R4.01 / R4.02
TP4 partie 3 – Amélioration des tests de l'API REST : Tests de substitution

Remarque : Si vous devez ajouter une méthode `Equal` ou des constructeurs, pensez à les créer dans une autre classe partielle (ne pas modifier les classes du Code First).

Méthode POST

Code du test unitaire de la méthode POST :

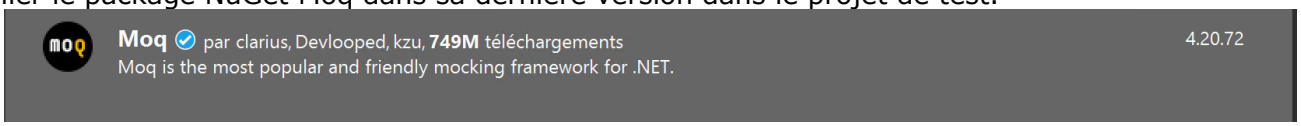
```
[TestMethod]
public void Postutilisateur_ModelValidated_CreationOK()
{
    // Arrange
    Random rnd = new Random();
    int chiffre = rnd.Next(1, 1000000000);
    // Le mail doit être unique donc 2 possibilités :
    // 1. on s'arrange pour que le mail soit unique en concaténant un random ou un timestamp
    // 2. On supprime le user après l'avoir créé. Dans ce cas, nous avons besoin d'appeler la méthode DELETE de l'API ou remove du DbSet.
    Utilisateur userAtester = new Utilisateur()
    {
        Nom = "MACHIN",
        Prenom = "Luc",
        Mobile = "0606070809",
        Mail = "machin" + chiffre + "@gmail.com",
        Pwd = "Toto1234!",
        Rue = "Chemin de Bellevue",
        CodePostal = "74940",
        Ville = "Annecy-le-Vieux",
        Pays = "France",
        Latitude = null,
        Longitude = null
    };

    // Act
    var result = controller.PostUtilisateur(userAtester).Result; // .Result pour appeler la méthode async de manière synchrone, afin d'attendre l'ajout

    // Assert
    Utilisateur? userRecupere = context.Utilisateurs.Where(u => u.Mail.ToUpper() == userAtester.Mail.ToUpper()).FirstOrDefault(); // On récupère l'utilisateur créé directement dans la BD grace à son mail unique
    // On ne connait pas l'ID de l'utilisateur envoyé car numéro automatique.
    // Du coup, on récupère l'ID de celui récupéré et on compare ensuite les 2 users
    userAtester.UtilisateurId = userRecupere.UtilisateurId;
    Assert.AreEqual(userAtester, userRecupere, "Utilisateurs pas identiques");
}
```

L'inconvénient de ce test unitaire est que l'on pollue la base avec des données de test : à chaque exécution de ce test, un nouvel utilisateur « MACHIN Luc » est ajouté.

Pour éviter cela, nous allons créer des tests de substitution (Moq) (= tests d'intégration à étroite couverture d'unité - *Narrow Integration Test*) : leur scope s'arrête à quelques composants tout en substituant d'autres dépendances (notion de substitution ou mock). Ces tests ressemblent à des tests unitaires au niveau de l'utilisation des frameworks et la façon de les écrire.
Ils permettent, par exemple d'émuler une API (non encore développée) qui serait invoquée par une seconde API en cours de développement : par exemple une API Utilisateurs, comme nous l'avons réalisé, qui appellerait une API d'authentification (OAuth API). Ou, autre exemple, simuler un « repository » d'objets habituellement liés à la base de données avec un « repository » factice ne retournant qu'un objet bien défini. C'est ce 2nd exemple que nous allons mettre en pratique.

Il est important d'utiliser des objets factices pour conserver le caractère de répétabilité d'un test (une propriété de date ou un capteur de température devant rester figés pour obtenir les mêmes résultats en sortie et donc la validation du test).
Plus d'explication ici : https://www.e-naxos.com/Blog/post/Xamarinforms-Unit-Testing-avec-MOQ.aspx

Installer le package NuGet Moq dans sa dernière version dans le projet de test.



Le package Moq fournit un framework de simulacre qui va nous permettre de créer de faux objets facilement.

Code du test de substitution de la méthode POST :

```
[TestMethod]
public void Postutilisateur_ModelValidated_CreationOK_AvecMoq()
{
    // Arrange
    var mockRepository = new Mock<IDataRepository<Utilisateur>>();
    var userController = new UtilisateursController(mockRepository.Object);

    Utilisateur user = new Utilisateur
    {
        Nom = "POISSON",
        Prenom = "Pascal",
        Mobile = "1",
        Mail = "poisson@gmail.com",
        Pwd = "Toto12345678!",
        Rue = "Chemin de Bellevue",
        CodePostal = "74940",
        Ville = "Annecy-le-Vieux",
        Pays = "France",
        Latitude = null,
        Longitude = null
    };

    // Act
    var actionResult = userController.PostUtilisateur(user).Result;

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(ActionResult<Utilisateur>), "Pas un ActionResult<Utilisateur>");
    Assert.IsInstanceOfType(actionResult.Result, typeof(CreatedAtActionResult), "Pas un CreatedAtActionResult");
    var result = actionResult.Result as CreatedAtActionResult;
    Assert.IsInstanceOfType(result.Value, typeof(Utilisateur), "Pas un Utilisateur");
    user.UtilisateurId = ((Utilisateur)result.Value).UtilisateurId;
    Assert.AreEqual(user, (Utilisateur)result.Value, "Utilisateurs pas identiques");
}
```

`var mockRepository = new Mock<IDataRepository<Utilisateur>>();` : permet d'instancier un simulacre sur le repository IDataRepository<Utilisateur>>
Ensuite, au lieu de passer le repository au contrôleur, on lui passe ce simulacre : `controller = new UtilisateursController(new IDataRepository<Utilisateur>)` devient `controller = new UtilisateursController(mockRepository.Object)`.
Enfin, les assertions sont « classiques » (pas de changement par rapport au TP2).
Bien que la méthode `PostUtilisateur` soit appelée, vous verrez qu'aucun enregistrement n'est créé dans la base.
Si la couche BD n'avait pas été développée, ou si la base ne contenait pas les données espérées, ce test de substitution fonctionnerait toujours, ce qui ne serait pas forcément le cas du TU de la 1ère page.
Sur les méthodes de tests des créations qui échouent, comme il n'y a pas de modification de la base, il n'est pas primordial de réaliser des tests de substitutions (même si on peut le faire !).

Méthode GetById

Test de substitution de la méthode `GetUtilisateurById` avec un ID qui existe :

```
[TestMethod]
```

```
public void GetUtilisateurById_ExistingIdPassed_ReturnsRightItem_AvecMoq()
{
    // Arrange
    Utilisateur user = new Utilisateur
    {
        UtilisateurId = 1,
        Nom = "Calida",
        Prenom = "Lilley",
        Mobile = "0653930778",
        Mail = "clilleymd@last.fm",
        Pwd = "Toto12345678!",
        Rue = "Impasse des bergeronnettes",
        CodePostal = "74200",
        Ville = "Allinges",
        Pays = "France",
        Latitude = 46.344795F,
        Longitude = 6.4885845F
    };
    var mockRepository = new Mock<IDataRepository<Utilisateur>>();
    mockRepository.Setup(x => x.GetByIdAsync(1).Result).Returns(user) ;

    var userController = new UtilisateursController(mockRepository.Object);

    // Act
    var actionResult = userController.GetUtilisateurById(1).Result;

    // Assert
    Assert.IsNotNull(actionResult);
    Assert.IsNotNull(actionResult.Value);
    Assert.AreEqual(user, actionResult.Value as Utilisateur);
}
```

Quand on appelle la méthode `GetUtilisateurById(1)`, la méthode `GetByIdAsync(1)` de la classe `UtilisateurManager` va ensuite être appelée.
`mockRepository.Setup(x => x.GetByIdAsync(1).Result).Returns(user)` : signifie « dès que la méthode `GetByIdAsync` est appelée, pour la valeur 1, retourne l'objet `user` ».

Test du `GetUtilisateurById` avec ID inconnu :

```
[TestMethod]
public void GetUtilisateurById_UnknownIdPassed_ReturnsNotFoundResult_AvecMoq()
{
    var mockRepository = new Mock<IDataRepository<Utilisateur>>();
    var userController = new UtilisateursController(mockRepository.Object);

    // Act
    var actionResult = userController.GetUtilisateurById(0).Result;

    // Assert
    Assert.IsInstanceOfType(actionResult.Result, typeof(NotFoundResult));
}
```

Méthode GetByEmail

Faire de même : 1 test Moq qui réussit et 1 qui échoue.

Méthode DELETE

Normalement le code du test de substitution devrait être :

```
[TestMethod]
public void DeleteUtilisateurTest_AvecMoq()
{
```

```
        // Arrange
        var mockRepository = new Mock<IDataRepository<Utilisateur>>();
        var userController = new UtilisateursController(mockRepository.Object);

        // Act
        var actionResult = userController.DeleteUtilisateur(1).Result;

        // Assert
        Assert.IsInstanceOfType(actionResult, typeof(NoContentResult), "Pas un NoContentResult"); // Test du type de retour
    }
```

Exemple ici :
https://learn.microsoft.com/fr-fr/aspnet/web-api/overview/testing-and-debugging/unit-testing-controllers-in-web-api

L'action retourne 200 (OK) sans corps de réponse

La `Delete` méthode appelle `Ok()` pour renvoyer une réponse HTTP 200 vide. Comme l'exemple précédent, le test unitaire vérifie le type de retour, dans ce cas **OkResult**.

C# Copier

```
[TestMethod]
public void DeleteReturnsOk()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Delete(10);

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(OkResult));
}
```

Sauf que ce test n'est pas fonctionnel. Mettre un point d'arrêt et le déboguer. Vous verrez alors que la méthode `GetByIdAsync` retourne *null*.

```
115         // DELETE: api/Utilisateurs/5
116         [HttpDelete("{id}")]
117         [ProducesResponseType(StatusCodes.Status204NoContent)]
118         [ProducesResponseType(StatusCodes.Status404NotFound)]
119         public async Task<IActionResult> DeleteUtilisateur(int id)
120         {
121             var utilisateur = await dataRepository.GetByIdAsync(id);
122             if (utilisateur == null)                                  id  1
123             {
124                 return NotFound();  ≤ 1 ms écoulées
125             }
126
127             await dataRepository.DeleteAsync(utilisateur.Value);
128             return NoContent();
129         }
```

Il faut à nouveau indiquer qu'elle doit retourner un user particulier. Le code devient alors :

```
    [TestMethod]
    public void DeleteUtilisateurTest_AvecMoq()
    {
        // Arrange
        Utilisateur user = new Utilisateur
        {
            UtilisateurId = 1,
            Nom = "Calida",
            Prenom = "Lilley",
            Mobile = "0653930778",
            Mail = "clilleymd@last.fm",
            Pwd = "Toto12345678!",
            Rue = "Impasse des bergeronnettes",
            CodePostal = "74200",
            Ville = "Allinges",
            Pays = "France",
            Latitude = 46.344795F,
            Longitude = 6.4885845F
        };

        var mockRepository = new Mock<IDataRepository<Utilisateur>>();
```

```
    mockRepository.Setup(x => x.GetByIdAsync(1).Result).Returns(user);
    var userController = new UtilisateursController(mockRepository.Object);

    // Act
    var actionResult = userController.DeleteUtilisateur(1).Result;

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(NoContentResult), "Pas un NoContentResult"); // Test du type de retour
}
```

Méthode PUT

Coder le test Moq d'un PUT qui réussit.
Indications : mélange code de test de POST et de DELETE. Il faudra 2 objets Utilisateur : celui recherché et le même mais avec des données modifiées. Bien mettre un point d'arrêt pour vérifier l'exécution.