

Introductie

Ontwikkelaars krijgen steeds meer tools aangereikt om asynchrone afhandeling van applicatielogica mogelijk te maken. Asynchrone programmalogica biedt voordelen t.o.v. synchrone code, omdat er efficiënter gebruik kan worden gemaakt van de beschikbare resources. Zo kun je bijvoorbeeld een database query uitvoeren en terwijl je wacht op het resultaat een webservice request kunnen uitvoeren. Er zijn verschillende mogelijkheden om asynchrone code te schrijven. Voor javascript ontwikkelaars is het bijvoorbeeld vanzelfsprekend om gebruik te maken van callbacks. Sinds versie 8 is deze mogelijkheid er ook in Java in de vorm van Lambda statements. Ook kan er gebruik worden gemaakt van multi-threading, van patterns zoals een event-loop, van co-routines (nodejs, kotlin) en van libraries als reactive extensions (RxJava, RxJS). Asynchrone afhandeling is vooral voordelig als we te maken hebben met interactie tussen applicaties. Het meest populaire protocol hiervoor is echter REST. Gebaseerd op HTTP is dit een synchroon protocol.

In dit artikel kijken we aan de hand van een voorbeeld naar een alternatief in de vorm van websockets. We zullen een systeem bouwen met een Angular frontend en een Vert.x backend. Voor de asynchrone afhandeling maken gebruik van Observables in RxJS en RxJava. Het voorbeeld gebruikt websockets omdat dit een populaire standaard is. Het principe kan echter ook worden toegepast op vergelijkbare protocollen zoals bijvoorbeeld TCP.

Use case

Aan de hand van de volgende usecase zullen we laten zien wat de kracht van Reactive Extentions is. We zullen verschillende operators gebruiken en bespreken. De usecase is gekozen zodat we te maken hebben met asynchrone data communicatie die over tijd wordt uitgevoerd. Dit is precies het domein waar Reactive Extentions het best tot zijn recht komt. Door gebruik te maken van RxJS en RxJava houden we de code compact en leesbaar.

We maken een webapplicatie waarin gebruikers kunnen zoeken naar films. De hoofdfunctionaliteit bestaat uit een tekstveld waar zoek termen ingetikt kunnen worden. De applicatie toont vervolgens direct de gevonden films.


MoviesSearch

Find a movie

Searchterm
the

The Dark Tower

Aug 3, 2017




ActionWesternScience Fiction

FantasyHorror

The last Gunslinger, Roland Deschain, has been

World War Z

Jun 20, 2013




ActionDramaHorror

Science FictionThriller

Life for former United Nations investigator

Death Note

Aug 25, 2017




MysteryFantasyHorrorThriller

A young man comes to possess a supernatural notebook, the Death Note, that grants him the power to kill any person simply by writing down

The Limehouse Golem

Aug 31, 2017



HorrorThriller

A series of murders has shaken the community to the point where people believe that only a legendary creature from dark times - the

De filmdatabase waar de applicatie gebruik van maakt kan veel films bevatten. Een zoekresultaat kan dus potentieel veel resultaten opleveren. Het resultaat zal dus asynchroon naar de browser moeten worden gepushed, zodat de gebruiker niet lang hoeft te wachten totdat er films op het scherm verschijnen.

Elke keer als de gebruiker een letter intikt, zal het zoekresultaat moeten worden aangepast. Hierbij is het belangrijk dat de gebruiker geen oude zoekresultaten voorgeschoteld krijgt.

Technologie

Voor het versturen van de zoekterm en het ontvangen van de resultaten zullen we gebruik maken van een WebSocket verbinding. Websockets maken het mogelijk om full-duplex asynchrone communicatie uit te voeren. Dit in tegenstelling tot REST, waarbij er altijd sprake is van een synchrone Request en Response. Met REST is het voor ons dus niet mogelijk om het zoekresultaat naar de browser te pushen in de vorm van een Stream.

In dit voorbeeld maken we gebruik van een Angular 4 frontend en een Vert.x 3 backend. Beide frameworks bevatten websocket API's die gebruik maken van Reactive Extensions. In Angular kun je dus gebruik maken van RxJS en in Vert.x van RxJava. Beide kanten van de lijn maken dus gebruik van Observables, wat ervoor zorgt dat de frontend en backend API's erg op elkaar lijken.

Frontend

Backend

De backend heeft voor iedere browserclient een websocket connectie. Via deze connectie komen berichten binnen. In ons geval representeren de berichten een zoekterm voor het zoeken naar films. Elke keer als de gebruiker een nieuwe letter typt, krijgen we een nieuwe zoekterm binnen. Het is dus zaak dat deze zoekacties snel achter elkaar kunnen worden uitgevoerd. Hiernaast is het nodig om elke vorige zoekactie te annuleren als er een nieuwe zoekactie voor dezelfde client binnenkomt. Dit doen we om te voorkomen dat de gebruiker oude resultaten ziet bij zijn nieuwe zoekterm.

Het volgende code snippet toont de belangrijkste functionaliteit voor het verwerken van deze zoekacties.

```
HttpServer server = vertx.createHttpServer(); //1
Observable<ServerWebSocket> wsStreamObservable
    = server.websocketStream().toObservable(); //2
wsStreamObservable.subscribe( //3
    socket -> { //4
        socket.toObservable() //5
            .map(buffer ->
                Json.decodeValue(buffer.toString("UTF-8"), WSAction.class)) //6
            .filter(action -> action.isSearch()) //7
            .map(action -> action.getBody()) //8
            .filter(searchTerm -> searchTerm.length() >= 3) //9
            .switchMap(movieService::findMovies) //10
            .map(movie -> movie.encode()) //11
            .subscribe(socket::writeTextMessage); //12
    }
);
```

Listing 1

Op regel 1 maken we een nieuwe Vert.x HTTP server aan. Deze kan HTTP(s) en WebSocket verbindingen voor verschillende clients faciliteren. Op regel 2 vragen we de websocket stream op voor deze server. Deze transformeren we tot een Observable. Dit maakt het mogelijk ons de abonneren op nieuwe websocket connecties (wsStreamObservable.subscribe, regel 4). Per client krijgen we een nieuwe socket binnen (regel 4). Deze socket representeert de verbinding tussen de server en de client. Via deze socket ontvangen we dus berichten en kunnen we berichten terugsturen. Hiervan maken we wederom een Observable (regel 5). Tot zover het opzetten van de verbindingen.

De berichten komen binnen als bytes. Deze gaan we eerst vertalen naar een JSON representatie, welke we vervolgens direct omzetten naar een Java object. Hiervoor maken we gebruik van de Rx map operator (regel 6).

De map operator voert een functie uit op elk van de items en retourneert het resultaat als een

Observable. In dit geval komt er voor elke zoekactie een buffer instantie binnen die wordt omgezet naar een WSAction object, zie hiervoor het marble diagram in Diagram 1. Marble diagrams zijn visuele representaties van functionele operators.

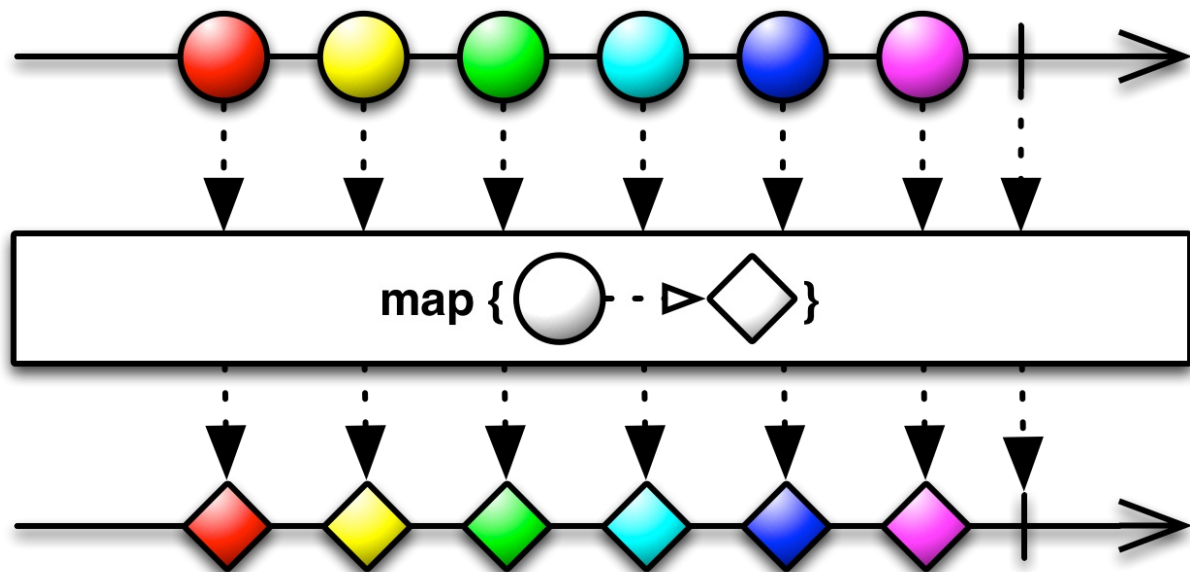


Diagram 1: map

Vervolgens willen we alleen zoekacties gaan afhandelen. Het is mogelijk dat we via dezelfde websocket connectie andere acties binnen krijgen. We willen in onze stream dus alleen de zoekacties doorlaten en andere acties negeren. Hiervoor gebruiken we een filter (regel 7). Een filter laat alleen items door die voldoen aan het meegegeven predicaat. In dit geval is dat `action.isSearch()`. Dit resulteert weer in een nieuwe Observable, die alleen de doorgelaten items bevat.

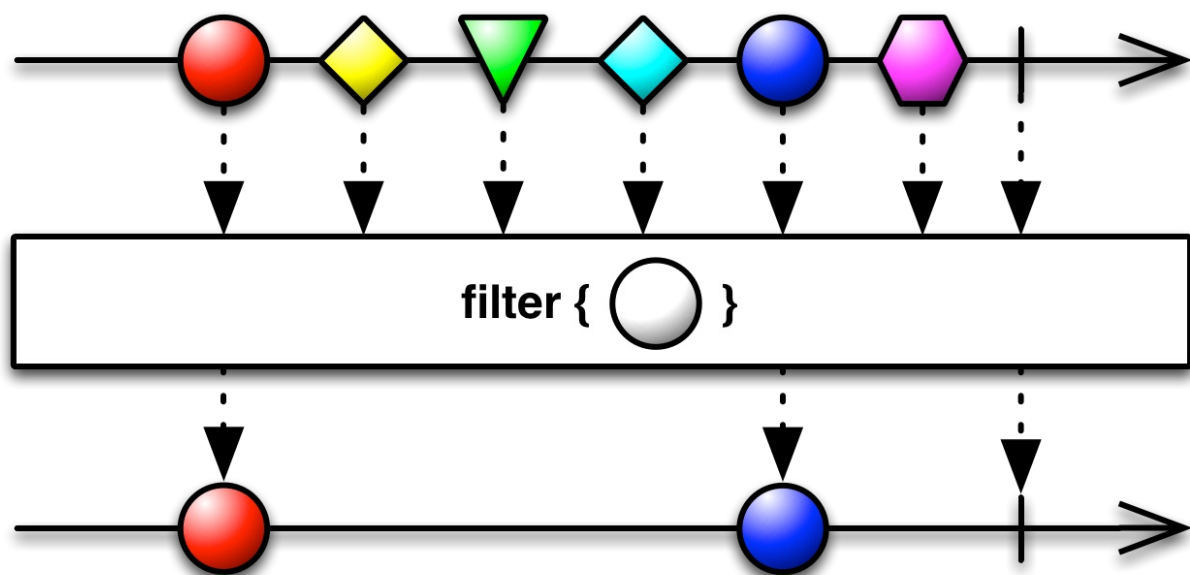


Diagram 2: filter

We willen van deze zoekactie de body hebben, deze bevat namelijk de zoek term. Hiervoor

gebruiken we wederom een map operator(8). Om te zorgen dat we alleen zoekacties van 3 of meer karakters verwerken gebruiken we wederom een filter (9).

Als de gebruiker een nieuw karakter intikt, krijgen we via de websocket verbinding een nieuwe zoekterm binnen. Op dat moment zijn we nog de vorige zoekacties aan het verwerken. Deze moeten we op dus annuleren. Als je dit uit zou implementeren zonder Observables, dan zou de code er uit kunnen zien als in Listing 2.

```
WSAction action = null;

try {
    action = Json.decodeValue(message, WSAction.class);
} catch (DecodeException e) {
    ws.writeTextMessage(new JsonObject()
        .put("status", "invalid request").encode());
}

if (action != null && action.isSearch()) {
    Subscription existingSearch = subscriptions.get(ws.textHandlerID());
    if (existingSearch != null) { //1
        existingSearch.unsubscribe(); //2
    }

    Subscription newSearch
        = movieService.findMovies(action.getBody()).subscribe(movie -> { //3
        ws.writeTextMessage(movie.encode());
    });

    subscriptions.put(ws.textHandlerID(), newSearch); //4
```

Listing 2

In deze implementatie van de zoek functionaliteit houden we een map van subscriptions bij op basis van het textHandlerID. Dit ID representeert een verbinding met een client.

We moeten hier dus expliciet controleren of er een bestaande zoekactie is (1). Als deze bestaat, roepen we hier unsubscribe aan, zodat de actie wordt geannuleerd (2).

We moeten vervolgens een nieuwe Subscription aanmaken (3) en deze aan de map toevoegen (4). Hierdoor introduceren we een expliciete variabele waarin we state moeten bijhouden, de map met subscriptions.

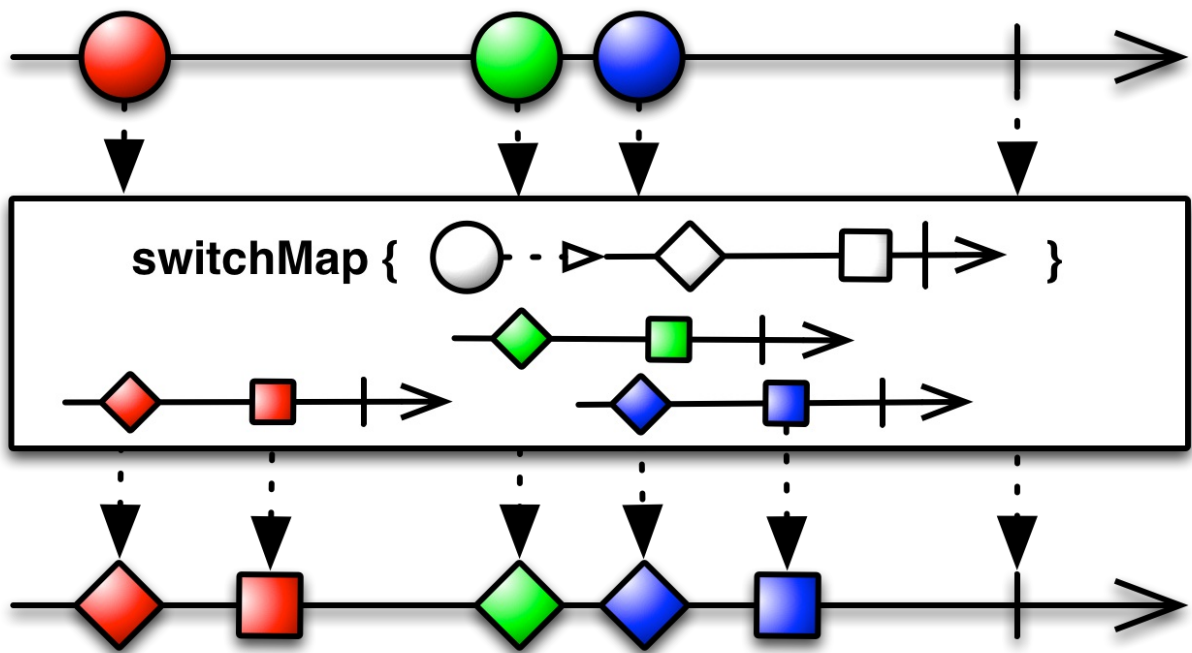


Diagram 3: switchmap

De switchmap kan dit voor ons afhandelen. Een switchmap subscribed op een Observable, in ons geval op het resultaat van het filter op regel 9. Voor elke zoekterm beginnen we met het ophalen van films door het aanroepen van de findMovies methode op onze movieService. Deze findMovies geeft een Observable van films terug in JSON formaat (Listing 3). Per zoekterm krijgen we dus meerdere films binnen over tijd. Op het moment dat er een nieuwe zoekterm binnenkomt uit de Observable van het filter in regel 9, dan zorgt de switchmap ervoor dat er een unsubscribe plaats vindt op de Observable die terugkomt uit findMovies methode en die behoorde bij de oude zoekterm. De findMovies methode wordt opnieuw uitgevoerd met de nieuwe zoekterm en er vindt een subscribe plaats op de nieuwe Observable die de films bevat die horen bij deze nieuwe zoekterm.

```
public Observable<JsonObject> findMovies(String keyword);
```

Listing 3

Tot slot gebruiken we nogmaals een map operator om de films om te zetten naar een JSON formaat wat we als websocket response kunnen teruggeven. Het terugsturen van de films over de websocket verbinding doen we in de subscribe methode (regel 12).

Conclusie