# Automated Testing for Android & iOS

KARUMI

# 1. Introduction

First things first: congratulations! You just took the first step in order to become a better developer, to be able to deal with code in a smarter, more resourceful way.

After all, that's why you're here!
To learn how to create rock solid mobile apps.

And the first rule
we would like us to introduce you to is this:

Test automation is the key to successful
software development

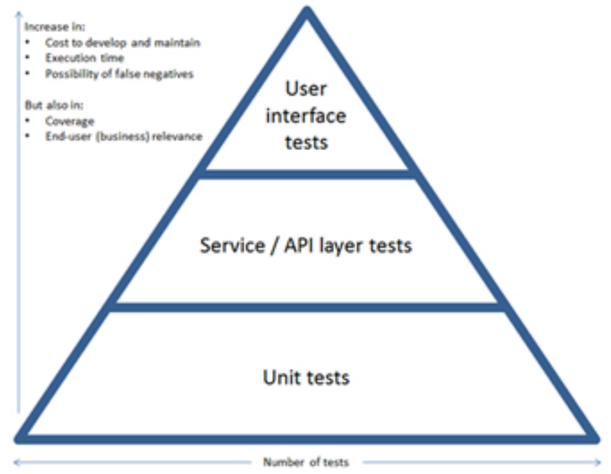
# 2. Needs

KARUMI

When thinking about automatization, the first thing is going to cross your mind is "I need to start writing some tests for mobile applications". Hang in there for a second! There are some things you need to consider before. If you happen to have limited testing knowledge, you could focus on acceptance and unit tests using the most common frameworks, a simple test runner and a mocking library.

But  then you will probably run into some problems.

# 3. Problems




- It could happen that you have no idea at all on what to test and how to test it.
- Or, maybe, it is the code itself, that it was not yet ready to be tested.
- It could be the case that, like us, you're obsessed with Mike Cohn's Test Pyramid without thinking about the type of software you were writing.
- Also, just because your tests were passing didn't mean the code was working.

# 4. Our testing development pipeline

```
public class RenewSessionTest extends ApiClientTest<BaseApiClient> {

  @Test public void shouldKeepTheUserLoggedInAfterRenewSession() {
    performLogin();
    enqueueUnauthorizedResponse();
    enqueueUnauthorizedResponse();
    enqueueSessionRenewedResponse();
    enqueueResponse();

    HttpRequest request = HttpRequest.Builder.to(ANY_ENDPOINT).get();
    getApiClient().send(request, AnyResponse.class);

    assertTrue(getApiClient().isUserLoggedIn());
  }

  @Test public void shouldSendRequestAfterRenewSessionUsingTheNewAccessToke
    performLogin();
    enqueueUnauthorizedResponse();
    enqueueSessionRenewedResponse();
    enqueueResponse();

    HttpRequest request = HttpRequest.Builder.to(ANY_ENDPOINT).get();
    getApiClient().send(request, AnyResponse.class);

    assertRequestContainsHeader(ACCESS_TOKEN_KEY, NEW_ACCESS_TOKEN);
  }
}
```

In order to do things the right way, you need a testing development pipeline. However, that is not always the first thing you must fix in order to improve your project quality.

Define what to test and how; that's the key. Also, consider this:

- What are the tools to use and why?
- What is the scope of our tests?

Another must-have is a testable code, so you can be confident enough to write your own tests.

# 5. Testing the business logic independently of the framework or library

Initially, our business logic was completely coupled to the framework. Wrong idea!

But, what is business logic (BL)?

We're talking about that portion of an enterprise system that determines the way data is calculated and transformed. You could use it, for example, to determine how a tax total was calculated from invoice line items. Moreover, BL also determines how data is routed to people or software systems, aka workflow.

Said that, you'll realize how important is checking if the BL is indeed implementing the predefined product requirements. Isolating the code you want to test and simulate different initial scenarios is essential to configure the behaviour of some components at runtime.

After, you should test the code by choosing the parts you'd like to exercise. Once completed, it is necessary to check the state of the software so you can know for sure that everything is correct after exercising the subject under test.

The key to this testing approach is the Dependency Inversion Principle.
If you write code that depends on abstractions,  you will be able to separate your software in different layers. In order to obtain instances of collaborators, you need to create code for some of your software.
Later, if you use test doubles, you will manage to create a simulation of both the behaviour and the state of the production code replaced with this test.

Also, remember that using a Service Locator or a Dependency Injection framework will help you to reduce all the boilerplate needed to apply Dependency Inversion.
However, these are not mandatory.
Said all this, if you do your homework correctly, you will be able to test your business logic no matter of the framework or the library used for the production code.

# Tools

```
public class GameBoyBIOSExecutionTest {

  @Test
  public void shouldIndicateTheBIOSHasBeenLoadedUnlockingTheRomMapping() {
    GameBoy gameBoy = givenAGameBoy();

    tickUntilBIOSLoaded(gameBoy);

    assertEquals(1, mmu.readByte(UNLOCK_ROM_ADDRESS) & 0xFF);
  }

  @Test
  public void shouldPutTheNintendoLogoIntoMemoryDuringTheBIOSThirdStage() {
    GameBoy gameBoy = givenAGameBoy();

    tickUntilThirdStageFinished(gameBoy);

    assertNintendoLogoIsInVRAM();
  }

  private GameBoy givenAGameBoy() {
    z80 = new GBZ80();
    mmu = new MMU();
    gpu = new GPU(mmu);
    GameLoader gameLoader = new GameLoader(new FakeGameReader());
    GameBoy gameBoy = new Gameboy(z80, mmu, gpu, gameLoader);
    return gameboy;
  }

}
```

- https://github.com/Quick/Nimble
- https://github.com/apple/swift-corelibs-xctest
- https://github.com/mockito/mockito
- https://github.com/junit-team/junit4

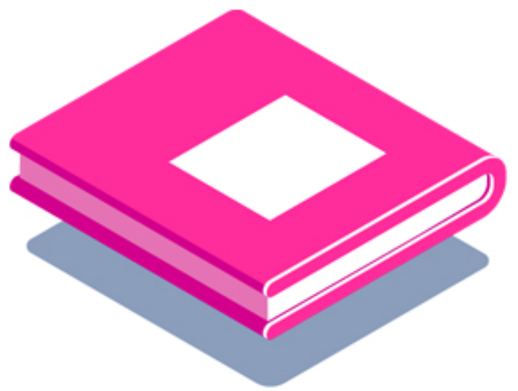
# 6. Testing the integration with a server side API

When dealing with server side API integrations, you must know if our client's side code is sending the correct messages to the API; parsing the API responses correctly. Also, it is important that the code is implementing the authentication mechanisms correctly. And finally, at least but not last, it should be handling API errors properly.

For all this, it is essential to simulate different server side responses. Furthermore, you have to perform assertions over the client side request. To achieve these, you should perform a test double known as mock and, also, a third party tool known as MockWebServer.

Later three of the most interesting points in anyAPI client must be taken into consideration:

- **The authentication mechanism.**
- **The renew session process**
- **The JSON parsing process.**
- **We send the expected information to the API**



In these tests we can see how the tested subject is the API client. Also, remember that you must configure some HTTP responses before you exercise the API client. Later, you should use more complicated based on the same approach. For that, you have to preconfigure responses using, as the body, the contents of a JSON file stored in the test resources directory.

# Tools

```
public class CartApiClientTest extends ApiClientTest<CartApiClient> {

  @Test public void shouldObtainCartFromTheCartEndpoint() {
    enqueueResponse(GET_CART_RESPONSE_FILE);

    CartDTO cart = givenACartApiClient().getCart();

    assertCartContainsExpectedValues(cart);
  }

  @Test public void shouldParseAddToCartResponse() {
    enqueueResponse(ADD_TO_CART_RESPONSE);

    CartDTO cart = givenACartApiClient().addToCart(ANY_SKU_ID);

    assertCartContainsExpectedValuesAfterAddAnItem(cart);
  }

  @Test public void shouldParseUpdateLineResponse() {
    enqueueResponse(UPDATE_CART_LINE_RESPONSE_FILE);

    CartDTO cart = givenACartApiClient().updateLine(ANY_SKU_ID, 3);

    assertCartContainsExpectedValuesAfterLineUpdated(cart);
  }
}
```

- https://github.com/AliSoftware/OHHTTPStubs
- https://github.com/square/okhttp/tree/master/mockwebserver

# 7.Testing the application User Interface

What key factors you have to consider in order to deal  with an application user interface? First, once the UI is loaded, you have to figure out if your code is showing the right information to the user. Does your code show the correct messages to the user as the result of a user interaction? And finally, make sure that your code is displaying the correct screens given a user interaction.

For you to achieve these three factors, it is necessary to start a valid environment, in order to execute your tests for which it can be used both an Android emulator or a device using Android as OS. When you have done that, you should perform different actions in the application user interface. Moreover, do not forget to perform assertions over the information shown to the user.

Again, to fully control the test scenario you need use test doubles in order to exercise the subject under test. Doing this, by reducing the test scenario and improving the determinism of our tests, you will decrease their flakiness.
Two tools can be used in this scenario:

- **Any dependency injector, as a core component to replace production code with test doubles**
- **Espresso, to interact with the device and perform assertions.**

Again, it is important for you to remember that these tests are possible thanks to the usage of testable code. What's the meaning of this? The Dependency Inversion Principle usage must be taken into consideration.

# Tools

```
@Mock SuperHeroesRepository repository;

@Test public void showsEmptyCaseIfThereAreNoSuperHeroes() {
   givenThereAreNoSuperHeroes();

   startActivity();

   onView(withText("¯\\_(?)_/¯")).check(matches(isDisplayed()));
}

@Test public void showsSuperHeroesNameIfThereAreSuperHeroes() {
   List<SuperHero> superHeroes = givenThereAreSomeSuperHeroes(ANY_NUMBER_OF_SUPER_HEROES);

   startActivity();

   RecyclerViewInteraction.<SuperHero>onRecyclerView(withId(R.id.recycler_view))
         .withItems(superHeroes)
         .check(new RecyclerViewInteraction.ItemViewAssertion<SuperHero>() {
            @Override public void check(SuperHero superHero, View view, NoMatchingViewException e) {
               matches(hasDescendant(withText(superHero.getName()))).check(view, e);
            }
         });
}


private void givenThereAreNoSuperHeroes() {
   when(repository.getAll()).thenReturn(Collections.<SuperHero>emptyList());
}
```

- https://github.com/kif-framework/KIF
- https://developer.android.com/training/testing/espresso
- https://github.com/karumi/shot
- https://github.com/uber/ios-snapshot-test-case
- https://github.com/InsertKoinIO/koin
- https://github.com/google/dagger

# 8 .Conclussions


Have you been able to successfully finished your Testing Development Pipeline?

"With all this information you should have an idea on what to test and how to do it. From testing your BL using plain old unit tests, to testing your API integration using stubbed HTTP responses or testing the application user interface using test doubles to simulate any possible scenario.

KARUMI

It has been our pleasure to share with you all this best practices with you!
By know, you should be able to easily attend to one of our hands-on training to achieve higher grounds in software development and to become a valuable professional eligible for exciting projects and job positions in this field.

- http://karumi.com/training

# Resources and Exercises

In the following repositories you'll find a set of awesome exercises ready to practice and improve your testing skills. From unit testing to integration or UI testing, solving these repositories proposed exercises you'll be able to master automated testing for Android and iOS.

- https://github.com/Karumi/KataContactsJava
- https://github.com/Karumi/KataContactsSwift
- https://github.com/Karumi/KataTODOApiClientiOS
- https://github.com/Karumi/KataTODOApiClientKotlin
- https://github.com/Karumi/KataSuperHeroesIOS
- https://github.com/Karumi/KataSuperHeroesKotlin