SWIFTCRAFT
21-24 May 2024

The Art of Functional State Management

Araks Avoyan

The Art of Functional State Management

by Araks Avoyan

Araks Avoyan

Senior iOS Engineer @Bambuser

ex - Spotify, Fishbrain, Workfront, VOLO

Agenda

- Demystify functional programming and reactive programming in the context of iOS apps
- See how they come together with frameworks like Combine for a practical and powerful solution
- Explore hands-on examples comparing traditional vs. functional state management

Functional Programming Fundamentals

Immutability

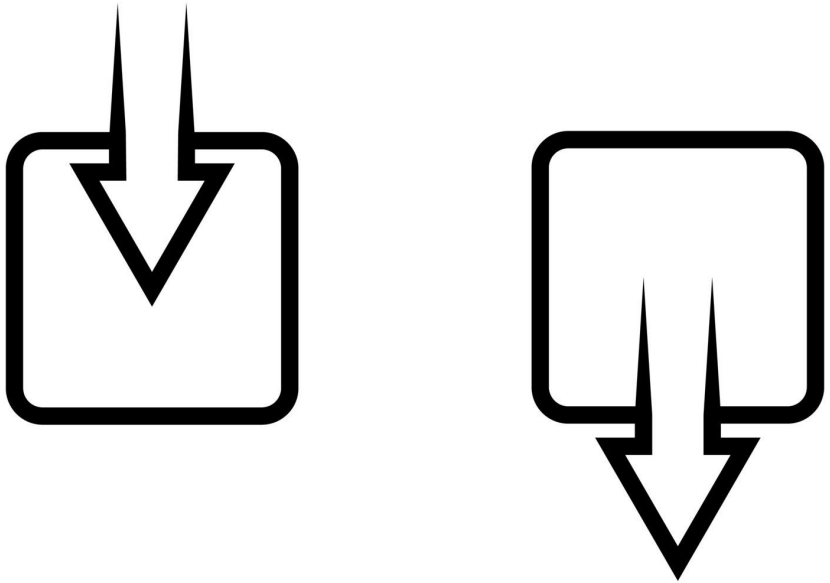
Data is immutable: New state objects for updates

Code snippet

```
// Traditional approach (mutable)
var tasks = [String]()
tasks.append("Buy milk")  // Modifies existing data

// Functional approach (immutable)
func addTask(_ newTask: String) -> [String] {
  return tasks + [newTask]  // Creates a new array with the update
}
```

Pure Functions



Pure functions: Consistent output for same input, no side effects

Code snippet

```
// Traditional approach (not pure)
func updateTask(at index: Int, with newTask: String) {
  tasks[index] = newTask  // Side effect: modifies global state
}

// Functional approach (pure)
func updateTask(at index: Int, with newTask: String, tasks: [String]) -> [String] {
  var updatedTasks = tasks
  updatedTasks[index] = newTask
  return updatedTasks  // Returns a new modified array
}
```

Benefits of Functional Programming

- Reduced risk of bugs: Immutability prevents accidental mutations.
- Easier reasoning about code: Pure functions make behavior predictable.
- Improved testability: Isolated and predictable functions are easier to test.

Cons of Functional Programming

- Learning Curve: Initial effort to grasp functional concepts.
- Potential Verbosity: Complex logic might require more lines of code.

Reactive Programming for iOS

Reactive Programming: Streams for Responsive UIs

Imagine Data as a Flowing River

Key principles

- Declarative approach: Describe how UI reacts to data changes.
- Data as streams: Treat data as continuous streams of events.
- Event-driven: React to events (changes in data streams).

Declarative approach

Instead of writing step-by-step instructions on how to update the UI, you describe how the UI should respond to changes in the data. This makes the code more concise and easier to reason about.

Data as Streams

Reactive programming treats data as continuous streams of events. This allows for efficient processing and updates, as you only need to react to the actual changes in the data, not the entire data set every time

Event-Driven

Unlike traditional approaches that rely on polling for updates, reactive programming is event-driven. Your app reacts to specific events (changes in data streams) triggered by user interactions, network requests, or other events. This leads to a more responsive and efficient user experience.

Example

User taps a button (event) -> triggers network request (event) -> data stream updates UI (reactive response).

Benefits of Reactive Programming

- Declarative and Event-Driven: Easier to manage UI updates.
- Efficient Data Processing: Streamlined transformations and updates.
- Improved Responsiveness: Apps react to events for a better user experience.

Cons of Reactive Programming

- Steeper Learning Curve: Concepts like publishers, subscribers, and operators require initial learning.
- Debugging Challenges: Tracing issues within data streams can be complex.

Combine: Functional State Management in Action

Combine: Reactive Power for State Management

What is Combine?

- Apple's reactive programming framework for iOS
- Data streams (publishers) emit values over time
- Subscribers react to changes in data streams

Core Functionalities

- Publishers: Define data streams emitting values over time.
- Subscribers: React to changes in data streams.
- Operators: Transform and manipulate data streams (map, filter, etc.).

Code snippet

```
// Simplified example (Combine)
let numberSubject = CurrentValueSubject<Int>(0)

numberSubject
  .map { String($0) }  // Transform number to string for display
  .assign(to: \.text, on: someLabel)  // Bind label's text to the data stream

numberSubject.send(10)  // Emit a new value (state change)
```

Async Operations & Error Handling

- Handle asynchronous operations (network requests, user interactions).
- Built-in error handling for robust apps.

Benefits of using Combine

- Declarative UI updates: Automatic updates on state changes.
- Streamlined asynchronous operations: Network requests and interactions in state management.
- Built-in error handling and cancellation: Robust apps.

Cons of Combine

- Newer Framework: Limited resources or libraries compared to established approaches

Hands-on example: Counter app

Traditional MVVM Approach

```
// ViewModel (MVVM)
class CounterViewModel {
  var count: Int = 0

  func increment() {
    count += 1
  }

  func decrement() {
    count -= 1
  }
}
```

Traditional MVVM Approach

```
// View (MVVM)
class CounterViewController: UIViewController {
  @IBOutlet weak var counterLabel: UILabel!

  private var viewModel = CounterViewModel()

  override func viewDidLoad() {
    super.viewDidLoad()
    counterLabel.text = String(viewModel.count)
  }

  @IBAction func incrementButtonTapped(_ sender: Any) {
    viewModel.increment()
    counterLabel.text = String(viewModel.count)
  }

  @IBAction func decrementButtonTapped(_ sender: Any) {
    viewModel.decrement()
    counterLabel.text = String(viewModel.count)
  }
}
```

Combine Approach

```
// ViewModel (Combine)
class CounterViewModel {
  @Published private(set) var count: Int = 0

  func increment() {
    count += 1
  }

  func decrement() {
    count -= 1
  }
}
```

Combine Approach

```swift
// View (Combine)
class CounterViewController: UIViewController {
  @IBOutlet weak var counterLabel: UILabel!

  private var viewModel = CounterViewModel()

  override func viewDidLoad() {
    super.viewDidLoad()
    viewModel.$count
      .map { String($0) }
      .assign(to: \.text, on: counterLabel)
  }

  @IBAction func incrementButtonTapped(_ sender: Any) {
    viewModel.increment()
  }

  @IBAction func decrementButtonTapped(_ sender: Any) {
    viewModel.decrement()
  }
}
```

Hands-on example: User Profile with Network Request

Traditional MVVM Approach

```swift
// NetworkService (MVVM)
protocol NetworkService {
  func fetchUser(completion: @escaping (User?, Error?) -> Void)
}

// User (Model)
struct User {
  let firstName: String
  let lastName: String
}
```

Traditional MVVM Approach

```
// UserViewModel (MVVM)
class UserViewModel {
  var firstName: String?
  var lastName: String?

  private let networkService: NetworkService

  init(networkService: NetworkService) {
    self.networkService = networkService
  }

  func fetchUserProfile() {
    networkService.fetchUser { [weak self] user, error in
      guard let self = self, let user = user else { return }
      self.firstName = user.firstName
      self.lastName = user.lastName
      // Manually notify view about changes (e.g., delegate or closure)
    }
  }
}
```

Combine Approach

```
// NetworkService (Combine)
protocol NetworkService {
  func fetchUser() -> AnyPublisher<UserData, Error>  // Returns a publisher of UserData or
Error
}

// User (Model)
struct User {
  let firstName: String
  let lastName: String
}

// UserData (Model for Network Response)
struct UserData: Decodable {
  let firstName: String
  let lastName: String
}
```

Combine Approach

```
// UserViewModel (Combine)
class UserViewModel {
  @Published var user: User? = nil

  private let networkService: NetworkService

  init(networkService: NetworkService) {
    self.networkService = networkService

    // Trigger network request on initialization and handle success/failure
    networkService.fetchUser()
      .map { User(firstName: $0.firstName, lastName: $0.lastName) }  // Transform response to
User object
      .sink(receiveCompletion: { [weak self] completion in
        if case .failure(let error) = completion {
          // Handle errors (show alert, etc.)
          print(error.localizedDescription)
        }
      }, receiveValue: { [weak self] user in
        self?.user = user
      })
      .store(in: &cancellables)  // Manage subscriptions for memory leaks
  }

  private var cancellables = Set<AnyCancellable>()
}
```

Key Takeaways: Functional & Reactive State Management

- Functional programming benefits: predictable, maintainable code
- Reactive programming benefits: declarative UI updates, efficient data handling
- Combine: a powerful tool for functional and reactive state management

Explore Further & Experiment!

- Experiment with functional programming and reactive programming in your projects
- Consider alternatives
- Many online resources available for further learning
- Balance efficiency with simplicity
- Enjoy the process!


PIVOT!!!

Thank you!

If you have questions, feel free to reach out!