RxJava Applied

Pre Java 8 data processing

```
List<Employee> employees = service.getEmployees();

Map<Integer, List<Employee>> ageDistribution = new HashMap<>();
for (Employee employee : employees) {
   if (employee.getAge() > 25){
       List<Employee> thisAge = ageDistribution.get(employee.getAge());
       if (thisAge != null){
           thisAge.add(employee);
       } else{
           List<Employee> createThisAge = new ArrayList<>();
           createThisAge.add(employee);
           ageDistribution.put(employee.getAge(), createThisAge);
       }
   }
}

System.out.println(ageDistribution);
```

Java 8 Stream ...

- Connects data source and client
- Do not hold any data
- Implements map / filter / reduce pattern
- Enforces functional style of data processing

Stream collectors

```
List<Employee> employees = service.getEmployees();

Map<Integer, List<Employee>> ageDistribution = new HashMap<>();
for (Employee employee : employees) {
   if (employee.getAge() > 25) {
       List<Employee> thisAge = ageDistribution.get(employee.getAge());
       if (thisAge != null){
           thisAge.add(employee);
       } else {
           List<Employee> createThisAge = new ArrayList<>();
           createThisAge.add(employee);
           ageDistribution.put(employee.getAge(), createThisAge);
       }
   }
}

System.out.println(ageDistribution);
```

Stream collectors

```
List<Employee> employees = service.getEmployees();


Map<Integer, List<Employee>> ageDistribution =
  employees.stream()
    .filter(e -> e.getAge() > 25)
    .collect(Collectors.groupingBy(Employee::getAge));


System.out.println(ageDistribution);
```

Stream collectors

```
List<Employee> employees = service.getEmployees();


Map<Integer, Long> ageDistribution =
  employees.stream()
    .filter(e -> e.getAge() > 25)
    .collect(Collectors.groupingBy(
          Employee::getAge,
          Collectors.counting()
    ));
```

Stream API - async processing

```
getEmployeeIds().stream()
    .map(this::doHttpRequest)
    .collect(Collectors.toList())
```

Output:

```
[main] processing request: c677c497
[main] processing request: 3b5320a9
[main] processing request: 9248b92e
[main] processing request: 97a68a53
```

Stream API - async processing

```
getEmployeeIds().stream()
    .parallel()
    .map(this::doHttpRequest)
    .collect(Collectors.toList())
```

Output:

```
[main] processing request: 7674da72
[ForkJoinPool.commonPool-worker-2] processing request: 747ae948
[ForkJoinPool.commonPool-worker-1] processing request: 33fe0bac
[ForkJoinPool.commonPool-worker-3] processing request: 812f69f3
[main] processing request: 11dda466
[ForkJoinPool.commonPool-worker-2] processing request: 12e22a10
[ForkJoinPool.commonPool-worker-1] processing request: e2b324f9
[ForkJoinPool.commonPool-worker-3] processing request: 8f9f8a97
```

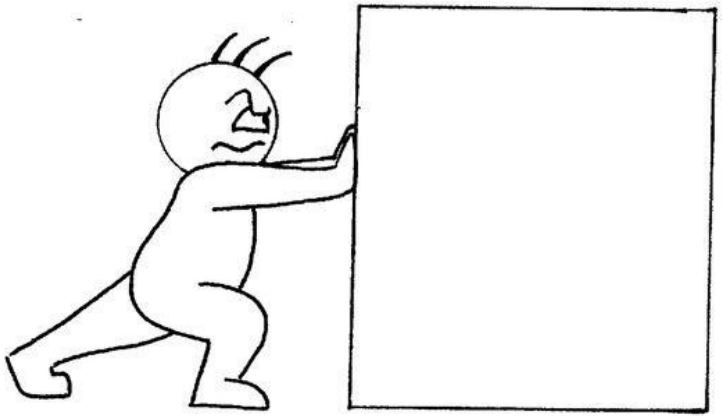
Stream API - async processing

```
ForkJoinPool forkJoinPool = new ForkJoinPool(80);

forkJoinPool.submit(() ->
        getEmployeeIds().stream()
                .parallel()
                .map(this::doHttpRequest)
                .collect(Collectors.toList())
).get();
```

Stream API has some limitations

Reactive Streams: what the difference



RxJava

http://reactivex.io

https://github.com/ReactiveX/RxJava

RxJava Observer

```
interface Observer<T> {

   void onNext(T t);

   void onCompleted();

   void onError(Throwable e);
}
```

RxJava Subscription

```
interface Subscription {

   void unsubscribe();

   boolean isUnsubscribed();
}
```

```
Subscription sub =
  Observable
    .create(s -> {
        s.onNext("A");
        s.onNext("B");
        s.onCompleted();
    })
    .subscribe(m  -> log.info("Message received: "  + m),
               e  -> log.warning("Error: " + e.getMessage()),
               () -> log.info("Done!"));
```

Output:

```
Message received: A
Message received: B
Done!
```

Stream API vs RxJava

RxJava:

- allows to process data in chosen thread, this is useful for IO, computations, specialized threads (GUI threads),
- allows synchronization on clocks and application events,
- works in push mode, producer initiates data transfer, but consumer may control data flow via backpressure.

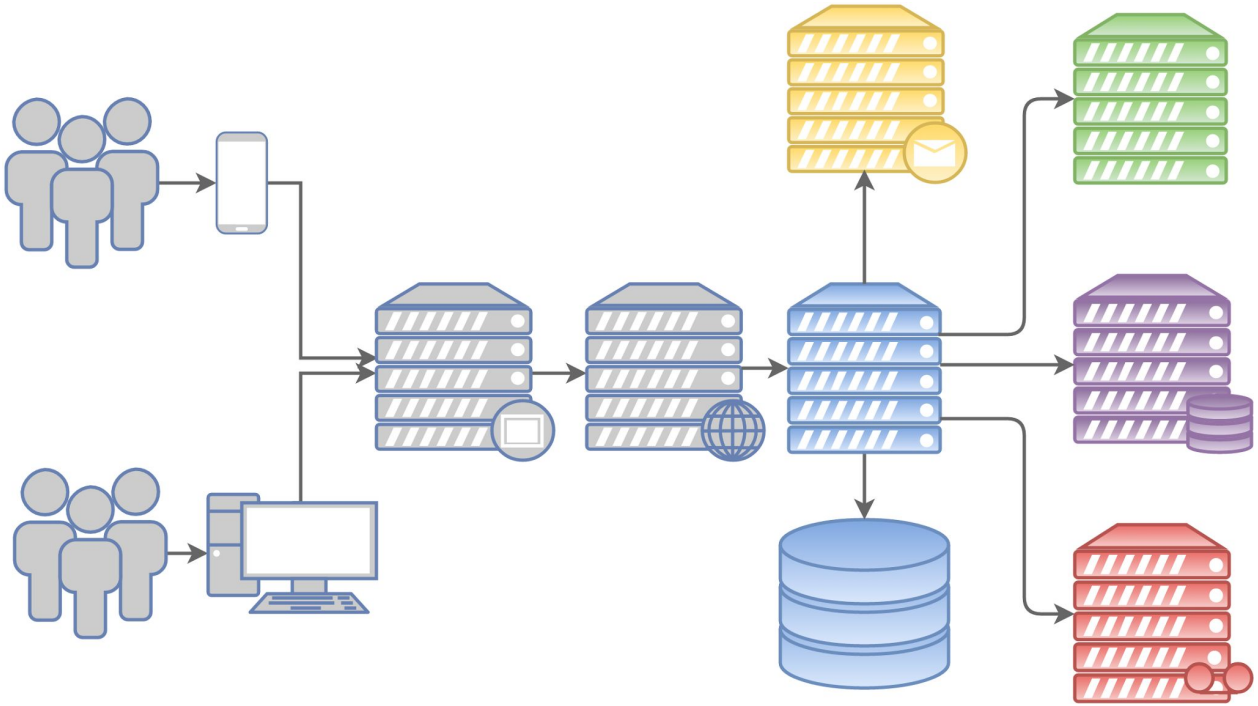
Stream API:

- tuned for hot data processing,
- good for parallel computation,
- has rich set of collectors for data.

Scenarios where RxJava shines

- Observables are better callbacks (easily wrap callbacks)
- Observables are highly composable (on contrast with callbacks)
- Provide async stream of data (on contrast with CompletableFuture)
- Observables can handle errors (have retry / backup strategies)
- Give complete control over running threads
- Good for managing IO rich application workflows
- Perfect for Android development (no Java 8 required, retrolambda compatible)
- Netflix uses RxJava for most their internal APIs

Request flow

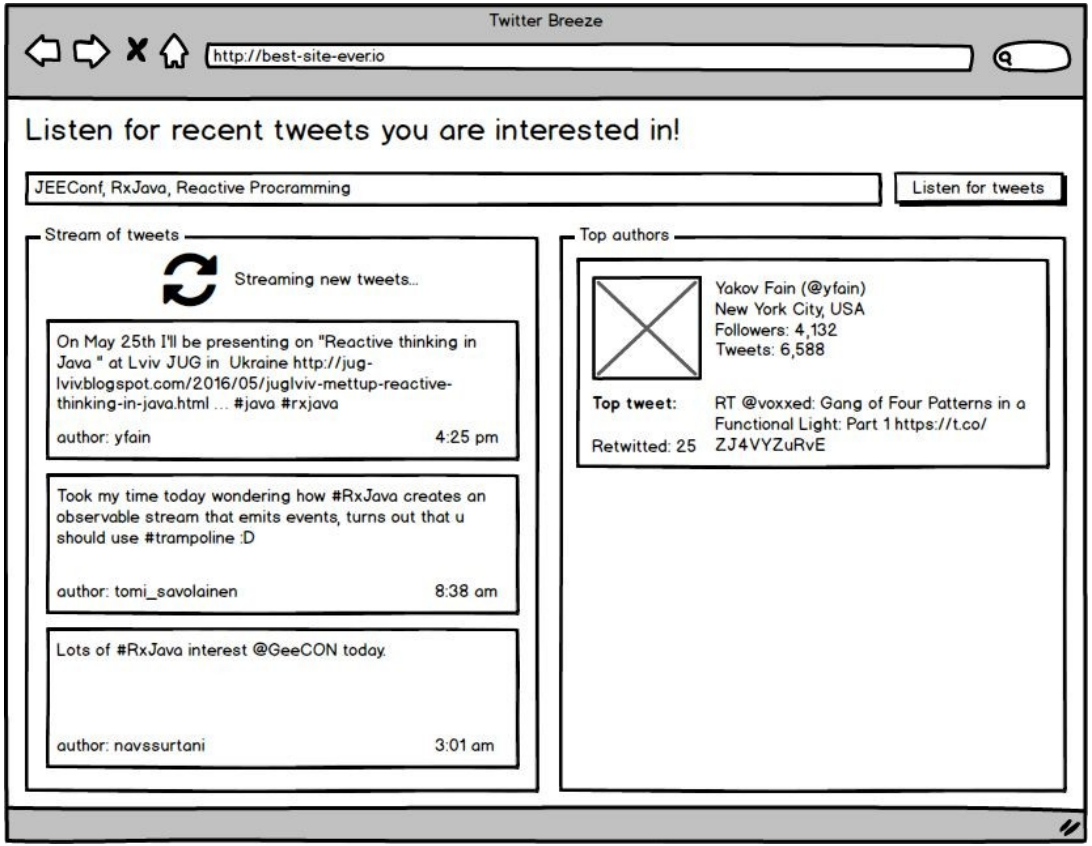


Created with draw.io

External libraries that work with RxJava

- hystrix - latency and fault tolerance bulkheading library
- camel RX - to reuse Apache Camel components
- rxjava-http-tail - allows you to follow logs over HTTP
- mod-rxvertx - extension for VertX that provides support Rx
- rxjava-jdbc - use RxJava with JDBC to stream ResultSets
- rtree - immutable in-memory R-tree and R*-tree with RxJava
- and many more ...

Use case: Stream of tweets



Twitter API

Twitter Stream API (WebSocket alike):

- Doc: `https://dev.twitter.com/streaming/overview`
- Library: *`com.twitter:hbc-core:2.2.0`*

Twitter REST API:

- `GET https://api.twitter.com/1.1/users/show.json?screen_name=jeeconf`
- `GET https://api.twitter.com/1.1/search/tweets.json?q=from:jeeconf`

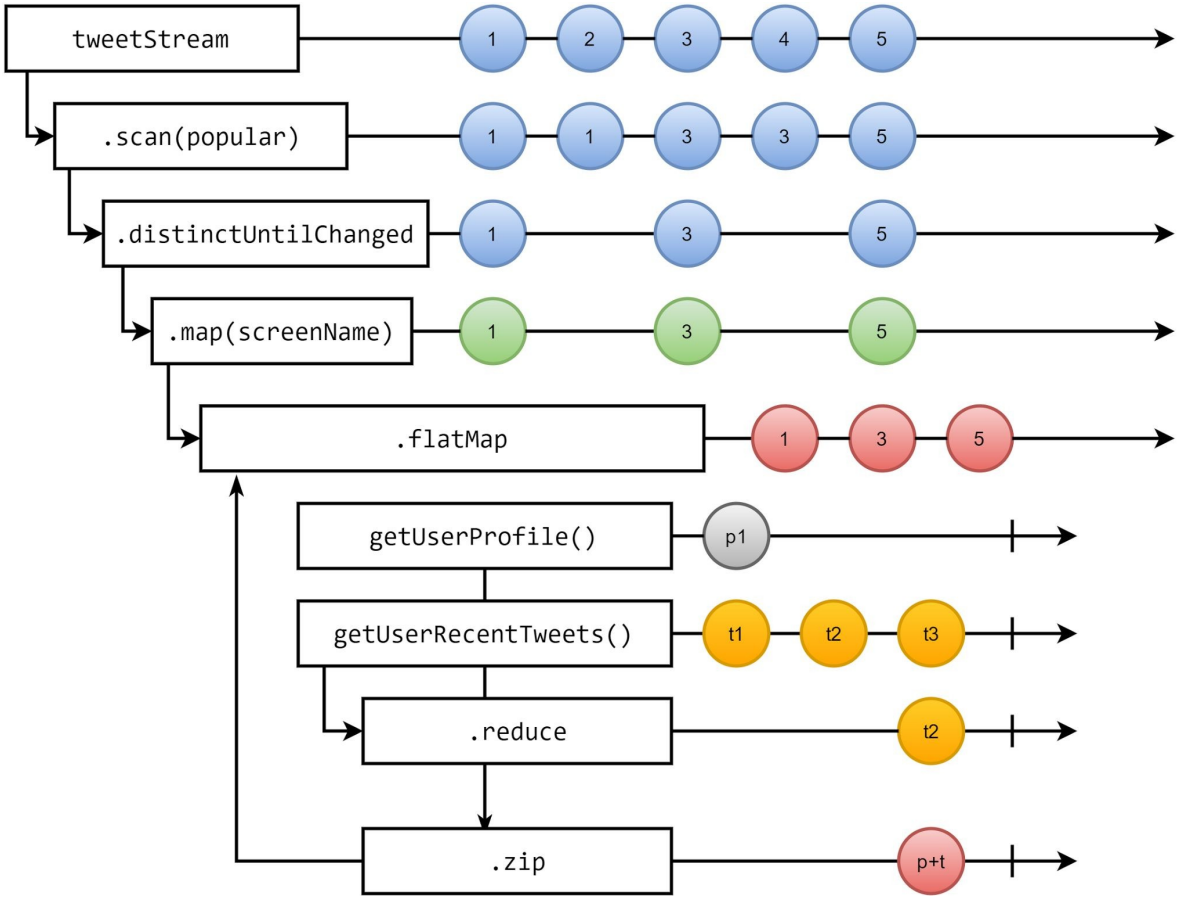
Let's look at entities

```
class Tweet {
    String text;
    int favorite_count;
    String author;
    int author_followers;
}
```

```
class Profile {
    String screen_name;
    String name;
    String location;
    int statuses_count;
    int followers_count;
}
```

```
class UserWithTweet {
    Profile profile;
    Tweet tweet;
}
```

Marble diagram



Get user profile synchronously

```
Profile getUserProfile(String screenName) {
      ObjectMapper om = new ObjectMapper();
      return (Profile) om.readValue(om.readTree(
            Unirest.get(API_BASE_URL + "users/show.json")
                  .queryString("screen_name", screenName)
                  .header("Authorization", bearerAuth(authToken.get()))
                  .asString()
                  .getBody()),
            Profile.class);
}
```

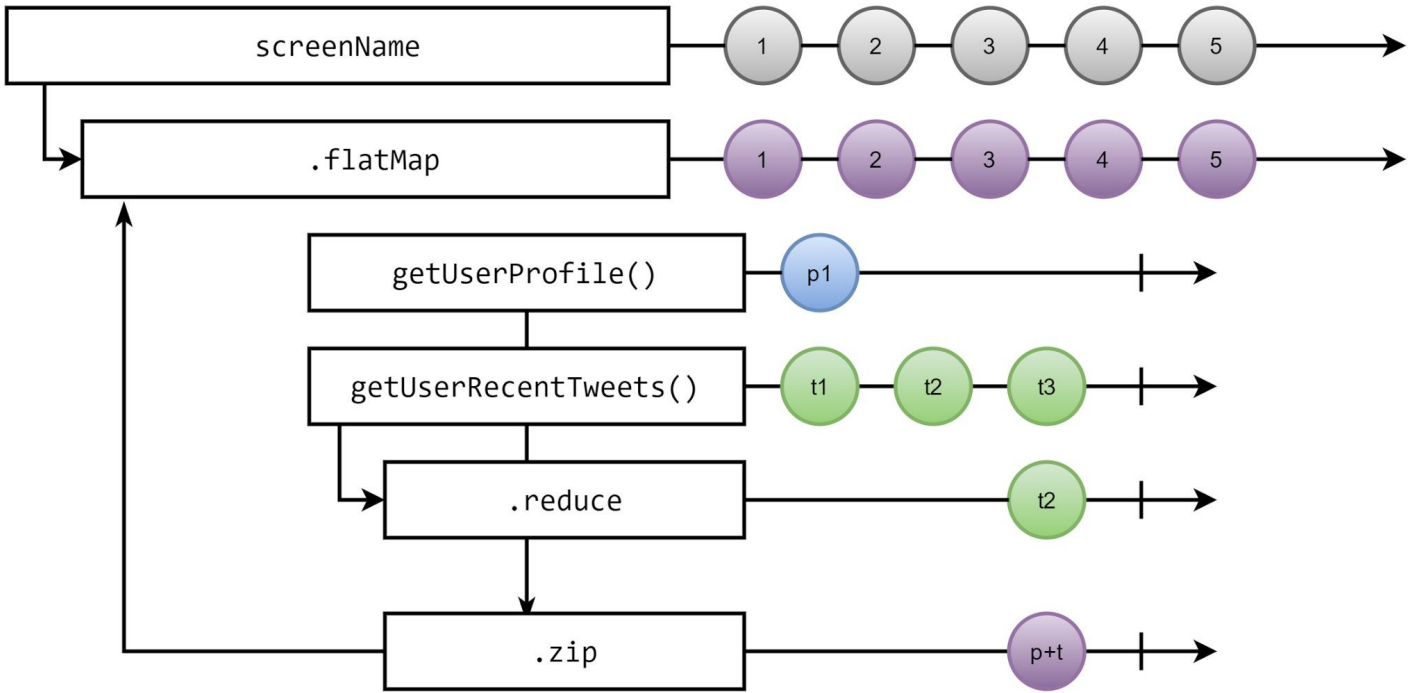
Get user profile **asynchronously**

```
Observable<Profile> getUserProfile(String screenName) {
    return Observable.fromCallable(() -> {
        ObjectMapper om = new ObjectMapper();
        return (Profile) om.readValue(om.readTree(
            Unirest.get(API_BASE_URL + "users/show.json")
                   .queryString("screen_name", screenName)
                   .header("Authorization", bearerAuth(authToken.get()))
                   .asString()
                   .getBody()),
            Profile.class);
       });
}
```

Add some errors handling

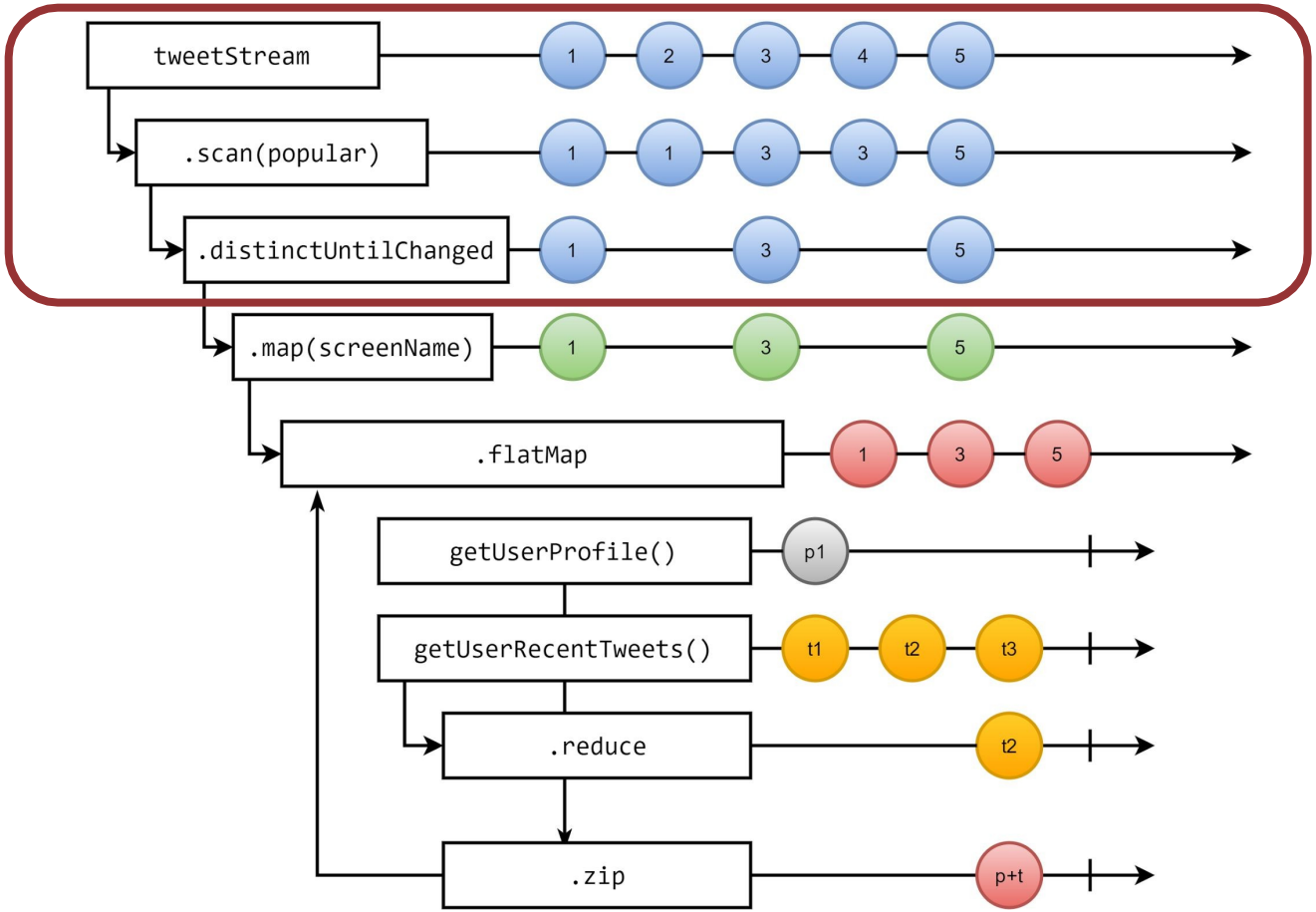
```
Observable<Profile> getUserProfile(String screenName) {
    if (authToken.isPresent()) {
        return Observable.fromCallable(() -> {
            ObjectMapper om = new ObjectMapper();
            return (Profile) om.readValue(om.readTree(
                    Unirest.get(API_BASE_URL + "users/show.json")
                            .queryString("screen_name", screenName)
                            .header("Authorization", bearerAuth(authToken.get()))
                            .asString()
                            .getBody()),
                    Profile.class);
        }).doOnCompleted(() -> log("getUserProfile completed for: " + screenName));
    } else {
        return Observable.error(new RuntimeException("Can not connect to twitter"));
    }
}
```

Concurrently



Get data concurrently


```
Observable<UserWithTweet> getUserAndPopularTweet(String author){
    return Observable.just(author)
    .flatMap(u -> {
        Observable<Profile> profile = client.getUserProfile(u)
            .subscribeOn(Schedulers.io());
        Observable<Tweet> tweet = client.getUserRecentTweets(u)
            .defaultIfEmpty(null)
            .reduce((t1, t2) ->
                t1.retweet_count > t2.retweet_count ? t1 : t2)
            .subscribeOn(Schedulers.io());
        return Observable.zip(profile, tweet, UserWithTweet::new);
    });
}
```


tweetStream
1
2
3
4
5
.scan(popular)
1
1
3
3
5
.distinctUntilChanged
1
3
5
.map(screenName)
1
3
5
.flatMap
1
3
5
getUserProfile()
p1
getUserRecentTweets()
t1
t2
t3
.reduce
t2
.zip
p+t

Let’s subscribe on stream of tweets!

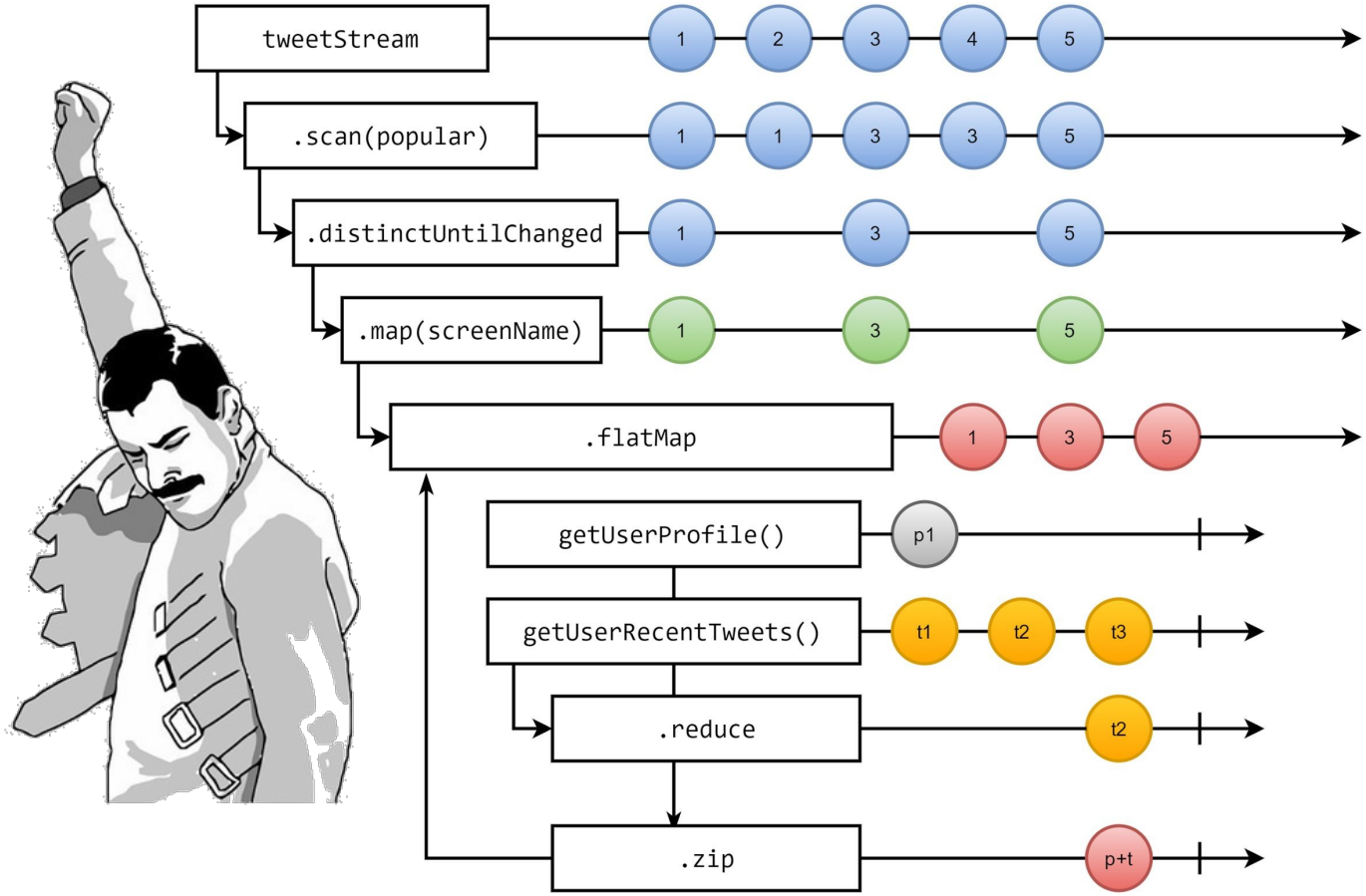
```
streamClient.getStream("RxJava", "JavaDay", "Java")
```

```
streamClient.getStream("RxJava", "JavaDay", "Java", "Trump", "Hillary")
```

```
streamClient.getStream("RxJava", "JavaDay", "Java", "Trump", "Hillary")
    .scan((u1, u2) -> u1.author_followers > u2.author_followers ? u1 : u2)
    .distinctUntilChanged()
    .map(p -> p.author)
    .flatMap(name -> getUserAndPopularTweet(name))
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.immediate())
    .subscribe(p -> log.info("The most popular tweet of user "
                                    + p.profile.name + ": " + p.tweet));
```

```
streamClient.getStream("RxJava", "JavaDay", "Java", "Trump")
    .scan((u1, u2) -> u1.author_followers > u2.author_followers ? u1 : u2)
    .distinctUntilChanged()
    .map(p -> p.author)
    .flatMap(name -> {
        Observable<Profile> profile = client.getUserProfile(name)
            .subscribeOn(Schedulers.io());
        Observable<Tweet> tweet = client.getUserRecentTweets(name)
            .defaultIfEmpty(null)
            .reduce((t1, t2) ->
                t1.retweet_count > t2.retweet_count ? t1 : t2)
            .subscribeOn(Schedulers.io());
        return Observable.zip(profile, tweet, UserWithTweet::new);
    })
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.immediate())
    .subscribe(p -> log.info("The most popular tweet of user "
                             + p.profile.name + ": " + p.tweet));
```

Marble diagram



Conclusions: pitfalls

- API is big (150+ methods to remember)
- Requires to understand underlying magic
- Hard to debug
- Don’t forget about back pressure

Conclusions: strength

- It is functional, it is reactive*
- Good for integration scenarios
- Allows to control execution threads
- Easy to compose workflows
- Easy to integrate into existing solutions
- Ok to test

Demo - TweetRx