Badass datasource

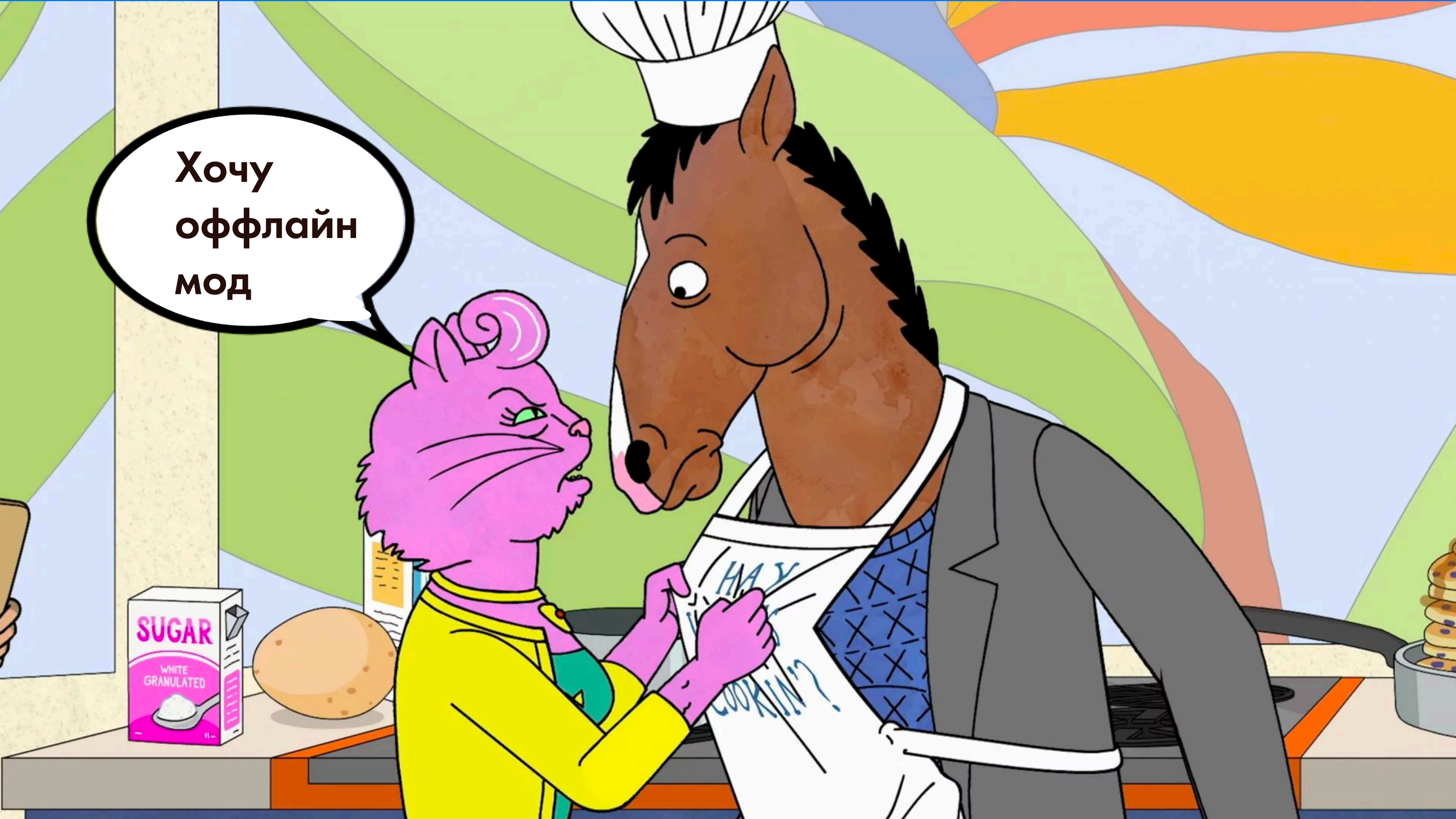
Alexey Bykov

Alexey Bykov

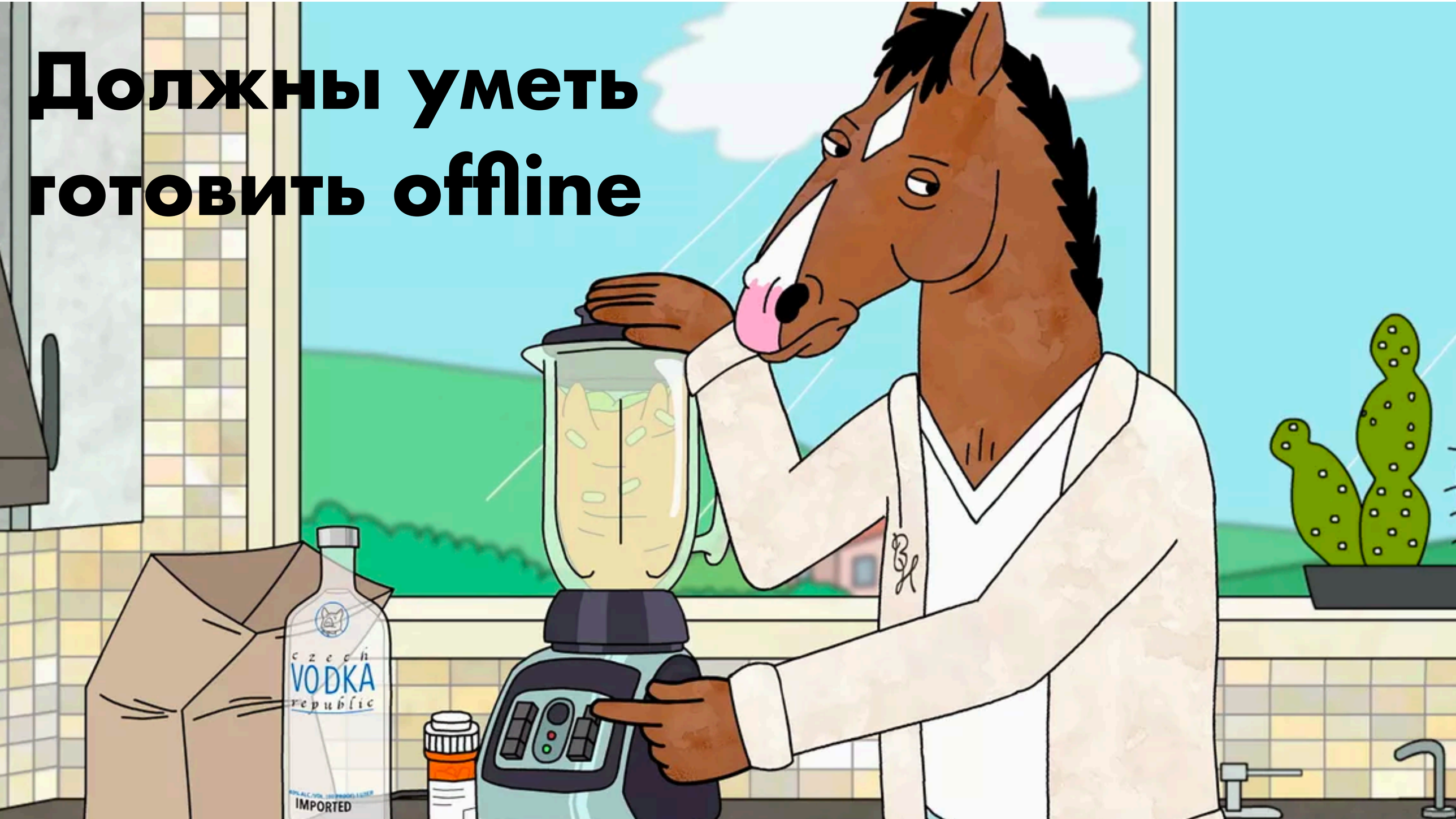
Android Software Engineer @Revolut

- Ex-hockey-player
- 4 years in Android
- EX: Programm Committee of Podlodka
- EX: AppsConf, Android Academy MSK

@nonewsss

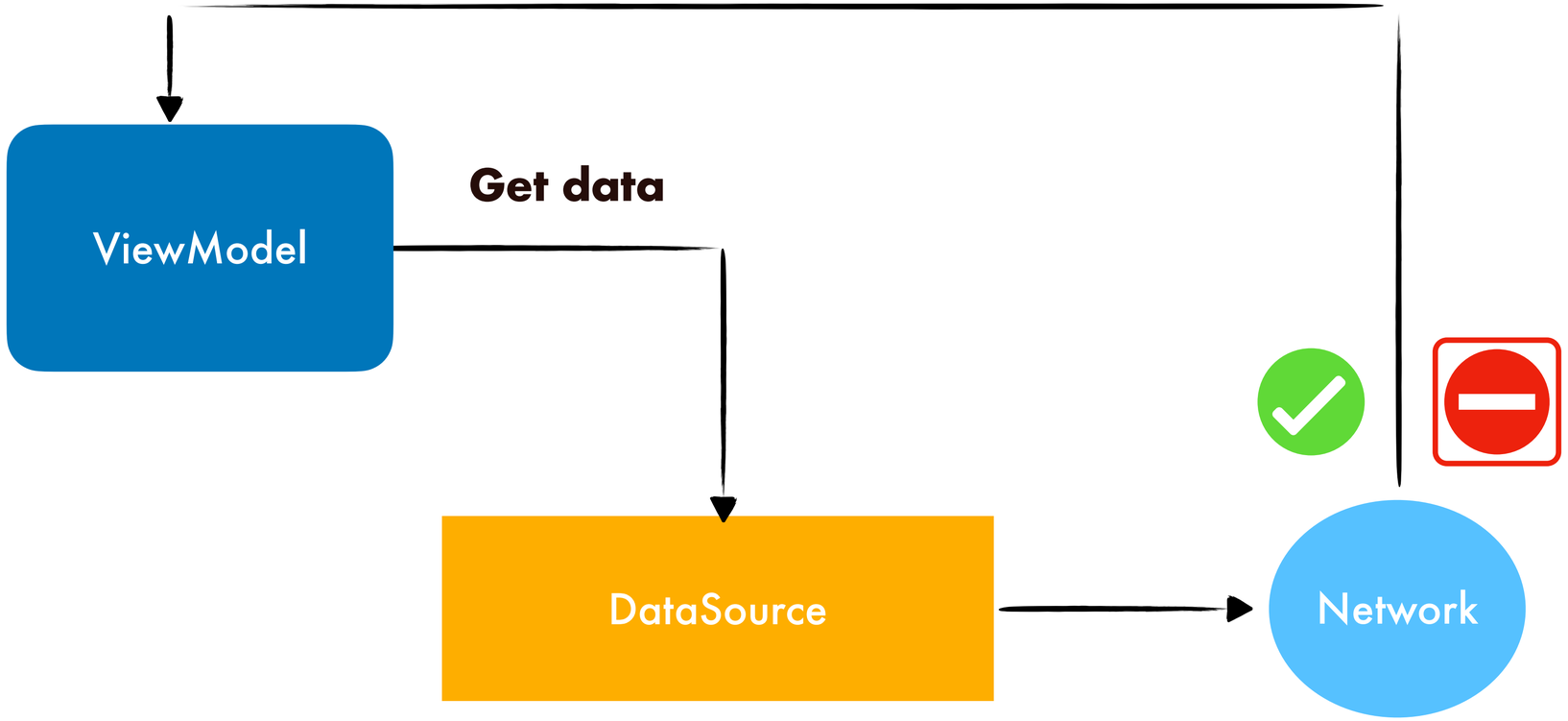

Хочу оффлайн мод
SUGAR
WHITE GRANULATED

Должны уметь готовить offline

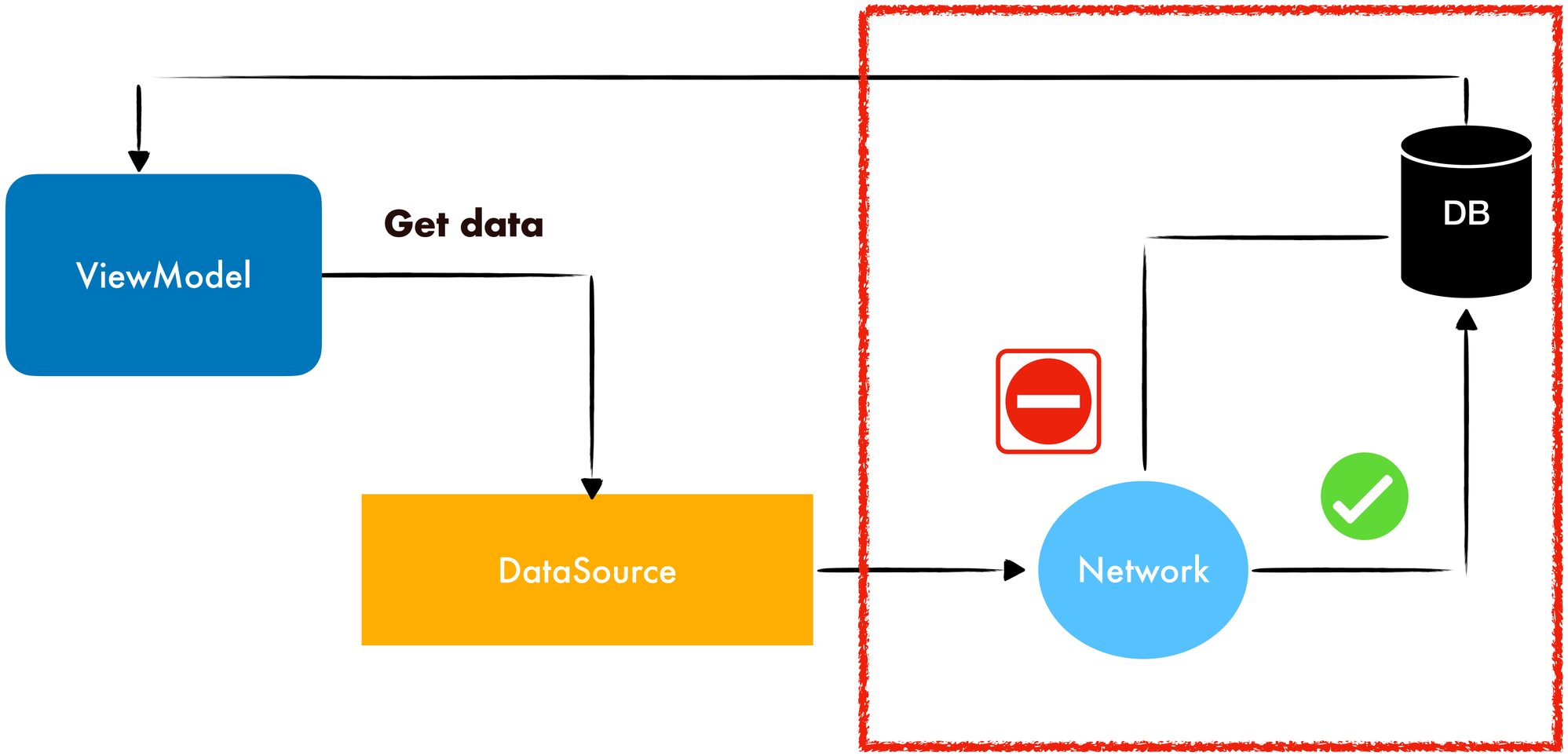


Способы работы с данными

1. Network only



2. Network first



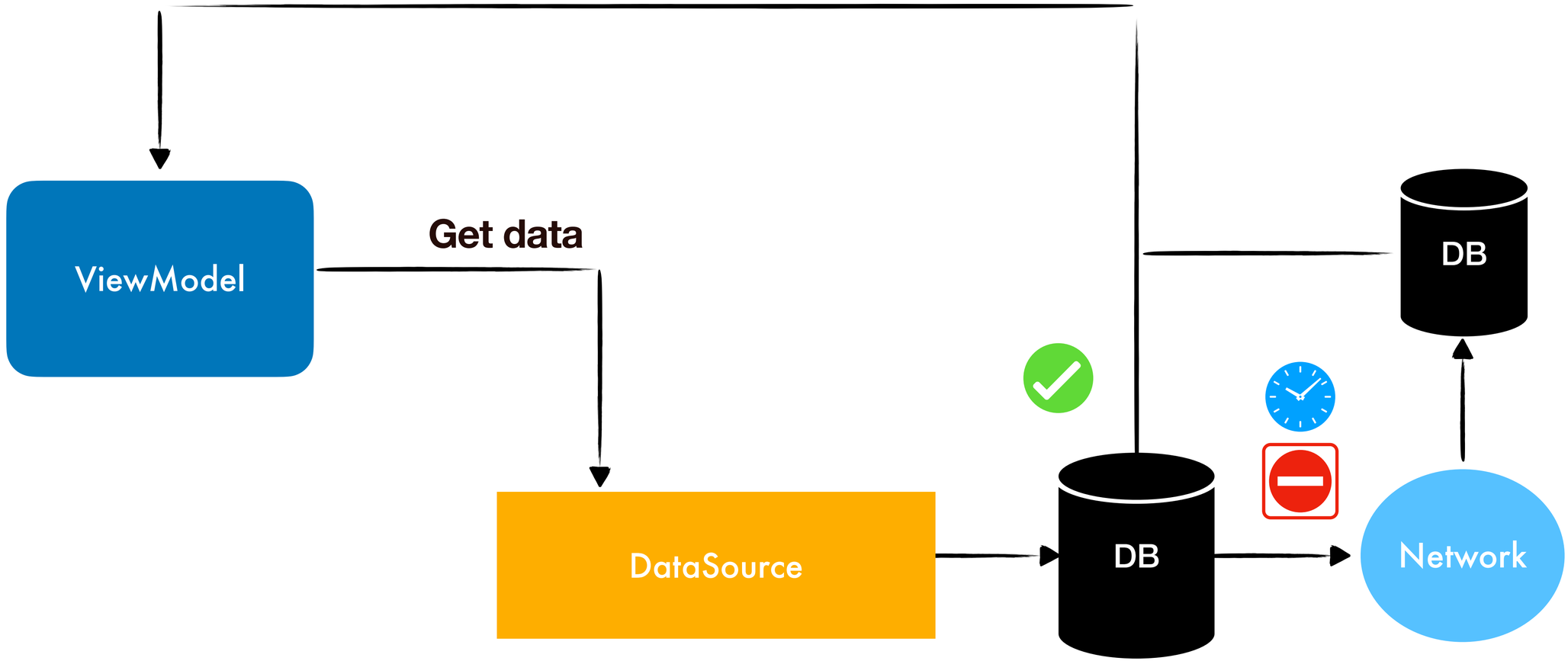
2. Network first

- Пользователь ждёт
- Тяжело масштабировать
- Лишние http-запросы

+ Прагматично для MVP

+ Актуальные данные

3. Cache only

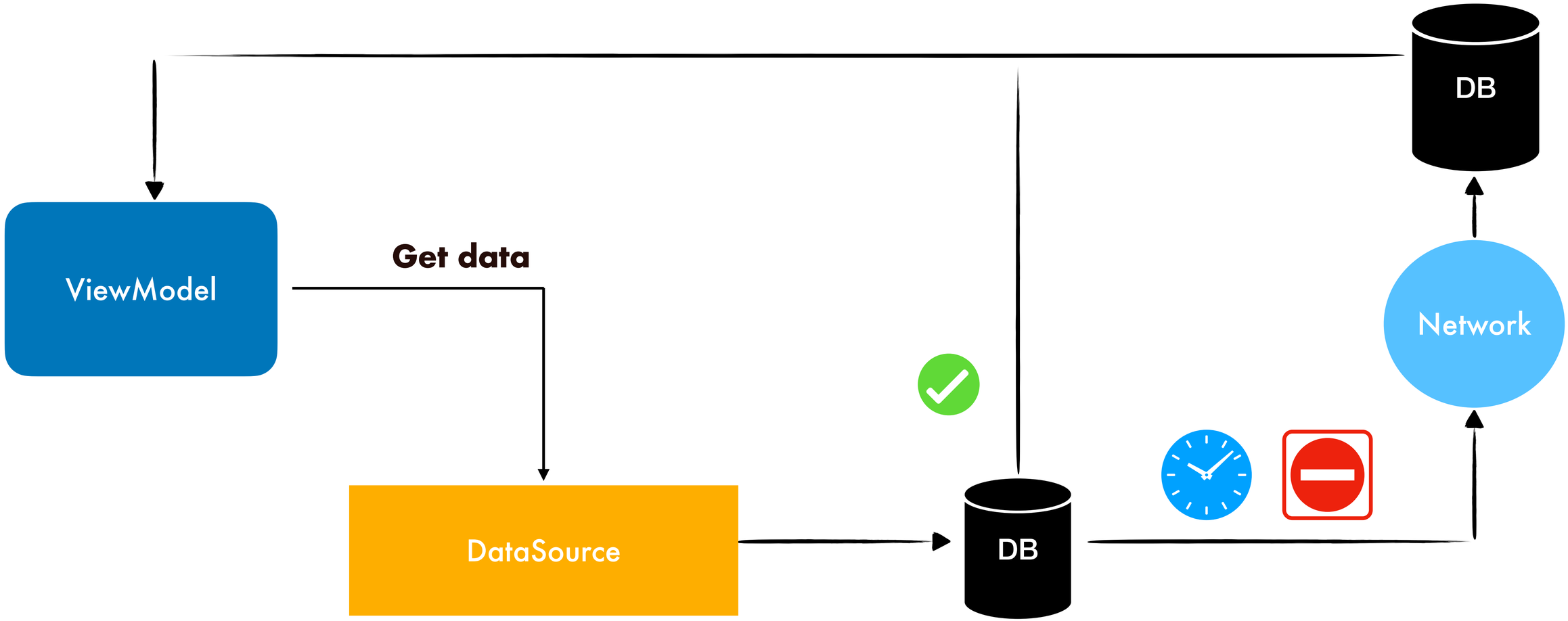


3. Cache only

- Данные неактуальны (иногда)
- Тяжело масштабировать
- Всегда показываем ProgressBar
- Моргания из-за лишнего трейдинга

+ Легко сделать
+ Всё ещё прагматично для MVP

4. Cache first



4. Cache first

– UI State diffing

– Легко выстрелить в ногу

\+ Отзывчивый UX

\+ Масштабируемо

“Какой подход мне выбрать?”

Всё зависит

Сроки

Ресурсы команды

Актуальность данных

...

Мы выбрали CacheFirst, с несколькими эмитами

Небольшой проект

Небольшая команда

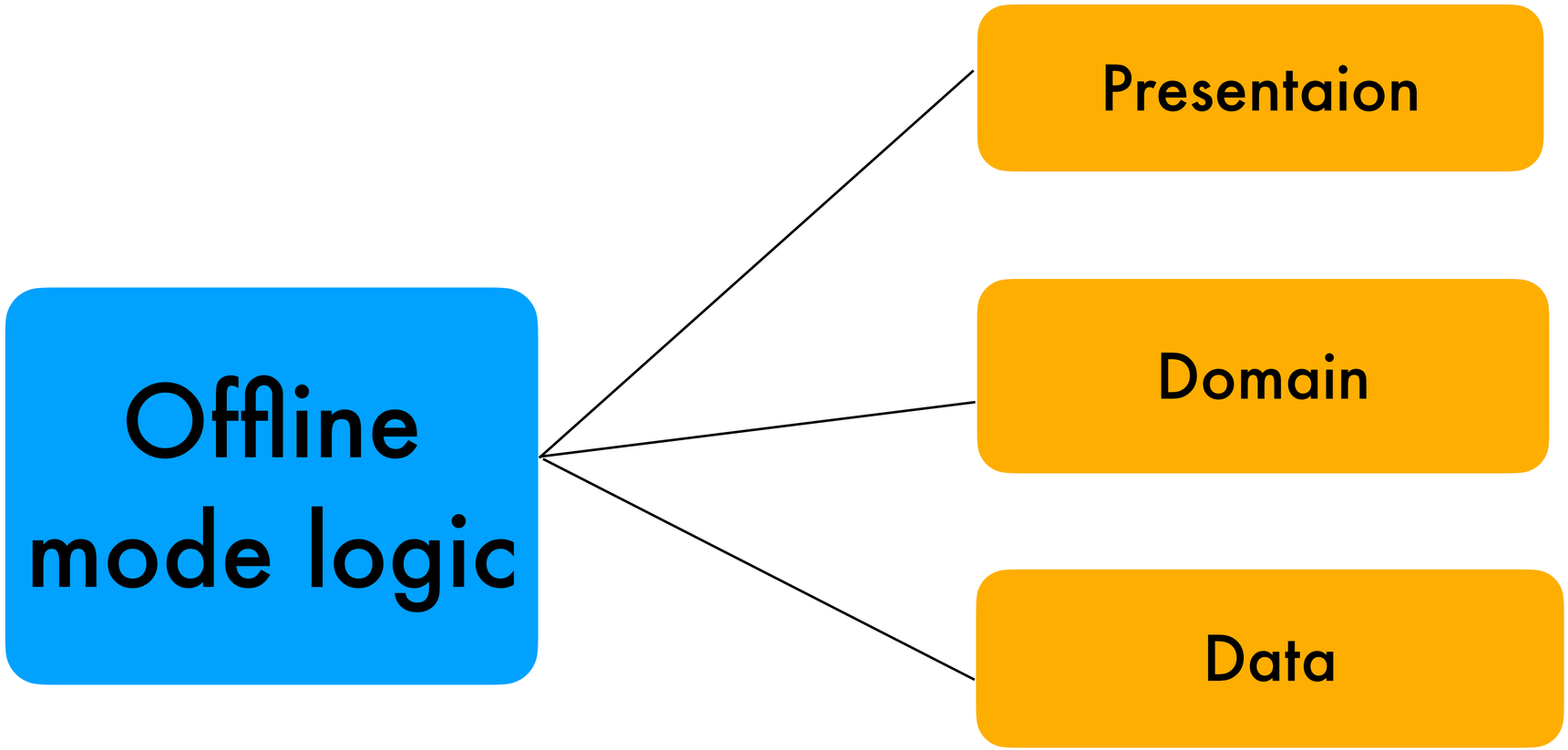
Пёс

Проект растёт

Растёт и команда

Проблемы

#1 DataSource в-е-з-д-е



#2 Разные подходы

BehaviorSubject

DBS concatMap network

Network stream + onErrorResumeNext DBS

…

#3 ErrorHandling везде?

Data layer

Domain layer

Presentation layer

View layer

Repository курильщика

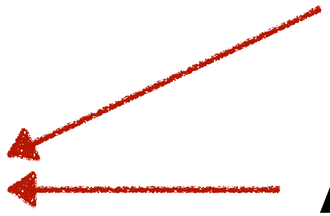
```
interface Repository {

    fun getUser(): Single<User>

}
```

Repository курильщика

```
interface Repository {

    fun getUser(): Single<User>

}
```

Repository курильщика

```
interface Repository {

    fun getUser(): Single<User>

}
```

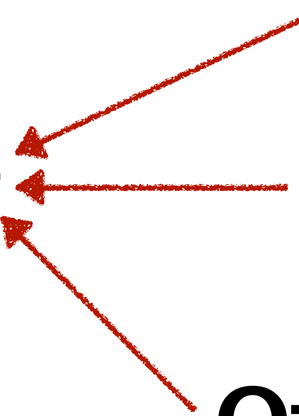


Данные свежие?

А ошибка возможна?

Repository курильщика

```
interface Repository {

    fun getUser(): Single<User>

}
```



Данные свежие?

А ошибка возможна?

Ответ синхронен?

Repository курильщика

```
interface Repository {

    fun getUser(): Single<User>

}
```

Данные свежие?

А ошибка возможна?

Ответ синхронен?

Боли #1

Дублирование Http-запросов

Лишний threading

Лишние loading-bars

Боли #2

Дублирование кода

Автобусный фактор

Медленная разработка

Думать больше, чем надо

Цель:
Показать решение этих болей

Вы узнаете:

1. Как делать это с Rx

2. Практики, проверенные на проде

3. Как Rx работает под капотом

Бонус:

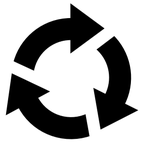
1. Как делать это с Rx/ Kotlin Flow

2. Практики, проверенные на проде

3. Как Rx/Kotlin Flow работает под капотом

Revolut

 3 Apps: Retail, Business, Junior

 > 400 модулей и > 150 Repository

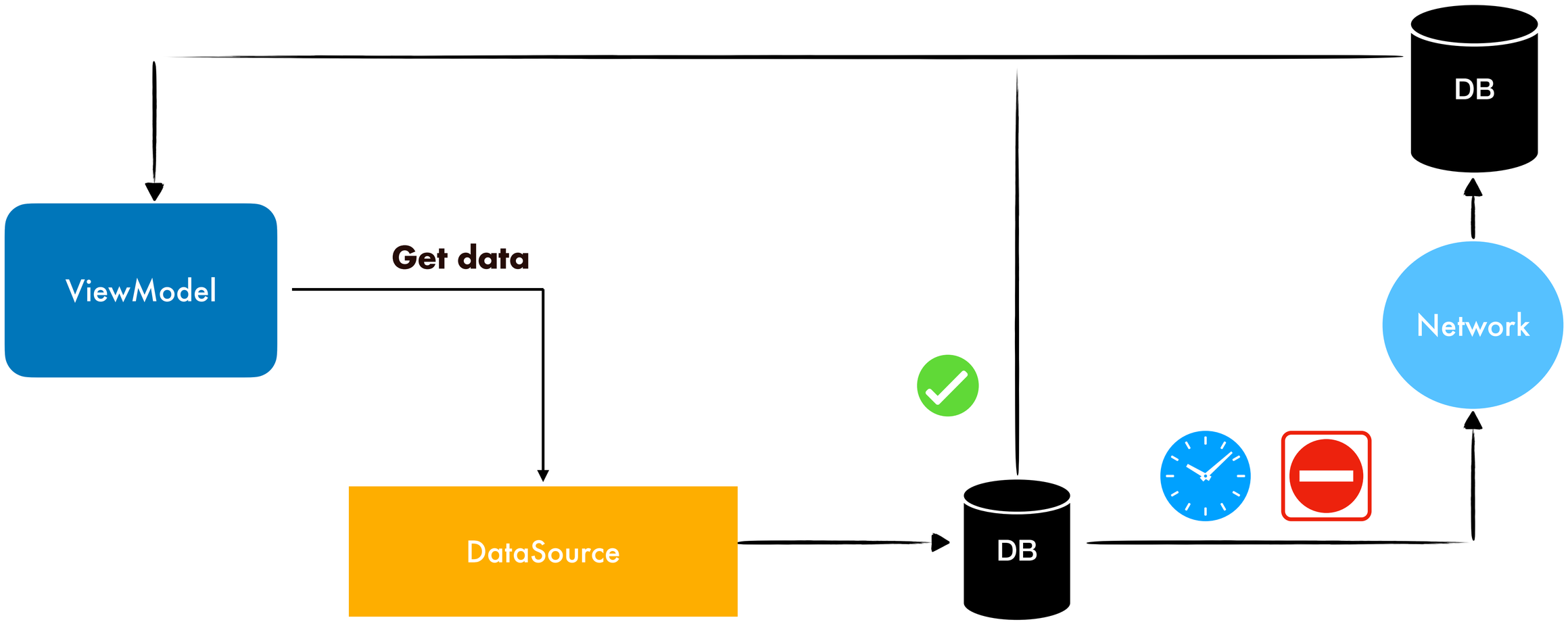
 99% на Kotlin

 63 devs  63 devs

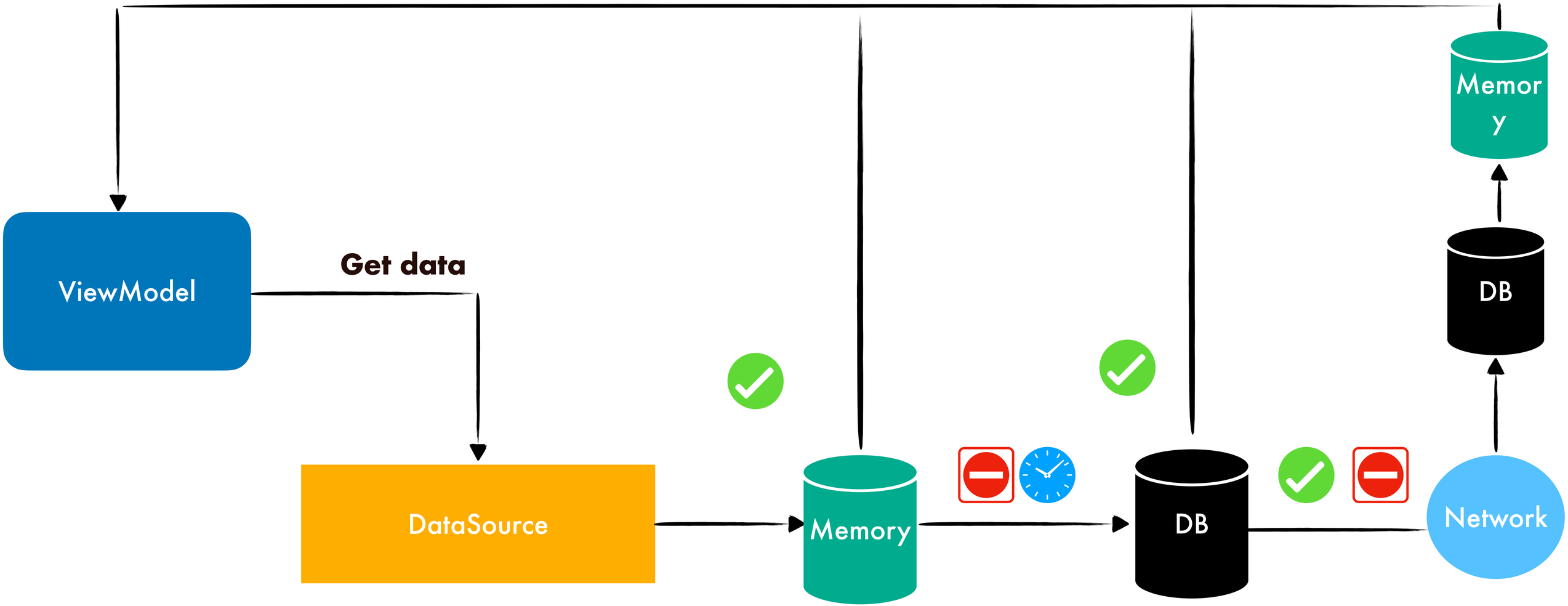
Поехали

1. Улучшаем стратегию кеширования

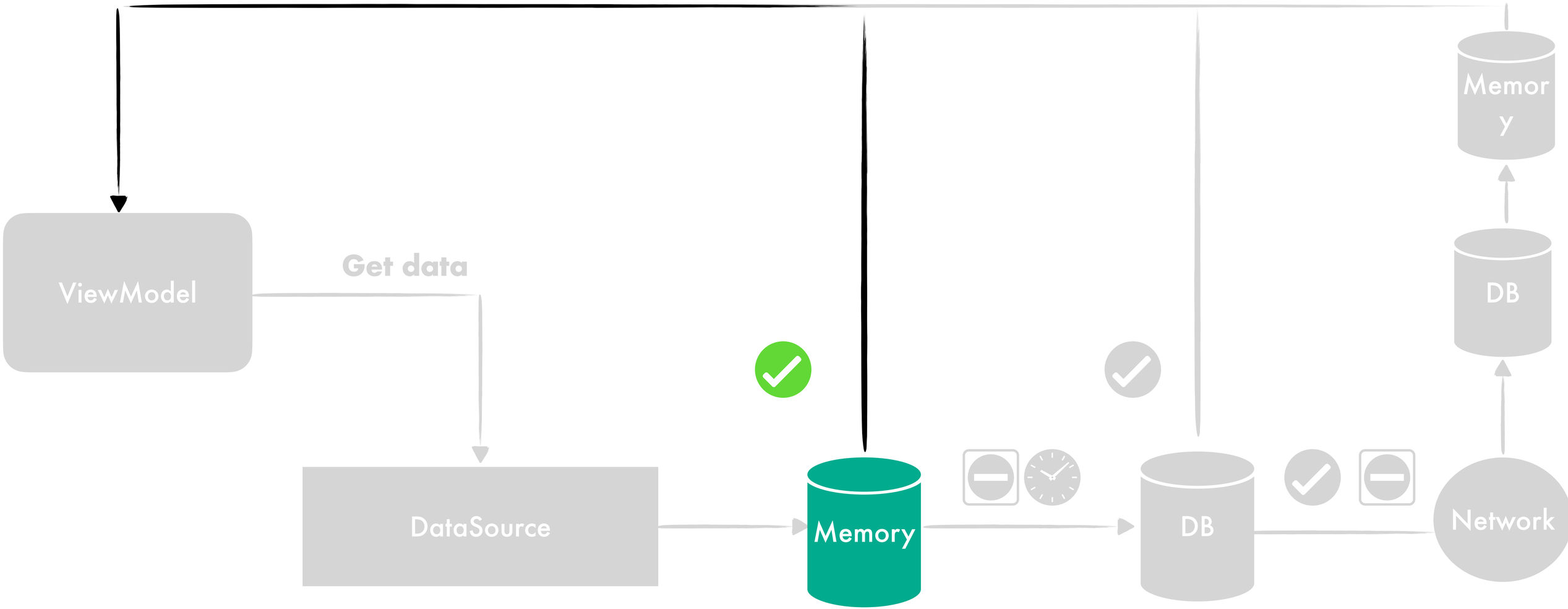
Cache first



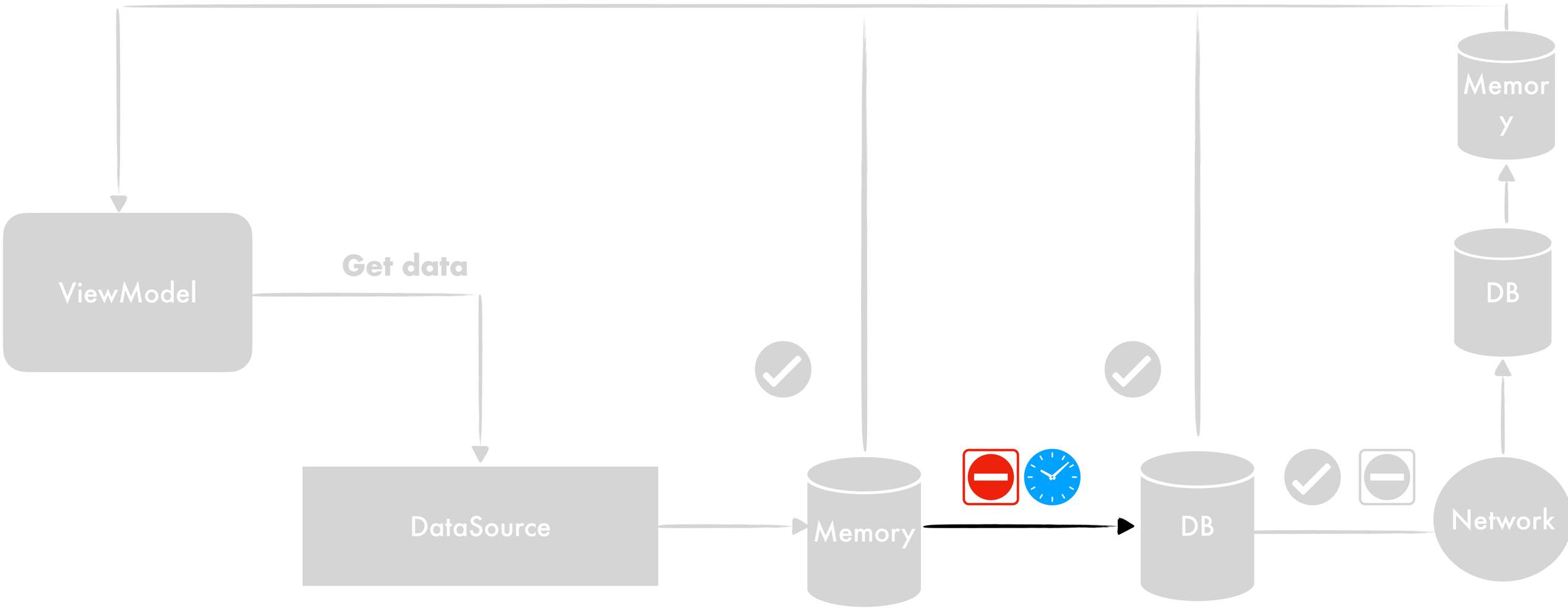
Two-level caching



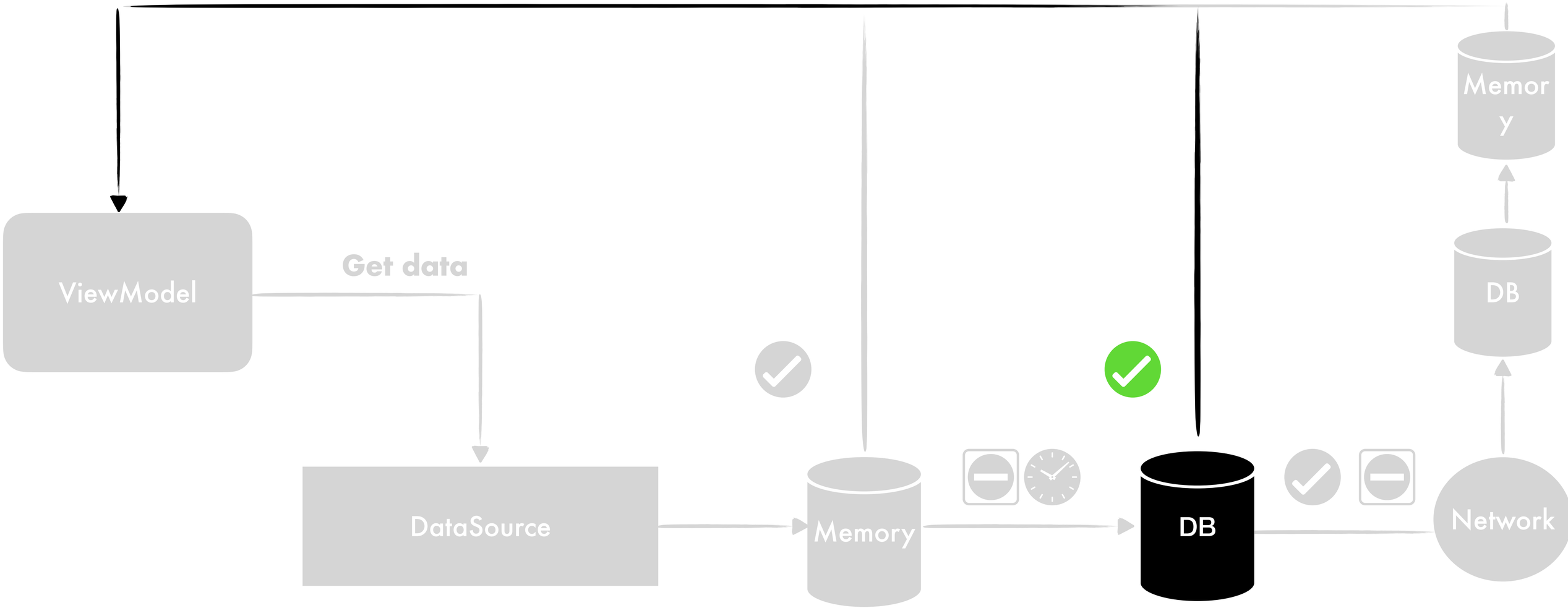
Из памяти если есть



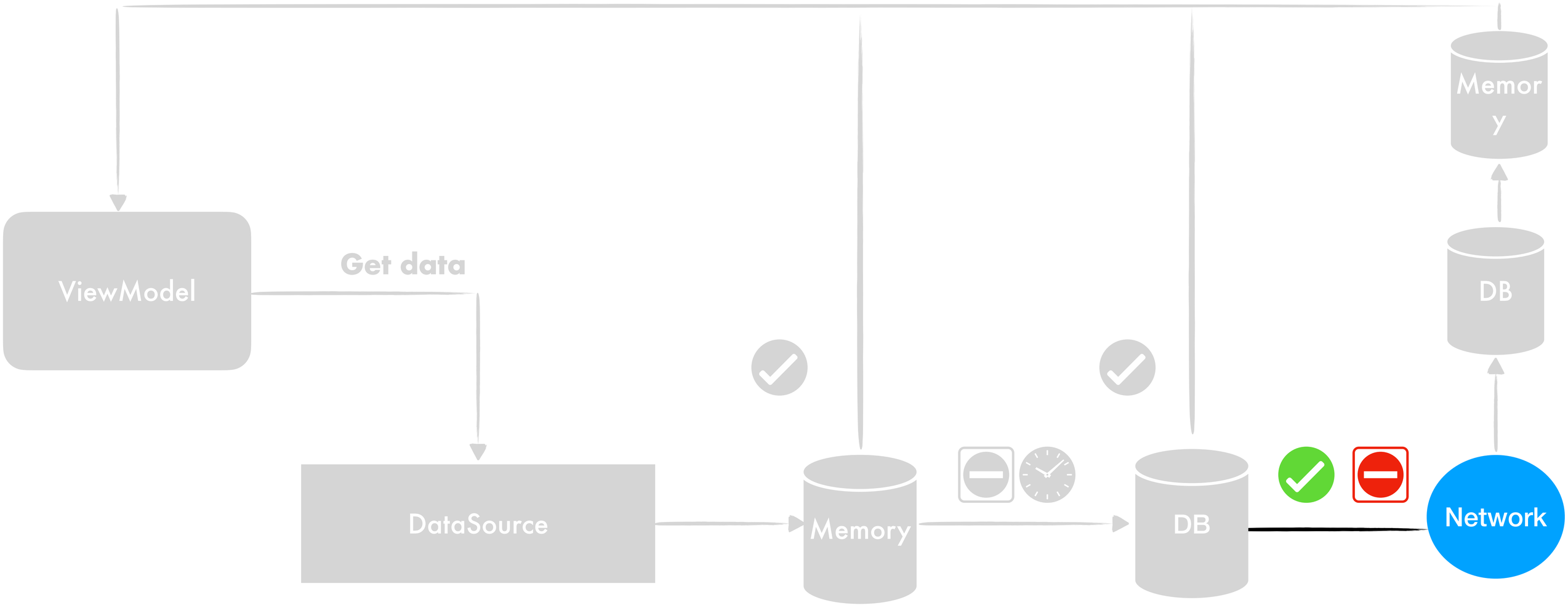
Empty/Expired?



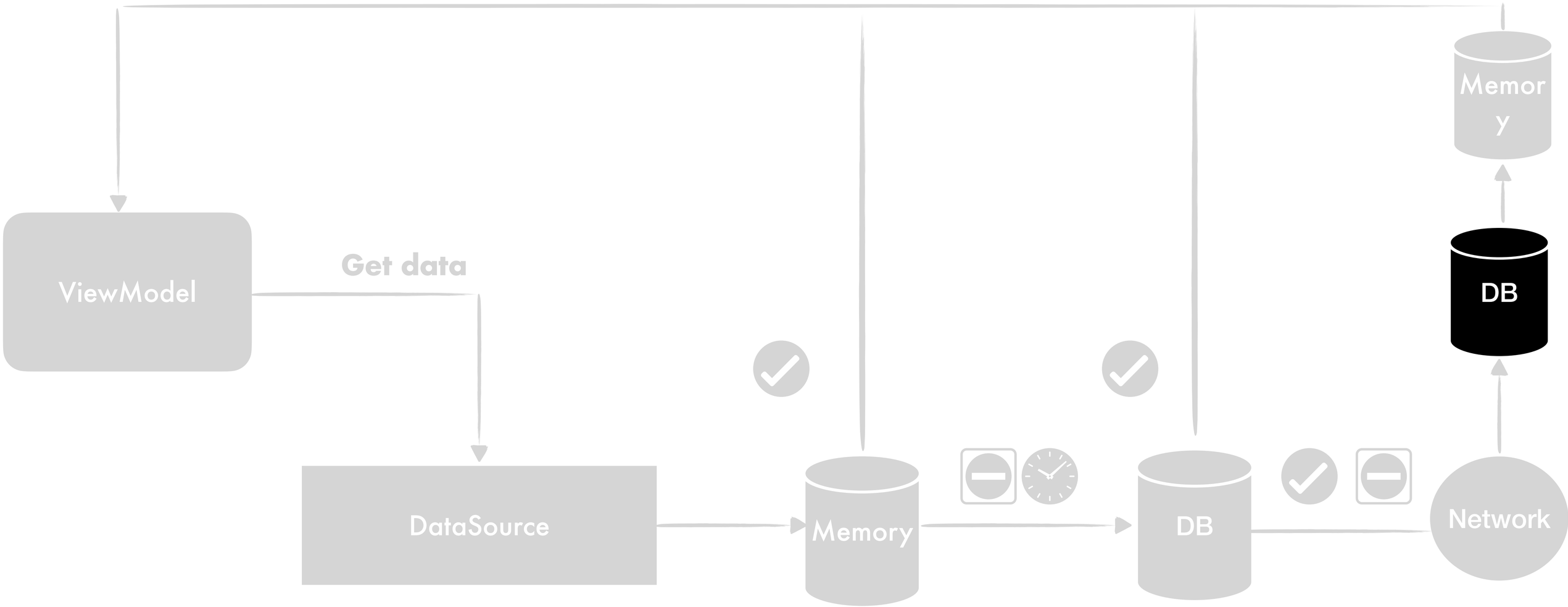
Из стораж



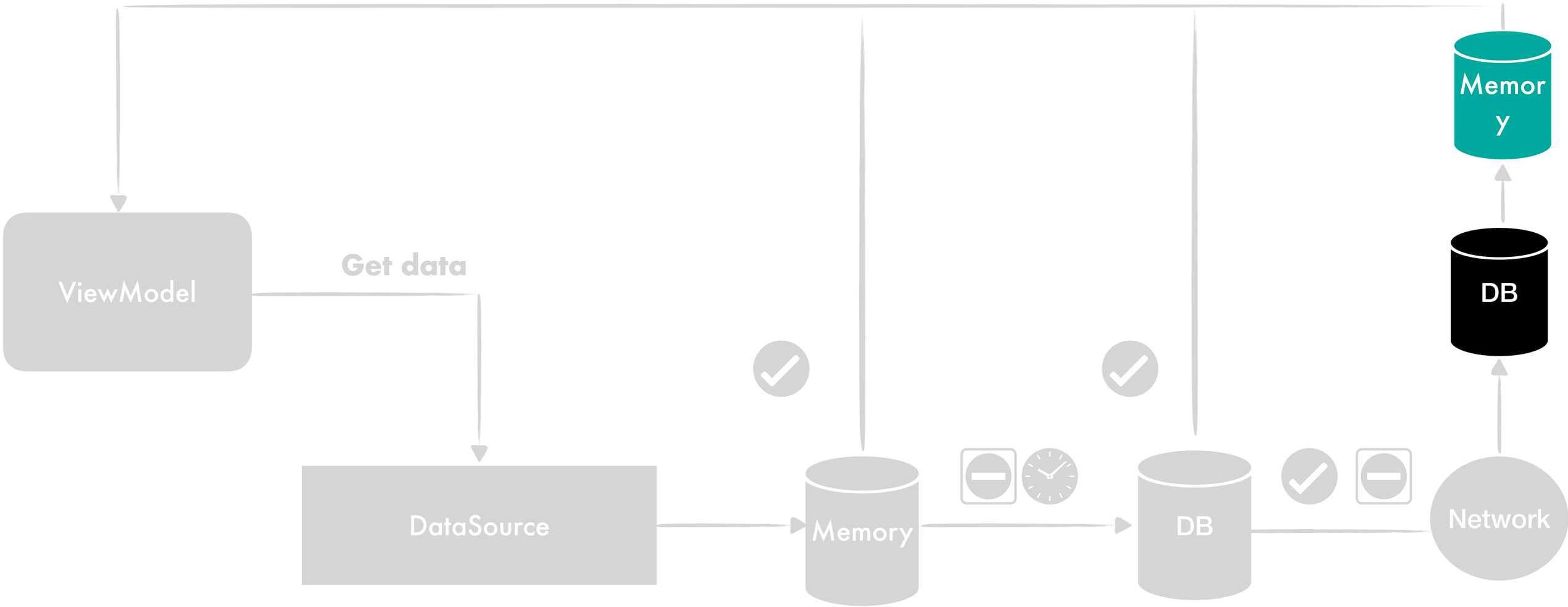
Идём в сеть



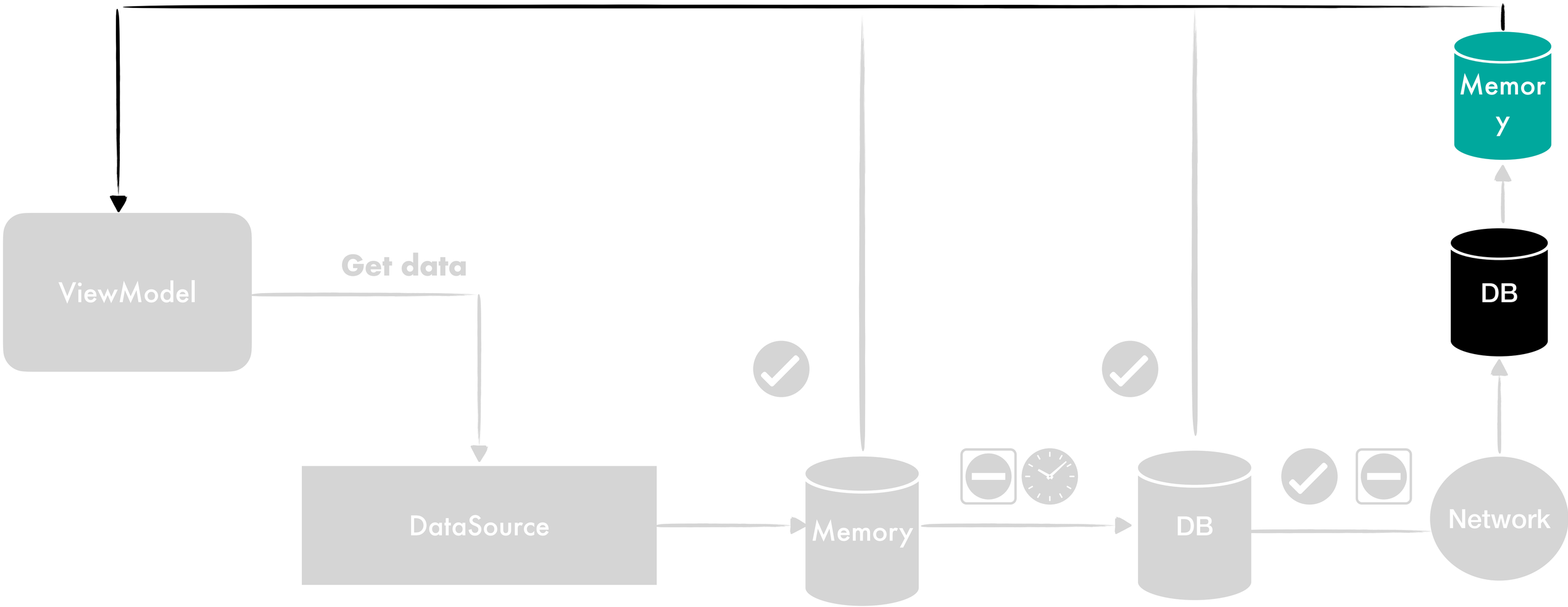
Пишем в стораж



Пишем в память



Отдаём снова



Two-level caching

1. Быстрее

2. Синхронное выполнение

Синхронный код на Rx?

Rx: Синхронное выполнение

```
print(“Before”)

Observable.just(1)
    .subscribe {
        print(“Body”)
    }

print("After")
```

Rx: Синхронное выполнение

```
print(“Before”)

Observable.just(1)
    .subscribe {
        print(“Body”)
    }

print("After")
```

1

Rx: Синхронное выполнение

```
print(“Before”)

Observable.just(1)
    .subscribe {
        print(“Body”)
    }

print(“After")
```

1

2

Rx: Синхронное выполнение

```
print(“Before”)

Observable.just(1)
    .subscribe {
        print(“Body”)
    }

print("After")
```

1

2

3

2. API для разработчиков

Repository здорового человека

```
interface Repository {

  fun observeUser(forceUpdate: Boolean): Observable<Data<User>>

}
```

Должны уметь обновлять

```
interface UserRepository {

  fun observeUser(forceUpdate: Boolean): Observable<Data<User>>

}
```

Предоставлять детали ответа

```
interface UserRepository {

  fun observeUser(forceUpdate: Boolean): Observable<Data<User>>

}
```

LCE-Pattern

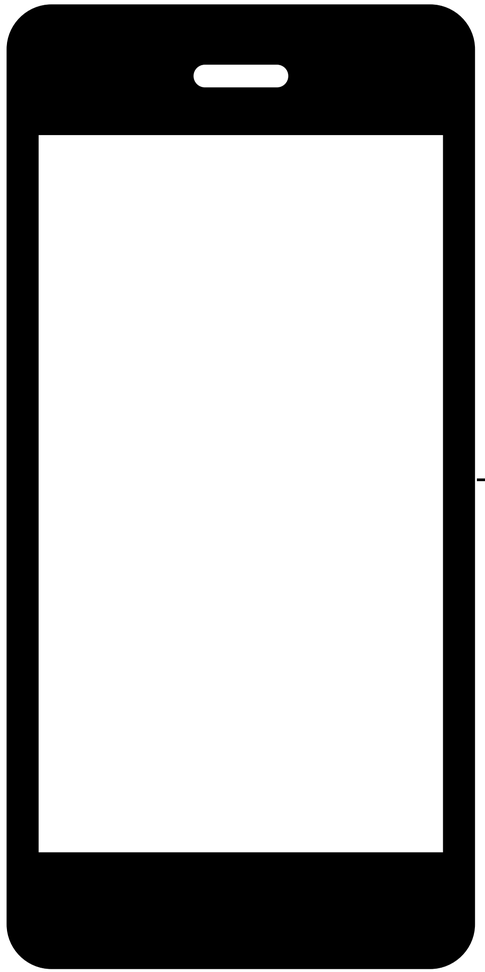
Loading, Content, Error Pattern

Simple implementation

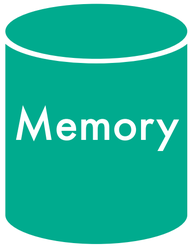
```
data class Data<out T>(
    val content: T? = null,
    val error: Throwable? = null,
    val loading: Boolean = false
)
```

Но зачем?

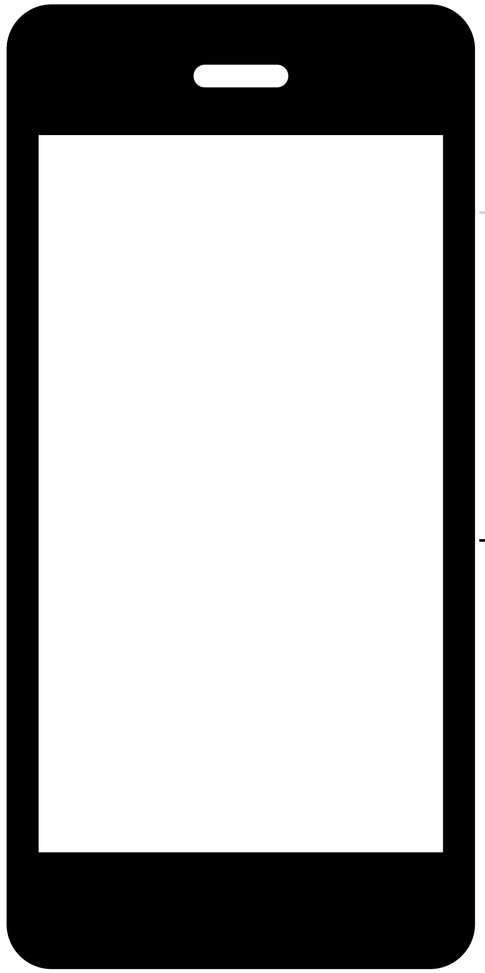
Открыли экран — данные из кеша



force reload = false



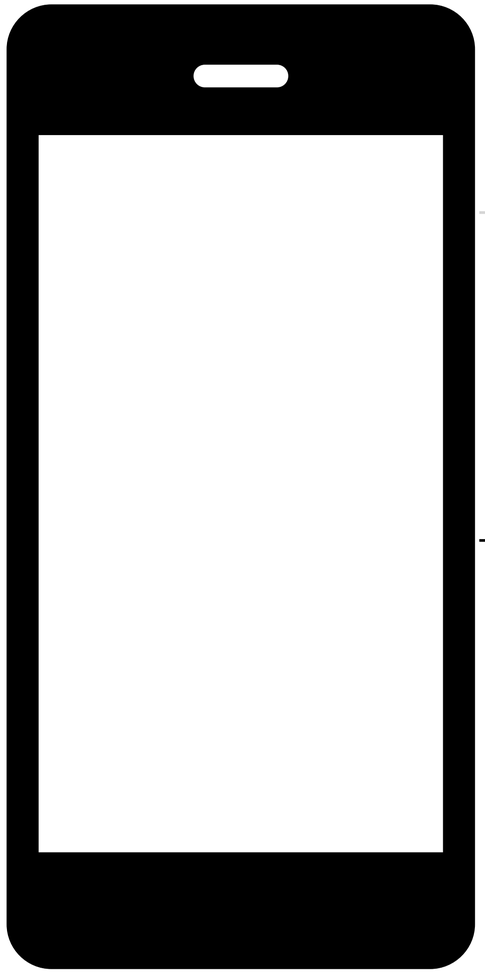
Данных в Кеше нет :/



force reload = false

force reload = false ¯_(ツ)_/¯

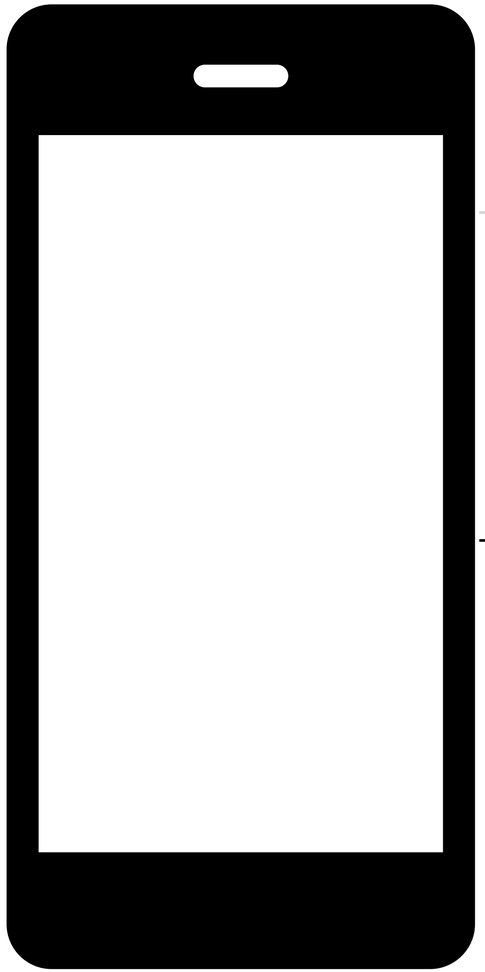
Показываем загрузку



force reload = false

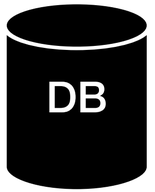
force reload = false ¯_(ツ)_/¯

Показываем данные из DB

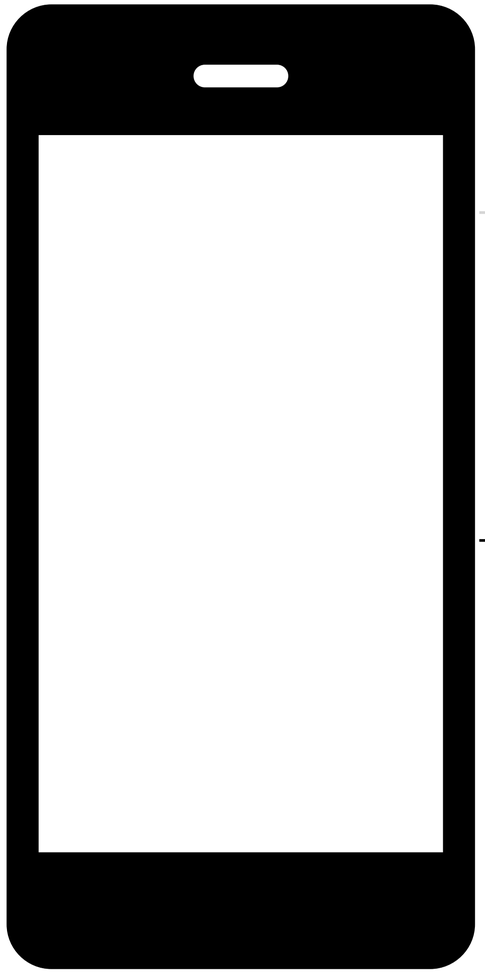


force reload = false

force reload = false ¯_(ツ)_/¯

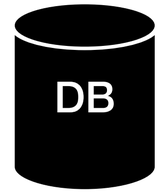


Показываем данные из сети

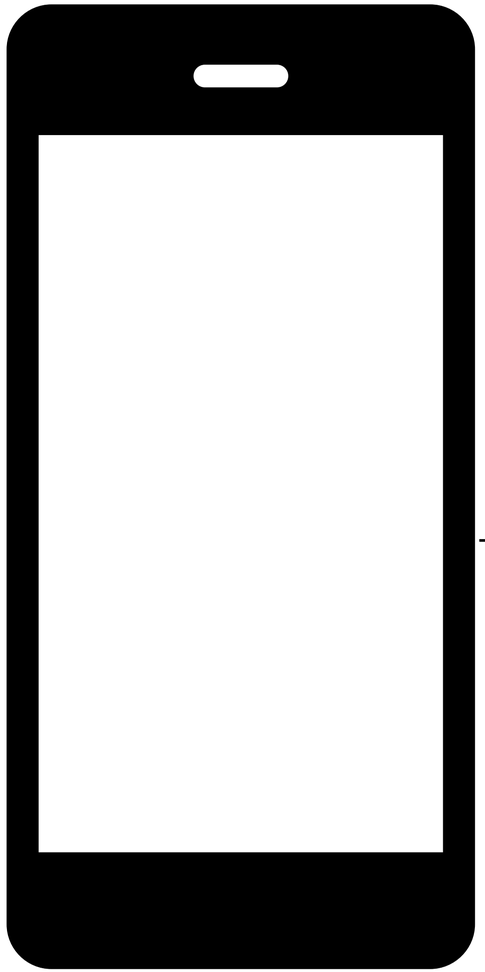


force reload = false

force reload = false ¯_(ツ)_/¯

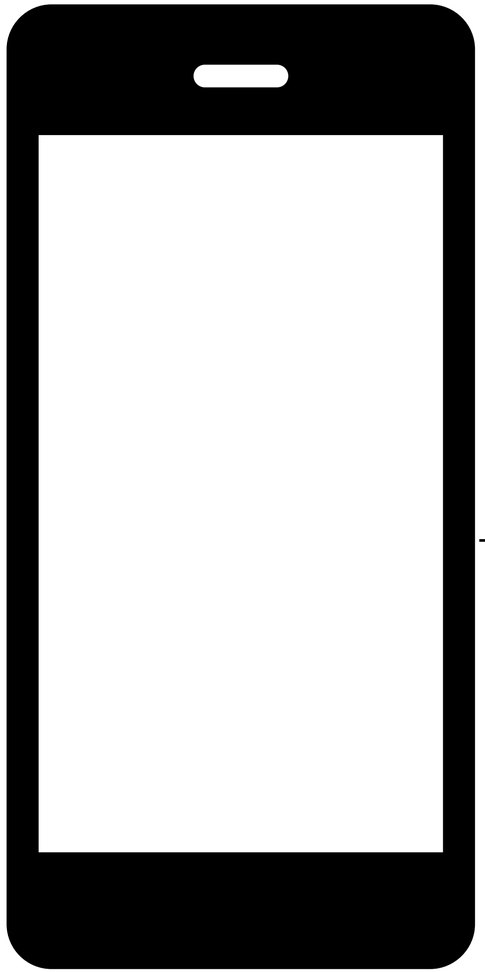


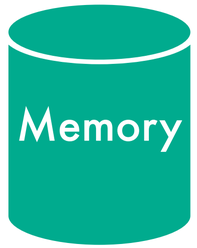
ForceReload = true



force reload = true

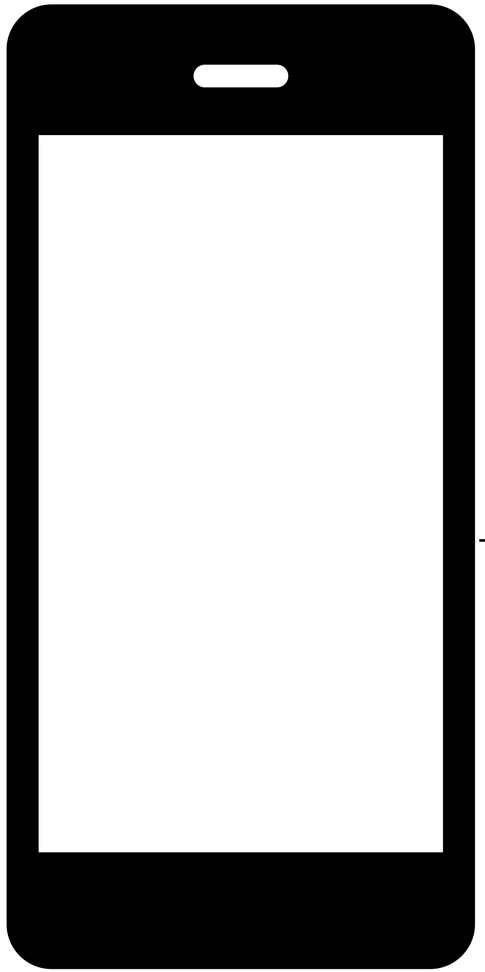
Кеш с загрузкой



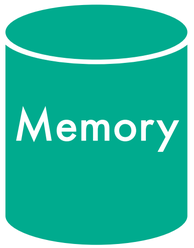


force reload = true

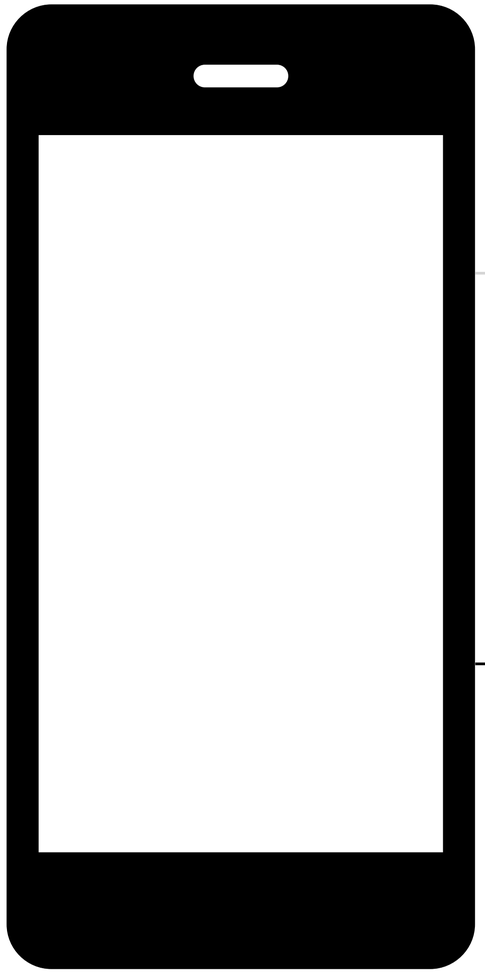
Данные из сети



force reload = true



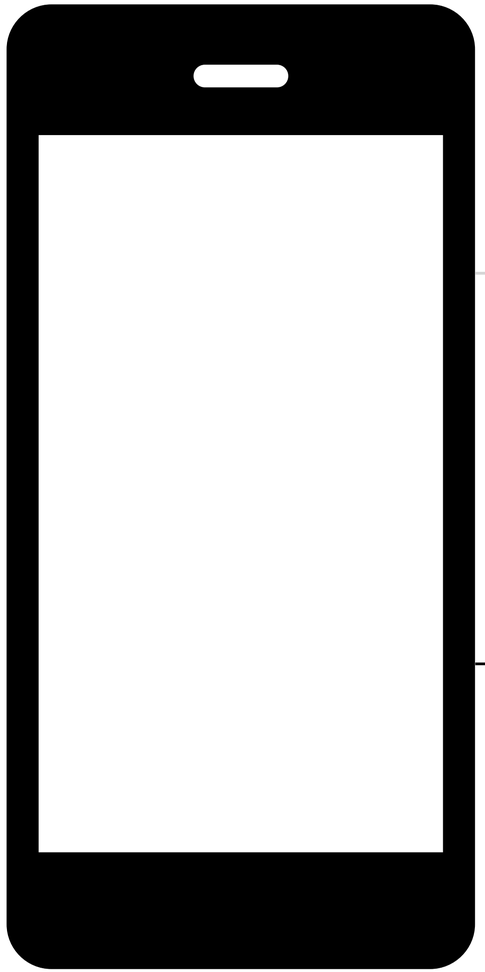
Ещё один вариант



force reload = true

force reload = true

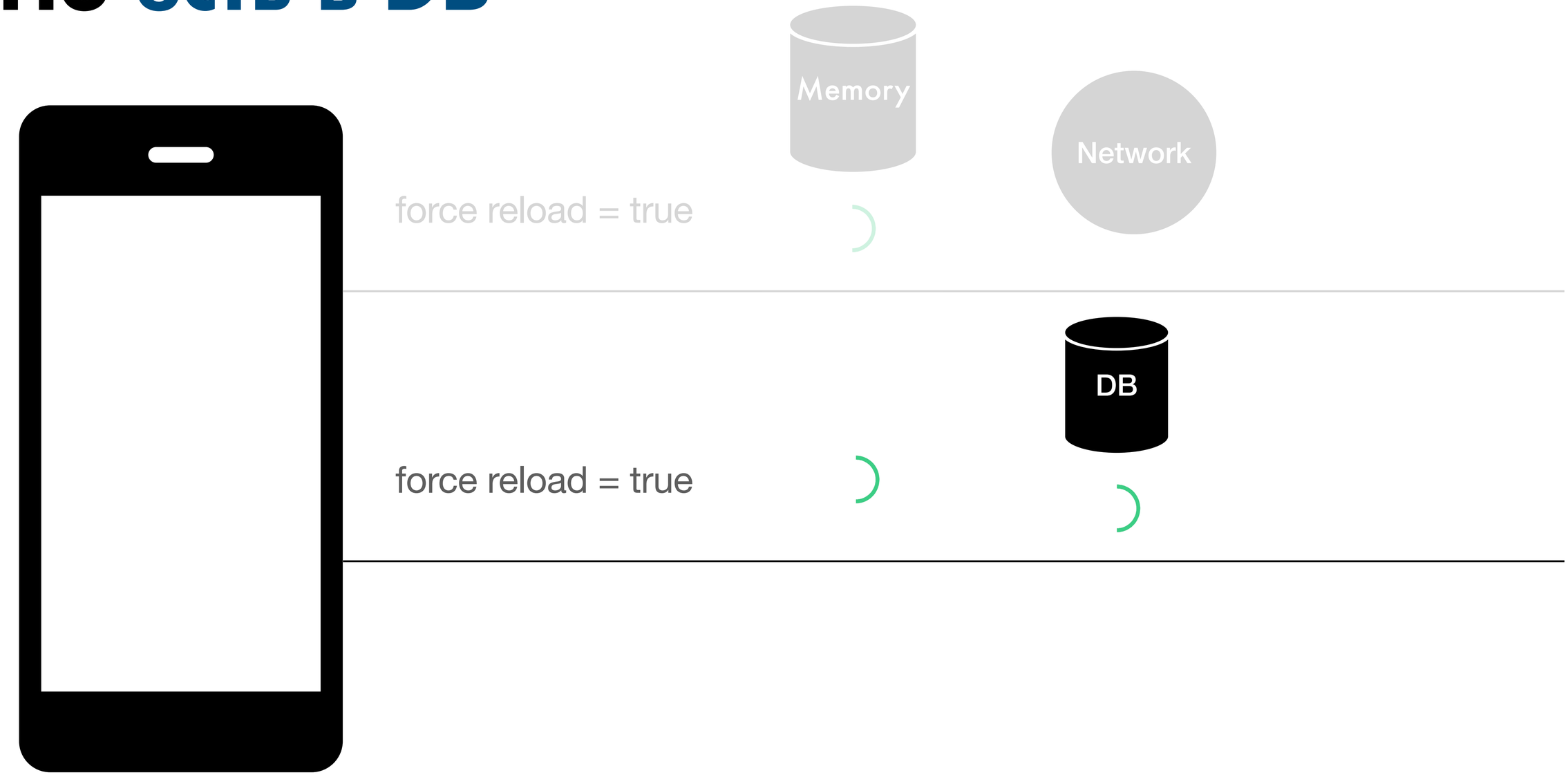
Данных в кеше нет



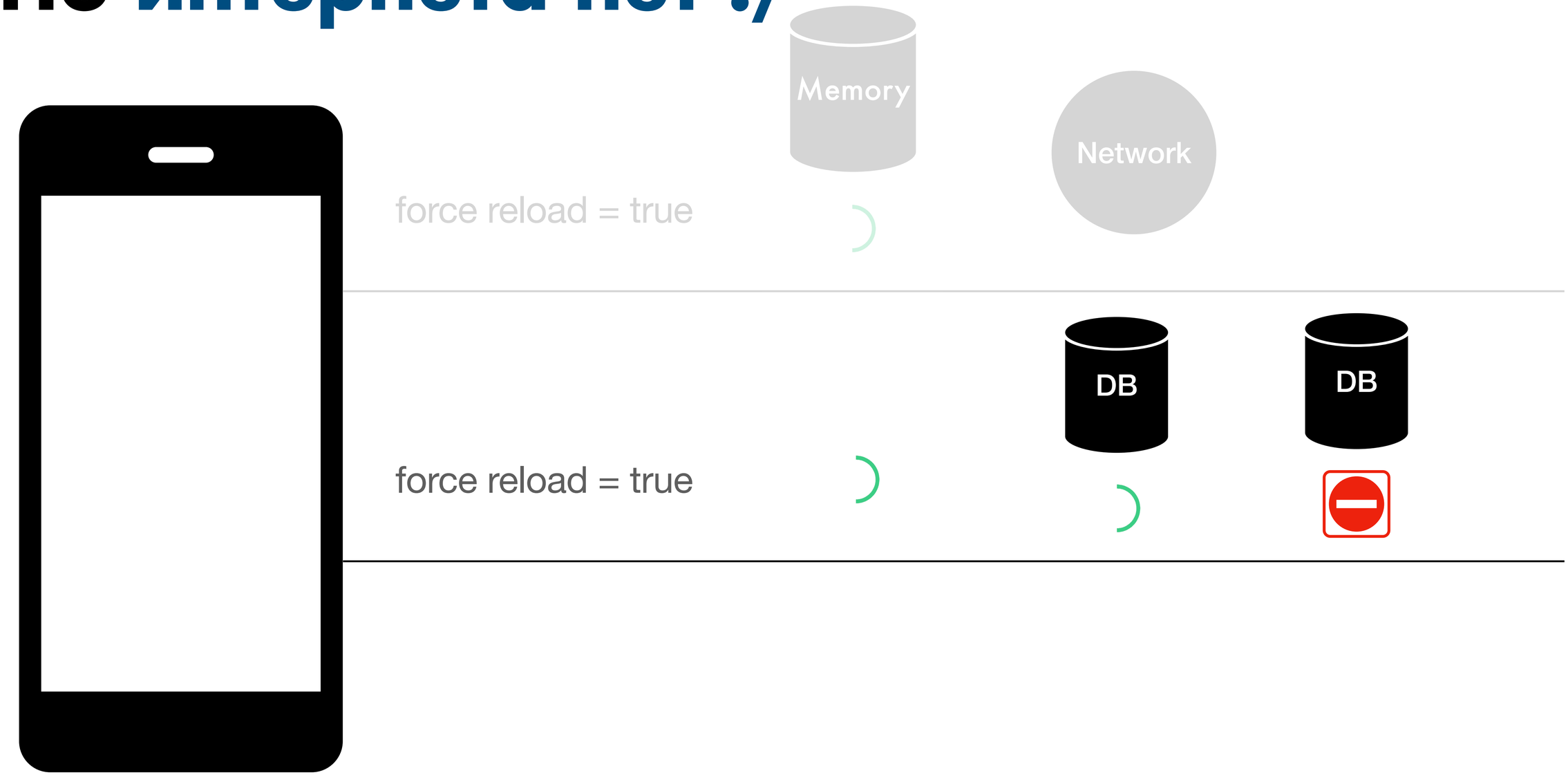
force reload = true

force reload = true

Но есть в DB



Но интернета нет :/



3. Вспоминаем работу RX

Цепочка

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

На каком потоке отработает doOnSubscribe?

1. На потоке подписки
2. На IO
3. На computation
4. Иногда на IO, иногда на computation

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

На каком потоке отработает doOnSubscribe?

На IO

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Удалили doOnSubscribe

```
Observable.just(“Hey")
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Какой subscribeOn будет выполнен?

1. Вот этот

2. Вот этот

3. Все 4 выполнятся

```
Observable.just(“Hey”)
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Какой subscribeOn будет выполнен?

Все выполнятся

```
Observable.just(“Hey”)
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
    .subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe {

}
```

Вернёмся к нашим баранам

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Тут 3 этапа

Тут 3 этапа

1. Создание источников
2. Подписка источников
3. Emitting данных

Тут 3 этапа

1. Создание источников

2. Подписка источников

3. Emitting данных

Цепочка

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Создание: Just

```
Observable.just(“Hey”)
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

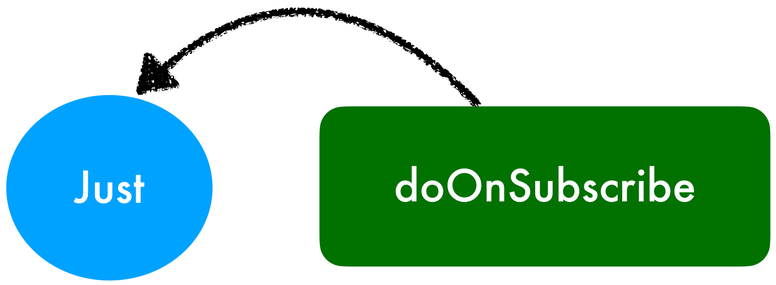
Создание: doOnSubscribe

```
Observable.just("Hey")
    .doOnSubscribe {   }

 .subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe{

}
```

Хранит ссылку на стрим выше

Store up stream

```
Observable.just(“Hey")
    .doOnSubscribe {   }

 .subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe{

}
```



Создание: subscribeOn

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Store up stream

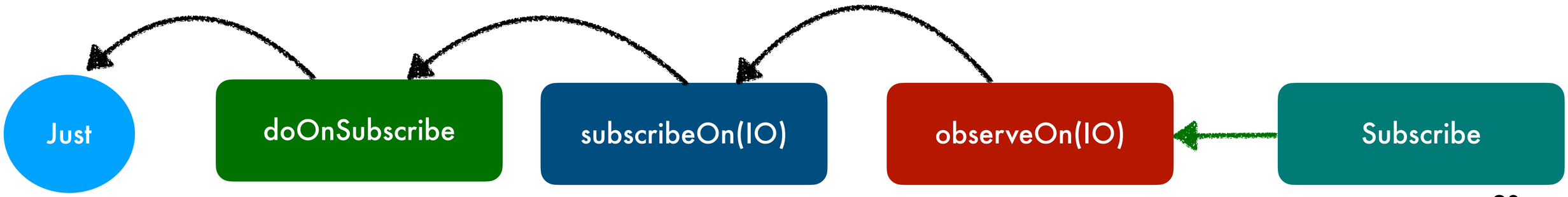
Создание: observeOn

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Store up stream

Начало подписки

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```



Тут 3 этапа

1. Создание источников

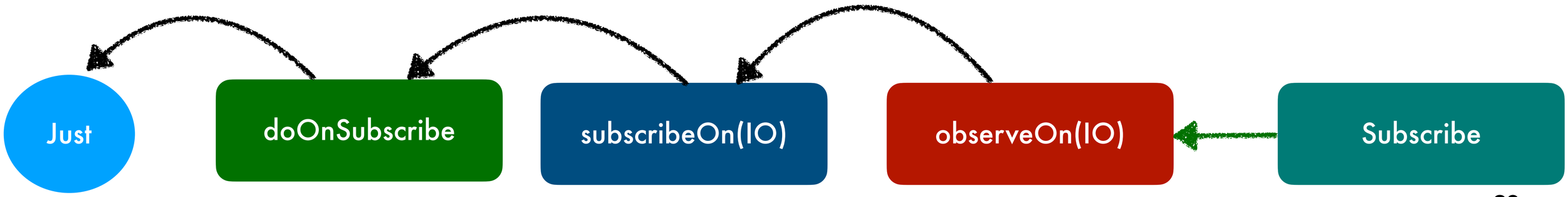
2. **Подписка** источников

3. Emitting данных

Подписались на observeOn

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

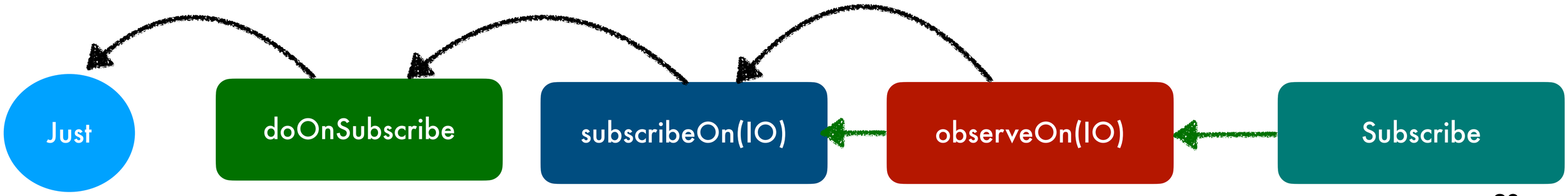
Call subscribe



Подписались на subscribeOn

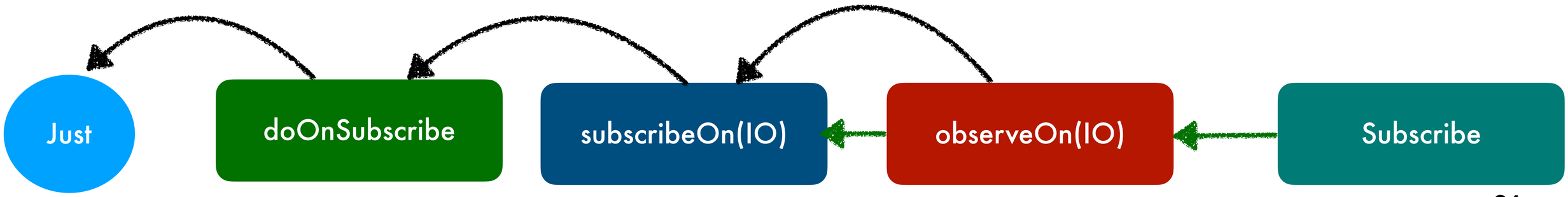
```
Observable.just("Hey")
    .doOnSubscribe {  }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Call subscribe



Сменили треду на IO

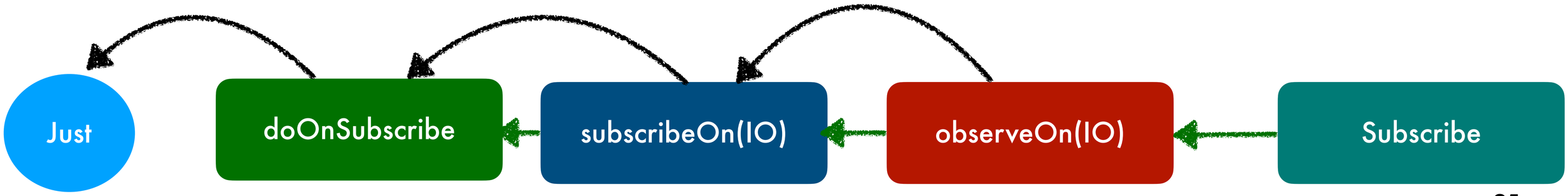
```
Observable.just("Hey")
        .doOnSubscribe {   }
Change thread to IO  .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .subscribe {

        }
```



Подписались на doOnSubscribe

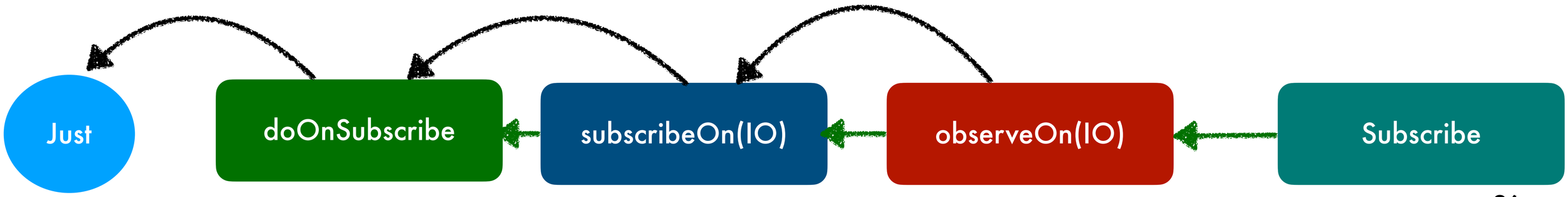
```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Call subscribe



Выполнился doOnSubscribe

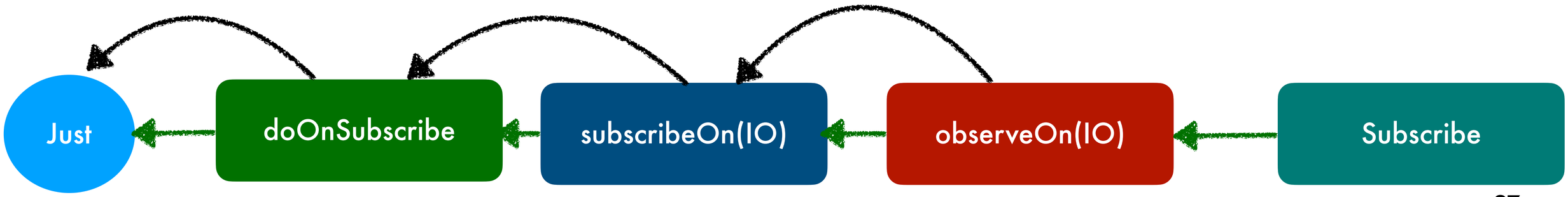
```
Observable.just("Hey")
        .doOnSubscribe  {
              Will be executed on IO
        }
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .subscribe
```



Подписались на Just

Call subscribe ↑

```
Observable.just("Hey")
        .doOnSubscribe {   }
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .subscribe {

        }
```



Тут 3 этапа

1. Создание источников

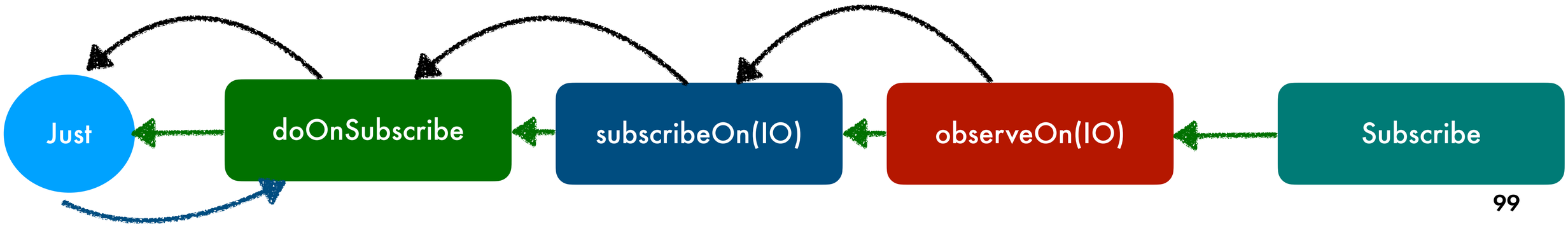
2. Подписка источников

3. Emitting данных

Just emits

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {


    }
```

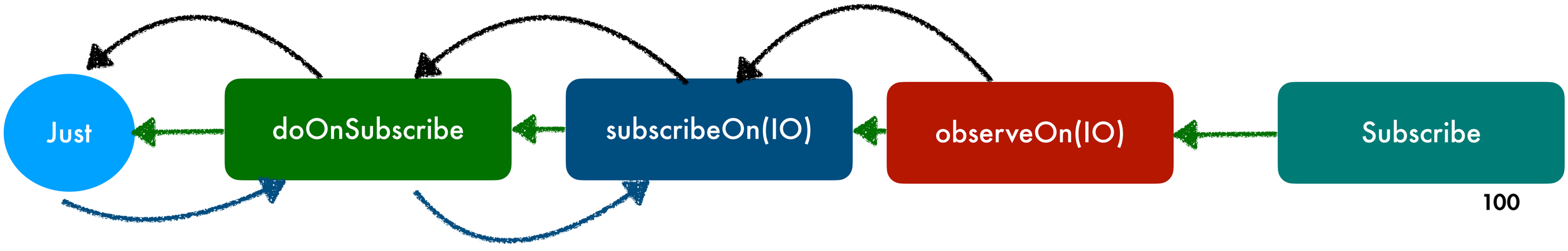
Emit "Hey"



doOnSubscribe emits

```
Observable.just("Hey")
        .doOnSubscribe {   }
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .subscribe {

        }
```

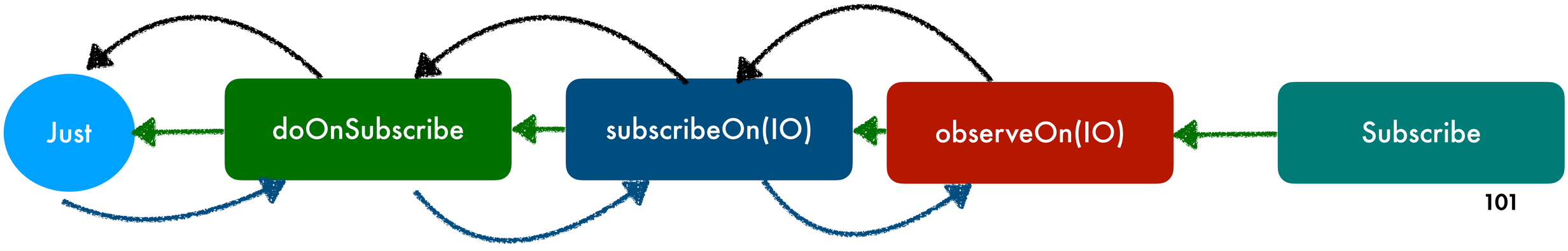
Emit "Hey"



subscribeOn emits

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

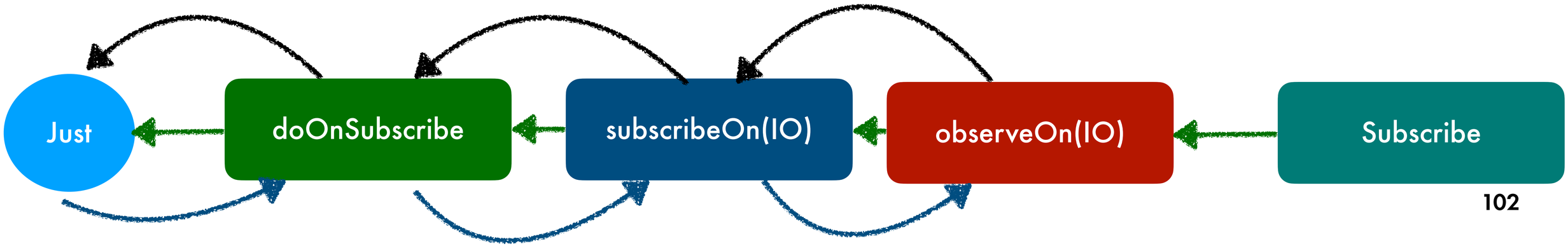
Emit "Hey"



subscribeOn emits

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

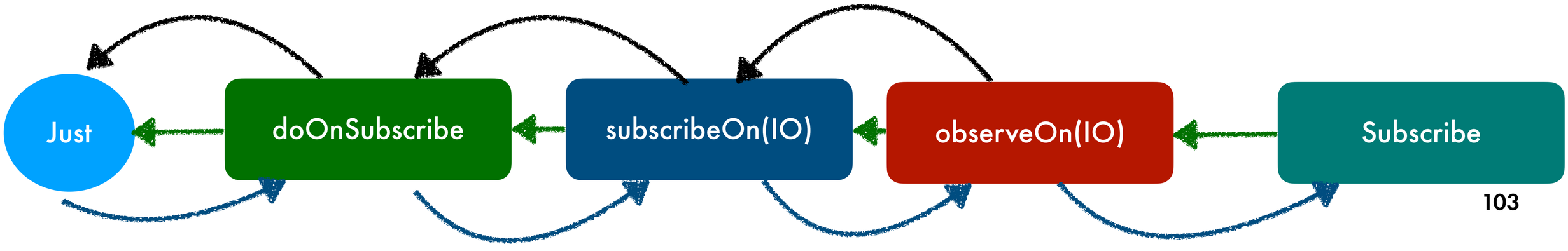
Change thread to computation



subscribeOn emits

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

Emit "Hey"



4. Реализуем

Что нам нужно?

1. RuntimeCache: ConcurrentHashMap

Что нам нужно?

1. RuntimeCache: ConcurrentHashMap

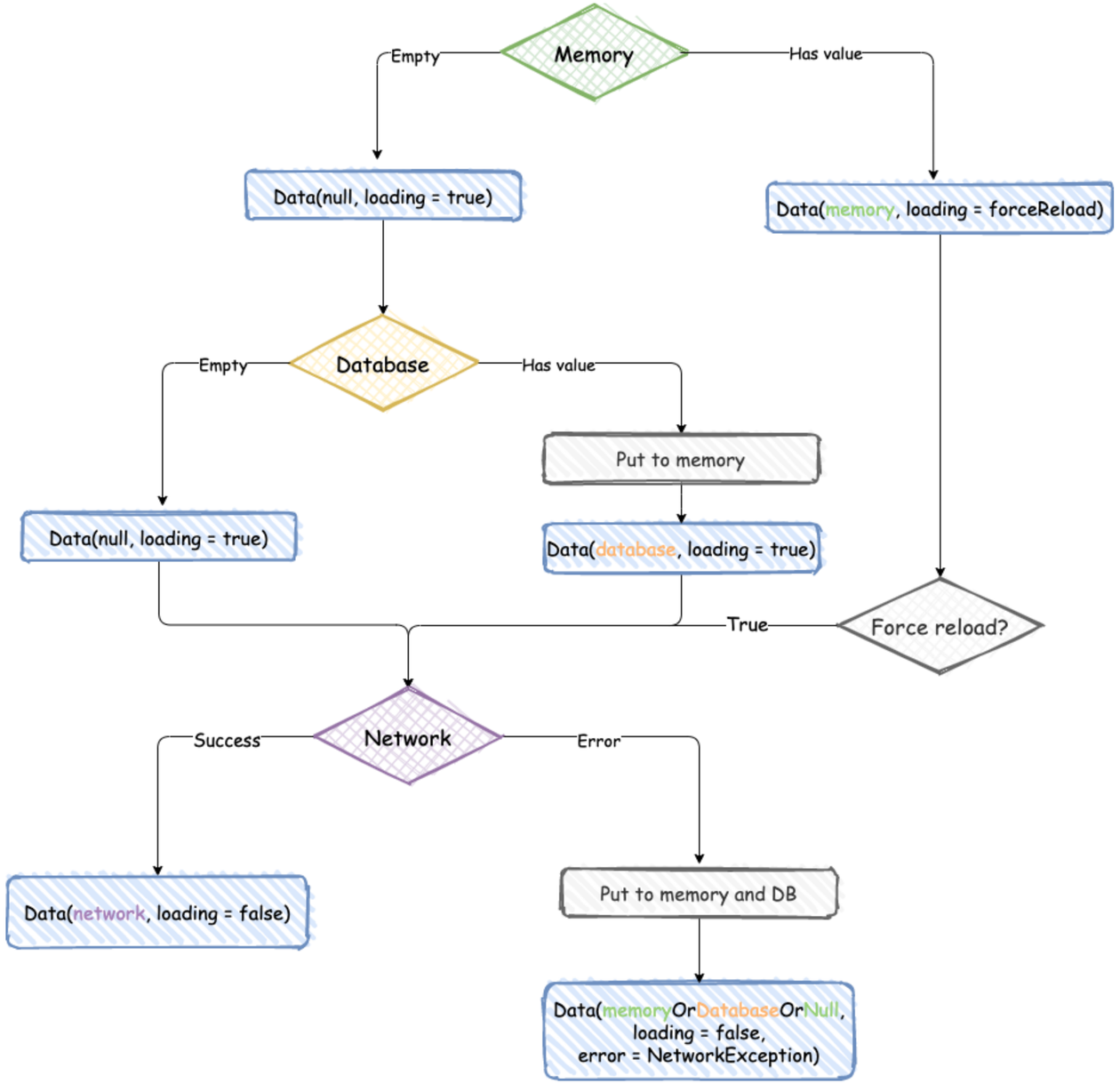
2. Subject: PublishSubject

Не забываем про потокобезопасность

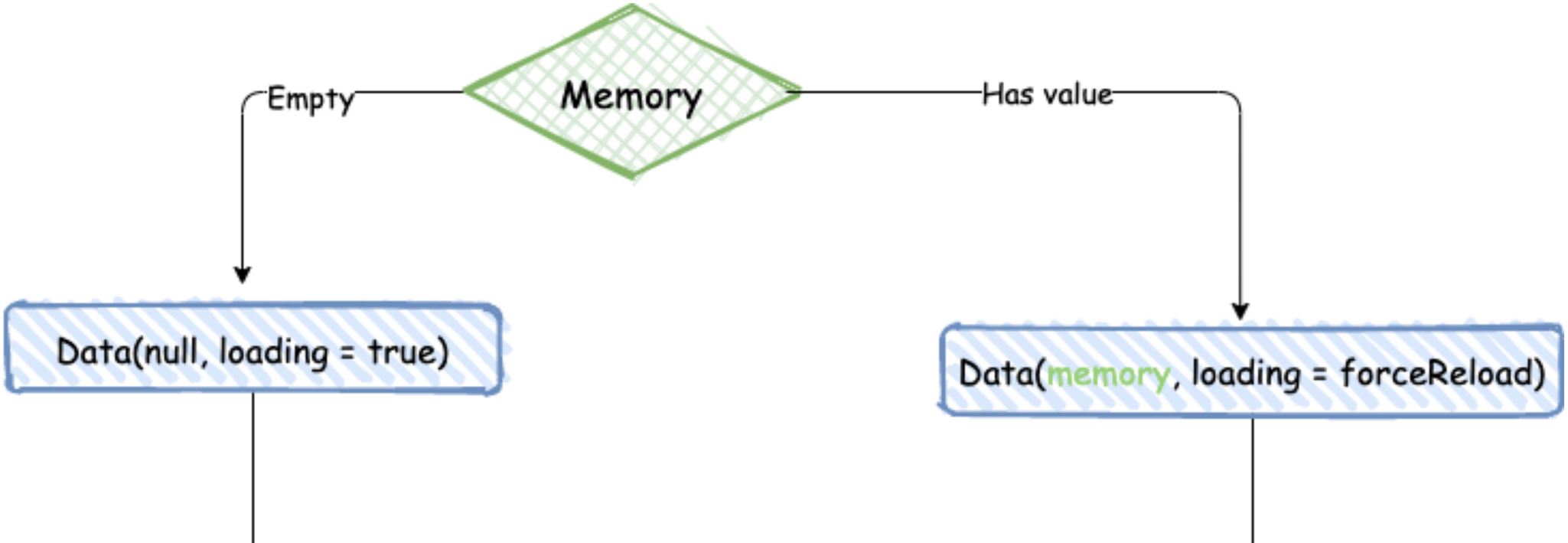
1. RuntimeCache: ConcurrentHashMap

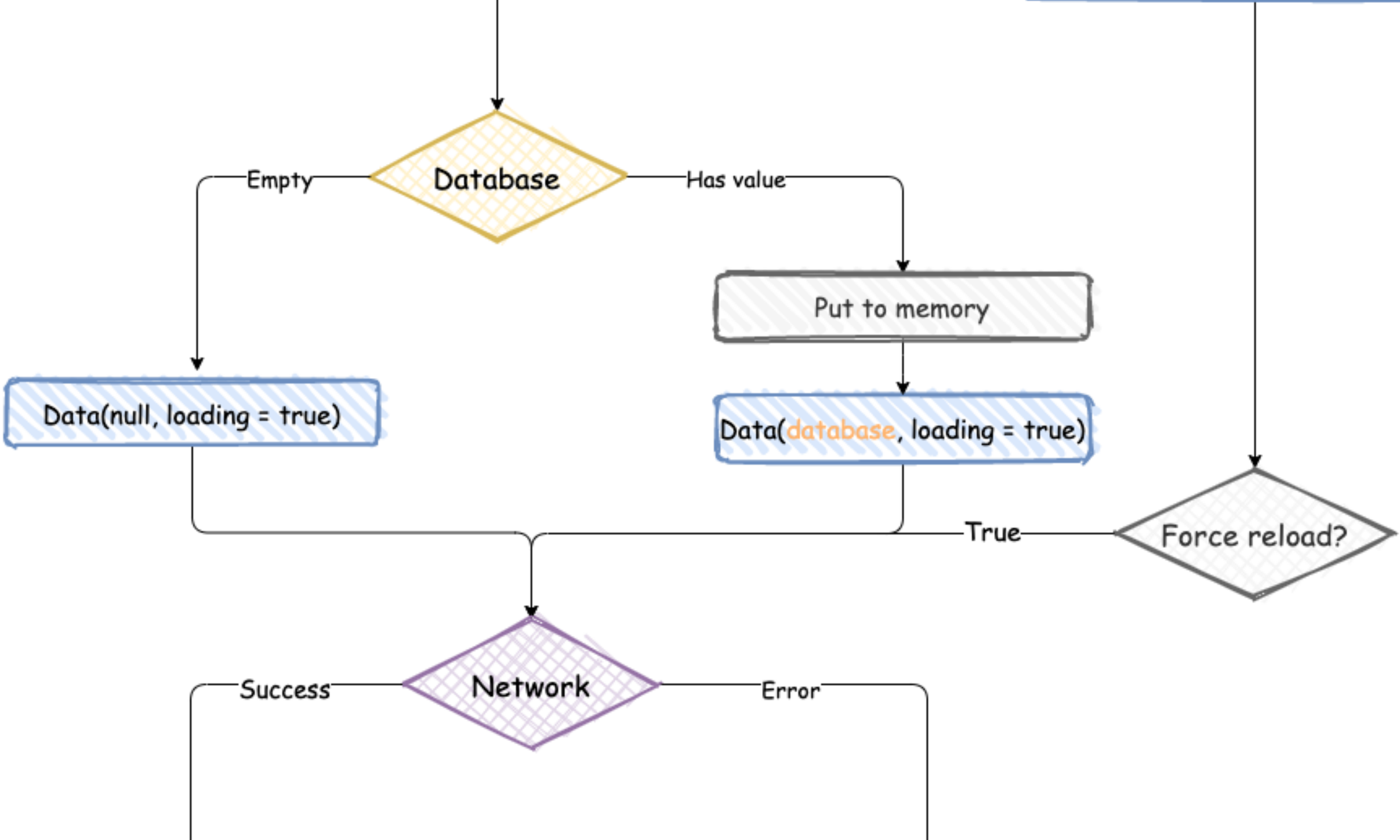
2. Subject: PublishSubject().toSeriallized()

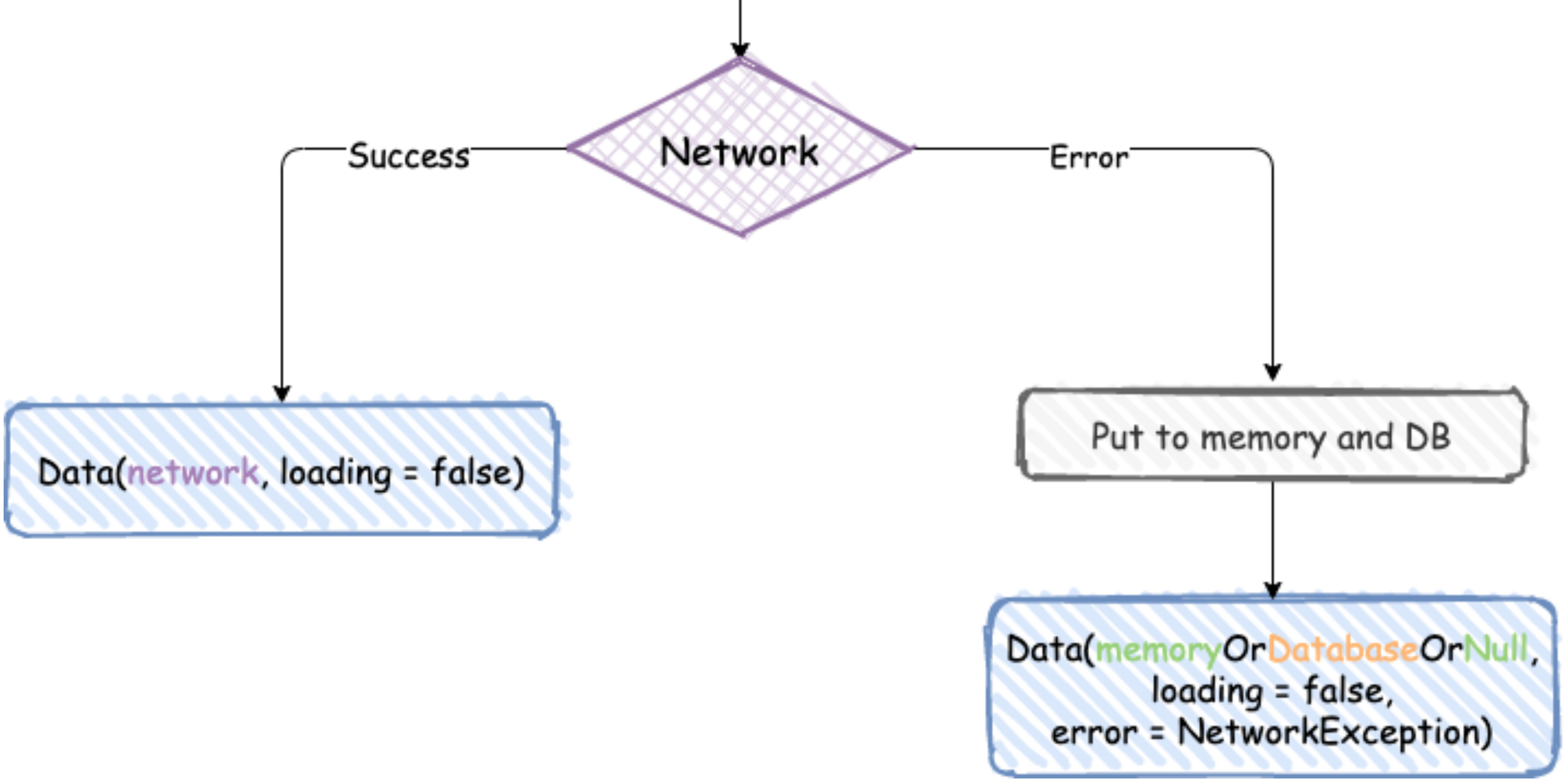
Схема работы



Проверяем память, эмитим

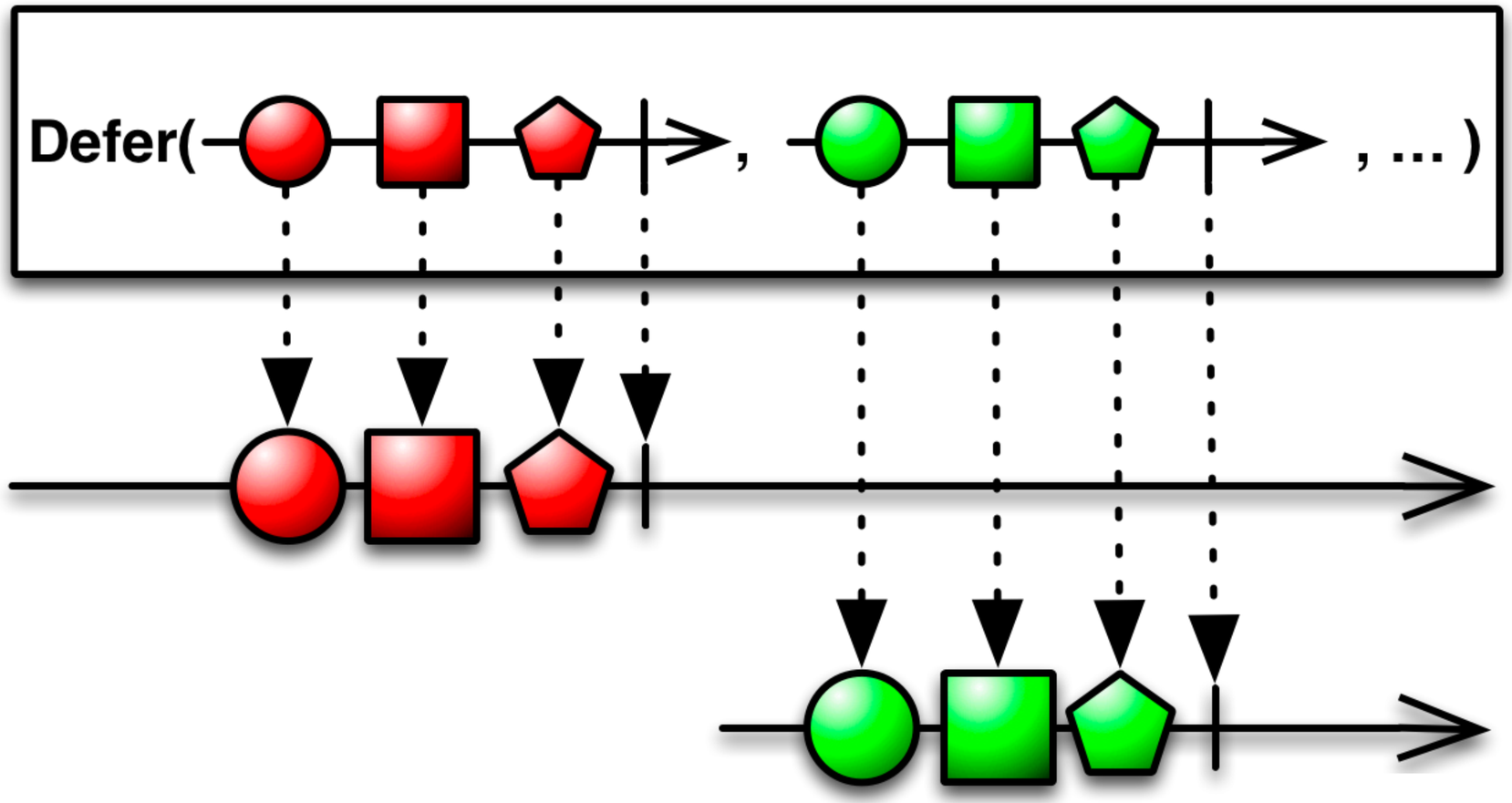



Database
Empty
Has value
Put to memory
Data(null, loading = true)
Data(database, loading = true)
Force reload?
True
Network
Success
Error


Network
Success
Error
Data(network, loading = false)
Put to memory and DB
Data(memoryOrDatabaseOrNull,
loading = false,
error = NetworkException)

1. Важно, чтобы данные были актуальными

Observable.defer



http://reactivex.io/documentation/operators/defer.html

Repository

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>>  =
    Observable.defer {
        //Тут будем создавать Observables
    }
```

Repository: loading

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        val fromMemory = memoryCache[“key”]
        val loading = fromMemory == null || forceReload
      //...
    }
```

Repository: loading

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        val fromMemory = memoryCache[“key”]
        val loading = fromMemory == null || forceReload

        fetchFromStorage(fromMemory)
        //...
    }
```

Repository: from storage

```
private fun fetchFromStorage(fromMemory: User?): Single<Data<User>> =
```

Repository: from storage

```
private fun fetchFromStorage(fromMemory: User?): Single<Data<User>> =

        if (fromMemory != null) {
            Single.just(Data(content = memCache))
        }
```

Repository: Иначе в DB

```
private fun fetchFromStorage(fromMemory: User?): Single<Data<User>> =
    if (fromMemory != null) {
        Single.just(Data(content = fromMemory))
    } else {
        dao.getUser()
    }
```

Repository: Сохраняем в кеш

```
private fun fetchFromStorage(fromMemory: User?): Single<Data<User>> =
    if (fromMemory != null) {
        Single.just(Data(content = fromMemory))
    } else {
        dao.getUser()
            .doOnSuccess { cachedValue ->
                memoryCache["key"] = cachedValue
            }
    }
```

Repository: subscribeOn(IO)

```
private fun fetchFromStorage(fromMemory: User?): Single<Data<User>> =
    if (fromMemory != null) {
        Single.just(Data(content = fromMemory))
    } else {
        dao.getUser()
            .doOnSuccess { cachedValue ->
                memoryCache["key"] = cachedValue
            }
            .subscribeOn(Schedulers.io())
    }
```

Repository: loading

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        val fromMemory = memoryCache["key"]
        val loading = fromMemory == null || forceReload

        fetchFromStorage(fromMemory)
        //...
    }
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        val fromMemory = memoryCache["key"]
        val loading = fromMemory == null || forceReload

        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                //Го в сеть если нужно
            }
    }
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
                .flatMapObservable { dataFromStorage ->
                    if (loading) {


                    }
                }
        //...
    }
```

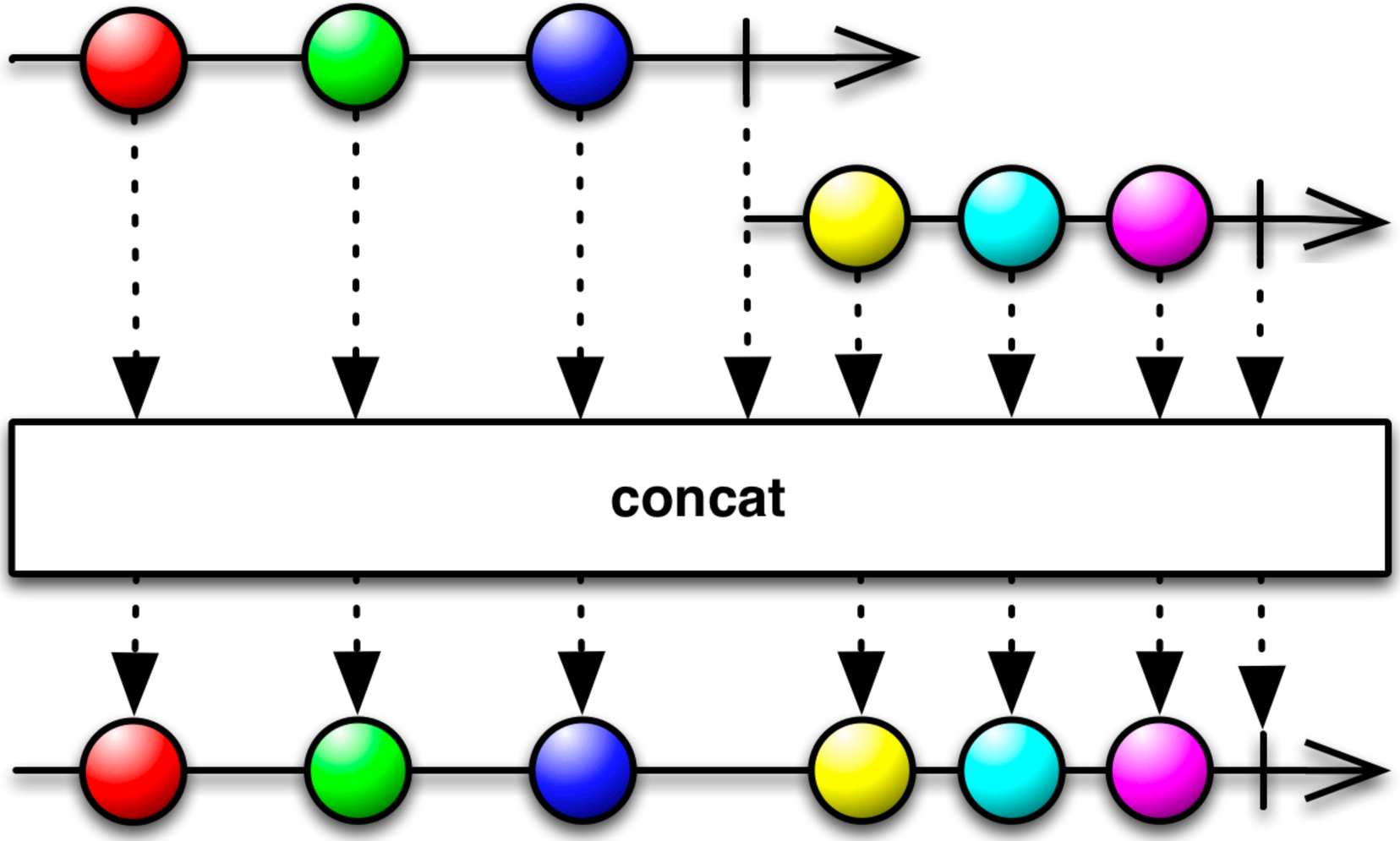
Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    Последовательно: база затем сеть
                }
            }
        //...
    }
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                }
            }
        //...
}
```

Concat



http://reactivex.io/documentation/operators/concat.html

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                }
            }
        //...
}
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                    concat(

                    )
                }
            }
        //...
```

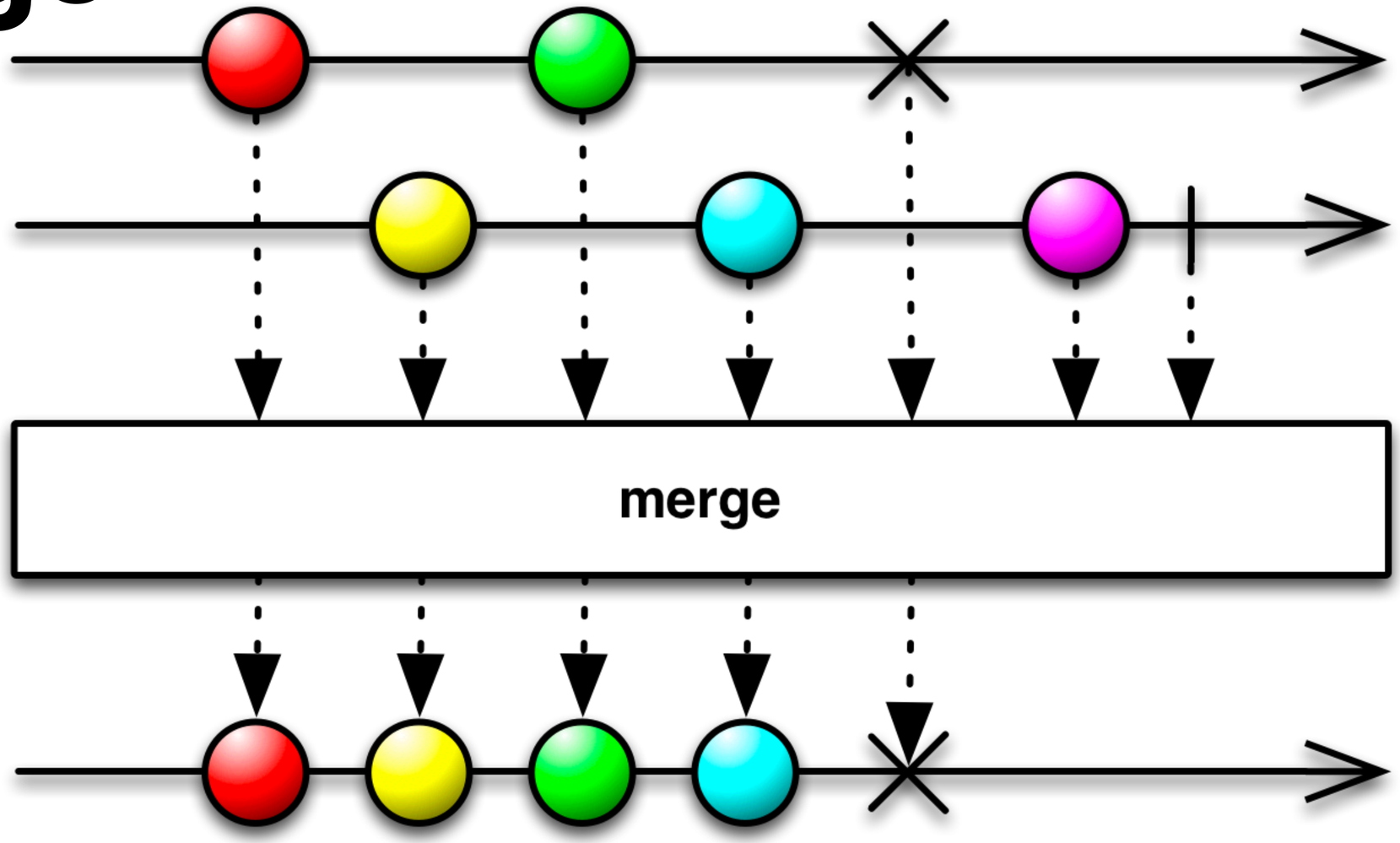
Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                    concat(
                        just(data),
                        fetchFromNetwork(storageData.content, params)
                    )
                }
            }
        //...
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                    subject.onNext(data)
                    concat(
                        just(data),
                        fetchFromNetwork(storageData.content, params)
                    )
                }
    }
```

Merge



http://reactivex.io/documentation/operators/merge.html

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                    subject.onNext(data)
                    concat(
                        just(data),
                        fetchFromNetwork(storageData.content, params)
                    )
                }
    }
```

Repository: fromNetwork

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    val data = storageData.copy(loading = true)
                    subject.onNext(data)
                    concat(
                        just(data),
                        fetchFromNetwork(storageData.content, params)
                                .mergeWith(subject)
                    )
                }
```

Repository: just return data

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { dataFromStorage ->
                if (loading) {
                    //...
                } else {
                    concat(
                        just(data),
                        subject
                    )
                }
```

Repository: No data in DB

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { _: Throwable ->

            }
        //...
}
```

Repository: No data in DB

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { _: Throwable ->
              val data = Data(content = null, loading = true)

            }
        //...
}
```

Repository: No data in DB

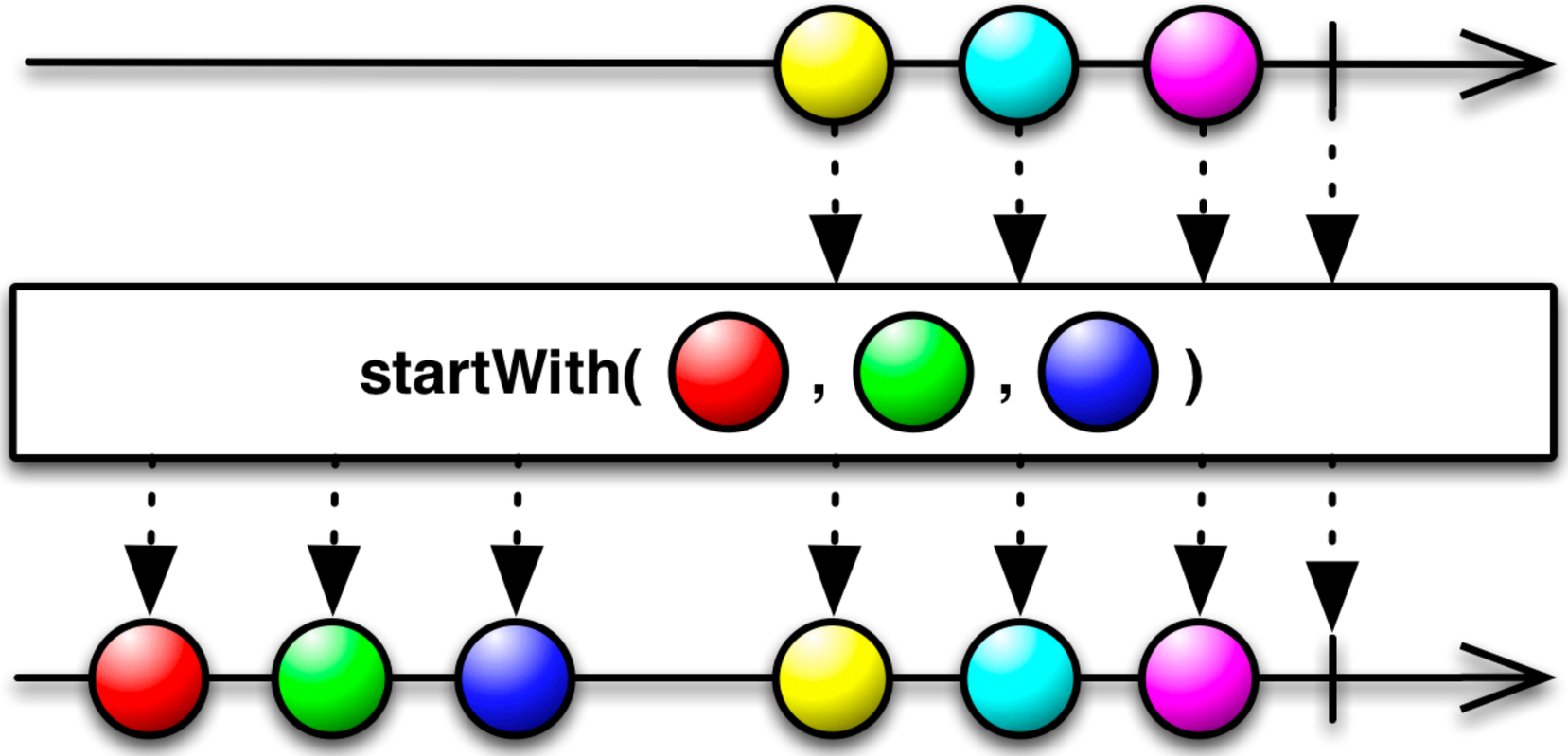
```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { _: Throwable ->
                val data = Data(content = null, loading = true)
                concat(
                        just(data),
                        fetchFromNetwork()
                )
            }
    //...
```

Repository: Reactive

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { _: Throwable ->
                val data = Data(content = null, loading = true)
                concat(
                        just(data),
                        fetchFromNetwork()
                            .mergeWith(subject)
                )
            }
        //
```

2. Синхронный ответ

StartWith



http://reactivex.io/documentation/operators/startwith.html

Repository: Reactive

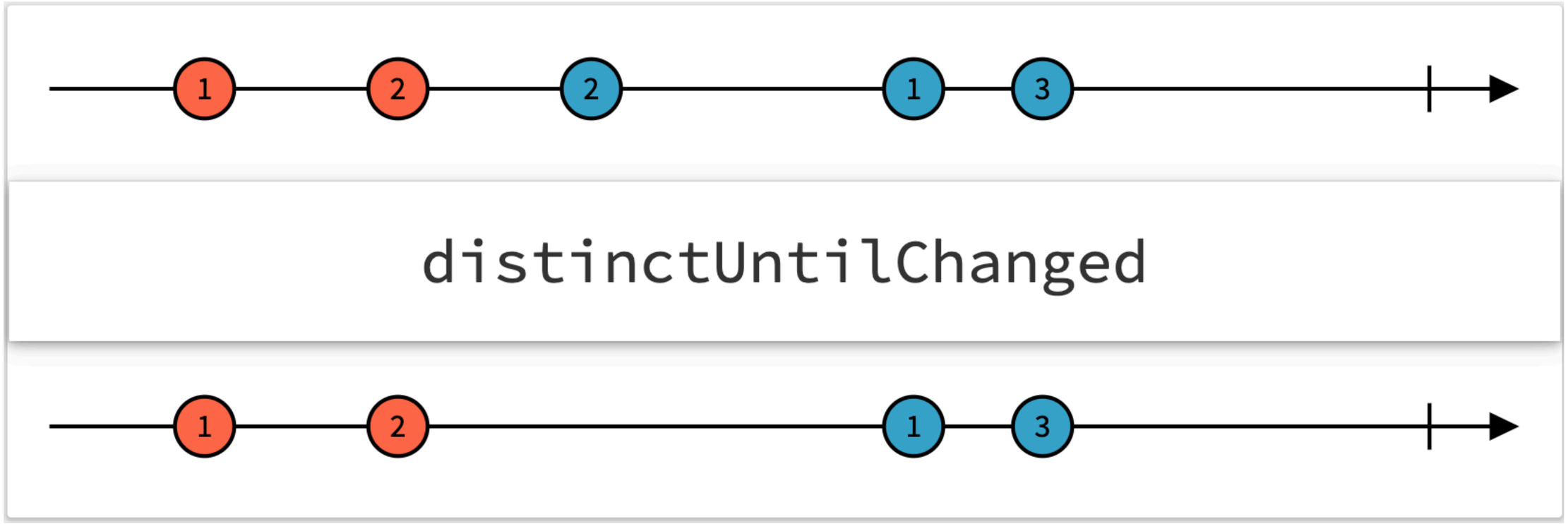
```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { //...}
        //...
}
```

Repository: Reactive

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { //...}
            .startWith(Data(content = fromMemory, loading = loading)
        //...
}
```

3. Не эмитим одни и те же данные

Distinct until changed



http://reactivex.io/documentation/operators/merge.html

Repository: Reactive

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { //...}
            .startWith(Data(content = fromMemory, loading = loading)
}
```

Repository: Reactive

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    Observable.defer {
        //...
        fetchFromStorage(fromMemory)
            .flatMapObservable { //... }
            .onErrorResumeNext { //...}
            .startWith(Data(content = fromMemory, loading = loading)
            .distinctUntilChanged()
}
```

4. Переиспользуем

Repository

```
Observable.defer {
    //феч из стоража + определение загрузки
    fetchFromStorage(memCache, params)
        .flatMapObservable { storageData ->
            //го в сеть или вернуть данные
        }
        }
        .onErrorResumeNext { _: Throwable ->
            //данных нет в базе, го в сеть
       }
        .startWith(Data(memCache, loading = loading))
        .distinctUntilChanged()
}
```

API: Input/Output

```
class DataObservableDelegate<Params : Any, Domain : Any>()
```

API: Input

Like userId

```
class DataObservableDelegate<Params : Any, Domain : Any>()
```

API: Output

Observe

```
class DataObservableDelegate<Params : Any, Domain : Any>()
```

Constructor: Предоставляем каллбэки

API: Из Сети

```
class DataObservableDelegate constructor(
    private val fromNetwork: (params: Params) -> Single<Domain>
)
```

API: Из Кэша

```
class DataObservableDelegate constructor(
    private val fromNetwork: (params: Params) -> Single<Domain>
    private val fromMemory: (params: Params) -> Domain? = { _ -> null },

)
```

API: Из базы

```
class DataObservableDelegate constructor(
    private val fromNetwork: (params: Params) -> Single<Domain>
    private val fromMemory: (params: Params) -> Domain? = { _ -> null },
    private val fromStorage: ((params: Params) -> Single<Data<Domain>>) =
        { params -> Single.error(…)  }

)
```

API: В кэш/базу

```
class DataObservableDelegate constructor(
    private val fromNetwork: (params: Params) -> Single<Domain>
    private val fromMemory: (params: Params) -> Domain? = { _ -> null },
    private val fromStorage: ((params: Params) -> Single<Data<Domain>>) =
        { params -> Single.error(…) } }
    private val toMemory: (params: Params, Domain) -> Unit = { _, _ -> Unit },
    private val toStorage: ((params: Params, Domain) -> Unit) = { _, _ -> Unit }
)
```

Код из Repository

```
class DataObservableDelegate constructor(
    fromNetwork: DataObservableDelegate<Params, Domain>.(params: Params) -> Single<Domain>,
    private val fromMemory: (params: Params) -> Domain? = { _ -> null },
    private val fromStorage: ((params: Params) -> Single<Data<Domain>>) =
        { params -> Single.error(…) } }
    private val toMemory: (params: Params, Domain) -> Unit = { _, _ -> Unit },
    private val toStorage: ((params: Params, Domain) -> Unit) = { _, _ -> Unit }
) {

  private val subject = PublishSubject.create<Data<Domain>>().toSerialized()


   fun observe(params: Params, forceReload: Boolean = true): Observable<Data<Domain>> =
      ….Stuff we implemented before



}
```

Result

Repository здорового человека

```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {

    private val userDelegate = DataObservableDelegate<Unit, User>(
        fromNetwork = {  },
        fromMemory = {  },
        toMemory = { params, user -> ....  },
        toStorage = { params, user -> .... },
        fromStorage = {params-> ....}
    )
```

Repository: observe User

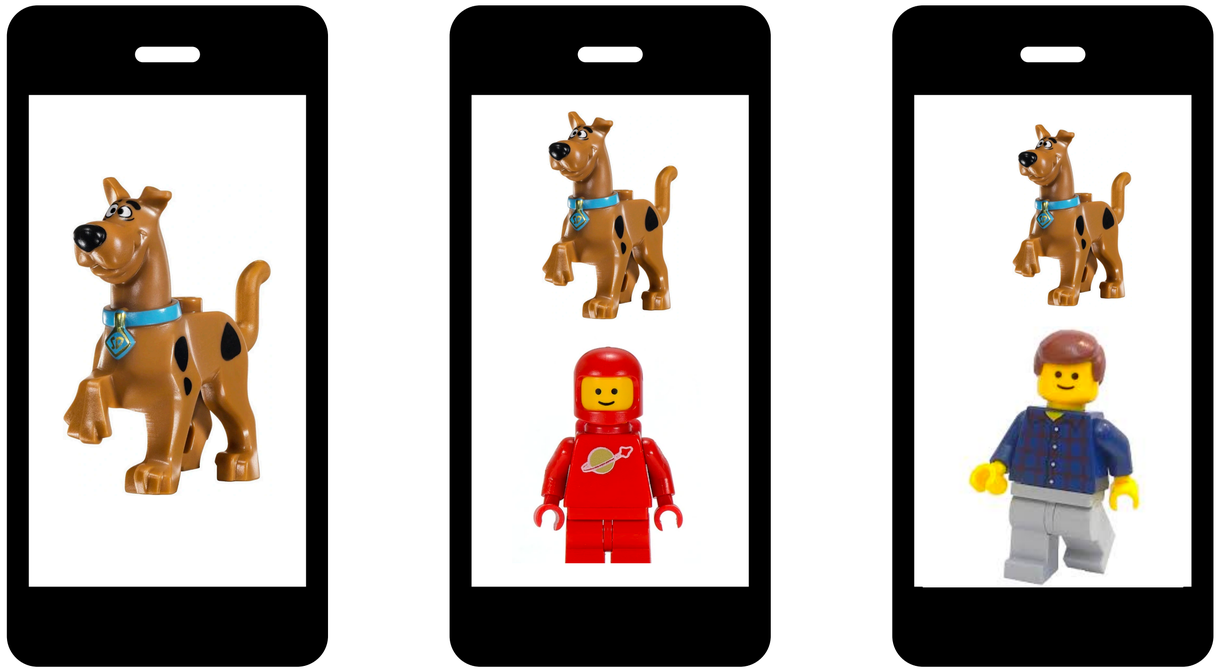
```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {

    private val userDelegate = DataObservableDelegate<Unit, User>(
        fromNetwork = {  },
        fromMemory = {  },
        toMemory = { params, user -> ....  },
        toStorage = { params, user -> .... },
        fromStorage = {params-> ....}
    )
```

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    userDelegate.observe(Unit, forceReload)
```

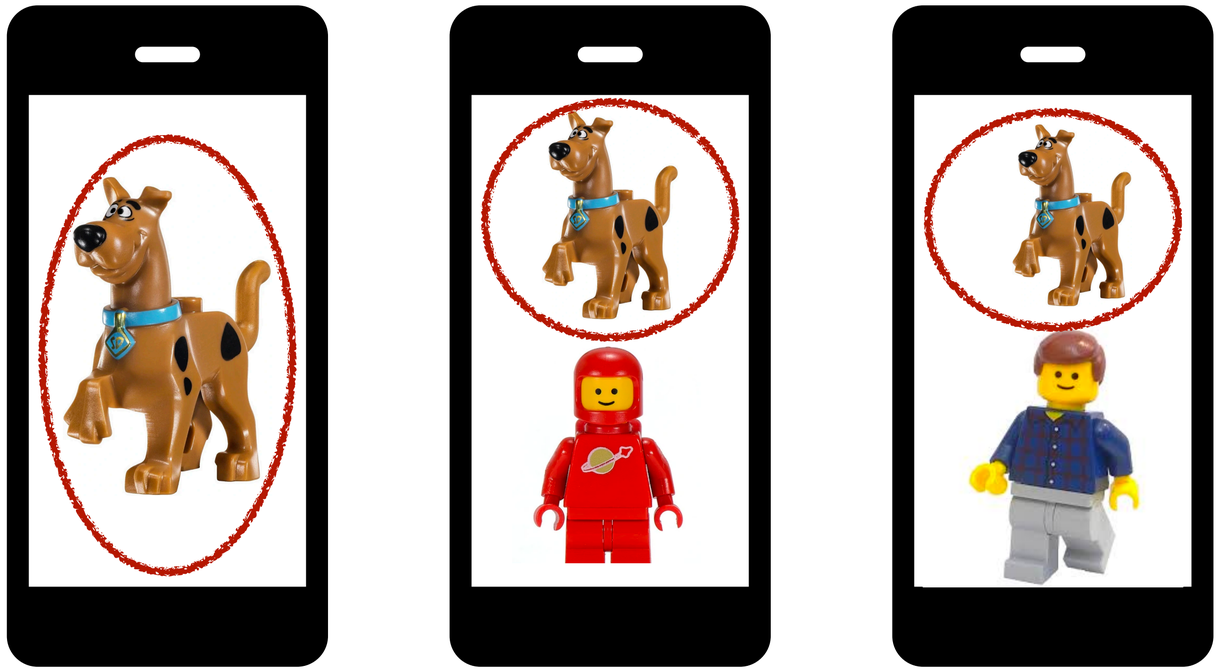
5. Shared Request

Проблема

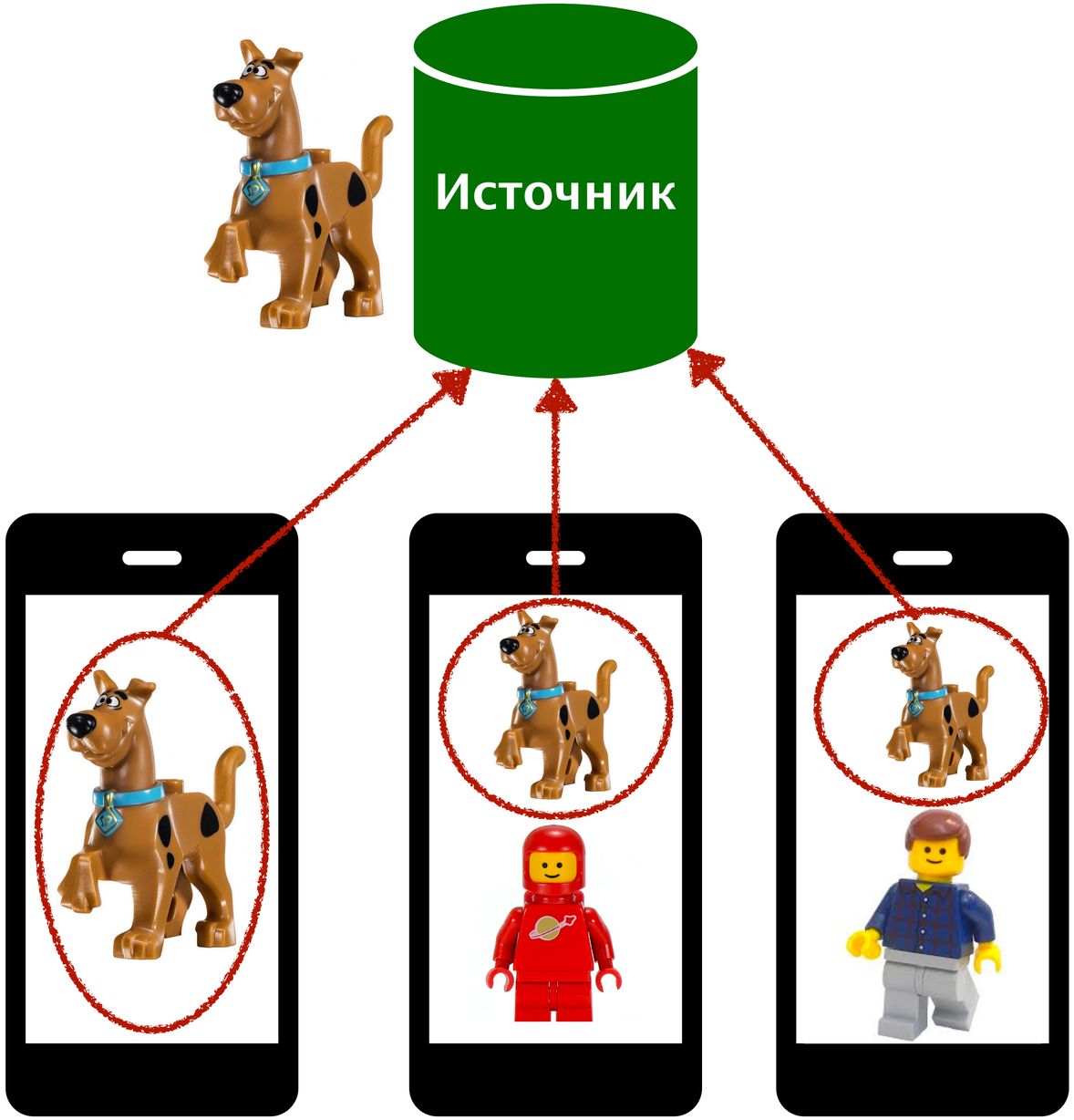
Одинаковый контент



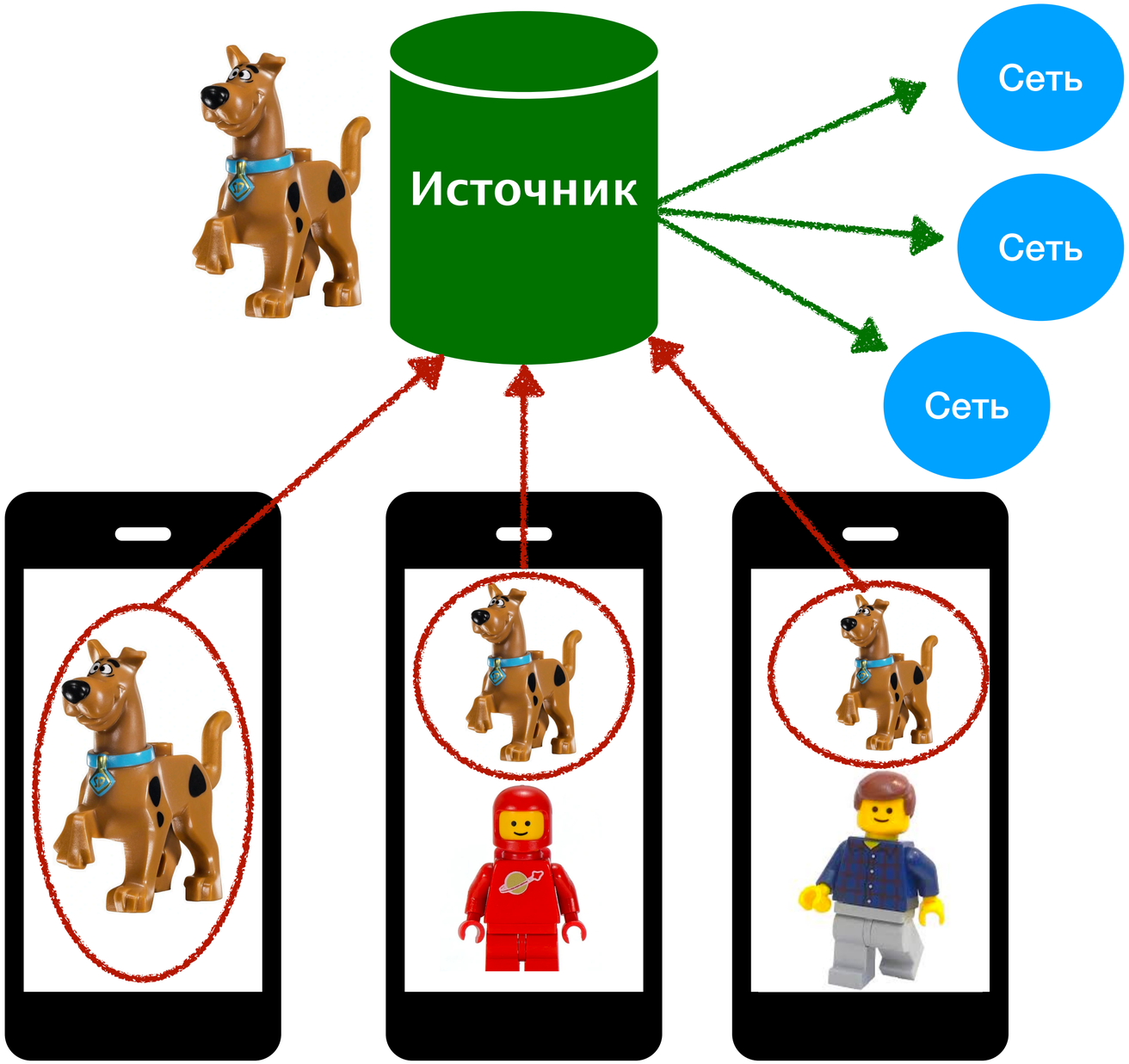
Одинаковый контент



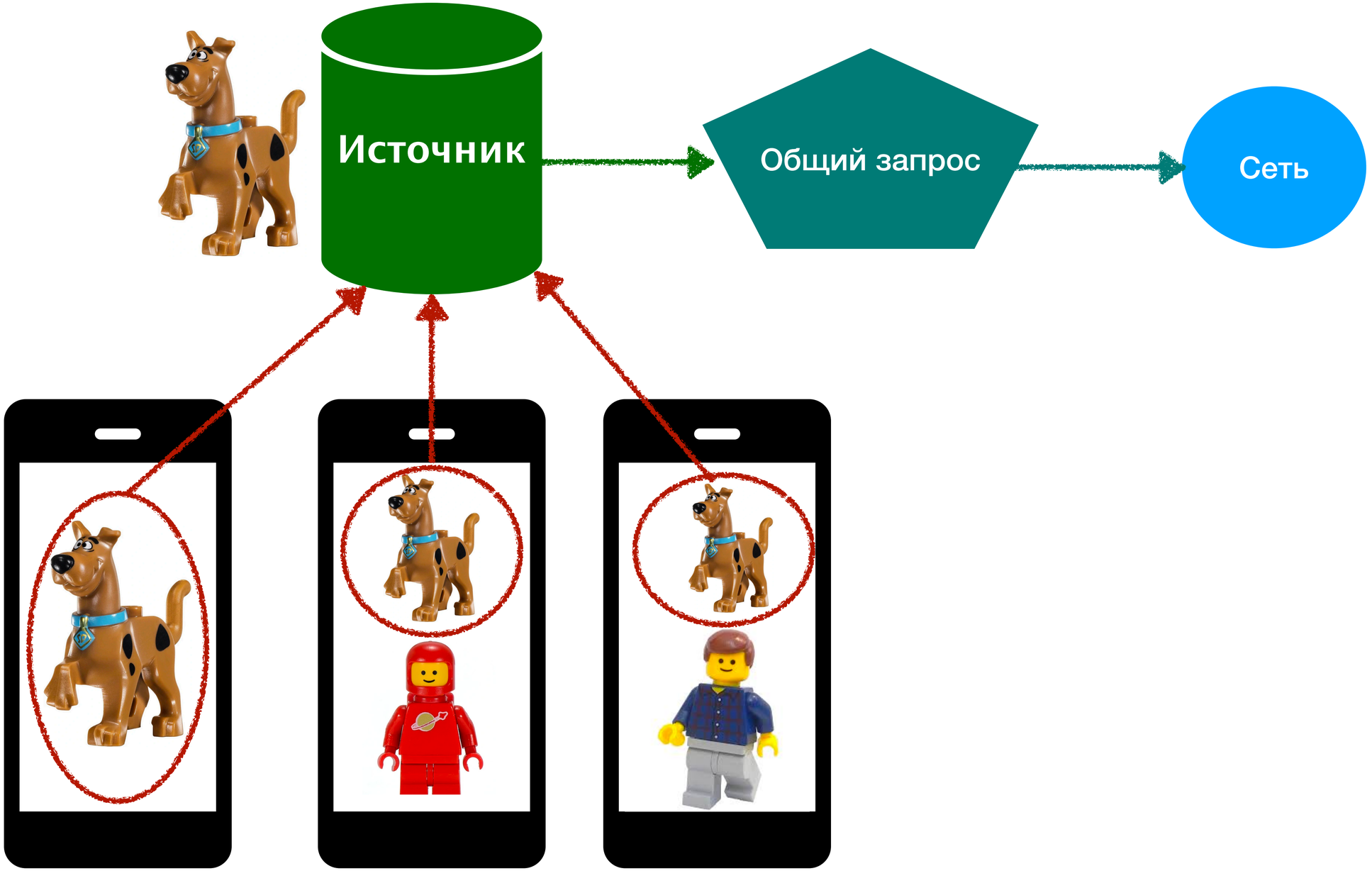
Общий источник



Force reload = true



Идея:



5.1 Connectable observable

Connectable observable

1. Общий стрим данных

2. Не начинают работу при subscribe()

3. Выполняются при connect()

Простой пример

```
val users: Observable<String> = getFromNetwork()
```

Делаем Connectable

```
val users: Observable<String> = getFromNetwork().publish()
```

Делаем Connectable

```
val users: ConnectableObservable<String> = getFromNetwork().publish()
```

Подписываемся

```
val users: ConnectableObservable<String> = getFromNetwork().publish()

users.subscribe()
users.subscribe()
users.subscribe()
```

Добавляем connect

```
val users: ConnectableObservable<String>  =  getFromNetwork().publish()

users.subscribe()
users.subscribe()
users.subscribe()

users.connect() // Без него не заработает источник
```

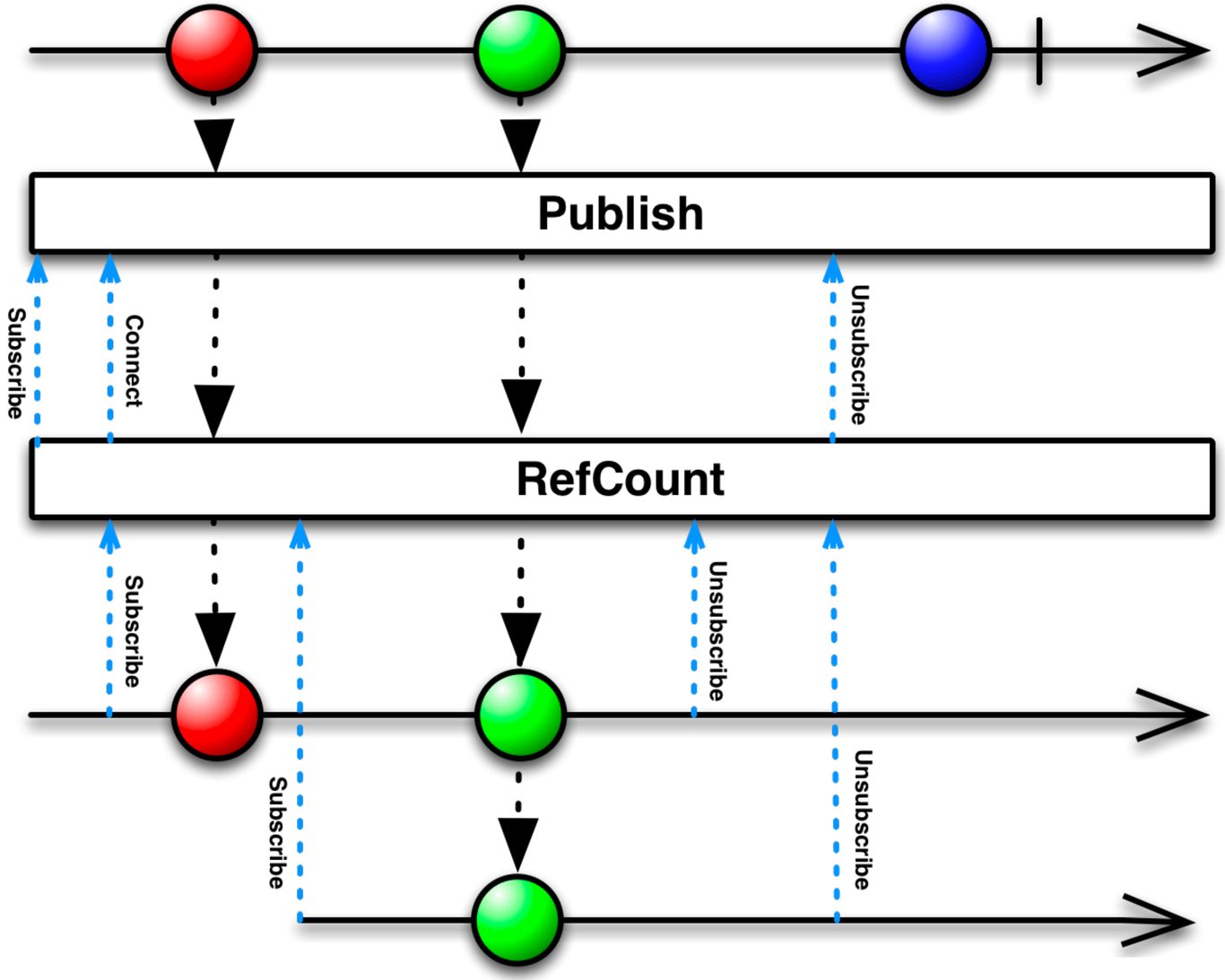
Добавляем connect

```
val users: ConnectableObservable<String>  =  getFromNetwork().publish()

users.subscribe()
users.subscribe()
users.subscribe()

users.connect() // Без него не заработает источник
```

Connectable observable

1. publish().refCount()

RefCount



http://reactivex.io/documentation/operators/refcount.html

Connectable observable

1. publish().refCount()

2. share() //Инкапсулирует первый вариант

Как было

```
val users: ConnectableObservable<String>  =   getFromNetwork().publish()

users.subscribe()
users.subscribe()
users.subscribe()

users.connect() // Без него не заработает источник
```

Добавляем share()

```
val users: Observable<String> =  getFromNetwork().share()

users.subscribe()
users.subscribe()
users.subscribe()

users.connect() // Без него не заработает источник
```

Удаляем connect

```
val users: Observable<String>  =   getFromNetwork().share()

users.subscribe()
users.subscribe()
users.subscribe()
```

Сделаем ещё одну абстракцию

SharedObservableRequest

```
class SharedObservableRequest<Params, Result>(
    private val load: (params: Params) -> Observable<Result>
)
```

Сохраняем запросы

```
class SharedObservableRequest<Params, Result>(
    private val load: (params: Params) -> Observable<Result>
) {

    private val requests = HashMap<Params, Observable<Result>>()

}
```

Return shared data

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {



}
```

Return shared data

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

        }
    }
```

Make request

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

            val newShared = load(params)
        }
    }
```

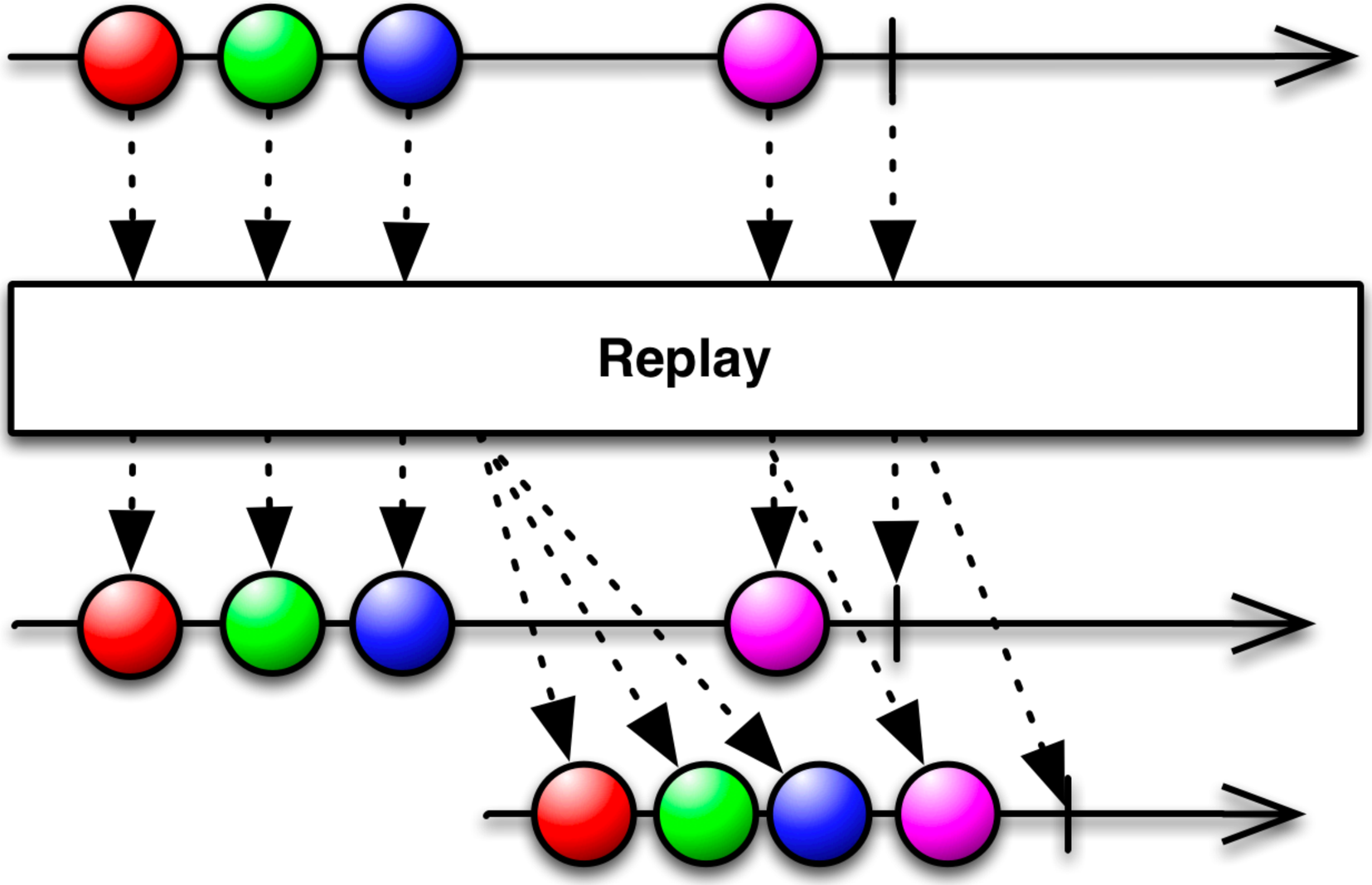
Make request

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

            val newShared = load(params)
                .doFinally {
                    synchronized(requests) { requests.remove(params) }
                }
        }
    }
```

Make request

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

            val newShared = load(params)
                .doFinally {
                    synchronized(requests) { requests.remove(params) }
                }

            requests[params] = newShared
            return@defer newShared
        }
    }
```

Replay



http://reactivex.io/documentation/operators/refcount.html

Make request

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

            val newShared = load(params)
                .doFinally {
                    synchronized(requests) { requests.remove(params) }
                }

            requests[params] = newShared
            return@defer newShared
        }
    }
```

Connectable

```
fun getOrLoad(params: Params): Observable<Result>  = Observable.defer {
        synchronized(requests) {
            requests[params]?.let { cachedShared ->
                return@defer cachedShared
            }

            val newShared = load(params)
                .doFinally {
                    synchronized(requests) { requests.remove(params) }
                }
                .replay(1)
                .refCount()

            requests[params] = newShared
            return@defer newShared
        }
```

После, встроим

SharedRequest в DataObservableDelegate

6. Custom scheduler

Вспомним: синхронный RX

```
Log(“Before”)

Observable.just(1)
    .subscribe {
        Log(“Body”)
    }

Log.i("After")
```

1

2

3

Реальный Presenter

```
view.setName(“Bojack”)

interactor.observeUser(forceReload = false)
    .subscribe(::updateProfile)

view.setSurname(“Horsrman”)
```

observeOn

```
view.setName("Bojack")

interactor.observeUser(forceReload = false)
        .observeOn(AndroidSchedullers.mainThread())
        .subscribe(::updateProfile)

view.setSurname("Horsrman")
```

Реальный Presenter

```
view.setName(“Bojack”)

 interactor.observeUser(forceReload = false)
        .observeOn(AndroidSchedullers.mainThread())
        .subscribe(::updateProfile)

view.setSurname(“Horsrman”)
```

Реальный Presenter

```
view.setName(“Bojack”)

interactor.observeUser(forceReload = false)
        .observeOn(AndroidSchedullers.mainThread())
        .subscribe(::updateProfile)

view.setSurname(“Horsrman”)
```

1

Реальный Presenter

```
view.setName(“Bojack”)

interactor.observeUser(forceReload = false)
        .observeOn(AndroidSchedullers.mainThread())
        .subscribe(::updateProfile)

view.setSurname(“Horsrman”)
```

1

2

Но почему?

AndroidSchedullers.mainThread()

```
public final class AndroidSchedulers {

HandlerScheduler(new
Handler(Looper.getMainLooper()), false);
```

Внутренности HandlerScheduller

```
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {

   Message message = Message.obtain(handler, scheduled);

    ....
    handler.sendMessageDelayed(message, unit.toMillis(delay));
    ....
}
```

1. Получаем сообщение из пула

```
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {

   Message message = Message.obtain(handler, scheduled);

    ....
    handler.sendMessageDelayed(message, unit.toMillis(delay));
    ....
}
```

2. Отправляем в очередь

```
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {

    Message message = Message.obtain(handler, scheduled);

     ....
    handler.sendMessageDelayed(message, unit.toMillis(delay));
     ....
}
```

6.1 Вспоминаем как работает Handler/Looper/MessageQueue

Producer/Consumer pattern

Looper

Управляет MessageQueue

Thread-local на поток

Looper потока

```
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
```

```
public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

Handler

Отправляет/Получает сообщения в MessageQueue

Напрямую связан с Looper

Create Handler() internals

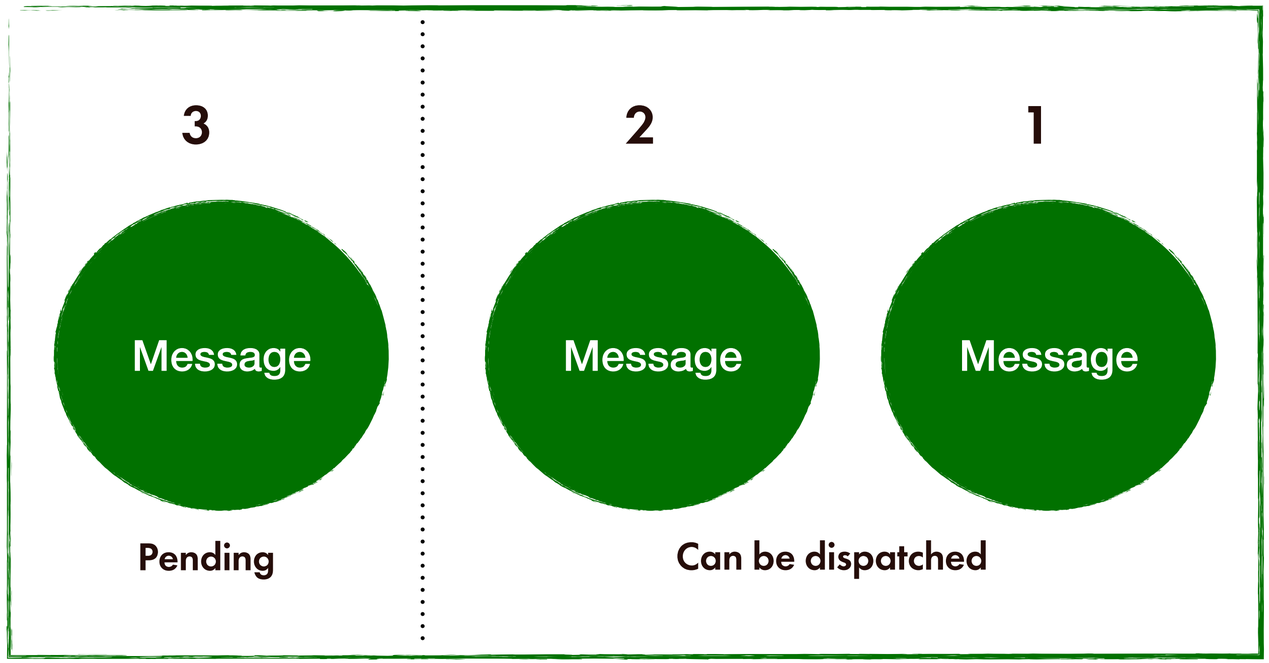
```
public Handler(@Nullable Callback callback, boolean async) {
    mLooper = Looper.myLooper();
}
```

MessageQueue

Отсутствует публичный API

Обрабатывает сообщения от Handler

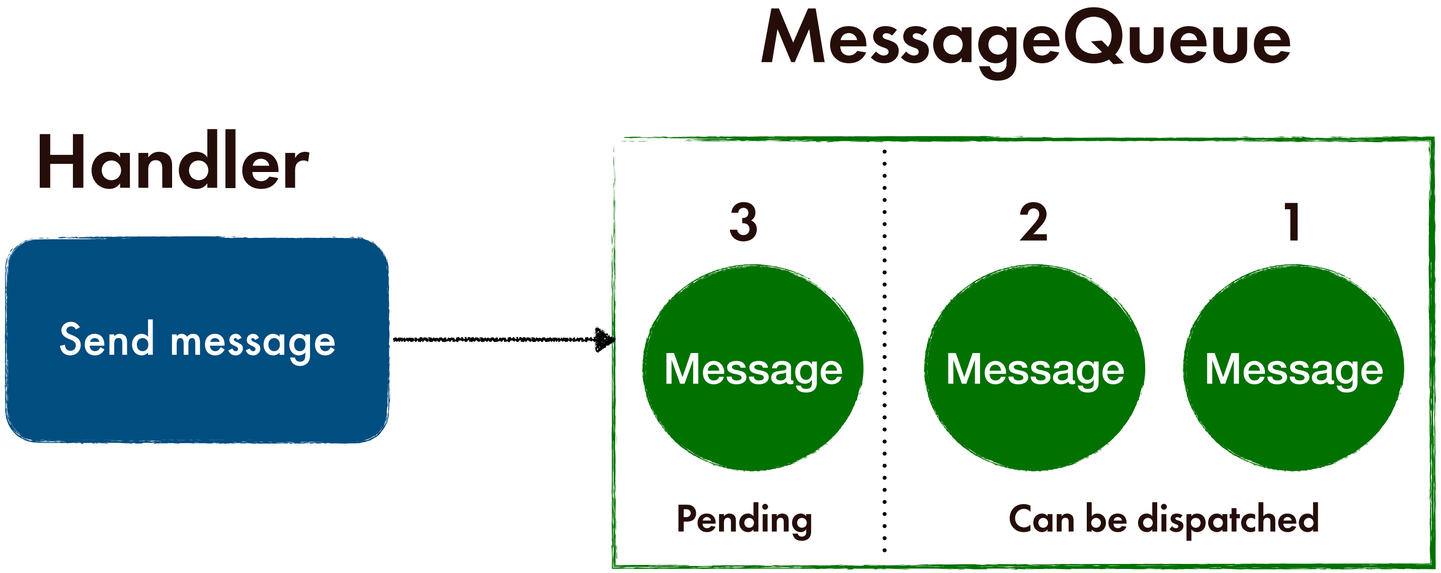
MessageQueue



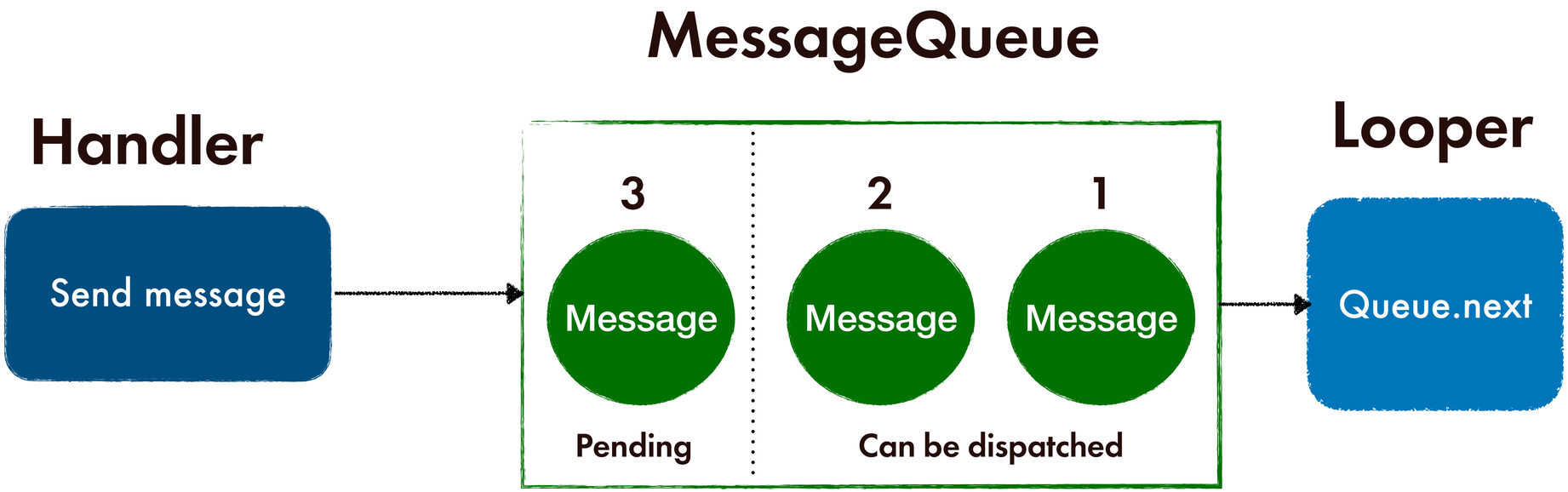
Handler: send message

Handler

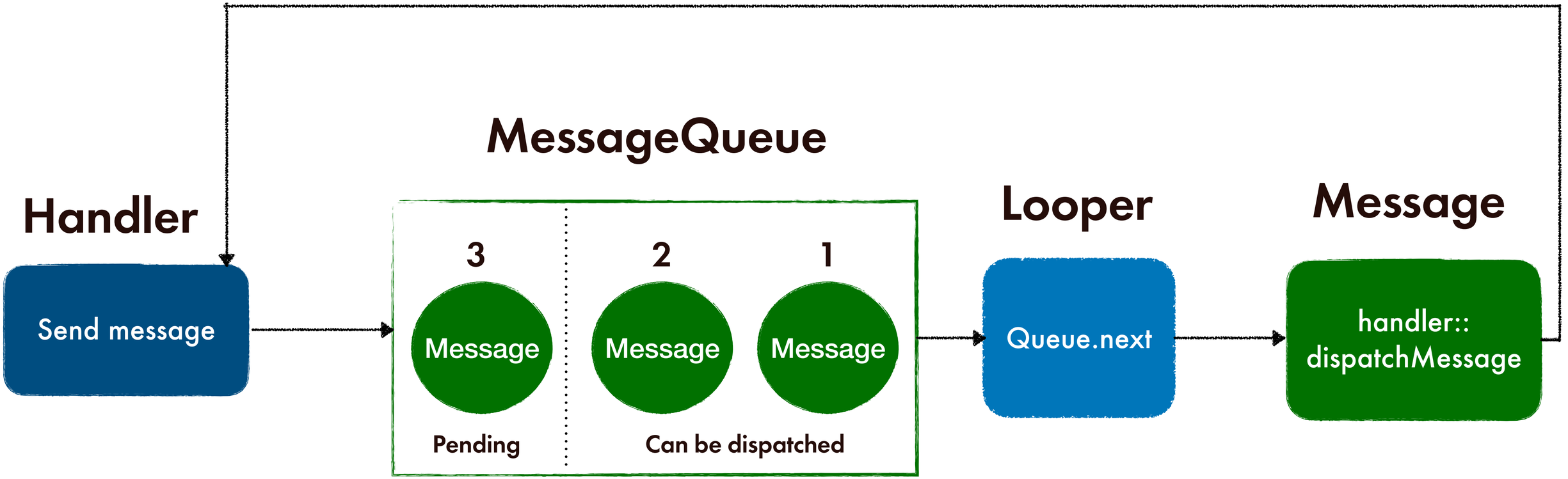
Обработка сообщений



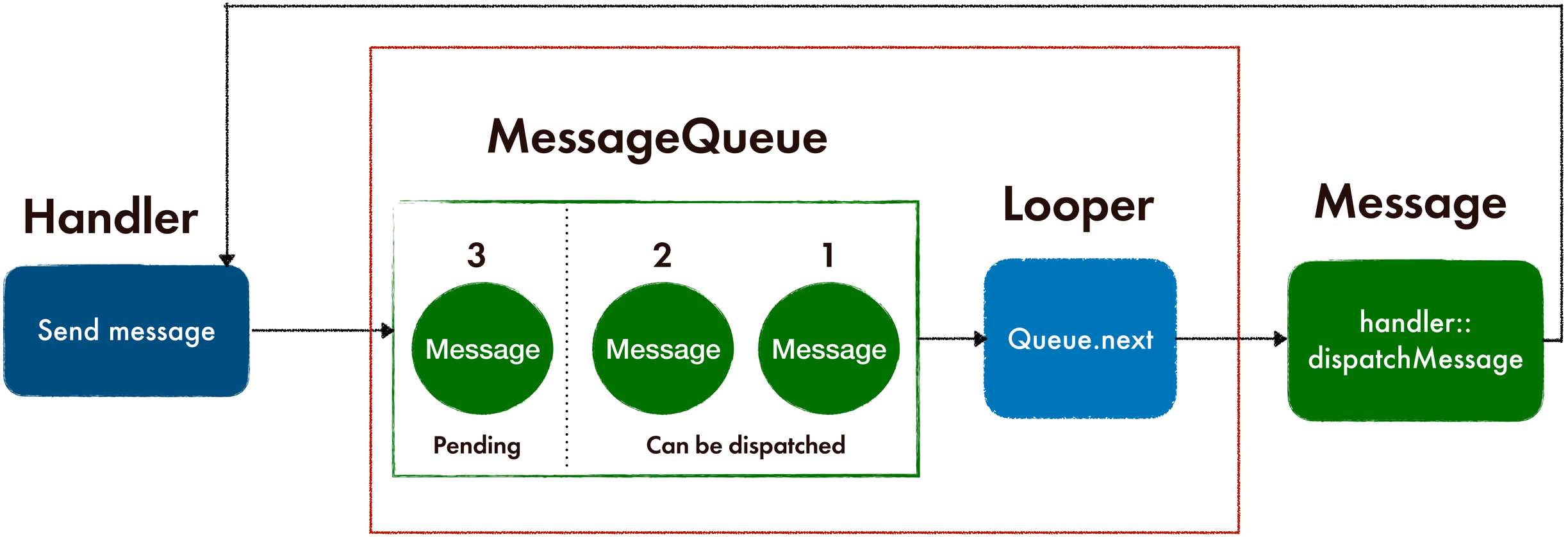
Looper: next message



Message:handler:dispatch



Причина задержки



6.2 Идём без очереди

ImmediateHandlerScheduller

```
public Disposable schedule(Runnable run, long delay, TimeUnit unit) {
   Message message = Message.obtain(handler, scheduled);
    ....
   if (Looper.myLooper() == handler.looper && delay <= 0) {
        handler.dispatchMessage(message)
   } else {
        handler.sendMessageDelayed(message, unit.toMillis(delay))
   }
   ....
}
```

Реальный Presenter

```
view.setName(“Bojack”)

interactor.observeUser(forceReload = false)
        .observeOn(RxDataAndroidSchedullers.mainThread())
        .subscribe(::updateProfile)

view.setSurname(“Horsrman”)
```

Что может пойти не так:

Рост ANR с ростом стримов

7. Стратегии обновления данных

Time-Based

7.1 Делаем ещё одну абстракцию

1. ForceReload

```
class TimeBasedForceReloadStrategy<Params : Any>
```

Update зависит отпараметров

Like userId

```
class TimeBasedForceReloadStrategy<Params : Any>
```

Store timestamp

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit)
)
```

Get timestamp

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?
)
```

Life time

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?,
    private val lifeTimeInMillis: Long

)
```

Store timestamp

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?,
    private val lifeTimeInMillis: Long
) {

    fun storeTime(params: Params) {
        storeTimeStamp.invoke(params, DateTime.now().millis)
    }
}
```

ForceReload = true :D

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?,
    private val lifeTimeInMillis: Long
) {

    fun calculateForceReload(forceReload: Boolean, params: Params): Boolean = when {
        forceReload -> true
    }
}
```

Last update is null

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?,
    private val lifeTimeInMillis: Long
) {

    fun calculateForceReload(forceReload: Boolean, params: Params): Boolean = when {
        forceReload -> true
        getLastUpdateTime(params) == LAST_UPDATE_TIME_EMPTY -> true
    }
}
```

Cache expired

```
class TimeBasedForceReloadStrategy<Params : Any>(
    private var storeTimeStamp: ((key: Params, timeInMills: Long) -> Unit),
    private var getTimeStamp: (key: Params) -> Long?,
    private val lifeTimeInMillis: Long
) {

    fun calculateForceReload(forceReload: Boolean, params: Params): Boolean = when {
        forceReload -> true
        getLastUpdateTime(params) == LAST_UPDATE_TIME_EMPTY -> true
        else -> isExpired(params)
    }
}
```

7.2 Используем в Repository

Создаём стратегию

```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {
```

```
 private val userForceReloadStrategy = TimeBasedForceReloadStrategy<String>(
     storeTimeStamp = { key, timeInMillis -> memoryCache[key] = timeInMillis },
     getTimeStamp = { key -> memoryCache[key] },
     lifeTimeInMillis = TimeUnit.MINUTES.toMillis(1)
 )
```

Может быть: Prefs, Memory Cache

```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {
```

```
 private val userForceReloadStrategy = TimeBasedForceReloadStrategy<String>(
     storeTimeStamp = { key, timeInMillis -> memoryCache[key] = timeInMillis },
     getTimeStamp = { key -> memoryCache[key] },
     lifeTimeInMillis = TimeUnit.MINUTES.toMillis(1)
 )
```

Observe user

```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {

private val userForceReloadStrategy = TimeBasedForceReloadStrategy<String>(
    storeTimeStamp = { key, timeInMillis -> memoryCache[key] = timeInMillis },
    getTimeStamp = { key -> memoryCache[key] },
    lifeTimeInMillis = TimeUnit.MINUTES.toMillis(1)
)
```

```
override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    userDelegate.observe(Unit, forceReload)
```

Observe user

```
internal class UserRepositoryImpl @Inject constructor(
    private val userService: UserService,
    private val dao: UserDao,
    private val memoryCache: MemoryCache<User>
) : UserRepository {

private val userForceReloadStrategy = TimeBasedForceReloadStrategy<String>(
  storeTimeStamp = { key, timeInMillis -> memoryCache[key] = timeInMillis },
  getTimeStamp = { key -> memoryCache[key] },
  lifeTimeInMillis = TimeUnit.MINUTES.toMillis(1)
)

override fun observeUser(forceReload: Boolean): Observable<Data<User>> =
    userDelegate.observe(Unit,
    userForceReloadStrategy.calculateForceReload(
              forceReload,
             ”params_key_withdrawal_limits”
      )
    )
)
```

Альтернатива

Latest Recently Used (LRUCache)

RxData

```
implementation 'com.revolut.rxdata:dod:1.2.5'
implementation 'com.revolut.rxdata:core:1.2.8'
```

https://github.com/revolut-mobile/RxData

Roman Iatcyna

@yatsinar

8. Extensions

Observable Extensions

extractContent() //Маппит контент если он есть

extractDataOrError() //Маппит контент если он есть или кидает ошибку

extractDataIfLoaded() //Маппит контент если он есть и если loading = false

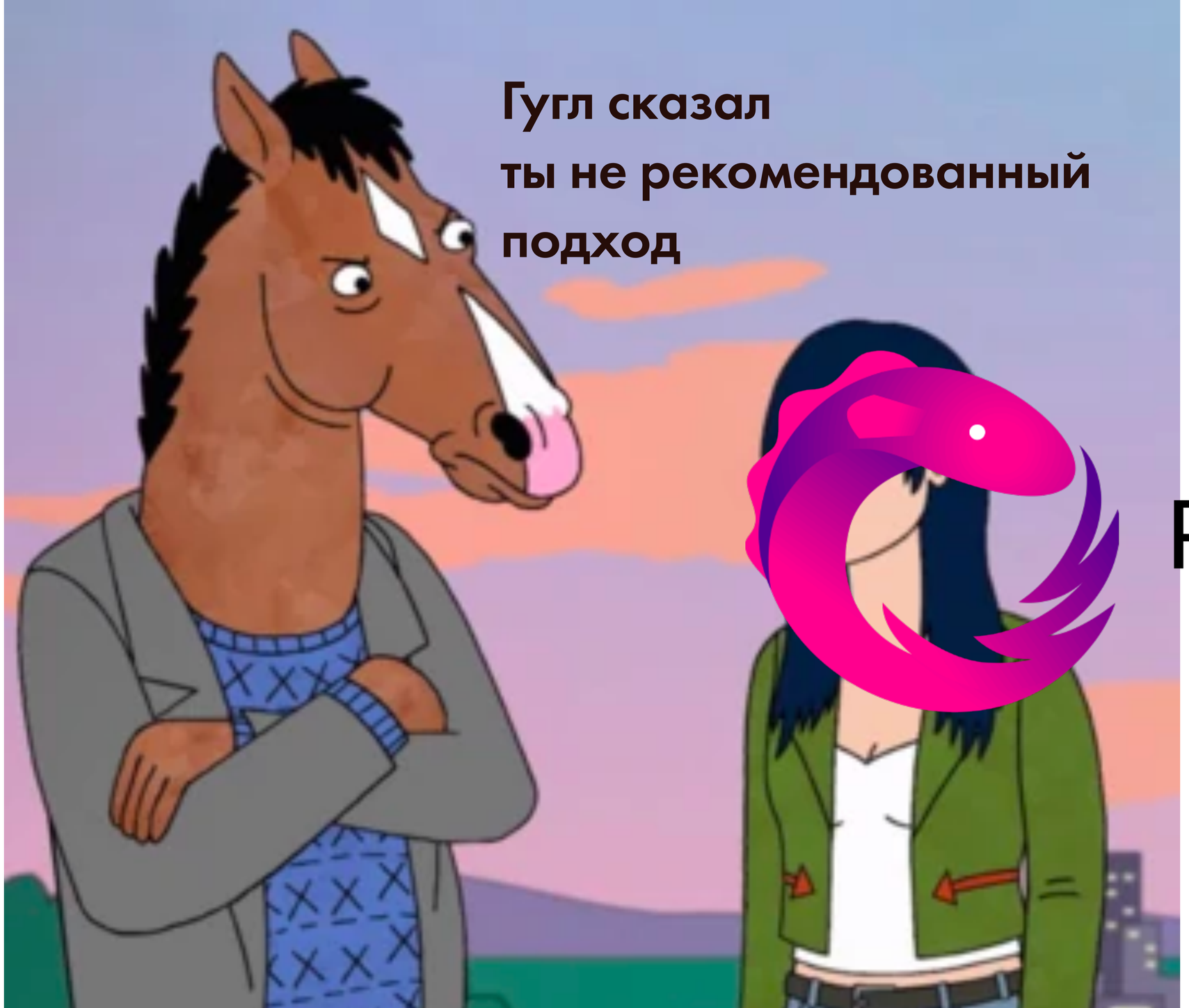
skipWhileLoading() //Ждёт, когда loading будет false

combineLatestData() //Комбинирует источники данных с Data

....

https://github.com/revolut-mobile/RxData/tree/master/core/src/main/java/com/revolut/rxdata/core/extensions

С RX закончили



RxJava


RxJava
5
RxJava
ДОМ ПРЕСТАРЕЛЫХ
RxJava
Чилик


Flow

9. Flow dat!

Факты о Flow

1. Стабилен с 1.3.0 (current 1.4.0)

2. Рекомендован гуглом

Помните цепочку?

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

FlowOf

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

```
flowOf("Hey")
```

OnStart

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

```
flowOf("Hey")
    .onStart {   }
```

FlowOn

```
Observable.just("Hey")
    .doOnSubscribe {  }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

```
flowOf("Hey")
    .onStart {  }
    .flowOn(Dispatchers.IO)
```

FlowOn

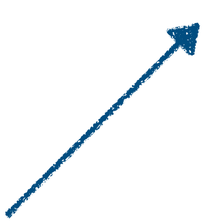
```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

```
flowOf("Hey")
    .onStart {  }
    .flowOn(Dispatchers.IO)
```

Это не совсем то же самое

CoroutineScope

```
Observable.just("Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```



```
CoroutineScope(Dispatchers.Default).launch {
    flowOf("Hey")
        .onStart {  }
        .flowOn(Dispatchers.IO)
}
```

Collect

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe {

    }
```

```
CoroutineScope(Dispatchers.Default).launch {
    flowOf("Hey")
        .onStart {  }
        .flowOn(Dispatchers.IO)
        .collect {

        }
}
```

Те же шаги что и в Rx

1. Создание источников
2. Подписка источников
3. Emitting данных

10. Threading: Rx vs Flow

RX: SubscribeOn

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .subscribe {

    }
```

RX: up stream

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .subscribe {

    }
```

RX: up stream and down

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .subscribe {

    }
```

RX: up stream and down

```
Observable.just(“Hey")
    .doOnSubscribe {   }
    .subscribeOn(Schedulers.io())
    .subscribe {

    }
```

Сломать поток в большой цепочке очень легко

Flow: Context preservation

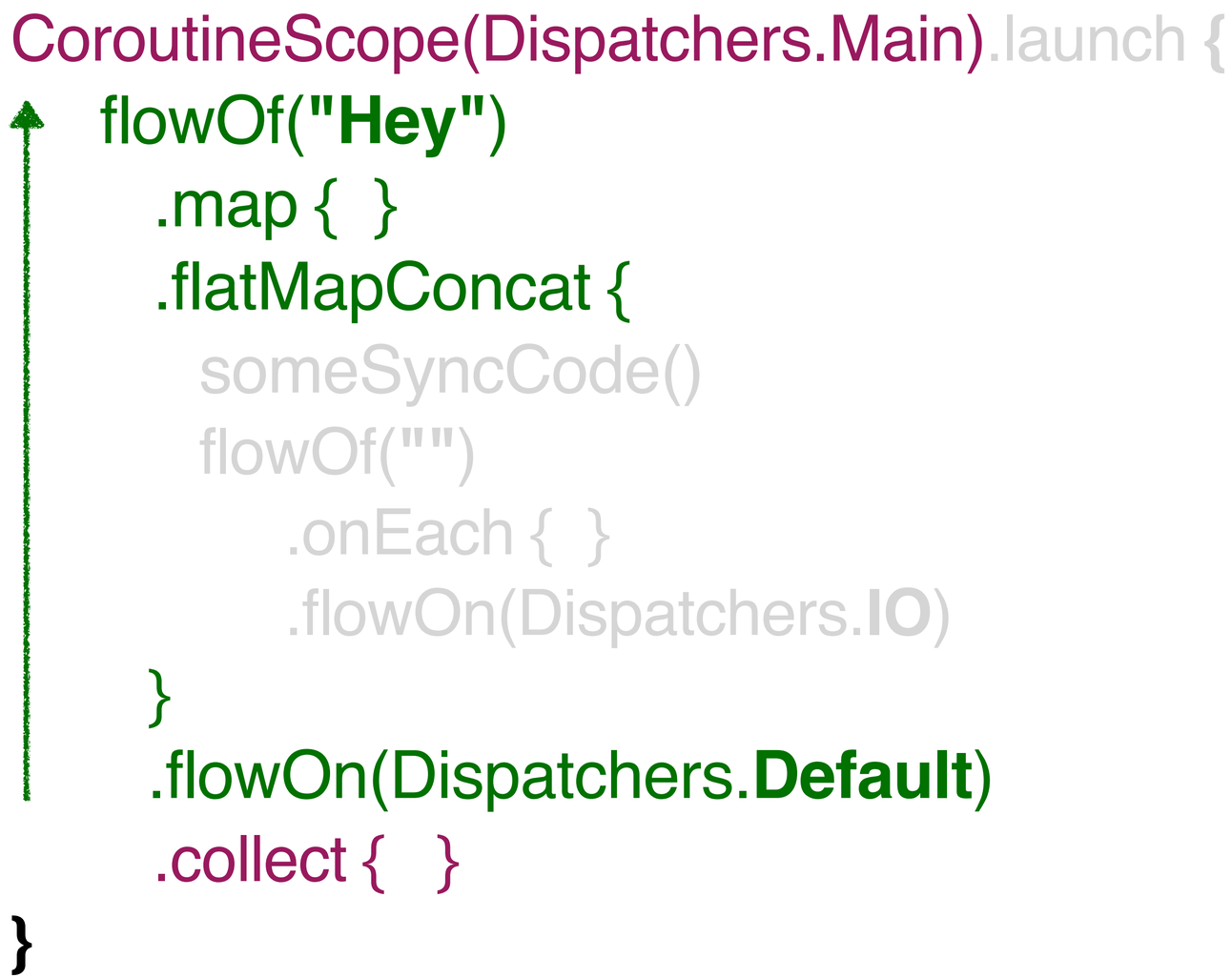
FlowOn

```
CoroutineScope(Dispatchers.Main).launch {
    flowOf("Hey")
        .map {  }
        .flatMapConcat {
            someSyncCode()
            flowOf("")
                .onEach { }
                .flowOn(Dispatchers.IO)
        }
        .flowOn(Dispatchers.Default)
        .collect {   }
}
```

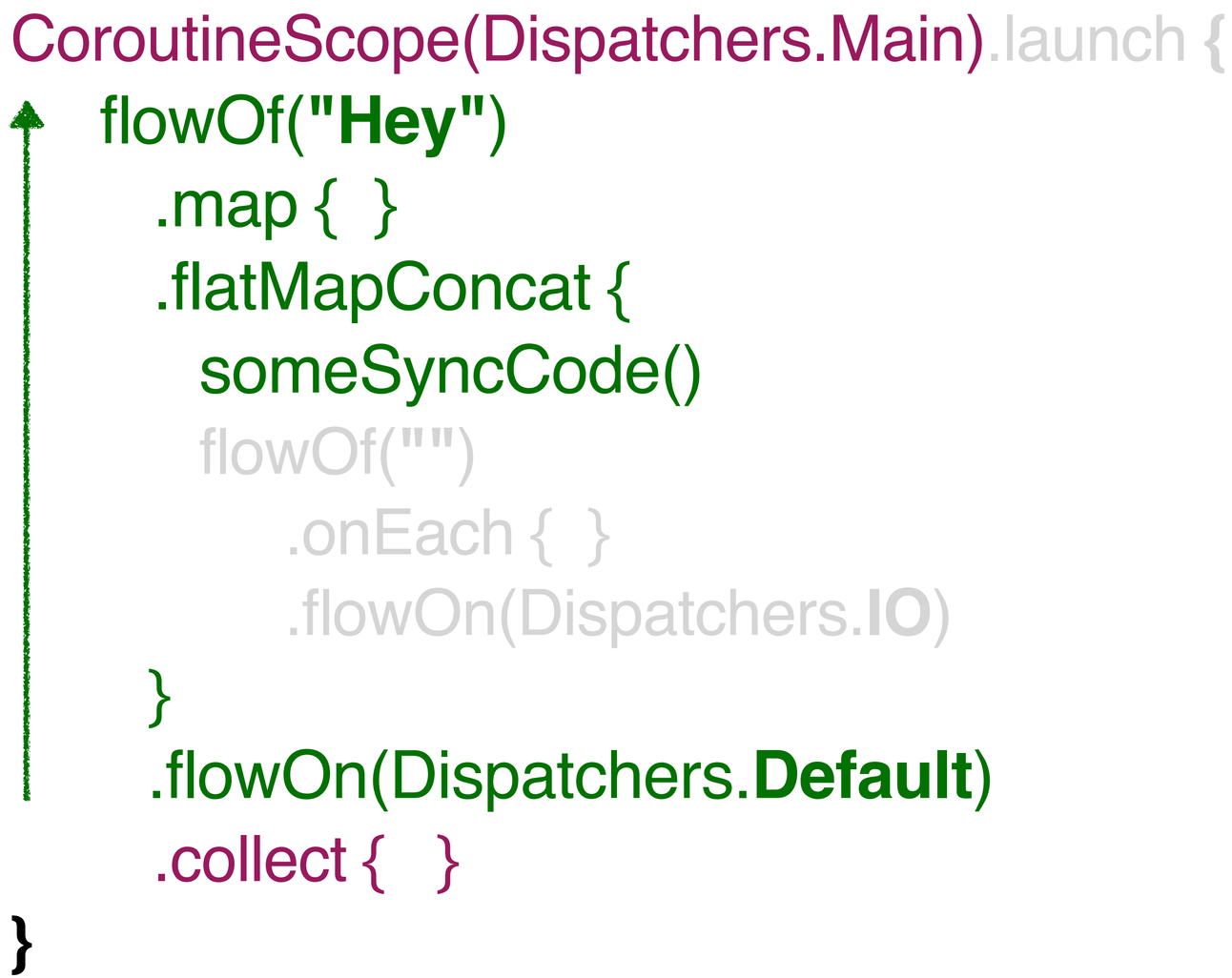
Collect: Гарантированно на main

```
CoroutineScope(Dispatchers.Main).launch {
    flowOf("Hey")
        .map {  }
        .flatMapConcat {
            someSyncCode()
            flowOf("")
                .onEach {  }
                .flowOn(Dispatchers.IO)
        }
        .flowOn(Dispatchers.Default)
        .collect {   }
}
```

flowOn: Изменяет контекст только upstream



SomeSyncCode — часть стрима flatMapConcat



```
CoroutineScope(Dispatchers.Main).launch {
    flowOf("Hey")
        .map {  }
        .flatMapConcat {
            someSyncCode()
            flowOf("")
                .onEach {  }
                .flowOn(Dispatchers.IO)
        }
        .flowOn(Dispatchers.Default)
        .collect {  }
}
```

FlowOn(IO)

```
CoroutineScope(Dispatchers.Main).launch {
    flowOf("Hey")
        .map {  }
        .flatMapConcat {
            someSyncCode()
            flowOf("")
                .onEach {  }
                .flowOn(Dispatchers.IO)
        }
        .flowOn(Dispatchers.Default)
        .collect {   }
}
```

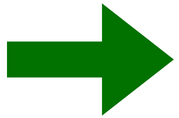
11. Что поменялось в реализации?

Что поменялось в реализации

PublishSubject.toSeriallized()

Что поменялось в реализации

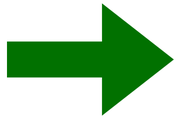
PublishSubject.toSeriallized()



MutableSharedFlow

Что поменялось в реализации

PublishSubject.toSeriallized()



MutableSharedFlow

subscribeOn

Что поменялось в реализации

PublishSubject.toSeriallized() → MutableSharedFlow

subscribeOn → flowOn

Что поменялось в реализации

PublishSubject.toSeriallized()	→	MutableSharedFlow
subscribeOn	→	flowOn
onErrorResumeNext	→	catch + emitAll
concat	→	onCompletion + emitAll
defer	→	flow + emitAll

12. Immediate scheduler

Типичная работа

```
CoroutineScope(Dispatchers.Main).launch {
    delegate.observe(params = Unit, forceReload = true)
        .collect {

        }
}
```

Типичная работа

```
CoroutineScope(Dispatchers.Main).launch {
    delegate.observe(params = Unit, forceReload = true)
        .collect {

        }
}
```

Immediate

```
CoroutineScope(Dispatchers.Main.immediate).launch {
    delegate.observe(params = Unit, forceReload = true)
        .collect {

        }
}
```

13. Shared request

Merged!

https://github.com/Kotlin/kotlinx.coroutines/pull/2069

SharedRequest

replay(1).refCount()

```
.shareIn(scope = CoroutineScope(Dispatchers.IO),
         started = SharingStarted.WhileSubscribed(),
         replay = 1)
```

FlowData

```
implementation  TBD
```

https://github.com/NoNews/FlowData

Перевод с RX на Flow

Migration.kt, идёт с flow из коробки

Пишите прямо в Rx-стиле

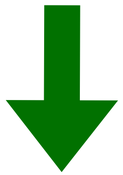
Migration.kt, идёт с flow из коробки

flowOf(“1”)~~.observeOn()~~

Пишите прямо в Rx-стиле

Migration.kt, идёт с flow из коробки

flowOf(“1”).~~observeOn()~~



```
* withContext(Dispatchers.IO) {
*     flow.collect { value -> println("Received $value") }
* }

@Deprecated(message = "Collect flow in the desired context instead", level = DeprecationLevel.ERROR)
public fun <T> Flow<T>.observeOn(context: CoroutineContext): Flow<T> = noImpl()
```

Что делать с overdraw?

RecyclerView-based

Epoxy

Litho

Compose (in beta)

Что получили

Требуется только **реализовать абстракции**

Легко отказаться от слоёв кеширования

Нет соблазна реализовать offline в domain

Нельзя нарушить контракт стратегии

Быстрый рендеринг экранов

Обновление всего контента из одного места

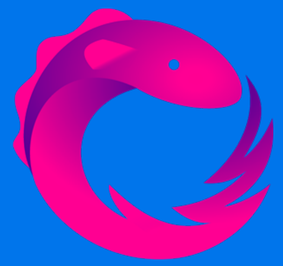
Полезные ссылки

1. https://bit.ly/36dDv2N *(С. Рябов, Rx + Flow)*
2. https://bit.ly/32pNlgL *(Р. Яцына, статья про Reactive Data Flow)*
3. https://bit.ly/2IfOJvH *(Д. Мовчан, А. Быков, П. Щегельский, Recycler-Based)*
4. https://bit.ly/38owRt1 *(Р. Яцына, доклад про Reactive Data Flow)*
5. https://bit.ly/38nfol1 *(V. Drobushkov, From RxJava to KotlinFlow)*
6. https://bit.ly/35bmLKm *(Р. Елизаров, Asynchronous Data Streams)*

Revolut

Alexey Bykov Android Software Engineer @ Revolut

RxData



FlowData

@nonewsss