TDD for Angular Application

A step-by-step blueprint for the confused souls

Kamal Raj Sekar

CONTENTS

PREFACE

Welcome to *TDD for Angular*. In this book, I will provide you with a radical and systematic way of building angular applications using Test Driven Development which can be applied in your daily job using an easy-to-follow blueprint.

It is not that difficult to learn about the definition of 'red-green-refactor' from Wikipedia or some other source. But in order to apply TDD in a real-life project requires much more specific knowledge than the definition of 'red-green-refactor'. This book intends to offer you that.

Although we will look at the various benefits of TDD, convincing you to follow TDD is not in the scope of this book. Furthermore, this book is not an in-depth tutorial to build an angular application with all the features in angular rather it would only use the minimal features(Module, Component, and Service with Dependency Injection) required to build an example angular application to demonstrate how to apply TDD. The test frameworks/libraries used in this book are used just to demonstrate the ideas in building the angular application using TDD, you are free to choose your own frameworks/libraries as per your need.

Audience

This book is meant for

- Front-end developers
- Architects
- Agile coaches/Consultants who aspire to become Technical Agility Expert

How are the ideas structured?

The book is structured into three sections

1. Fundamentals
2. Blueprint for Test-Driving Angular Application
3. Building a real Angular Application using our Blueprint

4. Conclusion

Downloads

I have relied on an example project to show you how to implement these concepts step-by-step using the blueprint and sequenced it in a way to help you easily understand the ideas/information that is shared in this book. All source code mentioned is in `courier-new` font like this to separate it from ordinary text. You can download the source code and run it on your own computer from the repository - https://github.com/trycatchkamal/test-drive-angular-application

Kamal Raj Sekar

PART – I

FUNDAMENTALS

1. TDD FROM A THOUSAND FEET

Let's start with some basic information that is required to understand Test-Driven Development (TDD)

What is Code?

A code is nothing but a set of instructions supplied to a computer in a particular language. Developers typically write this in a programming language like Javascript, Java, Python, C, C++, C#, etc.

What is a Test?

A test is something that verifies if the code is working correctly. Tests can be manual, where a person has to test the application manually by executing certain actions and checking the result of that, or automated, where some special tool or software runs the tests and checks the result automatically.

What is TDD?

TDD stands for Test-Driven Development. It is an approach to develop software in which the development is guided by tests. When following TDD, people always write the test first that defines the desired functionality of the actual code. Then, write the actual code that fulfills the requirements of the test. Finally, run the test to verify that the code is functioning correctly and meets the requirements specified in the test.

Three Rules of TDD

Robert C Martin, better known as Uncle Bob describes(refer to blog) TDD in terms of these three rules.

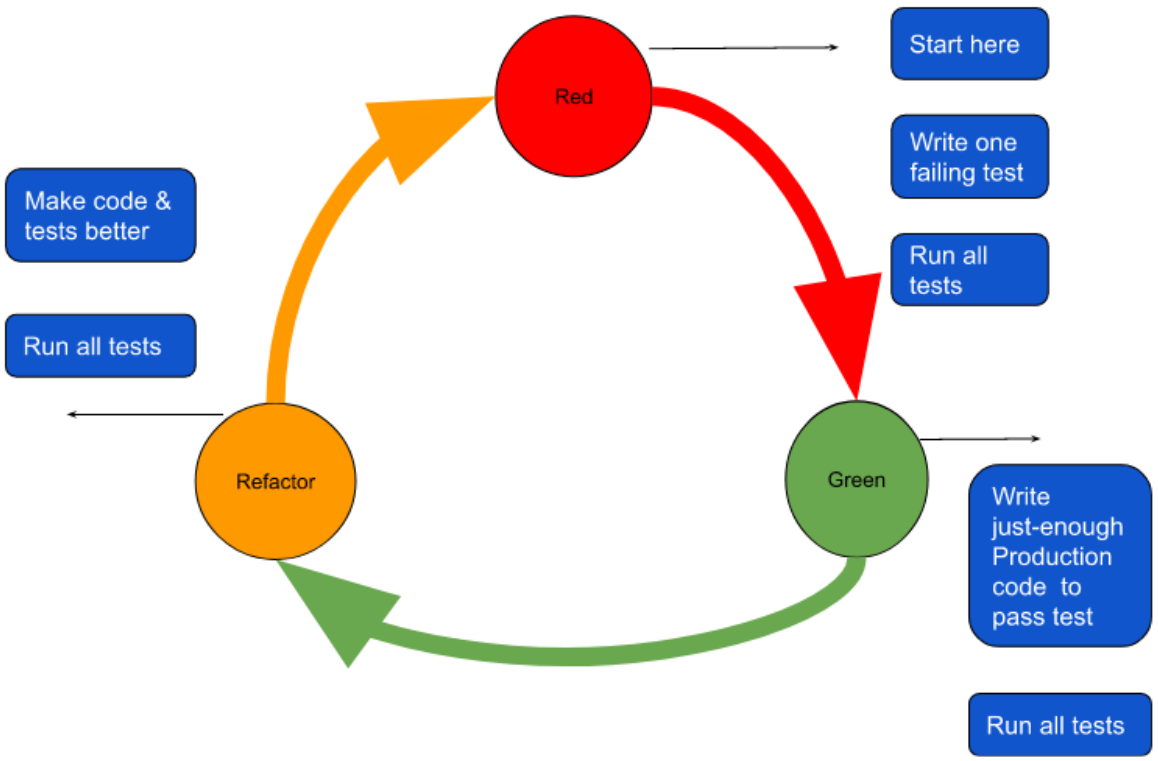
1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Any code that will eventually execute in the production environment is referred to as "production code" (the environment that the real users will use). We never execute test code in a production environment. Only the source code is active in the production environment. Production code, therefore, refers to *source code* in this context.

The Red-Green-Refactor Cycle

We understood the rules of TDD. Now, how do we implement it?



i. Red

First step is to write a simple test. Since at the moment of writing the test, the corresponding

production code that supports the tested functionality does not exist, the test is intended to fail. Run this test again while also running the earlier tests to see if the test fails. This state is referred to as "red".

ii. Green

Second step is to write just-enough production code - the intended functionality that will correspond to the above test. Run all the tests after writing the production code that satisfies the aforementioned test. All of the tests, including the one we just failed, would succeed if the code is correct. The primary focus in this step is to pass the test and turn it to "green".

iii. Refactor

Now that we know the code functions, try to refactor the code or tests at this stage in order to make them better. When the tests pass, we can be certain that nothing that was already functional has been broken.

2. LEVELS OF TESTS

In the previous chapter we learned that TDD requires us to write the test before writing any production code. As you might be aware, when we begin a project all we have is the set of requirements or user stories.

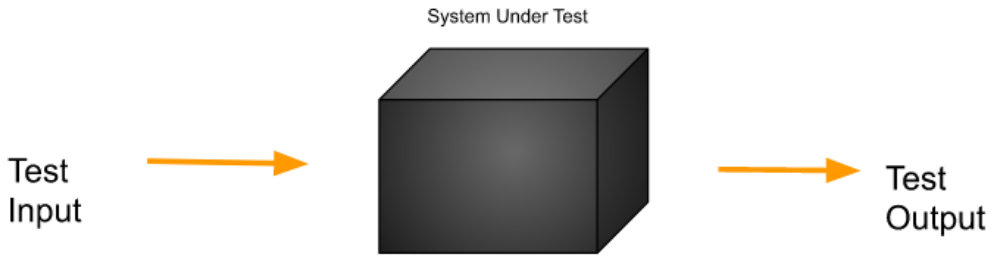
It is possible to write tests at the level of the user story which includes the database, the UI, and a server. At the same time, it is also possible to write a test at the lowest possible level, like just testing a single line of code.

But which test should you prefer? the end-end test or the unit test? In this chapter, we will get clarity about this.

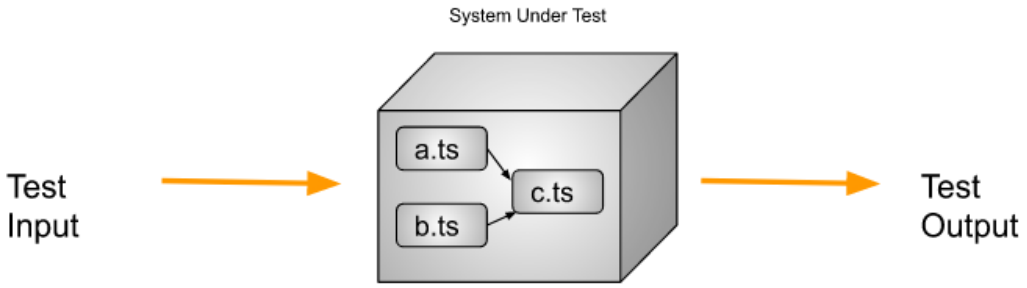
Classification of Tests

Software tests can be classified in various ways depending on the specific context and goals of the testing. It can be broadly classified into the Black box and White box testing

Black box testing - Test has no knowledge about the internals of the system being tested



White box testing - Test has knowledge about the internals of the system being tested



Black box tests

According to Wikipedia, “a Black box is a system which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings”. Hence, Black box tests are those that test the system without knowing any inner details or workings of the system.

Example: Suppose you are testing the software in the way that a user would, without knowing anything about the internal architecture or design of the software.

White box tests

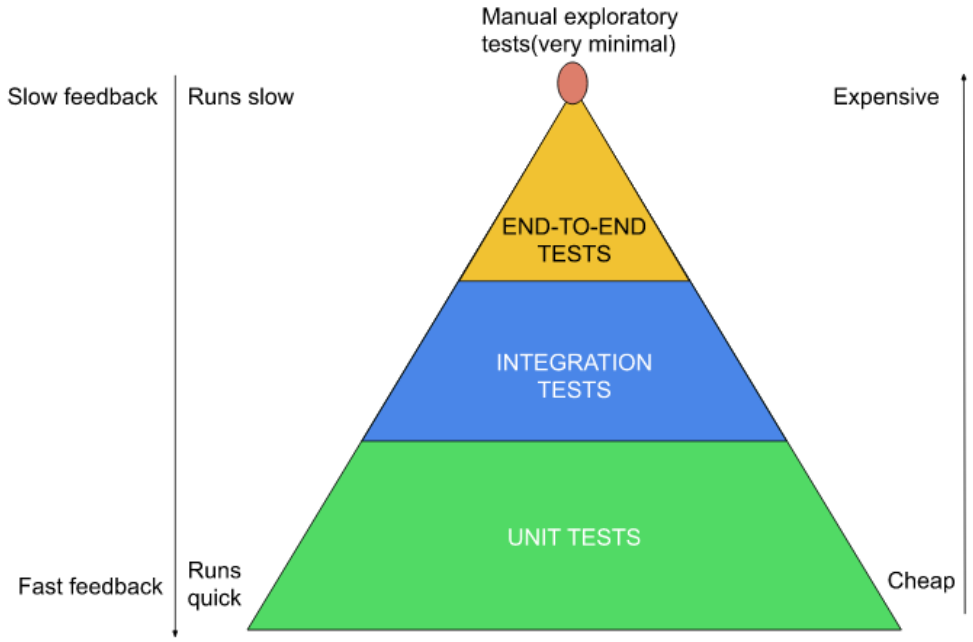
On the contrary to Black box tests, White box tests are those that are written using knowledge about the inner workings or the architecture of the software.

Example: Suppose you have a library management web application that shows you the available books in a library in the UI. If you have to test the API which is used to fetch the information from the database, one should know the details about this particular API used from the UI during an interaction. Only then we could create a test that just verifies the correctness of the API's responses to different input combinations.

White box tests don't have to be limited to the API level. They may also be any depth below that. Class-level tests are a typical example of deeper white-box tests. In these tests, we utilize tools like NUnit(.net), JUnit(java), etc., to test a specific class while being aware of the methods it exposes and the intended behavior of those methods.

Another way to classify tests - The Test Pyramid

The black box and the white box tests can be further classified into different levels as shown below. Mike Cohn came up with this concept *Test Pyramid*, in his book *Succeeding with Agile*. Test pyramid consists of three different layers of tests.



End-end tests

Tests that test the software from the user's perspective. These are black box tests.

Integration tests

These are white-box tests that are written to expose faults in the interaction between integrated units within the software that is being tested. Integration testing focuses on determining the correctness of the interface.

Unit tests

These are white-box tests too, that verify the smallest piece of code or functionality, usually by using test doubles in place of the dependencies for the piece of code being tested. Test doubles are objects that are used in place of real objects during testing. They are usually referred to as dummy, fake, mock, and stub objects.

Based on the specific context and the goals of testing, the aforementioned three levels may further be divided into more sublevels by a team. However, the general classification as shown above still holds.

For instance, a team may decide to create many levels of integration tests, such as one that verifies the wiring between the UI and the backend and another that verifies the wiring between the various classes (Service and Controller, for example) in the backend. Even then, both of these levels are still referred to as integration tests.

Shortly, we will understand that it doesn't matter what these layers are named exactly. What matters is that we place our tests in different layers.

Bear in mind that all these three levels of tests are automated. These are not performed manually. We can have a sprinkling on the top called 'Manual exploratory tests' to address cases where we need a human tester

Why not any other shape but a Pyramid for the test classifications?

When all we're talking about is layering the tests, why utilize a pyramid structure? Why not arrange it in any other shape or form? Because we also want to provide guidance for the number of tests in each layer, we structure the layers into a pyramid. Although numerous forms and models have been put out to show the balance between different levels of testing, the test pyramid is a commonly accepted and applied approach.

As we can see in the shape of the pyramid, the large base of the pyramid represents the lower level of testing which is more focused and more frequent testing, and as we go up from the bottom more comprehensive and less frequent testing needs to be written. In other words, we want more unit tests, fewer integration tests, and even fewer end-end tests, as indicated by the shape of the pyramid.

Why run more unit tests?

There are several reasons why it is important to run a large number of unit tests

Unit tests execute the fastest

When we compare unit tests with end-to-end tests, It is the unit tests that execute the fastest because all it needs is the code. Whereas the end-to-end tests need a complete environment--including the server and database—before we start the test.

Unit tests give faster feedback

Why are tests written? We create tests to quickly determine whether the code is behaving as expected or not. The faster the feedback, the better. In that spirit, we'll move the majority of the testing scenarios into the layer of unit tests. We create an integration test for a scenario if the unit test cannot cover it.

Unit tests are low cost

Another aspect is the cost to run the test. Unit tests are practically free as they just need code to run, whereas the end-end/integration tests may need a computer with a database and possibly a working network.

Unit tests help with refactoring

When we make changes to the code, it is important to ensure that the changes do not introduce new defects. Running a large number of unit tests can help to catch any unintended consequences of refactoring and can give us confidence that our changes have not broken any existing functionality

Unit tests improve code quality

By writing a large number of unit tests, we can ensure that the code is well-tested and that it behaves as intended. The higher the quality of individual components the better the overall system resiliency. Thus, resulting in reliable code which improves the overall quality of the code.

Overall, we want to test most of the scenarios at the unit level. Whatever cannot be tested at the unit level, can be moved up to the integration level, acknowledging that these tests will be slower and costlier. Whatever cannot be tested at the integration level, can be moved up to the end-end level.

This will result in a test pyramid that is optimal. Also, for any behavior that cannot be tested at the end-to-end level in an automatic way, like the tests that need human beings (Eg: alignment, look and feel, etc), we do that along with the manual exploratory test, shown as sprinkled on the top of the pyramid