Haut du formulaire

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

## Etape #4 – Rendre l’application faiblement couplée

## Dans cette étape

Dans cette 4ème étape, 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 »

Pour le moment, toute la partie d’accès aux données et de validation pour l’application de gestion de contacts est contenu dans la classe contrôleur. Ce n’est pas une très bonne idée car si vous décidez de modifier une partie de l’application, vous risquez d’introduire des bugs dans une autre partie de l’application. Par exemple, si vous modifiez une validation de données, vous risquez d’introduire un nouveau bug dans la partie d’accès aux données ou dans le contrôleur.

(SRP), une classe ne doit changer que pour une seule raison. Mixer contrôleur, validation et couche d’accès aux données n’est pas dans la philosophie du pattern « Single Responsibility Principle ». (Une classe par responsabilité)

Il a plusieurs raisons qui amènent à modifier une application. Vous voulez ajouter une nouvelle fonctionnalité, vous voulez résoudre un bug dans votre application ou modifier l’implémentation d’une fonctionnalité de l’application. Les applications sont rarement statiques. Elles ont tendance à grossir et évoluer dans le temps.

Imaginons, par exemple, qu’on veuille changer notre implémentation de notre accès aux données. Pour le moment l’application de Gestion de Contacts utilise Entity Framework pour accéder à la base. Vous pourriez très bien décider de migrer vers une technologie d’accès aux données différente telle que ADO.NET Data Service ou NHibernate. Cependant, parce que le code d’accès aux données entre la validation et le contrôleur n’est pas isolé, il est impossible d’en modifier le code sans modifier les autres parties de l’application qui ne sont pas directement liée à la couche d’accès aux données.

Quand une application est faiblement couplée, vous pouvez faire des modifications sur une partie de l’application sans modifier le reste. Par exemple, vous pouvez changer de couche d’accès aux données sans modifier la partie validation de données ou du contrôleur.

Dans cette étape, nous allons tirer partie des avantages de modèles de développement logiciel (Design Patterns) en faisant du refactoring sur l’application de Gestion de Contact pour rendre cette application faiblement couplée. Quand cela sera fait, l’application de Gestion de Contacts pourra être modifiée plus facilement.

Le “Refactoring” est un processus de réécriture de l’application mais sans perte de fonctionnalités.

## Utilisation du modèle de développement (Repository)

La première modification va consister à tirer partie du modèle de développement appelé (Repository). Nous allons utiliser ce modèle pour isoler le code d’accès aux données du reste de l’application.

L’implémentation de ce modèle impose de suivre deux étapes :

1. Créer une interface
2. Créer une classe qui implémente cette interface.

Tout d’abord, vous devez créer une interface qui décrit toutes les méthodes d’accès aux données. C’est l’interface IContactManagerRepository contenue dans le « Listing 1 ». Cette interface décrit les 5 méthodes CreateContact(), DeleteContact(), EditContact(), GetContact, and ListContacts().

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

using System;

using System.Collections.Generic;

namespace ContactManager.Models

{

public interface IContactRepository

{

Contact CreateContact(Contact contactToCreate);

void DeleteContact(Contact contactToDelete);

Contact EditContact(Contact contactToUpdate);

Contact GetContact(int id);

IEnumerable<Contact> ListContacts();

}

}

Ensuite, vous devez créer une classe qui implémente l’interface IContactManagerRepository. Comme nous avons utilisé Microsoft Entity Framework pour la couche d’accès aux données, nous devons créer une nouvelle classe EntityContactManagerRepository. Cette classe est contenue dans le listing 2.

**Listing 2 – Models\EntityContactManagerRepository.cs**

using System.Collections.Generic;

using System.Linq;

namespace ContactManager.Models

{

public class EntityContactManagerRepository : ContactManager.Models.IContactManagerRepository

{

private ContactManagerDBEntities \_entities = new ContactManagerDBEntities();

public Contact GetContact(int id)

{

return (from c in \_entities.ContactSet

where c.Id == id

select c).FirstOrDefault();

}

public IEnumerable ListContacts()

{

return \_entities.ContactSet.ToList();

}

public Contact CreateContact(Contact contactToCreate)

{

\_entities.AddToContactSet(contactToCreate);

\_entities.SaveChanges();

return contactToCreate;

}

public Contact EditContact(Contact contactToEdit)

{

var originalContact = GetContact(contactToEdit.Id);

\_entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit);

\_entities.SaveChanges();

return contactToEdit;

}

public void DeleteContact(Contact contactToDelete)

{

var originalContact = GetContact(contactToDelete.Id);

\_entities.DeleteObject(originalContact);

\_entities.SaveChanges();

}

}

}

On note que la classe EntityContactManagerRepository implémente l’interface IContactManagerRepositor. La classe implémente donc les 5 méthodes d’accès aux données décrites dans l’interface.

Vous pouvez alors vous demander pourquoi avons-nous besoin d’une interface ?

A une exception près, l’application devra maintenant interagir avec l’interface et non avec la classe. Plutôt que d’appeler les méthodes exposées par la classe EntityContactManagerRepository, vous devrez appeler les méthodes exposées par l’interface IContactManagerRepository.

Comme cela vous pourrez faire une nouvelle implémentation de l’interface sans être obligé de modifier les appels faits dans l’application. Vous pouvez par exemple avoir deux implémentations de l’interface IContactManagerRepository sur deux framework d’accès aux données ADO.NET Data Services avec la classe DataServicesContactManagerRepository et Microsoft Entity Framework avec la classe EntityContactManagerRepository. Il sera donc possible de basculer de l’une à l’autre sans modifier la couche de validation ou la couche du contrôleur.

La programmation par interface par rapport à la programmation par classe rend l’application plus robuste aux changements.

Vous pouvez créer une interface sur une classe depuis Visual Studio en sélectionnant le menu option « Refactor / Extract Interface ». Par exemple vous pouvez créer en premier la classe EntityContactManagerRepository et utiliser l’extraction d’interface de Visual Studio pour créer automatiquement l’interface IContactManagerRepository.

## Utilisation du modèle de programmation d’injection de dépendance.

Maintenant que vous avez migré le code d’accès aux données dans une classe « Repository », vous devez modifier le contrôleur Contact pour utiliser cette classe. Vous devez tirer partie du modèle de programmation « Injection de dépendance » afin d’utiliser la classe « Repository » dans le contrôleur.

Les modifications du contrôleur Contact sont contenues dans le listing 3.

**Listing 3 – Controllers\ContactController.cs**

using System.Text.RegularExpressions;

using System.Web.Mvc;

using ContactManager.Models;

namespace ContactManager.Controllers

{

public class ContactController : Controller

{

private IContactManagerRepository \_repository;

public ContactController()

: this(new EntityContactManagerRepository())

{}

public ContactController(IContactManagerRepository repository)

{

\_repository = repository;

}

protected void ValidateContact(Contact contactToValidate)

{

if (contactToValidate.FirstName.Trim().Length == 0)

ModelState.AddModelError("FirstName", "First name is required.");

if (contactToValidate.LastName.Trim().Length == 0)

ModelState.AddModelError("LastName", "Last name is required.");

if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone, @"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}"))

ModelState.AddModelError("Phone", "Invalid phone number.");

if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email, @"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"))

ModelState.AddModelError("Email", "Invalid email address.");

}

public ActionResult Index()

{

return View(\_repository.ListContacts());

}

public ActionResult Create()

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)

{

// Validation logic

ValidateContact(contactToCreate);

if (!ModelState.IsValid)

return View();

// Database logic

try

{

\_repository.CreateContact(contactToCreate);

return RedirectToAction("Index");

}

catch

{

return View();

}

}

public ActionResult Edit(int id)

{

return View(\_repository.GetContact(id));

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Edit(Contact contactToEdit)

{

// Validation logic

ValidateContact(contactToEdit);

if (!ModelState.IsValid)

return View();

// Database logic

try

{

\_repository.EditContact(contactToEdit);

return RedirectToAction("Index");

}

catch

{

return View();

}

}

public ActionResult Delete(int id)

{

return View(\_repository.GetContact(id));

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Delete(Contact contactToDelete)

{

try

{

\_repository.DeleteContact(contactToDelete);

return RedirectToAction("Index");

}

catch

{

return View();

}

}

}

}

On notera que le contrôleur du Listing 3 dispose de deux constructeurs. Le premier prend en paramètre une instance alors que le deuxième prend en paramètre une interface IContactManagerRepository. Le contrôleur Contact utilise le « Constructeur d’injection de dépendance ».

De ce fait, un seul utilise EntityContactManagerRepository (seulement le premier). Le reste de la classe utilise l’interface IContactManagerRepository. Il est ainsi facile de basculer d’une implémentation à une autre en changeant le premier constructeur. Passer de EntityContactManagerRepository à DataServicesContactRepository se fait à un seul endroit dans la classe.

Autre avantage : maintenant le contrôleur Contact est testable. Dans le cadre d’un test unitaire vous pouvez instancier le contrôleur Contact en passant par une fausse implémentation de IContactManagerRepository. Cette fonctionnalité est très importante et sera utilisée dans la prochaine étape quand vous allez écrire des tests unitaires sur l’application de Gestion de Contacts.

## Création d’une couche de service

Notez que la couche de validation restait mixée avec contrôleur dans le Listing 3. Pour les mêmes bonnes raisons qui nous ont amenées à isoler la couche d’accès aux données, nous allons isoler la couche de validation.

Pour se faire, nous allons créer une nouvelle couche: une couche de service. La couche de service doit se placer entre le contrôleur et la couche de « repository ». La couche de service contient la couche métier et toute la couche de validation.

La classe ContactManagerService se trouve dans le Listing 4. Elle contient la couche de validation initialement contenue dans le contrôleur Contact.

**Listing 4 – Models\ContactManagerService.cs**

using System.Collections.Generic;

using System.Text.RegularExpressions;

using System.Web.Mvc;

using ContactManager.Models.Validation;

namespace ContactManager.Models

{

public class ContactManagerService : IContactManagerService

{

private IValidationDictionary \_validationDictionary;

private IContactManagerRepository \_repository;

public ContactManagerService(IValidationDictionary validationDictionary)

: this(validationDictionary, new EntityContactManagerRepository())

{}

public ContactManagerService(IValidationDictionary validationDictionary, IContactManagerRepository repository)

{

\_validationDictionary = validationDictionary;

\_repository = repository;

}

public bool ValidateContact(Contact contactToValidate)

{

if (contactToValidate.FirstName.Trim().Length == 0)

\_validationDictionary.AddError("FirstName", "First name is required.");

if (contactToValidate.LastName.Trim().Length == 0)

\_validationDictionary.AddError("LastName", "Last name is required.");

if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone, @"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}"))

\_validationDictionary.AddError("Phone", "Invalid phone number.");

if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email, @"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"))

\_validationDictionary.AddError("Email", "Invalid email address.");

return \_validationDictionary.IsValid;

}

#region IContactManagerService Members

public bool CreateContact(Contact contactToCreate)

{

// Validation logic

if (!ValidateContact(contactToCreate))

return false;

// Database logic

try

{

\_repository.CreateContact(contactToCreate);

}

catch

{

return false;

}

return true;

}

public bool EditContact(Contact contactToEdit)

{

// Validation logic

if (!ValidateContact(contactToEdit))

return false;

// Database logic

try

{

\_repository.EditContact(contactToEdit);

}

catch

{

return false;

}

return true;

}

public bool DeleteContact(Contact contactToDelete)

{

try

{

\_repository.DeleteContact(contactToDelete);

}

catch

{

return false;

}

return true;

}

public Contact GetContact(int id)

{

return \_repository.GetContact(id);

}

public IEnumerable<Contact> ListContacts()

{

return \_repository.ListContacts();

}

#endregion

}

}

Notez que le constructeur de la classe ContactManagerService prend en paramètre un ValidationDictionary (Dictionnaire de données). La couche de service communique avec la couche contrôleur par ce dictionnaire de données. Nous reverrons ce dictionnaire de données en détails dans la suite de cette étape qui aborde le modèle « Décoration »

Notez de plus que la classe ContactManagerService implémente l’interface IContactManagerService toujours dans l’optique d’avoir une application faiblement couplée. Les autres classes l’application de Gestion de Contacts n’interagie pas directement avec ContactManagerService mais avec l’interface IContactManagerService.

L’interface IContactManagerService est contenue dans le Listing 5.

**Listing 5 – Models\IContactManagerService.cs**

using System.Collections.Generic;

namespace ContactManager.Models

{

public interface IContactManagerService

{

bool CreateContact(Contact contactToCreate);

bool DeleteContact(Contact contactToDelete);

bool EditContact(Contact contactToEdit);

Contact GetContact(int id);

IEnumerable ListContacts();

}

}

Les modifications pour le contrôleur Contact est contenue dans le Listing 6. Notez que le contrôleur n’interagit plus avec la classe ContactManager « Repository » mais bien avec l’interface IContactManagerService. Chaque couche est maintenant bien isolée.

**Listing 6 – Controllers\ContactController.cs**

using System.Web.Mvc;

using ContactManager.Models;

namespace ContactManager.Controllers

{

public class ContactController : Controller

{

private IContactManagerService \_service;

public ContactController()

{

\_service = new ContactManagerService(new ModelStateWrapper(this.ModelState));

}

public ContactController(IContactManagerService service)

{

\_service = service;

}

public ActionResult Index()

{

return View(\_service.ListContacts());

}

public ActionResult Create()

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)

{

if (\_service.CreateContact(contactToCreate))

return RedirectToAction("Index");

return View();

}

public ActionResult Edit(int id)

{

return View(\_service.GetContact(id));

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Edit(Contact contactToEdit)

{

if (\_service.EditContact(contactToEdit))

return RedirectToAction("Index");

return View();

}

public ActionResult Delete(int id)

{

return View(\_service.GetContact(id));

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Delete(Contact contactToDelete)

{

if (\_service.DeleteContact(contactToDelete))

return RedirectToAction("Index");

return View();

}

}

}

L’application suit désormais le modèle SRP (Single Responsibility Principe). Le contrôleur du Listing 6 n’a désormais plus que le rôle de contrôle du flux de l’application. La couche de validation lui à été enlevée et déportée dans une couche de service. Toute la couche d’accès aux données à été de la même manière déportée dans la couche « Repository ».

## Utilisation du modèle “Décoration”

Nous allons maintenant complètement découper la couche de service et la couche contrôleur. Logiquement, vous devriez être capable de compiler la couche de service dans une assembly séparée de la couche contrôleur sans avoir besoin d’y faire référence dans l’application MVC.

Cependant, la couche de service à besoin de passer les messages d’erreurs de validation à la couche contrôleur. Comment la couche de service va-t-elle communiquer les messages d’erreurs de validation sans coupler le contrôleur et la couche de service ? Pour cela, vous allez tirer avantage du modèle de programmation appelé « Modèle de décoration »

Un contrôleur utilise ModelStateDictionary appelé ModelState pour représenter les erreurs de validation. Nous sommes alors tentés de passer des « ModelState » entre la couche contrôleur et la couche de service. Cependant l’utilisation de ModelState dans la couche de service la rend dépendante aux fonctionnalités ASP.NET MVC. Ce n’est donc pas une bonne idée car cette couche de service pourrait être utilisée demain par une application WPF par exemple. Nous ne voulons pas dans notre cas rendre la couche de service dépendante d’ASP.NET MVC à travers l’utilisation de la classe ModelStateDictionary.

Le modèle de Décoration encapsule une classe existante dans une nouvelle classe afin d’implémenter une interface. Dans l’application de Gestion de Contacts, nous allons ajouter la classe ModelStateWrapper contenue dans le listing 7. La classe ModelStateWrapper implémente l’interface du Listing 8.

**Listing 7 – Models\Validation\ModelStateWrapper.cs**

using System.Web.Mvc;

namespace ContactManager.Models.Validation

{

public class ModelStateWrapper : IValidationDictionary

{

private ModelStateDictionary \_modelState;

public ModelStateWrapper(ModelStateDictionary modelState)

{

\_modelState = modelState;

}

public void AddError(string key, string errorMessage)

{

\_modelState.AddModelError(key, errorMessage);

}

public bool IsValid

{

get { return \_modelState.IsValid; }

}

}

}

**Listing 8 – Models\Validation\IValidationDictionary.cs**

namespace ContactManager.Models.Validation

{

public interface IValidationDictionary

{

void AddError(string key, string errorMessage);

bool IsValid {get;}

}

}

Si vous regardez en arrière dans le Listing 5, vous pourrez voir que la couche de service ContactManager utilise exclusivement l’interface IValidationDictionary. La couche ContactManagerService n’est donc pas dépendante de la classe ModelStateDictionary. Quand le Contact contrôleur créé la couche de Service, le contrôleur encapsule ModelState comme ceci :

\_service = new ContactManagerService(new ModelStateWrapper(this.ModelState));

## Conclusion

Dans cette étape nous n’avons pas ajouté de fonctionnalités à l’application de Gestion de Contacts. Le but de cette étape était bien de faire du “refactoring” (refonte) sur l’application pour la rendre plus facilement modifiable et maintenable.

Dans un premier temps nous avons implémenté le modèle de développement “Repository” et migré ainsi le code d’accès aux données dans une classe séparée.

Ensuite nous avons isolé la partie validation et logique dans une couche de service. La couche contrôleur interagit avec la couche de service et la couche de service interagissait avec la couche « Repository »

Par ailleurs, quand vous avez créé la couche de service, vous avez tiré avantage du modèle de Décoration pour isoler ModelState de la couche de service. Pour se faire, vous avez manipulé l’interface IValidationDictionary plutôt que ModelState.

Finalement, vous avez tiré avantage du modèle de programmation appelé “Injection de dépendance”. Ce modèle fait manipuler des interfaces par le développeur plutôt que des classes. L’implémentation du mode d’injection de dépendance rend aussi l’application plus testable. Dans la prochaine étape, vous allez ajouter des tests unitaires à votre application.

|  |
| --- |
| Ressources complémentairesAccéder aux autres étapes du tutoriel « [Développer d’une application de gestion de contacts avec ASP.NET MVC (C#)](http://msdn.microsoft.com/fr-fr/asp.net/dd627541.aspx) »[Découvrez et Formez-vous avec le Coach ASP.NET](http://msdn.microsoft.com/fr-fr/asp.net/msdn.coachaspnet.aspx)[Le site ASP.NET sur MSDN](http://msdn.microsoft.com/fr-fr/asp.net/default.aspx) |

Bas du formulaire