# Clojure: een beter alternatief voor Java

Dit is Rich Hickey, bedenker van programmeertalen en databases:



Op de Java One van 2014 mocht Rich Hickey een [praatje](https://www.youtube.com/watch?v=VSdnJDO-xdg) houden. Hij begon met een [quote](http://thenewstack.io/the-new-stack-makers-adrian-cockcroft-on-sun-netflix-clojure-go-docker-and-more/) over een programmeertaal, en liet het publiek raden welke:

"A lot of the best programmers and the most productive programmers I know are writing everything in \*\*\*\*\*\*\* and swearing by it, and then just producing ridiculously sophisticated things in a very short time. And that programmer productivity matters."

-- Adrian Cockroft, (voormalig) Netflix

Adrian Cockroft had het niet over Java, maar over Clojure, een functionele programmeertaal voor de JVM, die Rich Hickey zelf in 2007 introduceerde. De pitch van Rich was dat je met Clojure programma’s kunt schrijven die beter en flexibeler zijn, en dat je er bovendien ook productiever mee bent dan met Java.

Het zal zeker een reden zijn waarom Clojure her en der is omarmd als een beter alternatief voor Java. Naast het eerder genoemde Netflix geldt dat bijvoorbeeld ook voor de programmeurs bij [WalmartLabs](http://blog.cognitect.com/blog/2015/6/30/walmart-runs-clojure-at-scale):

"Clojure shrinks our code base to about one-fifth the size it would be if we had written in Java"

-- Anthony Marcar, architect, WalmartLabs

De boodschap van dit artikel is simpel: wil jij ook tot wel duizenden regels code per jaar besparen, stap dan nu over op Clojure. Speciaal voor jou en de andere lezers van Java Magazine bieden wij daarom hier de Clojure overstapservice.

Allereert de drie voordelen van overstappen op een rijtje:

## 1. Clojure is simpel

Clojure is niet object-georiënteerd, maar data-georiënteerd. Data wordt gerepresenteerd met *immutable hashmaps* in plaats van *classes* zoals in Java. Deze datastructuren zijn eenvoudig te manipuleren met functies. De filosofie van Clojure sluit daarom aan bij een uitspraak van Alan J. Perlis:

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

Als je steeds dezelfde datastructuren hergebruikt, hoef je niet telkens het wiel opnieuw uit te vinden om je data in de gewenste vorm – dus classes - te krijgen. Classes vormen bovendien een taal op zichzelf, je moet eerst de methods leren kennen voordat je de datastructuren erachter kunt gebruiken. En omdat de maps *immutable* zijn, hoef je ook geen rekening te houden met eventuele side-effects bij het aanroepen van methods.

Een klein voorbeeld. In de [Ring](https://github.com/ring-clojure) library, waarmee je webapplicaties kunt schrijven, is een *handler* niets anders dan een functie die een hashmap verwacht (de request) en een hashmap teruggeeft (de response):

(defn what-is-my-ip [request]

{:status 200

:headers {"Content-Type" "text/plain"}

:body (:remote-addr request)})

De namespace waarin we deze handler definiëren heeft geen afhankelijkheid van de Ring-library. Het enige wat we hoeven te weten is hoe een request er in Ring uitziet, dus welke keys we in de hashmap kunnen verwachten en welke keys we terug moeten geven om een geldige response te kunnen vormen. Alles wat tussen request en response gebeurt is niets anders dan rekenen met normale hashmaps. Dat is toch een stuk eenvoudiger dan het werken met HttpServletRequest en HttpServletResponse.

### Minder regels code

Met Clojure heb je geen last van de overhead die object-georiënteerde talen met zich meebrengen. Overhead die bestaat uit het schrijven van interface- en implementatiecode om je datastructuren te ‘beschermen’. Je hebt dus minder Clojure-regels dan Java-regels nodig om hetzelfde te bereiken. De voordelen van minder regels code zijn evident: kleinere programma’s hebben minder bugs, zijn beter te begrijpen en zijn door de eenvoud ook robuuster.

### 2. Clojure is overal

Bij het werken in het Clojure-ecosysteem kun je de kennis van dezelfde programmeertaal hergebruiken in meerdere omgevingen. Voor server side ontwikkeling maak je gebruik van Clojure op de JVM (of in .NET via ClojureCLR).

Voor front-end development kun je gebruik maken van ClojureScript, wat compileert naar JavaScript. De ClojureScript-compiler kan naast JavaScript voor de browser ook JavaScript genereren voor NodeJS, wat je dan ook weer voor server side toepassingen kunt gebruiken.

De implementaties van Clojure op diverse omgevingen zijn zoveel mogelijk hetzelfde gehouden en wijken alleen af als dat afgedwongen wordt door beperkingen van de omgeving. Een voorbeeld van zo’n beperking is de afwezigheid van multithreading in een browser.

Veel Clojure libraries bieden ondersteuning voor zowel Clojure als ClojureScript.

Dus ook de kennis die je hebt van deze libraries kun je hergebruiken over omgevingen heen.

### Reader conditionals

In de laatste stabiele versie van Clojure (1.7, uitgebracht op 30 juni 2015), werden reader conditionals toegevoegd. Dit geeft de mogelijkheid om binnen één bestand meerdere platformen te kunnen bedienen. Bij een full stack Clojure applicatie kun je hierdoor een gedeelte van je code hergebruiken op client en server.

Een voorbeeld. De volgende code definieert een variabele met de waarde NaN. Op het Clojure/JVM-platform is dat Double/NaN en in JavaScript NaN uit de globale namespace.

(def not-a-number

#?(:clj Double/NaN

:cljs js/NaN

:default nil))

Dus behalve [Universal JavaScript](https://medium.com/@mjackson/universal-javascript-4761051b7ae9) kunnen we evengoed spreken over Universal Clojure.

### Datomic

Clojure is overal, zelfs in de database. Meer precies: de NoSQL database Datomic, ook een vondst van Rich Hickey.

Datomic is net als Clojure gebaseerd op het concept van *immutability*: data wordt niet overschreven zoals dat gebeurt bij traditionele databases (‘update-in-place’), maar wordt uitsluitend toegevoegd aan vroegere waarden (‘append-only’). En omdat bij elke schrijfactie de tijd wordt geregistreerd, kun je met Datomic ook tijdreizen – dus teruggaan naar de toestand van een willekeurig moment in het verleden.

Data wordt in Datomic opgeslagen in de vorm van de bekende datastructuren die we van Clojure gewend zijn (maps, sets, lists, vectors en dergelijke), en kunnen als zodanig direct in Clojure programma’s worden geladen. Omgekeerd is het ook mogelijk om Clojure functies op te slaan in de database, waardoor je als het ware stored procedures krijgt, die je kunt aanroepen in je gedistribueerde database.

## 3. Clojure is full stack

Hoe je het ook wendt of keert, Java is niet full stack, het is een taal voor de back-end. Wel is geprobeerd om een plekje te veroveren aan de client side, maar wie ontwikkelt er tegenwoordig nog applicaties in AWT, Swing of JavaFX? Evenzo is Clojure begonnen als alternatief voor back-end, maar met de komst van ClojureScript, en een hoop front-end libraries en tools kun je nu recht spreken van een full stack programmeertaal. Bovendien biedt de stack interessante elementen waarmee full stack ontwikkeling aanzienlijk kan worden verbeterd.

### React

Met de komst van JavaScript library React is het bouwen van een SPA (single page app) in ClojureScript een fluitje van een cent. Reagent is een ClojureScript library die het schrijven van React-componenten vergemakkelijkt. Door middel van Hiccup-notatie, waarmee je uit een geneste Clojure-datastructuur HTML kan beschrijven en reactieve atoms, kun je met relatief weinig code een React-component schrijven.

Dit is een voorbeeld van een component wat het aantal clicks telt op een knop en dit toont in een div:



(def count-state (atom 10))

(defn counter []

[:div

@count-state

[:button {:on-click #(swap! count-state inc)}

"x"]])

(reagent/render-component [counter]

(js/document.getElementById "app"))

Met Om, een andere ClojureScript wrapper rondom React, zijn zelfs betere performance benchmarks gehaald dan React zelf. Dit heeft te maken met de efficiënte manier waarop verschillen in *state* kan worden bepaald in ClojureScript (maar ook in Clojure zelf).

### Geen callback hell

Een van de problemen waar JavaScript-developers mee worstelen is het fenomeen *callback hell*. Een browser heeft maar één thread. Daarom moeten we werken met geneste callbacks. Als we callbacks diep nesten, wordt de code steeds slechter leesbaar.

Clojure geeft hiervoor een oplossing in de vorm van de core.async library, die het werken met asynchrone code vereenvoudigt. Uiteraard werkt ook deze library op zowel server als client. De bouwblokken van core.async zijn channels, buffers en go-blocks. Go-blocks zijn source-transformaties die de illusie geven van synchrone code, maar uiteindelijk geneste callbacks opleveren.

Een voorbeeld. De volgende code haalt eerst het e-mailadres op van een gebruiker via een REST-API call. Hiervoor gebruiken we de library [http-cljs](https://github.com/r0man/cljs-http), die het maken van Ajax calls combineert met core.async. Als resultaat van een http-call krijgen we een een channel terug, waaruit we het resultaat uit kunnen lezen met de <! (take) operatie. Vervolgens gebruiken we het e-mailadres in een tweede call om de bestellingen van deze gebruiker op te halen. Als uiteindelijk resultaat geven we het aantal bestellingen terug.

(go (let [email (:body

(<! (http/get

(str "/api/users/"

"123"

"/email"))))

orders (:body

(<! (http/get

(str

"/api/orders-by-email/"

email))))]

(count orders)))

### Interactief programmeren

Clojure is een interactieve programmeertaal. Via een REPL (read-eval-print loop) krijg je direct feedback. Tijdens het ontwikkelen kun je functies herdefiniëren en opnieuw testen terwijl de applicatie draait, zonder een nieuwe compileercyclus te starten.

Ook ClojureScript kent het concept van een REPL waarmee je code in je browser kan testen en herdefiniëren. Een tool als [Figwheel](https://github.com/bhauman/lein-figwheel) maakt het tevens mogelijk om gewijzigde ClojureScript meteen in je browser te zien, zonder dat je de applicatie in je browser moet verversen.