ТИНЬКОФФ

Фасады, асинхронщина и трубы

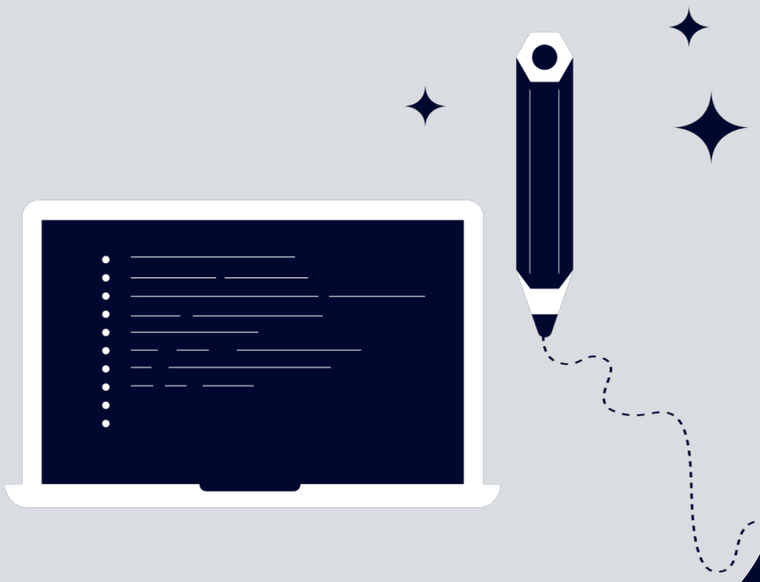
Декларативный подход к потокам данных и сопутствующие инструменты

ТИНЬКОФФ

Даровских Александр | Bring a Friend

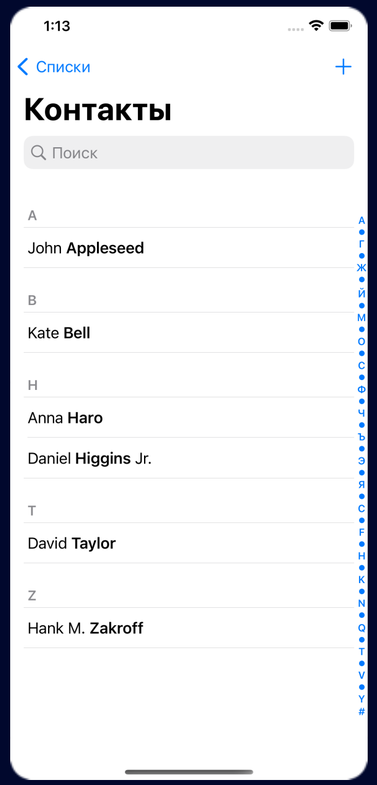
ext.adarovskikh@tinkoff.ru

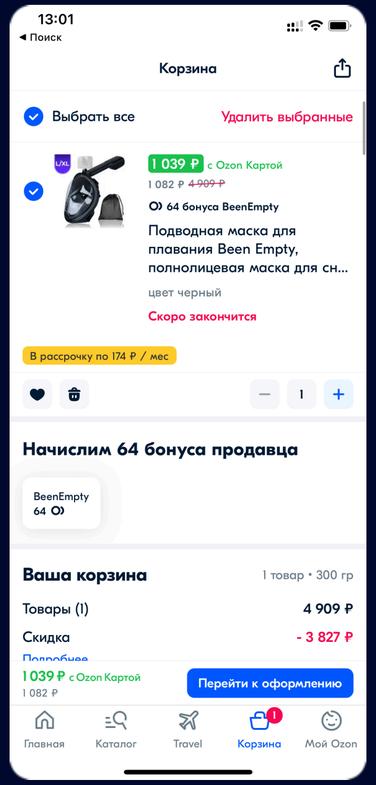
Краткое содержание

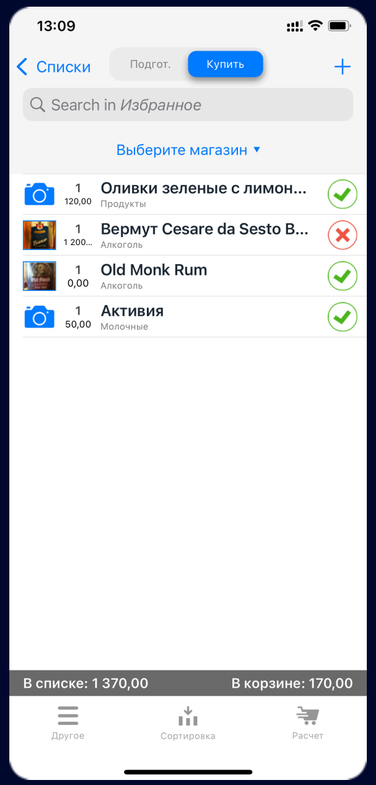


- Подсветим задачу
- Рассмотрим текущее положение дел
- Предложим вариант решения
- Создадим сопутствующие инструменты
- Немного затронем тестирование предложенного подхода

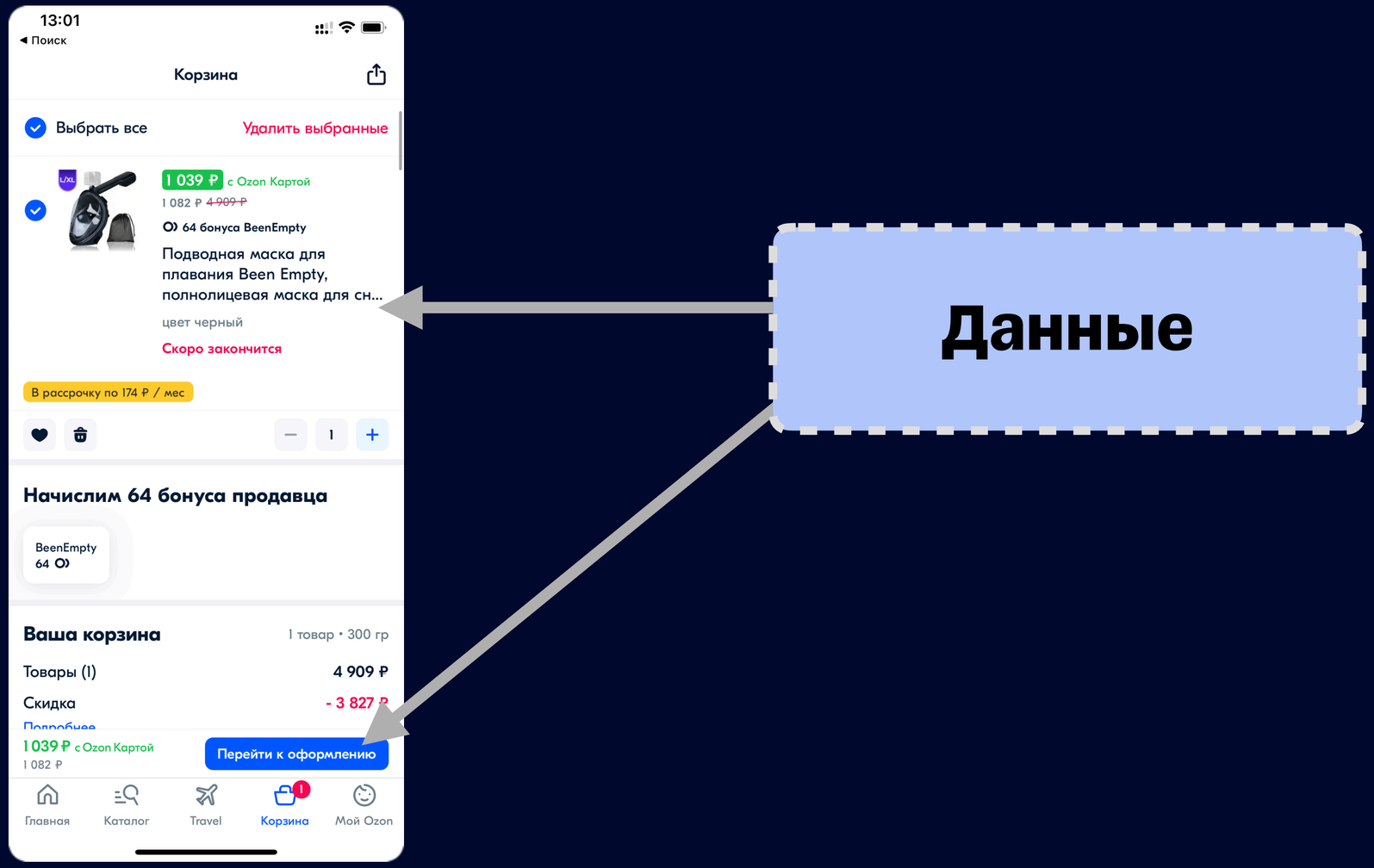
Зачем это всё?



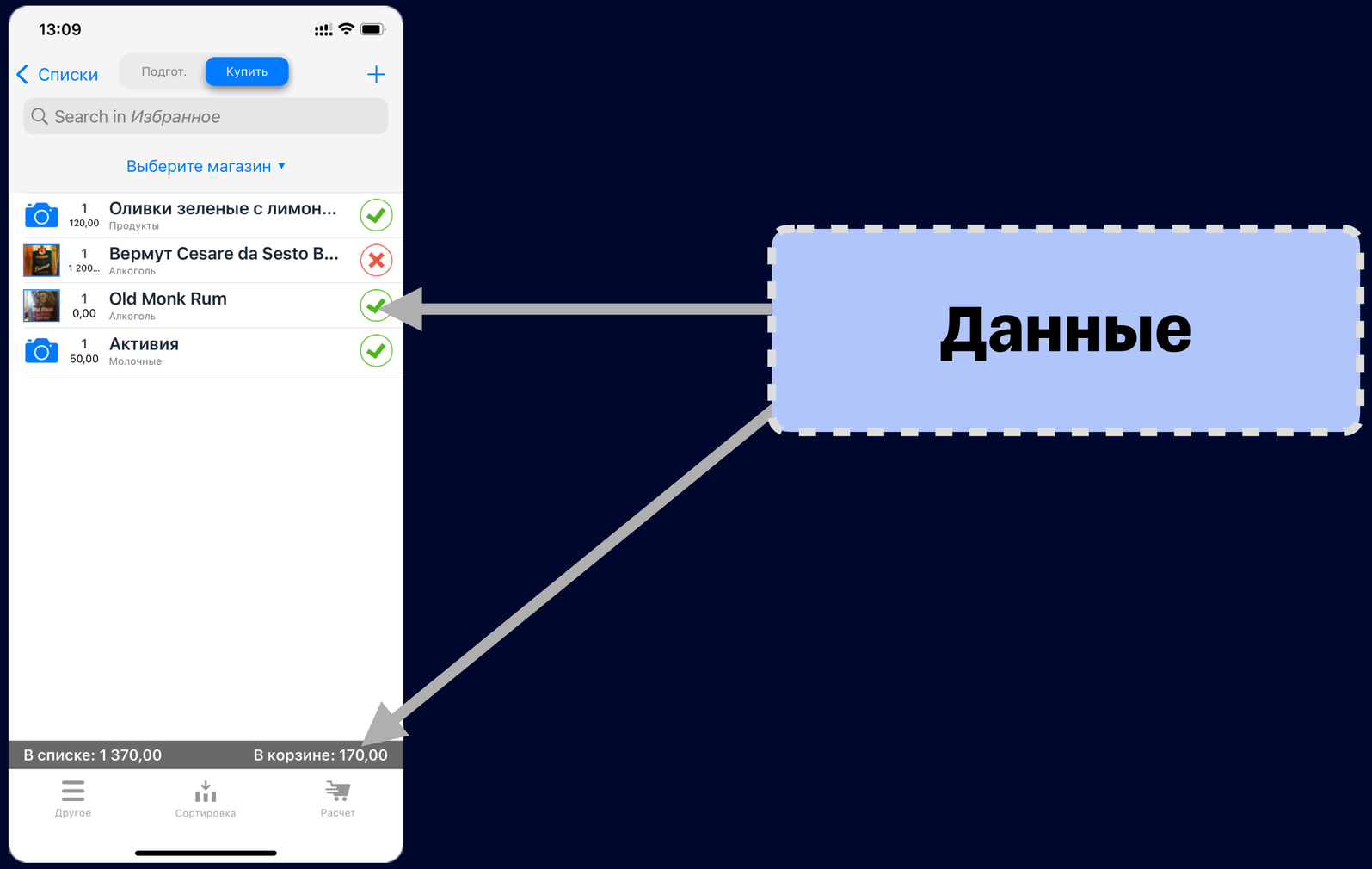




Зачем это всё?



Зачем это всё?



Текущий фасад

Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load(completion: @escaping (Result<[SomeViewModel], Error>) -> Void)
}
```

Текущий фасад

Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load(completion: @escaping (Result<[SomeViewModel], Error>) -> Void)
}


final class SomePresenter : ISomePresenter {
    func viewDidLoad() {
        facade.load([weak self] result in
            guard self else { return }
            // на каком мы потоке?
            // сколько раз вызовут коллбэк?
            // можно ли блокировать этот поток?
        }
    }
}
```

Текущий фасад

Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load(completion: @escaping (Result<[SomeViewModel], Error>) -> Void)
}
```

Текущий фасад

Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}
```

Текущий фасад

Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}
```

Текущий фасад

Что с ним не так?

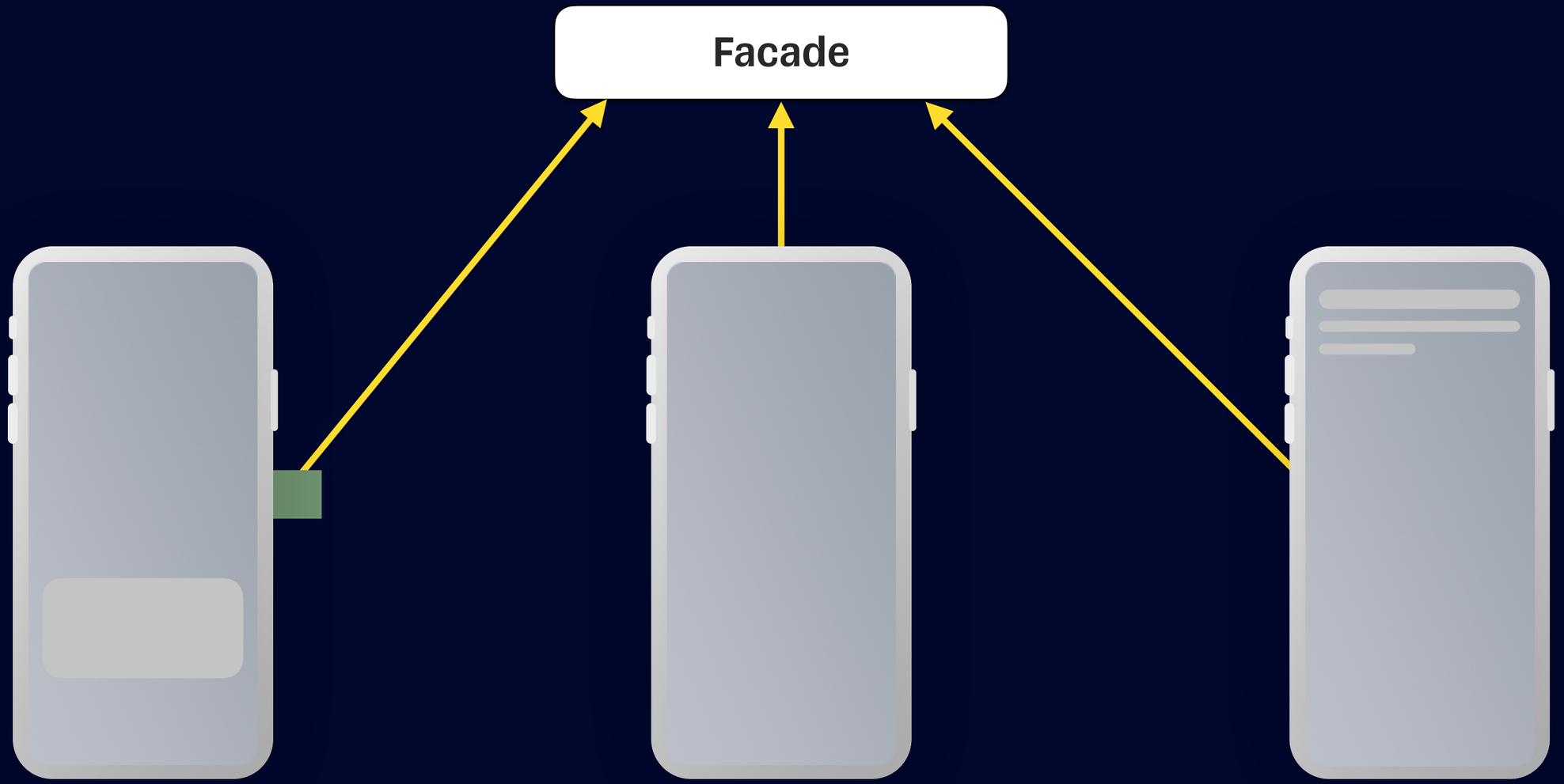
```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}

final class SomePresenter : ISomePresenter {
    func viewDidLoad() {
        facade.load([weak self] result in
            guard self else { return }
            // на каком мы потоке?
            // сколько раз вызовут коллбэк?
            // можно ли блокировать этот поток?
        }
    }
}
```

Текущий фасад

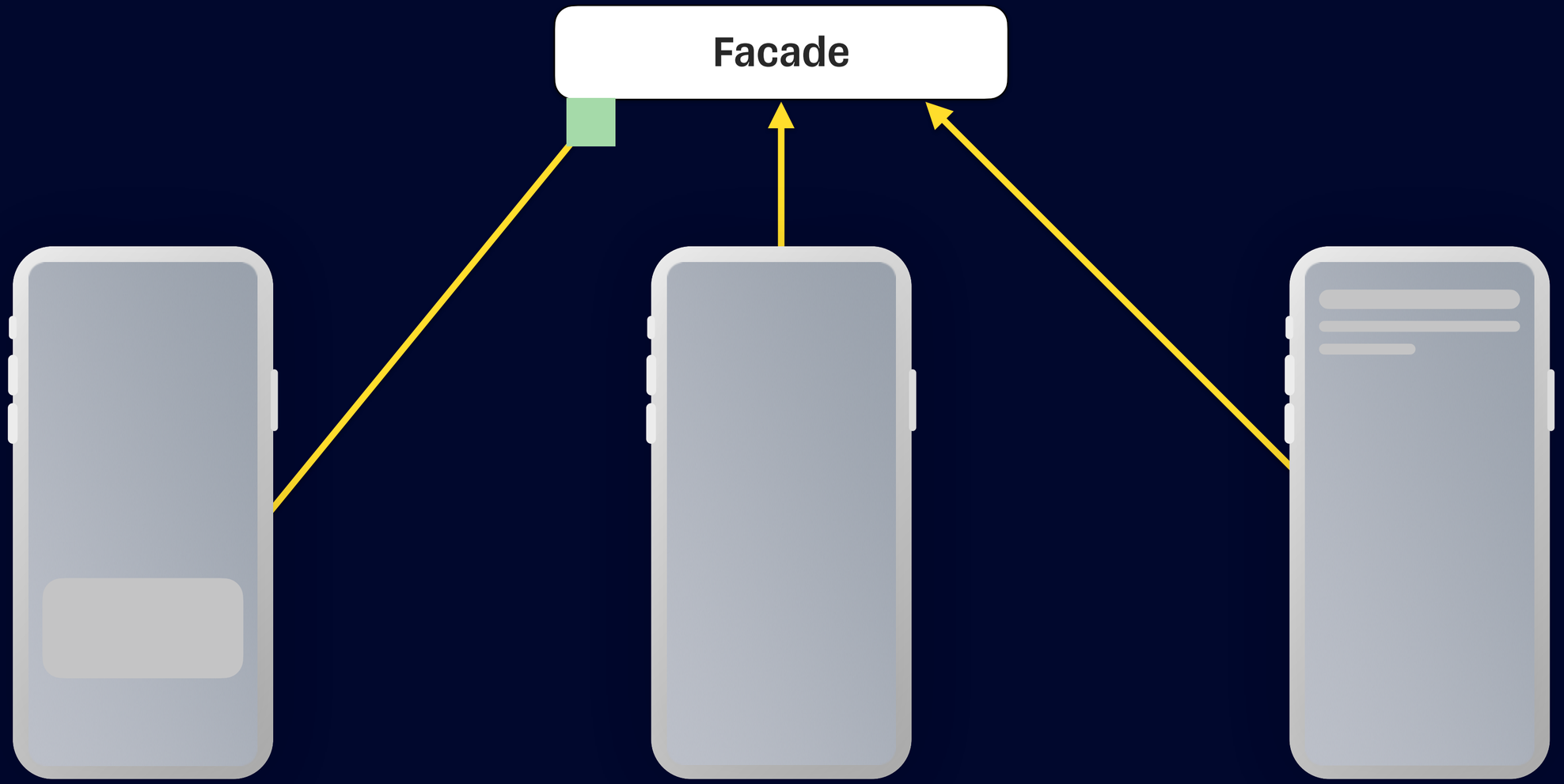
Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}

final class SomePresenter : ISomePresenter {
    func viewDidLoad() async {
        let models = await facade.load()
   }
}
```

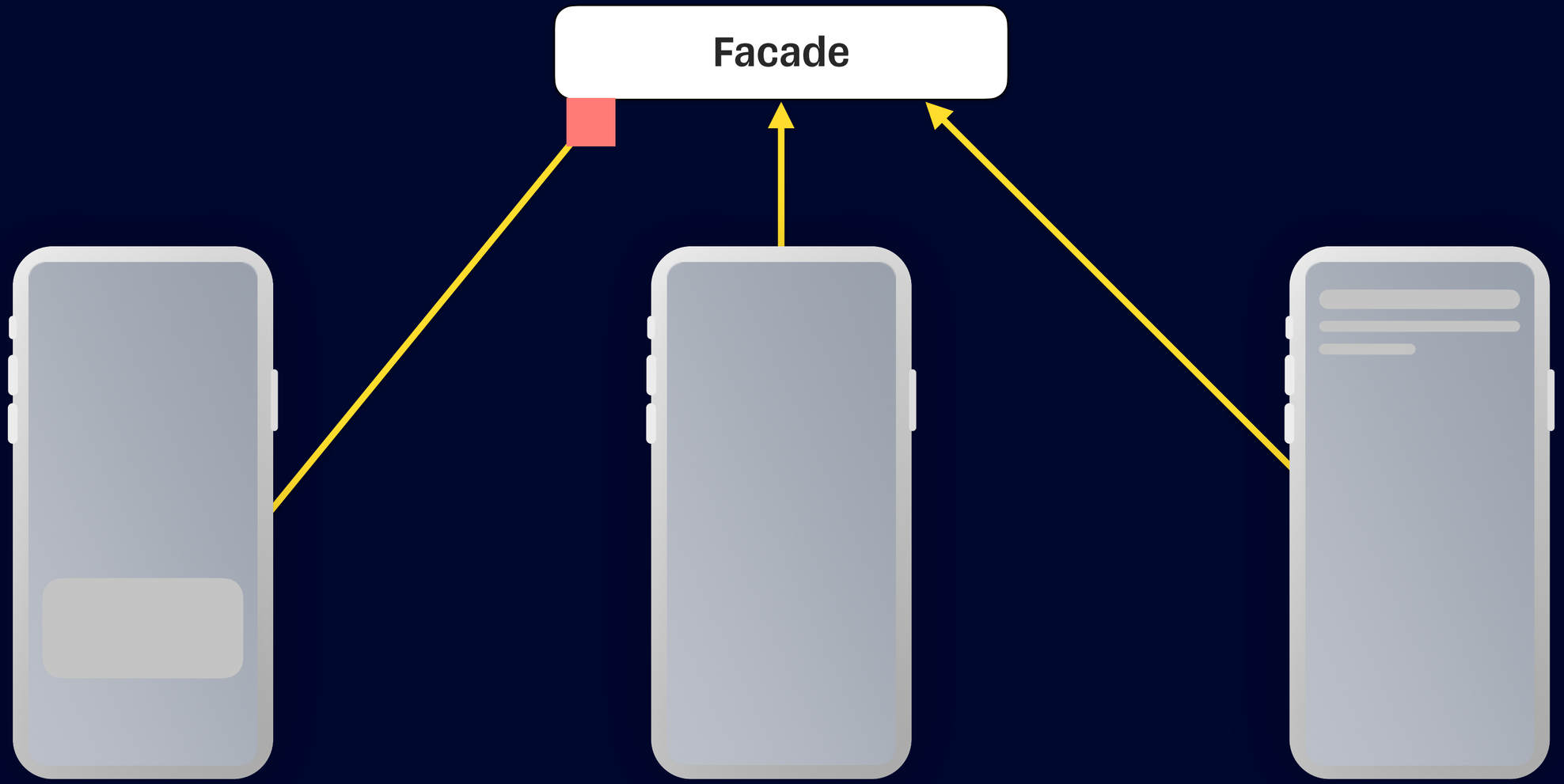
Обновление данных



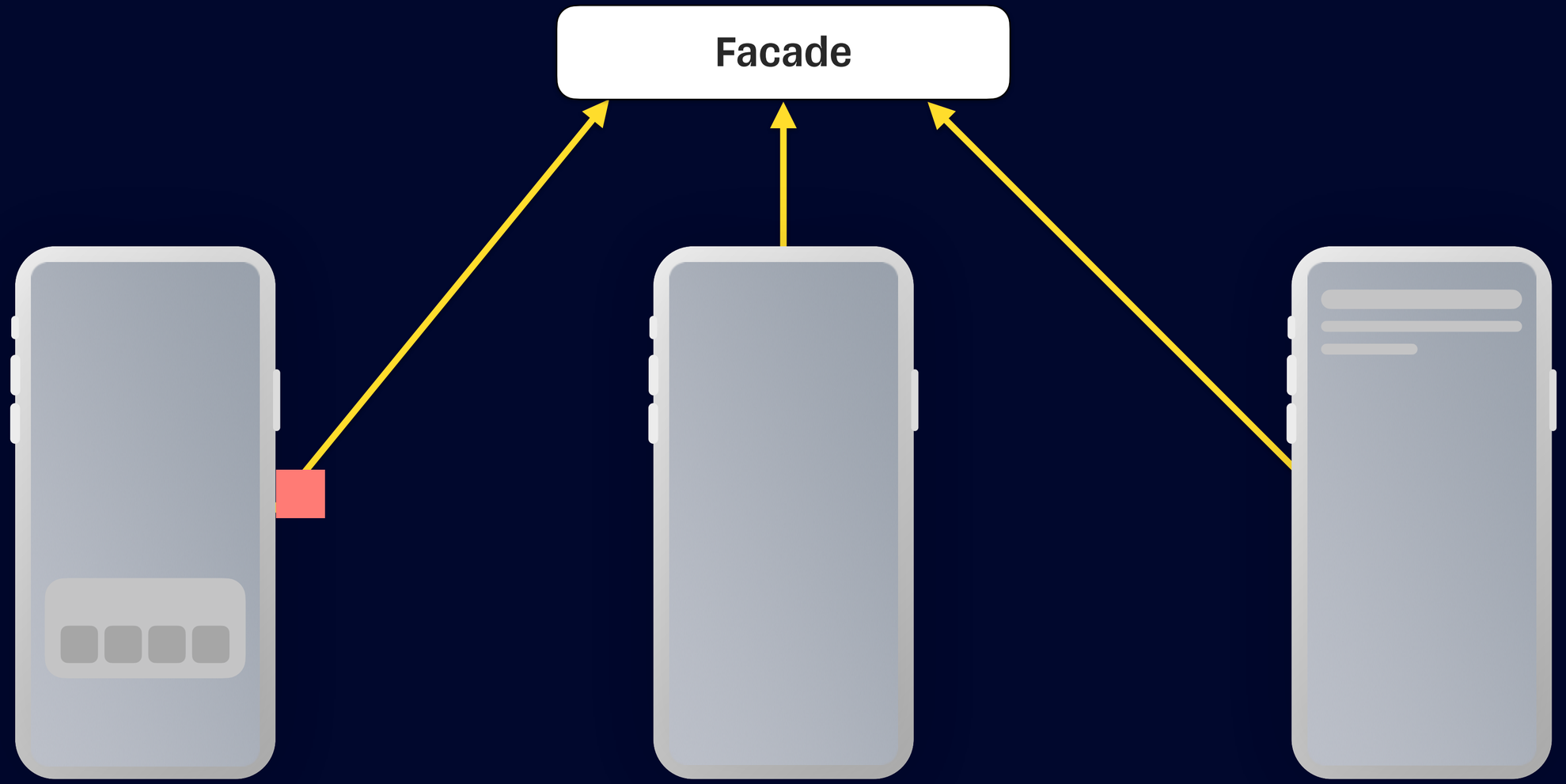
Обновление данных



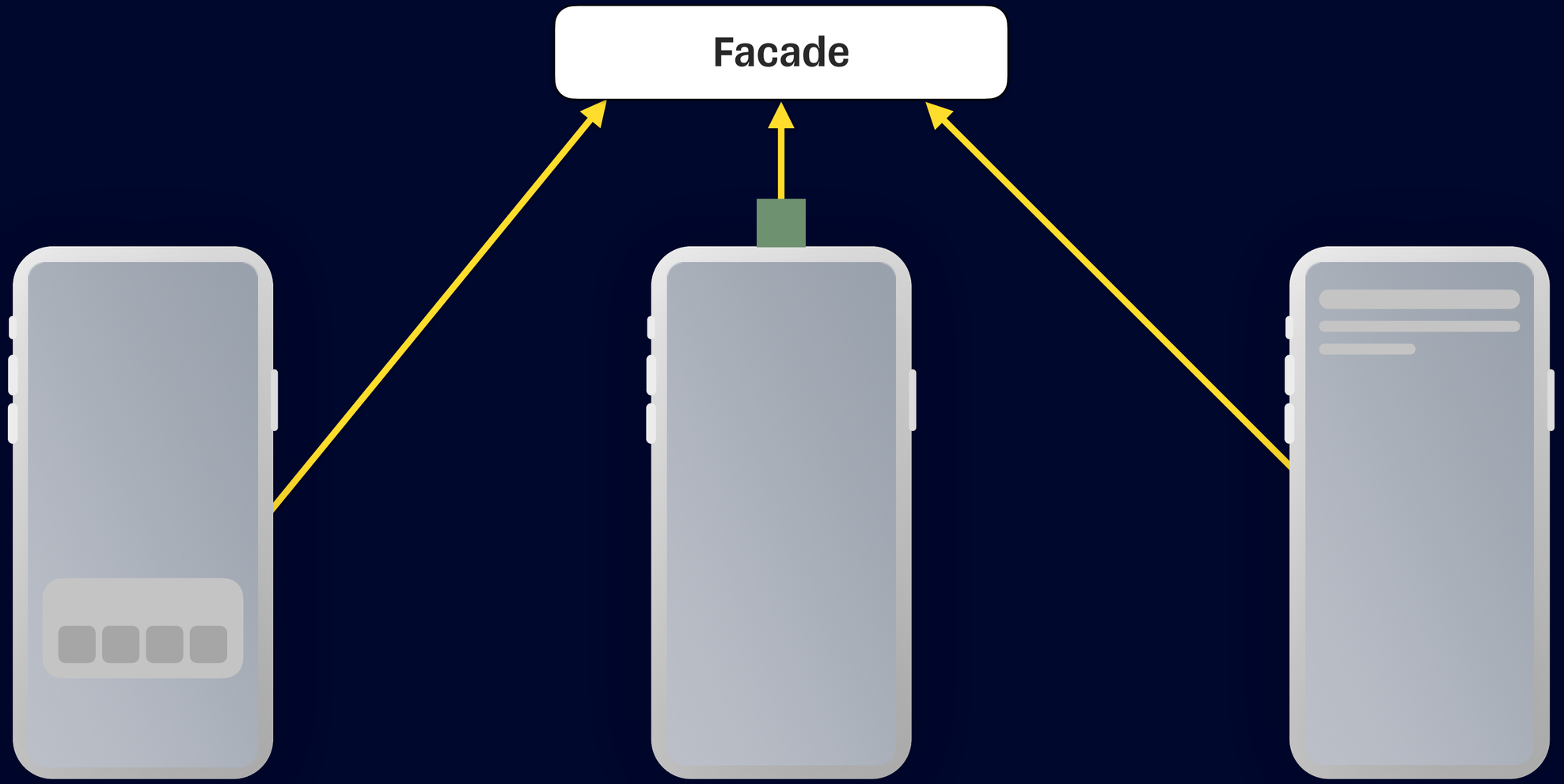
Обновление данных



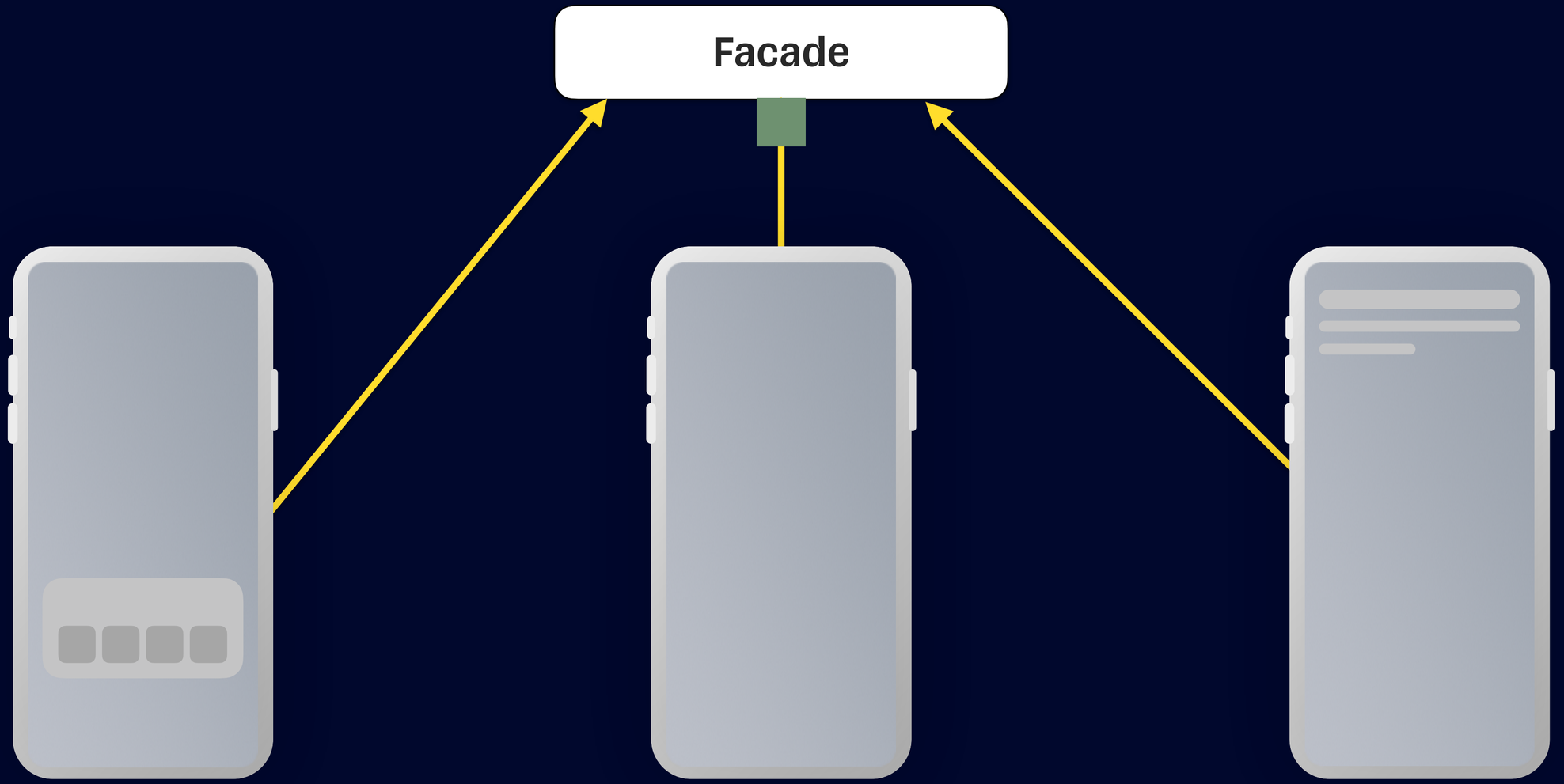
Обновление данных



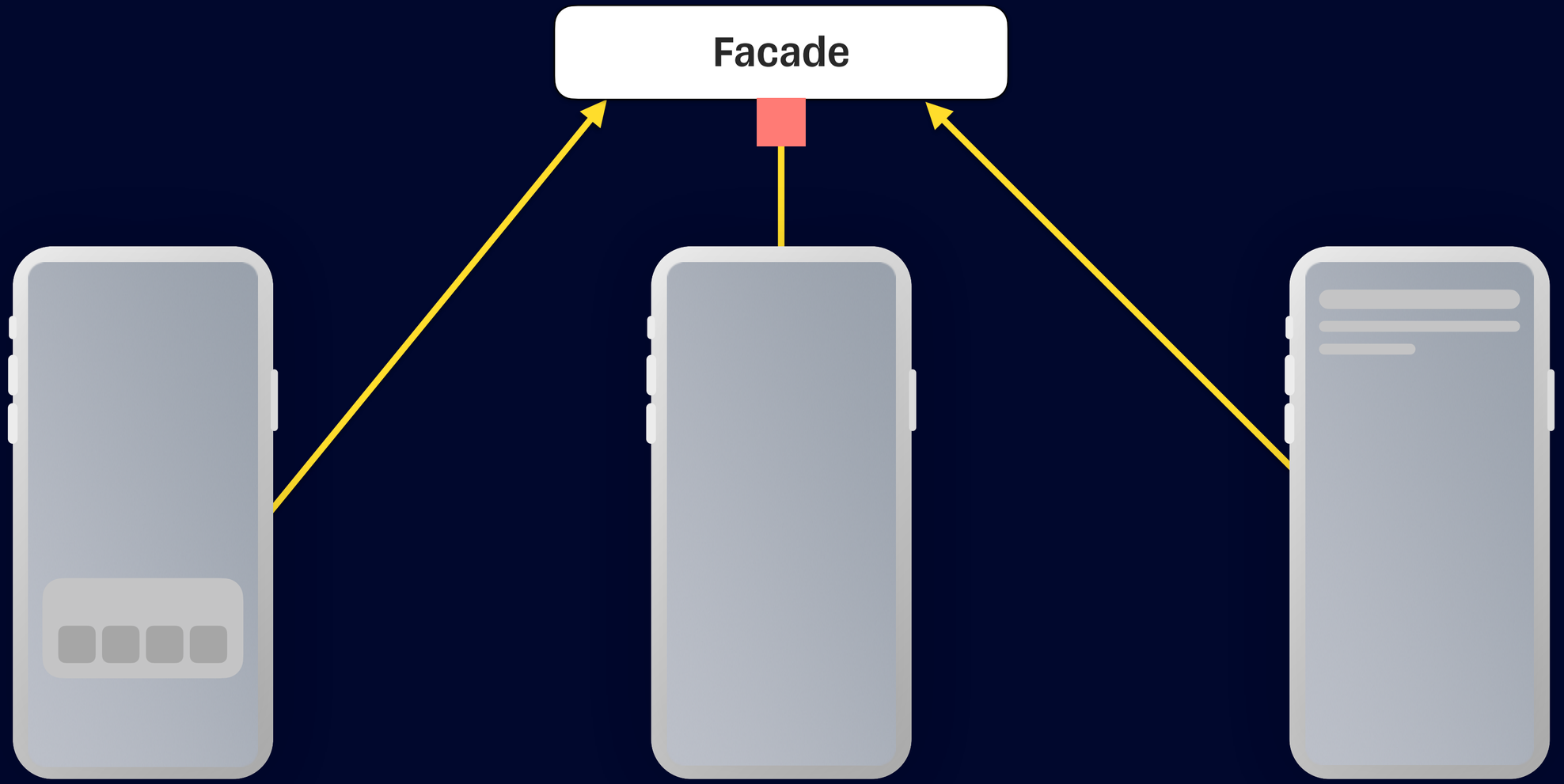
Обновление данных



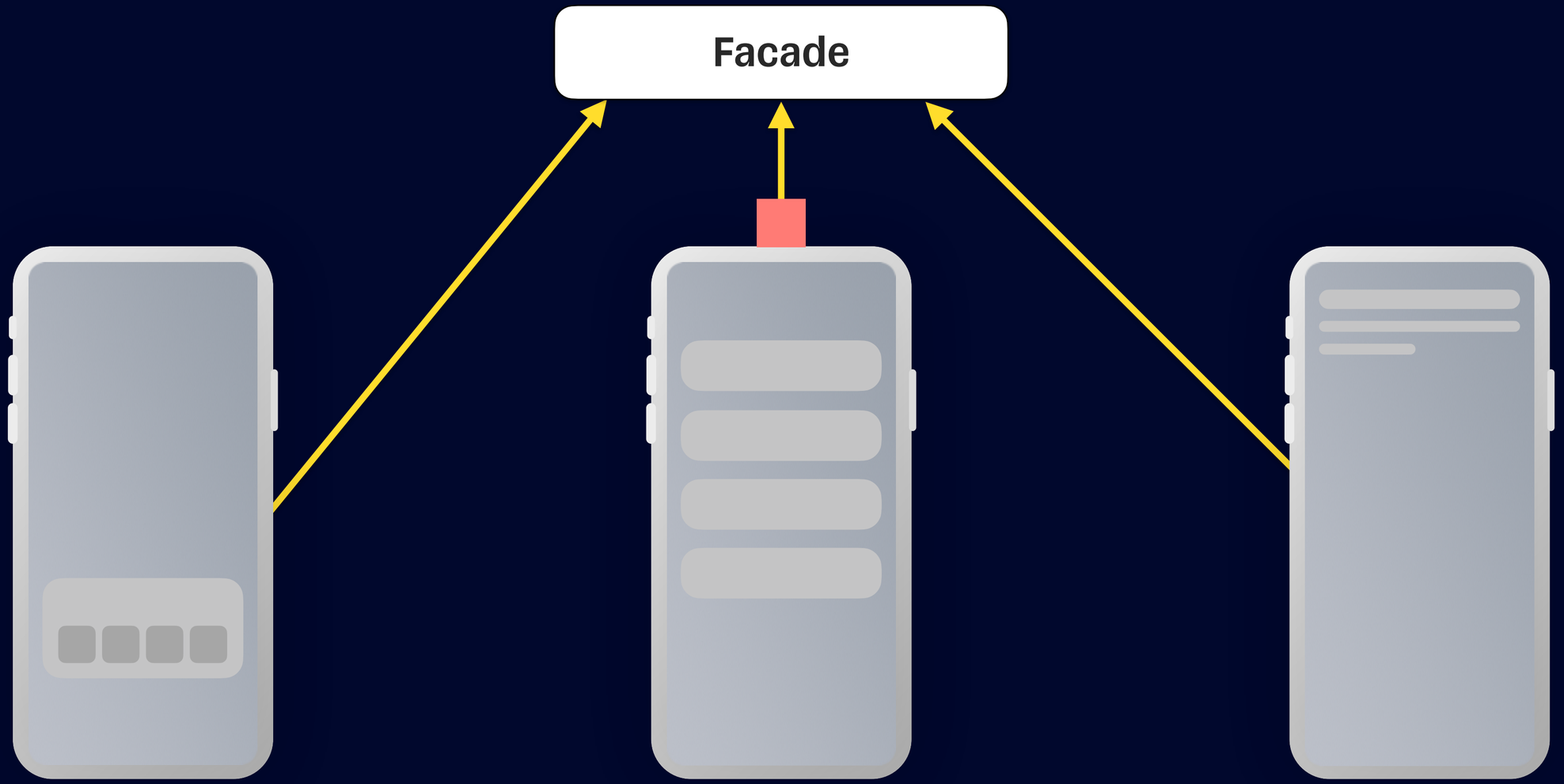
Обновление данных



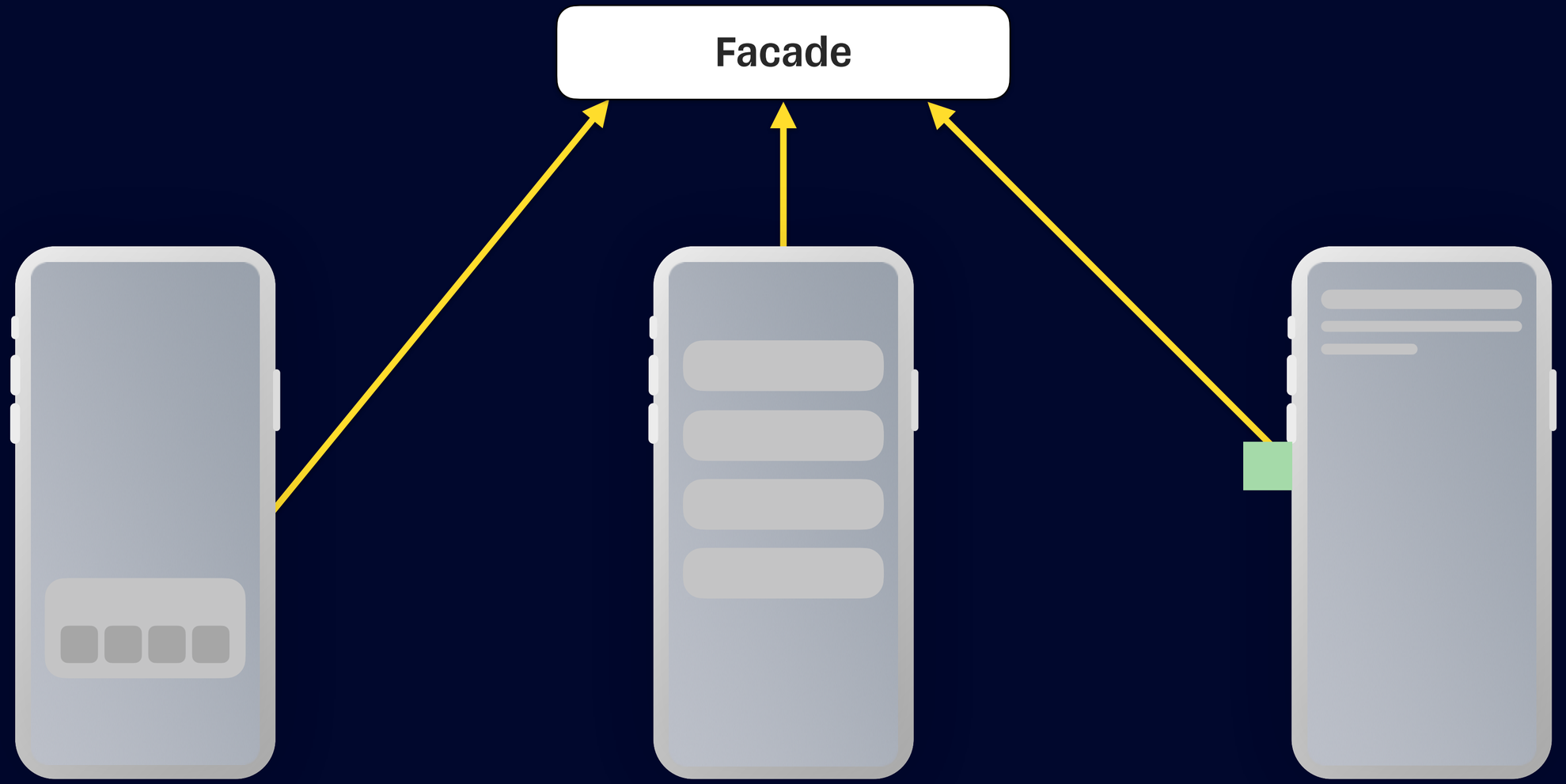
Обновление данных



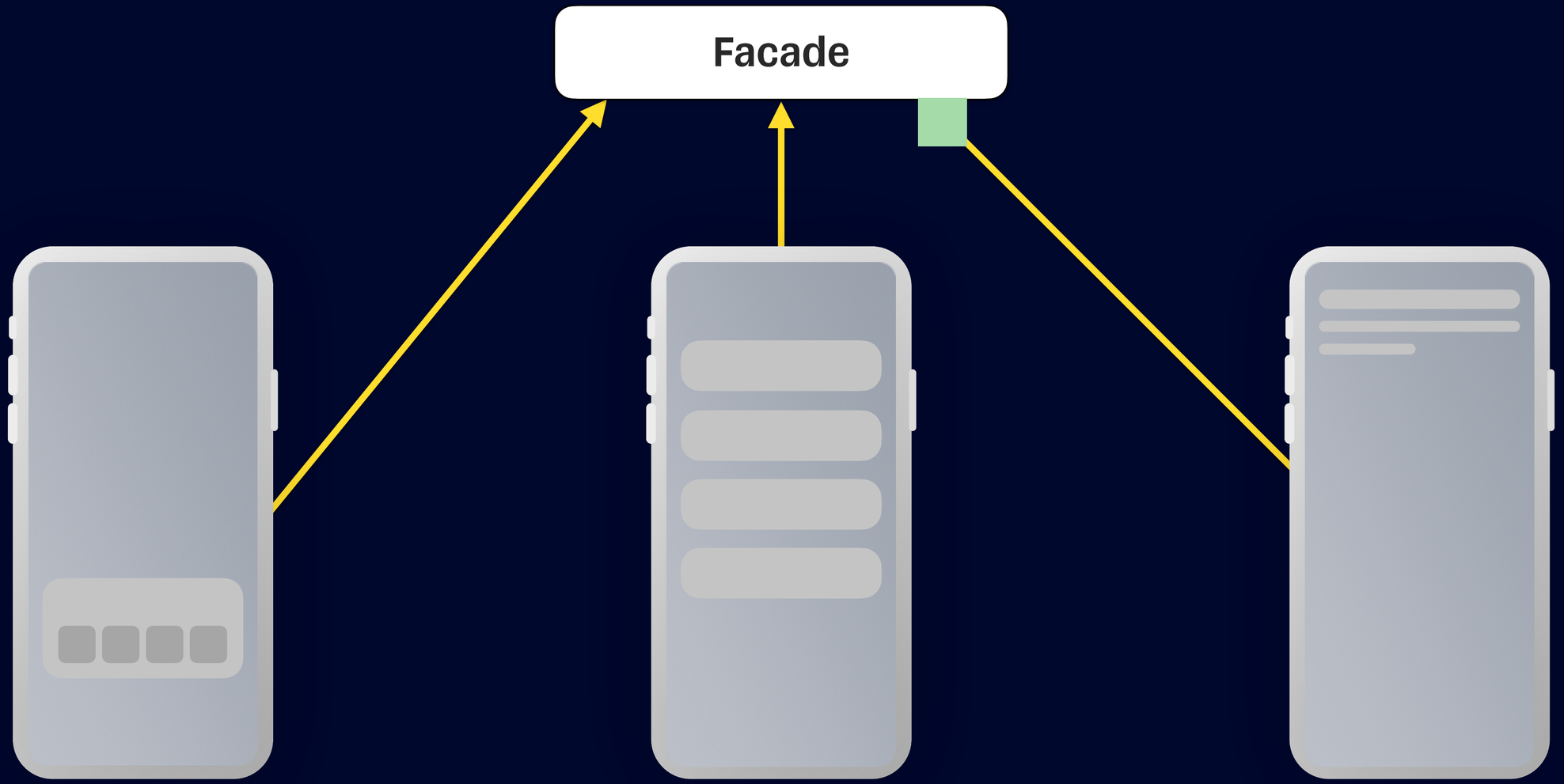
Обновление данных



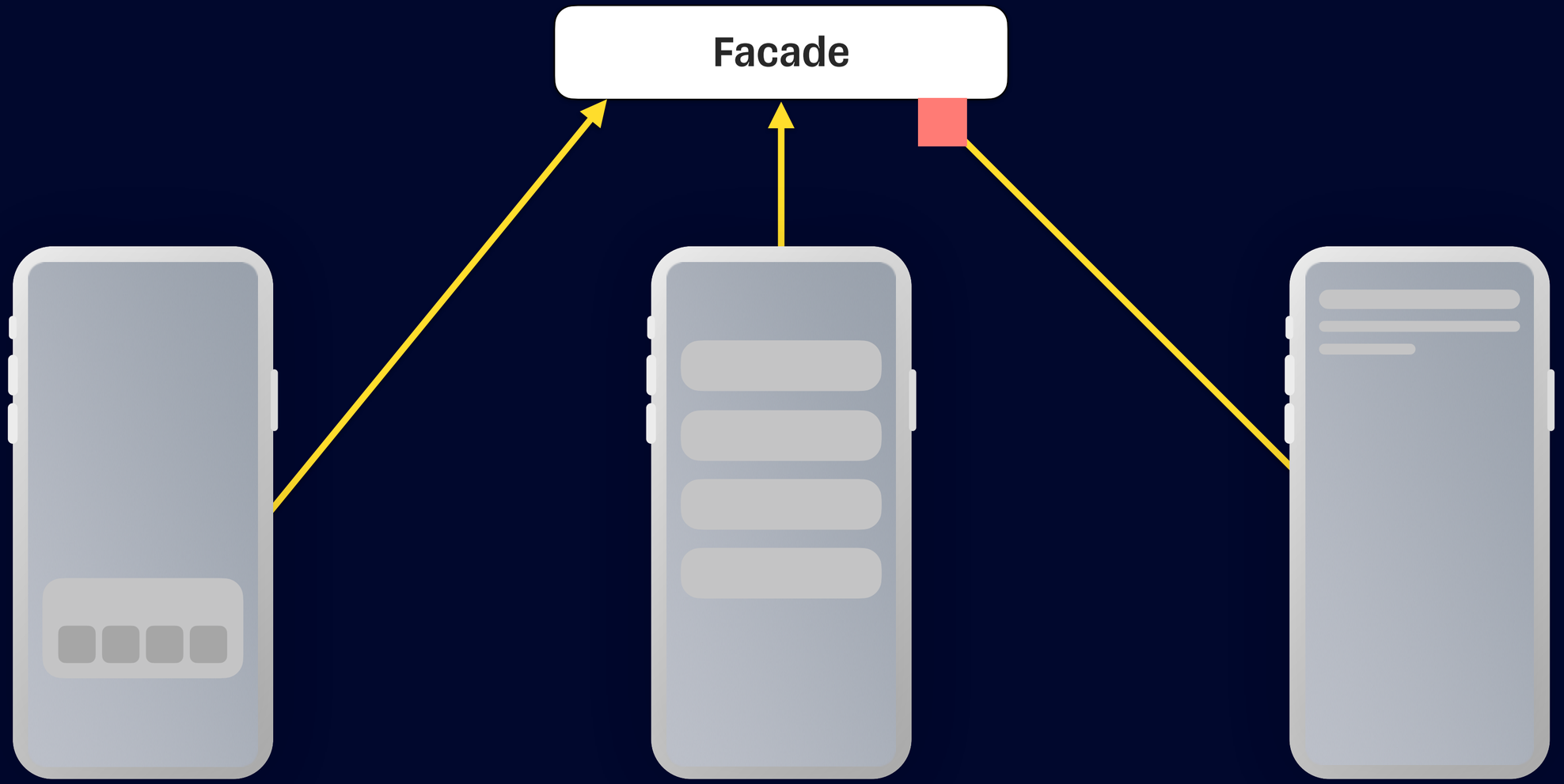
Обновление данных



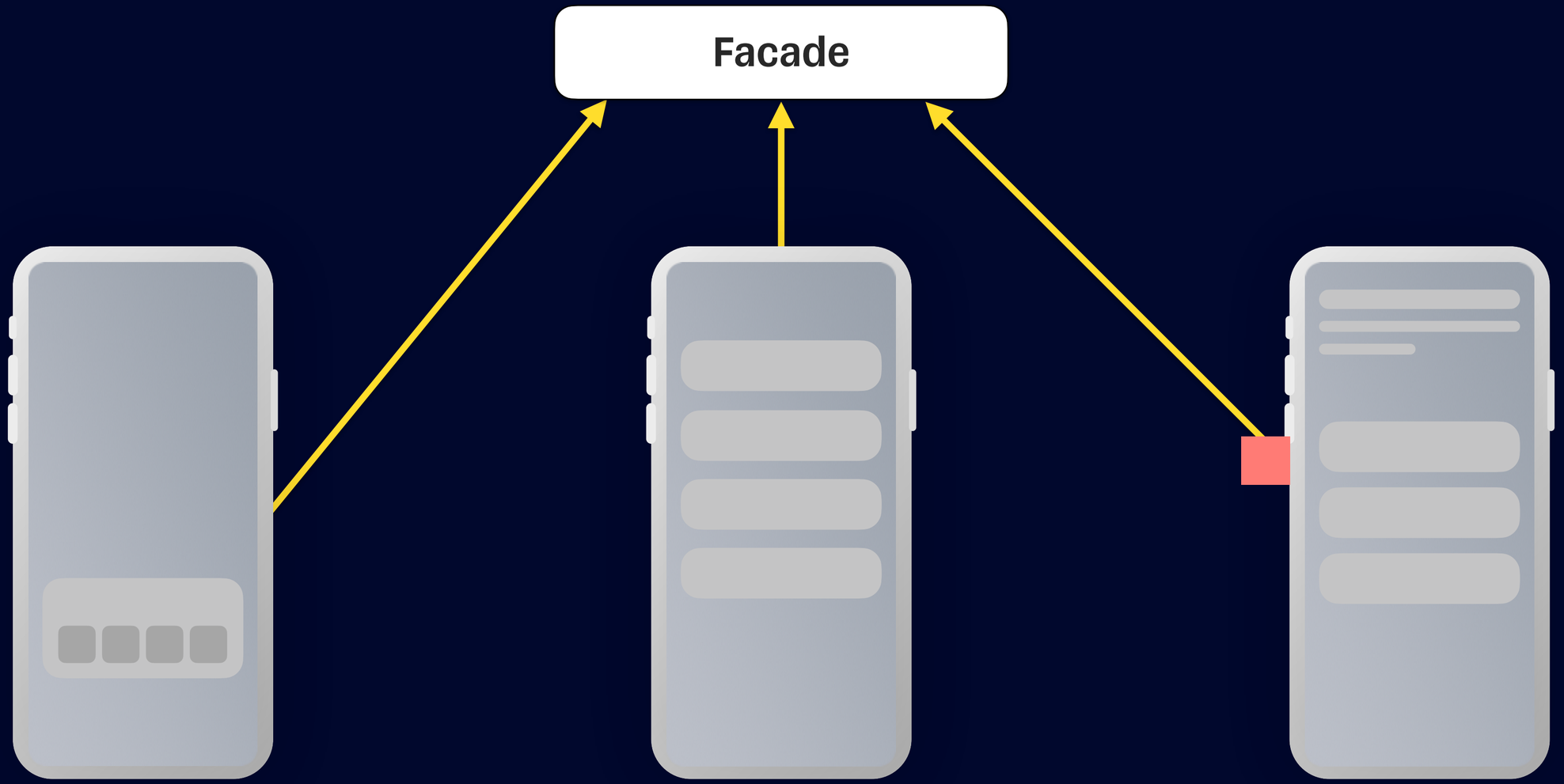
Обновление данных



Обновление данных



Обновление данных



Усугубим

Что с ним не так?

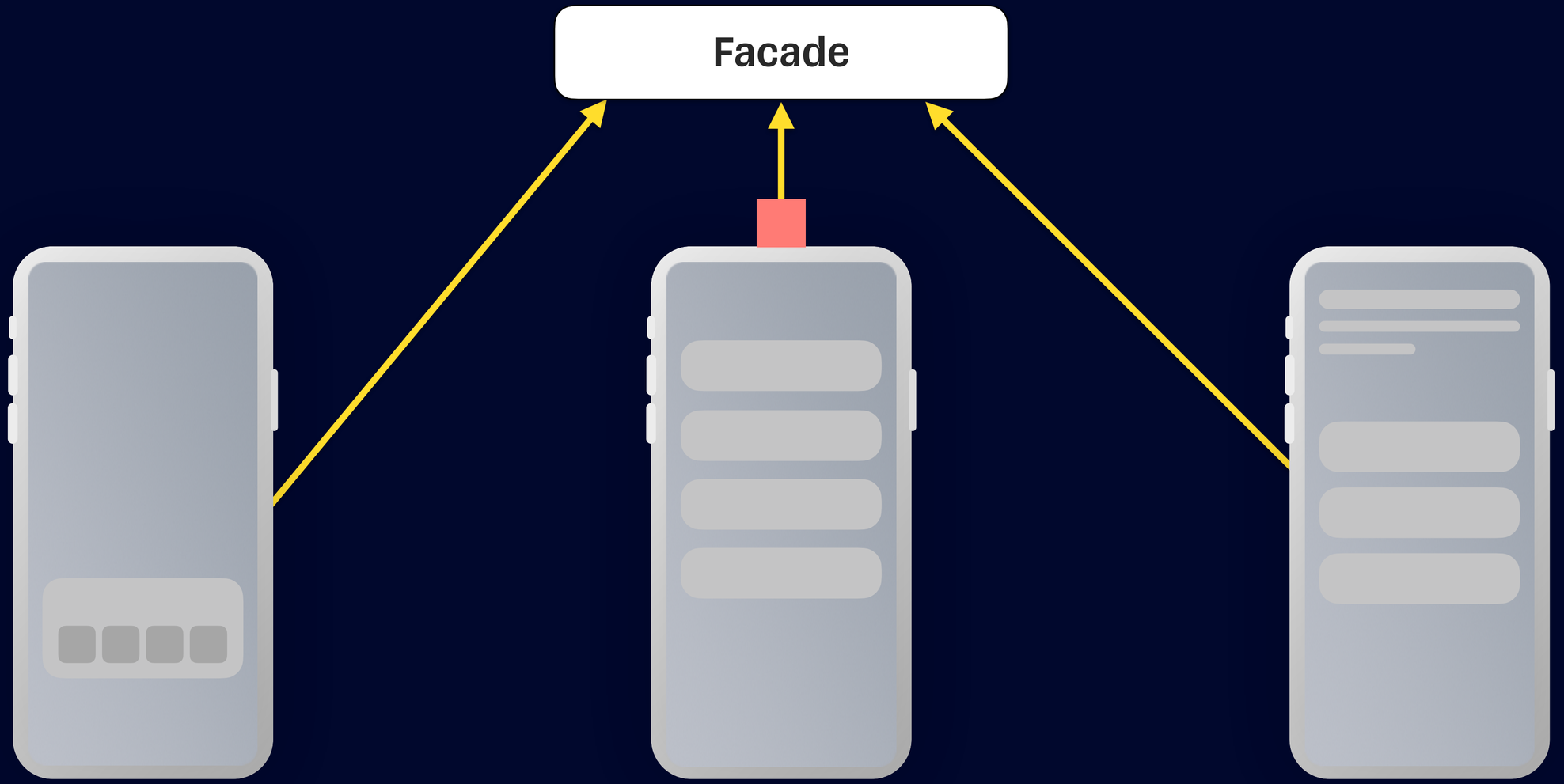
```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}
```

Усугубим

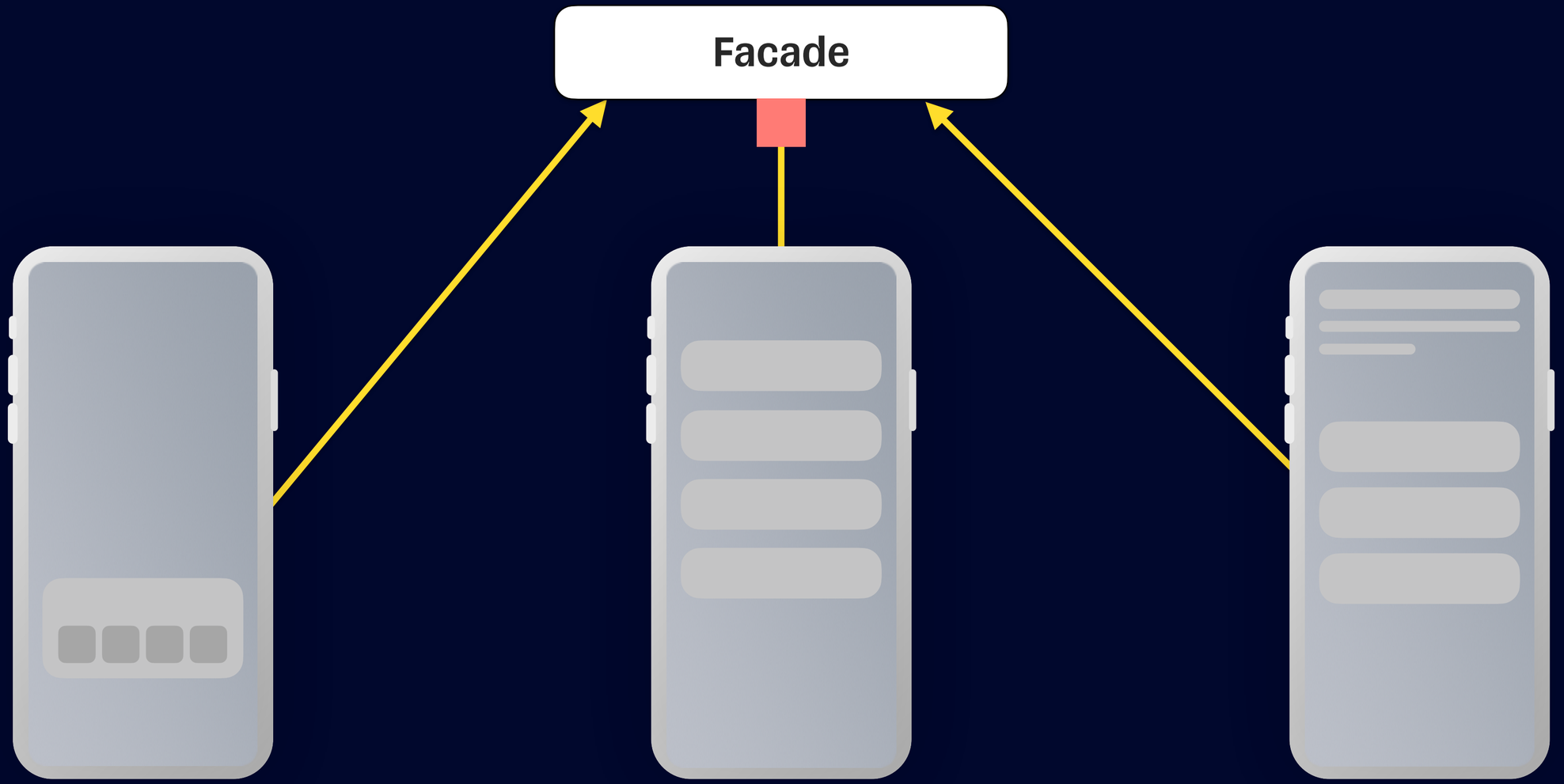
Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]

    func remove(_ object: SomeViewModel) async
}
```

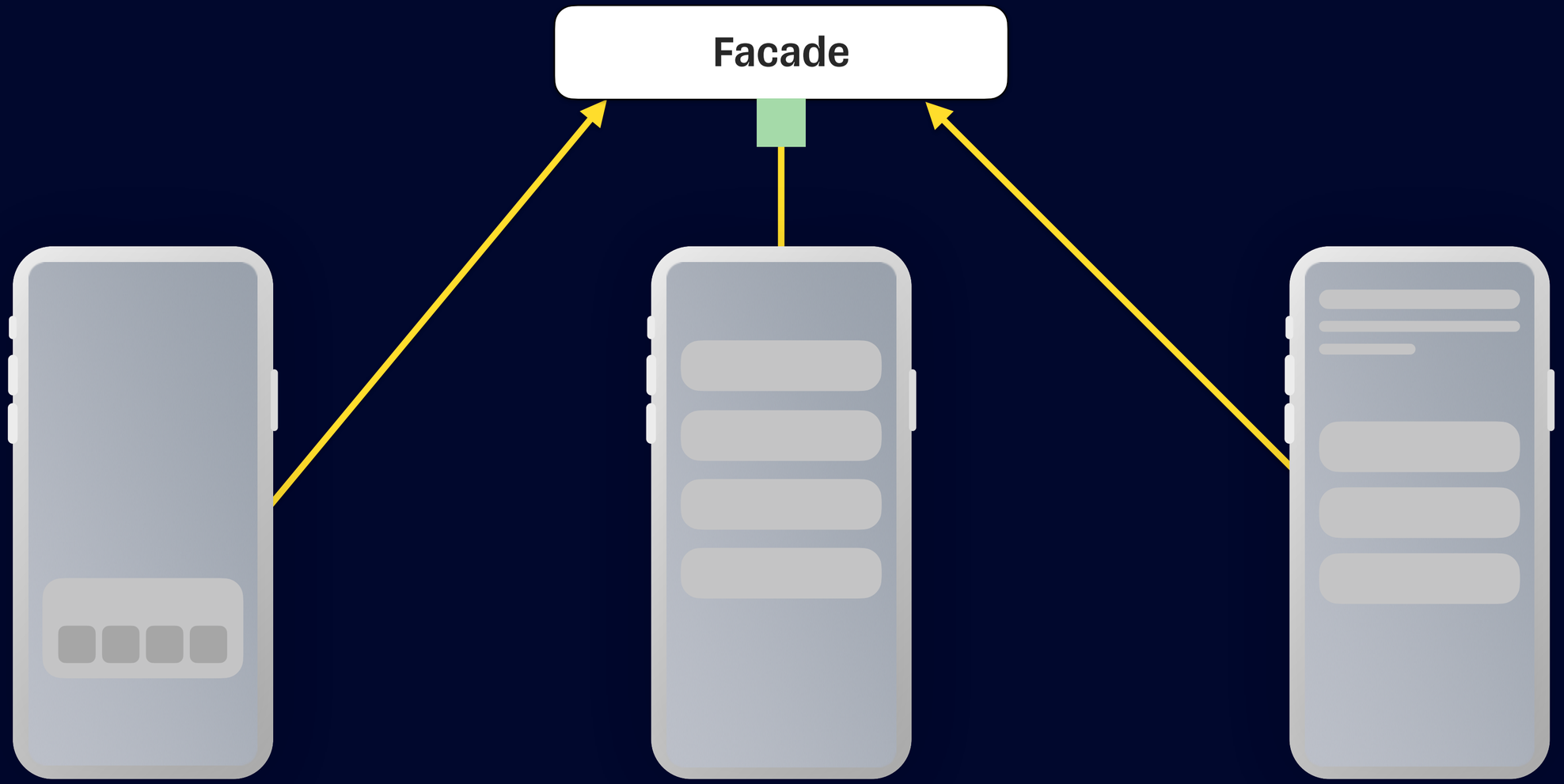
Удаление



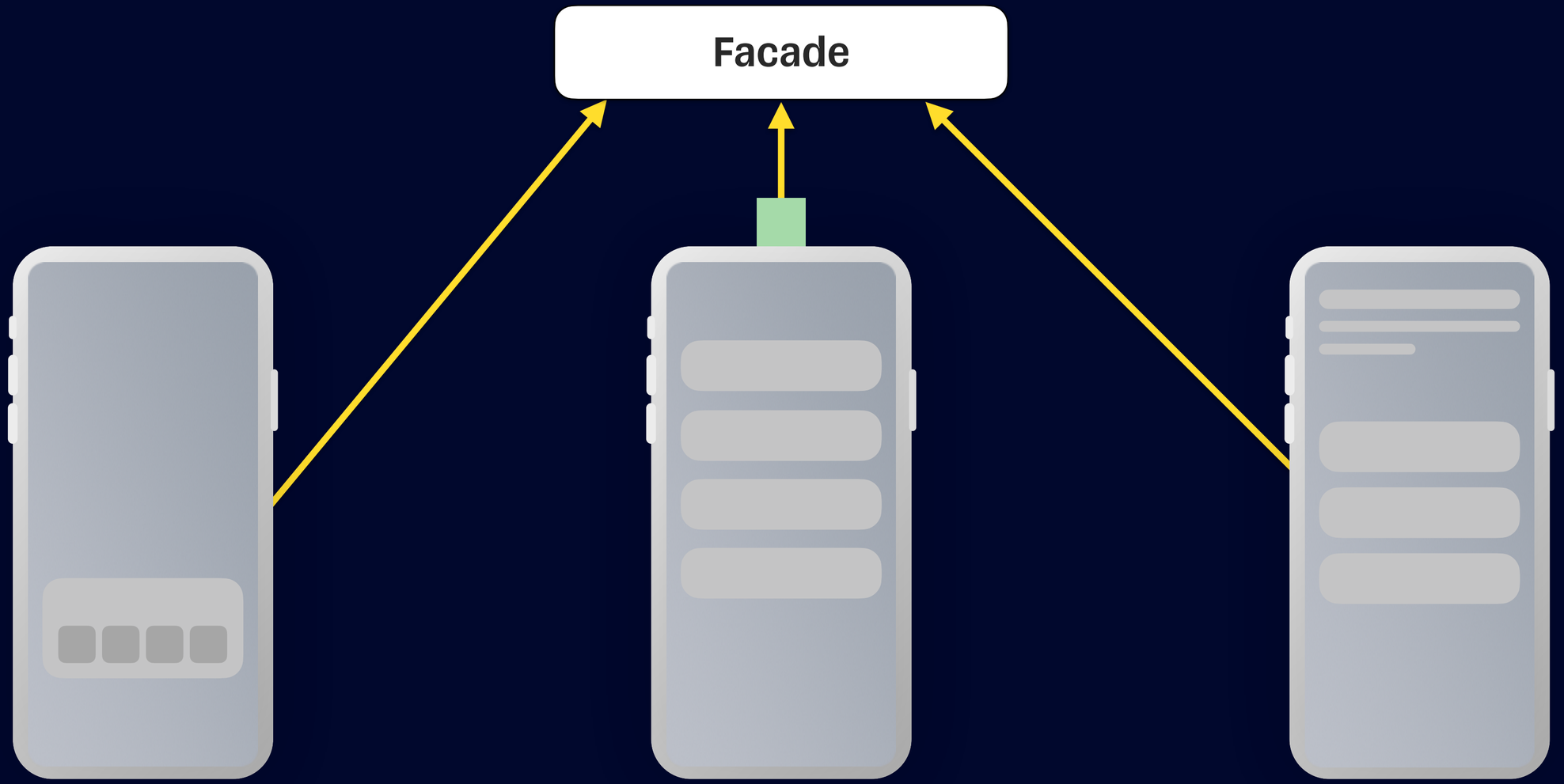
Удаление



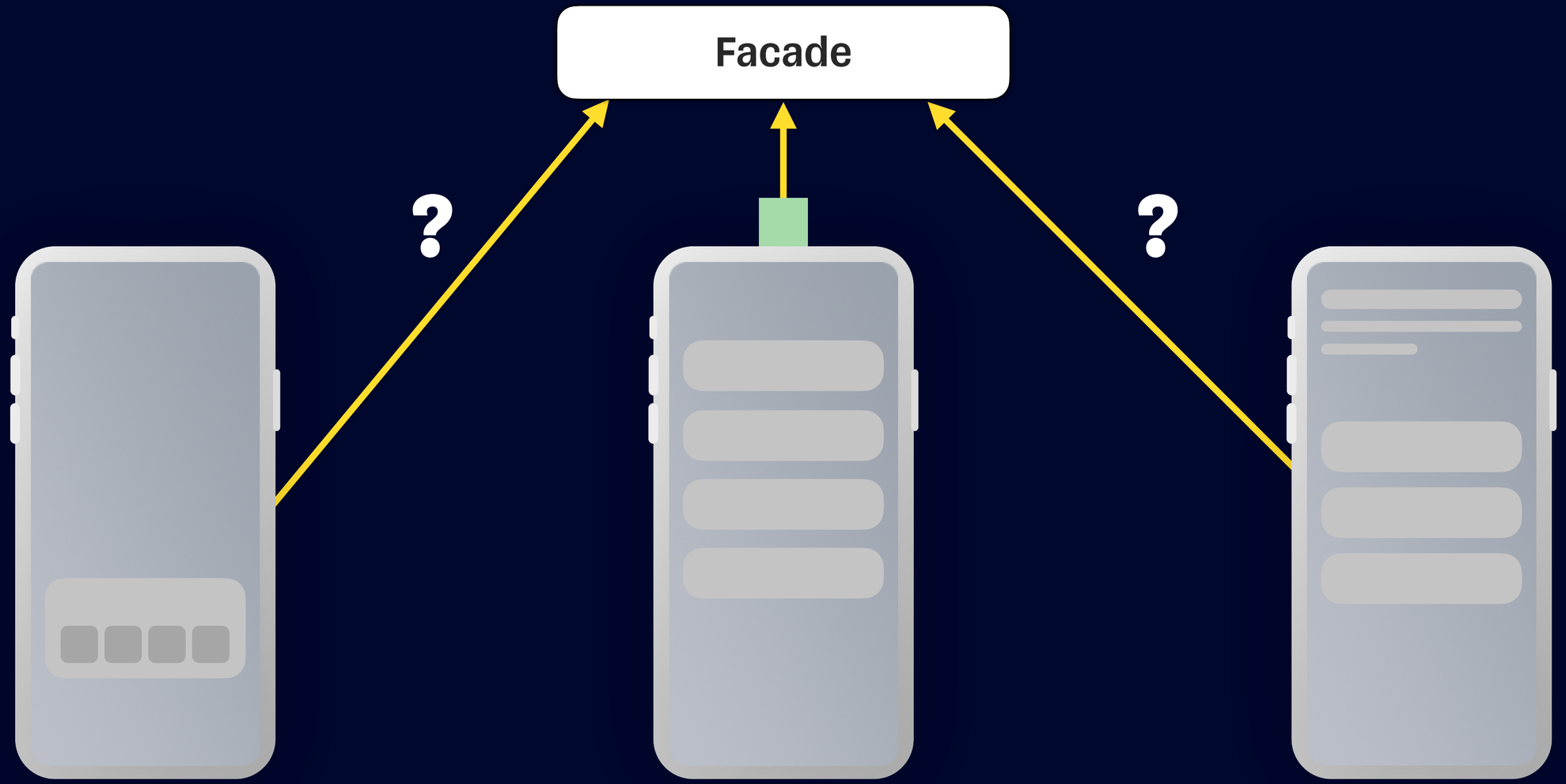
Удаление



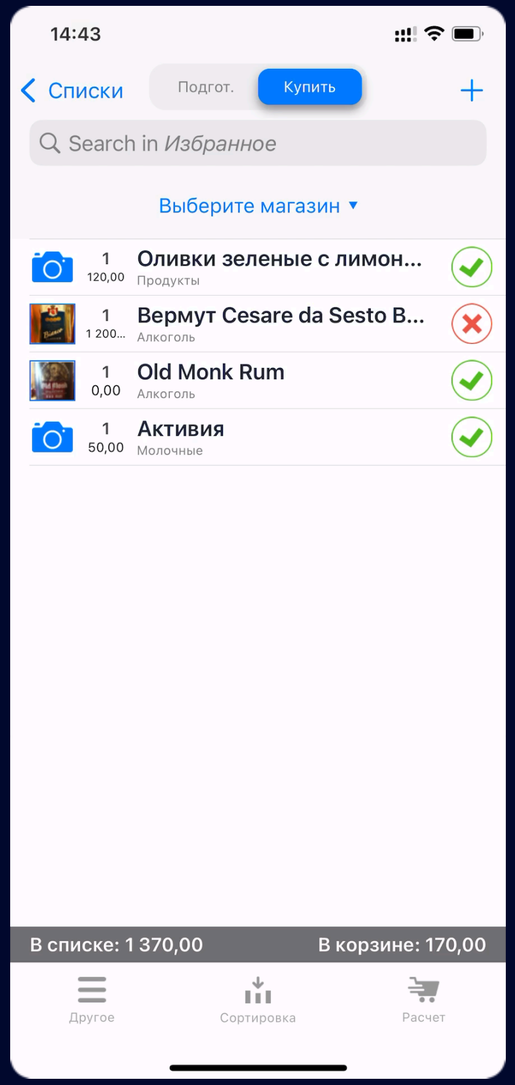
Удаление



Удаление



Удаление


14:43
Списки
Подгот.
Купить
Search in Избранное
Выберите магазин
1
120,00
Оливки зеленые с лимон...
Продукты
1
1 200...
Вермут Cesare da Sesto B...
Алкоголь
1
0,00
Old Monk Rum
Алкоголь
1
50,00
Активия
Молочные
В списке: 1 370,00
В корзине: 170,00
Другое
Сортировка
Расчет

Предложение

Что с ним не так?

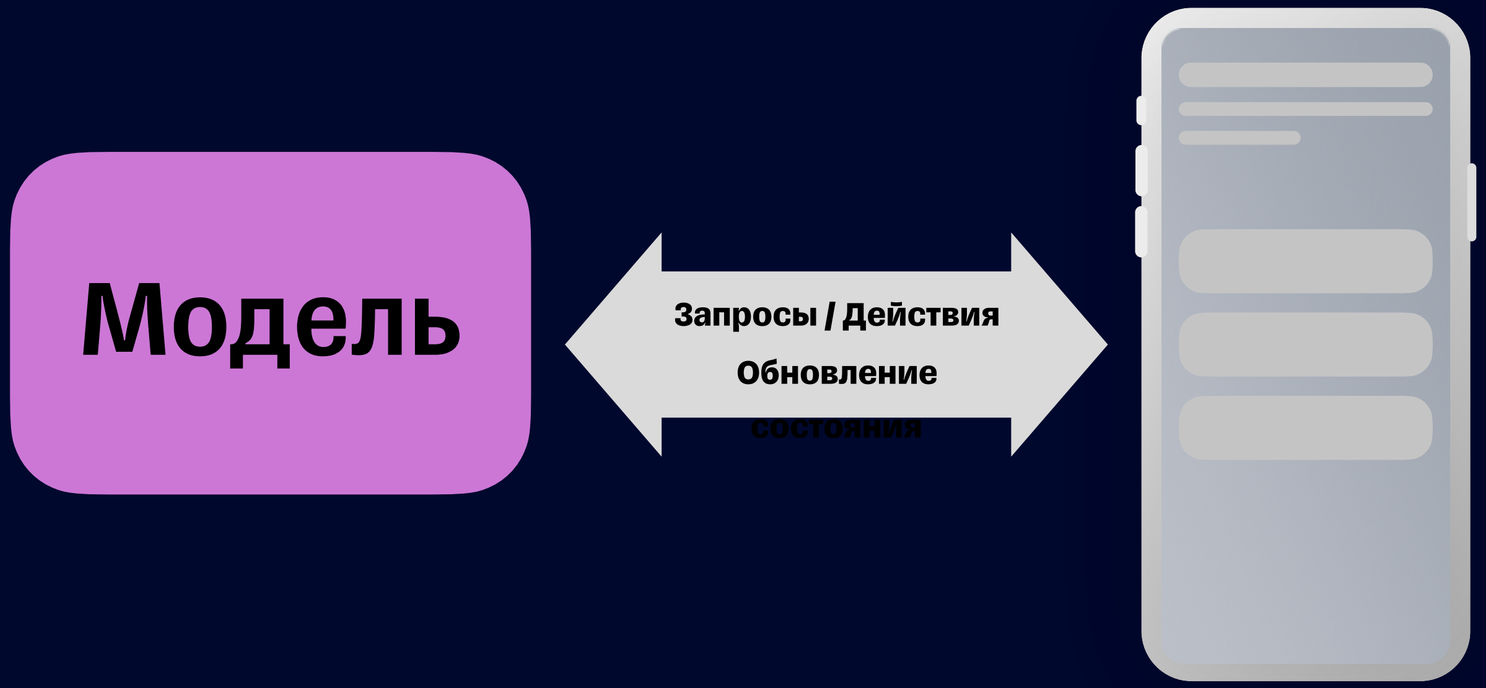
```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]
}
```

Предложение

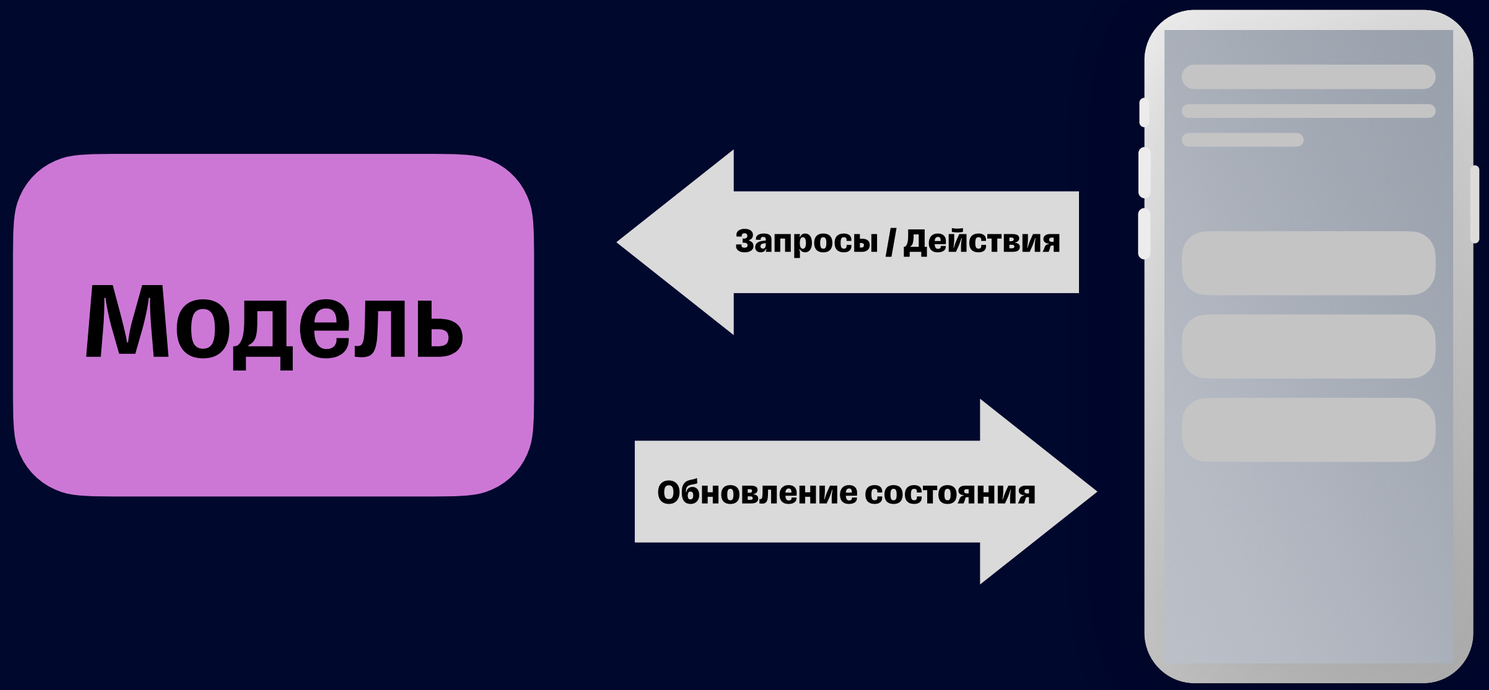
Что с ним не так?

```
protocol ISomeFacade {

    // MARK: - Methods

    func load() async throws -> [SomeViewModel]

    func remove(_ object: SomeViewModel) async -> [SomeViewModel]
}
```

Предложение



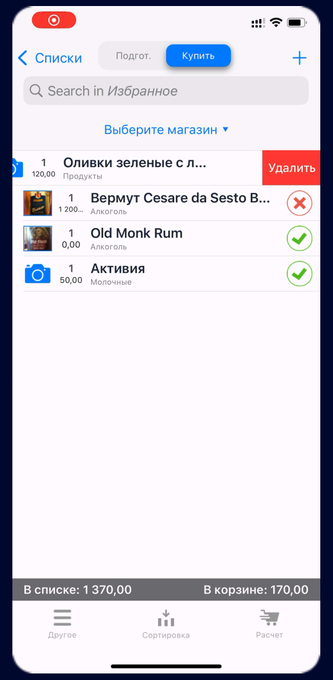
Предложение



Предложение

Модель

Запросы / Действия

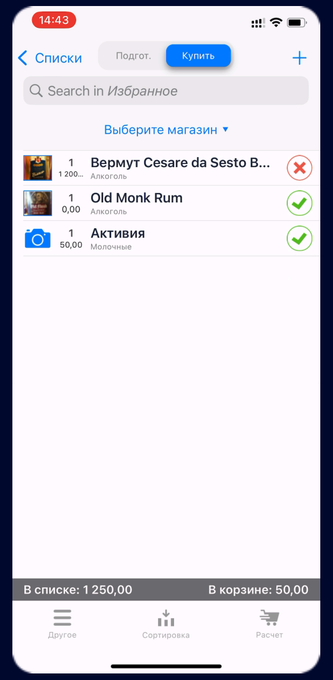


Предложение

Модель

Обновление состояния

Обновление состояния



Предложение

```
protocol ISomeFacade {

    func load() async throws -> [SomeViewModel]

    func remove(_ object: SomeViewModel) async
}
```

Предложение

```
protocol ISomeFacade {

    func load() async throws

    func remove(_ object: SomeViewModel) async
}
```

Предложение

```
protocol ISomeFacade: Actor {


    func load() async throws

    func remove(_ object: SomeViewModel) async
}
```

Предложение

```
protocol ISomeFacade: Actor {

    var items: AnyPublisher<[SomeViewModel], Never> { get }

    func load() async throws

    func remove(_ object: SomeViewModel) async
}
```

Где-то в презентере

```
func viewDidLoad() {
   Task { [weak view, someFacade] in
        let values = await someFacade.items.asyncValues
        for await values in values {
            await view?.updateItems(values)
        }
    }
}
```

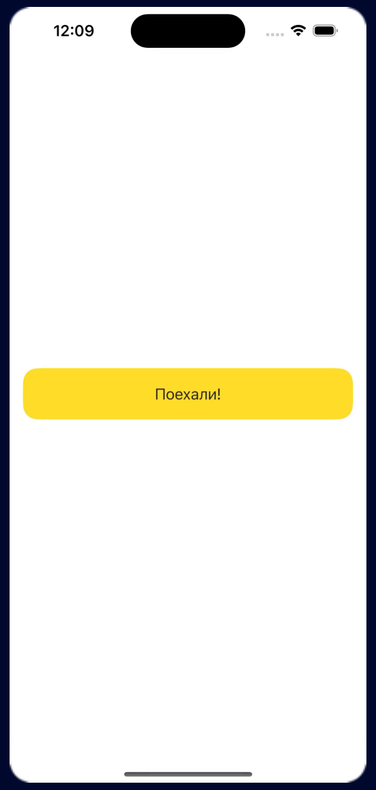
Где-то в презентере

```
func viewDidLoad() {
   Task { [weak view, someFacade] in
        let values = await someFacade.items.asyncValues
        for await values in values {
            await view?.updateItems(values)
        }
    }
}

func refreshRequested() {
    Task { @MainActor [weak view] in
        do {
            defer { view?.hideRefreshIndicator() }
            view?.showRefreshIndicator()
            try await someFacade.load()
        } catch is CancellationError {
            return
        } catch {
            router.showLoadError()
        }
   }
}
```

Где-то в презентере

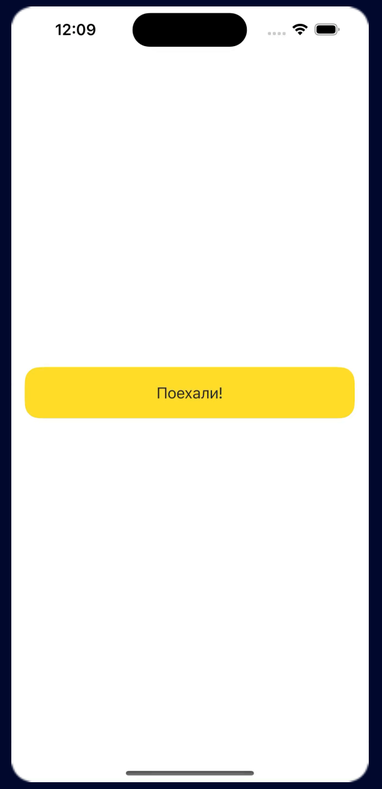
```
func removeActionTapped(object: SomeViewModel) {
    Task {
        await someFacade.remove(object)
    }
}
```

Усложнение логики



Предложение

Фасад → Поиск →



Усложнение логики

```
protocol ISearchingDataSource: Actor {

    // MARK: - Properties

    var items: AnyPublisher<[SomeViewModel], Never> { get async }

    // MARK: - Methods

    func setSearchText(_ searchText: String)
}
```

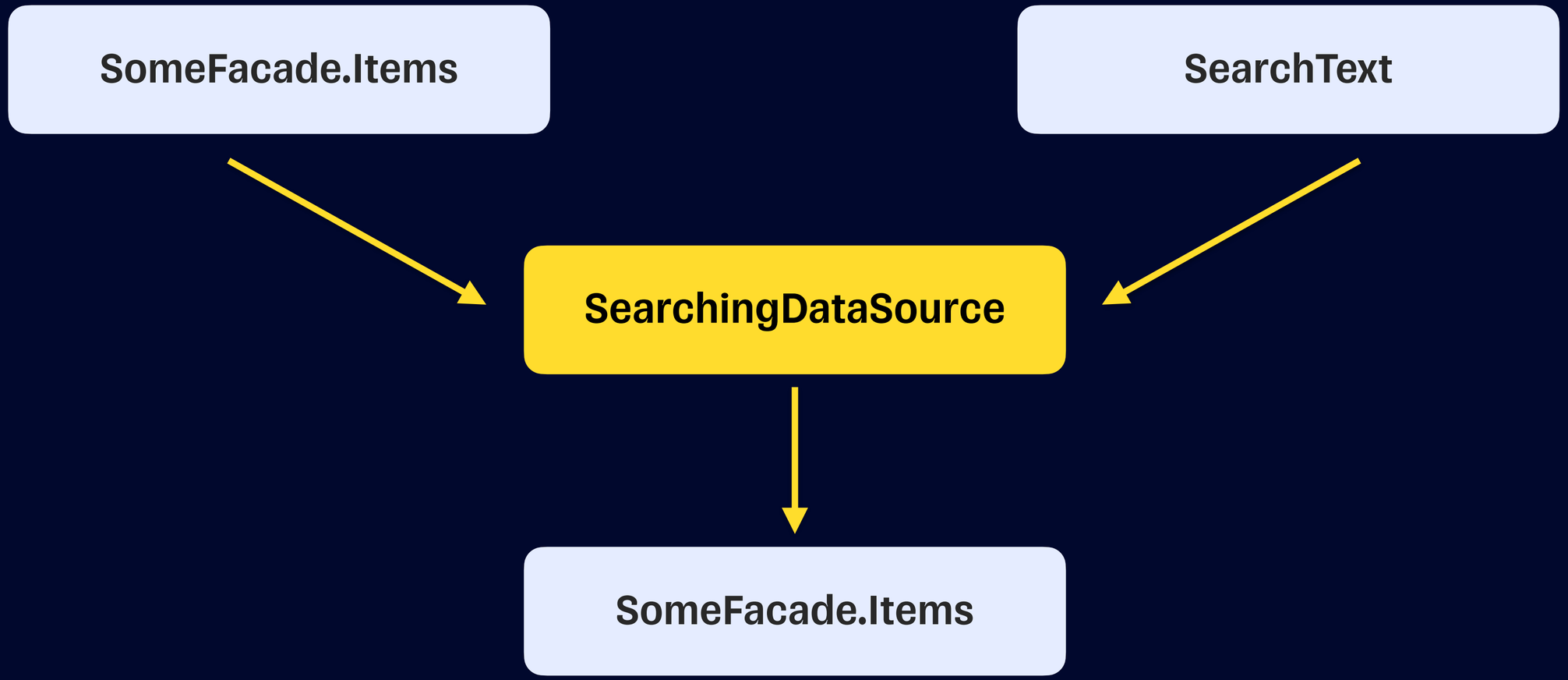
Где-то в презентере

```
func viewDidLoad() {
   Task { [weak view, someFacade] in
        let values = await someFacade.items.asyncValues
        for await values in values {
            await view?.updateItems(values)
        }
    }
}
```

Где-то в презентере

```
func viewDidLoad() {
   Task { [weak view, searchingDataSource] in
        let values = await searchingDataSource.items.asyncValues
        for await values in values {
            await view?.updateItems(values)
        }
    }
}
```

Усложнение логики



Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

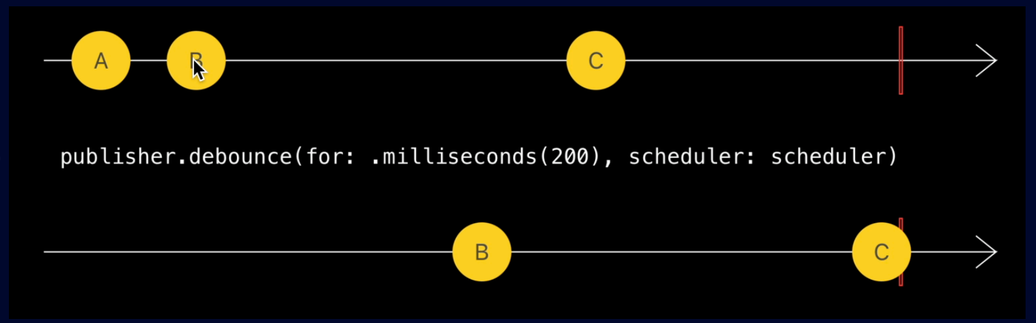
Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

CombineMarbles



Усложнение логики

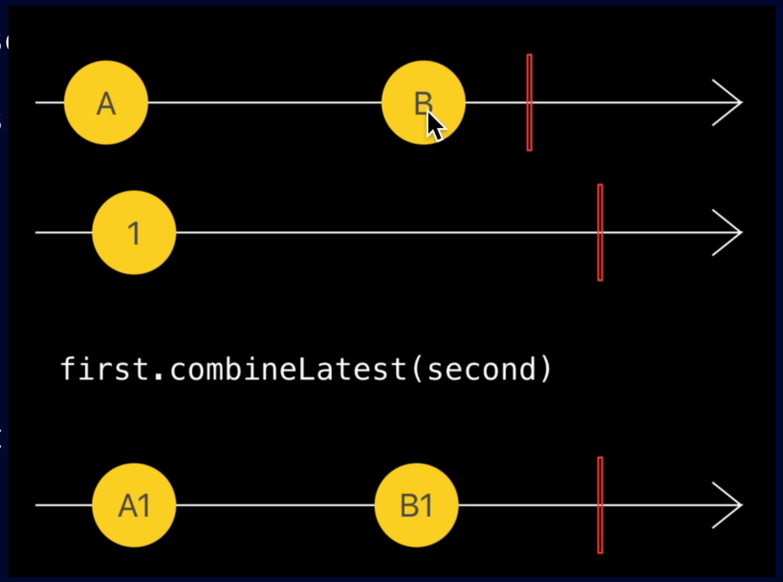
```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, s
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText

    private var searchText = CurrentValueSubject<String, Never>("")
}
```



Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Усложнение логики

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    func setSearchText(_ searchText: String) { self.searchText.value = searchText }

    private var searchText = CurrentValueSubject<String, Never>("")
}
```

Где-то в презентере

```
func searchFieldTextDidChange(_ text: String) async {
    await searchingDataSource.setSearchText(text)
}
```

Кто сказал «UDF»?

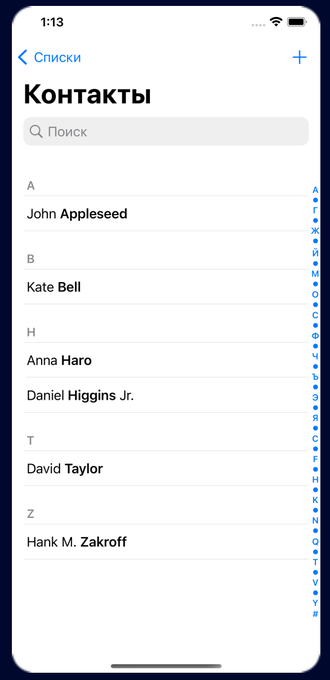
1. The Composable Architecture
2. Highway
3. У всех есть неисправимый баг — их писали не мы, сделаем своё :)

Инструменты

Совместимость — наше всё!

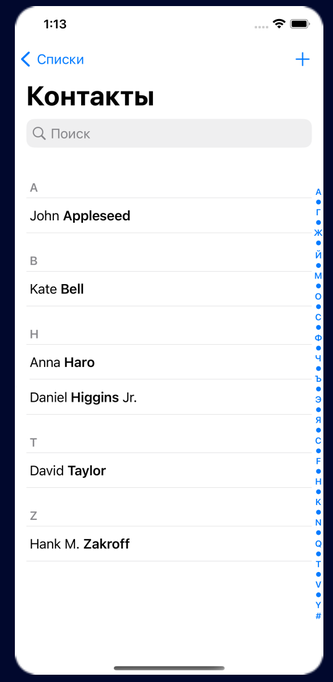
Предложение

Состояние



Предложение

Состояние



Combine и SwiftConcurrency

```
for await value in dataSource.values    'values' is only available in iOS 15.0 or newer
```

Combine и SwiftConcurrency

```
for await value in dataSource.asyncValues
```

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

Combine и SwiftConcurrency

```
public protocol Publisher<Output, Failure> {

   associatedtype Output

   associatedtype Failure : Error

   func receive<S>(subscriber: S)
        where S : Subscriber, Self.Failure == S.Failure, Self.Output == S.Input
}
```

Combine и SwiftConcurrency

```
public protocol Subscriber<Input, Failure> : CustomCombineIdentifierConvertible {

    associatedtype Failure : Error

    func receive(subscription: Subscription)

    func receive(_ input: Self.Input) -> Subscribers.Demand

    func receive(completion: Subscribers.Completion<Self.Failure>)
}
```

Combine и SwiftConcurrency

```
@frozen public struct Demand : Equatable, Comparable, ... {

    /// A request for as many values as the publisher can produce.
    public static let unlimited: Demand

    /// This is equivalent to `Demand.max(0)`.
    public static let none: Demand

   @inlinable public static func max(_ value: Int) -> Demand
}
```

Combine и SwiftConcurrency

```
searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues
```

Demand.max(1)

Combine и SwiftConcurrency

```
searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues
```

Demand.max(1)

Combine и SwiftConcurrency

```
searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues
```

Demand.max(1)

Combine и SwiftConcurrency

```
searchText
    "abcdef"
        .prepend(start)
        .removeDuplicates()
        .asyncValues
```

Combine и SwiftConcurrency

```
searchText
    .prepend(start)
"abcdef"    .removeDuplicates()
    .asyncValues
```

Combine и SwiftConcurrency

```
for await value in searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues {
    await doSomething(with: value)
}
```

"abcdef"

Combine и SwiftConcurrency

```
for await value in searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues {
    await doSomething(with: value)
}
```

"abcdef"

Combine и SwiftConcurrency

```
for await value in searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues {
    await doSomething(with: value)
}
```

Demand.max(1)

Combine и SwiftConcurrency

```
for await value in searchText
    .prepend(start)
    .removeDuplicates()
    .asyncValues {
    await doSomething(with: value)
}
```

Demand.max(1)

Combine и SwiftConcurrency

AsyncSequence & AsyncStream

Combine и SwiftConcurrency

```swift
let counter = Counter()
let stream = AsyncStream(unfolding: { await counter.value() })

let t1 = Task {
    for await i in stream.prefix(10) {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10) {
        print("consumer 2: \(i)")
    }
}

Task {
    await (t1.value, t2.value)

    PlaygroundPage.current.finishExecution()
}
```

```
consumer 1: 0
consumer 2: 1
consumer 1: 2
consumer 2: 3
consumer 1: 4
consumer 2: 5
consumer 1: 6
consumer 2: 7
consumer 1: 8
consumer 2: 9
consumer 1: 10
```

```swift
let stream = (0 ..< 100).publisher

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 2: \(i)")
    }
}

Task {
    await (t1.value, t2.value)

    PlaygroundPage.current.finishExecution()
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 2: 3
consumer 1: 3
consumer 2: 4
consumer 1: 4
```

Combine и SwiftConcurrency

```swift
let counter = Counter()
let stream = AsyncStream(unfolding: { await counter.value() })

let t1 = Task {
    for await i in stream.prefix(10) {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10) {
        print("consumer 2: \(i)")
    }
}

Task {
    await (t1.value, t2.value)

    PlaygroundPage.current.finishExecution()
}
```

```
consumer 1: 0
consumer 2: 1
consumer 1: 2
consumer 2: 3
consumer 1: 4
consumer 2: 5
consumer 1: 6
consumer 2: 7
consumer 1: 8
consumer 2: 9
consumer 1: 10
```

```swift
let stream = (0 ..< 100).publisher

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 2: \(i)")
    }
}

Task {
    await (t1.value, t2.value)

    PlaygroundPage.current.finishExecution()
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 2: 3
consumer 1: 3
consumer 2: 4
consumer 1: 4
```

Combine и SwiftConcurrency

```swift
let stream = PassthroughSubject<Int, Never>()

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        try! await Task.sleep(nanoseconds: 100_000)
        print("consumer 2: \(i)")
    }
}

Task {
    for x in 0 ..< 1000 {
        stream.send(x)
    }
}
```

```
consumer 1: 0
consumer 1: 1
consumer 1: 2
consumer 1: 3
consumer 1: 4
consumer 1: 5
consumer 1: 6
consumer 1: 7
consumer 1: 8
consumer 1: 9
consumer 2: 0
consumer 2: 15
consumer 2: 17
consumer 2: 25
consumer 2: 31
consumer 2: 55
consumer 2: 85
consumer 2: 103
consumer 2: 131
consumer 2: 156
```

```swift
let stream = TCAsyncChannel<Int, Never>()

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        try! await Task.sleep(nanoseconds: 100_000)
        print("consumer 2: \(i)")
    }
}

Task {
    for x in 0 ..< 1000 {
        try await stream.send(x)
    }
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 1: 3
consumer 2: 3
consumer 1: 4
consumer 2: 4
consumer 1: 5
consumer 2: 5
consumer 1: 6
consumer 2: 6
consumer 1: 7
consumer 2: 7
consumer 1: 8
consumer 2: 8
consumer 1: 9
consumer 2: 9
```

Combine и SwiftConcurrency

```swift
let stream = TCAsyncChannel<Int, Never>()

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        try! await Task.sleep(nanoseconds: 100_000)
        print("consumer 2: \(i)")
    }
}

Task {
    for x in 0 ..< 1000 {
        try await stream.send(x)
    }
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 1: 3
consumer 2: 3
consumer 1: 4
consumer 2: 4
consumer 1: 5
consumer 2: 5
consumer 1: 6
consumer 2: 6
consumer 1: 7
consumer 2: 7
consumer 1: 8
consumer 2: 8
consumer 1: 9
consumer 2: 9
```

Combine и SwiftConcurrency

```swift
let stream = TCAsyncChannel<Int, Never>()

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        try! await Task.sleep(nanoseconds: 100_000)
        print("consumer 2: \(i)")
    }
}

Task {
    for x in 0 ..< 1000 {
        try await stream.send(x)
    }
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 1: 3
consumer 2: 3
consumer 1: 4
consumer 2: 4
consumer 1: 5
consumer 2: 5
consumer 1: 6
consumer 2: 6
consumer 1: 7
consumer 2: 7
consumer 1: 8
consumer 2: 8
consumer 1: 9
consumer 2: 9
```

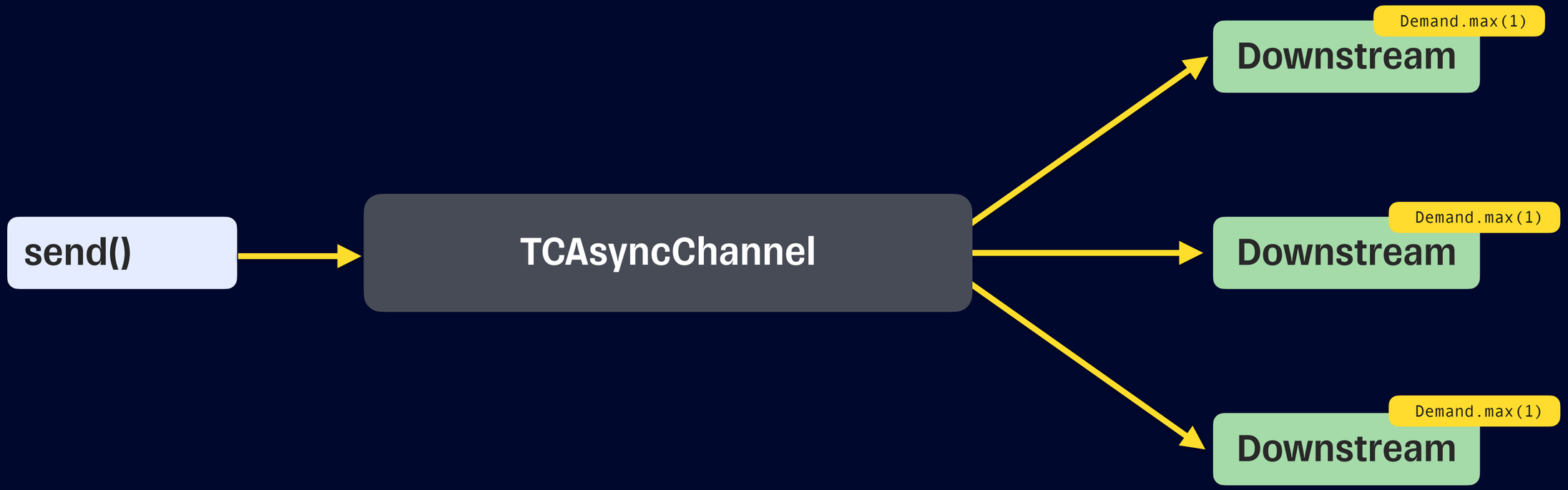
Combine и SwiftConcurrency

```swift
let stream = TCAsyncChannel<Int, Never>()

let t1 = Task {
    for await i in stream.prefix(10).values {
        print("consumer 1: \(i)")
    }
}

let t2 = Task {
    for await i in stream.prefix(10).values {
        try! await Task.sleep(nanoseconds: 100_000)
        print("consumer 2: \(i)")
    }
}

Task {
    for x in 0 ..< 1000 {
        try await stream.send(x)
    }
}
```

```
consumer 1: 0
consumer 2: 0
consumer 1: 1
consumer 2: 1
consumer 1: 2
consumer 2: 2
consumer 1: 3
consumer 2: 3
consumer 1: 4
consumer 2: 4
consumer 1: 5
consumer 2: 5
consumer 1: 6
consumer 2: 6
consumer 1: 7
consumer 2: 7
consumer 1: 8
consumer 2: 8
consumer 1: 9
consumer 2: 9
```

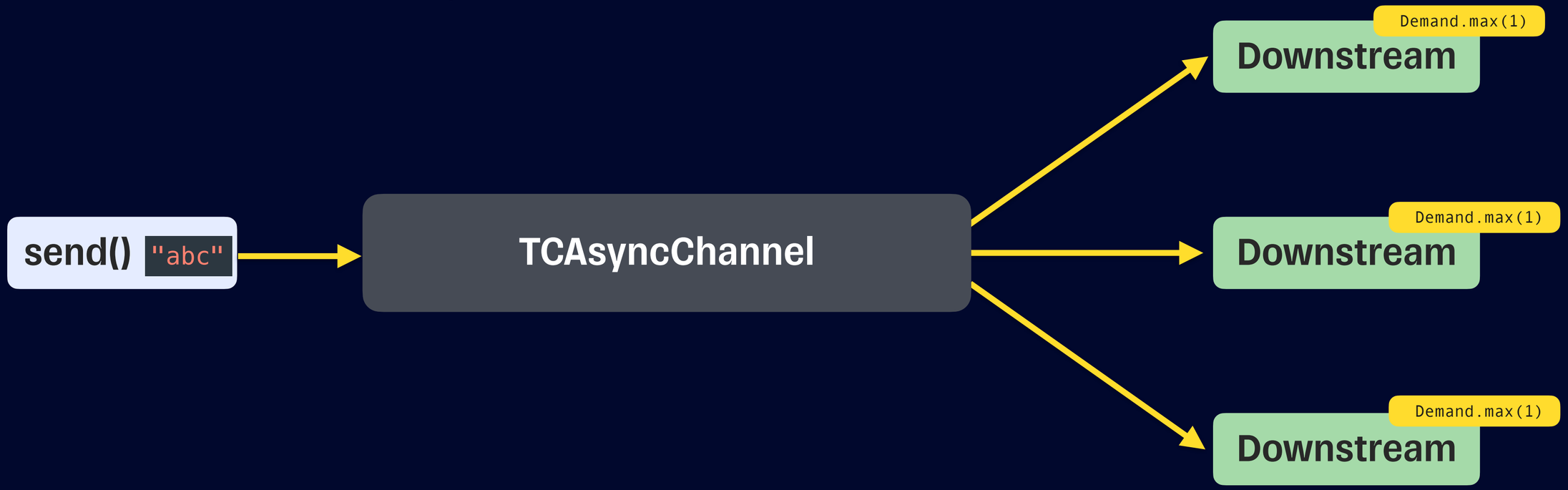
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



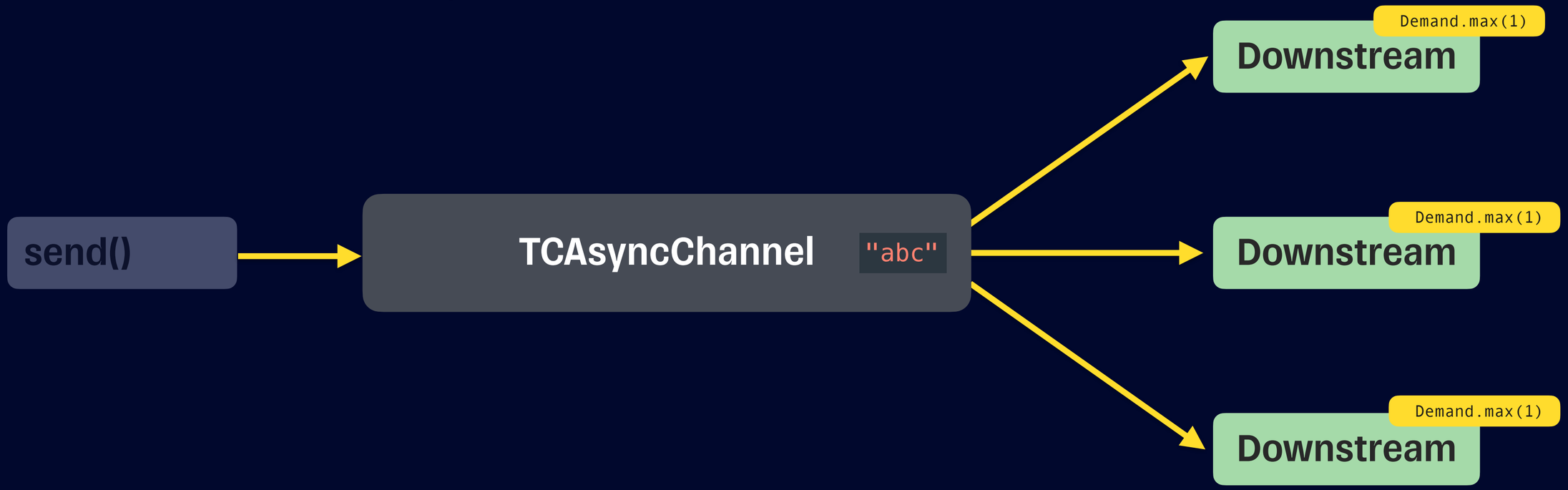
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



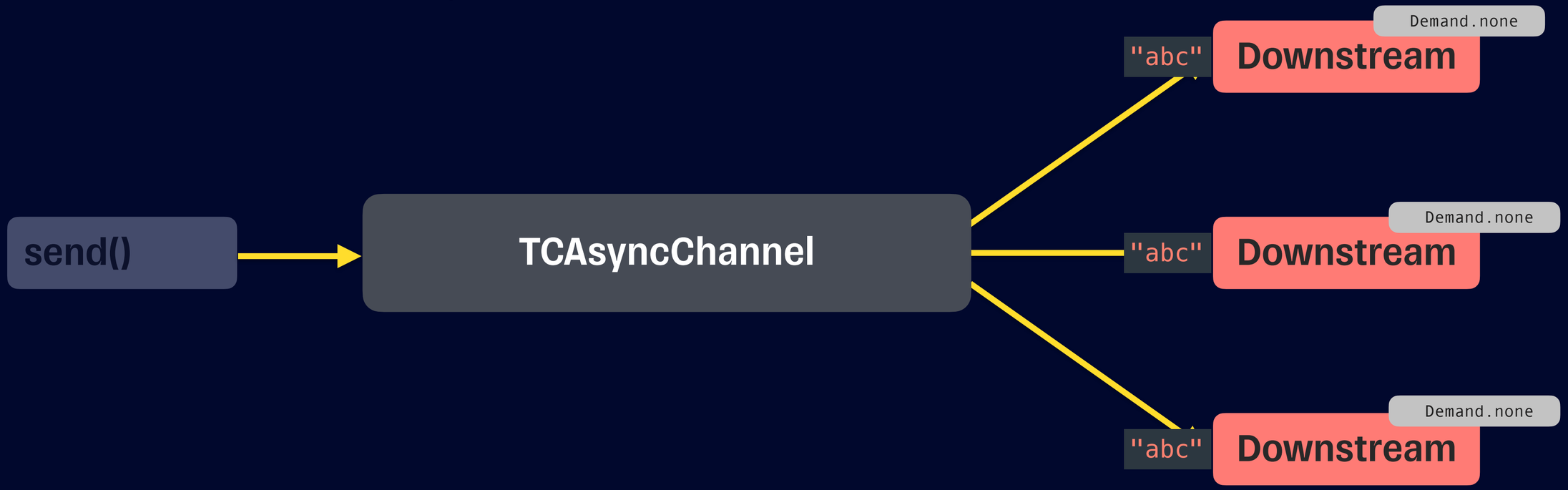
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



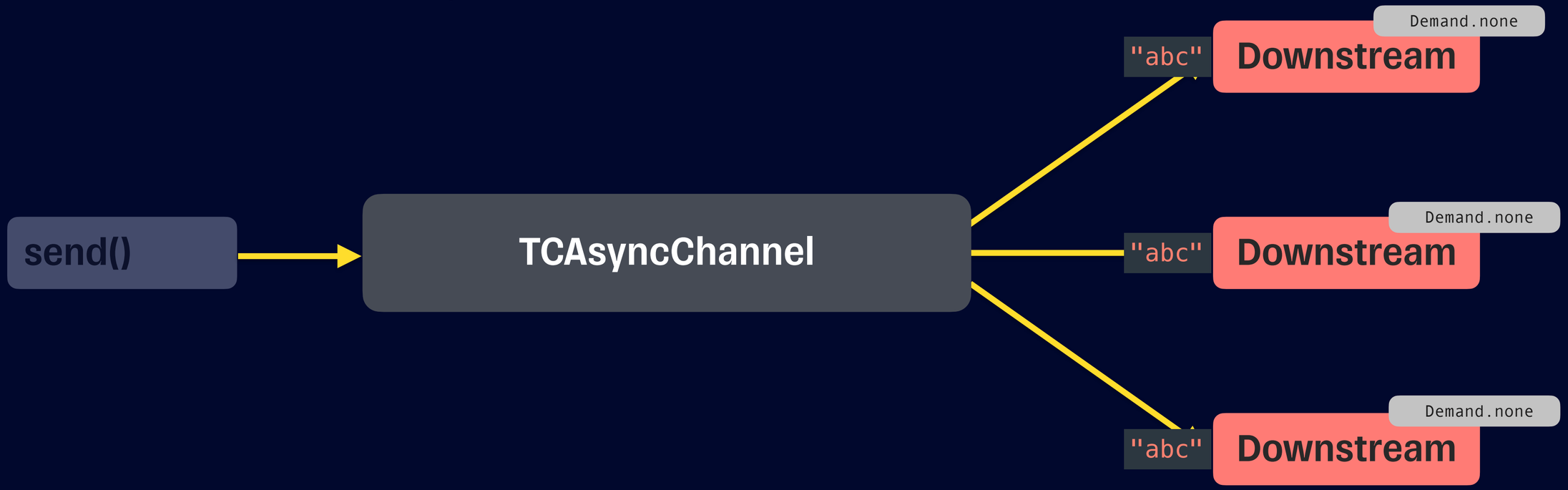
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



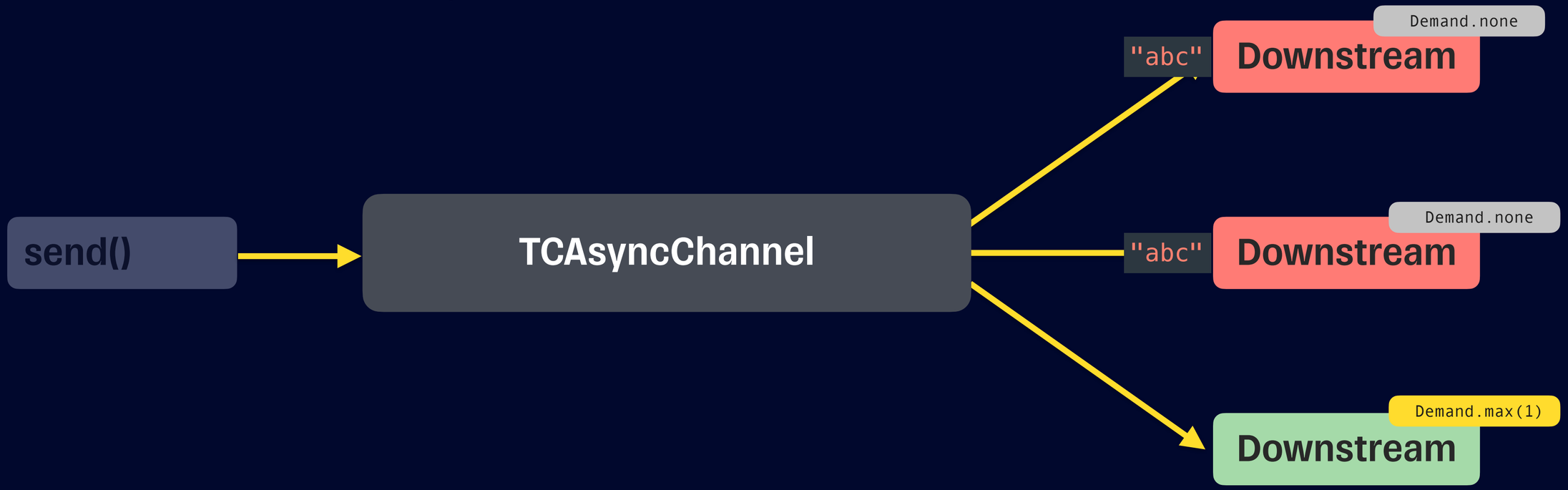
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



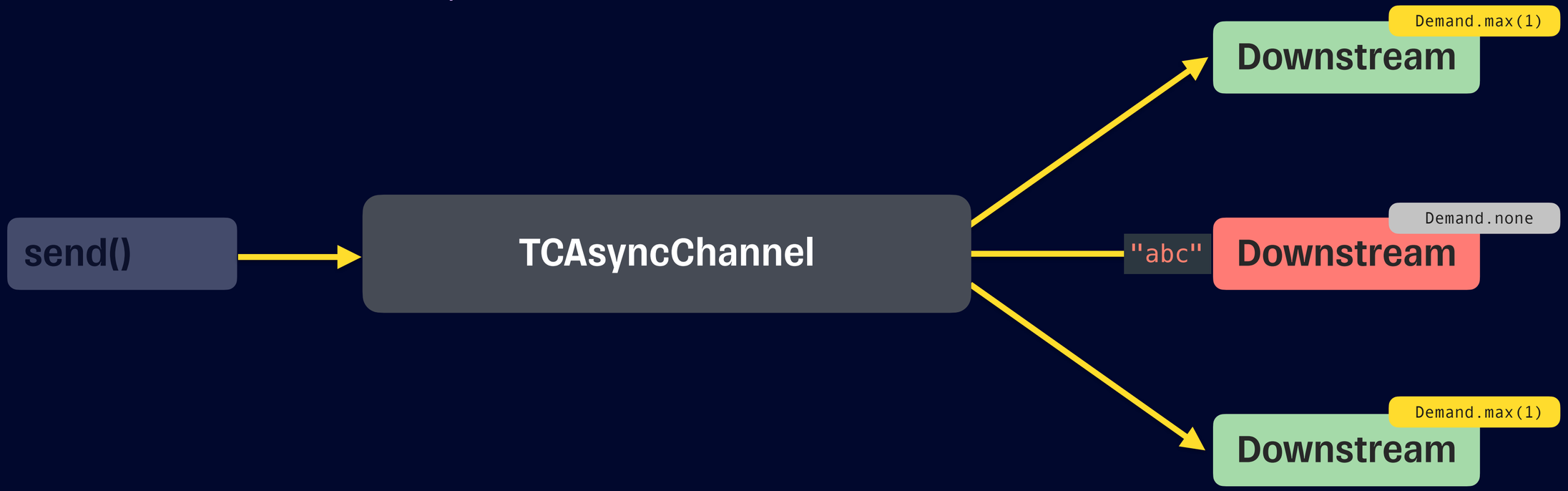
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



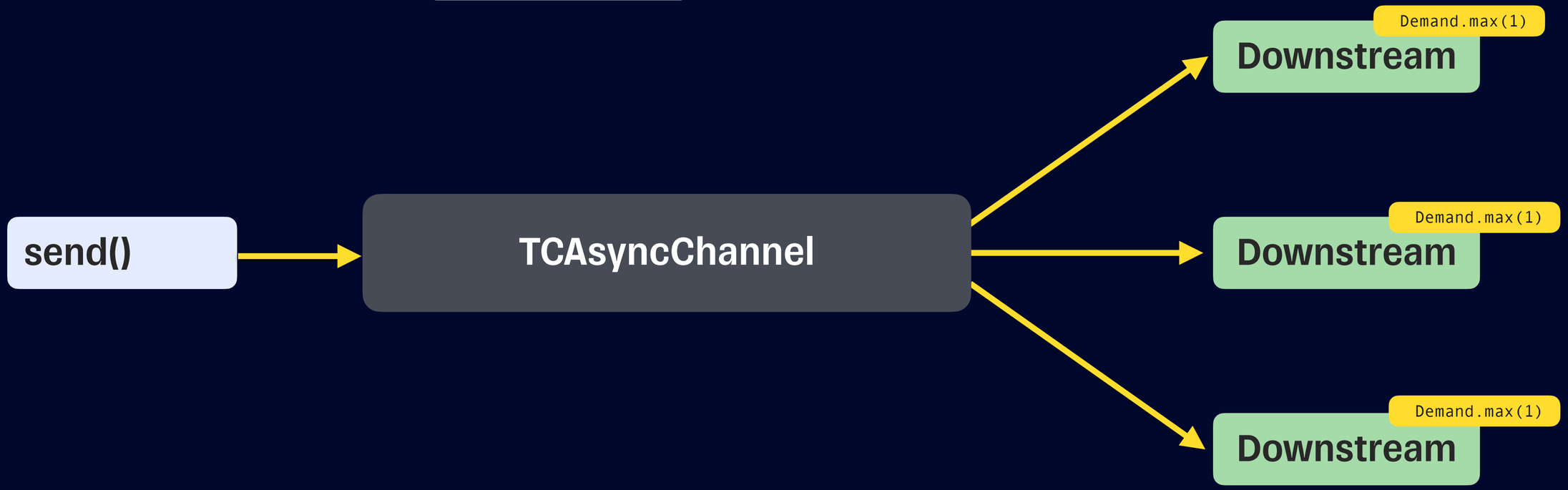
Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



Combine и SwiftConcurrency

```
print("sending")
await channel.send("abc")
print("sent")
```



Тестирование

Без него и не надо, а с ним и пусть будет

Тестирование

```
actor SearchingDataSource: ISearchingDataSource {
    ...
    var items: AnyPublisher<[ExampleViewModel], Never> {
        get async {
            let viewModels = await dependencies.someFacade.items

            let search = $searchText
                .debounce(for: 0.5, scheduler: dependencies.backgroundScheduler)
                .merge(with: $searchText.prefix(1))
                .removeDuplicates()

            let filteredModels = viewModels.combineLatest(search) { viewModels, search in
                viewModels.filter {
                    search.isEmpty || $0.content.localizedCaseInsensitiveContains(search)
                }
            }

            return filteredModels.eraseToAnyPublisher()
        }
    }

    // MARK: - Private Properties
    @Published
    private var searchText: String = ""
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_passthrough() async {
    // given
    let first = [ExampleViewModel.fake()]
    let second = [ExampleViewModel.fake(), ExampleViewModel.fake()]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    dependencies._backgroundScheduler = DispatchQueue.test.eraseToAnyScheduler()

    await dependencies._someFacade.access {
        $0.stubbedItems = subject.eraseToAnyPublisher()
    }

    let searchingDataSource = SearchingDataSource(
        dependencies: dependencies
    )

    var result: [[ExampleViewModel]] = []

    // when
    let cancellable = await searchingDataSource.items.sink { result.append($0) }

    // then
    XCTAssertEqual(result, [])

    // when
    subject.send(first)

    // then
    XCTAssertEqual(result, [first])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Тестирование

```
func test_searchingDataSource_search() async {
    // given
    let first = [ExampleViewModel.fake(content: "ggg")]
    let second = [ExampleViewModel.fake(content: "123"), ExampleViewModel.fake(content: "abc")]
    let subject = PassthroughSubject<[ExampleViewModel], Never>()

    let backgroundScheduler = DispatchQueue.test

    dependencies._backgroundScheduler = backgroundScheduler.eraseToAnyScheduler()

    ...

    // when
    subject.send(first)
    await searchingDataSource.setSearchText("12")

    // then
    XCTAssertEqual(result, [first])

    // when
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, []])

    // when
    subject.send(second)

    // then
    XCTAssertEqual(result, [first, [], [second[0]]])

    // when
    await searchingDataSource.setSearchText("")
    await backgroundScheduler.advance(by: .seconds(0.6))

    // then
    XCTAssertEqual(result, [first, [], [second[0]], second])
}
```

Combine и SwiftConcurrency

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            // выполнится где-то на другом потоке
            let values = await someFacade.items.asyncValues // на await можем уснуть
            for await values in values { // на await можем уснуть
                await view?.updateItems(values) // на await можем уснуть
            }
        }
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Combine и SwiftConcurrency

```
let channel = TCAsyncChannel<Int, Never>()

Task {
    try await channel.send(1)
    // к этому времени на консоль выведется x: 1
    try await channel.send(2)
    // к этому времени на консоль выведется x: 2
    try await channel.send(3)
    // к этому времени на консоль выведется x: 3
    ...
}

Task {
    for await x in channel.asyncValues {
        await doSomething()
        print("x: \(x)")
    }
}
```

Инструменты для тестирования

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyncValues
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

Инструменты для тестирования

```
final class SomePresenterTests {
    var view: SomeViewMock!
    var facade: SomeFacadeMock!
    var presenter: SomePresenter!
    ...
    func testViewDidLoad() async throws {
        // given
        let items = TCAsyncChannel<[SomeViewModel], Never>()
        facade.stubbedItems = items.eraseToAnyPublisher()

        let input = [SomeViewModel.fake()]
        // when
        presenter.viewDidLoad()
        try await items.send(input) // дождётся view.updateItems
        // then
        XCTAssertEqual(view.invokedUpdateItemsParameters, input)
    }
}
```

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyncValues
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

Инструменты для тестирования

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyncValues
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

```
final class SomePresenterTests {
    var view: SomeViewMock!
    var facade: SomeFacadeMock!
    var presenter: SomePresenter!
    ...
    func testViewDidLoad() async throws {
        // given
        let items = TCAsyncChannel<[SomeViewModel], Never>()
        facade.stubbedItems = items.eraseToAnyPublisher()

        let input = [SomeViewModel.fake()]
        // when
        presenter.viewDidLoad()
        try await items.send(input) // дождётся view.updateItems
        // then
        XCTAssertEqual(view.invokedUpdateItemsParameters, input)
    }
}
```

Инструменты для тестирования

```
final class SomePresenterTests {
    var view: SomeViewMock!
    var facade: SomeFacadeMock!
    var presenter: SomePresenter!
    ...
    func testViewDidLoad() async throws {
        // given
        let items = TCAsyncChannel<[SomeViewModel], Never>()
        facade.stubbedItems = items.eraseToAnyPublisher()

        let input = [SomeViewModel.fake()]
        // when
        presenter.viewDidLoad()
        try await items.send(input) // дождётся view.updateItems
        // then
        XCTAssertEqual(view.invokedUpdateItemsParameters, input)
    }
}
```

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyncValues
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

Инструменты для тестирования

```
final class SomePresenterTests {
    var view: SomeViewMock!
    var facade: SomeFacadeMock!
    var presenter: SomePresenter!
    ...
    func testViewDidLoad() async throws {
        // given
        let items = TCAsyncChannel<[SomeViewModel], Never>()
        facade.stubbedItems = items.eraseToAnyPublisher()

        let input = [SomeViewModel.fake()]
        // when
        presenter.viewDidLoad()
        try await items.send(input) // дождётся view.updateItems
        // then
        XCTAssertEqual(view.invokedUpdateItemsParameters, input)
    }
}
```

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyncValues
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

Инструменты для тестирования

```
final class SomePresenterTests {
    var view: SomeViewMock!
    var facade: SomeFacadeMock!
    var presenter: SomePresenter!
    ...
    func testViewDidLoad() async throws {
        // given
        let items = TCAsyncChannel<[SomeViewModel], Never>()
        facade.stubbedItems = items.eraseToAnyPublisher()

        let input = [SomeViewModel.fake()]
        // when
        presenter.viewDidLoad()
        try await items.send(input) // дождётся view.updateItems
        // then
        XCTAssertEqual(view.invokedUpdateItemsParameters, input)
    }
}
```

```
final class SomePresenter {
    ...
    func viewDidLoad() {
        Task { [weak view, someFacade] in
            let values = await someFacade.items.asyn
            for await values in values {
                await view?.updateItems(values)
            }
        }
    }
}
```

Инструменты для тестирования

```
func removeActionTapped(object: SomeViewModel) {
    Task {
        await someFacade.remove(object)
    }
}
```

Инструменты для тестирования

```
import TinkoffConcurrency

func removeActionTapped(object: SomeViewModel) {
    taskFactory.task {
        await someFacade.remove(object)
    }
}
```

Инструменты для тестирования

```
func testSomePresenter_removeActionTapped() async {
    // given
    let taskFactory = TCTestTaskFactory()
    let someFacade = SomeFacadeMock()

    let presenter = SomePresenter(taskFactory: taskFactory, someFacade: someFacade)

    // when
    presenter.removeActionTapped()

    await taskFactory.runUntilIdle()

    // then
    XCTAssertTrue(someFacade.invokedRemove)
}
```

Combine
и SwiftConcurrency

https://github.com/tinkoff-mobile-tech/TinkoffConcurrency

Демо

Приложение

https://github.com/adarovsky/MobiusShowcase.git