Practical RxJS In The Wild 🦁— Requests with concatMap() vs mergeMap() vs forkJoin() 🥊

Tomas Trajan Follow
Dec 19, 2017 · 7 min read

What? Stream? Did anyone say stream?! (Pipe cat 📷 by Mikhail Vasilyev)

I would like to share with you experience acquired by working on a yet another Hacker News client (code name `HAKAFAKA` 😂 still in alpha). I have been on the road for couple months now and realized that a small coding project wouldn't hurt. And sure it didn't, on the contrary it provided inspiration for this new post so let's get straight to it!

Contextual intro

In most of the apps we are building we have to perform at least some requests to the backend. Retrieving data to show it to the user, updating entities, submitting forms or any other activity involving communication with the server.

Retrieving of collections

Most APIs provide endpoints for retrieving whole collections of items with single request. Think users, posts or transactions. For large collections it is usually also possible to retrieve a subset of all items by sending pagination information in query parameters.

When we're developing our own API we are in full control and we should provide such endpoints because it reduces overhead of firing multiple requests and handling of individual responses per retrieved item.

Unfortunately, sometimes we will end up in situation where we have to consume 3rd party API which doesn't provide such convenience. In these cases we have to handle retrieving of collections ourselves.

> *Hacker News API is an example of API which doesn't let us retrieve collection of items in a single request. Instead, what we get is a list of IDs and we have to retrieve corresponding items one by one...*

Retrieving collection of items with RxJS

In our examples we will be using Angular to execute requests with provided `HttpClient` service. It provides us with expected methods like `.get()`, `.post()` or `.delete()` which all return observable of a response.

Returning observables enables us to use RxJS to combine and process requests out of box. Also, keep in mind that even though we're using Angular, the following concepts are **framework agnostic** and can work with anything which returns observables for it's async operations.

> *Code examples are implemented using RxJS 5 with* **lettable** *operators. In short, this means we will use* `.pipe()` *function eg:* `clicks$.pipe(debounce(250))` *instead of chaining operators directly on the Observable like this* `clicks$.debounce(250)` .

The concatMap() solution

Retrieving collection of items with a known set of IDs can be performed using RxJS `from()` method which passes our IDs one by one to the following operators.

In our case we want to perform a HTTP request for every ID and Angular `HttpClient` returns observable of a response by default. This leaves us with observable of observables but we're much more interested in the actual responses instead.

To solve this situation we have to flatten our observable stream and one of the available operators is `concatMap()` which does exactly what we need. The implementation will look something like this...

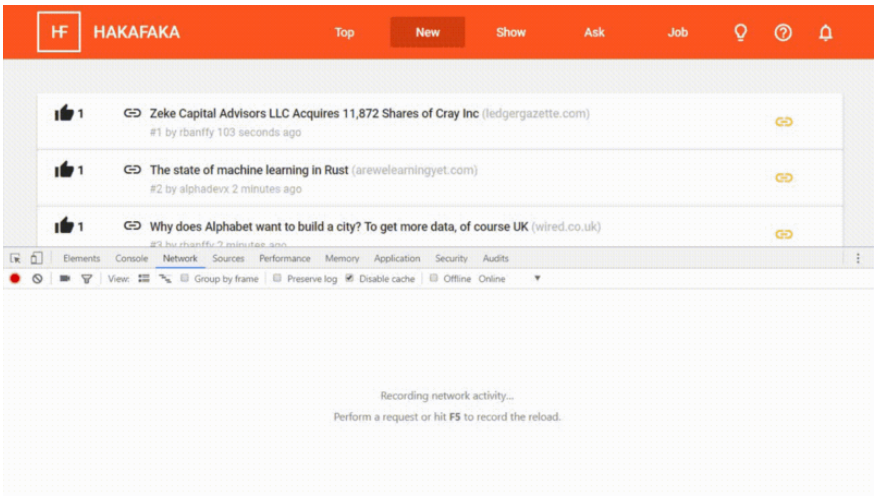
```
1  getItems(ids: number[]): Observable<Item> {
2    return from(ids).pipe(
3       concatMap(id => <Observable<Item>> this.httpClient
4    );
```

> *concatMap(): Projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next—Official RxJS Docs*

Official documentation might sound a tad bit academic. More simply, `concatMap()` flattens our stream from *observable of observables* to *observable of responses*.

Another important property is that it will wait for completion of previous observable before executing next one. This translates into every request waiting for completion of previous one which might not be exactly the best for performance.

In practice it will lead to a cascade of requests as shown in following animation...



RxJS concatMap() executes requests one by one, order is preserved but executing just one request at a time wastes extreme amount of time... BAD

Hopefully there is a better way to handle this requirement!

> *Follow me on Twitter because I want to be able to let you know about the newest blog posts and interesting frontend stuff*

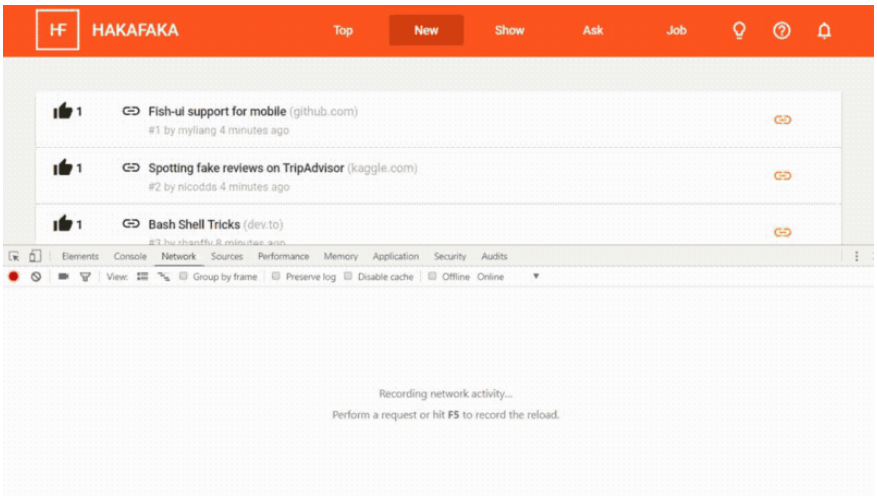
The mergeMap() solution

Another operator which seems to fit our situation is `mergeMap()`. It flattens our *observable of observables* stream into stream of responses too. Implementation is almost the same as previously, we only have to swap operator...

```
1  getItems(ids: number[]): Observable<Item> {
2    return from(ids).pipe(
3      mergeMap(id => <Observable<Item>> this.httpClient.g
4    );
```

> *mergeMap(): Projects each source value to an Observable which is merged in the output Observable—Official RxJS Docs*

Docs don't give away too many details but the key feature of `mergeMap()` is that it executes all nested observables immediately as they pass through the stream. This is a major performance win because all our requests are executed in parallel and the network tab will look something like this...



RxJS mergeMap() executes requests in parallel, the execution time is much better compared to concatMap(), but responses might arrive in changed order due to network conditions... SAD

The mergeMap() caveat

Our solution has a one major flaw though. Requests are executed in parallel but the responses might take different amounts of time to complete due to network conditions. In most cases it is important to preserve the order of collection we're retrieving.

> *The top post should be the one which received highest number of 👍 not the one with fastest response time from backend*

We can demonstrate this problematic behavior with simple code snippet where we delay the first request and all subsequent requests are performed as usual. The first request will arrive as last response if we're assuming standard network conditions.

```
1  getItems(ids: number[]): Observable<Item> {
2    return from(ids).pipe(
3      mergeMap((id, index) => {
4        if (index === 0) {
5          return  <Observable<Item>> this.httpClient.get
6        }
7        return <Observable<Item>> this.httpClient.get(`i
8      )
```

How can we fix this situation? Remember we have started with an ordered set of IDs and retrieved items usually contain ID property too. This enables us to use IDs to sort retrieved items every time a new response arrives.

Hypothetical service implementation then can look something like this...

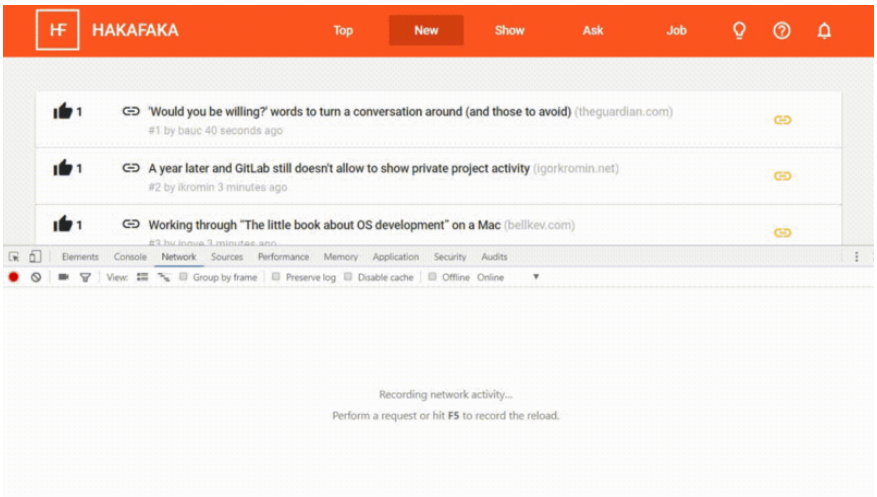
```
@Injectable()
export class FeatureService {

  items: Item[] = [];

  constructor(private backendService: BackendService)

  getItemsForIds(ids) {
    this.backendService
      .getItems(ids)
      .subscribe(item => {
        this.items.push(item);
        /*
          sort items by original IDs order
          because responses might arrived in unordered
          due to network conditions
        */
```

Now, our first delayed request is inserted correctly as a first item of the displayed list 🎉



RxJS mergeMap() with ordering of retrieved items, notice how the delayed item is correctly added to the beginning of the list... GREAT AGAIN 😂

The forkJoin() solution

Another way to execute requests in parallel is using `forkJoin()` instead of `from()` . It expect an array of observables as an argument

so we have to map IDs to requests first. This will return an array of responses when all requests have finished. What we want is to add posts one by one so we also have to use `concatAll()` operator at the end.

```
getItems(ids: number[]): Observable<Post> {
  return <Observable<Post>> forkJoin(
    ids.map(id => <Observable<Post>> this.httpClient.ge
  ).pipe(concatAll());
```

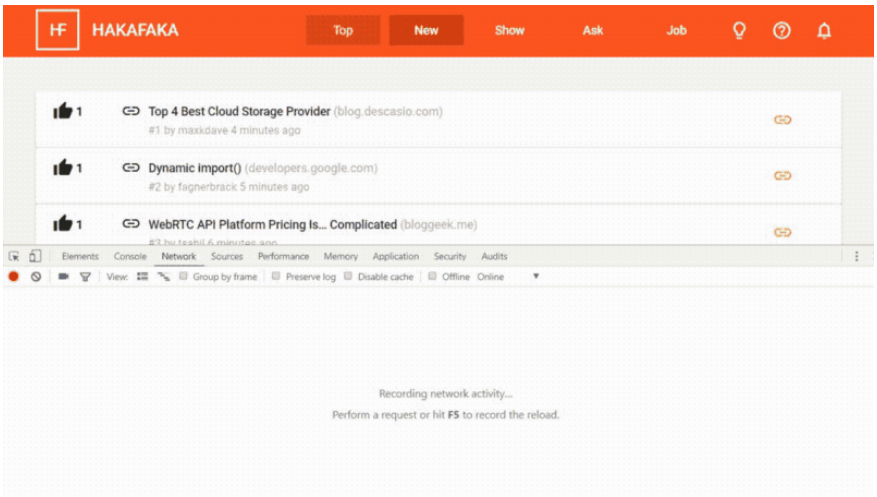
> *forkJoin(): as of December 2017 undocumented—Official RxJS Docs*

The forkJoin() caveat

This operator faces some problems with delayed requests too. The order will be preserved but if one request is delayed all the others have to wait for its resolution. We can simulate that situation with the snippet below...

```
getItems(ids: number[]): Observable<Post> {
  return <Observable<Post>> forkJoin(
    ids.map((id, index) => {
      if (index === 0) {
        return  <Observable<Post>> this.httpClient.get
      }
      return <Observable<Post>> this.httpClient.get(`i
```

As we can see, even though most requests were already resolved, the whole collection is blocked until resolution of the delayed item which is not very user friendly.



RxJS concatMap() executes requests in parallel but waits for all responses to arrive before passing results to the subscriber, one delayed request can stall the whole process... DISAPPOINTING

Another issue with `forkJoin()` is related to the way it handles failed requests. If any of the executed requests fails it will fail for the whole collection. Instead of items we will receive first encountered exception.

This might be desirable behavior if we need to guarantee that the list is consistent which may be the case when retrieving stuff like transactions. On the other hand, retrieving posts for the timeline of social network would be better of with showing all the successful responses. Even if some posts might be missing.

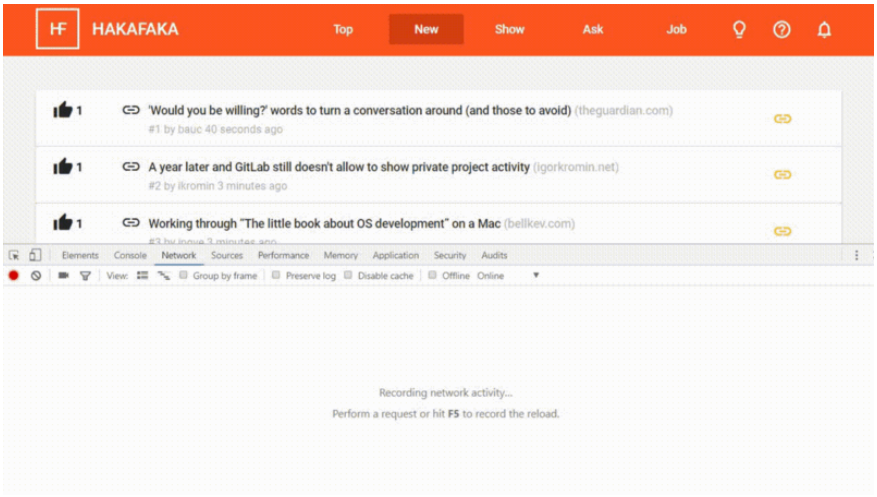
EDIT: After great feedback from Ihor Bodnarchuk it became clear that we can adjust `forkJoin()` code in such a way that every inner Observable will handle it's own error so that `forkJoin()` doesn't emit error on the first encountered error. The inner Observable can then look like this `this.httpClient.get(`item/${id}`).pipe(catchError(() => of(undefined)))` . This will return `undefined` instead of the error so we still need to filter collection after fork join resolves to remove these undefined items to end up with the collection of retrieved posts.

> *As always, it depends on the particular use case*

The Winner 🎉

And the winner is `mergeMap()` ! It executes requests in parallel and it is fault tolerant so we still display most of the posts even if some of the requests fail.

Just don't forget to handle ordering of retrieved items when necessary!



As shown above, RxJS .mergeMap() executes requests in parallel and when used in combination with response sorting provides the best results... GREAT

Did we forget about the switchMap() ?

You might be aware that there is also `switchMap()` operator which is used in many tutorials and guides to implement stuff like type-ahead component.

The use case is a bit different and it boils down to cancellation of previous requests once we execute new ones to prevent displaying of old responses due to race conditions.

If we used `switchMap()` in our case we would end up with a single (last) item of our list because every other request would have been cancelled. We can try it with following snippet.

```
1  getItems(ids: number[]): Observable<Item> {
2    return from(ids).pipe(
3      switchMap(id => <Observable<Item>> this.httpClient.
4    );
```

That's right! We made it to the end!

I hope you found this guide helpful and become aware of different approaches on how to implement multiple requests in your apps.

Please support this article with your 👏 👏 👏 to spread these tips to a wider audience and follow me on 🕊️ Twitter to get notified about newest blog posts.

You also might be interested in other Angular & Frontend related posts...

The Angular Model (ngx-model)

How to handle state in your Angular applications in standardized way with...

medium.com

6 Best Practices & Pro Tips when using Angular CLI

Learn how to organize your modules, use aliases, use sass, production build, testin...

medium.com

🎨 How To Style Angular Application Loading With Angular CLI Like a Boss 😎

Slow internet is a fact of life in many places around the world. Prompt users to wait...

medium.com

And never forget, future is bright

Obviously the bright future

Actually, future became present 😂

. . .