Instructor example

Web University by Harsha Vardhan

Name some Unit Testing benefits for developers that you personally experienced?

- Unit Tests allow you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours.
- TDD helps you to realize when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.
- The tests and the code work together to achieve better code. Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of both being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.
- TDD helps with coding constipation. When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.
- Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.
- Unit Tests give you instant visual feedback, we all like the feeling of all those green lights when we've done. It's very satisfying. It's also much easier to pick up where you left off after an interruption because you can see where you got to - that next red light that needs fixing.
- Contrary to popular belief unit testing does not mean writing twice as much code or coding slower. It's faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing. This is one you'll only believe when you're doing it :)
- I think it was Fowler who said: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all". I interpret this as giving me permission to write tests where I think they'll be most useful even if the rest of my code coverage is woefully incomplete.
- Good unit tests can help document and define what something is supposed to do
- Unit tests help with code re-use. Migrate both your code and your tests to your new project. Tweak the code till the tests run again.

What is Mocking?

Mocking is primarily used in unit testing. An object under test may have dependencies on other (complex) objects. To isolate the behavior of the object you want to replace the other objects by mocks that simulate the behavior of the real objects. This is useful if the real objects are impractical to incorporate into the unit test.

In short, mocking is creating objects that simulate the behavior of real objects.

What is the difference between Unit Tests and Functional Tests?

Unit Test - Testing an individual unit, such as a method (function) in a class, with all dependencies mocked up.

Integration Test - Checking if different modules are working fine when combined together as a group. This will test many methods and may interact with dependencies like Databases or Web Services.

- Integration tests tell what's not working. But they are of no use in guessing where the problem could be.
- Unit tests are the sole tests that tell you where exactly the bug is. To draw this information, they must run the method in a mocked environment, where all other dependencies are supposed to correctly work.

How to unit test an object with database queries?

If your objects are tightly coupled to your data layer, it is difficult to do proper unit testing. Your objects should be persistent ignorant. You should have a data access layer, that you would make requests to, that would return objects.

Then mock out your calls to the database. This way, you can leave that DB dependents part out of your unit tests, test them in isolation or write integration tests.

Should unit tests be written for Getter and Setters?

Properties (getters/setters) are good examples of code that usually doesn't contain any logic, and doesn't require testing. But watch out: once you add any check inside the property, you'll want to make sure that logic is being tested.

In other words, if your getters and setters do more than just get and set (i.e. they're properly complex methods), then yes, they should be tested. But don't write a unit test case just to test a getter or setters, that's a waste of time.

How would you unit test private methods?

If you want to unit test a private method, something may be wrong. Unit tests are (generally speaking) meant to test the interface of a class, meaning its public (and protected) methods. You can of course "hack" a solution to this (even if just by making the methods public), but you may also want to consider:

- If the method you'd like to test is really worth testing, it may be worth to move it into its own class.
- Add more tests to the public methods that call the private method, testing the private method's functionality. (As the commentators indicated, you should only do this if these private methods' functionality is really a part in with the public interface. If they actually perform functions that are hidden from the user (i.e. the unit test), this is probably bad).

Is writing Unit Tests worth it for already exciting functionality?

It's absolutely worth it. If you write tests that cover the functionality you're adding or modifying, you'll get an immediate benefit. If you wait for a re-write, you may never have automated tests.

You shouldn't spend a lot of time writing tests for existing things that already work. Most of the time, you don't have a specification for the existing code, so the main thing you're testing is your reverse-engineering ability. On the other hand, if you're going to modify something, you need to cover that functionality with tests so you'll know you made the changes correctly. And of course, for new functionality, write tests that fail, then implement the missing functionality.

What is Code Coverage?

Code coverage is a metric that determines the number of lines of code validated successfully by a testing process, which helps to analyze how software is verified in depth. Of course, the ultimate aim of any software company is the development of software products for businesses. But, in order to achieve that goal, companies must ensure that all the essential quality features of the software they develop are accurate, maintainable, effective, trustworthy, and safe.

When and where should I use Mocking?

Rule of thumb:

If the function you are testing needs a complicated object as a parameter, and it would be a pain to simply instantiate this object (if, for example, it tries to establish a TCP connection), use a mock.

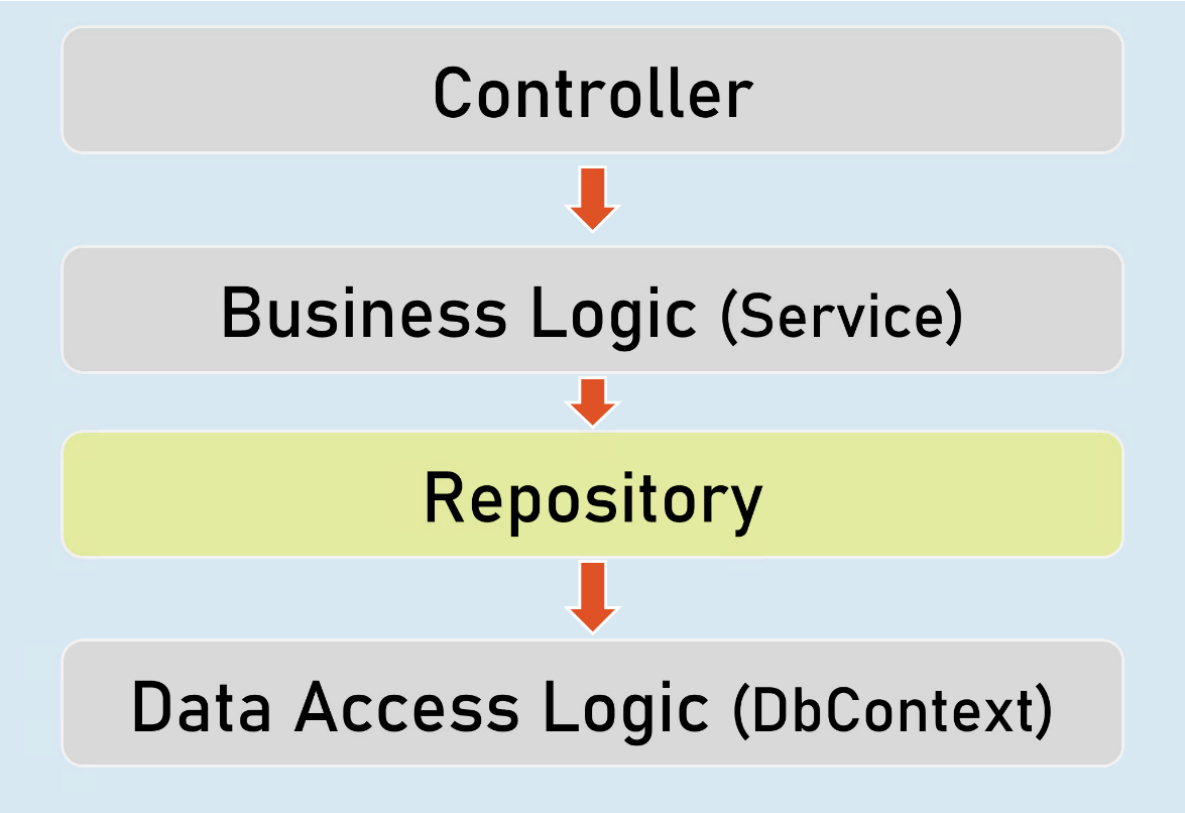
Typically you write a mock object if:

- The real object is too complex to incorporate it in a unit testing (For example a networking communication, you can have a mock object that simulate been the other peer)
- The result of your object is non-deterministic

Explain how and why to use repository pattern in Asp.Net Core?

Repository (or Repository Pattern) is an abstraction between Data Access Layer (EF DbContext) and business logic layer (Service) of the application.

Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer.



A Repository Pattern allows all of your code to use objects without having to know how the objects are persisted. All of the knowledge of persistence, including mapping from tables to objects, is safely contained in the repository.

Under the covers:

- For reading, it creates the query satisfying the supplied criteria and returns the result set.
- For writing, it issues the commands necessary to make the underlying persistence engine (e.g. an SQL database) save the data.

Very often, you will find SQL queries scattered in the codebase and when you come to add a column to a table you have to search code files to try and find usages of a table. The impact of the change is far-reaching.

How does EF Core support Transactions?

In EF Core, whenever you execute SaveChanges() to insert, update or delete data into the database, it wraps that operation in a transaction. So, you don't need to open a transaction scope explicitly.

If multiple CUD operations are done, and if one operation is failed due to exception, all the executed operations are rolled back automatically, by the transaction.

How do you execute plain SQL in Entity Framework Core?

For non-query operations such as INSERT, UPDATE, DELETE:

```
DbContext.Database.ExecuteSqlRaw(string sql, params object[ ] parameters)
```

For query operations such as SELECT:

```
DbSet.FromSqlRaw(string sql, params object[] parameters)
```