Architecturer des applications métier Silverlight

Par Manuel Felicio - Deepin Prayag (traducteur)

Date de publication : 1 mars 2013

Dernière mise à jour : 7 mars 2013

TOUT PUBLIC

Cet article, en plusieurs parties, explique comment architecturer une application d'entreprise Silverlight centrée sur les données. Silverlight, lorsqu'elle est combinée avec les patterns les plus populaires aujourd'hui et certaines technologies-clés pour le génie logiciel, peut être une technologie très productive, simple, fiable et efficace pour construire ce genre d'application, et en même temps fournir une expérience utilisateur exceptionnelle.
Langage/Outil/Technologie : Silverlight 4, C#, Visual Studio 2010.
Commentez cet article : **Commentez**

Traduction....3
Partie 0 - Introduction....3
Conclusion....4
Partie 1 - Concepts et technologies....4
Conclusion....7
Partie 2 - Vue d'ensemble de l'architecture....8
Conclusion....10
Partie 3 - Services de domaine et repository....10
Conclusion....13
Partie 4 - Métadonnées et localisation....13
Conclusion....15
Partie 5 - Modules et composition d'interface utilisateur....15
Conclusion....21
Partie 6 - Construire un framework MVVM....21
Conclusion....25
Liens....25
Remerciements....25

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Traduction

Cet article est la traduction la plus fidèle possible de l'article de **Manuel Felicio**, **Architecting Silverlight LOB applications** (1) .

Partie 0 - Introduction

Cette partie est une introduction à la série d'articles que je vais faire sur comment architecturer une application d'entreprise Silverlight centrée sur les données. Alors que Silverlight est une plateforme puissante pour créer d'excellentes expériences utilisateurs interactives pour le Web, le Bureau (et maintenant) le Mobile, elle est aussi le meilleur atout de Microsoft pour créer une technologie multiplateforme et multinavigateur.

Beaucoup de gens voient juste Silverlight comme un autre rival d'Adobe Flash à cause des aspects applications Internet riches que proposent les deux technologies et la solide prise en charge multimédia dont dispose Silverlight. En dépit de cela, nous pouvons tirer nettement plus de bénéfices de Silverlight si on la perçoit en tant que bonne technologie pour l'industrie du logiciel dans les technologies de l'information. Les projets informatiques sont entièrement axés sur la manipulation, la visualisation et la compréhension des données.

Silverlight, qui est un sous-ensemble de WPF (excellente pour traiter des données à travers la liaison de données) et toujours une application .NET, lorsqu'elle est combinée avec les patterns les plus populaires aujourd'hui et certaines technologies-clés pour le génie logiciel, peut être une technologie très productive, simple, fiable et efficace pour construire ce genre d'applications, et en même temps, fournir une expérience utilisateur exceptionnelle.

Dans cette série d'articles je vais essayer d'aborder les aspects importants lorsque vous architecturez une application qui peut démarrer petite, qui est prévue pour durer des années et au fil du temps évoluer en une plus grosse application en raison de nouvelles exigences et être développée par plusieurs équipes sans qu'il en résulte un gros chantier de code auquel personne ne souhaite toucher.

De telles applications sont souvent construites en utilisant les principes suivants.

- Séparation d'aspects

Le principe le plus important. Possibilité de diviser pour régner par la création de couches dans votre application qui sont destinées à faire face à un problème, une tâche ou une fonctionnalité, réduisant ainsi la complexité. Le reste de ces principes sont tous basés sur celui-ci.

- Couplage faible

Possibilité de créer plusieurs composants qui peuvent œuvrer ensemble mais ne dépendent pas les uns des autres.

- Flexibilité

Capacité à répondre aux changements futurs, soit en changeant complètement la technologie sous-jacente ou les besoins métier.

- Extensibilité

Possibilité de créer de nouvelles fonctionnalités par l'extension ou la réutilisation de celles existantes.

- Modularité

Possibilité de créer des secteurs d'activités indépendants bien définis (cohésifs) dans votre application qui peuvent être amenés à évoluer séparément, à s'intégrer les uns aux autres et même être déployés séparément.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

- Testabilité

Capacité à éviter/prévenir les bogues inattendus par des modifications ou de nouvelles fonctionnalités que vous effectuez, en testant la logique, le comportement ou les dépendances de votre application.

- Maintenabilité

Capacité à trouver rapidement le code que vous devez modifier, de le comprendre et de le modifier, même si celui qui l'a écrit en premier lieu n'est plus là (ce qui arrive fréquemment).

- Réutilisabilité

Possibilité de réduire le code en créant des composants qui seront réutilisés par d'autres composants nécessitant une fonctionnalité spécifique.

- Indépendance de l'interface utilisateur

Possibilité de changer de contrôles de présentation sans casser ni votre logique de présentation, ni la logique applicative.

- Localisation

Possibilité de cibler plusieurs langues, augmentant ainsi la diversité de votre clientèle.

Je pense à démarrer avec une architecture globale, choisir les bonnes technologies, construire une petite application dessus et ensuite continuer à la « composer ». Je vais commencer par .NET 3.5 et plus tard passer à .NET 4.

Conclusion

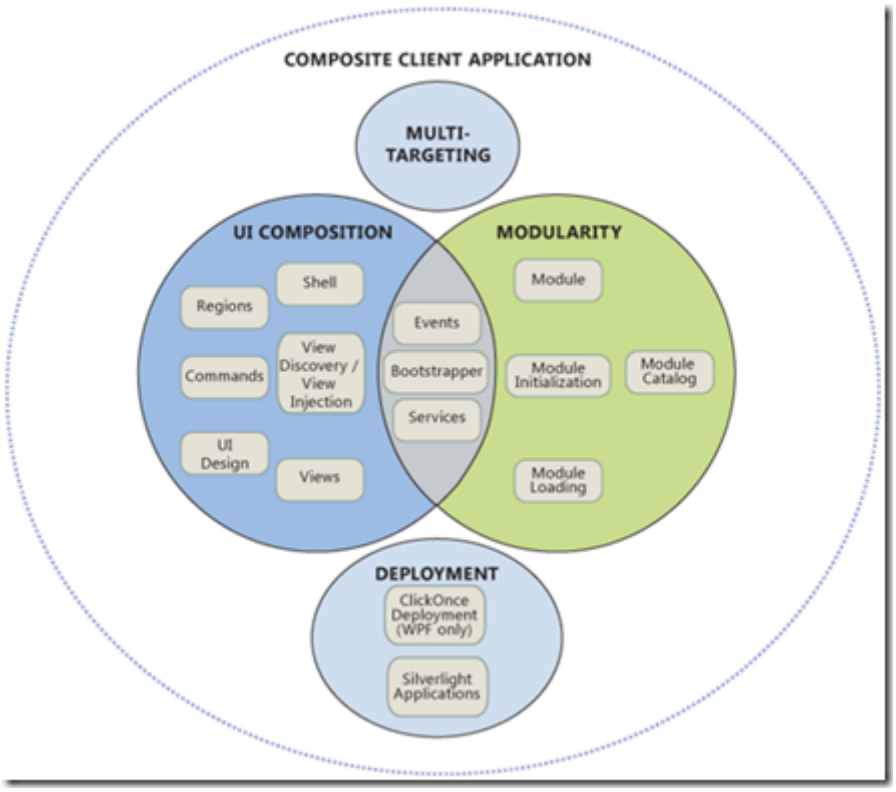
Dans la **prochaine partie** de cette série, je vais aborder les technologies qui sont disponibles et qui peuvent nous aider à accomplir notre objectif.

Partie 1 - Concepts et technologies

Dans cette première partie de la série je vais me concentrer sur certains projets open source dans « Patterns & Practices » de Microsoft et d'autres technologies qui sont disponibles et peuvent nous aider à accomplir notre objectif.

Nous avons d'abord **Prism** (« Composite Application Guidance for WPF et Silverlight »). Prism est composé d'une orientation et d'un ensemble de bibliothèques qui sont conçus pour vous aider à créer plus facilement des applications WPF et Silverlight modulaires. Il possède également des composants intégrés qui vous permettent de tirer profit de certains « patterns » connus tels que MVVM, l'agrégateur d'événements et l'injection de dépendances. Il en existe d'autres, mais je vais me concentrer sur ces trois, car je les trouve plus utiles.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Prism s'applique au développement client. Je pense que la meilleure caractéristique qu'offre Prism est la possibilité de spécifier vos modules applicatifs (XAP dans Silverlight) en tant que catalogue et de les charger à la demande. Nous allons explorer ceci en détail dans d'autres parties de cette série lorsque nous créerons un catalogue de modules à utiliser dans l'application.

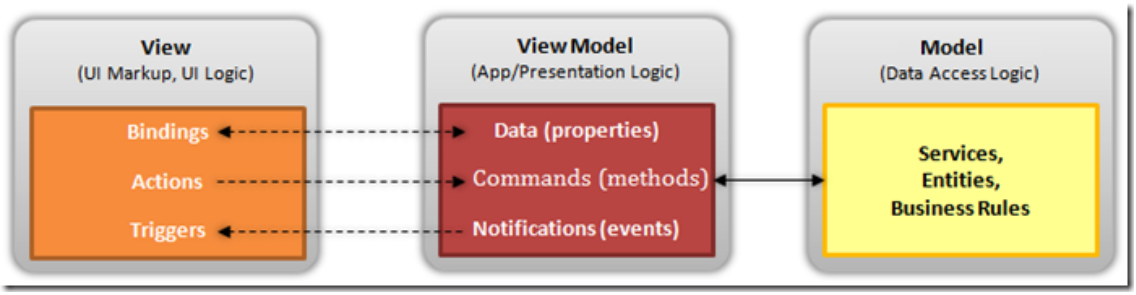
L'injection de dépendances (un usage spécifique de la notion d'inversion de contrôle) est une technique de choix pour construire des applications à couplage lâche, car il fournit des moyens de gérer les dépendances entre les objets. Il vous aide à construire du code testable, car vous pouvez facilement changer les implémentations concrètes desquelles vos objets dépendent. Prism est conçu en utilisant ce principe de conception et il utilise le conteneur Unity, qui implémente la technique d'injection de dépendances. Prism vous permet également d'utiliser d'autres conteneurs d'injection de dépendances, mais je vais me concentrer sur Unity, car je n'ai pas suffisamment d'expérience avec les autres.

Ce « pattern » peut être appliqué tant dans le développement client que serveur.

Je n'ai pas trouvé d'illustration bonne et simple à propos de l'injection de dépendances, je vais donc faire le lien vers **cet article** où j'ai utilisé cette technique dans un service WCF (OrderService) qui a utilisé un IOrderRepository implémenté par la classe OrderRepository dont les dépendances étaient injectées dans son constructeur.

Le « pattern » MVVM est en train de devenir l'alpha et l'oméga lorsqu'il s'agit de développement d'applications WPF et Silverlight. Il vous aide à séparer les aspects lors de la conception de votre couche de présentation et si elle est bien pensée, vous pouvez créer un bel ensemble de code réutilisable pour votre application et même le réutiliser pour d'autres applications.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Avec MVVM vous considérez essentiellement votre vue comme un ensemble de classes et de propriétés et non comme un ensemble de contrôles d'interface utilisateur. Cela vous permet de facilement changer votre interface utilisateur tout en conservant la même logique à laquelle elle est liée, et en même temps permet la testabilité. De plus, cela vous permet de réutiliser une partie de la logique d'interface utilisateur dans beaucoup de vues avec moins d'effort. Ce « pattern » est appliqué au développement client.

Le « pattern » **Event Aggregator** vous permet de construire votre application avec une architecture orientée événements où les composants communiquent entre eux de manière découplée. Il fournit une manière de publier et de s'abonner à des événements sans savoir qui s'abonne ou qui publie. La seule chose que vous devez savoir est l'événement que vous publiez ou auquel vous vous abonnez et comment réagir à cet événement. D'habitude, les modules ne dépendent pas les uns des autres et utiliser des événements comme contrats afin de communiquer entre eux est une bonne approche.

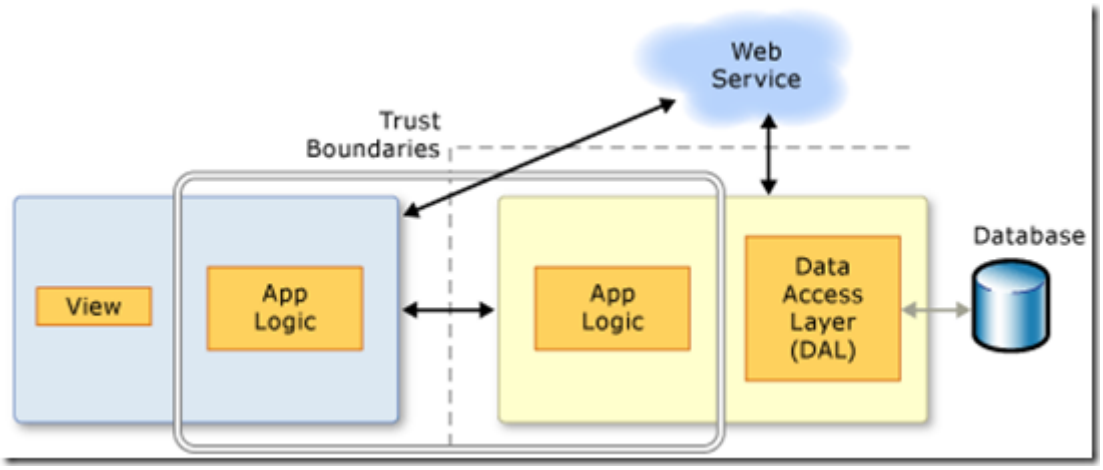
Nous allons maintenant nous concentrer sur le développement côté serveur.

Pour la communication client-serveur dans Silverlight, il est recommandé et imposé d'utiliser les services Web. Puisque nous construisons des applications métier il est logique d'utiliser des services qui sont bons pour traiter des données. Les services WCF sont certes bons, mais sont-ils idéaux#? Quel est le coût du maintien d'une couche de service complète qui nous permet d'exposer les opérations CRUD sur nos données et nous fournit également la capacité d'effectuer d'autres requêtes (telles qu'obtenir les produits par `productType`) ou d'autres opérations métier ? Combien de fois avons-nous fait cela ? Et combien de fois avons-nous fait cela pour différentes applications ? Qu'en est-il de la validation des données côté serveur et client ? Qu'en est-il de la génération de proxy ? Qu'en est-il de la quantité de données envoyées sur le réseau juste pour changer une valeur `DateTime` sur la base de données ?

Nous savons tous que le code généré par Visual Studio est terriblement laid et que le maintien des agents de service complets côté client peut être problématique, parce que vous avez d'habitude besoin de le garder synchronisé avec votre contrat de service. Dans Silverlight, il est un peu plus compliqué de partager un assembly avec des contrats de services et de données qu'en WPF, en raison d'incompatibilités de type. Aussi, d'habitude nous aimons que nos contrats de données côté client implémentent `INotifyPropertyChanged` et utilisent des `ObservableCollections` de manière à tirer profit de la liaison de données, mais du côté serveur on ne se soucie pas vraiment de ça.

WCF RIA Services vise à résoudre tous ces problèmes et bien d'autres. Je vais laisser tomber la partie « toutes les choses que peuvent faire les RIA Services » et **vous diriger au bon endroit** si vous ne le savez toujours pas. Les fonctionnalités les plus importantes que nous explorerons sont la composition des requêtes, la validation des données, le suivi des modifications, l'unité de travail, entre autres.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



J'ai utilisé RIA Services pendant les sept derniers mois et une chose que j'ai trouvée très utile est que la manière dont vous communiquez soit avec le serveur soit avec le modèle (du côté client) est par le biais d'une API très bien définie, ce qui rend facile la création de code très générique qui peut être utilisé pour créer des *viewmodels* réutilisables. C'est particulièrement utile lorsque vous avez plusieurs personnes travaillant sur le même projet et vous voulez assurer que chacun codera de la même façon (je compte bien explorer ceci par la suite).

Au moment de rédiger cet article, WCF RIA Services est disponible en version RC2 et a une licence « go-live », donc tout devrait fonctionner.

Cela ne veut pas dire que vous ne devriez pas également utiliser WCF. Il y a des cas où WCF RIA Services n'est pas adapté. Par exemple, le cas de téléchargement de fichier. Dans cette situation il est préférable de créer un service WCF ou un HttpHandler pour traiter ce cas particulier.

Pour notre couche de données les meilleures options sont de nos jours toujours NHibernate (utilisant Linq to NHibernate) ou Entity Framework. Néanmoins, nous devrions encapsuler notre accès aux données par des objets repository et ne pas utiliser notre technologie d'accès aux données directement dans nos services. J'ai un **article très simple** qui parle exactement de ça. RIA Services génère du code sur le client basé sur la réflexion et d'autres métadonnées qui sont appliquées dans nos entités côté serveur. Si vous utilisez Entity Framework avec RIA Services, il possède déjà un fournisseur de métadonnées spécifique qui peut comprendre les relations entre entités et les générer correctement dans le client. Si vous souhaitez utiliser NHibernate vous devrez spécifier des métadonnées supplémentaires dans vos entités. Pour la simplicité j'utiliserai Entity Framework. Il y a d'autres ressources sur le Web qui expliquent comment utiliser NHibernate.

Dernier point, mais non des moindres, je voudrais parler de MEF. MEF, qui signifie **Managed Extensibility Framework**, a été livré avec .NET 4 (et SL 4) et est conçu pour nous aider à composer nos applications d'une façon découplée beaucoup plus facile que l'utilisation d'un conteneur IoC pour l'injection de dépendances. Vous annotez simplement vos classes avec les attributs Import/Export et MEF résoudra les dépendances entre objets pour vous. Bien sûr, il y a plus que nous pouvons faire. Vous pouvez fournir des métadonnées en exportant un objet et importer uniquement des objets qui suivent certains critères. MEF peut être utilisé pour télécharger le XAP quand un « Import » doit être satisfait et que l'objet exporté est dans un autre XAP, ce qui aide la construction d'applications modulaires. Il paraît que la nouvelle version de Prism utilisera MEF pour ses fonctionnalités de modularité, ce qui le rend encore plus intéressant.

Conclusion

Dans la **prochaine partie**, je vais démarrer avec un scénario d'entreprise simple et suivre une approche de haut en bas en concevant une architecture construite sur ces technologies. Je n'ai pas été très disponible ces derniers temps, mais j'essaie de me rattraper. À bientôt, donc.

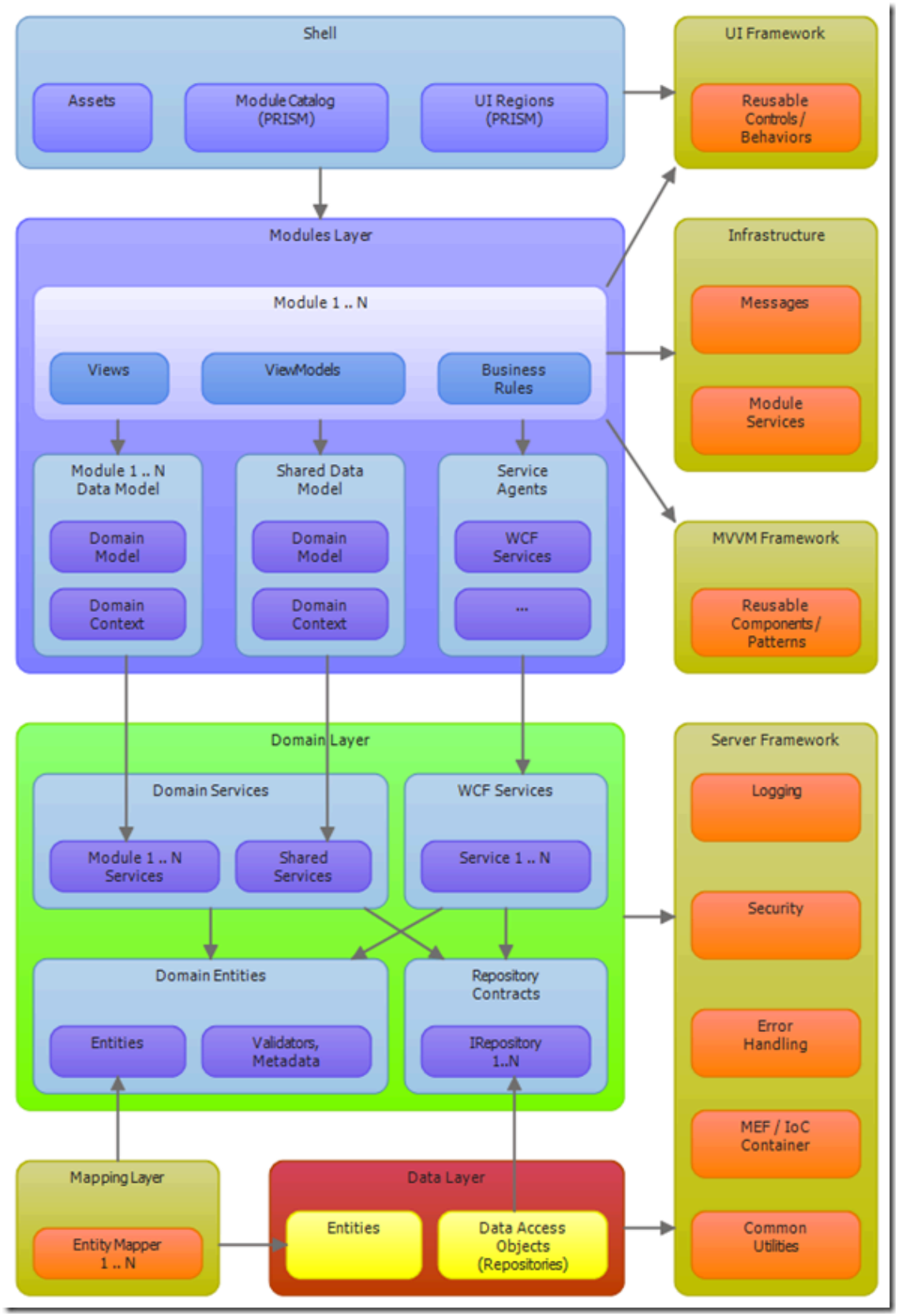
Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Partie 2 - Vue d'ensemble de l'architecture

Continuant notre série, je voudrais présenter une architecture sur laquelle je travaille depuis un certain temps et qui peut être utilisée dans des applications métier Silverlight utilisant le DDD (« **Domain Driven Development** ») et ayant plusieurs couches.

L'image suivante décrit comment une application peut être composée de multiples **modules**, tous hébergés dans des régions d'un **shell**. Ces modules peuvent communiquer les uns avec les autres par le biais de messages bien définis et des contrats définis dans le module de la couche d'**infrastructure**. En outre, ils extraient et manipulent des objets de données à travers des services WCF qui conviennent pour le DDD (appelés WCF RIA Services). Ces objets de données sont souvent appelés des entités et peuvent représenter votre **modèle de domaine** et sont pérennisés dans un magasin de données en utilisant des **objets d'accès aux données**. WCF RIA Services vous fournit des moyens de manipuler vos entités dans le client de la même façon que vous le feriez dans le serveur, lorsqu'il s'agit d'interroger, modifier des données et soumettre des modifications.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Une telle application compte souvent sur l'infrastructure du côté serveur pour l'authentification et l'autorisation, la journalisation et la gestion des erreurs. Du côté client elles comptent sur des *frameworks* qui aident à réduire, harmoniser et réutiliser du code dans le développement de modules tant pour la logique de présentation que pour les vues de présentation. Certaines couches peuvent avoir des composants qui dépendent de composants d'autres couches.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Dans de tels cas, nous réduisons ces dépendances avec des conteneurs IoC ou des *frameworks* similaires tels que MEF.

Le développement dans une architecture N-tiers peut être problématique lorsque vous souhaitez réutiliser du code qui doit être spécifié tant du côté serveur que du côté client quand il s'agit de la validation ou autre logique métier. WCF RIA Services comptent sur les métadonnées qui décrivent comment sont définies et validées les entités, et comment elles se comportent les unes avec les autres. Bien que cette technique soit très efficace, il y a des problèmes associés qui ne sont pas remarqués jusqu'à ce que vous les rencontriez, et j'essaierai d'en identifier et d'en résoudre quelques-uns.

Notez la couche de mapping entre les entités de données et les entités de domaine. Cette couche est optionnelle et peut être utile lorsque vous ne souhaitez pas exposer vos entités de données en tant que votre modèle de domaine. Cela aide à réaliser l'ignorance de persistance et vous permet aussi de créer des classes simples et légères pour transporter des données. Pour la simplicité, je ne vais pas utiliser cette couche dans les prochaines parties, donc mes entités de données seront mes entités de domaine.

Conclusion

Dans les prochaines parties, je vais construire une application qui respecte ces couches et qui évoluera au fil du temps. Je vais également construire un *framework* simple visant au développement MVVM afin de démontrer à quel point ce pattern peut être efficace.

Partie 3 - Services de domaine et repository

Dans la **partie précédente**, j'ai décrit une vue d'ensemble d'une architecture qui pourrait être utilisée lors du développement d'applications modulaires Silverlight. Dans cette partie je vais démarrer avec un modèle de domaine métier simple et je vais créer une couche de domaine autour de ce modèle.

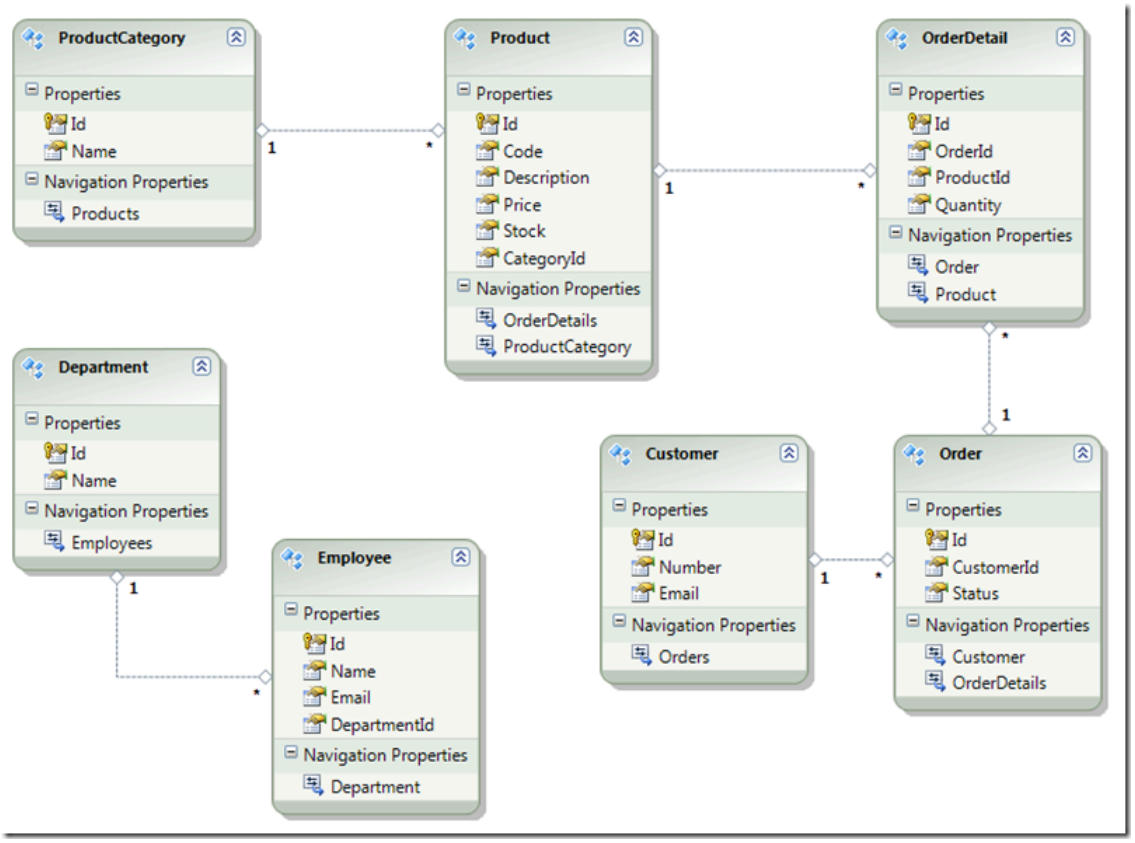
Dans cet exemple notre application sera une application Web pour une société qui vend des produits et qui peut être utilisée par ses clients pour passer des commandes et par ses employés pour les traiter. Ces commandes contiennent des détails qui consistent en des produits et des quantités. Chaque produit peut avoir une catégorie de produit. Quand un client soumet une commande, un employé de la société peut traiter cette commande et préparer les articles pour l'expédition.

La société a quelques départements et chaque employé fait partie d'un département. Cela signifie que les employés du service des ventes devront traiter les commandes ; les employés du département des ressources humaines pourront recruter de nouveaux employés ; et les employés du département des finances pourront observer quelques graphiques et imprimer des rapports avec des statistiques sur comment évoluent les ventes avec le temps, des rapports de ventes, etc.

L'authentification et l'autorisation seront par la suite ajoutées.

Par souci de simplicité, j'ai créé le modèle de données suivant :

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Ce modèle sera tant notre modèle de données que notre modèle de domaine. C'est très simple et comme je l'ai dit dans la **partie précédente**, créer un modèle de domaine séparé est facultatif, car vous pouvez toujours bien vous en sortir avec vos entités de données. Puisque nous utilisons Entity Framework nous allons tirer profit de LinqToEntitiesDomainService, un service de domaine spécifique qui traite des objets de l'Entity Framework.

Puisque nous commençons par notre couche de domaine, nous allons définir un service de domaine qui peut être utilisé depuis notre application Silverlight (ou d'autres clients également) et créer les opérations CRUD. Je suis sûr que vous avez remarqué combien de code (dupliqué) est généré lorsque vous utilisez le modèle de service de domaine de Visual Studio. Eh bien… plus besoin de souffrir ! Créez tout simplement une classe de base pour vos services de domaine et ajoutez les méthodes suivantes :

```
1. #region Basic CRUD
2. protected IQueryable<T> GetQuery<T>()
3.     where T : class
4. {
5.     return this.GetRepository<T>().GetQuery();
6. }
7. protected void Insert<T>(T entity)
8.     where T : class
9. {
10.     this.GetRepository<T>().Add(entity);
11. }
12. protected void Update<T>(T entity)
13.     where T : class
14. {
15.     this.GetRepository<T>().AttachAsModified(entity, this.ChangeSet.GetOriginal(entity));
16. }
17. protected void Delete<T>(T entity)
18.     where T : class
19. {
20.     this.GetRepository<T>().Delete(entity);
21. }
22. #endregion
23.
24. protected IRepository<T> GetRepository<T>()
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
25.     where T : class
26. {
27.     return this.Container.Resolve<IRepository<T>>();
28. }
```

La propriété Container est une instance de IUnityContainer, que je garde dans le service de domaine de base. J'ai d'habitude une classe singleton commune où le conteneur IoC est disponible mais je crée aussi un conteneur enfant pour chaque service de domaine. Cela me permet d'enregistrer des instances spécifiques dans le conteneur enfant et de l'utiliser uniquement dans la portée de mon service de domaine. C'est utile pour résoudre des *repositories* parce que mes *repositories* dépendent de l'ObjectContext qui est injecté via l'injection de dépendances. Voici comment on procède :

```
1. public abstract class DomainRepositoryService<TContext> :
 LinqToEntitiesDomainService<TContext>, IDbContextManager
2.     where TContext : ObjectContext, IDbContext, new()
3. {
4.     protected IUnityContainer Container { get; private set; }
5.     /// <summary>
6.     /// Create a child container to use THIS as the IDbContextManager implementation when
 resolving repositories
7.     /// Ensures that other threads / domain services can use the main container to resolve
 repositories
8.     /// and use its own instance as the IDbContextManager for the repositories they need
9.     /// </summary>
10.     /// <param name="context">The DomainServiceContext instance</param>
11.     public override void Initialize(DomainServiceContext context)
12.     {
13.         base.Initialize(context);
14.
15.         this.Container = IoC.Current.Container.CreateChildContainer();
16.         this.Container.RegisterInstance<IDbContextManager>(this);
17.     }
18.
19.     #region IDbContextManager Members
20.     public IDbContext GetDbContext()
21.     {
22.         return this.ObjectContext;
23.     }
24.     #endregion
25.     //...
26. }
```

Ceci dit, le service de domaine spécifique pour notre application Silverlight ressemblera à ceci :

```
1. [EnableClientAccess()]
2. public class MyAppService : DomainRepositoryService<MyAppEntities>
3. {
4.     #region Customer
5.     public IQueryable<Customer> GetCustomers()
6.     {
7.         return base.GetQuery<Customer>();
8.     }
9.     public void InsertCustomer(Customer entity)
10.     {
11.         base.Insert(entity);
12.     }
13.     public void UpdateCustomer(Customer entity)
14.     {
15.         base.Update(entity);
16.     }
17.     public void DeleteCustomer(Customer entity)
18.     {
19.         base.Delete(entity);
20.     }
21.     #endregion
22.     //repeat for the other entities...
23. }
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Remarquez comment ma couche de domaine ne se soucie pas vraiment de comment les entités sont stockées ou gérées avec EF (Entity Framework). Il délègue tout simplement la responsabilité à un objet qui implémente l'interface IRepository.

Cette interface peut être générique et ressemble souvent à :

```
1. public interface IRepository<TEntity>
2.     where TEntity : class
3. {
4.     TEntity GetEntity(int id);
5.     IQueryable<TEntity> GetQuery();
6.     void Add(TEntity entity);
7.     void Update(TEntity entity);
8.     void Delete(TEntity entity);
9.     /// <summary>
10.     /// Abstraction on RIA's AttachAsModified extension method
11.     /// </summary>
12.     /// <param name="current"></param>
13.     /// <param name="original"></param>
14.     void AttachAsModified(TEntity current, TEntity original = null);
15. }
```

Pour la simplicité, j'ai inclus la méthode AttachAsModified dans l'IRepository. Cette méthode est importante parce que RIA exige que l'entité en cours de mise à jour soit attachée dans l'ObjectContext. Une autre option serait de créer une interface séparée qui dérive d'IRepository et d'y ajouter la méthode. C'est que l'interface IRepository est un concept générique qui peut être utilisé avec plusieurs technologies d'accès de données.

Notez que dès que vous créez votre service de domaine et exposez vos entités, tout de suite une autre personne/équipe peut commencer à construire l'application utilisant les entités exposées, tandis qu'une autre équipe continue à travailler sur la couche serveur ajoutant des métadonnées et/ou la logique de validation qui sera également disponible pour l'application cliente.

Conclusion

Dans la **prochaine partie**, j'aborderai la spécification des métadonnées pour nos entités et cela devrait être suffisant pour la couche côté serveur, pour l'instant.

Partie 4 - Métadonnées et localisation

La spécification des métadonnées pour nos entités est la partie la plus facile et c'est un bon endroit pour commencer la localisation de notre application.

La première des choses que nous avons à faire est de créer une classe de métadonnées telle que décrite dans l'exemple suivant :

```
1. [MetadataType(typeof(Customer.CustomerMetadata))]
2. public partial class Customer
3. {
4.  internal class CustomerMetadata
5.  {
6.  [Display(ResourceType = typeof(Metadata), Name = "MyApp_Customer_Number")]
7.  public int Number;
8.
9.  [Display(ResourceType = typeof(Metadata), Name = "CommonProperties_Email")]
10.  public string Email;
11.
12.  [Include]
13.  public EntityCollection<Order> Orders;
14.  }
15. }
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
1. [MetadataType(typeof(Employee.EmployeeMetadata))]
2. public partial class Employee
3. {
4.     internal class EmployeeMetadata
5.     {
6.         [Display(ResourceType = typeof(Metadata), Name = "CommonProperties_Name")]
7.         [StringLength(128, ErrorMessageResourceType = typeof(Metadata),
 ErrorMessageResourceName = "CommonErrors_StringLength")]
8.         public string Name;
9.
10.         [Display(ResourceType = typeof(Metadata), Name = "CommonProperties_Email")]
11.         public string Email;
12.
13.         [Include]
14.         public Department Department;
15.     }
16. }
```

L'attribut DisplayAttribute est très utile, car il annote les propriétés avec des informations qui peuvent être consommées de différentes manières. Par exemple, lorsque vous utilisez votre entité dans une datagrid, les entêtes des cellules vont automatiquement chercher DisplayAttribute et extraire la chaîne localisée pour cette propriété. La même chose s'applique pour les labels dans une dataform et si vous voulez créer votre propre *framework* d'interface utilisateur, vous pouvez exploiter ce type d'annotation pour commencer la localisation de votre application. Remarquez comment j'ai réutilisé des noms de ressources pour les propriétés communes comme Email. La même chose aurait pu être faite pour Number et Name, mais cela dépend vraiment de chaque cas.

Vous pouvez également placer les règles de validation sur vos propriétés en utilisant des attributs comme StringLength, appliquer des expressions régulières ou même créer vos propres règles de validation personnalisées. Le plus intéressant ici étant que ces règles seront également disponibles côté client. Le DomainService fait en sorte qu'avant l'exécution des opérations sur les entités, toutes les règles de validation sont appliquées. Notez que vous ne pouvez pas faire toutes sortes de validation sur vos entités en utilisant des métadonnées. Certaines validations peuvent nécessiter la connexion à une base de données, transactions, etc., et cette information n'est pas disponible dans ce type de contexte de validation. Je recommande que de telles validations soient effectuées après l'appel à la méthode ExecuteChangeSet du DomainService et juste avant la méthode PersistChangeSet.

Plus de détails sur la spécification des métadonnées peuvent être trouvés dans la documentation ou d'autres ressources sur la toile. Je veux simplement mettre l'accent sur un aspect différent sur lequel peu de gens ont tendance à écrire. C'est le partage des ressources entre le client et le serveur. Le projet client ne compilera pas si les métadonnées ResourceManager ne sont pas disponibles. Comme elles sont définies sur le serveur, vous devez suivre ces étapes afin de les activer sur le client :

- ajoutez les fichiers de ressources en tant que lien dans le projet client ;
- modifiez l'espace de noms par défaut de votre projet client au même espace de noms qui est défini dans le projet serveur qui contient les fichiers de ressources ;
- modifiez le .csproj du projet client là où il fait référence au fichier lié Designer.cs et ajoutez ces lignes de code :

```
1.         <Compile Include="..\..\..\Server\Data\MyApp.Data\Resources\Metadata.Designer.cs">
2.
3.               <AutoGen>True</AutoGen>      <Link>Resources\Metadata.Designer.cs</Link>
4.
5.               <DesignTime>True</DesignTime>
6.
7.               <DependentUpon>Metadata.resx</DependentUpon>
8.
9.          </Compile>
```

- assurez-vous que les fichiers de ressources ont le même chemin relatif dans le projet client que dans le projet serveur. Dans ce cas, j'ai créé un dossier Resources dans les deux projets.
- assurez-vous également que le fichier de ressources a marqué son modificateur d'accès comme public.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Ces étapes sont nécessaires, car l'ajout de fichiers liés pour les fichiers dépendants, tels que Medatata.resx / Metadata.Designer.cs ne va pas automatiquement les marquer comme dépendants, vous devez donc le faire manuellement.

Maintenant, vous devriez être en mesure de compiler votre application et commencer à tirer profit des métadonnées dans vos entités.

Conclusion

Dans la **prochaine partie**, nous aborderons le développement client.

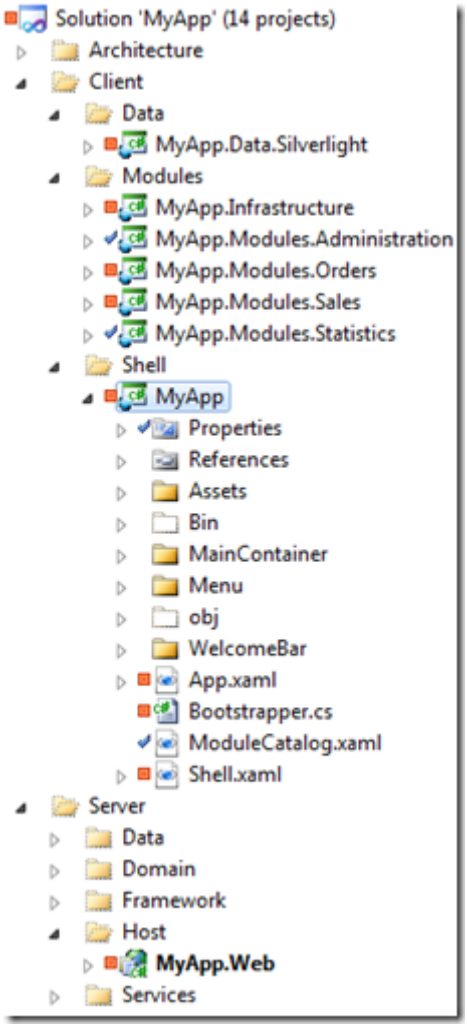
Partie 5 - Modules et composition d'interface utilisateur

Dans cette partie et **la prochaine**, je vais parler du développement client et de comment créer une application Silverlight modulaire et composite. Dans cette partie, je vais me concentrer sur la composition de l'interface utilisateur.

Lors de la construction d'applications métier Silverlight composites, vous devez être conscient que la modularité est une exigence importante. En tant que développeur, il devrait être plus facile pour vous de maintenir votre code et d'ajouter de nouvelles fonctionnalités à votre application sans avoir à changer le reste, de plus le fichier XAP de votre application ne grossit pas. Chaque module est un fichier XAP séparé, qui peut être téléchargé à la demande, selon l'intention de l'utilisateur d'utiliser une certaine fonctionnalité.

Ci-dessous est une architecture possible afin de réaliser ce dont nous avons besoin :

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Le dossier « Shell » contient le *xap* principal, que j'ai appelé MyApp. Ce *xap* est uniquement responsable de télécharger les autres et de leur fournir les moyens pour afficher leur contenu dans l'application principale. Le dossier « Modules » contient quatre modules essentiels pour le scénario d'entreprise que j'ai défini dans une partie précédente. Le module « Orders » permet aux utilisateurs non-employés de passer de nouvelles commandes. Le module « Sales » permet aux employés de consulter les commandes en cours et de les traiter. Le module « Statistics » permet aux employés de visualiser des graphiques et des rapports de l'évolution des ventes. Les fonctionnalités pour le département des ressources humaines sont accomplies par le module « Administration », qui devrait également permettre aux utilisateurs de l'application d'être créés et/ou gérés. Les modules et le Shell sont compilés dans le dossier ClientBin de « MyApp.Web ».

Le projet « Infrastructure » est un élément-clé. Il définit les moyens de communiquer entre les modules (les modules ne référencent pas d'autres modules) et l'application principale. En d'autres termes, il s'agit d'un contrat commun partagé par les modules. Lorsqu'il est défini ou partiellement défini, le développement des modules peut commencer. Notez que le développement du module « Orders » ne dépend pas du développement du module « Sales » ou du module « Statistics », ce qui signifie que des personnes différentes peuvent travailler sur chacun d'eux.

MyApp.Data.Silverlight est un projet qui devrait posséder du code généré par WCF RIA Services, ainsi que les clients d'autres services qui sont utilisés par les modules.

La couche serveur correspond à l'architecture que j'ai définie dans la partie 1.

Donc, comment notre Shell télécharge-t-il les modules ? Nous avons juste besoin de spécifier un catalogue de modules et le donner au programme d'amorçage de Prism, qui est responsable de l'initialisation du *framework* de PRISM. Ci-dessous le catalogue de modules que j'ai défini pour ce scénario :

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
<prism:ModuleCatalog xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    xmlns:prism="clr-
namespace:Microsoft.Practices.Composite.Modularity;assembly=Microsoft.Practices.Composite">

    <prism:ModuleInfo Ref="MyApp.Modules.Administration.xap"
                      ModuleName="Administration"
                      ModuleType="MyApp.Modules.Administration.Module,
 MyApp.Modules.Administration, Version=1.0.0.0"
                      InitializationMode="WhenAvailable">
    </prism:ModuleInfo>

    <prism:ModuleInfo Ref="MyApp.Modules.Statistics.xap"
                      ModuleName="Statistics"
                      ModuleType="MyApp.Modules.Statistics.Module, MyApp.Modules.Statistics,
 Version=1.0.0.0"
                      InitializationMode="WhenAvailable">
    </prism:ModuleInfo>

    <prism:ModuleInfo Ref="MyApp.Modules.Sales.xap"
                      ModuleName="Sales"
                      ModuleType="MyApp.Modules.Sales.Module, MyApp.Modules.Sales,
 Version=1.0.0.0"
                      InitializationMode="WhenAvailable">
    </prism:ModuleInfo>

    <prism:ModuleInfo Ref="MyApp.Modules.Orders.xap"
                      ModuleName="Orders"
                      ModuleType="MyApp.Modules.Orders.Module, MyApp.Modules.Orders,
 Version=1.0.0.0"
                      InitializationMode="WhenAvailable">
    </prism:ModuleInfo>
</prism:ModuleCatalog>
```

Notez les attributs `InitializationMode`. Ils nous permettent de spécifier si les modules doivent être téléchargés dès que possible ou s'ils doivent être téléchargés sur demande. Pour l'instant, je souhaite que mes modules soient téléchargés dès que possible, plus tard, je vais vous montrer des techniques efficaces pour les télécharger sur demande. Toutefois, cela n'affecte pas la façon dont je les construis.

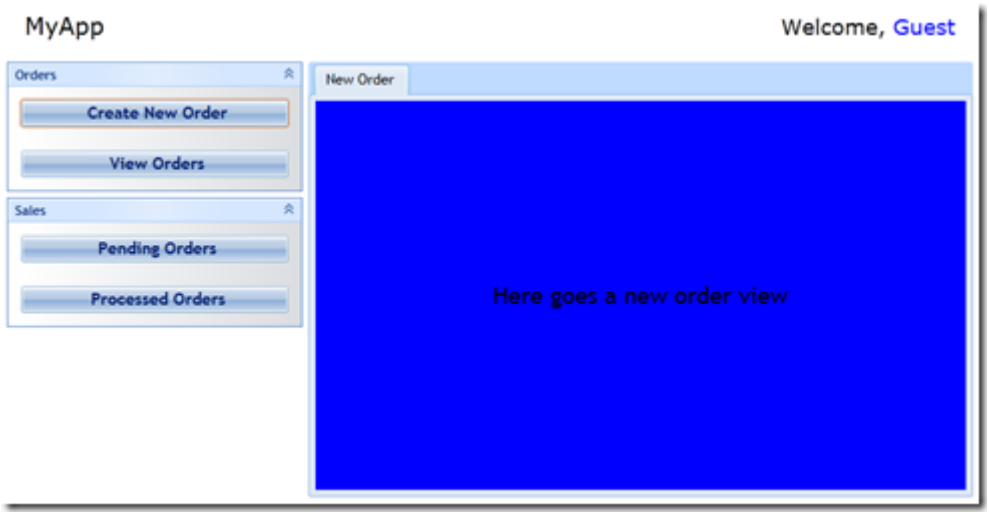
Pour commencer, notre Shell peut avoir une barre supérieure qui affiche un message de type « hello », un menu sur la gauche et une région où les modules peuvent afficher leur contenu.

```
    <Border BorderThickness="5" CornerRadius="10">
        <tlk:DockPanel LastChildFill="True">
            <shellWelcome:WelcomeBar tlk:DockPanel.Dock="Top" Margin="5"/>

  <shellMenu:Menu tlk:DockPanel.Dock="Left" VerticalAlignment="Stretch" Margin="5,0,5,5"/>
            <shellContainer:MainContainer Margin="0,0,5,5"/>
        </tlk:DockPanel>
    </Border>
```

Le résultat visuel est quelque chose comme ceci :

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/



Je voudrais que le menu et le conteneur principal soient dynamiques, en d'autres termes, avoir son contenu défini dynamiquement par des modules au lieu d'être défini dans le contrôle lui-même.

Nous pourrions créer une interface comme IMenu, implémentée par le *viewmodel* de notre menu et l'utiliser par des modules pour afficher du contenu. Bien que cela semble être une approche découplée, car vos modules peuvent afficher leur contenu sans dépendre du contrôle du menu lui-même, il est effectivement couplé à l'objet qui implémente l'interface IMenu. Cela signifie que si vous souhaitez afficher vos options du menu dans d'autres endroits, comme une barre d'outils ou une barre de ruban, vous aurez besoin d'une autre interface pour interagir avec ces contrôles. Une solution à ce problème consiste à utiliser une technique de publication/souscription de messages s'appuyant sur un objet médiateur, l'*EventAggregator* de Prism. Maintenant, ce que vous faites est la publication d'un message disant « Je veux afficher cette option du menu » et quel que soit le contrôle qui est intéressé par l'affichage des options du menu, cela s'abonne au message et agit en conséquence. Voyons comment nous pouvons faire cela :

```
1. public class MenuOptionMessage : CompositePresentationEvent<MenuOptionMessageArgs> { }
2.
3. public class MenuOptionMessageArgs
4. {
5.     public ICommand Command { get; set; }
6.     public string Text { get; set; }
7.     public string ImageSourceUri { get; set; }
8.
9.     public string Group { get; set; }
10. }
```

MenuOptionMessage est un événement Prism dont la charge utile est un message MenuOptionMessageArgs. Donc, chaque fois que vous souhaitez publier un MenuOptionMessage vous créez un MenuOptionMessageArgs et dites à l'*EventAggregator* de le diffuser à tous les abonnés. L'*EventAggregator* peut être accédé par l'injection de dépendances ou le conteneur IoC. J'ai l'habitude de créer une classe utilitaire simple pour garder le code plus propre :

```
1. public class Messenger
2. {
3.     public static T Get<T>()
4.         where T : EventBase
5.     {
6.         var ev = ServiceLocator.Current.GetInstance<IEventAggregator>();
7.         return ev.GetEvent<T>();
8.     }
9. }
```

Maintenant, nous pouvons faire publier des MenuOptionMessage par nos modules.

Initialisation du module « Order » :

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
1. Messenger.Get<MenuOptionMessage>().Publish(new MenuOptionMessageArgs
2. {
3.     Command = Commands.NewOrderCommand,
4.     Text = "Create New Order",
5.     Group = "Orders"
6. });
7. Messenger.Get<MenuOptionMessage>().Publish(new MenuOptionMessageArgs
8. {
9.     Command = Commands.ViewOrdersCommand,
10.     Text = "View Orders",
11.     Group = "Orders"
12. });
```

Initialisation du module « Sales » :

```
1. Messenger.Get<MenuOptionMessage>().Publish(new MenuOptionMessageArgs
2. {
3.     Command = Commands.ViewPendingOrdersCommand,
4.     Text = "Pending Orders",
5.     Group = "Sales"
6. });
7. Messenger.Get<MenuOptionMessage>().Publish(new MenuOptionMessageArgs
8. {
9.     Command = Commands.ViewProcessedOrdersCommand,
10.     Text = "Processed Orders",
11.     Group = "Sales"
12. });
```

Le contrôle Menu affiche toutes les commandes groupées par leur « Group » et crée une extension pour chaque « Group ». Vous pouvez accéder au code source pour confirmer cela.

Maintenant, il nous reste à faire afficher des vues par nos commandes.

Prism a un concept de région, où vous définissez des régions au sein de votre Shell, leur donnez un nom et ensuite vos modules injectent des vues dans ces régions en spécifiant le nom. Je ne suis pas amateur de cette fonctionnalité, car elle crée un couplage étroit entre les vues et les régions. En outre cela signifie que votre logique écran fonctionne uniquement avec Prism. J'utilise d'habitude une approche différente, qui nous permet non seulement d'utiliser les régions Prism, mais également d'autres *frameworks* d'interfaces utilisateurs. En outre, il n'est pas difficile de construire votre propre *framework* d'interface utilisateur, si vous le simplifiez. Par exemple, au lieu de penser en termes de régions, pensez en termes de position de vues.

```
1. public enum ViewPosition
2. {
3.     Left,
4.     Right,
5.     Top,
6.     Bottom,
7.     Center,
8.     Floating
9. }
```

En utilisant la même technique que celle pour le menu, chaque fois que nous voulons afficher une vue, nous avons tout simplement à publier un message où nous spécifions la position et la vue et quel que soit celui qui est responsable d'afficher cette vue dans la position spécifiée devrait s'abonner à ce message et agir en conséquence. Les messages devraient être définis dans l'*assembly* Infastructure. Voyons comment le message est défini :

```
1. public class CreateViewMessage : CompositePresentationEvent<CreateViewMessageArgs> { }
2.
3. public class CreateViewMessageArgs
4. {
5.     public Lazy<object> View { get; set; }
6.     public ViewPosition Position { get; set; }
7.     public string Title { get; set; } //optional
8. }
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

Notez l'objet Lazy. Cela vous permet d'avoir votre vue créée par l'abonné au lieu d'être créée par l'émetteur. Cela vous permet d'utiliser différents threads lors de la publication/abonnement, bien que dans ce cas l'abonné devrait utiliser le thread de l'interface utilisateur.

Ci-dessous un exemple de comment publier un CreateViewMessage à partir du module « Order ».

```
1. public class Commands
2. {
3.     public static ICommand NewOrderCommand = new ActionCommand(Commands.NewOrder);
4.     public static ICommand ViewOrdersCommand = new ActionCommand(Commands.ViewOrders);
5.
6.     private static void NewOrder()
7.     {
8.         Messenger.Get<CreateViewMessage>().Publish(new CreateViewMessageArgs
9.         {
10.             View = new Lazy<object>(() => ServiceLocator.Current.GetInstance<OrderView>()),
11.             Position = ViewPosition.Center,
12.             Title = "New Order"
13.         });
14.     }
15.
16.     private static void ViewOrders()
17.     {
18.         //to be implemented
19.     }
20. }
```

Notez l'OrderView qui est créé par le ServiceLocator pour permettre l'injection de dépendances, si nécessaire.

Le contrôle MainContainer est responsable de la souscription à ce message et d'afficher la vue dans un contrôle onglet.

```
1. public partial class MainContainer : UserControl
2. {
3.     public MainContainer()
4.     {
5.         InitializeComponent();
6.         if (DesignerProperties.IsInDesignTool)
7.             return;
8.
9.         Messenger.Get<CreateViewMessage>().Subscribe(this.HandleCreateView,
 ThreadOption.UIThread, true, this.CanHandleCreateView);
10.     }
11.
12.     private bool CanHandleCreateView(CreateViewMessageArgs args)
13.     {
14.         return args.Position == ViewPosition.Center;
15.     }
16.
17.     private void HandleCreateView(CreateViewMessageArgs args)
18.     {
19.         this.tabCtrl.Items.Add(new TabItem
20.         {
21.             Header = args.Title, Content = args.View.Value,
22.             VerticalContentAlignment = System.Windows.VerticalAlignment.Stretch,
23.             HorizontalContentAlignment = System.Windows.HorizontalAlignment.Stretch
24.         });
25.     }
26. }
```

Lorsque vous vous abonnez à un message, vous pouvez spécifier un prédicat qui dit au mécanisme de publication si vous souhaitez vraiment gérer ce message. Dans ce cas-ci, le conteneur principal ne veut gérer que des CreateViewMessage pour des vues qui doivent être affichées dans la position Center. Je pourrais avoir un autre composant s'abonnant au CreateViewMessage et afficher des vues à l'intérieur de fenêtres lorsque la position est Floating. Le paramètre ThreadOption est également important. Chaque message qui traite de l'interface utilisateur doit être souscrit dans le thread de l'interface utilisateur. Si vous souhaitez brancher un enregistreur afin de souscrire/

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

consigner les messages, vous n'avez pas besoin de vous souscrire sur le thread de l'interface utilisateur. Dans ce cas, vous devez utiliser un thread d'arrière-plan.

Il s'agit d'une approche très simple pour faire de la composition d'interface utilisateur qui fonctionne et qui est très efficace.

Conclusion

Dans la **prochaine partie** je vais me concentrer sur les *viewmodels* et comment nous pouvons construire l'OrderView.

Partie 6 - Construire un framework MVVM

Dans cette partie, je vais parler de la construction d'un *framework* MVVM.

Comme je le disais dans la **partie précédente**, cette partie devrait être sur l'OrderView. Cette vue devrait permettre aux utilisateurs de choisir des éléments d'une liste de produits, les ajouter à la commande en cours, choisir les quantités, voir le prix total actuel et soumettre la commande. Aussi, l'utilisateur doit être en mesure de filtrer, trier et avoir une pagination sur la liste des produits. Je suis sûr que vous avez vu plein d'articles et de billets de blogs d'autres personnes parlant de ce sujet, c'est-à-dire obtenir le produit sélectionné, l'ajouter à une autre collection ou créer un détail de la commande basé dessus, puis mettre à jour certaines autres données sur l'interface utilisateur et, enfin, soumettre les modifications au serveur. Le truc, c'est que tout le monde a sa propre façon de programmer et lorsque vous vous retrouvez éventuellement dans une équipe, il se peut que vous constatiez que deux personnes ont codé la même chose d'une manière différente, et que l'une a un bogue dans la situation A et l'autre a un bogue dans la situation B. Avoir un bon *framework* MVVM avec une méthodologie bien définie est indispensable pour prévenir ces situations. Dans cette partie, je souhaite parler des composants essentiels que vous devez avoir dans un bon *framework* MVVM. Je vous décrirai par la suite un *framework* MVVM sur lequel je travaille qui était basé sur WCF RIA Services, mais dont il ne dépend pas vraiment.

Puisque nous suivons les bonnes pratiques, nous savons qu'en utilisant une bonne architecture MVVM nous pouvons arriver à une solution dont la logique d'extraction, de filtrage, de tri, de pagination est entièrement séparée de la vue, nous permettant d'avoir aussi des vues différentes pour le même *viewmodel*. Par exemple, nous pouvons commencer par utiliser un DataGrid et un DataPager pour afficher nos articles, mais plus tard, fournir une nouvelle vue qui utilise des *combobox* pour sélectionner les options de tri, une *listbox* de type album pour afficher les articles et des boutons personnalisés pour la pagination. De même, nous devrions être en mesure de séparer toute cette logique de sa logique d'accès aux données réelles pour pouvoir utiliser des objets factices pour notre modèle et d'effectuer des tests unitaires pour nos *viewmodels*. Ce n'est pas une tâche facile mais c'est ce que je souhaite réaliser à partir de maintenant.

Eh bien, pour commencer, .NET/Silverlight nous offre déjà quelques classes et interfaces qui sont très pratiques dans le cadre de scénarios MVVM.

- INotifyPropertyChanged - Utilisée pour déclencher un événement lorsqu'une propriété change. Le *framework* de liaison WPF/Silverlight utilise cette interface pour mettre à jour la vue lorsqu'une propriété change.
- INotifyCollectionChanged - Utilisée pour déclencher un événement lorsqu'une opération d'insertion, de suppression, d'effacement ou de remplacement a été effectuée sur une collection. Les contrôles WPF/Silverlight qui ont une propriété ItemsSource utilisent d'habitude cette interface pour créer ou supprimer des éléments visuels dans un conteneur. Par exemple, des ListBox affichent de nouvelles ListBoxItem, des DataGrid affichent de nouvelles DataGridRow.
- ICollectionView - Utilisée pour fournir le filtrage, des descriptions de tri, des descriptions du groupe, et la sélection d'items pour une collection IEnumerable et faire que la vue n'affiche que les éléments filtrés, triés selon les descriptions de tri et mettre en évidence l'élément sélectionné. (Possède davantage de fonctionnalités, mais ce sont les plus pertinentes dans le cadre de cet article.)
- IPagedCollectionView - Utilisée pour fournir des options de pagination à une collection IEnumerable. Celles-ci sont utilisées essentiellement par des DataPager, qui font des appels à la méthode MoveToPage(int

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

pageIndex) et nous permettent d'enregistrer dans l'événement PageChanging et d'aller chercher une nouvelle page d'entités à afficher.
- Il existe d'autres interfaces importantes telles que IEditableObject et IEditableCollectionView, mais je ne les aborderai pas dans cet article. Elles sont utilisées pour mettre à jour les valeurs des propriétés d'un objet de façon atomique.

Malheureusement, les collections WCF RIA Services n'implémentent pas ICollectionView ou IPagedCollectionView, donc nous devons fournir notre propre mécanisme de filtrage, de tri et de pagination des données. Toutefois, cela est assez facile à faire avec LINQ et la classe **Expression** et nous pouvons composer des requêtes avec certains opérateurs de LINQ avec WCF RIA Services. Silverlight a sa propre implémentation de IPagedCollectionView, appelée System.Windows.Data.PagedCollectionView mais elle a été conçue pour fonctionner avec les collections en mémoire et non pas les « collections » (sources de données) côté serveur. Dans cette série, je fournis ma propre implémentation IPagedCollectionView/ICollectionView que j'ai conçue pour fonctionner avec n'importe quel type de sources de données. Quelle que soit l'option utilisée cela revient entièrement au modèle dans MVVM.

Donc, voyons comment fonctionne mon *framework* MVVM.

Tout d'abord, nous avons le concept du modèle :

```
1. public interface IDomainModel : IDisposable
2. {
3.     void RejectChanges();
4.     ISubmitOperation SaveChanges();
5.     ISubmitOperation SaveChanges(Action<ISubmitOperation> callback, object userState);
6.
7.     IDomainQuery<T> GetQuery<T>();
8.
9.     void Add<T>(T entity);
10.     void Remove<T>(T entity);
11.     void Attach<T>(T entity);
12.     void Detach<T>(T entity);
13.
14.     bool HasChanges { get; }
15.     bool IsSubmitting { get; }
16.     bool IsLoading { get; }
17. }
```

Si vous êtes habitué à l'API DomainContext, vous trouverez cette interface très facile à comprendre. Cette interface modèle est censée vous fournir des fonctionnalités de base pour extraire des données, les modifier et enregistrer les modifications. Pour des opérations plus spécifiques, vous pouvez étendre cette interface, ajouter des opérations spécifiques et faire en sorte que votre *viewmodel* en dépende.

L'interface IDomainQuery représente une requête sur un certain *repository* d'un type d'entité (quel qu'il soit). Comme une requête, elle vous permet de filtrer, trier et paginer les données.

```
1. public interface IDomainQuery : IDisposable
2. {
3.     void Cancel();
4.     void Load();
5.
6.     event Action Loading;
7.     event Action<ILoadOperation> Loaded;
8.     event Action Canceled;
9. }
10.
11. public interface IDomainQuery<T> : IDomainQuery
12. {
13.     /// <summary>
14.     /// Delegate that returns a lambda expression to apply in a Where operation
15.     /// </summary>
16.     Func<Expression<Func<T, bool>>> FilterExpression { get; set; }
17.
18.     /// <summary>
19.     /// SortDescription values that are applied in a OrderBy/ThenBy operation
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
20.     /// </summary>
21.     Func<SortDescriptionCollection> SortDescriptions { get; set; }
22.
23.     /// <summary>
24.     /// Contains the required data that are applied in a Skip and Take operation
25.     /// </summary>
26.     Func<PageDescription> PageDescription { get; set; }
27.
28.     new event Action<ILoadOperation<T>> Loaded;
29. }
30.
31. public sealed class PageDescription
32. {
33.     public int PageSize { get; set; }
34.     public int PageIndex { get; set; }
35. }
```

En outre, elle possède des événements pour vous permettre de savoir quand elle a commencé et terminé une opération de chargement. Il y a deux choses importantes à noter dans l'interface IDomainQuery. La première est le fait que la façon dont vous filtrez une IDomainQuery est la même que pour filtrer une RIA Services EntityQuery, ou tout autre objet IQueryable. Cela signifie qu'il est très facile de réaliser des implémentations IDomainQuery qui interrogent les *repositories* en mémoire ou des *repositories* côté serveur (notez qu'EntityQuery n'implémente pas IQueryable et qu'il n'est pas facile d'intégrer les deux concepts). L'autre chose importante est que le mécanisme de filtrage, de tri, de pagination est basé sur les délégués, ce qui signifie que chaque fois que la requête est exécutée/réexécutée ces délégués sont toujours réévalués et vont toujours refléter l'état actuel de votre *viewmodel*.

Dernier point, mais non des moindres, nous avons les opérations. J'ai créé des interfaces pour les opérations « Load » et « Submit ». J'ai implémenté toutes ces interfaces avec des classes qui s'appuient sur WCF RIA Services. Je ne vais pas poster le code de ces classes, mais ils sont disponibles dans le dépôt du code source, donc vous pouvez l'y consulter.

Si vous êtes habitué à DomainContext, vous êtes sans doute en train de vous demander où sont les EntitySet et où vous pouvez manipuler les collections. Eh bien, puisque mon implémentation IDomainModel s'appuie sur WCF RIA Services, vous pouvez étendre cette classe et manipuler votre DomainContext pour implémenter toute opération que vous souhaitez rendre disponible dans votre IDomainModel étendue. Mais avec ce *framework* elles ne sont pas censées être accessibles dans votre *viewmodel*. Au lieu de cela, j'utilise mon implémentation ICollectionView, que j'ai appelée DomainCollectionView.

```
1. public interface IDomainCollectionView : ICollectionView, IPagedCollectionView,
 INotifyPropertyChanged
2. {
3.     void Add(object item);
4.     void Remove(object item);
5. }
6.
7. public interface IDomainCollectionView<T> : IDomainCollectionView, IEnumerable<T>
8. {
9.     void Add(T item);
10.     void Remove(T item);
11. }
```

L'implémentation actuelle s'appuie sur une IDomainQuery et sera une vue pour les résultats qui sont récupérés au cours de chaque opération. Puisque vous pouvez implémenter une IDomainQuery pour toute IEnumerable vous pouvez utiliser cette implémentation pour n'importe quelle source de données.

```
1. public class DomainCollectionView<T> : ViewModelBase, IDomainCollectionView<T>
2. {
3.     private IDomainQuery<T> query;
4.     private ObservableCollection<T> sourceCollection;
5.
6.     #region DomainCollectionView Members
7.
8.     public DomainCollectionView(IDomainQuery<T> query, bool usePaging = true)
9.     {
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
10.         this.query = query;
11.         this.sourceCollection = this.CreateSourceCollection();
12.         this.CurrentItem = null;
13.         this.CurrentPosition = -1;
14.
15.         //Apply sort and paging
16.         query.SortDescriptions = () => this.SortDescriptions;
17.         if (usePaging)
18.         {
19.             this.canChangePage = true;
20.             this.pageSize = 10;// default page size to 10
21.             query.PageDescription = () => new PageDescription { PageSize = this.PageSize,
 PageIndex = this.PageIndex };
22.         }
23.
24.         query.Loaded += this.OnDataRefreshed;
25.     }
26.
27.     private void OnDataRefreshed(ILoadOperation<T> loadOp)
28.     {
29.         using (this.DeferRefresh())
30.         {
31.             this.sourceCollection.Clear();
32.             foreach (var entity in loadOp.Entities)
33.                 this.sourceCollection.Add(entity);
34.
35.             this.ItemCount = loadOp.TotalEntityCount;
36.             this.TotalItemCount = loadOp.TotalEntityCount;
37.         }
38.     }
39.
40.     //...
41. }
```

Maintenant que j'ai introduit le modèle et l'IDomainCollectionView, il est temps que nous parlions un peu du *viewmodel*.

Contrairement au modèle, les *viewmodels* ne sont pas tellement réutilisables, car ils dépendent généralement des besoins de la vue. Cependant, plusieurs vues dans les applications métier ont des besoins communs : la recherche/énumération des entités, édition des entités, créer des associations entre entités. Nous pouvons créer des *viewmodels* pour ces concepts et les étendre uniquement pour implémenter la logique métier. Toute l'infrastructure et la plomberie peuvent être centralisées et réutilisées partout.

Avant de terminer, je vais déposer simplement le premier jet d'un *viewmodel* de base qui peut être utilisé pour rechercher/énumérer des entités.

```
1. public class ListVM<T> : ViewModelBase, IListVM<T>
2. {
3.     private IDomainModel model;
4.
5.     public ListVM() { }
6.
7.     protected virtual IDomainModel CreateModel()
8.     {
9.         return ServiceLocator.Current.GetInstance<IDomainModel>();
10.     }
11.
12.     public void Initialize(IListVMInitializeArgs args)
13.     {
14.         this.model = args.ExistingModel ?? this.CreateModel();
15.         var query = model.GetQuery<T>();
16.         this.Entities = new DomainCollectionView<T>(query, args.RequiresPaging);
17.         query.Load();
18.     }
19.
20.     public IDomainCollectionView<T> Entities
21.     {
22.         get;
```


Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

```
23.         private set;
24.     }
25. }
```

Avec seulement quelques lignes de code, nous avons un *viewmodel* de base qui peut :

- afficher des résultats d'une requête qui peuvent être triés et paginés sur le serveur. Liez simplement une *datagrid* à la collection Entities et manipulez-la ;
- réutiliser un modèle existant ou en créer un nouveau. L'implémentation actuelle peut utiliser un modèle partagé pour toute l'application qui est enregistré dans le conteneur d'injection de dépendances actuel ou laisser un *viewmodel* dérivé décider quoi faire.

Je vous laisse y réfléchir.

Conclusion

Cette série est maintenant terminée. À l'avenir je pourrais creuser davantage pour voir comment nous pouvons créer des *viewmodels* de base pour tous les concepts cités plus haut et ajouter plus de fonctionnalités aux *viewmodels*.

Liens

Pour terminer, je tenais à dire que tout le code est disponible sur **http://code.google.com/p/mfelicio-wordpress** sous *trunk/2010/SL-MVVM-RIA*.

Remerciements

Je tiens ici à remercier **Manuel Felicio** de m'avoir autorisé à traduire son article.

Je remercie **tomlev** pour sa relecture technique et ses propositions.

Je remercie également **f-leb**, **zoom61**, **ClaudeLELOUP** et **Bovino** pour leurs relectures orthographiques et leurs propositions de corrections respectives.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/

1 : Le blog de l'auteur et l'article en anglais ne sont plus accessibles depuis quelques semaines.

Copyright ® 2013 Manuel Felicio. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://deepin.developpez.com/articles/silverlight/architecturer-des-applications-metier-silverlight/