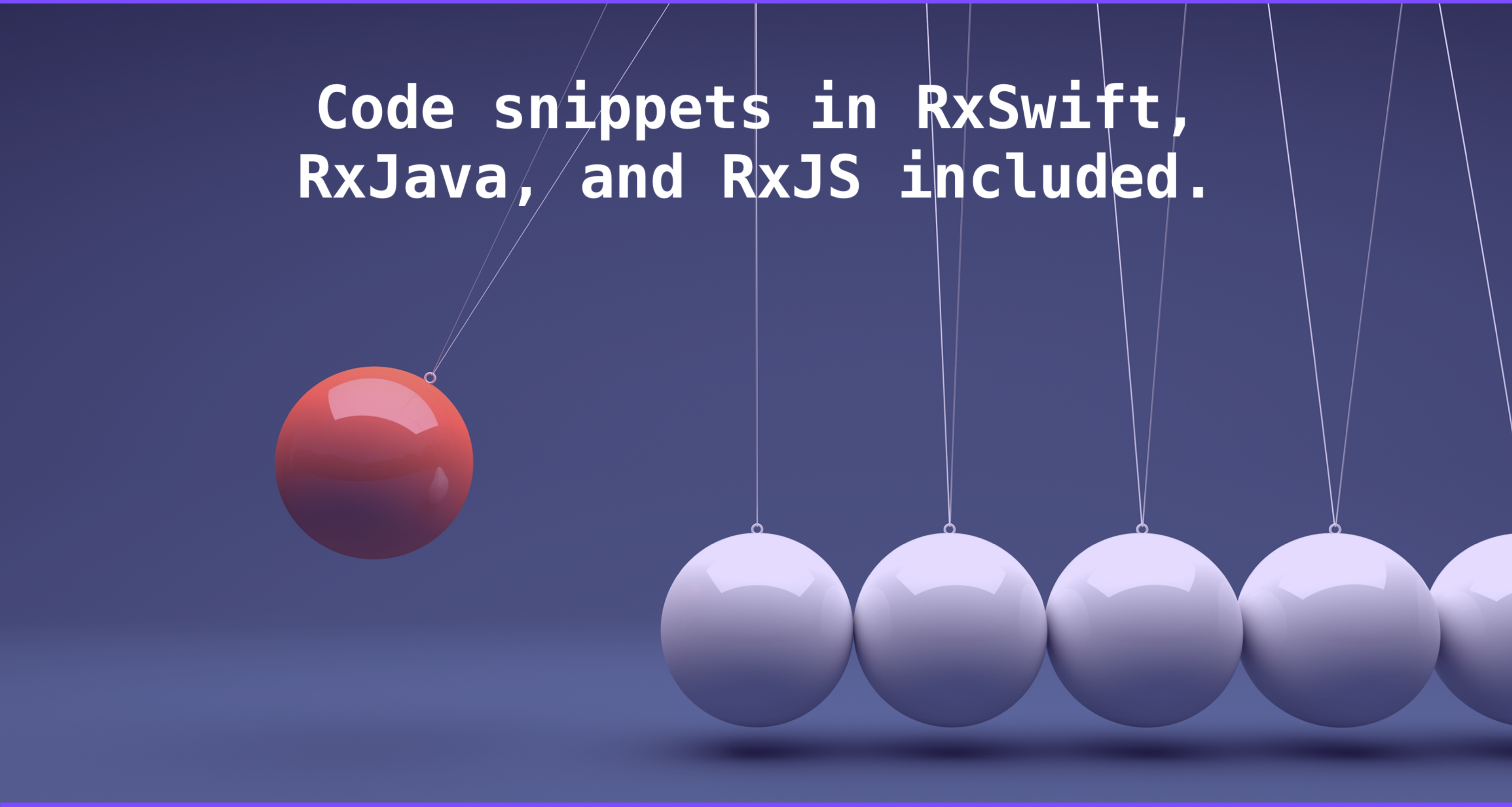
The Clean Way to Use Rx

Guide manual and tips for implementing with Rx extensions



Code snippets in RxSwift, RxJava, and RxJS included.

Yair Carreno

The Clean Way to Use Rx

Guide manual and tips for implementing with Rx extensions.

Yair Carreno

This book is for sale at http://leanpub.com/the-clean-way-to-use-rx

This version was published on 2020-07-24

Contents

Prefacio

About the book

This work, groups the different practices used in the implementation of software components through *Rx* extensions. These practices are organized into 35 application items, and for each item, it is analyzed, which are the recommended code practices and the practices to avoid.

It could also be said that this document is the compilation of good practices learned through the own experience acquired in business projects, recommendations received from forums, blogs, workshops and in general from analyzes given by experts in the area, including the recommendations given in the official sites of each extension [1].

This book is also intended as a practical reference manual for implementing *Rx* code. That is why much of the book's material consists of examples of *code snippets* written in the main programming languages for both Frontend (Mobile, Web; that is, JavaScript, Swift, Java) such as Backend.

The concepts studied in each of the items are agnostic to the programming language, there are few cases in which a certain capacity is not available in an extension and in which case the respective annotation is made in the item.

It is intended that once the concepts of each item have been studied, the reader benefits from a greater understanding of how *Rx* works.

Readers will also be equipped with tools to apply good practices that are ultimately reflected in the solutions in the area of:

- Clean Code.
- Best performance:
- Mitigate bugs scenarios.

About the source code

In the book, the *code snippet* examples are presented in *Swift*. The reader can also find the equivalent code in *JavaScript* and *Java* in the chapters:

- RxJS code snippet
- RxJava code snippet

[1]ReactiveX

The reader will also be able to access the public repositories written for each technology at:

- Clean Way Rx in JavaScript[2]
- Clean Way Rx in Swift[3]
- Clean Way Rx in Java[4]

Content development

Content development starts from the analysis of the fundamental concepts of both *Reactive programming* and *Functional programming*. The *Foundation* section introduces how these concepts are effectively used in Rx. The items described in this section are the basis for subsequent items.

In the next section named *Be Clean*, the emphasis is placed on clean, well-written, and easy-to-read code. *Rx* substantially simplifies the written code necessary to accomplish a task; however, you can quickly lose the advantage of simplicity and go to another extreme where the understanding of the code becomes complex because it is written in an unclean way.

One of the leading actors in *Rx* is the *data stream*. Said data flow can also be related to other flow generating sources and therefore establish dependencies between them. The *Flow dependencies* section discusses the considerations to consider when setting such dependencies between flows.

Operators are another critical actor in *Rx*. Learning to distinguish between them and knowing when to use one or the other is one of the essential skills acquired during development and design with *Rx*. In the *Operators* section, the items are involved with the recommendations for the operators' use.

Time becomes a variable to consider when data streams are orchestrated; this is discussed in the *Timing* section. A proper administration of the moments and execution times generate the expected results.

The following section deals with a topic relevant to any application and has to do with the management and administration of error scenarios. *Rx* has an important set of operators and functions that allow designing effective error control. These options are discussed in the *Error Handling* section.

The next two sections *Multicasting* and *Multi-threading* might look similar but are very different. *Multicasting* analyzes the practices of linking multiple sources of data and numerous recipients of such data. Instead, the *Multi-threading* section discusses concurrency and task parallelization in various execution contexts in the scope of processing.

And finally, there is the *Optimizing* section, which contains a couple of items referring to good practices that avoid the generation of memory leaks and proper use of memory resources used by the components designed with *Rx*.

[2]https://github.com/yaircarreno/Clean-Way-Rx-JavaScript
[3]https://github.com/yaircarreno/Clean-Way-Rx-Swift
[4]https://github.com/yaircarreno/Clean-Way-Rx-Java

As the reader will appreciate, the content is not developed following a line of basic, medium, and advanced topics. Still, on the contrary, everything is related, to the point that there are items related to others and therefore complement each other.

Glossary

During the development of the topics, the reader will be able to find some terms described below for clarity.

Contract: Refers to the events that could occur in broadcast and correspond to *onNext*, *onComplete* or *onError*.

Imperative: Refers to the traditional form of programming based on the execution of tasks sequentially.

Reactive: Refers to the style of programming based on the execution of tasks synchronously or asynchronously and whose purposes the *Rx* extensions are used.

Rx: This term will be used to refer to *Reactive programming*.

Audience

I recommend this reading for any actor that participates in the design, implementation, and validation processes of software components through *Reactive programming* and *Functional programming*, supported by the use of *Rx* extensions.

Whether it is a component in Frontend or Backend, the reader will be able to apply this book's knowledge when implementing code with *Rx* extensions. For example, it allows you to use pattern designs such as Unidirectional State Flow, MVVM, Optimistic UI, and other patterns required by the solution architecture[5].

Questions and contact

This work, intended to guide the reader in the search for good design practices in solutions, has been developed from my point of view. Taking into account the recommendations of expert sources cited in the bibliography, the reader could very surely find alternatives to the exposed techniques. They may vary a little from those listed here, and that is precisely the idea of this work.

If the reader finds something in the book that deserves to be reviewed, comments and feedback are welcome. For this and any questions or concerns, the following channels are available:

- Email: yaircarreno@gmail.com

[5]Clean Architecture in iOS

- Twitter: @yaircarreno[6]
- Blog: yaircarreno.com[7]

Versions of IDEs and technologies used.

- Xcode 11.5 - Swift 5
- Android Studio 4.0 - Java 1.8.0
- Visual Studio Code 1.47.1 - TypeScript 3.8.3 - Angular 9.1.0

Changelog

Chapter Changelog is provided to keep the reader informed about updates and changes in this book.

[6]https://twitter.com/yaircarreno
[7]https://www.yaircarreno.com/

Rules summary

Foundation

Item 1: From Imperative to Reactive

Relating *imperative code* to *reactive code* is possible as long as the capabilities of each of these styles are taken into account.

It is usual for the solution to contain both imperative and reactive code blocks. Actually, the source of reactive code is imperative code to which are added capabilities inherited from *functional programming* and *multithreaded* capabilities.

The basic rules for relating imperative code and reactive code could be summarized in the following points:

1. Orchestration should only occur between *Rx* code; that is, you should not orchestrate *Rx* tasks using *imperative* code.
2. *Imperative* code can be orchestrated with *Rx* tasks as long as *imperative* code is wrapped in *Rx* code.

The reader will find in item 8, item 9 and item 10, the effects of not complying with the two previous recommendations.

Now, how do I wrap imperative code in *Rx* code?

Through two available mechanisms:

- Through creation operators.
- Through the creation functions.

But before designing the envelope, consider the following analysis in Figure 1.1.

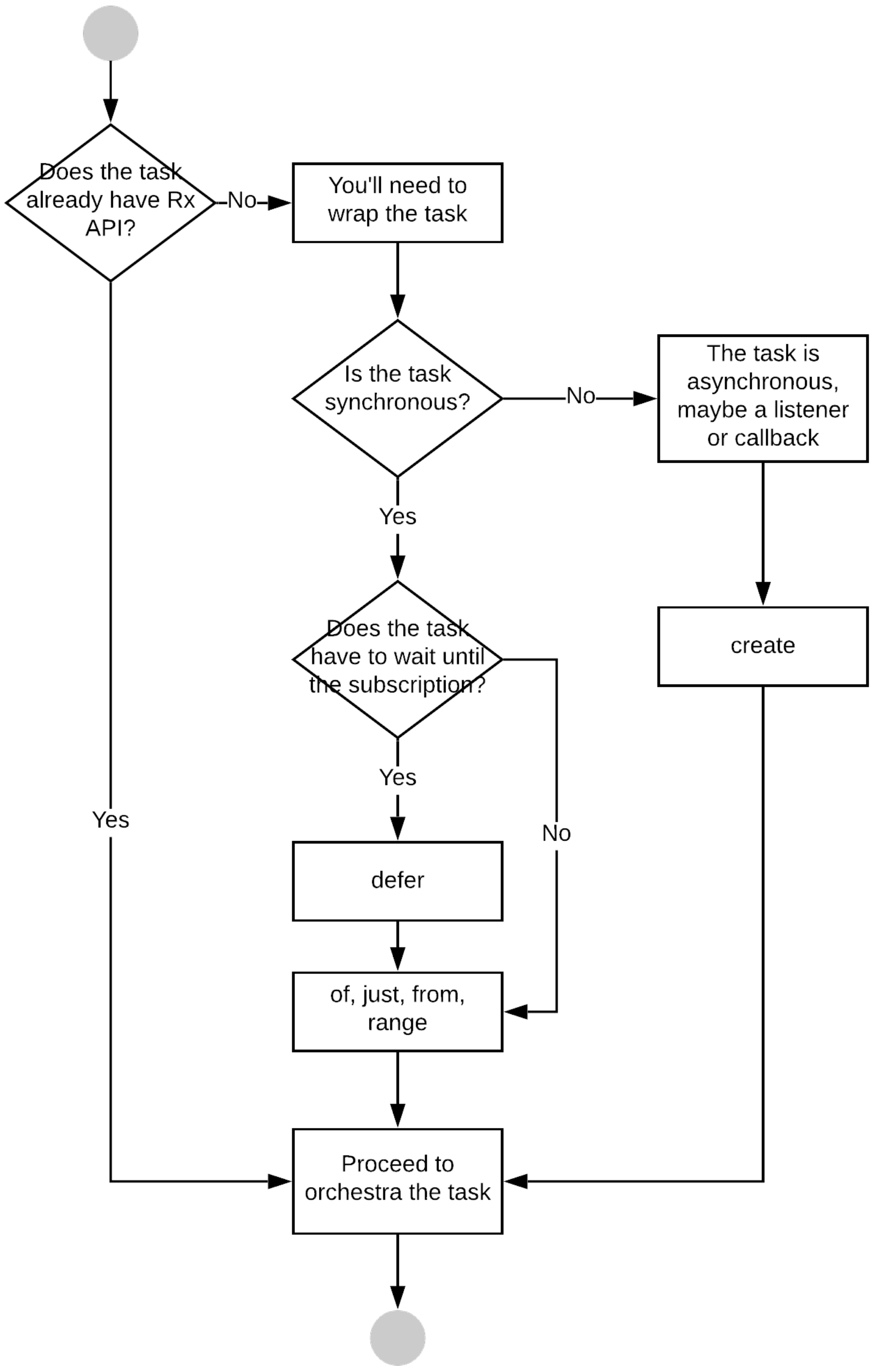

Does the task already have Rx API?
No
You'll need to wrap the task
Is the task synchronous?
No
The task is asynchronous, maybe a listener or callback
Yes
Does the task have to wait until the subscription?
create
Yes
Yes
No
defer
of, just, from, range
Proceed to orchestra the task

Figure 1.1 Wrapping tasks to Rx

- Validate first if the task already has an envelope. Third-party libraries, for example, provide these implementations when it comes to integrating third-party SDKs. Even in the project itself, a utility could already exist with said envelope.

- If the envelope is required, consider whether the task operates synchronously or asynchronously. Identify this by checking if the job is performed by a *listener*, *callback*, or any component of asynchronous nature.

  Sometimes it is mistakenly believed that every operation performed with *Rx* is *multithreaded* by default. The truth is that Rx is *single-threaded* by default [^8].

  Unless otherwise indicated, all operations will run on the same thread assigned by default. This feature is discussed in Item 30: Rx is single-threaded by default.
- If the task is synchronous, an Observable could be created through the creation operators that wraps the job.
- If the task is asynchronous, it could be wrapped through the creation functions and create *Traits*, that is, Observables of type *Single*, *Complete*, or *Maybe* according to the requirement and having full knowledge of the contracts to be fulfilled by the *Observer*.

Below are some of the options for wrapping tasks through operators.

Operators name

Please note that some operators are named differently or are not available in all languages.

Creation operators

Rx has the following mechanisms for wrapping tasks through operators:

Code Snippet: RxJS - RxJava

Example 1.1: of operator - CleanWayRx/Items/Item1.swift

```
Observable.of(["h", "e"], ["l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)


Console output:
---------------
["h", "e"]
["l", "l", "o"]
```

Example 1.2: just operator - CleanWayRx/Items/Item1.swift

```
Observable.just(["h", "e", "l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)

Console output:
---------------
["h", "e", "l", "l", "o"]
```

Example 1.3: from operator - CleanWayRx/Items/Item1.swift

```
Observable.from(["h", "e", "l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)

Console output:
---------------
h
e
l
l
o
```

Example 1.4: range operator - CleanWayRx/Items/Item1.swift

```
Observable.range(start: 1, count: 5)
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)


Console output:
---------------
1
2
3
4
5
```

Example 1.5: defer operator - CleanWayRx/Items/Item1.swift

```
Observable.deferred {
    return Observable.just(["h", "e", "l", "l", "o"])
}
.subscribe(onNext: { element in print(element) })
.disposed(by: disposeBag)


Console output:
---------------
["h", "e", "l", "l", "o"]
```

Support operators

These are also creation operators not intended to wrap but to support testing, debugging and error handling processes:

Code Snippet: RxJS - RxJava

Example 1.6: interval operator - CleanWayRx/Items/Item1.swift

```
let subscription = Observable<Int>
    .interval(.seconds(1), scheduler: scheduler)
    .subscribe { event in print(event) }

Thread.sleep(forTimeInterval: 5.0)
subscription.dispose()


Console output:
---------------
next(0)
next(1)
next(2)
next(3)
next(4)
```

Example 1.7: timer operator - CleanWayRx/Items/Item1.swift

```
let subscription = Observable<Int>.timer(.seconds(2), scheduler: scheduler)
    .subscribe { event in print(event) }

Thread.sleep(forTimeInterval: 5)
subscription.dispose()


Console output:
---------------
next(0)
completed
```

Example 1.8: empty operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.empty()
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)

Console output:
---------------
completed
```

Example 1.9: never operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.never()
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)

Console output:
---------------
```

Example 1.10: error operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.error(SampleError())
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)


Console output:
---------------
error(SampleError())
```

Creation functions

When it comes to wrapping an asynchronous task that is part of an external API and in which case a creation operator is not enough, the creation of the *Observable* is resorted to through the creation functions.

The anatomy for wrapping a task is as follows:

Example 1.11: CleanWayRx/Items/Item1.swift

```swift
private func taskWrapped() -> Observable<Any> {

    return Observable.create { observer in

        // Here the imperative code is embedded
        return Disposables.create()
    }
}
```

Code Snippet: RxJs - RxJava

Essential points for designing the *Observable* from *Observable.create*, and imperative code are:

1. Take into account the contracts that the *Observer* might need: *onNext* *, **onComplete* and *onError*.
2. As defined in point 1, analyze whether a general *Observable* or one of the types: *Single*, *Complete*, or *Maybe* is used.
3. Handle possible scenarios that generate errors and emits through the *onError* contract.
4. Make sure to release the particular strong references and any listeners bound by the imperative code.
5. Consider whether you need to handle *Backpressure* scenarios[^9].

Consider the following example of an envelope implementation:

Example 1.12: CleanWayRx/Items/Item1.swift

```swift
private func imperativeTask() -> Any? {

    let data = "any data"
    print("Do any imperative task or process")

    return data
}
```

The imperative task called *imperativeTask* could generate a result from a calculation or perform a task that does not generate a result but a completed action.

Whatever the case, an *Observable* is created with the result with data or if they. The conditions for emitting an error event, a completion event or an item emission event are defined:

Example 1.13: CleanWayRx/Items/Item1.swift

```swift
private func taskWrapped(task: Any?) -> Observable<Any> {

    return Observable.create { observer in

        guard let data = task else {
            observer.onError(SampleError())
            return Disposables.create {}
        }

        observer.onNext(data)
        observer.onCompleted()
        return Disposables.create()
    }
}
```

Code Snippet: RxJs - RxJava

Through the envelope, the task is available to be orchestrated and controlled in an Rx way:

Example 1.14: CleanWayRx/Items/Item1.swift

```swift
taskWrapped(task: self.imperativeTask())
    .subscribe(onNext: { data in print("next:", data) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)

Console output:
---------------
Do any imperative task or process
next: any data
completed
```

Here is an example of an envelope designed for a fairly well-known API, *Firebase Database Realtime*:

Example 1.15: Wrapping Firebase login task

```swift
public func rx_loginUser(user: User) -> Single<Bool> {

    return Single<Bool>.create { single in
        self.authReference.signIn(withEmail: user.email, password: user.password in
            if error == nil {
                single(.success(true))
            }
            else{
                single(.success(false))
            }
        }
        return Disposables.create()
    }
}
```

Another possible envelope for queries:

Example 1.16: Wrapping Firebase single event task

```swift
func rx_observeSingleEvent(of event: DataEventType) -> Observable<DataSnapshot> {
    return Observable.create({ (observer) -> Disposable in
        self.observeSingleEvent(of: event, with: { (snapshot) in
            observer.onNext(snapshot)
            observer.onCompleted()
        }, withCancel: { (error) in
            observer.onError(error)
        })
        return Disposables.create()
    })
}
```

Be Clean

Item 5: Keep clean the operators' chain

An *operators' chain* is a powerful tool for orchestrating tasks. Keeping the definition of the operator chain legible and clean allows for a better understanding of organized responsibilities.

Consider the following example:

Example 5.1: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token)
        .concatMap { saved in self.network.getUser(username) }
}
.subscribe(onNext: { user in print(user) })
.disposed(by: disposeBag)
```

The objective of the previous *code snippet* is to orchestrate a task that obtains a *token*, then caches it, and then proceeds to query the user.

That code is valid; however, it could be improved in the following way:

Example 5.2: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token) }
    .concatMap { saved in self.network.getUser(username) }
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

Code Snippet: RxJS - RxJava

What is the difference between code example 5.1 and code example 5.2?

Code in Example 5.1 is often used to orchestrate tasks with result dependencies, that is, when the next task (downstream) requires the results of the previous task (upstream).

In example 5.1, *getUser* operation does not need the results of the previous actions: the results of *getToken* or *storeToken*. For this reason, the code in Example 5.1 can be replaced by the code in Example 5.2 and clearer.

Now we show a case where the dependency on results is necessary:

Example 5.3: CleanWayRx/Items/Item5.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token)
        .concatMap { saved in self.network.getUser(username, token) }
}
.subscribe(onNext: { user in print(user) })
.disposed(by: disposeBag)
```

The *getUser* task needs the result of one of the previous tasks: the *token* in this case. Therefore an internal concatenation is used to gain access to the value since otherwise, the *token* variable would be unknown and would generate an error.

However, in these cases of task dependency, it is also possible to flatten the operations. That is possible through Tuplas[^11] like so:

Example 5.4: CleanWayRx/Items/Item5.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token).map{saved in (token, saved)} }
    .concatMap { pair in self.network.getUser(username, pair.0) }
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

A transformation is used with the map operator to generate and propagate a tuple with the necessary values to the next task.

When the results of a task do not influence the execution of the following tasks, the result could be ignored. For example, in the following code:

Example 5.5: CleanWayRx/Items/Item5.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token) }
    .ignoreElements()
    .andThen(self.network.getUser(username))
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

The *getUser* task by not relying on the results of the previous tasks could ignore the elements.

Error handling

Item 28: Handling errors

The key to effective error management is to identify the strategy to react to these scenarios adequately.

When an error occurs in the stream, it could react in the following ways:

Scenario 1: Do nothing, do not handle the error, and allow the exception to propagate to the Observer with immediate termination of the operator chain execution and shutdown of the stream. That is the one that should never be followed.

Scenario 2: Capture the error, control it, and assign a custom value without turning off the stream, albeit with the immediate completion of the operator chain's execution.

Scenario 3: Catch the error, log it, and ignore it to suspend the operator chain's execution without turning off the stream.

Scenario 4: An error occurs, the task execution is retried a limited number of times (*n times*). If the error persists after retries, the operator chain's execution is captured, controlled, and suspended without turning off the stream.

Scenario 5: An error occurs, the task execution is retried a limited number of times (*n times*) every specific time window (*period*). If the error persists after retries, the operator chain's execution is captured, controlled, and suspended without turning off the stream.

Rx provides operators that allow designing different strategies to apply the mentioned scenarios.

Below are examples of the recommended strategies. It is emphasized that scenario one should be avoided.

Scenario 1

Example 28.1: CleanWayRx/Items/Item28.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
error: SampleError()
```

In this case, *storeTokenWithError* task will generate an error. The result is emission suspension, and only the *onError* contract is executed, not even *onComplete*. Remember that *onError* and *onComplete* are mutually exclusive, or one runs, or the other runs but not both.

Scenario 2

Example 28.2: CleanWayRx/Items/Item28.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .catchError({ error in Observable.just(User()) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
next: User(name: "", email: "", posts: [])
completed
```

In this case, the error is handled and customized (*User* is propagated empty or with default values). Also, the contracts *onNext* and *onComplete* are reached.

It should also be noted that the *getUser* task cannot be executed since once the error is generated in *storeTokenWithError*, the execution of the operator chain is suspended.

Does the location of the error operator matter?

Yes. Capturing the error is done at the point where the operator is defined. In a chain of operators, it is recommended to locate it just before the subscription.

Scenario 3

Example 28.3: CleanWayRx/Items/Item28.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
SampleError()
completed
```

In this case, when the error occurs, the trace is left in logs, and the emission continues, ignoring the value.

Scenario 4

Example 28.4: CleanWayRx/Items/Item28.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .retry(2)
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
SampleError()
completed
```

Similar to the previous scenario, only the *retry* operator is added. It is crucial always to specify the number of attempts since if this parameter is not specified, the retry could be permanent and counterproductive for the application's performance.

Scenario 5

Example 28.5: CleanWayRx/Items/Item28.swift

```swift
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .retryWhen{ errors in errors
        .do(onNext: { ignored in print("retrying...") })
        .delay(.seconds(2), scheduler: MainScheduler.instance)
        .take(4)
        .concat(Observable.error(SampleError()))}
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
retrying...
retrying...
retrying...
retrying...
retrying...
SampleError()
completed
```

Code Snippet: RxJS - RxJava

RxJS code snippet

Ítem 1: From Imperative to Reactive

Code Snippet: RxSwift - RxJava

Example 1.1: of operator - clean-way-rx/src/app/items/item1.ts

```
of(["h", "e"], ["l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: (2) ["h", "e"]
next: (3) ["l", "l", "o"]
```

Example 1.2: just (of with single emision) operator - clean-way-rx/src/app/items/item1.ts

```
of(["h", "e", "l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: (5) ["h", "e", "l", "l", "o"]
```

Example 1.3: from operator - clean-way-rx/src/app/items/item1.ts

```
from(["h", "e", "l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: h
next: e
next: l
next: l
next: o
```

Example 1.4: range operator - clean-way-rx/src/app/items/item1.ts

```
range(1, 5)
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: 1
next: 2
next: 3
next: 4
next: 5
```

Example 1.5: defer operator - clean-way-rx/src/app/items/item1.ts

```
defer(() =>
    of(["h", "e", "l", "l", "o"]))
    .subscribe(
        next => console.log('next:', next));

Console output:
---------------
next: (5) ["h", "e", "l", "l", "o"]
```

Operadores de apoyo

Example 1.6: interval operator - clean-way-rx/src/app/items/item1.ts

```
const subscribe = interval(1000)
    .subscribe(
        next => console.log('next:', next));

setTimeout(() => {
    subscribe.unsubscribe();
}, 5000);

Console output:
---------------
next: 0
next: 1
next: 2
next: 3
next: 4
```

Example 1.7: timer operator - clean-way-rx/src/app/items/item1.ts

```
const subscribe = timer(1000, 1000)
    .subscribe(
        next => console.log('next:', next));

setTimeout(() => {
    subscribe.unsubscribe();
}, 5000);


Console output:
---------------
next: 0
next: 1
next: 2
next: 3
next: 4
```

Example 1.8: empty operator - clean-way-rx/src/app/items/item1.ts

```
empty()
    .subscribe({
        next: () => console.log('Next'),
        complete: () => console.log('Complete!')
    });


Console output:
---------------
Complete!
```

Example 1.9: never operator - clean-way-rx/src/app/items/item1.ts

```
never()
    .subscribe({
        next: () => console.log('Next'),
        complete: () => console.log('Complete!')
    });
```

Example 1.10: error operator - clean-way-rx/src/app/items/item1.ts

```
throwError('SampleError!')
    .subscribe({
        next: val => console.log(val),
        complete: () => console.log('Complete!'),
        error: val => console.log(`Error: ${val}`)
    });


Console output:
---------------
Error: SampleError!
```

Example 1.11: clean-way-rx/src/app/items/item1.ts

```
taskWrapped(): Observable<any> {

    return Observable.create(function (observer: any) {
        // Here the imperative code is embedded
    });
}
```

Code Snippet: RxSwift - RxJava

Example 1.12: clean-way-rx/src/app/items/item1.ts

```
imperativeTask() {
    const data = "any data";
    console.log("Do any imperative task or process");
    return data;
}
```

Example 1.13: clean-way-rx/src/app/items/item1.ts

```
taskWrapped(task: any): Observable<any> {

    return Observable.create(function (observer: any) {

        let data = task;
        if (data) {
            observer.next(data);
            observer.complete();
        } else {
            observer.error("SampleError");
        }
    });
}
```

Code Snippet: RxSwift - RxJava

Example 1.14: clean-way-rx/src/app/items/item1.ts

```
this.taskWrapped(this.imperativeTask())
    .subscribe({
        next: val => console.log(val),
        complete: () => console.log('Complete!'),
        error: val => console.log(`Error: ${val}`)
    });

Console output:
---------------
Do any imperative task or process
any data
Complete!
```

Ítem 5: Keep clean the operators' chain

Code Snippet: RxSwift - RxJava

Example 5.1: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)
            .pipe(saved => this.network.getUser(username)))
    )
    .subscribe(user => console.log(user));
```

Example 5.2: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)),
        concatMap(saved => this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Example 5.3: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)
            .pipe(saved => this.network.getUserWithToken(username, token)))
    )
    .subscribe(user => console.log(user));
```

Example 5.4: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
  .pipe(
    concatMap(token => this.cache.storeToken(token).pipe(map(saved => [token, saved]))),
    concatMap(pair => this.network.getUserWithToken(username, pair[0] as Token))
  )
  .subscribe(user => console.log(user));
```

Example 5.5: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)),
        ignoreElements(),
        concat(this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Ítem 28: Handling errors

Code Snippet: RxSwift - RxJava

Example 28.1: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Example 28.2: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        catchError(error => of(new User())))
    .subscribe(
        user => console.log('next:', user),
        error => console.log('error:', error),
        () => console.log('completed'));
```

Example 28.3: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

Example 28.4: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        retry(2),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

Example 28.5: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        retryWhen(errors => errors.pipe(
            tap(() => console.log('retrying...')),
            delay(2000),
            take(4),
            concat(throwError('SampleError!'))
        )),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

RxJava code snippet

Ítem 1: From Imperative to Reactive

Code Snippet: RxSwift - RxJS

Example 1.1: of (fromArray) operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
    Observable.fromArray(Arrays.asList("h", "e"), Arrays.asList("l", "l", "o"))
            .subscribe(element -> Log.d(TAG, "" + element)));

Console output:
---------------
D/Item1: [h, e]
D/Item1: [l, l, o]
```

Example 1.2: just operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.just(Arrays.asList("h", "e", "l", "l", "o"))
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: [h, e, l, l, o]
```

Example 1.3: from (fromIterable) operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.fromIterable(Arrays.asList("h", "e", "l", "l", "o"))
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: h
D/Item1: e
D/Item1: l
D/Item1: l
D/Item1: o
```

Example 1.4: range operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.range(1, 5)
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: 1
D/Item1: 2
D/Item1: 3
D/Item1: 4
D/Item1: 5
```

Example 1.5: defer operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.defer(() ->
                Observable.just(Arrays.asList("h", "e", "l", "l", "o")))
                .subscribe(element -> Log.d(TAG, "" + element)));

Console output:
---------------
D/Item1: [h, e, l, l, o]
```

Operadores de apoyo

Example 1.6: interval operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.interval(1, TimeUnit.SECONDS)
                .subscribe(element -> Log.d(TAG, "" + element)));

sleep(5000);

Console output:
---------------
D/Item1: 0
D/Item1: 1
D/Item1: 2
D/Item1: 3
D/Item1: 4
```

Example 1.7: timer operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.timer(1, TimeUnit.SECONDS)
                .subscribe(element -> Log.d(TAG, "" + element),
                           throwable -> Log.e(TAG, "error:" + throwable),
                           () -> Log.d(TAG, "completed")));

sleep(5000);


Console output:
---------------
D/Item1: 0
D/Item1: completed
```

Example 1.8: empty operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.empty()
                .subscribe(element -> Log.d(TAG, "" + element),
                           throwable -> Log.e(TAG, "error:" + throwable),
                           () -> Log.d(TAG, "completed")));


Console output:
---------------
D/Item1: completed
```

Example 1.9: never operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.never()
                .subscribe(element -> Log.d(TAG, "" + element),
                           throwable -> Log.e(TAG, "error:" + throwable),
                           () -> Log.d(TAG, "completed")));


Console output:
---------------
```

Example 1.10: error operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.error(() -> {
            throw new Exception("SampleError!");
        })
        .subscribe(element -> Log.d(TAG, "" + element),
                   throwable -> Log.e(TAG, "error:" + throwable),
                   () -> Log.d(TAG, "completed")));


Console output:
---------------
E/Item1: error:java.lang.Exception: SampleError!
```

Example 1.11: CleanWayRx/app/src/main/java/../items/Item1.java

```
private Observable<String> taskWrapped() {

    return Observable.create(emitter -> {
        // Here the imperative code is embedded
    });
}
```

Code Snippet: RxSwift - RxJS

Example 1.12: CleanWayRx/app/src/main/java/../items/Item1.java

```
private String imperativeTask() {

    String data = "Any data";
    Log.d(TAG, "Do any imperative task or process");
    return data;
}
```

Example 1.13: CleanWayRx/app/src/main/java/../items/Item1.java

```
private Observable<String> taskWrapped(final Object task) {

    return Observable.create(emitter -> {
        try {
            String data = (String) task;
            emitter.onNext(data);
            emitter.onComplete();
        } catch (Throwable e) {
            emitter.onError(e);
        }
    });
}
```

Code Snippet: RxSwift - RxJS

Example 1.14: CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        taskWrapped(this.imperativeTask())
                .subscribe(data -> Log.d(TAG, "next: " + data),
                           throwable -> Log.e(TAG, "error: " + throwable),
                           () -> Log.d(TAG, "completed")));

Console output:
---------------
D/Item1: Do any imperative task or process
D/Item1: next: Any data
D/Item1: completed
```

Ítem 5: Keep clean the operators' chain

Code Snippet: RxSwift - RxJS

Example 5.1: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .concatMap(saved -> network.getUser(username)))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.2: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token))
                .concatMap(saved -> network.getUser(username))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.3: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .concatMap(saved -> network.getUser(username, token)))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.4: CleanWayRx/app/src/main/java/../items/Item5.java

```java
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .map(saved -> new Pair<>(token, saved)))
                .concatMap(pair -> network.getUser(username, pair.first))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.5: CleanWayRx/app/src/main/java/../items/Item5.java

```java
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token))
                .ignoreElements()
                .andThen(network.getUser(username))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Ítem 28: Handling errors

Code Snippet: RxSwift - RxJS

Example 28.1: CleanWayRx/app/src/main/java/../items/Item28.java

```java
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .subscribe(element -> Log.d(TAG, "" + element),
                       throwable -> Log.e(TAG, "error:" + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.2: CleanWayRx/app/src/main/java/../items/Item28.java

```java
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .onErrorResumeNext(throwable -> Observable.just(new User()))
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.3: CleanWayRx/app/src/main/java/../items/Item28.java

```java
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.4: CleanWayRx/app/src/main/java/../items/Item28.java

```java
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .retry(2)
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.5: CleanWayRx/app/src/main/java/../items/Item28.java

```java
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .retryWhen(errors -> errors
                    .doOnNext(ignored -> Log.d(TAG, "retrying..."))
                    .delay(2, TimeUnit.SECONDS)
                    .take(4)
                    .concatWith(Observable.error(new Throwable())))
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```