READY, SET, CLOJURE!

Building Beautiful Abstractions with Lisp

Aaron Bedra

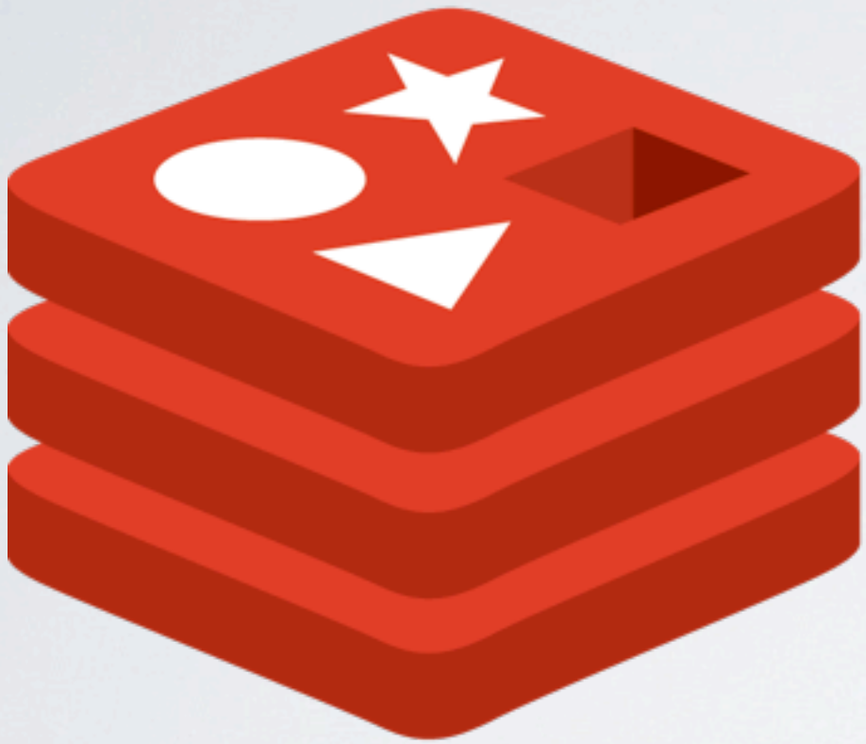
WHY CLOJURE?

- Built on top of a solid foundation (JVM) (JS)
- Full interoperability with the host platform(s)*
- Immutable by default
- Well controlled mutation when necessary
- Mostly functional
- Brings all the expressiveness and power of LISP

WHAT WE WILL COVER

- Clojure basics
- Clojure's built in test framework (clojure.test)
- Java Interoperability
- Polymorphism
- Macros
- Leiningen

WHAT PROBLEM ARE WE TRYING TO SOLVE?



redis

SURELY SOMEBODY ALREADY SOLVED THIS PROBLEM?

CURRENT LIBRARIES

- Accession
- clj-redis
- Carmine
- redis-clojure
- labs-redis-clojure

GETTING STARTED

```
$ lein new yow
$ cd yow
$ mkdir script
$ touch script/bootstrap
$ chmod +x script/bootstrap
```

```
REDIS_VERSION=2.6.4
URL=http://redis.googlecode.com/files/
$REDIS_VERSION.tar.gz

if [ ! -d "vendor" ]; then
    mkdir vendor
    pushd vendor
    if which wget > /dev/null; then
        wget $URL
    else
        curl -O $URL
    fi
    tar xvf $REDIS_VERSION.tar.gz
    pushd $REDIS_VERSION
    make
    popd
    popd
fi
```

```
if [ ! -d "config" ]; then
    mkdir config
    cp vendor/$REDIS_VERSION/redis.conf config/
fi

if [ ! -d "bin" ]; then
    mkdir bin
    cp vendor/$REDIS_VERSION/src/redis-server bin/
    cp vendor/$REDIS_VERSION/src/redis-cli bin/
fi
```

```
$ tree
|-- README.md
|-- bin
|   |-- redis-cli
|   `-- redis-server
|-- config
|   `-- redis.conf
|-- project.clj
|-- script
|   |-- bootstrap
|-- src
|   `-- yow
|       `-- core.clj
|-- test
|   `-- yow
|       `-- core_test.clj
`-- vendor
    `-- redis-2.6.4
```

Leiningen

```
(defproject yow "0.1.0-SNAPSHOT"
  :description "A Redis Adapter"
  :url "http://github.com/abedra/yow-2012"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.4.0"]
                 [org.clojure/data.json "0.2.0"]])
```

THE REDIS UNIFIED PROTOCOL

```
*<number of arguments> CR LF
$<number of bytes of argument 1> CR LF
<argument data> CR LF
...
$<number of bytes of argument N> CR LF
<argument data> CR LF

*3
$3
SET
$5
mykey
$7
myvalue

"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"
```

CLOJURE'S BUILT IN TEST FRAMEWORK

```clojure
(defn hello
  [name]
  (str "Hello " name "!"))

(deftest test-hello
  (is (= "Hello Yow!" (hello "Yow"))))
```

LETS TURN THE PROTOCOL EXAMPLE INTO A TEST!

```
(deftest test-command
  (testing "Produces proper SET command"
    (is (= "*3\r\n$3\r\nSET\r\n$5\r\nmykey\r
\n$7\r\nmyvalue\r\n"
           (command "set"
                    "mykey"
                    "myvalue")))))
```

NOW WE CAN CREATE THE IMPLEMENTATION

```clojure
(def crlf "\r\n")

(defn command
  [name & args]
  (str "*"
       (inc (count args)) crlf
       "$" (count name) crlf
       (str/upper-case name) crlf
       (str/join crlf
         (map (fn [x]
           (str "$" (count x) crlf x)) args))
       crlf))
```

```
Testing yow.core-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

JAVA INTEROP IN CLOJURE

```
user> (import '(java.net Socket))
;-> java.net.Socket
user> (Socket.)
;-> #<Socket Socket[unconnected]>
user> (Socket. "localhost" 6379)
;-> #<Socket Socket[addr=localhost/
127.0.0.1,port=6379,localport=45284]>
user> (def s (Socket. "localhost" 6379))
;-> #'user/s
user> (.setKeepAlive s true)
;-> nil
user> (.getKeepAlive s)
;-> true
```

HELLO REDIS!

```clojure
(defn- socket
  []
  (doto (Socket. "localhost" 6379)
    (.setTcpNoDelay true)
    (.setKeepAlive true)))

(defn request
  [query]
  (with-open [socket (socket)
              in (DataInputStream.
                   (BufferedInputStream.
                    (.getInputStream socket)))
              out (.getOutputStream socket)]
    (.write out (.getBytes (apply str query)))
    (println in)))
```

```
user> (request (command "set" "foo" "bar"))
;-> #<DataInputStream java.io.DataInputStream@580a00fd>
```

```
$ bin/redis-cli get foo
"bar"
```

POLYMORPHISM IN CLOJURE

EXAMPLES

- Most core datastructures implemented using Java Interfaces
- Interface generation via `proxy`
- Protocols
- Multimethods

```clojure
(defmulti encounter
  (fn [x y]
    [(:Species x) (:Species y)]))

(defmethod encounter [:Bunny :Lion]   [b l] :run-away)
(defmethod encounter [:Lion :Bunny]   [b l] :eat)
(defmethod encounter [:Lion :Lion]    [b l] :fight)
(defmethod encounter [:Bunny :Bunny] [b l] :mate)
```

```
(def b1 {:Species :Bunny :other :stuff})
(def b2 {:Species :Bunny :other :stuff})
(def l1 {:Species :Lion :other :stuff})
(def l2 {:Species :Lion :other :stuff})

(encounter b1 b2)
;-> :mate
(encounter b1 l1)
;-> :run-away
(encounter l1 b1)
;-> :eat
(encounter l1 l2)
;-> :fight
```

THE REDIS RESPONSE STRUCTURE

Redis will reply to commands with different kinds of replies. It is possible to check the kind of reply from the first byte sent by the server:

A single line reply first byte will be "+"
An error message first byte will be "-"
An integer first byte will be ":"
A bulk reply first byte will be "$"
A multi-bulk first byte will be "*"

Redis will reply to commands with different kinds of replies. It is possible to check the kind of reply from the first byte sent by the server:

A single line reply **first byte** will be "+"
An error message **first byte** will be "-"
An integer **first byte** will be ":"
A bulk reply **first byte** will be "$"
A multi-bulk **first byte** will be "*"

```
(defmulti response
  (fn [in] (char (.readByte in))))

(defmethod response \- [in]
  (.readLine in))

(defmethod response \+ [in]
  (.readLine in))

(defmethod response \: [in]
  (Long/parseLong (.readLine in)))

(defmethod response \$ [in]
  (.readLine in)
  (.readLine in))

(defmethod response \* [in]
  (throw (UnsupportedOperationException.
           "Not Yet Implemented")))
```

```
user> (request (command "set" "foo" "bar"))
;-> "OK"
```

```
user> (request (command "get" "foo"))
;-> "bar"
```

```clojure
;; Examples taken from http://try.redis-db.com/
(deftest test-basic-interaction
  (testing "SET then GET"
    (is (= "OK" (request (command "set" "server:name" "fido"))))
    (is (= "fido" (request (command "get" "server:name")))))
  (testing "INCR"
    (request (command "set" "connections" "10"))
    (is (= 11 (request (command "incr" "connections")))))
  (testing "DEL"
    (is (= 1 (request (command "del" "connections"))))))
```

```
Testing yow.core-test

Ran 2 tests containing 5 assertions.
0 failures, 0 errors.
```

REDIS.IO/COMMANDS

```clojure
(ns yow.commands
  (:use [clojure.data.json :only (read-str)]))

(defn fetch-redis-commands
  []
  (map first
   (read-str
    (slurp "https://raw.github.com/antirez/redis-doc/master/
commands.json"))))

user> (count (fetch-redis-commands))
;-> 144
```

ZREM ZREMRANGEBYRANK PUNSUBSCRIBE BRPOP BITCOUNT SET PEXPIREAT
FLUSHDB BGSAVE ZRANGE SLOWLOG SCARD HDEL HSETNX STRLEN CONFIG SET
HEXISTS SMOVE SUNIONSTORE ZINCRBY CONFIG RESETSTAT LINSERT
BRPOPLPUSH ECHO PSETEX LPOP SMEMBERS LPUSH ZRANK LINDEX RPOPLPUSH
DECRBY ZREVRANGEBYSCORE BLPOP ZADD SREM GETRANGE RENAMENX AUTH
HINCRBYFLOAT SINTER SDIFFSTORE LLEN MGET SUBSCRIBE ZCARD SETBIT
MIGRATE INCRBY DEL GETSET SETNX DEBUG OBJECT TTL RPUSH ZUNIONSTORE
RPUSHX HLEN TIME LREM INFO SLAVEOF HGET RESTORE LTRIM SADD BITOP
WATCH PUBLISH PEXPIRE QUIT SCRIPT FLUSH DECR EVALSHA HMGET LRANGE
EXEC SCRIPT EXISTS INCRBYFLOAT UNSUBSCRIBE BGREWRITEAOF MOVE PING
EXPIREAT SRANDMEMBER LPUSHX HGETALL LASTSAVE SCRIPT KILL HINCRBY
CLIENT KILL CLIENT LIST INCR ZREVRANGE PERSIST KEYS DUMP SETEX
ZCOUNT MSET ZREVRANK LSET UNWATCH SHUTDOWN GET SISMEMBER GETBIT
CONFIG GET SINTERSTORE ZRANGEBYSCORE ZSCORE SDIFF MULTI MONITOR
HVALS DEBUG SEGFAULT PSUBSCRIBE HSET APPEND TYPE SETRANGE SYNC
SCRIPT LOAD EXISTS EVAL SELECT SUNION HKEYS RANDOMKEY PTTL
FLUSHALL HMSET SAVE DISCARD SPOP SORT ZREMRANGEBYSCORE RENAME
RPOP EXPIRE ZINTERSTORE MSETNX DBSIZE OBJECT

```
"BITOP": {
    "summary": "Perform bitwise operations between strings",
    "complexity": "O(N)",
    "arguments": [
        {
            "name": "operation",
            "type": "string"
        },
        {
            "name": "destkey",
            "type": "key"
        },
        {
            "name": "key",
            "type": "key",
            "multiple": true
        }
    ],
    "since": "2.6.0",
    "group": "string"
}
```

HOW SHOULD WE IMPLEMENT ALL 144 COMMANDS?

DO WE WRITE 144 FUNCTIONS?

HELL NO! WE ABSTRACT!

CONSIDER THE FOLLOWING DSL...

```
(defcommands
  (set  [key value])
  (get  [key])
  (incr [key])
  (del  [key & keys]))
```

CLOJURE HAS A VERY POWERFUL MACRO SYSTEM

YOU CAN ESSENTIALLY EXTEND THE COMPILER

BUT REMEMBER THE FIRST RULE OF MACRO CLUB!

"Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary."

-Paul Graham, "Beating the Averages"

```clojure
;; clojuredocs.org/clojure_core/clojure.core/defmacro
(defmacro unless [pred a b]
  `(if (not ~pred) ~a ~b))

;; usage:
(unless false
  (println "Will print")
  (println "Will not print"))
```

```clojure
user> (macroexpand-1
        '(unless false
           (println "Will print")
           (println "Will not print")))
;; (if (clojure.core/not false)
;;   (println "Will print")
;;   (println "Will not print"))
```

LETS BRING OUR DSL TO LIFE

```clojure
(defn- parameters
  [params]
  (let [[args varargs] (split-with #(not= '& %) params)]
    (conj (vec args) (last varargs))))

(defmacro defcommand
  [name params]
  (let [com (str name)
        p (parameters params)]
    `(defn ~name ~params
       (apply command ~com ~@p))))
```

```
user> (macroexpand-1 '(defcommand set [key value]))
;; (clojure.core/defn set
;;   [key value]
;;   (clojure.core/apply
;;     yow.core/command "set" key value nil))

user> (macroexpand-1 '(defcommand del [key & keys]))
;; (clojure.core/defn del
;;   [key & keys]
;;   (clojure.core/apply
;;     yow.core/command "del" key keys))
```

```clojure
(defmacro defqueries
  [& queries]
  `(do ~@(map (fn [q] `(defquery ~@q)) queries)))

user> (macroexpand-1
       '(defcommands (set [set value]) (del [key & keys])))
;; (do
;;   (yow.core/defcommand set [set value])
;;   (yow.core/defcommand del [key & keys]))

user> (clojure.walk/macroexpand-all
       '(defcommands (set [set value]) (del [key & keys])))
;; (do
;;   (def set
;;     (fn* ([set value]
;;       (clojure.core/apply
;;         yow.core/command "set" set value nil))))
;;   (def del
;;     (fn* ([key & keys]
;;       (clojure.core/apply
;;         yow.core/command "del" key keys)))))
```

CLEANING UP

```clojure
(ns yow.core
  (:refer-clojure :exclude [set get])
  (:require [clojure.string :as str])
  (:import (java.net Socket)
           (java.io BufferedInputStream DataInputStream)))

;; Prefer require over use so that you don't have to propagate the
;; :refer-clojure into every namespace that pulls the library in
user> (require '[yow.core :as redis])
;-> nil
user> (redis/request (redis/set "foo" "bar"))
;-> "OK"
user> (redis/request (redis/get "foo"))
;-> "bar"
```

WITH THIS WE'RE OFF TO A GREAT START!

WE'VE CREATED A REDIS INTERACTION LIBRARY IN UNDER 80 LINES OF CODE

BUT IT'S FAR FROM FINISHED

YOU WILL HAVE TO USE YOUR NEW CLOJURE POWERS TO COMPLETE THIS EXERCISE

MAY THE ( ) BE WITH YOU

REFERENCES

- clojure.org
- redis.io
- github.com/abedra/accession
- github.com/abedra/yow-2012
- pragprog.com/book/shcloj2/programming-clojure