Governance on Unit Testing

Goal:

- **To be confident of quality of code in production**

Objectives:

- **raise the standard of confident coding by enforcing x% of code coverage on**
  - Pull Requests
  - Overall project
- discover and repackage tooling/toolchain for code quality
  - selected stages of testing
    - unit testing
    - integration testing
  - API
  - SDKs
  - Storage (Redis, ES, PG) and ORMs (EF, Dapper)
  - Kafka Consumers
  - Generic host services (console, background jobs)
  - Service classes
  - Actors
- Introduce a way to present wording as task on DevOps
  - e.g. Write unit test to cover "RegisterMomo" action method in WalletsController
- document governance around writing unit tests

Tools/Framework:

- Testing Framework:
  - XUnit (https://xunit.net/)
- Assertions:
  - FluentAssertions (https://fluentassertions.com/)
- Mocking:
  - **Moq** (https://github.com/moq/moq4)
- Data generators:
  - Bogus (https://github.com/bchavez/Bogus)

Common sense of knowing whether to write tests:

- **Any handwritten code should be testable**

Recommended Readings:

- https://docs.microsoft.com/en-us/dotnet/core/testing/
- https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Principles

Naming convention

- Test Project name
    - Must end with ".Tests" after the original project name
        - .e.g **Hubtel.ReceiveMoney.Mtn.Api.Tests**
- Test class names should end with "Tests" e.g. **WalletsControllerTests** or **CustomerServiceTests**
- Test methods should begin with method name being tested and meaningful test description in Pascal-SnakeCase e.g.
    - Format: [MethodName]_**Should**_[Action]_**When**_[Condition(s)]
        - e.g. VerifyAndConfirmRegistration_Should_Register_Account_When_GhanaCardId_Is_Valid
        - e.g. TopupDevice_Should_Respond_With_HttpStatusCode_NotFound_When_MeterId_Is_Not_Stored_In_Db
        -

Areas of Testing:

APIs:

1. What do I test?
    a. Controller actions
        i. One unit test class per controller (if actions methods are not too many)
        ii. Create #region for each action method and write the test methods/cases
        iii. Use InlineData for highly predictable, simple action methods
    b. Service classes
        i. One unit test class per service class (if public methods are not too many)
        ii. Create #region for each action method and write the test methods/cases
        iii. Use InlineData for highly predictable, simple action methods
2. How do I decide on which test to write?
    a. Proposed strategies:
        i. Write a single test to accommodate different input parameters with expectations OR
        ii. Break down target method's branching into individual test methods
3. How do I name my tests?
    a. Refer to naming convention

Writing the tests:

- Each test class/method should have the SUT (system under test) object
- Prepare "props" for the SUT using IClassFixture
    - All mocks, fake data, DI entities
- Use the AAA pattern for writing all test method(s)

Actors:

1. What do I test?
    a. Actors themselves
        i. For a better, wiser [testing] experience, abstract actor logic into a service class
    b. Service classes

Tips for writing effective tests:

- Always respect the business value first.
    - Should the code drive the test or the test drive the code?
- Stick to Repository Pattern as much as possible
    - Avoid direct calls to ORM methods
- Ensure service classes do not depend on Nuget package classes as return types (unless they can be easily mocked)
-