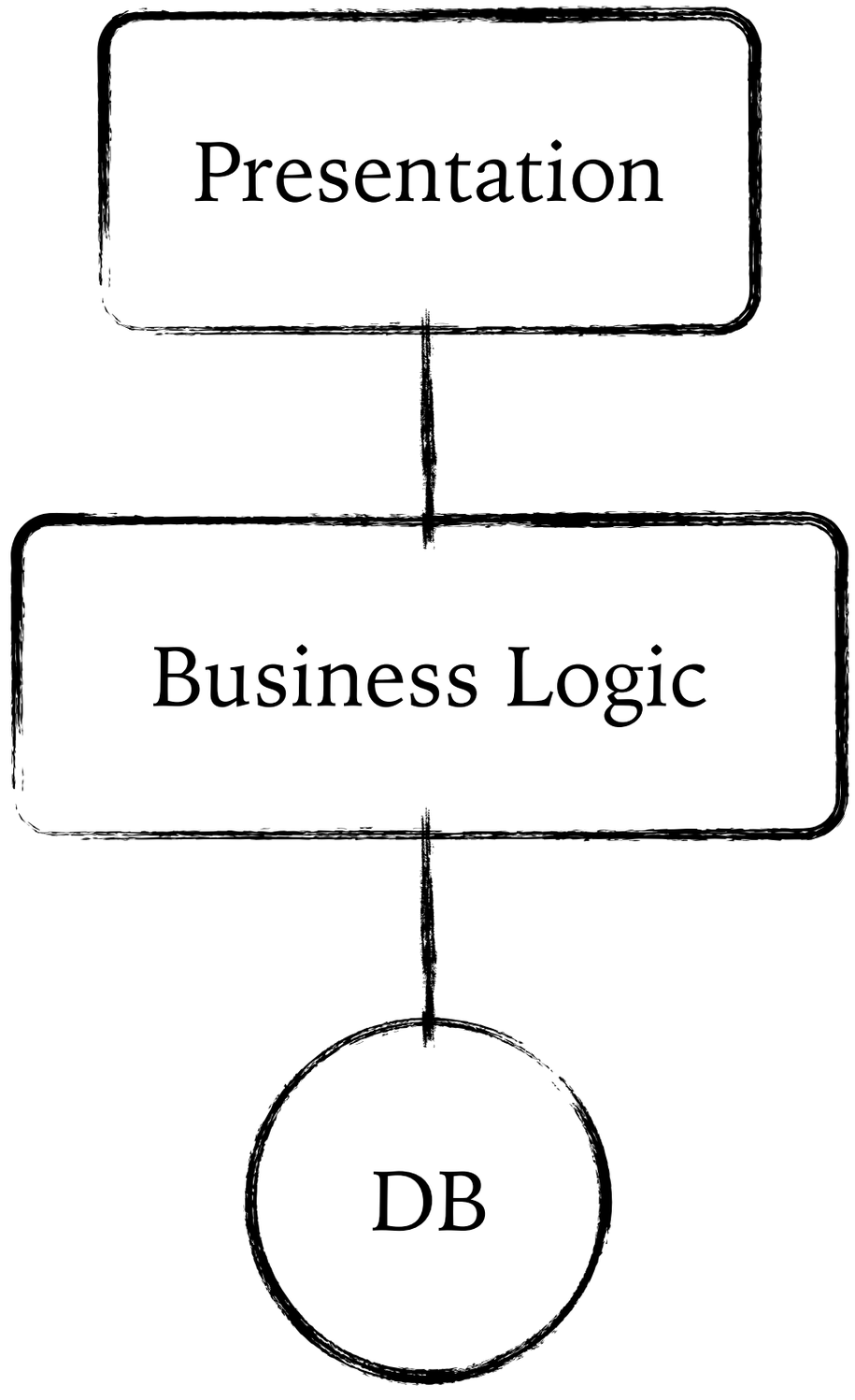
June 27, 2014
EuroClojure 2014
Krakow, Poland

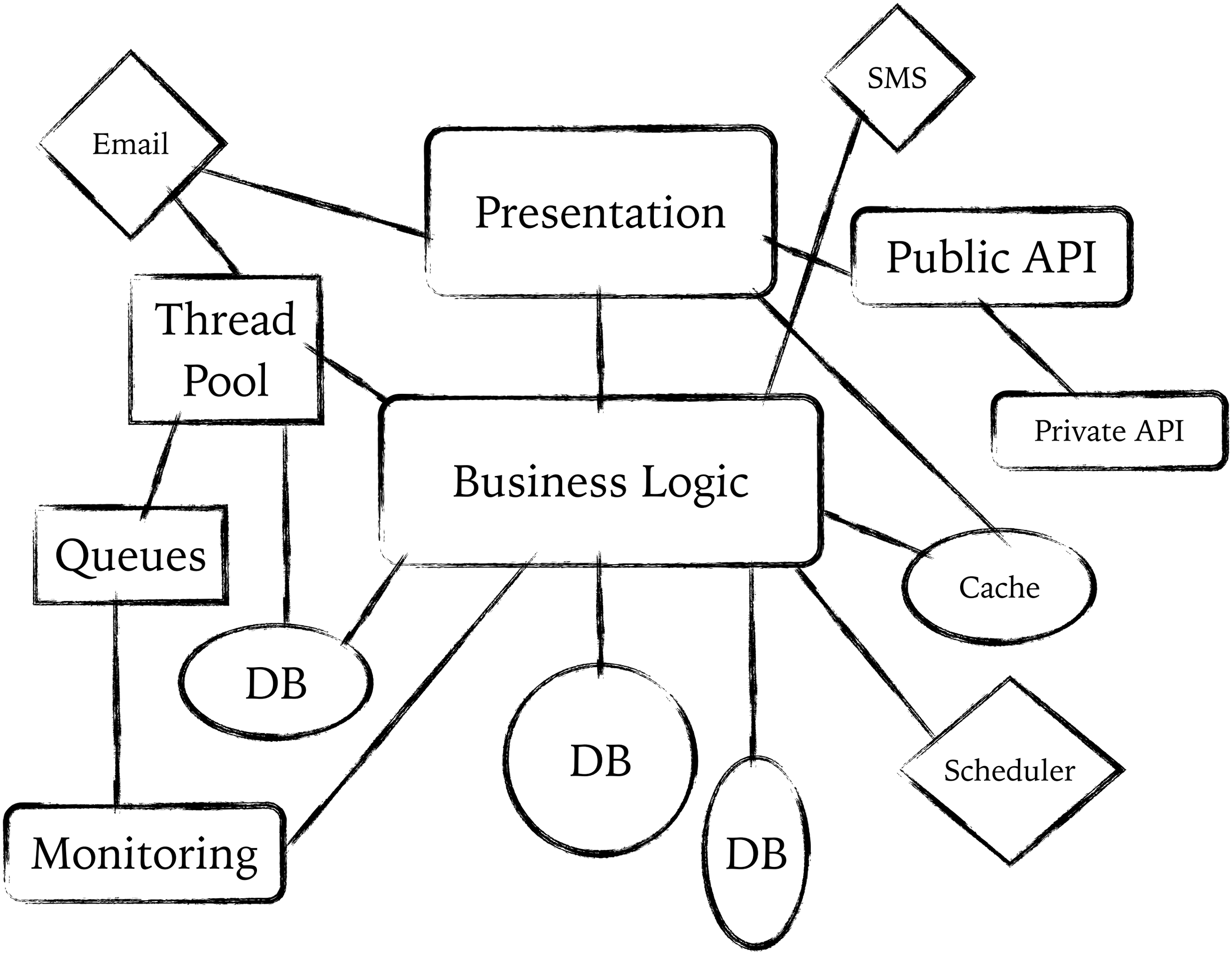
Components

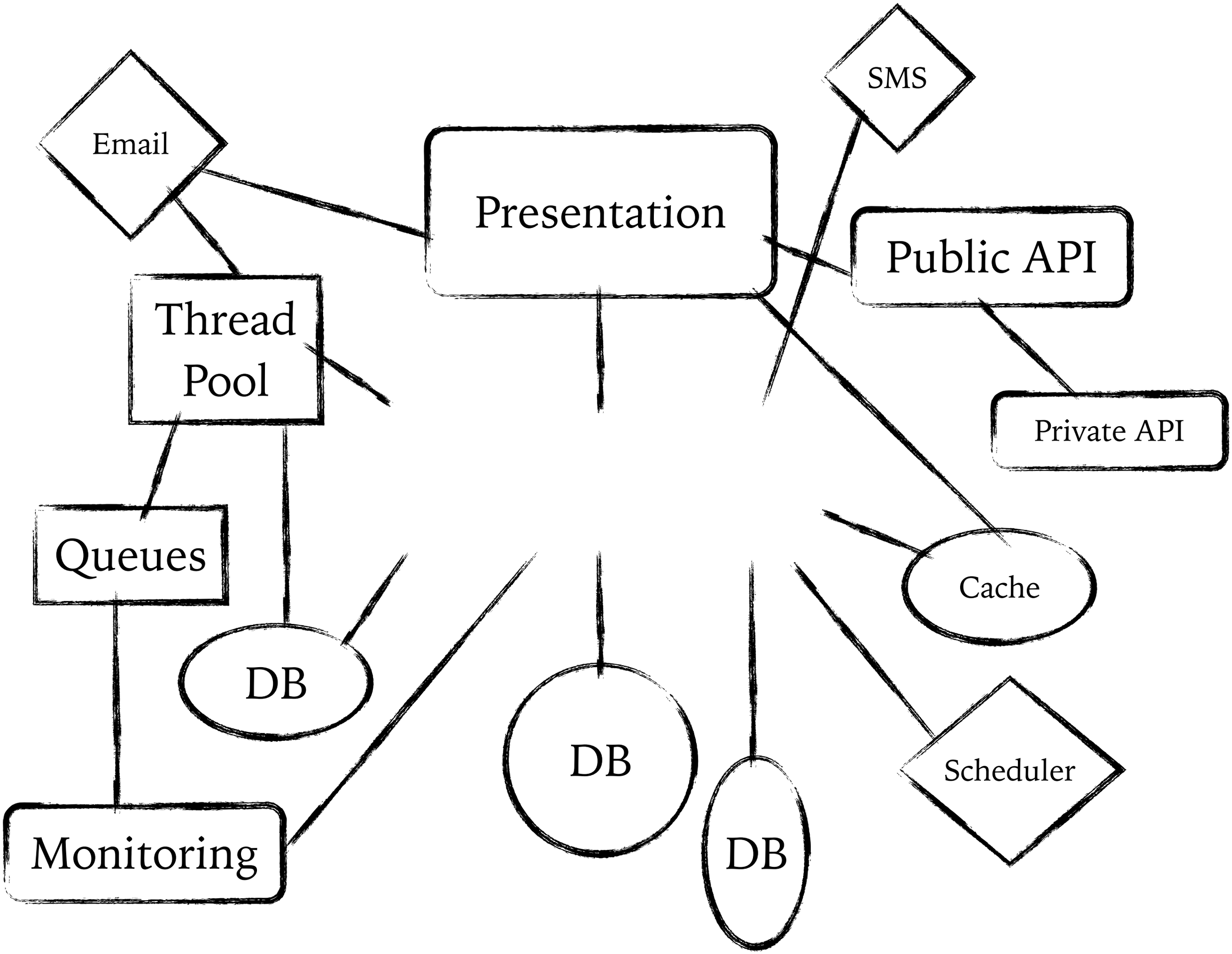
Just Enough Structure

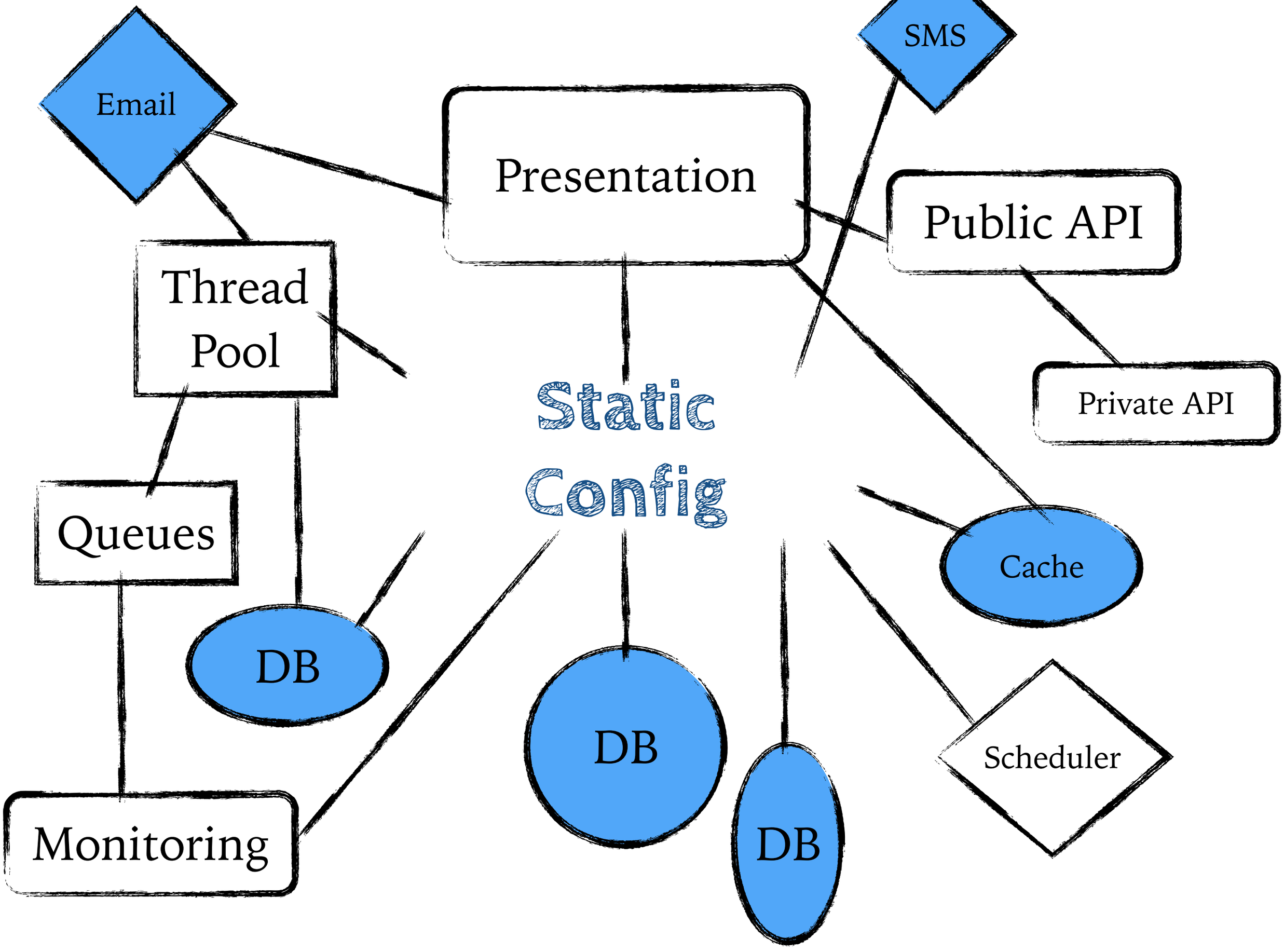
@stuartsierra

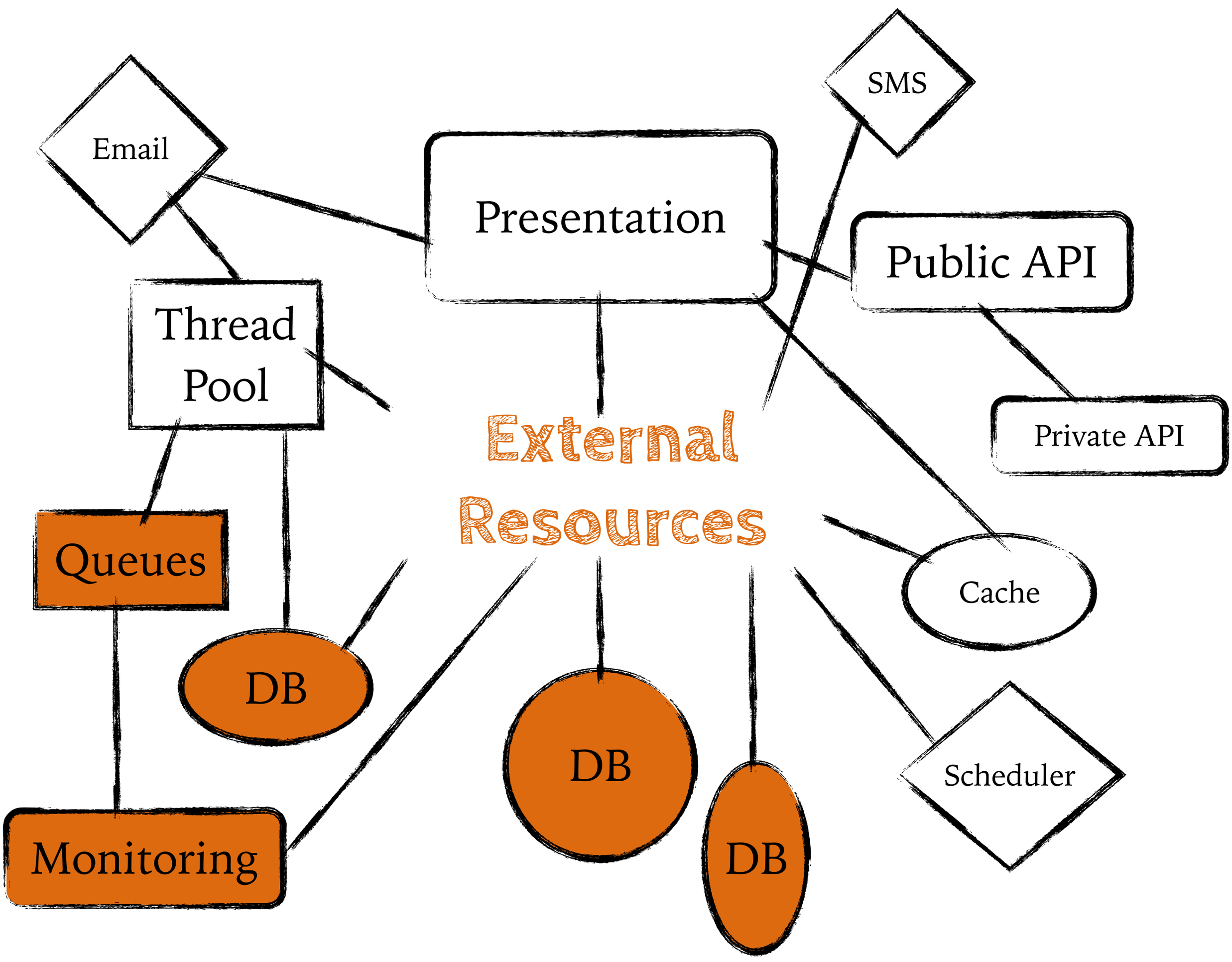
cognitect

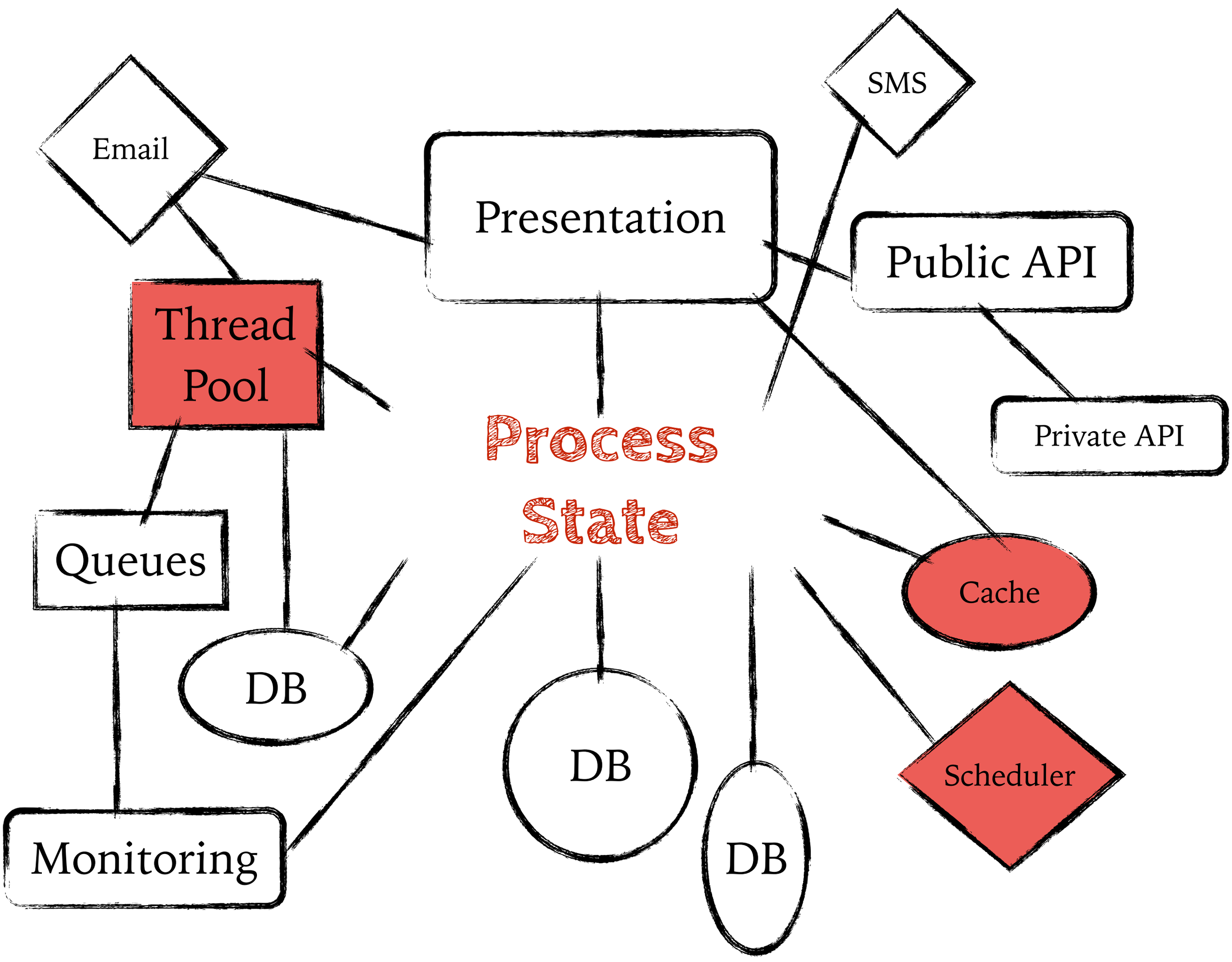

Presentation
Business Logic
DB


Email
SMS
Presentation
Public API
Thread Pool
Private API
Business Logic
Queues
Cache
DB
DB
Scheduler
Monitoring
DB


Email
SMS
Presentation
Public API
Thread Pool
Private API
Queues
Cache
DB
DB
DB
Scheduler
Monitoring


Email
SMS
Presentation
Thread Pool
Public API
Private API
Static
Config
Queues
DB
Cache
DB
DB
Scheduler
Monitoring


Email
SMS
Presentation
Public API
Thread Pool
Private API
External Resources
Queues
DB
Cache
DB
DB
Scheduler
Monitoring


Email
SMS
Presentation
Public API
Thread Pool
Private API
Process State
Queues
Cache
DB
DB
Scheduler
Monitoring
DB

Structure Built-In

Structure Built-In

```
public class Foo {

   private URL url;

   private Socket connection;

   public Foo(URL url) {...}

   public void connect() {...}

   public void shutdown() {...}
}
```

Structure Built-In

```
public class Foo {

   private URL url;
   private Socket connection;

   public Foo(URL url) {...}

   public void connect() {...}

   public void shutdown() {...}
}
```

Static config

Runtime state

Constructor

Lifecycle

Not a Lot of Structure

Not a Lot of Structure

```
(ns my-app.database)

(def url "…")

(def connection (atom nil))

(defn connect! []
  (swap! connection …))

(defn shutdown []
  (swap! connection …))
```

Not a Lot of Structure

```
(ns my-app.database)
```

Not instantiable

```
(def url "…")
```

Singletons

```
(def connection (atom nil))
```

Global mutable variables

```
(defn connect! []
  (swap! connection …))
```

Global effects

```
(defn shutdown []
  (swap! connection …))
```

Bootstrapping

```
(defn start-all! []
  (database/connect!)
  (create-queues!)
  (start-thread-pool!)
  ...
  (start-web-server!))
```

Bootstrapping

```
(defn start-all! []
  (database/connect!)
  (create-queues!)
  (start-thread-pool!)
  …
  (start-web-server!))
```

All modifying global state

Manual ordering

Bootstrapping

```
(defn start-all! []
  (database/connect!)
  (create-queues!)
  (start-thread-pool!)
  ...
  (start-web-server!))
```

All modifying global state

Manual ordering

Hidden dependencies

Hard to test

Just Enough Structure

Component

Component

- Immutable data structure (map or record)
- Public API
- Managed lifecycle
- Relationships to other components

Component

It's an object!

- Immutable data structure (map or record)
- Public API
- Managed lifecycle
- Relationships to other components

Component

It's an object!

- Immutable data structure (map or record)
- Public API
- Managed lifecycle
- Relationships to other components

Focus on behavior

State Wrapper Component

```
(defrecord DB [host conn] …)
```

State Wrapper Component

Encapsulates state

```
(defrecord DB [host conn] …)
```

State Wrapper Component

Encapsulates state

```
(defrecord DB [host conn] …)
```

Opaque to most consumers

Public API

```
(defrecord DB [host conn] …)

(defn query [db & …]
  (.doQuery (:conn db) …))


(defn insert [db & …]
  (.doStatement (:conn db) …))
```

Public API

```
(defrecord DB [host conn] …)
```

Take component as argument

```
(defn query [db & …]
  (.doQuery (:conn db) …))


(defn insert [db & …]
  (.doStatement (:conn db) …))
```

Public API

```
(defrecord DB [host conn] …)
```

```
(defn query [db & …]
  (.doQuery (:conn db) …))
```

May have side effects

```
(defn insert [db & …]
  (.doStatement (:conn db) …))
```

Public API

```
(defrecord DB [host conn] …)
```

```
(defn query [db & …]
  (.doQuery (:conn db) …))
```

```
(defn insert [db & …]
  (.doStatement (:conn db) …))
```

May use internal state

Lifecycle: Constructor

```
(defrecord DB [host conn] ...)


(defn db [host]
  (map->DB {:host host}))
```

Lifecycle: Constructor

```
(defrecord DB [host conn] ...)


(defn db [host]
  (map->DB {:host host}))
```

Creates initial state

Lifecycle: Constructor

```
(defrecord DB [host conn] ...)


(defn db [host]
  (map->DB {:host host}))
```

Creates initial state

No side effects

Lifecycle: Transitions

```
(ns com.stuartsierra.component)

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

Default implementation returns component

Lifecycle: Transitions

```
(defrecord DB [host conn]
  component/Lifecycle

  (start [this]
    (assoc this
      :conn (Driver/connect host)))

  (stop [this]
    (.stop conn)
    this))
```

Lifecycle: Transitions

```
(defrecord DB [host conn]
  component/Lifecycle

  (start [this]
    (assoc this
      :conn (Driver/connect host)))

  (stop [this]
    (.stop conn)
    this))
```

Lifecycle protocol

Lifecycle: Transitions

```
(defrecord DB [host conn]
  component/Lifecycle

  (start [this]
    (assoc this
      :conn (Driver/connect host)))

  (stop [this]
    (.stop conn)
    this))
```

May have side effects

Lifecycle: Transitions

```
(defrecord DB [host conn]
  component/Lifecycle

  (start [this]
    (assoc this
      :conn (Driver/connect host)))

  (stop [this]
    (.stop conn)
    this))
```

Always return updated component

Service Provider Component

```
(defrecord Email [endpoint api-key] …)
```

```
(defn send [email address body]
  …)
```

Service Provider Component

Encapsulates state

```
(defrecord Email [endpoint api-key] …)
```

```
(defn send [email address body]
  …)
```

Service Provider Component

Encapsulates state

```
(defrecord Email [endpoint api-key] …)
```

Public API provides services

```
(defn send [email address body]
  …)
```

Domain Model?

```
public class Customer {

   private String name;
   private Address address;

   public void notify() {…}
   …
}
```

Domain Model?

```
public class Customer {

    private String name;
    private Address address;

    public void notify() {…}
    …
}
```

Just structured data

Domain Model?

```
public class Customer {

    private String name;
    private Address address;

    public void notify() {…}
    …
}
```

Just structured data

Mingled with behavior

Let Data Be Data

Let Data Be Data

- Clojure maps, records, vectors, …
- Immutable values
- No behavior

Domain Model Component

```
(defrecord Customers [db email])
```

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Domain Model Component

```
(defrecord Customers [db email])
```

Represents aggregate operations

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Domain Model Component

```
(defrecord Customers [db email])
```

Encapsulates related dependencies

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Domain Model Component

```
(defrecord Customers [db email])
```

Public API takes component as argument

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Domain Model Component

```
(defrecord Customers [db email])
```

Gets dependencies out of component

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Domain Model Component

```
(defrecord Customers [db email])
```

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Uses public API of other components

Domain Model Component

```
(defrecord Customers [db email])
```

```
(defn notify [customers name message]
  (let [{:keys [db email]} customers
        address (query db … name)]
    (send email address message)))
```

Uses public API of other components

Treats other components as opaque

Constructor with Dependencies

```
(defn customers []
  (component/using
    (map->Customers {})
    [:db :email]))
```

Constructor with Dependencies

```
(defn customers []
  (component/using
    (map->Customers {})
    [:db :email]))
```

Declare dependencies by name

(Adds metadata)

Combining Components

System Map

```
(defn system […]
  (component/system-map
   :customers (customers)
   :db (db …)
   :email (->Email)
   …))
```

System Map

System constructor

```
(defn system […]
  (component/system-map
   :customers (customers)
   :db (db …)
   :email (->Email)
   …))
```

System Map

```
(defn system […]
  (component/system-map
    :customers (customers)
    :db (db …)
    :email (->Email)
    …))
```

Associates names with components

System Map

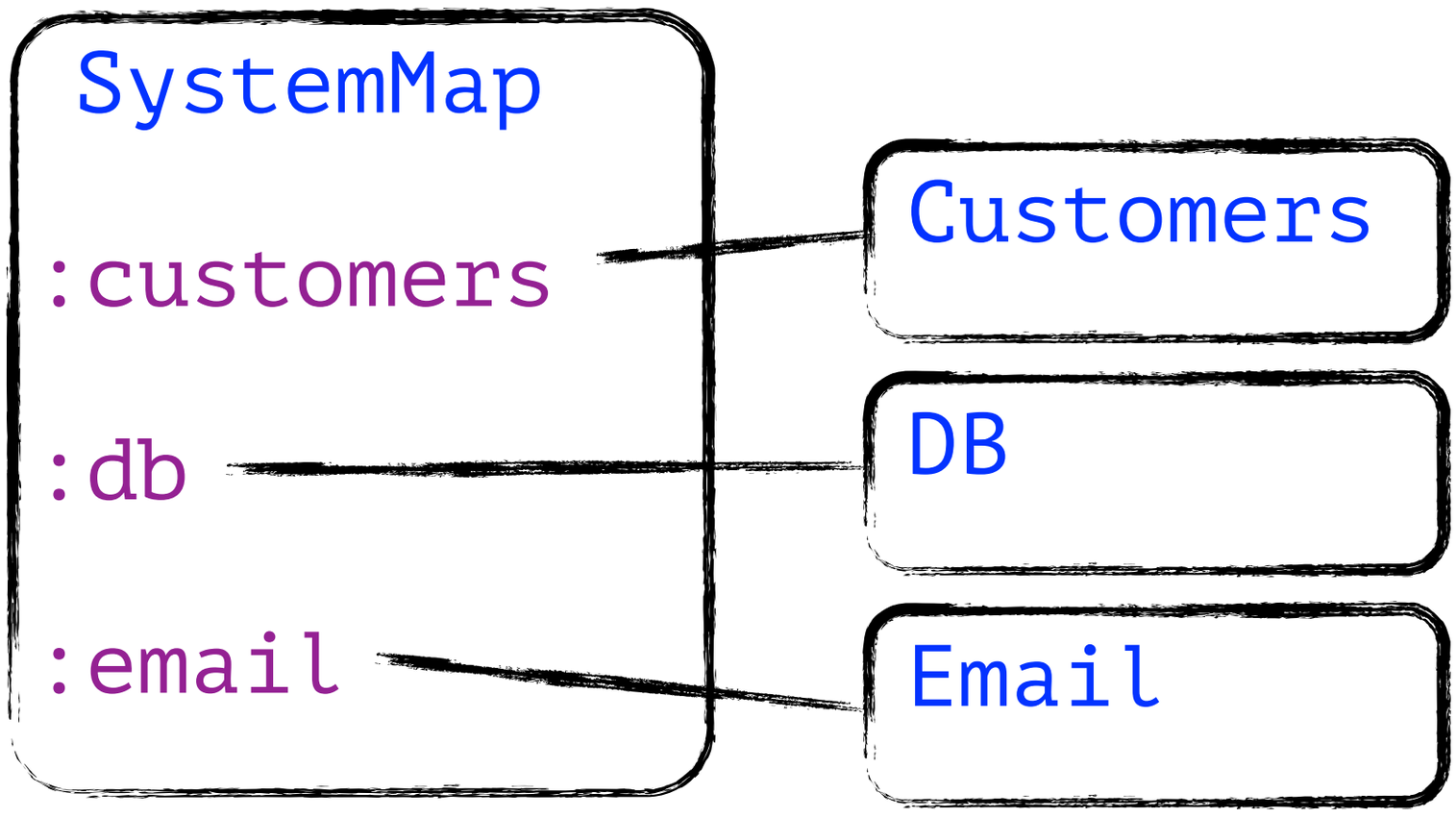
```
(defn system […]
  (component/system-map
    :customers (customers)
    :db (db …)
    :email (->Email)
    …))
```

Associates names with components

Manages lifecycle

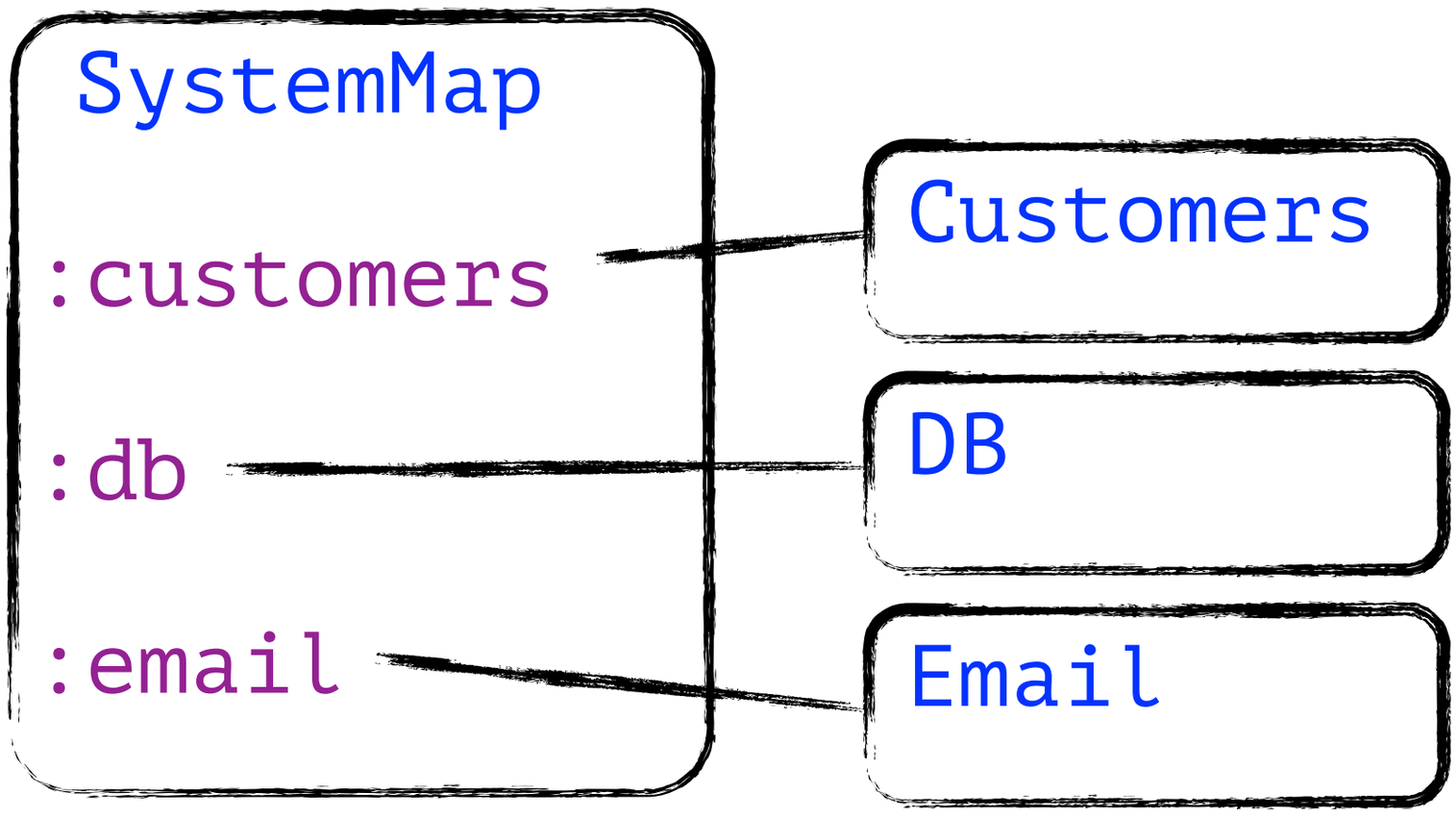
Provides dependencies

System

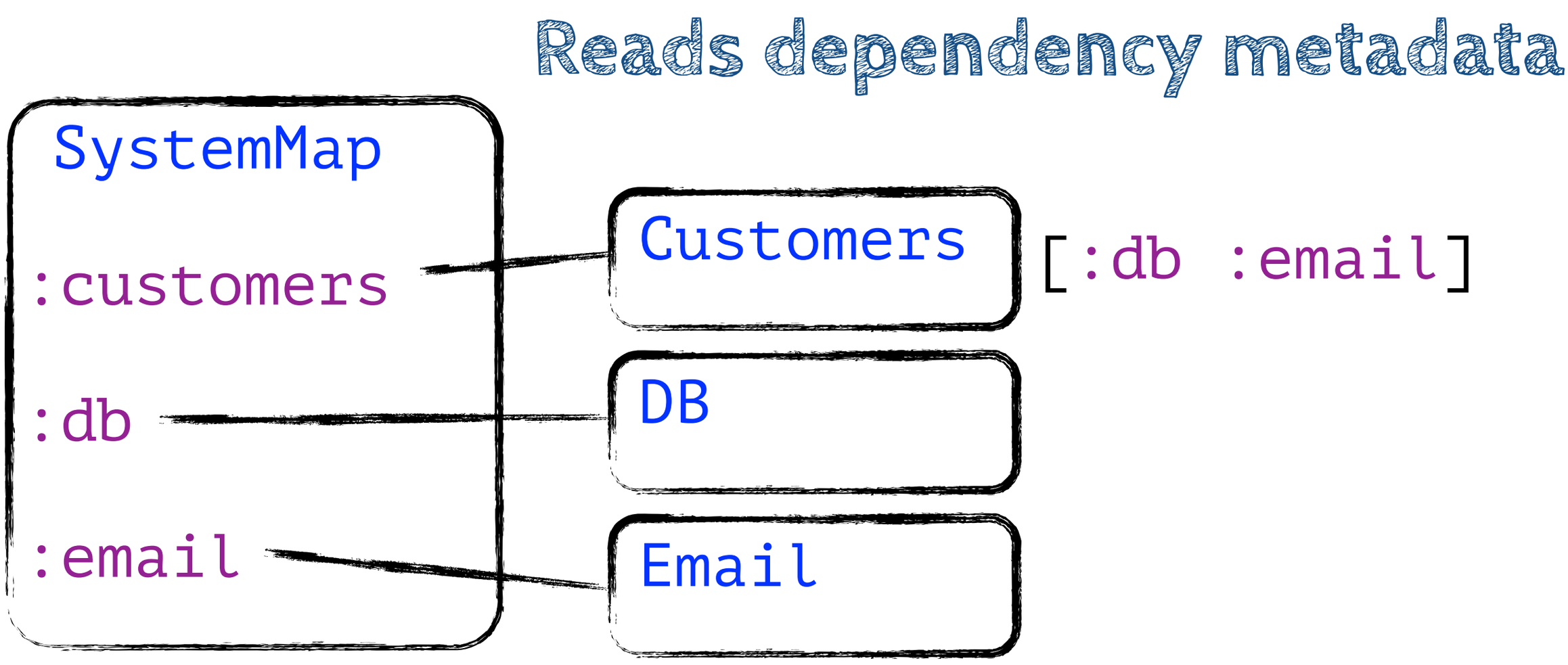


Starting a System

```
(component/start the-system)
```

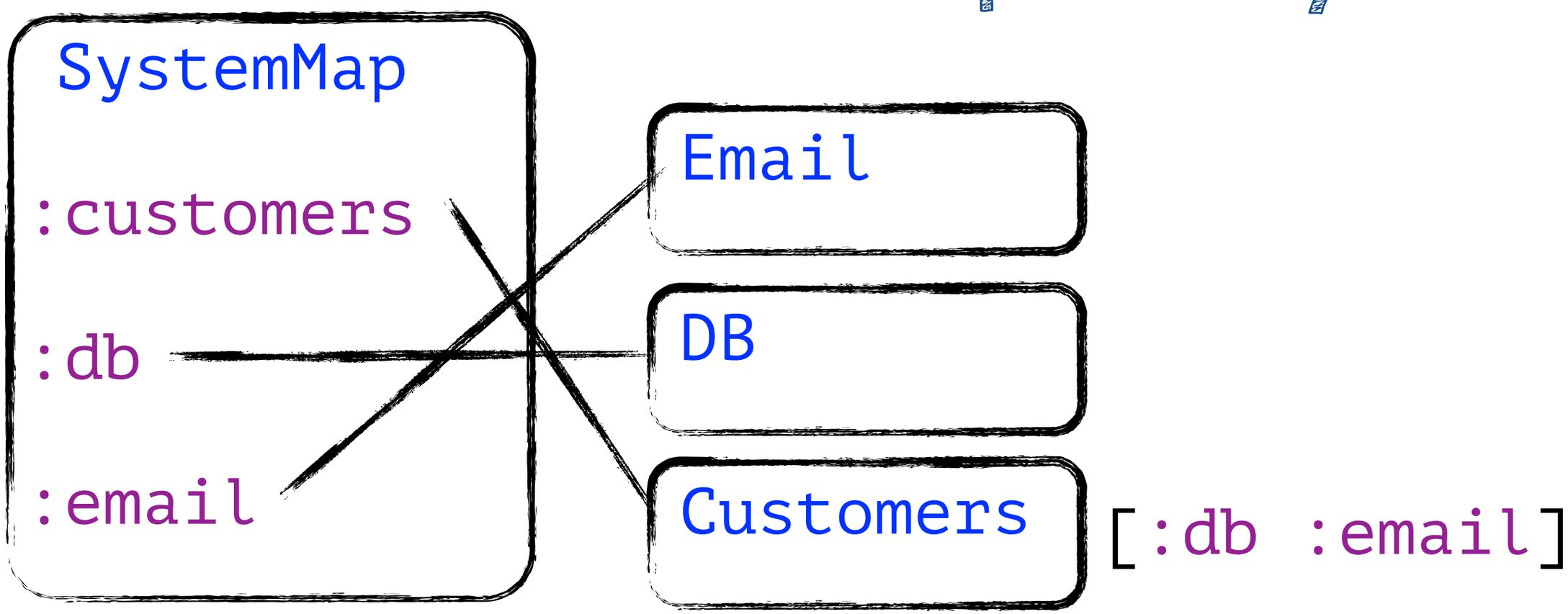


Starting a System



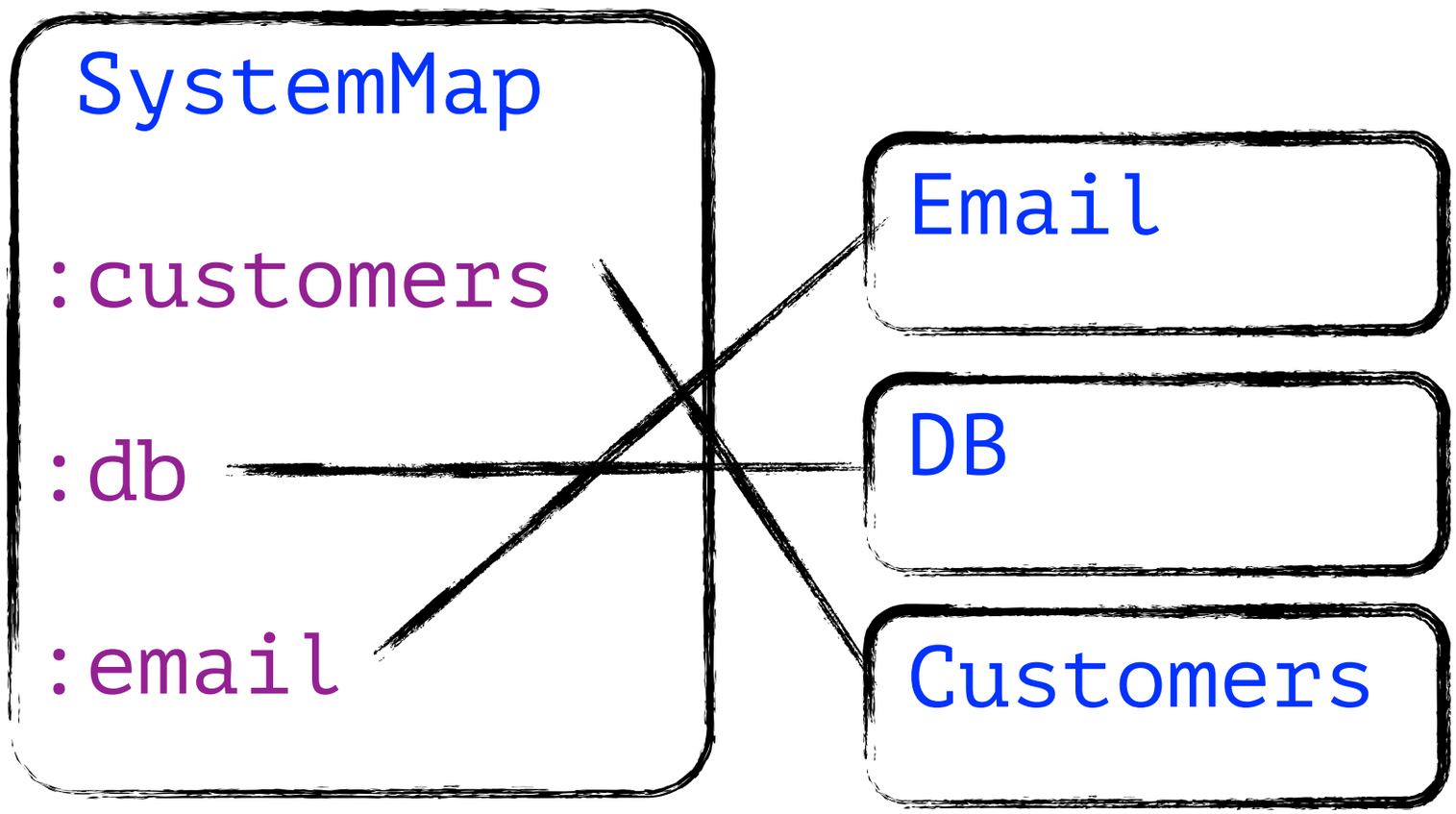
Starting a System

Sorts in dependency order

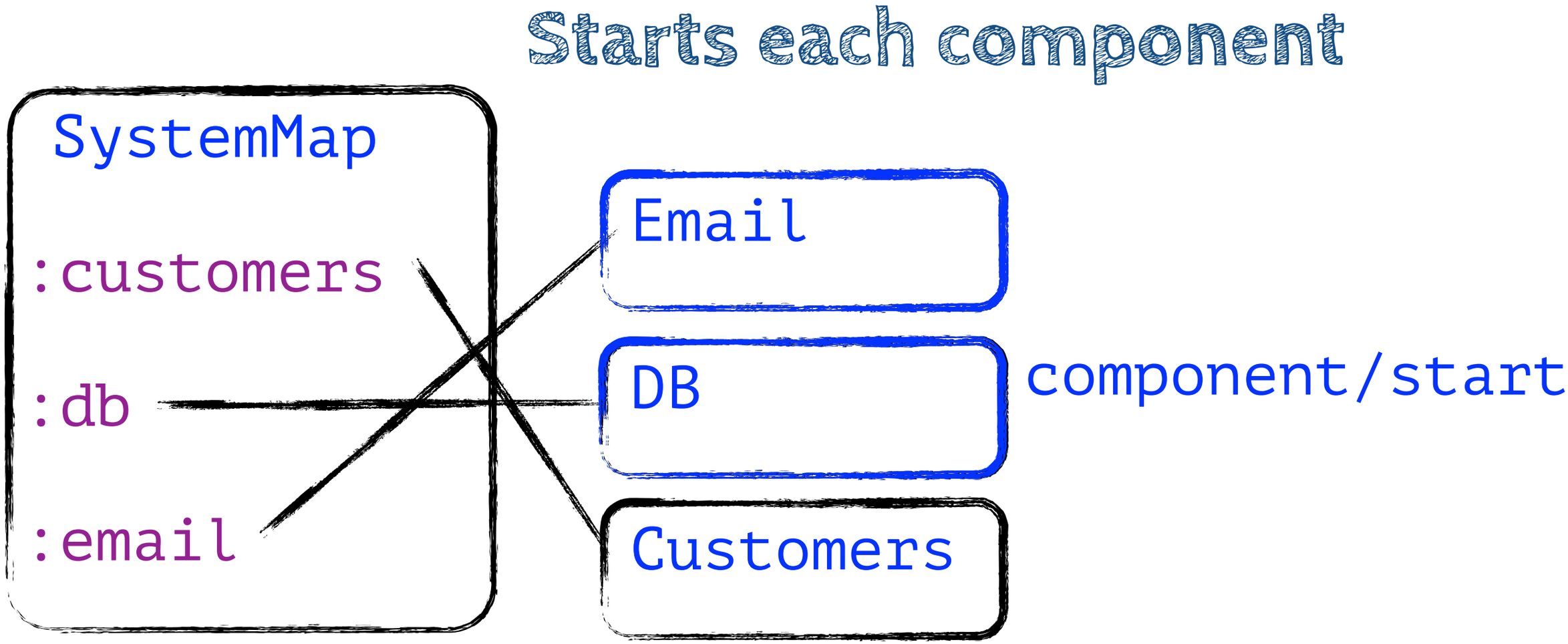


Starting a System

Starts each component

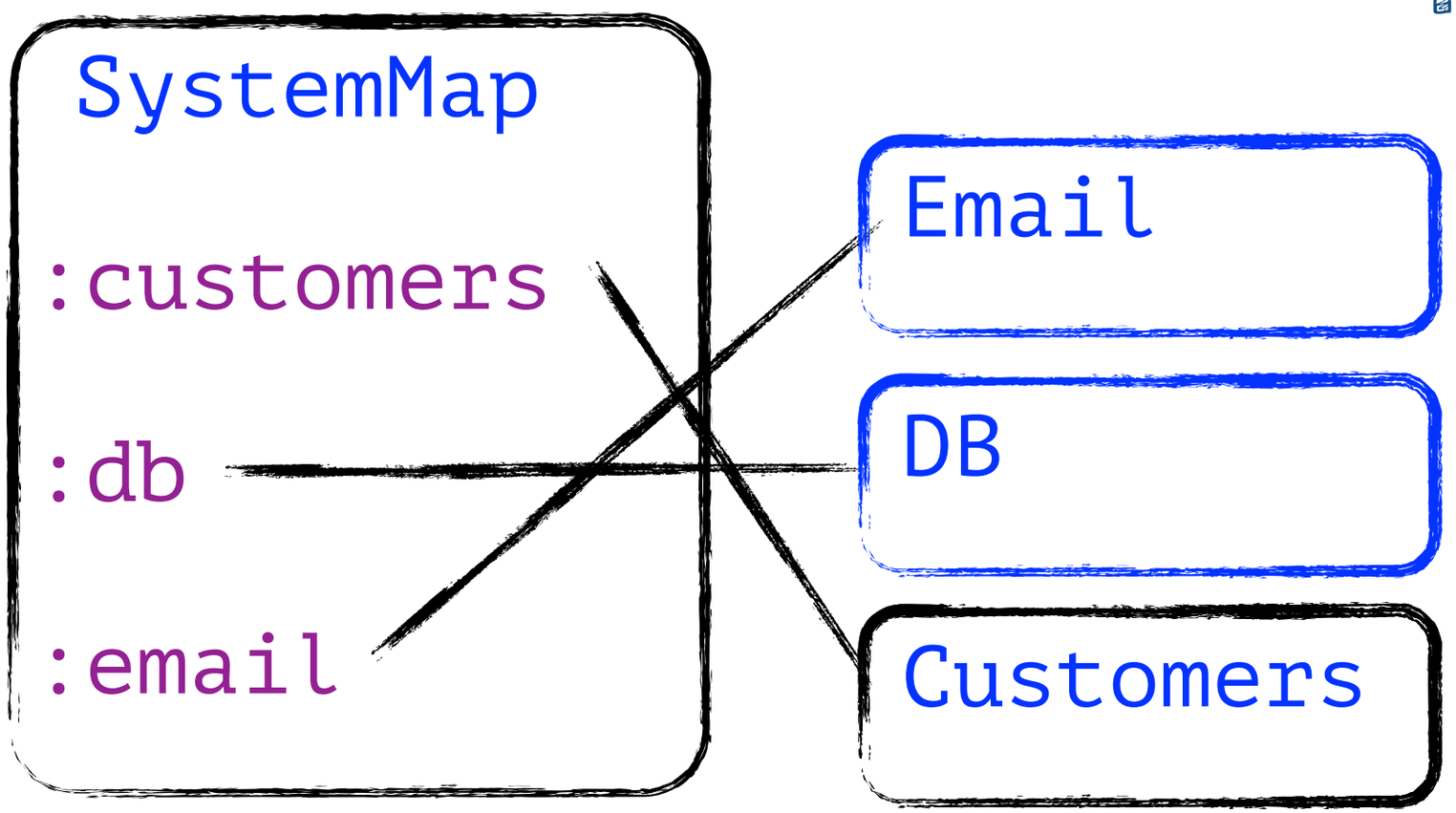


Starting a System



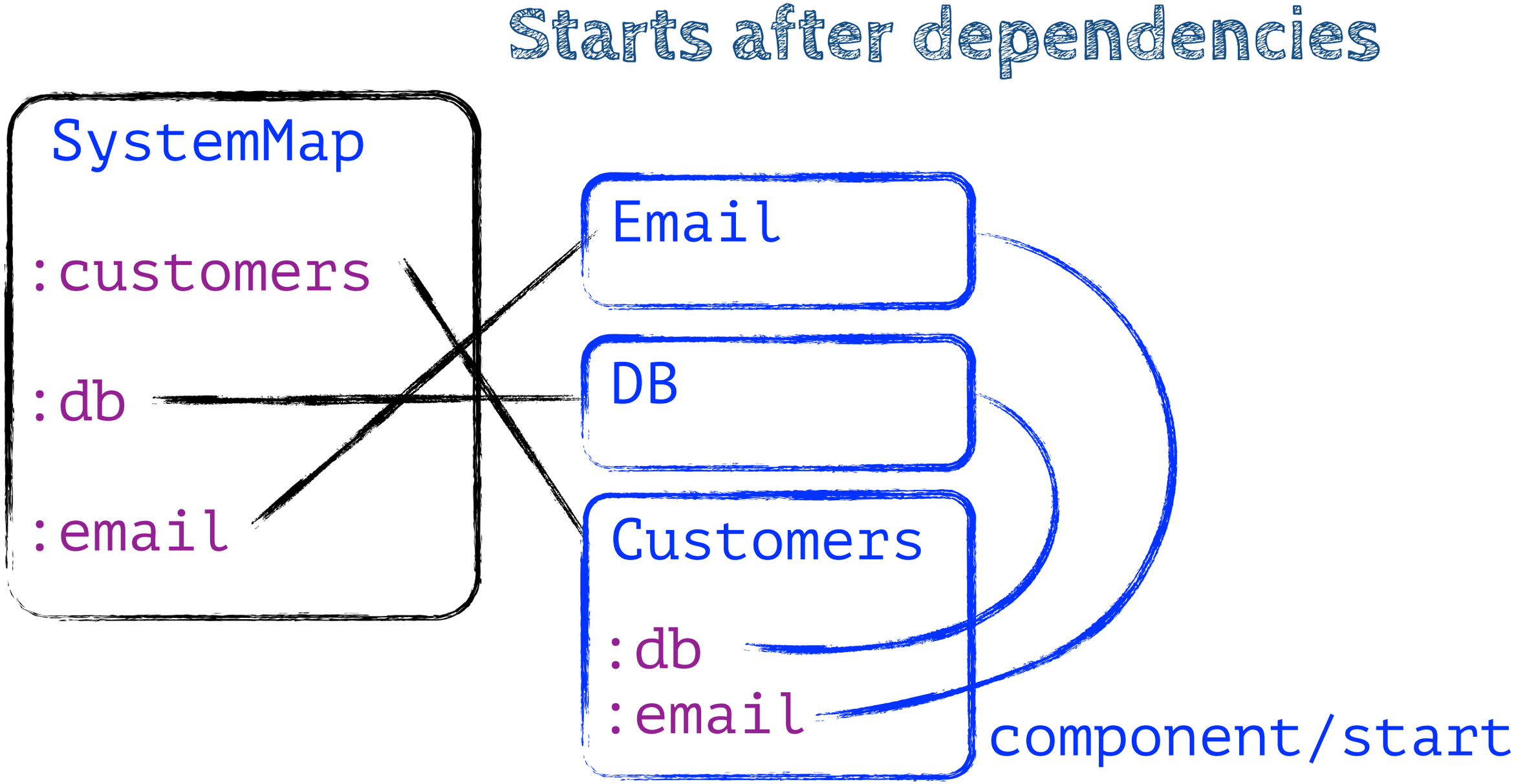
Starting a System

Associates dependencies



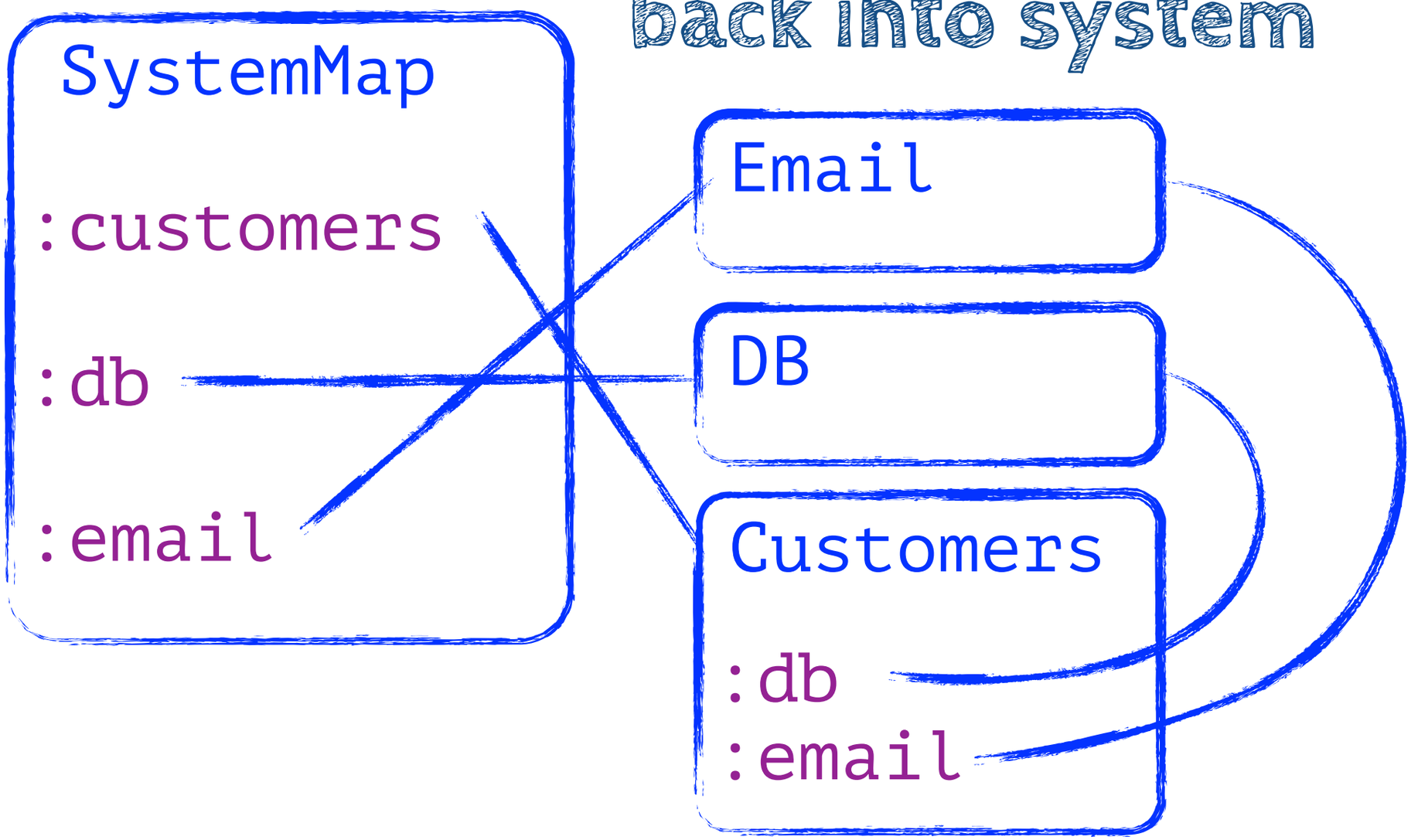

```
assoc :db
assoc :email
```

Starting a System

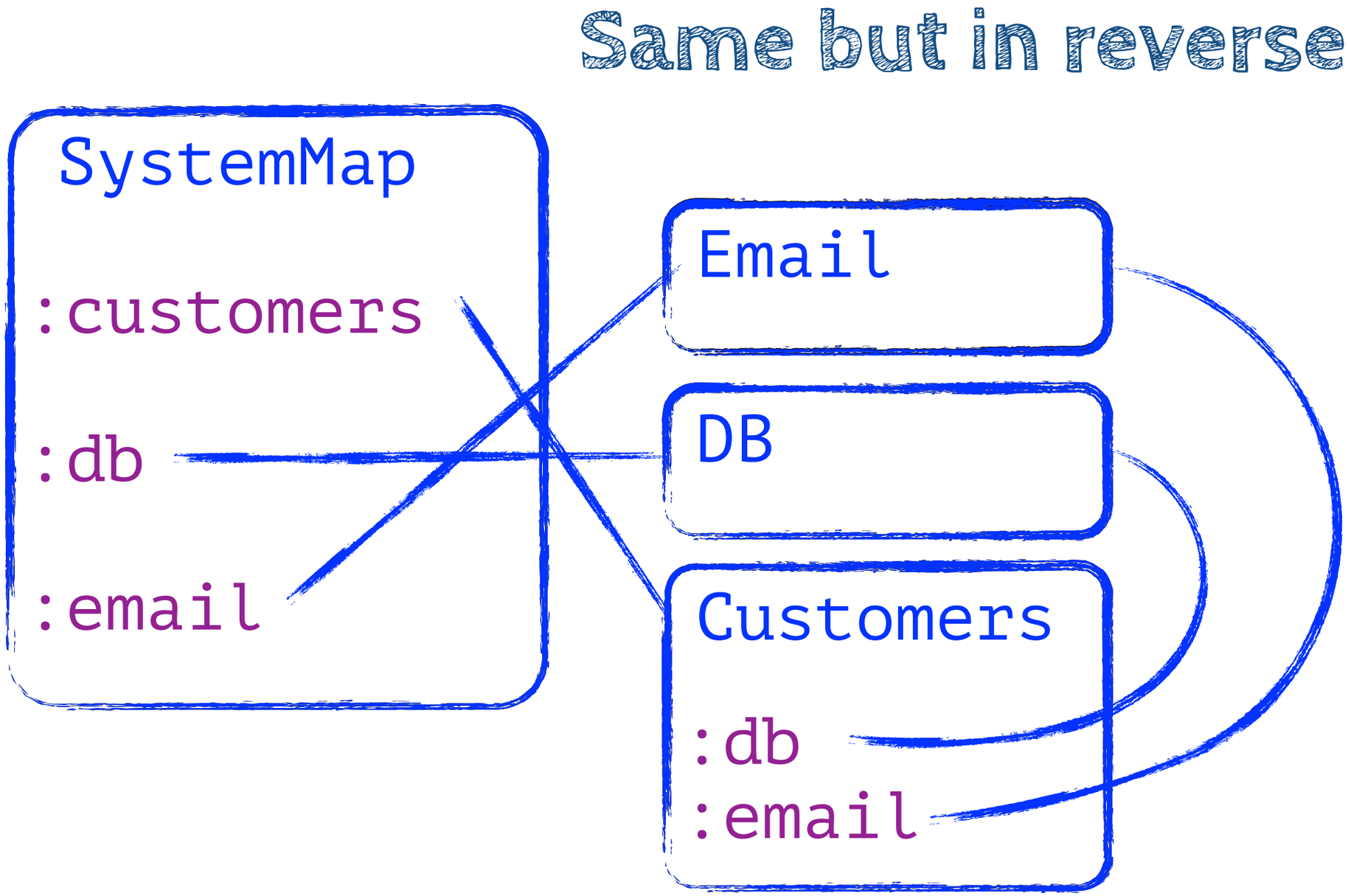


Starting a System

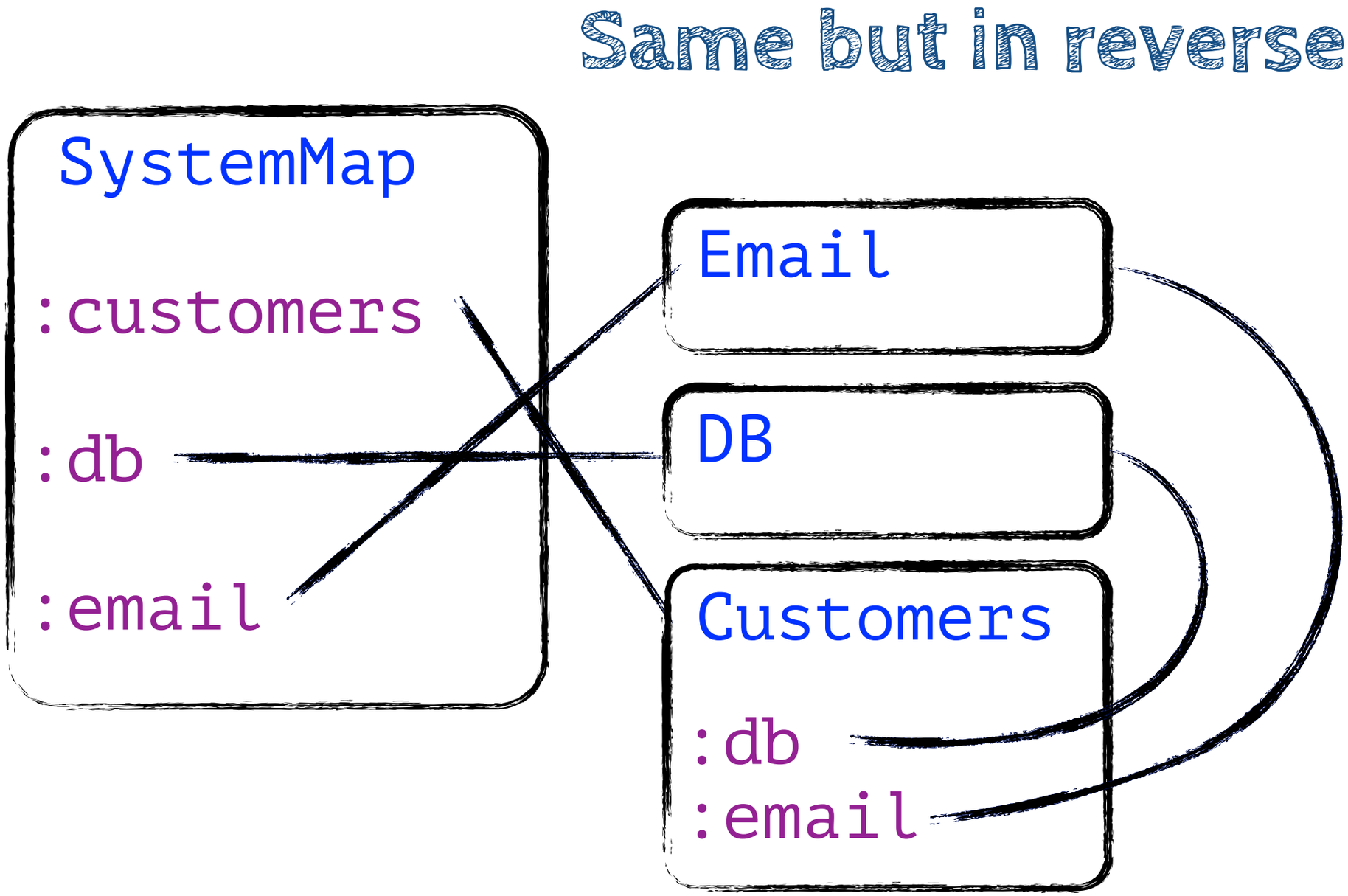
Associates updated components back into system



Stopping a System



Stopping a System



Before Start

```
#SystemMap{
   :customers
      #Customers{:db nil
                 :email nil}
   :db
      #DB{…}
   :email
      #Email{…}}
```

Before Start

```
#SystemMap{
  :customers
    #Customers{:db nil
               :email nil}
  :db
    #DB{…}
  :email
    #Email{…}}
```

Dependencies unset

After Start

```
#SystemMap{
   :customers
     #Customers{:db #DB{…}
                :email #Email{…}}
   :db
     #DB{…}
   :email
     #Email{…}}
```

After Start

Assoc'd dependencies

```
#SystemMap{
  :customers
    #Customers{:db #DB{…}
               :email #Email{…}}
  :db
    #DB{…}
  :email
    #Email{…}}
```

After Start

```
#SystemMap{
   :customers
      #Customers{:db #DB{…}
                 :email #Email{…}}
:db
  #DB{…}
:email
  #Email{…}}
```

Assoc'd dependencies

Same values

After Start

```
#SystemMap{
   :customers
      #Customers{:db #DB{…}
                 :email #Email{…}}
:db
  #DB{…}
:email
  #Email{…}
```

Assoc'd dependencies

Same values

As returned by 'start'

Injecting Alternatives

```
(defn test-system […]
  (assoc (system …)
    :email (stub-email)
    :db (test-db)))
```

Injecting Alternatives

```
(defn test-system […]
  (assoc (system …)
    :email (stub-email)
    :db (test-db)))
```

Assoc alternate components into system map

Injecting Alternatives

```
(defn test-system […]
  (assoc (system …)
    :email (stub-email)
    :db (test-db)))
```

Assoc alternate components into system map

Before start

Stub Service for Testing

```
(defprotocol Send
  (send [email-service address body]))

(defrecord StubEmail []
  Send
  (send [_ address body]
    ;; … record or save input …
    ))

(defn stub-email []
  (->StubEmail))
```

Stub Service for Testing

Function into protocol

```
(defprotocol Send
  (send [email-service address body]))

(defrecord StubEmail []
  Send
  (send [_ address body]
    ;; … record or save input …
    ))

(defn stub-email []
  (->StubEmail))
```

Stub Service for Testing

Function into protocol

```
(defprotocol Send
  (send [email-service address body]))

(defrecord StubEmail []
  Send
  (send [_ address body]
    ;; … record or save input …
    ))

(defn stub-email []
  (->StubEmail))
```

Stub implementation

Constructor

DB for Testing

```
(defrecord TestDB [host conn]
  component/Lifecycle

  (start [this]
    (let [conn (Driver/connect host)]
      (load-seed-data conn)
      (assoc this :conn conn)))

(stop [this]
  (drop-database conn)
  (.stop conn)
  this))
```

DB for Testing

```
(defrecord TestDB [host conn]
  component/Lifecycle
  (start [this]
    (let [conn (Driver/connect host)]
      (load-seed-data conn)
      (assoc this :conn conn)))

(stop [this]
  (drop-database conn)
  (.stop conn)
  this))
```

Doesn't mock DB

DB for Testing

```
(defrecord TestDB [host conn]
  component/Lifecycle

  (start [this]
    (let [conn (Driver/connect host)]
      (load-seed-data conn)
      (assoc this :conn conn)))

(stop [this]
  (drop-database conn)
  (.stop conn)
  this))
```

Doesn't mock DB

Creates/destroys for each use

Testing Business Logic

```
(deftest t-notify-customer
  (let [system (component/start
                (test-system))
        {:keys [customers]} system]
    (try
      (notify customers "bob" "Hi, Bob!")
      ;; … make some assertions …
      (finally
        (component/stop system)))))
```

Testing Business Logic

Isolated system

```
(deftest t-notify-customer
  (let [system (component/start
                (test-system))
        {:keys [customers]} system]
    (try
      (notify customers "bob" "Hi, Bob!")
      ;; … make some assertions …
      (finally
        (component/stop system)))))
```

Var Substitution & Asynchrony

```
(deftest t-notify-customer
  (with-redefs [send …]
    (binding [*db* …]
      (notify …)
      (is …))))
```

Var Substitution & Asynchrony

```
(deftest t-notify-customer
  (with-redefs [send …]
    (binding [*db* …]
      (notify …)
      (is …))))
```

Delimited in time

Var Substitution & Asynchrony

```
(deftest t-notify-customer
  (with-redefs [send …]
    (binding [*db* …]
      (notify …)
      (is …))))
```

Delimited in time

Potential race conditions

Var Substitution & Asynchrony

```
(deftest t-notify-customer
  (with-redefs [send …]
    (binding [*db* …]
      (notify …)
      (is …))))
```

Delimited in time

Potential race conditions

Tightly coupled to implementation

Var Substitution & Asynchrony

```
(deftest t-notify-customer
  (with-redefs [send …]
    (binding [*db* …]
      (notify …)
      (is …))))
```

Delimited in time

Potential race conditions

Tightly coupled to implementation

Wrong level of granularity

Entry Points

Entry Point: Main

```
(ns com.example.app
  (:require [com.stuartsierra.component
             :as component]))

(defn -main [...]
  (component/start (system ...)))
```

Entry Point: Global

```
(def sys (atom nil))

(defn init! []
  (reset! sys (system)))

(defn start! []
  (reset! sys
    (component/start @sys)))

(defn stop! []
  (reset! sys
    (component/stop @sys)))
```

Entry Point: Global

```
(def sys (atom nil))

(defn init! []
  (reset! sys (system)))

(defn start! []
  (reset! sys
    (component/start @sys)))

(defn stop! []
  (reset! sys
    (component/stop @sys)))
```

Exactly one mutable global for system

Entry Point: Global

```
(def sys (atom nil))

(defn init! []
  (reset! sys (system)))

(defn start! []
  (reset! sys
    (component/start @sys)))

(defn stop! []
  (reset! sys
    (component/stop @sys)))
```

Exactly one mutable global for system

Functions on values

Entry Point: Global

```
(def sys (atom nil))

(defn init! []
  (reset! sys (system)))

(defn start! []
  (reset! sys
    (component/start @sys)))

(defn stop! []
  (reset! sys
    (component/stop @sys)))
```

Exactly one mutable global for system

Functions on values

Transition from one state to the next

Entry Point: Global

```
(def sys (atom nil))

(defn init! []
  (reset! sys (system)))

(defn start! []
  (reset! sys
    (component/start @sys)))

(defn stop! []
  (reset! sys
    (component/stop @sys)))
```

Not swap! because of side-effects

Web App: Static Routes

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

(def server (start-server routes))
```

Web App: Static Routes

Static

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

(def server (start-server routes))
```

Web App: Static Routes

Static

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

(def server (start-server routes))
```

How to get components in here?

Web App: Static Routes

Static

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

How to get components in here?

```
(def server (start-server routes))
```

Don't just deref system everywhere!

Entry Point: Web Request

```
(defn system-middleware [ring-handler]
  (fn [request]
   (ring-handler
    (assoc request
      ::web-app (::web-app @sys)))))
```

Entry Point: Web Request

```
(defn system-middleware [ring-handler]
  (fn [request]
    (ring-handler
     (assoc request
       ::web-app (::web-app @sys)))))
```

Get component from system
at beginning of each request

Entry Point: Compiling Routes

```
(defn make-routes [web-app]
  (routes
    (GET "/foo" [request]
         (get-foo web-app request))
    (POST "/bar" [request]
         (do-bar web-app request))))
```

Entry Point: Compiling Routes

Construct routes
in a function

```
(defn make-routes [web-app]
  (routes
    (GET "/foo" [request]
         (get-foo web-app request))
    (POST "/bar" [request]
         (do-bar web-app request))))
```

Entry Point: Compiling Routes

Construct routes in a function

Close over component

```
(defn make-routes [web-app]
  (routes
   (GET "/foo" [request]
        (get-foo web-app request))
   (POST "/bar" [request]
         (do-bar web-app request))))
```

Entry Point: Compiling Routes

Construct routes in a function

Close over component

```
(defn make-routes [web-app]
  (routes
   (GET "/foo" [request]
        (get-foo web-app request))
   (POST "/bar" [request]
         (do-bar web-app request))))
```

Pass to handlers

Entry Point: Compiling Routes

```
(defrecord WebApp [customers server]
  component/Lifecycle
  (start [this]
    (assoc this :server
      (start-server (make-routes this))))
(stop [this]
  (stop-server server)
  this))
```

Entry Point: Compiling Routes

```
(defrecord WebApp [customers server]
  component/Lifecycle
  (start [this]
    (assoc this :server
      (start-server (make-routes this))))
  (stop [this]
    (stop-server server)
    this))
```

Create routes in 'start'

Entry Point: Compiling Routes

```
(defrecord WebApp [customers server]
  component/Lifecycle
  (start [this]
    (assoc this :server
      (start-server (make-routes this))))
  (stop [this]
    (stop-server server)
    this))
```

Create routes in 'start'

Capture component

Extensions

Renaming Dependencies

```
(defn test-system []
  (system-map
   :test/database (test-db)
   :test/email (stub-email)
   :customers (component/using
               (customers)
               {:db    :test/database
                :email :test/email})))
```

Renaming Dependencies

```
(defn test-system []
  (system-map
   :test/database (test-db)
   :test/email (stub-email)
   :customers (component/using
               (customers)
               {:db     :test/database
                :email  :test/email})))
```

Component's dependency keys

Renaming Dependencies

```
(defn test-system []
  (system-map
   :test/database (test-db)
   :test/email (stub-email)
   :customers (component/using
               (customers)
               {:db    :test/database
                :email :test/email})))
```

Component's dependency keys

System keys

Merging Systems

```
;; System A
{:a/web-app …
 :a/server …
 :db …
 :email …}
```

```
;; System B
{:b/web-app …
 :b/server …
 :db …
 :email …}
```

Merging Systems

```
;; System A
{:a/web-app …
 :a/server …
 :db …
 :email …}
```

```
;; System B
{:b/web-app …
 :b/server …
 :db …
 :email …}
```

```
(merge system-a system-b)
```

Merging Systems

```
;; System A
{:a/web-app …
 :a/server …
 :db …
 :email …}
```

```
;; System B
{:b/web-app …
 :b/server …
 :db …
 :email …}
```

Namespace-qualified keys prevent clashes

```
(merge system-a system-b)
```

Merging Systems

```
;; System A
{:a/web-app …
 :a/server …
 :db …
 :email …}
```

```
;; System B
{:b/web-app …
 :b/server …
 :db …
 :email …}
```

```
(merge system-a system-b)
```

Common keys reused

Great for testing

core.async

```
(defrecord Processor [input-ch output-ch]
  component/Lifecycle
  (start [this]
    (assoc this
      :go
      (go-loop []
        (when-some [val (<! input-ch)]
          … ))))
(stop [this]
  (close! input-ch)
  this)))
```

core.async

```
(defrecord Processor [input-ch output-ch]
  component/Lifecycle
  (start [this]
    (assoc this
      :go
      (go-loop []
        (when-some [val (<! input-ch)]
          … ))))
  (stop [this]
    (close! input-ch)
    this)))
```

Channels as dependencies

core.async

```
(defrecord Processor [input-ch output-ch]
  component/Lifecycle
  (start [this]
    (assoc this
      :go
      (go-loop []
        (when-some [val (<! input-ch)]
          … ))))
  (stop [this]
    (close! input-ch)
    this)))
```

Channels as dependencies

Create event loop in start

core.async

```
(defn system []
  :events (chan 100)
  :log (chan 100)
  :processor (component/using
              (->Processor)
              {:input-ch  :events
               :output-ch :log})
...)
```

core.async

```
(defn system []
  :events (chan 100)
  :log (chan 100)
  :processor (component/using
              (->Processor)
              {:input-ch  :events
               :output-ch :log})
  ...)
```

Create channels in system

core.async

```
(defn system []
  :events (chan 100)
  :log (chan 100)
  :processor (component/using
              (->Processor)
              {:input-ch  :events
               :output-ch :log})
  ...)
```

Create channels in system

Connect to component

core.async

```
(defn system []
  :events (chan 100)
  :log (chan 100)
  :processor (component/using
              (->Processor)
              {:input-ch  :events
               :output-ch :log})
  ...)
```

Create channels in system

Connect to component

Insert mult / pipe / etc. for indirection

Summary

Advantages

- Explicit dependencies and clear boundaries
  - Isolation, decoupling
  - Easier to test, refactor
- Automatic ordering of start/stop
- Easy to swap in alternate implementations
- Everything at most one map lookup away

Disadvantages

- Requires whole-app buy-in
- System map is too big to inspect visually
- Cannot start/stop just part of a system
- Boilerplate
  - Records, constructors, Lifecycles, dependencies
  - Destructuring component fields in functions

Possible Future

- Additional lifecycle protocols
  - “init” acquires resources but does not start
  - “release” closes resources and drops dependencies

Possible Future

- Handle mutable containers for systems
- Allow individual components to start, stop, or change at runtime
- Deref container and get “current” component with latest dependencies
- Catch errors, mark component as “failed”

@stuartsierra
stuartsierra.com

github.com/stuartsierra/component