Talen

De afgelopen jaren zijn er behoorlijk wat alternatieven op de JVM verschenen. Naast Java kun je in Python (Jython), Ruby (JRuby), Scala, Groovy en nog vele andere talen programmeren. Een van de meer recente ontwikkelingen is Clojure. In een serie van drie artikelen laten we zien hoe praktisch toepasbaar deze taal is.

Clojure: functioneel programmeren

Even wennen, maar helemaal niet eng!

Deze taal wijkt nogal af van de andere genoemde talen, omdat het hier om een puur functionele taal gaat. Dat is in het begin even wennen, zowel qua syntax als qua programmeerparadigma. In deze serie laten we aan de hand van een command-line applicatie die gebruik maakt van de API van de welbekende Last.fm website zien hoe praktisch toepasbaar Clojure is.

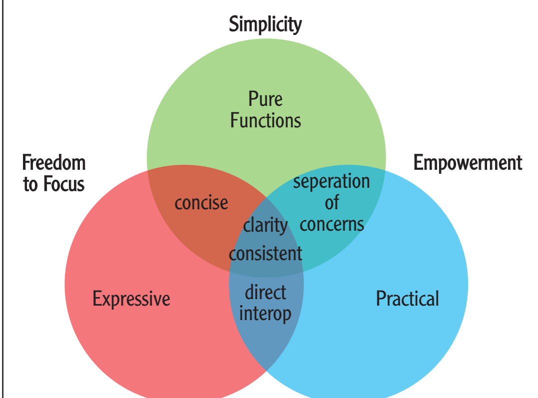
Maurits Rijk
is Agile consultant en metrieken expert bij Xebia.

Sander van den Berg
is senior consultant en architect bij Xebia.

Achtergrond

Clojure is een functionele programmeertaal voor de JVM, ontwikkeld vanaf begin 2007 door Rich Hickey. Hij zocht een Lisp-variant voor een bestaand en veel gebruikt platform, zodat de taal snel praktisch kon worden ingezet. Een tweede wens die Hickey had is dat de taal concurrent programmeren ondersteunde als basisuitgangspunt. Deze taal bestond volgens hem op dat moment nog niet. Hierop besloot Hickey zelf een nieuwe Lisp-achtige taal te ontwikkelen. In mei 2009 leidde dit na tweeënhalf jaar ontwikkeling tot de release van Clojure 1.0.
Clojure richt zich primair op de JVM, alhoewel er ook een CLR-variant bestaat. Het ontwerp van Clojure is geïnspireerd door de concepten uit een groot aantal talen. Uit Lisp heeft Clojure het gedachtegoed van 'code als data' overgenomen. Ruby heeft dynamische typering en korte feedback-cycles geleverd. Uit Haskell komen features zoals STM (Software Transactional Memory) ondersteuning.

In zijn boek 'The Joy of Clojure' legt Michael Fogus de basisontwerpprincipes van Clojure uit aan de hand van de afbeelding uit figuur 1. Clojure heeft drie pijlers als uitgangspunten. Allereerst streeft de taal naar eenvoud (simplicity). Een tweede uitgangspunt is dat de taal geschikt moet zijn om je dagelijks werk uit te voeren (empowerment), je moet er 'echte' applicaties mee kunnen bouwen. Als laatste uitgangspunt is de taal ontworpen om de ontwikkelaar zo min mogelijk in de weg te zitten (freedom to focus).



De basisprincipes van Clojure schematisch samengevat.

Eerste stappen...

We beginnen met het installeren van de Read Evaluate Print Loop (REPL) van Clojure. De REPL stelt ons in staat om snel code uit te proberen en te evalueren.
Stap 1: Download Clojure van http://clojure.org/downloads. In dit artikel zullen we Clojure 1.2 gebruiken.
Stap 2: Unzip clojure-1.2.0.zip.
Stap 3: cd naar de directory clojure-1.2.0 en start het volgende commando:

```
java -jar clojure.jar
```

Je zult nu de volgende prompt zien:

```
Clojure 1.2.0
user=>
```

Dit is de Clojure REPL.

In de eeuwenoude traditie van het leren van nieuwe programmeertalen gaan we "Hello World" implementeren.
Stap 1: Start de REPL.
Stap 2: Voer de volgende regel in:

```
(def hello (fn [target] (println (str "Hello" target))))
```

Stap 3: Voer uit:

```
(hello "Clojure")
```

Dit geeft het volgende resultaat: "Hello Clojure"

Bovenstaande code definieert een functie met de naam hello. Deze functie heeft 1 input parameter: target. Vervolgens print de functie de string-concatenatie van "Hello" en zijn argument target.
De primaire bouwstenen in Clojure zijn functies. Programma's in Clojure zijn een ketting van meerdere functies die aan elkaar zijn geregen. Een functie wordt in Clojure gedefinieerd door:

```
(def functie-naam (fn [parameter-lijst]
(functie-implementatie)))
```

Clojure kent gelukkig ook een syntax short-cut om functiedefinitie eenvoudiger te maken. Met behulp van defn kunnen we de volgende compactere schrijfwijze gebruiken:

```
(defn functie-naam  [parameter-lijst]
(functie-implementatie))
```

Een functie heeft een naam, een documentatiestring en nul of meerdere parameters. In Clojure worden functies uitgevoerd door ze te omringen met ronde haakjes (). Dit is ook de oorzaak van de hoeveelheid haakjes in Lisp-achtige applicaties en vergt soms enige gewenning.
Functies kunnen worden voorzien van een documentatie-string, vergelijkbaar met Java's JavaDoc.

```
(defn square "geeft het kwadraat van zijn input terug"
 [x]
 (* x x))
```

Met de functie doc kan de documentatiestring van de functie worden opgevraagd:
1:

```
(doc doc)
```

geeft:

```
------------------------
clojure.core/doc
([name])
Macro
Prints documentation for a var or special form given
its name
```

2:

```
(doc square)
```

geeft:

```
------------------------
user/square
([x])
```

geeft het kwadraat van zijn input terug
Functies zijn first-class citizens in Clojure. Ze kunnen als argumenten aan andere functies worden meegegeven. Hierdoor kunnen we functie-compositie gebruiken als bouwmiddel voor programma's.

Definieer in de REPL de volgende functie:

```
(defn twice [f a] (f (f a)))
```

Voer uit:

```
(twice square 2)
```

De functie twice neemt als parameters een functie en zijn argument. Vervolgens wordt de functie die als argument is meegegeven uitgevoerd op zichzelf, met als start argument de parameter a. De functie voert uiteindelijk de volgende berekening uit: (* (* 2 2) (* 2 2)), met als eindresultaat dus 16.

Datastructuren

Functionele talen zoals Clojure hebben twee belangrijke bouwblokken: de functies, die eerder in dit artikel besproken zijn, en datastructuren. Clojure kent vrijwel alle datastructuren die ook in Java bekend zijn. Er is echter een groot verschil: in Clojure zijn alle datastructuren immutable. Dit houdt in dat datatypen niet veranderd kunnen worden. Het toevoegen of verwijderen van elementen zorgt ervoor dat er een nieuwe datastructuur wordt gemaakt. We zullen later in deze artikelenreeks zien dat deze eigenschap van groot belang is bij concurrent programming.
Het werkpaard van de Clojure datastructuren zijn de Collections. Clojure kent drie veelgebruikte Collection typen: List, Vector en Map. Al deze drie Collection typen ondersteunen de seq functie, waar we later nog op terug komen.
Een List in Clojure is de meest elementaire datastructuur. De functie list maakt een nieuwe List aan, de functie conj voegt elementen aan het begin van de lijst toe. In het volgende voorbeeld maken we een lijst van 3 elementen en voegen we vervolgens een 4e element toe aan het begin van de lijst.

```
(list 1 2 3)
```

geeft: (1 2 3)

```
(conj (list 1 2 3) 4)
```

geeft: (4 1 2 3)
Lijsten zijn niet homogeen: hiermee bedoelen we dat de inhoud van een List niet perse uit 1 dataty-

Clojure kent vrijwel alle data-structuren die ook in Java bekend zijn.

Er is veel aandacht besteed aan een naadloze integratie met het Java-platform.

pe hoeft te bestaan. Zo kun je probleemloos strings combineren met getallen.

```
(list 1 "a" 2 "b")
```

geeft
(1 "a" 2 "b")

Een List kan dus ook andere lijsten bevatten! Dit mechanisme wordt veel toegepast in functionele talen om complexere datatypen te construeren zoals bijvoorbeeld tree's.
Definieer een tree van 2 diep:

```
(list 1 (list 2 3) (list 4 5))
```

geeft:
(1 (2 3) (4 5))
Net zoals Collections in Java kent ook Clojure een Collection type met random access. In Clojure is dit de Vector. Een Vector is een lijst waarbij de cellen geïndexeerd zijn door aaneengesloten integers. Hierdoor kunnen we de inhoud van een willekeurige cel opvragen met de functie nth. De eerste cel van een Vector heeft nummer 0. De functie conj voegt, anders dan bij een List, eenheden aan de achterkant van de Vector toe. In het volgende voorbeeld lichten we dit toe:

```
(vector 1 2 "a")
```

geeft:
[1 2 "a"]

```
(nth (vector 1 2 "a") 2)
```

geeft
"a"
Als laatste van de drie hoofddatastructuren bekijken we de Map. Een Map is een datastructuur die sleutels op waarden afbeeldt. Tot nu toe hebben we de datastructuren steeds gecreëerd door middel van de constructie functies zoals list en vector. We kunnen echter ook de single-quote (') gebruiken voor de vorm van een datastructuur om een instantie van de datastructuur te creëren. Zo geeft '(1 2) een List van twee elementen en '[1 2] een Vector van twee elementen. In het volgende voorbeeld zullen we op soortgelijke manier een Map creëren en laten we zien hoe je de bij een sleutel horende waarde kan ophalen.

```
(def mymap '{:aap "monkey" :ezel "donkey" :walvis
"whale" :onbekend "platypus"})

(:ezel mymap)
```

geeft:
"donkey"

```
(assoc mymap :onbekend "unkown")
```

geeft:

```
{:aap "monkey", :ezel "donkey", :walvis "whale", :
onbekend "unkown"}
```

Belangrijk is dat in het laatste voorbeeld niet de Map mymap wordt aangepast, maar er een nieuwe Map wordt aangemaakt met daarin het veranderde sleutel/waarde paar. Datastructuren in Clojure zijn immers immutable!
Alle drie de datastructuren die we net beschreven hebben voldoen aan de Seq interface. De Seq interface zorgt ervoor dat we al deze datastructuren als een logische List kunnen behandelen. De functie seq maakt van een collectie een List:

```
(seq '[1 2 3 4])
```

geeft:
([1 2] [3 4])
Elke collectie die voldoet aan de Seq Interface biedt de volgende functies:
(first coll)
Geeft het eerste element in de collectie. Roept seq aan op zijn argument.
(rest coll)
Geeft een sequence van alle elementen na het eerste element. Roept seq aan op zijn argument.
(cons item seq)
Geeft een nieuwe seq terug waar item het eerste element is en seq de rest.

Java-interoperabiliteit

Voor een snelle acceptatie van Clojure is een naadloze integratie met het Java-platform van groot belang. Hier is door de ontwerper dan ook veel aandacht aan besteed. Hij heeft er niet voor gekozen om alle mogelijke Java-libraries van een Clojure-schilletje te voorzien. Dit zou ook onbegonnen werk zijn. Als alternatief biedt hij een syntax die mooi aansluit op de rest van de taal. Daarnaast implementeren Clojure's datastructuren ook de standaard Java interfaces, waardoor aanroepen van Clojure vanuit Java eveneens probleemloos verloopt.

We zullen dit uitleggen aan de hand van het volgende voorbeeld:

```
(def rnd (new java.util.Random))
```

Alternatief:

```
(def rnd (java.util.Random.))
```

Aanroep method:

```
(.nextDouble rnd)

(.nextInt rnd 42)
```

In dit stukje code zien we hoe we de constructor van een Java-klasse aanroepen. We gebruiken hier de fully qualified naam voor de Random klasse. We zien hier ook gelijk dat er twee manieren zijn om Java objecten te instantiëren: via de functie new of via de iets compactere notatie met een puntje achter de naam van de klasse.
De aanroep volgt de gebruikelijke Clojure syntax:

eerst de naam van de functie, daarna de argumenten. Uiteraard bevindt dit alles zich tussen ronde haakjes. Het enige waarin je herkent dat het hier om een Java aanroep gaat, is de punt aan het begin van de functie.
Voor statische methodes en velden geldt een iets andere syntax. Daarvoor wordt de slash (/) gebruikt:

```
(System/getProperty "java.vm.version")

(Math/E)
```

Verder wordt ook de zogenaamde doto expressie veelvuldig toegepast in combinatie met Java objecten. Dit is met name geschikt om een aantal operaties achter elkaar op het zelfde object uit te voeren. Het volgende voorbeeld verduidelijkt dit:

```
(doto (java.util.HashMap.)
   (.put 1 "een")
   (.put 2 "twee"))
```

In dit voorbeeld construeren we een (anoniem) object van het type HashMap, en roepen daarop twee keer de put methode aan.

Installeren bouwomgeving

Tot nu toe hebben we gebruik gemaakt van de REPL als ontwikkelomgeving. Dit werkt prima voor kleine experimenten. Voor de applicatie die we in deze artikelreeks gaan ontwikkelen hebben we behoefte aan een solide bouwomgeving. Hiervoor maken we gebruik van Leiningen.

Stap 1: Download Leiningen van https://github.com/technomancy/leiningen. Voor Windows is dit een lein.bat script, voor OS-X en Linux een shell script. Bij het eerste keer uitvoeren van dit script zonder parameters installeert Leiningen zichzelf. Zorg er voor dat het lein commando op je PATH omgevingsvariabele te vinden is.

Stap 2: lein new lastfm.
Het lein new commando maakt een standaard directory structuur aan. Hierin staat een build-file met de naam project.clj. Hierin staan de standaard afhankelijkheden, maar voor onze applicatie hebben we ook een library nodig met last.fm Java bindings. Deze is opgenomen in de Maven repository van Xebia. In onderstaand codevoorbeeld staat hoe het project.clj er dient uit te zien:

```
(defproject LatestFMCLI "1.0.0-SNAPSHOT"
   :description "Last.fm applicatie voor JavaMagazine"
   :dependencies [[org.clojure/clojure "1.2.0"]
                  [org.clojure/clojure-contrib "1.2.0"]
             [net.roarsoftware/last.fm-bindings "1.0"]]
:repositories {"xebia-mvn" "http://os.xebia.com/maven2"}
   :main LatestFMCLI.core)
```

Voer nadat project.clj is aangepast, het commando lein deps uit. Dit zorgt ervoor dat alle benodigde libraries worden gedownload. In de project folder zijn deze terug te vinden in de lib subfolder.

Last.fm

Last.fm is een social site rond muziek. Gebruikers kunnen de door hen afgespeelde tracks laten "scroblen". Scroblen houdt in dat Last.fm kan bijhouden welke tracks een lid heeft afgespeeld. Aan de hand van deze informatie kan de site allerlei verbanden laten zien: wie houdt er nog meer van jouw muziek? Wat luisteren anderen die jouw smaak hebben? Zijn er nog leuke optredens in de buurt? In deze artikelen reeks zullen we Last.fm bruikbaar gaan maken vanaf de command-line van je favoriete OS. Old School!
We gaan nu een begin maken met onze applicatie. Net als in Java en vele andere talen kent Clojure het concept van een namespace. Het grote verschil met Java is dat namespaces zogenaamde first-class citizens zijn: je kunt ze bijvoorbeeld dynamisch genereren, maar ook naar believen symbolen aan toevoegen of juist verwijderen. Voor onze applicatie is een eenvoudige definitie van een namespace voldoende.

```
(ns LatestFMCLI.core
  (:import (net.roarsoftware.lastfm User))
  (:gen-class))
```

Wat direct opvalt in dit stukje code is dat we tegelijkertijd met het creëren van de namespace ook al direct de User klasse importeren vanuit het net.roarsoftware.lastfm package. Hiermee wordt de mapping van het symbool User naar de gelijknamige Java klasse toegevoegd aan de namespace.
Voordat we verder gaan met het maken van onze eerste last.fm queries, moeten we eerst zorgen voor een API key. Zonder API key kunnen we geen interactie hebben met de API van last.fm.

- Ga naar http://www.last.fm/ en meld je aan als lid.
- Ga vervolgens naar http://www.last.fm/api/account en meld daar je programma aan. Vervolgens zal je twee keys zien: je API key en je secret. Deze keys zul je later in de code nodig hebben, dus schrijf ze ergens op!

Nu we onze username weten en een API key hebben zullen we deze als constanten vastleggen in de code.

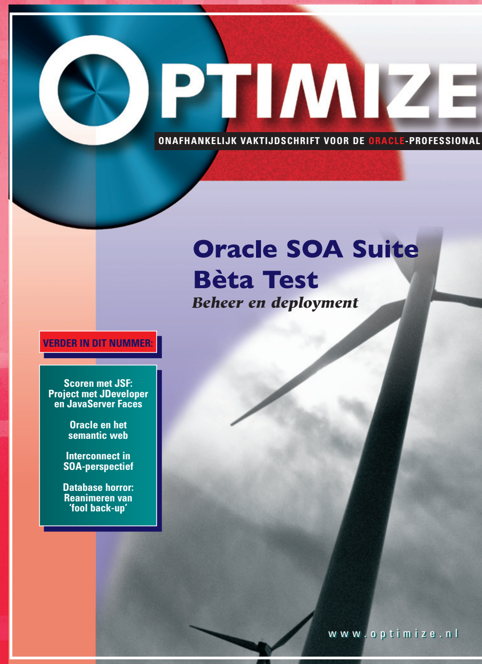
```
(def lastfm-api-key "85e....vul hier je eigen key in")
(def lastfm-user-name "svd... vul hier je eigen
username in")
```

Volgens de API-documentatie van Last.fm (http://www.last.fm/api) en de Java library die we in dit artikel gebruiken (http://www.u-mass.de/lastfm/doc) kunnen we via de User klasse een aantal interessante zaken van gebruikers opvragen.
Deze functionaliteit gebruiken we om de lijst met 50 meest afgespeelde tracks van een gebruiker weer te geven als HTML. De methode getTopTracks van de klasse User geeft een lijst met Tracks terug. Als parameters neemt de methode een user-string en een api-key-string. We zullen met de eerder uitgelegde java-interop de lijst opvragen.

Een solide bouwomgevingen maken we met Leiningen.

Neem nu een abonnement op

Optimize



Hét onafhankelijke vakblad voor de Oracle professional

Databases, application servers, software development en andere Oracle-gerelateerde onderwerpen worden allemaal uitvoerig behandeld in Optimize. Het vakblad is een must voor alle Oracle professionals in Nederland. Optimize gaat diep in op technische en op marktontwikkelingen met betrekking tot Oracle (partners) en concurrenten.

Optimize staat boordevol **praktische en professionele informatie**, geschreven voor en door Oracle professionals. In het blad vindt u het laatste (technische) nieuws uit de markt, een update van productlanceringen, praktijkverhalen, interviews met professionals uit de Oracle wereld en veel technische artikelen van hoog niveau over met name dba-onderwerpen, software development en Oracle Applications. Optimize is ook het publicatieplatform van de Oracle Business Club, de Nederlandse vereniging van Oracle leveranciers. De O.B.C. levert iedere editie een redactionele bijdrage.

Optimize heeft een **uitgebreide website** met onder andere:

- actueel nieuws uit de Oracle-markt,
- een actuele agenda met relevante events,
- het online archief met alle artikelen uit het blad, te downloaden door abonnees,
- een overzicht van relevante vacatures voor Oracle specialisten.

U kunt zich kosteloos abonneren op de **e-mail nieuwsbrief** die eenmaal per 3 weken verschijnt en u volledig op de hoogte houdt van ontwikkelingen op Oracle-gebied. Naast de zes edities van Optimize en de uitgebreide website krijgt u als abonnee korting op de **seminars** die speciaal voor u worden georganiseerd.

Nog geen abonnee?
Meld u online aan op **www.optimize.nl**. Het eerste jaar profiteert u van bijna 50% korting voor nieuwe abonnees.

www.optimize.nl

Array PUBLICATIONS

```
(defn top-tracks
  “Get the top played tracks of a Last.FM user”
  [user-name api-key]
 (User/getTopTracks user-name api-key))
```

Deze functie geeft ons een sequence in de vorm van een List terug met daarin de 50 meest gespeelde Tracks van de gebruiker. Deze lijst kunnen we gebruiken om een mooi geformatteerde lijst te laten zien in onze CLI. In Clojure is het gebruikelijk om met veel kleine functies te werken, dit is onderdeel van wat men in de Clojure community “Idiomatic” noemt. We definiëren een kleine functie die een Track netjes weergeeft.

```
(defn track-to-str
  “Format a Last.FM track as a pretty printed string”
  [track]
  (let [track-name (.getName track)
         artist-name (.getArtist track)]
    (str track-name “ by “ artist-name)))
```

In deze code zien we de concepten terug die eerder in dit artikel zijn uitgelegd. Daarnaast zien we twee nieuwe begrippen: de let-binding en de functie str. Een let-binding zorgt ervoor dat een symbol wordt verbonden met een functie of ander symbol, maar alleen in de scope van de haakjes die om let heen staan. Een let-binding is vergelijkbaar met een lokale variabele, met het verschil dat ook let-bindings unmutable zijn, zoals alles in Clojure. In bovenstaand voorbeeld verbinden we dus de naam van de track met track-name, en de artiest van de track met artist-name. De functie str doet niets meer dan uit zijn argumenten het datatype string bouwen. Het is een typische constructor functie, vergelijkbaar met list en vector.
De lijst met tracks die we van Last.fm krijgen, heeft geen nummering in zich. We willen graag de lijst voorzien van een volgnummer: de meest gespeelde track krijgt nummer 1, de volgende 2 enzovoorts. We zullen hiervoor het derde bouwblok van functionele talen gebruiken: higher order functions. Een higher order function is een (vaak generieke) functie die andere functie(s) als argument krijgt. Een typisch voorbeeld is de functie map, deze functie neemt als argumenten een functie en een lijst. Vervolgens voert map de functie die als argument is gegeven uit op elk element van de lijst. De functie die als argument aan map wordt meegegeven moet logischerwijs 1 argument nemen, het element van de lijst.
In het volgende stuk van onze Last.fm applicatie zullen we een speciale variant van map gebruiken: map-indexed. Deze functie geeft, naast het element uit de lijst, ook de index van het element mee als argument aan de functie die map-indexed heeft meegekregen. We zullen deze index gebruiken om de Tracks te voorzien van een volgnummer.

```
(defn number-a-sequence
  “Appends an increasing number to the elements
of  sequence, starting with 1”
  [seeq]
  (map-indexed #(str (+ 1 %1) “ “ %2) seeq))
```

Deze functie is generiek over alle mogelijke varianten van seq! Wat opvalt in de code is het gebruik van een anonymous function. Het eerste argument van map-indexed is #(str (+ 1 %1) “ “ %2). Deze notatie geeft aan dat dit een functie is zonder naam. Er is dus geen defn of let definitie van de functie. De functie bestaat alleen in deze aanroep. De %1 en %2 notatie staat voor de argumenten van de anonymous function.
Met de drie bouwblokken die we nu hebben gedefinieerd (top-tracks, track-to-str en number-a-sequence) kunnen we de lijst van meest afgespeelde tracks weergeven als een string met genummerde weergaves van tracknaam en artiest. Om onze lijst als HTML weer te geven moeten we deze lijst combineren met een HTML header en footer.

```
(defn to-html
  “converts a list of string items to HTML, by
appending a
   header and footer and adding linebreaks to
each item”
  [str-seeq]
  (let [ header “<html><body>”
          footer “</body></html>”]
    (str header (reduce str (map #(str % “<br/>”)
str-seeq)) footer )))
```

Ook in deze code zien we weer het gebruik van higher-order functions. De functie reduce voert zijn eerste argument, een functie met twee parameters, uit op de eerste twee elementen van de lijst. Vervolgens wordt de functie nogmaals uitgevoerd op element 3 van de lijst en het resultaat van de vorige aanroep van de functie etc. Bovenstaande code converteert elk item naar een string met een “<br/>” op het eind, vervolgens worden al deze strings geconcateneerd. Uiteindelijk worden daar header en footer aan vastgeplakt.
We kunnen nu op eenvoudige wijze onze lijst van 50 meest gespeelde tracks in HTML weergeven:

```
(defn -main []
  (to-html (number-a-sequence (map #(track-to-str %)
 (top-tracks lastfm-user-name
   lastfm-api-key)))))
```

Als we deze code willen uitvoeren kunnen we vanaf de command-line van je OS het commando lein repl uitvoeren. Dit commando zal een REPL starten in de juiste name-space. Je kan nu met (-main) de lijst van 50 meest gescroblde tracks als HTML laten zien.

Conclusie

Onze complete Last.fm applicatie bestaat uit 5 functies van tussen de 2 en 6 regels code. Met minder dan 25 regels Clojure hebben we een eerste versie van ons programma ontwikkeld. Dit is typisch voor Clojure en functioneel programmeren in het algemeen: kleine krachtige functies die eenvoudig gecombineerd kunnen worden. «

De volledige sourcecode is terug te vinden op: https://github.com/xebia/LatestFMCLI .

Typisch voor Clojure: kleine, krachtige functies die je simpel kunt combineren.

In het volgende nummer:
In het volgende nummer breiden we de applicatie uit. We laten zien hoe je niet alleen Last.fm data kunt opvragen, maar ook hoe je bijvoorbeeld tags toevoegt aan je favoriete artiesten. Hierbij introduceren we tegelijkertijd nieuwe Clojure taal constructies als “destructing” maar we beginnen met een solide software engineering practice: het schrijven van unit tests in Clojure.