Einführung in RxJS

Michael Ruttka, buschmais GbR

Den ersten Kontakt mit den sogenannten „Reactive Extensions for JavaScript“ (RxJS, siehe „http://reactivex.io/rxjs“) hat der Autor beim Umstieg von Angular.js auf Angular 2 gemacht. Wurden in Angular.js asynchrone Prozesse noch mit Promises abgebildet, so hat man sich in Angular 2 für Observables von RxJS entschieden. Es war also an der Zeit, mal einen genaueren Blick auf dieses Framework zu werfen. Ziel dieses Artikels ist es, die grundlegenden Konzepte von RxJS kennenzulernen und eine solide Wissensbasis zu erlangen, um seine ersten Schritte mit RxJS zu wagen.

Promises und Observables sind eine Art Platzhalter für noch nicht bekannte Ergebnisse. Ein einfaches Beispiel stellt die HTTP-Schnittstelle dar. Sendet der Client eine HTTP-Anfrage, ist deren Ergebnis erst nach der Antwort des Servers bekannt. Der Aufrufer erhält von der Schnittstelle einen solchen Platzhalter, kann auf dessen Ergebnis lauschen und unterdessen beispielsweise eine Lade-Animation anzeigen.

Was aber ist an Observables anders als an Promises? Der entscheidende Unterschied besteht darin, dass Observables über die Möglichkeit verfügen, abgebrochen werden zu können und mehr als nur ein Ergebnis zu verarbeiten. Dies kann beispielsweise bei der Event-Behandlung in einem Client-Framework wie Angular sehr von Vorteil sein.

Einrichtung

Bevor man mit RxJS loslegen kann, muss es zunächst einmal eingerichtet sein. In diesem Artikel wird „npm“ verwendet, um die Abhängigkeiten zu verwalten. Also wird auf der Kommandozeile „npm install rxjs –save“ ausgeführt, um RxJS in der neuesten Version in die „package.json“ aufzunehmen.

Da in diesem Artikel TypeScript zum Einsatz kommt, ist außerdem

noch der TypeScript-Compiler erforderlich. Er wird mit „npm install typescript -g" global auf unserer Entwicklungsmaschine installiert. Das Code-Beispiel in *Listing 1* zeigt, ob die Installation erfolgreich war.

Um den TypeScript-Code ausführen zu können, muss er zuerst mit „tsc hello-rxjs.ts" vom TypeScript-Compiler in JavaScript kompiliert werden. Danach sollte im Dateisystem eine Datei „hello-rxjs.js" auftauchen. Diese kann nun mit „node hello-rxjs.js" von Node.js ausgeführt werden. Es erscheint „hello RxJS" auf der Kommandozeile.

In diesem kurzen Beispiel ist bereits ein erster Fallstrick versteckt. Es gibt zwei Varianten, wie RxJS importiert werden kann. Innerhalb dieses Artikels wird ausschließlich die einfachere Variante, das „Komplettpaket", verwendet. Die Alternative bindet nur die verwendeten Komponenten in die Anwendung ein. Dies empfiehlt sich vor allem dann, wenn es auf die Auslieferungsgröße ankommt. In dieser Variante sind auch alle verwendeten Operatoren (dazu später mehr) einzeln zu importieren.

Observables

Das Herz von RxJS bilden die Observables. Ein Observable ist die Repräsentation einer beliebigen Menge von Werten, die über eine beliebige Zeitdauer verteilt sein können. Mit einer Subscription kann man diese Werte beobachten. Werte sind oft auch Ereignisse (etwa ein Mausklick oder eine Navigation), können aber auch Werte (wie 1, 2, 3, A, B,C) im engeren Sinne sein.

Das einführende Beispiel zeigt bereits ein solches Observable. Die dort verwendete Klasse „Subject" implementiert das Interface „Observable" und erweitert dieses um zusätzliche Methoden (wie „next") zur Auslösung des Observable. Aufgrund dieser Möglichkeit eignet sich Subject zur einfachen und nachvollziehbaren Demonstration der Funktionsweise von RxJS.

Die Klasse „Observable" besitzt die Methode „subscribe", mit deren Hilfe man eine Subscription erhält. Observables haben immer einen Zustand. Solange keine Subscription für ein Observable aktiv ist, hat es den Zustand „cold". In diesem Zustand interessiert sich niemand für die Werte des Observable, also werden sie auch nicht verarbeitet.

Mit dem Aufruf von „subscribe" und der damit verbundenen Erzeugung einer Subscription geht das Observable in den Zustand „hot" über. Die Werte des Observable werden nun überwacht und verarbeitet. Werte, die vor der Subscription in das Observable gereicht wurden, wird der neue Subscriber nie erfahren. Das Beispiel in *Listing 2* würde also lediglich die Werte „2" und „3" ausgeben.

Da zum Zeitpunkt der Subscription der Wert „1" bereits gesendet wurde, wird die Subscription lediglich über die nachfolgenden Werte benachrichtigt. Das Beispiel zeigt außerdem, dass die an „subscribe" übergebene „Arrow"-Funktion jeweils einmal pro Wert des Subject ausgeführt wird. Doch die Methode „subscribe" verfügt noch über zwei weitere Parameter, um auf Ereignisse des Observable zu reagieren. Das zweite Argument ist ein Callback für Fehler und das dritte Argument wird ausgeführt, wenn das Observable „completed" ist. *Listing 3* zeigt ein Beispiel, das diese beiden Callbacks demonstriert.

Die Ausgabe dieses Code-Schnipsels würde die Werte „1" und „2" sowie einen Fehler liefern, nicht aber die „finished"-Meldung. Das

```
import {Subject} from 'rxjs/Rx';
const subject = new Subject<string>();
subject.subscribe((value) => console.log(`hello
${value}`));
subject.next('RxJs');
```

Listing 1

```
import {Subject} from 'rxjs/Rx';

const subject = new Subject<string>();
subject.next('1');
subject.subscribe((value) => console.log(value));
subject.next('2');
subject.next('3');
```

Listing 2

```
import {Subject} from 'rxjs/Rx';

const subject = new Subject<string>();
subject.subscribe(
  (value) => console.log(value),
  (error) => console.error(error),
  () => console.log('finished'));

subject.next('1');
subject.next('2');
subject.error(new Error('something went wrong'));
subject.complete();
```

Listing 3

liegt daran, dass ein Observable nach einem Fehler nicht mehr beendet werden kann – es endet also entweder in dem Zustand „Error" oder „completed". Würde man also die Zeile weglassen, die den Error auslöst, würde neben „1" und „2" auch die Ausgabe „finished" erscheinen. An dieser Stelle eine weitere Besonderheit: Fehler, die im Value-Callback (erstes Argument der „subscribe"-Methode) auftreten, werden nicht vom Error-Callback behandelt.

Hat man erst einmal eine Subscription in der Hand, ist man auch in der Verantwortung, diese wieder aufzulösen; ansonsten riskiert man ein Speicherleck. Es ist also wichtig sicherzustellen, dass die Methode „unsubscribe" aufgerufen wird, wenn die Werte nicht mehr überwacht werden sollen. In Angular wird hierfür gerne die Lifecycle-Methode „ngOnDestroy" verwendet. Eine Ausnahme bilden Observables, bei denen eine Beendigung sichergestellt werden kann. Eine HTTP-Anfrage gehört beispielsweise zu dieser Klasse von Observables, da sie nach der Antwort durch den Server keine weiteren Ergebnisse liefern können und somit abgeschlossen sind.

Operatoren

Bis jetzt sind Observables noch nicht sonderlich spektakulär; erst mit Einführung der Operatoren werden Observables wirklich interessant. Denn neben dem Unterschied zu Promises, dass mehr als nur ein Wert verarbeitet werden kann, wird dem Entwickler mit den Operatoren eine große Werkzeugkiste in die Hand gegeben. Sie ermöglicht es, die Werte eines Observable zu verändern, sie mit anderen Observables zu vereinen, zu filtern, zu limitieren und noch vieles mehr.

Operatoren werden in aller Regel auf Observables ausgeführt und

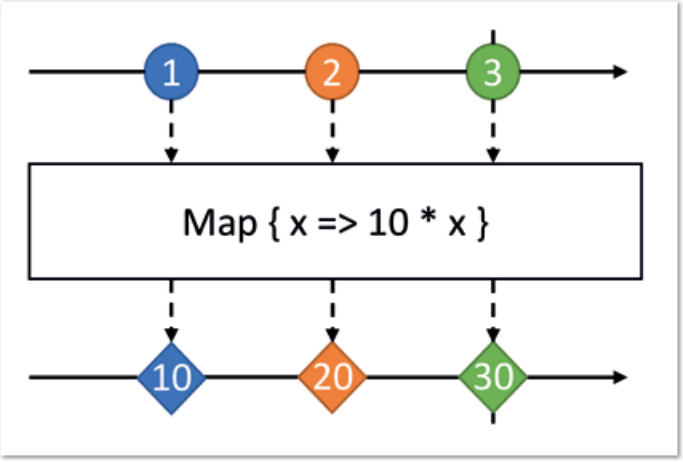

Abbildung 1: Einfaches Marble-Diagramm mit einem Mapping der Werte auf ihr Zehnfaches

geben ein neues Observable zurück. Dadurch können mehrere dieser Aufrufe aneinandergekettet werden. Es entsteht eine Observable-Sequenz, wodurch eine Notation ähnlich wie beim Builder-Pattern entsteht. Nachfolgend stelle ich die verschiedenen Kategorien, in die sich Operatoren aufteilen lassen, mit ihren wichtigsten Vertretern vor.

Marble-Diagramme

Es hat sich etabliert, die Funktionsweise von Observables durch Marble-Diagramme zu veranschaulichen. Ein Marble-Diagramm zeigt für ein einzelnes Observable immer einen Zeitstrahl, auf dem sich bunte „Murmeln" befinden, die die Werte symbolisieren. *Abbildung 1* zeigt ein einfaches Beispiel für ein Marble-Diagramm, das eine Observable-Sequenz darstellt, in der die eingehenden Werte „1:1" in einen anderen Wert konvertiert werden. *Tabelle 1* erklärt die Symbolik von Marble-Diagrammen.

Aus Platzgründen wird in diesem Artikel nicht für jeden hier genannten Operator zusätzlich ein Marble-Diagramm aufgeführt. Eine ausführliche Liste mit Marble-Diagrammen steht auf der Seite *„http://rxmarbles.com"*. Ein Besuch lohnt sich, denn die interaktiven Diagramme erleichtern den Einstieg in RxJS ungemein.

Erstellung

Die folgenden Operatoren erzeugen neue Observables. Sie können in den meisten Fällen statisch auf der Klasse „Observable" aufgerufen werden. „from" erzeugt ein Observable, das jeden Wert eines Arrays einzeln in die Verarbeitungskette reicht. Im obigen Beispiel werden also die Werte 1, 2 und anschließend 3 ausgegeben. Nachdem alle Werte verarbeitet wurden, wird das Observable automatisch „completed" *(siehe Listing 4)*.

„timer/interval" erzeugt ein Observable, das nach einer Verzögerung (etwa 500 ms, *siehe Listing 5)* den Wert „0" in die Sequenz reicht. Wird wie im Beispiel auch der zweite Parameter angegeben, so wird zusätzlich nach dieser Anzahl an Millisekunden je ein weiterer inkrementierter Wert zurückgegeben. Dieses Observable läuft demnach unendlich lang. Es ist also wichtig, dass ein „unsubscribe" die Beendigung dieser Subscription sicherstellt. Das obige Beispiel wird die Werte „0", „1", „2", „3" und so weiter ausgeben, bis die Anwendung beendet ist. Ist lediglich die Funktionalität des zweiten Parameters gewünscht, so kann man dies mit dem Operator „interval" erreichen,

Symbolik	Bedeutung
1	Einzelner ausgestoßener Wert
10	Transformierter Wert in der gleichen Farbe des Quellwerts
	Terminierung der Observable-Sequenz durch einen Fehler (im Beispiel nicht verwendet)
2	Normale Terminierung der Observable-Sequenz ("complete")

Tabelle 1

```
Observable.from([1, 2, 3])
  .subscribe((value) => console.log(value));
```

Listing 4

```
Observable.timer(500, 100)
  .subscribe((value) => console.log(value));
```

Listing 5

```
Observable.from([1, 2, 3])
  .map((x) => x * 10)
  .subscribe((value) => console.log(value));
```

Listing 6

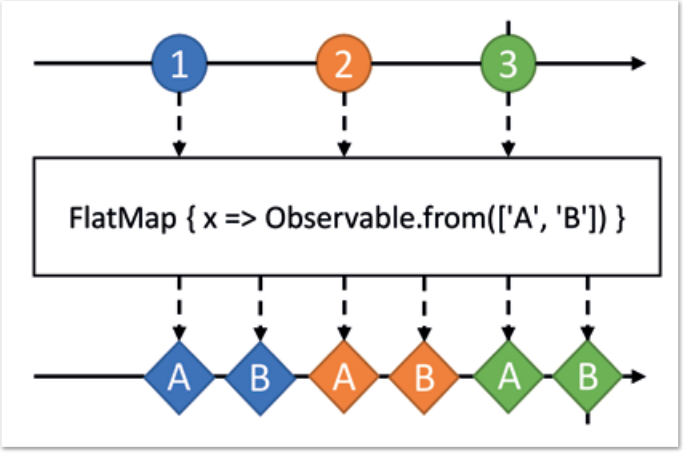

Abbildung 2: Marble-Diagramm zum Codebeispiel des „flatMap"-Operators

```
Observable.from([1, 2, 3])
  .flatMap((value) => Observable.from('A', 'B'))
  .subscribe((value) => console.log(value));
```

Listing 7

```
Observable.timer(500, 100)
  .filter((value) => value % 2 === 0)
  .take(5)
  .subscribe((value) => console.log(value));
```

Listing 8

```
Observable.interval(100)
  .take(5)
  .subscribe((value) => console.log(value));
```

Listing 9

```
const delayed = Observable.timer(1000);

Observable.interval(500)
  .takeUntil(delayed)
  .subscribe((value) => console.log(value));
```

Listing 10

```
const nameObservable = Observable.from(['Jim', 'Johnny',
'Jack']);
const surnameObservable = Observable.from(['Beam',
'Walker', 'Daniels', 'Jameson']);

Observable.zip(nameObservable, surnameObservable)
  .map((values) => `${values[0]} ${values[1]}`)
  .subscribe((value) => console.log(value));
```

Listing 11

der sich abgesehen von dem anfänglichen Delay identisch verhält.

Transformation

Es kommt sehr häufig vor, dass die Werte eines Observable in gänzlich neue Objekte umgewandelt werden müssen. Transformations-Operatoren übernehmen diese Aufgabe und erzeugen basierend auf den Eingangswerten ein oder mehrere neue Objekte für die weitere Sequenz.

„map" transformiert den eingehenden Wert in einen beliebigen anderen Wert. Im *Listing 6* wird der eingehende Wert mit „10" multipliziert. Die Konsole wird also die Werte „10", „20" und „30" anzeigen. Das Marble-Diagramm zu diesem Operator wurde übrigens als Beispiel zur Erklärung von Marble-Diagrammen verwendet.

„flatMap" transformiert einen einfachen Eingangswert in ein Observable, dessen Sequenzwerte anschließend jeweils einzeln in der Sequenz weiterverarbeitet werden *(siehe Abbildung 2)*. Im *Listing 7* wird ein simples Array mit den Werten 1, 2 und 3 mit flatMap aufgerufen. Für jeden dieser Werte wird im flatMap-Callback ein neues Observable zurückgegeben, das die Werte „A" und „B" beinhaltet. Die Ausgabe lautet demnach „A", „B", „A", „B", „A" und „B".

Filter

Filter-Operatoren verringern die Menge an verarbeiteten Werten. Sie können einzelne Werte aus der Sequenz entfernen oder die Sequenz gänzlich beenden. „filter" ist der wohl naheliegendste Operator der Filter-Kategorie. Er begrenzt die Elemente einer Verarbeitungskette anhand einer Funktion, die für jeden Wert einen Wahrheitswert zurückgibt. Ist dieser wahr, wird das Element in der Sequenz weitergereicht; ist er falsch, wird es gefiltert. Im *Listing 8* werden alle ungeraden Werte gefiltert, sodass am Ende nur die geraden Zahlen „0", „2", „4", „6" und „8" ausgegeben werden.

„take" begrenzt die verarbeiteten Werte auf eine übergebene Anzahl. Im *Listing 9* wird das unendliche Intervall, das alle 100 ms einen numerischen Wert liefert, nach fünf Ausführungen beendet. Die Ausgabe ist also auf „0", „1", „2", „3" und „4" begrenzt. Durch diese Begrenzung wird die eigentlich unendliche Subscription endlich und kann so automatisch beendet werden. Der Aufruf von „unsubscribe" ist demnach nicht notwendig, außer es kann nicht sichergestellt werden, dass mindestens fünf Werte durch das Observable erzeugt werden.

Eine andere Variante von „take" stellt „takeUntil" dar, die so lange Werte verarbeitet, bis ein anderes Observable einen Wert liefert. Im *Listing 10* werden also die Werte „0" und „1" nach je 500 ms ausgegeben. Da zu diesem Zeitpunkt eine Sekunde vergangen ist, liefert das zweite Observable einen Wert „0" und beendet somit die Sequenz. Beide Subscriptions sind dann automatisch beendet.

Kombination

Hat man RxJS erst einmal im Einsatz, wird auch schnell die Anforderung entstehen, die Werte zweier Observables zu mischen und in einer gemeinsamen Sequenz zu verarbeiten. Die folgenden beiden Operatoren ermöglichen ein solches Zusammenführen zweier Observables.

Möchte man die Werte von zwei Observables „1:1" miteinander kombinieren, so greift man zum „zip"-Operator. Dieser kann beliebig viele Observables entgegennehmen. Es wartet so lange, bis von allen ein Wert eingetroffen ist, und reicht diese dann in einem Array weiter.

Im *Listing 11* wird „zip" verwendet, um ein Observable von Vornamen mit einem Observable von Nachnamen zu kombinieren. Die Ausgaben sind demnach „Jim Beam", „Johnny Walker" und „Jack

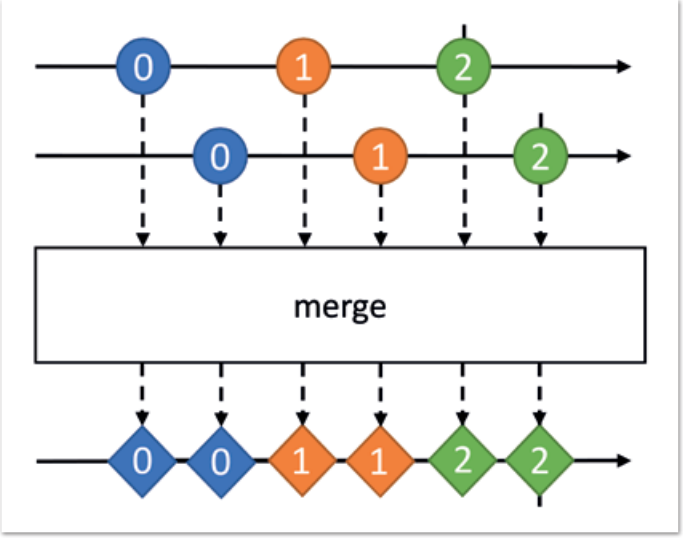

Abbildung 3: Marble-Diagramm zum Codebeispiel des „merge"-Operators

```
Observable.interval(150)
  .merge(Observable.interval(200))
  .take(6)
  .subscribe((value) => console.log(value));
```

Listing 12

```
Observable.from(['Jim', 'Johnny', 'Jack'])
  .do((name) => console.log(name))
  .subscribe();
```

Listing 13

```
Observable.interval(100)
  .delay(300)
  .take(5)
  .subscribe((value) => console.log(value));
```

Listing 14

Daniels". „Jameson" wird nicht ausgegeben, da hier der Vorname fehlt.

„merge" erweitert die Werte eines Observable um die Werte eines anderen. Im Gegensatz zu „zip" wartet „merge" nicht auf die Werte des anderen Observable; die Werte werden sofort weitergereicht, egal von welchem Observable sie stammen (siehe *Abbildung 3* und *Listing 12*).

„Utility" sind praktische Hilfs-Operatoren, die keiner der anderen Kategorien genau zugeordnet werden konnten. „do" ist ein sehr praktischer Operator für Seiteneffekte. Er führt lediglich eine Funktion aus, ohne die Werte der Sequenz in irgendeiner Art zu verändern. Im *Listing 13* wurde die Konsolen-Ausgabe von der „subscribe"-Methode in das „do" verlagert, was keinen spürbaren Unterschied macht.

„delay" verzögert die Weitergabe eines Werts um die übergebene Anzahl an Millisekunden und reicht anschließend alle bis dahin aufgetretenen Werte weiter. Nach Ablauf des Delay gehen alle folgenden Werte sofort weiter. *Listing 14* zeigt nach 300 ms die Werte „0", „1", „2" und „3" und nach weiteren 100 ms den Wert „4" an.

Fehlerbehandlung

Neben der Observable-weiten Fehlerbehandlung der „subscribe"-Methode existieren auch mehrere Operatoren, die auf vorhergehende Errors in der Observable-Sequenz reagieren. Operatoren dieser Kategorie ermöglichen eine Wiederaufnahme der Sequenz. Der Error-Handler der „subscribe"-Methode wird also unter Umständen nicht benachrichtigt.

Eine Fehlerbehandlung durch den Error-Handler der „subscribe"-Methode (zweiter Parameter) hat nicht die Möglichkeit, die Subscription weiter aufrechtzuerhalten. Nach der Ausführung der Fehlerbehandlung ist die Subscription zwangsweise beendet, obwohl das Observable möglicherweise noch weitere Werte liefern könnte.

Anders verhält sich der „catch"-Operator. Dieser ruft die übergebene Funktion auf, sofern irgendwo vor ihm in der Sequenz ein Fehler auftreten ist. Der Rückgabewert der Funktion wird anschließend in der weiteren Sequenz verarbeitet. Somit gibt das Beispiel den Wert „fixed it!" mehrfach zurück, obwohl der Wert ursprünglich ein numerischer Zähler des Operators „interval" war *(siehe Listing 15)*.

Vor allem bei Netzwerkanfragen kann es vorkommen, dass fehlgeschlagene Anfragen wiederholt werden müssen. Für diese Aufgabe

```
Observable.interval(100)
  .do(() => {throw Error('something went wrong')})
  .catch((error) => 'fixed it!')
  .subscribe((value) => console.log(value));
```

Listing 15

```
Observable.from(['Jim', 'Johnny', 'Jack'])
  .do((name) => console.log(name))
  .do(() => {throw Error('something went wrong')})
  .retry(2)
  .subscribe((value) => console.log(value));
```

Listing 16

eignet sich der „retry"-Operator. Er wiederholt im Fehlerfall die vorhergehende Sequenz, bis diese erfolgreich ist oder die übergebene Anzahl an Versuchen erreicht wurde. Das *Listing 16* gibt also dreimal „Jim" und anschließend einen Stacktrace des Fehlers aus. „Johnny" und „Jack" kommen gar nicht erst zum Zuge.

Fazit

RxJS ist eine hochinteressante Bibliothek, die – gut in die Anwendung integriert – ein mächtiges Werkzeug darstellt. Angular hat hier bereits gute Arbeit geleistet. Die HTTP-Kommunikation, das Data Binding und die Eventbehandlung setzen auf Observables. Hat man die Funktionsweise erst einmal richtig verstanden, gelingt einem das Umdenken von Promises auf Observables recht schnell.

Weiterführende Links

- *http://reactivex.io/rxjs*
- *https://www.learnrxjs.io*
- *http://rxmarbles.com*
- *https://channel9.msdn.com/blogs/j.van.gogh/reactive-extensions-api-in-depth-marble-diagrams-select–where*

Michael Ruttka
michael.ruttka@buschmais.com

Michael Ruttka ist Consultant bei der buschmais GbR. Seine fachlichen Schwerpunkte liegen in der Entwicklung und Konzeption von Web-Anwendungen.