Embracing Reactive Data and View State

James Paolantonio
james.paolantonio@betterment.com
@jpaolantonio

Betterment

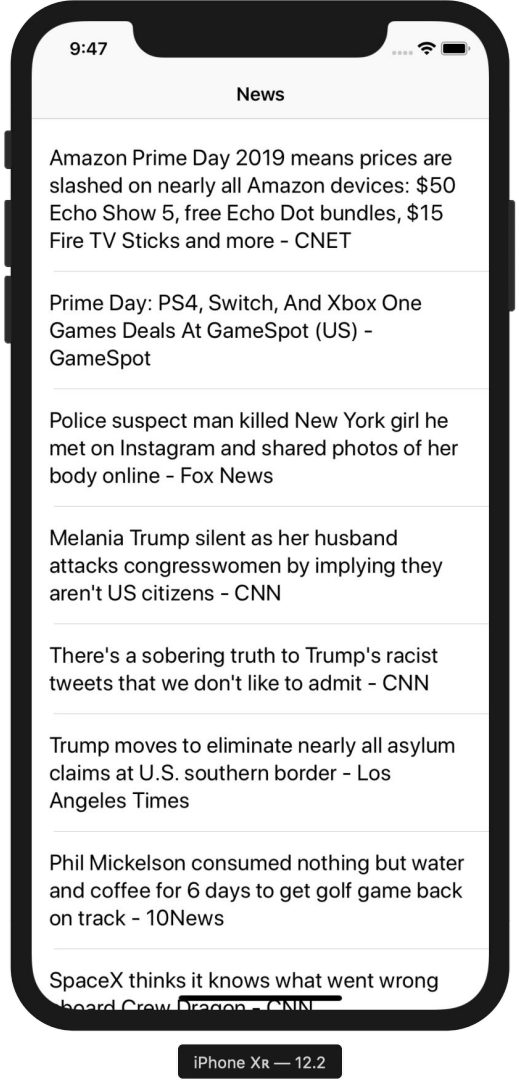
#Buzzwords

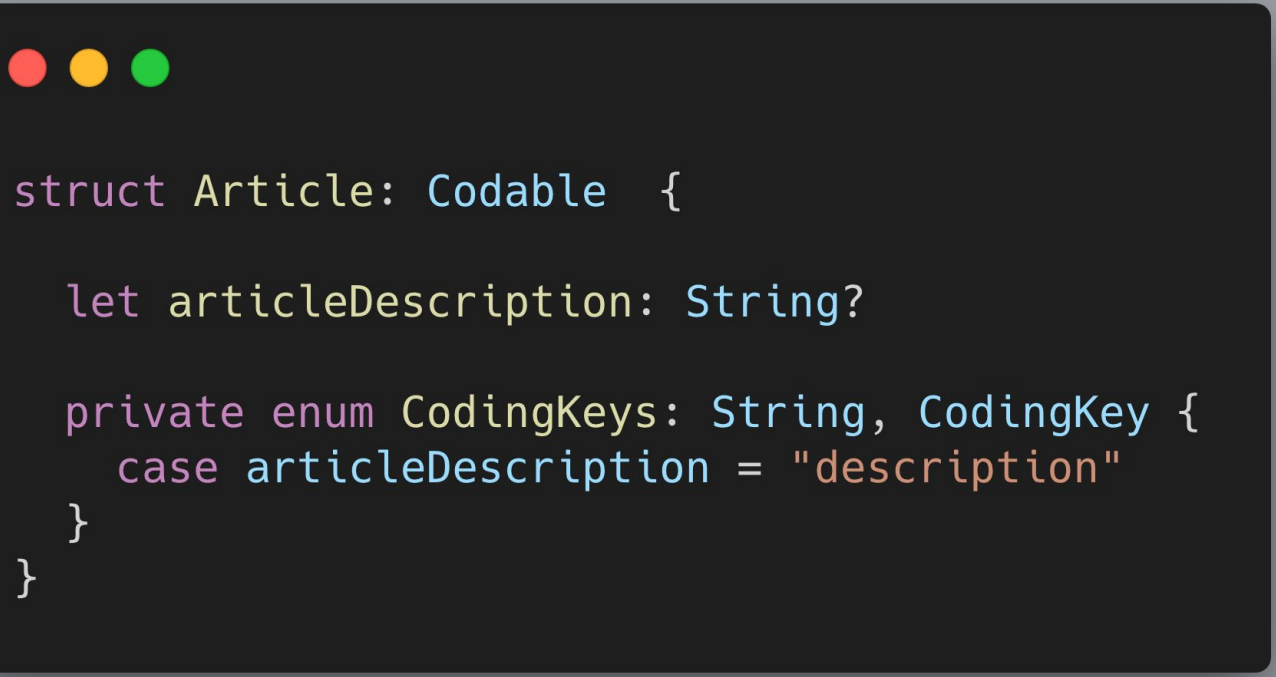
How we think about building mobile apps

How can we use newer tools to empower us

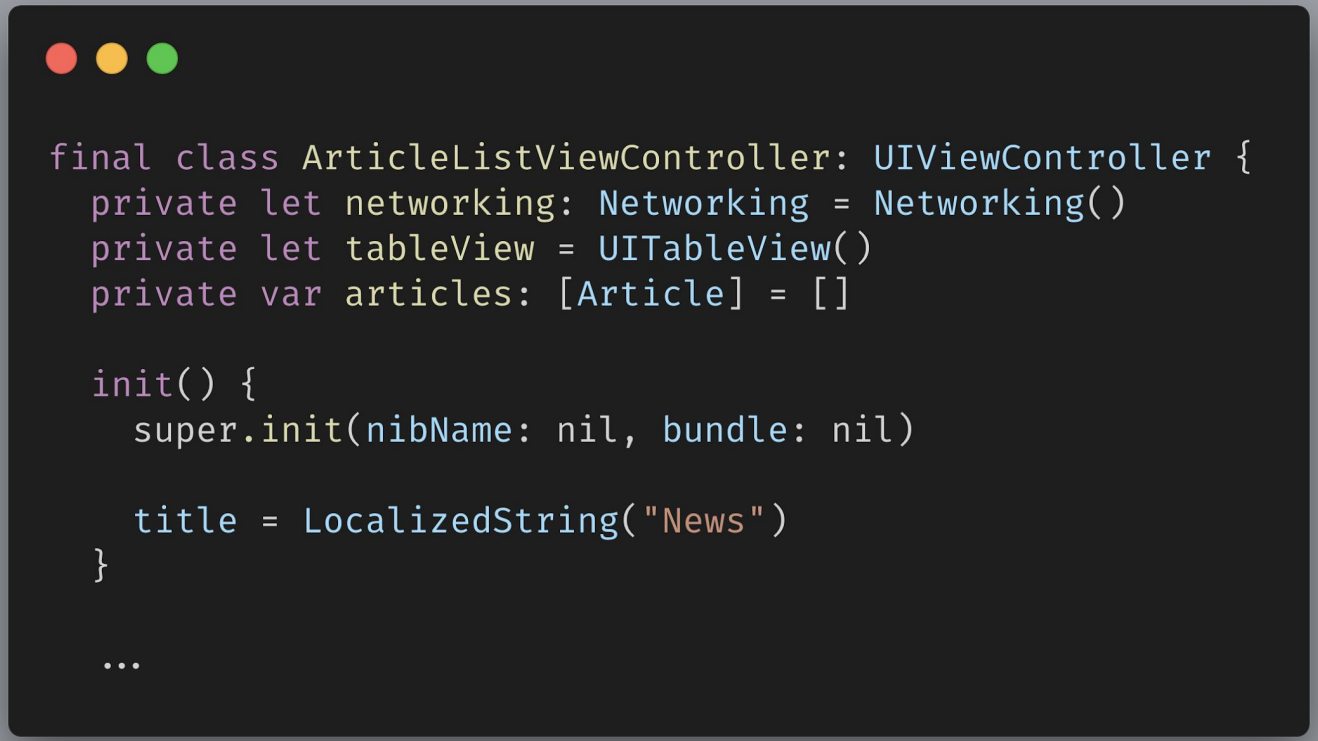
Let’s build an app

What could go wrong?





```
struct Article: Codable  {

  let articleDescription: String?

  private enum CodingKeys: String, CodingKey {
    case articleDescription = "description"
  }
}
```

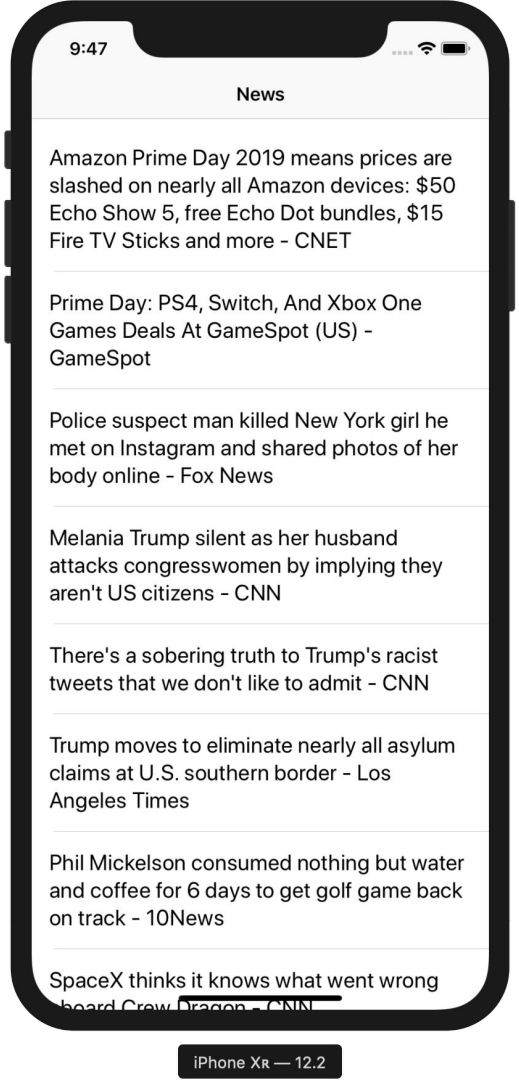


```swift
final class ArticleListViewController: UIViewController {
  private let networking: Networking = Networking()
  private let tableView = UITableView()
  private var articles: [Article] = []

  init() {
    super.init(nibName: nil, bundle: nil)

    title = LocalizedString("News")
  }

  ...
```

```
  override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    networking
      .request(NewsApi.topHeadlines(request: TopHeadlinesRequest())) { result in
        switch result {
        case let .success(data):
          self.handle(data)
        case .failure:
          break
        }
    }
  }

  private func handle(_ result: Response) {
    guard let headlinesResponse = try? JSONDecoder().decode(TopHeadlinesResponse.self, from:
result.data) else {
      return
    }

    articles = headlinesResponse.articles
    tableView.reloadData()
  }
```

Now What

What can we improve?



Combine

Coordinators

SwiftUI

Snapshot Tests

Reactive

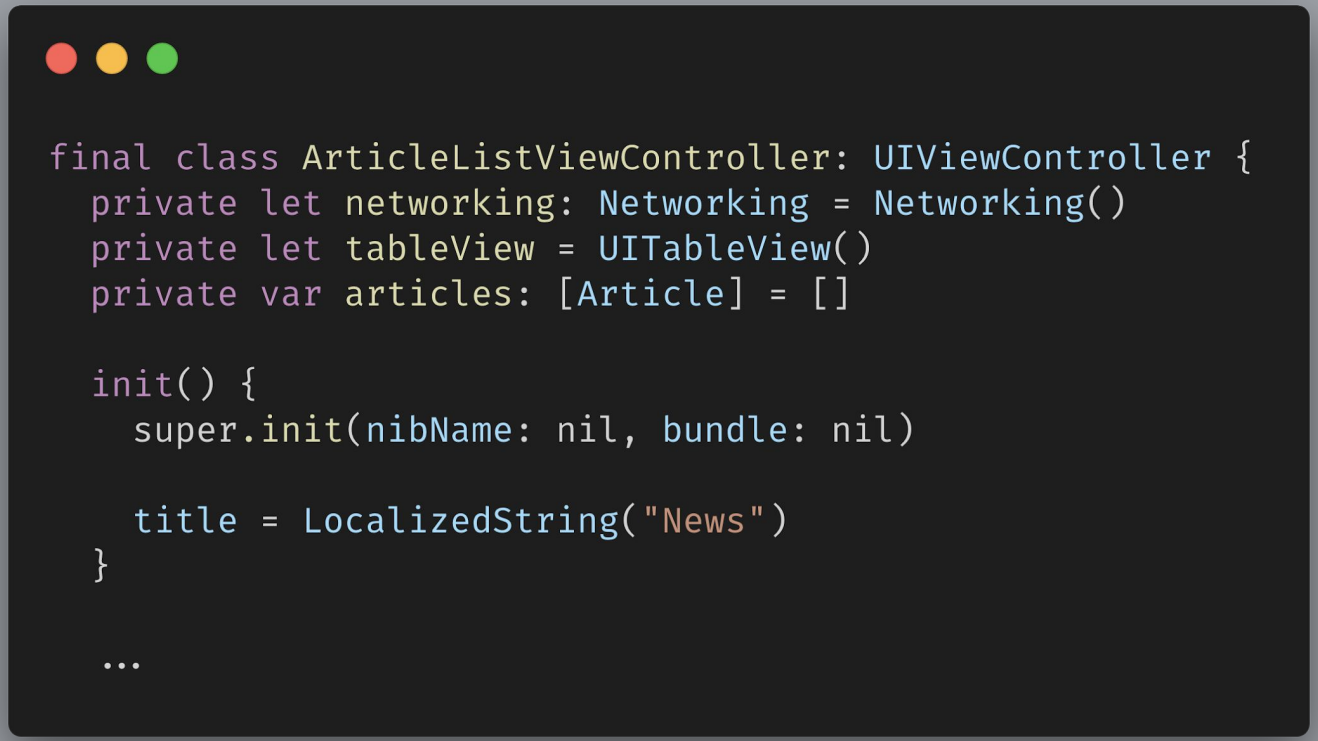
View Models

Functional

Coordinators

Declarative

```
  override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    networking
      .request(NewsApi.topHeadlines(request: TopHeadlinesRequest())) { result in
        switch result {
        case let .success(data):
          self.handle(data)
        case .failure:
          break
        }
    }
  }

  private func handle(_ result: Response) {
    guard let headlinesResponse = try? JSONDecoder().decode(TopHeadlinesResponse.self, from:
result.data) else {
      return
    }

    articles = headlinesResponse.articles
    tableView.reloadData()
  }
```



```swift
final class ArticleListViewController: UIViewController {
  private let networking: Networking = Networking()
  private let tableView = UITableView()
  private var articles: [Article] = []

  init() {
    super.init(nibName: nil, bundle: nil)

    title = LocalizedString("News")
  }

  ...
```

Building an app

Representing a combination of local and remote state to a user

Is a user logged in?

How much money is in their account?

Should I show view A or B?

Technical problems when we mess this up

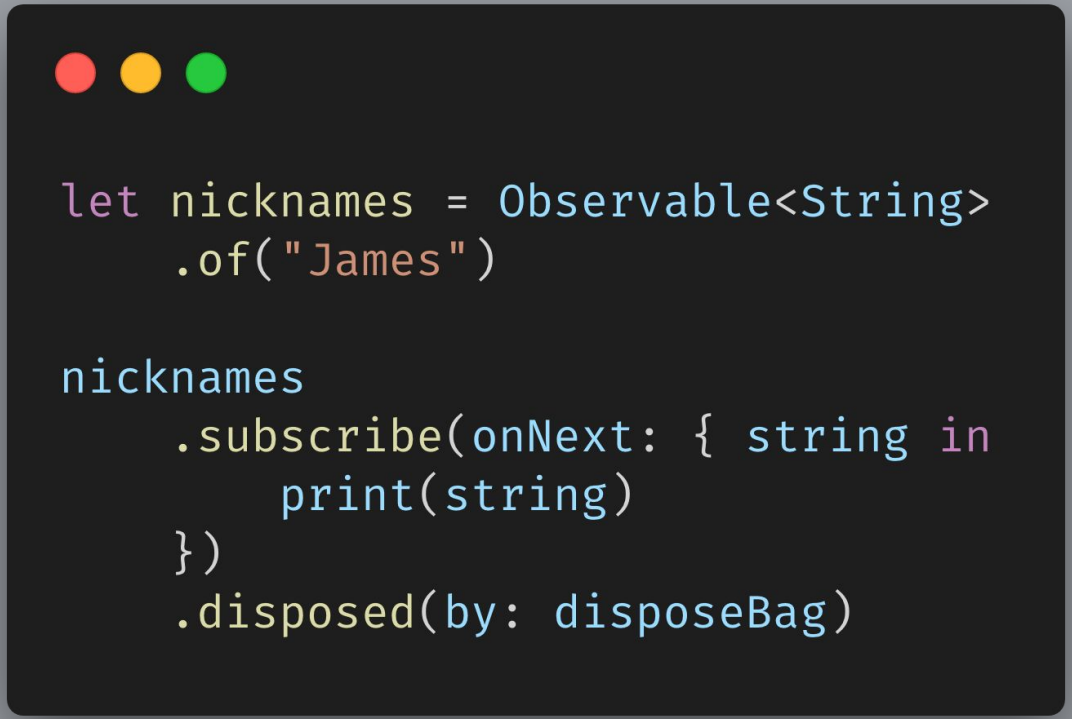
This view didn’t update?!?

Do I update data on viewWillAppear?

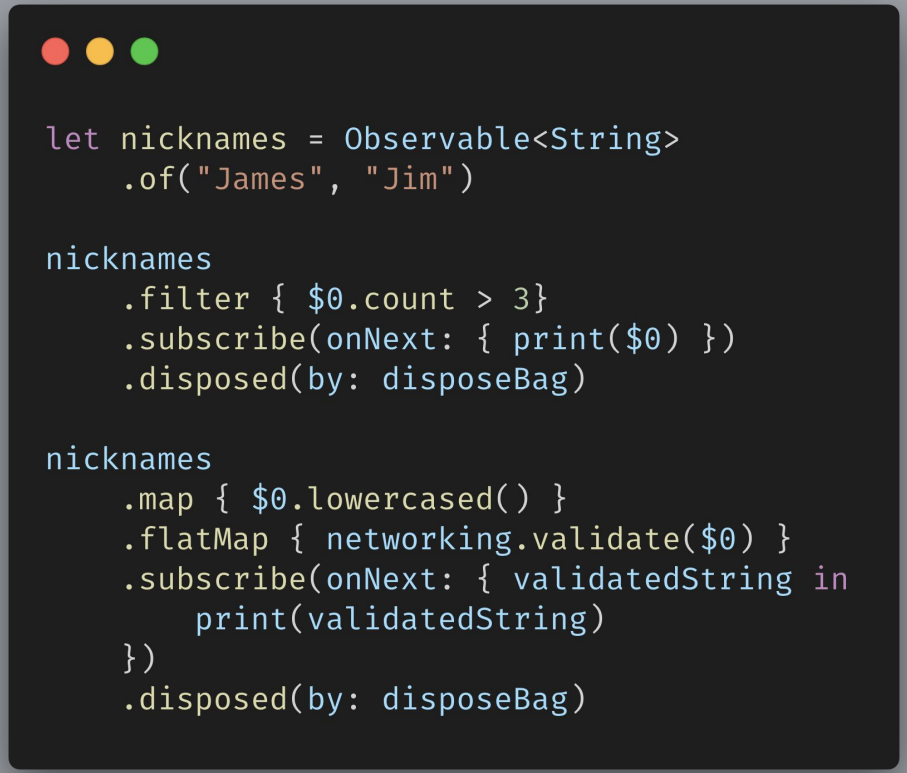
I think there is a bug because of some cached data

Reactive Data


Observable<String>.of("Hello")
Observable<String>.of("Hello", "World")
Observable<String>.error(TestError.error)
Observable<String>.never()
Observable<String>.empty()



```
let nicknames = Observable<String>
    .of("James")

nicknames
    .subscribe(onNext: { string in
        print(string)
    })
    .disposed(by: disposeBag)
```

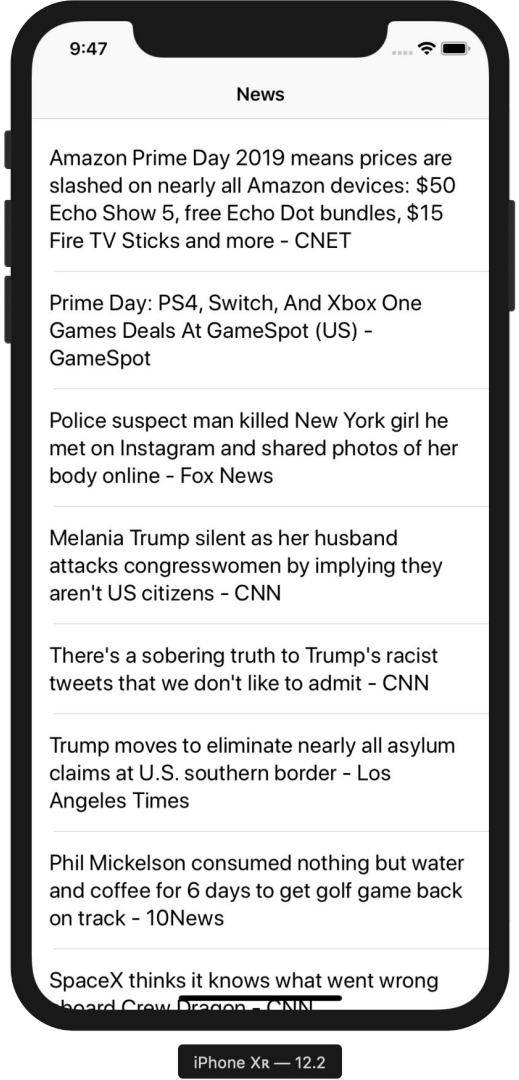


```
let nicknames = Observable<String>
    .of("James", "Jim")

nicknames
    .filter { $0.count > 3}
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)

nicknames
    .map { $0.lowercased() }
    .flatMap { networking.validate($0) }
    .subscribe(onNext: { validatedString in
        print(validatedString)
    })
    .disposed(by: disposeBag)
```

View State

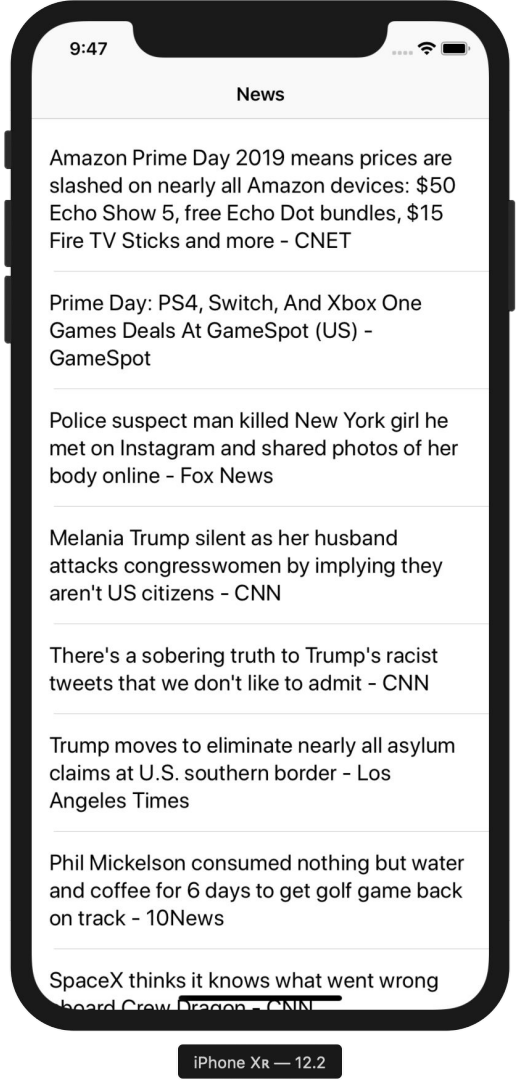
View State

Array of Articles



View State

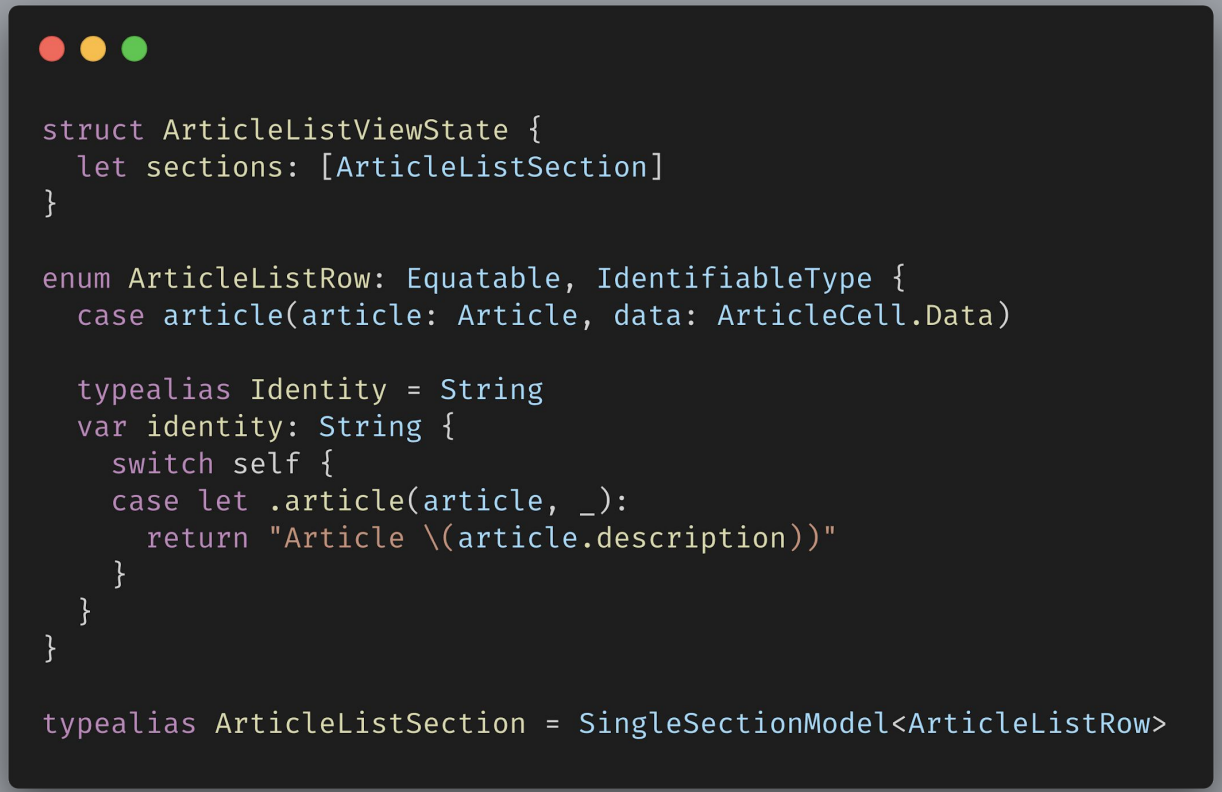
Array of **cell data**



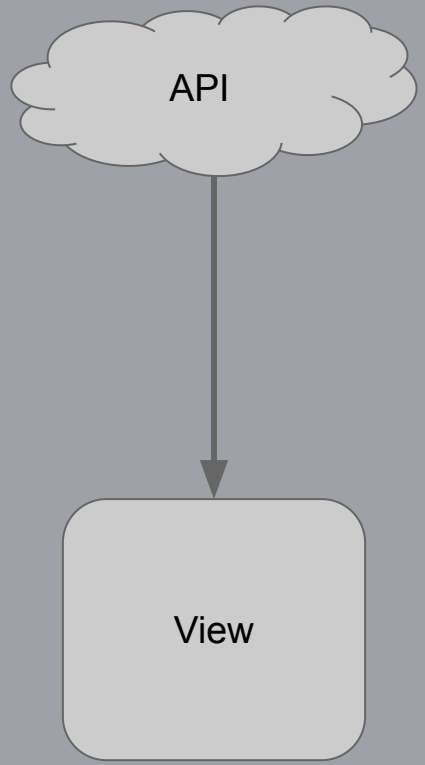
Starting to think about building a layered system

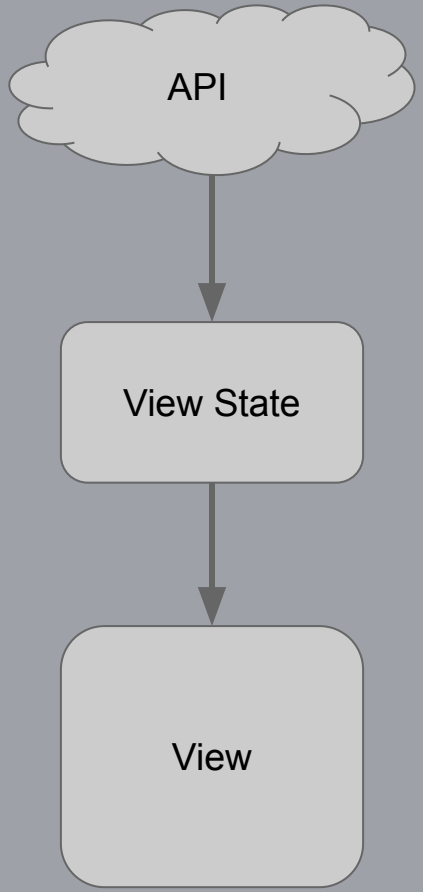
```swift
final class ArticleCell: UITableViewCell {
  private let titleLabel = UILabel()

  override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
    super.init(style: style, reuseIdentifier: reuseIdentifier)

    /// ...
  }

  /// ...

  func update(article: Article) {
    titleLabel.text = article.description
  }
}
```

```swift
final class ArticleCell: UITableViewCell {
  public struct Data: Codable {
    let title: String
  }

  private let titleLabel = UILabel()

  override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
    super.init(style: style, reuseIdentifier: reuseIdentifier)

    /// ...
  }

  /// ...

  func update(data: Data) {
    titleLabel.text = data.title
  }
}
```

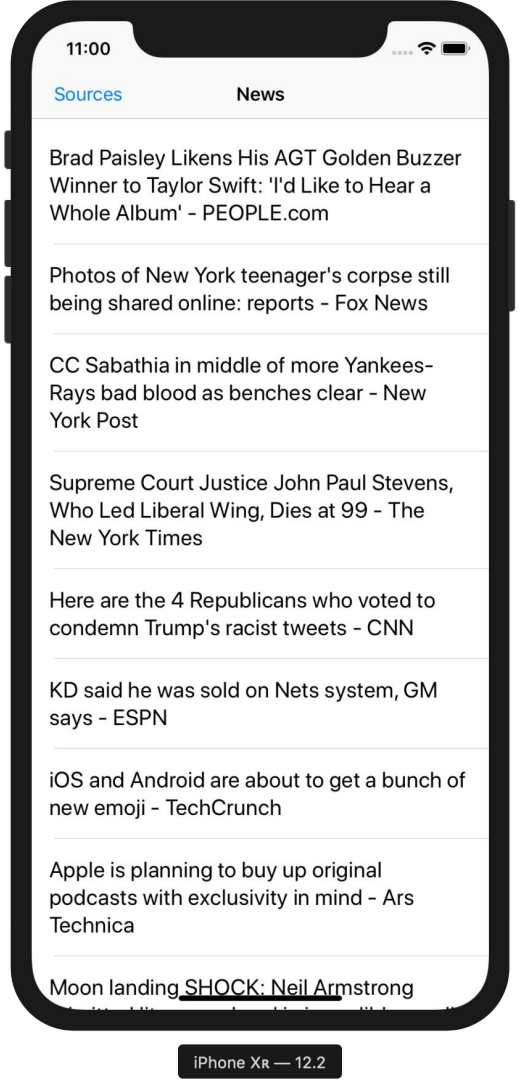


```
struct ArticleListViewState {
  let sections: [ArticleListSection]
}

enum ArticleListRow: Equatable, IdentifiableType {
  case article(article: Article, data: ArticleCell.Data)

  typealias Identity = String
  var identity: String {
    switch self {
    case let .article(article, _):
      return "Article \(article.description))"
    }
  }
}

typealias ArticleListSection = SingleSectionModel<ArticleListRow>
```

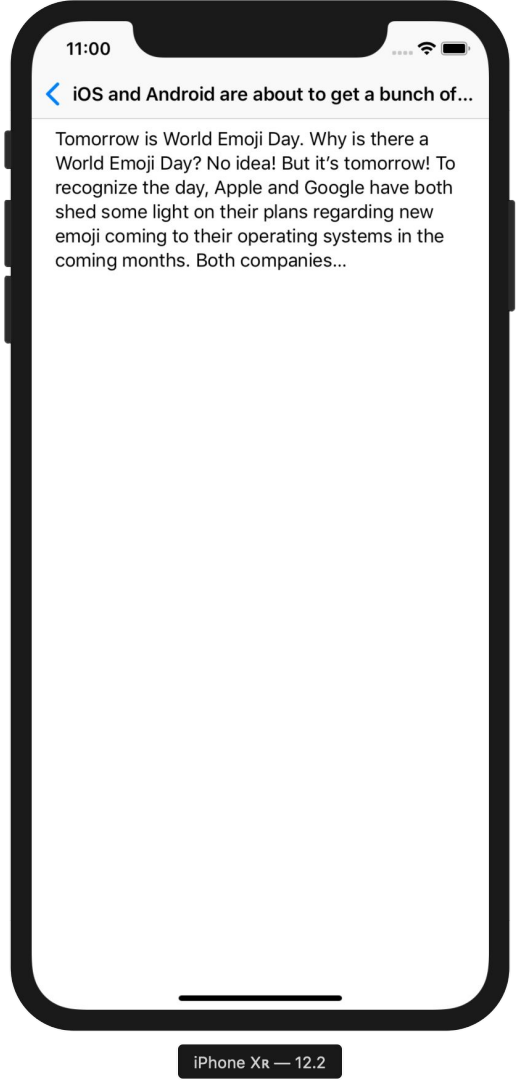

API
View


API
View State
View

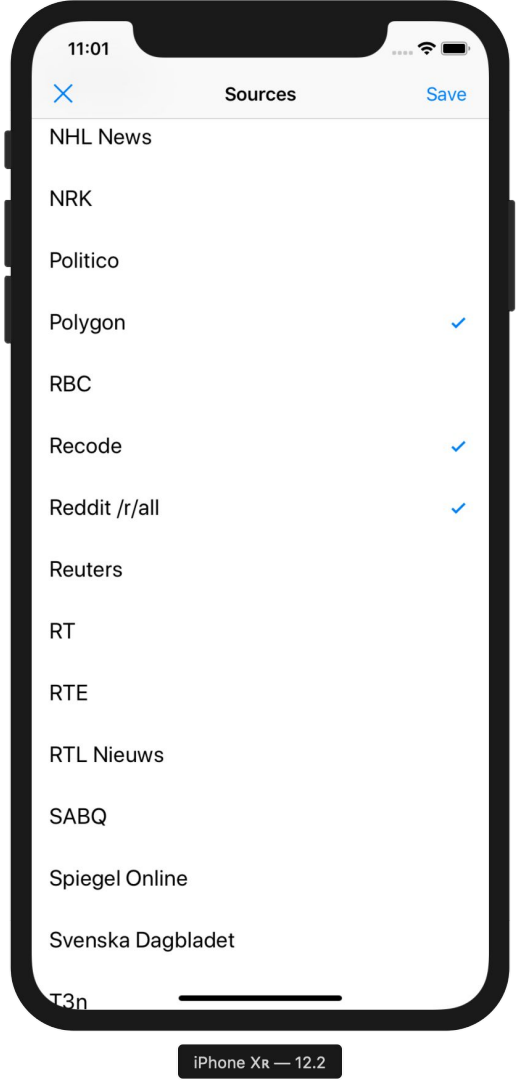
Let's *keep* building an app



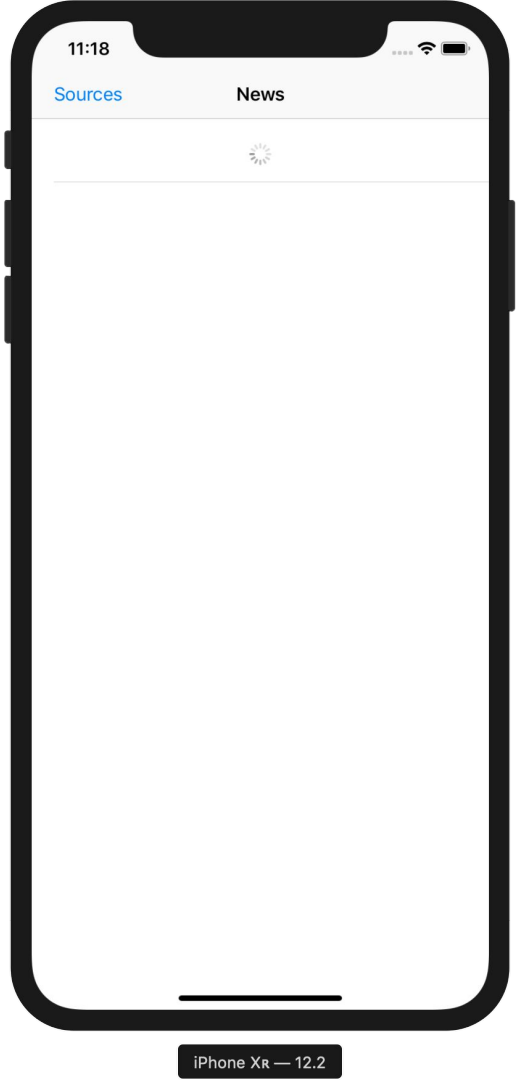
Add a detail screen



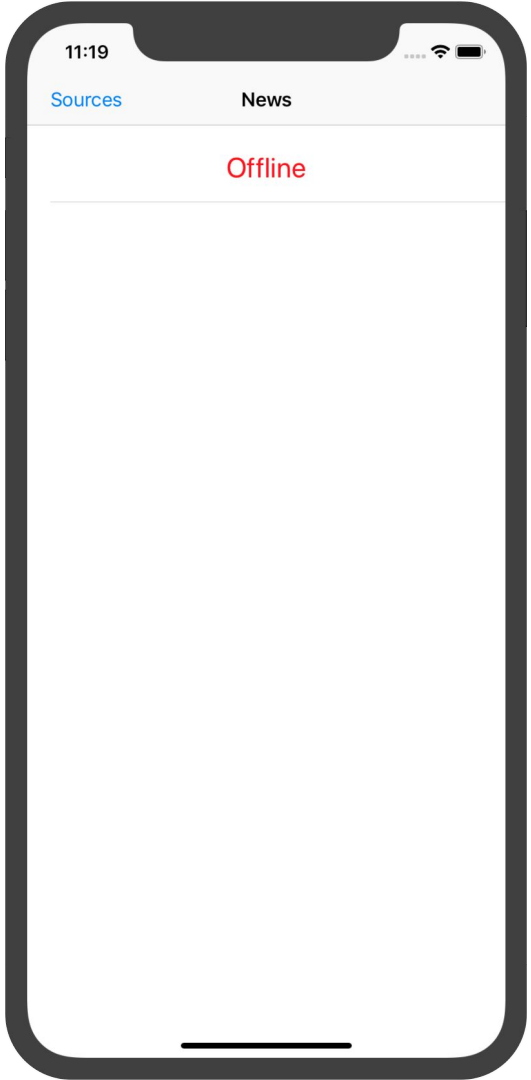
Filter by sources



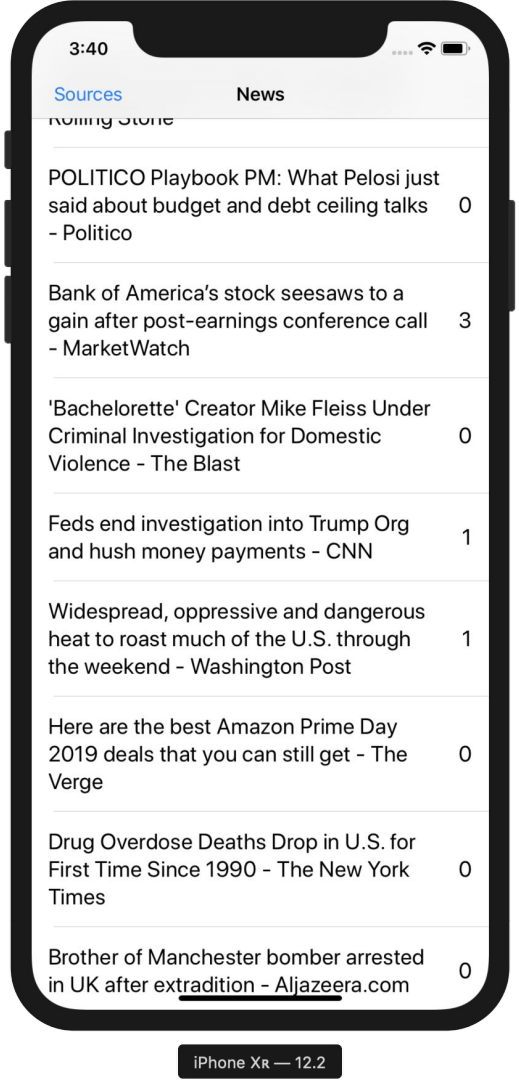
Loading State



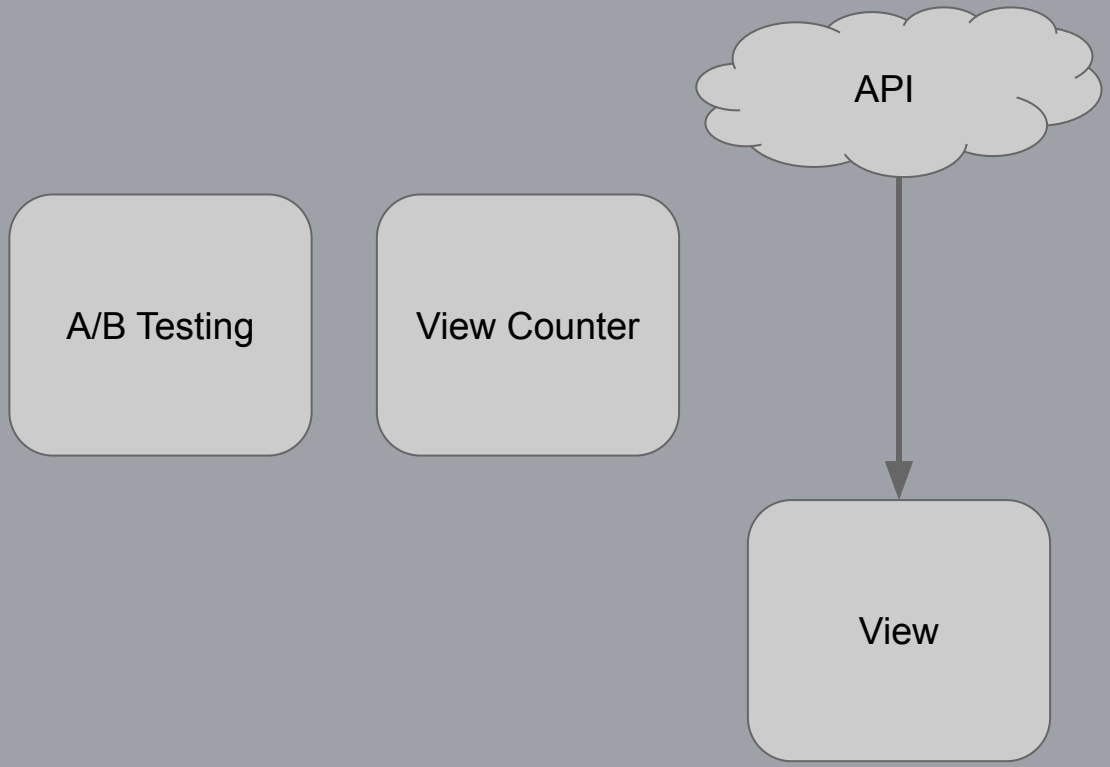
Error State

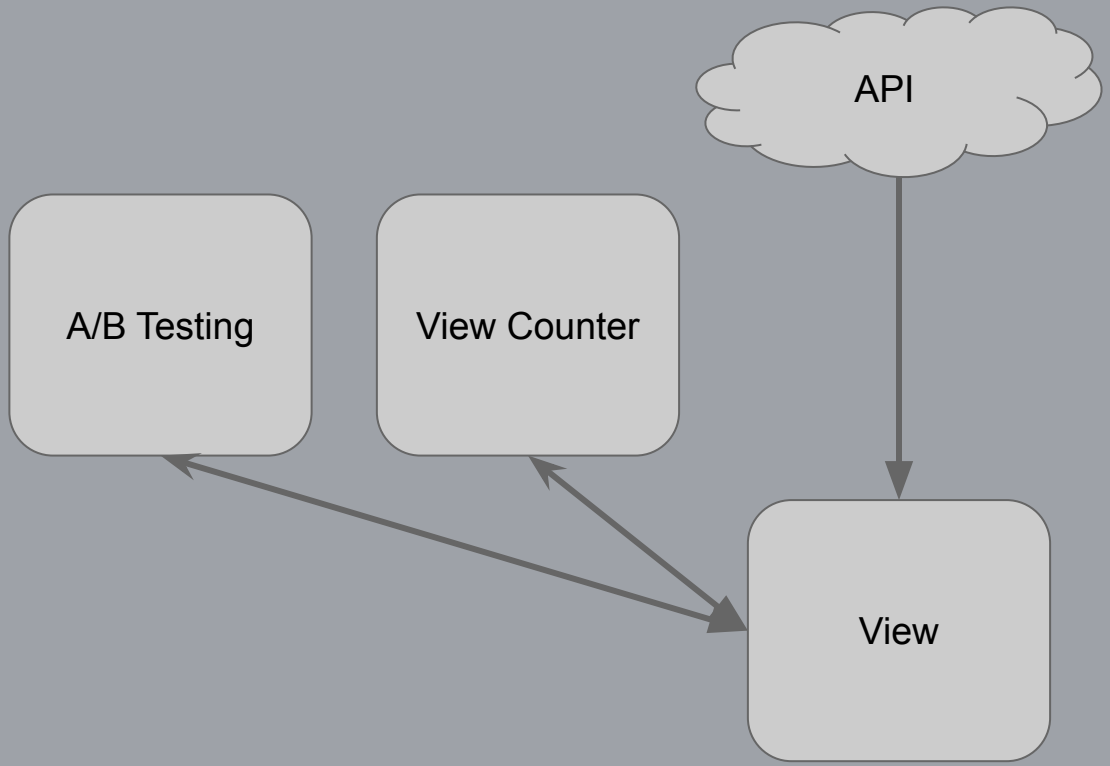


Add a view counter
+
A/B test the cells

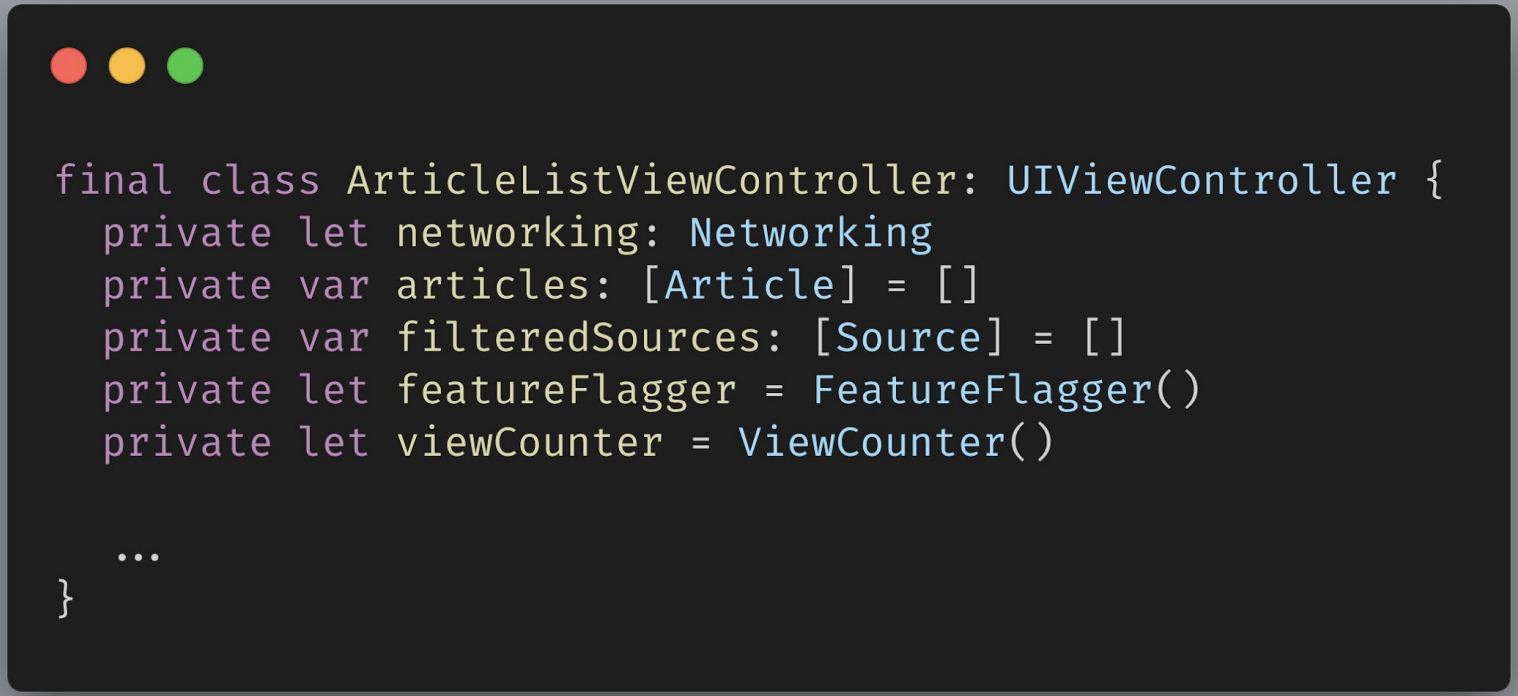


How do we build this “the easy way”


API
A/B Testing
View Counter
View

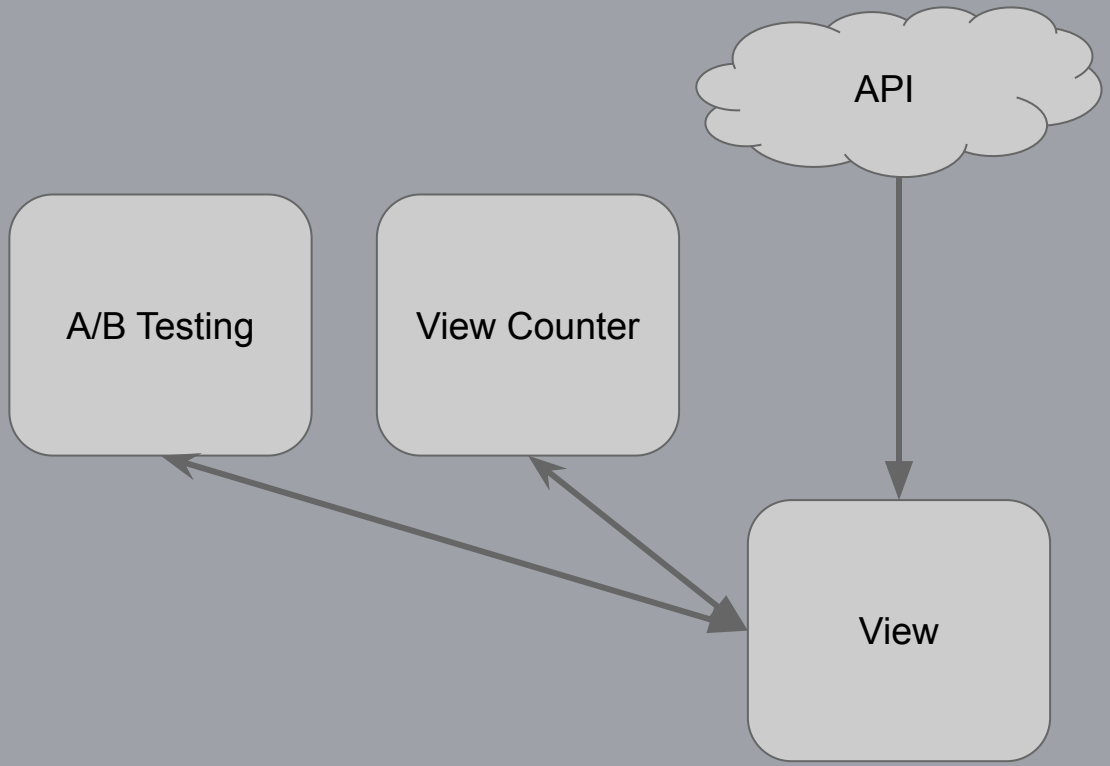

API
A/B Testing
View Counter
View

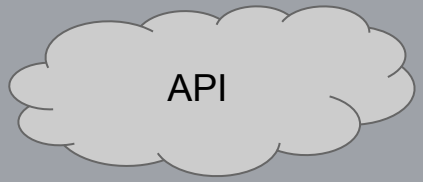
```swift
final class ArticleListViewController: UIViewController {
  private let networking: Networking
  private var articles: [Article] = []
  private var filteredSources: [Source] = []
  private let featureFlagger = FeatureFlagger()
  private let viewCounter = ViewCounter()

  func refreshArticles() {
    let request = TopHeadlinesRequest(sources: filteredSources)
    networking
      .request(NewsApi.topHeadlines(request: request))
      ...
  }

  func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let article = articles[indexPath.row]
    if featureFlag.enabled(.newDesignedCells) {

      let cell = tableView.dequeueReusableCell(cellClass: ArticleViewCell.self, for: indexPath)
      let viewCount = viewCounter.getViewCount(article: article)
      let cellModel = ArticleViewCell.Data(title: article.title, views: viewCount)
      cell.update(cellModel)
      return cell

    } else {

      let cell = tableView.dequeueReusableCell(cellClass: ArticleCell.self, for: indexPath)
      let cellModel = ArticleViewCell.Data(title: article.title
      cell.update(cellModel)
      return cell

    }
  }
}
```


final class ArticleListViewController: UIViewController {
private let networking: Networking
private var articles: [Article] = []
private var filteredSources: [Source] = []
private let featureFlagger = FeatureFlagger()
private let viewCounter = ViewCounter()
...
}

```
final class ArticleListViewController: UIViewController {

  ...

  func refreshArticles() {
    let request = TopHeadlinesRequest(sources: filteredSources)
    networking
      .request(NewsApi.topHeadlines(request: request))
      ...
  }

  func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let article = articles[indexPath.row]
    if featureFlag.enabled(.newDesignedCells) {

      let cell = tableView.dequeueReusableCell(cellClass: ArticleViewCell.self, for: indexPath)
      let viewCount = viewCounter.getViewCount(article: article)
      let cellModel = ArticleViewCell.Data(title: article.title, views: viewCount)
      cell.update(cellModel)
      return cell

    } else {

      let cell = tableView.dequeueReusableCell(cellClass: ArticleCell.self, for: indexPath)
      let cellModel = ArticleViewCell.Data(title: article.title
      cell.update(cellModel)
      return cell

    }
  }
}
```

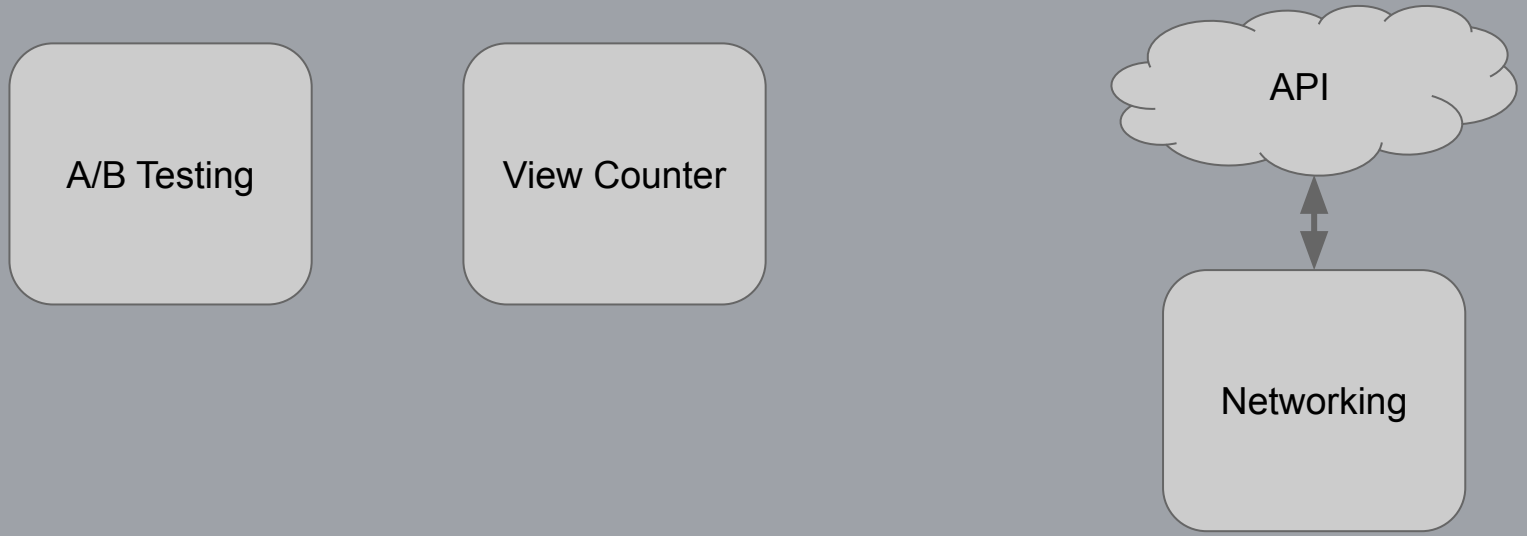
Let’s try looking at the data “streams”


API
A/B Testing
View Counter
View



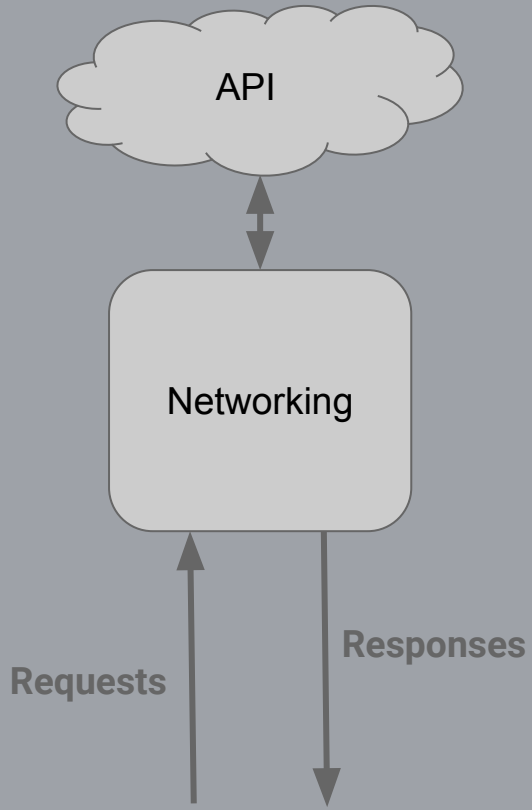
A/B Testing

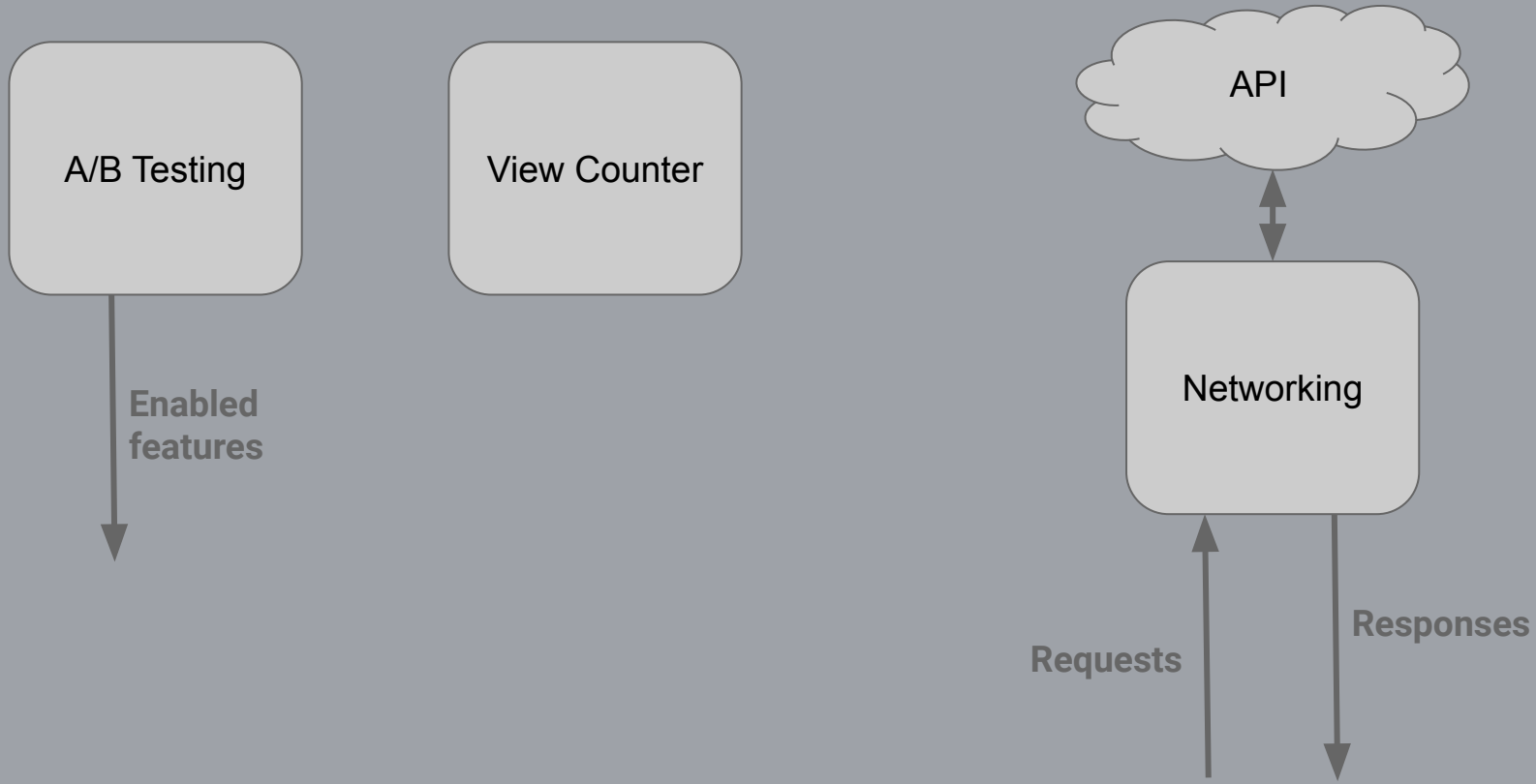
View Counter

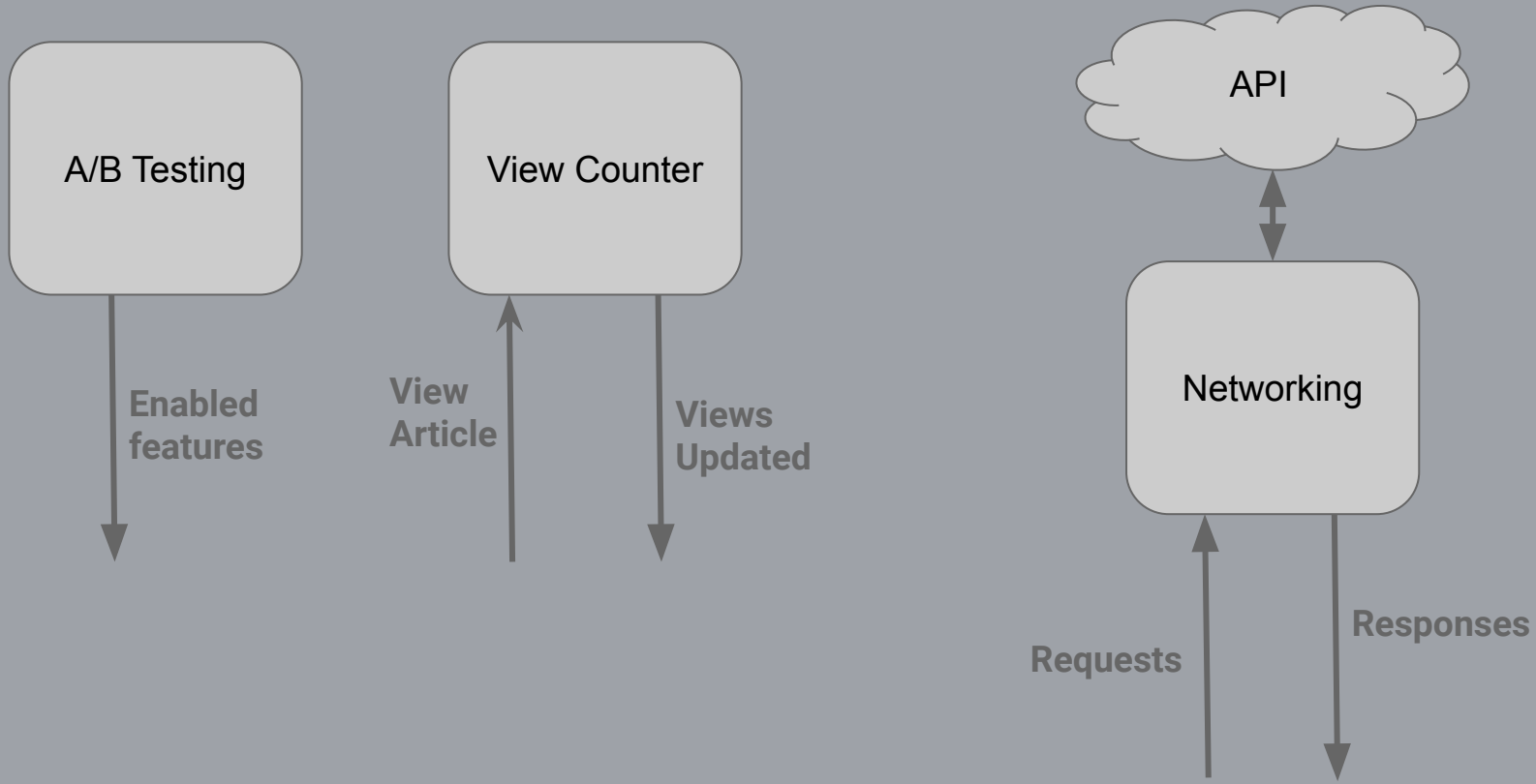

A/B Testing
View Counter
API
Networking

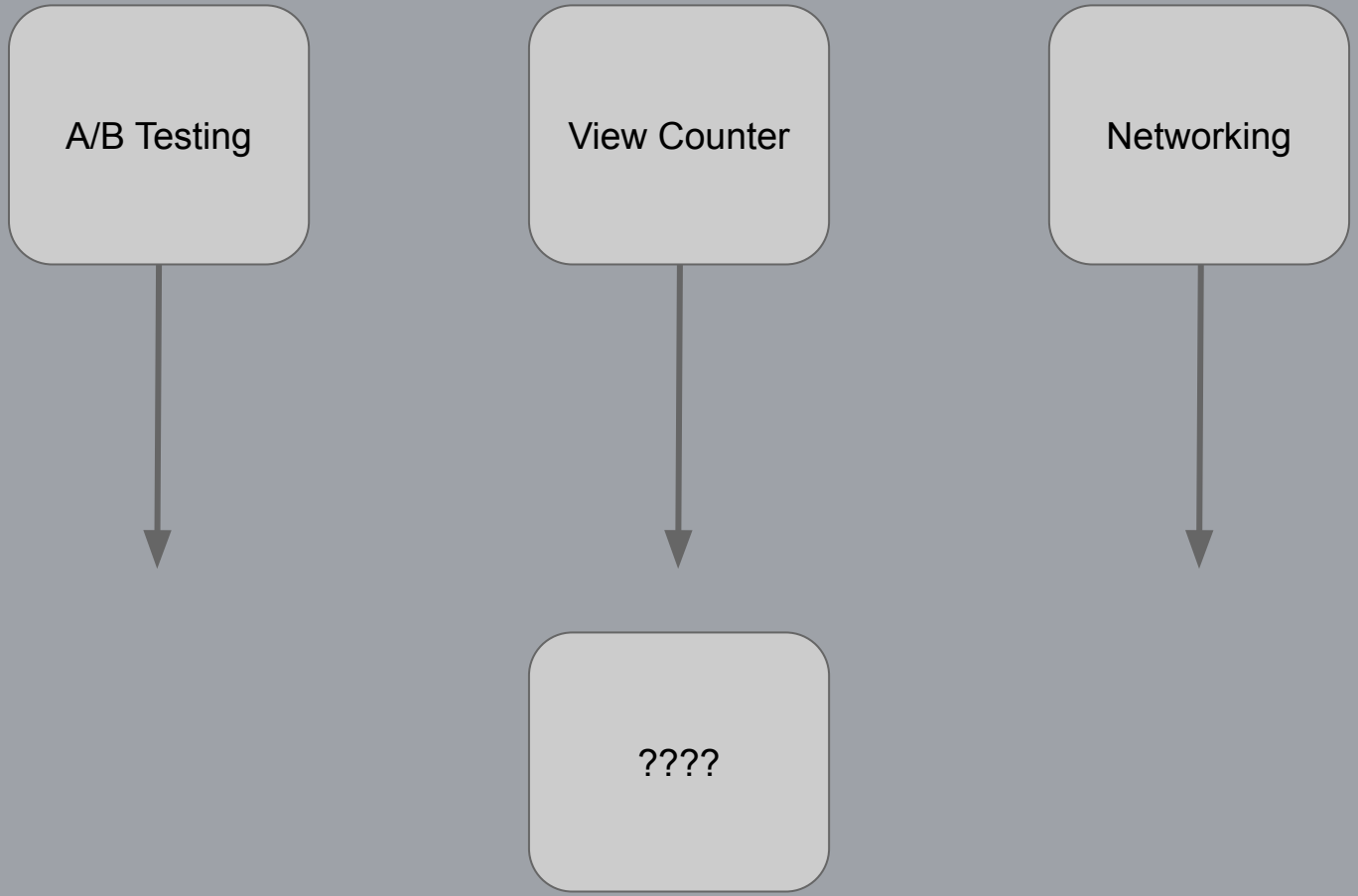
A/B Testing

View Counter

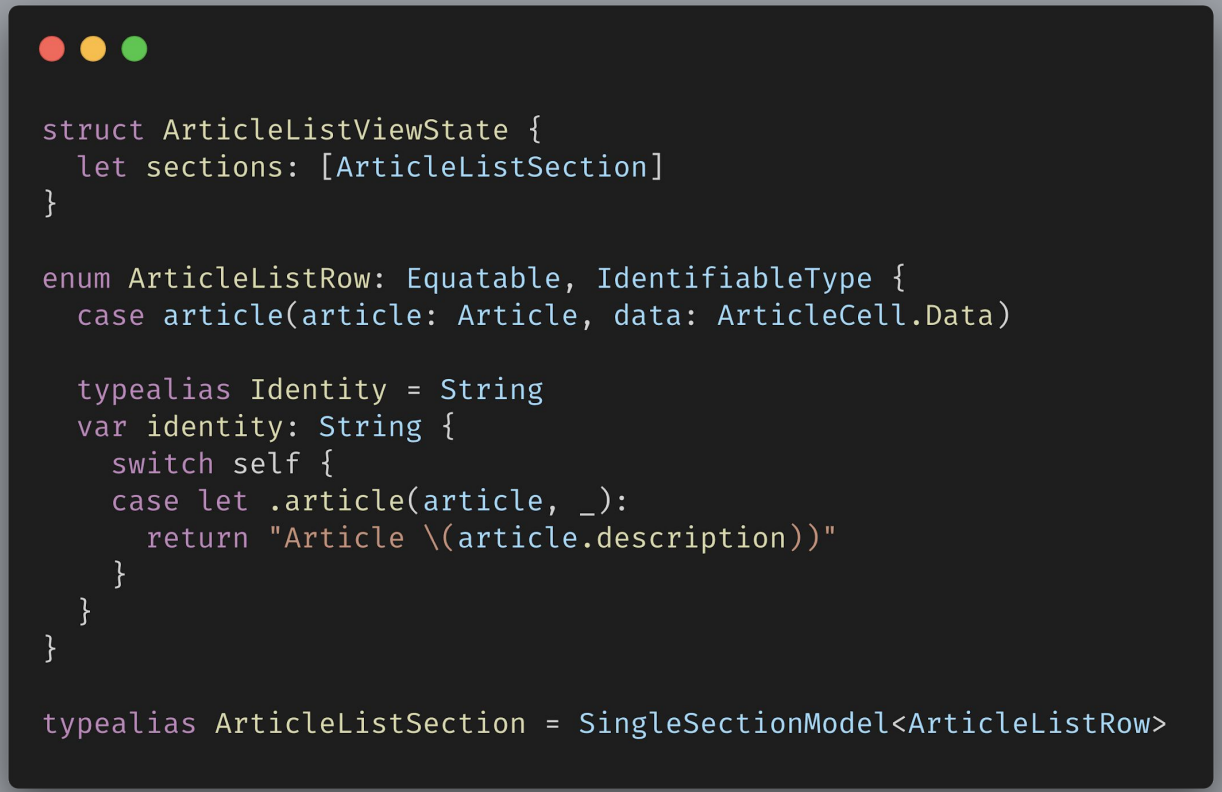



A/B Testing
View Counter
API
Networking
Enabled features
Requests
Responses

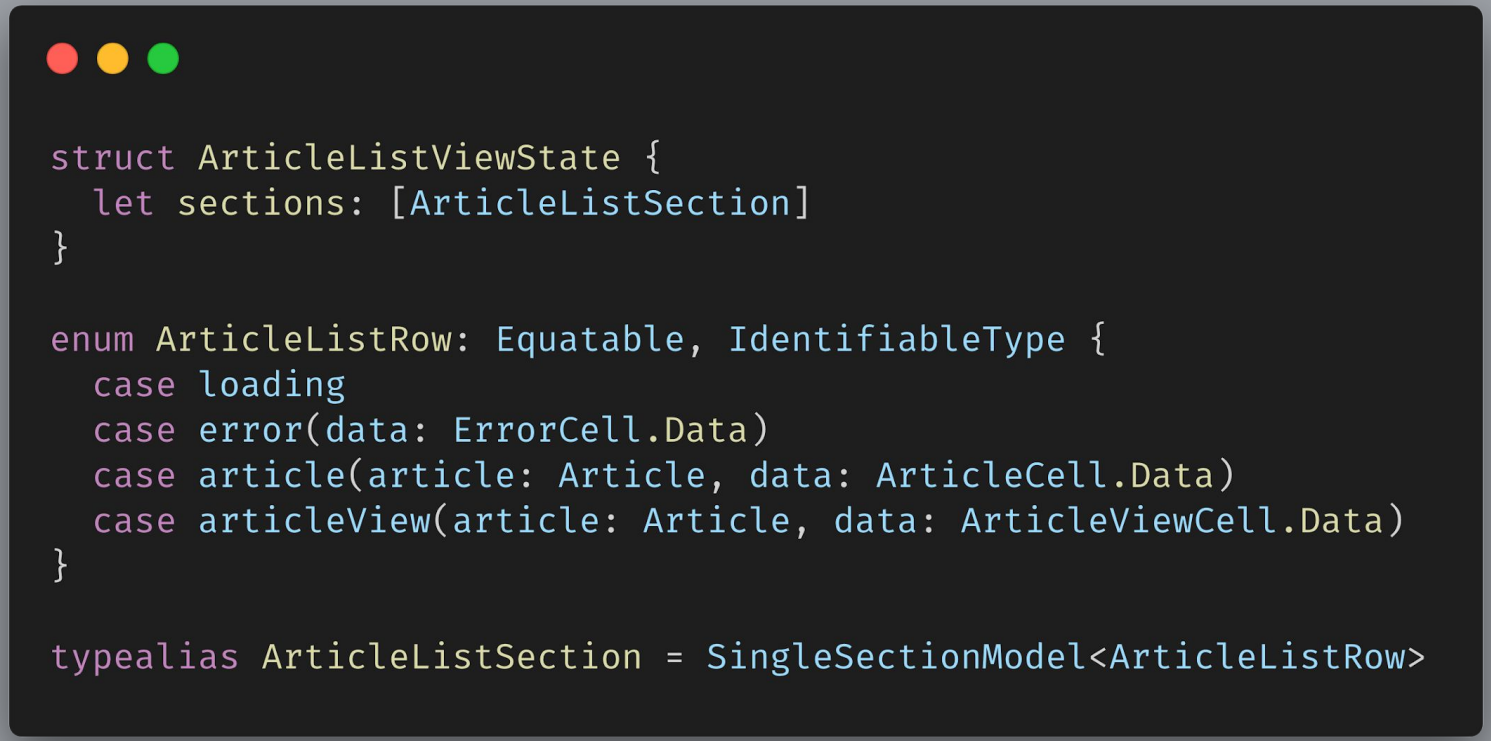

A/B Testing
Enabled features
View Counter
View Article
Views Updated
API
Networking
Requests
Responses


A/B Testing
View Counter
Networking
????

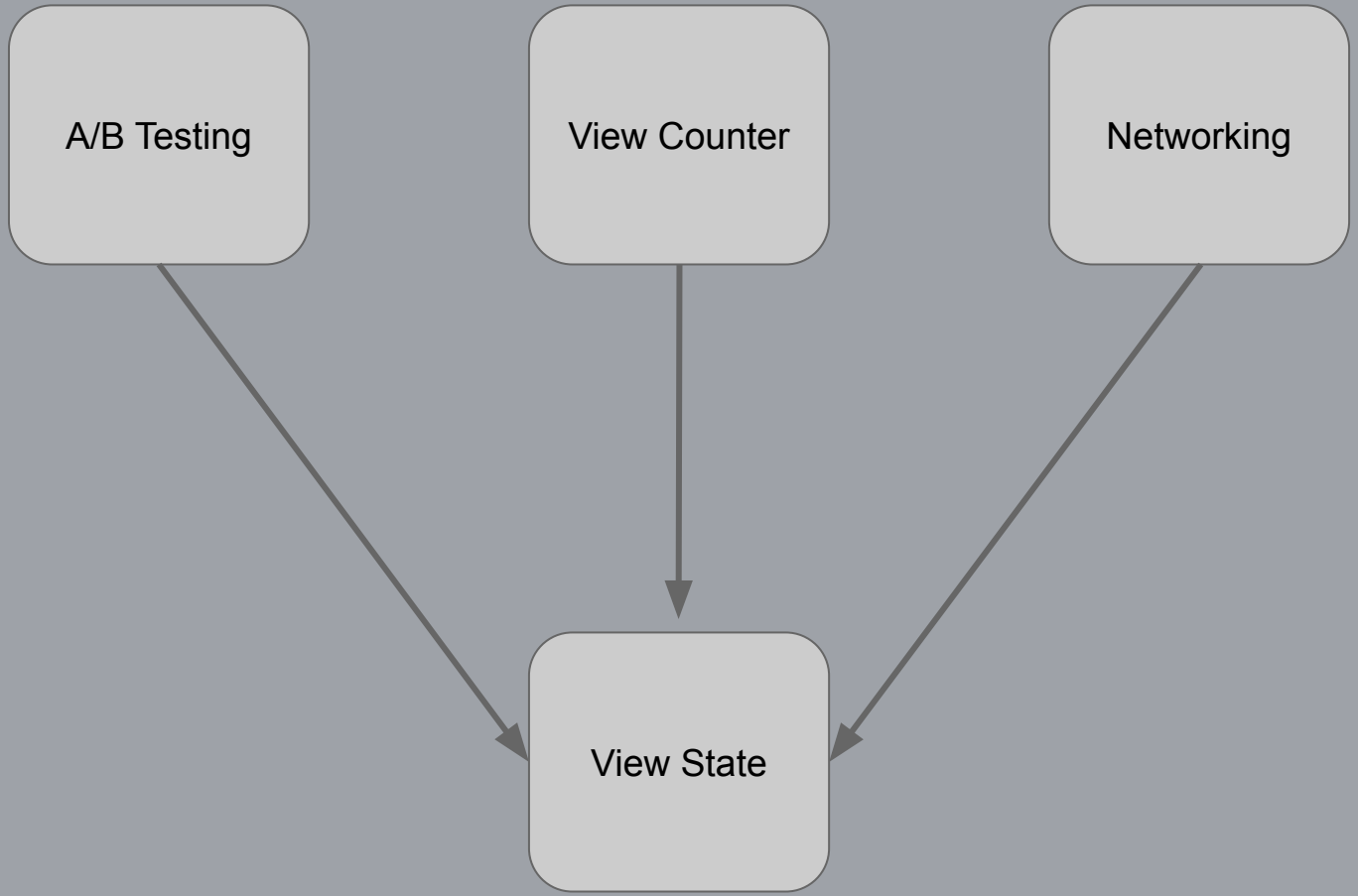
Let’s think about the view state

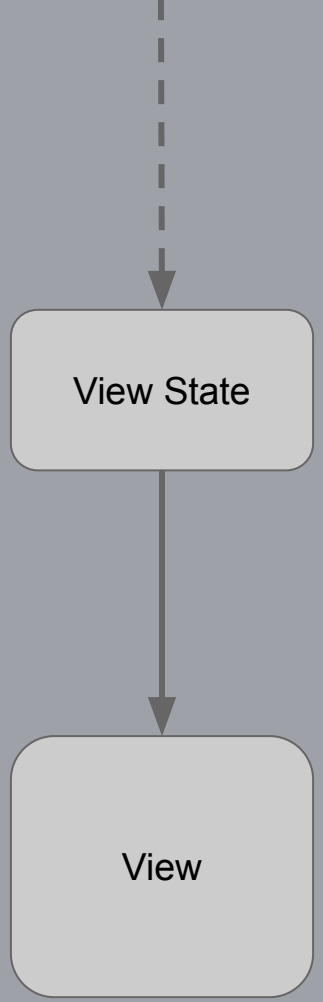


```
struct ArticleListViewState {
  let sections: [ArticleListSection]
}

enum ArticleListRow: Equatable, IdentifiableType {
  case article(article: Article, data: ArticleCell.Data)

  typealias Identity = String
  var identity: String {
    switch self {
    case let .article(article, _):
      return "Article \(article.description))"
    }
  }
}

typealias ArticleListSection = SingleSectionModel<ArticleListRow>
```



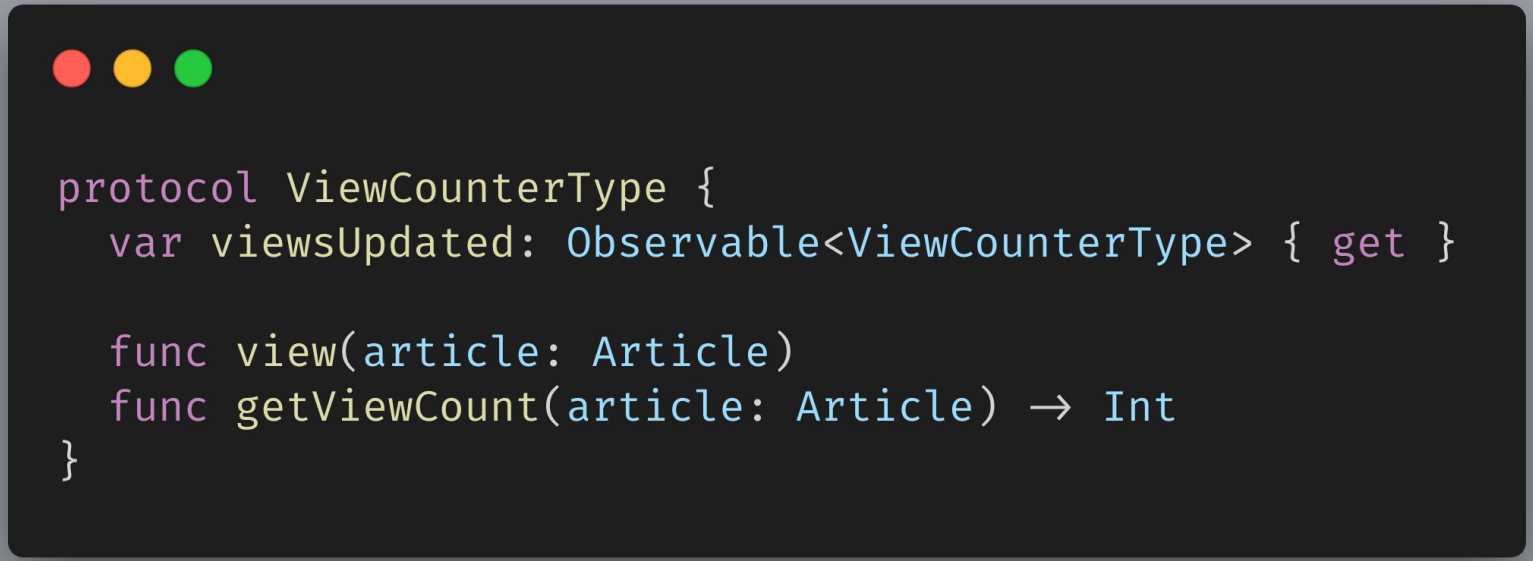
```
struct ArticleListViewState {
  let sections: [ArticleListSection]
}

enum ArticleListRow: Equatable, IdentifiableType {
  case loading
  case error(data: ErrorCell.Data)
  case article(article: Article, data: ArticleCell.Data)
  case articleView(article: Article, data: ArticleViewCell.Data)
}

typealias ArticleListSection = SingleSectionModel<ArticleListRow>
```


A/B Testing
View Counter
Networking
View State

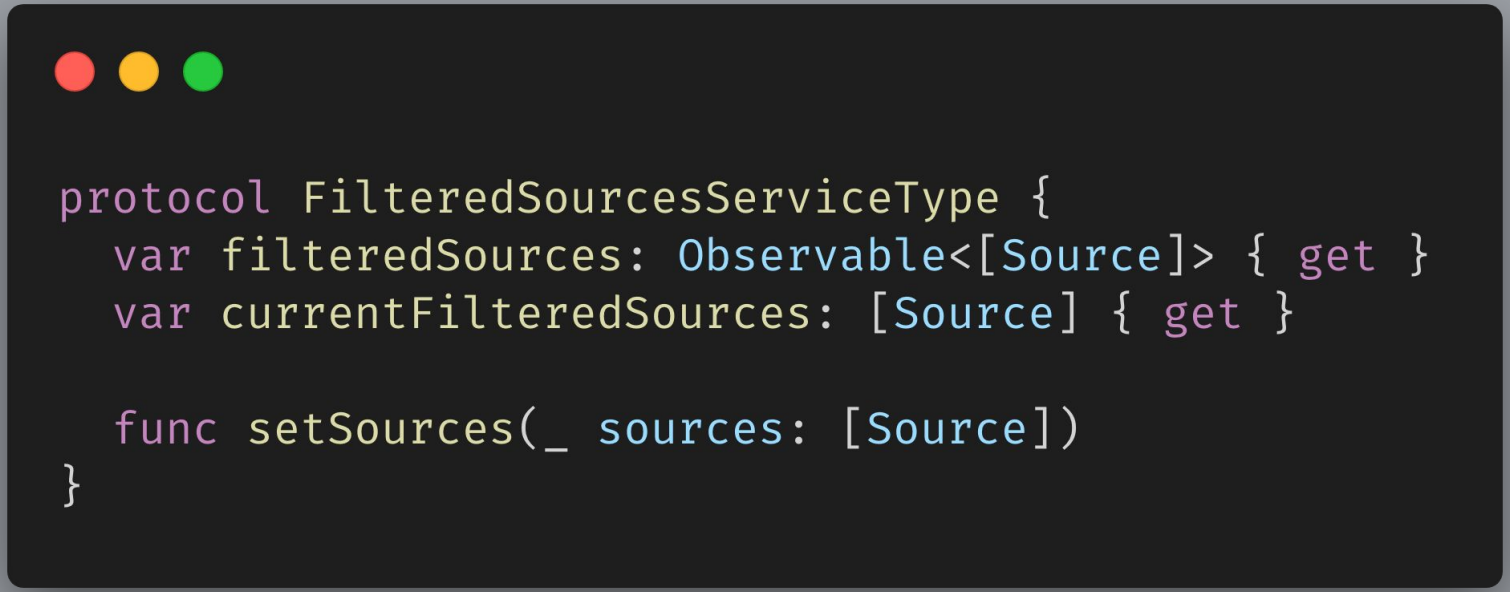

View State
View

Putting it all together (demo time)

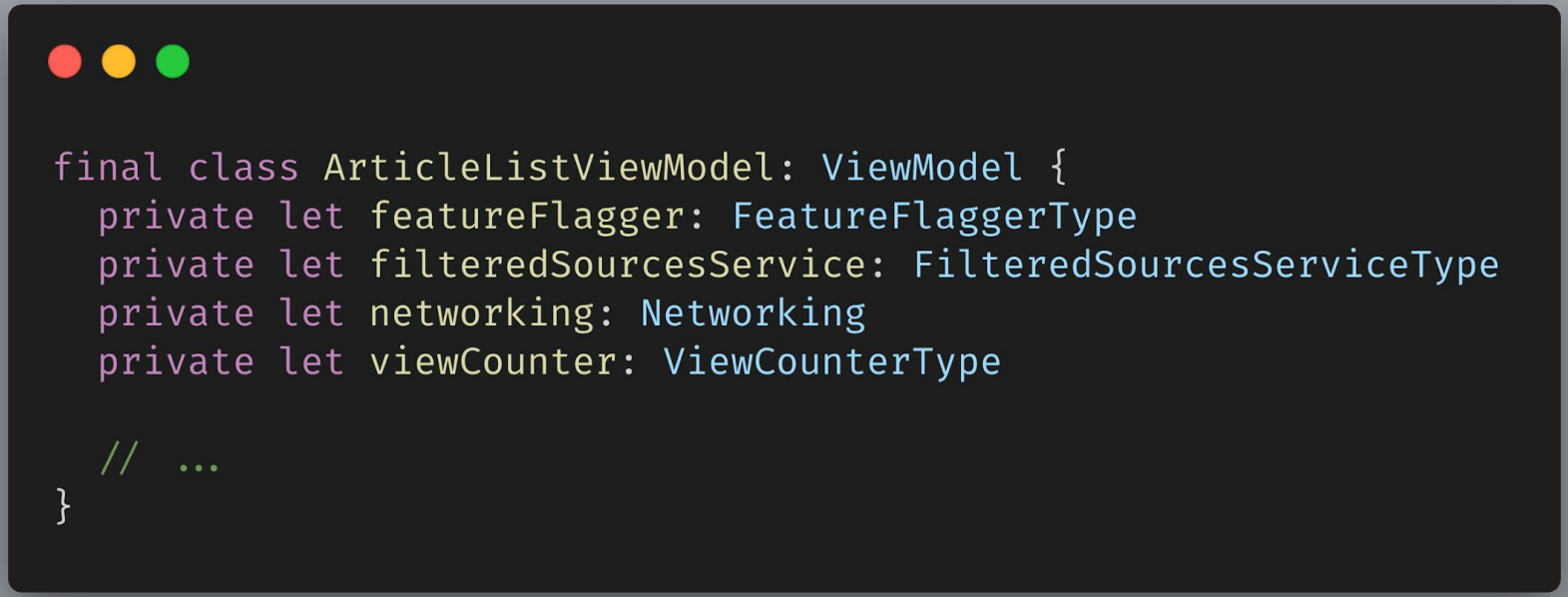
```
protocol FeatureFlaggerType {
  var featureFlags: Observable<[FeatureFlag]> { get }
  var currentFeatureFlags: [FeatureFlag] { get }
}
```

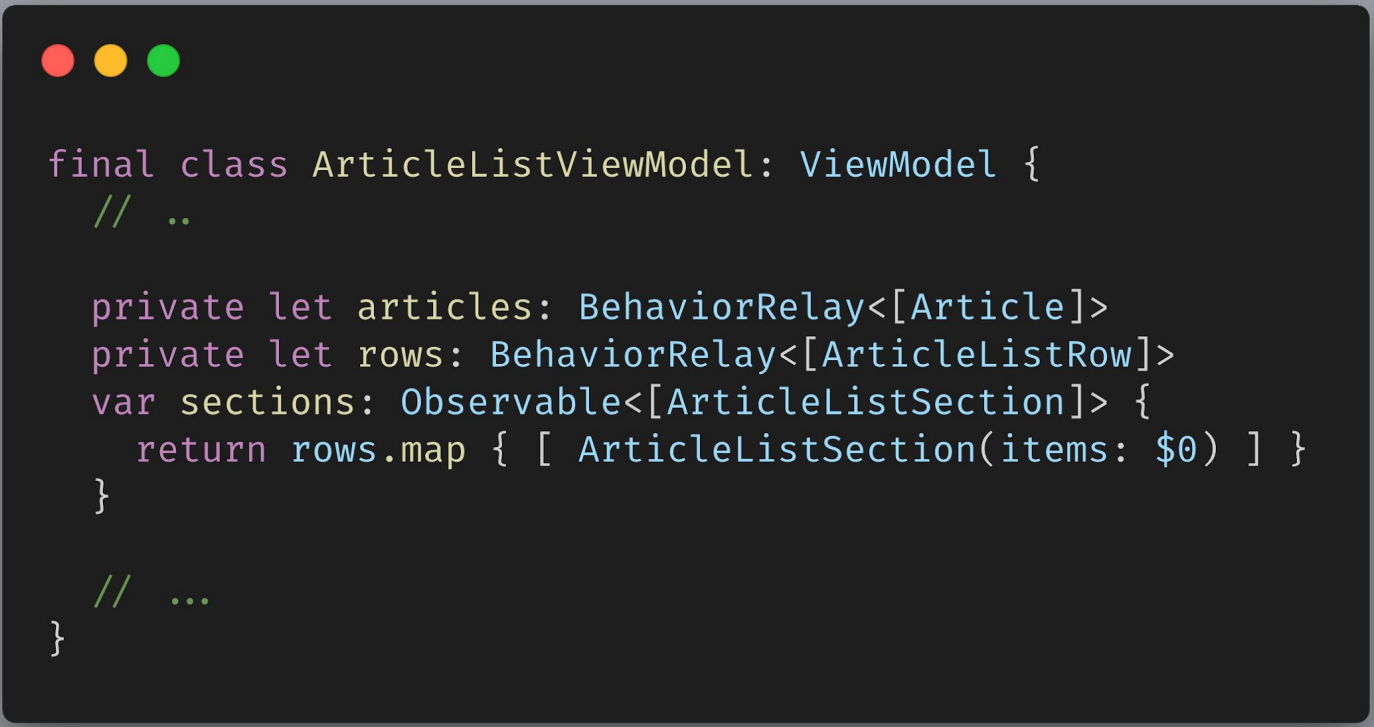


```
protocol ViewCounterType {
  var viewsUpdated: Observable<ViewCounterType> { get }

  func view(article: Article)
  func getViewCount(article: Article) → Int
}
```



```
protocol FilteredSourcesServiceType {
  var filteredSources: Observable<[Source]> { get }
  var currentFilteredSources: [Source] { get }

  func setSources(_ sources: [Source])
}
```


final class ArticleListViewModel: ViewModel {
private let featureFlagger: FeatureFlaggerType
private let filteredSourcesService: FilteredSourcesServiceType
private let networking: Networking
private let viewCounter: ViewCounterType
// ...
}

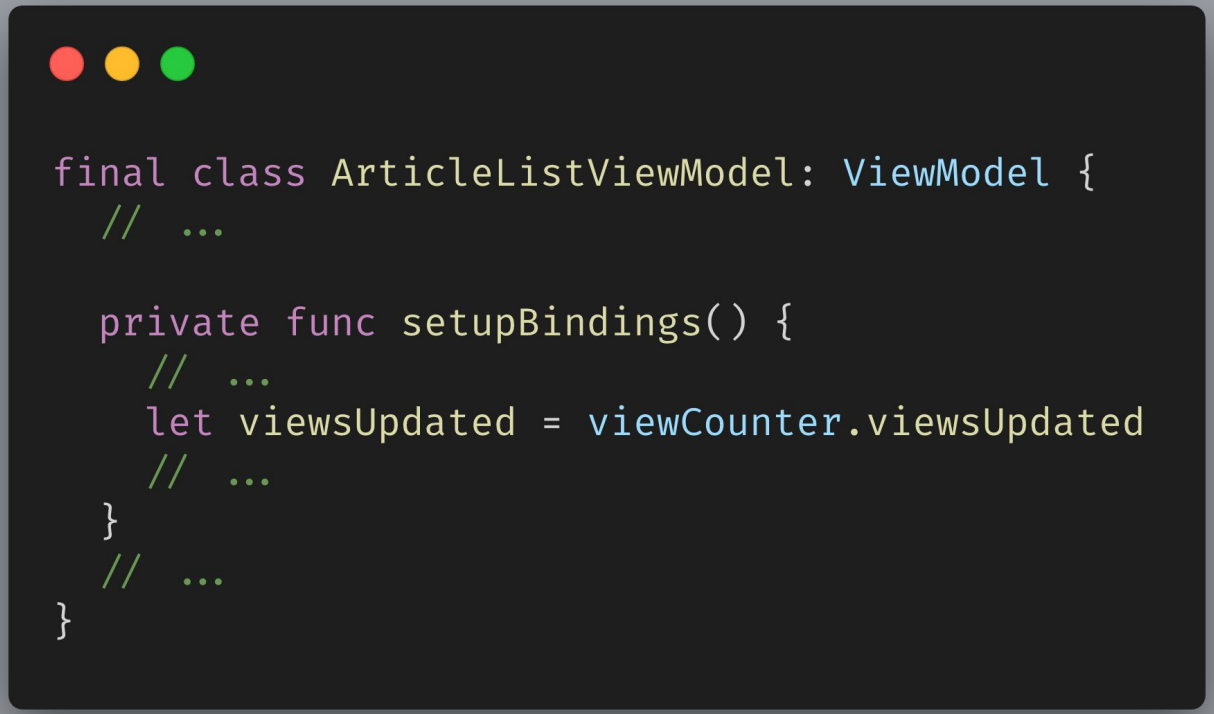

final class ArticleListViewModel: ViewModel {
// ..
private let articles: BehaviorRelay<[Article]>
private let rows: BehaviorRelay<[ArticleListRow]>
var sections: Observable<[ArticleListSection]> {
return rows.map { [ArticleListSection(items: $0)] }
}
// ...
}

```swift
final class ArticleListViewModel: ViewModel {
  // ..

  private func refetchArticles() {
    self.rows.accept([ArticleListRow.loading])

    let request = TopHeadlinesRequest(sources: filteredSourcesService.currentFilteredSources)

    networking
      .request(NewsApi.topHeadlines(request: request))
      .filterSuccessfulStatusCodes()
      .map(TopHeadlinesResponse.self)
      .map { $0.articles }
      .subscribe(onSuccess: { [weak self] articles in
        self?.articles.accept(articles)
      })
      .disposed(by: disposeBag)
  }

  // ...
}
```


final class ArticleListViewModel: ViewModel {
// ..
private func setupBindings() {
}
// ...
}

```swift
final class ArticleListViewModel: ViewModel {
  // ...

  private func setupBindings() {
    // ...
    filteredSourcesService
      .filteredSources
      .subscribe(onNext: { [weak self] _ in
        self?.start()
      })
      .disposed(by: disposeBag)
    // ...
  }
  // ...
}
```

```
final class ArticleListViewModel: ViewModel {
  //  ...

  private func setupBindings() {
    //  ...
    let newCellsEnabled = featureFlagger
      .featureFlags
      .map { $0.enabled(.newDesignedCells) }
    //  ...
  }
  //  ...
}
```


final class ArticleListViewModel: ViewModel {
// ...
private func setupBindings() {
// ...
let viewsUpdated = viewCounter.viewsUpdated
// ...
}
// ...
}

```
final class ArticleListViewModel: ViewModel {
  // ...

  private func setupBindings() {
    // ...
    Observable
      .combineLatest(articles, featureFlagEnabled, viewsUpdated)
      .map { articles, featureFlagEnabled, viewCounter -> [ArticleListRow] in
        if featureFlagEnabled {
          return articles |▷ articleViewAdapter(viewCounter)
        } else {
          return articles |▷ articleAdapter
        }
      }
      .bind(to: rows)
      .disposed(by: disposeBag)
    // ...
  }
  // ...
}
```

Takeaways

Reactive streams and view state helps to layer your application

Streams help you manage data dependencies

View state allows you to describe your UI and ease into reactive patterns

Easier to test

Allows for shrinking your domain

Allows engineers to experiment

I’m still skeptical

The iOS industry is adopting reactive ideas

Combine and Swift UI

Framework

Combine

Customize handling of asynchronous events by combining event-processing operators.

SDKs

iOS 13.0+ Beta

macOS 10.15+ Beta

UIKit for Mac 13.0+ Beta

tvOS 13.0+ Beta

watchOS 6.0+ Beta

On This Page

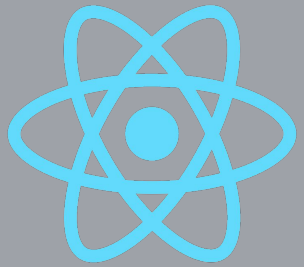
Overview

The Combine framework provides a declarative Swift API for processing values over time. These values can represent many kinds of asynchronous events. Combine declares *publishers* to expose values that can change over time, and *subscribers* to receive those values from the publishers.

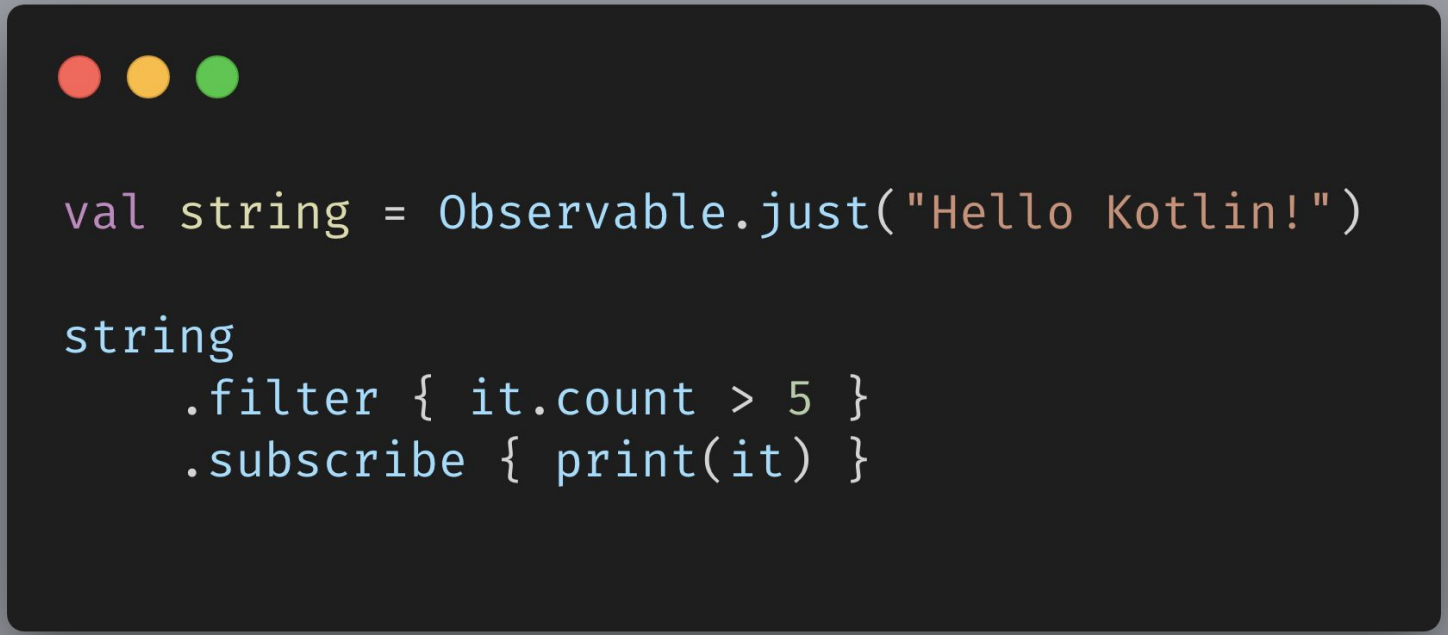
- The `Publisher` protocol declares a type that can deliver a sequence of values over time. Publishers have *operators* to act on the values received from upstream publishers and republish them.
- At the end of a chain of publishers, a `Subscriber` acts on elements as it receives them. Publishers only emit values when explicitly requested to do so by subscribers. This puts your subscriber code in control of how fast it receives events from the publishers it's connected to.

Several Foundation types expose their functionality through publishers, including `Timer`, `NotificationCenter`, and `URLSession`. Combine also provides a built-in publisher for any property that's compliant with Key-Value Observing.

It’s not just iOS



Flutter


val string = Observable.just("Hello Kotlin!")
string
.filter { it.count > 5 }
.subscribe { print(it) }

Demo Redux

Thanks

Source Code

We Are Hiring

Date

July 17th, 2019

Author

James Paolantonio | james.paolantonio@betterment.com
@jpaolantonio

Betterment