Linkit

Pierre Sugar
pierre@sugaryourcoffee.de

February 3, 2015

Abstract

linkit is a command line application that adds links to a web page that is created and updated by linkit. Links as said can be added but also updated and removed. It is possible to search links from the command line and list them. linkit can check whether the links are still active and list inactive links. Each link can have a category, tags and a discription. The category is used to group links and the discription can be searched for but tags a specifically used for searching for specific terms and topics.

1 Project outline

The programm runs from the command line. When provided a link linkit will check whether the link exists and adds it to the web page. A link can be anything that is accessible over an URI. That is web sites and files.

- add a link
- update a link
- remove a link
- check if link is alive
- search link based on description and tags
- list all links
- invoke link from command line (should open web site or file)

2 Source code management

We organize the source in Git at https://github.com/sugaryourcoffee/linkit. We first create our project with Mix, then we cd into the project directory, initialize our git repository, do an initial commit and push the repository to Github.

```
$ mix new linkit
$ cd linkit
$ git init
$ git add .
$ git commit -am "initial commit"
$ git remote add origin git@github.com:sugaryourcoffee/linkit.git
$ git push -u origin master
$
```

3 Implement the Command Line Interface

We write the test first for our command line interface to describe the functions the command line interface actually should provide. We first describe the commands of `linkit`.

```
$ linkit add "http://elixir-lang.org"\
             --tag Elixir\
             --description "Home page of Elixir"
Added "http://elixir-lang.org"
$ linkit add "http://ruby-lang.org"\
             --tag Ruby\
             --description "Home of Ruby"
$ linkit update "http://elxir-lang.org"\
             --description "Home of Elixir"
Updated "http://elixir-lang.org" with description "Home of Elixir"
$ linkit delete "http://elixir-lang.org"
Deleted "http://elixir-lang.org"
$ linkit check "http://elixir-lang.com"
Check error: "http://elixir-lang.com" is not available
$ linkit list --tag "Elixir"
URL                      | Description   | Tags
-------------------------+---------------+-------
"http://elixir-lang.org" | Home of Elxir | Elixir
$ linkit call "http://elixir-lang.org"
$ linkit list
URL                      | Description   | Tags
-------------------------+---------------+-------
"http://elixir-lang.org" | Home of Elxir | Elixir
"http://ruby-lang.org"   | Home of Ruby  | Ruby
```

The test is shown in 1 on page 2.

Listing 1: test/cli_test.exs

```
defmodule CliTest do
  use ExUnit.Case

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

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

```
test ":help returned when no known command is given" do
  assert parse_args(["--nocommand"]) == :help
end

test ":add returned with uri, tag and description" do
  long   = ["--add", "http://example.com",
            "--tag", "tag",
            "--description", "description"]
  short  = ["--add", "http://example.com",
            "--tag", "tag",
            "--description", "description"]
  result = {:add,
             [tag: "tag", description: "description"],
             ["http://example.com"]}

  assert parse_args(long)  == result
  assert parse_args(short) == result
end

test ":update returned with uri, tag and description" do
  long   = ["--update", "http://example.com",
            "--tag", "tag",
            "--description", "description"]
  short  = ["--update", "http://example.com",
            "--tag", "tag",
            "--description", "description"]
  result = {:update,
             [tag: "tag", description: "description"],
             ["http://example.com"]}

  assert parse_args(long)  == result
  assert parse_args(short) == result
end

test ":remove returned with uri" do
  long   = ["--remove", "http://example.com"]
  short  = ["-r",       "http://example.com"]
  result = {:remove, [], ["http://example.com"]}

  assert parse_args(long)  == result
  assert parse_args(short) == result
end

test ":check returned with uri" do
  long   = ["--check", "http://example.com"]
  short  = ["-c",      "http://example.com"]
  result = {:check, [], ["http://example.com"]}

  assert parse_args(long)  == result
  assert parse_args(short) == result
  end

  test ":list returned with given tag and description filter" do
    long   = ["--list", "http://example.com",
              "--tag", "tag",
              "--description", "description"]
    short  = ["--list", "http://example.com",
              "-t", "tag",
              "-d", "description"]
    result = {:list, [tag: "tag", description: "description"],
               ["http://example.com"]}

    assert parse_args(long)  == result
    assert parse_args(short) == result
  end

end
```

The implementation so far is in 2 on page 4.

Listing 2: lib/linkit/cli.ex

```
defmodule Linkit.CLI do

  @moduledoc """
  Parses the command line and dispatches to the respective functions to add,
  update, delete, check and list URIs
  """
  @vsn 0.1

  @doc """
  Entry for parsing the command line options
  """
  def main(argv) do
    argv
    |> parse_args
  end

  @doc """
  parses *argv* that is retrieved from the command line. *argv* can have
  following values.

  * ["--help  "]
  * ["--add   ", "uri", "--tag", "tag", "--description", "description"]
  * ["--update", "uri", "--tag", "tag", "--description", "description"]
  * ["--delete", "uri"]
  * ["--check ", "uri"]
  * ["--list  ", "uri", "--tag", "tag", "--description", "description"]

  Returns tuples for each of the commands, except for help which is returned as
  an atom.
  * :help
  * {:add,      URI, TAG, DESCRIPTION}
  * {:update, URI, TAG, DESCRIPTION}
  * {:remove, URI}
  * {:check,   URI}
  * {:list ,    URI, TAG, DESCRIPTION}
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                              switches: [help:         :boolean,
                                         add:          :boolean,
                                         update:       :boolean,
                                         remove:       :boolean,
                                         check:        :boolean,
                                         list:         :boolean,
                                         tag:          :string,
                                         description:  :string],

                              aliases:  [h:  :help,
                                         a:  :add,
                                         u:  :update,
                                         r:  :remove,
                                         c:  :check,
                                         l:  :list,
                                         t:  :tag,
                                         d:  :description])

    command(parse)
  end

  # Determine the command that has been invoked from command line and return
  # the command as an atom with the command arguments and values.
  defp command({[{:add,     true} | args], argv, _}), do: {:add,
args, argv}
  defp command({[{:update, true} | args], argv, _}), do: {:update, args, argv}
  defp command({[{:remove, true} | args], argv, _}), do: {:remove, args, argv}
  defp command({[{:check,  true} | args], argv, _}), do: {:check,
args, argv}
  defp command({[{:list ,   true} | args], argv, _}), do: {:list ,
args, argv}

  # When help is called command returns the atom :help. :help is also returned
  # when user invokes an unknown command
  defp command({[{:help,    true} |      _],      _, _}), do:
:help
  defp command({[_                          ],      _, _}), do:
:help
end
```

Now that the commands get parsed we want to process the commands. The help command is the easiest so we start with that. The help command just prints the usage of `linkit` and then exits the application.

The next command we will process is the `add` command as without this we cannot do anything with the application. In a first step we just save the links to a file. The generation of the web page will follow later on when we have implemented the other commands.

We first have to decide where the link-file lives. We could let the user decide about that. But we will take the approach to put the working directory in the users home folder under `/.syc/linkit/`. There we put all application specific data. We call the link file `links`. So we don't have to hardcode this we want to make this configurable. The place for that is the `config/config.exs`.

Listing 3: config/config.exs

```
# This file is responsible for configuring your application
# and its dependencies with the aid of the Mix.Config module.
use Mix.Config

# This configuration is loaded before any dependency and is restricted
# to this project. If another project depends on this project, this
# file won't be loaded nor affect the parent project. For this reason,
# if you want to provide default values for your application for third-
# party users, it should be done in your mix.exs file.

# Sample configuration:
#
#     config :logger, :console,
#       level: :info,
#       format: "$date $time [$level] $metadata$message\n",
#       metadata: [:user_id]

# It is also possible to import configuration files, relative to this
# directory. For example, you can emulate configuration per environment
# by uncommenting the line below and defining dev.exs, test.exs and such.
# Configuration from the imported file will override the ones defined
# here (which is why it is important to import them last).
#
#     import_config "#{Mix.env}.exs"

config :app, directory: File.expand_path("~/.syc/linkit/")
config :app, linksfile: Application.get_env(:app, :directory) <> "/links"
```

The `add` function needs to check whether the `URI` is already in the `links`-file. So we read the file and check whether the link is already there. If not we add the link otherwise we ask whether we should update the link with the new `tags` and `description` if given.

To list the `URIs` the user invokes the `list` function. We read the `link` file and print the contents in a nicely formatted table.

The `update` function we will implement in a similar way as the `add` function. We also check whether the link is available if not we ask the user whether to add it as a new link otherwise we update the link as intended.

When the `check` function is invoked we call the link and see whether it returns a `Error 404: Not found` error. After all links have been checked we print a statistic about the checks. The `check` function is a good candidate to use `Agents` and run the checks in parallel.

The purpose of the `remove` link is basically after the `check` function fails for a specific `link` we want to remove the dead link from our `link` file.

As usual we will start with the tests which are shown in 4 on page 7.

Listing 4: test/cli_process_test.exs

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

  test "help" do
    result = capture_io fn -> process(:help) end

    assert result == """
   NAME
        linkit - organizing links and adding them to a web page

   SYNOPSIS
        linkit command [command options] [arguments...]

   VERSION
        0.1

   COMMANDS
        add    - Adds a link to linkit
        check  - Checks if link can be reached
        help   - Prints this help
        list   - Lists the links
        remove - Removes a link
        update - Updates a link

    """
  end

  test "add" do
  end

  test "update all values" do
  end

  test "update the tags" do
  end

  test "update the description" do
  end
  test "remove" do
  end

  test "check a specific URI" do
  end

  test "check all URIs" do
  end

  test "check URIs based on a filter" do
  end

  test "list a specific URI" do
  end

  test "list all URIs" do
  end

  test "list URIs based on a filter" do
  end
end
```

.