Vavr

Zet Java op z'n kop!

Met de komst van Lambda's in Java 8 zette de Java-wereld een stapje richting het functioneel programmeren. De Vavr library (voorheen Javaslang) doet daar nog een schepje bovenop met rijker gedrag voor *functions*, zoals *composition* en *currying, immutable collections, for expressions* en rijke *value wrappers*. In dit artikel lees je hoe je functioneel kunt programmeren in je huidige Java project. Ook als je nog geen ervaring hebt met functioneel programmeren, dan kun je toch snel profiteren van de praktische API, die Vavr biedt.

Hinse ter Schuur
is Senior Software Engineer bij SDB Java.

De basis van Functioneel Programmeren is het evalueren van pure (mathematische) functies. Functies waarbij de output uitsluitend afhankelijk is van de argumenten, die in de functie worden gestopt. Deze zijn dus niet afhankelijk van enige vorm van state. Dit betekent ook dat een functie voor een gegeven input altijd dezelfde output heeft.

Programmeren binnen het Functioneel Programmeren paradigma betekent het gebruik maken van expressies; iets dat een waarde oplevert. Dit in tegenstelling tot statements, want dat is iets dat een actie uitvoert zonder een waarde terug te geven. Statements hebben een side effect: ze doen of wijzigen iets. Side effects, in het bijzonder state changes, hebben als nadeel dat ze lastig zijn in een multithreaded omgeving.

Een krachtige constructie in het Functioneel Programmeren is het kunnen werken met *higher order functions*. Dit zijn functions, die zelf een functie teruggeven of een functie als argument accepteren. De Lambda's in Java 8 maken het mogelijk om in Java met higher order functions te werken.

Vavr

Starten met Vavr is heel gemakkelijk. Het enige wat nodig is voor de basis van Vavr is het toevoegen van de dependency aan je project (zie **Listing 1**).

Syntactic sugar

Een aardige vorm van *syntactic sugar* biedt Vavr in de vorm van een aantal static methodes om eenvoudig nieuwe objecten aan te maken. Het idee is dat je `io.vavr.API.*` static importeert, waarna je dan bijvoorbeeld een nieuwe `List` aan kunt maken door `List("Apple", "Pear")` aan te roepen. Het keyword `new` hoeft dan niet gebruikt te worden. Het voorbeeld in **Listing 2** laat zien hoe een nieuwe `List`, een `Option` en een `Future` object gemaakt kunnen worden.

For expression

In Java kennen we de *for loop* en de *enhanced for loop*. Met het oog op functioneel programmeren hebben deze één nadeel. Het zijn namelijk statements in plaats van expressies. Om iets nuttigs te doen in de for-loop zijn *side effects* nodig: bijvoorbeeld een result lijst opbouwen, iets printen of een variabele muteren. Om deze uitdaging het hoofd te bieden,

```
<dependency>
        <groupId>io.vavr</groupId>
        <artifactId>vavr</arifactId>
        <version>0.9.2</version>
</dependency>
```

Listing 1. Vavr dependency

```
List<String> fruit = List("Apple", "Pear");
Option<String> maybeFoo = Some("foo");
Future<Integer> result = Future(() -> doSomeHeavyCalculation());
```

Listing 2. Gebruik constructor methods

```
Iterator<String> fruitColors = For(shops, shop ->
    For(shop.getFruit())
        .yield(fruit -> shop.getName() + " has fruit color " + fruit.
getColor()));
```

Listing 3. For expression

heeft Vavr de `For` comprehension, ook wel `for` expression genoemd. Dit is een soort for loop, maar dan eentje die het resultaat terug geeft in de vorm van een `Iterable`, een expressie dus. In **Listing 3** is het dus een for loop, die een iterator van strings teruggeeft.

Functions

Met de komst van Java 8 deed de `Functional-Interface` zijn intrede. Zeker in combinatie met Lambda's is dit een krachtig hulpmiddel. Standaard biedt Java 8 echter maar twee basis functional interfaces, die respectievelijk één en twee argumenten accepteren: `Function` en `BiFunction`. Vavr bouwt verder op dit idee met `FunctionN`, die tot wel acht argumenten accepteren: `Function0, Function1 .. Function8`. *Interoperability* met de Java types wordt gemaximaliseerd, doordat de `Function0` interface de `Supplier` interface implementeert. Verder implementeert `Function1` de Java `Function` en `Function2` implementeert de Java `BiFunction`. Naast de te verwachten `apply` functie, zijn deze interfaces voorzien van methodes, die het functioneel programmeren mogelijk maken met zaken als *partial application*, *currying* en *liften*.

Composition

Net als de Java Functions zijn de Vavr functions te composen met de `andThen` methode. Daarnaast is ook de `compose` methode te gebruiken. Deze methode ligt dichter bij de mathematische notatie. De functies `f: X → Y` en `g: Y → Z` kunnen gecomposed worden tot `h: X → Z`. In Java code ziet dit er dan uit als in **Listing 4**. De functies `trim` en `length` worden gecombineerd tot een nieuwe functie `h`.

Tuples

Tuples zijn een algemeen concept in functionele programmeertalen, zoals Haskell en Scala. Tuples zijn een immutable datastructuur, die een aantal objecten van verschillende types kan bevatten op een type-safe manier. Een goede use-case voor tuples is het teruggeven van meerdere objecten uit een functie, zoals in **Listing 5**. Het kan dan eenvoudiger zijn om een tuple te gebruiken dan voor dat ene geval een nieuwe (inner-) class te definiëren.

Values

Vavr komt met een heel aantal *immutable* datastructuren: values. Immutable wil zeggen dat een eenmaal gemaakt object niet meer wijzigt. Dit heeft als groot voordeel dat je thread-safety gratis krijgt. Als je bijvoorbeeld een element aan een Vavr `List` toevoegt, dan krijg je een nieuwe instantie van een `List`.

```
Function1 <String, String> f = String ::trim;
Function1 <String, Integer> g = String ::length;
Function1 <String, Integer> h = g.compose (f);

assertEquals((Integer)5, h.apply(“ HELLO “));
```

Listing 4. Compose functies

```
Public Tuple2<Integer, String> score(String rawInput {
       Return Tuple .of(rawInput.length(), rawInput .toLowerCase());
}
```

Listing 5. Tuple return type

Naast immutability hebben deze value classes ook een aantal interessante eigenschappen. Om deze eigenschappen te bundelen en zoveel mogelijk hergebruik te stimuleren, heeft Vavr de `Value` base class. Deze implementeert de `Iterable` interface. Hierdoor zijn alle Vavr value objects in `for` loops te gebruiken. Daarnaast hebben ze tal van conversie functies om ze naar andere datatypes te converteren. Operaties op `Value` objecten zijn erop gericht om zoveel mogelijk van de 'state' te sharen.

Collections

Een groot gedeelte van de Vavr API bestaat uit immutable collections. Deze collections zijn van de grond af aan opnieuw opgebouwd. De diverse collection types implementeren (op de Iterable interface na) geen Java interfaces. De belangrijkste reden hiervoor is dat de Java API grotendeels gebaseerd is op een mutable datastructuur. Denk hierbij bijvoorbeeld aan de `add()` en `remove()` methodes. Om toch goed samen te kunnen werken met bestaande API's, die op basis van Java collections werken, zijn views en converters voor de Vavr collections gedefinieerd.

Vavr Collections

De twee pijlers waarop de Vavr collection types zijn gebaseerd zijn: *Immutable Data Structures* en *Persistent Data structures*. Persistent Data structure wil zeggen dat de huidige versie van een object bewaard blijft en hergebruikt waar mogelijk. Bij een wijziging, zoals het toevoegen aan een collectie, wordt geen volledige kopie gemaakt van de oude lijst. Er wordt dan een nieuwe referentie gemaakt, die verwijst naar het toegevoegde object en de oude lijst.

DE IMMUTABLE COLLECTIES ZIJN EEN GOED STARTPUNT

Naast de immutable persistent eigenschappen hebben de collection types een erg rijke interface. Hierdoor zijn veel dingen rechtstreeks via de collectie te doen zonder daarvoor een `for` loop te hoeven gebruiken. Methodes zoals `map`, `filter`, `zip` en nog vele anderen kunnen gebruikt worden om collecties te transformeren. Daarnaast is er bijvoorbeeld de `mkString` methode, die items uit een collectie combineert tot een `String` met een separator naar keuze. **Listing 6** laat een voorbeeld zien van een aantal van deze methodes.

```
List<String> list = List("Apple", "Pear", "Banana", "Strawberry",
"Pineaple", "Melon");
List<String> specialFruit = list
        .filter(f -> f.startsWith("P"))
        .map(f -> f.toUppercase());

assertEquals ("PEAR;PINEAPLE", specialFruit.mkString (";"));
```

Listing 6. Filter en map op List

```
Option<seq<String>> maybeStrings = Option .sequence(Seq(Option
.of("Hello"), Option .of("World")));

String actual = maybeStrings
        .map(seq -> seq.mkString (","))
        .getOrElse("<none>");

assertEquals ("Hello, World", actual);
```

Listing 7. Sequence van Option

Container types

Naast de collections in Vavr zijn er nog container types. Types die een waarde encapsuleren en daar een bepaalde betekenis aan geven. Bijvoorbeeld de `Option` om aan- of afwezigheid van een waarde uit te drukken, `Try` om het al dan niet slagen van een operatie uit te drukken en `Future` om aan te geven dat operaties niet direct klaar zijn.

Option

Een eenvoudig, maar krachtig concept, is het kunnen uitdrukken van het optioneel kunnen zijn van een waarde. Je wilt niet op allerlei plekken in de code telkens op `null` hoeven te checken. Java 8 heeft hier al het `Optional` type voor. Vavr biedt een krachtiger variant aan met de `Option` class. `Option<T>` is een container voor values van type `T`, die aanwezig zijn of niet. Ook over deze `Option` kun je *mappen* om gemakkelijk resultaten te verwerken. Ook de `sequence` methode is één van de handige utilities op `Option`. Deze methode combineert een sequence van `Option` tot één nieuwe `Option`, die een sequence bevat. Een voorbeeld hiervan is te zien in **Listing 7**.

De semantiek van de `Option` is net iets anders dan Java's `Optional`. Het verschil zit hem in hoe de container om gaat met `null` values uit de `map` methode. Voor de details verwijs ik graag naar de blog post *"The agonizing death of an astronaut"* (zie referenties).

Try

Functioneel programmeren is vooral gericht op het geven van een output op basis van een input. Het *thrown* van excepties werken tegen dit principe. Het laat functies anders eindigen dan met een output van het verwachtte type. Om functioneel te kunnen programmeren, maar toch fout situaties af te handelen, biedt Vavr het type `Try<T>`. Een Try kan of een `Success<T>`, als de operatie geslaagd is, of een `Failure` zijn, waarbij de `Exception` in de `Failure` wordt teruggegeven. Typisch gebruik je een resultaat van type `Try` door middel van de `map` of `andThen` methodes. Deze functies worden alleen uitgevoerd als de `Try` een `Success` is. Om ook iets met de foutsituatie te kunnen doen, kan bijvoorbeeld de `getOrElse` methode gebruikt worden. Een combinatie van deze functies, zoals weergegeven in **Listing 8**, kan natuurlijk ook.

Future

Voor bewerkingen waarvoor het resultaat op een later moment teruggeven wordt, biedt Vavr de `Future`. Afhankelijk van een (optioneel) mee te geven `ExecutionContext` wordt de daadwerkelijke operatie op een aparte thread uitgevoerd. Het is mogelijk om te wachten op het resultaat, met bijvoorbeeld de `get` methode. Het is echter gebruikelijker om door het gebruik van transformatie functies (zoals `map`) het resultaat te transformeren. Het resultaat is dan nog steeds een `Future`, alleen dan een `Future` die het uiteindelijk getransformeerde resultaat bevat. Je hoeft dus uiteindelijk geen blocking call te doen.

Listing 9 laat zien hoe je bewerkingen kunt doen, die uitgevoerd worden zodra de (potentieel langdurende) operatie van de `Future` afgerond is. Ook de `Future` class heeft weer een rijke set aan operaties, die ondersteuning biedt om futures op verschillende wijzes te combineren of te filteren, net zoals de `Try` en `Option`.

VAVR IS EEN KRACHTIGE LIBRARY MET EEN DUIDELIJKE FOCUS: FUNCTIONEEL PROGRAMMEREN

Validation

In alle gevallen waar je met user input te maken hebt, wil je input validatie doen. Validatie van een inputveld kan twee uitkomsten hebben: 1) input is goed → ga verder of 2) de input is invalid → geef een duidelijke foutmelding. De `Validation` class van Vavr drukt dit goed uit. Je kunt zelf de types voor zowel de succes- als de foutsituatie specificeren. In **Listing 10** is ervoor gekozen om validatie fouten als `String` terug te geven en gevalideerde input als `Integer`.

Voor één validatie is dat nog niet zo spannend. Interessanter wordt het echter als je meerdere validaties hebt. Bij voorkeur zou je in één keer alle validatiefouten aan de gebruiker melden. Met Vavr kun je hiervoor de `combine` functie gebruiken. Deze heeft als input een aantal `Validation` objecten en levert een `Validation.Builder` op. Met de `ap` functie op die builder kun je de gevalideerde velden combineren tot een object naar keuze. Mocht een of meerdere van de validaties zijn gefaald, dan krijg je uiteraard de foutmeldingen terug en wordt er geen object gemaakt. Bijvoorbeeld in **Listing 11** wordt op basis van gevalideerde `name` en `age` een `Person` object gemaakt als de validaties slagen. Als één of meer validaties falen dan worden de validatiefouten geprint.

Conclusie

Vavr is een krachtige library met een duidelijke focus: functioneel programmeren. De library brengt geen additionele dependencies met zich mee en is daardoor lichtgewicht om mee te starten.

Vavr's immutable collecties zijn een goed startpunt om met de library te beginnen. De immutability en de utility methods op de collecties zijn erg prettig om mee te werken. Het is mooi om te zien hoe getracht wordt om het functioneel programmeren zo gemakkelijk mogelijk te maken. De `For` expression is daar een mooi voorbeeld van. Het is een goede poging, maar vergeleken met bijvoorbeeld Scala voelt de `For` toch wat onhandig aan in het gebruik. Het werken met de `functions` is krachtig, maar de uitgebreide type-declaraties met generics leiden soms wat af van wat je werkelijk wilt bereiken.

Het is onmogelijk om in dit artikel alle ins en outs van de hele library te beschrijven. Naast de core module zijn er nog aparte modules voor JSON support, Property based testing en GWT support. Het beste beeld krijg je uiteraard als je er gewoon mee aan de slag gaat. De User Guide en de JavaDoc (zie referenties) zijn een mooi startpunt om alle utility methodes te ontdekken.

Op het moment van schrijven was versie 0.9.2 de meest recente versie. Echter wordt er tegelijkertijd hard gewerkt aan de 1.0 versie van Vavr. Deze versie zal niet volledig backward compatible zijn met 0.9.x. De releasedatum is nog niet vastgesteld, maar zal na de release van Java 11 zijn. ■

```
Try<String> response = doTrickyStuff();
Tuple2<Integer, String> actual = response
       .map(r -> Tuple .of(200, "OK: " +r))
       .getOrElse(Tuple .of(500, "Server error"));

assertEquals (Tuple .of(200, "OK: It works"), actual);
```

Listing 8. Verwerken van een try

```
final String fruit = "Apple";
final Float price = 0.35f;
Future<float> eventualTotalValue = Future(() ->
longRunningOperation(fruit))
       .onSucces(amount -> printf("found %d %s", amount, fruit))
       .map(amount -> amount * price);
```

Listing 9. Eenvoudig Future voorbeeld

```
private static Validation<String, Integer> validateAge(int age) {
    return age > 18 ? Valid(age) : Invalid("Person should be older
than 18");
}
```

Listing 10. Valideer leeftijd

```
Validation <seq<String>, Person> personValidation = Validation
       .combine(validateName (someName), validateAge(someAge))
       .ap((name, age) -> new Person(name, age));

If (personValidation.isInvalid()) {
       printIn (personValidation.getError().mkString());
}
```

Listing 11. Valideer Person

REFERENTIES

- User guide: **http://www.vavr.io/vavr-docs/**
- JavaDoc **https://www.javadoc.io/doc/io.vavr/vavr/**
- The agonizing death of an astronaut: **http://blog.vavr.io/the-agonizing-death-of-an-astronaut/**