TDD for C# and .Net Core Microservices

A simplified manual for the confused souls

Vivek Ganesan

Kamal Raj Sekar

CONTENTS

PREFACE

This book is written in an attempt to give you a radical, principled approach to learn, practice and build Test Driven Development (TDD) Driven .NET Core Microservices Development that can be applied to your day job.

If you chose this book wondering '*Does TDD differ when done for .NET Core Microservices when compared to other frameworks or technologies?*', the answer is '*No*'.

If you are still wondering, '*Then, do we still need a separate book for this topic?*', we say '*Yes*'.

It is not that difficult to understand the 'red-green-refactor' definition from various sources. However, to apply that practice in a real-life project, you would need more specific knowledge than just 'red-green-refactor'. This book intends to offer you with that.

Also, this book will help you practice these concepts step-by-step in an example project using an easy-to-follow blueprint.

Goals of this book

This book has two goals.

1. Provide a step-by-step blueprint for creating a .Net Core Microservices using TDD
2. Provide code samples that you can execute in your own machine

Audience

You could potentially get some value out of this book, if you are a

- .Net Developer
- Architect
- Agile coach / Consultant who aspire to become Technical Agility expert

What is this book not about?

This book is not going to convince you to follow TDD. Though we talk about benefits in various places, we believe convincing is not in the scope of this book. It is not going to cover

related topics like Branching strategies, Behavior Driven Development, or Continuous Integration. We may talk in bits and pieces about these practices. However, we do not deep-dive on any of these topics.

This book is not going to cover API gateways or containerization aspects of the microservices. We will limit ourselves to just applying TDD to develop the RESTful HTTP endpoints built using .Net Core

How are the ideas structured?

The ideas in this book are structured into three sections.

1. Fundamentals
2. Blueprint for Test-Driving .Net Core Microservices
3. Building a Real Microservice
4. Conclusion

How to read this book?

The sequencing of the book is designed in three sections so that you can align with our definition of certain terms, and to get a good idea about the blueprint before you deep-dive into any chapter of your choice. It will help you appreciate why we do things in certain ways.

You can also refer to the code samples at any point in time, in case you want a quick copy-pasteable sample that you could later modify to fit your context.

You could download the project referred to in this book from the repository(https://github.com/techcoachcircle/test-drive-dotnet-microservices) and run it on your own computer too.

We hope you enjoy reading this book.

Vivek Ganesan
Kamal Raj Sekar

PART - I

FUNDAMENTALS

1. TDD FROM A THOUSAND FEET

Let's start with the some basic information that you require 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 C, C#, Java, Python, etc.

What is a Test?

A test is something that verifies if the code is working correctly. Tests may also be automated.

What is TDD?

TDD, or test-driven development, is a method of developing software in which the development is guided by the tests. In other words, when following TDD, people never develop code without first writing a test.

Three Rules of TDD

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

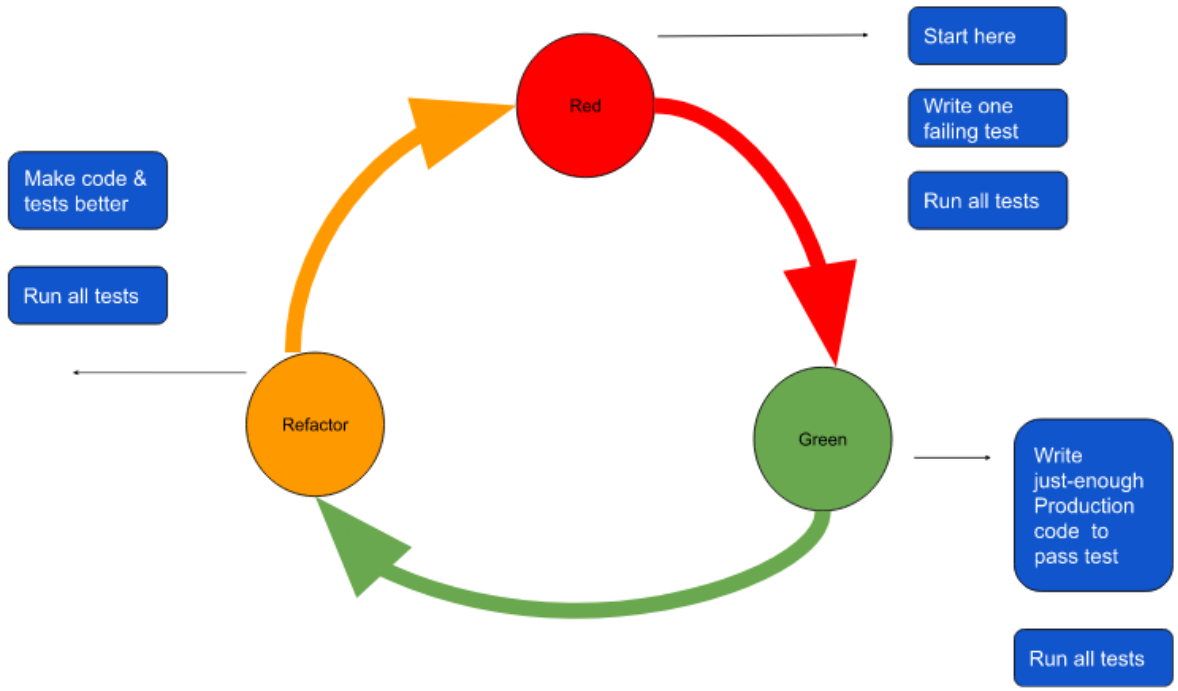
1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to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

Any code that will 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. There, only the source code is active. 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 damaged.

2. LEVELS OF TESTS

We learned in the previous chapter that TDD requires us to write a simple test before writing any production code. But when we actually begin a project, all we have is a requirement or a user story.

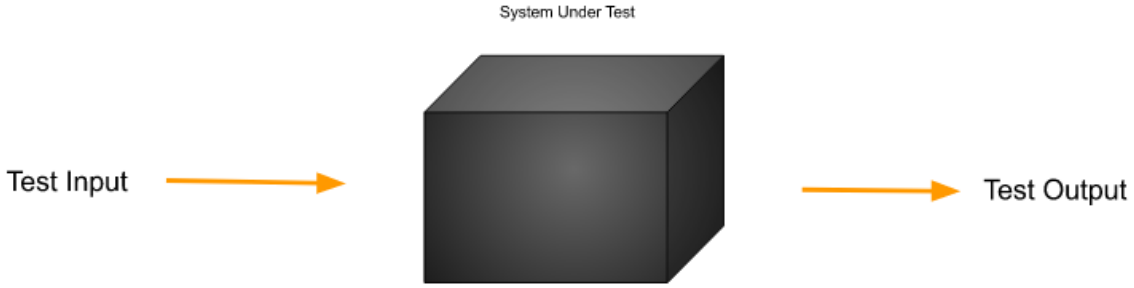
It is possible to write a test at the level of user story including a database, web/mobile UI and a server. At the same time, it is also possible to write a test at the lowest possible level, like just testing a single line of code.

Which of these tests should we prefer? The end-end tests? Or, the unit tests? This chapter seeks to clear this confusion.

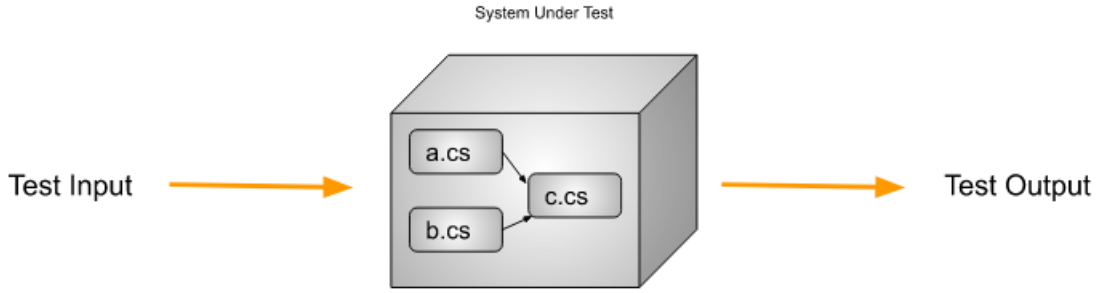
Classification of Tests

Tests can be broadly categorized into two groups - White box and black box tests.

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.

For example, using the software in the way that a user would, without knowing anything about the internal architecture or design of the software.

White box tests

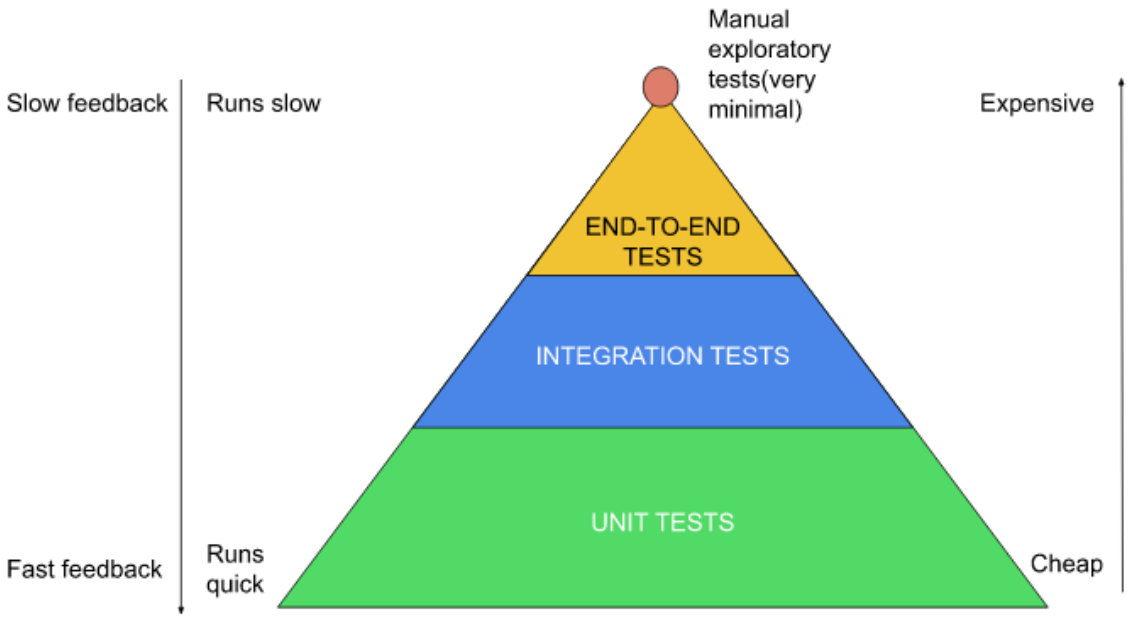
On the contrary to Black box tests, *White box tests* are those that are written using knowledge about the inner workings or the architecture of the software.

For example, if one knows a particular API is used from the UI during an interaction, one 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 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 faking/mocking the dependencies.

Depending on the situation, 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 being

Why not any other shape but a Pyramid?

Why use pyramid if the only thing we are talking about is grouping the tests into multiple layers? Why not group it as rectangles in a stack? Because we also want to provide guidance for the amount of tests in each layer, we structure the layers into a pyramid. The pyramid's base is broad. 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?

Which layer's tests execute the quickest? Unit tests or the end-end tests? It is the unit tests that execute the fastest because all it needs is the code. Whereas end-end tests may need a complete environment—including the server and database—before we start the test.

Why are tests written? To quickly determine whether the code is performing as expected or not, we write tests. *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.

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.

Additionally, having more unit tests has the benefit of isolating the behaviors(function/method) of the particular class and only testing that piece of code. The higher the quality of individual components the better the overall system resiliency. Thus, resulting in reliable code.

Due to the above reasons, 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 in 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 in the end-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, that is shown as sprinkled on the top of the pyramid.