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

Etape 4 – Rendre l'application faiblement couplée

Dans cette 4ème étape, nous allons mettre en œuvre plusieurs modèles de développement logiciel (Design Patterns) pour que l'application devienne plus facile à maintenir et à modifier.

Point de départ

Au départ, on commence avec la classe du contrôleur Contact qui fait tout, de la validation des données saisies à leur lecture et à leur mise à jour dans la base de données.

Listing 1 – Controllers\ContactController.cs

```
using System.Linq;
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private ContactManagerDBEntities _entities = new ContactManagerDBEntities();

        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(_entities.ContactSet.ToList());
        }

        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
            {
                _entities.AddToContactSet(contactToCreate);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Edit(int id)
        {
            var contactToEdit = (from c in _entities.ContactSet
                                 where c.Id == id
                                 select c).FirstOrDefault();

            return View(contactToEdit);
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(Contact contactToEdit)
        {
            ValidateContact(contactToEdit);
            if (!ModelState.IsValid)
                return View();

            try
            {
                var originalContact = (from c in _entities.ContactSet
                                       where c.Id == contactToEdit.Id
                                       select c).FirstOrDefault();
                _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName,
contactToEdit);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Delete(int id)
        {
            var contactToDelete = (from c in _entities.ContactSet
                                   where c.Id == id
                                   select c).FirstOrDefault();

            return View(contactToDelete);
        }

        [AcceptVerbs(HttpVerbs.Post)]
```

```
        public ActionResult Delete(Contact contactToDelete)
        {
            try
            {
                var originalContact = (from c in _entities.ContactSet
                                       where c.Id == contactToDelete.Id
                                       select c).FirstOrDefault();

                _entities.DeleteObject(originalContact);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

    }
}
```

Utilisation du modèle « Repository »

On va suivre le modèle de développement « Repository » qui consiste à isoler le code chargé de l'accès aux données du reste de l'application.

Pour cela, on commence par créer une interface qui décrit les différentes méthodes dont on a besoin pour gérer les données, telles qu'on les trouve actuellement dans le contrôleur :

- Dans l'action Index, on récupère la **liste** de tous les contacts,
- Dans l'action Create, il y a du code pour **créer** un contact,
- Dans l'action Edit, on a du code pour **retrouver** un contact et pour **modifier** ce contact,
- Dans l'action Delete, on doit aussi **retrouver** un contact et **supprimer** ce contact.

Listing 2 – Models\IContactManagerRepository.cs

```
using System;
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactManagerRepository
    {
        Contact CreateContact(Contact contactToCreate);
        void DeleteContact(Contact contactToDelete);
        Contact EditContact(Contact contactToEdit);
        Contact GetContact(int id);
        IEnumerable<Contact> ListContacts();

    }
}
```

Dans cette interface, on décrit donc les 5 méthodes qui nous permettent d'accéder aux contacts : CreateContact(), DeleteContact(), EditContact(), GetContact() et ListContacts().

On crée ensuite une classe qui implémente cette interface IContactManagerRepository. Comme la couche d'accès aux données est confiée à Microsoft Entity Framework, on va nommer cette classe EntityContactManagerRepository.

Listing 3 – 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<Contact> 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();
        }
    }
}
```

Utilisation du modèle « Injection de Dépendance »

Maintenant que l'accès aux données est géré par notre classe Repository, il faut faire évoluer le code du contrôleur pour utiliser cette classe Repository. Pour cela, nous employons une technique appelée « l'injection de dépendance » : la classe contrôleur ne vas pas instancier la classe Repository dont elle a besoin, mais « injecter » un objet Repository

Listing 4 – 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();
            }
        }

    }
}
```

Par rapport à avant, on peut remarquer que notre contrôleur contient maintenant deux constructeurs. Le premier constructeur n'attend pas de paramètre. Il se contente d'instancier un objet EntityContactManagerRepository et de faire passer (« injecter ») celui-ci au second constructeur :

```
        public ContactController() : this(new EntityContactManagerRepository())
        {
        }
```

Le second constructeur attend comme paramètre une interface IContactManagerRepository :

```
        public ContactController(IContactManagerRepository repository)
        {
            _repository = repository;
        }
```

Grace à cette technique (appelée « Constructeur d'injection de dépendance »), seul le premier constructeur utilise la « vraie » classe EntityContactManagerRepository alors que tout le reste du code du contrôleur travaille uniquement avec l'interface IContactManagerRepository (l'application est faiblement couplée).

Avantages

1. Si un jour on passe de EntityFramework à NHibernate ou Subsonic, on n'aura qu'une ligne à modifier dans tout le contrôleur. Concrètement, ça veut dire qu'on peut démarrer un projet même si on ne sait pas encore trop quel framework d'accès aux données on va utiliser. Et quand on sera fixé sur le framework qui nous convient le mieux, le fait de faire un changement en cours de route devrait avoir un impact minime (l'application est plus robuste aux changements).
2. L'autre intérêt, c'est qu'on peut utiliser le second constructeur en lui passant une « fausse » implémentation de IContactManagerRepository, ce qui sera extrêmement pratique pour réaliser des tests unitaires (le contrôleur est testable).

Utilisation d'une couche de « Service »

Le contrôleur ne contient plus aucun code pour accéder aux données. Par contre, on y trouve encore du code lié à la validation de ces données (dans la procédure ValidateContact()).

Pour les mêmes bonnes raisons qui nous ont fait sortir la gestion des données du contrôleur, on va créer une « couche de service » entre le contrôleur et la couche de « repository ». Cette couche de service va contenir la couche métier et toute la couche de validation.

Listing 5 – Models\ContactManagerService.cs

```
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Web.Mvc;

namespace ContactManager.Models
{
```

```
    public class ContactManagerService : IContactManagerService
    {
        private ModelStateDictionary _validationDictionary;
        private IContactManagerRepository _repository;

        public ContactManagerService(ModelStateDictionary validationDictionary)
            : this(validationDictionary, new EntityContactManagerRepository())
        {}

        public ContactManagerService(ModelStateDictionary validationDictionary,
IContactManagerRepository repository)
        {
            _validationDictionary = validationDictionary;
            _repository = repository;
        }

        public bool ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                _validationDictionary.AddModelError("FirstName", "First name is required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                _validationDictionary.AddModelError("LastName", "Last name is required.");
            if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone,
@"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}"))
                _validationDictionary.AddModelError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email,
@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"))
                _validationDictionary.AddModelError("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
    }
}
```

Le constructeur de la classe ContactManagerService attend un paramètre de type ModelStateDictionary qui va servir pour échanger des informations entre le contrôleur et le service.

Par ailleurs, tout comme dans le cas du Repository, la classe ContactManagerService va en fait implémenter l'interface IContactManagerService. Toute l'application de « Gestion de Contacts » travaillera en priorité avec l'interface IContactManagerService pour que l'application soit faiblement couplée.

Listing 6 – 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();
    }
}
```

Il nous reste donc à modifier une nouvelle fois le code source du contrôleur pour que d'une part il ne contienne plus de code lié à la validation des données et que d'autre part il n'ai plus de lien direct avec le Repository mais laisse le Service interagir avec celui-ci.

Listing 7 – 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(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), c'est-à-dire que chaque « morceau » du code est chargé de faire une seule chose :

- Le contrôleur n'a plus qu'un rôle de contrôle du flux de l'application.
- La couche de validation a été déportée dans une couche « Service ».
- Toute la couche d'accès aux données à été déportée dans la couche « Repository ».

Utilisation du modèle « Décoration »

Notre objectif d'avoir une classe par responsabilité est donc bien atteint. Il nous reste malgré tout un petit souci : pour faire passer les messages d'erreurs de la couche service au contrôleur, on utilise des objets qui dépendent de ASP.NET MVC (un objet ModelStateDictionary en l'occurrence).

Cela posera problème le jour où on aura besoin de mettre à jour des contacts depuis autre chose qu'une application ASP.NET MVC (si on voulait faire une moulinette en mode console par exemple). Il faut donc trouver un moyen pour casser cette dépendance.

Pour cela, on utilise le modèle de conception « Décoration ». Cela consiste simplement à encapsuler une classe existante dans une nouvelle classe, ce qui dans le cas présent nous permet alors d'implémenter cette nouvelle classe en tant qu'interface.

On va donc encapsuler la classe ModelStateDictionary dans une classe ModelStateWrapper qui implémente l'interface IValidationDictionary.

Listing 8 – 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 9 – Models\Validation\IValidationDictionary.cs

```
namespace ContactManager.Models.Validation
{
    public interface IValidationDictionary
    {
        void AddError(string key, string errorMessage);
        bool IsValid {get;}
    }
}
```

Il faut ensuite modifier le code de la classe ContactManagerService pour ne plus utiliser d'objet ModelStateDictionary mais travailler avec l'interface IValidationDictionary à la place.

Listing 10 – 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;
        }

etc...
```

Par conséquent, lorsque la classe contrôleur créé la couche de service, elle ne doit plus utiliser directement un objet ModelStateDictionary mais l'encapsuler dans une classe ModelStateWrapper :

```
~~_service = new ContactManagerService(this.ModelState);~~
_service = new ContactManagerService(new ModelStateWrapper(this.ModelState));
```

Et il n'y a pas d'autres modifications à apporter : le framework ASP.NET MVC continuera à utiliser l'objet ModelState pour afficher les erreurs via les helpers Html.ValidationXxxxxx.

Récapitulatif

Le but de cette étape était de faire du « refactoring » (refonte) sur l'application pour la rendre plus facilement modifiable et maintenable. On est partit d'un contrôleur qui s'occupait de tout et on l'a réorganisé morceau par morceau.

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
- La couche de service interagit avec la couche « Repository »

Parallèlement, on a utilisé le modèle de programmation « **Injection de dépendance** » qui nous permet de manipuler des interfaces plutôt que des classes.

Et pour finir, nous avons modifié la couche de service pour tirer avantage du modèle de « **Décoration** » afin d'isoler ModelState de la couche de service.