CLOUD COMPUTING AND BIG DATA PROCESSING

RXJS: TWITTER POPULARITY TRACKER APP

January 12, 2020

GEISSER AGUILAR NIKLAUS
Niklaus.Geisser@vub.be
0554142

Contents

1 INTRODUCTION

2 REACTIVE PROGRAMMING

Reactive Programming is a paradigm which main approach is to handle continuous time-varying values and propagation of change. It is frequently used in event driven applications since changes in the state are propagated across dependant computations by the execution of the model. Take for instance the sum of two variables (a = 1; b = 2). In the sequential imperative case, the resulting variable will always be 3, even though the values of the variables can change over time. However, in reactive programming, the resulting variable will always be kept up to date, meaning that the application will "react" to the changes of the variables *a* and *b* and the sum will be recomputed. This means that all dependant computations will be recomputed when the state changes [1]. For instance, it can be said that spread sheets are reactive

2.1 First class abstraction

There are two classes of first class abstractions in reactive programming which are behaviours and events. The term behaviour refers to time-varying or continuous values. Examples of behaviours include timers, mouse position, input field content, etc. On the other hand, events refer to streams of timed or discrete values. For instance, mouse clicks, key presses and asynchronous responses. Events and behaviours are dual to each other [1].

2.2 Evaluation Model

The way changes are propagated through the dependency graph in reactive programming is called evaluation model. It is important to identify the actor that will initiate the propagation of changes. In other words, whether the master node will "push" new data to the child nodes (driven by availability) or they will "pull" the data from the master node (driven by demand) [1]. These two approaches are called "Pull based" and "Push based" evaluation models. Most cloud computing solutions will use event-push based approaches

2.3 Glitch avoidance

Glitches are temporary inconsistencies due to the update propagation order. Glitches can only happen in push based models, meaning that when the source gets new data it propagates it through all dependent computations. A way to avoid glitches is to arrange expressions in a topologically sorted graph, in that way an expression can only be evaluated after all the others that depend on that one have been evaluated [1]. Most reactive programming languages eliminate glitches by arranging ex- pressions in a topologically sorted graph [Cooper and Krishnamurthi 2006; Meyerovich et al. 2009; Maier et al. 2010], thus ensuring that an expression is always evaluated after all its dependents have been evaluated.

2.4 Lifting Operation

The mapping of an operator to a behaviour variant is called lifting. This operation changes a function's type signature and it pushes a dependency graph in the general dataflow of the application. $f(s) => f_{lifted}(Behaviour < S >)$

3 RxJS

ReactiveX is a library for handling asynchronous and event-based programs by making use of observable sequences combining the observer and iterator patterns along with functional programming[2].

3.1 Observer Pattern

In this pattern, there exists an object called Producer that has a list of listeners or "observers" subscribed to it. Whenever the state of the producer changes, it notifies the listeners by calling their "update" methods [3].

3.2 Iterator Pattern

In this pattern, an object is provided to a consumer in order to traverse its contents in an easy way. The interface is composed of two methods:

- **next()** Gets the next item in the list
- **hasNext()** Check if there is a next item in the list

The main objective of the pattern is to encapsulate a traversing logic for different data structures [3].

3.3 Observable

The combination of the Observer and Iterator pattern results in another pattern called Rx which is named after the Reactive extension libraries. Observables are a central part of this pattern. They can emit values to consumers when they become available. When a listener or observer is subscribed to an Observable it will receive the values of the sequence whenever the data become available without making a request.

4 RxJS + ReactJS

The present project was developed implementing two libraries. ReactJS to build the user interface and RxJS to handle the events and asynchronous flow of the application. Even though this project could have been developed without using a web framework, it is interesting to see how RxJs can be adapted to work within any javascript solution. ReactJS was chosen for two main reasons:

- Is the second most used library for building UI.

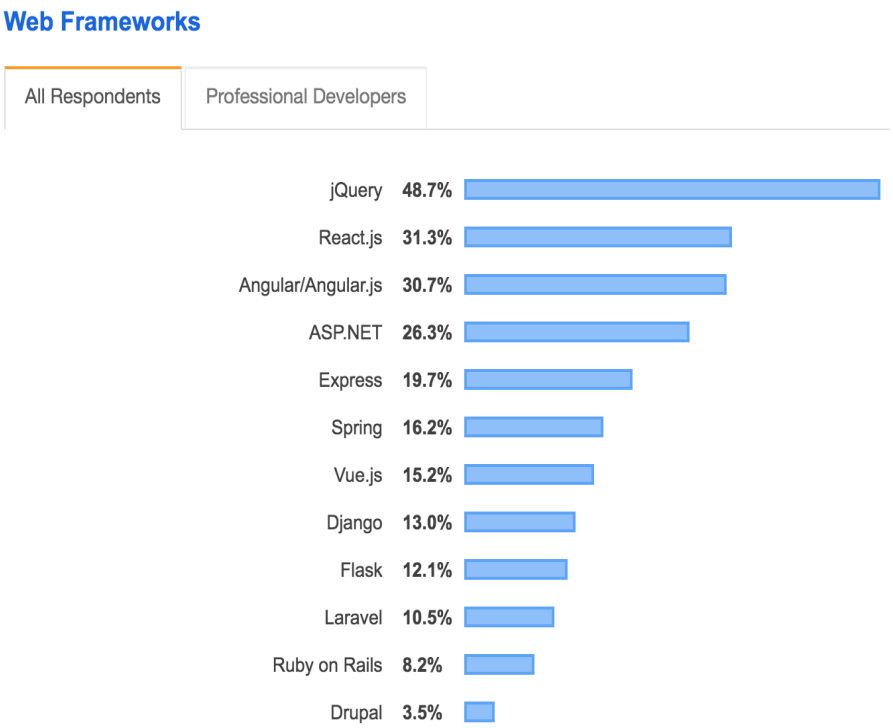

Figure 1: StackOverflow developer survey 2019

- It does not come with an integration out of the box of RxJS unlike Angular, thus it makes it a little bit more challenging to join these two libraries.

5 WEB APPLICATION DESCRIPTION

The web application developed in the present project has the following features that implement several operators of RxJS:

- A tweet counter that displays the number of incoming tweets in the stream based on a topic.
- A component that connects to a web socket and let's the user change the topic of the tweets.
- A component that shows the incoming tweets with a link that takes the user to the actual tweet description in twitter.
- A component that shows the popular hashtags based on the topic of the tweet stream.
- A component that lists the distinct languages in which the tweets are written.

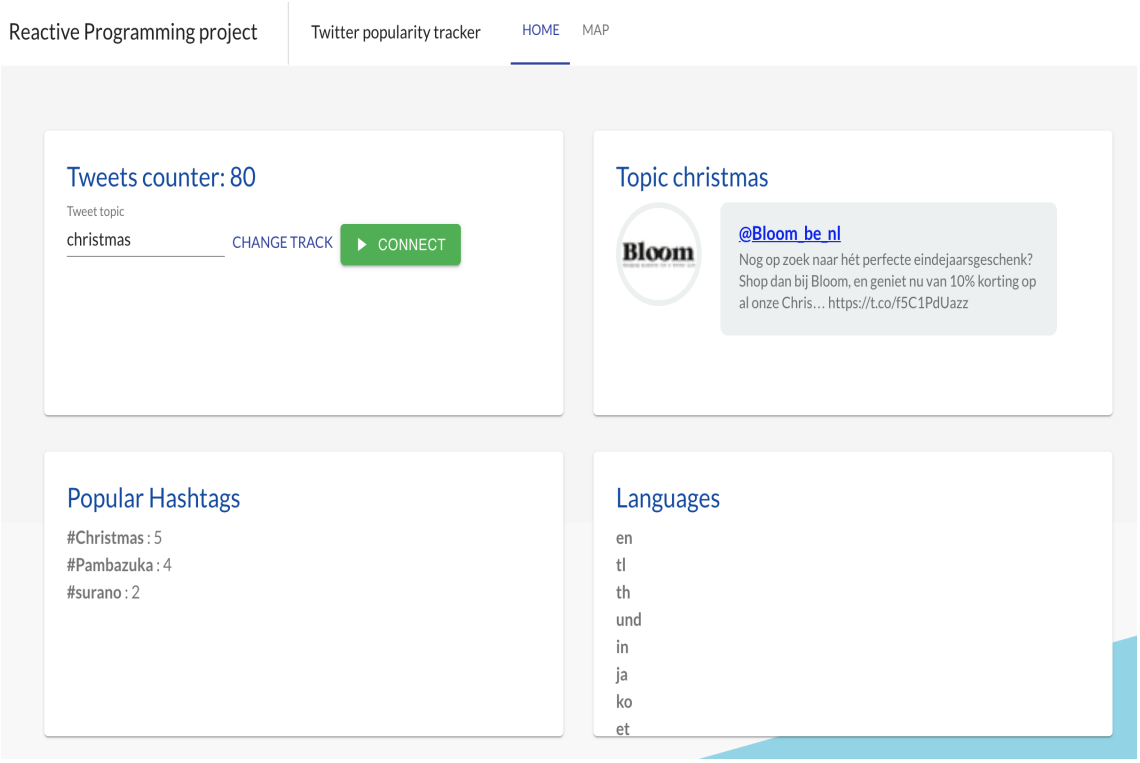

Figure 2: Home page of web application

- A component that shows in a map the location of the incoming tweets along with a list displaying them.

The backend of the application was implemented in NodeJs using web sockets

6 Implementation

6.1 Twitter interaction

In order to consume the twitter API, it is necessary to apply for a developer account describing what are the use cases intended to use the API. Once the application is approved, one can call the necessary endpoints for the streaming. In particular, the service used is the following: https://stream.twitter.com/1.1/statuses/filter.json, which returns public statuses related to a filter predicate. To consume the service, the following library was used: https://www.npmjs.com/package/node-tweet-stream. Additionally, socket.io library was used to handle web sockets communication. Whenever a client connects to a socket in the server we start consuming the streaming. In the frontend side, we declare a Subject (special type of observable that allows values to be multicasted to multiple observer [4]), that will emit the values received by the sockets.

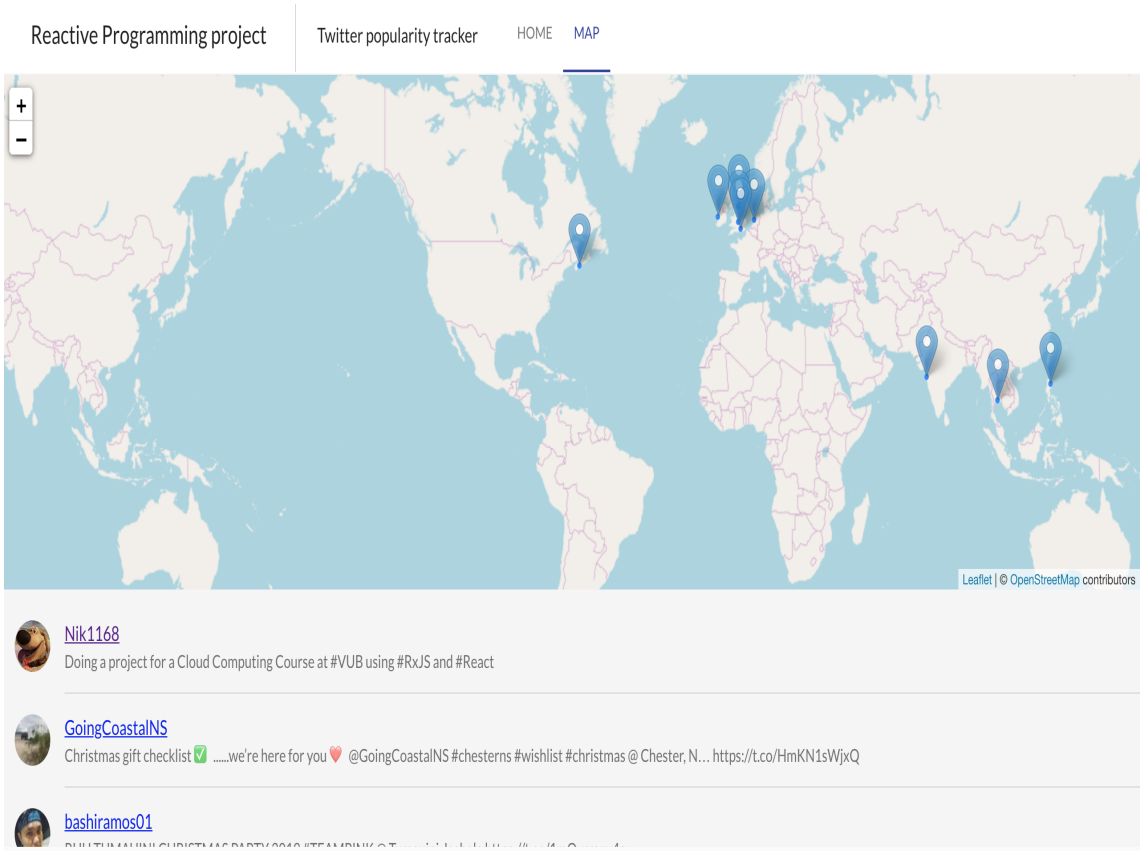

Figure 3: Map showing location of tweets

```
// Front end
let tweetsStream = new Subject();
socket.on('tweet', data => tweetsStream.next(data));
```

```
// Backend
io.on("connection", function (socket) {
    t.on('tweet', function (tweet) {
        socket.emit('tweet', tweet);
    });
});
t.track('christmas');
```

6.2 Operators used

For each of the features of the application, several operators were used that will be described as follows:

6.2.1 Display tweets

In order to displays the incoming tweets from the stream, we subscribe to the observable and use a **debounce** in order to wait a specific amount of time to send each tweet.

```
tweetsStream
            .pipe(
                debounce(() => timer(50))
            )
            .subscribe((tweet) => {
               // handle list tweets
            })
```

Additionally, a request has to be made in order to change the topic of the tweet streaming. In order to do that, we transform the request which is a promise into an observable using the **from** method from rxJS.

```
export const changeTrack = (track) =>
    from(fetch(`${URL_SERVER}/changeTrack`, {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({track})
    })
        .then((res) => res.json()));
```

6.2.2 Tweet counter

In order to implement the tweet counter we declare an object that will subscribe to the data stream to receive each Tweet.

```
tweetsStream
            .pipe(scan(counter => counter + 1, 0))
            .subscribe(counter => {
                this.setState({count: counter})
            })
```

In this case, we use the **scan** transformation which applies an accumulator over a source Observable [4]. Once the counter is received in the subscription, we can update the state of the component to show it. (Implementation based on practical sessions of the course).

6.2.3 Popular hashtag tracker

There are different factors to take into consideration for the implementation of this feature: first of all, not all tweets have hashtags, therefore, we need to perform a filter. (Implementation based on practical sessions of the course)

- Not all tweets have hashtags, therefore, we need to perform a **filter**.
- A tweet can have several hashtags, therefore, in order to get all hashtags we **map** and transform the array of hashtags into an observable.
- In order to be able to group the hashtags we use the **contactAll** operator to pass the subscription of this observable to the next one.
- Next we use **groupBy** to group the streams by each hashtag. This operator returns a new grouped observable
- Once this is done, we apply a **scan** operator to count the number of occurrences of each hashtag. In order to not list all possible hashtags, we consider a Tweet to be popular if it has more than two occurrences in the stream. It is important to remark that this is a high order observable.

```
tweetsStream
            .pipe(
                filter(tweet => tweet.entities.hashtags.length > 0),
                map(tweet => from(tweet.entities.hashtags)),
                concatAll(),
                groupBy(hashtag => hashtag.text)
            )
            .subscribe(groupedObservable => {
                groupedObservable
                    .pipe(
                        scan((count, current) => {
                            return {key: current, size: count.size + 1}
                        }, {key: '', size: 0}),
                        filter(res => {
                            return res.size > 1
                        })
                    )
                    .subscribe((result) => {
                        // handle results

                    });
            })
```

6.2.4 Display languages

To display the languages of the incoming tweets we use the **distinct** operator since several tweets in the stream have the same language.

```
tweetsStream
            .pipe(distinct(tweet => tweet.lang))
            .subscribe(tweet => {
               // show tweets
            })
```

6.2.5 Map with tweet's location

In this section, we will show how can we use observables with behaviours, we will use DOM events to change the opacity of a marker in the map by hovering over the list of the respective Tweet. In order to do so, we will use the library react DOM, which has a mouseover event. We create two observables for both events (when hovering or not) and we merge them to make a subscription [3].

```
tweetsStream
            function isHovering(element) {
        const over = Rx.DOM.mouseover(element).map(identity(true)); ]t
        const out = Rx.DOM.mouseout(element).map(identity(false));
        return over.merge(out);
    }
```

To display the tweets in the map we simply make a subscription and filter tweets that have a location. Refer to source code for details of the implementation or go here to see a live demo.

7 Run project

There are two folders called "frontend" and "server". First, install all necessary dependencies in both folders by running **npm install**. Next start server by entering "server" folder and typping **npm start**. Do the same for frontend folder. Refer to README file in the source code for more details.

8 Conclusions

It can be concluded that RxJS is a powerful tool to deal with asynchronous events and streams of data. It certainly has advantages over promises and callbacks functions. A lot can be done with simple approaches.

Bibliography

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM computing surveys*, 45(4):1–34, 2013.

[2] Reactive X. http://reactivex.io/intro.html. Accessed: 2019-11-23.

[3] Sergi Mansilla. *Reactive Programming with RxJS 5: Untangle Your Asynchronous JavaScript Code*. Pragmatic Bookshelf, 2018.

[4] Rxjs home page. https://rxjs-dev.firebaseapp.com/. Accessed: 2019-11-23.