Smart-Commerce

제 1 장 서 론

지난 프로젝트에서 Escaping Closure, Notification Center를 구현하면서 비동기 통신 과정을 명확하게 추론하는 것에 한계를 느꼈다. 또한 UI 구현 코드와 데이터 통신 코드가 무분별하게 섞여 가독성 떨어지는 문제점을 파악했다. 이를 보완하고자 RxSwift와 MVVM 디자인 패턴을 학습했으며, 본 프로젝트는 학습한 기술을 적용하여 문제점을 개선하는데 첫 번째 목적을 둔다.

또한 현재 애플리케이션 시장에서 전반적으로 사용되는 UI를 벤치마킹하여 고객 친화적인 인터페이스에 대한 견문을 넓히는데 두 번째 목적을 둔다.

제 2 장 네트워크

본 프로젝트에서 서버로부터 JSON Data를 가져와 View에 로드하는 일련의 과정은 아래 양식을 크게 벗어나지 않는다. 따라서 위 글에서는 HomeViewController.CategoryCollectionView의 Cell에 데이터가 로드되는 과정을 예시로 든다.

2.1 구현 순서

- View에 로드할 Data 구조체를 정의
- 서버로부터 가져온 JSON Data를 Decoding할 ResponseDTO 구조체 정의
- API 스펙 정의, 정의한 API를 통해 URLSession으로 HTTP 통신을 할 수 있는 함수를 선언

2.2 View에 로드할 Data 구조체를 정의한다.

아래 이미지는 본 프로젝트의 RecommendViewController.CategoryCollectionView의 화면이다. UICollectionView 각각의 Cell들을 살펴보면 썸네일을 담당하는 UIImageView와 제목을

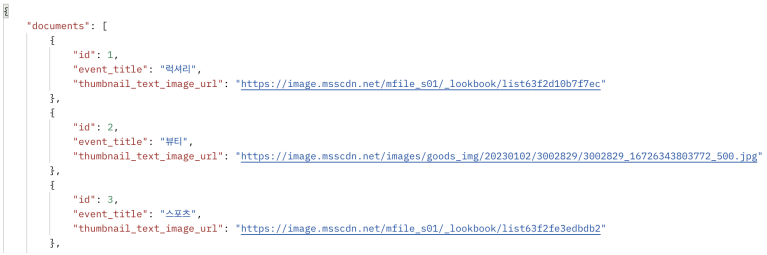
담당하는 UILabel로 구성된 것을 확인할 수 있다.

Cell들에게 일관된 데이터를 전달하기 위해 형식에 맞는 EventCategoryCellData 구조체를 정의한다.

```
struct EventCategoryCellData {
    let title: String?
    let thumbnailURL: URL?
}
```

2.3 서버로부터 가져온 JSON Data를 Decoding할 ResponseDTO 구조체를 정의한다.

아래 이미지는 서버에서 제공하는 JSON 양식이다. documents 객체에 각각의 데이터 들이 배열 형식으로 감싸져있으며, 배열 안에 있는 객체들을 iOS가 이해할 수 있도록 Decodable로 만드는 작업이 필요하다.



JSON에서 documents 객체에 감싸져서 데이터들이 넘어오므로 documents Key는 그대로 사용하고 안에 있는 객체는 EventCategoryDocument Decodable 구조체로 정의한다.

```
import Foundation

struct EventCategoryDTO: Decodable {
    let documents: [EventCategoryDocument]
}

struct EventCategoryDocument: Decodable {
    let title: String?
    let thumbanilTextURL: String?

    enum CodingKeys: String, CodingKey {
        case title = "event_title"
        case thumbanilTextURL = "thumbnail_text_image_url"
    }

    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)

        self.title = try? values.decode(String?.self, forKey: .title)
        self.thumbanilTextURL = try? values.decode(String?.self, forKey: .thumbanilTextURL)
    }
}
```

2.4 API 스펙 정의, 정의한 API 스펙을 통해 URLSession으로 HTTP 통신을 할 수 있는 함수를 선언

```
class FetchEventCategoryNetwork {
    private let session: URLSession
    let api = FetchEventCategoryAPI()

    init(session: URLSession = .shared) {
        self.session = session
    }

    func fetchEventCategory() -> Single<Result<EventCategoryDTO,
        FetchEventCategoryError>> {
        guard let url = api.fetchEventCategory().url else {
            return .just(.failure(.invalidURL)) }
        let request = NSMutableURLRequest(url: url)
        request.httpMethod = "POST"

        return session.rx.data(request: request as URLRequest)
            .map { response in
                do {
                    let eventCategoryData = try
                        JSONDecoder().decode(EventCategoryDTO
                        .self, from: response)
                    return .success(eventCategoryData)
                } catch {
                    return .failure(.invalidJSON)
                }
            }.catch { _ in
                    .just(.failure(.networkError))
            }.asSingle()
    }
}
```

네트워크는 Success 또는 Failure의 결과만 갖고 있으므로 Single Trait을 사용하는것이 적적하다 생각했다. 또한 Swift의 Result는 Single의 정의와 잘 맞는다 판단하여 Single이 Result 타입을 갖도록 설정했다.

```
enum FetchEventCategoryError: Error {
    case invalidURL
    case invalidJSON
    case networkError
}
```

이때 Result의 Success는 서버로부터 JSON 데이터를 Decoding에 성공했을때 반환할 객체(EvnetCategoryDTO), Failure는 enum case로 정의한 FetchEventCategoryError를 반환하도록 구현했다.

제 3 장 MVVM

3.1 정의

본 프로젝트에서 View, ViewModel, Model의 구분을 다음와 같이 통일했다.

- View: 오직 UI를 구현하는 코드만 담는다.
    - Parent View의 bind(_:) 함수 내에서 Child View의 bind(_:) 함수를 호출하여 Parent ViewModel과 Child ViewModel을 바인드 한다.
- ViewModel: View와 바인딩 돼서 데이터와 사용자 액션을 처리한다.
    - Parent ViewModel은 Child ViewModel를 Components로 갖는다.
- Model: 기능에 대한 순수한 비즈니스 로직은 Model에서 구현한다.

3.2 네트워크 호출 시기

위 장에서 언급한 Network 로직 호출 시기에 대한 고민을 했다. CategoryCollectionView는 RecommendViewController의 SubView 이며, RecommendViewController는 HomeViewController의 MenuButtonSectionView의 버튼이 눌릴때 로드된다.

따라서 RecommendViewController가 로드되기 전 즉, RecommendViewController.viewWillAppear(_:) 메소드가 호출될 때 Network 로직 호출돼서 JSON Data를 가져오는 것이 적절하다 판단했다.

이를 위해 Rx Extension을 통해 viewWillAppear<Void>를 ControlEvent로 선언했다.

```
extension Reactive where Base: UIViewController {
    var viewWillAppear: ControlEvent<Void> {
        let source = self.methodInvoked(#selector(Base.viewWillAppear(_:))).map { _ in }

        return ControlEvent(events: source)
    }
}
```

선언한 viewWillAppear를 RecommendViewController.bind(_:) 내에서 호출하여 RecommendViewModel.recommendViewWillAppear Observable에 바인드 했다.

```
self.rx.viewWillAppear
    .bind(to: viewModel.recommendViewWillAppear)
    .disposed(by: disposeBag)
```

3.3 비즈니스 로직 분리

RecommendViewModel.recommnedViewWillAppear가 방출하는 이벤트를 가공해서 다시 RecommnedViewConroller에 전달하는 과정중 ViewModel내에 Network의 비즈니스 로직이 섞여 코드의 가독성이 떨어지는 것을 파악했다.

위처럼 ViewModel에 데이터를 처리하는 비즈니스 로직이 담기게되면 추후에 발생할 코드의 재사용성, 유지보수, Unit Test 관점에서 적절한 조치가 아니라 판단했다.

이를 해결하기 위해 Network에 대한 순수 비즈니스 로직은 Model에 별도로 분리하고 ViewModel에서는 Model에서 정의한 로직을 호출만 하는 코드를 작성해서 View와 ViewModl의 역할을 확실하게 분리했다.

3.4 Model의 비즈니스 로직

RecommendModel에서 Network 비즈니스 로직을 처리하는 함수는 다음과 같다.

- 위 장에서 언급한 fetchEventCategory()를 호출하는 함수

```
func fetchEventCategory() -> Single<Result<EventCategoryDTO, FetchEventCategoryError>> {
    fetchEventCategoryNetwork.fetchEventCategory()
}
```

- fetchEventCategory()를 통해 방출되는 Event중 Success case만 분류해서 Decoding된 EventCategoryDTO 데이를 반환하는 함수

```
func getEventCategoryValue(_ result: Result<EventCategoryDTO, FetchEventCategoryError>) -> EventCategoryDTO? {
    guard case .success(let value) = result else { return nil }

    return value
}
```

- EventCategoryDTO에서 필요한 데이터(title, thumbnailTextURL)을 가져와 Cell에 뿌릴 EvnetCategoryCellData를 얻는 함수 (thumbnailURL프로퍼티 사용은 밑에서 설명)

```
func getEventCategroyList(_ value: EventCategoryDTO) -> [EventCategoryCellData] {
    value.documents
        .map { eventCategory in
            let thumbnailURL = URL(string: eventCategory.thumbanilTextURL ?? "")

            return EventCategoryCellData(title: eventCategory.title, thumbnailURL: thumbnailURL)
        }
}
```

3.5 데이터 처리

이제 RecommnedViewModel에서는 RecommendModel에서 정의한 함수들을 호출해서 데이터를 가공하는 작업을 수행한다.

```
let evnetCategoryResult = recommendViewWillAppear
    .withLatestFrom(model.fetchEventCategory())
    .share()
```

```
let evnetCategoryValue = evnetCategoryResult
    .compactMap(model.getEventCategoryValue(_:))
```

최종적으로 가공된 데이터([EventCategoryCellData])는 CategoryCollectionViewModel.categoryCellData Observable에 바인드된다.

```
evnetCategoryValue
    .map(model.getEventCategroyList(_:))
    .bind(to: categoryCollectionViewModel.categoryCellData)
    .disposed(by: disposeBag)
```

CategoryCollectionView에서 방출된 이벤트는 UI변경에 영향을 주므로 Main Thread에서 동작을 보장해야한다. 따라서 categoryCellData Observable을 asDriver로 변경한 categoryDataList를 통해 CategoryCollectionView 에 넘겨준다.

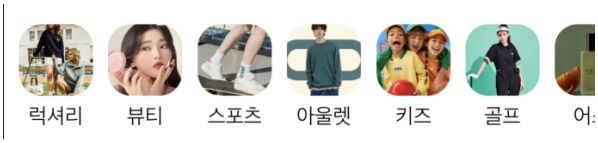
```
// ParentViewModel -> ViewModel
let categoryCellData = PublishSubject<[EventCategoryCellData]>()

// ViewModel -> View
let categoryDataList: Driver<[EventCategoryCellData]>

init() {
    self.categoryDataList = categoryCellData
        .asDriver(onErrorJustReturn: [])
}
```

3.6 MVVM ETC

아래의 왼쪽 사진은 위에서 작업한 RecommendViewController.CategoryCollectionView의 화면이고 오른쪽 사진은 CategoryViewController.EventCategoryListView의 화면이다. 두 사진속 UICollectionView를 비교해보면 각각의 Cell들은 이미지만 다를뿐 텍스와 배열이 일치하고 Cell을 탭 했을 때 동일한 View로 이동한다.



따라서 두 UICollectionView는 동일한 형식의 EventCategoryCellData를 필요로 하며 이미지만 다르게 처리하는 과정을 구현했다. 이를 위해 본 프로젝트에서는 서버에서 테이블 컬럼(thumbnail_imag_url)을 추가했다.


```
"documents": [
    {
        "id": 1,
        "event_title": "럭셔리",
        "thumbnail_image_url": "https://image.msscdn.net/mfile_s01/_lookbook/list63f2d10b7f7ec",
        "thumbnail_text_image_url": "https://image.msscdn.net/images/event_banner/20220913164934000000037242.png"
    },
    {
        "id": 2,
        "event_title": "뷰티",
        "thumbnail_image_url": "https://image.msscdn.net/images/goods_img/20230102/3002829/3002829_16726343803772_500.jpg",
        "thumbnail_text_image_url": "https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcTFr438zqpO3qp7jKQM3lsui_JLXzmAhQr36oaI
    },
    {
        "id": 3,
        "event_title": "스포츠",
        "thumbnail_image_url": "https://image.msscdn.net/mfile_s01/_lookbook/list63f2fe3edbdb2",
        "thumbnail_text_image_url": "https://lh3.googleusercontent.com/
            IMqAmHFBD03_soZg3kBEDzrw3a-pCsWWix7XGVf5hz4HtyXHQgZuphZAPBkp4e3FNr1dDnl2j45XS4fviqGYuSSsnEINkMstDSejmyI"
    },
```


그리고 기존에 RecommnedViewModel에서 EventCateogryDTO를 방출하는 eventCategoryValue Observable을 CategoryViewController.EventCategoryListView에 전달하기 위해 최상위 ViewModel인 TabBarViewModel 에서 RecommendViewModel.evnetCateogryValueList를 Subscrib하도록 설정했고 eventCategoryValueList에서 이벤트가 방출할 때마다 EventCategoryViewModel.eventCategoryCellData Observabler 에서 이벤트를 방출하도록 설정했다.

```
evnetCategoryValue
    .bind(to: self.eventCategoryValueList)
    .disposed(by: disposeBag)
```

```
homeViewModel.recommendViewModel.eventCategoryValueList
    .bind(to: categoryViewModel.eventCategoryListViewModel
        .eventCategoryCellData)
    .disposed(by: disposeBag)
```

이제 CategoryViewController.EventCateogryListView에서 사용할 데이터 맞은 데이터(title, thumbnailURL)을 가져와 EventCategoryCellData로 반환하는 함수

getEventCategoryListCellData(_:) 함수를 EvnetCategoryListModel에서 선언하고 EventCategoryListViewModel에서 호출하여 EvnetCategoryListView에 넘겨주었다.

```
func getEventCategoryListCellData(_ value: EventCategoryDTO) -> [EventCategoryCellData] {
    value.documents
        .map { eventCategoryDocument in
            let thumbnailURL = URL(string: eventCategoryDocument.thumbnailURL ?? "")

            return EventCategoryCellData(title: eventCategoryDocument.title, thumbnailURL: thumbnailURL)
        }
}
```

```
init(model: EventCategoryListModel = EventCategoryListModel()) {

    let sample = eventCategoryCellData
        .map(model.getEventCategoryListCellData(_:))

    self.cellData = sample
        .asDriver(onErrorJustReturn: [])
}
```

위 작업을 통해 두 번의 Network 통신으로 구현될 작업을 한 번에 처리하므로서 코드의 길이를 줄이고 기존 코드를 재활용하는 기회를 가지면서 MVVM 아키텍쳐 설계의 견문을 넓히는 기회를 가졌다.