need two commands with `List.keyfind`.

This is not yet the data we need. What we want to do is iterate or recurse through the data of `"results"` and `"metadata"`'s `resultset`. That is we just need the data of these two and then put it into a collection. To retrieve the data we will now do all in one swoop with pattern matching. Note that the `resultset` in nested in the `metadata`, so we have also a nested pattern to destructure the data.

```
iex(6)> [ {r, rd}, { m, [ { rs, rsd } ] } ] = body
[{"results",
  [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
    {"name", "Annual Summaries"}, {"datacoverage", 1},
    {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
 {"metadata", [{"resultset", [{"limit", 1}, {"count", 11}, {"offset", 1}]}]}]
iex(7)> r
"results"
iex(8)> rd
```

```
[[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
  {"name", "Annual Summaries"}, {"datacoverage", 1}, {"mindate", "1831-02-01"},
  {"maxdate", "2014-07-01"}]]
iex(9)> m
"metadata"
iex(92)> rs
"resultset"
iex(93)> rsd
[{"limit", 1}, {"count", 11}, {"offset", 1}]
```

I think that is quite cool how `Elixir` can destructure a collection. Now both the `"results"` data and the `"metadata"` data we want to put into a `HashDict` so we can access the values with `results["id"]` or just use comprehensions. To put the data into a `HashDict` we use `Enum.map` and `Enum.into`.

```
iex(97)> result = rd |> Enum.map(&Enum.into(&1, HashDict.new))
[#HashDict<[{"name", "Annual Summaries"}, {"maxdate", "2014-07-01"},
  {"id", "ANNUAL"}, {"mindate", "1831-02-01"}, {"uid", "gov.noaa.ncdc:C00040"},
  {"datacoverage", 1}]>]
iex(98)> resultset = rsd |> Enum.into(HashDict.new)
#HashDict<[{"limit", 1}, {"count", 11}, {"offset", 1}]>
```

Now we can access the data with the `Dict` interface.

```
iex(112)> resultset
#HashDict<[{"limit", 1}, {"count", 11}, {"offset", 1}]>
iex(113)> HashDict.keys h
["name", "maxdate", "id", "mindate", "uid", "datacoverage"]
iex(114)> HashDict.keys resultset
["limit", "count", "offset"]
iex(115)> resultset["count"]
11
iex(116)> [ h | t ] = result
[#HashDict<[{"name", "Annual Summaries"}, {"maxdate", "2014-07-01"},
  {"id", "ANNUAL"}, {"mindate", "1831-02-01"}, {"uid", "gov.noaa.ncdc:C00040"},
  {"datacoverage", 1}]>]
iex(117)> HashDict.keys h
["name", "maxdate", "id", "mindate", "uid", "datacoverage"]
iex(118)> h["maxdate"]
"2014-07-01"
iex(119)> HashDict.get(h, "mindate")
"1831-02-01"
```

We now now how to retrieve the data. Back in the `Noaa.CLI` module we will decode the repsonse accordingly in the `process` functions.

We will first write a test but again we cannot test for error response when we issue a `System.halt` what we will do when the response returns an error. Therefore we test only the success response `:ok`.

Listing 29: test/decode_test.exs

```
defmodule DecodeTest do
```

```
  use ExUnit.Case

  import Noaa.CLI, only: [ decode_response: 1 ]

  def test_body do
    [{"results",
      [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
        {"name", "Annual Summaries"}, {"datacoverage", 1},
        {"mindate", "1831-02-01"}, {"maxdate", "2014-07-01"}]]},
     {"metadata", [{"resultset", [{"limit", 1}, {"count", 11},
                   {"offset", 1}]}]}]
  end

  def test_results do
    [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
      {"name", "Annual Summaries"}, {"datacoverage", 1},
      {"mindate", "1831-02-01"},
      {"maxdate", "2014-07-01"}]]
  end

  def test_metadata do
    [{"limit", 1}, {"count", 11}, {"offset", 1}]
  end

  test "decode for successful response" do
    assert decode_response({:ok, test_body}) == [ test_results, test_metadata ]
  end

end
```

The implementation of the `decode_response` follows.

Listing 30: lib/noaa/cli.ex

```
defmodule Noaa.CLI do

  @default_count 10
  @max_count 1000

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
```

```
`argv` can be one of the following options.

* datasets  --count   COUNT
* locations --count   COUNT
* locations --search  CITY
* data      --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
```

```
                from: from,
                to: to ]), do: { :data, [dataset:  dataset,
                                         location: location,
                                         from:     from,
                                         to:       to] }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                    fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA at http://www.ncdc.noaa.gov

  usage: noaa command args

  noaa --datasets  [ --count [ count | #{@default_count} ] ]
  noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
  noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                   --to YYYY-MM-DD
  """
  System.halt(0)
end

def process({:datasets, count}) do
  datasets_url(count)
  |> Noaa.Webservice.fetch
  |> decode_response
end

def process({:locations, count}) when is_integer(count) do
  locations_url(count)
  |> Noaa.Webservice.fetch
  |> decode_response
end

def process({:locations, city}) do
  locations_url(@max_count)
end

def process({:data, values}) do
  data_url(values)
  |> Noaa.Webservice.fetch
  |> decode_response
end

def datasets_url(count) do
```

```
    "http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets?limit=#{count}"
  end

  def locations_url(count) do
    "http://www.ncdc.noaa.gov/cdo-web/api/v2/locations?limit=#{count}"
  end

  def data_url(values) do
    """
    http://www.ncdc.noaa.gov/cdo-web/api/v2/data?\
    datasetid=#{values[:dataset]}&\
    locationid=#{values[:location]}&\
    startdate=#{values[:from]}&\
    enddate=#{values[:to]}\
    """
  end

  def decode_response({:ok, body}) do
    [ { _, results }, { _, [ { _, metadata } ] } ] = body
    [ results, metadata ]
  end

  def decode_response({:error, reason}) do
    {_, message} = List.keyfind(reason, "message", 0)
    IO.puts "Error fetching data from NOAA: #{message}"
    System.halt(2)
  end

end
```

And when we run the test it will pass.

Next we want to transform the data `results` and `metadata` into a `HashDict`. As we do that we have seen in ...

Again we start with a test.

Listing 31: test/transform_test.exs

```
defmodule TransformTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ transform_to_hashdicts: 1 ]

  def test_results do
    [[{"uid", "gov.noaa.ncdc:C00040"}, {"id", "ANNUAL"},
      {"name", "Annual Summaries"}, {"datacoverage", 1},
      {"mindate", "1831-02-01"},
      {"maxdate", "2014-07-01"}]]
  end

  def test_metadata do
    [{"limit", 1}, {"count", 11}, {"offset", 1}]
```

```
  end

  test "transformation of results and metadata into HashDict" do
    [ results, metadata ] = transform_to_hashdicts([test_results,
                                                    test_metadata])
    assert is_list results
    assert is_list metadata
  end

end
```

And here the respective implementation of `convert_to_list_of_hashdicts`.

Listing 32: lib/cli7.ex

```
defmodule Noaa.CLI do

  @default_count 10
  @max_count 1000

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be used with one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
  * locations --search  CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
                                           count:     :integer,
                                           search:    :string,
                                           dataset:   :string,
```

```
                                        location:  :string,
                                        from:      :string,
                                        to:        :string ],
                                aliases:  [ h:         :help,
                                            c:         :count,
                                            s:         :search,
                                            d:         :dataset,
                                            l:         :location,
                                            f:         :from,
                                            t:         :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, [dataset:  dataset,
                                             location: location,
                                             from:     from,
                                             to:       to] }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA at http://www.ncdc.noaa.gov

  usage: noaa command args
```

```
  noaa --datasets  [ --count [ count | #{@default_count} ] ]
  noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
  noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                   --to YYYY-MM-DD
  """
  System.halt(0)
end

def process({:datasets, count}) do
  print(datasets_url(count), "Datasets",
        [ "name", "id", "mindate", "maxdate", "datacoverage" ])
end

def process({:locations, count}) when is_integer(count) do
  print(locations_url(count), "Locations",
        [ "name", "id", "mindate", "maxdate", "datacoverage" ])
end

def process({:locations, city}) do
  locations_url(@max_count)
end

def process({:data, values}) do
  print(data_url(values), "Weather Data",
        [ "datatype", "value", "station", "attributes", "date" ])
end

def datasets_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets?limit=#{count}"
end

def locations_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/locations?limit=#{count}"
end

def data_url(values) do
  """
  http://www.ncdc.noaa.gov/cdo-web/api/v2/data?\
  datasetid=#{values[:dataset]}&\
  locationid=#{values[:location]}&\
  startdate=#{values[:from]}&\
  enddate=#{values[:to]}\
  """
end

def decode_response({:ok, body}) do
  [ { _, results }, { _, [ { _, metadata } ] } ] = body
  [ results, metadata ]
end
```

```
  def decode_response({:error, reason}) do
    {_, message} = List.keyfind(reason, "message", 0)
    IO.puts "Error fetching data from NOAA: #{message}"
    System.halt(2)
  end

  def transform_to_hashdicts([ results | metadata ]) do
    [ results  |> Enum.map(&Enum.into(&1, HashDict.new)),
      metadata |> Enum.map(&Enum.into(&1, HashDict.new)) ]
  end

  def print(url, title, columns) do
    [results, metadata] = url
    |> Noaa.Webservice.fetch
    |> decode_response
    |> transform_to_hashdicts

    IO.puts(IO.ANSI.format(["\n", :blue, :bright, title], true))
    Noaa.TableFormatter.print(results, columns)
    IO.puts(IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true))
    Noaa.TableFormatter.print(metadata, [ "limit", "count", "offset" ])
   end
end
```

The next step in our implementation is to print the data to the console.

7 Print the Results

Printing the data is the actual result of our application. We will format the data as a table. To nicely format it we have to determine the max length of each value to have each column the same size. We also want to choose which values to show dependent on the command, that is `datasets`, `locations` and `data`. All of these have metadata that is printed with each of the data.

7.1 Design the metadata Table

The `metadata` gives information about the data retrieved from `NOAA`. The `metadata` is shown in table 3.

Table 3: Data of `metadata`

Data	Example
"limit"	"1"
"count"	"11"
"offset"	"1"

7.2 Design the datasets Table

The `datasets` describes the type of data that can be obtained with the `data` webservice. The `datasets` contains the data in table 4.

Table 4: Data of `datasets`

Data	Example
"name"	"Annual Summaries"
"id"	"ANNUAL"
"mindate"	"1831-02-01"
"maxdate"	"2014-07-01"
"datacoverage"	1

7.3 Design the locations Table

The `locations` is the location for which the weather data can be obtained from. Table 5 shows the layout.

Table 5: Data of `locations`

Data	Example
"name"	"Ajman, AE"
"id"	"CITY:AE000002"
"mindate"	"1944-03-01"
"maxdate"	"2014-12-30"
"datacoverage"	0.6855

7.4 Design the data Table

This is actually the weather data we want to display for a location. Table 6 shows the design of the `data` table.

Table 6: Data of `data`

Data	Example
"datatype"	"PRCP"
"value"	14
"station"	"GHCND:GM00004199"
"attributes"	",,E,"
"date"	"2014-10-01"

7.5 Implementation of TableFormatter

Based on the design of the table formats of the different data types we are ready for the implementation. We provide the fields we want to display to the `TableFormatter`. The `TableFormatter` looks for the largest fields in each column and creates the column layout accordingly.

As usual test first, we write the test for the `TableFormatter`.

Listing 33: test/table_formatter_test.exs

```
defmodule TableFormatterTest do
  use ExUnit.Case
  import ExUnit.CaptureIO
```

```
  alias Noaa.TableFormatter, as: TF

  def test_data do
    [ [ c11111: "c1",  c2: "c12", c3: "c1345", c4: "c1456", c5: "c15678"
],
      [ c11111: "c2",  c2: "c22", c3: "c234",  c4: "c2456", c5: "c25678"
],
      [ c11111: "c3",  c2: "c32", c3: "c334",  c4: "c3456", c5: "c356789" ],
      [ c11111: 3.56,  c2: "c42", c3: "c434",  c4: "c4456", c5: "c45678"
],
      [ c11111: "c56", c2: "c52", c3: "c534",  c4: "c5456", c5: "c55678"
] ]
  end

  def test_headers, do: [ :c11111, :c3, :c5 ]

  test "Extract columns" do
    columns = TF.extract_columns(test_data, test_headers)
    assert List.first(columns) == [ "c1", "c2", "c3", "3.56", "c56" ]
    assert List.last(columns)  == [ "c15678", "c25678", "c356789", "c45678",
                                    "c55678" ]
  end

  test "column width" do
    widths = TF.max_column_widths(TF.extract_columns(test_data, test_headers),
                                  test_headers)
    assert widths == [6, 5, 7]
  end

  test "formatter string" do
    assert TF.formatter_string([6, 5, 7], "|") == "~-6s|~-5s|~-7s~n"
  end

  test "print table header" do
    result = capture_io fn ->
      TF.print_header(test_headers, TF.formatter_string([6, 5, 7], " | "))
      TF.print_horizontal_line({ "-", "-+-" }, [6, 5, 7])
    end

    assert result == """
    c11111 | c3    | c5
    -------+-------+-------
    """
  end

  test "formatted table" do
    columns = TF.extract_columns(test_data, test_headers)
    formatter = TF.formatter_string([6, 5, 7], " | ")
    result = capture_io fn ->
      TF.print_header(test_headers, formatter)
```

```
    TF.print_horizontal_line({ "-", "-+-" }, [6, 5, 7])
    TF.print_table(columns, formatter)
  end

  assert result == """
  c11111 | c3     | c5
  -------+--------+--------
  c1     | c1345  | c15678
  c2     | c234   | c25678
  c3     | c334   | c356789
  3.56   | c434   | c45678
  c56    | c534   | c55678
  """
 end

end
```

Now to the implememtation of the `TableFormatter` module.

Listing 34: lib/noaa/table_formatter.ex

```
defmodule Noaa.TableFormatter do

  def print(rows, header) do
    columns   = extract_columns(rows, header)
    widths    = max_column_widths(columns, header)
    formatter = formatter_string(widths, " | ")
    print_header(header, formatter)
    print_horizontal_line({ "-", "-+-" }, widths)
    print_table(columns, formatter)
  end

  def extract_columns(data, header) do
    for h <- header do
      for d <- data, do: to_string d[h]
    end
  end

  def max_column_widths(rows, header) do
    row_column_widths = rows
    |> Enum.map(&Enum.max_by(&1, fn(x) -> String.length(x) end))
    |> String.length)

    header_column_widths = header
    |> Enum.map(&to_string/1)
    |> Enum.map(&String.length/1)

    List.zip([row_column_widths, header_column_widths])
    |> Enum.map(&Tuple.to_list/1)
    |> Enum.map(&Enum.max/1)
  end
```

```
  def formatter_string(column_widths, separator) do
    (column_widths
    |> Enum.map(&("~-#{&1}s"))
    |> Enum.join(separator)) <> "~n"
  end

  def print_table(data, formatter) do
    data
    |> List.zip
    |> Enum.map(&Tuple.to_list/1)
    |> Enum.each(fn(row) -> :io.fwrite(formatter, row) end)
  end

  def print_header(header, formatter) do
    :io.fwrite(formatter, header)
  end

  def print_horizontal_line({ line, separator }, widths) do
    widths
    |> Enum.map(&String.duplicate(line, &1))
    |> Enum.join(separator)
    |> IO.puts
  end

end
```

Now we have all pieces available and can finalize the application. We update the `process` functions in `lib/noaa/cli.ex` after we have written the respective test as shown in 35.

Listing 35: test/cli_process_test

```
defmodule CliProcessTest do
  use ExUnit.Case
  import ExUnit.CaptureIO
  import Noaa.CLI, only: [ process: 1 ]

  test "process :datasets to fetch datasets" do
    result = capture_io fn -> process({:datasets, 1}) end

    assert result == """
    #{IO.ANSI.format(["\n", :blue, :bright, "Datasets"], true)}
    name              | id      | mindate    | maxdate    | datacoverage
    ------------------+---------+------------+------------+-------------
    Annual Summaries  | ANNUAL  | 1831-02-01 | 2014-07-01 | 1
    #{IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true)}
    limit | count | offset
    ------+-------+-------
    1     | 11    | 1
    """
```

```
end

test "process :locations to fetch locations" do
  result = capture_io fn -> process({:locations, 1}) end

  assert result == """
  #{IO.ANSI.format(["\n", :blue, :bright, "Locations"], true)}
  name       | id              | mindate    | maxdate    | datacoverage
  -----------+-----------------+------------+------------+-------------
  Ajman, AE  | CITY:AE000002   | 1944-03-01 | 2014-12-30 | 0.6859
  #{IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true)}
  limit | count | offset
  ------+-------+-------
  1     | 38497 | 1
  """
end

test "process :locations to search for a city" do
  result = process({:locations, "Munich"})

  assert result == "Munich location"
end

test "process :data to fetch data for a specified city" do
  result = capture_io fn ->
                        process({:data,
                                 [dataset: "GHCND",
                                  location: "CITY:GM000019",
                                  from: "2014-10-01",
                                  to: "2014-10-01"]})
                      end

  assert result == """
  #{IO.ANSI.format(["\n", :blue, :bright, "Weather Data"], true)}
  datatype | value | station            | attributes | date
  ---------+-------+--------------------+------------+--------------------
  PRCP     | 14    | GHCND:GM000004199  | ,,E,       | 2014-10-01T00:00:00
  SNWD     | 0     | GHCND:GM000004199  | ,,E,       | 2014-10-01T00:00:00
  TMAX     | 193   | GHCND:GM000004199  | ,,E,       | 2014-10-01T00:00:00
  TMIN     | 120   | GHCND:GM000004199  | ,,E,       | 2014-10-01T00:00:00
  PRCP     | 3     | GHCND:GME00111524  | ,,E,       | 2014-10-01T00:00:00
  SNWD     | 0     | GHCND:GME00111524  | ,,E,       | 2014-10-01T00:00:00
  TMAX     | 191   | GHCND:GME00111524  | ,,E,       | 2014-10-01T00:00:00
  TMIN     | 104   | GHCND:GME00111524  | ,,E,       | 2014-10-01T00:00:00
  #{IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true)}
  limit | count | offset
  ------+-------+-------
  25    | 8     | 1
  """
end
```

```
end
```

Then we do the final implementation in `lib/cli.ex`.

Listing 36: lib/noaa/cli.ex

```
defmodule Noaa.CLI do

  @default_count 10
  @max_count 1000

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be used with one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
  * locations --search  CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
                                           count:     :integer,
                                           search:    :string,
                                           dataset:   :string,
                                           location:  :string,
                                           from:      :string,
                                           to:        :string ],
                               aliases:  [ h:         :help,
                                           c:         :count,
                                           s:         :search,
                                           d:         :dataset,
                                           l:         :location,
                                           f:         :from,
```

```
                                        t:          :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, [dataset:  dataset,
                                             location: location,
                                             from:     from,
                                             to:       to] }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA at http://www.ncdc.noaa.gov

  usage: noaa command args

  noaa --datasets  [ --count [ count | #{@default_count} ] ]
  noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
  noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                   --to YYYY-MM-DD
  """
  System.halt(0)
end

def process({:datasets, count}) do
```

```
  print(datasets_url(count), "Datasets",
        [ "name", "id", "mindate", "maxdate", "datacoverage" ])
end

def process({:locations, count}) when is_integer(count) do
  print(locations_url(count), "Locations",
        [ "name", "id", "mindate", "maxdate", "datacoverage" ])
end

def process({:locations, city}) do
  locations_url(@max_count)
end

def process({:data, values}) do
  print(data_url(values), "Weather Data",
        [ "datatype", "value", "station", "attributes", "date" ])
end

def datasets_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets?limit=#{count}"
end

def locations_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/locations?limit=#{count}"
end

def data_url(values) do
  """
  http://www.ncdc.noaa.gov/cdo-web/api/v2/data?\
  datasetid=#{values[:dataset]}&\
  locationid=#{values[:location]}&\
  startdate=#{values[:from]}&\
  enddate=#{values[:to]}\
  """
end

def decode_response({:ok, body}) do
  [ { _, results }, { _, [ { _, metadata } ] } ] = body
  [ results, metadata ]
end

def decode_response({:error, reason}) do
  {_, message} = List.keyfind(reason, "message", 0)
  IO.puts "Error fetching data from NOAA: #{message}"
  System.halt(2)
end

def transform_to_hashdicts([ results | metadata ]) do
  [ results  |> Enum.map(&Enum.into(&1, HashDict.new)),
    metadata |> Enum.map(&Enum.into(&1, HashDict.new)) ]
```

```
  end

  def print(url, title, columns) do
    [results, metadata] = url
    |> Noaa.Webservice.fetch
    |> decode_response
    |> transform_to_hashdicts

    IO.puts(IO.ANSI.format(["\n", :blue, :bright, title], true))
    Noaa.TableFormatter.print(results, columns)
    IO.puts(IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true))
    Noaa.TableFormatter.print(metadata, [ "limit", "count", "offset" ])
   end
end
```

When we run the test they should all pass. Except for one test for that we didn't do the implemenation. `process(:locations, city)` we haven't implemented yet. We also should extend our interface by the `limit` switch. Currently when invoking `:datasets` or `:locations` we always start at the first record. This is actually not usefull. We cover those two pending topics (maybe) later.

We are ready to go with our nicely implemented application. We can test it from the commandline or within `iex`. Lets test it in `iex`.

```
\$ iex -S mix
iex(1)> Noaa.CLI.process({:datasets, 12})
Datasets
name                        | id          | mindate    | maxdate    | datacoverage
----------------------------+-------------+------------+------------+-------------
Annual Summaries            | ANNUAL      | 1831-02-01 | 2014-07-01 | 1
Daily Summaries             | GHCND       | 1763-01-01 | 2014-12-31 | 1
Monthly Summaries           | GHCNDMS     | 1763-01-01 | 2014-12-01 | 1
Weather Radar (Level II)    | NEXRAD2     | 1991-06-05 | 2015-01-04 | 0.95
Weather Radar (Level III)   | NEXRAD3     | 1994-05-20 | 2015-01-01 | 0.95
Normals Annual/Seasonal     | NORMAL_ANN  | 2010-01-01 | 2010-01-01 | 1
Normals Daily               | NORMAL_DLY  | 2010-01-01 | 2010-12-31 | 1
Normals Hourly              | NORMAL_HLY  | 2010-01-01 | 2010-12-31 | 1
Normals Monthly             | NORMAL_MLY  | 2010-01-01 | 2010-12-01 | 1
Precipitation 15 Minute     | PRECIP_15   | 1970-05-12 | 2013-07-01 | 0.25
Precipitation Hourly        | PRECIP_HLY  | 1900-01-01 | 2013-10-01 | 1

Metadata
limit | count | offset
------+-------+-------
12    | 11    | 1
iex(2)> Noaa.CLI.process({:locations, 10})

Locations
name             | id             | mindate    | maxdate    | datacoverage
-----------------+----------------+------------+------------+-------------
Ajman, AE        | CITY:AE000002  | 1944-03-01 | 2014-12-30 | 0.6859
Dubai, AE        | CITY:AE000003  | 1944-03-01 | 2014-12-30 | 0.6859
```

```
Sharjah, AE     | CITY:AE000006 | 1944-03-01 | 2014-12-30 | 0.6859
Algiers, AG     | CITY:AG000001 | 1877-04-01 | 2014-12-30 | 1
Annaba, AG      | CITY:AG000002 | 1909-11-01 | 1937-12-31 | 0.9527
Bejaia, AG      | CITY:AG000005 | 1909-11-01 | 1938-12-29 | 0.9596
Constantine, AG | CITY:AG000006 | 1880-05-01 | 1938-12-30 | 0.8736
Laghouat, AG    | CITY:AG000008 | 1888-01-01 | 1938-12-30 | 0.9036
Tamanrasset, AG | CITY:AG000016 | 1940-01-01 | 2014-02-18 | 0.9989
Baku, AJ        | CITY:AJ000001 | 1881-07-01 | 1992-01-31 | 0.991

Metadata
limit | count | offset
------+-------+-------
10    | 38497 | 1

iex(3)> Noaa.CLI.process({:data, [dataset: "GHCND", \
        location: "CITY:AE000002", from: "2014-10-01", to: "2014-10-01"]})

Weather Data
datatype | value | station           | attributes | date
---------+-------+-------------------+------------+--------------------
PRCP     | 0     | GHCND:AE000041196 | ,,S,       | 2014-10-01T00:00:00
TMAX     | 381   | GHCND:AE000041196 | ,,S,       | 2014-10-01T00:00:00
TMIN     | 285   | GHCND:AE000041196 | ,,S,       | 2014-10-01T00:00:00

Metadata
limit | count | offset
------+-------+-------
25    | 3     | 1
```

8 Make the Command-Line Executable

What we want to do is call our application from the command line without the `Elixir` command. To enable this for our application we have to add `[ main_module: Noaa.CLI ]` to `mix.exs` like shown in listing 37.

Listing 37: mix.exs

```
defmodule Noaa.Mixfile do
  use Mix.Project

  def project do
    [app: :noaa,
     version: "0.0.1",
     elixir: "~> 1.0",
     escript: escript_config,
     deps: deps]
  end

  # Configuration for the OTP application
  #
```

```
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger, :httpoison]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    [
      { :httpoison, "~> 0.5.0" },
      { :jsx,       "~> 2.4.0" }
    ]
  end

  defp escript_config do
    [ main_module: Noaa.CLI ]
  end
end
```

`escript` is `Erlang` utility that can run `Zip` archives that contain pre-compiled applications. `escript` requires a `main` function in the `module` depicted in the `escript` directive in `mix.exs`. The `main` function in `Noaa.CLI` is actuall the `run` function. So the only thing we have to do is to rename `run` to `main`.

Listing 38: lib/noaa/cli.ex

```
defmodule Noaa.CLI do

  @default_count 10
  @max_count 1000

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def main(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be used with one of the following options.
```

```
* datasets  --count   COUNT
* locations --count   COUNT
* locations --search  CITY
* data      --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :datasets, count }
    { [ datasets: true ], _, _ }
      -> { :datasets, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :locations, count }
    { [ locations: true ], _, _ }
      -> { :locations, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :locations, location }
    _ -> parse_remains(parse)
  end
end

@doc """
Command line parameters can be given in an arbitrary sequence. We check the
switches given for completeness and order them in the required sequence. Then
we kick them off again whether they match a given command. In this case we
```

```
check for the :data command.
"""
def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, [dataset:  dataset,
                                             location: location,
                                             from:     from,
                                             to:       to] }

@doc """
All commands and their parameters are not matched within argsv/1 are checked
in the parse_remains/1 functions and try to put the command and switches
given in the right order. The we kick them off again whether it matches a
command.
"""
def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

@doc """
If none of the other commands match, then :help is invoked
"""
def parse_remains(_), do: :help

@doc """
Prints the help message if the command line parameters do not match a command
Exits the application after printing the help message.

## Example
    iex> Noaa.CLI.process(:help)
"""
def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA at http://www.ncdc.noaa.gov

  usage: noaa command args

  noaa --datasets  [ --count [ count | #{@default_count} ] ]
  noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
  noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                   --to YYYY-MM-DD
  """
  System.halt(0)
end

@doc """
Processes the datasets command and prints the available datasets limitted by
```

```
the count parameter.

## Example
    iex> Noaa.CLI.process({:datasets, 1})
"""
def process({:datasets, count}) do
  print(datasets_url(count), "Datasets",
        ["name", "id", "mindate", "maxdate", "datacoverage"])
end

@doc """
Processes the locations command and prints the available locations limitted
by the count parameter.

## Example
    iex> Noaa.CLI.process({:locations, 10})
"""
def process({:locations, count}) when is_integer(count) do
  print(locations_url(count), "Locations",
        ["name", "id", "mindate", "maxdate", "datacoverage"])
end

@doc """
Searches for the location specified by the city parameter. This will obtain
the locationid which is necessary for the data command.
Note: Not yet implemented.

## Example
    iex> Noaa.CLI.process({:locations, "Munich"})
"""
def process({:locations, city}) do
  locations_url(@max_count)
end

@doc """
Processes the data command and finally prints the weather data of the
specified location. A location can be obtained by the locations command.

## Example
    iex> Noaa.CLI.process({:data, [datasets: "GHCND",
    ...>                           location: "CITY:AE000002",
    ...>                           from: "2014-10-01", to: "2014-10-01"]
"""
def process({:data, values}) do
  print(data_url(values), "Weather Data",
        ["datatype", "value", "station", "attributes", "date"])
end

@doc """
Returns the datasets URL
```

```
"""
def datasets_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets?limit=#{count}"
end

@doc """
Returns the locations URL
"""
def locations_url(count) do
  "http://www.ncdc.noaa.gov/cdo-web/api/v2/locations?limit=#{count}"
end

@doc """
Returns the data URL
"""
def data_url(values) do
  """
  http://www.ncdc.noaa.gov/cdo-web/api/v2/data?\
  datasetid=#{values[:dataset]}&\
  locationid=#{values[:location]}&\
  startdate=#{values[:from]}&\
  enddate=#{values[:to]}\
  """
end

@doc """
Returns the bodies for the results and metadata in raw format
"""
def decode_response({:ok, body}) do
  [ { _, results }, { _, [ { _, metadata } ] } ] = body
  [ results, metadata ]
end

@doc """
In case there is an error when fetching data from NOAA the error message is
printed and the application exits with code 2.
"""
def decode_response({:error, reason}) do
  {_, message} = List.keyfind(reason, "message", 0)
  IO.puts "Error fetching data from NOAA: #{message}"
  System.halt(2)
end

@doc """
Transforms the bodies results and metadata into HashDicts.
"""
def transform_to_hashdicts([ results | metadata ]) do
  [ results  |> Enum.map(&Enum.into(&1, HashDict.new)),
    metadata |> Enum.map(&Enum.into(&1, HashDict.new)) ]
end
```

```
  @doc """
  Prints the data in a table format containing the results and the metadata
  """
  def print(url, title, columns) do
    [results, metadata] = url
    |> Noaa.Webservice.fetch
    |> decode_response
    |> transform_to_hashdicts

    IO.puts(IO.ANSI.format(["\n", :blue, :bright, title], true))
    Noaa.TableFormatter.print(results, columns)
    IO.puts(IO.ANSI.format(["\n", :blue, :bright, "Metadata"], true))
    Noaa.TableFormatter.print(metadata, [ "limit", "count", "offset" ])
  end
end
```

To make our application executable we package it with `$ mix escript.build`.

```
$ mix escript.build
Compiled lib/noaa.ex
Compiled lib/noaa/webservice.ex
Compiled lib/noaa/table_formatter.ex
lib/noaa/cli.ex:105: warning: variable city is unused
Compiled lib/noaa/cli.ex
Generated noaa.app
Consolidated List.Chars
Consolidated Access
Consolidated Collectable
Consolidated String.Chars
Consolidated Inspect
Consolidated Enumerable
Consolidated Range.Iterator
Consolidated protocols written to _build/dev/consolidated
Generated escript noaa with MIX_ENV=dev
```

Now let's test it from the command line with `$ ./noaa`

```
$ ./noaa --datasets 12

Datasets
name                       | id          | mindate    | maxdate    | datacoverage
---------------------------+-------------+------------+------------+-------------
Annual Summaries           | ANNUAL      | 1831-02-01 | 2014-07-01 | 1
Daily Summaries            | GHCND       | 1763-01-01 | 2014-12-31 | 1
Monthly Summaries          | GHCNDMS     | 1763-01-01 | 2014-12-01 | 1
Weather Radar (Level II)   | NEXRAD2     | 1991-06-05 | 2015-01-04 | 0.95
Weather Radar (Level III)  | NEXRAD3     | 1994-05-20 | 2015-01-01 | 0.95
Normals Annual/Seasonal    | NORMAL_ANN  | 2010-01-01 | 2010-01-01 | 1
Normals Daily              | NORMAL_DLY  | 2010-01-01 | 2010-12-31 | 1
Normals Hourly             | NORMAL_HLY  | 2010-01-01 | 2010-12-31 | 1
```

```
Normals Monthly           | NORMAL_MLY | 2010-01-01 | 2010-12-01 | 1
Precipitation 15 Minute   | PRECIP_15  | 1970-05-12 | 2013-07-01 | 0.25

Metadata
limit | count | offset
------+-------+-------
10    | 11    | 1
```

9 Commenting the functions

This is something I should have done while coding. After being done it is quite cumbersome. But nevertheless `Elixir` comes with a fantastic documentation feature that allows you not only to insert code like you would test your application in `iex` but it also tests whether this code actually runs. Let's document the `lib/noaa/table_formatter.ex` module as this is the only one where we can create `iex` commands without reaching for the `NOAA` webservice over the internet.

Listing 39: lib/noaa/table_formatter.ex

```
defmodule Noaa.TableFormatter do

  @moduledoc """
  Is formatting the results of the fetched data and the metadata into a table.
  """
  def print(rows, header) do
    columns   = extract_columns(rows, header)
    widths    = max_column_widths(columns, header)
    formatter = formatter_string(widths, " | ")
    print_header(header, formatter)
    print_horizontal_line({ "-", "-+-" }, widths)
    print_table(columns, formatter)
  end

  @doc """
  The header contains the header names of the data. The header filters the
  columns that should be extracted and displayed.

  ## Example
      iex> data   = [[ aaa: "a1", b: "b1", c: "c1"],
      ...>           [ aaa: "a2", b: "b2", c: "c2222"]]
      iex> header =  [ :aaa, :c ]
      iex> Noaa.TableFormatter.extract_columns(data, header)
      [ [ "a1", "a2" ], [ "c1", "c2222" ] ]
  """
  def extract_columns(data, header) do
    for h <- header do
      for d <- data, do: to_string d[h]
    end
  end
```

```
@doc """
Determines based on the header columns and the row columns the maximum
column widths so the data fits into the columns.

## Example
    iex> columns = [[ "a1", "a2"],
    ...>            [ "c1", "c2222"]]
    iex> header  = [ :aaa, :c ]
    iex> Noaa.TableFormatter.max_column_widths(columns, header)
    [ 3, 5 ]
"""
def max_column_widths(columns, header) do
  row_column_widths = columns
  |> Enum.map(&Enum.max_by(&1, fn(x) -> String.length(x) end)
  |> String.length)

  header_column_widths = header
  |> Enum.map(&to_string/1)
  |> Enum.map(&String.length/1)

  List.zip([row_column_widths, header_column_widths])
  |> Enum.map(&Tuple.to_list/1)
  |> Enum.map(&Enum.max/1)
end

@doc """
Creates a formatter string based on the column widths.

## Example
    iex> column_widths = [ 3, 5 ]
    iex> separator     = " | "
    iex> Noaa.TableFormatter.formatter_string(column_widths, separator)
    "~-3s | ~-5s~n"
"""
def formatter_string(column_widths, separator) do
  (column_widths
  |> Enum.map(&("~-#{&1}s"))
  |> Enum.join(separator)) <> "~n"
end

@doc """
Prints the extracted rows into a table format.
"""
def print_table(data, formatter) do
  data
  |> List.zip
  |> Enum.map(&Tuple.to_list/1)
  |> Enum.each(fn(row) -> :io.fwrite(formatter, row) end)
end
```

```
  @doc """
  Prints the table header.
  """
  def print_header(header, formatter) do
    :io.fwrite(formatter, header)
  end

  @doc """
  Prints a horizontal line between the header and the table body.
  """
  def print_horizontal_line({ line, separator }, widths) do
    widths
    |> Enum.map(&String.duplicate(line, &1))
    |> Enum.join(separator)
    |> IO.puts
  end

end
```

To test the comments, that is the included `iex` commands we write a short test that checks that the commands are valid.

Listing 40: test/doc_test.exs

```
defmodule DocTest do
  use ExUnit.Case

  doctest Noaa.TableFormatter
end
```

We run the test with `mix` as we test all our code.

```
pierre@saltspring:~/Learn/Elixir/noaa$ mix test test/doc_test.exs
Compiled lib/noaa/table_formatter.ex
lib/noaa/cli.ex:149: warning: variable city is unused
Compiled lib/noaa/cli.ex
Generated noaa.app
...

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
3 tests, 0 failures

Randomized with seed 362781
```

10 Project Documentation

`Elixir` also comes equipped with a documentation generator called `ExDoc`. To utilize this we have to add it to our dependencies in our `mix.exs` file. `ExDoc` also provides a nice feature to link to the `Github` hosted source code. To use that we add the `Github-URL` into the `application` section in `mix.exs`.

Listing 41: mix.exs

```
defmodule Noaa.Mixfile do
  use Mix.Project

  def project do
    [app: :noaa,
     version: "0.0.1",
     name: "Noaa",
     source_url: "https://github.com/sugaryourcoffee/noaa",
     elixir: "~> 1.0",
     escript: escript_config,
     deps: deps]
  end

  # Configuration for the OTP application
  #
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger, :httpoison]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    [
      { :httpoison, "~> 0.5.0" },
      { :jsx,       "~> 2.4.0" },
      { :ex_doc,    github: "elixir-lang/ex_doc" }
    ]
  end

  defp escript_config do
    [ main_module: Noaa.CLI ]
  end
end
```

To install `ExDoc` issue the command `$ mix deps.get` from the command line. To create the documentation call `$ mix docs`. If you get an error saying

```
** (RuntimeError) Could not find a markdown processor to be used by ex_doc.
You can either:

* Add {:earmark, ">= 0.0.0"} to your mix.exs deps
```

```
  to use an Elixir-based markdown processor

* Add {:markdown, github: "devinus/markdown"} to your mix.exs deps
  to use a C-based markdown processor

* Ensure pandoc (http://johnmacfarlane.net/pandoc) is available in your syst$
m
  to use it as an external tool
```

Then follow the instructions. For instance add `earmark` as dependency.

Listing 42: mix.exs

```
defmodule Noaa.Mixfile do
  use Mix.Project

  def project do
    [app: :noaa,
     version: "0.0.1",
     name: "Noaa",
     source_url: "https://github.com/sugaryourcoffee/noaa",
     elixir: "~> 1.0",
     escript: escript_config,
     deps: deps]
  end

  # Configuration for the OTP application
  #
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger, :httpoison]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    [
      { :httpoison, "~> 0.5.0" },
      { :jsx,       "~> 2.4.0" },
      { :ex_doc,    github: "elixir-lang/ex_doc" },
      { :earmark,   "~> 0.1" }
    ]
  end
```

```
  defp escript_config do
    [ main_module: Noaa.CLI ]
  end
end
```

Then try again `$ mix deps.get` and it should create your documentation.

```
pierre@saltspring:~/Learn/Elixir/noaa$ mix docs
==> earmark
Compiled lib/earmark/context.ex
Compiled lib/earmark.ex
Compiled lib/earmark/cli.ex
Compiled lib/earmark/helpers.ex
Compiled lib/earmark/inline.ex
Compiled lib/earmark/parser.ex
Compiled lib/earmark/html_renderer.ex
Compiled lib/earmark/line.ex
Compiled lib/earmark/block.ex
Generated earmark.app
==> noaa
Compiled lib/noaa.ex
Compiled lib/noaa/webservice.ex
Compiled lib/noaa/table_formatter.ex
lib/noaa/cli.ex:149: warning: variable city is unused
Compiled lib/noaa/cli.ex
Generated noaa.app
Docs successfully generated.
View them at "doc/index.html".
```

Now open `doc/index.html` and you will see a nicely formatted application documentation.

11 Closing

So that's it as a brief overview. That was a lot stuff. But we scratched only the surface of `Elixir`. What we didn't cover is `logging`and multi processing. This is all nicely covered by Dave Thomas' book `Programming Elixr` which can be found at `https://pragprog.com/book/elixir/programming-elixir`. This short description is based on the book from Dave Thomas. As Elixir is a quite new language there are only 2 books available at the moment. You can find them all at `http://elixir-lang.org/`.