**Développement d’une application de gestion de contacts avec ASP.NET MVC (C#)**

**Etape #5 – Créer des tests unitaires**

Dans cette série de tutoriaux, nous allons construire de bout en bout une application complète de gestion de contacts. Cette application vous permettra de stocker les informations – noms, numéro de téléphone et adresses emails – pour une liste de personnes.

Nous allons construire cette application à travers plusieurs étapes. Chacune d’entres elles allant enrichir notre application MVC. Le but de cette construction par étape est de vous permettre de comprendre la raison derrière chaque changement.

* **Etape #1 – Création de l’application**. Dans cette 1ère étape, nous allons créer l’application de gestion de contacts de la manière la plus simple qui soit. Nous allons mettre en place le support d’opérations classiques vers la base de données : création, lecture, mise à jour et suppression d’enregistrements.
* **Etape #2 – Rendre l’application plus attrayante**. Ici, nous allons améliorer l’apparence de l’application en modifiant la page maitre par défaut d’ASP.NET MVC ainsi que le CSS l’accompagnant.
* **Etape #3 – Ajout de la validation de formulaires**. Dans cette 3ème étape, nous allons ajouter une logique simple de validation. Nous allons empêcher les utilisateurs de soumettre un formulaire sans remplir certains champs obligatoires. Nous allons également valider les adresses email et numéros de téléphone.
* **Etape #4 – Rendre l’application faiblement couplée**. Ici, nous allons profiter de plusieurs modèles de développement logiciel (Design Patterns) pour maintenir et modifier plus facilement notre application. Par exemple, nous allons revoir l’application pour utiliser 2 patterns connus sous le nom de « Repository pattern » et « Dependency Injection pattern »
* **Etape #5 – Créer des tests unitaires**. Dans cette 5ème étape, afin de rendre encore plus simple la gestion du code source, nous allons ajouter des tests unitaires. Nous allons utiliser pour cela un « Mock Object Framework » puis fabriquer des tests unitaires pour nos contrôleurs et logiques de validation.
* **Etape #6 – Utiliser un développement guidé par les tests**. Ici, nous allons ajouter de nouvelles fonctionnalités d’abord en écrivant des tests unitaires puis ensuite en écrivant du code pour ces mêmes tests unitaires. Dans cette étape, nous allons ajouter la notion de groupes de contacts.
* **Etape #7 – Ajouter le support d’Ajax**. Dans cette dernière phase, nous allons améliorer la réponse et la performance de notre application en y ajoutant le support d’Ajax.

### Développer une application de gestion de contacts avec ASP.NET MVC (C#)

Lire et télécharger les tutoriels : <http://msdn.microsoft.com/fr-fr/asp.net/dd627541.aspx>

**Sommaire**

1. **Dans cette étape**
2. **Qu’apportent les tests unitaires**
3. **Utilisation du Mock Object Framework**
4. **Création des tests unitaires pour la couche de service**
5. **Création des tests unitaires pour les contrôleurs**
6. **Conclusion**

## Dans cette étape

Dans l’étape précédente de l’application de gestion de contacts, nous avons repris le code pour le rendre faiblement couplé. Nous avons séparé l’application en plusieurs morceaux : le contrôleur, le service et la couche d’accès aux données. Chaque couche interagie avec les autres à travers des interfaces.

Ces modifications ont permis de rendre l’application plus maintenable et mieux modifiable. Par exemple, si l’on a besoin de changer la façon d’accéder aux données, il suffit de changer la couche d’accès associée (Repository Layer) et ce, sans toucher ni au contrôleur ni au service. En rendant l’application de gestion de contacts faiblement couplée, nous rendons l’application plus agile au changement.

Mais, que se passe-t-il lorsque l’on voudra ajouter une fonctionnalité à l’application de gestion de contacts ? Ou lorsque l’on souhaitera corriger un bug ? A chaque modification, on risque de générer un nouveau bug.

Par exemple si un jour le chef de projet demande de rajouter une fonctionnalité comme la notion de groupes de contacts pour que les utilisateurs puissent organiser leurs contacts et les associer à des groupes tels que « Amis », « Travail », ou autre.

Afin d’implémenter cette nouvelle fonctionnalité, il faudra modifier les trois couches de l’application de gestion de contacts. Ajouter une nouvelle fonctionnalité dans le contrôleur, à la couche de service et bien sûr à la couche d’accès aux données. Dès que le code commencera à être modifié, les fonctionnalités qui fonctionnaient avant risqueront d’être touchées et cassées.

Le fait d’avoir repris le code pour lui donner une architecture en couche est une bonne chose car cela permet de changer une couche sans toucher au reste de l’application. Cependant, pour rendre le code plus facile à maintenir et à modifier dans une couche donnée, il va falloir créer des tests unitaires.

Il faut donc utiliser les tests unitaires pour tester une petite partie de code. Les tests unitaires doivent être plus petits qu’une couche complète d’une application. Par exemple, on peut utiliser les tests unitaires pour tester une méthode. On pourrait prendre de manière unitaire CreateContact() de la classe ContactManagerService.

Les tests unitaires pour une application doivent être autonomes. Quelques soient les modifications apportées dans l’application, vous devez pouvoir lancer les tests et vérifier si les modifications n’entrainent pas de problèmes dans les fonctionnalités existantes. Les tests rendent votre code plus robuste aux modifications.

Dans cette étape, nous allons ajouter des tests à l’application de gestion de contacts. C’est grâce à ces tests que, dans la prochaine étape, nous allons pouvoir ajouter la fonctionnalité de groupes de contacts sans crainte de casser des fonctionnalités existantes.

Il existe de nombreux frameworks de tests tel que NUnit, xUnit.net and MbUnit. Dans ce tutorial, nous allons utiliser le frameworks de tests inclus dans Visual Studio. Cependant, vous pouvez facilement utiliser d’autre frameworks de tests.

## Qu’apportent les tests unitaires ?

Dans un monde parfait, tout notre code devrait être couvert par les tests. Chaque modification d’une ligne de code dans votre application devrait pouvoir être testée instantanément par l’exécution du test unitaire.

Cependant, nous ne vivons pas dans un monde parfait. En pratique, quand on écrit des tests, on se focalise à tester la couche métier. En général, on n’écrit pas de tests pour la couche d’accès aux données, ni pour la couche de présentation.

Pour être utiles, les tests doivent pouvoir être exécutés rapidement. Vous pouvez sans problème créer des centaines (voir des milliers) de tests pour une application. En effet, si l’exécution d’un test unitaire est trop longue, il ne sera pas exécuté. En d’autres termes, les tests unitaires trop longs sont inutiles au développement de tous les jours.

C’est pour cette raison que les tests unitaires ne doivent pas interagir avec la base de données. Faire tourner des centaines de tests unitaires sur une base de données peut être très long. Par contre, abstraire la base et écrire du code qui interagie avec cette abstraction est beaucoup plus intéressant.

C’est exactement pour cette même raison qu’on n’écrira pas de tests unitaires sur les vues. En effet, pour tester une vue, il est nécessaire d’avoir un serveur Web. Cela peut être aussi très lent à mettre en place et n’est donc pas recommandé pour les tests unitaires.

Si votre vue contient de la logique complexe, vous devez déplacer cette logique dans des méthodes de type Helper. Vous pourrez alors écrire des tests unitaires sur ces méthodes Helper sans avoir à monter un serveur Web.

Alors que l’écriture des tests unitaires sur la couche d’accès aux données ou sur les vues n’est pas une bonne idée, ces tests peuvent être d’une grande importance pour évaluer votre application à des fins fonctionnelles ou d’intégration.

Note : ASP.NET MVC repose sur le moteur ASP.NET WebForms. Alors que le moteur des WebForms est dépendant du serveur Web, d’autres moteurs de vues MVC pourrait ne pas l’être.

## Utilisation de Mock Object Framework

Quand vous écrivez des tests unitaires, vous aurez presque toujours besoin de tirer avantage de Mock Object Framework. Un Mock Object Framework permet la création de méthodes et de « fausses » classes dans votre application.

Par exemple, vous pouvez utiliser un Mock Object Framework pour générer une version fictive de la couche d’accès aux données. Vous pouvez ensuite utiliser ces classes fictives à la place des vraies classes dans vos tests unitaires. Utiliser la couche d’accès aux données fictives vous permettra d’éviter d’exécuter la vraie couche d’accès pour l’exécution des tests.

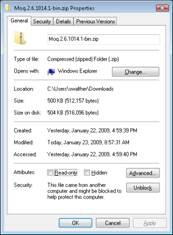
Visual Studio n’intègre pas de Mock Object Framework. Cependant, il existe de nombreux frameworks payants ou open source pour le framework .NET :

1. Moq – Ce framework est sous licence BSD. Vous pouvez le télécharger ici : <http://code.google.com/p/moq/>.
2. Rhino Mocks – Ce framework est sous licence BSD. Vous pouvez le télécharger ici : <http://ayende.com/projects/rhino-mocks.aspx>.
3. Typemock Isolator – Ce framework est payant. Vous pouvez télécharger une version d’évaluation ici : <http://www.typemock.com/>.

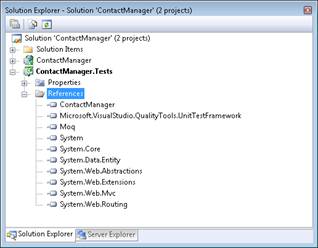
Dans ce tutoriel, nous utiliserons Moq. Cependant, vous pouvez facilement utiliser Rhino Mocks ou Typemock Isolator sur le projet de gestion de contacts.

Avant d’utiliser Moq, vous devez suivre ces étapes :

1. Débloquez le fichier Zip

[](http://static.asp.net/asp.net/images/mvc/30/CS/image001.png)

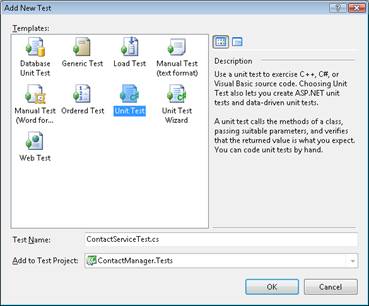
1. Décompressez le fichier téléchargé.
2. Ajoutez une référence à l’assembly Moq en faisant un bouton droit sur « References » sur le projet « ContactManager.Tests ». Sélectionnez « Add Reference », parcourrez le répertoire qui vient d’être décompressé et sélectionnez l’assembly Moq.dll.
3. Après cette étape vous devriez obtenir ceci

[](http://static.asp.net/asp.net/images/mvc/30/CS/image003.png)

## Création des tests unitaires pour la couche de service

Nous allons maintenant créer un jeu de tests unitaires pour la couche de service de l’application de gestion de contacts. Nous utiliserons ces tests afin de vérifier notre logique de validation.

1. Créez un nouveau répertoire « Models » dans le projet « ContactManager.Tests ». Puis cliquez droit sur ce répertoire et sélectionnez « Add, New Test ».

[](http://static.asp.net/asp.net/images/mvc/30/CS/image005.png)

1. Sélectionnez le template « Unit Test » et donnez lui comme nom : « ContactManagerServiceTest.cs » puis cliquez « Ok »

En général, on fait correspondre le projet de tests avec la structure du projet ASP.NET MVC. Par exemple, on mettra le test du contrôleur dans un répertoire « Controller », le test du modèle dans un répertoire « Model » etc…

Pour commencer, on va tester la méthode CreateContact() de la classe ContactManagerService. Nous allons donc créer ces 5 tests :

1. CreateContact()  - Test si CreateContact() retourne la valeur vrai quand un contact « valide » est passé à la méthode.
2. CreateContactRequiredFirstName() – Test si un message d’erreur est généré quand on envoi un Contact sans la propriété « FirstName » remplie (prénom) à la méthode CreateContact()
3. CreateContactRequredLastName() – Test si un message d’erreur est généré quand on envoi un Contact sans la propriété « LastName » remplie (nom de famille) à la méthode CreateContact()
4. CreateContactInvalidPhone() – Test si un message d’erreur est généré quand on envoi un Contact avec un numéro de téléphone non valide à la méthode CreateContact()
5. CreateContactInvalidEmail() – Test si un message d’erreur est généré quand on envoi un Contact avec une adresse email non valide à la méthode CreateContact()

Le premier test vérifie qu’un contact valide ne génère pas de messages d’erreur. Les tests suivant vérifient chaque règle de saisie.

Code des tests dans le Source 1.

**Listing 1 – Models\ContactManagerServiceTest.cs**

using System.Web.Mvc;

using ContactManager.Models;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

namespace ContactManager.Tests.Models

{

[TestClass]

public class ContactManagerServiceTest

{

private Mock<IContactManagerRepository> \_mockRepository;

private ModelStateDictionary \_modelState;

private IContactManagerService \_service;

[TestInitialize]

public void Initialize()

{

\_mockRepository = new Mock<IContactManagerRepository>();

\_modelState = new ModelStateDictionary();

\_service = new ContactManagerService(new ModelStateWrapper(\_modelState), \_mockRepository.Object);

}

[TestMethod]

public void CreateContact()

{

// Arrange

var contact = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555", "steve@somewhere.com");

// Act

var result = \_service.CreateContact(contact);

// Assert

Assert.IsTrue(result);

}

[TestMethod]

public void CreateContactRequiredFirstName()

{

// Arrange

var contact = Contact.CreateContact(-1, string.Empty, "Walther", "555-5555", "steve@somewhere.com");

// Act

var result = \_service.CreateContact(contact);

// Assert

Assert.IsFalse(result);

var error = \_modelState["FirstName"].Errors[0];

Assert.AreEqual("First name is required.", error.ErrorMessage);

}

[TestMethod]

public void CreateContactRequiredLastName()

{

// Arrange

var contact = Contact.CreateContact(-1, "Stephen", string.Empty, "555-5555", "steve@somewhere.com");

// Act

var result = \_service.CreateContact(contact);

// Assert

Assert.IsFalse(result);

var error = \_modelState["LastName"].Errors[0];

Assert.AreEqual("Last name is required.", error.ErrorMessage);

}

[TestMethod]

public void CreateContactInvalidPhone()

{

// Arrange

var contact = Contact.CreateContact(-1, "Stephen", "Walther", "apple", "steve@somewhere.com");

// Act

var result = \_service.CreateContact(contact);

// Assert

Assert.IsFalse(result);

var error = \_modelState["Phone"].Errors[0];

Assert.AreEqual("Invalid phone number.", error.ErrorMessage);

}

[TestMethod]

public void CreateContactInvalidEmail()

{

// Arrange

var contact = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555", "apple");

// Act

var result = \_service.CreateContact(contact);

// Assert

Assert.IsFalse(result);

var error = \_modelState["Email"].Errors[0];

Assert.AreEqual("Invalid email address.", error.ErrorMessage);

}

}

}

Comme on utilise la classe Contact dans ce code source, on doit ajouter une référence au Framework « Microsoft Entity ». Ajoutez donc une référence à l’assembly System.Data.Entity.

Le code du Listing 1 contient une méthode « Initialize() » auquel on a ajouté un attribut [TestInitialize]. Cette méthode sera donc appelée automatiquement avant chaque test unitaire (ici elle sera appelée 5 fois, 1 fois par test unitaire). La méthode Initialize() créé une couche de simulation de l’accès aux données grâce à cette ligne de code :

\_mockRepository = new Mock<IContactManagerRepository>();

Cette ligne de code utilise le Framework « Moq » pour générer un accès aux données virtuel et ceci depuis l’interface IContactManagerRepository. La « fausse » couche d’accès aux données sera utilisée plutôt que celle de type « EntityContactManagerRepository » qui aurait besoin de faire un accès physique à la base chaque fois. La « fausse » couche d’accès aux données implémente l’interface IContactManagerRepository, mais les méthodes ne font rien.

Quand on utilise le framework « Moq », il y a une distinction à faire entre \_mockRepository et \_mockRepository.Object. Le premier fait référence à la classe Mock<IContactManagerRepository> qui contient les méthodes pour spécifier le comportement de la « fausse » classe. Tant dis que l’autre implémente simplement l’interface IContactManagerRepository.

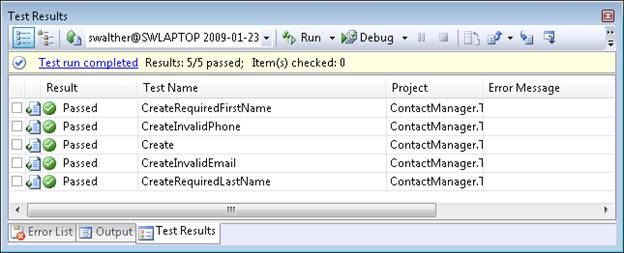
La « fausse » couche d’accès aux données utilise la méthode Initialize() quand une instance de la classe ContactManagerService est créé. Tous les tests unitaires utilisent cette instance de la classe ContactManagerService.

Le code du Listing 1 contient 5 méthodes. Chacune correspond à un test unitaire. Chaque méthode est décorée de l’attribut [TestMethod]. Quand les tests unitaires sont lancés, chaque méthode décorée de cet attribut est appelée. En d’autres termes, chaque méthode qui est décorée de cet attribut [TestMethod] devient un test unitaire.

Le premier test unitaire, nommé CreateContact(), vérifie que l’appel à CreateContact() retourne la valeur « vrai » dans le cas où on lui passe un contact « valide ».

Les autres tests vérifient que la méthode CreateContact() renvoie faux quand on lui passe un Contact qui n’est pas valide et qu’une exception est bien est soulevée. Par exemple, le test CreateContactRequiredFirstName() s’occupe de tester la création de la classe Contact avec une chaine vide dans la propriété FirstName. Ensuite, il appelle la méthode CreateContact() avec ce Contact non valide. Finalement, le test vérifie bien que la méthode CreateContact() renvoie « faux » et que le modèle lève une exception en conséquence avec comme message « First Name is required »

Vous pouvez lancer le test du code Listing 1 en allant dans le menu « Test / Run / All Tests in Solution » ou (CTRL+R, A). Le résultat des tests doit s’afficher comme ceci (Figure 4):

[](http://static.asp.net/asp.net/images/mvc/30/CS/image007.png)

**Figure 04**: Résultat des tests.

## Création des tests unitaires pour les contrôleurs.

Une application ASP.NET MVC contrôle les enchainements des interactions utilisateur. En testant un contrôleur, vous voulez savoir si ce dernier retourne la bonne action ou les bonnes données vis-à-vis du comportement attendu. Vous voulez aussi peut-être vérifier si le contrôleur interagit bien avec les classes du modèle de la manière attendue.

Par exemple, le Listing 2 contient deux tests unitaires sur le contrôleur Contact et sur la méthode Create(). Le premier test vérifie que lorsque l’on passe un Contact « valide » vers la méthode Create(), cette méthode redirige bien sur l’action « Index ». En d’autre termes, quand on lui passe un Contact « valide », la méthode Create() doit retourner une action « Index » de type RedirectToRouteResult.

Il ne faut pas tester la couche de service de l’application de gestion de contacts quand on test la couche contrôleur. Par conséquent, on va utiliser une « fausse » couche de service. On mettra ceci dans la méthode Initialize().

\_service = new Mock();

Dans le test unitaire CreateValidContact(), on appellera la « fausse » couche de service CreateContact() de cette manière :

\_service.Expect(s => s.CreateContact(contact)).Returns(true);

Cette ligne de code permet de toujours renvoyer « vrai » quand la méthode CreateContact() est appelée. En créant cette fausse couche de service, on va pouvoir tester le contrôleur dans avoir besoin d’exécuter la couche de service.

Le second test unitaire vérifie que l’action Create() retourne à nouveau la vue de création de contact quand on lui passe un contact « non valide ». La couche de service doit retourner « faux » à l’appel de la méthode CreateContact().

\_service.Expect(s => s.CreateContact(contact)).Returns(false);

Si la méthode s’exécute comme prévue, elle doit retourner une vue de type « Create » même quand la couche de service revoie faux. Dans ce cas, le contrôleur doit afficher les messages d’erreur de saisie dans la vue « Create » et l’utilisateur pourra donc de corriger les erreurs qu’il a commises.

Si vous avez pour but de créer des tests unitaires sur les contrôleurs, vous devez donc renvoyer explicitement le nom de la vue. Par exemple, vous ne devez par retourner ceci :

return View();

Mais plutôt ceci :

return View(“Create”);

En renvoyant le nom de la vue de façon explicite, la propriété ViewResult.ViewName sera renseignée.

**Code Listing 2 – Controllers\ContactControllerTest.cs**

using System.Web.Mvc;

using ContactManager.Controllers;

using ContactManager.Models;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

namespace ContactManager.Tests.Controllers

{

[TestClass]

public class ContactControllerTest

{

private Mock<IContactManagerService> \_service;

[TestInitialize]

public void Initialize()

{

\_service = new Mock<IContactManagerService>();

}

[TestMethod]

public void CreateValidContact()

{

// Arrange

var contact = new Contact();

\_service.Expect(s => s.CreateContact(contact)).Returns(true);

var controller = new ContactController(\_service.Object);

// Act

var result = (RedirectToRouteResult)controller.Create(contact);

// Assert

Assert.AreEqual("Index", result.RouteValues["action"]);

}

[TestMethod]

public void CreateInvalidContact()

{

// Arrange

var contact = new Contact();

\_service.Expect(s => s.CreateContact(contact)).Returns(false);

var controller = new ContactController(\_service.Object);

// Act

var result = (ViewResult)controller.Create(contact);

// Assert

Assert.AreEqual("Create", result.ViewName);

}

}

}

## Conclusion

Dans cette étape, nous avons créé des tests unitaires sur l’application de gestion de contacts. Les tests unitaires peuvent être lancés de façon autonome à tout moment pour vérifier le comportement de l’application. Les tests unitaires permettent de couvrir les scénarios d’évolution ou de modification sans risque.

Nous avons créé deux types de tests. Le premier a testé la logique de validation de l’application en créant des tests unitaires sur la couche de service. Ensuite, nous avons testé la logique de navigation en créant des tests sur la couche contrôleur. Les tests sur les couches de service et contrôleur sont autonomes grâce à l’utilisation d’un Framework de « Moq  permettant de simuler les couches de dépendance.

Dans la prochaine étape, nous allons ajouter de nouvelles fonctionnalités à l’application de gestion de contacts afin d’introduire la notion de groupes de contacts. Nous ajouterons cette fonctionnalité en utilisant le pattern « Test-Driven developpment» ou programmation pilotée par les tests. Le développement sera donc en permanence contrôlé par les tests unitaires.