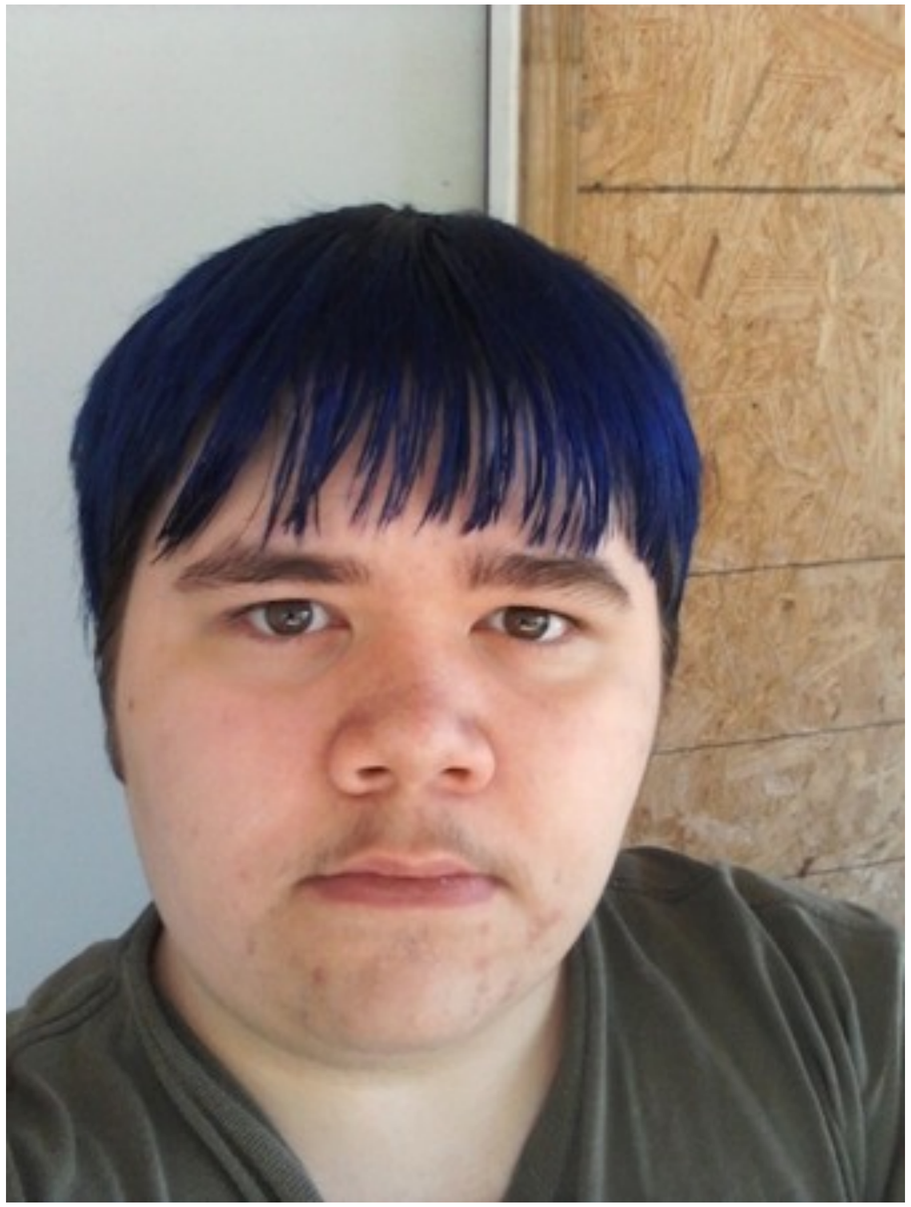
Clojail

Life in the Clojure Prison

Me

- Clojure programmer for around 3 years.
- Wrote http://tryclj.com (Try Clojure).
- Is writing a book called Meet Clojure.
- Is the youngest person in this room.
- Is a Stuart Sierra groupie.



Code is dangerous!

- If code weren't dangerous, you wouldn't be able to
  - read/write to the file system.
  - talk to the internet.
  - do pretty much anything useful at all.
- Dangerous is good.

How often do you think of it like that?

- You are your code’s sandbox.
    - You control everything that happens.
    - It’s why you don’t usually care about sandboxing.
- We almost never need or want to allow people to evaluate code on our machines...

But Clojure makes it so easy

```
user> (eval (read-string "(+ 3 3)"))
6
```

But what about this?

```
[13:05:31] <Raynes> &(+ 3 3)
[13:05:32] <lazybot> ⇒ 6
[13:05:34] <Raynes> &(System/exit 0)
[13:05:35] * lazybot has left IRC (you've killed him)
```

It gets worse

```
[13:15:40] <Raynes> &(map #(.delete %) (file-seq (System/getProperty "user.home")))
[13:15:41] <lazybot> deletin' ur datas...
```

We need to be more cautious!

IM HIT!
ICANHASCHEEZBURGER.COM

We need a sandbox to keep us safe

A sandbox can prevent...

- I/O, such as
  - Interaction with the file system.
  - Interaction with the internet.
- The execution of arbitrary programs.
- Destruction of the JVM.

Step 1: The JVM

The JVM sandbox

- is thorough.
- has been around since before I learned to walk.
- prevents I/O.
- denies access to certain methods and classes.
- is customizable.

- Basically...

It stops this from happening

```
user=> (System/exit 0)
       [cake] error connecting to socket
```

And this stuff

```
user=> (slurp "http://hackmycreditcard.com")
"processing payments"

user=> (-> "user.home"
           System/getProperty
           (java.io.File. ".emacs")
           .delete)
true
```

Among other things...

It saves your computer from evil...

Unfortunately, it isn't enough for every use-case.

Step 2: Clojure

The hardest part of sandboxing is sandboxing the *state* of Clojure

What if they rebind things?

```
user=> (def + -)
#'repl-1/+
user=> (+ 10 10)
0
```

What about infinite loops?

```
user=> (loop [] (recur))
```

Facing these problems

- Try Clojure is a Clojure REPL in your browser.
- 4Clojure is Clojure koans for your browser.
- Lazybot is a Clojure REPL in your IRC channel.

So, what do figure out what we need to do!

Rip apart code

- Look at
  - namespaces
  - symbols
  - classes
  - packages
  - vars
  - etc

def

- Need to keep defs from being abused because they can
  - rebind things in the namespace.
  - be used to abuse memory.

Loops

- Timeouts!

Threads

- They can be used to avoid timeouts.
- They must die.

The dot (.) special form

- Is evil because you can abuse Clojure's Java classes.
  - Observe:

```
user=> (.intern *ns* '+)
#'user/+
```

- Cannot be gotten rid of.
- Cannot be rebound because it is a special form.
- Must be replaced entirely.

That's what is needed to make it safe, but why stop there?

Extensibility

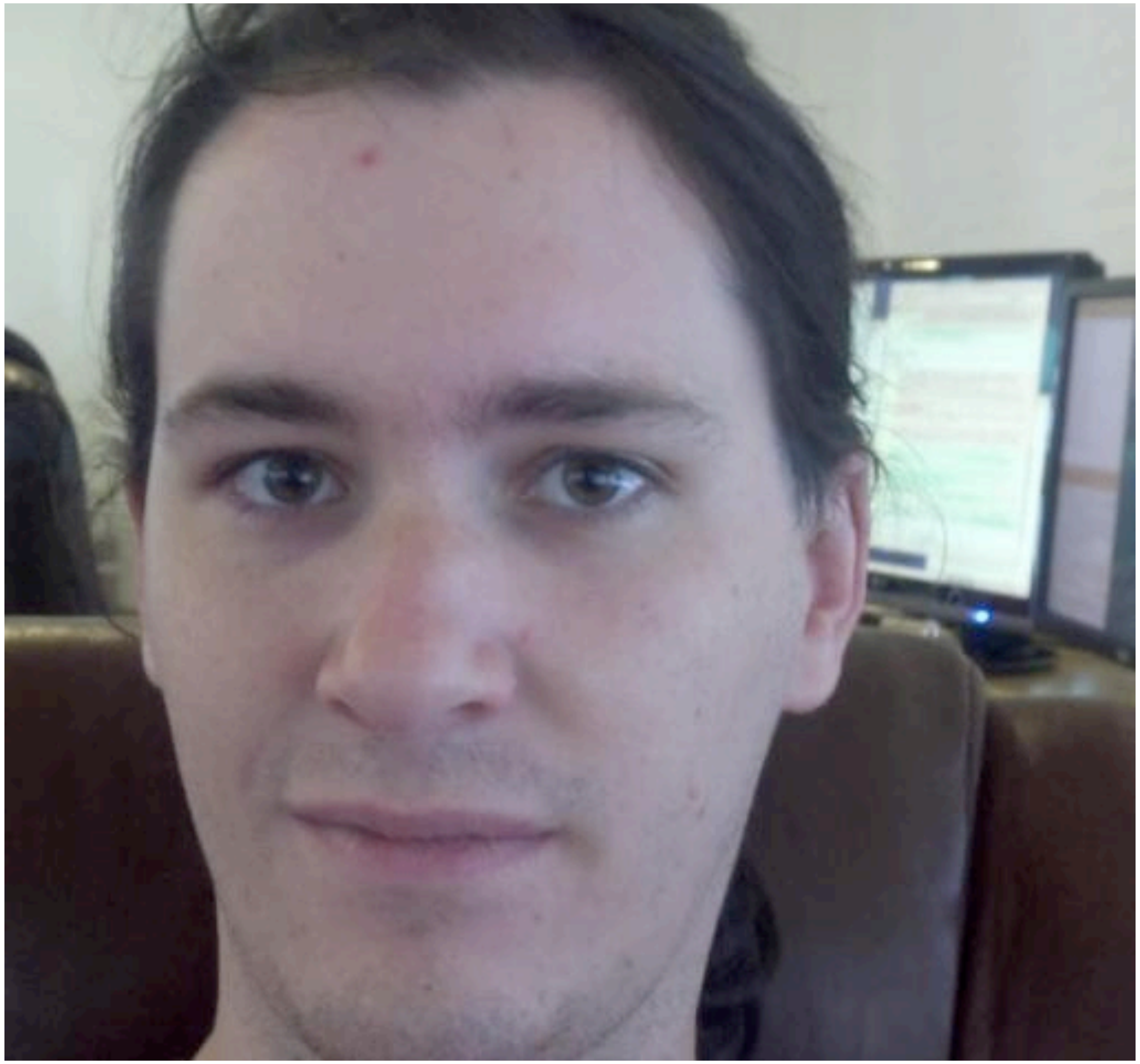
- Safety isn't the only concern.
- Customized evaluation contexts.
    - Allow users to block whatever they want.
    - 4Clojure is built on this.

Clojail

An all you can eat sandboxing library

written by this guy

Anthony Grimes



and this guy

Alan Malloy

inspired by ideas from this guy
Heinz N. Gies

which were inspired by an IRC bot from this guy
Kevin Downey

for you guys.

Well, mostly for me, but you guys can use it too!

It can...

- take advantage of the JVM sandbox.
- sandbox the Clojure side of things.
- do very selective blacklisting.
- basically do everything we’ve talked about.

Using it

Leiningen

Or

Easy enough, right?

```
(defproject scared "0.1.0"
  :description "Feeling vulnerable"
  :dependencies [[clojail "0.5.0"]])
```

Open For Business

```
user=> (use 'clojail.core)
user=> (def sb (sandbox #{}))
#'user/sb
user=> (sb '(+ 3 3))
6
user=> (sb '(System/exit 0))
AccessControlException access denied ...
user=> (def sb (sandbox #{} :jvm false))
#'user/sb
user=> (sb '(System/exit 0))
```

Open For Business

```
user=> (use 'clojail.core)
user=> (def sb (sandbox #{}))
#'user/sb
user=> (sb '(+ 3 3))
6
user=> (sb '(System/exit 0))
AccessControlException access denied ...
user=> (def sb (sandbox #{} :jvm false))
#'user/sb
user=> (sb '(System/exit 0))
```

Open For Business

```
user=> (use 'clojail.core)
user=> (def sb (sandbox #{}))
#'user/sb
user=> (sb '(+ 3 3))
6
user=> (sb '(System/exit 0))
AccessControlException access denied ...
user=> (def sb (sandbox #{} :jvm false))
#'user/sb
user=> (sb '(System/exit 0))
```

Blocking Clojure

- ‘sandbox’ takes a set of things like Java classes, packages, symbols, sets, namespaces.
- These are called ‘testers’.

Math Can Be Difficult

```
user=> (def sb (sandbox #{'+ '- Math}))
#'user/sb
user=> (sb '(+ 3 3))
SecurityException You tripped the alarm! + is bad!
user=> (sb '(- 4 2))
SecurityException You tripped the alarm! - is bad!
user=> (sb '(Math/cos 10.3))
SecurityException You tripped the alarm! class java.lang.Math
is bad!
```

Math Can Be Difficult

```
user=> (def sb (sandbox #{'+ '- Math}))
#'user/sb
user=> (sb '(+ 3 3))
SecurityException You tripped the alarm! + is bad!
user=> (sb '(- 4 2))
SecurityException You tripped the alarm! - is bad!
user=> (sb '(Math/cos 10.3))
SecurityException You tripped the alarm! class java.lang.Math
is bad!
```

Security Blanket

```
user=> (use 'clojail.testers)
nil
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (sb '(do (future (range)) nil))
SecurityException You tripped the alarm! future-call is bad!
```

Security Blanket

```
user=> (use 'clojail.testers)
nil
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (sb '(do (future (range)) nil))
SecurityException You tripped the alarm! future-call is bad!
```

```
user=> (use 'clojail.testers)
nil
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (sb '(do (future (range)) nil))
SecurityException You tripped the alarm! future-call is bad
```

```
user=> (def sb (sandbox*))
#'user/sb
user=> (sb '(+ 3 3) secure-tester)
6
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (sb '(+ 3 3))
6
```

```
user=> (def sb (sandbox*))
#'user/sb
user=> (sb '(+ 3 3) secure-tester)
6
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (sb '(+ 3 3))
6
```

```
user=> (sb '(loop [] (recur)))
TimeoutException Execution timed out.  clojail.core/thunk-timeout
(core.clj:57)
user=> (def sb (sandbox secure-tester :timeout 5000))
#'user/sb
user=> (sb '(loop [] (recur)))
TimeoutException Execution timed out.  clojail.core/thunk-timeout
(core.clj:57)
```

Wait for it...

```
user=> (sb '(loop [] (recur)))
TimeoutException Execution timed out.  clojail.core/thunk-timeout
(core.clj:57)
user=> (def sb (sandbox secure-tester :timeout 5000))
#'user/sb
user=> (sb '(loop [] (recur)))
TimeoutException Execution timed out.  clojail.core/thunk-timeout
(core.clj:57)
```

Definitely

```
user=> (def sb (sandbox secure-tester-without-def))
#'user/sb
user=> (doseq [name '[a b c d e f]]
         (sb `(def ~name 0)))
nil
user=> (sb 'e)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: a in this context, compiling:(NO_SOURCE_PATH:0)
user=> (sb 'f)
0
user=> (sb (cons 'do (map #(list 'def % 0)
                          '[a b c d e f])))
#'sandbox207/f
user=> (sb 'f)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: f in this context, compiling:(NO_SOURCE_PATH:0)
```

Definitely

```
user=> (def sb (sandbox secure-tester-without-def))
#'user/sb
user=> (doseq [name '[a b c d e f]]
         (sb `(def ~name 0)))
nil
user=> (sb 'e)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: a in this context, compiling:(NO_SOURCE_PATH:0)
user=> (sb 'f)
0
user=> (sb (cons 'do (map #(list 'def % 0)
                          '[a b c d e f])))
#'sandbox207/f
user=> (sb 'f)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: f in this context, compiling:(NO_SOURCE_PATH:0)
```

Definitely

```
user=> (def sb (sandbox secure-tester-without-def))
#'user/sb
user=> (doseq [name '[a b c d e f]]
         (sb `(def ~name 0)))
nil
user=> (sb 'e)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: a in this context, compiling:(NO_SOURCE_PATH:0)
user=> (sb 'f)
0
```

```
user=> (sb (cons 'do (map #(list 'def % 0)
                          '[a b c d e f])))
#'sandbox207/f
user=> (sb 'f)
user=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: f in this context, compiling:(NO_SOURCE_PATH:0)
```

A Little Space

```
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (def sb-two (sandbox secure-tester))
#'user/sb-two
user=> (sb '*ns*)
#<Namespace sandbox376>
user=> (sb-two '*ns*)
#<Namespace sandbox379>
user=> (def sb (sandbox secure-tester
                        :namespace 'my-ns))
#'user/sb
user=> (sb '*ns*)
#<Namespace my-ns>
```

A Little Space

```
user=> (def sb (sandbox secure-tester))
#'user/sb
user=> (def sb-two (sandbox secure-tester))
#'user/sb-two
user=> (sb '*ns*)
#<Namespace sandbox376>
user=> (sb-two '*ns*)
#<Namespace sandbox379>
user=> (def sb (sandbox secure-tester
                        :namespace 'my-ns))
#'user/sb
user=> (sb '*ns*)
#<Namespace my-ns>
```

Secure Dispensations

```
user=> (use 'clojail.jvm)
nil
user=> (def con
          (-> (java.io.FilePermission. "foo"
                                       "read,write")
              permissions
              domain
              context))
#'user/con
user=> (def sb (sandbox secure-tester :context con))
#'user/sb
user=> (sb '(do (spit "foo" "Hi!") (slurp "foo")))
"Hi!"
user=> (jvm-sandbox #(do (spit "foo" "Hi!") (slurp "foo")) con)
"Hi!"
```

Secure Dispensations

```
user=> (use 'clojail.jvm)
nil
user=> (def con
         (-> (java.io.FilePermission. "foo"
                                      "read,write")
             permissions
             domain
             context))
#'user/con
user=> (def sb (sandbox secure-tester :context con))
#'user/sb
user=> (sb '(do (spit "foo" "Hi!") (slurp "foo")))
"Hi!"
user=> (jvm-sandbox #(do (spit "foo" "Hi!") (slurp "foo")) con)
"Hi!"
```

Secure Dispensations

```
user=> (use 'clojail.jvm)
nil
user=> (def con
         (-> (java.io.FilePermission. "foo"
                                      "read,write")
             permissions
             domain
             context))
#'user/con
user=> (def sb (sandbox secure-tester :context con))
#'user/sb
user=> (sb '(do (spit "foo" "Hi!") (slurp "foo")))
"Hi!"
user=> (jvm-sandbox #(do (spit "foo" "Hi!") (slurp "foo")) con)
"Hi!"
```

Laying The Foundation

```
user=> (def sb (sandbox secure-tester
                        :init '(def foo "foo")))
#'user/sb
user=> (sb 'foo)
"foo"
user=> (def init '(require '[clojure.string :as string]))
#'user/init
user=> (def sb (sandbox secure-tester :init init))
#'user/sb
user=> (sb '(string/join [\a \b \c]))
"abc"
```

Laying The Foundation

```
user=> (def sb (sandbox secure-tester
                        :init '(def foo "foo")))
#'user/sb
user=> (sb 'foo)
"foo"
```

```
user=> (def init '(require '[clojure.string :as string]))
#'user/init
user=> (def sb (sandbox secure-tester :init init))
#'user/sb
user=> (sb '(string/join [\a \b \c]))
"abc"
```

A Simple Binding Spell

```
user=> (let [writer (java.io.StringWriter.)]
         (sb '(println "Hello, world!") {#'*out* writer})
         (println (str writer)))
Hello, world!
```

So that’s Clojail.

Mmmm, implementation details!

Let's look at the individual pieces that make up the sandbox, starting with check-form

Border Patrol

```
(defn check-form [form tester nspace]
  (some tester (separate form nspace)))
```

Check ALL The Things!

```clojure
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
            (let [resolved-s (safe-resolve % nspace)
                  s-meta (meta resolved-s)]
              (if s-meta
                [resolved-s
                 ((juxt (comp symbol str :ns) :ns :name)
                 s-meta)]
                (let [[bottom] (map symbol (.split (str %) "/"))
                      resolved-s (safe-resolve bottom nspace)]
                  (if (class? resolved-s)
                    [resolved-s %]
                    %))))
            %)
         (flatten-all (collify (macroexpand-most s)))))))
```

Check ALL The Things!

```
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
             (let [resolved-s (safe-resolve % nspace)
                   s-meta (meta resolved-s)]
               (if s-meta
                 [resolved-s
                  ((juxt (comp symbol str :ns) :ns :name)
                  s-meta)]
                 (let [[bottom] (map symbol (.split (str %) "/"))
                       resolved-s (safe-resolve bottom nspace)]
                   (if (class? resolved-s)
                     [resolved-s %]
                     %))))
             %)
         (flatten-all (collify (macroexpand-most s)))))))
```

Check ALL The Things!

```
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
            (let [resolved-s (safe-resolve % nspace)
                  s-meta (meta resolved-s)]
              (if s-meta
                [resolved-s
                 ((juxt (comp symbol str :ns) :ns :name)
                 s-meta)]
                (let [[bottom] (map symbol (.split (str %) "/"))
                      resolved-s (safe-resolve bottom nspace)]
                  (if (class? resolved-s)
                    [resolved-s %]
                    %))))
            %)
         (flatten-all (collify (macroexpand-most s)))))))
```

Check ALL The Things!

```
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
            (let [resolved-s (safe-resolve % nspace)
                  s-meta (meta resolved-s)]
              (if s-meta
                [resolved-s
                 ((juxt (comp symbol str :ns) :ns :name)
                 s-meta)]
                (let [[bottom] (map symbol (.split (str %) "/"))
                      resolved-s (safe-resolve bottom nspace)]
                  (if (class? resolved-s)
                    [resolved-s %]
                    %))))
            %)
         (flatten-all (collify (macroexpand-most s)))))))
```

Check ALL The Things!

```
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
            (let [resolved-s (safe-resolve % nspace)
                  s-meta (meta resolved-s)]
              (if s-meta
                [resolved-s
                 ((juxt (comp symbol str :ns) :ns :name)
                 s-meta)]
                (let [[bottom] (map symbol (.split (str %) "/"))
                      resolved-s (safe-resolve bottom nspace)]
                  (if (class? resolved-s)
                    [resolved-s %]
                    %))))
            %)
         (flatten-all (collify (macroexpand-most s)))))))
```

Check ALL The Things!

```
(defn- separate [s nspace]
  (set
   (flatten-all
    (map #(if (symbol? %)
            (let [resolved-s (safe-resolve % nspace)
                  s-meta (meta resolved-s)]
              (if s-meta
                [resolved-s
                 ((juxt (comp symbol str :ns) :ns :name)
                 s-meta)]
                (let [[bottom] (map symbol (.split (str %) "/"))
                      resolved-s (safe-resolve bottom nspace)]
                  (if (class? resolved-s)
                    [resolved-s %]
                    %))))
            %)
         (flatten-all (collify (macroexpand-most s)))))))
```

So that’s how your non-interop code is handled. But what about your Java interop code?

- Clojail sandboxes in two stages.

  1. It checks the code before evaluation.

  2. Modifies the code so that it can sandbox things that couldn’t be checked/it could have missed before evaluation

- We will replace the ‘.’ special form with our specialized ‘dot’ macro.

- This is just a simple recursive walk. It’s what ‘dot’ does that is interesting.

The Interop Police

```
(defn- make-dot [tester-str]
  `(defmacro ~'dot [object# method# & args#]
     `(let [~'tester-obj# (binding [*read-eval* true]
                             (read-string ~~tester-str))
            ~'obj# ~object#
            ~'obj-class# (class ~'obj#)]
        (if-let [~'bad# (some ~'tester-obj#
                              [~'obj-class#
                               ~'obj#
                               (.getPackage ~'obj-class#)])]
          (security-exception ~'bad#)
          (. ~object# ~method# ~@args#)))))
```

The Interop Police

```
(defn- make-dot [tester-str]
  `(defmacro ~'dot [object# method# & args#]
     `(let [~'tester-obj# (binding [*read-eval* true]
                             (read-string ~~tester-str))
            ~'obj# ~object#
            ~'obj-class# (class ~'obj#)]
        (if-let [~'bad# (some ~'tester-obj#
                              [~'obj-class#
                               ~'obj#
                               (.getPackage ~'obj-class#)])]
          (security-exception ~'bad#)
          (. ~object# ~method# ~@args#)))))
```

And that's how dot is handled. But what about timeouts?

Patience...

```
(defn thunk-timeout
  ...
  ([thunk time unit tg]
     (let [task (FutureTask. thunk)
           thr (if tg (Thread. tg task) (Thread. task))]
       (try
         (.start thr)
         (.get task time (or (uglify-time-unit unit) unit))
         (catch TimeoutException e
           (.cancel task true)
           (.stop thr)
           (throw (TimeoutException. "Execution timed out.")))
         (catch Exception e
           (.cancel task true)
           (.stop thr)
           (throw e))
         (finally (when tg (.stop tg)))))))
```

Patience...

```
(defn thunk-timeout
  ...
  ([thunk time unit tg]
     (let [task (FutureTask. thunk)
           thr (if tg (Thread. tg task) (Thread. task))]
       (try
         (.start thr)
         (.get task time (or (uglify-time-unit unit) unit))
         (catch TimeoutException e
           (.cancel task true)
           (.stop thr)
           (throw (TimeoutException. "Execution timed out.")))
         (catch Exception e
           (.cancel task true)
           (.stop thr)
           (throw e))
         (finally (when tg (.stop tg)))))))
```

Patience...

```
(defn thunk-timeout
  ...
  ([thunk time unit tg]
     (let [task (FutureTask. thunk)
           thr (if tg (Thread. tg task) (Thread. task))]
       (try
         (.start thr)
         (.get task time (or (uglify-time-unit unit) unit))
         (catch TimeoutException e
           (.cancel task true)
           (.stop thr)
           (throw (TimeoutException. "Execution timed out.")))
         (catch Exception e
           (.cancel task true)
           (.stop thr)
           (throw e))
         (finally (when tg (.stop tg)))))))
```

Patience...

```clojure
(defn thunk-timeout
  ...
  ([thunk time unit tg]
     (let [task (FutureTask. thunk)
           thr (if tg (Thread. tg task) (Thread. task))]
       (try
         (.start thr)
         (.get task time (or (uglify-time-unit unit) unit))
         (catch TimeoutException e
           (.cancel task true)
           (.stop thr)
           (throw (TimeoutException. "Execution timed out.")))
         (catch Exception e
           (.cancel task true)
           (.stop thr)
           (throw e))
         (finally (when tg (.stop tg)))))))
```

Sandboxing threads

- thunk-timeout can handle typical (Thread. ...) threads.
- Do not allow anything that uses threadpools.
    - secure-tester tries to do this.

The moment you've all been waiting for.
The evaluator!

Finally, An Eval!

```
(defn- evaluator [code tester-str context nspace bindings]
  (fn []
    (binding [*ns* nspace
              *read-eval* false]
      (let [bindings (or bindings {})
            code `(do ~(make-dot tester-str)
                      ~(dotify (macroexpand-most code)))]
        (with-bindings bindings
          (jvm-sandbox #(eval code) context))))))
```

Finally, An Eval!

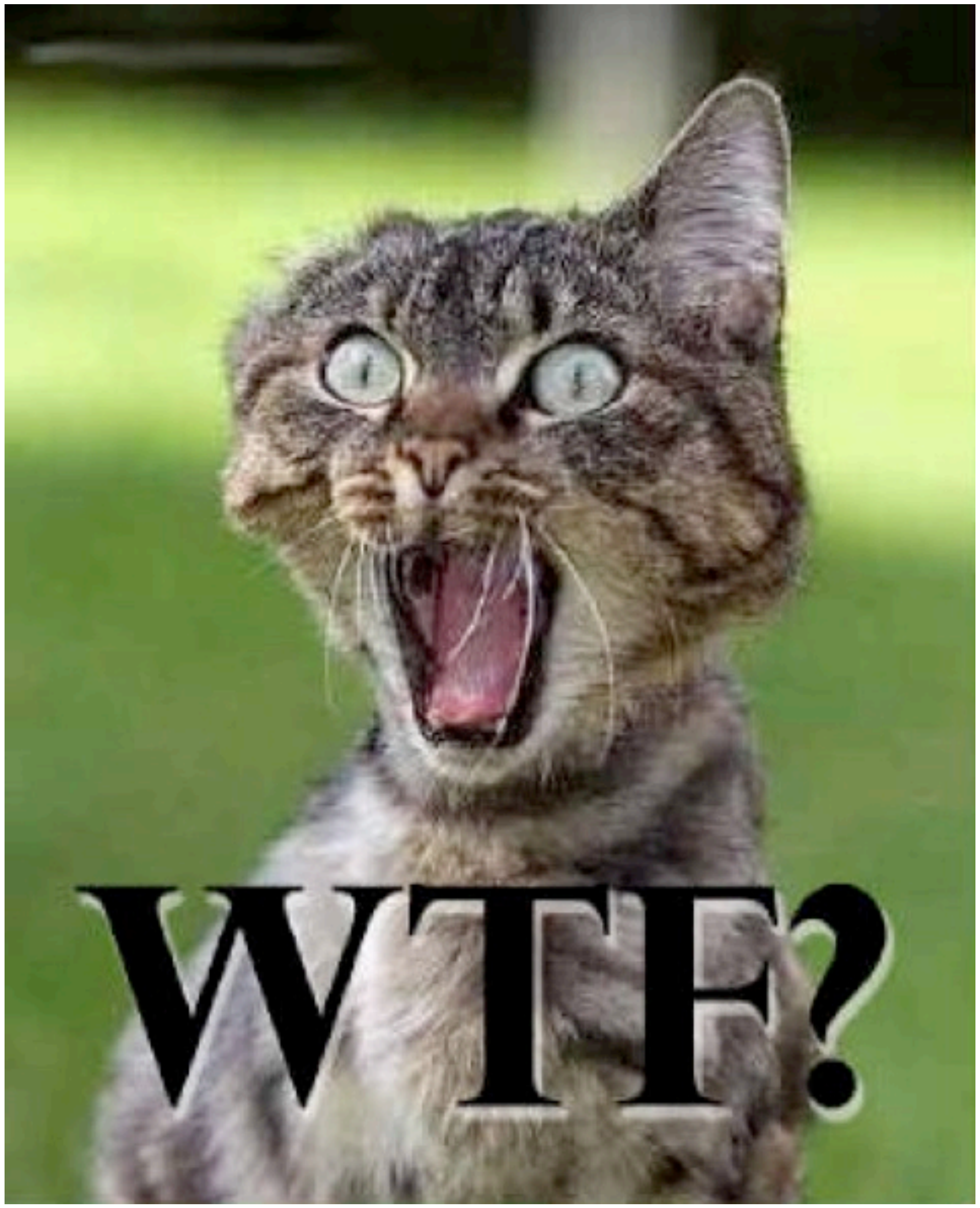
```
(defn- evaluator [code tester-str context nspace bindings]
  (fn []
    (binding [*ns* nspace
              *read-eval* false]
      (let [bindings (or bindings {})
            code `(do ~(make-dot tester-str)
                      ~(dotify (macroexpand-most code)))]
        (with-bindings bindings
          (jvm-sandbox #(eval code) context))))))
```

That is a lot to take in. How does clojail put it all together?
The sandbox* function!

Shield your eyes, it’s a really dense function.

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (let [result (if-let [problem (check-form code tester nspace)]
                           (security-exception problem)
                           (thunk-timeout
                            (evaluator code tester-str context nspace bindings)
                            timeout :ms
                            (ThreadGroup. "sandbox")))]
              result)
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```



And that was without the docstring!

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
     (fn [code tester & [bindings]]
       (let [tester-str (read-tester tester)
             old-defs (user-defs nspace)]
         (when jvm (set-security-manager (SecurityManager.)))
         (try
           (if-let [problem (check-form code tester nspace)]
             (security-exception problem)
             (thunk-timeout
              (evaluator code tester-str context nspace bindings)
              timeout :ms
              (ThreadGroup. "sandbox")))
           (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

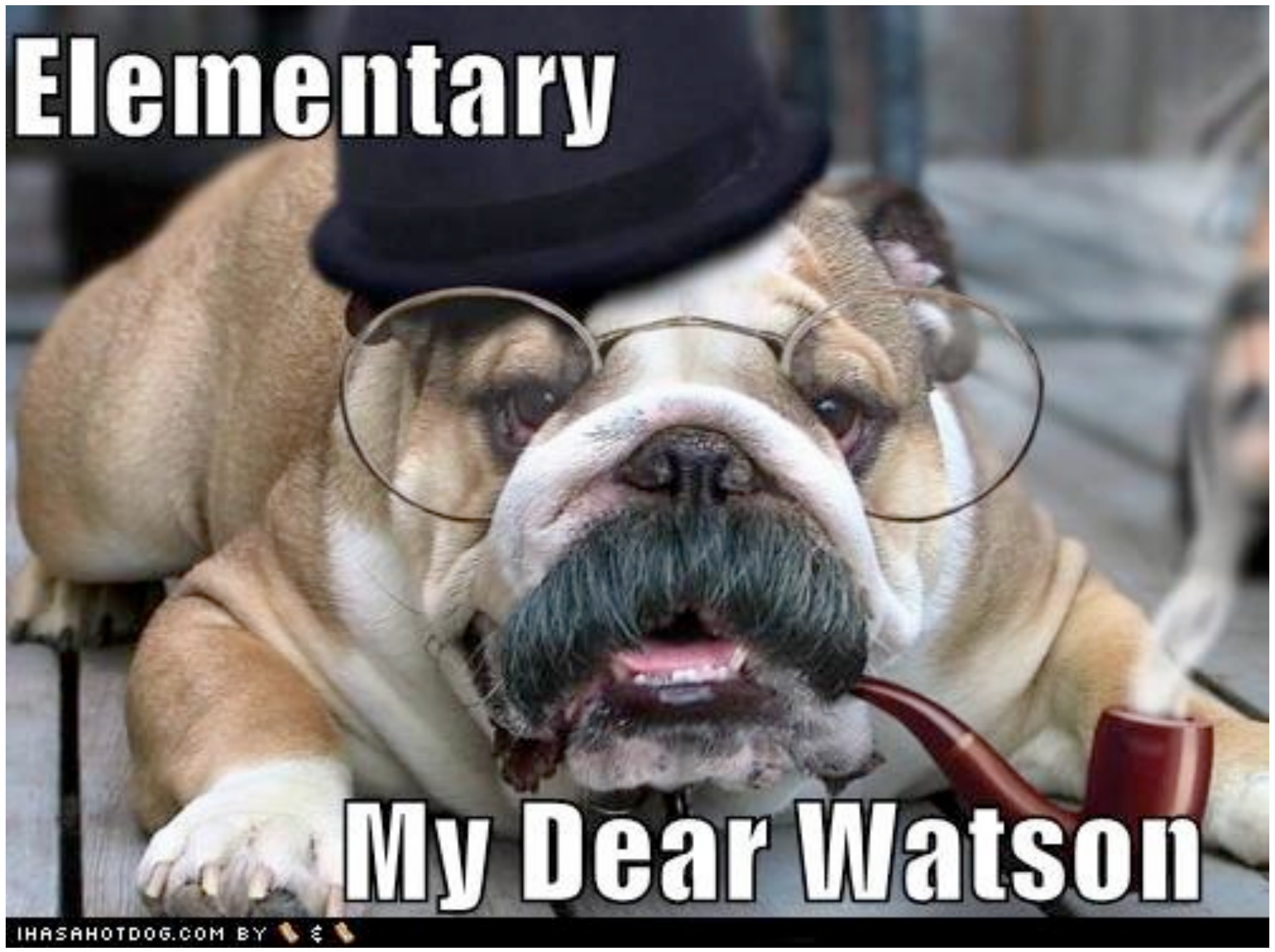
```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                          init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```

Make It So

```
(defn sandbox* [& {:keys [timeout namespace context jvm
                           init ns-init max-defs refer-clojure]
                   :or {timeout 10000
                        namespace (gensym "sandbox")
                        context (-> (permissions) domain context)
                        jvm true
                        refer-clojure true
                        max-defs 5}}]
  (let [nspace (create-ns namespace)]
    (binding [*ns* nspace]
      (when refer-clojure (clojure.core/refer-clojure))
      (eval init))
    (let [init-defs (conj (user-defs nspace) 'dot)]
      (fn [code tester & [bindings]]
        (let [tester-str (read-tester tester)
              old-defs (user-defs nspace)]
          (when jvm (set-security-manager (SecurityManager.)))
          (try
            (if-let [problem (check-form code tester nspace)]
              (security-exception problem)
              (thunk-timeout
               (evaluator code tester-str context nspace bindings)
               timeout :ms
               (ThreadGroup. "sandbox")))
            (finally (wipe-defs init-defs old-defs max-defs nspace))))))))
```



The result of all that, my good friends, is a Clojure sandbox.
It's awesome and all, but we need to think about a few things.

- If being safe is important, you should take every possible precaution imaginable.
- The JVM sandbox is mature and thorough, but that doesn't mean it is invincible.
- Run your code in its own user account.

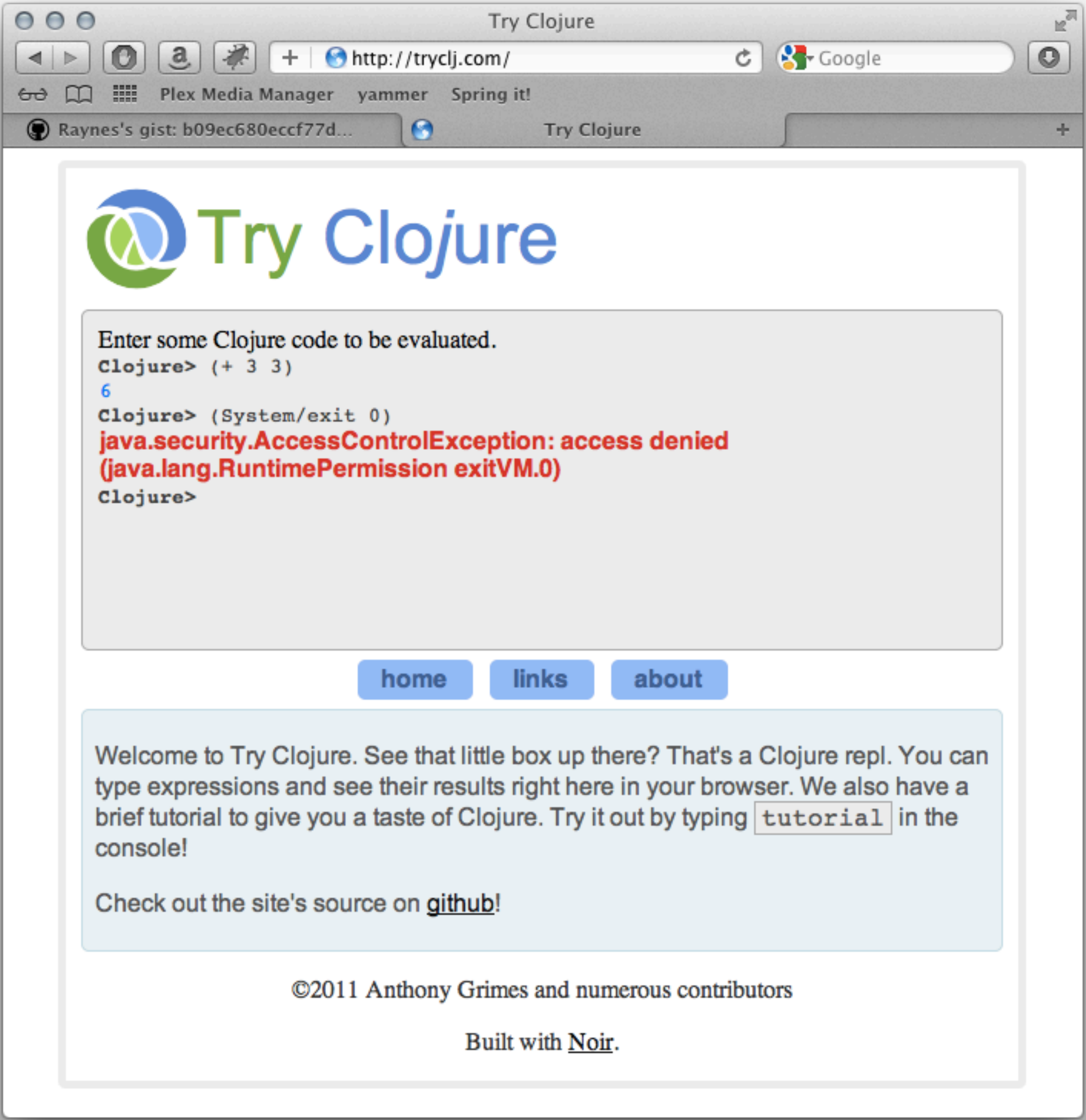
- You’re okay as long as you use the JVM sandbox.
- Allowing everyone to safely evaluate code in the same namespace is clojail’s goal.
- We are limited by not being Rich Hickey.

Luckily, Clojurians are drawn to holes in clojail like moths to a flame.

People (mostly me) trust Clojail enough to use it in their own projects.

Try Clojure

- An interactive tutorial website for Clojure with a Clojail-powered REPL.
- Similar in nature to the other TryLanguage websites, particularly TryHaskell.
  - Has a space in the name, unlike the other sites. This makes it cooler.
- Built on Chris Granger's awesome Noir web framework.
- Runs on Heroku. Also makes it cooler.



```
Enter some Clojure code to be evaluated.
Clojure> (+ 3 3)
6
Clojure> (System/exit 0)
java.security.AccessControlException: access denied
(java.lang.RuntimePermission exitVM.0)
Clojure>
```

Welcome to Try Clojure. See that little box up there? That's a Clojure repl. You can type expressions and see their results right here in your browser. We also have a brief tutorial to give you a taste of Clojure. Try it out by typing `tutorial` in the console!

Check out the site's source on github!

©2011 Anthony Grimes and numerous contributors

Built with Noir.

Approach

- def is allowed.
- Each user has his own namespace.
- Timeouts happen very fast.
- Tries to emulate a REPL as closely as possible.

```clojure
(defn make-sandbox []
  (sandbox try-clojure-tester
           :timeout 2000
           :init '(future (Thread/sleep 600000)
                          (-> *ns* .getName remove-ns))))

(defn find-sb [old]
  (if-let [sb (get old "sb")]
    old
    (assoc old "sb" (make-sandbox))))

(defn eval-request [expr]
  (try
    (eval-string expr (get (update-session! find-sb) "sb"))
    (catch TimeoutException _
      {:error true :message "Execution Timed Out!"})
    (catch Exception e
      {:error true :message (str (root-cause e))})))
```

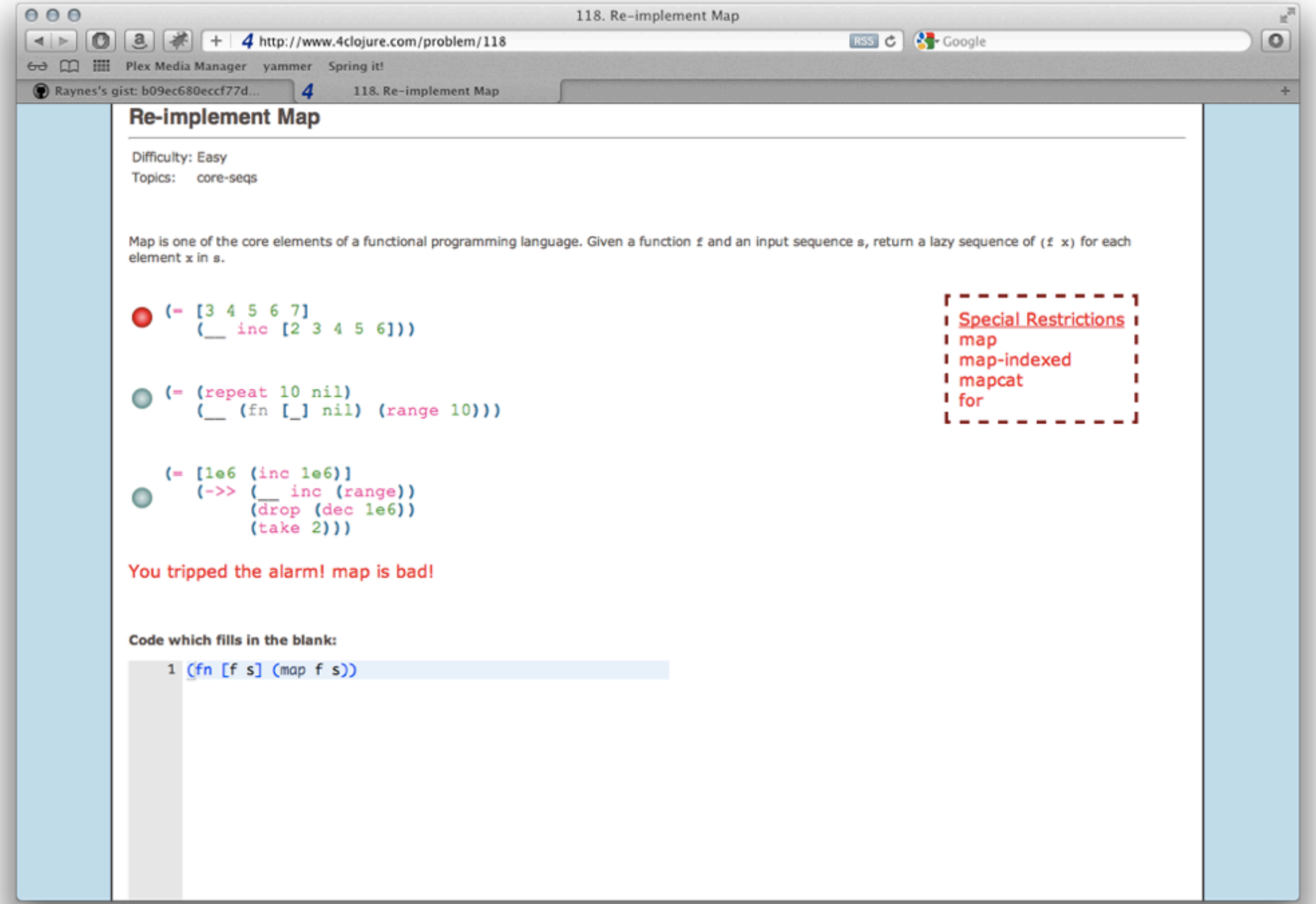
```clojure
(defn make-sandbox []
  (sandbox try-clojure-tester
           :timeout 2000
           :init '(future (Thread/sleep 600000)
                          (-> *ns* .getName remove-ns))))

(defn find-sb [old]
  (if-let [sb (get old "sb")]
    old
    (assoc old "sb" (make-sandbox))))

(defn eval-request [expr]
  (try
    (eval-string expr (get (update-session! find-sb) "sb"))
    (catch TimeoutException _
      {:error true :message "Execution Timed Out!"})
    (catch Exception e
      {:error true :message (str (root-cause e))})))
```

Credits

- Andrew Gwozdziewycz (apgwoz)
  - Design the whole thing.
- Chris Done
  - Awesome jquery-console used for the REPL interface.
  - Awesome design on TryHaskell that we took inspiration from.
- Allen Johnson (mefesto)
  - Wrote the interactive tutorial stuff.

4Clojure

- Perhaps the most interesting Clojail use-case.
- Solve koan-like Clojure problems/tasks in your browser.
- Has a long list of problems of variable difficulity, ranging from easy to very hard.
- Wonderful as a companion to any Clojure learning material, and is a great learning experience even for veteran Clojurians.


118. Re-implement Map

http://www.4clojure.com/problem/118

Plex Media Manager yammer Spring it!

Raynes's gist: b09ec680eccf77d... 118. Re-implement Map

Re-implement Map

Difficulty: Easy
Topics: core-seqs

Map is one of the core elements of a functional programming language. Given a function `f` and an input sequence `s`, return a lazy sequence of `(f x)` for each element `x` in `s`.

```
(= [3 4 5 6 7]
   (__ inc [2 3 4 5 6]))
```

```
(= (repeat 10 nil)
   (__ (fn [_] nil) (range 10)))
```

```
(= [1e6 (inc 1e6)]
   (->> (__ inc (range))
        (drop (dec 1e6))
        (take 2)))
```

Special Restrictions
map
map-indexed
mapcat
for

You tripped the alarm! map is bad!

Code which fills in the blank:

```
(fn [f s] (map f s))
```

Thursday, November 10, 11

Approach

- Relies on dynamic sandboxing.
- If a problem calls for the reimplementation of a core function, the core function or similar functions can be blacklisted to prevent cheating.

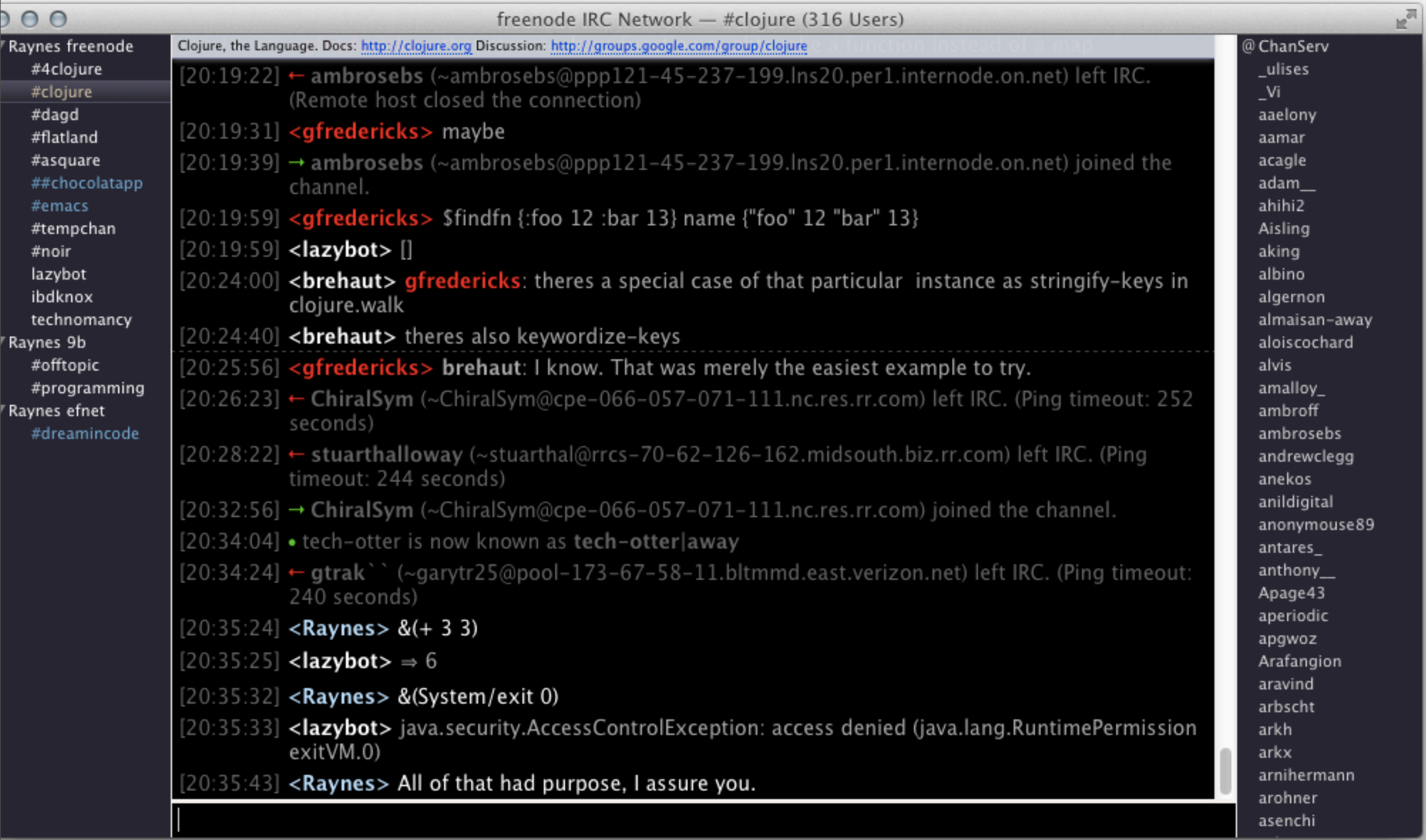
```
(for [test tests]
  (try
    (when-not (->> user-forms
                   (s/replace test "__")
                   read-string-safely
                   first
                   (sb sb-tester))
      "You failed the unit tests")
    (catch Throwable t (.getMessage t))))
```

Credits

- David Byrne and Alan Malloy (project leads)
- Alex McNamara (top contributor)
- Carin Meier (frontend)

Lazybot

- An IRC bot written in Clojure.
- Extensible via plugins.
- Totally dynamic and can be run/manipulated from a repl.
- Has a Clojure evaluation plugin.
- Can be found in #clojure, stealing people's codez.



Clojure, the Language. Docs: http://clojure.org Discussion: http://groups.google.com/group/clojure

```
[20:19:22] ← ambrosebs (~ambrosebs@ppp121-45-237-199.lns20.per1.internode.on.net) left IRC.
           (Remote host closed the connection)
[20:19:31] <gfredericks> maybe
[20:19:39] → ambrosebs (~ambrosebs@ppp121-45-237-199.lns20.per1.internode.on.net) joined the
           channel.
[20:19:59] <gfredericks> $findfn {:foo 12 :bar 13} name {"foo" 12 "bar" 13}
[20:19:59] <lazybot> []
[20:24:00] <brehaut> gfredericks: theres a special case of that particular  instance as stringify-keys in
           clojure.walk
[20:24:40] <brehaut> theres also keywordize-keys
[20:25:56] <gfredericks> brehaut: I know. That was merely the easiest example to try.
[20:26:23] ← ChiralSym (~ChiralSym@cpe-066-057-071-111.nc.res.rr.com) left IRC. (Ping timeout: 252
           seconds)
[20:28:22] ← stuarthalloway (~stuarthal@rrcs-70-62-126-162.midsouth.biz.rr.com) left IRC. (Ping
           timeout: 244 seconds)
[20:32:56] → ChiralSym (~ChiralSym@cpe-066-057-071-111.nc.res.rr.com) joined the channel.
[20:34:04] • tech-otter is now known as tech-otter|away
[20:34:24] ← gtrak`` (~garytr25@pool-173-67-58-11.bltmmd.east.verizon.net) left IRC. (Ping timeout:
           240 seconds)
[20:35:24] <Raynes> &(+ 3 3)
[20:35:25] <lazybot> ⇒ 6
[20:35:32] <Raynes> &(System/exit 0)
[20:35:33] <lazybot> java.security.AccessControlException: access denied (java.lang.RuntimePermission
           exitVM.0)
[20:35:43] <Raynes> All of that had purpose, I assure you.
```

Approach

- def is not allowed.
- Everybody uses the same namespace in all channels.

```clojure
(defn execute-text [box? bot-name user txt pre]
  (try
    (with-open [writer (StringWriter.)]
     ;; I am aware of the existence of with-out-str.
     ;; Look closer -- it won't work here.
      (let [bindings {#'*out* writer}
            res (if box?
                  (sb (safe-read txt) bindings)
                  (pr-str (no-box (read-string txt) bindings)))
            replaced (string/replace (str writer) "\n" " ")
            result (str replaced (when (= last \space) " ") res)]
        (str (or pre "\u21D2 ") (trim bot-name user txt result))))
   (catch TimeoutException _ "Execution Timed Out!")
   (catch Exception e (str (root-cause e)))))
```

Credits

- Alan Malloy
- A zillion contributors whose names wont all fit in this slide (or talk). You know who you are.

Guidelines for using Clojail in your own code

- The JVM sandbox is your friend. Always use it.
- Follow Clojail's release cycle closely and update at every convenient chance.
- Report any and every issue you find with it.
- Don't be paranoid. Remember that the JVM sandbox will protect you from real danger.
- If you avoid sharing the same namespace with everybody, it is less likely that one person will blow away the state of the whole thing for everybody.
- Don't allow def and give everyone the same namespace. That's asking for it.

Further reading

- https://github.com/flatland/clojail/wiki

Thanks

- The internet:
  - For all of the adorable cat pictures.
- Baishampayan Ghose
  - For having the longest name I've ever had to type.
  - For the '10 conj pictures.
- My Geni co-workers (Alan, Lance, Justin):
  - For listening to this talk and reviewing it.
  - Helping me prepare.
- Alan Malloy
  - For turning my insane ideas into good ones.