Clojure Building Blocks

An Introduction to Clojure and Its Capabilities for Data Manipulation

by Jean-François “Jeff” Héon

> Jeff introduces Clojure fundamentals and uses them to show why you might want to explore this language further.

I mainly use Java at work in an enterprise setting, but I’ve been using Clojure at work for small tasks like extracting data from log files or generating or transforming Java code. What I do could be done with more traditional tools like Perl, but I like the readability of Clojure combined with its Java interoperability. I particularly like the different ways functions can be used in Clojure to manipulate data.

I will only be skimming the surface of Clojure in this short article and so will present a simplified view of the concepts. My goal is for the reader to get to know enough about Clojure to decide if it is worth pursuing further using longer and more complete introduction material already available.

I will start with a mini introduction to Clojure, followed by an overview of sequences and functions combination, and finish off with a real-world example.

Ultra Crash Course

Clojure, being a Lisp dialect, has program units inside lists. A function call will be the first element of a list, optionally followed by parameters.

For setup instructions, look here. Clojure programs can be run as a script from the command line, as a file from your IDE, or precompiled and packaged to be run as a normal Java jar. They can also be simply loaded or typed in the REPL, the interactive development shell. The REPL might be invoked from your IDE or simply called from the command line, provided you have java 1.5 or higher installed:

```
java -cp clojure.jar clojure.main
```

I invite you to follow along with a REPL on a first or second read and try the examples and variations. You can display the documentation of a function with the doc function.

Entering the following at the REPL:

```
(doc +) ;In Clojure, + is a function and not an operator
```

will echo the documentation. For the article, I precede REPL output with the > symbol.

```
>-------------------------
clojure.core/+
([] [x] [x y] [x y & more])
Returns the sum of nums. (+) returns 0.
```

For the curious, you can also display the source of a function with source.

```
(source +) ;Try it yourself.
```

First, let’s start with the mandatory addition example.

```
(+ 2 4 6)
> 12
```

Values can be associated to a symbol with def.

```
(def a 3)
>#'user/a
```

The REPL will write the symbol name preceded by the namespace, #'user/a, in this case.

Typing “a” will return back its value.

```
a
>3
```

The symbol is bound to the result of the expression after its name.

```
(def b (* 2 a))
>#'user/b
b
>6
```

The str function will concatenate the string representation of its arguments.

```
(str "I have " b " dogs")
>"I have 6 dogs"
```

We can also string together characters. You’ll notice that character literals in Clojure are preceded by a backslash.

```
(str \H \e \l \l \o)
>"Hello"
```

It is common to manipulate data as collections, be it lists, vectors, or whatever. The apply function will call the given function with the given collection unpacked.

```
(def numbers [2 4 6]) ;define a vector with 3 integers
>#'user/numbers
```

(I will omit the echoing of the symbol name for the remainder of the article.)

```
(apply + numbers) ;Sames as (+ 2 4 6)
>12
```

Vectors are accessed like a function by passing the zero-based index as an argument.

```
(numbers 0)
>2
```

Sequences

Clojure has many core functions operating on sequences. A sequence allows uniform operations across different kinds of collections, be it a list, a vector, a string, etc. In our examples, we will be using mostly vectors, an array-like data structure with constant-time access.

For example, the take function will return the n first elements.

```
(take 3 [1 2 3 4 5]) ;Take 3 from a vector
```

```
>(1 2 3)
(take 3 "abcdefg") ;Take 3 from a string
>(\a \b \c)
```

If you were expecting to get back the string “abc”, you might be disappointed by the result, as I was the first time I tried. What happened here? Operations producing sequences, like take, do not return elements in the original collection data type, but return a sequence of elements. That is why calling take on a string returns a sequence of characters. This means that take on the vector did not return a vector, but a sequence.

Let’s define a test vector to explore more sequence manipulations.

```
(def days-of-the-week ["sunday", "monday", "tuesday",
"wednesday", "thursday", "friday", "saturday"])
```

Oops! I forgot to capitalize the days. Let’s use map, which applies a function to each element of a collection and returns a sequence of the results. For example, the following returns a sequence of our numbers incremented by one.

```
(map inc numbers)
>(3 5 7)
```

First let’s develop a function to capitalize a word. Note that there already exists a capitalize function in the clojure.string namespace, but we’ll roll our own to demonstrate a few points. We’ll develop our function incrementally using the REPL.

We’ll start by getting the first letter of a word. The function first will create a sequence over the given collection and return the first element.

```
(first "word")
>\w
```

Let’s use a bit of Java interop and call the static function toUpperCase from the Java Character class.

```
(java.lang.Character/toUpperCase (first "word"))
>\W
```

So far so good. Now let’s get the rest of our word.

```
(rest "word")
>(\o \r \d)
```

What happens if we want to string our capitalized word together?

```
(str (java.lang.Character/toUpperCase (first "word")) (rest "word"))
> "W(\\o \\r \\d)"
```

We get back the string representation of the first argument, the letter W, concatenated with the string representation of the sequence of the rest of the word.

We need to use a variant of the function apply, which takes an optional number of arguments before a sequence of further arguments.

```
(apply str (java.lang.Character/toUpperCase (first "word"))
(rest "word")) ;Same as (str \W \o \r \d)
>"Word"
```

Now let’s make a function from our trials and tribulations.

```
(defn capitalize [word]
(apply str (java.lang.Character/toUpperCase
(first word)) (rest word)))
```

The first line defined the function named capitalize taking one parameter named word. The second line is simply our original expression using the parameter.

Let's try it out.

```
(capitalize (first days-of-the-week))
> "Sunday"
```

Good. We're ready to capitalize each day of the week now.

```
(def capitalized-days (map capitalize days-of-the-week))
capitalized-days
>("Sunday" "Monday" "Tuesday" "Wednesday"
"Thursday" "Friday" "Saturday")
```

Map is an example of a high-order function, which has one or more functions in its parameter list. It's a convenient way of customizing a function's behavior via another function instead of using flags or more involved methods like passing a class containing the desired behavior inside a method.

Notice that the original collection is left untouched.

```
days-of-the-week
> ["sunday" "monday" "tuesday" "wednesday"
"thursday" "friday" "saturday"]
```

Clojure collections are persistent, meaning they are immutable and that they share structure. Let's add a day to have a longer weekend.

```
(conj capitalized-days "Jupiday")
>("Jupiday" "Sunday" "Monday" "Tuesday"
"Wednesday" "Thursday" "Friday" "Saturday")
```

Adding Jupiday has not modified the original collection capitalized-days, which is guaranteed not to ever change, even by another thread. The longer week was not produced by copying the 7 standard days, but by keeping a reference to the 7 days and another to the extra day. Various collection "modifications", which really return a new data structure, are guaranteed to be as or almost as performant as the mutable version would be.

Filtering operations can be done with the filter high-order function, which return a sequence of elements satisfying the passed-in function.

```
(filter odd? [0 1 3 6 9])
>(1 3 9)
```

When a function passed to an higher function is simple and only used once, there is no need to give it a name. We can define the function in-place. We just use fn instead of defn and forego specifying a name.

For example, here is another way of capitalizing our week days using an anonymous function.

```
(map (fn [word] (apply str (java.lang.Character/toUpperCase
(first word)) (rest word))) days-of-the-week)
>("Jupiday" "Sunday" "Monday" "Tuesday"
```

```
  "Wednesday" "Thursday" "Friday" "Saturday")
```
Another handy sequence operation is reduce. It applies a function between the first two elements of a vector and then applies the function with the result and the 3rd element and so on.

```
(reduce * [1 2 4 8]) ;Same as (* (* (* 1 2) 4) 8)
> 64
```
Another form of reduce takes a parameter as the first value to combine with the first element.

```
(reduce * 10 [1 2 4 8]) ;Same as (* (* (* (* 10 1) 2) 4) 8)
> 640
```
Let's sum the number of characters for each day.

```
(reduce (fn [accumulator element]
(+ accumulator (count element))) 0 days-of-the-week)
> 50
```
We can redefine the previous anonymous function using syntactic sugar.

```
#(+ %1 (count %2))
```
Note that we can omit the number 1 from the usage of the first argument.

```
#(+ % (count %2))
```
Here is an example to extract the word three in three languages from a vector of vectors.

```
(map #(% 3) [["Zero" "One" "Two" "Three"]
["Cero" "Uno" "Dos" "Tres"]["Zéro" "Un" "Deux" "Trois"]])
>("Three" "Tres" "Trois")
```

Composition of Functions

Let's explore function assembly with a wild example: capitalize and stretch.

Let's define our additional function.

```
(defn stretch [word]
(apply str (interpose " " word)))
```
And test.

```
(stretch "word")
>"w o r d"
```
This would be a standard way of combining stretch and capitalize.

```
(map (fn [word] (stretch (capitalize word))) days-of-the-week)
>("S u n d a y" "M o n d a y" "T u e s d a y" "W e d n e s d a y"
"T h u r s d a y" "F r i d a y" "S a t u r d a y")
```
Clojure also provides the comp function, which produce a new function from the successive application of the functions given.

```
(map (comp capitalize stretch) days-of-the-week)
>("S u n d a y" "M o n d a y" "T u e s d a y" "W e d n e s d a y"
"T h u r s d a y" "F r i d a y" "S a t u r d a y")
```

Had we wanted to keep a capitalize-n-stretch function, we could have associated the result of the composition to a symbol.

```
(def capitalize-n-stretch (comp capitalize stretch))
(capitalize-n-stretch "Hello")
>"H e l l o"
```

We can compose more than one function together and we can even throw in anonymous functions into the mix.

```
(map (comp inc (fn [x] (* 2 x)) dec) numbers)
>(3 7 11)
```

We can produce a new function by partially giving arguments.

```
(def times-two (partial * 2))
(times-two 4) ;Same as (* 2 4)
>8
```

We can revisit our compose example differently.

```
(map (comp inc (partial * 2 ) dec) numbers)
>(3 7 11)
```

A Real-World Example

Here is an example of a real function I wrote to collect all the referenced table names for a specific schema. The SQL statements are peppered in various Java files. I call the extract-table-names function for each file, and a corresponding .out file is produced with the referenced table names, uppercased, sorted, and without duplicates. After processing the file, the name of the file and the table count is returned to be displayed by the REPL. The goal is not for you to understand all the program, just to have a feel of it.

```
(ns article
(:use [clojure.string :only [split-lines join upper-case]]))
;Import a few helper functions

;;Extract table names matching MySchema for a given line
(defn extract[line]
(let [matches (re-seq #"(\s|\")+((?i)(MySchema)\.\w+)" line)]
;We're using a regular expression
(map #(% 2) matches)))
;Extract the table name (third item in each match)

(defn extract-table-names [file-path file-name]
"Extract MySchema.* table names from the java file
and write sorted on an out file."
(let [file (slurp (str file-path file-name ".java"))
```

```
  ;Get the file
  lines (split-lines file)
  ;Split the file by lines
  names (remove nil? (flatten (map extract lines)))
  ;Extract and remove non-matches
  cleaned-names (-> (map upper-case names) distinct sort)
  ;Uppercased, distinct only and sorted
  ]

  ;Write the file with unique sorted table names
  (spit (str file-path file-name ".out")
  (join "\n" cleaned-names))
  (str file-name ". Table count: " (count cleaned-names))))

  ;Usage example
  (extract-table-names "/DataMining/" "DataCruncher")
```

I've also used Clojure to extract running time statistics of our system and then generate distribution charts with Incanter, a wonderful interactive statistical platform.

This conclude my brief tour of data manipulation with Clojure. There is a lot more to sequences than what I've shown. For example, they are realized as needed, in what is referred to as lazy evaluation. There is an excellent summary of functions in the sequence section of the Clojure cheatsheet. Clojure functions can also be combined in other interesting ways like the thread-first or thread-last macros.

Clojure Collections

Looking Deep Inside Clojure Data Collections

by Steven Reynolds

> Steven explains the benefits of immutability and explores how Clojure's data collections handle it.

Clojure embraces a functional programming style, controlling mutability tightly. Unless you take special steps to permit it, data collections in Clojure are not mutable; they cannot be changed.

Why bother with immutability? Clojure does so for two key reasons. Having a network of references to mutating objects is fundamentally very complex. Complexity is the enemy of software development. Secondly, such a network is exquisitely difficult to make correct while allowing concurrency.

In the familiar imperative or Object Oriented programming styles, when a structure is updated, it is mutated. The application holds a stable reference to a collection and the collection itself is changed. Clojure instead generally uses collections that cannot be changed; the update operations return a new version of the collection. Whenever a version of a collection is created, that version of the collection must remain accessible (because it cannot be changed). Hence these type of collections are sometimes called persistent. Of course, old versions of the collection can be garbage collected when there are no

references to them.

The interesting challenge is to ensure that, when a new collection must be returned, Clojure doesn't need to copy the entire collection. Excessive copying causes performance degradation.

Lists

Clojure contain a fairly classical representation for lists. The next code sample creates some lists and exports a graph of each of them.

```
(defn list-ex []
(let [w '(1 2 3) ; Figure 1
x (rest a) ; Figure 2
y (conj a '(3 4)) ; Figure 3
z (cons 1 a) ; Figure 4
lsaver (PersistentListSaver.)
csaver (ConsSaver.)]
(. lsaver save w "list_before.dot")
(. lsaver save x "list_after_rest.dot")
(. lsaver save y "list_after_conj.dot")
(. csaver save z "list_after_cons.dot")
))
```

The original list, w, is created in the let form and contains the elements 1, 2, and 3. It is shown in Figure 1. The second list, x, is the rest of w. This is all of w except its head element (1 in this case).

PersisentListSaver is a Java class that uses reflection to dump the Clojure list to a graph. This graph is then drawn using GraphViz.

If you look at the list in Figure 1, you can see the representation has a first that contains the head of the list, and a rest that contains other elements.

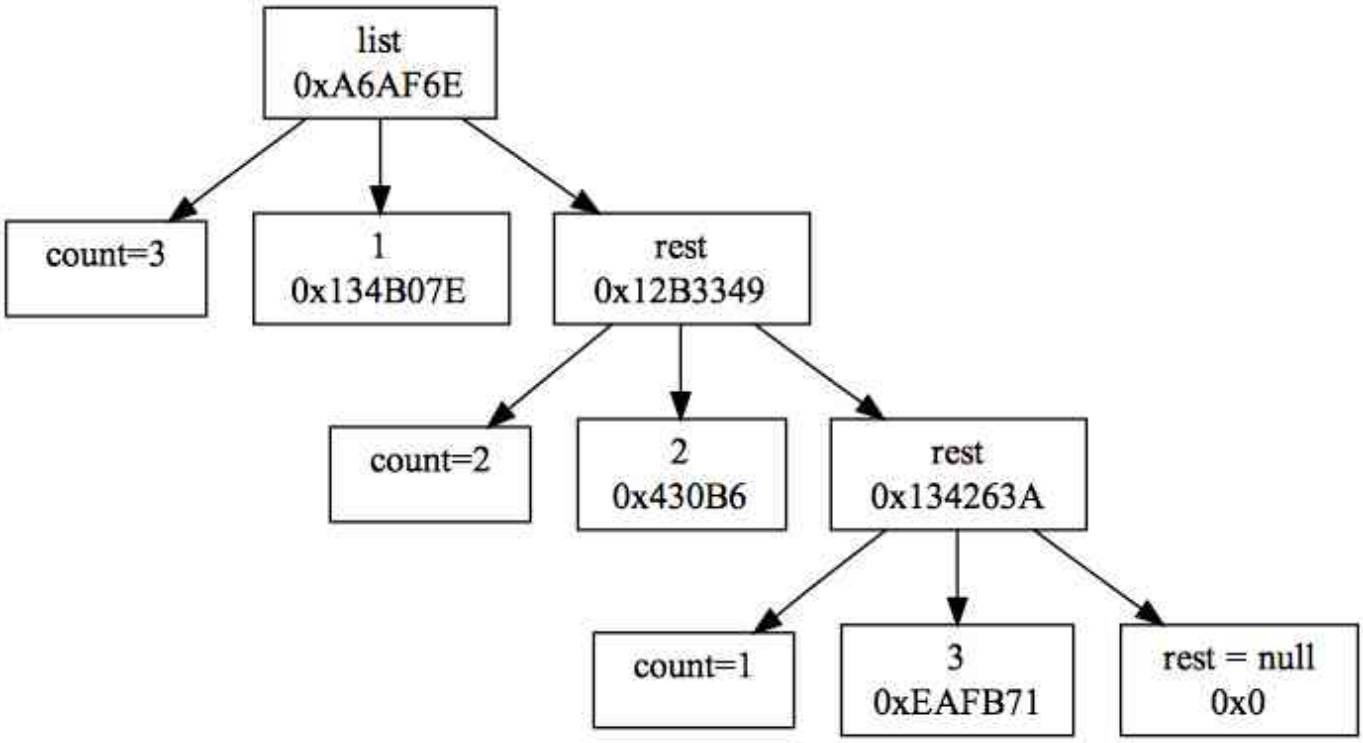

Figure 1: Clojure List

As you can see in Figure 2, when Clojure takes the rest of the original list, it doesn't need to copy any data. The hex numbers in the graph nodes are the result from calling System.identityHashCode on the node. The Java contract of this hash code is that it will always return the same number for the same Object. The usual JDK implementation is to return the address of the Object.

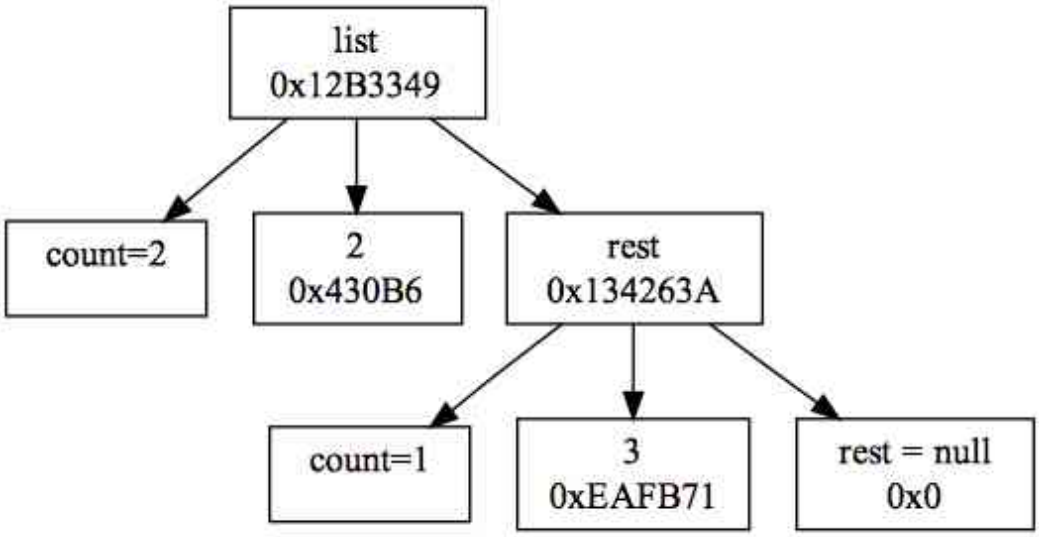

Figure 2: Clojure List from Figure 1 after rest applied

The graphs are simplified: some details left out, and element nodes are shown inline rather than with a separate node. In reality primitives are not stored in Clojure (or Java) collections. They're always storing a boxed Object. That boxing is left out of the graphs for greater clarity and to save paper.

If you add elements to a list, Clojure also arranges to share data. The code sample adds 3 and 4 to the

list w using conj and saves this new list in y. The conj function adds elements to the collection at the most efficient location.

If you compare Figure 3 to Figure 1, you can see that conj added the elements at the front. Again, all the preexisting elements are shared.

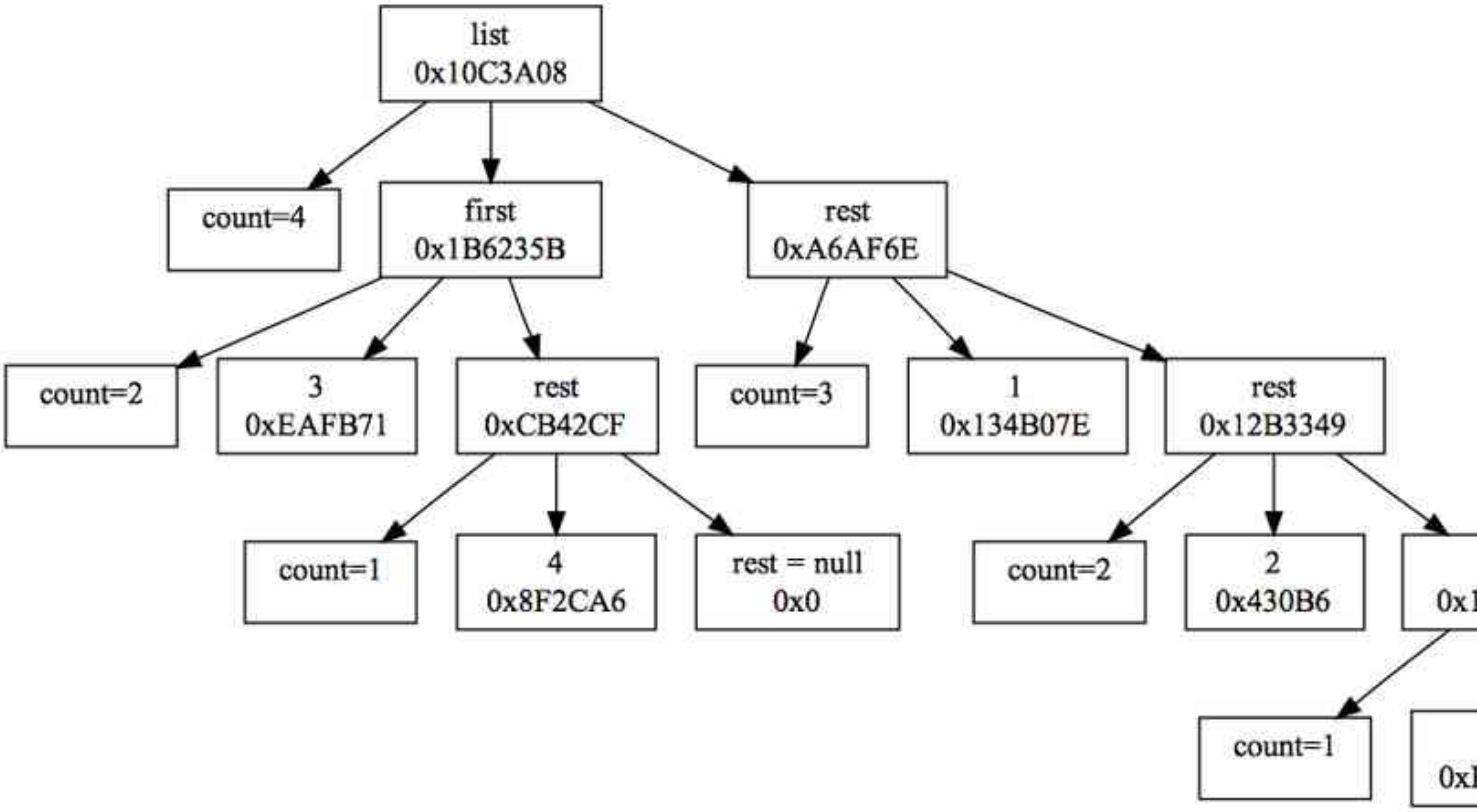

Figure 3: Clojure List from Figure 1 after conj applied

When you add elements to a list using the cons function (list z), Clojure creates a mixed structure with a Cons cell, a first element, and an ISeq, as shown in Figure 4. This shares elements, but future operations will not be quite as efficient. The cons function is specified to always add to the front of any data collection.

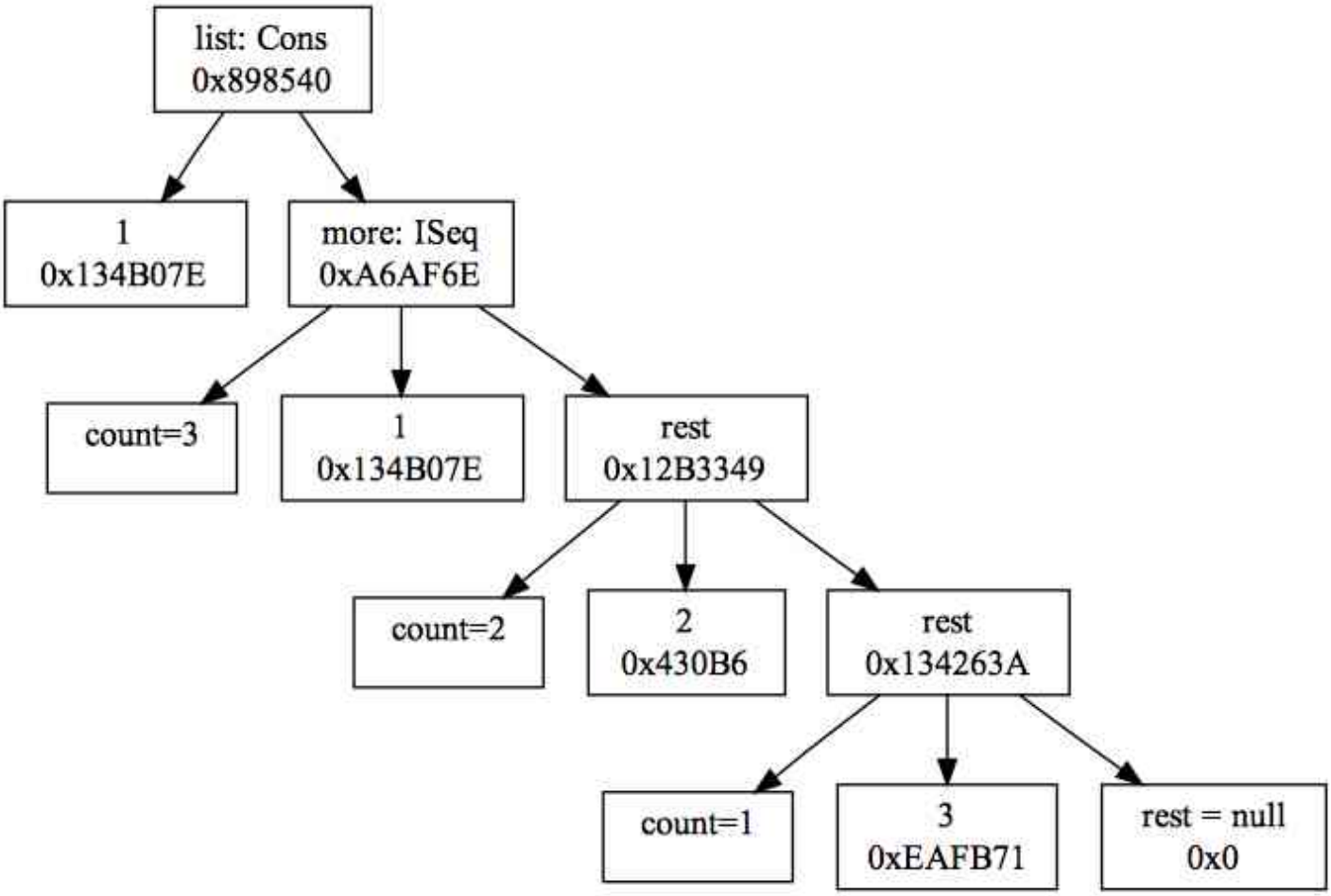

Figure 4: Clojure List from Figure 1 after cons applied

Maps

Another very important type of collection is maps. The next example shows a small map.

```
(defn amap-assoc []
(let [x '{1 "one" 2 "two"} ; Figure 5
y (assoc x 3 "three") ; Figure 6
pamsaver (PersistentArrayMapSaver.)]
(. pamsaver save x "amap_before_assoc.dot")
(. pamsaver save y "amap_after_assoc.dot")
))
```

In Clojure, small maps are stored in a simple structure backed by an array that contains the keys and values interleaved. You can see this in Figure 5. Lookups are done by a linear scan of the array. That's fast when the map is small. If you add an element to the map (use the function assoc), Clojure just makes a new map without sharing (Figure 6).

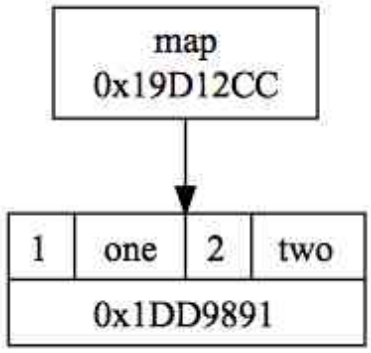

Figure 5: Clojure Array Map

Figure 6: Clojure Array Map from Figure 5 after assoc applied

Longer maps are stored in a PersistentHashMap. They stored as a tree with a 32-way branching factor. PersistentHashMap uses a sparse array to store data at each tree level. The bitmap indicates which elements of the nominal full array are actually present in the realized sparse array.

You can read more about Clojure's PersistentHashMap at the Wikipedia article "Hash array mapped trie" and at the blog by arcanesentiment. The Clojure implementation is based on a paper by Phil Bagwell. This data structure is technically a trie rather than a tree because the keys can be variable length, for example a String.

The next example shows a small hash map.

```
(defn hmap-assoc []
(let [x (hash-map 1 "one" 2 "two") ; Figure 7
y (assoc x 3 "three") ; Figure 8
phmsaver (PersistentHashMapSaver.)]
(. phmsaver save x "hmap_before_assoc.dot")
(. phmsaver save y "hmap_after_assoc.dot")
))
```

In Figures 7 and 8, you can see the maps before and after an element is added. In this example, only one level of the trie is needed. If you look at Figure 8, you can see that the new map is not able to share any elements with the old map.

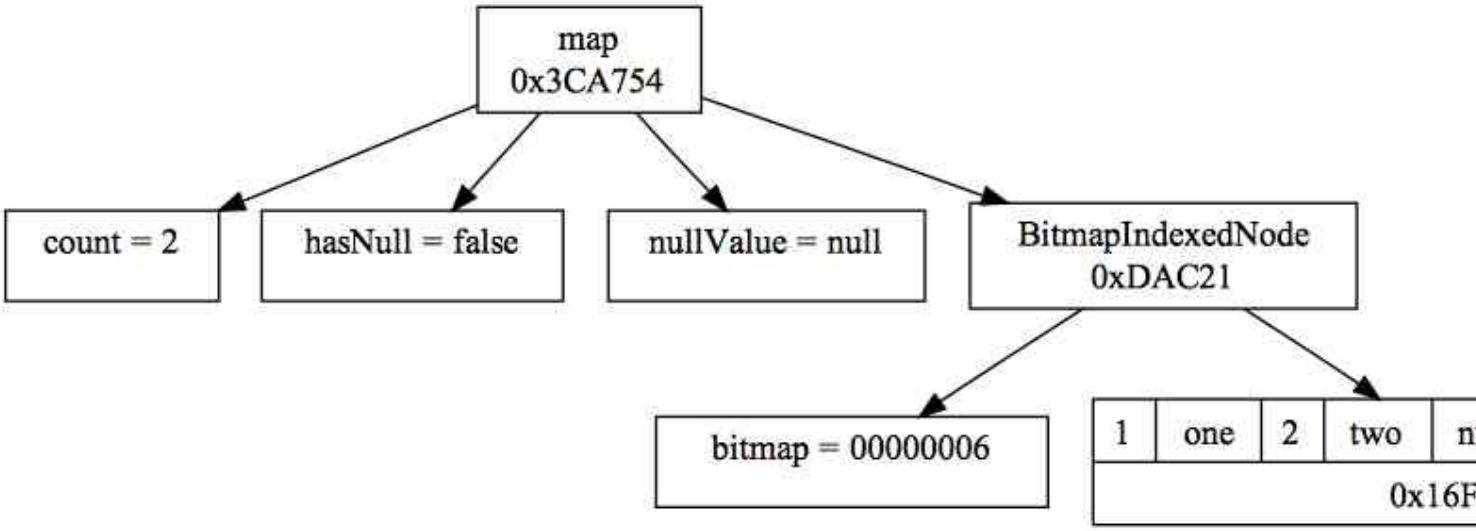

Figure 7: Clojure Hash Map

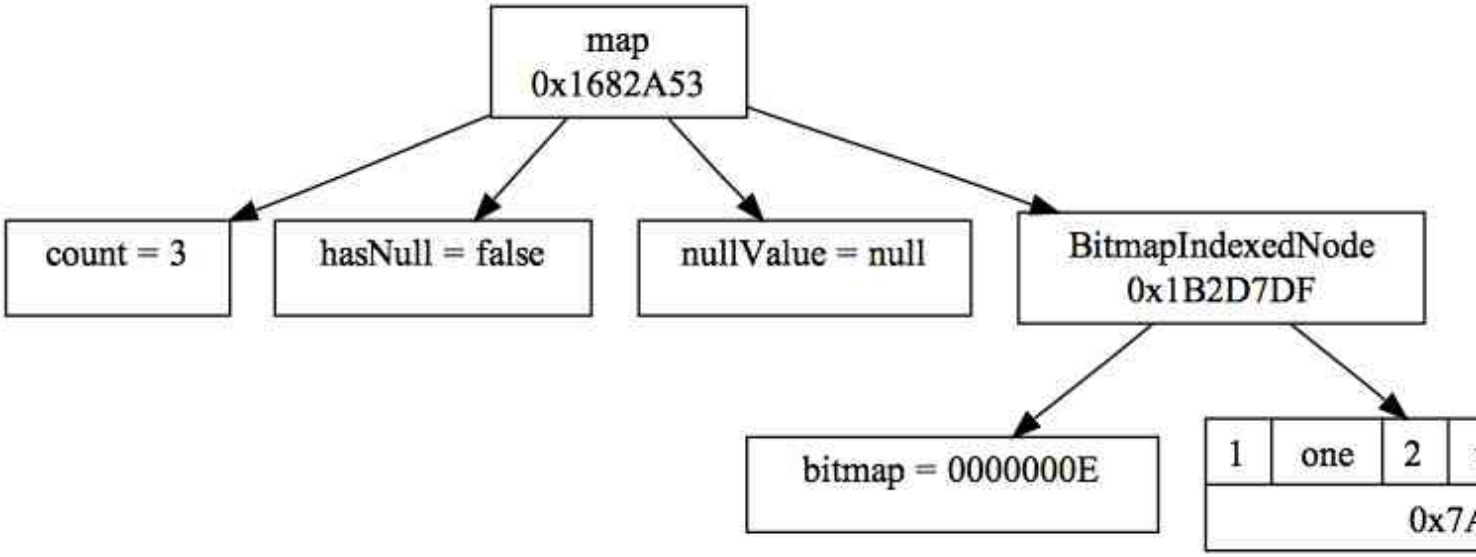

Figure 8: Clojure Hash Map from Figure 7 after assoc applied

Figure 9 shows a more realistic larger hash map that has 17 elements. I removed most of the nodes so that the figure will fit on the page. With a 32-way branching factor, this map instance only needs two levels to store its data; two levels create room for 32*32 elements.

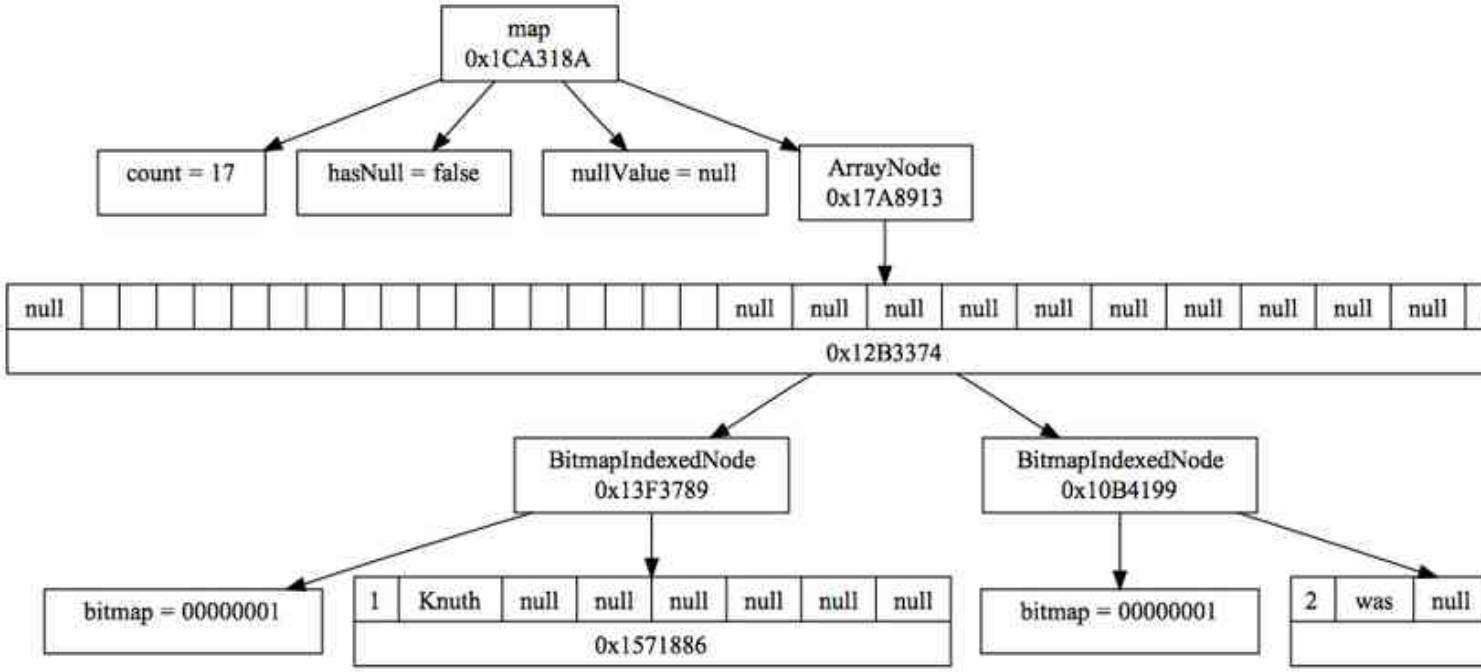

Figure 9: Larger Clojure Hash Map

If you add the key/value {18 "physics"} to this larger map, you get Figure 10. This figure shows that most of the hash map was shared. The new node is of course different, also the nodes above the new node are necessarily different. This technique is well known and is called path copying.

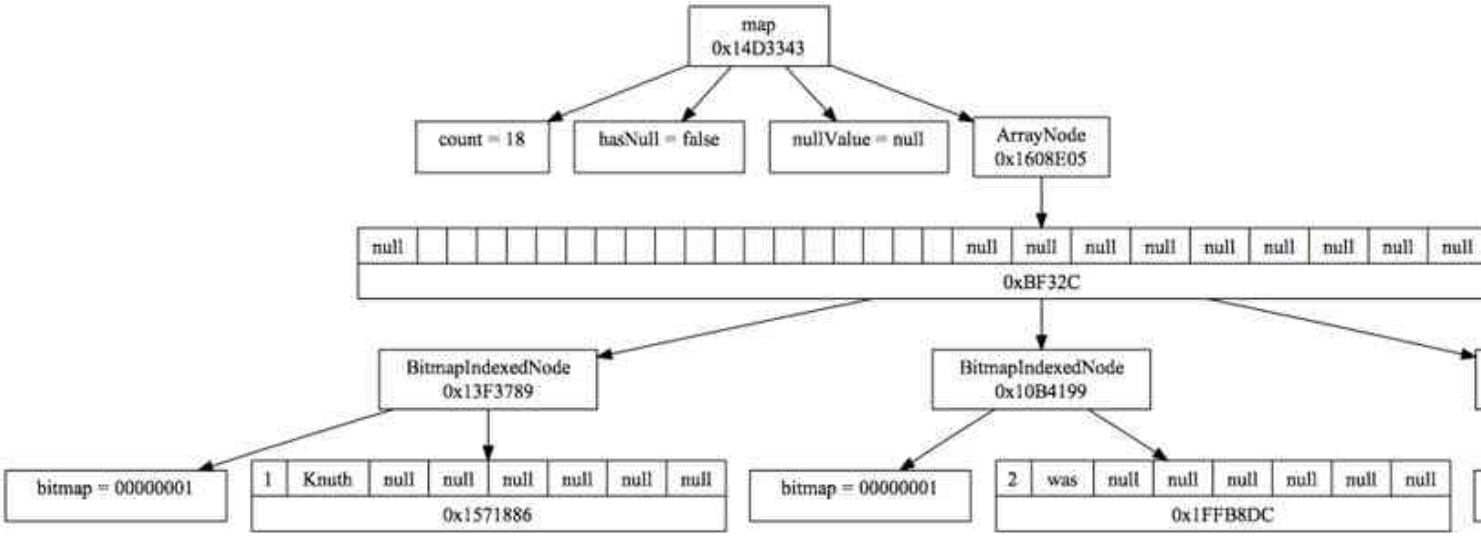

Figure 10: Clojure Hash Map from Figure 9 after assoc applied

Transient Collections

Clojure also has transient versions of the collections. These collections are allowed to change, and so can be faster in some situations. In particular, a transient is used when a collection is built up from another collection. That transient is converted to persistent when it is returned. That approach provides a nice speedup in a way that is invisible to the application code.

Conclusion

I have shown the internal representation of some Clojure collections; I believe that it is very illuminating to see the backing data for objects. It is like a physician using an MRI to see the internals of their patient. The graphs do show internal representations, and these representations will likely change with future versions of Clojure. You should not depend on the details. These graphs were made with Clojure 1.2.

The Clojure data collections are very sophisticated, and play a key role in how Clojure can both avoid mutability and also have excellent performance.

Create Unix Services with Clojure

Clojure's Affinity for Java Opens a Lot of Doors

by Aaron Bedra

> Aaron is the coauthor (with Stuart Halloway) of the forthcoming Programming Clojure, Second Edition. Here he gives a practical, hands-on experience with Clojure.

There are many reasons for turning to Clojure to solve problems. Clojure is a general purpose programing language, based on Lisp, that runs on the Java Virutual Machine. It would take up all of your attention, and a lot of your time to go through them all, so you should browse the videos for yourself. One of Clojure's selling points is its ability to interoperate with any Java code. Billions of dollars have been invested into building supporting infrastructure and libraries around the Java stack, and it is trivial for you to tap into them using Clojure. This article will dive right in to writing Cloure code with a dash of Java interop.

For this article, we will use the Leiningen build tool. There are fantastic installation instructions right on the github project page. Leinigen works on Linux, OS X, and Windows, so you should be covered. Before starting this article, make sure you have the current (1.5.2 or greater) version of Leiningen installed.

In this example we will be building an application to test the availability of websites. The goal here is to check to see if the website returns an HTTP 200 OK response. If anything other than our expected response is received, it should be noted. Let's start by creating a new project.

```
lein new pinger
```

Open your project.clj file and modify the contents to match what we are going to be working on. Be sure to update Clojure to the latest version.

```
(defproject pinger "0.0.1-SNAPSHOT"
:description "A website availability tester"
:dependencies [[org.clojure/clojure 1.3.0-beta1]])
```

Grab the dependencies by running lein deps.

```
lein deps
```

First we need to write the code that connects to a url and captures the response code. We can accomplish this by using Java's URL Library.

```
(ns pinger.core
(:import (java.net URL)))
(defn response-code [address]
(let [connection (.openConnection (URL. address))]
(doto connection
(.setRequestMethod "GET")
(.connect))
(.getResponseCode connection)))
(response-code "http://google.com")
-> 200
```

Now let's create a function that uses response-code and decides if the specified url is available. We will define available in our context as returning an HTTP 200 response code.

```
(defn available? [address]
(= 200 (response-code address)))
(available? "http://google.com")
=> true
(available? "http://google.com/badurl")
=> false
```

Next we need a way to start our program and have it check a list of urls that we care about every so often and report their availability. Let's create a main function.

```
(defn -main []
(let [addresses '("http://google.com"
"http://amazon.com"
"http://google.com/badurl")]
(while true
(doseq [address addresses]
(println (available? address)))
(Thread/sleep (* 1000 60)))))
```

In this example we create a list of addresses (two good and one bad), and use a simple while loop that never exits to simulate a never-ending program execution. It will continue to check these urls once a minute until the program is terminated. Since we are exporting a -main function, don't forget to add :gen-class to your namespace declaration.

```
(ns pinger.core
(:import (java.net URL))
(:gen-class))
```

Now that we have the fundamentals in place we need to tell leningen where our main function is located. Open up project.clj and add the :main declaration:

```
(defproject pinger "0.0.1-SNAPSHOT"
:description "A website availability tester"
:dependencies [[org.clojure/clojure "1.3.0-beta1"]]
```

```
  :main pinger.core)
```
It's time to compile our program into a jar and run it. To do this, run

```
lein uberjar
java -jar pinger-0.0.1-SNAPSHOT-standalone.jar
true
false
true
```
You should see your program start and continue to run until you press ctrl-c to stop it.

Adding real continuous loop behavior

A while loop that is always true will continue to run until terminated, but it's not really the cleanest way to obtain the result as it doesn't allow for a clean shutdown. We can use a scheduled thread pool that will start and execute the desired command in a similar fashion as the while loop, but with a much greater level of control. Create a file in the src directory called scheduler.clj and enter the following code:

```
(ns pinger.scheduler
(:import (java.util.concurrent ScheduledThreadPoolExecutor TimeUnit)))
(def ^:private num-threads 1)
(def ^:private pool (atom nil))
(defn- thread-pool []
(swap! pool (fn [p] (or p (ScheduledThreadPoolExecutor. num-threads)))))
(defn periodically
"Schedules function f to run every 'delay' milliseconds after a
delay of 'initial-delay'."
[f initial-delay delay]
(.scheduleWithFixedDelay (thread-pool)
f
initial-delay delay TimeUnit/MILLISECONDS))
(defn shutdown
"Terminates all periodic tasks."
[]
(swap! pool (fn [p] (when p (.shutdown p)))))
```
This code sets up a function called periodically that will accept a function, initial-delay, and repeated delay. It will execute the function for the first time after the initial delay then continue to execute the function with the delay specified thereafter. This will continue to run until the thread pool is shut down. Since we have a handle to the thread pool, we can do this gracefully via the shutdown function.

Let's update our application to take advantage of the scheduling code as well as make the -main function only responsible for calling a function that starts the loop.

```
(defn check []
(let [addresses '("http://google.com"
```

```
  "http://google.com/404"
  "http://amazon.com")]
  (doseq [address addresses]
  (println (available? address)))))
  (def immediately 0)
  (def every-minute (* 60 1000))
  (defn start []
  (scheduler/periodically check immediately every-minute))
  (defn stop []
  (scheduler/shutdown))
  (defn -main []
  (start))
```

Make sure to update your namespace declaration to include the scheduler code:

```
  (ns pinger.core
  (:import (java.net URL))
  (:require [pinger.scheduler :as scheduler])
  (:gen-class))
```

Not everything in the previous sample is necessary, but it makes for more readable code. Adding the start and stop functions makes it easy to work interactively from the REPL which will be a huge advantage should you choose to extend this example. Give everything one last check by running lein uberjar and executing the jar. The program should function exactly as it did before.

Logging

So far we have produced a program capable of periodically checking the availability of a list of websites. It is, however, lacking the ability to keep track of what it has done and to notify us when a site is unavailable. We can solve both of these issues with logging. There are a lot of logging options for Java applications, but for this example we will use log4j. It gives us a real logger to use, and it gives us email notification. This is great because we will have the ability to send email alerts when a website isn't available. In order to do this we will need to pull log4j and mail into our application. To make it easier to take advantage of log4j we will also pull in clojure.tools.logging. Open your project.clj file and add clojure.tools.logging, log4j, and mail:

```
  (defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester
  :dependencies [[org.clojure/clojure "1.3.0-beta1"]
  [org.clojure/tools.logging "0.1.2"]
  [log4j "1.2.16"]
  [javax.mail/mail "1.4.1"]]
  :main pinger.core)
```

and pull the dependencies in with leiningen:

```
  lein deps
```

The great part about the clojure logging library is that it will use any standard Java logging library that is on the classpath so there is no additional wiring required between log4j and your application. Create a folder in the root of your project called resources. Leiningen automatically adds the contents of this folder to the classpath, and you will need that for your log4j properties file. Create a file under the resources directory named log4j.properties and add the following contents:

```
log4j.rootLogger=info, R, email
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=logs/pinger.log
log4j.appender.R.MaxFileSize=1000KB
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{ISO8601} %-5p [%c] - %m%n
log4j.appender.email=org.apache.log4j.net.SMTPAppender
log4j.appender.email.SMTPHost=localhost
log4j.appender.email.From=system@yourapp.com
log4j.appender.email.To=recipient@yourapp.com
log4j.appender.email.Subject=[Pinger Notification] - Website Down
log4j.appender.email.threshold=error
log4j.appender.email.layout=org.apache.log4j.PatternLayout
log4j.appender.email.layout.conversionPattern=%d{ISO8601} %-5p [%c] - %m
%n
```

This sets up standard logging to pinger.log and will send an email notification for anything logged as error, which in our case is when a website doesn't respond with an HTTP 200 response or when an exception is thrown while checking the site. Make sure to change the email information to something that works in your environment.

Let's update the code and add logging. The goal here is to replace any println statements with log messages. Open core.clj, add the info and error functions from clojure.tools.logging into your namespace declaration, and create a function to record the results.

```
(ns pinger.core
(:import (java.net URL))
(:use [clojure.tools.logging :only (info error)])
(:require [pinger.scheduler :as scheduler])
(:gen-class))


...
(defn record-availability [address]
(if (available? address)
(info (str address " is responding normally"))
(error (str address " is not available"))))
```

Also update check to reflect the changes:

```
(defn check []
```

```
(let [addresses '("http://google.com"
"http://google.com/404"
"http://amazon.com")]
(doseq [address addresses]
(record-availability address))))
```

Rebuild and try your program again. You should notice a newly created logs directory that you can check for program execution. You should also notice an email come in with an error message. If you get a "connection refused" error on port 25, you will need to set up a mail transport agent on your machine to enable mail sending. You now have a way to notify people of a website failure!

Configuration

We have hard-coded our list of websites to monitor, and that simply won't work! We need a way to give a list of sites to monitor from some external source. We could use a properties file, database or webservice to accomplish this. For ease of explanation we will go with a properties file. Create a file named pinger.properties in the root of the application and add the following to it:

```
urls=http://google.com,http://amazon.com,http://google.com/404
```

We need a way to load this file in and create a collection of sites to feed into the check function. Create a file named config.clj in the src directory:

```
(ns pinger.config
(:use [clojure.java.io :only (reader)])
(:import (java.util Properties)))
(defn load-properties []
(with-open [rdr (reader "pinger.properties")]
(doto (Properties.)
(.load rdr))))
(def config (load-properties))
```

As long as pinger.properties is on the classpath, the previous example will read pinger.properties into a Java properties object. Since we don't want to do this every time we run the website checking routine, we create a var to hold the value for us. All we have left to do is get the url's attribute and put it into a list. Add the following function into the config namespace:

```
(defn urls []
(str/split (.get config "urls") #","))
```

Make sure to require clojure.string in your namespace declaration

```
(ns pinger.config
(:use [clojure.java.io :only (reader)])
(:require [clojure.string :as str])
(:import (java.util Properties)))
```

Finally, update the check function in core.clj to use the new configuration function.

```
(ns pinger.core
(:import (java.net URL))
```

```
  (:use [clojure.tools.logging :only (info error)])
  (:require [pinger.scheduler :as scheduler]
  [pinger.config :as config])
  (:gen-class))

  ...
  (defn check []
  (doseq [address (config/urls)]
  (record-availability address)))
```

Rebuild your application with leiningen and give it a try. Remember to put pinger.properties on the classpath:

```
  java -cp pinger.properties:pinger-0.0.1-standalone.jar pinger.core
```

Wrapping Up

We now have what we need to succeed. In this example we covered:

- Using Java's URL to check a website to see if it was available
- Using Java's ScheduledTheadPoolExecutor to create a periodically running task
- Using log4j with clojure.tools.logging to send error notifications
- Using Java's property system for configuration
- Using leiningen to create standalone executable jars

There are quite a few things you could do to expand on this example. We could redefine what it means for a website to be available by adding requirements for certain HTML elements to be present, or for the response to return in a certain time to cover an SLA. Try adding to this example and see what you can come up with.