Clojure : a first tutorial

GnuVince

dernière version:
October 31, 2008

Contents

1. First Clojure tutorial

This post is the first of what I hope will become a series about Clojure. Clojure is a young language, and though there is a lot of documentation already on the Internet and in the blogs of many enthusiasts, I figured there would be no harm in having some more. In this series, I will start with a simple script and with each post, I will improve the program by introducing new Clojure features. The whole thing is not mapped out, and as such, I am very receptive to constructive comments on how to make these posts better.

1.1. The problem

The problem we will tackle on is a fairly simple one: scraping web sites. I am a of web comics enthusiast, I read quite a few of them, but I don't like to go on 20 web sites to view them all, and their RSS feeds are often not to my satisfaction: some feeds only give you a link to the latest strip, others have news, information or publicity in them that you don't want. So we will write a script that extracts the latest strip from web comic sites and create an RSS feed with those.

1.2. What we'll do today

In this first post, we'll get something very simple working: the program will download the content of a web site, extract the strip link and print it. This will allow us to view data structures and Java interop. I will not assume that you know any Clojure, so I'll try to explain as we go along. If something is unclear, you can always check out the documentation on the official Clojure web site. The pseudo code of our application will be as follows:

```
for each comic:
    get the html
    extract the image with a regex
    display the complete image URL
```

1.3. Data

We'll start our program by defining our data. We will want to scrape several comic strips and not have to write one function per web comic, so we'll need a standard way to represent the different comics we have. We will need four pieces of data:

- The name of the web comic
- The URL where the latest comic can be found
- A regular expression to capture the strip image link
- An optional URL prefix to construct an absolute URL

Because most sites use relative links for their images, if no URL prefix is given, we will assume that the URL of the latest strip page is to be used as the prefix. We will represent the data of one comic with a hash-map and we will put all those hash-maps inside a vector. Here's the result with two comics:

```
(def *comics*
  [{:name "Penny-Arcade"
    :url "http://www.penny-arcade.com/comic/"
    :regex #"images/\\d{4}/.+?(?:png|gif|jpg)"
    :prefix "http://www.penny-arcade.com/"
    }
   {:name "We The Robots"
    :url "http://www.wetherobots.com/"
    :regex #"comics/.+?[.](?:jpg|png|gif)"
    }
  ])
```

1.4. A few notes about this piece of code:

- `def` is special form to assign a value to a name. In this case, we assign our vector of hash-maps to *comics*.
- `*` is a valid character in an identifier. It is a Lisp convention to used asterisks around a variable name to indicate that it is a global variable. The list of valid identifier characters is described in the page about the reader.
- Clojure has literal syntax for vectors: space-separated values enclosed in square braces.
- Clojure has literal syntax for maps: space-separated values enclosed in curly braces. Clojure considers commas to be white space, so you can use them to clearly separate the different pairs: `{:false 0, :true 1}`.
- Clojure has a special data type called a keyword. Keywords begin with a colon follow one or more identifier characters.
- Clojure strings are enclosed inside double quotes.
- Clojure has literal syntax for regular expressions: `#"regex"`. In the latest stable release of Clojure (20080916), the text inside the quotes is not automatically escaped, and the backslashes need to be doubled. In the Subversion repository and in future releases, this behavior has been changed and you no longer need to double the backslashes (except to represent a literal backslash.)
- We omitted the `:prefix` key/value pair for "We The Robot": accessing a non-existing key in a Clojure map returns nil, which is what we said we want when we want to use the value of the :url field as the prefix.

That's actually quite a lot of notes for such a short piece of code! Now that we have our data, let's look at the next step, fetching the HTML from a URL.

1.5. Fetching the HTML

Java has a class to read documents through the HTTP protocol, which means that Clojure has a class to read documents through the HTTP protocol. Sadly, Java does not have a

method to download an entire document as a string. We'll have to create our own function to do the deed. The classes that we'll need can be accessed by their fully-qualified names (e.g.: java.io.BufferedReader), but this tends to make the code long-winded. We'll use the import function to load the class names into the current namespace to keep our code shorter.

```
(  import   '(java.net URL)
          '(java.lang StringBuilder)
          '(java.io BufferedReader InputStreamReader))
```

import takes an arbitrary number of lists where the first element is a symbol representing the name of the package and the rest are the classes to be added to the namespace. Here, we import `URL, StringBuilder, BufferedReader and InputStreamReader`. Now, let's look at the code to download an HTML page:

```
(defn fetch-url
  "Return the web page as a string."
  [address]
  (let [url (URL. address)]
    (  with-open   stream (. url (openStream))
      (let [buf (BufferedReader. (InputStreamReader. stream))]
        (  apply   str (  line-seq   buf))))))
```

We'll look at the code line by line in just a moment, but let me first explain quickly what this function does. fetch-url is a function that takes an argument, address, uses this argument to create a new URL object and open a stream to that object. We then read all the lines from that stream, join them together and return one big string.

- `(defn fetch-url`: `defn` is a macro used to define a new function. It is followed by the name of the function, `fetch-url`.
- `"Return the web page as a string."`: functions can have a documentation string which can be consulted in the REPL with the doc function. It appears between the name of the function and the formal parameters.
- `[address]`: the list of formal parameters is actually a vector of formal parameters.
- `(let [url (URL. address)]` : let is a special form used to introduce a new scope with some bindings. The bindings are defined inside square brackets with the format `[name1 value1 name2 value2 ...]`. `(URL. address)` creates a new URL (the Java class) from address. Suffixing a dot to a class name is the same as `(new ClassName)`.
- `(with-open stream (. url (openStream))`: `with-open` is a macro that wraps code inside a `try/finally` block and calls the close method after the block has finished executing. Here we open a stream to the URL of our comic and

`with-open` will automatically close that stream when we're done. There are other ways to call the method: `(. url openStream)` and `(.openStream url)` are both valid.

- Next we have one more definitions, a buffered reader. This should be familiar to Java people.
- `(apply str (line-seq buf))))))` : the function `line-seq` returns a lazy sequence of all the lines in a `BufferedReader`. We then apply the `str` function to all those lines to join them together into one string and this value is returned. You'll note that there are a lot of closing parentheses on this line: it's a Lisp convention to close every parentheses on the same line instead of putting each one on a separate line as is conventional in the Java world.

Phew, that was a lot to take in! Now that we've completed the second line of our pseudo code, we're ready to extract the image links.

1.6. Extracting the image link

The function used to get the image link is much shorter than fetch-url. We will pass a comic (a map), we will use the Clojure function re-find to find the string we are looking for and we will return it with the prefix.

Let's look at the code:

```
(defn image-url
  "Return the absolute URL of the image of a comic.
  If the comic has a prefix, prepend it to the URL,
  otherwise use the :url value."
  [comic]
  (let [src (fetch-url (:url comic))
        image ( re-find (:regex comic) src)]
    ( str ( or (:prefix comic) (:url comic))
         image)))
```

This should now look familiar to you. A function of one argument with a documentation string. We won't look at every line, instead I'll explain the important parts:

- Maps are functions of their keys: to access a value in a map, you can say `(:keyword map)` or `(map :keyword)`, both are valid.
- `re-find` returns either the matching string if there were not captures in the regular expression, a vector if there were captures or nil if no match was found. We don't do any captures in our examples, so image is a string.
- The function `str` is used to concatenate strings. `(str "foo" "bar")` returns `"foobar"`.
- `or` returns its first argument if it's true, the second one otherwise. `nil` and `false` are the only false values, all other values are true. This returns the prefix if there is one or the url if there is no defined prefix.

1.7. Printing the URLs

Finally, we can print the URLs. We will use the doseq macro for this purpose, which is practically a foreach loop. doseq takes three argument: the name of an individual item, a collection and a body. We will print the name of the comic and the URL of its latest strip.

```
(doseq [comic *comics*]
  (println (str (:name comic) ": " (image-url comic))))
```

This should give us the following output:

```
Penny-Arcade: http://www.penny-arcade.com/images/2008/20081029.jpg
We The Robots: http://www.wetherobots.com/comics/2008-10-22-Storytime.jpg
```

1.8. Next time

Next time, we'll look at how multimethods can help us to handle cases such as Xkcd where we also want to get the URL of the strip, but also the alt text to have a complete strip.