Clojure en production

Mathieu Corbin, @Exoscale

EXOSCALE

- Cloud provider Européen
- Infrastructure as a service
- 4 zones
- Performances
- Tooling

Clojure @Exoscale

- API Gateway
- Object Store
- Stream processing
- Monitoring (Riemann)
- Frontend
- ...

Clojure ?

- Langage de la famille des LISP sur la JVM/Javascript/CLR
- Programmation fonctionnelle, immutabilité
- Dynamiquement typé
- Programmation concurrente
- Syntaxe élégante et concise

```clojure
(defn fizzbuzz?
  [nb]
  (condp = 0
    (mod nb 15) "FizzBuzz"
    (mod nb 3)  "Fizz"
    (mod nb 5)  "Buzz"
    nb))

(defn fizzbuzz
  [start end]
  (map fizzbuzz? (range start end)))

(fizzbuzz 1 16)
```


```clojure
(defn fizzbuzz?
  [nb]
  (condp = 0
    (mod nb 15) "FizzBuzz"
    (mod nb 3)  "Fizz"
    (mod nb 5)  "Buzz"
    nb))

(defn fizzbuzz
  [start end]
  (map fizzbuzz? (range start end)))

(fizzbuzz 1 16)
```

Syntaxe

Expression

(fonction arg1 arg2 arg3 …)

Exemples

```
user> (+ 1 1)
2
user> (+ 1 1 1 1)
4
user> (* 10 10)
100
user> (- 100 20 30)
50
```

Exemples

```
(* (/ 10 5) (+ 20 20) (* (- 5 2) 3))
```

```
(*     2          40     (*     3     3))
```

```
(*     2          40          9 )
```

720

Définir des variables et des fonctions

```
(def pi 3.14159)

(defn perimeter
  [r]
  (* 2 r pi))

(println (perimeter 10))
"62.8318"
```

Structure de données

- Vector: [1 2 3]
- List: ‘(1 2 3)
- Set: #{1 2 3}
- Map: {**:key1** “foo” **:key2** “bar”}

Immutabilité

```
(def cities {:grenoble 160649
             :nancy    105162})
```

Immutabilité

```
(def cities {:grenoble 160649
             :nancy    105162})

(update cities :grenoble inc)
=> {:grenoble 160650, :nancy 105162}
```

Immutabilité

```
(def cities {:grenoble 160649
             :nancy    105162})

(update cities :grenoble inc)
=> {:grenoble 160650, :nancy 105162}

cities
=> {:grenoble 160649, :nancy 105162}
```

Map/Reduce/Filter

```
(map inc [1 2 3])
=> (2 3 4)

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

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

Threading macros

```
(reduce + (filter even? (map inc (range 6))))
```

Threading macros

```
(reduce + (filter even? (map inc (range 6))))
```

==

```
(->> (range 6)          ; => (0 1 2 3 4 5)
     (map inc)          ; => (1 2 3 4 5 6)
     (filter even?)     ; => (2 4 6)
     (reduce +))        ; => 12
```

Macros ?

- Rappelez vous des listes...

```
'(1 2 3 4 5 6 7 8 9 10)
```

Macros ?

- Rappelez vous des listes...

```
'(1 2 3 4 5 6 7 8 9 10)
```

- Ceci est aussi une liste valide:

```
'(println (assoc {} :grenoble 160649))
```

Macros ?

- Rappelez vous des listes...

```
'(1 2 3 4 5 6 7 8 9 10)
```

- Ceci est aussi une liste valide:

```
'(println (assoc {} :grenoble 160649))
```

Notre code est donc une liste, qui peut être manipulée via des macros

```
user> (infix (1 + 1))
2
```

```
user> (infix (1 + 1))
2
user> (macroexpand '(infix (1 + 1)))
(+ 1 1)
```

```
user> (infix (1 + 1))
2
user> (macroexpand '(infix (1 + 1)))
(+ 1 1)

(defmacro infix
  [expr]
  (list (second expr)
        (first expr)
        (last expr)))
```

expr = ‘(1 + 1)

+

1

1


Multithreaded programming
Theory
Actual

Atom: compare and set

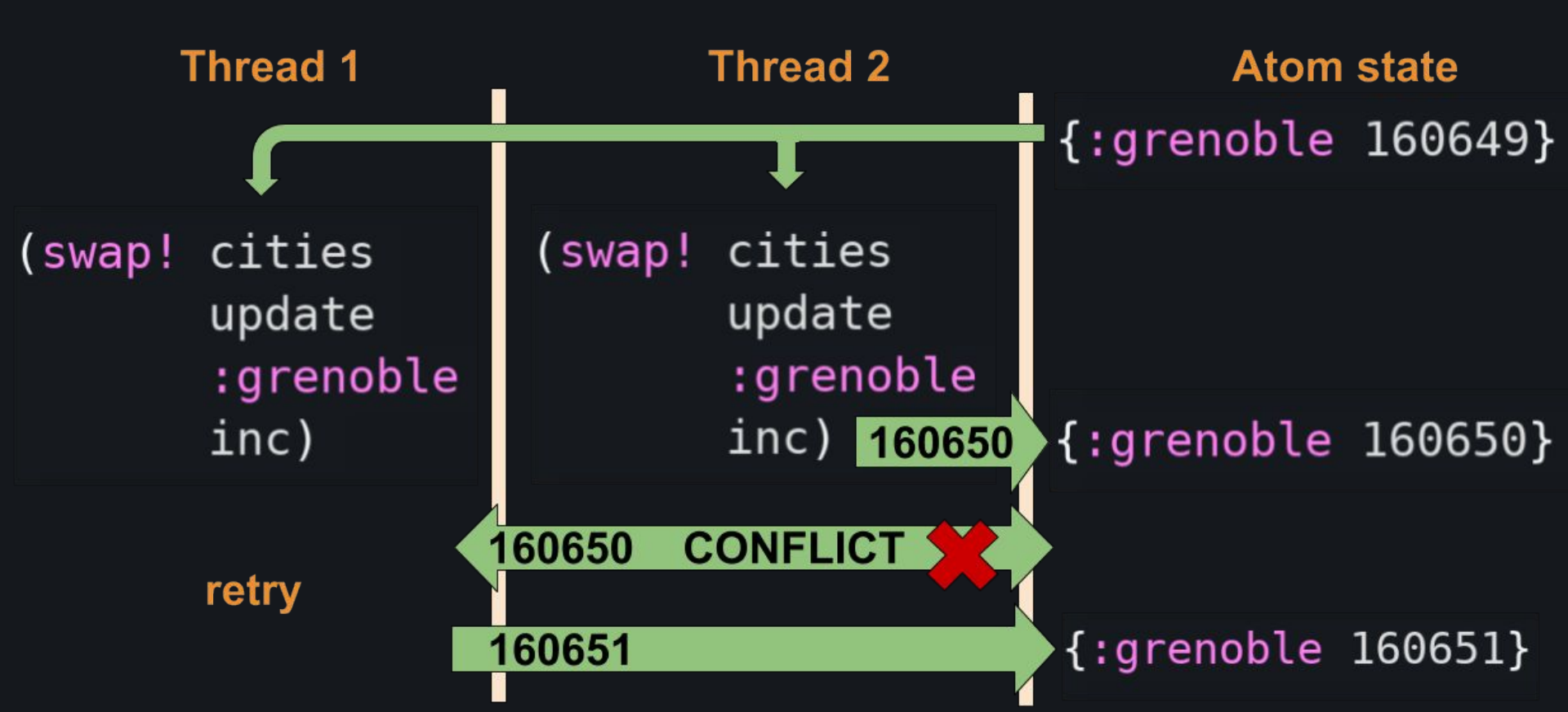
```
(def cities (atom {}))
```

Atom: compare and set

```
(def cities (atom {}))

(swap! cities assoc :grenoble 160649)
=> {:grenoble 160649}
```

Atom: compare and set

```
(def cities (atom {}))

(swap! cities assoc :grenoble 160649)
=> {:grenoble 160649}

@cities
=> {:grenoble 160649}
```


Thread 1
Thread 2
Atom state
{:grenoble 160649}
(swap! cities
update
:grenoble
inc)
(swap! cities
update
:grenoble
inc)
160650
{:grenoble 160650}
160650 CONFLICT
retry
160651
{:grenoble 160651}

Ref: software transactional memory

```
(def account1 (ref 1000))
(def account2 (ref 500))

(dosync
 (alter account1 + 500)
 (alter account2 - 500))
```

Multithreadisme: et bien plus encore...

- Agents: actions asynchrones sur une ressource partagée
- Futures et promises
- Core.async: ~ Golang Goroutines
- Manifold/Aleph: Programmation asynchrone

Un langage commun: la donnée

- Requête HTTP

```
{:request-method :get
 :server-port 3000
 :uri "/foo"
 :server-name "localhost"
 :query-string "talk=clojure"}
```

- Réponse HTTP

```
{:status 200
 :headers {"Content-Type" "text/plain"}
 :body "Hello !"}
```

- HTML

```clojure
(defn footer
  [author]
  [:div {:id "footer"}
   [:p (str "Copyright " author)]])

[:div
 [:h1 "Hello !"]
 [:p "HTML example in Clojure"]
 (footer "mcorbin")]
```

- DSL (Hayt, Cassandra)

```
{:select :users
 :columns [:a :b]
 :where [[= :foo :bar]
         [:in :baz [5 6 7]]]}
```

Un langage commun: la donnée

- Les structures de données servent à modéliser les données de nos programmes
  - Immutables
  - Facilement manipulables/composables
  - Expressives

Spec

```
(def city
  {:name "grenoble"
   :population 160649
   :streets ["Jean Jaurès" "Hoche" "Liberté"]})
```

Spec

```
(s/def ::not-empty-string (s/and string?
                                 not-empty))

(s/def ::name ::not-empty-string)
(s/def ::population pos-int?)
(s/def ::streets (s/coll-of ::not-empty-string
                            :min-count 1))
(s/def ::country ::not-empty-string)

(s/def ::city (s/keys :req-un [::name
                               ::population
                               ::streets]
                      :opt-un [::country]))
```

Spec

- Vérifier la validité de nos structures de données
  - Ne se limite pas qu’aux types

- Spec n’est pas un type system
  - Inutile d’utiliser spec sur chaque variable

- Possible de créer des générateurs à partir de Spec
  - Property Based Testing

Multimethod

```
(def request {:command :ping})

(defmulti compute! :command)

(defmethod compute! :ping
  [request])

(defmethod compute! :list
  [request])

(defmethod compute! :default
  [request])
```

Intéropérabilité Java

```java
HashMap<String, Integer> cities = new HashMap();
cities.put("Grenoble", 160649);
long timestamp = System.currentTimeMillis();
```

```clojure
(def cities (new java.util.HashMap))
(.put cities "Grenoble" 160649)
(def timestamp (System/currentTimeMillis))
```

ClojureScript

- Mêmes avantages que Clojure (langage, immutabilité, REPL…)

- Code/libs partagées entre le front et le back

- Reagent/Reframe: React/Redux+++
  - Immutabilité
  - Concepts simples
  - Performant

Et chez Exoscale, on en pense quoi ?

- On aime
  - Concepts simples, “just works”, productif, REPL
  - Programmation fonctionnelle, “one way flow”
  - Librairies et non Framework
  - L’intéropérabilité/l’écosystème Java

Et chez Exoscale, on en pense quoi ?

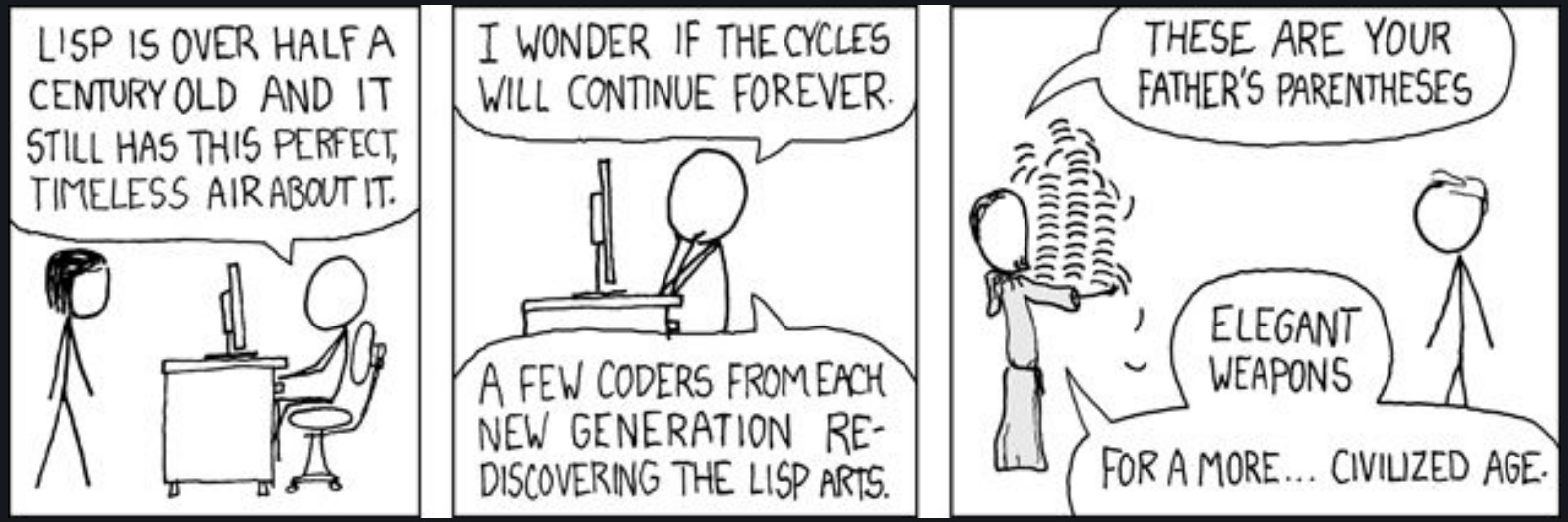
- Le typage dynamique ? Pas un problème
  - ⚠ Nous avons des dev ayant fait du F#/Haskell/Scala...
  - Langages fortement typés: excellent **sur le papier**
    - C’est autre chose en pratique
  - *“Having State and IO monads to get a random number doesn't make my program more robust, readable or debuggable.”*

Et chez Exoscale, on en pense quoi ?

- On aime moins
  - Site officiel contenant peu d’informations
    - Sites communautaires beaucoup plus complets
  - Environnement de développement pas forcément “clé en main” (REPL)
    - Emacs FTW
  - Les messages d’erreurs

Demo/REPL !

Merci !



https://www.braveclojure.com/foreword/
https://aphyr.com/tags/Clojure-from-the-ground-up
https://tour.mcorbin.fr/
https://clojure.org/