

 **audreywelch / ios-astronomy-unit-testing**
forked from [LambdaSchool/ios-astronomy-unit-testing](#)

No description, website, or topics provided.

Edit

[Manage topics](#)

🕒 5 commits

🌿 1 branch

📦 0 releases

👤 1 contributor

Branch: master ▼

New pull request

Create new file

Upload files

Find file

Clone or download ▼

This branch is even with LambdaSchool:master.

 Pull request  Compare



armadsen Add README

Latest commit `a03c6cc` on Sep 17, 2018

📁 Astronomy.xcodeproj	Remove image filtering code	5 months ago
📁 Astronomy	Remove image filtering code	5 months ago
📄 .gitignore	Add starter project	5 months ago
📄 README.md	Add README	5 months ago
📄 TestJSON.swift	Add README	5 months ago

📄 README.md



Astronomy Unit Testing

The goal of this project is practice writing unit tests, including the use of dependency injection, protocol oriented programming, mocking, and expectations. This project covers the concepts covered in the Unit Testing III - Dependency Injection and Mocking module. After completing the lesson material and this project, you should be able to:

- write types and functions that support dependency injection for testing
- use protocol oriented programming to create mocks for testing code with complex dependencies
- use expectations to test asynchronous code

Part 0 - Fork and Clone (Optional)

This repo contains a completed copy of the Astronomy project. You are welcome to use your own code written for the last module for this project if you wish. Otherwise, fork and clone this repo.

Part 1 - Prepare To Write Tests

Determine What to Test

Before you begin writing tests, you should determine what functionality you'd like to test, and how. Remember that unit testing should focus on small, independent parts of your app. Make a list of tests you'd like to write. You can do this in note form, or by creating skeleton test case classes with empty (but well-named) test methods.

Be sure that you test the networking parts of the app, paying particular attention to testing `MarsRoverClient`.

Add a Test Target

Add a unit test target to your project, following the steps learned in [Unit Testing I](#).

Refactor MarsRoverClient to Improve Testability

`MarsRoverClient` is where all the networking code in the project lives. Before you can write effective tests for it, you'll need to refactor it support dependency injection via a protocol-conforming object.

1. Create a new Swift file called `NetworkDataLoader.swift`.
2. Inside `NetworkDataLoader.swift`, define a protocol called `NetworkDataLoader`.
3. It should include two methods, `func loadData(from request: URLRequest, completion: @escaping (Data?, Error?) -> Void)`, and `func loadData(from url: URL, completion: @escaping (Data?, Error?) -> Void)`.
4. Add an extension to `URLSession` to add conformance to `NetworkDataLoader`. Implement both `loadData()` variants.
5. In `MarsRoverClient`, add a constant property called `networkLoader` of type `NetworkDataLoader`.
6. Add an initializer to `MarsRoverClient` that takes a `NetworkDataLoader` object. Give the initializer's argument a default value of `URLSession.shared`. This way, `MarsRoverClient` will continue to function as always in existing code, but test code can provide (inject) a different network loader.
7. Update all `MarsRoverClient` methods to use the `networkLoader` property instead of obtaining `URLSession.shared` directly. If you're using the starter code, only one method, `fetch<T>()` needs to be changed.
8. Build and run the app and verify that it still works as it did before.

Part 2 - Write Tests

Write tests for the app. The instructions here are focused on tests for `MarsRoverClient`, because it requires the application of the new concepts learned in this lesson (dependency injection, mocking, expectations), but you can and should test as much of your code as you're able.

1. In the unit test target, if you haven't already done so, create a test case class called `MarsRoverClientTests`.
2. Add test functions for the things you want to test. You should at least test `fetchMarsRover()` and `fetchPhotos()`.
3. Create a `MockLoader` struct that conforms to the `NetworkDataLoader` protocol.
4. Add properties for `data` and `error` so that the struct can be initialized with explicitly known data and/or an error.
5. Implement the two `loadData()` functions. They should call the passed completion closure with the values of object's `data` and `error` properties. Make them call the completion closure asynchronously on the global background queue.
6. In your tests, create a `MockLoader` instance with appropriate data or an error (depending on the test), and pass it into the initializer for `MarsRoverClient`. If you need test JSON, you can find it in `TestJSON.swift` in this repo.
7. In the completion closure passed to the client being tested, use assertions to ensure that the parameters passed to the completion closure behave as expected.
8. Use expectations to ensure that results from the (asynchronously called) closure passed to `MarsRoverClient`'s `fetch` methods are reported.

Part 3 - Verify Tests Pass

When you're finished, make sure your project passes all the tests you wrote. Your project **must** pass its tests to be considered complete.

Go Farther

If you finish early or want to challenge yourself, think about completing one or more of the following tasks:

- Add tests for `FetchPhotoOperation`. Note that it can't use the existing `NetworkDataLoader` protocol due to its need for the ability to cancel its data task. How would you test cancellation?
- Add tests for `PhotoDetailViewController`, particularly its `save()` method. Should these be UI tests, or unit tests, or both?