Better Unit Testing (Part 2)

Learn how to mock dependencies and save time during testing

Dhananjay Trivedi
Dec 9, 2019 · 6 min read ★

Photo by Mometrix Test Prep on Unsplash

This story is a continuation of the *Better Unit Testing* series where we have already covered the basics, check it out before you read this one.

Better Unit Testing — Part 1

Reading this without any prior knowledge of unit testing? That's even better, let's get started

medium.com

Here is what you will learn from this piece:

1. What are dependencies?
2. How to test classes that have dependencies?
3. A domain-independent example to clear things further.
4. Important learning that you should definitely know if you are just getting started with unit testing.
5. How to wrap up your work after you have successfully completed unit testing.

. . .

What Are Dependencies?

In our projects, being professional developers, we maintain good separation of logic and responsibility by building different classes that interact/communicate with each other to fulfill our requirements while trying to keep our system clean and maintainable.

At times, when you are testing a class that is dependent on other classes, how are you supposed to test that class?

. . .

How To Test Classes With Dependencies?

If we run the whole project, all the required classes are set up and we can test things we do usually but that can be time-consuming (first the build time of the whole project, then setting up dependent classes during runtime).

Hence, we would be wasting a lot of time setting up classes that we don't care about at the moment, just to test the class we care about, at that moment.

Mocking is the answer. We mock our dependent classes like they are already set up with some dummy data just to see if our required class is working properly or not.

. . .

An Example

Let's take the simplest example, which is always used in most of our projects — the login screen. Let's write the test for our login system but, first, let's understand the components involved in our login system.

- We have `LoginSync` which is responsible for syncing our device after login, depending upon the server response, which sounds like a long-running, thread-blocking process (which it actually is, but we won't care about blocking for now).
- `LoginApiSync` manages networking with APIs and getting a response.
- `AuthTokenCache` is responsible for saving your authentication token received from the server on the device.
- There is only one function, called `loginSync()`, that takes the username and password and returns `LoginResult` and then, depending on the `LoginResult`, it either caches the `AuthToken` or leaves it unchanged.

```
class LoginSync {
    private var loginApiSync: LoginApiSync
    private var authTokenCache: AuthTokenCache

    constructor(
        loginApiSync: LoginApiSync,
        authTokenCache: AuthTokenCache
    ) {
        this.loginApiSync = loginApiSync
        this.authTokenCache = authTokenCache
    }
```

```
12
13      fun loginSync(username: String, password: String) {
14          // Don't see the implementation of function you are writing the Unit Tests for.
15      }
16  }
17
18  /**
19   * Sealed class in Kotlin to manage the return type, cleanly.
20   */
21
22  sealed class LoginResult {
23      class SUCCESS(var authToken : String) : LoginResult()
24      object FAILURE : LoginResult()
25      object NETWORK_FAILURE : LoginResult()
26  }
27
28  /**
29   * Interfaces
30   */
31
32  interface LoginApiSync {
33      fun getServerResponse(username: String, password: String): LoginResult
34  }
35
36  interface AuthTokenCache {
37      fun getAuthToken(): String
38      fun saveAuthTokenInDevice()
39  }
40
41  /**
42   * Classes our LoginSync is depedent on which will be implementing the interfaces defir
43   */
44
45  class LoginHttpEndPointSyncImpl : LoginApiSync {
46      override fun getServerResponse(username: String, password: String): LoginResult {
47          // Some networking logic here...
48          // When success full we will return the following
49          return LoginResult.SUCCESS("AuthToken")
50      }
51  }
52
53  class AuthTokenCacheImpl : AuthTokenCache {
54
55      override fun saveAuthTokenInDevice() {
56          // Logic to save auth token to device
57      }
58
59      override fun getAuthToken(): String {
```

```
59      override fun getAuthToken(): String {
60          return "auth_token"
61      }
62
63  }
```

LoginSync.kt hosted with ❤ by GitHub view raw

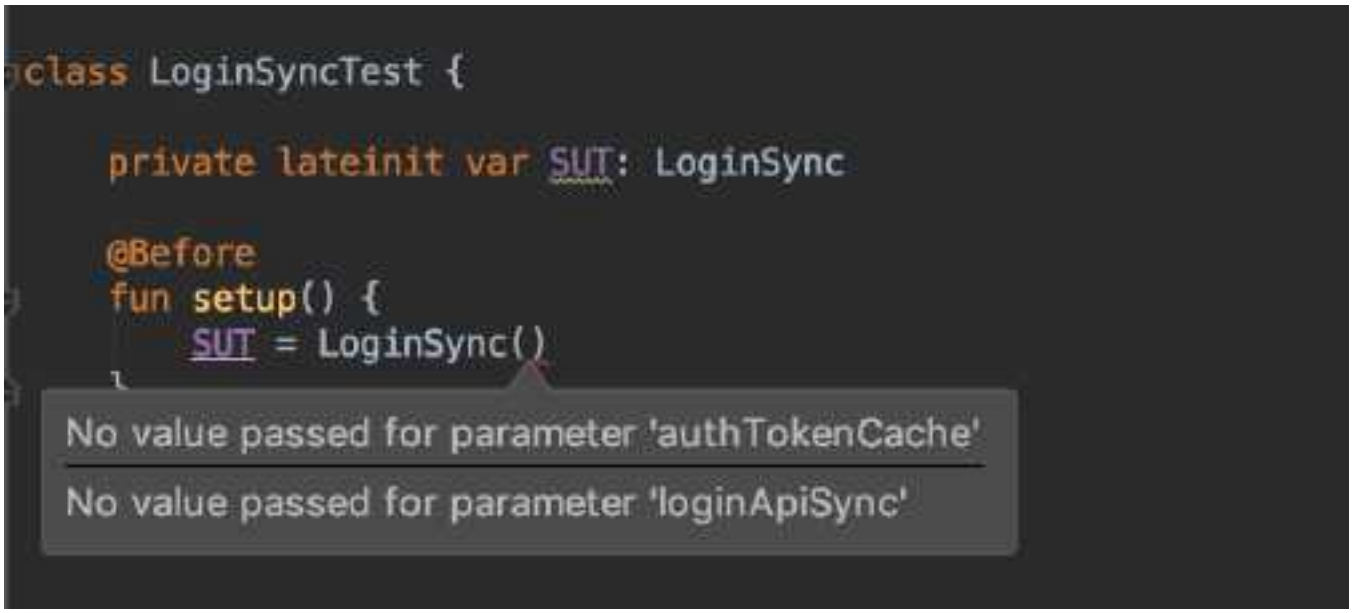
Source: https://gist.github.com/devDeeJay/a1fd7fd702c508680911818d04cd732854

. . .

Let's Start Writing the Test for LoginSync

1. Create a `LoginSyncTest` file and a `LoginSyncTest` class inside it. A good variable name to easily identify the system under test would be `SUT`, hence, naming our `LoginSync` reference variable `SUT`.

2. Usually, `setup()` is where we instantiate the instance of the class we want to test but here, we can't do it directly as it requires instances of two other classes as well. As you can see, we have a red error saying it expects two parameters.



3. Let's substitute the dependencies required, these substituted dependencies are called *test doubles*.

There are different types of test doubles like Fake, Mock, and Stub. We will understand each of them later.

To create `TestDoubles` we create classes that implement the same interface of `LoginApiSync` and `AuthTokenCache` but the implemented logic will be a dummy one.

Creating test double classes

```
class LoginApiSyncTd : LoginApiSync {
    lateinit var username: String
    lateinit var password: String

    override fun getServerResponse(username: String, password: String): LoginResult {
        this.username = username
        this.password = password

        return LoginResult.SUCCESS( authToken: "authToken")

    }
}

class AuthTokenCachingTd : AuthTokenCache {

    var authTokenFromApi: String = ""

    override fun getAuthToken(): String {
        return authTokenFromApi
    }

    override fun saveAuthTokenInDevice() {

    }

}
```

Now, we can instantiate our SUT by doing:

```
class LoginSyncTest {

    private lateinit var SUT: LoginSync

    // Test Doubles
    private lateinit var loginApiTd: LoginApiSyncTd
    private lateinit var authCacheTd: AuthTokenCachingTd

    @Before
    fun setup() {
        // initializing Test Doubles
        loginApiTd = LoginApiSyncTd()
        authCacheTd = AuthTokenCachingTd()

        // initialising System Under Test with the Test Doubles
        SUT = LoginSync(loginApiTd, authCacheTd)
    }
```

Now, let's write our first test function. But that's another challenge.

4. How will you test the `loginSync()` function without even knowing what it does? We certainly can't read the implementation of the function because then we treat a potential bug as a feature.

The other way is to just think, hit, and trial all potential scenarios, which is the point of unit testing, isn't it? So, let's do that.

Let's think of all the use cases for the `LoginSync` class.

- Username and password are passed to the networking API.
- If login succeeds, `AuthTokenId` is updated and cached.
- If login fails, `AuthToken` shouldn't change.

Guess that's all!

5. Let's write each of those test cases.

A good practice is to add comments of all the test scenarios and then write the function for each one of them.

```
1  import org.hamcrest.CoreMatchers.`is`
2  import org.junit.Assert.assertThat
3  import org.junit.Before
4  import org.junit.Test
5
6  /**
7   * Constants which will be shared amongst classes
8   */
9
10 const val USERNAME = "USERNAME"
11 const val PASSWORD = "PASSWORD"
12 const val AUTH_TOKEN = ""
13 var isUserLoginError = false
14
15 class LoginSyncTest {
16
17     private lateinit var SUT: LoginSync
18
19     // Test Doubles
20     private lateinit var loginApiTd: LoginApiSyncTd
21     private lateinit var authCacheTd: AuthTokenCachingTd
```

```
    private lateinit var authCacheTd: AuthTokenCachingTd

    @Before
    fun setup() {
        // initializing Test Doubles
        loginApiTd = LoginApiSyncTd()
        authCacheTd = AuthTokenCachingTd()

        // initialising System Under Test with the Test Doubles
        SUT = LoginSync(loginApiTd, authCacheTd)
    }

    // Username and password are passed to the networking API
    @Test
    fun loginSync_success_usernameAndPasswordIsSentToApi() {
        SUT.loginSync(username = USERNAME, password = PASSWORD)
        assertThat(loginApiTd.username, `is`(USERNAME))
        assertThat(loginApiTd.password, `is`(PASSWORD))
    }

    // If Login succeeds, AuthTokenId must be updated and cached
    @Test
    fun loginSync_success_authTokenIsCachedAndUpdated() {
        SUT.loginSync(USERNAME, PASSWORD)
        assertThat(authCacheTd.authTokenFromApi, `is`(AUTH_TOKEN))
    }

    // If Login fails, AuthToken shouldn't change.
    @Test
    fun loginSync_loginFails_authTokenNotChanged() {
        isUserLoginError = true
        /**
         * Checking if authToken remains Empty
         */
        assertThat(authCacheTd.getAuthToken(), `is`(""))
    }

}


class LoginApiSyncTd : LoginApiSync {
    lateinit var username: String
    lateinit var password: String

    override fun getServerResponse(username: String, password: String): LoginResult {
        this.username = username
        this.password = password

```

```
69          return if (isUserLoginError) {
70              /**
71               * When userloginError happens, we set no authToken hence authTokenRemainsl
72               */
73              LoginResult.FAILURE
74          } else {
75              /**
76               * When everything is good, we return SUCCESS and set the authToken
77               */
78              LoginResult.SUCCESS("authToken")
79          }
80
81      }
82  }
83
84  class AuthTokenCachingTd : AuthTokenCache {
85
86      var authTokenFromApi: String = ""
87
88      override fun getAuthToken(): String {
89          /**
90           * Just giving some value in return for now
91           */
92          return authTokenFromApi
93      }
94
95      override fun saveAuthTokenInDevice() {
96
97      }
98
99  }
```

Then, you can go ahead and run your `Test` class after writing all the tests and see the result for your own Classes.

In Android Studio, we get this button popping up and we get three options:

- Run — To run all your tests.

- Debug — To debug and observe the behavior of your test class.
- Run with coverage — This tells you about your code coverage. The % of lines and the number of methods you are covering in your tests. Super helpful. We should always aim for 100% as we want 100% of components to work.

. . .

Important Learning

1. Use instance methods over static methods

You should try as much as possible not to write static methods in classes as they become much harder to unit test.

Hence, try to write instance methods and access them using the object of the class, this way, you can always mock an object of the class to test its methods.

2. Singletons are your enemy in unit testing

When you have multiple unit tests and multiple of those unit tests are accessing a singleton, and when you run each of those functions one-by-one, they all might pass but if you run them all together then you will face conflicts.

In some situations, any of those multiple tests will fail randomly! Hence, if you sit down to debug them, you are in for a ride.

It's because the singleton is being shared, a cross-over if you will, which violates a principle of unit testing — your tests should run independently of each other.

The bottom line, don't use singletons if you want to keep your code unit test and developer-friendly.

3. Learn about testing frameworks and libraries

Learn about testing frameworks that will make your life easier.

For example, if you are an Android developer, Mockito is a great library to help you mock your test doubles to make your life easy and productive as a developer.

. . .

Wrapping Up Your Work

I can't talk enough about the importance of clean code and that includes both your production code as well as your test codebase. If you let your test codebase rot, your production code will rot too!

Hence, clean up your unit test code by removing all the unnecessary comments, white spaces, maybe think of better, more descriptive, unambiguous names for your variables, functions, and classes.