



Mash-Up
iOS Team

2024.06.11



가자! 빵집으로~ 🥖

Vanilla SwiftUI vs TCA



Index

1

Vanilla SwiftUI vs TCA

2

Stack-based vs. Tree-based Navigation

3

프로젝트 시연

Vanilla SwiftUI vs TCA

Vanilla SwiftUI vs TCA

Vanilla SwiftUI... 무엇이 문제인가

-프로퍼티 래퍼의 장점

 바인딩을 간편하게 구현 가능

 즉시 UI에 반영

Vanilla SwiftUI vs TCA


Vanilla SwiftUI... 무엇이 문제인가

-데이터 흐름의 방향

@State, @ObservedObject와 같은 프로퍼티 래퍼를 통해
∞ 양방향 바인딩

Vanilla SwiftUI... 무엇이 문제인가

-데이터 흐름의 방향

 Source of Truth를 지킨다고 해도 데이터가 언제 어떻게 데이터가 변경될 지 모른다는 문제가 존재

 데이터 변경에 따른 Side Effect 관리가 어려움

 테스트 어렵다~

Vanilla SwiftUI vs TCA

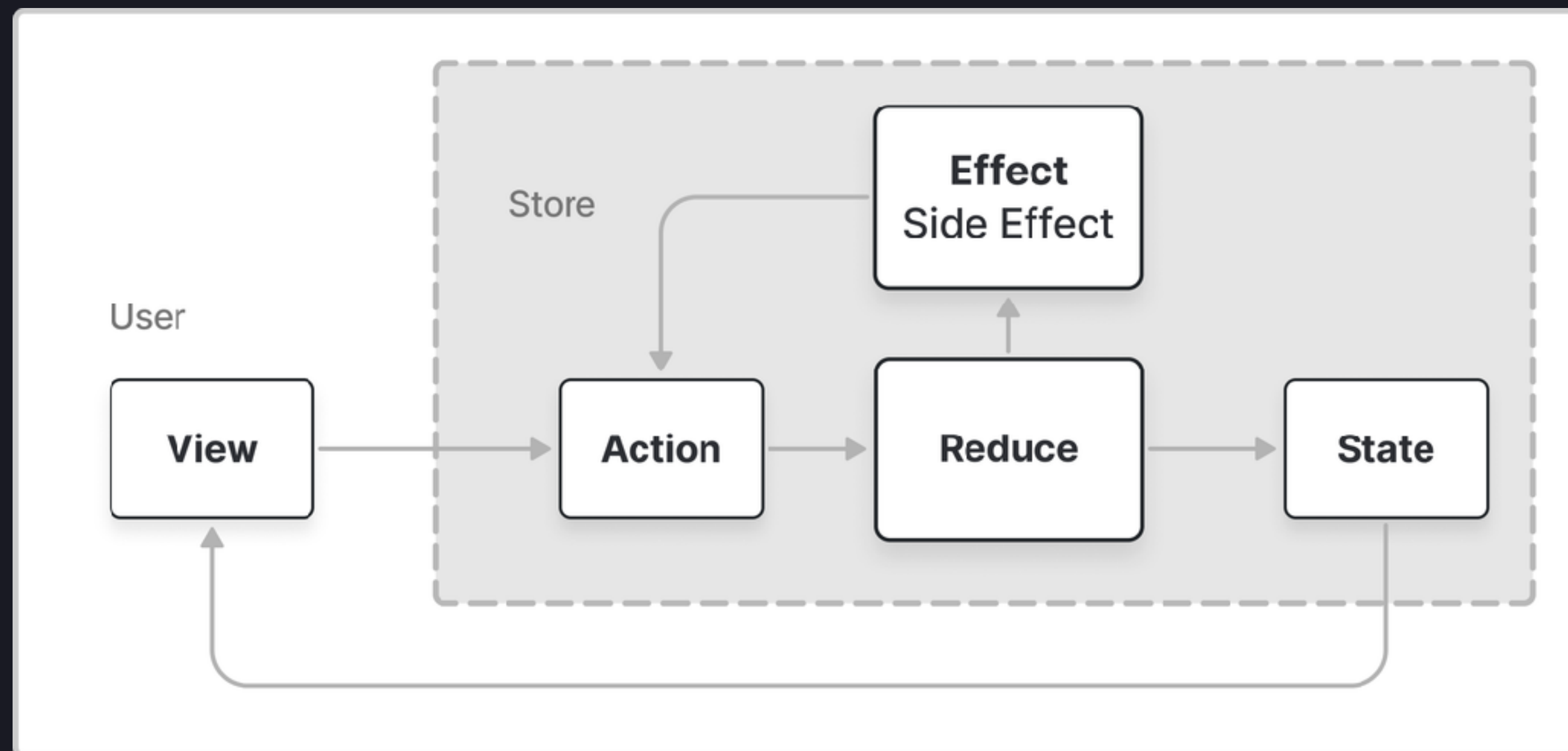
TCA에서는...

TCA에서 State는 값 타입(struct)으로 관리됨
→ State의 변화에 따른 side-effect 걱정이 없다

👁️ Reducer를 담는 store는 참조타입이지만..

TCA로 해결해보자

-Single Direction Source of Truth

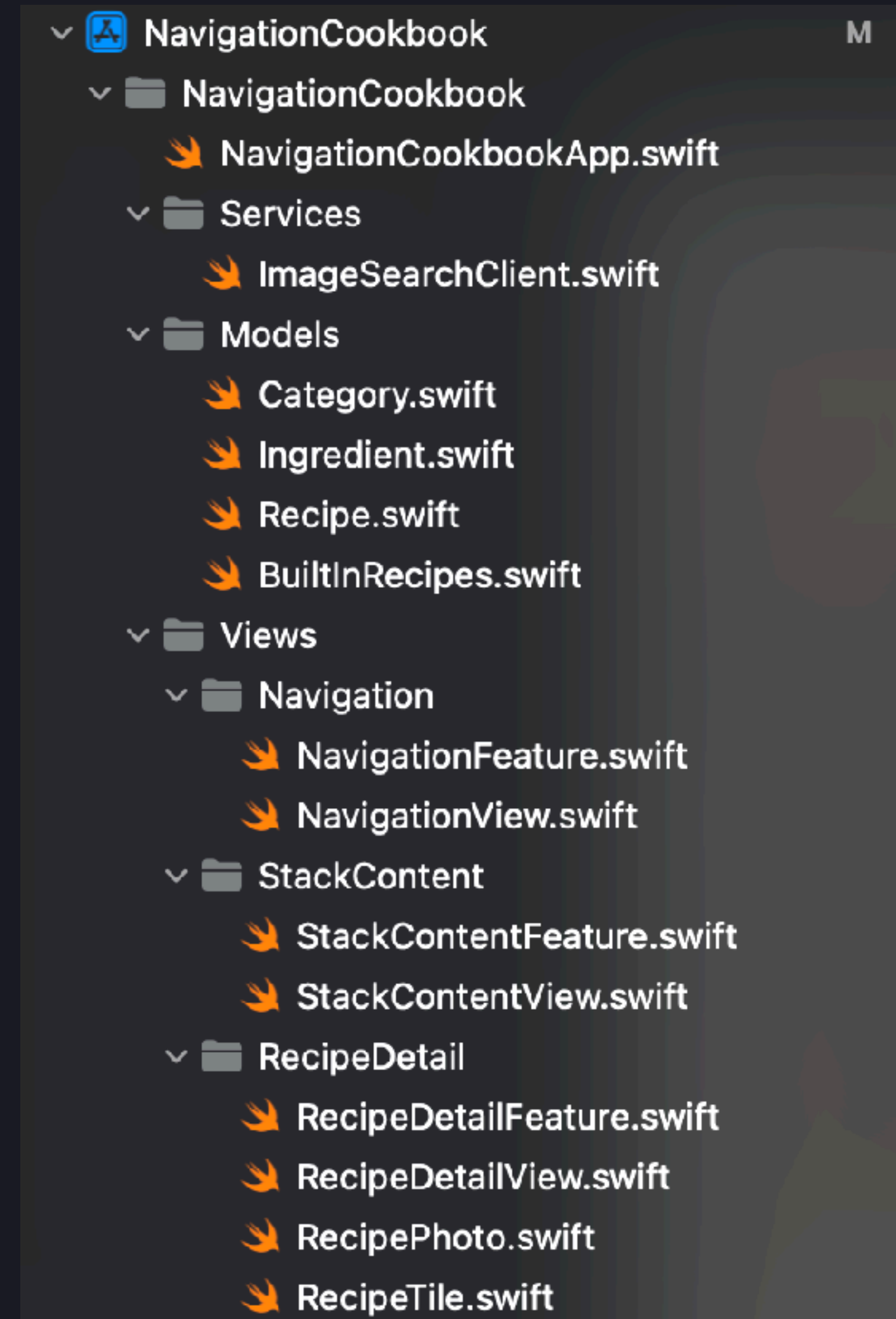
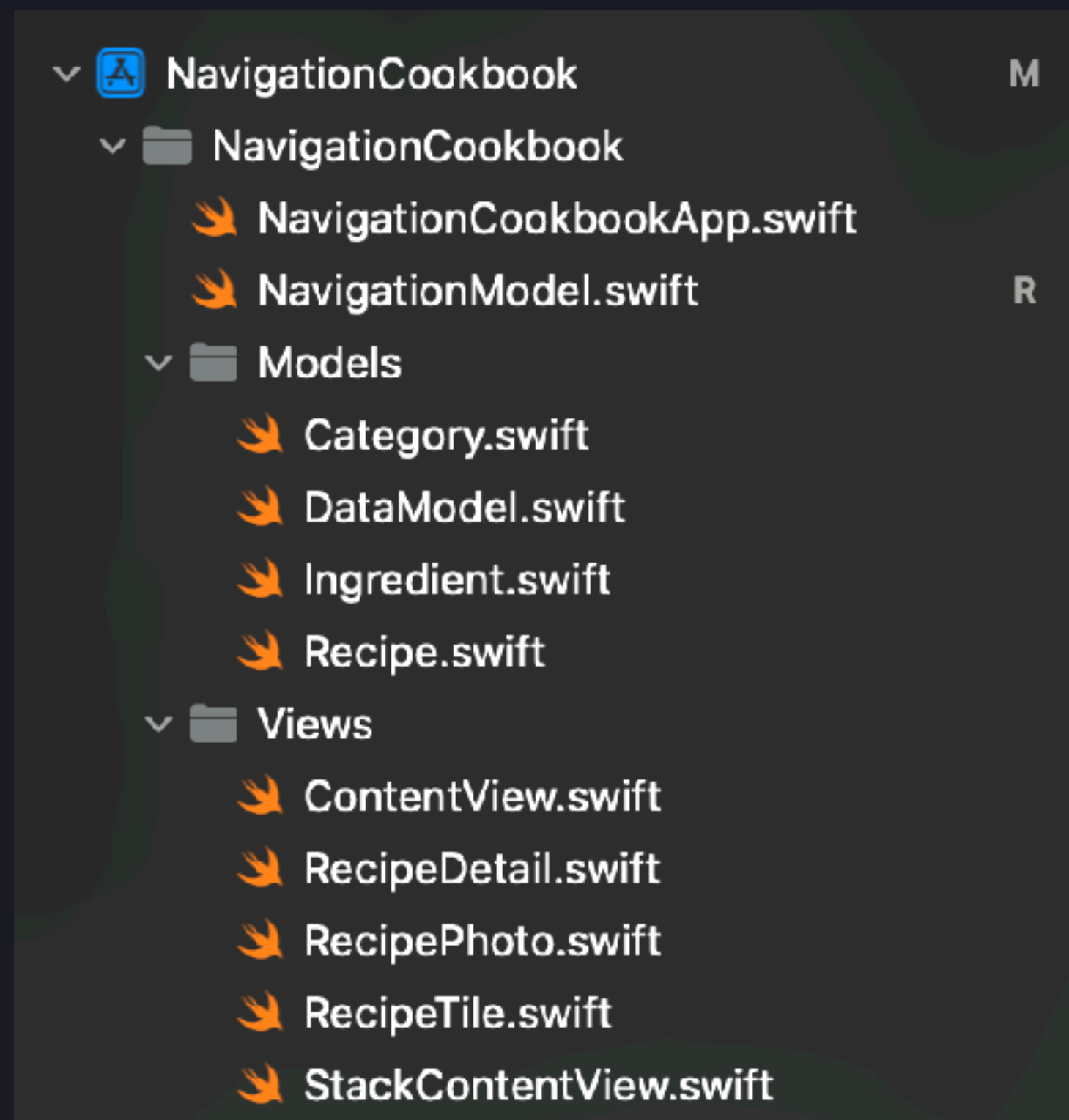


State를 변경하는 유일한 방법은
Store에 Action을 보내는 것

Vanilla SwiftUI vs TCA

하지만 TCA도...

-불가피한 코드 복잡성 증가



Vanilla SwiftUI... 무엇이 문제인가

- Navigation

```
import SwiftUI

struct StackContentView: View {
    @EnvironmentObject private var navigationModel: NavigationModel
    var dataModel = DataModel.shared

    var body: some View {
        NavigationStack(path: $navigationModel.recipePath) {
            List(Category.allCases) { category in
                Section {
                    ForEach(dataModel.recipes(in: category)) { recipe in
                        NavigationLink(recipe.name, value: recipe)
                    }
                } header: {
                    Text(category.localizedName)
                }
            }
        }
        .navigationTitle("Categories")
        .navigationDestination(for: Recipe.self) { recipe in
            RecipeDetail(recipe: recipe) { relatedRecipe in
                NavigationLink(value: relatedRecipe) {
                    RecipeTile(recipe: relatedRecipe)
                }
            }
            .buttonStyle(.plain)
        }
    }
}
```

NavigationLink를 사용하면

🚨 데이터 컬렉션에 대한 바인딩을
사용해야 함

🚨 NavigationLayer가 연결되는
모든 객체를 알고 있게 됨

→ 모듈화에 방해가 됨

TCA에서는

- Navigation

State에만 접근 가능하면 된다!

→ Action, Reducer, Dependency,
View, Feature 등은 필요하지 않음

```
struct StackContentView: View {
    var store: StoreOf<StackContentFeature>

    var body: some View {
        List(Category.allCases) { category in
            Section {
                ForEach(getRecipes(recipes: store.recipes, for: category)) { recipe in
                    NavigationLink(recipe.name, state: NavigationFeature.Path.State.recipeDetail(.init(recipe: recipe)))
                }
            } header: {
                Text(category.localizedName)
            }
        }
        .navigationTitle("Categories")
    }
}
```

Vanilla SwiftUI vs TCA

TCA에서는

- Navigation

따라서 Stack에 있는 모든 Feature의
State struct만 자체 모듈로 추출하면 컴파일 가능

→ 서로 결합되지 않고 연결 가능함

Stack-based vs. Tree-based Navigation

Vanilla SwiftUI vs TCA

Stack-based vs Tree-based Navigation

스택 전체에 있는 기능들을 단일 플랫폼 값 배열로 모델링

Vanilla SwiftUI vs TCA

Stack-based vs Tree-based Navigation

배열에 item 추가 → **Stack**에 push
배열에서 item 삭제 → **Stack**에서 pop

Drill-down 방식에 주로 사용됨

Vanilla SwiftUI vs TCA

Stack-based vs Tree-based Navigation

앱의 Root에서 플랫 배열을 사용하여
이미 스택에 푸시된 상태로 앱을 시작하도록
진입점을 변경할 수 있음

Stack-based vs Tree-based Navigation

```
struct RootFeature: Reducer {  
    struct State {  
        var path = StackState<Path.State>()  
        /* code */  
    }  
    enum Action {  
        case path(StackAction<Path.State, Path.Action>)  
        /* code */  
    }  
}
```

```
NavigationView(  
    store: Store(  
        initialState: NavigationFeature.State(  
            path: StackState([]),  
            stackContent: StackContentFeature.State()  
        )  
    ) {  
        NavigationFeature()  
    }  
)
```

Navigation을 관리하는 Feature의 State, Action 내부에
StackState, StackAction을 포함

ex. StackState에 path가 쌓일 빈 배열 넣어주면서 시작

Stack-based vs Tree-based Navigation

```
struct Path: Reducer {
    enum State {
        case addItem(AddFeature.State)
        case detailItem(DetailFeature.State)
        case editItem(EditFeature.State)
    }
    enum Action {
        case addItem(AddFeature.Action)
        case detailItem(DetailFeature.Action)
        case editItem(EditFeature.Action)
    }
    var body: some ReducerOf<Self> {
        Scope(state: /State.addItem, action: /Action.addItem) {
            AddFeature()
        }
        Scope(state: /State.editItem, action: /Action.editItem) {
            EditFeature()
        }
        Scope(state: /State.detailItem, action: /Action.detailItem) {
            DetailFeature()
        }
    }
}
```

스택에 추가될 수 있는 모든 기능들의 도메인을 포함한 Path Reducer 생성

Stack-based vs Tree-based Navigation

```
struct RootFeature: Reducer {
    /* code */

    var body: some ReducerOf<Self> {
        Reduce { state, action in
            // Core logic for root feature
        }
        .forEach(\.path, action: /Action.path) {
            Path()
        }
    }
}
```

forEach 메서드로 부모 도메인 & Navigation 도메인을 결합

Stack-based vs Tree-based Navigation

```
struct RootView: View {
    let store: StoreOf<RootFeature>

    var body: some View {
        NavigationStackStore(
            path: self.store.scope(state: \.path, action: { .path($0) })
        ) {
            // Root view of the navigation stack
        } destination: { state in
            switch state {
            case .addItem:
            case .detailItem:
            case .editItem:
            }
        }
    }
}
```

도메인 내의 StackStore에 새로운 데이터를 푸시할 수 있음

Stack-based vs Tree-based Navigation

장점

복잡하고 재귀적인 경로를 처리할 수 있음

→ Drill-down되는 앱의 스택을 쉽게 관리할 수 있음

```
path: [  
    .detail(StandupDetailModel(standup: .designMock)),  
    .record(RecordMeetingModel(standup: .designMock)),  
    .detail(StandupDetailModel(standup: .designMock)),  
    .record(RecordMeetingModel(standup: .designMock)),  
    .detail(StandupDetailModel(standup: .designMock)),  
    .record(RecordMeetingModel(standup: .designMock)),  
],
```

Vanilla SwiftUI vs TCA

Stack-based vs Tree-based Navigation

장점

👍 Tree 기반 Navigation 보다 버그가 적음

일반적으로 화면 전체 스택을 복원할 수 있음

Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

각 Feature는 탐색 가능한 모든 위치들에 대한
분기 지점 역할

→ 트리와 같은 구조를 형성하기 때문에
트리 기반 탐색이라고 부름

Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

유효한 탐색 경로를
State의 존재 유무에 의해 제어

- ✓ nil: not navigated to a feature
- ✓ non-nil: active navigation

Stack-based vs **Tree-based** Navigation

```
struct State {  
    @PresentationState var detailItem: DetailFeature.State?  
    @PresentationState var editItem: EditFeature.State?  
    @PresentationState var addItem: AddFeature.State?  
    /* code */  
}
```

옵셔널 상태로 관리할 경우,
두 개 이상의 상태가 동시에 nil이 아닌 것과 같은
유효하지 않은 상태가 발생할 수 있음
→ 유효한 경우 : 전부 다 nil / 한개만 non-nil



[아오스] 양준혁 오늘 오전 12:08

nil nil nil
non-nil nil nil
nil non-nil nil
nil nil non-nil
non-nil non-nil nil
non-nil nil non-nil
nil non-nil non-nil
non-nil non-nil non-nil

Stack-based vs **Tree-based** Navigation

```
struct Destination: Reducer {
    enum State {
        case addItem(AddFeature.State)
        case detailItem(DetailFeature.State)
        case editItem(EditFeature.State)
    }
    enum Action {
        case addItem(AddFeature.Action)
        case detailItem(DetailFeature.Action)
        case editItem(EditFeature.Action)
    }
    var body: some ReducerOf<Self> {
        Scope(state: /State.addItem, action: /Action.addItem) {
            AddFeature()
        }
        Scope(state: /State.editItem, action: /Action.editItem) {
            EditFeature()
        }
        Scope(state: /State.detailItem, action: /Action.detailItem) {
            DetailFeature()
        }
    }
}
```

nil값 체크 + 관리의 용이성
🌻 enum 화면 케이스 관리

→ 경로 수를 쉽게 관리할 수 있음
→ 모듈을 독립적으로 관리할 수 있음 👍

Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

```
struct Destination: Reducer {  
    enum State {  
        case addItem(AddFeature.State)  
        case detailItem(DetailFeature.State)  
        case editItem(EditFeature.State)  
    }  
    enum Action {  
        case addItem(AddFeature.Action)  
        case detailItem(DetailFeature.Action)  
        case editItem(EditFeature.Action)  
    }  
    var body: some ReducerOf<Self> {  
        Scope(state: /State.addItem, action: /Action.addItem) {  
            AddFeature()  
        }  
        Scope(state: /State.editItem, action: /Action.editItem) {  
            EditFeature()  
        }  
        Scope(state: /State.detailItem, action: /Action.detailItem) {  
            DetailFeature()  
        }  
    }  
}
```

Destination Reducer로
한번 감쌘

Stack-based vs **Tree-based** Navigation

```
struct InventoryFeature: Reducer {
    struct State {
        @PresentationState var destination: Destination.State?
        /* code */
    }
    enum Action {
        case destination(PresentationAction<Destination.Action>)
        /* code */
    }

    /* code */
}
```

destination 하나로 관리할 수 있음
→ 이를 사용하면 응용 프로그램이 시작될 때
딥링크 위치를 지정할 수 있음

Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

장점

👍 앱이 지원하는 Navigation 경로를 정적으로 정의할 수 있음

👍 기능별로 독립적으로 실행할 수 있음

> 프리뷰로 해당 뷰 내에서 이동할 수 있는 전체적인 흐름을 테스트 해볼 수도 있음

Stack-based vs **Tree-based** Navigation

장점

여러 API를 동일한 형태로 사용할 수 있음

```
.navigationDestination(  
    unwrapping: self.$model.destination,  
    case: /StandupDetailModel.Destination.meeting  
) { $meeting in  
    ...  
}  
  
.navigationDestination(  
    unwrapping: self.$model.destination,  
    case: /StandupDetailModel.Destination.record  
) { $model in  
    ...  
}
```

```
.alert(  
    unwrapping: self.$model.destination,  
    case: /StandupDetailModel.Destination.alert  
) { action in  
    ...  
}  
  
.sheet(  
    unwrapping: self.$model.destination,  
    case: /StandupDetailModel.Destination.edit  
) { $editModel in  
    ...  
}
```


Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

단점

재귀 경로에 대한 사용 사례를 분석하기 어려움

😊 재귀 enum을 제공하던가...

여러분이 가꾸어 나가는 **지식의 나무**

나무위키에 오신 것을 환영합니다!
다크는 **화색**이 아니라 **검정**입니다!
나무위키는 누구나 기여할 수 있는 위키입니다.
검증되지 않았거나 편향된 내용이 있을 수 있습니다.

?	나무위키에 처음 오셨나요? 먼저 나무위키의 규정 과 CCL 위반 등 자주 하는 실수, 도움말 을 확인해 보세요.
🗨️	나무위키 게시판 공지 · 그루터기 · 문의 · 신고 · 편집 요청 · 차단 소명/해제 요청 · 다중 계정 검사 · 토론 문의 · 개발 문의
🔍	권리침해 문의 권리자의 임시조치 및 저작권 침해 관련 문의 방법이 권리침해 도움말 에 설명되어 있습니다.
⚖️	중재 제도 토론 중재 요청을 받고 있습니다. 게시판 에서 요청하실 수 있습니다.
🛡️	운영진 지원 나무위키에서 운영진 을 상시 모집합니다. 여기 에서 지원하실 수 있습니다. [지원 전 참고 사항 · 펼치기 · 접기]

[↑](#)
[↓](#)

Vanilla SwiftUI vs TCA

Stack-based vs **Tree-based** Navigation

단점

Tree기반 Navigation을 잘못 구현하여 기능들이 결합된 경우
한 기능을 빌드하려면 다른 기능도 빌드해야하는 경우가 발생할 수 있음

leaf 기능들의 경우 이러한 결합이 생겨도
다른 종속성이 많지 않기 때문에 무관하지만



root 기능이 결합되면 컴파일 시간이 길어질 수 있음



프로젝트 시연





Mash-Up
iOS Team

2024.06.11

Thank you.

