

프로젝트 코드 리뷰 문서

- Smooth 팀 / 김두리 (iOS)

(github - <https://github.com/stove-smooth/sgs-smooth/tree/develop/src/frontend/ios>)

0. 들어가기

- 프로젝트 소개

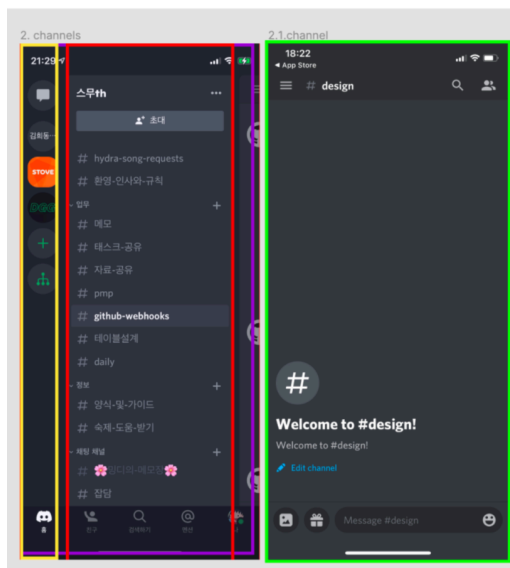
본 프로젝트는 디스코드의 주요 기능을 클론하는 것을 목표로 하였습니다. 주요 기능으로는, 실시간 채팅 및 실시간 음성/화상 채팅입니다.

- 개념 및 용어 설명

모든 화면 및 단위의 명칭은 실제 디스코드 서비스에서 사용하고 있는 명칭을 사용하였습니다. 따라서, 주요 기능 단위의 명칭은 다음과 같습니다.

용어	설명
Community(or Server)	슬랙의 워크스페이스의 개념으로, 사람들이 모이는 곳
Channel	일종의 단체방 개념으로 하나의 Room이 될 수 있음
Chatting	실제 채팅이 이뤄지는 곳

위 용어를 바탕으로, 메인 화면은 다음과 같이 설계됩니다.



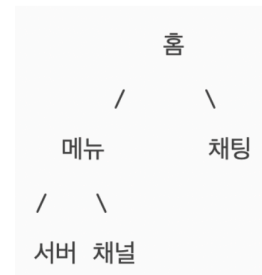
노랑 - 서버(커뮤니티) vc

빨강 - 채널 vc

- 노랑+빨강 - 메뉴 vc

초록 - 채팅 vc

보라 - 홈 vc



크게 [Menu]와 [Chatting]을 관리하는 [Home] VC와, Server와 Channel 정보를 관리하는 [Menu] VC 마지막으로 실제 대화가 이뤄지는 [Chatting]으로 이뤄져 있습니다.

각각의 ViewController들은 부모 ViewController에서 전체를 관리하게 되며, 해당 화면 및 단위에서의 데이터 처리는 해당 ViewController에서 하게 됩니다.

하나의 화면에 여러 정보를 담고 있는 컴포넌트들이 많아, 조금은 복잡하지만 위와 같이 구조화하였습니다. 이와 관련하여 구체적으로 어떻게 동작하는지는 아래 3. 주요기능 및 코드 부분에서 기술됩니다.

1. 프로젝트 파일 구조

- Project

```
smooth-ios
├── Modules : 화면 단위
├── Service : 네트워크 레이어 구현체
├── API : 네트워크 레이어
├── Model : Entity
├── Coordinator : 일종의 라우터
├── Base : MVVM 패턴에 의해 공통적으로 사용하는 파일들
├── Common : 프로젝트 기본 구성파일
├── Utils : 자주 사용하는 기능들을 커스텀한 파일
│   ├── UserDefaults, Alert ... Utils files
│   └── Extension : 퍼스트파티 커스텀한 파일들
```

MVVM+Coordinator 패턴을 바탕으로 하여, 개발하였습니다. 따라서, 패턴의 구조 별로 디렉토리를 두어 관리하였고, Modules의 경우 화면 단위로 구성하였습니다.

- Modules

```
Modules
├── Splash : 로그인 메인 화면
├── Signin : 로그인 화면
├── Signup : 회원가입 진입 화면
│   ├── SignupInfo : 회원 정보(닉네임, 비밀번호)
│   └── Verifycode : 이메일 인증
├── Home
│   ├── HomeViewController.swift
│   ├── Menu
│   ├── Channel : Menu의 채널
│   ├── Channel-make : 채널 생성
│   ├── Category
│   │   └── Make : 카테고리 생성
│   ├── Chatting : 채팅 화면
│   ├── Server-Info : 서버 정보
│   ├── Server-Setting : 커뮤니티 설정 화면
│   │   ├── Category-edit : 카테고리 편집
│   │   ├── Category-reorder : 카테고리 재배치
│   │   ├── Channel-reorder : 채널 재배치
│   │   ├── Category-setting : 카테고리 편집
│   │   ├── Server-Invite : 서버 초대
│   │   └── Server-edit : 서버 수정
│   └── SubView : Home(main)화면에서 주로 사용된 View 파일들
├── AddServer : 서버 추가 화면
│   ├── Invite : 서버 초대장 만들기
│   ├── Join : 서버 초대 받기
│   ├── Make : 서버 생성
│   └── Regist : 초대장 등록하기
├── Friend : 친구 리스트
│   ├── Info : 친구 정보
│   ├── Request : 친구 요청
│   └── SubView : 친구 리스트 구성에 필요한 View 파일들
```

2. 프로젝트 개발 컨셉

1) 개발 컨셉

- 스토리보드 사용 지양

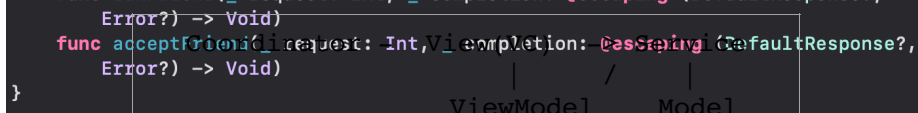
재사용 가능한 개발을 목표로 하여, 모든 View 파일들은 코드로서 작성하였습니다. 이때 View를 보다 더 편리하게 작성하기 위해 SnapKit과 Then 라이브러리를 사용하여 작성하였습니다.

- 프로토콜 프로그래밍 지향 프로그래밍

프로토콜을 정의하여 해당 기능을 모듈화 하고자 하였습니다. 같은 레이어에서 하나의 프로토콜을 정의하여, Extension과 Generic을 통해 자유롭게 모듈 단위로 커스텀하여 사용하였습니다. 이를 통해 코드의 범용성을 가질 수 있게끔 작성하였습니다.

```
protocol FriendServiceProtocol {
    func fetchFriend(_ completion: @escaping ([FriendSection]?, Error?) -> Void)
    func deleteFriend(_ request: Int, _ completion: @escaping (DefaultResponse?, Error?) -> Void)
    func requestFriend(_ request: RequestFriend, _ completion: @escaping (DefaultResponse?, Error?) -> Void)
    func banFriend(_ request: Int, _ completion: @escaping (DefaultResponse?, Error?) -> Void)
    func acceptFriend(_ request: Int, _ completion: @escaping (DefaultResponse?, Error?) -> Void)
}

struct FriendService: Networkable, FriendServiceProtocol {
    typealias Target = FriendTarget
}
```

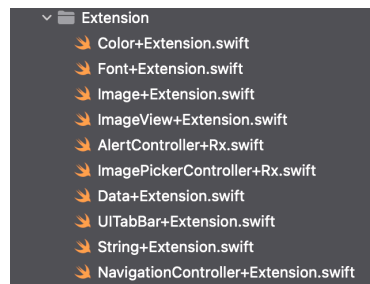
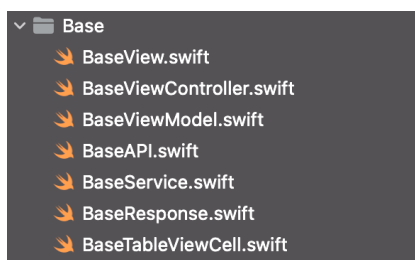


네트워크(Service, API) 레이어는 API에 정의한 내용들을 바탕으로 Service(구현체)에서 사용한다.
(이때 Target이 associatedtype으로, ServiceProtocol에서 채택한다.)

- 컴포넌트의 모듈화 및 재사용

자주 사용하는 코드나 디자인 패턴에 기반하여 중복해서 사용해야 하는 코드들은 별도로 파일을 생성하여 관리하였습니다.

주로, 중복해서 사용해야 하는 코드들은 [Base] 디렉토리에 정의하였고, 프로젝트를 개발하면서 자주 사용되는 코드의 경우 해당 클래스를 Extension 하여 중복 코드를 최소화 하고자 노력하였습니다.



(오른쪽) : 각 계층에서 동일하게 사용한 코드들을 미리 정의하였다.
(왼쪽) : 자주 사용하는 컴포넌트들의 코드들을 Extension하여 관리하였다.

2) 적용 디자인 패턴 : MVVM+Coordinator

본 프로젝트는 MVVM+Coordinator 패턴을 지향하여, 프로젝트 설계 하였습니다. 각 계층 별로 역할은 다음과 같이 하였고, 이를 바탕으로 구조화 하였습니다.

- View/ViewController : ViewModel에 의존하며, 값을 변경하거나 전달 해야할 때 Rx 사용 지향
- ViewModel : input과 output을 분리하여 화면 데이터 관리

- Service : Network Layer와 함께 사용되며, Entity를 로직에서 사용하는 Model로 변환해줌
- Coordinator : 화면 전환에 모든 것을 관리

3. 주요 기능 및 코드

- Home : 하나의 화면에서 여러 정보(서버, 채널, 채팅)들을 보여줍니다. ([github](#))

1) [Menu]와 [Chatting]은 animation을 통해 화면 전환이 이뤄진다.
로그인 성공 시 최초로 보여지는 화면은 [Chatting]입니다. 따라서, 유저는 [Menu]로 진입하여 서비스를 소비하게 되는데 이에 따른 화면 이동은 Delegate 패턴을 통해 구현하였습니다.

```
// MARK: - Data 동기화 (menu <- home -> chatting)
extension HomeViewController: MenuViewControllerDelegate {
    func swipe(channel: Channel?, communityId: Int) {
        self.didTapMenuButton(channel: channel, communityId: communityId) // 화면전환 애니메이션
        // delegate로 전달
        self.delegate?.loadChatting(channel: channel!, communityId: communityId)
    }
}

// MARK: - Menu Animation
extension HomeViewController: ChattingViewControllerDelegate {
    func dismiss(channel: Channel?, communityId: Int?) {
        self.menuState = .closed
        toggleMenu(completion: nil)
        self.delivery?.appear(channel: channel, communityId: communityId)
    }

    func didTapMenuButton(channel: Channel?, communityId: Int?) {
        toggleMenu(completion: nil)
        self.delivery?.appear(channel: channel, communityId: communityId)
    }
}
```

Delegate를 통해 화면 전환 시 필요한 데이터들을 전달 받습니다. 이 모든 것들은 HomeViewController가 중간에 관리하며, 이에 따라 각각의 Child ViewController에게 delegate이 선언됩니다.

2) Server와 Channel의 데이터 정보는 [Menu]가 관리한다.
MenuViewModel이 Server와 Channel에 필요한 데이터를 모두 관리합니다. 따라서, MenuViewModel에 생성/변동된 값들은 Rx Binder를 통해 전달합니다.
전달된 데이터들은 해당 View에서 Rx 문법을 통해 화면을 그리게 되고, 최종적으로 해당 View에 대한 event는 다시 MenuViewController/Model에서 합니다.

- [MenuView Binder] ([소스코드 링크](#))
- [Server/Channel 값 바인딩] ([소스코드 링크](#))
- [화면 전환에 따른 Observable 객체 관리] ([소스코드 링크](#))

- Chatting : StompClient를 이용하여 실시간 통신

1) chatWebSocket Service 등록 시점
chatting은 메인 화면의 진입점임으로, 앱 시작과 동시에 WebSocket 연결을 합니다. ([코드 링크](#))
HomeViewController는 각 ViewController를 중간에 매니징하는 화면에 불과하기 때문에, 앱 시작점으로부터 socket 연결을 하였습니다.

2) 채팅의 경우, MessageKit을 이용하였기 때문에, 기존 라이브러리의 제약선에서 model을 관리합니다. ([소스코드](#)) ([작성중](#))

- WebRTC