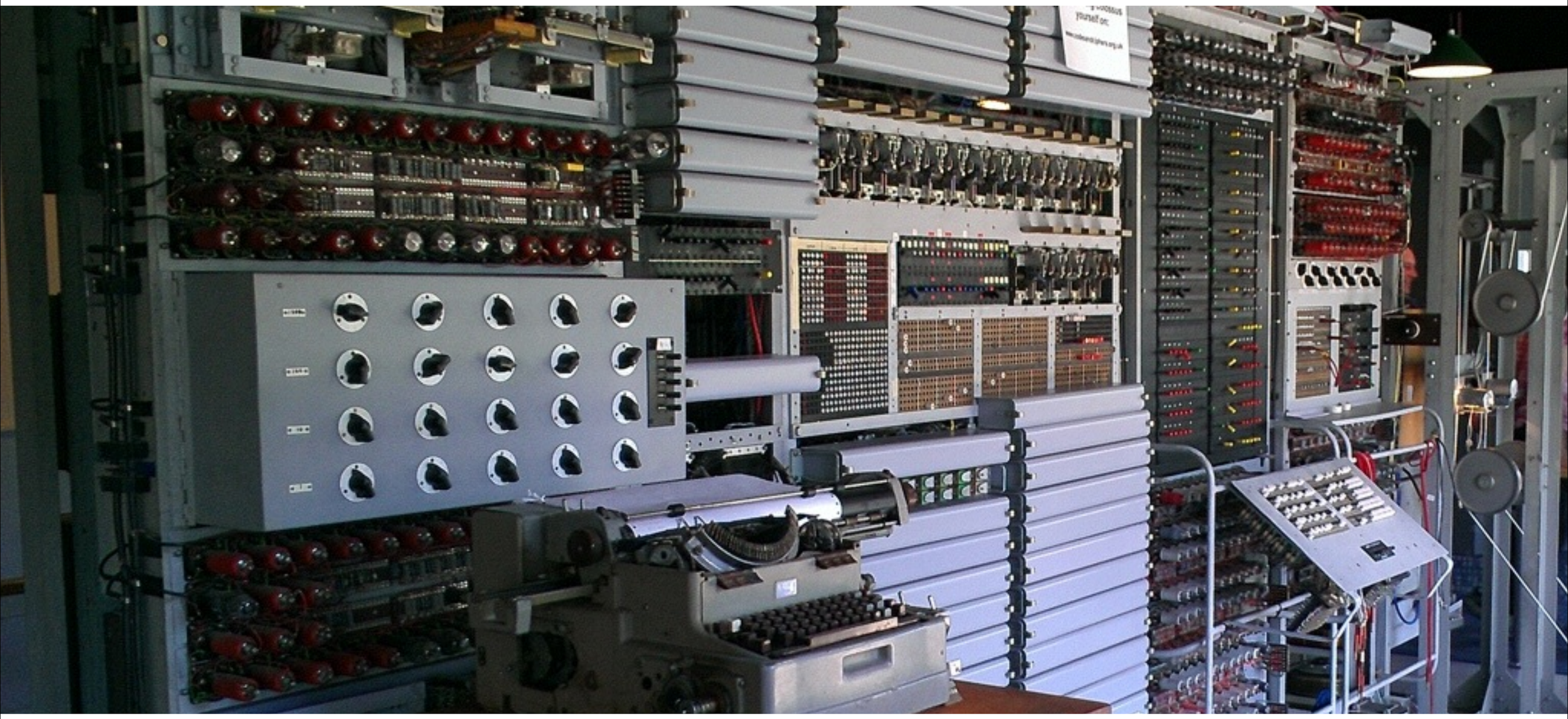
Clojure in the Large



cognitect

@stuartsierra

DevNexus 2014, Atlanta

APL is like a beautiful diamond—
flawless, beautifully symmetrical.
But you can't add anything to it.

Joel Moses, 1970s (disputed)

Lisp is like a ball of mud.
Add more and it's still a ball of mud—
it still looks like Lisp.

Joel Moses, 1970s (disputed)

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.

Brian Foote and Joseph Yoder, 1999

These systems show unmistakable signs of unregulated growth, and repeated, expedient repair.

Brian Foote and Joseph Yoder, 1999

Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

Brian Foote and Joseph Yoder, 1999

```clojure
;; Not a lot of structure

(ns com.example.app)

(def state (ref {}))

(defn- private-op [arg] ...)

(defn op1 [arg] ...)

(defn op2 [arg] ...)
```

```clojure
;; Not a lot of structure

(ns com.example.app)

(def state (ref {}))

(defn- private-op [arg] ...)

(defn op1 [arg] ...)

(defn op2 [arg] ...)
```

Not first-class modules

```clojure
;; Not a lot of structure

(ns com.example.app)

(def state (ref {}))

(defn- private-op [arg] ...)

(defn op1 [arg] ...)

(defn op2 [arg] ...)
```

Not first-class modules

Global by default

Larger …

- Programs: 10,000s of lines
- Systems: 10s–100s of machines
- Teams: 10s of developers

Components serve as the building blocks for the structure of a system.

Components serve as the building blocks for the structure of a system.

Component

- Encapsulated state
- Interfaces
- Managed lifecycle

Encapsulation

Encapsulation deals with grouping the elements of an abstraction ... and with separating different abstractions from each other.

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))
```

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(defn cache-user [user]
  (dosync
    (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...))
```

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(defn cache-user [user]
  (dosync
    (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...))
```

Effectively global mutable variables

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(defn cache-user [user]
  (dosync
    (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...))
```

Effectively global mutable variables

Temptation to share promiscuously

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(defn cache-user [user]
  (dosync
    (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...))
```

Effectively global mutable variables

Temptation to share promiscuously

Cannot ever have more than one

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(defn cache-user [user]
  (dosync
    (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...))
```

Hidden dependencies

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))

(deftest t-user-cache
  (reset! db-conn ...)
  (dosync (ref-set user-cache ...))
  ... actual test ...))
```

Harder to setup/ teardown for testing

```clojure
;; Global state

(def user-cache (ref {}))

(def db-conn (atom nil))
```

```clojure
(require ... :reload)
```

Reloading code destroys state

```clojure
;; Global state

(defonce user-cache (ref {}))

(defonce db-conn (atom nil))
```

```clojure
(require ... :reload)
```

```
;; Global state

(defonce user-cache (ref {}))

(defonce db-conn (atom nil))
```

Cannot recover initial state

```
(require ... :reload)
```

```clojure
;; Global state

(let [user-cache (ref {})]
      db-conn (atom nil)]

  (defn cache-user [user]
    (dosync
      (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...)))
```

```
;; Global state

(let [user-cache (ref {})]
      db-conn (atom nil)]

  (defn cache-user [user]
    (dosync
      (alter user-cache ...)))

(defn connect []
  (swap! db-conn ...)))
```

Global state hidden in closure

```
;; Local state
```

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})
```

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})
```

Well-defined initial state

```
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})
```

Well-defined initial state

Encapsulate related state

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})

(defn cache-user [user-db]
  (dosync
    (alter (:cache user-db) ...)))

(defn connect [user-db]
  (assoc user-db :conn ...))
```

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})

(defn cache-user [user-db]
  (dosync
    (alter (:cache user-db) ...)))

(defn connect [user-db]
  (assoc user-db :conn ...))
```

Makes function dependencies clear

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})

(deftest t-cache-user
  (let [users (new-user-db)]
    (is (= ... (get-user users)))))
```

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})

(deftest t-cache-user
  (let [users (new-user-db)]
    (is (= ... (get-user users)))))
```

Easy to test

```clojure
;; Local state

(defn new-user-db []
  {:cache (ref {})
   :conn nil})
```

```clojure
(require ... :reload)
```

Safe to reload

```
;; Thread-bound state
```

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)
```

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)

(defn- internal-op1 []
  ... *resource* ...)
```

```clojure
(defn op1 [arg]
  (internal-op1 ...))
```

```
;; Thread-bound state

(def ^:dynamic *resource*)

(defn- internal-op1 []
  ...*resource*...)

(defn op1 [arg]
  (internal-op1 ...))
```

Hidden dependency

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)

(defn op1 [arg]
  (internal-op1 ...))
```

```clojure
(defmacro with-resource [src & body]
 `(binding [*resource* (acquire src)]
    (try ~@body
      (finally (dispose *resource*)))))
```

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)

(defn op1 [arg]
  (internal-op1 ...))
```

```clojure
(defmacro with-resource [src & body]
```

Assumes
body starts & ends
on a single thread

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)

(defn op1 [arg]
  (internal-op1 ...))
```

```clojure
(defmacro with-resource [src & body]
```

Assumes
body does not return
a lazy sequence

```clojure
;; Thread-bound state

(def ^:dynamic *resource*)

(defn op1 [arg]
  (internal-op1 ...))
```

```clojure
(defmacro with-resource [src & body]
```

Limits caller
to one resource
at a time

```
;; Local state with dynamic extent

(defmacro with-resource [[sym src] & body]
  `(let [~sym (acquire ~src)]
     (try ~@body
       (finally
         (dispose ~sym)))))
```

```
;; Local state with dynamic extent

(defmacro with-resource [[sym src] & body]
  `(let [~sym (acquire ~src)]
     (try ~@body
       (finally
         (dispose ~sym)))))
```

Not limited to one resource

```
(with-resource [a source-for-a]
  (with-resource [b source-for-b]
    ...))
```

```clojure
;; Local state with dynamic extent

(defmacro with-resource [[sym src] & body]
  `(let [~sym (acquire ~src)]
     (try ~@body
       (finally
         (dispose ~sym)))))
```

Still assumes body completes on one thread

```clojure
(with-resource [a source-for-a]
  (with-resource [b source-for-b]
    ...))
```

```
;; “Request scope”



(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

```
;; “Request scope”
```

Pass around
“context” argument

```
(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

```
;; “Request scope”
```

Encapsulates all state for a given process

```
(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

```
;; “Request scope”



(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

Accumulate
intermediate results

```
;; “Request scope”

(defn add-resource [context src]
  (assoc context ::resource (acquire src)))

(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

Acquire resources

```
;; “Request scope”

(defn add-resource [context src]
  (assoc context ::resource (acquire src)))

(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

Use namespace-qualified keys for isolation

```
;; “Request scope”
```

Not confined to a single thread

```
(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

```clojure
;; “Request scope”

(defn add-resource [context src]
  (assoc context ::resource (acquire src)))

(defn op1 [context]
  (let [resource (::resource context)]
    (assoc context ::result1 ...)))
```

Still need to manage resource lifecycle

```clojure
(defn finish [context]
  (update-in context [::resource] dispose))
```

```
;; Safe global Vars
```

```
;; Safe global Vars

;; True constants
(def ^:const gravity 6.67384e-11)
```

```
;; Safe global Vars

;; True constants
(def ^:const gravity 6.67384e-11)

;; True singletons (rare)
(def runtime (Runtime/getRuntime))
```

```clojure
;; Safe global Vars

;; True constants
(def ^:const gravity 6.67384e-11)

;; True singletons (rare)
(def runtime (Runtime/getRuntime))

;; For interactive REPL use
(def conn (db/connect uri))
```

```clojure
;; Private dynamic Vars

(def ^:private ^:dynamic *state*)

(defn- internal-op1 [arg]
  ... *state* ...)
```

```clojure
(defn op1 [arg]
  (binding [*state* (initial-state)]
    (internal-op1 arg)))
```

```clojure
;; Private dynamic Vars

(def ^:private ^:dynamic *state*)

(defn- internal-op1 [arg]
  ... *state* ...)

(defn op1 [arg]
  (binding [*state* (initial-state)]
    (internal-op1 arg)))
```

Binding is confined to a single operation

```clojure
;; Private dynamic Vars

(def ^:private ^:dynamic *state*)

(defn- internal-op1 [arg]
  ... *state* ...)

(defn op1 [arg]
  (binding [*state* (initial-state)]
    (internal-op1 arg)))
```

Binding is confined to a single operation

Operation is single-threaded by definition

Decoupling

```clojure
;; Interface

(defprotocol KeyValueStore
  (get [store key])
  (put [store key value])
  (cas [store key old new]
    "Compare-and-set"))
```

```clojure
;; Interface

(defprotocol KeyValueStore
  (get [store key])
  (put [store key value])
  (cas [store key old new]
    "Compare-and-set"))
```

primitives, essential operations

```clojure
;; Public API

(defn swap [store key f & args]
  (loop []
    (let [old (get store key)
          new (apply f old args)]
      (if (cas store key old new)
        new
        (recur)))))
```

```clojure
;; Public API

(defn swap [store key f & args]
  (loop []
    (let [old (get store key)
          new (apply f old args)]
      (if (cas store key old new)
        new
        (recur)))))
```

Built on primitive operations

```clojure
;; Implementation

(defrecord SQLStore [url conn]
  KeyValueStore
  (get [_ key] ...)
  (put [_ key value] ...)
  (cas [_ key old new] ...))

(defn sql-store [url]
  (->SQLStore url (atom nil)))
```

```clojure
;; Implementation

(defrecord SQLStore [url conn]
  KeyValueStore
  (get [_ key] ...)
  (put [_ key value] ...)
  (cas [_ key old new] ...))

(defn sql-store [url]
  (->SQLStore url (atom nil)))
```

Create state in constructor

```clojure
;; Another implementation

(defrecord MemoryStore [r]
  KeyValueStore
  (get [_ key]
    (clojure.core/get @r key))
  (put [_ key value]
    (dosync (alter r assoc key value)))
  (cas [this key old new]
    (dosync (if (= old (get this key))
              (do (alter r assoc key new)
                  true)
              false))))
```

```clojure
;; Another implementation

(defn mem-store []
  (->MemoryStore (ref {})))
```

```
;; Another implementation

(defn mem-store []
  (->MemoryStore (ref {})))
```

Create state in constructor

```clojure
;; Operations using component

(defn op1 [store arg]
  (swap store ...))
```

```clojure
;; Operations using component

(defn op1 [store arg]
  (swap store ...))
```

Pass component as argument

```clojure
;; Operations using component

(defn op1 [store arg]
  (swap store ...))
```

Pass component as argument

Depend only on public API

```clojure
;; Operations using component

(defn op1 [store arg]
  (swap store ...))

(deftest t-op1
  (let [store (mem-store)]
    (op1 store 42)))
```

```clojure
;; Operations using component

(defn op1 [store arg]
  (swap store ...))

(deftest t-op1
  (let [store (mem-store)]
    (op1 store 42)))
```

Easy to test

```
;; Kingdom of Nouns

(defprotocol P
  (op [this]))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defrecord R2 [x y z]
  P
  (op [this] ...)))
```

```
;; Kingdom of Nouns

(defprotocol P
  (op [this]))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defrecord R2 [x y z]
  P
  (op [this] ...)))
```

Protocol with only one method

```
;; Kingdom of Nouns

(defprotocol P
  (op [this]))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defrecord R2 [x y z]
  P
  (op [this] ...)))
```

Protocol with only one method

Types with only one protocol

```clojure
;; Republic of Functions

(defn r1 [x y]
  (fn [] ...))

(defn r2 [x y z]
  (fn [] ...))
```

```clojure
;; Republic of Functions

(defn r1 [x y]
  (fn [] ...))

(defn r2 [x y z]
  (fn [] ...))
```

Can't see inside

```
;; Constitutional Monarchy of Compromise

(defprotocol P
  (op [this] ...))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defn r2 [x y z]
  (reify P
    (op [this] ...)))
```

```
;; Constitutional Monarchy of Compromise

(defprotocol P
  (op [this] ...))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defn r2 [x y z]
  (reify P
    (op [this] ...)))
```

records are transparent

```
;; Constitutional Monarchy of Compromise

(defprotocol P
  (op [this] ...))

(defrecord R1 [x y]
  P
  (op [this] ...)))

(defn r2 [x y z]
  (reify P
    (op [this] ...)))
```

records are transparent

reify is opaque

Lifecycle

```clojure
;; Rally the troops

(defn init! []
  (connect-to-database!)
  (create-thread-pools!)
  (start-background-processes!)
  (start-web-server!)
  ...)
```

```clojure
;; Rally the troops

(defn init! []
  (connect-to-database!)
  (create-thread-pools!)
  (start-background-processes!)
  (start-web-server!)
  ...)
```

All global side effects

```
;; github.com/stuartsierra/component

(defprotocol Lifecycle
  (start [component])
  (stop [component]))
```

```
;; github.com/stuartsierra/component

(defprotocol Lifecycle
  (start [component])
  (stop [component]))
```

Acquire resources
Begin operation
Return updated component

```
;; github.com/stuartsierra/component

(defprotocol Lifecycle
  (start [component])
  (stop [component]))
```

Release resources
Cease operation
Return updated component

```
;; github.com/stuartsierra/component

(defprotocol Lifecycle
  (start [component])
  (stop [component]))
```

Return updated component

Use immutable data structures

```clojure
(defrecord Database [host port conn]
  component/Lifecycle
  (start [component]
    (assoc component
      :conn (db/connect host port)))
  (stop [component]
    (.close conn)
    component))

(defn new-database [host port]
  (map->Database {:host host :port port}))
```

```clojure
(defrecord Database [host port conn]
  component/Lifecycle
  (start [component]
    (assoc component
      :conn (db/connect host port)))
  (stop [component]
    (.close conn)
    component))

(defn new-database [host port]
  (map->Database {:host host :port port}))
```

Assoc updated state

```clojure
(defrecord Database [host port conn]
  component/Lifecycle
  (start [component]
    (assoc component
      :conn (db/connect host port)))
  (stop [component]
    (.close conn)
    component))
```

Assoc updated state

Always return component

```clojure
(defn new-database [host port]
  (map->Database {:host host :port port}))
```

```clojure
(defrecord Database [host port conn]
  component/Lifecycle
  (start [component]
    (assoc component
      :conn (db/connect host port)))
  (stop [component]
    (.close conn)
    component))

(defn new-database [host port]
  (map->Database {:host host :port port}))
```

Assoc updated state

Always return component

Constructor function free of side-effects

```clojure
(defrecord TestDB [host port conn]
  component/Lifecycle
  (start [component]
    (let [conn (db/connect host port)]
      (create-database conn "test_db"))
      (add-seed-data conn)
      (assoc component :conn conn)))
  (stop [component]
    (drop-database conn)
    (.close conn)
    component))
```

```clojure
(defrecord TestDB [host port conn]
  component/Lifecycle
  (start [component]
    (let [conn (db/connect host port)]
      (create-database conn "test_db"))
      (add-seed-data conn)
      (assoc component :conn conn)))
(stop [component]
  (drop-database conn)
  (.close conn)
  component))
```

Alternate implementations for dev & testing

```clojure
(defrecord UsersAPI [database cache]
  component/Lifecycle
  (start [component]
    (reset! cache (top-users database))
    component)
  (stop [component]
    component))

(defn users-api []
  (component/using
    (map->UsersAPI {:cache (atom {})})
    [:database :email-service]))
```

```clojure
(defrecord UsersAPI [database cache]
  component/Lifecycle
  (start [component]
    (reset! cache (top-users database))
    component)
  (stop [component]
    component))
```

Encapsulate related functionality & state

```clojure
(defn users-api []
  (component/using
    (map->UsersAPI {:cache (atom {})})
    [:database :email-service]))
```

```clojure
(defrecord UsersAPI [database cache]
  component/Lifecycle
  (start [component]
    (reset! cache (top-users database))
    component)
  (stop [component]
    component))

(defn users-api []
  (component/using
    (map->UsersAPI {:cache (atom {})})
    [:database :email-service]))
```

Initialize state from other components

```
(defrecord UsersAPI [database cache]
  component/Lifecycle
  (start [component]
    (reset! cache (top-users database))
    component)
  (stop [component]
    component))

(defn users-api []
  (component/using
    (map->UsersAPI {:cache (atom {})})
    [:database :email-service]))
```

Declare dependencies by name

```clojure
(defn notify-user [users-api username message]
  (let [{:keys [database email-service]} users-api
        address (get-email database username)]
    (send-email email-service address message)))
```

Define operations in terms of component

```
(defn notify-user [users-api username message]
  (let [{:keys [database email-service]} users-api
        address (get-email database username)]
    (send-email email-service address message)))
```

Destructure dependencies

```
(defn notify-user [users-api username message]
  (let [{:keys [database email-service]} users-api
        address (get-email database username)]
    (send-email email-service address message)))
```

Destructure dependencies

```
(defn notify-user [users-api username message]
  (let [{:keys [database email-service]} users-api
        address (get-email database username)]
    (send-email email-service address message)))
```

Call other components via public API

```clojure
(defn new-system [config]
  (let [{:keys [host port]} (:db config)]
    (component/system-map
     :users-api (users-api)
     :database (database host port)
     :queue-service (queue-service ...)
     :email-service (email-service ...))))
```

Encapsulate the whole application

```
(defn new-system [config]
  (let [{:keys [host port]} (:db config)]
    (component/system-map
     :users-api (users-api)
     :database (database host port)
     :queue-service (queue-service ...)
     :email-service (email-service ...))))
```

Encapsulate the whole application

```
(defn new-system [config]
  (let [{:keys [host port]} (:db config)]
    (component/system-map
     :users-api (users-api)
     :database (database host port)
     :queue-service (queue-service ...)
     :email-service (email-service ...))))
```

Each component declares its own dependencies

Starting a System

- You call Lifecycle/start on a system
- Component library will:
  - Build dependency graph of components
  - Sort in dependency order
  - Start each component

Starting Each Component

- Get dependencies from system map (already started)
- Assoc dependencies into component
- Call Lifecycle/start on component
- Assoc "started" component back into system

```clojure
(defn new-system [config]
  (let [{:keys [host port]} (:db config)]
    (component/system-map
     :users-api (users-api)
     :database (database host port)
     :queue-service (queue-service ...)
     :email-service (email-service ...))))
```

```clojure
(defn new-system [config]
  (let [{:keys [host port]} (:db config)]
    (component/system-map
     :users-api (users-api)
     :database (database host port)
     :queue-service (queue-service ...)
     :email-service (email-service ...))))
```

Components declare dependencies

System provides them

```clojure
(defn test-system []
  (assoc (new-system default-config)
    :database (test-database ...)))
```

Alternate systems for dev & testing

```
(defn test-system []
  (assoc (new-system default-config)
    :database (test-database ...)))
```

```clojure
(defn test-system []
  (assoc (new-system default-config)
    :database (test-database ...)))
```

Assoc in alternate implementations

```
(ns user)  ;; "Workflow Reloaded"

(def system nil)

(defn go []
  (alter-var-root #'system
    (constantly (dev-system))))
  (component/start system))

(defn reset []
  (stop system)
  (clojure.tools.namespace.repl/refresh
    :after 'user/go))
```

```clojure
(ns user)  ;; "Workflow Reloaded"

(def system nil)

(defn go []
  (alter-var-root #'system
    (constantly (dev-system)))
  (component/start system))

(defn reset []
  (stop system)
  (clojure.tools.namespace.repl/refresh
    :after 'user/go))
```

Global Var for REPL only

```clojure
(ns user)  ;; "Workflow Reloaded"

(def system nil)

(defn go []
  (alter-var-root #'system
    (constantly (dev-system)))
  (component/start system))

(defn reset []
  (stop system)
  (clojure.tools.namespace.repl/refresh
    :after 'user/go))
```

Global Var for REPL only

Create the system

```clojure
(ns user)  ;; "Workflow Reloaded"

(def system nil)

(defn go []
  (alter-var-root #'system
    (constantly (dev-system)))
  (component/start system))

(defn reset []
  (stop system)
  (clojure.tools.namespace.repl/refresh
    :after 'user/go))
```

Global Var for REPL only

Create the system

Stop, reload, restart

```clojure
;; REPL introspection

(-> system :users-api :cache deref)
```

```
;; REPL introspection

(-> system :users-api :cache deref)
```

Everything is encapsulated,
but still accessible

```clojure
;; Entry Point: main

(defn -main [config-file]
  (component/start
    (new-system (read-conf config-file))))
```

```
;; You don't control main

(defroutes routes
  (GET "/foo" [req] get-foo)
  (POST "/bar" [req] do-bar))

(def handler (-> routes
                 wrap-params
                 wrap-session
                 wrap-blah-blah-blah
                 wrap-wrap-wrap-wrap-wrap))

(def server (run-jetty handler))
```

```clojure
;; You don't control main

(defroutes routes
  (GET "/foo" [req] get-foo)
  (POST "/bar" [req] do-bar))

(def handler (-> routes
                 wrap-params
                 wrap-session
                 wrap-blah-blah-blah
                 wrap-wrap-wrap-wrap-wrap))

(def server (run-jetty handler))
```

Entry points are static

```
(defrecord WebApp [users-api
                   email-service
                   database ...]
  component/Lifecycle
  (start [this] ...)
  (stop [this] ...))
```

```clojure
(defroutes routes ...)

(defn wrap-app-component [handler web-app]
  (fn [request]
    (handler (assoc request
                    ::web-app web-app))))
```

```clojure
(defn make-handler [web-app]
  (-> routes
      (wrap-app-component web-app)
      wrap-params
      wrap-session
      ...))
```

```clojure
(defroutes routes ...)

(defn wrap-app-component [handler web-app]
  (fn [request]
    (handler (assoc request
                    ::web-app web-app))))

(defn make-handler [web-app]
  (-> routes
      (wrap-app-component web-app)
      wrap-params
      wrap-session
      ...))
```

Construct root handler at startup

```
(defroutes routes ...)
```

Inject component into request context

```
(defn wrap-app-component [handler web-app]
  (fn [request]
    (handler (assoc request
                    ::web-app web-app))))
```

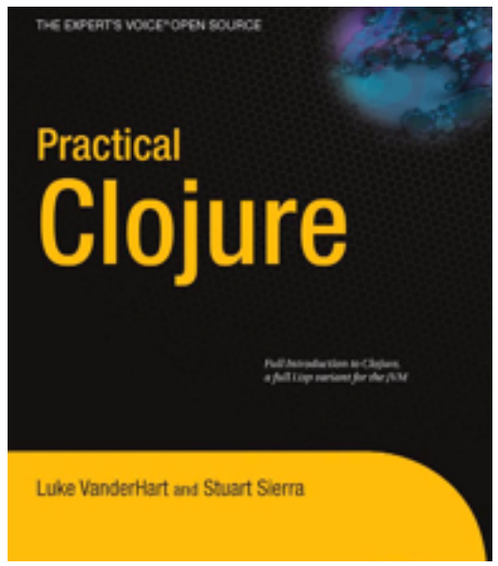
```
(defn make-handler [web-app]
  (-> routes
      (wrap-app-component web-app)
      wrap-params
      wrap-session
      ...))
```

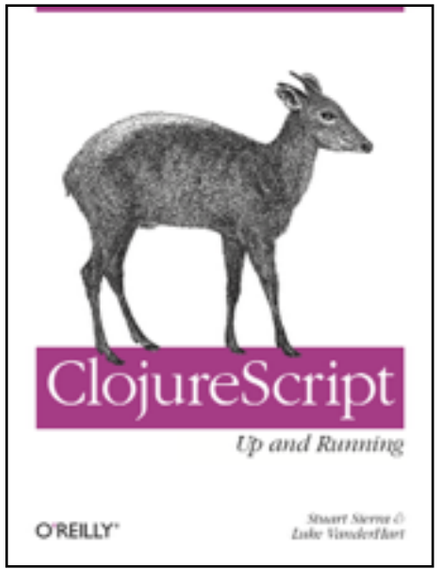
Construct root handler at startup

Summary

- Separate components
- Encapsulate local state
- Decouple by injecting dependencies
- Manage lifecycle of components
- Compose components into systems

clojure.org
Google Group: Clojure
#clojure on Freenode IRC





stuartsierra.com
@stuartsierra
github.com/stuartsierra/component

cognitect.com
@cognitect
clojure.com
datomic.com