RxJs 3/4
ReactiveX for Javascript

1 Observables kombinieren

Sehr häufig macht eine Fortsetzung des Programms erst Sinn, wenn Daten aus **mehreren** Quellen vorhanden sind. D.h. man muss mehrere Observables gleichzeitig behandeln. Dazu gibt es wieder eine große Anzahl von Funktionen, wie man diese verbindet, z.B. Webrequest bei Buttonklick.

1.1 debounce + throttle

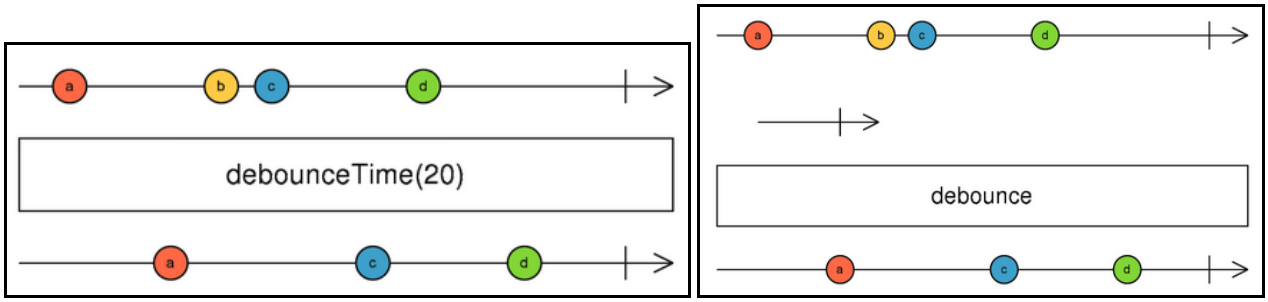
Die Funktionen **`debounce`**() und **`throttle`**() sorgen dafür, dass bei einer Vielzahl von Werten (meist handelt es sich um Events) nicht alle zum Subscriber durchgereicht werden, sondern diese entsprechend reduziert werden.
Den Unterschied kann man sich in etwa so vorstellen:
Man chattet mit einem Freund über WhatsApp, der alle 5 Sekunden eine neue Nachricht schickt. Dann hat man folgende Möglichkeiten:

- **ohne** throttle/debounce: man liest immer sofort jede Nachricht. Das ist zeitaufwändig, aber man ist immer auf dem aktuellen Stand.
- **throttle**: Falls neue Nachrichten angekommen sind (sieht man am Notification-Icon), liest man alle 5 Minuten die Nachricht.
- **debounce**: man liest prinzipiell einmal keine Nachricht und lässt den Freund seine in viele Einzelteile zerhackte Geschichte fertig erzählen. Erst wenn er 5 Minuten nichts mehr schickt, schaut man nach.

Oder bei MouseMove-Events:

- **debounce**: sendet prinzipiell einmal keine Events. Erst wenn die Maus eine bestimmte Zeit nicht bewegt wurde, wird das Event durchgereicht.
- **throttle**: sendet nicht alle Mouse-Events, sondern nur zu bestimmten Zeitpunkten genau einen Wert.
- Demo: http://demo.nimius.net/debounce_throttle/

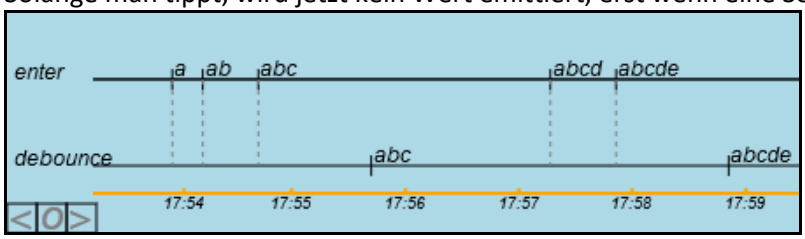
1.1.1 debounce/debounceTime



Verzögert das Emittieren von Werten auf eine Mindest-Wartezeit. Es wird also sichergestellt, dass innerhalb der angegebenen Zeitspanne (in Millisekunden) „Ruhe herrscht“, damit danach ein Wert emittiert wird. Das ist bei Eingabefeldern oft sinnvoll, wo man nicht sofort auf jeden Tastendruck reagieren will.

```
const txtInput = document.querySelector('#txtInput');
Rx.fromEvent(txtInput, 'keyup')
  .pipe(
    map(x => x.currentTarget.value),
    draw(0, 'enter'),
    debounceTime(1000),
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '---->'));
```

Solange man tippt, wird jetzt kein Wert emittiert, erst wenn eine Sekunde keine Eingabe erfolgt, wird ausgelöst.

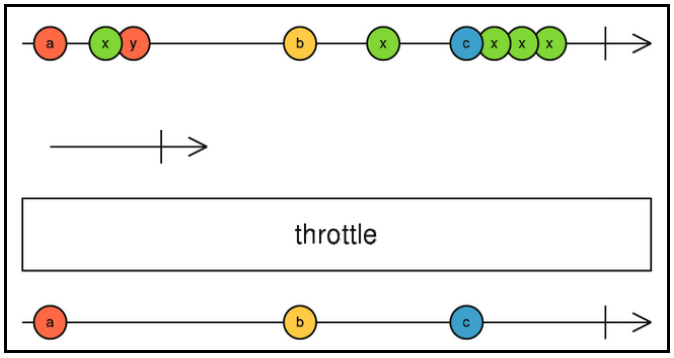


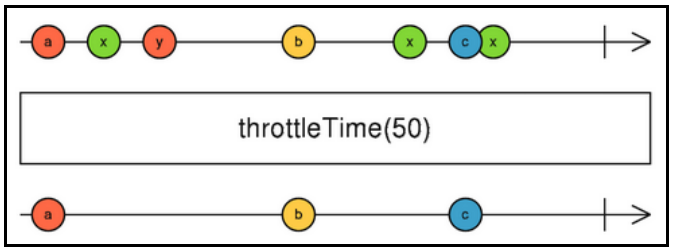

```
19:17:53.936  enter a
19:17:54.209  enter ab
19:17:54.703  enter abc
19:17:55.709  ----> abc
19:17:57.304  enter abcd
19:17:57.890  enter abcde
19:17:58.893  ----> abcde
```

Anstelle einer fixen Zeit kann man auch ein Observable angeben, das diese Verzögerungszeiten liefert:

```
Rx.fromEvent(txtInput, 'keyup')
  .pipe(
    map(x => x.currentTarget.value),
    draw(0, 'enter'),
    debounce(_ => Rx.interval(1000)),
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '---->'));
```

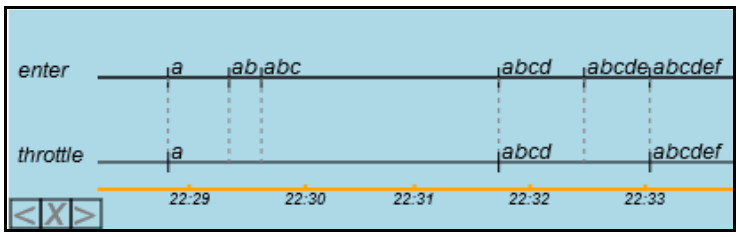
1.1.2 throttle/throttleTime





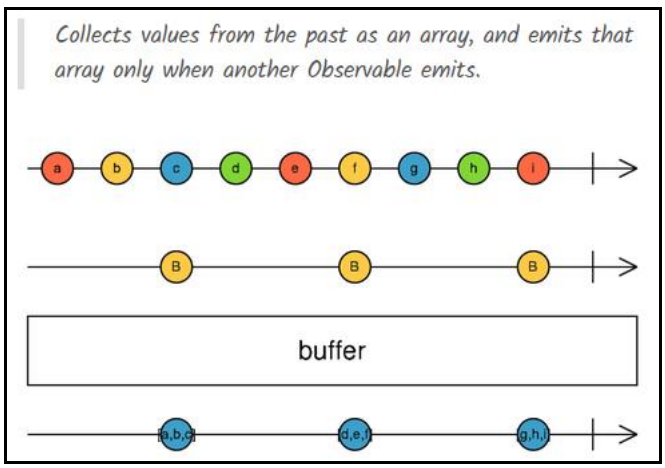
Sendet nach einer bestimmten Zeit einen Wert, wenn in der Zwischenzeit einer oder mehrere Werte angefallen sind.

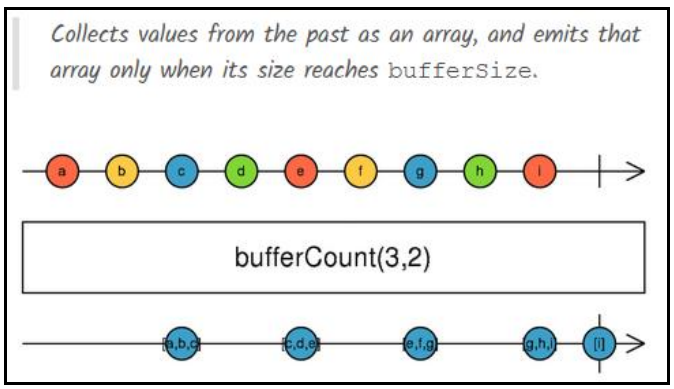
```
const txtInput = document.querySelector('#txtInput');
Rx.fromEvent(txtInput, 'keyup')
  .pipe(
    map(x => x.currentTarget.value),
    draw(0, 'enter'),
    throttleTime(1000),
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '---->'));
```




```
19:22:28.843  enter a
19:22:28.845  ----> a
19:22:29.378  enter ab
19:22:29.663  enter abc
19:22:31.750  enter abcd
19:22:31.750  ----> abcd
19:22:32.501  enter abcde
19:22:33.078  enter abcdef
19:22:33.080  ----> abcdef
```

1.2 buffer

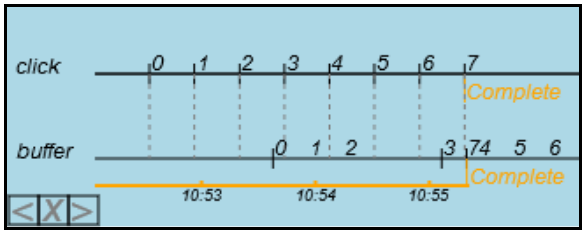




1.2.1 bufferTime

bufferTime() funktioniert ähnlich wie **debounce**(), jedoch werden die vorherigen Werte nicht verworfen, sondern in einem Array gesammelt.

```
RxJsVisualizer.prepareCanvas(['enter', 'buffer']);
Rx.interval(400)
  .pipe(
    take(8),
    draw(0, '', true),
    bufferTime(1500)
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '', true));
```

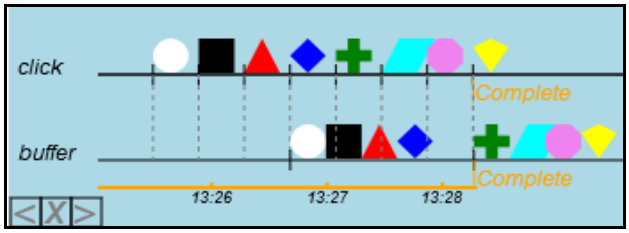



```
12:10:52.544   0
12:10:52.942   1
12:10:53.346   2
12:10:53.643   0,1,2
12:10:53.745   3
12:10:54.145   4
12:10:54.544   5
12:10:54.945   6
12:10:55.143   3,4,5,6
12:10:55.347   7
12:10:55.348   Completed
12:10:55.363   7
12:10:55.366   Completed
```

1.2.2 bufferCount

Bei **bufferCount**() wird immer dann emittiert, wenn sich eine bestimmte Anzahl von Werten angehäuft hat:

```
Rx.interval(400)
  .pipe(
    take(8),
    draw(0, ''),
    bufferCount(4),
  )
  .subscribe(RxJsVisualizer.observerForLine(1, ''));
```

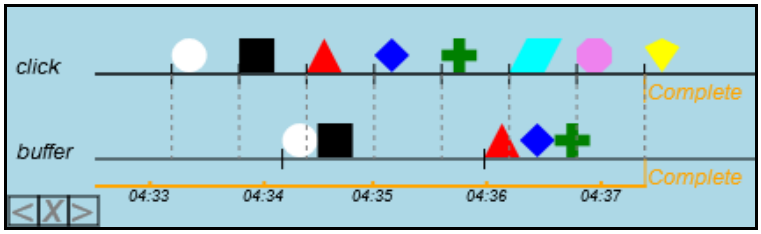



```
12:13:25.514   0
12:13:25.914   1
12:13:26.314   2
12:13:26.715   3
12:13:26.717   0,1,2,3
12:13:27.117   4
12:13:27.516   5
12:13:27.916   6
12:13:28.318   7
12:13:28.319   Completed
12:13:28.326   4,5,6,7
12:13:28.329   Completed
```

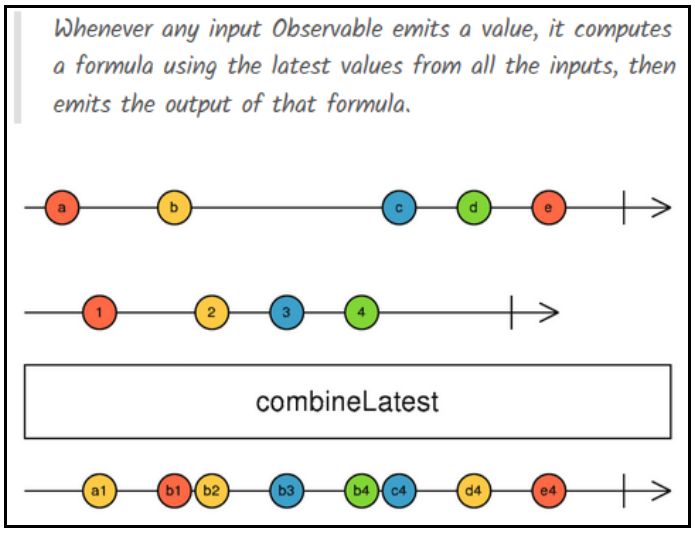
1.2.3 buffer

Bei **buffer**() wird nicht eine bestimmte Zeitdauer gewartet, sondern es werden bei Auftreten eines Wertes eines anderen Observables die gesammelten Werte geliefert.

```
Rx.interval(600).pipe(take(8))
  .pipe(
    draw(0),
    buffer(Rx.fromEvent(document.querySelector('#btnCollect'), 'click'))
  )
  .subscribe(RxJsVisualizer.observerForLine(1));
```

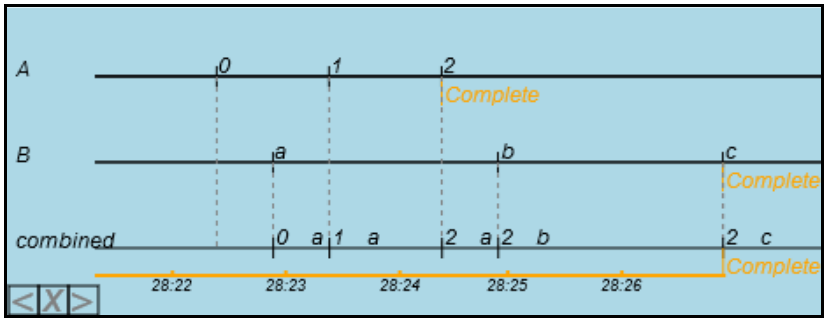


1.3 combineLatest

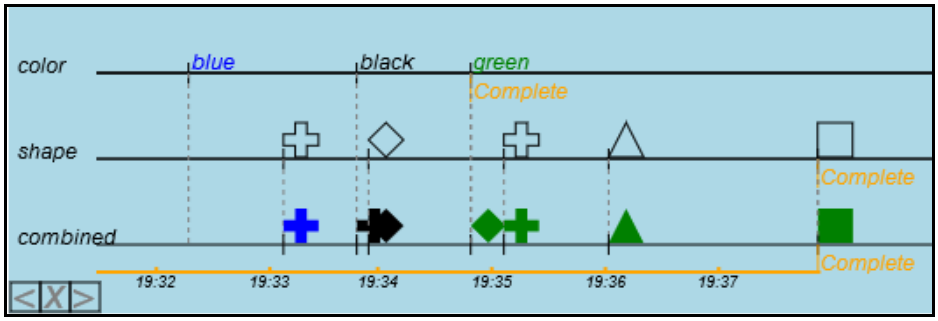


Aus zwei oder mehreren Observables werden bei Auftreten eines Wertes die jeweils aktuellen Werte eines jeden Observable genauso emittiert.

```
RxJsVisualizer.prepareCanvas(['A', 'B', 'combined']);
const items = ['a', 'b', 'c', 'd'];
const timerA = Rx.timer(1000, 1000).pipe(take(3));
const timerB = Rx.timer(1500, 2000).pipe(map(x => items[x]), take(3));
timerA.subscribe(RxJsVisualizer.observerForLine(0, 'timerA', true));
timerB.subscribe(RxJsVisualizer.observerForLine(1, 'timerB', true));
Rx.combineLatest(timerA, timerB)
  .subscribe(RxJsVisualizer.observerForLine(2, 'combined', true));
```




```
12:28:22.418  timerA 0
12:28:22.919  timerB a
12:28:22.922  combined 0,a
12:28:23.418  timerA 1
12:28:23.422  combined 1,a
12:28:24.420  timerA 2
12:28:24.421  Completed timerA
12:28:24.423  combined 2,a
12:28:24.921  timerB b
12:28:24.924  combined 2,b
12:28:26.922  timerB c
12:28:26.922  Completed timerB
12:28:26.924  combined 2,c
12:28:26.925  Completed combined
```



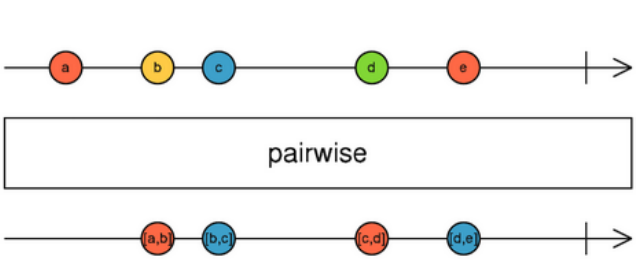
Als zusätzlichen Parameter kann man eine Kombinationsfunktion angeben:

```
RxJsVisualizer.prepareCanvas(['A', 'B', 'combined']);
const timerA = Rx.timer(1000, 1000).pipe(take(3));
const timerB = Rx.timer(1500, 2000).pipe(take(3));
timerA.subscribe(RxJsVisualizer.observerForLine(0, 'timerA', true));
timerB.subscribe(RxJsVisualizer.observerForLine(1, 'timerB', true));
Rx.combineLatest(timerA, timerB, (a, b) => `${a}+${b}=${a + b}`)
  .subscribe(RxJsVisualizer.observerForLine(2, 'combined', true));
```

```
12:33:38.252  timerA 0
12:33:38.753  timerB 0
12:33:38.756  combined 0+0=0
12:33:39.251  timerA 1
12:33:39.257  combined 1+0=1
12:33:40.252  timerA 2
12:33:40.252  Completed timerA
12:33:40.257  combined 2+0=2
12:33:40.754  timerB 1
12:33:40.759  combined 2+1=3
12:33:42.756  timerB 2
12:33:42.756  Completed timerB
12:33:42.760  combined 2+2=4
12:33:42.761  Completed combined
```

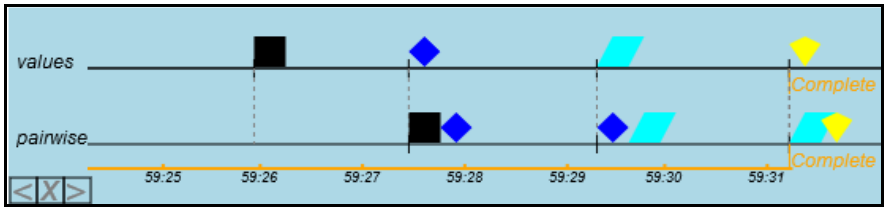
1.4 pairwise

> *Puts the current value and previous value together as an array, and emits that.*



pairwise() funktioniert ähnlich wie **combineLatest**(), jedoch wird aus einem Observable der aktuelle und der vorherige Wert als Array geliefert.

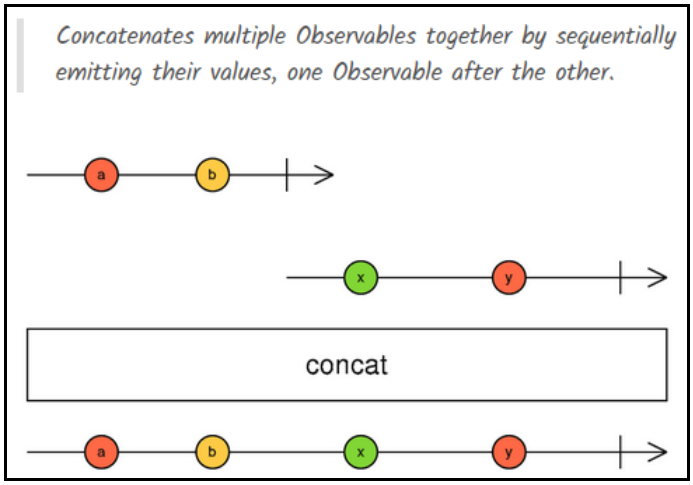
```
RxJsVisualizer.prepareCanvas(['values', 'pairwise']);
RxJsVisualizer.createStreamFromArraySequence([1, 3, 5, 7], 1500, 2000)
  .pipe(
    draw(0, 'values'),
    pairwise())
  .subscribe(RxJsVisualizer.observerForLine(1, 'pairwise'));
```




```
17:59:25.947  values 1
17:59:27.478  values 3
17:59:27.480  pairwise 1,3
17:59:29.340  values 5
17:59:29.342  pairwise 3,5
17:59:31.243  values 7
17:59:31.244  pairwise 5,7
17:59:31.245  Completed values
17:59:31.246  Completed pairwise
```

Man beachte, dass bei Emission des ersten Wertes noch kein **pairwise**() ausgelöst wird, sondern erst wenn das erste Paar zur Verfügung steht, also bei 3.

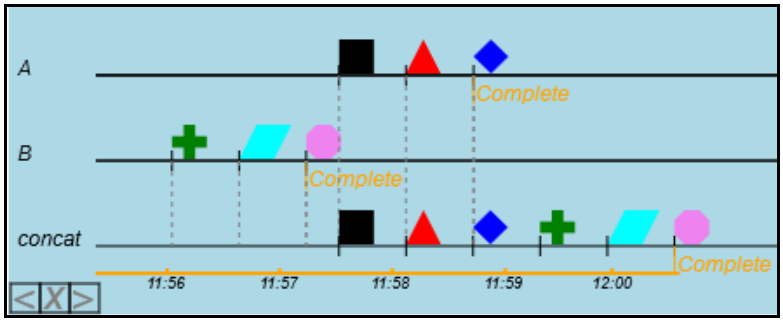
1.5 concat



Die Werte zwei oder mehrerer Observables werden sequentiell verknüpft und in der richtigen Reihenfolge emittiert. Es wird aber abgewartet, bis **das erste Observable completed** ist. Das zweite Observable startet also nicht, bevor das erste beendet ist.

```
const valuesA = RxJsVisualizer.createStreamFromArraySequence([1, 2, 3], 600, 600)
  .pipe(delay(1500));
const valuesB = RxJsVisualizer.createStreamFromArraySequence([4, 5, 6], 600, 600);
valuesA.subscribe(RxJsVisualizer.observerForLine(0, 'A'));
valuesB.subscribe(RxJsVisualizer.observerForLine(1, 'B'));
Rx.concat(valuesA, valuesB).subscribe(RxJsVisualizer.observerForLine(2, 'concat'));
```

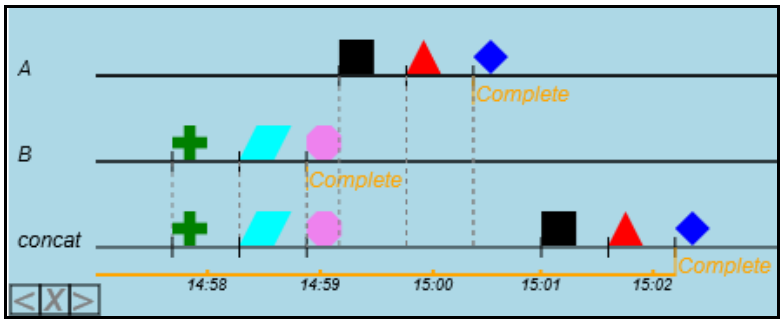
Es wird sichergestellt, dass die Reihenfolge (zuerst A, dann B) eingehalten wird.




```
18:11:56.051  B 4
18:11:56.653  B 5
18:11:57.256  B 6
18:11:57.256  Completed B
18:11:57.551  A 1
18:11:57.551  concat 1
18:11:58.151  A 2
18:11:58.153  concat 2
18:11:58.754  concat 3
18:11:58.758  A 3
18:11:58.760  Completed A
18:11:59.356  concat 4
18:11:59.959  concat 5
18:12:00.562  concat 6
18:12:00.564  Completed concat
```

Die Reihenfolge ist wichtig, es wird die Reihenfolge beibehalten, und sobald wie möglich emittiert:

```
Rx.concat(valuesB, valuesA).subscribe(RxJsVisualizer.observerForLine(2, 'concat'));
```



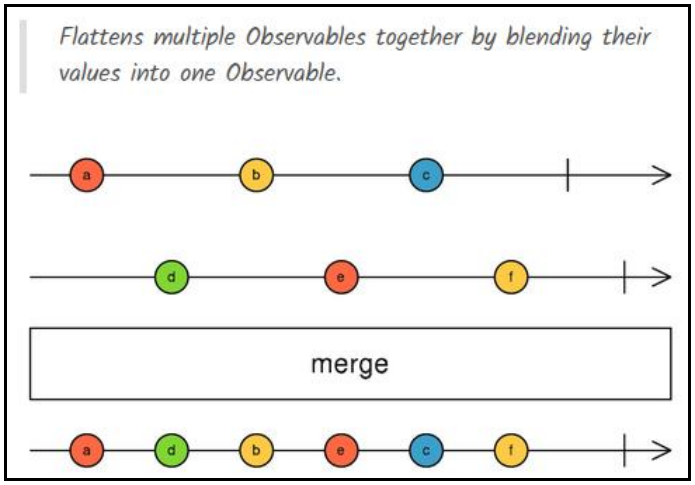

```
18:14:57.703  B 4
18:14:57.703  concat 4
18:14:58.305  B 5
18:14:58.305  concat 5
18:14:58.908  B 6
18:14:58.910  concat 6
18:14:58.912  Completed B
18:14:59.201  A 1
18:14:59.804  A 2
18:15:00.405  A 3
18:15:00.405  Completed A
18:15:01.016  concat 1
18:15:01.617  concat 2
18:15:02.217  concat 3
18:15:02.219  Completed concat
```

2.1 s

Completed B: 14:58.9

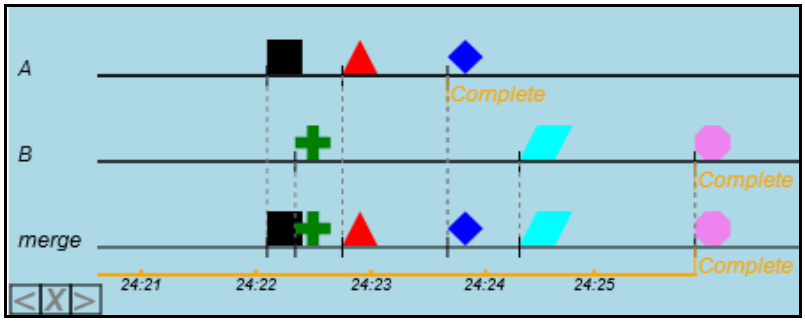
concat 1: 15:01.0 (=14:58.9 + 1.5 start delay + 0.6 emit delay)

1.6 merge



Im Gegensatz zu **concat**() wird bei **merge**() die flache Struktur mit Werten in jener Reihenfolge erzeugt, in der sie anfallen, egal welches Observable es emittiert.

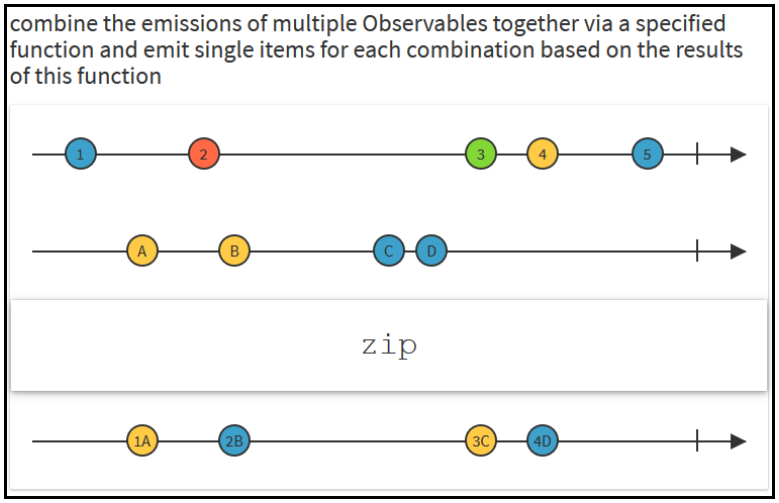
```
RxJsVisualizer.prepareCanvas(['A', 'B', 'merge']);
const valuesA = RxJsVisualizer.createStreamFromArraySequence([1, 2, 3]);
const valuesB = RxJsVisualizer.createStreamFromArraySequence([4, 5, 6]);
valuesA.subscribe(RxJsVisualizer.observerForLine(0, 'A'));
valuesB.subscribe(RxJsVisualizer.observerForLine(1, 'B'));
Rx.merge(valuesA, valuesB).subscribe(RxJsVisualizer.observerForLine(2, 'merge'));
```




```
18:24:22.112  A 1
18:24:22.114  merge 1
18:24:22.362  B 4
18:24:22.363  merge 4
18:24:22.778  A 2
18:24:22.778  merge 2
18:24:23.707  A 3
18:24:23.707  merge 3
18:24:23.707  Completed A
18:24:24.342  B 5
18:24:24.342  merge 5
18:24:25.890  B 6
18:24:25.891  merge 6
18:24:25.892  Completed B
18:24:25.892  Completed merge
```

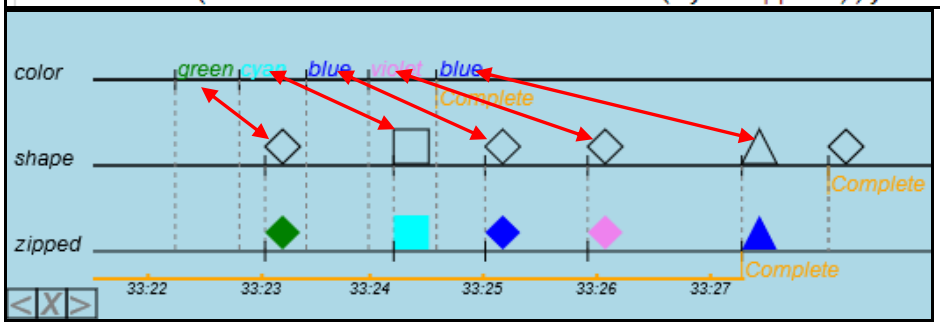
Auch hier kann man wieder mehr als zwei Observables kombinieren.

1.7 zip



Es werden die Daten aus den Streams im Reißverschlusssystem kombiniert. Immer dann also, wenn bei jedem Stream ein Element vorhanden, wird eine Kombination emittiert.

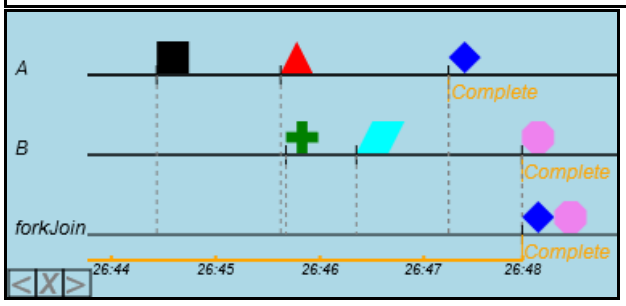
```
RxJsVisualizer.prepareCanvas(['color', 'shape', 'zipped']);
const obsColors = RxJsVisualizer.createStreamFromArrayRandom(colors, RxJsVisualizer.rnd(3, 6));
const obsLogos = RxJsVisualizer.createStreamFromArrayRandom(shapes, RxJsVisualizer.rnd(3, 6));
obsColors.subscribe(RxJsVisualizer.observerForLine(0, 'color'));
obsLogos.subscribe(RxJsVisualizer.observerForLine(1, 'shape'));
Rx.zip(obsColors, obsLogos, (a, b) => `${a} ${b}`)
  .subscribe(RxJsVisualizer.observerForLine(2, 'zipped'));
```



1.8 forkJoin

Diese Methode macht Sinn, wenn man an mehreren Observables interessiert ist, jedoch nur dann, **wenn alle beendet sind** und man nur den letzten Wert braucht. Es funktioniert also nur mit Observables, die allesamt das completed event auslösen!

```
RxJsVisualizer.prepareCanvas(['A', 'B', 'forkJoin']);
const valuesA = RxJsVisualizer.createStreamFromArraySequence([1, 2, 3]);
const valuesB = RxJsVisualizer.createStreamFromArraySequence([4, 5, 6]);
valuesA.subscribe(RxJsVisualizer.observerForLine(0, 'A'));
valuesB.subscribe(RxJsVisualizer.observerForLine(1, 'B'));
Rx.forkJoin(valuesA, valuesB).subscribe(RxJsVisualizer.observerForLine(2, 'merge'));
```




```
18:26:44.453  A 1
18:26:45.657  A 2
18:26:45.708  B 4
18:26:46.393  B 5
18:26:47.288  A 3
18:26:47.289  Completed A
18:26:48.002  B 6
18:26:48.003  Completed B
18:26:48.004  merge 3,6
18:26:48.004  Completed merge
```

Das Ergebnis liegt als Array vor.

1.8.1 Webrequest

Sehr brauchbar ist dieser Operator, wenn man parallel mehrere Webrequests durchführt und alle gemeinsam abwarten möchte.
Angenommen man möchte folgende drei Requests gemeinsam behandeln:

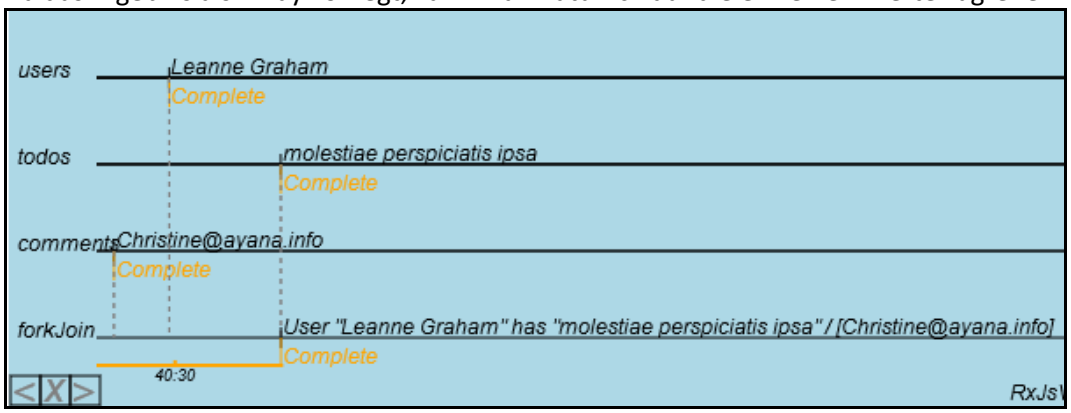
- https://jsonplaceholder.typicode.com/users/1
- https://jsonplaceholder.typicode.com/todos/9
- https://jsonplaceholder.typicode.com/comments/16

Dann könnte man das mit folgendem Code erreichen:

```
function requestHttp(resource, id, prop, lineNr, offset) {
  return Rx.from(
    fetch(`https://jsonplaceholder.typicode.com/${resource}/${id}`)
    .then(x => x.json())
  )
  .pipe(
    delay(offset),
    map(x => x[prop]),
    draw(lineNr)
  );
}
```

```
RxJsVisualizer.prepareCanvas(['users', 'todos', 'comments', 'forkJoin']);
const httpA = requestHttp('users', 1, 'name', 0, 500);
const httpB = requestHttp('todos', 9, 'title', 1, 1500);
const httpC = requestHttp('comments', 16, 'email', 2, 0);
Rx.forkJoin(httpA, httpB, httpC)
  .pipe(map(x => `User "${x[0]}" has "${x[1]}" / [${x[2]}]`))
  .subscribe(RxJsVisualizer.observerForLine(3));
```

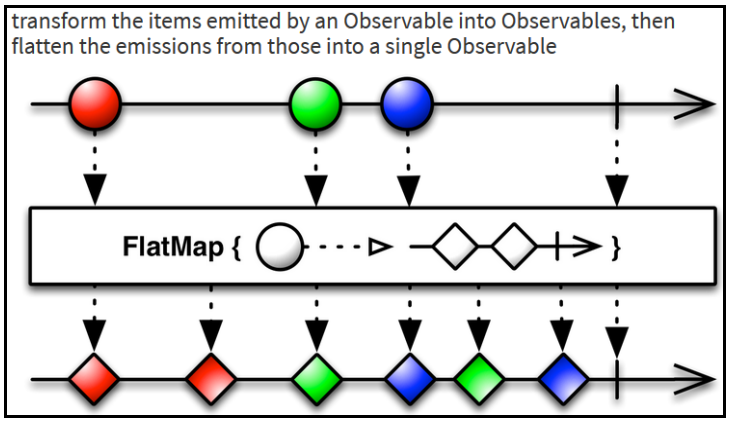
Da das Ergebnis als Array vorliegt, kann man natürlich auf die einzelnen Werte zugreifen:



1.9 flatMap/switchMap

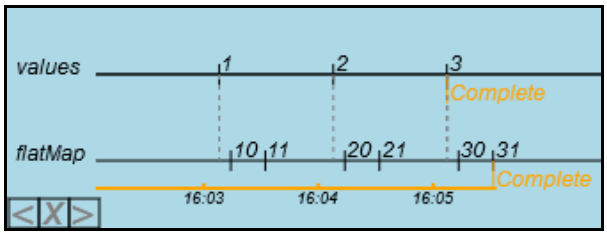
Die Operatoren flatMap und switchMap verhalten sich sehr ähnlich.

1.9.1 flatMap



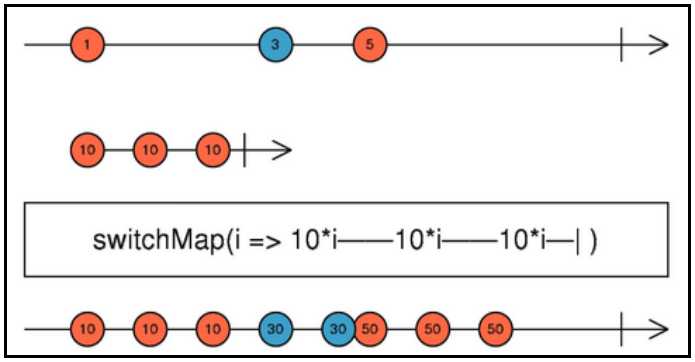
`flatMap`() erzeugt aus Werten eines Observables weitere Observables und kombiniert diese Teilsequenzen mittels merge zu einer Gesamtsequenz.
Es entspricht in etwa dem LINQ-Operator **`SelectMany`**, der aus einer Liste von Listen eine einzige zusammenhängende Liste erzeugt.

```
RxJsVisualizer.prepareCanvas(['values', 'flatMap']);
RxJsVisualizer.createStreamFromArraySequence([1, 2, 3], 1000, 1000)
  .pipe(
    draw(0, '', true),
    flatMap(x => Rx.timer(100, 300).pipe(
      take(2),
      map(y => `${x}${y}`))))
  .subscribe(RxJsVisualizer.observerForLine(1, 'flatMap', true));
```



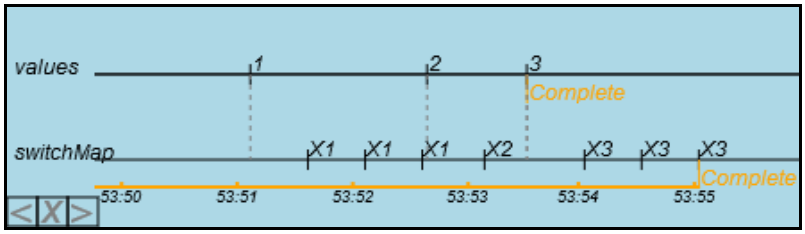

```
20:16:03.137   1
20:16:03.241  flatMap 10
20:16:03.544  flatMap 11
20:16:04.142   2
20:16:04.244  flatMap 20
20:16:04.548  flatMap 21
20:16:05.142   3
20:16:05.143  Completed
20:16:05.245  flatMap 30
20:16:05.548  flatMap 31
20:16:05.550  Completed flatMap
```

1.9.2 switchMap



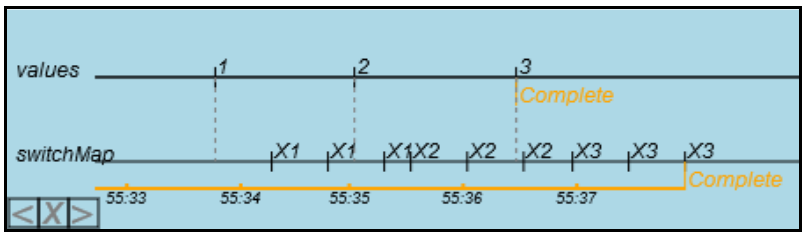
switchMap() funktioniert im Prinzip wie **flatMap**(), jedoch wird das Emittieren des inneren Observables unterbrochen, sobald ein neuer Wert des äußeren Observables auftritt. In obigem Diagramm fehlt also ein „blauer 30er“ im Vergleich zu flatMap, weil eben zu diesem Zeitpunkt ein 5er emittiert wurde.

```
RxJsVisualizer.prepareCanvas(['values', 'switchMap']);
RxJsVisualizer.createStreamFromArraySequence([1, 2, 3])
  .pipe(
    draw(0, '', true),
    switchMap(x => Rx.interval(500).pipe(map(y => `X${x}`), take(3))))
  .subscribe(RxJsVisualizer.observerForLine(1, 'flatMap'));
```




```
18:53:51.134   1
18:53:51.638  flatMap X1
18:53:52.139  flatMap X1
18:53:52.639  flatMap X1
18:53:52.681   2
18:53:53.183  flatMap X2
18:53:53.554   3
18:53:53.556  Completed
18:53:54.059  flatMap X3
18:53:54.558  flatMap X3
18:53:55.059  flatMap X3
18:53:55.060  Completed flatMap
```

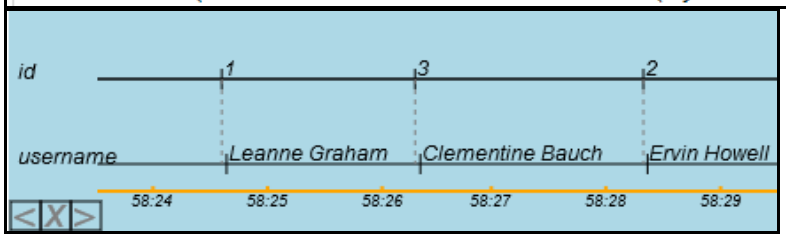
Zum Vergleich flatMap:



1.9.3 Webrequest

Das braucht man üblicherweise bei einem WebRequest, der bei Button-Klick oder bei Eingabe in ein Textfeld (meist mit debounceTime) ausgelöst werden soll. Man bekommt ja bei einem Web-Aufruf ein Observable mit dem Response zurück. Das ist zwar nur ein Wert, aber ist trotzdem ein Observable.

```
RxJsVisualizer.prepareCanvas(['id', 'username']);
Rx.fromEvent(document.querySelector('#txtInputId'), 'keyup')
  .pipe(
    map(x => x.currentTarget.value),
    filter(x => x.length > 0),
    draw(0, 'id', true),
    map(id => `https://jsonplaceholder.typicode.com/users/${id}`),
    flatMap(url => Rx.from(
      fetch(url).then(x => x.json())
    )),
    map(x => x.name)
  )
  .subscribe(RxJsVisualizer.observerForLine(1, 'username'));
```




```
18:58:24.623  id 1
18:58:24.666  username Leanne Graham
18:58:26.327  id 3
18:58:26.373  username Clementine Bauch
18:58:28.340  id 2
18:58:28.377  username Ervin Howell
```

2 Best Practices

Jetzt sollen noch ein paar Anwendungen besprochen werden, die immer wieder in ähnlicher Form auftreten.

2.1 Sequenz

Ein Standardproblem ist, dass man mehrere Web-Aufrufe hintereinander braucht, wobei jeder Aufruf Daten des vorherigen Aufrufs benötigt.
Beispiel:

1. Mit einem Username liest man ein User-Objekt
2. Aus dem User-Objekt erhält man die UserId, mit der man sich die Orders holt
3. Aus der OrderId der ersten Order liest man schließlich die OrderDetails
4. Die OrderDetails werden angezeigt.

Das löst man üblicherweise mit einer switchMap-Sequenz, bei der man evtl. noch die einzelnen Daten mit map() auflöst.

```
readUserByName('hhuber') //a single User
  .pipe(
    map(x => x.userId),
    switchMap(x => readOrdersOfUser(x)), //Array of Orders
    map(x => x[0]),
    map(x => x.orderId),
    switchMap(x => readOrderDetailsOfOrder(x)) //Array of OrderDetails
  )
  .subscribe(x => x.forEach(y => console.log(`${y.amount} x ${y.product}`)));
```

```
09:38:24.303  4 x Chai
09:38:24.305  10 x Tofu
09:38:24.306  2 x Ikura
```

2.2 Sequenz mit Zwischenergebnissen

Wenn man auch die Zwischenergebnisse braucht, speichert man sich diese mit tap() in lokalen Variablen. Diese sind dann im Subscribe verfügbar und zu dem Zeitpunkt dann auch sicher zugewiesen.

```
let currentUser = {};
let currentOrders = [];
let currentOrder = {};
readUserByName('hhuber') //a single User
  .pipe(
    tap(x => currentUser = x),
    map(x => x.userId),
    switchMap(x => readOrdersOfUser(x)), //Array of Orders
    tap(x => currentOrders = x),
    map(x => x[0]),
    tap(x => currentOrder = x),
    map(x => x.orderId),
    switchMap(x => readOrderDetailsOfOrder(x)) //Array of OrderDetails
  )
  .subscribe(x => {
    console.log(`Nr of orders of ${currentUser.lastname}: ${currentOrders.length}`);
    const date = currentOrder.orderDate;
    console.log(`order at ${date.getDate()}.${date.getMonth()+1}.${date.getFullYear()}`);
    x.forEach(y => console.log(`${y.amount} x ${y.product}`));
  });
```

```
09:52:20.188  Nr of orders of Huber: 3
09:52:20.190  order at 21.3.2019
09:52:20.191  4 x Chai
09:52:20.192  10 x Tofu
09:52:20.192  2 x Ikura
```

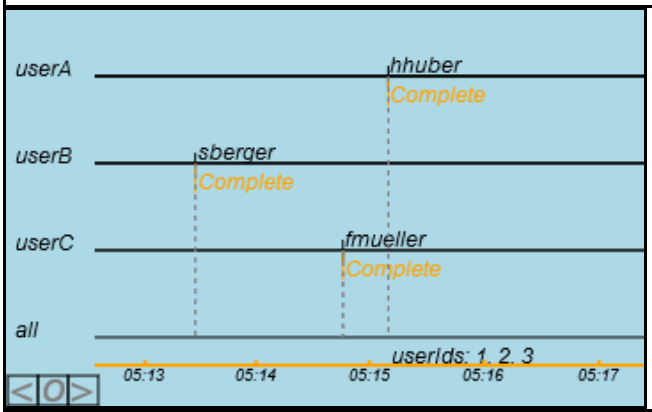
2.3 Parallel lesen, Gesamtergebnis abwarten

Hier ist forkJoin() die richtige Funktion, aber **nur wenn man sicher ist, dass die Observables completed werden**. Das ist bei HTTP-requests aber der Fall.

```
Rx.forkJoin(
    readUserByName('hhuber', 2500),
    readUserByName('sberger', 800),
    readUserByName('fmueller', 2100)
  )
  .subscribe(
    x => {
      const [huber, berger, mueller] = x;
      console.log(`userIds: ${huber.userId}, ${berger.userId}, ${mueller.userId}`);
    }
  );
```

Mit Visualisierung sieht es so aus:

```
RxJsVisualizer.prepareCanvas(['userA', 'userB', 'userC', 'all']);
Rx.forkJoin(
    readUserByName('hhuber', 2500).pipe(draw(0, 'userA', true, x => x.username)),
    readUserByName('sberger', 800).pipe(draw(1, 'userB', true, x => x.username)),
    readUserByName('fmueller', 2100).pipe(draw(2, 'userC', true, x => x.username))
  )
  .subscribe(
    x => {
      const [huber, berger, mueller] = x;
      RxJsVisualizer.writeToLine(3, `userIds: ${huber.userId}, ${berger.userId}, ${mueller.userId}`);
    }
  );
```




```
17:05:13.476  userB {"username":"sberger","userId":2,"firstname":"Susi","lastname":"Berger"}
17:05:13.477  Completed userB
17:05:14.776  userC {"username":"fmueller","userId":3,"firstname":"Fritzi","lastname":"Müller"}
17:05:14.777  Completed userC
17:05:15.176  userA {"username":"hhuber","userId":1,"firstname":"Hansi","lastname":"Huber"}
17:05:15.176  Completed userA
17:05:15.178  userIds: 1, 2, 3
```