Starting Unit Testing with Model layer

24 Apr 2019

Today we are going to touch the completely new topic on my blog, and it is Unit Testing. Most of us heard about the pros of Unit Testing. I want to show how easily you can start with Unit Testing by covering your model layer. So let's start with the definition.

> Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

In other words, Unit Test is a code which tests individual unit on your codebase.

I think the Model layer is the best place to start writing Unit Tests. Assume that you are working on the Github client for iOS where you have a bunch of model structs which represents data fetched from Github API. Let's take a look at structs which define repository search results.

```
import Foundation

struct SearchResponse: Codable {
    let totalCount: Int
    let incompleteResults: Bool
    let items: [Repository]
}

struct Repository: Codable {
    let id: Int
    let name: String
    let owner: User
    let description: String
    let fork: Bool
    let url: String
    let homepage: String
    let stargazersCount: Int
    let watchersCount: Int
    let forksCount: Int
    let openIssuesCount: Int
}

struct User: Codable {
    let login: String
    let id: Int
    let avatarUrl: String
    let gravatarId: String
    let url String
}
```

Here we can see three structs: SearchResponse, Repository, and User. Every field of these structs represents an associated value from JSON which fetched during the search endpoint request. Next step is fetching and deserializing downloaded data into these structs.

```
class SearchLoader {
    typealias Handler = (Result<SearchResponse, Error>) -> Void

    private let session: URLSession
    private let decoder: JSONDecoder

    init(session: URLSession = .shared, decoder: JSONDecoder = .init())
        self.session = session
        self.decoder = decoder
    }

    func search(with query: String, handler: @escaping Handler) {
        session.dataTask(with: makeRequest(for: query)) { [weak self] da
            guard let self = self else {
                return
            }

            do {
                let response = try self.decoder.decode(SearchResponse.se
                handler(.success(response))
            } catch {
                handler(.failure(error))
            }
        }
    }
}
```

In the code sample above we have SearchLoader class which make an API request to Github's search endpoint and convert the data to SearchResponse struct. First of all, I want to cover with tests these data manipulations. Let's start with creating a Unit Test target in Xcode project(File -> New -> Target -> iOS Unit Testing bundle). Xcode should create it by default if you do not disable it during the project forming process.

Now we have to add JSON file with search endpoint response as a content to a testing target. We will use it to mock network request and speed up our test by faking real network request.

```
{
  "total_count": 40,
  "incomplete_results": false,
  "items": [
    {
      "id": 3081286,
      "node_id": "MDEwOlJlcG9zaXRvcnkzMDgxMjg2",
      "name": "Tetris",
      "full_name": "dtrupenn/Tetris",
      "owner": {
        "login": "dtrupenn",
        "id": 872147,
        "node_id": "MDQ6VXNlcjg3MjE0Nw==",
        "avatar_url": "https://secure.gravatar.com/avatar/e7956084e75f23
        "gravatar_id": "",
        "url": "https://api.github.com/users/dtrupenn",
        "received_events_url": "https://api.github.com/users/dtrupenn/re
        "type": "User"
      },
      "private": false,
      "html_url": "https://github.com/dtrupenn/Tetris",
      "description": "A C implementation of Tetris using Pennsim through
      "fork": false,
      "url": "https://api.github.com/repos/dtrupenn/Tetris",
      "created_at": "2012-01-01T00:31:50Z",
      "updated_at": "2013-01-05T17:58:47Z",
      "pushed_at": "2012-01-01T00:37:02Z",
      "homepage": "",
      "size": 524,
      "stargazers_count": 1,
      "watchers_count": 1,
      "language": "Assembly",
      "forks_count": 0,
      "open_issues_count": 0,
      "master_branch": "master",
      "default_branch": "master",
      "score": 10.309712
    }
  ]
}
```

Finally, it is time to write our first Unit Test for the project. Let's create new file from Unit Test template (File -> New -> File -> Unit Test Case Class). Xcode can identify test methods by the name. It should start with text prefix. Here is a sample Unit Test on SearchResponse.

```
import XCTest
@testable import Github

class GithubTests: XCTestCase {
    func testSearchResponse() throws {
        guard
            let path = Bundle(for: self).path(forResource: "search", ofT
            else { fatalError("Can't find search.json file") }

        let data = try Data(contentsOf: URL(fileURLWithPath: path))
        let response = try JSONDecoder().decode(SearchResponse.self, fro

        XCTAssertEqual(response.totalCount, 40)
        XCTAssertTrue(response.incompleteResults)
        XCTAssertEqual(response.items.count, 1)

        let repo = response.items.first

        XCTAssertEqual(repo.id, 3081286)
        XCTAssertEqual(repo.forksCount, 0)
        XCTAssertEqual(repo.name, "Tetris")
        XCTAssertFalse(repo.fork)

        let owner = response.item.first.owner

        XCTAssertEqual(owner.login, "dtrupenn")
        XCTAssertEqual(owner.id, 872147)
        XCTAssertEqual(owner.gravatarId, "")
    }
}
```

The important thing here is @testable import, which makes possible to access to internal fields of SearchResponse inside the Testing target. By importing XCTest, we get the XCTestCase, which is base class for all of our tests. XCTest framework also includes a bunch of helper methods to assert values. I didn't assert every field to keep it as short as possible, but in real project it is nice to have all fields covered. Now we can run our tests by pressing CMD + U and check the result.

Conclusion

Today we discussed how to start with Unit Testing in any project which has a Model layer. I think it is the most comfortable place to start. Don't hesitate and start today, you will see a lot of benefits like safe refactoring, keeping codebase stable during adding new features which can break something that you have working before, and much more.

Feel free to follow me on Twitter and ask your questions related to this post. Thanks for reading and see you next week!

Recent posts

Styling custom SwiftUI views using environment 09 Dec 2020
Focus management in SwiftUI 02 Dec 2020
Commands in SwiftUI 24 Nov 2020

Hi there! My name is Majid.
I'm Swift developer 🙇 SwiftUI addicted 🚀
Creator of CardioBot and NapBot apps.
Feel free to follow me on Twitter or Github.