Notes sur le module : *Test Driven Developpement.*

Armel Pitelet

3 février 2022

Table des matières

Sur les types de test, voir https://blog.octo.com/la-pyramide-des-tests-par-la-pratique-1-5/ → la pyramide des test d'intégration. Faire un test est un compromis entre cout et précision /fiabilité

- Test unitaire : test d'un block de code le plus petit possible.
- Test de composant : on test une partie d'une application.
- Test d'intégration : on test l'ensemble de l'application après l'ajout d'une nouvelle fonctionnalité.
- IHM (de bout en bout) : On démarre l'application et on regarde directement si tt fonctionne correctement.
- Test de monté en charge (de performance) : pour ne pas dégrader les performances au cours du temps.
- Test utilisateur : analyse de donnée faite en conception (une fois que tt est développé).

Note :
Pour ce cours on repart de la V3 DemoBinding dans WPF.

Remarque : *Sur les template de test dans Visual Studio.*
On a NUnit, MSTest ou bien XUnit (Il faut privilégier les deux derniers).

Remarque : *Sur MSTest.*
Dans ce template on a une classe annoté `TestClass` et une méthode `TestMethod`.

Remarque : *Sur les packages interessant.*
Via NUget : FluentAssertions : Apporte plus de test par rapport à ce qui est disponible de base

Remarque : *Sur le test des méthodes Async.*
Il faut également que le test associés soit asynchrone.

1 Unitaire et intégration

Remarque : *Sur les conventions de nommage des tests.*
Il y a plusieurs méthodes : https://dzone.com/articles/7-popular-unit-test-naming.

Note :

Attention : parfois pour débusquer un bug il est nécessaire d'écrire des programme de test *manuelle* (dans le sens test logique et pas unitaire).

Remarque : *Sur la visibilité dans les tests unitaires.*

On peut passer tt les propr en private mais ce n'est pas forcément une nécissité.

Remarque : *Sur l'annotation ClassInitialize.*

Permet de lancer la méthode annotée une seule fois pour la classe de test en cours. Attention cette méthode doit être static. On pourrait également passer par le constructeur (sans lui donner l'annotation). La version avec annotation fait qq petites optimisations et est donc à préférer. Cependant si il faut initialiser des choses non static il faut passer par le constructeur.

Remarque : *Sur l'annotation TestInitialize.*

Permet de lancer la méthode annotée à chaque tests dans la classe de test.

Note :

Pour excécuter les test un par un Il faut passer par le gestionnaire de test de Visual Studio.

Remarque : *Sur la philosophie des test unitaires.*

Pour chaque classe, il faut tester chaque méthode, dans un cas nominal, et dans les cas un peu plus *borderline* qui sont sensé introduire des erreurs.

Remarque : *Sur l'annotation ExpectedException().*

Permet d'attendre une exception du type typeof(UneException, potentiellement un message).

Remarque : *Sur les log.*

En générale on log les erreurs, les paramètres passées en méthodes (avec `logger.LofInformation`). A utiliser un peu partout (en console+fichier en debug. Uniquement en ficheir en Release). Voir **Serilog** et/ou **Nlog**. Il vaut mieux passer par injection ded épendance pour utiliser les logs.

Remarque : *Sur la couverture de code.*

A partir du moiment ou l'on fait des tests unitaire, on peut regarder la couverture du code (par les tests).

Remarque : *Si l'on a du mal à tester.*

C'est qu'en générale les principes solides n'ont pas été respectés et qu'il faut refactorer le code.

Remarque : *Sur la répartition des test.*

Un projet de test par projet testé, une classe de test par class testée.

Remarque : *Sur les annotation allant avec la IActionResult.*

Elle peuvent servir pour la doc (cad swagger) mais ne change pas le fonctionnement du code.

Remarque : *Sur les ActionResult et IActionResult.*

IActionResult est l'interface implémenté par ActionResult et ActionResult<>. On a pas besoin de spécifier un type de retour autre. IActionResult est plus général et donc à favoriser.

Remarque : *Sur la notion de code Synchrone.*

Le code s'excécute de manière séquentielle.

Remarque : *Sur la notion de code Parallèle.*
Le code s'excécute de manière séquentielle sur plusieurs thread différents (on éxécute deux codes identique ou non en parallèle).

Remarque : *Sur la notion de code Asynchrone.*
Dans le code asynchrone les partie du code marquées `await` peuvent attendre (appel api, bdd) et dans ce cas le reste du code continue à s'excécuter. Async et await laisse le framework décider ou non de créer un nouveau thread pour attendre puis exécuter le code dès que possible.

Note :
Au niveau de l'API on voudra des controlleurs asynchrones pour augmenter les performances car on va pouvoir faire plus de requêtes si ces dernières peuvent être mise en attentes.

Remarque : *Pour les test avec bdd.*
Si l'on fait une connection direct il faut bien penser à nettoyer la bdd par la suite. Pour être plus propre on peut passer par le Driver InMemory ou bien SqlLite (en package nugget). Attention au InMemory qui peut avoir des comportement différents d'une base de donnée relationnelle. Voir plutot avec SqlLite (package de EFCore) : https://docs.microsoft.com/fr-fr/ef/core/testing/sqlite.

Note :
Attention à utiliser le moins possible les méthodes déjà existante pour initialiser les test unitaire (pour ne pas risquer d'ajouter un bug dès le début du test).

Remarque : *Sur les test bdd.*
Pour tester certain composant on peut se passer des drivers précédent et isoler les composant à tester via certain frameworks : comme **Moq** qui permet de créer des objets fictifs qui répondent à la même interface que les objets que l'on veut simuler. Voir DemoBindong dans TDD pour implémentation donnée par Nico. On peut donner aux objets Mocks un comportement à avoir. L'intéret est de pouvoir isoler un objet de toutes ses dépendances (ici on test un controller en l'isolant du repository qu'il utilise en créant un mock pour le repository et en lui donnant un comportement attendue pour les fonctions qu'il est sensé avoir via `Setup(repo => repo.func()).Return(result_attendue)`.).

Remarque : *Du détail sur les Mocks.*
On peut également vérifier que certaines méthodes on bien été appelée (une et une seule fois, plusieurs fois,... Pour passer des paramètres lors dun `Setup` on peut soit passer un véritable objet soit utiliser le mot clef `It` avec par exemple It.IsAny<Type que l'on veut>()) via `Mock.Verify`.

Remarque : *Sur la bonne utilisation des mocks.*
Ils sont trés utiles pour les test unitaire et de composant mais ne doivent pas être utilisé lors de test d'intégration.

Remarque : *Pour générer le code coverage.*
Il existe **OpenCover**, un outils github pour vérifier le coverage des tests.

2 Test de performance

Remarque : *Pour des tests de performances cross version.*
On peut utiliser la bibliothèque **benchmarkdotnet**. Voir https://benchmarkdotnet.org/articles/

guides/getting-started.html pour un guide de démarrage. Il semble qu'il faille utiliser une console App pour ce genre de test.

Note :
Il vaut mieux être dans les même conditions que dans la production pour faire des tests de performances. Par contre il ne faut pas être dans la prod en elle même mais sur un environnement iso-prod (en terme de code mais aussi en terme d'architecture logiciel et infrastructure).

Remarque : *Sur l'attribut* ***GlobalSetup***.
Permet de donner un environnement global aux tests de performance. La méthode marqué par cette attribut sera appelée une seule , avant l'excécution de tt les tests marqués **Benchmark** qui marque une méthode comme à 'benchmarker'.

Note :
Il faut penser à passer la génération en mode release pour les tests de performances.

Remarque : *Sur l'attribut* ***SimpleJob***.
Permet via RuntimeMonikor de tester différent type de runtime (comme .net 5 vs .net 6).

Note :
Après un test on peut exporter les résultats d'un test sous forme de graphique. Peut être utile pour garder une trace et comparer en fonction des versions des projets.

Remarque : *Pour faire du Parallel.*
On doit passer par la bibliothèque parallel.

Remarque : *Pour tester les performance de l'API.*
On peut passer par la bibliothèque **Gatling**. C'est plutôt un outil Java, même si on peut l'utiliser dans dotnet en passant par du Scala (encore un nouveau language). On passera plutôt par **NBomber**. NBomber fournit un .html à la fin des tests qui résume les tests.

Remarque : *Sur le ScenarioBuilder.*
Cela permet de configurer les conditions de tests.

Remarque : *Sur la notion de WarmUp.*
Le WarmUp est une étape de *chauffe* de l'API. Vu qu'en C# le compilateur fait du JIT (Just in Time), il y a qq optimisations qui ne sont appliqués qu'après qq utilisation du code. La période pendant laquelle ces optimisations ne sont pas encore effective est appelée le WarmUp.

Note :
NBomber permet également de monitoré l'utilisation matériel pendant les tests.

Note :
Ce genre de tests de performance sur l'API est qq chose de très fréquent. L'idée est de suivre dans le temps l'évolution des performance du programme. C'est aussi intéressant pou r*sizer* une application c'est à dire avoir une idée du matériel nécessaire pour utiliser une application (en fonction de la charge demandée).

Remarque : *Sur les outils natif pour des test de performance.*
voir https://docs.microsoft.com/fr-fr/dotnet/core/diagnostics/dotnet-trace.

Remarque : *Pour tracer les fuites mémoire.*
Il faut passer par les commandes **dotnet dump** qui permettent de récupérer ce qu'il y a dans la mémoire.

Remarque : *Sur une alternative au pattern factory.*
Voir le pattern **builder**

Remarque : *Sur le test d'un site internet (Notre site Angular).*
On peut utiliser **cypress**. Il est possible de le mettre dans la chaine d'intégration continue. Il peut prendre des screenshot des events qui ne se déroulent pas comme attendue. Pour un guide démarage voir : https://docs.cypress.io/guides/getting-started/writing-your-first-test#Add-a-test-file.

Remarque : *Liens utiles.*
awesome X dans une recherche permet d'avoir des pages qui regroupent l'ensemble des technos les plus appréciés et utilisés par la communauté. Voir par exemple https://github.com/quozd/awesome-dotnet.

Remarque : *Sur les tests en WPF.*
On passe par ce drivers : **WinAppDriver** (Exe à récupérer, à installer puis lancer au click ou en ligne de commande [penser à l'ajouter au path]). Une fois mis dans le path on peut le lancer en ligne de commande via WinAppDriver Nport ou Nport est le port à donner. Il faut ensuite dans les package nugget passer par Appium.Webdriver (dans visual studio) et utiliser une classe de configuration (voir exemple de nico). A partir de la on crée des classes de test.