Leren

In het vorige Java Magazine hebben we een begin gemaakt met het bouwen van een eenvoudige client voor Last.FM met Clojure. Al doende hebben we een aantal interessante facetten laten zien van de taal Clojure en hebben we kennis gemaakt met het Clojure-ecosysteem. Nu gaan we de applicatie uitbreiden met nieuwe functionaliteit.

Clojure: functioneel programmeren (2)

Applicatie uitbreiden met nieuwe functionaliteit

Maurits Rijk
is Agile consultant en metrieken-expert bij Xebia.

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

De eerste stap in onze nieuwe ontwikkeling is het creëren van een vangnet. We gaan immers bestaande code aanpassen. Om te voorkomen dat we met onze wijzigingen fouten introduceren in bestaande code gaan we een set van unit-tests maken, die dienen als regressie-testen.

Unit-testing is in Clojure ingebouwd. Om de functionaliteit die Clojure biedt op het gebied van unit-testing te gebruiken, moeten we de *namespace clojure.test* gebruiken. Dit doen we op de volgende manier:

```
(ns LatestFMCLI.test.core
   (:use LatestFMCLI.core)
   (:use clojure.test))
```

We zullen nu een test ontwikkelen voor de functie *number-a-sequence*. Deze functie heeft als doel om de elementen van een willekeurige *sequence* om te vormen tot strings en te voorzien van een index. Om het geheugen op te frissen: een *sequence* is in Clojure de abstractie van collecties: een logische lijst.

De functie *number-a-sequence* heeft de volgende werking:

```
(number-a-sequence [“test” 3 “foo” “bar”])
```

geeft:

```
(“1 test” “2 3” “3 foo” “4 bar”)
```

De macro *deftest* creëert een unit-test. Deze macro maakt een test aan met een naam. Binnen een test kunnen we een aantal assertie-functies gebruiken om de uitkomst van functies te vergelijken met verwachte waarden. De belangrijkste assertie-functie is de functie *is*. De functie *is* stelt ons in staat om arbitraire verwachtingen vast te leggen over de uitkomst van een functie. Een eenvoudig voorbeeld is:

```
(is (= 5 (+ 2 2)))
```

Waarin we de assertie doen dat 5 gelijk moet zijn aan 2 + 2. Deze assertie is in dit specifieke geval natuurlijk onterecht. De functie is zal dit rapporteren met het volgende bericht:

```
FAIL in clojure.lang.PersistentList$EmptyList@1
(NO_SOURCE_FILE:10)
expected: (= 5 (+ 2 2))
  actual: (not (= 5 4))
false
```

De eenvoudigste test die we kunnen schrijven voor *number-a-sequence* is het scenario dat we hierboven beschreven hebben. De assertie zal zijn of het toepassen van *number-a-sequence* op de sequence [“test” 3 “foo” “bar”] het resultaat (“1 test” “2 3” “3 foo” “4 bar”) oplevert. Dit leidt tot de volgende code:

```
(deftest test-number-a-sequence
         (is (= [“1 test” “2 3” “3 foo” “4 bar”]
          (number-a-sequence [“test” 3 “foo” “bar”]))))
```

We kunnen deze test eenvoudig uitbreiden met nieuwe testgevallen. Zo willen we bijvoorbeeld nagaan of de functie ook op een lege sequence werkt.

```
(deftest test-number-a-sequence
         (is (= [] (number-a-sequence [])))
         (is (= [“1 test” “2 3” “3 foo” “4 bar”]
          (number-a-sequence [“test” 3 “foo” “bar”]))))
```

Door tracks te taggen voegen we informatie toe aan de tracks in LastFM

Een gedeelte van onze code kent afhankelijkheden op functies van externe bibliotheken (de Last.FM API). Deze afhankelijkheden willen we in unit-tests niet. De traditionele manier om dit probleem op te lossen is het toepassen van een Mocking Framework. Clojure kent standaard een eenvoudig mocking framework. Dit framework is onderdeel van *clojure.contrib*.
Als voorbeeld gaan we de functie top-tracks testen. Deze functie kent een afhankelijkheid op de functie *User/getTopTracks* uit de Last.FM API. Deze afhankelijkheid willen we in de unit-test van *top-tracks* mocken.

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

We verwachten dat als de static methode getTopTracks in de klasse User wordt aangeroepen, dit binnen de scope van top-tracks slechts eenmaal voorkomt en dat de argumenten bestaan uit username en api-key. We kunnen dit met de mock library als volgt uitdrukken:

```
(deftest test-top-tracks
  (let [usrname "testuser"
        apikey "testkey"]
    (expect [User/getTopTracks (times once
        (has-args [usrname apikey]))]
      (top-tracks "testuser" "testkey"))))
```

We mocken in bovenstaande code dus de methode getTopTracks en spreken onze verwachtingen uit over de argumenten en het aantal keer aanroepen. Helaas zal bovenstaande code fouten geven bij het uitvoeren. Oorzaak is dat de mocking library gebruik maakt van het herdefiniëren van Clojure var's. *User/getTopTracks* is geen Clojure var, maar een Java static method. We zullen deze functie dus moeten wrappen in een Clojure functie met *defn*.
We hebben nu voldoende bouwstenen om een volwaardige testsuite voor onze Last.FM CLI applicatie te bouwen. De rest van dit artikel breiden we de bestaande code uit met nieuwe functionaliteit.

Clojure types

Tot nu toe hebben we in onze voorbeelden gebruik gemaakt van Java-objecten voor tracks, albums en artiesten. Dit zijn de domeinobjecten die door de Java API van Last.FM worden verwacht. In Clojure wordt gestructureerde data over het algemeen in maps opgeslagen. Maps hebben echter geen type (afgezien van de onderliggende Java-klasse PersistentArrayMap). Indien je dit toch nodig hebt, bijvoorbeeld voor het bouwen van een OO hierarchie, dan biedt Clojure uitkomst in de vorm van zogenaamde Records en Types. We gebruiken hier records om Track gegevens op te slaan:

```
(defrecord LastFMTrack [name artist])
```

In het volgende stukje code laten we zien hoe je met records om kunt gaan:

```
; Creatie van nieuw object
(def track (LastFMTrack. "Canon in D major"
"Johann Pachelbel"))

; Print de naam van de track
(println (:name track))

; Verander artiest
(def new-track (assoc track :artist
"christina aguilera"))
```

We zien aan dit voorbeeld dat een record zich gedraagt als een gewone map in Clojure. Omdat de Clojure compiler een record echter vertaalt naar een Java-klasse is de toegang tot de velden aanzienlijk sneller.
Clojure kent nog twee aanvullende type constructies: *deftype* levert alleen maar een constructor en eventueel door de gebruiker gespecificeerde functionaliteit. En met het gebruik van *reify* kunnen anonieme klassen en instanties daarvan gecreëerd worden. Over het algemeen zal *defrecord* het meest gebruikt worden. In de volgende paragraaf over Protocols voegen we functionaliteit aan het gedefinieerde type toe.

Protocols

Met de huidige code (uit het eerste artikel) kunnen we de meest gespeelde tracks ophalen van Last.FM. Deze tracks krijgen we, als we het in dit artikel geïntroduceerde record-type gebruiken, als een lijst van *LastFMTrack's*. Dit is natuurlijk een leuke eerste stap, maar uiteindelijk willen we ook veranderingen uitvoeren op tracks. Een van deze veranderingen is het taggen van tracks. Door tracks te taggen kunnen we extra informatie aan tracks in Last.FM toevoegen. Zo kunnen we een track taggen met 'concert' als we het live hebben gezien.

De Last.FM API heeft voldoende aan een api-key en username bij het opvragen van gegevens. Bij modificatie zijn er echter strengere eisen. Zo moet er een session-key worden gegenereerd volgens een bepaald vraag-antwoord spel. Om deze stap te vereenvoudigen hebben we in de namespace LatestFMCLI.auth een eenvoudige library toegevoegd om de authenticatie mogelijk te maken. Deze library heeft een API met vier functies:

- check-session-existence: deze functie controleert of er reeds eerder een session-key is aangevraagd en of deze klopt;
- request-user-permission: als het resultaat van check-session-existence negatief is moet er aan de gebruiker toestemming worden gevraagd;
- get-session: deze functie geeft, als bovenstaande stappen zijn uitgevoerd, de sessie terug om wijzigingen te kunnen doen op Last.FM. (get-session) moet als argument worden meegegeven aan elke wijzigingsfunctie van Last.FM;

- lastfm-api-key: een helper functie die de api-key en secret uit een tekst-file leest (keys.txt) met het volgende formaat: *{:lastfm-api-key: "your key" :lastfm-secret "your secret"}*. Dit formaat zal geen verassing zijn na het vorige artikel.

We willen tagging natuurlijk generiek opbouwen. Onder generiek verstaan we in de context van Last.FM dat we meerdere soorten data (Tracks, Albums, Artists) willen taggen. Daarnaast willen we in de toekomst waarschijnlijk meer dan alleen tags toevoegen aan de Tracks, Albums en Artists. Wellicht willen we Tracks en Albums weergeven in HTML. We willen dus zowel nieuwe typen toevoegen als nieuwe functionaliteit. Dit probleem komt in de software engineering vaker voor, zo vaak dat het een naam heeft gekregen: The Expression Problem.
Het Expression Problem is als volgt gedefinieerd: definieer een datatype, waarbij we nieuwe varianten van het datatype willen toevoegen en nieuwe functies over deze datatypen willen definiëren, zonder dat we bestaande code moeten hercompileren. In bovenstaande voorbeeld willen we bijvoorbeeld onze tag functie met zowel Track als Album laten werken en willen we bijvoorbeeld het datatype Artist introduceren waar we de tag functie op willen toepassen.

Gelukkig heeft Clojure een aantal mechanismen om dit probleem op te lossen. Een mechanisme hebben we hierboven besproken: defrecord. Met defrecord kunnen we nieuwe datatypen toevoegen. Een ander mechanisme is protocollen. Een protocol in Clojure is een abstractie van een functie. Een protocol is vergelijkbaar met een Java interface. Het beschrijft de namen van functies en hun parameters, maar heeft zelf (in principe) geen implementatie.

We kunnen bijvoorbeeld onze functie tag beschrijven met een protocol.

```
(defprotocol Taggable
   "Defines that a datatype is \"taggable\""
(tag [this taglist] "tag an item with a list of tags"))
```

Bovenstaand protocol definieert het Clojure *protocol* Taggable met de functie *tag*. Deze protocol definitie geeft dus geen invulling aan de functie tag. Met behulp van dit protocol kunnen we nu een implementatie koppelen aan een datastructuur. Als voorbeeld nemen we de eerder in dit artikel gedefinieerde datastructuur LastFMTrack. We kunnen in Clojure een record associëren met een protocol door deze in de defrecord definitie op te nemen. Zie het voorbeeld hieronder:

```
(defrecord LastFMTrack [name artist])

;met protocol:
(defrecord LastFMTrack [name artist]
   Taggable
   (tag [this taglist]
      (Track/addTags (:artist this) (:name this)
         (vector-to-string taglist) (get-session)))
```

We hebben nu de implementatie van het protocol Taggable geassocieerd met het record LastFMTrack. Met het protocol mechanisme kunnen we heel eenvoudig nieuwe implementaties van bestaande functies definiëren over nieuwe datatypen.

In bovenstaand voorbeeld hebben we ons niet gehouden aan het probleem zoals beschreven in het Expression Problem: we hebben immers de bestaande definitie van LastFMTrack aangepast! Clojure biedt de optie om protocol implementaties te definiëren over bestaande datatypen zonder deze aan te passen. We doen dit met de *extend-protocol* constructie. *Extend-protocol* associeert een protocol implementatie met een datatype. We hadden bovenstaande implementatie van *tag* voor *LastFMTrack* ook kunnen definiëren met *extend-protocol*. Zie onderstaande voorbeeld code:

```
(extend-protocol Taggable
   LastFMTrack
   (tag [this taglist]
      (Track/addTags (:artist this) (:name this)
      (vector-to-string taglist) (get-session))))
```

Multimethods

Nu we het protocol om een item te taggen hebben gedefinieerd, willen we dit protocol implementeren voor de verschillende items die de API van Last.FM biedt. Last.FM kent Tracks, Albums en Artists. We kunnen natuurlijk voor elk van deze typen een aparte protocol implementatie maken. Er zijn echter andere manieren om dit op te lossen. Clojure ondersteunt *multi-methods*. Multi-methods kunnen aan de hand van een zogenaamde dispatch functie een specifieke implementatie van een functie aanroepen. Dit mechanisme kennen we ook in een beperkte vorm in Java: type gebaseerd polymorfisme, ofwel inheritance. Bij inheritance is de dispatch functie simpelweg het type van het object. Als voorbeeld laten we zien hoe je type-based polymorfisme in Java en Clojure implementeert.

```
// Plain old Java code

public class Item {
   public void tag(List<Tag> taglist) {
      ...
   }
}

public class Track extends Item {
   @Override
   public void tag(List<Tag> taglist) {
      ...
   }
}

public class Album extends Item {
   @Override
   public void tag(List<Tag> taglist) {
      …
   }
}
```

Clojure ondersteunt multi-methods

```
public static void main(String[] args) {
    Album a = new Album();
    //a.tag zal nu de methode tag van Album aanroepen,
niet van Item.
    a.tag(taglist);
}
```

Bovenstaande code is standaard Java. We zullen nu dezelfde code implementeren met behulp van Clojure's multimethods. *defmulti* beschrijft de naam van de multi-methode en de dispatch-functie. *defmethod* beschrijft vervolgens de implementatie per geval van de dispatch-functie. De dispatch-functie in onderstaand voorbeeld is de functie class. Deze functie geeft de fully-qualified classname terug van een var. Onderstaande *defmethod's* hebben dus als dispatch-waarde een fully-qualified classname, zoals bijvoorbeeld net.roarsoftware.lastfm.Track.

```
;definieer de multimethode add-tag en de dispatch
functie,
;dispatching is hier op type.
(defmulti add-tag (fn [n & more] (class n)))

(defmethod add-tag net.roarsoftware.lastfm.Album [alb
tl]
  (Album/addTags (.getName alb) (.getArtist alb)
(vector-to-string tl) (get-session)))

(defmethod add-tag net.roarsoftware.lastfm.Track [track
tl]
  (Track/addTags (.getArtist track) (.getName track)
(vector-to-string tl) (get-session)))

(defn- main []
   (add-tag album ["testtag"]))
```

Multimethoden kunnen dus gebruikt worden om inheritance na te bootsen. Ze zijn echter veel krachtiger dan dat. Dispatching kan bij multimethoden op elke denkbare functie.

Multi methoden kunnen ook gebruikt worden om inheritance na te bootsen

Met behulp van de bovenstaande multimethode kunnen we ons oorspronkelijke Taggable protocol eenvoudig implementeren.

```
(extend Taggable
   LastFMTrack
   (tag [this taglist]
      (add-tag this taglist)))
```

Destructuring

We zijn in het eerste artikel ingegaan op een aantal data structuren die door Clojure ondersteund worden, bijvoorbeeld vectoren en maps. Zo hebben we bijvoorbeeld de volgende map gedefinieerd:

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

Stel dat we nu zowel de waarde van :aap als :ezel aan een nieuw symbool willen toekennen. Met bijvoorbeeld een Java achtergrond zou je dan al snel voor de volgende oplossing kiezen:

```
(let [aap  (:aap mymap)
      ezel (:ezel mymap)]
  (println aap ezel))
```

Op zich werkt dit prima, maar Clojure kent een handigere manier om data structuren uit elkaar te halen. Dit concept heet binnen Clojure destructuring. Dit ziet er als volgt uit:

```
(let [{aap :aap, ezel :ezel} mymap]
  (println aap ezel))
```

In bovenstaand stukje code zien we hoe je in één keer de gewenste velden uit de map kunt halen en tegelijkertijd ook kunt toekennen aan één of meerdere symbolen.
Als we de namen van de symbolen gelijk houden aan de namen van de sleutels in onze map, dan kunnen we dit nog verder vereenvoudigen door middel van de *:keys feature*:

```
(let [{:keys [aap ezel]} mymap]
  (println aap ezel))
```

In dit voorbeeld hebben we destructuring gebruikt om snel de waarden van een map te binden aan locale symbolen. Hetzelfde principe kan ook toegepast worden op vectoren. Stel we hebben de volgende vector om het begin van een leesplankje te representeren:

```
(def leesplankje ["Aap" "Noot" "Mies" "Wim"
  "Zus" "Jet"])
```

We kunnen nu door middel van destructuring in één keer de eerste drie waarden van de vector toekennen aan drie symbolen:

```
(let [[aap noot mies] leesplankje]
  (println aap noot mies))
```

Daarnaast kent Clojure nog een aantal handige constructies voor destructuring van vectoren, waarvan we er hier twee laten zien in het volgende voorbeeld:

```
(let [[aap noot mies & more :as all] leesplankje]
  (println more)
  (println all))
```

We zien hier hoe we achter de ampersand (&) het symbool *more* definiëren. Deze wordt gebonden aan alle resterende waarden van de vector ("Wim", "Zus", "Jet"). Daarnaast gebruiken we ook het *:as* keywoord. Het symbool daarachter (all) wordt gebonden aan de complete inhoud van de vector.
In de voorgaande voorbeelden hebben we laten hoe je destructuring kunt toepassen op datastructuren. Op exact dezelfde manier kunnen we dit ook toepassen op functie parameters. We gaan dit gebruiken om onze eerder gedefinieerde track-to-str functie te herschrijven. Eerst nog even de oude versie die gebruik maakt van Java interoperability:

```
(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)))
```

En hier de nieuwe versie waarbij destructuring mogelijk is omdat we track als record hebben gedefinieerd:

```
(defn track-to-str
  "Format a Last.FM track as a pretty printed string"
  [{:keys [name artist]}]
  (str name " by " artist))
```

Geavanceerd itereren

In ons eerste artikel zijn we al ingegaan op het itereren door een sequence. Clojure kent niet de gebruikelijke *for* of *while* lussen zoals we die in de meeste procedurele of object-georiënteerde talen vinden. In plaats daarvan kun je recursie gebruiken. Daar gaan we in dit artikel niet verder op in omdat er vaak een beter alternatief is. Zo hebben we al eerder de zeer krachtige map/reduce constructie gezien.

Soms wil je echter over meerdere sequences tegelijkertijd itereren. Programmeertalen als Java vangen dit op door een aantal lussen in elkaar te draaien. Gebruikelijke constructies zijn:

```
// Plain old Java code

for (int x = 0; x < 3; x++)
{
   for (int y = 0; y < 3; y++)
   {
      if (x != y)
      {
          // doe iets met x en y
      }
   }
}
```

In Clojure kan dit met de for macro als volgt elegant worden opgelost:

```
(for [x (range 3) y (range 3) :when (not= x y)] [x y])

; resulteert in ([0 1] [0 2] [1 0] [1 2] [2 0] [2 1])
```

Met behulp van *:when* geven we hier de conditie aan waaraan de (x, y) paren moeten voldoen. Het bijkomende voordeel hier is dat *for* een lazy sequence maakt. Dat betekent dat alleen dat deel van de sequence wordt geconstrueerd dat ook daadwerkelijk wordt gebruikt. Dit is duidelijk niet vergelijkbaar met *for* lussen in Java.
Voor onze Last.FM applicatie willen we voor onze meest gespeelde tracks de meest populaire tags opvragen. We definiëren eerst een functie *get-top-tags* die voor een gegeven track een lijst van meest voorkomende tags terug geeft. Hier wordt ook weer gebruik gemaakt van parameter destructuring om de naam en artiest van een track op te halen:

```
(defn get-top-tags
  "Creëer sequence van toptags"
  [{:keys [name artist]} api-key]
  (Track/getTopTags artist name api-key))
```

Vervolgens definiëren we *get-top-tracks-tags* om met behulp van de *for* macro een lazy sequence te genereren van (track, tag) tuples:

```
(defn get-top-tracks-tags
  "Creëer sequence van [track tag] tuples"
  [user-name api-key]
  (for [track (toptracks user-name api-key)
        tag (get-top-tags track api-key)]
          [track tag]))
```

Het symbool *track* wordt hier gebonden aan de elementen uit de sequence van top tracks. Voor ieder van deze tracks wordt vervolgens het symbool *tag* gebonden aan de elementen uit de sequence van meest populaire tags. Hieronder een voorbeeld hoe je dat zou kunnen gebruiken:

```
; Haal de lijst op
(def my-list (get-top-tracks-tags "your_user_name"
"your_api_key"))

; … en daaruit het eerste tupel
; omdat de lijst 'lazy' is, wordt de rest
   niet berekend!
(def first-top-track-tag (first my-list))

; gebruik een anonieme functie en parameter
; destructing om tupel uit elkaar te halen.
((fn [[track tag]] (str (.getName track) ": "
   (.getName tag)))
   first-top-track-tag)

; output in REPL: "Dog Days Are Over: female vocalists"
```

We hebben al aangegeven dat de *for* macro een lazy sequence oplevert. Soms is dat niet gewenst en wil je bijvoorbeeld direct een functie met zogenaamde *side effects* zoals weergave op het scherm, direct uitvoeren. Hiervoor heeft Clojure de *doseq* macro. De syntax is verder gelijk aan de *for* macro:

```
(doseq [x (range 3) y (range 3) :when (not= x y)]
   (println [x y]))

; resulteert in
; [0 1]
; [0 2]
; …
; [2 1]
```

Conclusie

In dit artikel hebben we laten zien hoe Clojure geavanceerde taalconstructies als destructuring biedt, die tot compacte en elegante code leiden. Daarnaast zijn we uitgebreid ingegaan datatypen, protocollen en multimethoden. Deze laten een zeer flexibele manier van polymorfisme toe. Omdat we niet alle details kunnen weergeven hebben we aan het einde van dit artikel een aantal referenties toegevoegd.

De volledige sourcecode is terug te vinden op: https://github.com/xebia/LatestFMCLI/tree/article2 .

Vooruitblik

We hebben in dit tweede over Clojure een groot aantal concepten voorbij laten komen. In het derde (en laatste) artikel uit deze serie zullen we de Last.fm applicatie nog verder uitbouwen aan de hand van geavanceerde taalconstructies als macro's. Hiermee is het mogelijk om de hoeveelheid 'boiler-plate' code nog verder te reduceren en het stelt de ontwikkelaar in staat om zijn eigen DSL (Domain Specific Language) te definieren.

Gebruik van Clojure leidt tot compacte en elegante code

Referenties

- "Programming Clojure", Stuart Halloway, Pragmatic Programmers, 2009
- "The Joy of Clojure", Michael Fogus and Chris Houser, Manning, 2010
- "Practical Clojure", Luke Vanderhart and Stuart Sierra, Apress, 2010
- "Clojure in Action", Amit Rathore, Manning, 2011