Clojure Introduction

@codemomentum

sebula.com

Motivation

- Software Development is about taking things apart.
- Clojure is a simple and elegant language.

  “Simplicity is a pre-requisite for reliability” (Dijkstra)

- Clojure demands that you raise your game, and pays you back for doing so.

Why Clojure?

- Extremely expressive language
- First-class functions
- Awesome Java Interop abilities
- Power of lisp (macros, function composition)
- FUNctional

Why Lisp?

- Little Syntax to Learn
- Say a lot more with less code, fewer lines, fewer bugs
- Bend your language to the problem, not the other way around
- Code as Data

Why Lisp?

When I hear people complain about Lisp's parentheses, it sounds to my ears like someone saying: "I tried one of those bananas, which you say are so delicious. The white part was ok, but the yellow part was very tough and tasted awful.

Paul Graham

Functional Programming

- OOP makes code understandable by encapsulating moving parts
- FP makes code understandable by minimizing moving parts

- Better fit for concurrency and multicore architectures

Agility

- Architectural agility is the agility you get from building a system that is **fundamentally simple**.

- Simplicity enables change, it's the primary source of real agility.

- If you're dragging an elephant around you're never going to be agile

* "Simple Made Easy" by Rich Hickey

Simple Use Case

```java
/**
 * used to execute a concrete command, such as an http call
 * a jmx-rpc
 * or  sending a jms message
 */
public interface CommandExecutor {
    Result execute(Document commandDescription);
}
```

Step1: Design

CommandExecutor.java

Executes the command that is described in the “command description”

Result and Document are wrappers around java.util.Map

Step2: Implement & Test

HttpCommandExecutor.java JmxCommandExecutor.java JmsCommandExecutor.java

Step3: Integrate

```java
/**
 * job executor will take a job from the queue, instantiate or reuse an existing command executor to execute it
 *
 */
public interface CommandExecutorService {
    Result execute(Map job);
}
```

CommandExecutorService.java

```java
/**
 * TODO- to be extended
 *
 */
public interface ExecutorRegistry {

    CommandExecutor getExecutor(String type);

}
```

ExecutorRegistry.java

Simple Use Case

Step3: Integrate

```java
public class DelegatingCommandExecutorService implements CommandExecutorService {

    final static Logger logger = LoggerFactory.getLogger(DelegatingCommandExecutorService.class);

    private ExecutorRegistry executorRegistry;

    public DelegatingCommandExecutorService(ExecutorRegistry executorRegistry) {
        this.executorRegistry = executorRegistry;
    }

    @Override
    public Result execute(Map jobMap) {
        DocumentBasedCommand job = new DocumentBasedCommand(new DefaultDocument(jobMap));
        CommandExecutor executor = executorRegistry.getExecutor(job.getType());
        if (null == executor) {
            logger.error("Dont know how to execute the job of type {}", job.getType());
            throw new RuntimeException("Dont know how to execute the job of type:" + job.getType());
        } else {
            logger.info("Delegating job: {} of type: {} to executor: {}", new Object[]{job.getId(), job.getType(), executor});
            return executor.execute(job.getCommandDescription());
        }
    }
}
```

DelegatingCommandExecutorService.java

```java
public class SimpleExecutorRegistry implements ExecutorRegistry {

    private Map<String, CommandExecutor> registry;

    public SimpleExecutorRegistry(Map<String, CommandExecutor> registry) {
        this.registry = registry;
    }

    @Override
    public CommandExecutor getExecutor(String type) {
        return registry.get(type);
    }
}
```

SimpleExecutorRegistry.java

Simple Use Case

Step4: In Action

```
<bean id="httpCommandExecutor" class="command.executor.HttpCommandExecutor">
    <constructor-arg index="0" ref="threadSafeHttpClient" />
</bean>


<bean id="executorRegistry" class="command.executor.registry.SimpleExecutorRegistry">
    <constructor-arg index="0">
        <map>
            <entry key="http_command" value-ref="httpCommandExecutor"/>
            <entry key="jmx_command" value-ref="jmxContextCommandExecutor"/>
        </map>
    </constructor-arg>
</bean>

<bean id="jobExecutor"
      class=".command.executor.service.DelegatingCommandExecutorService">
    <constructor-arg index="0" ref="executorRegistry"/>
</bean>
```

someApplicationContext.xml

This is brilliant if you are charging by the hour

Simple Use Case - Clojure

Delegator

```
(defonce pipelines {:analytics  store-analytics-event!
                    :crawlerlog store-crawler-log!})

(defn dispatch-event [event] "dispatch the event to the related handler based on event type"
  (log/debugf "Event body: %s" (String. (.getBody ^Event event)) )
  (let
      [parsed-event (json/read-str (String. (.getBody ^Event event)) :key-fn keyword)
       event-type (keyword (:event-type parsed-event))
       handler (event-type pipelines)]
    (if-not (nil? handler)
      (handler parsed-event)
      (log/errorf "Uknown event type: %s from event %s" event-type parsed-event)
      )
    )
  event
  )
```

Code for one of the handlers (Bonus)

```
(defn store-crawler-log! [parsed-event] "stores the event to Cassandra"
  (log/debugf "Storing Event: %s to cassandra " parsed-event)
  (let [event {:account (:account parsed-event)
               :date    (System/currentTimeMillis)
               :job     (json/write-str (:job parsed-event))
               :message (:message parsed-event)
               }
        ]
    (cas/insert-to-cassandra (:event-type parsed-event) event))
  )
```

What Clojure Looks Like

```
int function ( ) {
      return 5;
}
```

```
( fn [ ]
      5
)
```

```
1+1;
```

```
(+ 1 1)
```

```
System.out.println(“Hello World!”);
```

```
(println “Hello World!”)
```

Clojure Introduction - Language Primitives

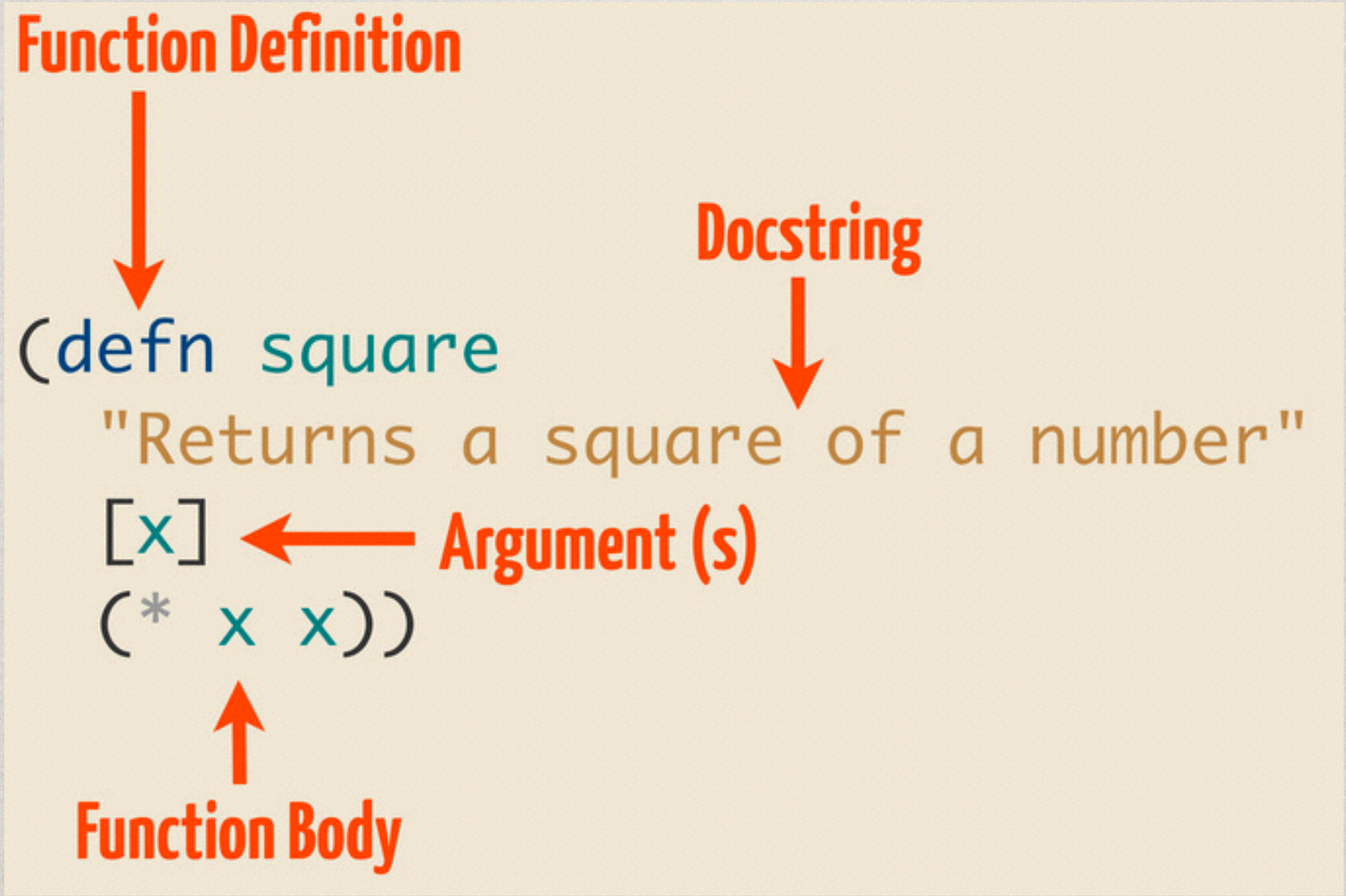
Boolean	true, false
Character	\a
Keyword	:a, :b
List	(1 2 3), (println x)
Map	{:marka "Bmw", :model “320d”, :renk “mavi”}
Nil	nil
Number	1, 4.5
Set	#{“a” “b” “c”}
String	“Ola!”
Symbol	user/foo, java.lang.String
Vector	[1 2 3]

Clojure Introduction - Special Forms

Every form not handled specially by a **special form** or **macro** is considered by the compiler to be an **expression**, which is evaluated to **yield a value**.

def	(def symbol init?)	var	(var symbol)
if	(if test then else?)	fn	(fn name? [params*] exprs*)
do	(do exprs*)	loop	(loop [bindings*] exprs*)
let	(let [bindings*] exprs*)	recur	(recur exprs*)
quote	(quote form)	try/catch/throw	(try expr* catch-clause* finally-clause?)

Clojure Introduction - Functions



Clojure Introduction - Collections

{ }
Map
{“a” 1 “b” 2}

#{ }
Set
{“a” “b” “c”}

‘( )
List
‘(1 2 3)

[ ]
Vector
[“a” “b” “c”]

The trick is to efficiently use primitive data structures and library functions

Clojure Introduction - Examples

=>(doc map)

……

=>(map inc [3 4 5])
(4 5 6)

```
public static List<Integer> incByOne(List<Integer> input){
  List<Integer> mapped = new ArrayList<Integer>();
   for(int i:input){
     mapped.add(i + 1);
   }
   return mapped;
}
```

=> (doc reduce)

…….

=> (reduce + [3 4 5])
12

Clojure Introduction - Examples

=>(doc filter)

……

=> (filter even? [1 2 3 4])
(2 4)

=>(doc reductions)

……

=> (reductions + [1 2 3 4])
(1 3 6 10)

=> (doc remove)

……

=> (remove nil? [1 nil 2 nil])
(1 2)

=> (doc remove)

……

=> (remove nil? [1 nil 2 nil])
(1 2)

Clojure Introduction - Key Concepts

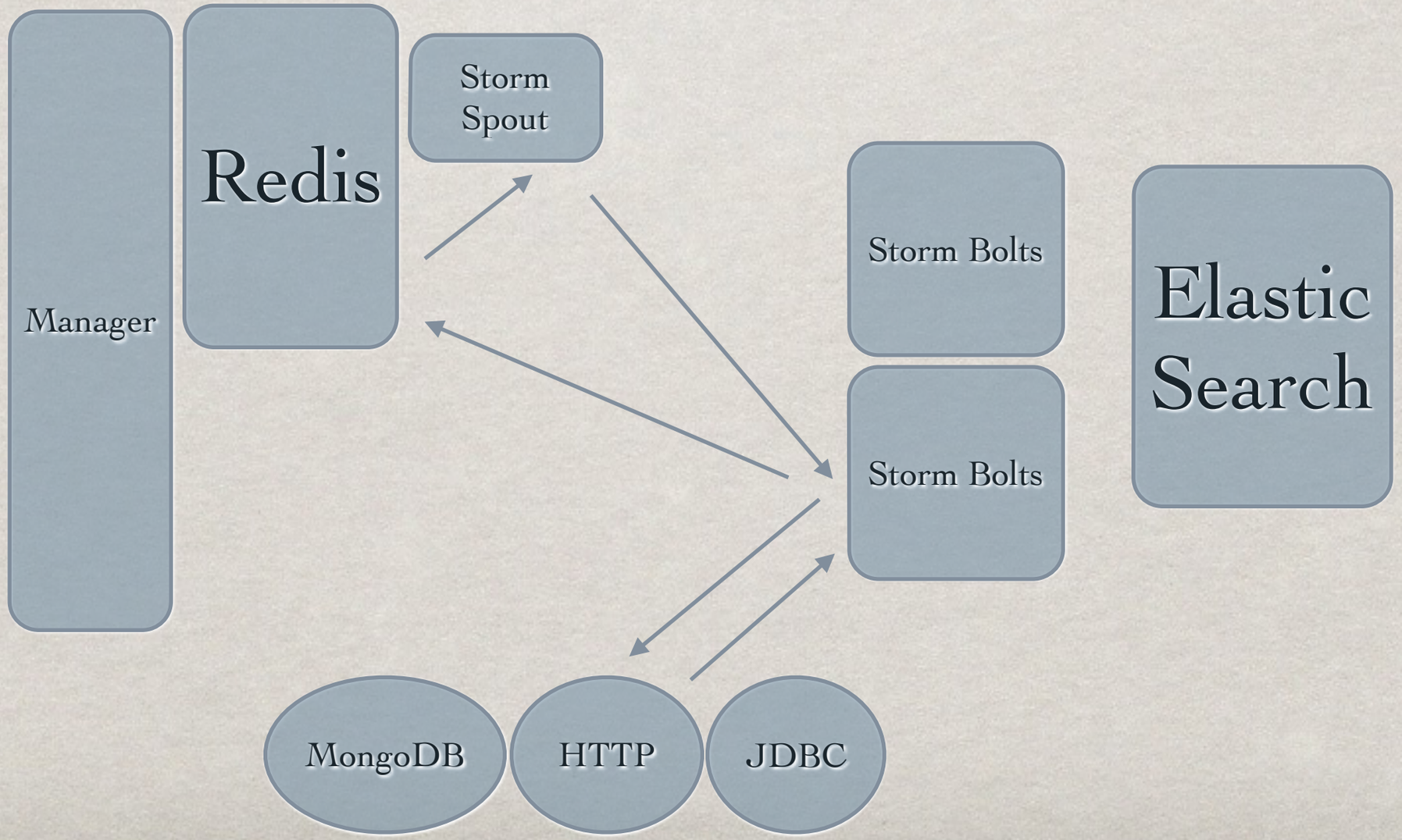
- Leiningen (Clojure Build Tool)
- Repl
- Emacs
- CounterClockwise (Eclipse Plugin)
- La Clojure / Cursive (Idea Plugins)
- JVM tools

Who is Using Clojure

- Nokia Maps
- Twitter
- Factual
- Lumosity
- YieldBot
- USwitch
- Climate
- Akamai
- Citi
- Nugg
- Prismatic
- Netflix

Clojure @Sebula

- Distributed Crawler that supports Http, MongoDB, JDBC



Clojure @Sebula

- Unique Delay queue with Carmine (Redis Library)

```clojure
(def server1-conn {:pool {} :spec {:host     (:redis-uri env/props)
                                   :port     (:redis-port env/props)
                                   :password (:redis-pass env/props)
                                   }})
(defmacro wcar* [& body] `(wcar server1-conn ~@body))

(defn enqueue! [queue message delay]
  (wcar* (car/zadd queue (+ delay (System/currentTimeMillis)) message))
  )

(defn remove-from-queue! [queue message]
  (wcar* (car/zrem queue message))
  )

(defn pop-from-queue [queue]
  (let [min 0
        max (System/currentTimeMillis)
        ]
    (wcar* (car/zrangebyscore queue min max "LIMIT" 0 1))
    )
  )
```

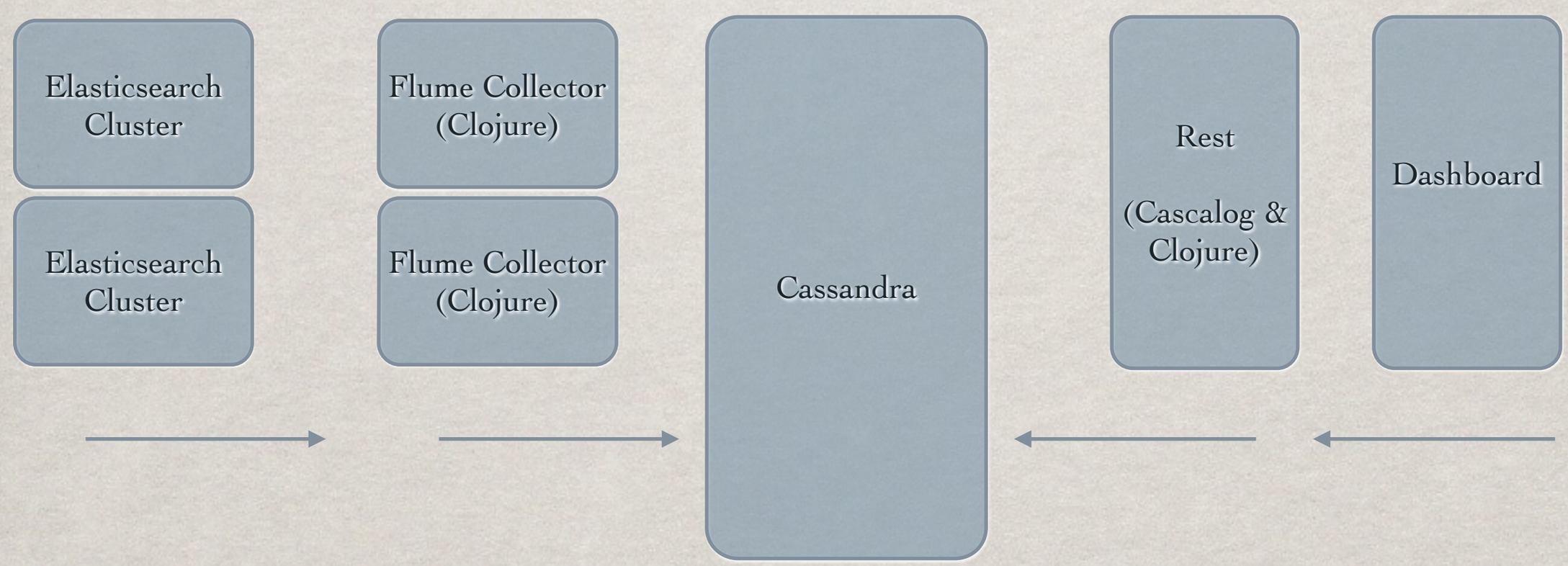
Clojure @Sebula

- Quartize (Quartz Library)

```
(defn init-main-schedule-on-quartz! []
  "frequency is in minutes"
  (let [job (j/build
              (j/of-type PeriodicInjectorJob)
              (j/with-identity "main-job"))
        trigger (t/build
                  (t/with-identity "main-trigger")
                  (t/start-at (-> 1 minutes from-now))
                  (t/with-schedule (schedule
                                     (with-interval-in-minutes (:main-schedule-interval-in-minutes props)
                                     ))))]
    (qs/schedule job trigger))
  )
```

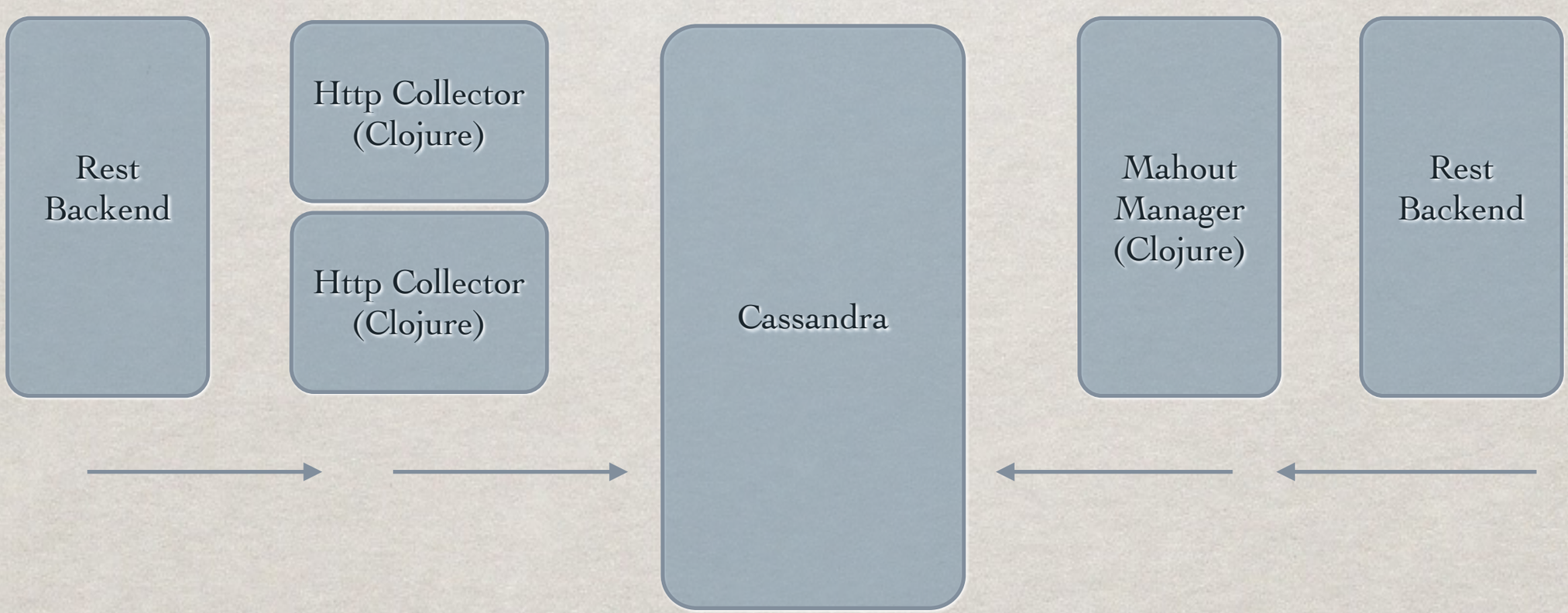
Clojure @Sebula

- Search Analytics
  - Flume, Cassandra, Cascading, Cascalog



Clojure @Sebula

- Rcmmndr: Hadoop & Cassandra Integration



Clojure In Depth

- Collections & Data Structures
- Destructuring
- State & Concurrency & Parallelism
- Macros
- Protocols & Datatypes
- MultiMethods
- Java Interoperability

Talks

- Simple Made Easy by Rich Hickey
- Narcissistic Design - Stuart Halloway

Books

- Land of Lisp
- Clojure Programming
- Programming Clojure

Resources

- Programming Clojure
- ClojureWerkz Clojure Workshop: http://clojureworkshop.com/
- Working with Legacy Code
- clojureist.com
- IstanbulCoders.org
- istanbul-coders+subscribe@googlegroups.com