Premier-League-Match-Grinta-iOS-Task

By: Mohamed Salah

Introduction:

The Premier League Match App is an iOS application built using Swift and UIKit. It provides users with real-time data about Premier League matches, including information about both played and unplayed matches, match scores, and match schedules. Users can also add matches to their favorites, and unplayed matches are automatically updated when the match results are available. The app leverages RxSwift for reactive programming, RxDataSource for animations and Realm for local data storage. It also includes unit tests to ensure code reliability.

Architecture Design:

MVVM

Technology Used:

Swift || UIKit || RxSwift || RxDataSource || Realm || UnitTesting

Network Layer

1- Protocol:

- **SOLID Principles**: The design of this struct aligns with SOLID principles, specifically the Open-Closed Principle and Dependency Inversion Principle. This means that any network layer implementing the protocol can work with it without requiring a specific type. It promotes flexibility and extensibility, allowing for various parsing types without modifying the core network client.

- **Unit Testing**: Another goal is to facilitate unit testing. By using Swift's generics and allowing for dependency injection, it becomes simple to create mock network clients for testing purposes. During testing, you can easily pass a mock implementation of APIClientProtocol, which helps in isolating the code under test from actual network requests, making your codebase more testable and maintainable.

```swift
struct FetchGlobalRequest<T: Codable> {
    let parsingType: T.Type
    let baseURL: URL
    let attributes: [String]?
    let queryParameters: [String: String]?
    let jsonBody: [String: Any]?
    let headers: [String: String]?

    init(
        parsingType: T.Type,
        baseURL: URL,
        attributes: [String]? = nil,
        queryParameters: [String: String]? = nil,
        jsonBody: [String: Any]? = nil,
        headers: [String: String]? = nil
    ) {
        self.parsingType = parsingType
        self.baseURL = baseURL
        self.attributes = attributes
        self.queryParameters = queryParameters
        self.jsonBody = jsonBody
        self.headers = headers
    }
}
protocol APIClientProtocol {
    func fetchGlobal<T: Codable>(request: FetchGlobalRequest<T>) -> Observable<T>

    func fetchLocalFile<T: Codable>(
        parsingType: T.Type,
        localFilePath: URL
    ) -> Observable<T>
}
```

2- Network Functions:

- **Flexibility**: This function is capable of handling a wide range of network requests. You can use it with different combinations of headers, parameters, HTTP methods (GET or POST), and request bodies. This adaptability makes it versatile for various use cases.
- **Generics for Parsing**: It uses generics to allow you to specify the type for parsing the response data. This means you don't need to worry about the specific type when making a request; it's determined when you call the function. This keeps your code more generic and reusable.

SOLID Principles: The function also follows some important SOLID principles:

- **Single Responsibility Principle (SRP)**: It focuses on two key responsibilities - sending network requests and parsing responses. This separation of concerns makes the code easier to understand and maintain.
- **Open-Closed Principle (OCP)**: It's open for extension, meaning you can use it for various types of network requests without altering the core functionality. It's closed for modification, ensuring that changes to one part don't affect the rest of the code.
- **Dependency Inversion Principle (DIP)**: By using generics for response parsing, it promotes the DIP. You depend on abstractions (protocols and generics) rather than concrete types, enhancing flexibility and maintainability.

```
private func buildURL(baseURL: URL, attributes: [String]?, queryParameters: [String:
    var components = URLComponents(url: baseURL, resolvingAgainstBaseURL: false)!
    if let attributes = attributes, !attributes.isEmpty {
        components.path += "/" + attributes.joined(separator: "/")
    }
    if let queryParameters = queryParameters {
        components.queryItems = queryParameters.map { URLQueryItem(name: $0.key, val
    }
    return components.url
}

private func buildRequest(url: URL, method: String, jsonBody: [String: Any]?, header
    var request = URLRequest(url: url)
    request.httpMethod = method
    if let jsonBody = jsonBody {
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        if let jsonData = try? JSONSerialization.data(withJSONObject: jsonBody) {
            request.httpBody = jsonData
        }
    }
    if let headers = headers {
        for (key, value) in headers {
            request.setValue(value, forHTTPHeaderField: key)
        }
    }
    return request
}
func fetchGlobal<T>(request: FetchGlobalRequest<T>) -> Observable<T> {
    return Observable<T>.create { observer in
        guard let url = self.buildURL(baseURL: request.baseURL, attributes: request.
            request.queryParameters) else {
            observer.onError(NSError(domain: "Invalid URL", code: -1, userInfo: nil))
            return Disposables.create()
        }
        let method = request.jsonBody != nil ? "POST" : "GET"
        let request = self.buildRequest(url: url, method: method, jsonBody: request.
        AF.request(request).responseDecodable(of: T.self) { response in
            switch response.result {
            case .success(let value):
```

View Model

1- VM First Part:

```
class MatchViewModel {
    let apiManager: APIClientProtocol
    init(apiManager: APIClientProtocol = APIManager()) {
        self.apiManager = apiManager
    }
    var matchesModel: Matches?
    //In
    let selectedSegmentIndex = BehaviorRelay<Int>(value: 0)

    //out
    var currentMatchesList = PublishRelay<[Match]>.init()
//    var matchesList = PublishRelay<[Match]>.init()
    var matchesList = BehaviorRelay<[Match]>(value: [])
    var favoruiteMatchesList = BehaviorRelay<[Match]>(value: FavouritesManager.shared
    var showLoading = BehaviorRelay<Bool>(value: false)
    var errorSubject = PublishSubject<Error>()
    private let disposeBag = DisposeBag()
    func inialize() {
        getMatches()
        setupBinding()
    }
```

I've used dependency injection for the apiManager to ensure flexibility and easy testing. This means you can provide any APIClientProtocol implementation when creating an instance of this class.
For the various components, here's what they do:

- selectedSegmentIndex: Represents the currently selected segment index.
- currentMatchesList: Publishes the current list of matches.
- matchesList: Holds a list of matches.
- favoruiteMatchesList: Maintains a list of favorite matches.
- showLoading: Signals whether loading is in progress.
- errorSubject: Publishes errors when they occur.

In the **inialize** method, we're calling **getMatches()** and setting up some initial **bindings** for these components. This sets the stage for further operations and interactions in the code.

No table of contents entries found.

2- VM Seound part

getMatches():

- Fetches match data, displays a loading indicator, and updates match-related properties based on the received data or errors.

setupBinding():

- This function sets up a binding for the favoruiteMatchesList.
- It listens to changes in the list of favorite matches and updates the currentMatchesList when the user is on the favorites page (when selectedSegmentIndex.value is equal to 1).
- This ensures that the currentMatchesList reflects the favorites only when the user is actively viewing the favorites page, as you want to keep it separate from the regular matches when on other pages.

```
func getMatches() {
    self.showLoading.accept(true)
    let headers = [
     "X-Auth-Token": SecurityConstants.Links.apiKey
    ]
    let request = FetchGlobalRequest(parsingType: Matches.self, baseURL
        "PL").stringToUrl, headers: headers)
    apiManager.fetchGlobal(request: request)
        .observe(on: MainScheduler.instance)
        .subscribe(
            onNext: { matchesModel in
                self.showLoading.accept(false)
                self.matchesModel = matchesModel
                self.matchesList.accept(matchesModel.matches)
                self.currentMatchesList.accept(matchesModel.matches)
            },
            onError: { error in
                self.showLoading.accept(false)
                self.errorSubject.onNext(error)
            }
        )
        .disposed(by: disposeBag)
}
private func setupBinding() {
    favoruiteMatchesList
        .subscribe(onNext: { [weak self] favoriteMatches in
            guard let self = self else { return }

            if self.selectedSegmentIndex.value == 1 {
                self.currentMatchesList.accept(favoriteMatches)
            }
        })
        .disposed(by: disposeBag)
}
```

3- VM Third part

formatDate(dateString:yearly:):

- Converts a date string to a human-readable format.
- Accepts an optional yearly parameter for including the year.
- Returns the formatted date or nil if the input is invalid.

formatTime(dateString:timeZone:):

- Formats the time part of a date string.
- Allows specifying a time zone for the output.
- Returns the formatted time with the specified time zone or nil for invalid input.

categorizeMatches(_ matches:):

- Groups a list of Match objects into sections based on their date.
- Sorts sections by date, placing the nearest matches at the beginning.
- Returns categorized matches as an array of date-section tuples.

```
func formatDate(dateString: String, yearly: Bool = false) -> String? {
    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss'Z'"

    if let date = dateFormatter.date(from: dateString) {
        dateFormatter.dateFormat = yearly ? "MM/dd/yyyy" : "MM/dd"
        return dateFormatter.string(from: date)
    }

    return nil
}
func formatTime(dateString: String, timeZone: TimeZone = .current) -> String? {
    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss'Z'"
    dateFormatter.timeZone = TimeZone(secondsFromGMT: 0)

    if let date = dateFormatter.date(from: dateString) {
        dateFormatter.timeZone = timeZone
          dateFormatter.dateFormat = "h:mm a"
        dateFormatter.dateFormat = "HH:mm"
        return dateFormatter.string(from: date)
    }

    return nil
}

func categorizeMatches(_ matches: [Match]) -> [(String, [Match])] {
    let categorizedMatches = Dictionary(grouping: matches) { match in
        return formatDate(dateString: match.utcDate, yearly: true) ?? "Unknown"
    }

    var sections = categorizedMatches.map { key, value in
        return (key, value)
    }

    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "MM/dd/yyyy"
    sections.sort { section1, section2 in
```

Note: There are still another two functions, but we will talk about them in the Realm part.

RxSwift Functions in View

RxSwift Part 1:

BindTableView Function:

This function binds data to a table view and enables animated section transitions. It uses the RxTableViewSectionedAnimatedDataSource to configure cell content and section headers dynamically. Each cell is customized based on match data, including team information, scores, and favoriting functionality. When the user taps the favorite button, it adds or removes matches from their favorites list, updating the view accordingly. The table view sections are animated based on the incoming data using RxSwift.
In summary, bindTableView handles dynamic table view content, animation, and interaction related to displaying match data and managing favorites.

```
func bindTableView() {
            }
            cell.homeTeamNameLabel.text = match.homeTeam.name
            cell.awayTeamNameLabel.text = match.awayTeam.name
            if match.score.winner == nil {
                cell.statusLabel.text = matchVM.formatTime(dateString: match.utcDate
                cell.timeZoneLabel.text = TimeZone.current.identifier
            } else {
                let homeScore = String(match.score.fullTime.home ?? 0)
                let awayScore = String(match.score.fullTime.away ?? 0)
                cell.statusLabel.text = homeScore + "-" + awayScore
                cell.timeZoneLabel.text = ""
            }
            cell.saveToFavouritesButton.isChecked = FavouritesManager.shared().isMat
            let favouriteMatch = matchVM.createFavouriteObject(match: match)
            cell.onFavButtonTapped = {
                if cell.saveToFavouritesButton.isChecked {
                    FavouritesManager.shared().addToFavorites(match: favouriteMatch)
                } else {
                    FavouritesManager.shared().removeFromFavorites(match: favouriteM
                }
                self.matchVM.favoruiteMatchesList.accept(FavouritesManager.shared().
            }
            return cell
        },
        titleForHeaderInSection: { dataSource, sectionIndex in
            return dataSource[sectionIndex].model
        }
    )

    matchVM.currentMatchesList
        .map { matches in
            let categorizedMatches = self.matchVM.categorizeMatches(matches)

            return categorizedMatches.map { date, matches in
                MatchSectionModel(model: date, items: matches)
            }
        }
        .bind(to: matchTableView.rx.items(dataSource: dataSource))
        .disposed(by: disposeBag)
}
```

RxSwift Part 2:

bindViewsToViewModels():

- Binds the selected segment index of a segmented control to the selectedSegmentIndex property in the view model. This connection ensures that changes in the segmented control reflect the selected segment in the view model.

bindViewModelsToViews():

- Subscribes to changes in the selectedSegmentIndex property of the view model.
- Updates the view model's currentMatchesList based on the selected segment index. When the user changes segments, the view model is notified and updates the displayed matches accordingly. Note: We will talk more about this in the Realm Part

handleErrors():

- Displays an alert message with an error description when an error occurs in the view model.

handleLoadingIndicator():

- Manages the visibility of an activity indicator based on the showLoading property in the view model. The indicator is shown during loading and hidden when loading is complete.

```
func bindViewsToViewModels() {
    segmentOutlet.rx.selectedSegmentIndex
        .bind(to: matchVM.selectedSegmentIndex)
        .disposed(by: disposeBag)
}
func bindViewModelsToViews() {
    matchVM.selectedSegmentIndex
        .subscribe(onNext: { [weak self] index in
            guard let self = self else { return }
            switch index {
            case 0:
                self.matchVM.currentMatchesList.accept(self.matchVM.matchesList.v
            case 1:
                self.matchVM.updateMatches()
                self.matchVM.currentMatchesList.accept(self.matchVM.favoruiteMatc
            default:
                break
            }
        })
        .disposed(by: disposeBag)

}
func handleErrors() {
    matchVM.errorSubject
        .subscribe { error in
            self.show(messageAlert: "Error", message: error.localizedDescription)
        }
        .disposed(by: disposeBag)
}
func handleLoadingIndicator() {
    matchVM.showLoading
        .observe(on: MainScheduler.instance)
        .subscribe(onNext: { [weak self] isLoading in
            if isLoading {
                self?.activityIndicator.startAnimating()
            } else {
                self?.activityIndicator.stopAnimating()
            }
```

Realm

Realm part 1:

Singleton Pattern for Consistency:
A singleton pattern is implemented to provide a single instance of FavouritesManager. This ensures that favorited match data is consistently managed across the entire application.

addToFavorites Function:
The addToFavorites function adds a match to the user's list of favorites while first checking if the match is already favorited. If it is, the existing entry is removed to prevent issues if a user repeatedly taps the favorite button. Afterward, a new match entry is created and added to the database, updating the user's list of favorites.

isMatchFavorited Function:
This function checks if a specific match is already in the list of favorites by searching for it in the database. It returns true if the match is found.

removeFromFavorites Function:
The removeFromFavorites function allows users to straightforwardly remove a match from their list of favorites. If the match exists in the list of favorites, it is removed from the database.

This code ensures the reliable management of favorited matches and safeguards against potential issues if users interact rapidly with the favorite button.

```
class FavouritesManager {

    private static let sharedInstance = FavouritesManager()
    private let realm = try! Realm()

    static func shared() -> FavouritesManager {
        return FavouritesManager.sharedInstance
    }

    func addToFavorites(match: FavouriteMatch) {
        try! realm.write {
            if let existingMatch = realm.object(ofType: FavouriteMatch.self, forPrimaryK
                if !existingMatch.isInvalidated {
                    realm.delete(existingMatch)
                }
            }
            //Safer Approach - To avoid crashes - if the user spam the RadioButton
            let newMatch = FavouriteMatch(id: match.id,
                                          homeTeamImage: match.homeTeamImage,
                                          awayTeamImage: match.awayTeamImage,
                                          homeTeamName: match.homeTeamName,
                                          awayTeamName: match.awayTeamName,
                                          utcDate: match.utcDate,
                                          matchday: match.matchday,
                                          winner: match.winner,
                                          fullTimeScoreHome: match.fullTimeScoreHome,
                                          fullTimeScoreAway: match.fullTimeScoreAway,
                                          competitionCode: match.competitionCode,
                                          competitionName: match.competitionName)
            realm.add(newMatch, update: .modified)
        }
    }

    func isMatchFavorited(id: Int) -> Bool {
        return realm.object(ofType: FavouriteMatch.self, forPrimaryKey: id) != nil
    }

    func removeFromFavorites(match: FavouriteMatch) {
        if let matchToDelete = realm.object(ofType: FavouriteMatch.self, forPrimaryKey:
            try! realm.write {
                realm.delete(matchToDelete)
            }
```

Realm part 2:

getAllFavoriteMatches():

- Retrieves all favorite matches from the Realm database.
- Converts the retrieved FavouriteMatch objects to Match objects to make them compatible with UI display.
- Returns an array of Match objects representing the user's favorite matches.

convertFromRealmObjectToNormal(favoriteArray:):

- Converts an array of FavouriteMatch objects to an array of Match objects.
- Maps each FavouriteMatch to a Match object with the necessary data transformations, such as creating Competition, Team, and Score objects.

fetchMatchesToUpdate():

- Queries the Realm database to find matches that require an update (e.g., matches with no winner information).
- Returns a list of FavouriteMatch objects that need to be updated.

updateMatchInRealm(match:with updatedMatch:):

- Updates a specific FavouriteMatch object in the Realm database with information from an updated Match object.
- Modifies the winner, full-time scores, and any other relevant data to keep the database synchronized with the latest match information.

These functions allow for seamless retrieval and manipulation of favorite match data stored in Realm, facilitating a smooth user experience in your application.

```swift
func getAllFavoriteMatches() -> [Match] {
    let favoriteMatches = realm.objects(FavouriteMatch.self)
    return convertFromRealmObjectToNormal(favoriteArray: Array(favoriteMatches))
}
func convertFromRealmObjectToNormal(favoriteArray: [FavouriteMatch]) -> [Match] {
    return favoriteArray.map { favoriteMatch in
        let competition = Competition(name: favoriteMatch.competitionName, code: fav
        let homeTeam = Team(name: favoriteMatch.homeTeamName, crest: favoriteMatch.h
        let awayTeam = Team(name: favoriteMatch.awayTeamName, crest: favoriteMatch.a

        var winner: Winner? = nil
        if let favoriteWinner = favoriteMatch.winner {
            winner = Winner(rawValue: favoriteWinner)
        }

        let fullTimeHome = Time(home: favoriteMatch.fullTimeScoreHome, away: favorit
        let score = Score(winner: winner, fullTime: fullTimeHome)

        return Match(competition: competition, id: favoriteMatch.id, utcDate: favori
            favoriteMatch.matchday, homeTeam: homeTeam, awayTeam: awayTeam, score: s
    }
}
//Update Old Data in DataBase
func fetchMatchesToUpdate() -> Results<FavouriteMatch> {
    let matchesToUpdate = realm.objects(FavouriteMatch.self).filter("winner == nil")
    return matchesToUpdate
}

func updateMatchInRealm(match: FavouriteMatch, with updatedMatch: Match) {
    try! realm.write {
        match.winner = updatedMatch.score.winner?.rawValue
        match.fullTimeScoreHome = updatedMatch.score.fullTime.home
        match.fullTimeScoreAway = updatedMatch.score.fullTime.away
    }
}
```

Realm part 3 (VIewModel):

createFavouriteObject(match:):

- Creates a FavouriteMatch object from a given Match item.
- Maps relevant properties from the Match item to the corresponding properties of the FavouriteMatch.

updateMatches():

- This function is responsible for updating old matches that were saved before they were played.
- It utilizes the FavouritesManager to fetch a list of matches that need to be updated.
- Iterates through the matches to check if an updated version is available in the current matchesModel.
- If an updated match is found, it updates the old match in the Realm database using the FavouritesManager.
- If any updates occurred, it notifies the favoruiteMatchesList to ensure the UI reflects the changes.

These functions work together to update matches in your favorite list when new data becomes available, ensuring that your stored match data is always up-to-date.

```
//MARK: - DataManager Helper Functions
extension MatchViewModel {
    func createFavouriteObject(match: AnimatableSectionModel<String, Match>.Item) ->
        FavouriteMatch {
        return FavouriteMatch(id: match.id,
                              homeTeamImage: match.homeTeam.crest,
                              awayTeamImage: match.awayTeam.crest,
                              homeTeamName: match.homeTeam.name ?? "N/A",
                              awayTeamName: match.awayTeam.name ?? "N/A",
                              utcDate: match.utcDate,
                              matchday: match.matchday,
                              winner: match.score.winner?.rawValue,
                              fullTimeScoreHome: match.score.fullTime.home,
                              fullTimeScoreAway: match.score.fullTime.away,
                              competitionCode: match.competition.code,
                              competitionName: match.competition.name)
    }
    //Update the old matches, that was saved before it was played
    func updateMatches() {
        let favouriteDataManger = FavouritesManager.shared()
        let matchesToUpdate = favouriteDataManger.fetchMatchesToUpdate()
        var somethingChanged: Bool = false
        matchesToUpdate.forEach { match in
            if let updatedMatch = matchesModel?.matches.first(where: { $0.id == match.id
                }), updatedMatch.score.winner != nil {
                somethingChanged = true
                favouriteDataManger.updateMatchInRealm(match: match, with: updatedMatch)
            }
        }
        if somethingChanged {
            favoruiteMatchesList.accept(favouriteDataManger.getAllFavoriteMatches())
        }
    }
}
```

Thank You