Tome 8

Silverlight

Olivier Dahan

Collection
ALL DOT BLOG

e-naxos

www.e-naxos.com

Formation – Audit – Conseil – Développement

XAML (Windows Store, WPF, Silverlight, Windows Phone), C#

Cross-plateforme Windows / Android / iOS / Xamarin

UX Design

ALL DOT.BLOG

Tome 8

Silverlight

Tout Dot.Blog mis à jour par thème sous la forme de livres PDF gratuits

Olivier Dahan
odahan@gmail.com

Table des matières

Présentation du Tome 8 - Silverlight 13
XAML pour Silverlight 14
Silverlight, Blend et Design, 11 vidéos et un manuel 15
44 Videos US et 11 Ateliers en français sur Silverlight! 15
Silverlight/Blend : bien plus que des exemples… 18
Les Splash screens Silverlight 19
De quoi s'agit-il ? 19
Pourquoi personnaliser le splash screen ? 21
Le splash screen, un objet pas comme les autres 21
Le projet de démonstration 23
Le Splash screen 28
Conception technique 32
Code Source 41
Conclusion 41
Centrer un splash screen personnalisé avec Silverlight 42
Silverlight et la sérialisation 43
Pas de sérialisation ? 43
A quoi sert la sérialisation ? 43
Pas de gestion de disque local en RIA 44
En tout logique 44
Une solution 45
Le DataContract 45
Un exemple de code 45
Conclusion 47
Gestion des cookies 48
Application transparente 50
Contrôle DataForm et Validation des données 51
DeepZoom 57
Projection 3D 59

Un média player complet bien caché ! 60
Silverlight et le multithread (avec ou sans MVVM) 62
Nouveau Paradigme ? 62
Mais paradoxal ! 64
En pratique on fait comment ? 65
« Il neige » ou les boucles d'animation, les fps et les sprites sous Silverlight 78
Le but 79
Le principe 79
Le sprite Flocon 80
Le constructeur 80
La méthode « Tombe » 81
La boucle d'animation 84
Le contrôle des FPS 85
Le Rendering 85
Le plein écran 86
Le code 86
Happy Holidays ! 86
Un peu de technique 87
Pas d'héritage de UserControl 87
Améliorations diverses 88
Image de fond 88
Conclusion 89
Convertir une appli Flash en sources Silverlight ou WinRT! 89
Blocage IE avec écran multitouche sous Silverlight 89
Plantage avec Silverlight ? 91
Sécurité cachée ! 91
Problème numéro 1: Identifier cette fichue fenêtre. 91
Comment autoriser l'application bloquante ? 92
Et ça marche ! 93
Gérer des paramètres personnalisés du plugin Silverlight 94

Exemple : gadgets et menus 94
Déclarer le paramètre 94
Choix du lancement de l'application 95
La prise en charge du paramètre 96
Conclusion 98
Isolated Storage et stockage des préférences utilisateur 98
Démonstration 99
Stockage isolé ? 99
Mais pas un coffre-fort ! 100
Les quotas 100
Comme un disque privé 100
Les bonnes pratiques 101
La gestion simple des paramètres 102
Dialogue JavaScript / Silverlight 104
Appeler et se faire appeler 104
Appeler Silverlight depuis JavaScript 104
Appeler JavaScript depuis Silverlight 105
Code Exemple 105
Charger une image dynamiquement 108
Réagir au changement de taille du browser 110
Localiser une application 111
Localiser 111
Respecter la culture 112
La culture sous .NET 112
Localiser une application 113
Plus loin que la traduction 121
Conclusion 122
Gestion du clic droit 122
L'exemple 122
Le code 123

Conclusion 124
Charger une image sur l'hôte 124
Un bref exemple 124
Le code 125
Conclusion 127
Le multitâche 127
Exemple live 127
La structure du programme 129
La classe Job 129
La création des jobs 131
Le dialogue avec l'application 133
Le lancement des jobs 135
Conclusion 136
Optimiser les performances des traitements graphiques 136
Le Debug 136
Les vidéos 138
Le Texte 139
Le plugin Silverlight 140
Autres conseils 141
Conclusion 144
Les accès cross domaines 145
Un (faux) espoir pour le futur (qui n'arrivera pas) 145
Les plans B 145
L'explication du danger 146
Cross domain policy 148
ClientAccessPolicy.xml 148
Conclusion 149
Silverlight libère le chemin (Path n'est plus sealed) 150
Petit mais costaud 150
Simple et pratique 150

Ecrivez vos propres formes 150
Conclusion 153
Webcam, Capture et Prise de vue... 153
L'exemple 153
Le code 155
Conclusion 164
WCF Service pour Silverlight 164
Web Service classique vs. WCF Service 164
Services WCF 166
"Silverlight-Enabled" WCF Services 169
Conclusion 169
Produire et Utiliser des données OData en Atom avec WCF et Silverlight 170
Qu'est-ce que OData ? 170
Exposer des données OData 172
Le client Silverlight 180
L'aspect sécurité 185
Conclusion 186
Intégrer Bing Maps dans vos applications Silverlight 187
L'exemple live 187
Je veux le faire aussi ! 188
Passons aux choses sérieuses 189
Conclusion 191
Chargement Dynamique de DLL 192
Halte aux applications joufflues ! 192
L'exemple live 193
Comment ça marche ? 195
Conclusion 199
Accéder à l'IP du client 200
La technique du Web Service 200
La technique de la page Aspx 200

Conclusion 203
Compression des fichiers Silverlight Xap 204
Conclusion 205
Silverlight et le mode fullscreen 205
Silverlight et la Sécurité 205
Un compromis qu'on aimerait voir plus souvent… 206
Le code 207
Imprimer avec Silverlight 209
Print Screen 209
Comment ça marche ? 210
C'est tout ? 211
Des Extensions ? 211
Un générateur d'état minimaliste 211
Conclusion 214
Du Son pour Silverlight ! 214
Son et Image sont les mamelles d'une application Web réussie 214
Un "moteur" de mobylette 216
Les solutions actuelles 217
D'autres références 220
Conclusion 221
Personnaliser un thème du toolkit Silverlight 221
Les thèmes 222
Motivation 223
Personnaliser un thème du Toolkit 223
Conclusion 227
Lire un fichier en ressource sous Silverlight 228
ClientBin 228
Content ou Resource ? 228
Mode Content 230
Mode Resource 231

Conclusion 232
Silverlight : gérer une base de données locale 233
Utiliser l'Isolated Storage 233
Le cas de l'Out Of Browser (OOB) 233
Les solutions payantes 233
Le vrai gratuit et open source 234
Silverlight Database 236
Compatibilité ascendante 236
Les fonctionnalités 237
Conclusion 239
Silverlight : L'Isolated Storage en pratique 240
L'Isolated Storage 240
La gestion de l'espace 241
Les paramètres utilisateurs 243
La gestion des Logs 246
Le stockage de fichiers 253
Conclusion 254
Quand browsers et proxies jouent à cache-cache avec vos Xap... 255
Vider moi ce cache ! 255
Tromper les caches 255
Quelques balises de plus 256
Conclusion 260
MEF et Silverlight 4+ [Applications modulaires] 260
Code Source 260
Préambule 261
MEF – Le besoin 263
Du lego, pas un puzzle ! 266
MEF et MVVM 267
Conclusion 355
Validation des données sous Silverlight 355

IDataErrorInfo 355
Les membres de IDataErrorInfo 356
Implémenter IDataErrorInfo 356
Retour visuel sous Xaml 359
Faiblesse 360
Conclusion 360
Silverlight 5 : exemples de code des nouveautés 361
Ancestor RelativeSource Binding 361
Implicit DataTemplates 362
ClickCount 366
Binding on Style Setter 366
Trusted applications in-browser 367
Realtime Sound (low-latency Audio) 368
Variable Speed Playback (“Trick Play”) 368
Linked Text Containers 369
Custom Markup Extensions 369
XAML Binding Debugging 373
3D Graphics API 374
Additional Silverlight 5 Features Included in this Beta 375
Silverlight 5 Features not included in this Beta 376
Legal Notice 376
Conclusion 377
Silverlight 5 : Les nouveautés 377
Des innovations majeures 378
Les nouveautés 379
Conclusion 382
Les extensions de balises Xaml de Silverlight 5, vous connaissez ? 383
Custom Markup Extensions 383
Les bases 384
Mise en œuvre simplifiée 385

Un peu plus sophistiqué, simuler "'Static" de WPF 388
Conclusion 391
Lorem Ipsum generator : un autre exemple Silverlight 391
Fonctionnalités 392
Conclusion 393
Le code source de "Lorem Ipsum Generator" 393
Le logiciel Silverlight 393
Le code source 393
Silverlight 5 – Fuite mémoire avec les images (une solution) 394
Bogue fantôme ? 394
Un code simple 394
Le problème ? 395
Une solution ? 396
Ca plante toujours ! 397
Enfin la solution… 398
Conclusion 400
Encoder en ANSI ou autre avec Silverlight 400
La classe Encoding 400
Un générateur de classes dérivant de Encoding 401
Conclusion 402
Avertissements 403
E-Naxos 403

Table des Figures

Figure 1 - Le splash screen de l'application de démo « codes postaux français » 19
Figure 2 - Création du projet Silverlight 23
Figure 3 - Le choix du type de solution 24
Figure 4 - Arborescence de la solution 24
Figure 5 -Ecran de test 25
Figure 6 - Paramétrage de la page de test 26
Figure 7 - La conception du splash 32
Figure 8 - Création de la page XAML du splash 33
Figure 9 - Configuration d'Expression Design 34
Figure 10 - Adapter le code XAML exporté depuis Design 34
Figure 11 - Les fichiers "ralentisseurs" 38
Figure 12 - Ralentir le chargement avec Sloppy 39
Figure 13 - Le Splash en action ! 40
Figure 14 - capture application de test de la gestion des cookies 49
Figure 15 - Test de la validation des données 53
Figure 16 - Exemple DeepZoom 58
Figure 17 - Média player templaté 61
Figure 18 - Mécanisme de l'accès inter thread 67
Figure 19 - Schéma de fonctionnement du Smart Dispatcher 69
Figure 20 - Il Neige ! 79
Figure 21 - Happy Holidays ! 87
Figure 22 - Capture - Isolated Storage démonstration 99
Figure 23 - Structure de l'Isolated Storage 100
Figure 24 - Communiquer avec la page HTML 105
Figure 25 - Début de séquence : affichage de la page HTML et du plugin Silverlight 106
Figure 26 - Saisie d'un tester envoyé vers HTML 106
Figure 27 - Modification en retour de l'application Silverlight 107
Figure 28 - Création d'un fichier de ressources de traduction 114
Figure 29 - Affichage d'un texte en ressource 115
Figure 30 - Erreur d'accès aux ressources 115
Figure 31 - Localisation d'une ressource 118
Figure 32 - Application localisée en US 120
Figure 33 - Application localisée en FR 121
Figure 34 -Gestion du clic droit 122
Figure 35 - Chargement d'une image 125
Figure 36 - Voir le multitâche en action 128
Figure 37 - Les dangers du cross domaine 146
Figure 38 - Prendre des photos 154
Figure 39 - Les techniques remplacées par WCF 166
Figure 40 - Créer un Entity Model depuis une connexion SGBD 174
Figure 41 - Un modèle d'entités généré automatiquement 175
Figure 42 - Ajouter le service OData au projet Silverlight 181
Figure 43 - L'affichage des données OData par l'application Silverlight 185
Figure 44 - Un bel exemple (mais cassé !) 188

Figure 45 - Une jolie application *194*
Figure 46- Le dialogue utilisateur du full-screen *208*
Figure 47 - Page exemple pour l'impression *209*
Figure 48 - Exemple d'impression *213*
Figure 49 - Mode Content *230*
Figure 50 - L'application idéale *264*
Figure 51 - Dans la vraie vie... ..... *264*
Figure 52 - Une application est faite de "parties" *266*
Figure 53 - Le principe d'exportation *268*
Figure 54 - le principe d'importation *269*
Figure 55 - le principe de composition *271*
Figure 56 - Hello MEF ! *276*
Figure 57 - Hello MEF (clic) *276*
Figure 58 - Modules multiples *280*
Figure 59 - DynamicXAP - Le Shell au lancement *310*
Figure 60 - DynamicXAP - Chargement des extensions *311*
Figure 61 - DynamicXAP - Le rôle utilisateur *312*
Figure 62 - DynamicXAP - Rôle Administrateur *312*
Figure 63 - DynamicXAP - Rôle Anonyme *313*
Figure 64 - Widget1 - Visuel *338*
Figure 65 - Widget2 de Extensions.XAP *347*
Figure 66 - Widget3 de Extensions.XAP *348*
Figure 67 - Widget4 ..... *354*
Figure 68 - Widget5 ..... *354*
Figure 69 - Valider des données *359*
Figure 70 - Extension de balises *387*
Figure 71 - Effet d'une balise personnalisée *387*
Figure 72 - Lorem Ipsum *391*
Figure 73 - générateu d'encodeur *401*

Présentation du Tome 8 - Silverlight

Bien qu'issu des billets et articles écrits sur Dot.Blog au fil du temps, le contenu de ce PDF a entièrement été réactualisé et réorganisé lors de la création du présent livre PDF entre octobre et décembre 2013. Il s'agit d'une version inédite corrigée et à jour qui tient compte des évolutions les plus récentes autour de Silverlight, jusqu'à sa version 5.

Le Tome 8 complète les deux autres Tomes consacrés à XAML, le Tome 7 qui se concentre sur les bases de XAML et sur WPF, le Tome 6 plus axé sur WinRT et Modern UI. Bien que largement inspiré des billets de Dot.Blog le présent Tome est l'un de ceux qui a demandé le plus de travail avec le Tome précédent (XAML) à la fois par sa taille gigantesque et par la réorganisation des thèmes, la réécriture et la mise en conformité des informations pour en faire non pas un recueil de textes morts mais bien un *nouveau livre totalement d'actualité* ! Un énorme bonus par rapport au site Dot.Blog ! Plus d'un mois de travail a été consacré à la réactualisation du contenu. Corrections du texte mais aussi des graphiques, des pourcentages des parts de marché évoquées le cas échéant, contrôle des liens, et même ajouts plus ou moins longs, c'est une véritable *édition spéciale différente* des textes originaux toujours présents dans le Blog !

C'est donc bien plus qu'un travail de collection déjà long des billets qui vous est proposé ici, c'est *un vrai nouveau livre sur Silverlight à jour*. Avec ses plus de 400 pages A4 et ses près de 100.000 mots, le Tome 8 comme les autres Tomes de la série ALL DOT BLOG montre à la fois ma volonté claire de partager une information complète et riche autant que la somme de travail énorme que représente Dot.Blog puisqu'avec ce 8ème Tome c'est près de 2300 pages A4 (!!!) qui viennent d'être publiées en 8 livres gratuits et cela n'est qu'une partie des plus de 650 billets et articles publiés depuis 2008 !

Astuce : cliquez les titres pour aller lire sur Dot.Blog l'article original et ses commentaires, vous pourrez d'ailleurs vous amuser à comparer les textes et prendre la mesure des modifications ! Tous les autres liens Web de ce PDF sont fonctionnels, n'hésitez pas à les utiliser (certains sont cachés, noms de classe du Framework par exemple, mais ils sont fonctionnels, promenez votre souris…).

XAML pour Silverlight

XAML est un, mais chacune de ses variantes appelées « profil » possède ses petites particularités, parfois uniquement en raison du framework sous-jacent et de l'environnement pour lequel ce dernier a été spécialisé.

Silverlight est un plugin pour browser, cela impose beaucoup de restrictions sur la taille du framework embarqué mais aussi certaines spécificités liées au monde Web.

Selon les versions de XAML ce qui est dit pour Silverlight est généralement valable pour Windows Phone 7.x et même souvent Windows Phone 8. Il y a donc toujours à apprendre ou à transposer même si hélas Silverlight a été arrêté. Il sera maintenu jusqu'en 2021, ce qui lui laisse encore quelques beaux jours puisqu'il rend un service unique. Se passer de Silverlight c'est accepter de revenir au moyen-âge de HTML/CSS/JS. Un jour peut-être Silverlight renaîtra, une nouvelle Direction doit être nommée avant l'été 2014 chez Microsoft, nul ne peut prédire les axes que le nouveau CEO privilégiera… Ballmer, Sinofksy, tous sont remplaçables, Silverlight ne l'est pas.

Silverlight, Blend et Design, 11 vidéos et un manuel

Microsoft a mis en ligne une série de 11 vidéos et un guide pour présenter Silverlight et le processus de création d'application, notamment au travers de Blend et Expression Design. C'est vraiment bien fait et reste d'actualité même si on n'utilise pas Silverlight car la manipulation de Blend reste la même, XAML reste le même malgré quelques différences, même sous Windows Phone 8 ou WinRT.

La page d'accueil de la série de vidéo : http://msdn.microsoft.com/en-us/expression/ff843950

Orienté Designer cet ensemble est toutefois utilisable par tout le monde et peut intéresser tous les utilisateurs de XAML quel que soit le profil. Il s'agit d'une mini formation assez complète permettant de mieux comprendre Silverlight, XAML et les outils qui tournent autour ainsi que la façon dont le processus d'élaboration d'une application se déroule.

Les vidéos couvrent les domaines suivants :

- What is Silverlight? An Overview (this video)
- Understanding and Working with XAML Code in Blend
- Creating Vector-Based Artwork with Expression Design
- Applying Color and Effects to Projects using Expression Design
- Organizing Your Project using Layout Containers in Blend
- Editing the Appearance of Your Project Items using Blend
- Exploring the Objects and Timeline Task Pane in Blend
- Customizing Silverlight Video Players using Blend
- Optimizing Video for Silverlight Playback using Expression Encoder
- Adding Interactivity to Silverlight Projects using Blend
- Publishing Silverlight Projects to the Web using Blend

Ces videos sont complétées par un starter kit (document Word).

Un bon moyen pour se faire rapidement une idée de Silverlight, XAML et du processus de création d'une application en mode vectoriel. Le tout est en anglais, comme d'habitude, sorry.

44 Videos US et 11 Ateliers en français sur Silverlight!

Dans un billet de 2008 je vous parlais de l'extraordinaire travail publié par Mike Taulty, 44 vidéo autour de Silverlight (V 2 à l'époque) ! Cette série reste toujours

d'actualité pour se former, mais elle peut gêner ceux qui ne pratiquent pas assez bien l'anglais pour capter chaque leçon. Si j'en reparle c'est que les liens que j'avais donnés à l'époque ne sont plus bons et qu'il était nécessaire de les réactualiser (grâce à la vigilance d'un lecteur que je remercie au passage !). C'est chose faite dans le présent billet.

Mais j'en profite aussi pour vous parler d'une série d'ateliers que Microsoft avait mis en ligne. Le **coach Silverlight** ce sont des « ateliers » d'auto-formation bien fichus. Certes ils ont été écrits pour Silverlight 3, mais cette version était déjà très complète et les nouveautés de la 4 ou de la 5 ne changent pas beaucoup l'intérêt de cette série.

Si on compte bien, 44 + 11 ça donne **55 modules de formation gratuits** pour prendre en main Silverlight ou **même d'autres profils XAML** car après tout tous partagent un (gros) noyau commun... Les longues soirées vont être chargées et vous pouvez d'ores et déjà prévoir de laisser votre XBox à vos enfants, vous n'allez pas beaucoup y toucher dans les semaines à venir !

L'accès direct au sommaire des 44 casts US : http://misfitgeek.com/blog/44-silverlight-videos/

Le sommaire pour se mettre l'eau à la bouche :

- Silverlight - Hello World
- Silverlight - Anatomy of an Application
- Silverlight - The VS Environment
- Silverlight - Content Controls
- Silverlight - Built-In Controls
- Silverlight - Width, Height, Margins, Padding, Alignment
- Silverlight - Using a GridSplitter
- Silverlight - Grid Layout
- Silverlight - StackPanel Layout
- Silverlight - Canvas Layout
- Silverlight - Databinding UI to .NET Classes
- Silverlight - Simple Styles
- Silverlight - Custom Types in XAML
- Silverlight - Binding with Conversion
- Silverlight - List Based Data Binding
- Silverlight - Simple User Control
- Silverlight - Templating a Button
- Silverlight - Resources from XAP/DLL/Site Of Origin

- Silverlight - Animations & Storyboards
- Silverlight - Uploads with WebClient
- Silverlight - Downloads with WebClient
- Silverlight - Calling HTTPS Web Services
- Silverlight - Calling Web Services
- Silverlight - Making Cross Domain Requests
- Silverlight - Using HttpWebRequest
- Silverlight - File Dialogs and User Files
- Silverlight - Using Sockets
- Silverlight - Using Isolated Storage
- Silverlight - .NET Code Modifying HTML
- Silverlight - Using Isolated Storage Quotas
- Silverlight - Calling JavaScript from .NET
- Silverlight - Evaluating JavaScript from .NET Code
- Silverlight - Handling HTML Events in .NET Code
- Silverlight - Handling .NET Events in JavaScript
- Silverlight - Calling .NET from JavaScript
- Silverlight - Displaying a Custom Splash Screen
- Silverlight - Passing Parameters from your Web Page
- Silverlight - Loading Media at Runtime
- Silverlight - Dynamically Loading Assemblies/Code
- Silverlight - Reading/Writing XML
- Silverlight - Multiple Threads with BackgroundWorker
- Silverlight - Insert/Update/Delete with the DataGrid
- Silverlight - Getting Started with the DataGrid
- Silverlight - Embedding Custom Fonts

Le coach français : http://msdn.microsoft.com/fr-fr/silverlight/msdn.coachsilverlight.aspx

Et le sommaire pour vous donner encore plus envie d'y aller tout de suite :

- Présentation générale de Silverlight 3
- Quels outils utiliser ?
- Les bases d'une application Silverlight
- Atelier 1 : Concepts principaux : XAML, formes de base, gestion évènements
- Atelier 2 : Gestion de la vidéo avec Silverlight et Web Client
- Atelier 3 : Les styles, le templating et le DataBinding

- Atelier 4 : Binding entre éléments, transformations et projection 3D
- Atelier 5 : Animations classiques, Easy Animations et moteur physique
- Atelier 6 : WCF et Silverlight 3
- Atelier 7 : SandBoxing, Open/SaveFileDialog, Quota
- Atelier 8 : Pixel Shaders
- Atelier 9 : L'accélération matérielle avec le GPU
- Atelier 10 : Out of Browser et les autres nouveautés de Silverlight 3
- Atelier 11 : Introduction à .NET RIA Services

Silverlight/Blend : bien plus que des exemples…

Juste quelques mots pour vous parler d'une librairie de code qui se trouve sur CodePlex et qui n'est pas assez connue à mon avis.

Il s'agit de "Expression Blend Samples" qu'on trouve ici : http://expressionblend.codeplex.com/

Cette librairie, disponible en binaire avec installeur et en code source, contient de nombreux Behaviors et des effets visuels (Pixel shaders) tout à fait pratiques et utilisables dans vos projets Silverlight.

Ne vous laissez pas tromper par le mot "samples" (exemples) dans le titre, ce sont bien plus que de simples exemples, c'est du code très utile !

On trouve par exemple des Behaviors (comportements) pour contrôler le composant média (play, pause…) sans passer par du code VB ou C# donc. Mais ce n'est pas tout, bien entendu, il y a aussi de nombreux Behaviors orientés données, d'autres spécifiques au prototyping… Une vraie mine d'or.

Des Triggers (déclencheurs) sont aussi disponibles dont le `StateChangeTrigger` qui détecte le passage à un état donné ou tout changement d'état d'un contrôle ou d'autre plus exotiques comme le `MouseGestureTrigger` qui permet de dessiner une gesture et ainsi de piloter une application via des mouvements préétablis de la souris !

Enfin, la librairie contient tout un ensemble d'effets visuels issus de la librairie WPF http://www.codeplex.com/fx avec deux petites différences qui comptent : d'une part "Expression Blend Samples" est fourni avec un module d'installation ce qui évite d'avoir à compiler les shaders, et d'autre part l'effet "Wave", un des plus sympas,

fonctionne à merveille alors même qu'il refuse de se compiler dans sa librairie originale…

Une librairie à posséder donc…

Les Splash screens Silverlight

De quoi s'agit-il ?

La technique des « splash screens » est aujourd'hui bien connue, de nombreux logiciels en possèdent. En deux mots pour être sûr de ce dont nous parlons il s'agit d'un premier écran affiché rapidement lors du chargement d'une application et dont le but est de faire patienter l'utilisateur jusqu'à ce que l'application soit opérationnelle et utilisable.

Figure 1 - Le splash screen de l'application de démo[1] « codes postaux français »

La puissance des machines aidant, peu de logiciels ont réellement besoin aujourd'hui d'un splash screen. Certains Designers pensent même que la présence d'un Splash est une erreur de Design car une application, surtout sur le Web ou sur smartphones, doit être immédiatement utilisable (pour éviter que l'utilisateur impatient ne zappe). En revanche les applications qui téléchargent des vidéos au lancement, ou qui

[1] Voir en fin d'article l'adresse Web de cette application que vous pouvez tester en ligne.

ouvrent des connexions aux données par exemple en tirent souvent partie. Mais il est vrai qu'il peut s'avérer préférable de travailler en multitâche pour être réactif quitte à avoir un petit délai dans l'accès à certaines possibilités plutôt qu'afficher un Splash durant lequel l'utilisateur peut passer à autre chose…

Les Splash screen sont ainsi des artifices qu'il faut savoir manier avec précaution et discernement. Pour un logiciel ouvert à un large publique c'est déconseillé, pour un logiciel d'entreprise qui doit absolument charger des éléments ou ouvrir des connexions afin d'être ensuite fluide et agréable, cela peut se faire. Sur Internet où les débits sont variables dans de grandes proportions le Splash peut aussi servir l'utilisateur en lui permettant de se rendre compte de la mauvaise qualité de sa connexion.

On trouve aussi des Splash screen dont la seule utilité est esthétique, ou utilisés comme une « griffe », une signature de la part de son concepteur. Une façon d'exposer son logo, le nom de sa marque auprès des utilisateurs. Ce type de splash est souvent une redite de la boîte « à propos » et s'affiche de façon fugace. Certains logiciels permettant d'ailleurs de supprimer cet affichage et c'est une bonne idée…

Sous Silverlight on retrouve l'ensemble de cette palette d'utilisations mais l'une des raisons premières d'ajouter un splash screen reste le plus souvent liée au *temps de chargement de l'application*. En effet, une application Silverlight est avant tout une application Web, donc utilisant un réseau assez lent. En Intranet la question se pose moins si le réseau utilisé est de bonne qualité. De plus, c'est une technologie dont le principe est de télécharger l'application en locale avant de l'exécuter, ce qui prend du temps. Ces deux facteurs cumulés imposent même pour des applications de taille modeste d'afficher un splash screen afin d'informer l'utilisateur que la page Web qu'il a demandée est bien en train d'arriver (avec le danger qu'il zappe malgré tout si cela s'allonge…). Le splash indique ainsi le plus souvent une progression (barre, pourcentage…) pour que l'utilisateur puisse se faire une idée de l'attente et c'est vraiment indispensable pour ne pas qu'il passe à un autre site ! Si c'est rapide il attendra, si c'est trop lent ou s'il ne sait pas si « ça charge », sur Internet l'utilisateur zappe… Raison de plus pour l'informer !

C'est pourquoi toute application Silverlight est dotée, de base, d'un splash screen minimaliste sans avoir besoin de coder quoi que ce soit. Il s'agit d'une animation circulaire, avec, en son centre, le pourcentage d'avancement.

Pourquoi personnaliser le splash screen ?

Un certain nombre d'exemples déjà été évoqués laissent supposer la nécessité d'une personnalisation du splash. Mais indiquer à l'utilisateur que le logiciel qu'il a demandé est bien en train d'arriver est la fonction première d'un splash sous Silverlight. Le splash screen par défaut est efficace et discret mais il se peut qu'il ne cadre pas avec le visuel de l'application.

Nous sommes bien ici dans un cadre particulier où le choix n'existe finalement pas vraiment : de toute façon il y aura un splash screen… Alors autant qu'il soit à la hauteur et qu'il attise la curiosité de l'utilisateur pour lui donner envie de rester.

Enfin, il faut noter que tout développeur aime bien laisser son empreinte et prouver qu'il maîtrise la technologie utilisée, proposer un splash screen est un moyen simple de le faire tout en faisant patienter l'utilisateur.

Le splash screen, un objet pas comme les autres

La nature spécifique du splash screen Silverlight

Dans une application classique, qu'elle soit de type Windows Forms ou même WPF, dès son lancement le développeur peut se reposer sur l'ensemble des outils de la plateforme utilisée. Dès lors, le splash screen n'est rien d'autre qu'un objet visuel affiché temporairement. Il peut même s'agir d'une fiche (type `Form` Windows Forms ou `Page` WPF) ou de tout autre objet visuel. Le moment de la création du splash, celui de son affichage, la façon de le libérer une fois l'application prête à travailler, etc, tout cela est relativement bien codifié aujourd'hui.

Sous Silverlight les choses sont un peu différentes.

Tout d'abord parce que le splash screen ne peut pas appartenir à l'application elle-même. La raison est simple : si tel était le cas, il faudrait attendre le chargement final de l'application pour pouvoir l'activer et cela lui ferait perdre beaucoup de son intérêt…

Sous Silverlight il est possible de charger des modules dynamiquement. Cela est même conseillé pour les applications un peu lourdes. Dans un tel cas on peut concevoir un premier paquet XAP très léger qui affichera le splash screen pendant qu'il lancera le chargement des autres modules. C'est une technique à ne pas négliger, mais ce n'est pas celle dont je vais vous parler ici.

Dans le cas le plus classique, une application Silverlight se présente sous la forme d'un fichier XAP contenant l'essentiel de l'application si celle-ci est de taille raisonnable. De fait, le splash screen doit être externalisé pour pouvoir s'afficher durant le chargement en local sur le poste client du fichier XAP et des éventuelles ressources annexes. Une telle externalisation du splash est prévue par le plugin.

Externaliser… oui mais où ?

Une application Silverlight, même si ce n'est pas une obligation, se construit généralement sous la forme de deux projets : le projet Silverlight lui-même et un projet Web ASP.NET plus classique contenant une page Html ou Aspx hébergeant le plugin paramétré pour lancer l'application, le fichier XAP étant placé dans un sous-répertoire. Visual Studio propose par défaut de créer cette organisation, je vous conseille de la conserver.

Donc lorsqu'on dit qu'il faut externaliser le splash screen, on suppose par là qu'il existe autre chose que le projet Silverlight pour l'abriter. Et en effet, un tel endroit existe, c'est justement le projet Web créé par Visual Studio au sein de la solution.

Une nature doublement spéciale

Nous avons vu que sous Silverlight le splash screen est un peu différent des objets visuels utilisés pour cette fonction dans les applications de type Desktop. Nous avons vu qu'il fallait aussi l'externaliser.

Ce qui reste à dire sur cette nature particulière est que le splash screen va utiliser les capacités de Silverlight 1.0 qui fonctionnait avec du JavaScript. Cela implique que le splash ne peut pas utiliser toute la palette technique mise à disposition depuis Silverlight 2 car au moment de son exécution toutes les ressources ne sont pas encore chargées (thèmes du toolkit, données, etc). Et cela implique aussi que sa logique, s'il en possède une, doit être codée en JavaScript et non en C#.

La réalité est un peu plus nuancée : le splash pourra utiliser tout ce qui est de base dans le runtime de Silverlight. Mais cela posera éventuellement des problèmes sous Blend. Nous verrons cet aspect plus loin et comment le contourner.

Le projet de démonstration

Création du projet

Nous allons débuter la démonstration en créant un projet Silverlight sous Visual Studio. La séquence est basique : *Fichier*, *Nouveau*, *Projet*. Le dialogue s'affiche, choisir *Silverlight*, saisir ou sélectionner un répertoire (figure 2).

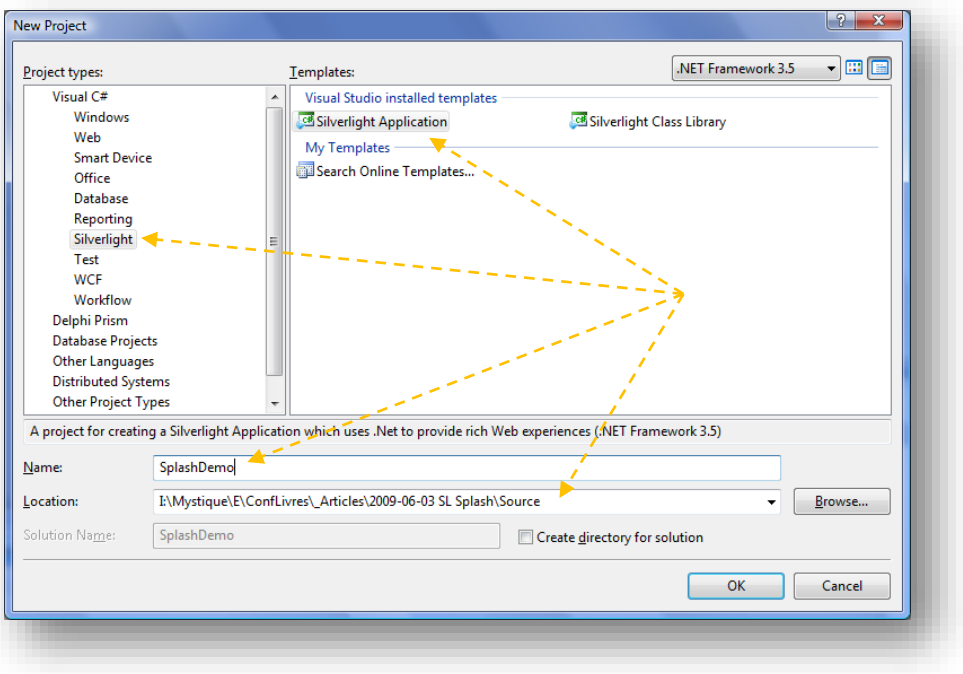


Figure 2 - Création du projet Silverlight

Une fois ce dialogue validé, VS nous en propose un second permettant de choisir le type de solution. C'est ici qu'on peut demander une solution Web complète constituée de deux sous projets (Silverlight et Asp.Net) ou bien opter pour un projet Silverlight simple.

> *Selon la version de Visual Studio les dialogues peuvent être légèrement différents, le lecteur adaptera en fonction de son setup.*

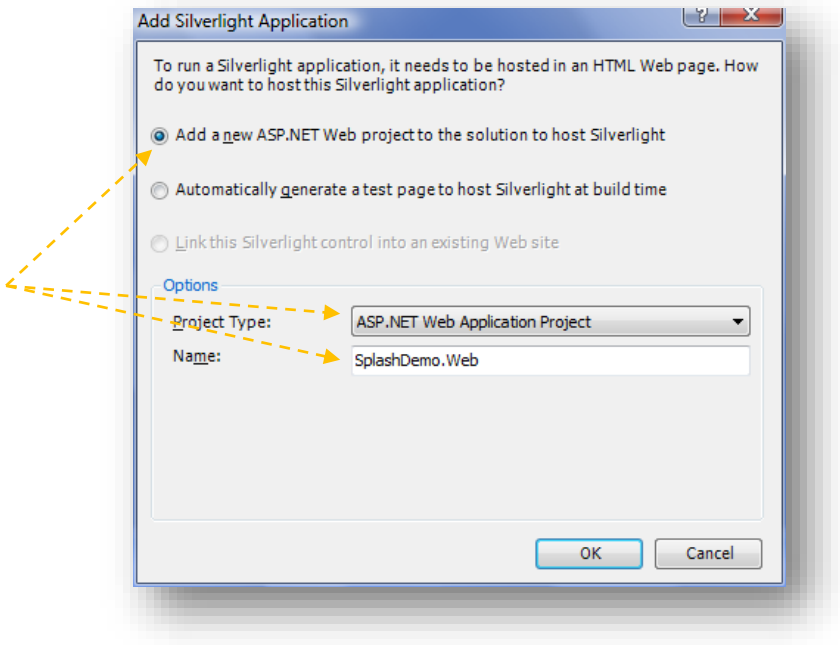


Figure 3 - Le choix du type de solution

Le contenu de la solution

Arrivé à ce stade la solution contient deux projets, l'un Silverlight, l'autre ASP.NET. Vous devez avoir une arborescence qui correspond à la figure suivante :

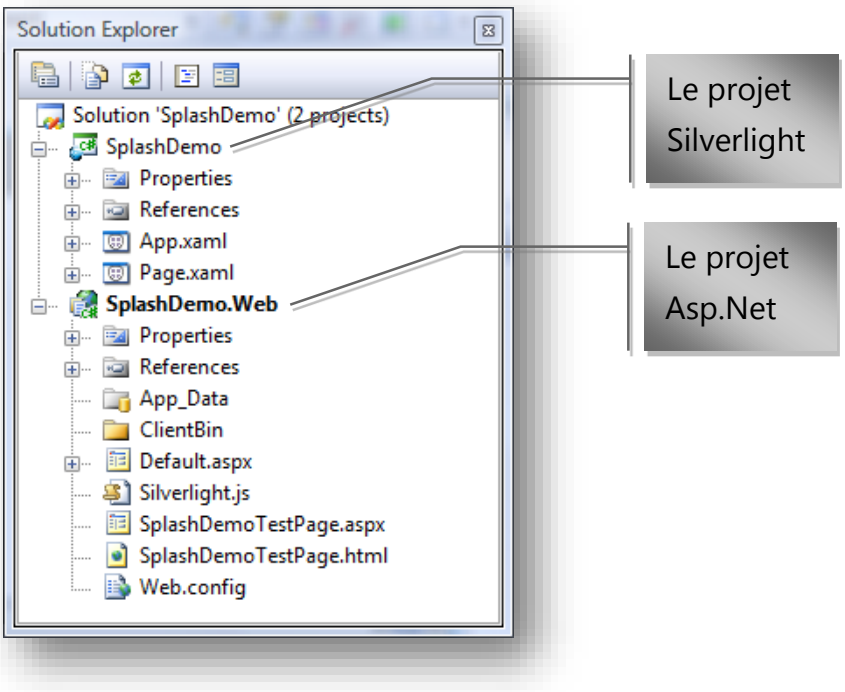


Figure 4 - Arborescence de la solution

La page de démonstration

Pour s'apercevoir que l'écran principal de l'application est affiché il faut bien y placer un peu de visuel, alors lâchez-vous, l'application Silverlight en elle-même ne nous intéresse pas ici, seule la présence d'un visuel compte !

Vous pouvez utiliser Blend, ce qui est plus agréable, ou le designer XAML de Visual Studio qui a considérablement été amélioré au fil du temps.

Si vous ne vous sentez pas l'âme créatrice, voici quelques lignes de XAML qui feront parfaitement l'affaire pour remplacer la grille vide par défaut :

```
<Grid x:Name="LayoutRoot" Background="Purple">
        <Canvas>
            <Ellipse Height="200" Width="200" Fill="Orange" />
            <Button Canvas.Left="180" Canvas.Top="150"
                    Content="Démo !"></Button>
        </Canvas>
    </Grid>
```

Lorsqu'on exécute l'application on obtient le magnifique affichage de la figure suivante :

Figure 5 -Ecran de test

Centrer l'application

En fait je vous mens un peu, vous n'obtiendrez pas l'affichage ci-dessus même si vous avez suivi à la lettre ce que j'ai dit. Ce que vous verrez sera très proche, mais le

rectangle de l'application sera affiché coincé en haut à gauche et non pas centré harmonieusement dans l'espace du browser.

Pour cela il faut appliquer une petite astuce. Il existe plusieurs façon de faire, mais j'aime bien celle que je vais vous présenter car elle est très simple à mettre en œuvre et fonctionne aussi bien dans la page de test HTLM créée par Silverlight que dans la page ASPX équivalente.

A ce propos, vous avez du remarquer que le projet de test Web comporte deux pages qui portent le nom de l'application avec le suffixe « `TestPage` », l'une avec l'extension HTML, l'autre en ASPX (dans le projet exemple les pages s'appellent « `SplashDemoTestPage.aspx` / `html` »).

Par défaut c'est la page ASP.NET que VS appelle lorsqu'on lance l'application. Comme il faut bien faire un choix, le présent article va utiliser la page HTML. Pour ce faire ouvrez les propriétés du projet Web et cliquez sur l'onglet « Web ». Là vous verrez que la page de lancement est celle en ASP.NET (extension ASPX). Changez l'extension en « `.html` » et refermez les propriétés. De cette façon c'est bien la page HTML sur laquelle nous allons travailler qui sera affichée et non sa sœur en ASP.NET. Sans cette petite mise au point vous pourriez rester longtemps à chercher pourquoi aucune modification ne semble s'afficher…

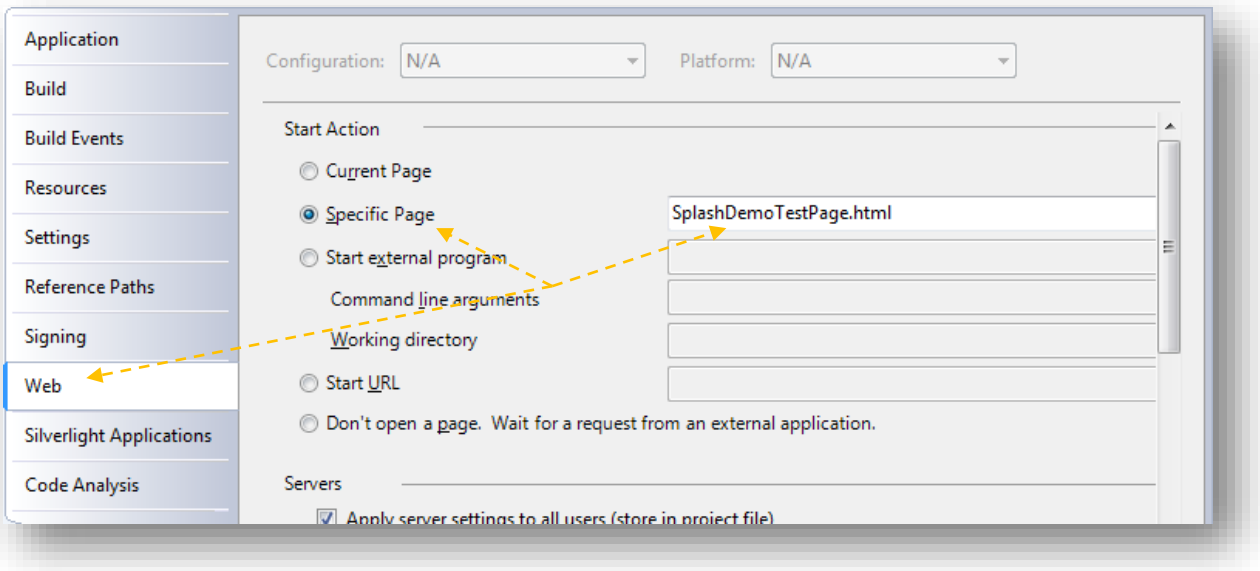


Figure 6 - Paramétrage de la page de test

Bref, il faut centrer l'application, c'est plus joli. La méthode que je préfère consiste à créer un style pour une division qui englobera le plugin Silverlight :

```
DIV.sl
        {
            position: absolute;
            left: 50%;
            top: 50%;
            width: 400px;
            height: 300px;
            margin-left: -200px; /* half of width */
            margin-top: -150px; /* half of height */
        }
```

Le code de ce style peut être ajouté à la section style existante dans le fichier HTML ou bien à une feuille CSS dans une application réelle.

L'astuce consiste à positionner la division au centre de l'espace disponible étant données quatre informations : la hauteur et la largeur du plugin Silverlight (la taille de votre `UserControl` principal) ainsi que la moitié de ces deux valeurs transformées en négatif. Ainsi une largeur de 400 pixels donnera une marge gauche à « -200 ». Si vous modifiez la taille de votre application Silverlight il faut juste penser à rectifier les valeurs que vous aurez saisies dans le style.

Une fois le style créé, il suffit d'englober la déclaration du plugin Silverlight au sein de la page HTML par une balise `DIV` de la classe que nous venons de créer :

```
<div class="sl">
        <div id="silverlightControlHost">
            <object data="data:application/x-silverlight-2," … code
        </div>
</div>
```

Vous remarquerez que j'ai laissé la division « `silverlightControlHost` » déjà posée par VS lors de la création de la page HTML, cela pour vous faire voir où placer la nouvelle division et avoir un repère dans le code existant. Dans la pratique il sera bien entendu préférable de fusionner ces deux balises, tant au niveau de la déclaration du style lui-même que des balises HTML.

Dans tous les cas, arrivé à ce stade nous disposons d'une application Silverlight qui affiche quelque chose en centrant tout cela proprement dans l'espace du navigateur. C'est le point de départ pour commencer à réfléchir à notre splash screen qui, ne le perdons pas de vue, est le vrai sujet de ce tutoriel !

Malgré tout je voudrais vous donner une dernière astuce sur ce sujet. Ici je vous ai proposé de centrer une application Silverlight en partant du principe que sa taille a été fixée à la conception. Cela est souvent le cas mais pas toujours. Il vaut mieux d'ailleurs développer des apps qui savent tirer parti de tout l'espace disponible.

Mais quand bien même la taille de l'application serait fixe, il existe une autre façon de procéder qui donne, à mon goût, un résultat de meilleure qualité notamment parce que tout est fait en XAML et peut être contrôlé par l'application.

L'astuce consiste à placer l'application Silverlight dans une `Grid`. Cette dernière sera en mode `Stretch` sur les 2 dimensions, et le `UserControl` de l'application sera simplement centré dans la `Grid` de façon tout à fait classique. De fait l'application en elle-même occupe tout l'espace (la `Grid`) mais le visuel de l'application (le `UserControl`) est toujours centré, le tout sans bricolage HTML ou CSS.

On peut aussi tirer parti de cette astuce pour choisir la couleur de fond de la `Grid` plutôt, là encore, que de bricoler la couleur de fond du Browser.

A vous de choisir la méthode qui vous plait !

Le Splash screen

Tester un splash screen

Vous allez penser qu'il est étonnant de partir de la fin et d'aborder dès maintenant la façon de tester un splash screen alors que nous n'avons pas encore commencé sa conception…

En fait c'est une préoccupation parallèle à la conception car durant cette dernière il faut bien pouvoir afficher le splash pour le mettre au point, tant comportementalement que visuellement.

Un splash n'est visible que durant le temps de chargement de l'application Silverlight, il faut donc que celui-ci soit assez long pour voir si tout se passe bien, et vu la légèreté de notre application de démonstration, le chargement sera immédiat surtout en local sur la machine de développement ! Pire, dès que nous aurons fait le test une fois l'application sera placée dans le cache du browser, ce qui fait que même déployée sur un serveur distant son chargement sera presque immédiat aux lancements suivants. Pas facile dans ces conditions de tester le splash…

Il n'y a pas beaucoup de solution pour ralentir le chargement de notre application, en tout cas « out of the box ». La méthode couramment utilisée consiste à déplacer dans

le projet Silverlight des vidéos bien grosses placées en ressource de façon à faire grossir énormément la taille du fichier XAP. C'est une technique simple mais qui n'évite pas totalement le problème du cache du browser.

Il y a une autre solution qui permet de tester un site Web dans des conditions de vitesse de connexion parfaitement connue. Il s'agit d'une petite application Java qui joue le rôle de proxy et dans laquelle on peut paramétrer la vitesse de connexion qui sera simulée. Dans notre cas on peut par exemple choisir une liaison modem 56k. Il suffit alors d'un bon gros fichier ajouté au XAP selon l'astuce précédente et nous obtenons un tel ralentissement que le splash est bien visible !

Ce petit utilitaire bien sympathique et gratuit s'appelle SLOPPY et il peut être téléchargé ici : http://www.dallaway.com/sloppy/

Etant en Java il réclame qu'une JVM soit installée mais cela est très souvent le cas sur une machine de développement.

Une fois téléchargé, Sloppy se présente sous la forme d'un petit exécutable Java « `sloppy.jnlp` », placez-le où vous voulez. Un double-clic le lance et affiche le premier écran qui vous permet de sélectionner la vitesse de la connexion ainsi que la page à atteindre.

Sloppy… pensez-y, en dehors de voir les splash screen, c'est très intéressant de voir comment le site Web que vous avez conçu s'affiche chez un utilisateur ayant une connexion à faible débit !

Conception graphique

Maintenant que nous savons comment ralentir le chargement de l'application Silverlight, nous savons que nous pourrons voir notre splash en cours de fonctionnement. Nous pouvons donc passer à la conception de ce dernier.

Contraintes techniques

Un splash Silverlight est forcément situé en dehors du XAP de l'application, sinon il serait soumis à la même contrainte, à savoir qu'il faudrait attendre le chargement de ce dernier pour que l'écran puisse être affiché. Ce n'est bien entendu pas ce que nous souhaitons.

Dès lors le splash doit se présenter sous la forme d'un fichier XAML autonome placé dans le projet Web et non dans le projet Silverlight.

Cela implique quelques contraintes techniques. La première est que ce fichier XAML doit contenir un code qui se contente des outils de base du plugin Silverlight. Il ne peut en aucun cas se reposer sur le Toolkit, sur les thèmes ou tout autre composant ou classe qui réclamerait une DLL particulière.

Ensuite ce fichier XAML va être considéré comme étant de type Silverlight 1.0, c'est-à-dire que son comportement ne pourra pas être décrit en C# mais en JavaScript.

Tout cela force à l'économie, ce qui est une bonne chose pour un splash qui ne doit pas mettre plus de temps que l'application elle-même à se charger !

Toutefois ne dramatisons pas non plus. XAML, de base, c'est déjà très riche. Et côté comportement le splash va tout juste afficher un pourcentage ou une barre de progression (voire les deux au maximum). Une ligne de JavaScript peut suffire. Maintenant si vous voulez programmer un effet de neige qui tombe ou des effets à couper le souffle, c'est possible, mais ça sera un peu plus long (et il faut aimer JScript, que je n'apprécie guère, mais chacun ses goûts !).

Quel outil de conception ?

Concevoir un objet visuel réclame d'utiliser les bons outils. Et pour créer un splash screen dont l'essentiel est d'être visuel, cela ne peut se faire en tapant du XAML à la main. C'est possible, bien entendu, mais avant d'arriver à un résultat intéressant cela peut prendre du temps !

Je vous conseille donc fortement de créer votre splash avec Blend, de toute façon indispensable pour créer des applications WPF ou Silverlight.

Qui dit visuel dit graphique. Qui dit graphique dit outil *ad hoc*, là encore. XAML est vectoriel, il semble intéressant d'utiliser cet avantage plutôt que d'utiliser une image JPG qui prendra du temps à télécharger…

Blend offre des outils de dessin vectoriel, cela peut être suffisant. Sinon je vous conseille d'utiliser Expression Design (aujourd'hui disponible en téléchargement gratuit puisque la suite Expression a été arrêtée). Ce dernier est une sorte d'Illustrator en plus simple (mais pas forcément moins attractif) qui sait transformer un dessin en XAML dans une opération de copier/coller (par exemple vers Blend).

D'autres possibilités sont envisageables, on peut aussi travailler directement sous Illustrator et faire un copier/coller vers Blend pour ensuite obtenir le code XAML, ou utiliser un addon pour Illustrator qui sait produire directement du XAML. On

n'oubliera pas non plus l'excellent InkScape, équivalent d'Illustrator en Open Source qui sait lui aussi exporter en XAML.

Bref, il faut des outils de dessin pour faire des dessins, c'est une évidence. Mais il faut aussi quelqu'un pour faire les dits dessins ! Si vous n'êtes pas doué, vous pouvez toujours chercher des illustrations ou des cliparts Illustrator et les passer dans Design pour en tirer du code XAML. Sinon, préférez les services d'un infographiste…

Le dessin

Sous Expression Design on peut rapidement créer une petite scène ou un objet qui servira de splash. Plus le dessin est soigné, plus il est en rapport avec l'application, mieux ce sera ! C'est pourquoi l'assistance d'un professionnel est forcément un plus.

Pour ce tutoriel imaginons une petite scène tranquille (les images relaxantes mettent l'utilisateur en situation de détente), évitez les dessins agressifs qui crispent et rendre l'attente plus longue ou les animations trop complexes. Sauf exception (ou erreur de conception) le splash n'est visible que peu de temps, on doit tout de suite voir où se trouve la barre de progression ou le pourcentage.

La figure qui suit montre la scène en cours de conception sous Expression Design.

Bien entendu dans cet article je n'ai pas pris le temps de vous faire un dessin ultra sophistiqué ni n'ai sollicité mon infographiste, c'est donc une illustration dans un style très naïf ! L'important étant le principe.

Nous disposons maintenant d'une scénette, d'un objet ou tout ce que vous voudrez, et dans cette scène nous allons placer deux choses importantes pour un splash : une zone de texte pour indiquer le pourcentage d'avancement et une barre de progression pour symboliser graphiquement ce dernier. Pour cette démonstration je me limiterai au pourcentage, amusez-vous à ajouter la barre de progression (par exemple faire passer le nuage de gauche à droite, c'est une idée simple).

Ici j'ai choisi de placer uniquement un texte, et parce que je suis un peu contrariant, il n'indiquera pas le pourcentage d'avancement mais le pourcentage restant. Je trouve ça plus optimiste de voir la quantité diminuer qu'augmenter au fil du temps.

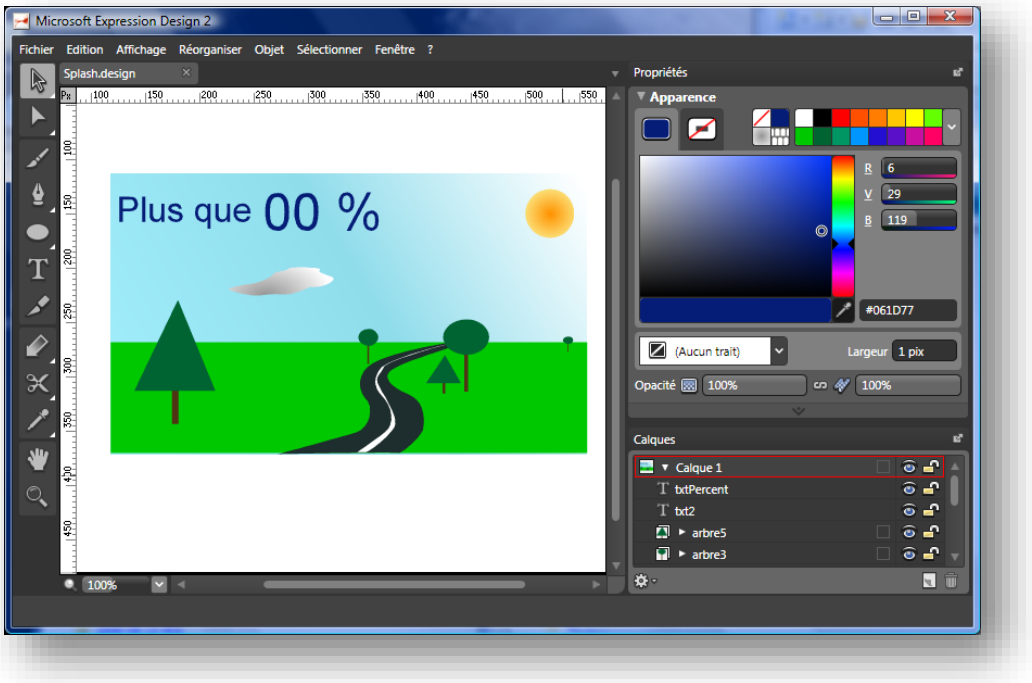


Figure 7 - La conception du splash

Un soleil, une route qui se perd à l'infini dans l'immensité verte, quelques arbres et un joli petit nuage qui habille le ciel, n'est-ce pas là un décor qui inspire la tranquillité et la *zénitude* propre à une attente sereine ? Il ne manque que le chant des cigales, mais un MP3 gonflerait inutilement notre splash… Et les sites Web qui utilisent du son cassent en général les pieds de l'utilisateur qui peut être en train d'écouter de la musique. Aussi évitez autant que possible de brouiller l'écoute de l'utilisateur[2] !

Conception technique

Ajouter le splash au projet

Il nous faut maintenant créer un fichier XAML pour accueillir notre splash. Nous allons bien sûr le créer dans le projet Web et non dans le projet Silverlight. Le plus simple est de faire un clic droit sur le projet Web, *Ajouter*, *Nouvel item*. Dans la boîte de dialogue choisir la catégorie Silverlight puis « *Silverlight JScript page* ». Saisir un nom pour le fichier avant de valider, par exemple « `splash.xaml` ».

Nous disposons maintenant de l'écrin, reste à y placer le joyau…

[2] Comme quoi on peut trouver des contextes où cette contrepèterie hyper classique est tout à fait à propos !

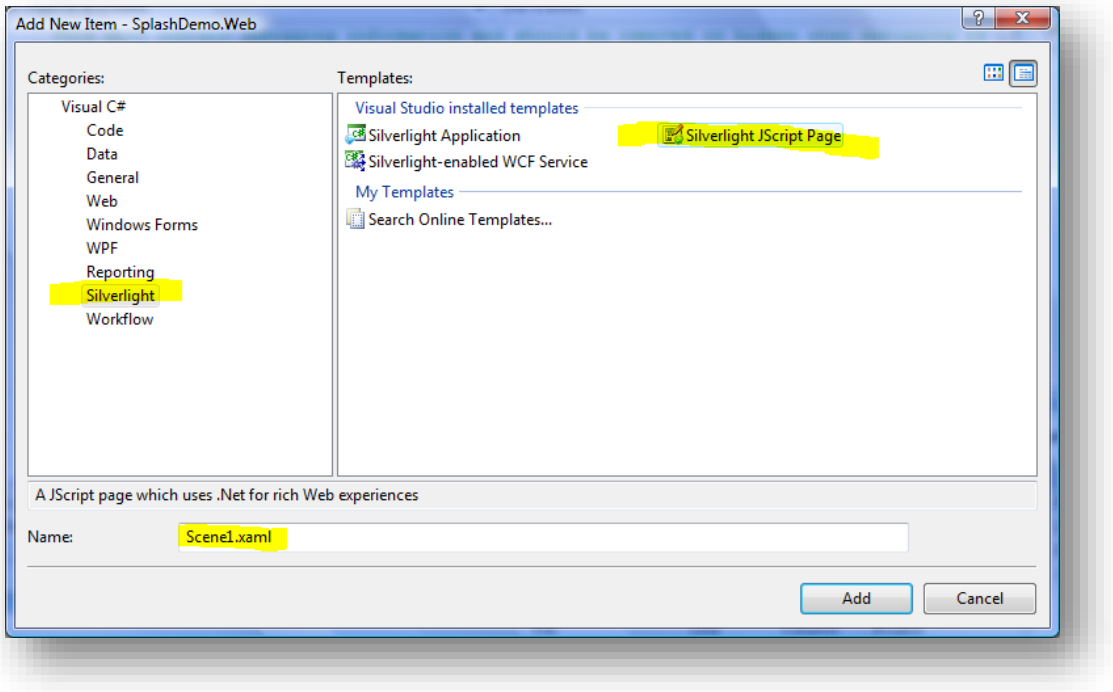


Figure 8 - Création de la page XAML du splash

> *On notera que selon les versions de Visual Studio l'option de création d'une page Silverlight JScript peut ne pas être présente. Dans ce cas il suffit de créer un fichier vide avec l'extension .xaml pour le dessin et un fichier vide de même nom avec l'extension .js pour le code JScript.*

Exportation du dessin

Pour utiliser le dessin que nous avons créé sous Expression Design nous devons l'exporter en XAML. Pour ce faire il faut s'assurer que la fonction d'exportation XAML du presse-papiers est bien configurée (figure suivante). Attention à certaines options comme la transformation des textes en tracés, cela peut être avantageux ou calamiteux selon ce qu'on veut faire des textes sous Blend ou VS ensuite… Si on a placé des textes fixes en utilisant des fontes ou des déformations on a tout intérêt à faire une transformation en tracé, mais si on désire agir sur ces textes, comme le pourcentage dans notre cas, il faut absolument que la zone texte reste de ce type et ne soit pas transformée en un dessin vectoriel, nous ne pourrions plus changer ou mettre à jour le texte.

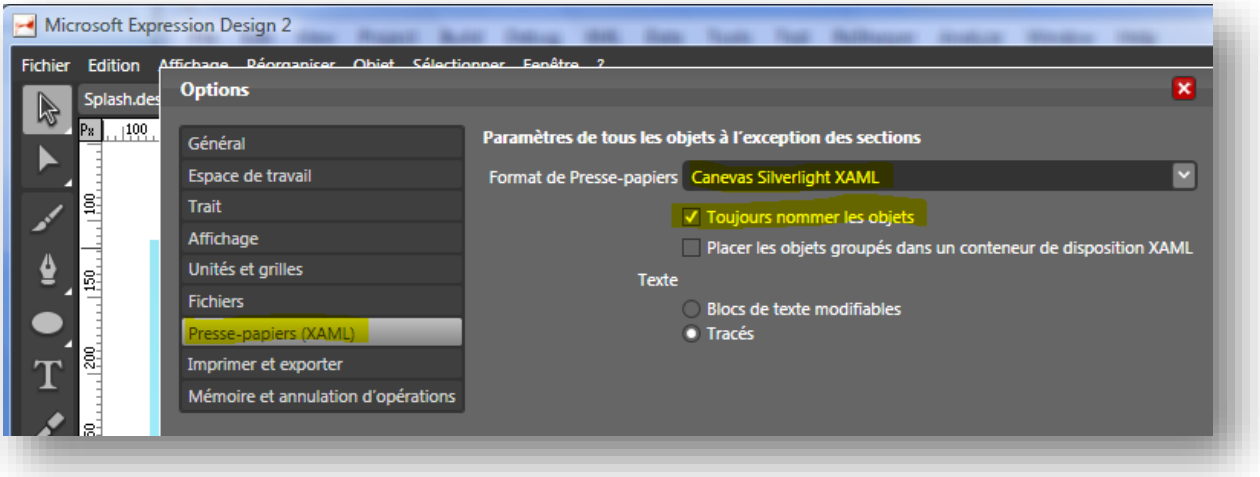


Figure 9 - Configuration d'Expression Design

Il ne reste plus qu'à sélectionner les parties constituant le dessin à l'aide du lasso et de taper `Ctrl-Maj-C` pour exporter la sélection en XAML.

> *Attention : l'exportation XAML depuis Design prend en compte le placement des objets dans le document graphique. Si vous avez créé votre dessin en plein milieu de ce dernier, toutes les coordonnées exportées le seront par rapport au point zéro du document et vous risquez d'avoir des surprises… Je conseille donc soit de bien fixer la taille du document au départ et d'utiliser 100% de cet espace pour créer votre splash, soit, avant d'exporter, de penser à prendre votre dessin et à la placer en haut à gauche (en 0,0) afin que les coordonnées en XAML soient relatives à cette position « neutre ».*

Il faut maintenant revenir à Visual Studio et remplacer le contenu du fichier « `Splash.xaml` » par celui qui est dans le presse-papiers. Il sera utile de faire quelques adaptations comme renommer l'objet `Canvas` que Design a appelé « `Document` » en quelque chose de plus parlant, comme « `Splash` » par exemple.

Vous noterez aussi que le `Canvas` a les dimensions du document Design et non celles de votre dessin… Si vous avez opté pour un dessin remplissant 100% du document Design, cela ne pose aucun souci, dans le cas contraire il faut modifier le code XAML pour donner les véritables dimensions de votre dessin :

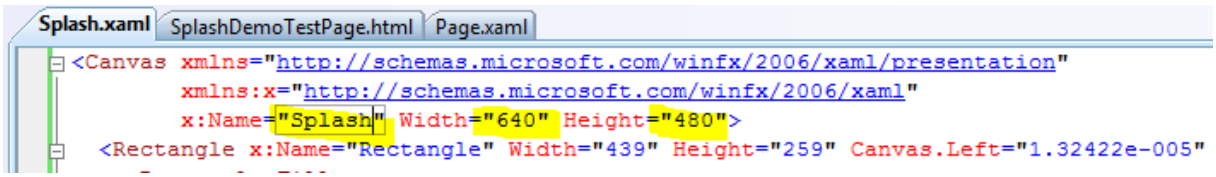


Figure 10 - Adapter le code XAML exporté depuis Design

Pour se repérer on peut utiliser les dimensions du plus gros objet, par exemple ici nous voyons (figure ci-dessus) qu'il existe un rectangle dont la largeur est de 439 pixels sur 259, s'il s'agit d'une forme qui englobe le splash alors nous utiliserons ces valeurs pour corriger la hauteur et la largeur du `Canvas` root (`640x480` dans le code montré figure ci-dessus).

Enfin, vous noterez que l'espace de nom copié depuis Design génère des messages d'avertissement dans Visual Studio indiquant que les objets ne peuvent pas être trouvés. Il suffit de changer le premier « `xmlns` » qu'on voit en haut de la figure précédente par la définition suivante :

```
xmlns="http://schemas.microsoft.com/client/2007"
```

Corrections et améliorations sous Expression Blend

Visual Studio ne sait pas travailler en mode design sur un fichier XAML hors d'un projet Silverlight (ou autre profil XAML), et s'agissant d'un fichier XAML isolé, il ne sait pas non plus l'afficher. Autant dire qu'il est temps de passer sous Expression Blend !

Si vous possédez Visual Studio 2010 ou plus récent vous pourrez en partie vous passer de Blend car cette version intègre un designer Silverlight. Toutefois rien ne remplacera Blend pour travailler le visuel XAML, rien, sauf la prochaine version de Blend bien entendu ☺

Parmi les petites choses à vérifier se trouve le nom de l'objet `TextBlock` qui va être mis à jour. Sous Blend ou directement dans le code XAML vérifiez que l'objet possède bien un nom, donnez-lui en un si nécessaire. Dans le projet exemple la zone qui sera modifiée s'appelle « `txtPercent` ».

Paramétrage du Plugin Silverlight

Après tous ces préparatifs nous arrivons maintenant au moment crucial où nous allons connecter tout cela pour faire marcher le splash !

Le plugin Silverlight possède de nombreux paramètres, seuls quelques-uns sont utilisés par défaut. Il en existe notamment qui permettent d'indiquer le nom du fichier qui sert de splash screen. Nous allons l'utiliser maintenant.

Tout d'abord, dans la page HTML repérez la balise `object` qui créée le plugin. Celle-ci doit se présenter comme suit :

```
<div id="silverlightControlHost">
    <object data="data:application/x-silverlight-2,"
            type="application/x-silverlight-2"
            width="100%" height="100%">
       <param name="source" value="ClientBin/SplashDemo.xap" />
       <param name="onerror" value="onSilverlightError" />
       <param name="background" value="white" />
       <param name="minRuntimeVersion" value="2.0.31005.0" />
       <param name="autoUpgrade" value="true" />
       <a href="http://go.microsoft.com/fwlink/?LinkID=124807"
          style="text-decoration: none;">
                    <img
                       src="http://go.microsoft.com/fwlink/?LinkId=108181"
                       alt="Get Microsoft Silverlight"
                       style="border-style: none" />
                </a>
    </object>
```

En fin de section des paramètres nous allons ajouter les lignes suivantes :

```
<param name="splashscreensource" value="Splash.xaml"/>
<param name="onSourceDownloadProgressChanged"
       value="onSourceDownloadProgressChanged" />
```

La source XAML

La ligne suivante fixe le nom du fichier splash :

```
<param name="splashscreensource" value="Splash.xaml"/>
```

L'événement de progression

La ligne suivante indique le nom de la fonction JScript appelée pendant la progression :

```
<param name="onSourceDownloadProgressChanged"
       value="onSourceDownloadProgressChanged" />
```

Le code JScript

Maintenant que le plugin est configuré, il ne reste plus qu'à coder la fonction JScript.

Dans le projet Web, vous remarquerez qu'avec le fichier `Splash.xaml`, Visual Studio a créé le fichier `Splash.js` (si ce n'est pas le cas créez-le). C'est ce dernier qui nous intéresse maintenant. Ouvrez-le sous VS puis remplacer le contenu par le suivant :

```
function onSourceDownloadProgressChanged(sender, eventArgs) {
    sender.findName("txtPercent").Text =
        100-Math.round((eventArgs.progress * 100))  + " %";
}
```

Ce code très simple va localiser le composant `txtPercent` et lui affecter le pourcentage de chargement restant (donc de 100 % à 0 %). La progression est retournée par l'argument `eventArgs`.

Toutefois, pour que ce code soit actif encore faut-il déclarer notre source JScript dans la page HTML. Pour ce faire nous ajoutons dans la section `HEAD` de la page :

```
<script type="text/javascript" src="Splash.js"></script>
```

Où « `Splash.js` » est le nom du fichier script accompagnant le fichier XAML de même nom (mais avec l'extension `.xaml`, bien entendu).

Exécution

Il est temps d'exécuter l'application pour voir notre splash à l'œuvre…

Pour rappel nous avons pensé à ajouter quelques gros fichiers dans le projet Silverlight, des objets déjà compressés comme des mp3 ou des vidéos sont tout à fait indiqués car Silverlight créé des fichiers XAP qui sont en réalité des fichiers « zippés ». Pour grossir efficacement le XAP il est donc préférable d'utiliser des fichiers difficilement compressibles (donc déjà compressés par un mécanisme ou un autre, comme le codage mp3 ou un codec vidéo).

Dans le projet j'ai placé ces fichiers dans un sous-répertoire que j'ai appelé « `slowdown` ». Une fois le splash débogué il suffira de détruire ce dernier et son contenu.

Dernière remarque, les fichiers ajoutés doivent l'être au XAP et non laissés en ressources externes, il faut donc pour chacun (une sélection globale marche aussi dans le cas de plusieurs fichiers) indiquer dans le « `Build Action` » l'action « `Content` » (contenu). Voir figure suivante.

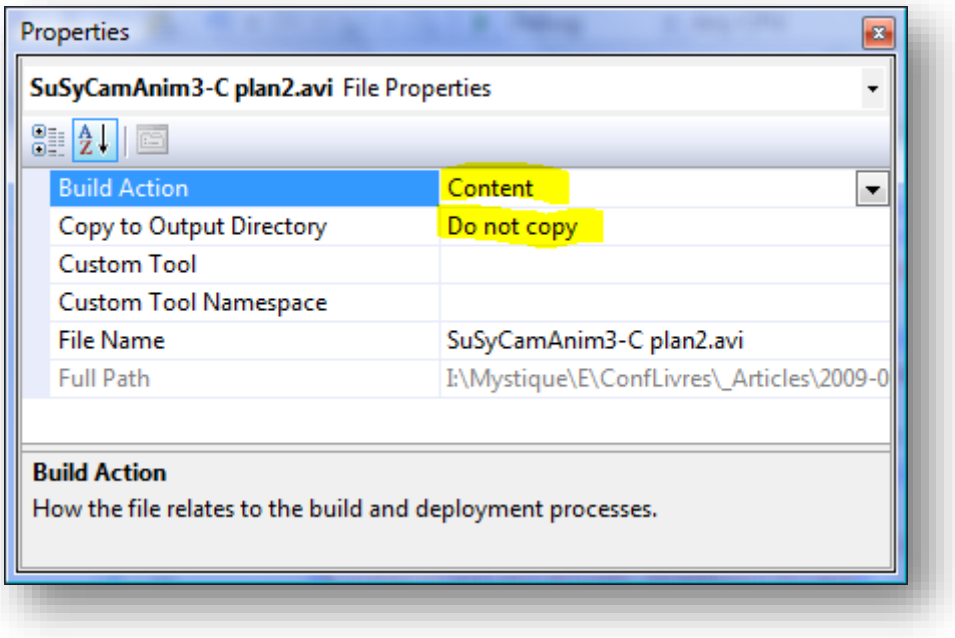


Figure 11 - Les fichiers "ralentisseurs"

Ralentissement

La présence de gros fichiers peut ne pas être suffisante pour ralentir assez le chargement de l'application, surtout en local. Dans ce cas utilisez l'utilitaire Sloppy dont j'ai parlé plus haut.

Pour ce faire lancer votre application normalement depuis VS. IE apparait et affiche cette dernière. Copier l'adresse qui est indiquée dans la barre d'adresse et collez-là dans le champ URL de Sloppy puis cliquez sur le bouton d'exécution de ce dernier. Un nouveau browser sera affiché, bien plus lent que l'autre pour peu que vous sélectionniez la simulation d'un modem 256 k ou moins si vous le désirez.

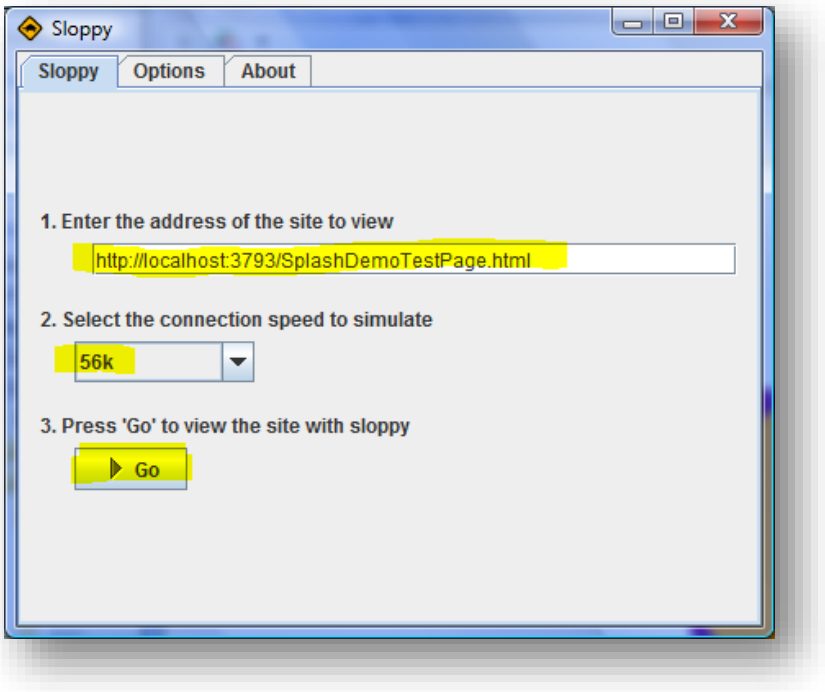


Figure 12 - Ralentir le chargement avec Sloppy

Astuce : Lorsque vous aurez effectué l'opération une fois il ne sera même plus nécessaire d'exécuter l'application via Visual Studio. Il suffira de faire une reconstruction de la solution puis de cliquer directement sur le bouton « Go » de Sloppy…

Silence on tourne !

Si tout se passe bien lors que vous exécutez l'application (suffisamment ralentie par les astuces que je vous ai indiquées), vous devez voir l'affichage suivant :

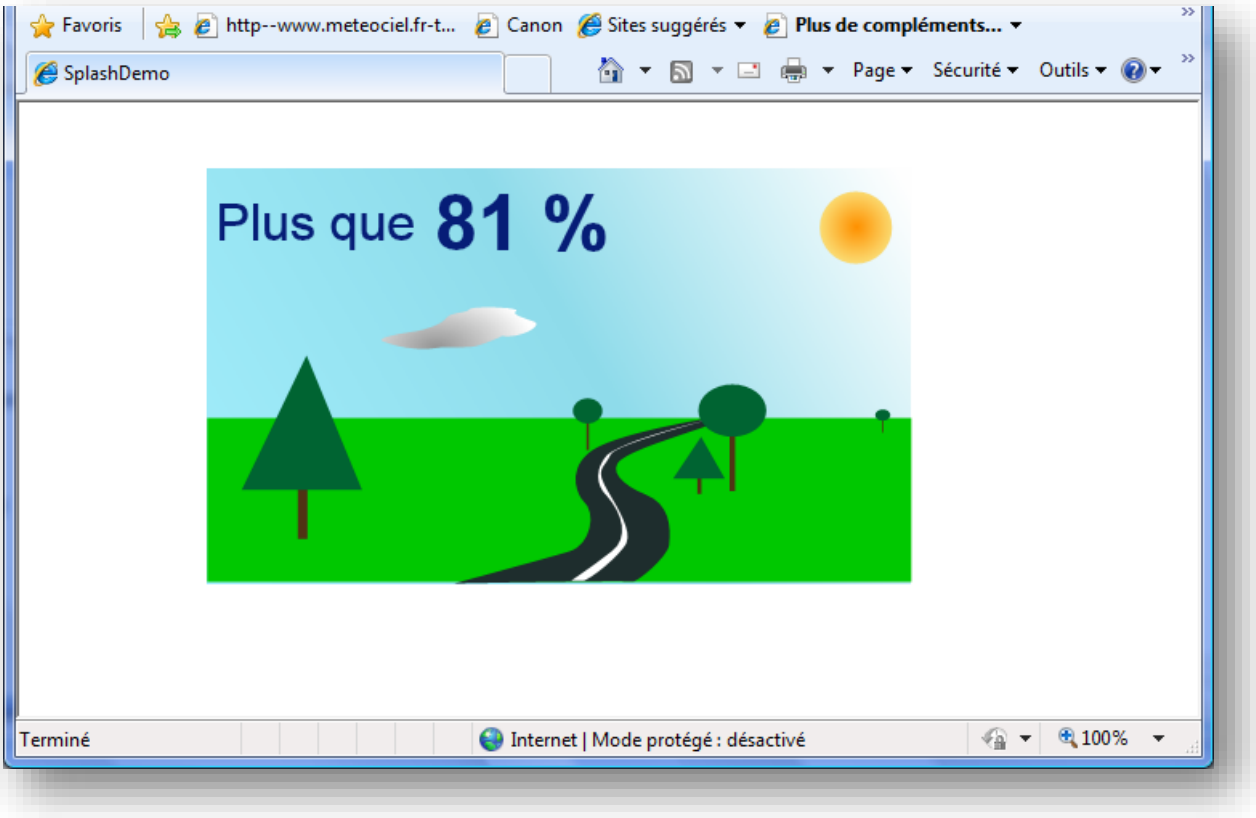


Figure 13 - Le Splash en action !

Il est temps de faire les dernières corrections, si nécessaire.

Making-off

Beaucoup de films se terminent par un making-off ce que le spectateur adore en général (ce qui permet d'éviter qu'il ne se lève avant la fin du générique, rusé !). On y voit le super méchant en train de tronçonner la pauvre victime qui se tord de douleur se prendre un fou-rire par exemple. Cela nous rappelle que ce n'est « que » du cinéma. Une mise en perspective intéressante qui rappelle le fameux tableau « ceci n'est pas une pipe » de Magritte. Et qui prouve accessoirement qu'un (bon ?) film ne se tourne pas en une seule prise…

Je me suis demandé comment on pouvait transposer ce concept à cet article, et je me suis dit que vous faire part de tout ce qui a coincé à un moment où un autre redonnerait une dimension humaine à tout cette prose. Un article tel que celui que vous lisez ne s'écrit pas en une seule « prise ». Il y a des ratés !

Par exemple lors des premières exécutions j'avais oublié de rajouter la déclaration du script « `splash.js` » dans le fichier HTML, et bien entendu le splash s'affichait mais le pourcentage n'était pas mis à jour…

Autre problème rencontré : les textes ont été créés dans Design puis importés dans Blend avec des transformations et lorsque le pourcentage se mettait à jour la mise en page de cette partie de texte disparaissait (le bloc de texte revenait à la police par défaut). Pour régler le problème il a fallu que sous Blend je supprime le `txtPercent` et que je recréé un `TextBlock` de même nom configuré correctement (taille de fonte, couleur…).

On notera aussi que le dessin créé sous Design était légèrement plus grand que la page XAML de l'application. Le soleil à droite de l'image était coupé presque au trois-quarts. J'ai dû agrandir la page de l'application pour qu'elle soit à peu près de la taille du splash. Mais ça coinçait toujours. Petit oubli : il fallait aussi modifier le style CSS qui permet de centrer l'application…

Voilà. Comme quoi il y a toujours mille détails auxquels il faut faire attention !

Code Source

Vous pouvez télécharger le code source de l'article (accompagné du PDF de l'article dans sa version originale non mise à jour par rapport au présent livre).

J'ai laissé le répertoire `slowdown` dans le projet Silverlight pour que vous pensiez à y mettre les fichiers ralentisseurs car je n'ai bien sûr pas laissés ceux que j'ai utilisés (pour le coup, c'est le zip de l'article qui serait devenu très lent à télécharger !).

N'oubliez pas Sloppy à l'adresse que j'ai indiquée dans cet article (sinon GIYF – ou pour rester dans le monde Microsoft, BIYF).

Autre exemple

Vous pouvez regarder la petite application de démonstration que j'ai mise en ligne (les codes postaux), je lui ai ajouté un splash qui sera certainement plus facile à voir si vous utilisez Sloppy pour invoquer l'adresse suivante : http://www.e-naxos.com/svcp/SVCPTestPage.html

Conclusion

Ce long article prend fin… J'espère vous avoir donné envie de faire vos propres splash !

La technique étudiée ici se concentre sur la page de test HTML, on peut bien entendu faire la même chose avec la page hôte en ASPX. De même la fonction JScript se résume à modifier le pourcentage écrit, on peut facilement en compliquant un peu le

code modifier la largeur d'un rectangle pour recréer une barre de progression ou créer tout un tas d'effets visuels. JScript n'était pas le propos de cet article, je laisse les virtuoses de ce langage s'en donner à cœur-joie !

Centrer un splash screen personnalisé avec Silverlight

Faire de belles applications sous Silverlight est un plaisir, mais une belle application n'est terminée que lorsqu'elle dispose de son splash screen personnalisé. La "final touch" qui fait voir au monde que vous n'êtes pas du genre à vous contenter des comportements par défaut et que vous êtes un vrai développeur, un dur, un tatoué ! – Cela étant dit avec toutes les réserves exprimées dans l'article précédent…

Partons du principe que vous avez déjà un beau splash screen (l'article précédent devrait vous aider si vous ne savez pas encore le faire). Donc une présentation sous la forme d'un fichier Xaml contenant la définition d'un Canvas avec plein de jolies choses dedans. C'est le format Silverlight 1.0 qui est utilisé pour les splash screens.

Tout va bien, vous avez fait tout ce qu'il faut, mais quand vous lancez votre application le splash est affiché en haut à gauche ! Damned ! Alors on commence à bricoler. Certains vont tenter de fixer une taille plus grande au Canvas de base et de centrer les objets dedans. Pas glop! Ça ne s'adapte que très mal à la largeur réelle du browser... D'autres vont plonger les mains dans JavaScript pour calculer dynamiquement la position du Canvas. Compliqué et fatiguant...

Je vais vous donner une astuce. Plus facile à mettre en œuvre je n'ai pas en stock. Le truc consiste simplement à englober votre Canvas dans une balise `Grid` sans paramètres !

Et oui, la `Grid` est utilisable dans un splash sous Silverlight 2+. Voilà comment faire :

```
1: <Grid>
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     >
5:     <Canvas x:Name="MonSpash" ..... >
6:     </Canvas>
7:     </Grid>
8: </Grid>
```

XAML 1 - Centrage d'un splash screen

C'est tout ! Votre splash, originellement dans le `Canvas` "`MonSplash`" (lignes 5 et 6) se trouve entouré par une `Grid`. Et le tour est joué, le splash apparait bien centré sur le browser quelle que soit ses dimensions.

Attention à un détail : Le fichier Xaml du splash est placé dans le l'application Web et non dans le projet Xap de Silverlight (normal sinon il serait chargé avec l'appli et ne pourrait plus servir de splash). Mais dans cette configuration Expression Blend ne reconnaît le fichier que comme une source Silverlight 1.0, du coup si vous voulez rouvrir le splash sous Blend ce dernier affiche une erreur, `Grid` étant du SL 2 et ne pouvant être *root* d'un fichier Xaml SL 1.0. Je vous recommande donc de placer la `Grid` une fois que vous aurez terminé la conception du splash sous Blend... Dans le pire des cas vous pouvez toujours supprimer la balise `Grid`, travailler sur le splash, et remettre la balise. C'est tellement simple que cela ne devrait vraiment pas gêner.

Pour voir un splash personnalisé centré, regardez l'application de démo : les **Codes Postaux Français sous Silverlight**. L'application a été mise à jour pour intégrer un splash.

Silverlight et la sérialisation

La sérialisation est un besoin fréquent que le framework .NET sait parfaitement gérer. La sérialisation binaire s'obtient d'ailleurs le plus naturellement du monde en ajoutant un simple attribut `[Serializable]` à la définition de la classe. Mais voilà, essayez de décorer une classe avec `SerializableAttribute` sous Silverlight... Surprise ! Un message d'erreur vous prévient que l'espace de nom ne peut être trouvé.

Pas de sérialisation ?

Non. Pas de sérialisation binaire en tout cas sous Silverlight. La classe `SerializableAttribute` n'est tout simplement pas définie dans `System`.

Avant d'aller crier au scandale sur les blogs ou forums officiels essayons de réfléchir...

A quoi sert la sérialisation ?

Sérialiser, cela sert à prendre un "cliché" d'une instance dans le but d'être capable de recréer une copie parfaite de cette dernière au bout d'un certain temps.

Un certain temps... cela peut être une milliseconde ou bien 1 jour ou 1 an. Dans le premier cas le stockage du "cliché" est en mémoire généralement, mais dans le

second, il s'agit le plus souvent d'un stockage persistant tel qu'un fichier disque ou une base de données.

Pas de gestion de disque local en RIA

La nature même de Silverlight et le style bien particulier d'applications qu'il permet d'écrire fait qu'il n'existe aucune gestion de disque local. En réalité il n'existe rien qui permette d'accéder à la machine hôte, en lecture comme en écriture, question de sécurité.

On notera quelques exceptions à cette règle : l'`OpenFileDialog` qui permet d'ouvrir uniquement certains fichiers sur les disques de l'hôte et l'`IsolatedStorage`, espace de stockage local protégé pouvant être utilisé par une application Silverlight. Toutefois `OpenFileDialog` ne règle pas le problème de persistance et si l'`IsolatedStorage` peut être exploité en ce sens il s'agit d'un espace restreint, inlocalisable par un utilisateur "normal" donc in-sauvegardable sélectivement. Autant dire que l'`IsolatedStorage` peut rendre des services ponctuels (cache de données principalement) mais que ce n'est certainement pas l'endroit où stocker des données sensibles ou à durée de vie un peu longue.

Bref, en dehors de ces exceptions qui ne règlent pas tout à fait le besoin de persistance des instances, une application de type RIA ne peut sérieusement gérer des données que distantes...

En tout logique...

Reprenons : la sérialisation sert à persister des instances pour un stockage à plus ou moins longue échéance ou une communication avec un serveur distant. La nature d'une application RIA supprime la liberté d'un stockage local fiable et facilement maitrisable.

En toute logique sérialiser des instances sous Silverlight n'a donc aucun intérêt, sauf pour communiquer avec un serveur distant.

Comme le framework .NET pour Silverlight est une version light du framework complet il a bien fallu faire des coupes sombres... La sérialisation n'a pas échappé à cette rigueur. La principale raison d'être de la sérialisation sous Silverlight étant la communication (quelle que soit la technologie cela se fait sur une base XML le plus souvent) et cela réclamant une sérialisation XML plutôt que binaire, `SerializableAttribute` et sa sérialisation binaire ont ainsi été "zappés" !

Une solution

La situation est grave mais pas désespérée. S'il reste difficile le plus souvent d'utiliser directement sous Silverlight des classes conçues pour le framework complet (et pas seulement à cause de la sérialisation binaire), il est tout à fait possible de sérialiser des instances sous Silverlight à condition d'utiliser les mécanismes qui servent aux communications avec les serveurs distants.

Le DataContract

Silverlight utilise une partie du mécanisme WCF du `DataContract` (mais l'attribut `DataMember` n'existe pas). On trouve même une classe le `DataContractSerializer` qui fournit le nécessaire pour sérialiser et désérialiser des instances même si celles-ci ne sont décorées d'aucun attribut particulier.

Au final la sérialisation sous Silverlight est plus simple que la sérialisation binaire par `SerializableAttribute` sous le framework complet !

Un exemple de code

Einar Ingebrigtsen, un MVP nordiste travaillant en Scandinavie, a eu la bonne idée de proposer deux méthodes utilitaires qui montrent comment utiliser le `DataContractSerializer`. Plutôt que d'imiter et réinventer la roue, regardons son code :

```
1:  public string Serialize<T>(T data)
2:          {
3:              using (var memoryStream = new MemoryStream())
4:              {
5:                  var serializer = new
                        DataContractSerializer(typeof(T));
6:                  serializer.WriteObject(memoryStream, data);
7:
8:                  memoryStream.Seek(0, SeekOrigin.Begin);
9:
10:                 var reader = new StreamReader(memoryStream);
11:                 string content = reader.ReadToEnd();
12:                 return content;
13:             }
14:         }
15:
16:         public T Deserialize<T>(string xml)
17:         {
18:             using (var stream = new
                      MemoryStream(Encoding.Unicode.GetBytes(xml)))
19:             {
20:                 var serializer = new
                        DataContractSerializer(typeof(T));
21:                 T theObject = (T)serializer.ReadObject(stream);
22:                 return theObject;
23:             }
24:         }
```

Code 1 – Sérialisation et désérialisation

C'est simple et efficace. Attention, pour utiliser ce code il faudra ajouter une référence à `System.Runtime.Serialization.dll`, puis un using `System.Runtime.Serialization`.

L'utilisation de ces méthodes tombe dès lors sous le sens, mais un petit exemple est toujours plus parlant :

```
1:  { ....
2:  // nouvelle instance
3:  var t = new Test { Field1 = "test de serialisation", Field2 = 7 };
4:  t.List.Add(new test2 { Field1 = "item 1" });
5:  t.List.Add(new test2 { Field1 = "item 2" });
6:  t.List.Add(new test2 { Field1 = "item 3" });
7:
8:  // sérialisation et affichage
9:  var s = Serialize<Test>(t);
10: var sp = new StackPanel { Orientation = Orientation.Vertical };
11: LayoutRoot.Children.Add(sp);
12: sp.Children.Add(new TextBlock { Text = s });
13:
14: // désérialisation et affichage
15: Var t2 = Deserialize<Test>(s);
16: var result = t2 == null ? "null" : "instantiated";
17: sp.Children.Add(new TextBlock { Text = result });
18: }
19:
20: // la classe de test
21: public class Test
22: {
23:   public string Field1 { get; set; }
24:   public int Field2 { get; set; }
25:   private ObservableCollection<test2> list =
26:                 new ObservableCollection<test2>();
27:   public ObservableCollection<test2> List { get { return list; } }
28: }
29:
30: // la classe des éléments de la liste de la classe Test
31: public class test2
32: {
33:   public string Field1 { get; set; }
34: }
```

Code 2 - Test de sérialisation

Conclusion

Une fois que l'on a compris pourquoi la sérialisation binaire est absente de Silverlight, et une fois qu'on a trouvé une solution de rechange, on se sent beaucoup mieux !

Il reste maintenant tout à fait possible de stocker en local le résultat de la sérialisation dans un fichier texte (en utilisant l'`IsolatedStorage`) ou bien de le transmettre à un service Web ou autre...

On notera qu'il existe aussi des librairies externes comme JSON.NET qui permettent d'étendre et de simplifier la sérialisation des instances ou des grappes d'objets. Mais dans de nombreux cas pourquoi charger la mule alors qu'on peut trouver dans Silverlight des solutions qui ne réclament rien de plus que le plugin...

Gestion des cookies

Silverlight propose un espace de stockage privé pour chaque application sur la machine cliente. Cet espace s'appelle **l'Isolated Storage** (*stockage isolé*, car protégé et isolé du reste de la machine cliente). C'est dans cet espace qu'il est conseillé de sauvegarder les données propres à. Toutefois on peut être amené pour certaines informations à préférer le stockage sous la forme de **Cookies** traditionnels. On peut aussi avoir besoin de lire certains cookies posés par l'éventuelle application hôte en ASP.NET par exemple.

Quelles que soient les raisons, il peut donc s'avérer utile de manipuler les cookies depuis une application Silverlight.

Capture de l'application exemple (testable en live en cliquant sur le titre de ce chapitre) :

Figure 14 - capture application de test de la gestion des cookies

Techniquement, et après avoir ajouté à votre code le `using` du namespace "`System.Windows.Browser`", il suffit pour créer (ou modifier) un cookie d'appeler la méthode `SetProperty` de `HtmlPage.Document` :

```
 1:  private bool setCookie(string key, string value)
 2:          {
 3:              if (string.IsNullOrEmpty(key) ||
                            string.IsNullOrEmpty(value))
 4:              { return false; }
 5:              DateTime dt = calExpiry.SelectedDate ?? DateTime.Now
 6:                  + new TimeSpan(1, 0, 0);
 7:              string s = key.Trim() + "=" + value.Trim()
 8:                  + ";expires=" + dt.ToString("R");
 9:              HtmlPage.Document.SetProperty("cookie", s);
10:              return true;
11:          }
```

Code 3 - Enregistrer une valeur dans un cookie

Pour la lecture on accède à la chaîne "`Cookies`" de `HtmlPage.Document` qu'on peut ensuite traiter pour séparer chaque cookie :

```
 1:  private string getCookie(string key)
 2:          {
 3:              string[] cookies = HtmlPage.Document.Cookies.Split(';');
 4:              foreach (string cookie in cookies)
 5:              {
 6:                  string[] pair = cookie.Split('=');
 7:                  if (pair.Length == 2)
 8:                  {
 9:                      if (pair[0].ToString() == key)
10:                          return pair[1];
11:                  }
12:              }
13:              return null;
14:          }
```

Code 4 - Lire une valeur depuis un cookie

Bref, rien de bien compliqué mais qui pourra vous servir dans certains cas.

Le code source du projet : Cookies.zip (62,65 kb)

Application transparente

Il est tout à fait possible de rendre une application (ou une fenêtre) Silverlight transparente, c'est à dire que son contenu apparaîtra sur la couleur de fond du site hôte. Cela peut s'avérer très utile pour des petits morceaux de Silverlight placés sur un site Html ou Asp.net, pour créer une barre de menu par exemple. Si la charte couleur du site est modifiée il ne sera pas nécessaire de modifier et recompiler la ou les applications Silverlight.

Pour arriver à ce résultat il faut faire deux choses simples dont la syntaxe dépend de la nature de la page où s'insère le plugin (ASP.NET ou HTML)

Exemple ASP.NET :

```
<asp:Silverlight PluginBackground="Transparent" Windowless="true"
  ID="Menu" runat="server"
  Source="~/ClientBin/Menuxap" MinimumVersion="xxxxx"
/>
```

Code 5 - Application transparente en ASP.NET

Exemple HTML :

```
<object data="data:application/x-silverlight," type="application/x-silverlight-2-
b2" >
<param name="source" value="ClientBin/Menu.xap"/>
<param name="onerror" value="onSilverlightError" />
<param name="pluginbackground" value="Transparent" />
<param name="windowless" value="true" />
</object>
```

Code 6 - Application transparente en HTML

Les deux paramètres importants étant : `PluginBackgrounf` et `Windoless`.

Par défaut une application Silverlight occupe 100% de la fenêtre hôte, dans le cas d'une barre de menu ou d'un autre type de fenêtre intégré à une page ASP.NET/HTML on ajoutera donc dans les balises la déclaration de la hauteur et de la largeur exacte qu'on souhaite réserver pour le plugin. Bien entendu le contenu de l'application Silverlight doit être dépourvu d'arrière-plan, cela va sans dire, sinon, transparence ou non, c'est lui qui sera affiché et non le background de la page hôte...

Contrôle DataForm et Validation des données

Le composant DataForm a été introduit dans Silverlight 3, il est en quelque sorte le pendant mono enregistrement du composant `DataGrid`. Avec cette dernière on montre plusieurs enregistrements à la fois, avec la `DataForm` on présente les données d'une seule fiche. On conserve malgré tout la possibilité de naviguer de fiche en fiche, et le composant est hautement personnalisable grâce aux styles et templates.

Imaginons une petite fiche permettant de créer un login : pseudo, e-mail et mot de passe sont le lot de ce genre de cadres de saisie. Pour les besoins de la démo commençons par créer une classe représentant l'ensemble des informations à saisir (voir le code en fin de billet).

Rien d'exceptionnel ici donc, juste une petite classe. Mais vous remarquerez plusieurs choses :

Les propriétés sont décorées d'attributs qui permettent de modifier notamment l'affichage des labels dans la `DataForm`, un nom interne de propriété n'est pas forcément adapté pour une lecture par l'utilisateur final.

Les propriétés sont décorées de l'attribut `Required`. Autre particularité pour la DataForm : certains champs peuvent être obligatoires.

Dans le code (ci-après) chaque propriété prend en charge sa validation : elle lève une exception dès que la valeur passée n'est pas conforme à celle attendue. Il existe d'ailleurs un doublon fonctionnel dans ce code : la présence de la méthode `CheckNull` qui s'assure qu'une propriété n'est pas vide, et l'attribut `Required`. En l'état c'est la méthode `CheckNull` qui prend le dessus, on pourrait donc supprimer l'attribut.

Il existe bien d'autres attributs pour la DataForm et le propos de ce billet n'est pas de tous les détailler, mais il est important de noter comment le code peut prévoir certains comportements qui seront reportés sur l'interface utilisateur sans réellement interférer avec elle... subtile.

Regardons maintenant le code XAML de la page affichée par Silverlight (en fin de billet).

Tout d'abord vous remarquerez que l'instance de `LoginInfo` n'est pas créée dans le code C# mais sous forme de ressource dans la fiche, en XAML.
Ensuite nous définissons une balise `DataForm` dont l'item courant est lié à l'instance de `LoginInfo`.

Et c'est tout. Pour la décoration j'ai ajouté un bouton "`ok`" qui accessoirement lance la validation totale de la fiche. Ne saisissez rien et cliquez dessus : vous verrez apparaître le cadre des erreurs avec la liste de toutes les validations qui ont échouées. Bien entendu l'aspect de ce cadre est templatable aussi.

Capture de l'application exemple :

Figure 15 - Test de la validation des données

Vous pouvez aussi télécharger le code du projet : SL3DataValidation.zip (60,40 kb)

A noter : la présence du petit symbole info, il répond à l'attribut `Description` prévu dans le code de la classe et permet d'informer l'utilisateur.
Vous remarquerez aussi qu'en passant d'un champ à un autre les champs invalidés sont décorés par une lisière rouge dont un coin est plus marqué : cliquez sur ce dernier et vous obtiendrez le message d'erreur avec sa petite animation. L'aspect de ce cadre ainsi que l'animation sont bien entendu modifiables à souhait.

Code de la classe LogInfo :

```
 1:  public class LoginInfo : INotifyPropertyChanged
 2:      {
 3:          private string loginName;
 4:
 5:          [Required()]
 6:          [Display(Name="Login name:",
                     Description="Votre login personnel.")]
 7:          public string LoginName
 8:          {
 9:              get { return loginName; }
10:              set
11:              {
12:                  checkNull(value);
13:                  loginName = value.Trim();
14:                  dochange("LoginName");
15:              }
16:          }
17:
18:
19:          private string email;
20:
21:          [Required()]
22:          [Display(Name="e-mail:",
                     Description="Adresse mail pour vous joindre.")]
23:          public string Email
24:          {
25:              get { return email; }
26:              set
27:              {
28:                  checkNull(value);
29:                  checkMail(value);
30:                  email = value.Trim();
31:                  dochange("Email");
32:              }
33:          }
34:
35:          private string password;
36:
37:          [Required()]
38:          [Display(Name="Mot de passe:",
                     Description="Votre mot de passe de connexion.")]
39:          public string Password
```

```
40:         {
41:             get { return password; }
42:             set
43:             {
44:                 checkNull(value);
45:                 password = value.Trim();
46:                 dochange("Password");
47:             }
48:         }
49:
50:         private string passwordCheck;
51:
52:         [Required()]
53:         [Display(Name="Contrôle:",
                     Description="Retapez ici votre mot de passe.")]
54:         public string PasswordCheck
55:         {
56:             get { return passwordCheck; }
57:             set
58:             {
59:                 checkNull(value);
60:                 passwordCheck = value.Trim();
61:                 if (string.Compare(password,passwordCheck)!=0)
62:                     throw new Exception("Les mots de passe diffèrent !");
63:                 dochange("PasswordCheck");
64:             }
65:         }
66:
67:         private void checkMail(string value)
68:         {
69:             string pattern = @"^(([^<>()[\]\\.,;:\s@\""]+"
70:                       + @"(\.[^<>()[\]\\.,;:\s@\""]+)*)|(\"".+\""))@"
71:                       + @"((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}"
72:                       + @"\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+"
73:                       + @"[a-zA-Z]{2,}))$";
74:                   Regex regMail = new Regex(pattern);

75:                   if (!regMail.IsMatch(value))
                          throw new Exception("E-mail non valide !");
76:         }

77:
```

```
78:          private void checkNull(string value)
79:          {
80:              if (string.IsNullOrEmpty(value))
                     throw new Exception("Ne peut être vide !");
81:          }
82:
83:          #region INotifyPropertyChanged Members
84:
85:          public event PropertyChangedEventHandler PropertyChanged;
86:          private void dochange(string property)
87:          {
88:              if (PropertyChanged!=null)
                       PropertyChanged(this,
                                  new PropertyChangedEventArgs(property));
89:          }
90:
91:          #endregion
92:      }
```

Code 7 - Classe de Login décorée avec les attributs de validation de données

Le code XAML de la fiche :

```
 1:  <UserControl
 2:      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3:      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4:      xmlns:dataFormToolkit="clr-namespace:System.Windows.Controls;assembly=
                              System.Windows.Controls.Data.DataForm.Toolkit"
 5:      xmlns:local="clr-namespace:DataValidation"
 6:      x:Class="DataValidation.MainPage"
 7:      Width="300" Height="320">
 8:
 9:      <UserControl.Resources>
10:       <local:LoginInfo x:Key="loginRec"  />
11:      </UserControl.Resources>
12:
13:      <StackPanel x:Name="LayoutRoot">
14:          <dataFormToolkit:DataForm
15:              x:Name="PwForm"
16:              CurrentItem="{StaticResource loginRec}"
17:              Foreground="Black"
18:              Height="292"
19:              VerticalAlignment="Top"
20:              TabNavigation="Cycle"/>
21:          <Button
22:              Content="Ok"
23:              VerticalAlignment="Center"
24:              HorizontalAlignment="Center"
25:              Click="Button_Click"/>
26:      </StackPanel>
27:  </UserControl>
```

XAML 2 - UserControl de login affichant les erreurs de saisie

DeepZoom

Deep Zoom, vous connaissez ? Peut-être pas car il n'a pas eu le succès qu'il mérite et l'arrêt de Silverlight n'a pas arrangé les choses. Si vous ne savez pas ce que c'est, voici en deux mots quelques explications : Deep Zoom est une technologie permettant d'afficher des images énormes sans avoir besoin d'attendre que tout soit téléchargé. On peut ainsi faire un "zoom profond" (deep zoom) sur une image, ou plus amusant, sur une composition d'images. Si vous connaissez Binq Maps ou Google Maps cela vous donne une bonne idée (partir d'une vision de la France depuis le ciel et zoomer jusqu'à voir le nombre de voitures garées devant chez vous).

Des tas de possibilités s'ouvrent si je vous dis que le logiciel Deep Zoom Composer est gratuit et qu'il permet de créer toute sorte de compositions et qu'il est même capable de créer des exécutables directement utilisables, voire même les sources complètes d'un projet Blend (version 3 car cela n'a jamais évolué depuis, à ma connaissance en tout cas) !

Figure 16 - Exemple DeepZoom

C'est à partir de cette dernière option que j'ai créé la démo à laquelle vous accéderez en cliquant sur le titre de ce chapitre (l'application de démo se trouve en fin de billet). Sur la base du projet créé par Deep Zoom Composer j'ai pu modifier certains comportements, améliorer certaines petites choses (notamment à l'aide de Behaviors dont certains sont déjà dans le projet). Vraiment incroyable et d'une rapidité fulgurante, si on fait abstraction du processus de création des milliers de fichiers nécessaires rien que pour la présente démo. En tout 4593 fichiers, 303 sous-répertoires pour un poids total de 52,2 Mo. Du lourd donc. Surtout comparé au fichier exécutable Silverlight produit (le .xap) qui ne fait que 79 Ko.

Alors attention aux compositions géantes qu'on est tenté de concevoir, après il faut assumer niveau hébergement !

En revanche les possibilités d'interactions et de personnalisations au travers de Blend sont innombrables. On sent un potentiel énorme et on aurait pu penser que les idées d'utilisation un peu originales viendraient certainement avec le temps mais ce ne fut pas le cas. Mais Deep Zoom est toujours utilisable, comme Silverlight, et cela peut

rendre des services (présenter un plan d'architecte très grand avec plusieurs niveaux de zoom, sans rien avoir à coder par exemple, le tout hébergé sur le web et utilisable depuis n'importe quel PC ou Mac).

Ce billet ne présente pas de code, il suffit de disposer de Deep Zoom Composer, de quelques photos, de suivre la procédure et de bricoler un peu le C# du projet pour obtenir le même résultat. Le but était avant tout de vous parler de Deep Zoom.

nota: les photos font partie de celles que j'ai prises à Barcelone lors des TechDays 2008. Vous pourrez découvrir une photo d'une fière assemblée, en train de festoyer. Ne me cherchez pas sur la photo, il fallait bien quelqu'un pour immortaliser l'instant et tenir l'appareil photo...

Projection 3D

Aujourd'hui je vais vous parler de la **projection 3D**. Ce qu'on appelle de la "2,5D" car ce n'est déjà plus de la 2D mais ce n'est pas encore de la vraie 3D (qui implique la manipulation d'objets vectoriels 3D comme WPF sait le faire). Sous Silverlight, la 2,5D est un groupe de transformation qui s'ajoute à celui déjà présent (et permettant les translations, le skew, les rotations, etc). Ce groupe se présente globalement de la même façon (en tout cas sous Blend) mais les effets sont différents puisqu'il s'agit de créer des effets de perspectives 3D avec les objets 2D de Silverlight.

A quoi cela peut-il bien servir ?

A plein de choses ! Au niveau graphique et visuel s'entend.

Pour illustrer tout cela voici un exemple qui montre comment on peut animer une petite fiche d'information qui donne l'impression d'avoir deux faces, comme une carte de jeu par exemple : Exemple Projection.

L'exemple est entièrement réalisé sous Blend, sans une ligne de C# ni aucune de XAML tapée à la main (car forcément, il y a du XAML mais c'est Blend qui s'en occupe). J'arrive à obtenir en deux storyboards et en utilisant un Behavior pour déclencher les animations depuis les boutons au lieu de code behind C#. Je me suis même offert le luxe d'utiliser les modes de EaseIn/Out qui permettent d'obtenir des mouvements plus "organiques" (effets de rebonds ou d'élastique par exemple).

Comme il n'y a rien dans ce projet d'autre que deux timelines, plutôt qu'un long discours je vous propose de charger le projet exemple : 3DProjection.zip (61,42 kb)

En deux mots, l'astuce consiste à avoir deux fiches identiques au niveau de leur forme extérieure. L'une est la face A, l'autre la face B. Le contenu des deux fiches est totalement libre.

Au début la fiche A est visible "à plat", "normalement" donc. Sur l'appui du bouton "aide" (simple exemple) l'animation `FlipToBack` est lancée. Elle consiste à appliquer une transformation perspective sur l'axe Y pour donner l'impression que la fiche tourne. Pour que la fiche B ne soit pas visible durant cette phase, elle a subi en début d'animation une simple translation pour l'amener en dehors de l'espace de l'application (par exemple en position 1000 sur l'axe Y).

Au moment où la fiche A passe à 90 degrés, c'est à dire quand elle est vue par la tranche, et comme elle n'a aucune épaisseur, elle disparait visuellement. C'est à ce moment précis que la magie s'opère : on fait faire une rotation à la fiche B pour qu'elle soit sur sa tranche aussi et on l'amène à la place de la fiche A (en annulant la translation sur Y de B) et dans le même temps on fait une translation de 1000 sur Y de la fiche A. Les keyframes utilisées pour cette substitution sont de type `HoldIn`, c'est à dire qu'il n'y a pas de transition douce avec l'état précédant, tout est instantané. Maintenant que la fiche B est à la place de la fiche A, on peut continuer l'animation de B pour qu'elle s'affiche. Le "spectateur" est dupé : durant la microseconde où la fiche A était vue par la tranche nous lui avons substitué la fiche B vue sous le même angle. Le début de l'animation fait tourner A, alors que la seconde partie de l'animation fait tourner B, mais on a l'impression visuelle que c'est la même fiche qui tourne et qu'on en découvre le dos...

Pour revenir de la page B à la page A, il suffit de créer une seconde timeline qui reprend le même principe mais qui fait les choses dans l'autre sens...

Un média player complet bien caché !

Silverlight est livré de base avec un contrôle très versatile, `MediaElement`, capable de jouer de nombreux formats comme les mp3 ou les vidéos.

Si ce contrôle est très puissant il est un peu "nu" de base et il faut soi-même ajouter les boutons de commande comme "play" et looker l'ensemble. Avec **Blend** c'est un jeu d'enfant. Enfin, de grand enfant qui a un peu de temps devant lui tout de même. D'où la question : n'existerait-il pas un `MediaElement` déjà tout habillé ?

Si vous possédez **Expression Media Encoder** vous savez que cet extraordinaire outil (servant principalement à encoder des médias) donne le choix entre plusieurs formats de sortie dont des projets HTML tout fait intégrant un média player tout looké, et mieux encore, avec un choix parmi de nombreux modèles.

Quel rapport entre Media Encoder et un projet Blend/VS ? C'est tout simple : lorsque vous installez Media Encoder, sous Blend dans l'onglet *Assets* vous disposez, en plus de `MediaElement` d'un nouveau contrôle "`MediaPlayer`" !

Par défaut ce composant ressemble à l'image ci-dessous. Pour le relooker, il suffit de faire un clic droit et d'éditer une copie du template !

Figure 17 - Média player templaté

Reste la question à 10 centimes d'euro : oui mais Media Encoder est livré avec de nombreux modèles dont certains ont déjà un look sympa, ne pourrait-on pas récupérer ces modèles au lieu de templater à la main le `MediaPlayer` ?

Si, c'est possible (© Les Nuls, "Hassan Cehef").

Comment ? Là c'est plus cher... Non, comme je suis un chic type, voici la solution gratuite :

Encoder s'installe avec le code source des modèles qui se trouvent dans le répertoire "`C:\Program Files\Microsoft Expression\Encoder 3\Templates\en`" (à moduler selon la version de votre OS), il suffit donc de piocher le modèle qu'on désire utiliser, et grâce aux sources d'extraire le contrôle avec son template et de l'intégrer à son propre projet !

Le célèbre Tim Heuer décrit (en anglais) la méthode à suivre, je vous renvoie ainsi à son billet Using Encoder Templates in your Silverlight Application si jamais vous n'arrivez pas à vous dépatouiller seul avec le code source des projets Encoder.

Intégrer de la vidéo, même HD, dans une application Silverlight n'a jamais été aussi simple... et beau.

Silverlight et le multithread (avec ou sans MVVM)

La "parallélisation" des traitements est devenu un enjeu majeur et paradoxal du développement et Silverlight n'échappe pas à ce nouveau paradigme.

Nouveau Paradigme ?

Nouveau paradigme, cela fait toujours un peu "*attention je suis en train de dire quelque chose d'intelligent et j'utilise une tournure à la mode qui fait savante pour le faire remarquer*"... Mais parfois cela a aussi un vrai sens et une vraie justification !

Nouveau paradigme donc que la programmation multitâche, ou pour être encore plus chébran, parlons de programmation parallèle, voire de parallélisme tout en évitant le trop prétentieux "massivement parallèle" (nos PC ne comptent pas encore la centaine de "cœurs" qu'Intel prévoit pour dans quelques années).

On le sait depuis un moment, la fréquence des processeurs n'évoluera plus comme elle l'a fait jusqu'à lors. Le fondeur de processeurs se retrouve ainsi face à un mur infranchissable, un peu comme l'était le mur du son pour l'aviation jusqu'à ce jour d'octobre 47 ou le pilote Yeager à bord du Bell-X-1 finit par traverser les briques de ce mur qu'on avait cru impénétrable. Pour nous il s'agit du "*mur des performances*". On miniaturise, on diminue l'épaisseur des couches, on tasse les transistors, mais on butte sur la fréquence, l'élever encore plus ferait tout fondre... Le seul moyen d'aller plus vite reste de multiplier les processeurs... Et comme multiplier les cartes mères atteint vite une limite en termes de prix et de consommation électrique, on essaye désormais de tasser des cœurs sur les puces comme on tassait les transistors à

l'étape précédente. Ce n'est pas une option et il n'y a pas de "plan B", c'est la seule solution jusqu'à ce qu'existent les ordinateurs quantiques (dont la faisabilité reste totalement à prouver malgré quelques bricolages savants surmédiatisés).

Deux, quatre, huit, trente-deux, bientôt cent cœurs dans les machines de monsieur Tout-Le-Monde. Déjà 4 ou 8 dans le moindre smartphone haut de gamme, même 16 si on compte les 8 du GPU de certains modèles !

Hélas ce n'est pas avec les techniques de programmation utilisées jusqu'aujourd'hui que les informaticiens vont s'en sortir. Les fondeurs ont trouvé le moyen de percer le "mur des performances", et si les informaticiens ont une idée de comment ils vont relever le défi, ils ne se sont pas encore dotés des outils permettant de le faire : une formation *ad hoc* et une approche radicalement différente de la programmation.

On peut ainsi réellement, et sans emphase exagérée, parler de "nouveau paradigme" de la programmation parallèle car seule cette nouvelle approche va permettre aux logiciels de bénéficier des accélérations offertes par les processeurs modernes et encore plus par ceux à venir. Et pour cela il va falloir que les développeurs se mettent à niveau alors qu'ils sont déjà à la traîne de cette révolution…

Depuis que j'ai écrit ces lignes, il n'y a pourtant pas si longtemps, les choses ont tellement vite que j'ai presque l'impression que tout cela est surréaliste… WinRT par exemple est farci d'API asynchrones, le traitement multitâche n'y est même plus une option, c'est une obligation. Comme quoi j'avais raison de vous sensibiliser à ce problème ! Aujourd'hui il ne s'agit plus de se poser la question de savoir s'il faut s'en préoccuper, il faut déjà être devenu un expert ! C# avec async/await aide beaucoup les choses mais encore faut-il maîtriser la chose…

Au moment où j'écrivais ce billet je venais d'acquérir une machine à 4 cœurs hyperthreadés, donc à 8 cœurs apparents. Avec certaines applications elle allait à peine aussi vite que mon double cœurs E6700 overclocké (et pas toujours)… Je regardais la jolie fenêtre des performances de Windows, je voyais ces huit magnifiques graphiques qui sont autant de preuves formelles de la puissance de la bête (et un enchantement de geek), mais un seul s'activait vraiment le plus souvent, un autre le faisant mollement, et les 6 autres piquaient un roupillon désespérant la plupart du temps ! D'autres machines sont passées depuis, des huit cœurs aussi. Le véritable gain provient du passage du disque dur au SSD, mais toujours pas des cœurs multiples. Pourquoi ? Parce qu'aujourd'hui encore les applications sont écrites avec les pieds ! Le multitâche n'est réellement utilisé que rarement et de façon

ponctuelle. Ça traîne… Et je vois tous ces cœurs qui dorment. Ces applications je les maudis et dans la foulée leurs concepteurs incompétents qui ont programmé comme on le faisait il y a 20 ans : en pensant qu'un seul processeur anime la machine et que Windows va se débrouiller pour partager le temps (et que si l'utilisateur trouve que "ça rame" il n'a qu'à acheter une machine plus récente…).

En revanche je suis plein de compliments confraternels à l'égard des développeurs qui ont programmé les quelques applications qui savent se servir de mes huit cœurs ! Ces programmes qui s'essoufflaient sur le E6700 deviennent des bombes sur mon i870 ou mon i7-3770 version K ou vont nettement plus vite au minimum, ce qui n'est déjà pas si mal.

Faire partie de ceux qu'on maudit ou de ceux qu'on félicite, il va falloir choisir son camp !

Programmer "parallèle" devient ainsi une obligation de fait, techniquement parlant. Mais il ne faut pas ignorer non plus les cas plus classiques où le parallélisme est le seul moyen de programmer certaines fonctions (ce qui est évident sous WinRT par exemple). C'est ce que nous verrons dans un instant au travers d'un exemple de code.

Mais paradoxal !

D'un côté des PC de bureau équipés de 16 Go de RAM et d'octo-cœurs surpuissants, de l'autre des smartphones aux interfaces à faire pâlir les logiciels PC fonctionnant avec peu de mémoire et de puissance.

Le progrès avance vite et les smartphones haut de gamme qui sortent aujourd'hui on effacer ce paradoxe. Certains modèles sont désormais plus puissants (et plus chers) que la majorité des PC portables grand public.

Si le paradoxe n'a duré que quelques années, si le grand écart ne doit plus être fait, le combat se déplaçant sur le cross-plateforme et le cross-form-factor, il n'en reste pas moins vrai que les smartphones sont largement moins puissant qu'un très bon PC mais que leurs utilisateurs ont une exigence de fluidité et de rapidité qui pose de vrais problèmes de programmation. Fluidité, immersion, capteurs multiples, simultanéité, ubiquité…

Ce sont les défis que le développeur doit dès aujourd'hui relever, et simultanément ! Même en s'y mettant tout de suite, il part en retard (les "utilisateurs de base" possèdent depuis plusieurs années des machines à double cœurs !). Il faut donc

programmer en visant la “scalability” ou “capacité à s’adapter”, s’adapter à la charge, s’adapter au matériel. Tirer le meilleur d’un Phone 7, faire cracher ses chevaux à un PC de bureau 8 cœurs. En gros il faut maîtriser la programmation parallèle. D’autant que même les smartphones eux-mêmes sont aujourd’hui livrés en double, quadri et même octo cœurs… Comme on le voit cette nouvelle approche devient vraiment obligatoire quelles que soient les machines !

En pratique on fait comment ?

On sort ses neurones du mode économie d’énergie certifié Energy Star et on se sert de l’ordinateur le plus massivement parallèle de l’univers : son cerveau :-)

Cela signifie adopter une nouvelle “attitude” face à la programmation quotidienne. La moindre classe doit être pensée dans un contexte multitâche. Cela doit devenir un automatisme, comme de finir chaque instruction par un point-virgule en C#.

Mais comme il n’est pas question dans ce blog de faire un cours complet sur le multitâche, attardons-nous sur un exemple concret et voyons comment introduire un peu de ces réflexes qui tendent vers une meilleur utilisation des ressources physiques sous-jacentes offertes par la machine hôte de nos chefs-d ‘œuvres compilés….

Un exemple concret

Programmer en vue d’une utilisation multitâche n’est pas forcément simple. On peut être ainsi tenté d’y échapper jusqu’au jour où on tombe sur un cas tout simple qui va déboucher sur un magnifique plantage.

Comme le titre du billet l’indique, cela n’a rien de spécifique à MVVM, même si l’exemple sera programmé dans un mode de ce type (mais très édulcoré, ce n’est donc pas un exemple MVVM, pour cela se reporter à l’article complet que j’ai écrit sur le sujet, ou mieux au Tome 2 de ALL DOT BLOT !).

Imaginons ainsi une vue présentant des données issues d’un ViewModel ou directement d’un Modèle (ce que MVVM autorise). Supposons ce dernier cas pour simplifier la “plomberie” de l’exemple. Ce Modèle simule une source de données un peu particulière puisqu’elle fonctionne en permanence et dépend de données externes qui arrivent un peu quand elles veulent. Cela n’a rien d’exotique, prenez un simple Chat, il y aura bien un Modèle fournissant les messages de l’interlocuteur fonctionnant en permanence et la fréquence d’arrivée des données (les messages) est aléatoire (quand l’interlocuteur a envie d’écrire). La lecture d’une sonde sur

smartphone pose le même genre de problème et plus généralement tout capteur physique interne ou externe.

Le modèle

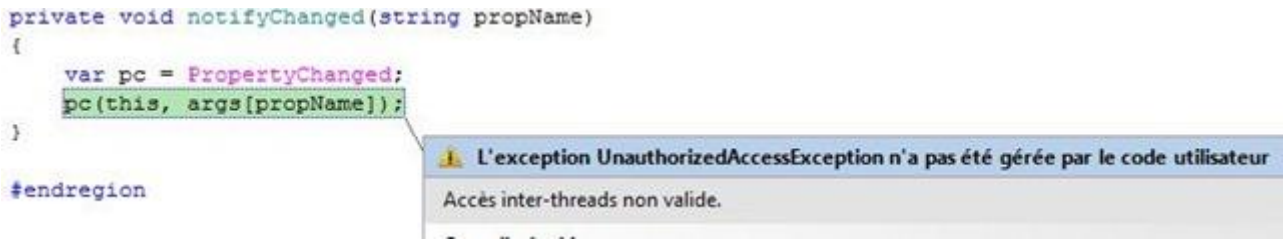
Le Modèle de notre exemple sera bien plus simple encore : il fabrique des nombres aléatoires. Mais il les fabrique dans un thread séparé qui s'endort pour une durée aléatoire après chaque génération de nombre. Pour ce faire la classe `RandomNumberPusher` utilise un `BackgroundWorker`, classe du Framework permettant assez simplement de créer des tâches de fond (code source du projet à télécharger plus bas).

La Vue

Créons une vue (la `MainPage` sera suffisante pour cet exemple minimaliste) et attachons une instance de la classe `RandomNumberPusher` au `DataContext` de la grille principale. Ajoutons un `TextBlock` bindé (voir mon article sur le Binding Xaml) à la propriété `Number`. Et exécutons…

Le plantage…

A intervalle irrégulier l'instance du Modèle va générer en tâche de fond un nombre aléatoire et va déclencher la notification de changement d'état de la propriété `Number` (événement PropertyChanged). La Vue et son `TextBlock` étant bindés à cette dernière, la mise à jour du texte va se faire automatiquement. Oui mais… cela va planter. Pour une raison simple : seul le thread principal ayant créé les objets d'interface peut modifier ces derniers, or le générateur aléatoire fonctionne depuis un thread différent. On peut mettre le plantage encore plus en évidence si on tentait de modifier directement la propriété `Text` du `TextBlock` depuis la tâche de fond. Le modèle de programmation MVVM nous oblige à penser le code autrement (et c'est une bonne chose) et le plantage ne se verra pas forcément (le `TextBlock` ne changera pas de valeur). Mais à un moment donné il risque de s'afficher ceci :



Code 8 - Accès inter-thread - Exception

Accès inter-thread non valide. Le couperet tombe.

Pour obtenir ce message de façon sûre il faut placer un point d'arrêt dans le thread de fond, s'arrêter dessus et repartir car le plus souvent le texte n'est tout simplement pas mis à jour. Comme je le disais plus haut, le plantage serait systématique si la tâche de fond mettait à jour directement le `TextBlock` ce que le montage de type MVVM évite. Le bug n'en est que plus sournois car il ne se passe rien, et rien, ce n'est pas grand-chose ! Sur un écran un peu chargé il sera difficile de détecter le problème durant les tests.

Le mécanisme est simple :

Figure 18 - Mécanisme de l'accès inter thread

Le thread de fond modifie la propriété Number, ce qui déclenche l'événement de changement de propriété (`PropertyChanged`), l'objet de binding est alors activé et tente de mettre à jour la valeur de la propriété Text du TextBlock. Le plantage résulte du fait que la modification de ce dernier est effectuée au travers d'un thread différent de celui gérant l'UI.

Le code xaml

```
 1:  <UserControl x:Class="SLMultiThread.MainPage"
 2:      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3:      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4:      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 5:      xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
 6:      xmlns:model="clr-namespace:SLMultiThread.Models"
 7:      mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">
 8:      <UserControl.Resources>
 9:          <model:RandomNumberPusher x:Key="random" />
10:      </UserControl.Resources>
11:      <Grid x:Name="LayoutRoot" DataContext="{StaticResource random}">
12:          <TextBlock Text="{Binding Number, Mode=OneWay}"/>
13:      </Grid>
14:  </UserControl>
```

XAML 3 - Tester les accès cross thread

On note : la déclaration du namespace de notre Modèle, la déclaration d'une instance du Modèle en ressource, la définition du DataContext du LayoutRoot depuis cette dernière et enfin, le TextBlock bindé à la propriété Number.

Le code C# de la classe RandomNumberPusher n'est pas en cause dans le mécanisme du plantage. Elle est programmée en répondant aux exigences d'une classe dite Thread-Safe. Donc pour l'instant oublions là.

Les solutions

Plusieurs aspects sont à prendre en compte lors d'une programmation multithread. Le plantage que nous venons de mettre en évidence n'est qu'un aspect des choses, limité uniquement à la spécificité de Xaml qui ne prend pas en charge la mise à jour de l'UI via un thread autre que le principal. On trouvait déjà ce problème sous Windows Forms d'ailleurs.

Concernant ce problème spécifique il y a plusieurs approches, mais au final il n'y a qu'un moyen : faire en sorte que la manipulation de l'UI ne passe que par son thread propre. Ponctuellement on peut utiliser, comme sous Windows Forms, des mécanismes d'invocation transférant le contrôle au thread de l'UI. Mais ce que nous cherchons est une solution globale et "industrialisable". La plupart des auteurs qui a abordé le sujet arrive à la même conclusion, il faut passer par un mécanisme

centralisé de dispatch. Cette solution peut se mettre en œuvre de différentes façons, voyons directement celle qui m'apparait la plus simple à appliquer.

Le SmartDispatcher et la classe de base

Le principe peut se schématiser comme suit :

Figure 19 - Schéma de fonctionnement du Smart Dispatcher

Au début les choses sont, par force, identiques : la tâche de fond modifie la propriété (`Number` dans notre exemple) et déclenche `PropertyChanged` dans le Modèle. Mais au lieu que cet événement soit traité directement par le binding, nous intercalons le `Dispatcher`, c'est lui qui va vérifier que l'appel a lieu depuis le thread d'UI. Dans l'affirmative le traitement se poursuivra normalement, dans la négative, ce qui est le cas de notre exemple, il transfèrera la notification de changement de propriété au binding via une invocation appropriée. Comme le `Dispatcher` se trouve dans le thread de l'UI, et qu'il va faire en sorte de rendre "sien" les appels qui proviennent d'autres threads, le binding ne verra que des appels intra-thread UI et non inter-threads.

Cette solution fonctionne en deux étapes. Il y a le `Dispatcher`, mais il y a aussi le routage des notifications vers ce dispatcher. Modifier dans chaque classe l'appel à `PropertyChanged` pour qu'il pointe vers le `Dispatcher` serait fastidieux et source d'erreur. Le plus simple est donc de créer une classe de base dont hériterons les Modèles ou les ViewModels plus généralement (la connexion directe d'une Vue à un Modèle utilisée dans l'exemple est certes autorisé par MVVM mais ce n'est pas le "montage" le plus courant, le ViewModel étant souvent nécessaire pour la prise en charge des commandes, l'adaptation des données et la persistance de la Vue).

Le mécanisme du Binding et des patterns comme MVVM reposent sur un présupposé : que les classes reliées par binding à l'UI prennent en charge `INotifyPropertyChanged`. Il est donc raisonnable de mettre en œuvre une classe de

base implémentant cette interface et s'occupant automatiquement de la liaison avec le `Dispatcher`.

Le seul problème que peut poser cette approche est que le ViewModel soit déjà dans l'obligation d'hériter d'une classe de base. Cela peut être le cas si vous utilisez un framework MVVM ou autre. C# ne gérant pas l'héritage multiple (bien heureusement), il faudra alors fusionner la solution du Dispatcher à la classe de base déjà utilisée. A moins que celle-ci ne prenne déjà en charge un mécanisme équivalent, ce qui arrive parfois (prise en charge directe ou proposition d'un mécanisme identique).

Le code du Dispatcher

Parmi les implémentations existantes j'ai une préférence pour celle de Jeff Wilcox qui est un Senior Software Development Engineer chez Microsoft dans l'équipe de Silverlight… Autant se servir aux bonnes sources ! Son implémentation est sous licence Ms-PL (licence libre publique spécifique à Microsoft). J'avais dans mes tablettes un code assez proche mais pas aussi complet et on peut trouver des choses assez équivalentes dans le portage de Cairngorm sur Silverlight (Cairngorm sur CodePlex) , un framework MVC traduit depuis l'original de même nom conçu pour Flex de Adobe (un produit proche de Silverlight pour ceux qui ne connaissent pas).

```
 1:  using System;
 2:  using System.Windows;
 3:  using System.ComponentModel;
 4:  using System.Windows.Threading;
 5:
 6:  namespace SLMultiThread.ThreadTools
 7:  {
 8:      // (c) Copyright Microsoft Corporation.
 9:      // This source is subject to the Microsoft Public License (Ms-PL).
10:      // Please see http://go.microsoft.com/fwlink/?LinkID=131993 for details.
11:      // All other rights reserved.
12:
13:
14:          /// <summary>
15:          /// A smart dispatcher system for routing actions to the user interface
16:          /// thread.
17:          /// </summary>
18:          public static class SmartDispatcher
19:          {
20:              /// <summary>
21:              /// A single Dispatcher instance to marshall actions to the user
22:              /// interface thread.
23:              /// </summary>
24:              private static Dispatcher instance;
25:
26:              /// <summary>
27:              /// Backing field for a value indicating whether this is a design-time
28:              /// environment.
29:              /// </summary>
30:              private static bool? designer;
31:
32:              /// <summary>
33:              /// Requires an instance and attempts to find a Dispatcher if one has
34:              /// not yet been set.
35:              /// </summary>
36:              private static void requireInstance()
37:              {
38:                  if (designer == null)
39:                  {
40:                      designer = DesignerProperties.IsInDesignTool;
41:                  }
42:
43:                  // Design-time is more of a no-op, won't be able to resolve the
44:                  // dispatcher if it isn't already set in these situations.
45:                  if (designer == true)
46:                  {
```

```
47:                         return;
48:                     }
49:
50:                     // Attempt to use the RootVisual of the plugin to retrieve a
51:                     // dispatcher instance. This call will only succeed if the current
52:                     // thread is the UI thread.
53:                     try
54:                     {
55:                         instance =
                                Application.Current.RootVisual.Dispatcher;
56:                     }
57:                     catch (Exception e)
58:                     {
59:                         throw new InvalidOperationException(
                            "The first time SmartDispatcher is used must be from a user
                              interface thread. Consider having the application call
                              Initialize, with or without an instance.", e);
60:                     }
61:
62:                     if (instance == null)
63:                     {
64:                         throw new InvalidOperationException(
                                    "Unable to find a suitable Dispatcher instance.");
65:                     }
66:                 }
67:
68:                 /// <summary>
69:                 /// Initializes the SmartDispatcher system, attempting to use the
70:                 /// RootVisual of the plugin to retrieve a Dispatcher instance.
71:                 /// </summary>
72:                 public static void Initialize()
73:                 {
74:                     if (instance == null)
75:                     {
76:                         requireInstance();
77:                     }
78:                 }
79:
80:                 /// <summary>
81:                 /// Initializes the SmartDispatcher system with the dispatcher
82:                 /// instance.
83:                 /// </summary>
84:                 /// <param name="dispatcher">The dispatcher instance.</param>
85:                 public static void Initialize(Dispatcher dispatcher)
```

```
86:                 {
87:                     if (dispatcher == null)
88:                     {
89:                         throw new ArgumentNullException("dispatcher");
90:                     }
91:
92:                     instance = dispatcher;
93:
94:                     if (designer == null)
95:                     {
96:                         designer = DesignerProperties.IsInDesignTool;
97:                     }
98:                 }
99:
100:                /// <summary>
101:                ///
102:                /// </summary>
103:                /// <returns></returns>
104:                public static bool CheckAccess()
105:                {
106:                    if (instance == null)
107:                    {
108:                        requireInstance();
109:                    }
110:
111:                    return instance.CheckAccess();
112:                }
113:
114:                /// <summary>
115:                /// Executes the specified delegate asynchronously on the user interface
116:                /// thread. If the current thread is the user interface thread, the
117:                /// dispatcher if not used and the operation happens immediately.
118:                /// </summary>
119:                /// <param name="a">A delegate to a method that takes no arguments and
120:                /// does not return a value, which is either pushed onto the Dispatcher
121:                /// event queue or immediately run, depending on the current
                                                                    thread.</param>
122:                public static void BeginInvoke(Action a)
123:                {
124:                    if (instance == null)
125:                    {
126:                        requireInstance();
127:                    }
128:
```

```
129:                     // If the current thread is the user interface thread, skip the
130:                     // dispatcher and directly invoke the Action.
131:                     if (instance.CheckAccess() || designer == true)
132:                     {
133:                         a();
134:                     }
135:                     else
136:                     {
137:                         instance.BeginInvoke(a);
138:                     }
139:                 }
140:             }
141:  }
```

Code 9 - SmartDispatcher

Le SmartDispatcher est une enveloppe intelligente autour de la classe Dispatcher du Framework de Silverlight. Cette dernière a été conçue justement pour gérer le code de mise à jour de l'UI depuis d'autres threads. SmartDispatcher "habille" le Dispatcher de base en s'assurant qu'une instance est disponible. En fait, appeler le Dispatcher de SL serait suffisant, il est conçu pour cela, mais cela n'est pas possible depuis une tâche de fond, il faut donc que l'instance existe déjà. Le SmartDispatcher permet de s'assurer de cette condition liminaire et fournit un point d'accès connu et centralisé à l'instance en question.

Bien que SmartDispatcher s'occupe d'obtenir l'instance du Dispatcher automatiquement il est préférable de l'initialiser très tôt dans le code de l'application, pour s'assurer que le SmartDispatcher est bien opérationnel et disponible avant tout événement impliquant l'interface. C'est pourquoi il est préférable d'ajouter dans App.xaml le code suivant dans le constructeur de l'objet App, juste avant l'appel à `InitializeComponent()` :

```
SmartDispatcher.Initialize(Deployment.Current.Dispatcher);
```

Le SmartDispatcher est initialisé avec le Dispatcher fourni par le namespace Deployment qui s'occupe majoritairement du manifest et d'informations sur l'application en cours.

La classe de base

Il y a plusieurs façons de l'implémenter, mais puisque j'ai fait le choix de vous montrer l'implémentation de SmartDispatcher, autant utiliser la classe de base qui a été écrite par Wilcox pour fonctionner avec ce dernier, ça semble logique…

```
 1:  using System;
 2:  using System.Collections.Generic;
 3:  using System.ComponentModel;
 4:
 5:  namespace SLMultiThread.ThreadTools
 6:  {
 7:      // (c) Copyright Microsoft Corporation.
 8:      // This source is subject to the Microsoft Public License (Ms-PL).
 9:      // Please see http://go.microsoft.com/fwlink/?LinkID=131993 for details.
10:      // All other rights reserved.
11:
12:          /// <summary>
13:          /// A base class for data objects that implement the property changed
14:          /// interface, offering data binding and change notifications.
15:          /// </summary>
16:          public class PropertyChangedBase : INotifyPropertyChanged
17:          {
18:              /// <summary>
19:              /// A static set of argument instances, one per property name.
20:              /// </summary>
21:              private static readonly Dictionary<string,
                                PropertyChangedEventArgs> argumentInstances =
22:                  new Dictionary<string, PropertyChangedEventArgs>();
23:
24:              /// <summary>
25:              /// The property changed event.
26:              /// </summary>
27:              public event PropertyChangedEventHandler PropertyChanged;
28:
29:              /// <summary>
30:              /// Notify any listeners that the property value has changed.
31:              /// </summary>
32:              /// <param name="propertyName">The property name.</param>
33:              protected void NotifyPropertyChanged(string propertyName)
34:              {
35:                  if (string.IsNullOrEmpty(propertyName))
36:                  {
37:                      throw new ArgumentException(
                                        "PropertyName cannot be empty or null.");
38:                  }
39:
40:                  var handler = PropertyChanged;
41:                  if (handler == null) return;
42:                  PropertyChangedEventArgs args;
```

```
43:                     if (!argumentInstances.TryGetValue(propertyName, out
args))
44:                     {
45:                         args = new PropertyChangedEventArgs(propertyName);
46:                         argumentInstances[propertyName] = args;
47:                     }
48:
49:                     // Fire the change event. The smart dispatcher will directly
50:                     // invoke the handler if this change happened on the UI thread,
51:                     // otherwise it is sent to the proper dispatcher.
52:                     SmartDispatcher.BeginInvoke(
                                            () => handler(this, args));
53:                 }
54:             }
55:
56:  }
```

Code 10 - PropertyChangeBase

Cette classe implémente `INotifyPropertyChanged` qui est à la base du mécanisme de propagation “automatique” des changements de valeurs utilisé par le binding et sur lequel repose des patterns comme MVVM.

La classe utilise une astuce pour améliorer les performances : elle créée un dictionnaire des `PropertyChangedEventArgs` de telle façon à ce que ce soit toujours la même instance d’arguments qui soit utilisée pour une propriété donnée. Ainsi, et surtout lorsque les valeurs changent très souvent, le Garbage Collector n’est pas submergé de petits objets à durée de vie ultra-courte. Il est certes optimisé pour cela, mais moins il y a de créations / destructions moins le GC a de travail et tout va plus vite.

Une autre optimisation est importante, elle concerne la détection de la nécessité ou non de transiter via le Dispatcher. Dans le cas où cela n’est pas nécessaire le message de notification suit son cours. Il est ainsi traité immédiatement alors que le passage via le Dispatcher créé un délai (même infime).

Le code exemple

Le projet complet (VS2008 SL3) est téléchargeable ici: SLMultiThread.zip (9,55 kb)

Les autres facettes du multithread

Ici nous n'avons fait qu'effleurer le sujet en abordant un problème très spécifique du multithreading sous Silverlight. Si vous regardez le code de plus près, notamment la classe RandomNumberPusher que nous n'avons pas étudiée dans ce billet, vous y trouverez des techniques simples propres au multithreading comme l'utilisation d'un objet "locker" pour assurer le verrouillage des lectures et écritures entre les threads.

« Il neige » ou les boucles d'animation, les fps et les sprites sous Silverlight

Il n'est jamais trop tard pour préparer Noël, quel que soit le moment de l'année on finit toujours par se diriger vers un Noël ! Evoquer la neige en hiver est de bon ton et le faire l'été est rafraichissant. Sous ce prétexte glacé (mais pas glaçant) se cache des notions techniques très utiles sous Silverlight (ou WPF).

Les boucles d'animation, le contrôle des fps (frame per second, images par seconde) et les sprites (petits objets autonomes animés) sont certes autant d'incursions dans le monde du jeu mais la créativité permet d'envisager mille utilisations de ces techniques dans d'autres contextes.
Je me suis librement inspiré d'un billet s'intitulant « Making it snow in Silverlight » écrit par Mike … Snow (facile à se souvenir !). Heureusement que Dot.Blog est une sorte de mémoire du Web XAML car par exemple cet article, son site et son auteur ne sont plus sur le Web… Reste de ce dernier un livre référencé par Amazon US sur la programmation de jeu avec Silverlight. J'utilise de toute façon un code différent et (à mon sens) amélioré, mais il était important de rendre à Snow la neige qui lui appartient (et de faire un peu d'humour bas de gamme au passage).

Avant d'expliquer certains aspects, voici une démonstration live que vous pouvez passer en mode plein écran (cliquez sur le titre du chapitre du livre pour vous rendre sur Dot.Blog où vous pourrez jouer avec l'exemple live) :

Figure 20 - Il Neige !

Le but

Le but du jeu est donc de faire tomber de la neige sur un fond, c'est-à-dire de faire se mouvoir des centaines d'objets autonomes ayant chacun un aspect légèrement différent et une trajectoire un peu aléatoire. Le tout assez vite pour que cela paraisse naturel.

Le principe

Chaque flocon de neige est en réalité une instance d'un `UserControl`, il y a ainsi des centaines d'instances au même instant. Chaque flocon est responsable de son affichage et de la façon dont il avance à chaque « pas ». Des acteurs indépendants, nous sommes en pleine programmation objet ! La coordination générale étant réalisée par une boucle d'animation principale, c'est elle qui scande le rythme et demande à chaque flocon d'avancer d'un « pas ».
Tout se trouve donc dans deux endroits stratégiques : la boucle principale et le `UserControl` « `Flocon` ».

> *Je vous conseille de télécharger le projet complet (voir en fin de billet) pour mieux suivre les explications.*

Le sprite Flocon

Il s'agit d'un `UserControl` possédant une interface très simple puisqu'il s'agit d'un bitmap en forme d'étoile (ou pourrait utiliser n'importe quelle forme vectorielle créée sous Design, Illustrator ou InkScape). Côté XAML on retrouve ainsi une grille avec un PNG à l'intérieur, rien de palpitant. A noter qu'au niveau du `UserControl` lui-même on définit deux transformations : une d'échelle (`FlakeScale`) et une de rotation (`FlakeRotation`). Dans l'exemple de la neige cette dernière n'a que peu d'effet visuel sauf lorsqu'on choisit une taille assez grosse pour les flocons. L'œil ne discerne pas forcément la rotation de façon consciente mais comme elle est choisie aléatoirement pour chaque flocon cela renforce l'aspect global plus « organique ». De plus il s'agit d'illustrer des techniques utilisables dans d'autres contextes ou avec d'autres sprites pour lesquels l'effet serait plus évident. La technique est extrapolable à toutes les transformations supportées, même les transformations 2,5D de Silverlight.
Le code de la classe `Flocon` se divise en deux points : le constructeur et la méthode `Tombe`.

Le constructeur

Il est responsable de la création de chaque instance. C'est à ce niveau qu'on introduit un peu d'aléatoire dans la rotation du flocon.

```
public Flocon()
        {
            InitializeComponent();

            // position initiale
            SetValue(Canvas.TopProperty, 0d);
            SetValue(Canvas.LeftProperty, (double)random.Next(BoardWidth));

            // échelle initiale
            FlakeScale.ScaleX =
                 FlakeScale.ScaleY = scale * random.Next(1, 3);
            // rotation initiale
            FlakeRotation.Angle = random.Next(1, 180);
        }
```

Code 11 - Le constructeur de la classe Flocon

On choisit aussi à ce moment la position de départ en X (la position en Y étant toujours à zéro au départ).
La gestion des flocons est séquentielle (par la boucle d'animation que nous verrons plus loin). De fait il n'y a pas de multithreading et on peut se permettre un développement très simple de la classe. Par exemple les limites du tableau de jeu sont stockées dans des propriétés statiques de la classe. Le programme initialise ces valeurs au lancement et les modifie à chaque fois que la taille du tableau est changée (redimensionnement du browser).
Le constructeur fixe aussi de façon aléatoire l'échelle de l'instance afin que chaque flocon soit de taille légèrement différente par rapport à une échelle de base. Cela permet d'obtenir un effet de profondeur de champ.

La méthode « Tombe »

Il s'agit bien de tomber et non d'outre-tombe et de zombies, quoi qu'on aurait pu faire tomber des crucifix sur un fond de cimetière, chacun pouvant adapter selon ses goûts… Pour ma part je resterai dans la poésie d'une chute de neige. Pour chaque flocon il s'agit donc ici de savoir choir, avec si possible le sens du rythme et du mouvement pour plus d'élégance et de vraisemblance.

Pour ce faire la méthode « `Tombe` » doit calculer la nouvelle position `X,Y` du flocon.

Les flocons évoluent dans un espace de type `Canvas` particulièrement indiqué pour tout ce qui doit être positionné de cette façon à l'écran.

> *On notera que le Canvas est déconseillé pour toute autre forme de mise en page car fonctionnant au pixel près il est moins adapté aux mises en page dynamiques. Toutefois pour une surface de jeu où les objets sont placés et se déplacent au pixel près ce conteneur est, au contraire, un allié fort pratique.*

Sur l'axe des `Y` le déplacement s'effectue d'un pixel à la fois. Une incrémentation simple est utilisée. Sur l'axe horizontal le flocon change régulièrement de direction pour rendre l'ensemble visuel plus réaliste. On utilise ici un tirage aléatoire pour savoir s'il faut ou non changer de direction. Dans l'affirmative on modifie la valeur du pas de déplacement (`hzDeplacement`). La position sur X est alors calculée en ajoutant cette valeur à la position courante.

```
public void Tombe()
        {
            // un peu de mystère
            var mereNature = random.Next(40);

            if (mereNature == 1)
            {
                // changement de direction
                if (hzDeplacement >= 0.25 || hzDeplacement <= -0.25)
                    hzDeplacement = 0d;
                else
                {
                    var r = random.Next(3);
                    if (r == 1) hzDeplacement = 0.25;
                    else
                        if (r == 2) hzDeplacement = -0.25;
                        else
                            hzDeplacement = 0d;
                }
            }
            var x = (double)GetValue(Canvas.LeftProperty);
            var y = (double)GetValue(Canvas.TopProperty);
            y++;
            x += hzDeplacement;
            SetValue(Canvas.LeftProperty, x);
            SetValue(Canvas.TopProperty, y);

            HorsZone = (x < 0 || x > BoardWidth || y > BoardHeight);
        }
```

Code 12 - Méthode d'animation des flocons

Reste à savoir quand un flocon n'a plus d'intérêt, c'est-à-dire lorsqu'il échappe au champ de vision et qu'il peut être détruit (n'oublions pas que des centaines d'instances sont gérées à la fois et que tout flocon inutile doit être détruit le plus vite possible pour d'évidentes raison d'efficacité et de consommation mémoire). C'est le rôle du test en fin de méthode qui initialise la valeur « `HorsZone` ». Lorsqu'elle passe à true, le flocon peut être détruit. Cette information est utilisée par la boucle principale.

La boucle d'animation

Maintenant que nous disposons d'un modèle de flocon (la classe `Flocon`) reste à animer de nombreuses instances pour donner l'illusion de la neige qui tombe.

La technique pourrait utiliser un `Timer` ou bien une `StoryBoard` vide dont le déclenchement est réalisé à chaque pas (et comme elle s'arrête immédiatement on obtient une sorte de `Timer` rythmant la boucle principale). Cette dernière façon de faire était très utilisée sous Silverlight 1. Selon la vitesse de la machine cliente la boucle est ainsi déclenchée plus ou moins vite ce qui évite le problème de fixer un pas déterminé comme avec un `Timer`. Toutefois il est en réalité inutile de calculer la position des flocons plus rapidement que les FPS, pure perte d'énergie, ni de calculer moins souvent (autant abaisser les FPS).

Sous Silverlight 2 un événement a été ajouté donnant accès au rythme du moteur de rendu : `CompositionTarget.Rendering`. En gérant la boucle principale dans ce dernier on s'assure d'être parfaitement dans le tempo du moteur de rendu, ni plus rapide, ni plus lent.

Cette technique a toutefois dans le principe un défaut connu dans tous les jeux programmés de la sorte : la célérité de l'ensemble dépend de la vitesse de la machine hôte. Un PC très lent fera donc tomber la neige trop lentement, alors qu'un PC fulgurant des années 2020 ne permettra même plus de distinguer le déplacement des flocons tellement cela sera rapide, rendant le jeu « injouable ». Pour le premier problème il n'y a pas vraiment de solution, on ne peut donner plus puissance à la machine hôte (sauf à permettre à l'utilisateur de diminuer la charge du jeu lui-même en limitant par exemple le nombre maximum de flocons, le paramètre de densité peut jouer ce rôle dans notre exemple). Quant au second problème il trouve sa réponde dans la limite à 60 FPS par défaut de Silverlight, limite modifiable via les paramètres du plugin (voir ce billet).

Débarrassés des écueils propres à cette technique, l'événement de rendering devient l'endroit privilégié pour placer le code d'une boucle principale d'animation.

Le contrôle des FPS

Lorsque vous exécutez une application de ce type il est important de contrôler les FPS pour voir comment certaines améliorations du code peuvent accélérer ou non le rendu. Pour ce faire Silverlight permet d'activer l'affichage de deux informations essentielles dans la barre d'état du browser : les FPS actuels et la limite fixée dans le plugin. L'activation se fait via `Application.Current.Host.Settings` en plaçant `EnableFrameRateCounter` à true. Il est bien entendu vivement conseillé de déconnecter la fonction en mode Release. Si votre navigateur et ses réglages le supporte vous devez en ce moment pouvoir lire ces informations en bas à gauche de la barre d'état (et si l'exemple live de l'application en début de billet s'est chargé correctement).

> *Attention : Selon les réglages de sécurité Internet Explorer autant que FirexFox peuvent ne pas autoriser la modification de la barre d'état. Sous IE (sous FF cela est quasi identique) il faut vérifier le paramètre Outils / Options Internet / Sécurité / Zone Internet / Personnaliser le niveau / Autoriser les mises à jour à la barre d'état via le script. Ce paramètre doit être activé pour voir les FPS.*

Le Rendering

A chaque pas du rendering nous allons réaliser deux tâches essentielles : déplacer les flocons déjà présents et supprimer ceux devenus inutiles et créer de nouveaux flocons.

Pour déplacer les flocons nous ne faisons que balayer la liste de tous les flocons et appeler la méthode « `Tombe()` » de chacun. Au passage nous archivons dans une liste temporaire tous les flocons dont l'état `HorsZone` est passé à true. En fin d'animation les flocons inutiles sont supprimés de la liste principale les laissant ainsi disponible au Garbage Collector.

Pour créer de nouveaux flocons nous déterminons si la densité désirée est atteinte ou non. Le processus est randomisé pour plus de réalisme bien que le seuil de densité soit lui fixé par le programme (et modifiable dans notre exemple). Comme le placement de chaque flocon est réalisé dans le constructeur de la classe `Flocon` il n'y a rien d'autre à faire que de créer les flocons et de les ajouter à la collection `Children` du `Canvas` parent.

Le plein écran

Silverlight permet de passer une application en mode plein écran. Sachez toutefois que ce mode pose de sérieuses conditions à son utilisation. Notamment il ne peut être activé qu'en réponse directe à un clic de l'utilisateur. Cela permet d'éviter (en partie) qu'une application Silverlight soit utilisée de façon à tromper l'utilisateur final en prenant le contrôle de tout l'écran sans son autorisation (on pourrait penser à un fake du bureau Windows permettant par exemple d'extorquer un login). De même, lorsque l'application bascule en mode plein écran un message impossible à supprimer est affiché pour avertir l'utilisateur et lui indiquer que l'appui sur Escape permet de sortir de ce mode spécial.

Si cela n'était pas suffisant, les saisies clavier sont impossibles en plein écran (ce qui a été modifié dans les dernières versions de SL sous certaines conditions). Seules certaines touches ont un effet (voir la documentation sur MSDN). En revanche pour visionner des vidéos ou pour créer un jeu les touches disponibles et la gestion de la souris sont suffisants. C'est le cas de notre démo.

Une fois connues ces limites raisonnables (Microsoft ne veut pas, à juste titre, que Silverlight soit utilisé par des fraudeurs et autres adeptes du phishing) le passage en plein écran s'effectue par `Application.Current.Host.Content.IsFullScreen` en mettant cette propriété à `true`, et ce en réponse au clic sur un bouton en général.

Le code

Télécharger le projet complet de l'exemple "IINeige" : IINeige.zip (229,54 kb)

Happy Holidays !

Patientez quelques instants que le téléchargement de la petite application Silverlight ci-dessous se termine... (Enfin pas dans ce livre... cliquez sur le titre ci-dessus pour accéder au billet original sur Dot.Blog et accéder à l'exemple live).

Ensuite passez la souris sur ma tête pour lire le message !

Figure 21 - Happy Holidays !

Un peu de technique

Il faut bien que cela soit l'excuse de parler technique, au moins un peu...

Donc cette petite carte de vœux a été créée en utilisant l'exemple "Il neige" largement commenté plus haut. La transformation majeure est bien entendu de faire tomber des cadeaux et ma trombine de temps en temps à la place de certains flocons de neige. La possibilité d'interagir avec la dite trombine complète ce tableau de Noël.

Pour aller vite j'avoue que j'étais parti pour faire dériver les nouveaux sprites de la classe "`Flocon`" existante dont les instances sont multipliées pour créer l'effet de neige dans cette petite gestion de particules.

Pas d'héritage de UserControl

Trois fois hélas, créer un UserControl en héritant d'un autre n'est pas impossible sous Silverlight mais c'est un peu sportif... Pour éviter d'entrer dans les manipulations nombreuses et hasardeuses permettant de contourner ce problème j'ai utilisé la solution des interfaces ce qui a donné un charme spécial à cet exemple et a créé u réel intérêt qui le différencie de l'article « Il Neige ! ».

J'ai donc créé une interface `ISprite` qui reprend les deux choses importantes pour la boucle d'animation : l'objet est-il passé hors zone (il doit alors être détruit) et l'action "`Tombe`" qui est appelée à chaque pas d'animation.

Le flocon original a été modifié pour supporter l'interface (ce qu'il faisant déjà avant) et la boucle d'animation (utilisant l'event de rendering de SL) a été modifiée pour utiliser des `ISprite` au lieu des `Flocon` originaux.

Ne restait plus qu'à créer autant de `UserControl` que de types de cadeaux qui peuvent s'afficher (un rouge, un vert, un bleu, plus ma bobine affublée d'une barbe rapidement dessinée par dessus). Chaque nouveau `UserControl` supporte ainsi `ISprite`. L'application fait des tirages au sort pour attribuer 25% de chances d'apparaitre à chacun des 4 sprites.

Améliorations diverses

Le reste est conforme à l'application "Il neige", à la création de chaque sprite celui-ci détermine une rotation et une taille aléatoire. Pour ma tête j'ai modifié la séquence pour que l'angle de rotation soit compris entre -45 et +45 degrés, de telle façon à ce que je ne me retrouve pas tête en bas ! Ce sprite a en outre été complété d'un phylactère annonçant "Joyeuses Fêtes" dont l'apparition et la disparition sont gérés via deux behaviors de type "`GotoState`". Lorsque la souris entre ou sort du contrôle ces derniers changent l'état du sprite et le message devient visible ou se cache avec une transition automatique gérée par le Visual State Manager.

Comme vous l'avez constaté lorsque la souris entre dans l'application une guirlande clignotante apparaît. Il s'agit d'une image vectorielle Illustrator passée (par simple copier / coller) vers Expression Design puis ensuite passée de la même façon sous Blend à l'intérieur d'un nouveau `UserControl`. L'effet de chenillard est une animation réalisée "à la main" en mode "forever" en jouant sur l'opacité des bulbes et de leur reflet. Le contrôle est caché pour la souris ce qui fait qu'il est possible de cliquer sur ma tête "à travers" la guirlande bien qu'elle soit en premier plan.

Image de fond

Le logo E-naxos en 3D a été réalisé sous Swift 3D Version 6. Il a été transformé en Jpeg, car même si SL sait gérer la 3D la lourdeur d'une telle gestion ne s'imposait vraiment pas ici. C'est un fond statique et une image fixe est parfaite pour cela. En revanche un outil de 3D est nécessaire pour créer l'image !

Conclusion

Une façon ludique mais instructive de vous souhaiter à tous d'excellentes fêtes de fin d'années ... Quelle que soit l'année !

Le code complet : IlNeige2010.rar (287,17 kb)

Voici le projet compressé en mode Zip, plus classique que le format RAR ci-dessus : IlNeige2010.zip (290,57 kb)

Convertir une appli Flash en sources Silverlight ou WinRT!

J'avais découvert un site qui était une mine d'or de petits freewares de type utilitaires, ce site existe toujours et a évolué avec le temps. Un truc assez sympa qui pourra certainement vous intéresser : un **convertisseur d'applis Flash en code source Silverlight ou WinRT prêt à être compilé** !

SilverX est capable de charger un SWF et de créer une appli SL en récupérant les dessins vectoriels, les remplissages, les gradients, les bitmaps, les textes, les fonts, le clipping, les sons, etc, et même certaines animations (show/hide, matrices de transformation, changement de couleur...).

Un moyen simple, si ce n'est une application complexe, au moins de récupérer un bandeau de pub SWF et de le transformer en Silverlight ou en WinRT afin d'avoir un site tout en XAML sans petits bouts de Flash qui traînent de ci de là ! :-)

Ma première bonne résolution de 2010 était : Mettre du Silverlight PARTOUT !
En acceptant les aléas et autres erreurs de Microsoft, transformons pour 2014 ce vœu en « Mettre du XAML partout ! ».

De mon côté je vous assure que c'est une résolution que je vais tenir (ça sera peut-être la seule parmi les sempiternelles promesses qu'on se fait à soi-même et aux autres !)

Blocage IE avec écran multitouche sous Silverlight

Voici un cas intéressant pour lequel j'ai trouvé la solution qui surement vous intéressera...

Les écrans multitouche ne sont pas encore légion sur PC, sauf chez quelques geeks, dont je suis (cet article date d'avant la mode des tablettes et de Windows 8). Ainsi, pour tester dans un mode proche de la réalité les applications Phone 7 en cours de développement (et faute de téléphones de ce type disponible sur le marché au moment où j'écrivais ces lignes) j'ai acquis un écran 22 pouces IIYama multitouch T2250MTS. C'est très sympa, pas très cher et ça marche vraiment bien de base avec Windows 7 sans installer aucun driver (c'est à dire que Windows 7 possède déjà le fameux driver et que brancher l'USB de l'écran sur le PC suffit sans aucun pilote extérieur ni CD d'installation, c'est beau le progrès !). Cet écran fonctionne d'ailleurs toujours très bien, même sous Windows 8.x et le fait qu'il ne gère que 2 points est suffisant dans la vaste majorité des cas d'utilisation.

L'écran est même fourni avec un stylet qui se cache dans le bas de son boitier, ce qui évite les traces de gros doigts graisseux sur l'écran ! Et je me félicite d'être si précis dans mes billets car quelques années après j'avais totalement oublié ce stylet bien caché dont je redécouvre l'existence en corrigeant ce billet pour la version Livre PDF sur XAML !

Bref, tout cela fonctionne à merveille et du premier coup. Quand on arrive sur des zones de saisie et qu'on touche l'écran il y a même un petit symbole de clavier qui apparait pour permettre d'ouvrir un grand clavier virtuel, voire le module de reconnaissance d'écriture. On voit que Windows 7 a été conçu pour investir les Tablet PC (ce qui se fera plus tard sur les vraies tablettes avec Windows 8 mais l'idée était déjà là)… Vraiment cool. Seul bémol, sur un écran vertical manipuler les applis avec les doigts à bout de bras c'est vite crevant (j'ai testé l'impossibilité du tout tactile sur PC bien avant la sortie de Windows 8 c'est pourquoi je sais de quoi je parle aujourd'hui et que mes positions sur la question ne sont pas des a priori mais bien issu d'une longue expérience de ce type d'interface)… Mais ce n'est pas pour cela que j'avais acheté l'écran, c'était pour utiliser l'émulateur de Phone 7 en mode multitouch (car tester une appli pour téléphone touch avec la souris c'est le meilleur moyen de foirer le design, les doigts sont plus gros et moins précis, il faut y penser à la conception !). Il s'est avéré avec le recul que cela ne m'a pas vraiment servi… Ce n'était pas si pratique que ça. Pour tester une App mobile il faut une vraie machine, rien ne peut remplacer cette dernière, mais c'est une autre histoire.

Vous allez me dire c'est quoi le rapport avec Silverlight et Internet Explorer ?

C'est tout simple.

Plantage avec Silverlight ?

Silverlight gère le multitouch. Au moment où internet exploreur se lance et affiche l'application quand je suis en debug (sous VS ou Blend) je ne sais pas trop quel niveau de sécurité IE reconnaît (en tout cas pas Intranet local car j'ai abaissé toutes les sécurités à ce niveau et cela n'a rien changé) mais à l'activation du plugin *Silverlight IE s'arrête, comme planté*.

Impossible de s'en sortir. Il faut aller le shooter depuis les processus de Windows ou bien stopper le debug sous VS ce qui a le même effet.

Sécurité cachée !

Et là l'œil averti du chasseur de bug voit passer à toute vitesse comme une petite fenêtre cachée sous IE qui, hélas puisqu'on vient de flinguer IE, disparait sans qu'on puisse la lire…

Il y a donc une question qui est posée, en modal, ce qui bloque IE. Il "suffit" donc d'accéder à ce dialogue.

Je n'ai pas réussi. Et si on minimise IE pour que ça ne soit plus qu'un petit carré minuscule, on ne voit pas derrière la fenêtre du dialogue… Elle ne vient donc que quand IE est shooté. Bug…

Problème numéro 1: Identifier cette fichue fenêtre.

Ça passe tellement vite que c'est illisible bien entendu.

Ruse : Utiliser l'enregistreur d'écran livré avec Expression Encoder 3 et enregistrer la vidéo du bug pour faire ensuite un arrêt sur image et lire le texte du dialogue…

Ça ne marche pas au premier essai car ça va vraiment vite. Il faut ruser encore plus en modifiant les réglages d'enregistrement de la vidéo, notamment en passant à 100 images secondes et un débit d'au moins 20.000/s.

Là, on finit par choper sur la vidéo un "fantôme" du dialogue plus ou moins opaque selon la chance qu'on a. Mais on l'a eu ! Et que lit-on ?

"Un site Web veut ouvrir un contenu Web en utilisant ce programme sur votre ordinateur" et en dessous : Nom: **Composant de saisie tactile ou avec sty…**, Editeur: Microsoft Windows.

Le nom du composant n'est pas complet mais on a compris de ce quoi il s'agit. Silverlight, le plugin, au moment du chargement (car la première instruction de l'appli n'est pas encore atteinte, ce que prouve le debugger) doit forcément inspecter la machine pour savoir s'il y a de la vidéo, du son, et tous les périphériques ou facilités utilisées par le plugin. Parmi celles-ci se trouve vraisemblablement le multitouch et la gestion du stylet.

C'est à ce moment que le fameux dialogue est activé pour me demander si je donne le droit ou non à ce module de s'activer. Normalement on dispose sur ce genre de dialogue d'une case à cocher permettant d'indiquer si on souhaite ne plus voir la question.

Hélas, comme IE 8 semble avoir un léger bug, ce fichu dialogue est totalement incessible, on le voit à peine quand on shoote l'application. Et encore faut-il la ruse de la vidéo pour lire de quoi il s'agit.

Comment autoriser l'application bloquante ?

Second problème alors : comment autoriser le programme en question (la gestion du stylet) alors même que le dialogue et sa case à cocher ne sont pas accessibles ?

Réponse : en plongeant dans les arcanes de la gestion de sécurité de IE …

Et là, ce n'est pas de la tarte… J'aime bien IE, mais ce n'est pas d'une grande limpidité dès qu'on aborde ce genre de question.

Je vous fais grâce des multiples recherches sur le Web pour arriver à trouver une réponse utilisable. Que je vais maintenant vous exposer car ça peut servir ! (et pas seulement avec un écran multitouch, je suppose que le bug du dialogue doit se voir dans d'autres cas).

Pour faire simple, IE range dans la registry la liste des programmes ayant des droits faibles ainsi que la règle d'élévation de droit qui doit être appliquée. Comment certaines applications arrivent dans cette première liste, c'est un mystère que je n'ai pas essayé d'éclaircir, ce genre de plomberie en informatique me donnant rapidement la nausée. Donc, il existe dans la registry une première liste, elle se trouve là :

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Low Rights\ElevationPolicy\
```

C'est une liste d'ID, donc de GUID. Il faut la balayer pour lire les clés qui correspondent afin de localiser le programme dont on veut changer les droits. (La registry doit être ouverte en mode administrateur cela va sans dire).

Pour ce qui est du stylet, j'ai fini par trouver le coupable, c'est le programme "`wisptis.exe`" qui se trouve dans `%SystemRoot%\System32`. Les clés d'un ID sont `AppName`, le nom de l'appli, ici celui de l'exe (ce qui ne simplifie pas la recherche vu à quel point le nom n'est pas parlant), `AppPath`, le chemin que je viens d'indiquer et enfin `Policy`, un DWord codant l'autorisation.

Vous trouverez des explications ici : http://www.hotline-pc.org/mode-protege.html#

Une fois l'application localisée dans cette première liste on a la moitié de la solution. Changer la `Policy` à ce niveau ne semble pas avoir d'effet (immédiat en tout cas).

Il faut savoir qu'en réalité, pour l'utilisateur courant, IE stocke une seconde liste, similaire, dans une autre clé de la registry :

```
HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy
```

L'astuce consiste donc à créer une nouvelle clé dans cette liste en utilisant le GUID repéré dans la première liste puis de recréer à la main les clés, dont `Policy` à laquelle on donne une valeur dépendant de ce qu'on cherche à obtenir. Pour la gestion du stylet j'ai mis 3, qui semble donner le maximum de droits.

Je reviens sous VS, je lance mon appli SL, et là, Ô magie… mon appli s'affiche et IE ne pose plus l'infamante question cachée…

Et ça marche !

Problème résolu. Comme on ne peut pas dire que la solution est évidente, je pense que la partager ici sera un jour utile à quelqu'un.

Visiteur qui tombera au hasard d'une recherche Bing ou Google sur cette page et qui gagnera quelques heures de ton précieux temps, n'hésite pas à me laisser un message, ça fait toujours plaisir ! Quant aux fidèles lecteurs de Dot.Blog, le jour "J" j'espère que vous vous rappellerez que ce billet existe !

Et pour d'autres infos le mieux c'est : Stay Tuned !

Une dernière info : Le multitouch de mon écran utilise une barrière infrarouge, c'est à dire que les doigts ou le stylet coupe une grille infrarouge invisible. Ça marche vraiment bien. Sauf qu'en été les mouches arrivent ! Et dans ma campagne elles ne sont pas en retard ! Hélas quand une mouche se pose sur l'écran : elle clique ! Ce qui fiche un beau bazar parfois ! Prévoyez ainsi avec l'achat d'un écran de ce type un désinsecteur électrique qui grillera tous ces nuisibles qui prennent votre écran pour un tarmarc ! Je sais ça fait un peu rire comme truc... mais le geek n'a pas que des aventures palpitantes, il doit aussi combattre, outre les bugs, les mouches et autres diptères !

Gérer des paramètres personnalisés du plugin Silverlight

Une application Silverlight est hautement configurable et de nombreuses stratégies existent pour rediriger un utilisateur vers une page ou une autre, notamment le système de Navigation introduit dans SL3. Mais il est aussi possible de configurer le comportement d'une application depuis la balise qui déclare le Plugin, offrant tout un panel d'autres possibilités…

Exemple : gadgets et menus

Pour rendre les choses concrètes supposons un gadget écrit en Silverlight ou plus simplement un menu animé qui sera insérer dans des pages HTML ou ASPX. Selon la page où il apparaît, ce menu peut certainement avoir des comportements différents. On peut souhaiter un mode "novice" affichant l'essentiel sur une page d'accueil et un mode "expert" affichant plus de choix pour les pages intérieures par exemple.

Bref, on doit pouvoir passer à l'application des paramètres personnalisés à l'initialisation, c'est à dire depuis la balise du plugin.

Déclarer le paramètre

Qui de la poule ou de l'œuf… avant de déclarer le paramètre encore faudrait-il lui donner un nom et en fixer les valeurs possibles.

Déclaration dans le code C#

Dans notre exemple l'application Silverlight peut fonctionner en mode *Novice* ou en mode *Expert* (rien n'interdit que cela ne change au runtime, nous parlons bien ici d'un paramètre d'initialisation et non d'un choix fermé et définitif).

Pour simplifier et rendre le paramètre et sa valeur disponible dans toute l'application, c'est bien entendu l'objet `App` dérivant de `Application` qui sera choisi. On ajoute ainsi à `App.xaml.cs` les définitions suivantes :

```
1:  public enum DisplayMode
2:  { Novice, Expert }
```

Ainsi que :

```
1:  private DisplayMode displayMode = DisplayMode.Novice;
2:
3:  public DisplayMode DisplayMode
4:  { get { return displayMode; } }
```

Code 13 - gestion du paramètre d'initialisation

Déclaration dans le code Html

Pour un petit projet exemple de ce type je choisi toujours une application Silverlight simple sans site Web. C'est à dire que VS génère pour nous "`TestPage.Hml`" qui active le plugin et l'application. Cette page est placée dans `\bin\debug` (ou *release* mais il est rare de travailler autrement qu'en mode *debug* lors de tests de ce type).

Le plus simple consiste ainsi à générer une première fois l'application puis à se placer dans `\bin\debug` et à copier `TestPage.Html` en changeant son nom, ici "`ExpertView.Html`". Ensuite on ouvre cette nouvelle page HTML et on peut modifier la balise du plugin pour y ajouter notre (ou nos) paramètre(s) :

Dans la balise `object`, à la suite de toutes les autres sous balises "`<param>`" nous ajoutons :

```
<param name="initParams" value="DisplayMode=Expert" />
```

XAML 4 - Initialisation du paramètre dans la balise HTML du plugin

Choix du lancement de l'application

Bien entendu ce montage est rudimentaire. Lorsque vous exécutez (F5) l'application, c'est `TestPage.Html` qui est lancé et qui ne contient pas le nouveau paramètre. Cela

permettra de tester la gestion de la valeur par défaut du paramètre et de s'assurer que tout marche même si le paramètre est omis.

Pour tester la page en mode "*expert*", après une construction du projet, ouvrez le répertoire `bin\debug` et lancez à la main "`ExpertView.html`", l'application sera en mode expert.

Mais… mais… oui ! Ne vous inquiétez pas, il en manque un bout :-) j'y viens…

La prise en charge du paramètre

Nous disposons d'un paramètre déclaré dans `App.xaml.cs` donc visible dans toute l'application. Il est basé sur une énumération précisant les valeurs possibles. Une propriété a été ajoutée à `App` afin de stocker et rendre disponible le paramètre pour toutes les pages. Cette propriété est basée sur un backing field initialisé à "`Novice`".

Nous avons aussi deux pages Html, l'une qui ne contient aucun réglage particulier (et qui devrait déboucher sur un fonctionnement en mode par défaut, donc `Novice`) et une autre dans laquelle nous avons ajouté le paramètre en mode "`Expert`".

C'est bien joli, mais pour l'instant le paramètre n'est pas interprété et son effet ne sera visible nulle part. Nous allons y remédier maintenant :

Interpréter le paramètre

Nous devons maintenant récupérer le paramètre que le plugin Silverlight aura passé à notre application. La méthode `Application_Statup` de l'objet `App` semble tout à fait correspondre au besoin.

Voici le code :

```
 1:  private void Application_Startup(object sender, StartupEventArgs e)
 2:  {
 3:    // Récupération du paramètre plugin
 4:    if (e.InitParams.ContainsKey("DisplayMode"))
 5:     {
 6:      try
 7:        {
 8:           displayMode =
 9:             (DisplayMode)Enum.Parse(typeof(DisplayMode),
10:                                e.InitParams["DisplayMode"],
11:                                true);
12:         }
13:         catch { }
14:      }
15:     // Create the root page.
16:     RootVisual = new MainPage();
17:  }
```

Code 14 - Récupération de la valeur d'initialisation du plugin

L'événement `StartUp` propose des arguments parmi lesquels les fameux paramètres. Il suffit donc de récupérer par `InitParams` le paramètre qui nous intéresse (`DisplayMode`). Dans la foulée nous le transformons en valeur de l'énumération `DisplayMode`. En cas de problème (paramètre non reconnu) la valeur restera celle par défaut. D'où l'importance de fixer une valeur par défaut et de faire en sorte qu'elle soit toujours celle qui offre le moins "de risque" pour l'application. Ici, le mode "`Novice`" est bien le plus simple, celui qui offre le moins de possibilité et qui ainsi protège le mieux l'application.

Voilà… le paramètre est récupéré, la propriété `DisplayMode` de `App` est disponible pour toutes les pages de l'application.

Utiliser le paramètre

Reste à utiliser le paramètre quelque part pour se prouver que tout ce montage fonctionne bien.

Le plus simple ici consiste à faire afficher à la page principale le nom du mode dans lequel on se trouve. `MainPage.xaml` contient ainsi un `TextBlock` qui sera initialisé dans le `Loaded` de la page :

```
 1:  public MainPage()
 2:  {
 3:   InitializeComponent();
 4:   Loaded += Page_Loaded;
 5:  }
 6:
 7:  void Page_Loaded(object sender, RoutedEventArgs e)
 8:  {
 9:    txtDsiplayMode.Text =
10:          "Mode d'affichage : " +
11:          ((App)Application.Current).DisplayMode;
12:  }
```

Code 15 - Adapter le fonctionnement de l'application en fonction de la valeur passée

On remarque que nous sommes obligés de transtyper `Application.Current` en `(App)`, sinon nous ne verrions pas la propriété ajoutée, qui n'existe bien entendu pas dans la classe `Application` parente.

Conclusion

Je vous fais grâce d'une capture écran, selon qu'on exécute `TestPage.html` ou `ExpertView.html` on obtient l'affiche du mode "`Novice`" ou "`Expert`".

La gestion des paramètres personnalisés est donc très simple et permet de rendre une application Silverlight plus versatile et plus facile à intégrer dans une application Web classique. Il y a mille utilisations possibles, à vous de les inventer selon vos besoins !

Isolated Storage et stockage des préférences utilisateur

L'Isolated Storage (ou stockage isolé) est une zone réservée à une application Silverlight sur la machine cliente, on retrouve une logique similaire dans les environnements protégés de type Windows Phone ou WinRT. Les quotas sont réglés par l'utilisateur ou peuvent être modifiés par l'application sous réserve d'acceptation par l'utilisateur. Cette zone est particulièrement bien adaptée au stockage des préférences de l'application ou de l'utilisateur. Malgré les apparences, au moins dans ce cadre, la manipulation de l'Isolated Storage est enfantine…

Démonstration

(Cliquez sur le titre du chapitre pour tester l'application live directement le site Dot.Blog dans le billet original)

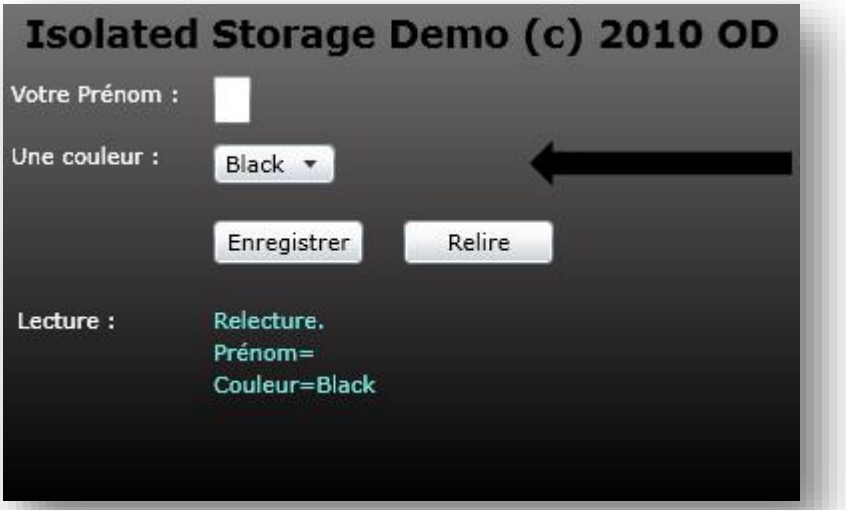


Figure 22 - Capture - Isolated Storage démonstration

Une fois les informations saisies (votre prénom, le choix d'une couleur) cliquez sur le bouton de sauvegarde. Cliquez sur le bouton de lecture pour vérifier l'enregistrement. Faites un copier de l'adresse dans votre browser, fermez le. Rouvrez-le, coller l'adresse (c'est juste pour vous faire gagner du temps, vous pouvez aussi naviguer normalement vers ce billet). Comme vous le constaterez les valeurs que vous avez choisies sont reprises automatiquement par l'application ... Un peu comme un cookie mais en beaucoup plus puissant.

Stockage isolé ?

La zone de stockage est dite isolée dans le sens où elle est placée dans un endroit assez peu accessible sur la machine utilisateur (Mac ou PC) et qu'elle est directement reliée à une application Silverlight et pas une autre et qu'enfin la taille occupée par cette zone est elle-même contrainte par une gestion de quotas sous la responsabilité de l'utilisateur.

Mais pas un coffre-fort !

A noter toutefois que l'IS n'est pas "inaccessible", il est dur à trouver, c'est tout. Il n'est pas crypté non plus. Cela signifie qu'il ne faut JAMAIS y stocker des informations sensibles de type mots de passe, cartes de crédit, etc… En tout cas pas de base. Votre application peut (et doit dans ces cas précis) utiliser le service de cryptographie .NET pour rendre les informations inviolables et infalsifiable (les deux aspects étant différents, mais c'est bon de le rappeler).

Les quotas

Limité par défaut à 1 Mo cet espace peut être étendu par l'utilisateur.

Il n'y a rien à faire de spécial pour bénéficier de l'espace attribué par défaut, mais si on doit stocker des quantités importantes de données il est nécessaire de vérifier la place dont on dispose, voire de demander l'augmentation du quota attribué à l'application (ce qui doit être validé par l'utilisateur).

Comme un disque privé

L'IS se comporte comme un disque privé pour l'application. On peut y créer des répertoires, lire et écrire des fichiers de tout type, etc.

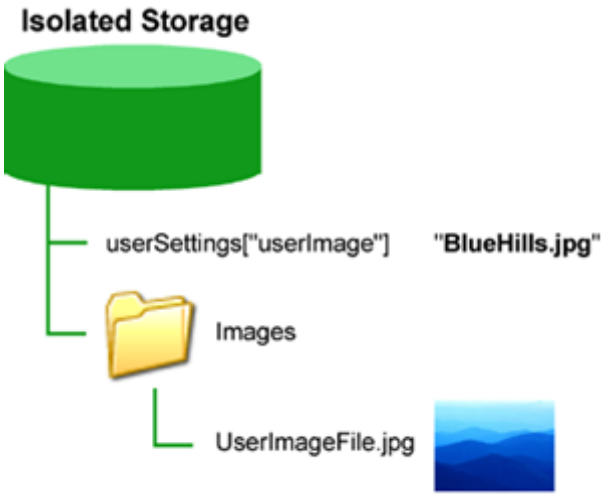


Figure 23 - Structure de l'Isolated Storage

Le schéma ci-dessus représente un Isolated Storage dans lequel sont exploités différents objets, une collection pour les préférences utilisateurs, un répertoire pour des images (avec une image stockée). Il s'agit donc bien d'une sorte de disque dur

privé pouvant stocker des données de tout type. On pourrait même y stocker une petite base de données. En tout cas on peut y placer des informations qui sont censées être là lors de la prochaine exécution.

Attention là aussi : autant l'IS n'est pas crypté par défaut et ne doit pas servir pour stocker des données sensibles, autant il n'est pas question non plus d'y stocker des données stratégiques qui n'existeraient nulle part ailleurs ! En effet, rien ne garantit que l'utilisateur n'aura pas, même pas erreur de manipulation, effacé l'espace attribué à l'application… Si l'application doit stocker des données à durée de vie "longue" ou ayant une importance pour l'utilisateur, il est préférable d'utiliser la manipulation de fichiers traditionnels qui seront stockés dans des espaces personnels de l'utilisateur qui eux sont sauvegardés par les backups de façon plus certaine (et que l'utilisateur peut facilement dupliquer ou copier car ces espaces sont faciles d'accès à la différence de l'IS).

Pour les curieux, l'IS place ses fichiers dans des répertoires "profonds" mais tout dépend de l'OS hôte. Par exemple sous Vista ou 7 il s'agit de : `<SYSDRIVE>\Users\<user>\AppData\LocalLow\Microsoft\Silverlight\is`
Alors que sous XP la racine se trouve là :
`<SYSDRIVE>\Document and Settings\<user>\Local Setting\Application data\Microsoft\Silverlight\is`
Pour les utilisateurs Mac, le stockage est effectué à cet endroit :
`/Users/<user>/Library/Application Support/Microsoft/Silverlight/is`

Les bonnes pratiques

Voici quelques conseils quand il s'agit de gérer 'proprement' l'IS :

Placer tous les appels à l'IS dans des blocs `Try/Catch` car pour de nombreuses raisons l'IS peut lever des exceptions (quotas dépassés, droits insuffisants…)

Si plusieurs applications doivent partager un même espace de stockage, l'IS n'est pas fait pour cela, l'espace disque en question doit être géré sur un serveur.

S'il est possible de créer des répertoires dans l'IS, rappelez-vous que sa "racine" est déjà très profonde dans l'arborescence du système… On peut très vite franchir la limite des 260 caractères d'un Path si l'on n'y prend garde !

Toutes les données sensibles doivent être cryptées.

Les données un peu volumineuses peuvent être compressées avec une librairie de type *SharZipLib* pour ne pas consommer le quota alloué trop vite.

Il est préférable d'utiliser l'`IsolatedStorageSettings` (comme montré dans ce billet) pour stocker des objets ou des paramètres. Cette méthode est bien plus simple que les autres.

Utiliser `IsolatedStorageFile` si vous devez avoir une logique de type flux utilisant les API des `Streams`. Généralement pour de grandes quantités de données ou si vous avez besoin d'un contrôle fin du contenu de l'IS. Cette technique n'est pas montrée ici, il existe beaucoup d'exemple sur le Web qui devraient faire l'affaire.

La gestion simple des paramètres

Venons-en à l'exemple montré plus haut. Le but du jeu est ici de conserver en mémoire le prénom de l'utilisateur et son choix de couleur. On pourrait placer n'importe quoi dans l'IS de la même façon, comme des images, des fichiers XML, etc.

Deux boutons : l'un pour lire, l'autre pour écrire.

Le code d'écriture :

```
 1: private void writeUserPreferences()
 2:         {
 3:             var iStorage = // niveau application
                       IsolatedStorageSettings.ApplicationSettings;
 4:             // IsolatedStorageSettings.SiteSettings; au niveau site
 5:             iStorage["PRENOM"] =
                       string.IsNullOrWhiteSpace(tbNom.Text)
 6:                                         ? "Nobody"
 7:                                         : tbNom.Text.Trim();
 8:             iStorage["COULEUR"] =
                           cbColor.SelectedItem != null ?
                           cbColor.SelectedItem.ToString() :
                           "Black";
 9:             iStorage.Save();
10:             txtResults.Text = "Préférences sauvegardées.";
11:         }
```

Le code de lecture :

```
 1: private void readUserPreferences()
 2:         {
 3:             var iStorage = // niveau application
                         IsolatedStorageSettings.ApplicationSettings;
 4:             // IsolatedStorageSettings.SiteSettings; au niveau site
 5:             tbNom.Text = iStorage.Contains("PRENOM")
                             ? iStorage["PRENOM"].ToString()
                             : string.Empty;
 6:             var cName = iStorage.Contains("COULEUR")
                             ? iStorage["COULEUR"].ToString()
                             : "Black";
 7:             txtResults.Text = "Relecture." + Environment.NewLine +
                                  "Prénom=" + tbNom.Text +
                                  Environment.NewLine +
 8:                               "Couleur=" + cName;
 9:             cbColor.SelectedItem = cName;
10:         }
```

Code 16 – Charger les préférences utilisateur

Comme vous le voyez, si on supprime le code propre à l'application, l'accès à l'Isolated Storage s'effectue de la façon la plus simple qu'il soit :

D'abord on récupère une instance du stockage spécifique pour la gestion des paramètres (l'autre mode d'accès à l'IS impose de gérer des répertoires, des streams, etc) :

```
1: var iStorage = IsolatedStorageSettings.ApplicationSettings;
```

Ensuite on lit ou on écrit directement dans un dictionnaire clé / valeur :

```
1: iStorage["PRENOM"] = tbNom.Text.Trim();      // Lecture
2: tbNom.Text = iStorage["PRENOM"].ToString();  // Ecriture
```

Quand on écrit, ne pas oublier d'appeler la méthode `Save()` du stockage pour persister les données.

L'exemple ici ne contient pas de gestion des exceptions : c'est mal ! :-) A vous d'en ajouter dans tout projet réel.

Dialogue JavaScript / Silverlight

Depuis la version 2.0 de Silverlight et sa programmation en C# (ou VB.net) j'avoue ne plus jamais m'être intéressé à JavaScript dans ce contexte. Je déteste JavaScript comme Jean Yanne détestait les départementales (le sketch du permis de conduire), ce n'est pas forcément très technique (quoi que…), c'est plutôt "tripesque", une sorte d'allergie. Mais parfois dialoguer entre une application Silverlight et la page qui l'accueille peut s'avérer très utile !

Ne soyons donc pas sectaires et sachons utiliser JavaScript quand il a un réel intérêt.

Appeler et se faire appeler

Une application Silverlight n'est pas forcément une grosse application monolithique, cela peut être, aussi, un gadget, un menu, une partie d'une application HTML ou ASP.NET. On peut améliorer grandement un site existant avec des morceaux de Silverlight sans pour autant refaire tout le site ! Et dans ce cas il peut s'avérer essentiel d'envoyer au plugin des informations en provenance de la page hôte ou bien l'inverse, agir sur la page hôte depuis l'application Silverlight.

Attention : ne pas confondre avec le Local Messaging

Le dialogue entre deux plugins Silverlight sur une même page utilise un autre procédé, le "*Local Messaging*", une sorte de canal de communication entre les différents plugins posés sur une page hôte. Solution qui peut s'ajouter à celle que nous voyons aujourd'hui. Nous verrons cette forme spéciale de communication dans un autre billet.

Pour l'instant ce qui nous intéresse c'est de faire communiquer un plugin Silverlight avec la page hôte, et dans les deux sens.

Appeler Silverlight depuis JavaScript

Pour qu'une classe Silverlight puisse être invoquée depuis JavaScript il faut qu'elle décide de se rendre visible, et même qu'elle déclare quelles méthodes seront visibles. C'est un schéma de sécurisé classique, sans déclaration volontaire du code Silverlight JavaScript ne verra rien.

Les étapes à réaliser sont les suivantes :

Dans le constructeur de l'application Silverlight il faut faire un appel à `RegisterScriptableObject()`. Cet appel autorise le dialogue entre JavaScript et l'application en même temps qu'il créé un nom par lequel le plugin SL sera visible.

Toute méthode qui doit pouvoir être appelée de l'extérieur doit être marquée par l'attribut `ScriptableMember`.

Depuis JavaScript il est maintenant possible d'accéder directement aux membres visibles de la classe SL au travers du DOM. Nous verrons l'exemple de code plus bas.

Appeler JavaScript depuis Silverlight

Dans ce sens-là les choses plus simples puisque SL peut appeler n'importe quel code JavaScript de la page hôte en utilisant `HtmlPage.Windows.Invoke()`.

Code Exemple

J'ai l'habitude d'intégrer les petits exemples Silverlight directement dans le billet mais ici il faudrait intégrer la page Html hôte qui intègre elle-même l'exemple SL. Rien d'impossible mais la structure du blog ne le permet pas directement. Si vous voulez voir fonctionner l'exemple il faudra télécharger le projet (en fin de billet)…

Pour simplifier nous allons mettre en place une sorte de boucle : l'application Silverlight va envoyer un texte à la page HTML en invoquant un code JavaScript de cette dernière et ce code JS va utiliser la communication avec le plugin pour retourner en écho le même texte.

Figure 24 - Communiquer avec la page HTML

Ci-dessus l'application Silverlight. On saisira un texte dans le champ supérieur, on l'enverra à la page HTML en cliquant sur le bouton, et en retour on recevra de la page le message dans la zone inférieure (avec une petite transformation au passage pour se prouver que cela fonctionne).

Scénario filmé

Pour comprendre la cinématique voir notre scénario en quelques images :

1 – L'application est lancée, la page ASP.NET est affichée avec notre plugin.

Figure 25 - Début de séquence : affichage de la page HTML et du plugin Silverlight

2 – On saisit un texte et on clique sur le bouton Go!

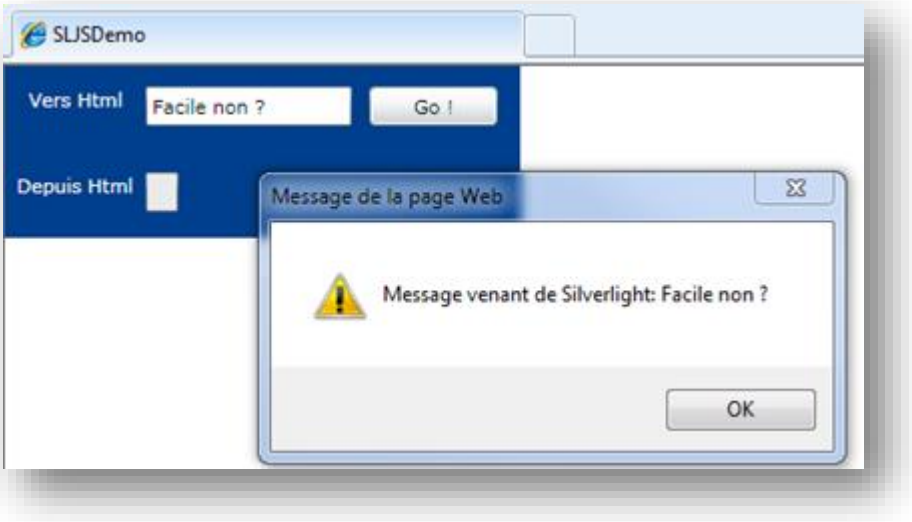


Figure 26 - Saisie d'un tester envoyé vers HTML

3 – On valide le dialogue ouvert par la fonction JavaScript qui va maintenant modifier et retourner la chaîne à l'application SL :

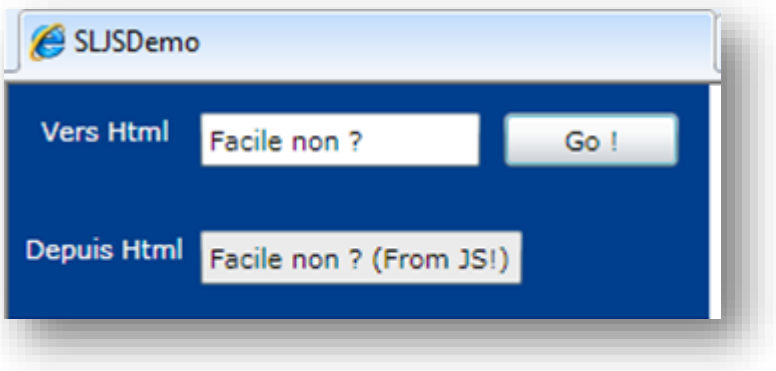


Figure 27 - Modification en retour de l'application Silverlight

Le code de la classe Silverlight

```
 1: namespace SLJSDemo
 2: {
 3:     public partial class MainPage : UserControl
 4:     {
 5:         public MainPage()
 6:         {
 7:             InitializeComponent();
 8:
 9:             HtmlPage.RegisterScriptableObject("PageSL",this);
10:         }
11:
12:         [ScriptableMember]
13:         public void UpdateMessage(string message)
14:         {
15:             textBox2.Text = message;
16:         }
17:
18:         // envoi du message à JavaScript
19:         private void button1_Click(object sender, RoutedEventArgs e)
20:         {
21:             HtmlPage.Window.Invoke("SilverlightToJavaScript",
                                                textBox1.Text);
22:         }
23:     }
24: }
```

Code 17 - Code C# d'appel et réception vers HTML

Le code JavaScript de la page ASP.NET

```
1: <script type="text/javascript">
2:     function SilverlightToJavaScript(msg) {
3:         alert("Message venant de Silverlight: " + msg);
4:         var control = document.getElementById("silverlightControl");
5:         control.Content.PageSL.UpdateMessage(msg+" (From JS!)");
6:     }
7: </script>
```

Code 18 - Code JScrip de dialogue avec le pluging Silverlight

Ce code a été placé dans la balise `<HEAD>` de la page. Il faut aussi noter qu'a été ajouté un "`id`" à la balise `<OBJECT>` du plugin Silverlight pour donner un nom à ce dernier (utilisé dans le code ci-dessus "`silverlightControl`").

Quant au code Xaml, deux Textbox et un bouton c'est pas trop compliqué à comprendre…

Charger une image dynamiquement

La plupart du temps utiliser une image dans une application Silverlight est quelque chose de simple : l'image est ajoutée au projet, depuis les *Assets* on la pose sur la surface de design et c'est tout… Mais ce n'est pas forcément la façon la plus subtile de gérer l'affichage d'images de grandes tailles ou d'images nombreuses, voire d'images au contenu variable (images mises à jour par une application serveur par exemple). Dès lors qu'on dépasse l'utilisation du simple glyph il est en réalité souvent nécessaire de charger dynamiquement la ou les images en question.

Comme toujours avec le couple Xaml / C# il existe plus d'une façon d'atteindre le but recherché. Mais à l'origine on trouve un besoin commun : créer une URI pour pointer l'image, qu'elle soit locale à l'application ou bien distante (quelque part sur le Web). Si l'image est locale elle doit bien entendu être ajoutée au projet SL. Dans ce cas le chargement dynamique ne réduira pas la taille du Xap ni son temps de téléchargement. Mais parfois le chargement dynamique peut servir d'autres objectifs que l'optimisation de la taille d'une application.

La version la plus simple consiste à faire comme suit :

```
Image image = new Image();
Uri uri = new Uri("images/myImage.png", UriKind.Relative);
ImageSource img = new System.Windows.Media.Imaging.BitmapImage(uri);
// ajouter ensuite "image" à l'arbre visuel
image.SetValue(Image.SourceProperty, img);
```

Code 19 - Chargement d'une image par code

Dans cette version on ne gère pas l'événement de la classe image permettant de savoir si l'image a bien été téléchargée ou non. S'il s'agit d'une image locale on peut faire l'économie d'une telle gestion. Mais s'il s'agit d'une image distante je vous conseille fortement de programmer l'événement `ImageFailed` de la classe Image ! Un réseau, et encore plus Internet, pose souvent des problèmes, qu'ils soient ponctuels (engorgement, pannes temporaires) ou plus spécifiquement liés à la ressource à laquelle on accède (disparition, changement de place, altération...). Dans de tels cas vous aurez certainement à vous frotter à l'infâme erreur `AG_E_NETWORK_ERROR`.

Après la création de l'objet Image (dans l'exemple ci-dessus) il suffit d'ajouter un gestionnaire :

```
img.ImageFailed
+= new EventHandler<ExceptionRoutedEventArgs>(img_ImageFailed);
```

Code 20 - Prendre en compte les erreurs de chargement

Puis de le programmer :

```
void img_ImageFailed(object sender, ExceptionRoutedEventArgs e)
{
        // gestion de l'erreur
}
```

Le code d'erreur est "open", à vous de réagir selon les circonstances et la gravité de l'absence de l'image. Très souvent j'opte pour une solution simple qui évite beaucoup de code et qui avertit immédiatement l'utilisateur : je stocke dans le projet un Png ou un Jpeg simple affichant un glyph d'erreur avec ou sans message. En cas d'erreur de chargement d'une image je charge l'image "erreur" à la place...

Dernière chose : lorsqu'on charge une image dynamiquement comme dans les exemples de ce billet il reste une étape à ne pas oublier : ajouter l'image au visuel en lui affectant un parent, un conteneur déjà présent et visible, par exemple :

```
LayoutRoot.Children.Add(image);
```

Enfin, dernière chose, vous pouvez aussi gérer l'événement `ImageOpened` de la classe Image. Cet événement intervient une fois l'image téléchargée et décodée avec succès. On peut l'exploiter en divers occasions comme prévenir l'utilisateur qu'une image longue à télécharger est enfin disponible et peut être consultée ou simplement et plus pratiquement pour prendre connaissance de la taille de l'image avant qu'elle ne soit affichée (ce qui est très pratique si cette taille est inconnue de l'application).

Bien évidemment, entre l'ajout d'une balise Xaml de type `<Image Source="images/MonImage.png"></Image>` et la gestion des événements ici décrits il y un monde, la différence entre une application de test et une application professionnelle...

Réagir au changement de taille du browser

Silverlight et Xaml sont très souples et il existe souvent plusieurs moyens d'atteindre le même objectif surtout lorsqu'il s'agit de mise en page. Par exemple centrer une application SL dans le browser peut se faire en créant une application occupant tout l'espace puis gérer dans l'application le centrage d'une grille ou bien on peut se reposer sur HTML et CSS pour centrer dans une balise `<div>` le plugin SL.

Il se peut aussi qu'on préfère un autre type de placement qui réclame de connaître la taille de l'espace utilisable du browser et de réagir à tout changement de dimension de ce dernier.

Le moyen le plus simple pour réagir au changement de taille du browser consiste à accrocher un gestionnaire d'événement sur l'objet `App.Current.Host.Content` :

Dans `Page.xaml.cs`:

```
public Page()
{
     InitializeComponent();
     App.Current.Host.Content.Resized += new EventHandler(Content_Resized);
}
```

Code 21 - Prendre en charge les changements de taille de l'application

Il est alors facile de prendre en compte la modification de taille lorsqu'elle intervient. Dans l'exemple ci-dessous on se contente de relever la hauteur et la largeur sans rien faire, dans une application réelle il y aurait évidemment un code plus subtil !

```
void Content_Resized(object sender, EventArgs e)
{
     double height = App.Current.Host.Content.ActualHeight;
     double width = App.Current.Host.Content.ActualWidth;
}
```

Code 22 - Retailler le plugin pour utiliser l'espace du browser

Localiser une application

L'informatique suit le monde, et le monde se mondialise… Une entreprise française devient une filiale d'un fond de pension américain, un éditeur de logiciel de Toulouse fusionne avec concurrent catalan, un fabriquant de téléphone finlandais devient américain, etc. Les outils eux-mêmes se globalisent, prenez Windows Phone 8 ou les tablettes Surface, pensez au market place centralisé de Microsoft pour vendre vos applications. Les applications ne peuvent plus se contenter d'être purement locales, elles doivent intégrer dès leur création les mécanismes permettant de les localiser. Le Framework lui-même, à la différence des plateformes précédentes, gère la localisation de façon naturelle et intégrée. Nous allons voir dans ce billet comment rendre vos applications Silverlight utilisables dans le monde entier.

Localiser

Pourquoi les informaticiens disent-ils "localiser", anglicisme puisé de "*localize*", plutôt que dire simplement "*traduire*" une application ? Encore un snobisme propre à nos profession ?

Non. Bien sûr que non. La traduction est à la localisation ce que l'algèbre est à la physique : un simple ingrédient permettant de créer le résultat final qui dépend de nombreux autres facteurs ! En effet, la "localisation" ne consiste pas seulement à traduire des bouts de texte, mais aussi à gérer ce qui est afférent à la **culture de l'utilisateur**.

Respecter la culture

Déjà traduire est une tâche difficile. Il y a le contexte, la mode, certains mots "valables" ne s'emploient plus ou plus que dans certaines situations, les langues sont vivantes donc changeantes. Traduire une application est un job difficile qui demande une parfaite maîtrise de la culture cible et une totale compréhension du texte qu'on traduit (donc dans notre cas du logiciel dont il fait partie).

Mais le respect de la culture cible va bien au-delà des mots : il faut prendre aussi en compte le format des dates, les civilités, l'ordre des mots, le format des numéros de téléphone, de sécurité sociale, la monnaie, les systèmes de mesure (S.I. ou non), les tailles habituelles du papier pour la mise en page par exemple (un format Letter américain n'a rien à voir avec de l'A4 traditionnel utilisé en France), voire même les couleurs, les connotations de certains gestes (s'il y a des photos, des vidéos, les italiens par exemple disent au revoir avec un petit geste de la main qui donne une impression efféminée en France, faudra-t-il tourner une version spéciale de chaque vidéo intégrée dans le logiciel ou éviter d'être trop "typé" dès le départ ?)…

On le voit ici, "localiser" c'est beaucoup plus que "traduire" qui, de fait, n'est qu'un élément du processus de localisation qui englobe des réflexions qui dépassent de loin le remplacement de "oui" par "yes" ou "si" !

La culture sous .NET

Heureusement pour le développeur, la plateforme .NET prend en charge de base de nombreux aspects liés à la culture, encore faut-il utiliser convenablement ce socle et construire son logiciel par dessus et non pas à côté en réinventant la poudre !

Sous .NET les cultures sont représentées par des objets qui contiennent déjà des informations essentielles comme le formatage des nombres, le symbole monétaire, et d'autres informations qui ne s'inventent pas. Néanmoins il reste une part du travail à fournir : intégrer la localisation dans les mécanismes de l'application, traduire les textes, adapter les comportements du logiciel.

Les cultures sont codifiée sous la forme "langue-région", par exemple le français parlé en France se code "`fr-FR`", mais le français de nos amis québécois se code "`fr-CA`" alors que celui de nos voisins belges se note "`fr-BE`". Il existe aussi un "`fr-CH`" pour les chuisses, et un "`fr-LU`" pour le luxe bourgeois des luxembourgeois.

On ne compte pas non plus les déclinaisons de l'anglais, "`en-US`" pour les GI de l'oncle SAM mais "`en-GB`" pour les grands bretons.

Ceux-là sont malgré tout faciles à retenir, mais si je vous demande le code de l'ouzbek, ça se complique un peu !

Comme vous le constater, la langue est représentée par un code en minuscules, alors que la région (souvent un pays) est noté en majuscules.

.NET sait aussi comprend les codes simples, c'est à dire uniquement formés d'un code langue comme "`fr`" ou "`us`" sans précision de la région. Cela est beaucoup moins précis mais peut s'utiliser. On peut supposer un logiciel français qui n'a pas besoin de tenir compte des différences entre français de France et français suisse ou wallon. Mais cette pratique est déconseillée. Il suffira que le même logiciel nécessite un jour d'écrire une valeur monétaire pour que toute la localisation soit à revoir (les suisses risquent de faire une attaque s'ils voient des prix en francs suisse affichés avec le symbole de l'euro !).

Bref, on code toujours ensemble langue et culture pour former un code de localisation. Plus techniquement il s'agit d'associer un code ISO 639 en minuscules pour la langue et un code ISO 3166 en majuscules pour la culture associée à une région ou un pays.

Pour aller plus loin, c'est ISO 639-1 qui est utilisé pour la langue. Il existe des extensions utilisant plus de lettres (dont le prochain ISO 639-5) mais qui ne sont pas utilisées sous .NET à ma connaissance. Il en va de même pour l'ISO 3166-1 et ses extensions. (Les liens proposés vers Wikipédia vous permettront d'obtenir les listes complètes).

Localiser une application

Une fois ces considérations générales abordées, voyons concrètement comme localiser une application Silverlight. Pour la démo l'application sera rudimentaire, un simple affichage avec quelques informations. Le processus que nous allons voir est indépendant de la complexité de l'application.

Le dossier des ressources

Le principal élément de la solution aux problèmes de localisation passe par la gestion de ressources quelle que soit la plateforme. Le Framework .NET n'y échappe pas. Notre application non plus et dans un premier temps nous allons ajouter un

répertoire que nous appellerons "Resources" (j'ai l'habitude de faire tout en anglais par défaut, vous pouvez choisir un autre nom si cela vous chante).

Les ressources par défaut

A l'intérieur de ce dossier de ressources nous allons créer un fichier Resource qui contiendra les valeurs de la langue par défaut de l'application. Nommons-le "Strings.resx".

Ajoutons une première chaîne "AppName" qui contiendra le nom de l'application.

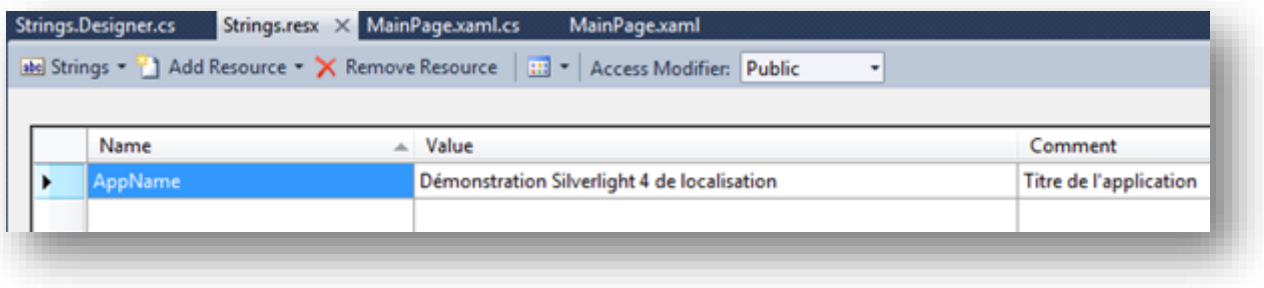


Figure 28 - Création d'un fichier de ressources de traduction

Bien entendu nous utiliserons le data binding pour lier l'interface aux ressources. Pour ce faire nous ajoutons la déclaration d'un espace de noms dans le fichier Xaml du UserControl (appelons ce namespace "local") :

```
1: xmlns:local= "clr-namespace:SLLocalization.Resources"
```

XAML 5 - Namespace local

Puis nous devons déclarer une ressource locale dans le UserControl :

```
1: <UserControl.Resources>
2:         <local:Strings x:Key="Strings" />
3: </UserControl.Resources>
```

XAML 6 - Création de la ressource en XAML

Maintenant nous pouvons ajouter un TextBlock en haut de notre page pour afficher le titre de l'application en nous liant à la ressource "Strings" :

```
1: <TextBlock Text="{Binding AppName,
                            Source={StaticResource Strings}}" />
```

XAML 7 - Binding avec les ressources

J'ai bien entendu retiré de la balise ci-dessus tout ce qui concerne la mise en forme de l'objet texte (fonte, placement…).

Le résultat est immédiatement visible (une construction du projet s'avère nécessaire) :

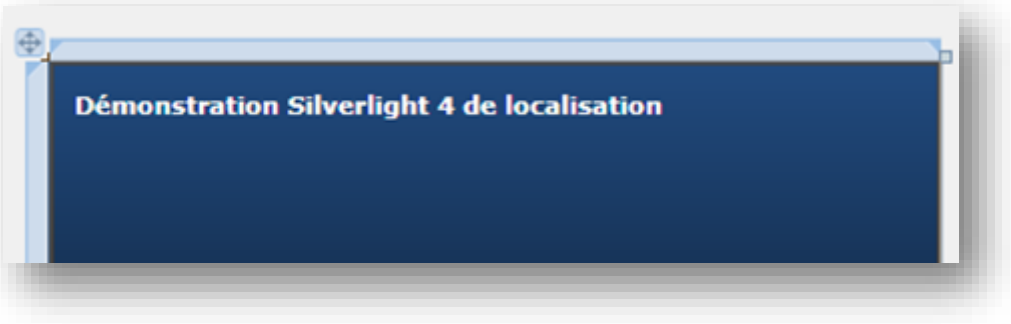


Figure 29 - Affichage d'un texte en ressource

Pour se rendre compte du résultat, le mieux est toujours d'effectuer un run. Et là que se passe-t-il ? Bug !

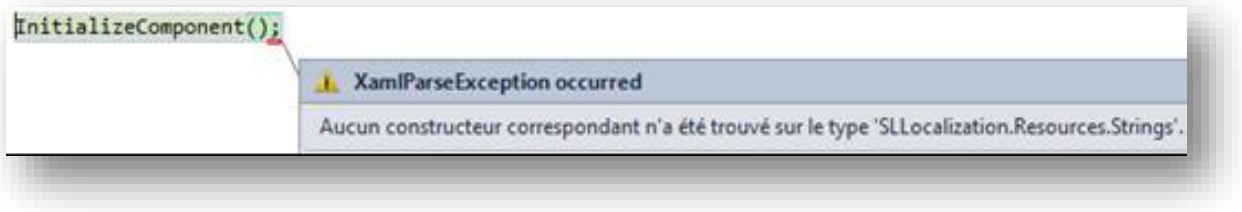


Figure 30 - Erreur d'accès aux ressources

Comment ça "aucun constructeur" pour le type `Strings` ? Regardez bien la première capture, plus haut, lorsque le fichier de ressources `Strings` a été modifié pour fixer le nom de l'application. Et oui, nous avons bien spécifié pour "`Access Modifier`" la valeur "`Public`".

Hélas, si vous regarder dans le code généré par Visual Studio, ici "`Strings.Designer.cs`" et que vous cherchez la déclaration du type `Strings`, vous tomberez sur un constructeur marqué "`internal`" !

Il s'agit d'un bug connu. Mais bien embêtant. A vérifier si cela est toujours le cas dans la version de Silverlight que vous utiliserez…

Vous avez toujours la possibilité de modifier "`internal`" en "`public`" et de relancer l'application, cela va fonctionner. Mais à chaque modification du fichier des ressources il faudra recommencer puisque le code source lié à la ressource est régénérer par Visual Studio automatiquement.

Contourner le bug

Avant d'aller plus loin dans notre quête de localisation il nous faut faire une pause et se demander comment nous pouvons contourner ce bug de façon élégante.

La façon la plus simple consiste à créer une classe publique qui elle exposera la ressource. Comme cette dernière possède un constructeur "`internal`" et que notre classe appartiendra bien au même namespace, nous n'aurons aucune difficulté à mettre en place cette "feinte".

Le code de la classe "ResourceLocalizer" (localisateur de ressources) est le suivant :

```
1: public class ResourceLocalizer
2:     {
3:         private readonly Resources.Strings appStrings = new Strings();
4:         public Resources.Strings AppStrings
5:         {
6:             get { return appStrings; }
7:         }
8:     }
```

Code 23 - ResourceLocalizer

Mais ce n'est pas la seule modification à effectuer.

En effet, nous devons corriger la déclaration du namespace dans le fichier Xaml de la page principale, il devient :

```
1: xmlns:local= "clr-namespace:SLLocalization"
```

De même, la déclaration de la ressource dans le `UserControl` devient :

```
1: <UserControl.Resources>
2:     <local:ResourceLocalizer x:Key="Strings" />
3: </UserControl.Resources>
```

Et enfin, le binding du TextBlock se transforme un peu pour devenir :

```
1: <TextBlock
2:   Text="{Binding AppStrings.AppName,
                Source={StaticResource Strings}}" />
```

XAML 8 - Binding vers le ResourceLocalizer

Cette fois-ci, tout fonctionne à merveille, aussi bien au design (à condition d'avoir fait un build du projet) qu'au runtime où l'infamante exception a disparu !

Nous pouvons continuer le travail…

Indiquer les langues supportées

Aussi bizarre que cela puisse paraître et alors que toutes les attentions ont été portées dans le Framework pour supporter la localisation et alors même que Visual Studio est le plus complet des EDI qui soit, et bien nous tombons encore sur un os… Après le bug que nous venons de contourner, voici maintenant qu'aucune option n'a été prévue pour indiquer les langues supportées dans la page des propriétés du projet !

Pour arriver à nos fins, il va falloir encore une fois ruser.

- Clic droit sur le nom du projet Silverlight
- Puis choisir '"unload project" (décharger le projet)
- Nouveau clic droit sur le projet
- Choisir "Edit <nom du projet>" (modifier le projet)

Dans le nœud "`PropertyGroup`" de ce fichier XML qui décrit le projet, ajoutons la ligne suivante (par exemple à la fin de la liste, juste avant la fermeture de la balise PropertyGroup) :

```
1: <SupportedCultures>fr-FR;en-US</SupportedCultures>
```

XAML 9 - Indiquer les cultures supportées

`SupportedCultures` n'apparaît pas dans IntelliSense, encore un petit problème. Mais ce n'est pas grave. Dans cette nouvelle balise il suffit d'indiquer les couples langue-culture qui sont supportés par l'application en les séparant d'un point-virgule.

Comme vous le notez, j'ai placez deux langues dans la balise : le français de France (`fr-FR`) et l'anglais américain (`en-US`) car tout ce travail n'a de sens que si nous supportons au moins deux langues !

On n'oublie pas de sauvegarder le fichier projet, de fermer sa fenêtre et de refaire un clic droit sur le projet puis "*reload project*" (recharger le projet).

Nous pouvons à nouveau travailler sur notre application et lui ajouter enfin une nouvelle langue !

Ajouter le support de la nouvelle langue

Le décor étant en place, il ne reste plus qu'à ajouter une nouvelle langue. Pour ce faire nous allons créer dans le répertoire des ressources un nouveau fichier ressources

qui s'appellera "`Strings.en-US.resx`", c'est à dire de la même façon que le fichier que nous avons déjà créé mais en ajoutant la culture comme extension.

Le fichier de ressources doit contenir les mêmes identificateurs que le fichier d'origine, par exemple nous devons recréer une chaîne ayant pour clé "`AppName`" :

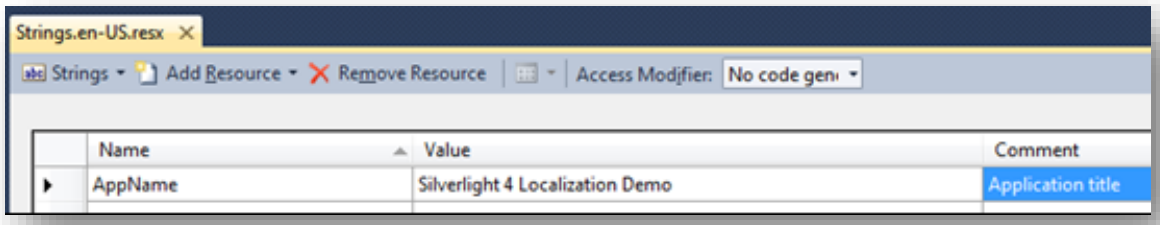


Figure 31 - Localisation d'une ressource

On remarquera aussi que "*Access Modifier*" est placé à la valeur "*No code generation*" (pas de génération de code). De fait aucun fichier "`.cs`" ne se trouve associé.

Quelques chaines de plus

En réalité une seule suffira bien pour cet exemple. Ici nous allons faire intervenir d'autres éléments de la localisation notamment le format des dates et le nom des jours et des mois. De plus, comme nous avons déjà montré comment accéder aux éléments localisés sous Xaml via binding, nous allons voir comment bénéficier des mêmes services par code.

La chaîne que nous allons créer a pour clé "`Today`". Elle permettra d'indiquer la date et l'heure.

Dans `Strings.resx` (le fichier par défaut) nous déclarons la chaîne comme suit :

```
1: Nous sommes le {0} et il est {1}
```

Vous remarquez l'utilisation des marqueurs propres à "`string.Format`" qui permettent justement de modifier le placement d'éléments variables.

Dans le fichier `Strings.en-US.resx` la chaîne sera définie de la façon suivante :

```
1: We are {0} and local time is {1}
```

Dans le code de la page principale (`MainPage.xaml.cs`) nous utilisons cette ressource pour fixer la valeur d'un `TextBlock` appelé "`txtDate`" :

```
1: void MainPage_Loaded(object sender, RoutedEventArgs e)
2: {
3:   txtDate.Text =
              string.Format(SLLocalization.Resources.Strings.Today,
4:              DateTime.Now.ToLongDateString(),
5:              DateTime.Now.ToShortTimeString());
6:  }
```

Cette initialisation s'effectue dans le gestionnaire `Loaded` du `UserControl` définissant la page.

Ce qui est intéressant ici est de voir que nous ne passons plus par la classe localisatrice de ressources nécessaire en Xaml (pour contourner le bug de VS).

Nous accédons directement à la propriété `Today`, qui est statique par défaut, de la classe `Strings` (la classe porte le nom du fichier, par défaut aussi), du namespace `Resources` (un sous répertoire de projet est fournisseur de namespace par défaut) de l'application `SLLocalization`.

Pour fixer le texte nous utilisons `string.Format` à qui nous passons d'une part la chaîne localisée et en arguments la date en format long et l'heure en format court.

Prise en charge de la langue en cours

Si l'on ne fait rien de plus, le projet tournera en français et devrait tourner en anglais sur un Windows anglais. Mais j'avoue ici une lacune, je n'ai pas réussi à savoir si cela était réellement automatique. N'ayant pas de machine totalement en US sous la main, je vous laisse tester…

Pour tester le passage en anglais sur ma machine (et cela serait le cas chez vous aussi, que cela soit pour l'anglais, l'espagnol ou autre d'ailleurs) il faut forcer un peu les choses.

Forçage par balise du plugin

Il est possible de forcer la prise en charge d'une culture au niveau du plugin, c'est à dire de la balise de définition de ce dernier, soit dans la page HTML hôte soit dans une page ASP.NET.

Les paramètres concernés s'appellent en toute logique :

- culture
- uiculture

Leur valeur doit être généralement la même, il s'agit du code culture complet, "`fr-FR`" par exemple.

De cette façon on s'assure que l'application sera toujours dans une langue donnée quel que soit le browser et la machine hôte.

Cela peut être utile dans certains cas certainement. Mais je ne vais pas utiliser cette méthode ici.

Forçage par code

Il suffit de forcer la culture dans App.xaml.cs au niveau du constructeur "`Public App()`" et de préférence avant l'appel à `InitializeComponent()`;

Le forçage s'effectue assez classiquement par le biais du thread courant :

```
1: Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
2: Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
```

XAML 10 - Forcer la culture

Vérification

Un run avec le code ci-dessus donnera l'affichage suivant :

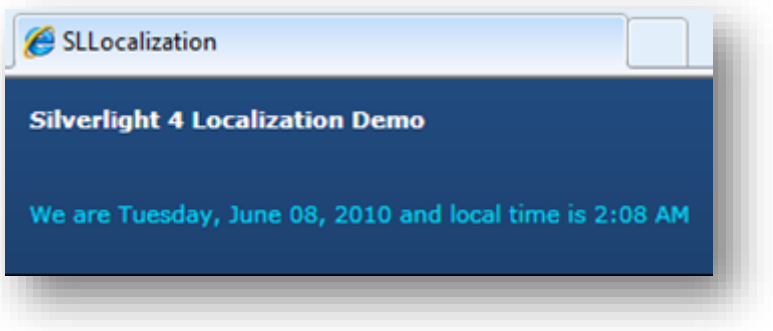


Figure 32 - Application localisée en US

On notera le titre traduit (en blanc sur fond bleu, au-dessus il s'agit de l'onglet de IE) ainsi que le respect de la culture américaine dans le format de la date, le nom du jour, du mois et l'indication de l'heure en AM/PM.

Si nous commentons les deux lignes de code fixant la culture un nouveau run nous donnera ceci :

Figure 33 - Application localisée en FR

C'est pas magnifique non ? (oui, il est bien 2:10 du matin ce n'est pas un nouveau bug !).

Plus loin que la traduction

Comme je le disais en introduction, la localisation va bien plus loin que la simple traduction des chaînes de caractères.

La technique exposée ici permet de gérer tous les autres cas de figure. Rappelez-vous qu'un fichier de ressources peut contenir des chaînes mais aussi des images, des icônes, de l'audio, etc. Si on désire avoir une image de fond avec le drapeau français pour la version française et un drapeau américain pour la version anglaise, il suffira de définir une clé "`ImageDeFond`" qui dans un fichier sera un drapeau français et dans l'autre sera un drapeau américain…

De même on peut vouloir définir de la même façon un jeu de couleur, des indicateurs booléens, tout ce qui peut servir à personnaliser l'application selon la culture de l'utilisateur.

Certaines informations, comme par exemple les chaînes, pourront être utilisées directement par binding dans les pages. D'autres peuvent simplement servir par code à formater des données comme nous l'avons vu ici.

Avec des fichiers Wav on pourra proposer des informations audio traduites elles aussi par le même mécanisme (reste à trouver les traducteurs et surtout les speakers ayant le bon accent pour faire les prises de son !).

Conclusion

Localiser une application Silverlight n'est pas très compliqué mais il faut admettre que cela n'est pas aussi simple qu'on pourrait l'espérer surtout avec les quelques petits bugs que VS met en travers de notre route.

Mais grâce à quelques explications on s'en sort finalement sans trop de peine. Et puis c'est le mécanisme de base qui demande un peu de travail, ensuite il suffit de remplir les fichiers de ressources et d'utiliser le contenu. On a ainsi un développement qui reste simple mais un résultat "pro" d'une application localisée.

Gestion du clic droit

Silverlight 4 a apporté de nombreuses améliorations, certaines plus visibles que d'autres. Parmi les plus discrètes, mais pas forcément les moins utiles, se trouve désormais la gestion du Clic Droit.

L'exemple

(À utiliser en live sur le site Dot.Blog en cliquant sur le titre de ce chapitre)

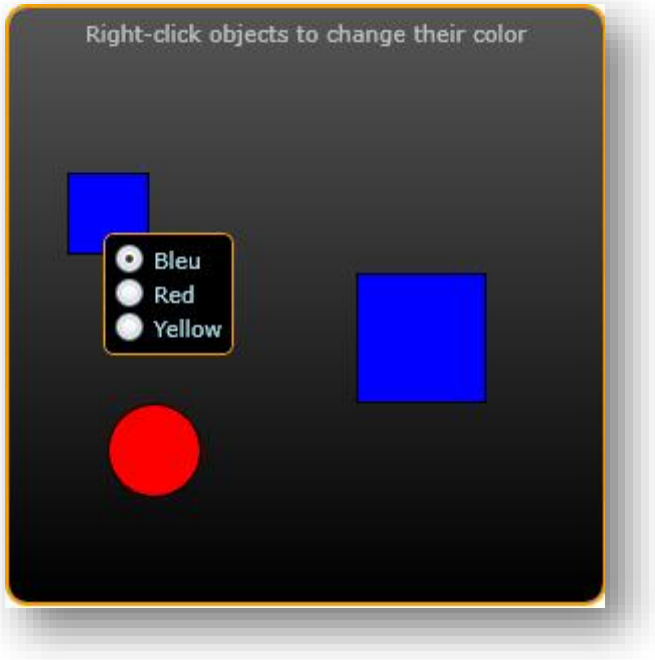


Figure 34 -Gestion du clic droit

Cliquez avec le bouton droit de la souris sur l'une des trois formes, un popup "glissant" viendra se coller à votre souris, à l'intérieur trois boutons radio permettant de choisir chacun une couleur (bleu, rouge, jaune). A l'arrivée du popup le radio bouton qui est sélectionné correspond toujours à la couleur actuelle de la forme choisie.

On peut bien entendu cliquer sur l'un des boutons pour changer la couleur de l'objet, ce qui entraine la fermeture du popup. On peut aussi décider de faire un autre clic droit sur une forme différente, le popup suivra alors la souris et le bouton coché sera mis à jour en fonction de la nouvelle cible.

Le code

Je ne vais pas vous abreuver de lignes de codes, le projet exemple est à téléchargé en fin de billet, il sera bien plus facile de le consulter au travers de Visual Studio ou Blend.

Notons simplement que cette nouvelle prise en charge du clic droit par Silverlight s'effectue grâce à deux nouveaux événements :

- `MouseRightButtonDown`
- `MouseRightButtonUp`

Il s'agit du pendant logique aux événements existants permettant de gérer le clic du bouton gauche. Tout cela se programme de la même façon et réagit identiquement.

Bien entendu lorsqu'un objet gère le clic droit l'utilisateur n'a plus accès au menu Silverlight. Toutefois ce dernier reste accessible partout où le clic droit n'est pas géré (reprenez l'exemple ci-dessus et faites un clic droit en dehors des trois formes colorées et vous obtiendrez le popup Silverlight classique).

En marge du code de démonstration du clic droit l'application présente certaines astuces qui vous intéresseront peut-être :

- Gestion du `clip` de la forme extérieure à bord arrondi
- Utilisation du behavior `FluideMovebehavior` qui anime le popup automatiquement
- Astuce d'un canvas permettant de positionner facilement le popup près de la souris

- Utilisation de la propriété `Tag` des radio-boutons et des formes pour mémoriser la couleur choisie
- Gestion du clic gauche sur toute la surface pour annuler le popup (abandon du popup par l'utilisateur)

Plus quelques autres petites choses que je vous laisse découvrir.

Conclusion

Gentiment mais surement, Silverlight s'est doté de capacités en faisant un outil de développement complet, paré pour la conception d'applications internet riches, belles et souples n'ayant rien à envier, côté pratique, aux application dites desktop. Il est dommage que l'arrêt de Silverlight en tant que plugin mette fin à ce qui devenait tout juste mature. Heureusement on retrouve Silverlight sous d'autres noms dans Windows Phone ou WinRT, avec quelques nuances.
Le clic droit ce n'est pas grand-chose, mais la gestion de menus popup locaux peut faire toute la différence entre un logiciel infernal à utiliser et un autre, à l'interface dépouillée, agréable à utiliser, n'affichant les menus locaux que lorsque cela est nécessaire et à l'endroit où ils ont un sens.

Bien entendu le clic droit peut être utilisé librement sans pour autant servir à afficher un popup. A chacun de faire preuve d'inventivité (en veillant à ce que malgré tout l'utilisateur puisse s'y retrouver, donc sans trop "casser" les codes ergonomiques classiquement admis).

Charger une image sur l'hôte

Charger une image depuis un serveur est un sujet que j'ai déjà abordé (voir: Silverlight : Charger une image dynamiquement). Charger une image depuis la machine hôte me semblait simple et je n'avais pas encore abordé le sujet. Pourtant, en lisant certaines questions que je reçois, force est de convenir que cela ne doit pas être si évident. Alors ne restons pas dans le questionnement et passons à la réponse, qui est courte.

Un bref exemple

Cliquez sur le bouton dans l'application SL ci-dessous, cela ouvrira le dialogue d'ouverture de fichier sur votre machine. Choisissiez une image png ou jpg et validez.

Elle remplira la zone de l'application. Cliquez à nouveau sur le bouton pour recommencer l'action. Bien entendu dans ce livre rien ne se passera, cliquez sur le titre de chapitre pour accéder au billet original sur Dot.Blog et jouez avec l'exemple live !

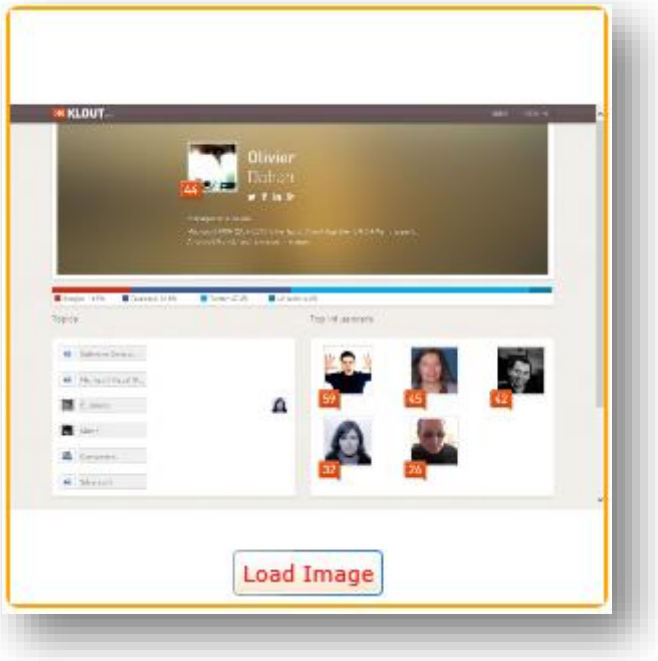


Figure 35 - Chargement d'une image

Le code

Je vous fais grâce du code Xaml (un `UserControl` avec une image et un bouton, ce n'est pas très compliqué à refaire…), voici plutôt le code C# :

```
 1: namespace SLLoadImg
 2: {
 3:     public partial class MainPage : UserControl
 4:     {
 5:         public MainPage()
 6:         {
 7:             InitializeComponent();
 8:         }
 9:
10:         private void btnLoadImage_Click(object sender,
                                              RoutedEventArgs e)
11:         {
12:             var ofd = new OpenFileDialog
13:                           {
14:                               Filter =
       "Images Files (*.png;*.jpg)|*.png;*.jpg | All Files (*.*)|*.*",
15:                               FilterIndex = 1
16:                           };
17:             if (ofd.ShowDialog()==true)
18:             {
19:                 var stream = ofd.File.OpenRead();
20:                 var bi = new BitmapImage();
21:                 bi.SetSource(stream);
22:                 imgExt.Source = bi;
23:                 stream.Close();
24:             }
25:         }
26:     }
27: }
```

Code 24 - Chargement d'une image depuis la machine hôte

Le code se trouve dans le click du bouton car l'`OpenFileDialog` ne peut, pour des raisons de sécurité, qu'être appelé suite à un tel clic de l'utilisateur.

Une fois l'instance du dialogue configurée, on l'exécute afin de récupérer le choix de l'utilisateur. C'est là que la partie se complique visiblement puisque c'est le passage du nom de fichier vers une image utilisable comme telle qui pose souci à certains lecteurs.

Mais vous allez voir que ce n'est pas si compliqué. Ligne 19 nous commençons par récupérer un flux (`stream`) sur le fichier sélectionné. Ce stream nous est fourni par `OpenRead` de la propriété `File` du dialogue.

Ensuite, ligne 20, nous créons une instance de `BitmapImage`. A la ligne suivante nous affectons sa source (`SetSource`) en lui passant en paramètre le flux récupéré juste avant.

Il ne reste plus qu'à transmettre l'image au composant image déjà présent dans l'application (`imgExt`) en initialisant sa propriété Source pour la faire pointer vers l'objet `BitmapImage` obtenu précédemment.

Au passage (ligne 23) on referme poliment le flux obtenu au départ de la séquence.

Conclusion

C'est très simple, mais l'enchainement entre l'obtention du flux, le passage vers un `BitmapImage` pour revenir sur un composant Image n'est effectivement pas limpide. Je conçois qu'on puisse chercher un moment en tournant autour sans y arriver.

Le multitâche

Silverlight, depuis la version 3, permet via le modèle de programmation classique des Threads de concevoir des applications multitâches prenant en compte les processeurs multi-cœur. Cela est essentiel à plus d'un titre.

Il faut en effet savoir que le thread principal de programmation est utilisé pour la mise à jour de l'UI et que cela est bloquant : un traitement long en C# suspend le rafraichissement créant saccades visuelles, image fixe et autres désagréments. C'est pour cela que Microsoft conseille de "casser" les "gros morceaux" de code procédural en petites unités de travail. Mais cela n'est pas toujours envisageable. Le multitâche peut en revanche facilement permettre de contourner le problème…

Cela est tellement important que des environnements comme WinRT sont totalement asynchrones.

Exemple live

En cliquant sur le titre de ce chapitre vous accéderez au billet original sur Dot.Blog et vous pourrez tester l'exemple live. Jouer avec permet de mieux comprendre les concepts évoqués ici.

Explications

A gauche le bouton de démarrage, à droite un stack panel contenant 6 instances d'un UserControl permettant de visualiser l'activité d'un job. Un job représente une tâche exécutée en arrière-plan dans un thread séparé. Chaque job simule aléatoirement un travail connaissant des pics d'activité et des pauses.

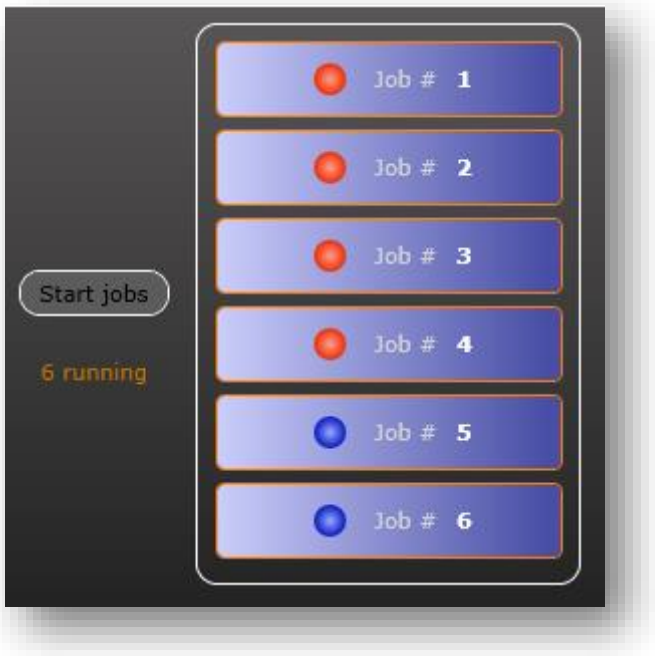


Figure 36 - Voir le multitâche en action

Lorsqu'un job est terminé, sa led devient gris pâle. Quand un job tourne sa led est bleue lorsqu'il est en mode pause et rouge quand il travaille. Dans cet exemple il n'y a aucune différence entre les deux états puisque autant la pause que le travail sont simulés par un `Thread.Sleep()`. Dans la réalité il s'agirait de vrais traitements ou d'attentes de données par exemple.

La durée totale d'un job ainsi que la durée de chaque pause et de chaque phase d'activité sont choisies de façon aléatoire. Deux runs différents donneront des résultats visuels différents donc ce qui est plus amusant !

Durant l'activité de tous les jobs un message sous le bouton indique le nombre de jobs restants actifs. La durée de chacun étant aléatoire, les extinctions de job arrivent dans un ordre quelconque.

Quand tous les jobs sont terminés le texte indique "*Inactive*" et le bouton de lancement est de nouveau `Enabled` (pendant le travail des jobs le bouton se grise et n'est plus actif).

Amusez-vous quelques fois… (on se lasse assez vite malgré tout ☺)).

La structure du programme

L'application se décompose comme suit :

`MainPage` (cs/xaml), c'est la fiche principale et unique de l'application.

`JobView` (cs/xaml) est le `UserControl` permettant de visualiser l'activité d'un thread

`Job.cs` est la classe qui effectue une tâche, nous allons revenir en détail sur son fonctionnement ainsi que la manière dont elle communique avec le thread principal pour mettre à jour l'affichage.

Dans ce billet je n'étudierai pas la mise en page (sous Blend) ni la création du `UserControl` (sous Blend aussi) avec ses propriétés de dépendance et sa gestion des états via le Visual State Manager. Le code source est téléchargeable en fin d'article, je vous laisse découvrir tout cela (et venir poser des questions si vous en avez, les commentaires sur Dot.Blog sont là pour ça !).

La classe Job

Lorsqu'on veut faire du multitâche on s'aperçoit bien vite que communiquer entre les tâches ou entre une tâche et l'application peut s'avérer complexe. De même, chaque tâche doit le plus souvent être capable de gérer un contexte (c'est à dire un ensemble de valeurs).

On sait aussi que le multitâche est beaucoup plus simple à gérer lorsqu'on travaille avec des objets immuables (*immutable* en anglais). Par exemple les chaines de caractères de .NET sont immuables, c'est un choix très malin dans un environnement multitâche. En effet les problèmes du multitâche ne pointent leur nez qu'à partir du moment où des objets (des valeurs) sont accédées par des threads différents. L'une des astuces consiste donc à ne traiter que des objets immuables et plus aucun conflit n'est possible ! Cette stratégie n'est pas celle adoptée totalement ici car les threads n'ont pas d'interaction entre eux et je ne voulais pas trop alourdir l'exemple.

En revanche, pour gérer le contexte de chaque thread, plutôt que de se mélanger avec des solutions biscornues, il existe une possibilité très simple : créer une classe qui contient tout le contexte ainsi que la méthode qui sera lancée dans un thread.

Ainsi, chaque thread fait tourner une même méthode mais celle-ci appartient à une instance unique de l'objet contexte. De fait, la méthode qui tourne dans son thread peut accéder sans risque de mélange à toutes les variables et autres méthodes de son contexte. Pas de "`lock`" à gérer, tout devient simple…

Regardons le code de `Job.cs` :

```
 1: public class Job
 2:     {
 3:         private readonly Random rnd =
                        new Random(DateTime.Now.Millisecond);
 4:
 5:         public int EndValue { get; set; }
 6:         public Action<Job> JobDone { get; set; }
 7:         public Action<Job, JobState> ReportJob { get; set; }
 8:         public int JobID { get; set; }
 9:         public int Counter { get; set; }
10:         public bool Done { get; set; }
11:         public JobView View { get; set; }
12:
13:         public void DoJob()
14:         {
15:             Done = false;
16:             Counter = 0;
17:             while (Counter < EndValue)
18:             {
19:                 Counter++;
20:                 if (ReportJob != null)
                            ReportJob(this, JobState.Idle);
21:                 Thread.Sleep(rnd.Next(250));
22:                 if (ReportJob != null)
                            ReportJob(this, JobState.Working);
23:                 Thread.Sleep(rnd.Next(250));
24:             }
25:             Done = true;
26:             if (JobDone != null) JobDone(this);
27:         }
28:     }
```

Code 25 - La classe Job

Pour son fonctionnement, la classe déclare une variable `Random` qui sera utilisée pour déterminer la durée des pauses et des phases d'activité. On notera qu'elle pourrait être déclarée en `static` car une seule classe est préférable, toutefois il faudrait protéger ses accès par des locks au minimum. Puis elle déclare une liste de propriétés publiques : `EndValue` (qui est le compteur de boucles, la durée du job pour simplifier), deux `Action<Job>` sur lesquelles nous allons revenir, un `JobID` attribué par le programme principal (un simple numéro), un compteur (la boucle principale de 0 à `EndValue` simulant le job) une variable booléenne `Done` qui sera positionnée à `True` lorsque le job sera terminé, et `View`, de type `JobView` qui pointe l'instance du `UserControl` servant à visualiser l'activité du thread. Ce contrôle est créé par le programme principal en même temps que l'objet Job qui initialise l'ensemble des propriétés que nous venons de lister.

La méthode `DoJob()` est la méthode de travail. C'est elle qui sera lancée dans un thread. Mais comme vous pouvez le constater ce modèle de programmation multithread est totalement transparent : la classe `job` est parfaitement "normale". Rien de spécial n'a été fait pour le multitâche. C'est là l'intérêt d'encapsuler la méthode de travail dans un objet, chaque tâche sera lancée dans une instance dédiée ce qui limite énormément les cafouillages possibles.

La création des jobs

Les jobs sont créés une fois la page principale de l'application chargée. Voici comment :

```
 1: private List<Job> jobs = new List<Job>();
 2:
 3:         void MainPage_Loaded(object sender, RoutedEventArgs e)
 4:         {
 5:             var r = new Random(DateTime.Now.Millisecond);
 6:             for (var i = 0; i < 6; i++)
 7:             {
 8:                 var view = new JobView
                            { Margin = new Thickness(3),
                              JobID = (i + 1).ToString() };
 9:                 spJobs.Children.Add(view);
10:                 jobs.Add(new Job
11:                             {
12:                                 EndValue = 20 + r.Next(30),
13:                                 JobID = i,
14:                                 JobDone = OnJobDone,
15:                                 ReportJob = OnJobReport,
16:                                 View = view
17:                             });
18:
19:             }
20:             txtState.Text = "Inactive";
21:         }
```

Code 26 - Extrait du code de lancement des tâches

L'application gère une liste "`jobs`" de tous les jobs créés. Dans le `Loaded` de la page on voit la boucle qui créée 6 instances du `UserControl` visualisateur d'activité et six instances de la classe Job. Le contrôle à l'aspect d'un rectangle au coin arrondi indiquant l'état actuel du job auquel il est rattaché. Là aussi pas de risque de cafouillage puisque chaque job possèdera son propre objet de visualisation.

Nous disposons maintenant d'une situation de départ simple : 6 instances de la classe `Job` existent en mémoire, chacune représentant un *contexte d'exécution* initialisé de façon propre et 6 instances de la classe `JobView` ont été créées et ajoutées au stack panel vertical qui occupe la partie droite de l'application. Chaque instance de `Job` est reliée au `JobView` lui correspondant. Ce lien existe plutôt comme membre du contexte de la tâche qu'en tant que véritable lien permettant aux deux instances de discuter ensemble. En tout cas dans notre exemple `Job` n'interviendra pas directement sur son JobView (il pourrait le faire).

Le dialogue avec l'application

Comme toujours il existe plusieurs façons d'arriver au but. Ainsi, les instances de Job doivent être capable (lorsque le job tourne) de rapporter leur activité à l'application principale. J'ai retenu deux événements : un reporting de l'activité ("je dors" / "je travaille") et un reporting spécial indiquant la fin du job.

Tout cela pourrait fort bien être géré avec un seul point d'accès ou au contraire avec un point d'accès différent pour chaque possibilité. De même la classe `Job` pourrait définir des `Events` auxquels le programme principal serait abonné de façon classique.

Ici, histoire de changer, j'ai préféré déclarer deux propriétés `Action<Job>` dans la classe `Job`. Comme le montre le code d'initialisation des jobs ci-avant, `JobDone` et `ReportJob` pointent vers les méthodes `OnJobDone` et `OnJobReport` de la page principale.

Si vous revenez sur le code de la classe `Job` vous verrez qu'en effet, si des actions sont définies, elles sont directement appelées. Il n'y a guère de différence avec une gestion d'événements, c'est simplement une autre façon de faire.

Prenons la méthode la plus simple d'abord :

OnJobReport

Cette méthode (une `Action<Job>`) sera appelée par chaque `job` pour signaler son activité. Voici son code :

```
1: public void OnJobReport(Job job, JobState jobState)
2:         {
3:             if (job.View.Dispatcher.CheckAccess())
                       job.View.JobState = jobState;
4:             else
5:                 job.View.Dispatcher.BeginInvoke(
                       () => job.View.JobState = jobState);
6:         }
```

Code 27 - Accès intra ou inter Thread

Le but est ici de mettre à jour la led du `JobView` correspondant, donc de faire changer l'état du `UserControl JobView` associé à l'instance de `Job`. C'est pour garder ce lien que la propriété `View` existe dans la classe `Job`, cela évite de gérer un dictionnaire supplémentaire associant l'objet d'interface avec l'instance de `Job`. Bien entendu, puisque chaque `Job` connait son objet d'interface il pourrait agir directement dessus

sans passer par la page principale. Mais il devrait le faire de la même façon ! Car attention, c'est le seul endroit où le multitâche commence à poser problème : le job veut mettre à jour un objet d'interface depuis un thread différent … et bien entendu cela n'est pas possible.

Que l'on passe par une `Action` gérée dans la page principale ou bien que la classe `Job` agisse directement sur l'objet d'interface il faut prendre en compte le fait que ce dernier a été créé dans un autre thread et qu'il ne peut pas être manipulé par un autre thread que celui où il a vu le jour.

Ceci explique le code de `OnJobReport` : On récupère l'instance de `Job` qui se signale dans le paramètre de la méthode (`Action<Job>` rend possible d'appeler une action possédant un paramètre de type `Job`), puis on récupère l'objet `View` associé et là on demande à ce dernier "`CheckAccess()`". Si la méthode renvoie "`true`" c'est que l'appel a lieu depuis le même thread que celui qui contrôle l'objet `JobView`, sinon la méthode retourne "`false`". Dans le 1er cas on peut se permettre d'agir directement sur l'objet d'interface, dans le second cas il faut passer par une invocation : `BeginInvoke` qui prend un *delegate* en paramètre, delegate que je remplace ici par une expression Lambda.

On notera que tel qu'est conçue l'application le premier cas n'arrivera jamais (le signalement d'un Job émane toujours d'un thread différent de celui de l'interface). C'est donc pour que l'exemple soit complet que j'ai ainsi structuré le code avec l'appel à `CheckAccess()`.

Le travail de `BeginInvoke` est de transférer le code à exécuter (le delegate remplacé ici par une expression Lambda) au thread principal afin que la modification de l'objet d'interface soit effectuée par ce dernier.

Imaginez un fleuve. D'un côté un troupeau de moutons qui courent dans tous les sens. De l'autre un camion qui attend pour les emmener plus loin. Le camion possède une passerelle qui ne peut accepter qu'un seul mouton à la fois. `BeginInvoke` est un pont construit sur la rivière qui ne laisse passer qu'un seul mouton à la fois mais qui peut en entasser plusieurs en file le temps que ceux qui sont en tête puissent monter, un par un, dans le camion.

L'avantage est que d'un côté cela peut sautiller dans tous les sens, mais que de l'autre on ne gère bien qu'un mouton à la fois. Le pont jouant à la fois le rôle de transporteur d'information d'un coté à l'autre, d'un thread à l'autre, et de zone tampon entre des tas moutons excités et le camion avec sa passerelle mono-mouton.

OnJobDone

Je vous laisse découvrir ce code dans le projet fourni. Rien de spécial, les mêmes stratégies sont utilisées. Pour savoir si tous les threads sont terminés `OnJobDone` utilise une requête Linq sur la liste des jobs et compte ceux dont la propriété `Done` est encore à "`false`". Cela permet à la fois d'inscrire sous le bouton de lancement le nombre de threads restant en course, et de réactiver le bouton de démarrage lorsque plus aucun job ne tourne.

Le lancement des jobs

Lorsqu'on clique sur le bouton de démarrage c'est à ce moment que les threads doivent être créés et exécutés. Le code suivant effectue ce travail :

```
 1: private void btnStartJobs_Click(object sender, RoutedEventArgs e)
 2:         {
 3:             btnStartJobs.IsEnabled = false;
 4:             foreach (var job in jobs)
 5:             {
 6:                 var t = new Thread(job.DoJob);
 7:                 t.Start();
 8:             }
 9:             Thread.Sleep(100);
10:             OnJobDone(null);
11:         }
```

Code 28 - Création des threads

La liste des jobs créée et initialisée au chargement de l'application est balayée, un nouvel objet `Thread` est créé pour chaque `Job`. En paramètre lui est passé le nom de la méthode qui va effectuer le travail en multitâche "`DoJob()`" (de la classe `Job`). Le thread est ensuite démarré.

En fin de méthode on laisse une petite pause pour ensuite rafraichir l'affichage (le comptage des threads en cours d'exécution). S'agissant d'un exemple je n'ai pas trop chargé le code et je réutilise ici la méthode `OnJobDone`. Cela n'est pas à conseiller dans une application réelle pour plusieurs raisons. La principale étant le respect de la sémantique. Que l'application principale appelle "`OnJobDone`" alors qu'elle vient de démarrer les jobs est un contresens. De plus cette méthode est conçue pour être appelée depuis les jobs. Ce genre de mélange ne peut que rendre le code difficile à maintenir et à comprendre. Evitez ce genre tournure rapide… Ici il aurait fallu extraire

ce qui est utile à la fois aux threads et à l'application, le placer dans une autre méthode. Cette dernière aurait été appelée depuis `OnJobDone` et depuis le code ci-dessus. Ma fainéantise a un prix exorbitant en plus : il me faut cinquante fois plus de mots dans le billet pour expliquer et justifier ce mauvais code que de code propre écrit directement. Pan sur les doigts ! C'est une belle leçon… Faire propre tout de suite est toujours une économie !

Conclusion

Faire du multitâche sous Silverlight s'avère très simple pour peu qu'on utilise les bonnes patterns. Mais pas d'illusion non plus : ce billet ne traite pas de toutes les façons de faire du multitâche. Il existe des cas plus complexes, d'autres moyens (comme le `background worker`) et des situations plus difficiles à gérer. Toutefois les bases posées ici devraient vous permettre de vous lancer dans ce mode de développement devenu obligatoire avec la stagnation des fréquences des processeurs et l'augmentation du nombre de cœurs en contrepartie… Et depuis SL3 il est possible de jouer avec toute la puissance de l'hôte pour des applications encore plus réactives et riches. Ne vous en privez pas !

Optimiser les performances des traitements graphiques

Silverlight est totalement graphique, optimiser les traitements graphiques pourrait s'appliquer à toutes les applications Silverlight donc. Mais en fait ici je parle de celles qui font réellement une utilisation intensive du graphique. Et il existe plusieurs choses importantes à savoir dans un tel contexte pour obtenir un résultat fluide et de bonnes performances.

Le Debug

Plusieurs options de Silverlight peuvent vous aider à trouver les failles de performances :

EnableRedrawRegions

Quand cette option est en route Silverlight affiche les rectangles qui sont redessinés. Les régions ainsi délimitées apparaissent en bleu, en jaune ou rose. C'est à la création du plugin qu'il faut ajouter l'option et non dans l'application elle-même en utilisant le paramètre de nom `EnableRedrawRegions` et en indiquant la valeur "`true`" (la façon

exacte dépend de si vous utiliser une balise ASP.NET Silverlight ou d'une balise HTML Object).

MaxFrameRate

Cette option (au niveau plugin aussi) permet de fixer un frame rate (nombre d'images par seconde) autre que celui par défaut (60 d'après MSDN). Il est important de savoir que la valeur par défaut peut s'avérer excessive pour une application de type gestion (affichage d'informations, grilles de données, etc) et qu'il est souvent préférable de la descendre entre 10 et 30 pour éviter de consommer trop de puissance processeur pour rien. Le mieux est de descendre la valeur jusqu'à temps que cela se voit (artefacts visuels, animations perdant de leur fluidité, etc). Quand vous arrivez à cette valeur (enfin juste au-dessus) vous avez optimisé le frame rate. Rappelons au passage qu'il s'agit bien du "Max" frame rate, c'est à dire la vitesse maximale qui dépend de l'hôte et de sa charge, le frame rate réel peut être bien inférieur si l'hôte rame comme un fou ! Aussi, rien ne sert de mettre au point une application avec un frame rate à 120 sur votre machine de compétition alors que les utilisateurs ne pourront au mieux obtenir peut-être que la moitié, voire moins…

EnableFrameRateCounter

Pour évaluer l'impact d'un traitement graphique ou vérifier la vitesse actuelle de rafraichissement de votre application, ou pour effectuer un réglage fin du `MaxFrameRate`, encore faut-il savoir combien d'images par secondes sont traitées par le plugin… En activant cette option (au niveau du plugin aussi) Silverlight affichera un compteur des FPS à l'écran.

Attention cela ne fonctionne que sur Internet Explorer et sur un OS Windows. Si vous testez l'application sous Chrome, Firefox ou sur un Mac, cette information ne sera pas affichée même si l'option est activée.

EnableGPUAcceleration

Par défaut les optimisations GPU sont désactivées… car avant d'activer cette option (dans le plugin aussi) il est préférable de comprendre comment elle agit. D'autant qu'elle a un fonctionnement assez limité. En effet, si WPF se repose totalement sur DirectX, puisque ces deux technologies fonctionnement exclusivement sur Windows, Silverlight par sa nature cross-plateforme est obligé de se baser sur un consensus minimaliste.

Ainsi, l'accélération matérielle ne fonctionne que sous Windows Vista, Windows 7/8 et Windows XP. Pas sur Mac par exemple. Et sous XP encore faut-il que le pilote de la carte (Nvidia, ATI ou Intel uniquement) soit plus récent que novembre 2004.

Autant dire que si cela fonctionne sur votre machine cela peut ramer sur celles des utilisateurs si leurs machines ne sont pas dans les rails de ces exigences !

Utiliser l'accélération GPU doit se faire dans un esprit particulier : si la machine le supporte l'application sera encore plus fluide, mais elle doit être au minimum "correcte" si l'accélération n'est pas possible. Ce qui implique de concevoir et d'optimiser les applications sans cette option puis de l'ajouter et de voir comment elle peut aider et rendre l'interface plus fluide.

N'oubliez pas non plus que l'accélération GPU fonctionne au niveau de l'objet et chaque objet visuel doit avoir son `CacheMode` défini à `BitmapCache`, avec des implications non négligeables et parfois imprévisibles sur le visuel (pixellisation, transformations plus lentes que sans accélérations…).

EnableCacheVisualization

Lorsque vous souhaitez activer l'accélération GPU il peut être intéressant de vérifier quelles zones en profitent réellement. Cette option s'active aussi au niveau du plugin. Toutes les textures dont le rendu est effectué par logiciel (et qui donc ne bénéficient pas de l'accélération) seront affichées avec une couleur choisie au hasard (différente pour chaque couche).

XPerf

Il s'agit d'un utilitaire Microsoft fourni avec le SDK de Windows 7, on le trouve dans les Windows Performance Analysis Tools (WPT).

L'étude de cet outil dépasse le cadre de ce billet et je renvoie le lecteur intéressé au lien précédent pour plus d'information. Xperf n'est pas limité à Silverlight, c'est un outil complet d'analyse des performances. Il peut s'avérer essentiel pour débusquer les lenteurs et manques de fluidité dans les applications de tout type, dont Silverlight.

Les vidéos

Le pipe-line de Silvelight lorsqu'il joue un média de type vidéo est le suivant :

1. *Décodage de l'image*
2. Conversion *YUV*
3. Calcul du resize si l'image n'est pas affichée dans sa taille originale

4. Mélange des couches si nécessaires (par exemple surimpression d'une barre de commande, de symboles…)
5. Dessin effectif de l'image.

Il en découle quelques évidences :

Encodez toujours vos vidéos dans la taille à laquelle vous les affichez… Si vous changez la taille d'affichage en cours de conception, réencodez la vidéo dans la nouvelle taille.

Encodez les vidéos avec un le frame rate le plus bas acceptable visuellement sauf si vous souhaitez faire de la HD réclamant une connexion ADSL ayant un très bon débit ce que beaucoup de gens n'ont pas. L'élitisme en la matière est stupide. Même avec de l'ADSL2+ chez moi j'ai souvent des problèmes sur les vidéos HD tout simplement parce les prétentieux qui les fournissent n'ont pas forcément les moyens de mettre des serveurs à la hauteur… Alors soyez humble, choisissez un format correct le plus sobre possible que vos serveurs pourront supporter en offrant une video fluide, c'est mieux qu'un truc en HD avec des saccades ! Ou alors, pour la HD, "crachez le budget" serveur qui va avec ! :-) Et cela avec ou sans Silverlight.

Sur-impressionner des couches sur de la vidéo est très coûteux, évitez de le faire… (Genre barre de commande en opacité réduite et autres zinzins qui peuvent parfaitement être affichés sur l'un des côtés plutôt qu'en surimpression).

Quand vous utilisez des vidéos dans votre application essayez de ne pas descendre sous un frame rate de 60 fps pour un affichage correct.

Le Texte

Le texte c'est simple, vectoriel, rapide. Mais en fait ce n'est qu'un dessin comme les autres, et souvent assez complexe ! Le moteur de rendu de Silverlight est amené à recréer l'image d'un texte dans de nombreuses conditions ce qui dégradera plus ou moins visiblement les performances. Par exemple une simple translation (ce qui n'est pas forcément "logique") ou tout changement d'échelle oblige le moteur à recalculer le texte ce qui ralentit le rendu. Mettre du texte sur une vidéo implique un ralentissement important (rendu du texte, séquence de fusion image décodée / texte rendu).

Dans bon nombre de cas, et même si cela fait perdre beaucoup de souplesse, il est préférable d'utiliser un PNG avec transparence pour gérer du texte qu'un objet texte.

Un PNG supportera facilement un changement de taille pas trop important sans impact visuel et sera bien plus rapide à afficher. Bien entendu cela s'entend pour les applications utilisant intensivement les graphiques (le titre de ce billet). La perte de souplesse (mise à jour du texte notamment) étant tellement importante qu'elle doit se justifier sérieusement.

Le plugin Silverlight

Certains réglages du plugin ont un impact plus ou moins important sur les performances. Certains sont évidents (frame rate, accélération GPU) d'autres sont plus "sournois".

Sournois à double titre d'ailleurs. D'une part parce qu'il n'est pas évident de savoir qu'ils ont un impact sur les performances, et d'autre part parce que cet impact peut fort bien varier d'une version à l'autre de Silverlight sans que cela ne soit forcément mentionné quelque part.

Windowless

Laissez ce paramètre à `false`. Lorsqu'il est à "`true`" les performances sont dégradées. Le positionner à "`true`" n'a de toute façon d'intérêt que s'il est nécessaire de mixer le contenu de l'application avec de l'affichage Html.

Pourquoi cette dégradation ? Il faut d'abord répondre à la question "à quoi sert ce paramètre ?" ! Et ça, je viens de le dire (deux qui suivent seulement pfff!), et oui, ça sert à pouvoir mixer du contenu Html avec l'affichage du plugin. Quand je dis mixer, c'est mixer, comme de la musique. Quand on mixe de la musique on superpose plusieurs couches, on ne les juxtapose pas. Mixer Html et affichage Silverlight signifie donc pouvoir écrire par dessus l'affichage de Silverlight depuis la page Html (ou le contraire).

Nous pouvons maintenant répondre à la première question. Qui se résume à en poser une troisième "comment ça marche le `windowless` ?". C'est tout simple : au lieu que le plugin dispose de son propre espace rectangulaire dans lequel il fait ce qu'il veut, en mode `windowless` (`=true` donc) Silverlight délègue l'affichage au browser.

Vous imaginez maintenant la perte de performance : pour chaque frame, SL doit avertir le browser, lui passer l'image, et le browser doit recalculer son propre rendu, les éventuelles surimpressions et réafficher le tout.

Bien entendu `Windowless` a son utilité (ajouter des symboles vectoriels SL au dessus d'un affichage telle une carte géographique gérée par la page Html par exemple) mais ce mode ne doit pas être utilisé sans en comprendre son impact. Notons parmi ceux-ci, et outre les performances globales qui sont largement touchées :

- Pas de possibilité de gérer la souris en Html si l'affichage SL est au dessus (même avec un fond transparent)
- Pas de capture de la souris en dehors des limites du plugin SL (même si ces limites sont invisibles)
- Pas d'IME ou de support pour l'Accessibilité
- Pas de support du mode plein écran
- Problèmes divers selon les browsers et les OS
- Effet de déchirement visuel dans les animations et les vidéos sous XP ou Vista avec DWM déconnecté et sous IE peu importe la plateforme
- Rendu peu fiable sous Safari
- Problèmes de focus sur les browsers Mozilla

Etc. Ca fiche la trouille une liste pareille hein ? !

Pluginbackground

Evitez autant que possible de mettre le fond du plugin en mode "`Transparent`" sauf si cela est absolument nécessaire. Le plus souvent j'ai vu cette pratique dans le seul but de s'assurer que le fond de l'application SL est le même que celui de la page Htm hôte. Si vous créez une application sérieuse devant s'intégrer sur des pages Html, il existe forcément une charte graphique pour ces dernières. Utiliser la couleur de fond indiquée par la charte comme couleur de fond du plugin. Vérifiez aussi que la taille d'affichage du plugin correspond bien celle fixée dans la page SL (si elle n'est pas en mode auto). Car si le plugin prend plus de place que la taille des pages SL, un bord blanc apparaitra autour !

Autres conseils

Bien d'autres choses sont à considérées si on souhaite qu'une application Silverlight soit efficace, fluide et rapide. Citons en vrac :

- Ne gonflez pas votre code par des ajouts et des ajouts, principalement le code Xaml. Si lors de la conception vous devez complexifier le code, refactorez, réorganisez et tentez de conserver un code compact. Ce conseil porte sur le code behind autant que sur Xaml. Tout code Xaml a un impact sur le visuel et sa fluidité. Les lourdeurs, mauvaises approches et autres propriétés fixées pour

les tests qu'on oublie sont autant de boulets que votre application traînera !

- Lorsque votre application n'a rien de spécial à faire, coupez son activité : stoppez les animations, stoppez les calculs en tâche de fond, etc. Bien entendu cela peut s'avérer contreproductif, à vous de juger au cas par cas, mais coupez toujours le maximum de choses, ne laissez pas tourner des animations pour rien notamment (objets cachés par d'autres, attention de l'utilisateur focalisée ailleurs…).

- Toute brosse qui est resizée réclame la création d'une nouvelle texture ce qui coûte cher en temps de calcul et en mémoire. Eviter les resize autant que possible.

- Quand Silverlight joue une animation il ne redessine que les parties rectangulaires qui le nécessitent ; animez des petits objets et évitez d'animer des placards !

- Les videos peuvent nécessiter un FPS de 60 au minimum, mais une application standard se contente de 10 à 30 FPS, 20 à 30 étant généralement la fourchette la plus consensuelle (en termes de fluidité et de consommation de ressources).

- Testez sur plusieurs plateformes et plusieurs browsers !

- Pendant le développement affichez toujours le compteur de frames, c'est un bon indicateur qui, s'il se dégrade d'un seul coup, vous permettra de cibler immédiatement le code incriminé (le dernier ajouté…).

- Quand vous animez l'opacité ou une transformation de tout descendant de `UIElement`, utilisez le `CacheMode` car si l'accélération GPU est en route (et disponible sur la plateforme hôte) ces opérations seront réalisées par le processeur graphique et non par votre application.

- En même temps, méfiez-vous du `CacheMode` qui peut engendrer des contre-performances dans certains cas ! Testez au cas par cas.

- Pour cacher un élément, utilisez `Visibility` plutôt qu'une opacité à `0`. Dans ce dernier cas l'objet existe toujours et gère le `Hit Test`. Un coût inutile s'il est invisible. SL 4 a ajouté la possibilité d'animer par des transitions les propriétés

de type `Visibility` ce qui n'était pas le cas avant. Vous pouvez donc cacher totalement un objet plutôt que de le rendre transparent sans pour autant que ce changement se fasse avec violence !

- Sachez que Silverlight utilise pleinement les machines multi-cœurs pour son moteur de rendu et l'affichage des médias de type videos. Si vous avez comme moi une machine principale à 4 cœurs en hyperthreading (donc 8 apparents pour l'OS) il est préférable de tester vos applis sur le portable de votre petite sœur ou celui que vous avez refilé à votre papy l'année dernière pour vous assurer qu'elles ne seront pas utilisables uniquement que par quelques geeks suréquipés !

- En mode plein-écran il est encore plus important de cacher (`Visibility=false`) les éléments inutiles qu'en mode normal (entendu pour les performances).

- Evitez de fixer une taille précise à un élément média : laissez-le s'afficher dans sa résolution native et adaptez votre affichage à la place restante. Le resize des vidéos comme expliqué plus haut est coûteux.

- Attention : Lorsque du code-behind s'exécute Silverlight ne met plus à jour l'affichage ! Cela ne se voit pas si ce code est court et rapide, mais si vous devez exécuter des méthodes lourdes, pensez à découper le travail en petites unités comportant des pauses pour éviter que le frame rate global ne chute dramatiquement ! Voir mon billet sur le multitâche…

- Parfois Silverlight a du mal à charger des XAP trop gros… Il est important de concevoir son application de telle sorte que le XAP principal soit le plus petit possible. Utilisez MEF, utilisez du lazy loading, séparer les ressources de grandes tailles (images, fichiers) de votre XAP pour les télécharger à part, etc…

- Un détail étonnant : `Double.Tostring()` est moins rapide que `Double.ToString(CultureInfo.InvariantCulture)` ! Si votre application doit stocker, traiter ou comparer des doubles transformés en string sans avoir à les afficher, spécifiez toujours la culture invariante, elle est optimisée pour la vitesse. Pour une présentation de données à l'utilisateur utilisez `ToString()` qui par défaut utilise la culture courante bien entendu (ou un convertisseur ou une chaîne de format Xaml).

Un autre détail intéressant : Si vous utilisez beaucoup d'images, placez leur mode Stretch à `Fill`. Cela peut paraître contre intuitif mais dans ce mode de nombreux calculs sont évités ce qui le rend plus rapide même que "`None`". La différence n'est pas énorme mais lorsque beaucoup d'images sont traitées cela peut avoir un impact non négligeable.

Conclusion

Le sujet est vaste… et ce long billet le prouve ! Mais développer une application Silverlight ce n'est certainement pas reproduire les automatismes acquis sous Windows Forms, les MFC, Delphi ou Java ! Je vois beaucoup trop de développeurs qui abordent Silverlight comme un simple module d'affichage qu'on pluguerait sur un code existant à la place de ASP.NET ou de Windows Forms… J'ai vu la même chose d'ailleurs en ASP.NET avec des applications conçues comme s'il s'agissait d'applications desktop, avec des résultats désastreux, forcément.

Silverlight est une technologie "légère", surtout laissez toujours quelques neurones allumés dans un coin pour vous en souvenir ! "Légère" cela veut dire qu'elle se destine au Web, à l'intérieur d'un browser, qu'elle se destine aux smartphones (sous Phone 7/8), autant de contextes où il n'est pas question de donner dans le dantesque ou le goulu ! Si vous voulez faire du gros, du lourd, faites-le en WPF, une technologie robuste pour le desktop utilisant DirectX et tout le framework, toute la puissance machine.

Se tromper d'outil est parfois une erreur grave qui se paye cher. Silverlight est étudié pour faire "certaines choses". Programmez-le en dehors de ce contexte et vous irez au mur tout en lui faisant une mauvaise publicité (ce qui n'est plus très grave aujourd'hui mais ce n'est pas une raison !). Vouloir reprogrammer votre énorme soft de gestion sous Silverlight serait certainement une erreur. Comme les applications pour unités mobiles, Silverlight doit être utilisé pour des actions ciblées, des applications limitées à un champ pas trop large.

En réalité les problèmes de performances les plus graves ne sont pas ceux que j'ai évoqués dans ce billet. Les plus graves prennent souvent racines dans un mauvais choix technologique à la base même d'un projet !

Les accès cross domaines

Silverlight est d'une telle puissance qu'il pourrait servir à créer des codes malicieux redoutables, du phishing plus vrai que nature et autres désagréments. Microsoft a eu tellement peur que cela n'arrive qu'ils ont verrouillés de nombreuses possibilités du produit.

Parmi celles-ci notons l'interdiction de télécharger quoi que ce soit en dehors du domaine de provenance de l'application sauf si le serveur cible rend cet accès possible via un fichier de configuration.

La solution existe mais elle n'est pas toujours applicable, car si le serveur en question n'a pas prévu le fameux fichier de configuration et si vous n'avez aucun droit sur ce serveur, il n'existe tout simplement pas d'alternative simple. Même pour du contenu public, volontairement publié pour être accessible depuis n'importe où. De n'importe où mais pas d'une application Silverlight. Ainsi en a voulu Microsoft. A avoir trop limité Silverlight Microsoft a certainement été la cause de l'adoption limitée du plugin sur le Web grand public. J'ai moi-même des tas de projets en réserve jamais réalisés car ils violaient tel ou tel point de sécurité. Résultat : arrêt de Silverlight. Dommage.

Il y a même une situation pire encore : votre Application doit accéder à des services hébergés sur un serveur qui n'est pas un serveur Web. Là, pas question de poser un fichier de configuration sur la "racine", il n'y a pas de "racine" au sens web du terme et le serveur ne répondra de toute façon pas aux requêtes HTTP.

Un (faux) espoir pour le futur (qui n'arrivera pas)

J'en avais parlé avec l'un des développeurs de l'équipe Silverlight qui s'occupait justement de la sécurité. Je lui avais suggéré qu'au moins la possibilité soit laissée de faire du cross domaine (sur un serveur non configuré) suite à un message de confirmation de l'utilisateur. Silverlight est déjà plein de messages de ce genre (pour passer en mode plein écran par exemple), un écran de confirmation pour autoriser le cross domaine serait vraiment un plus. Il a dit que ce n'était pas une mauvaise idée… J'espérais voir cet ajout dans SL 5. Mais cela n'a pas été le cas, et nous savons aujourd'hui qu'il n'y aura pas de SL6… Vraiment dommage.

Les plans B

Donc soyons réalistes. Sans fichier de configuration sur le serveur pas de solution.

En fait si, il y a un plan B : écrire une application serveur qui joue le rôle d'un proxy pour toutes les requêtes cross-domaine. C'est tellement lourd que je ne considère pas cela comme une solution (d'autant que le proxy va devenir un goulot d'étranglement). Mais dans certains cas il n'y a pas d'autres choix.

De même, si votre application Silverlight doit appeler un service qui ne se trouve pas sur un serveur IIS il n'y a pas d'autres voies que de créer une application serveur de fichier de configuration. Silverlight essayera de télécharger la configuration cross-domaine sur le port `943`, il "suffit" donc de répondre à cette demande. Une application Silverlight accédant à un service WCF sur une machine non serveur Web pourra donc le faire si vous ajoutez à côté des services un serveur de fichier de configuration de police d'accès. Mike Snow propose une implémentation d'un tel serveur, je vous renvoie ainsi vers son article "Full implementation of Silvelight Policy Server".

L'explication du danger

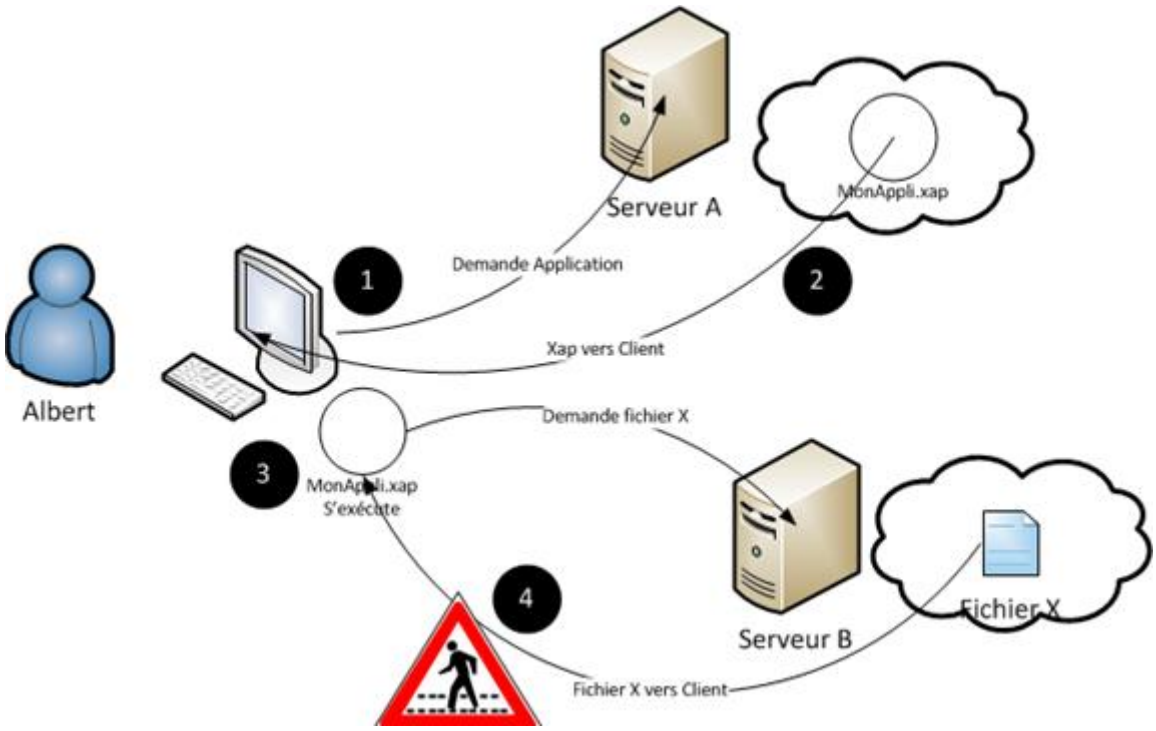


Figure 37 - Les dangers du cross domaine

Regarder le dessin ci-dessus. Prenons les opérations dans l'ordre :

Albert est assis devant son PC. Il se connecte au Serveur A pour télécharger l'application Silverlight `MonAppli.Xap`.

L'application `MonAppli.Xap` arrive sur le poste client d'Albert et s'exécute. Il s'agit d'une application Web utilisée au travers d'un browser disposant de certains mécanismes comme par exemple les cookies. Le browser d'Albert est installé sur la machine d'Albert, c'est Albert qui est connecté à son propre PC avec tous ses crédits, dont ses cookies.

L'application Silverlight essaye maintenant de télécharger le fichier X se trouvant sur le serveur B. Imaginons un fichier protégé contenant des informations secrètes que seul Albert et quelques autres privilégiés ont le droit de télécharger.

La demande de téléchargement émanant du PC de Albert et Albert disposant déjà des droits (stockés dans des cookies) lui permettant d'accéder au fichier X, et la demande provenant du Browser de Albert qui est bien connecté à sa machine, le serveur B va penser que c'est Albert qui demande le fichier X. Il va l'envoyer. Et celui-ci sera reçu à l'insu d'Albert par l'application `MonAppli.xap` qui pourra par exemple le transmettre en douce au serveur A.

Albert vient de se faire dépouiller d'un fichier X ultra secret se trouvant sur le serveur B sans même le savoir !

C'est contre les attaques de ce type que Microsoft (et Adobe pour Flash) a prévu un filtrage strict. Celui-ci réclame que le Serveur B expose dans sa racine un fichier de configuration particulier autorisant explicitement le Serveur A (et ses applications, donc `MonAppli.Xap`) à accéder au fichier X.

Vous allez me dire que si X est si secret que ça, on se demande bien pourquoi la sécurité est si pauvre et ne repose que sur un cookie. On est d'accord. Mais quand je pose cette question à des gens mieux informés que moi en termes de sécurité on me répond "mais ce n'est pas la question". Si ça l'est. C'est à mon sens au serveur d'être bien protégé et non à Silverlight de faire la police, mais bon c'est comme ça… Quoi ? Ah oui… Et si X n'est pas un fichier secret mais un truc banal que tout le monde peut télécharger ? J'ai posé la question bien entendu et on m'a parlé d'attaques bizarres autant qu'étranges dont j'ai oublié le nom après avoir vérifié qu'il s'agissait de trucs vraiment rares et très difficiles à réaliser. Bref, je suis navré de ne pas répondre de façon claire à tout cela, je n'ai jamais réussi à avoir de réponse claire moi non plus en dehors de jargon sécuritaire qui sent plutôt la paranoïa qu'autre chose. Les types qui font de la sécurité informatique doivent dormir avec un flingue sous l'oreiller je pense et doivent demander à leur femme de montrer sa carte d'identité avant d'ouvrir quand elle sonne à la porte… Le monde est une jungle, certes. Je suis trop resté

soixante-huitard certainement :-) “Peace & Love” me plaisait quand même bien plus comme slogan que “vos papiers svp” !

Cross domain policy

Si vous avez accès au serveur, alors vous pourrez configurer le fameux fichier qui doit est être placé sur la racine du serveur. Pas celle de votre application mais bien la racine du serveur.

Silverlight accepte deux fichiers :

- `CrossDomain.xml`
- `ClientAccessPolicy.xml`

Le second est testé en premier. Une raison à cela : `ClientAccessPolicy.xml` est “le” fichier de configuration de police d’accès cross-domaine de Silverlight. Il est naturellement téléchargé en premier. Mais Flash utilise une technique proche et les serveurs qui sont configurés pour Flash peuvent être utilisés par Silverlight sans modification… De fait si Silverlight ne voit pas son propre fichier de configuration il tente en seconde chance de télécharger `CrossDomain.xml` qui est le fichier de police cross-domaine pour Flash. Tout simplement.

Le format du fichier Flash est accessible sur le site Adobe : Cross-domain policy file specification. Bien entendu si vous avez accès au serveur et que vous avez le droit de placer un fichier xml dans sa racine, utilisez de préférence le format de fichier Silverlight plutôt que le format Adobe… D’abord parce que MS pourrait décider un jour de ne plus supporter cette astuce, et ensuite parce que le fichier Silverlight autorise une configuration plus fine.

ClientAccessPolicy.xml

La version la plus simple (qui autorise tout le monde à tout faire) est la suivante :

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <access-policy>
 3:   <cross-domain-access>
 4:     <policy>
 5:       <allow-from http-request-headers="*">
 6:         <domain uri="*"/>
 7:       </allow-from>
 8:       <grant-to>
 9:         <resource path="/" include-subpaths="true"/>
10:       </grant-to>
11:     </policy>
12:   </cross-domain-access>
13: </access-policy>
```

Code 29 - Fiche de police d'accès cross domaine

Peut-être que, comme moi, vous pensez que trop de sécurité tue la sécurité. Je suis convaincu que pour ne pas s'enquiquiner pleins de développeurs vont placer un fichier comme celui-ci sur leurs serveurs… Moralité c'est comme s'il n'y avait aucune sécurité. Tout ça pour ça…

Même si je sais que vous ne le ferez peut-être pas (à moins de gérer des données ultra sensibles), je vous déconseille quand même cette version "passoire" et de configurer plus finement les droits accordés par le fichier de police cross-domaine.

Pour plus de détail sur le format de ce fichier lisez la documentation MS "Mise à disposition d'un service au-delà des limites de domaine".

Conclusion

On bougonne, on plaisante, mais en réalité un serveur connecté au Web est comme une brebis abandonnée seule la nuit en pleine forêt depuis la réintroduction des loups et des ours… Et les loups comme les ours rodent…

Avoir le fantasme d'un monde cool et fraternel est une chose, avoir le sens des réalités en est une autre.

Lorsque vous concevez des applications Silverlight qui doivent accéder à des données se trouvant sur un autre domaine c'est en tout cas une réalité bien rugueuse (un message d'erreur) qui vous rappellera que le monde n'est pas un grand Disney Land dans le cas où le serveur n'a pas explicitement autorisé cet accès par un fichier de police d'accès cross-domaine.

Autant le savoir et concevoir ses applications en fonction de cette réalité (et configurer ses serveurs correctement).

Silverlight libère le chemin (Path n'est plus sealed)

Parmi toutes les choses qui ont changé et toutes les nouveautés de Silverlight 4, forcément il y en a qui passent plus inaperçues. Des petits riens qui parfois peuvent changer la vie du développeur. Ici c'est le simple mot "`sealed`" qui a été supprimé dans le code source de la classe `Path`. Un truc insignifiant. Mais riche en perspectives…

Petit mais costaud

Le changement est petit, certes. Minuscule face à l'ajout du `RichTextArea` et de bien autres nouveautés de Silverlight 4. Mais en supprimant "`sealed`" (scellé) l'équipe de SL nous permet de créer nos propres contrôles dérivant de `Path` sans avoir à coder tout depuis zéro, ce qui était le cas jusqu'à lors.

Par exemple, les propriétés de dépendances telles `Fill`, `Stroke`, et toute la logique d'une "*shape*" (forme), tout cela est disponible et exploitable, il suffit juste de fournir le nouveau dessin.

Simple et pratique

Les applications Silverlight sont orientées média au sens large. Le graphisme y tient un rôle essentiel. Tout ce qui peut simplifier la création d'objets graphiques est donc un avantage.

Certes je ne vais pas vous dire que fabriquer ses propres formes est une activité ultra essentielle et extrêmement fréquente. Mais il y a beaucoup d'applications qui nécessitent des symboles par exemple. Disposer d'un peu plus que du `Rectangle` et de l'`Ellipse` c'est tout de même plus confortable !

D'ailleurs Silverlight 4 est fourni avec plusieurs nouvelles formes comme la flèche, des polygones, etc.

Ecrivez vos propres formes

Ecrire ses propres formes fait ainsi partie du travail de mise en place d'une interface visuelle. Aujourd'hui ce travail est simplifié par la possibilité d'écrire des classes

héritant de `Path`. On ne va pas en faire des tonnes non plus, mais si on n'en parle pas, je suis certain que peu de gens se rendront compte que `Path` n'est plus `sealed` !

Pour terminer voici un exemple, extrait du code source d'ailleurs des nouvelles formes proposées avec Silverlight 4. Le triangle est une figure magique, parfaite même (pour les anciens Grecs) et chargée de symbolisme. C'est donc un bon exemple !

```
 1: public class Triangle : Path
 2:     {
 3:         public Triangle()
 4:         {
 5:             CreatePathData(0,0);
 6:         }
 7:
 8:         private double lastWidth = 0;
 9:         private double lastHeight = 0;
10:         private PathFigure figure;
11:
12:         private void AddPoint(double x, double y)
13:         {
14:             LineSegment segment = new LineSegment();
15:             segment.Point = new Point(x + 0.5 * StrokeThickness,
16:                 y + 0.5 * StrokeThickness);
17:             figure.Segments.Add(segment);
18:         }
19:
20:         private void CreatePathData(double width, double height)
21:         {
22:     // in order to fit in our layout slot we need to reduce the size of the stroke
23:             height -= this.StrokeThickness;
24:             width -= this.StrokeThickness;
25:
26:             // needed to avoid a layout loop
27:             if (lastWidth == width && lastHeight == height) return;
28:             lastWidth = width;
29:             lastHeight = height;
30:
31:             var geometry = new PathGeometry();
32:             figure = new PathFigure();
33:             figure.StartPoint =
    new Point(0 + 0.5 * StrokeThickness, height + 0.5 * StrokeThickness);
34:             AddPoint(width, height);
35:             AddPoint(width / 2, 0);
36:             AddPoint(0, height);
37:             figure.IsClosed = true;
38:             geometry.Figures.Add(figure);
39:             this.Data = geometry;
40:         }
```

```
41:
42:         protected override Size MeasureOverride(Size availableSize)
43:         {
44:             return availableSize;
45:         }
46:
47:         protected override Size ArrangeOverride(Size finalSize)
48:         {
49:             CreatePathData(finalSize.Width, finalSize.Height);
50:             return finalSize;
51:         }
52:     }
53:
```

Code 30 - Une forme personnalisée (triangle)

Le principe est simple : sous-classez `Path`, ajoutez la logique que vous voulez pour le rendering de votre forme, et fournissez des surcharges pour les méthodes `MeasureOverride` et `ArrangeOverride`. C'est tout.

De la tête à Toto aux montres molles de Dali, le choix vous appartient !

Conclusion

Savoir donner de l'importance à l'insignifiant est souvent la clé des idées géniales… L'absence d'un petit mot dans le source de Silverlight 4 ouvre des voies nouvelles. Cela méritait bien un billet !

Webcam, Capture et Prise de vue…

Silverlight 4 nous a amené beaucoup de nouveautés intéressantes. La gestion de la Webcam en fait partie. Comment activer un camera, comment afficher un flux vidéo, comment permettre des capture d'image et les sauvegarder en jpeg ? Autant de question auxquelles ce billet va tenter de répondre !

L'exemple

J'aime bien commencer par l'exemple car on comprend mieux la suite…

Cliquez sur le titre de ce chapitre pour accéder au billet original et jouer avec l'exemple live.

La démo ci-dessous permet de faire toutes les choses présentées en introduction. Elle n'est pas parfaite, c'est juste une démo à laquelle j'ai ajouté plusieurs fonctions au fur et à mesure. Donc pas un modèle, ni de programmation, ni de design. Mais enfin, elle fait le job et elle illustrera parfaitement le code (que nous verrons plus bas).

Nota: A nos amis qui lisent le blog avec un flux RSS, sachez qu'aussi pratique que ces flux soient, vous vous privez à chaque fois des démos vivantes que je publie dont vous ne voyez que la balise. N'hésitez à venir sur le billet, ici sur Dot.Blog, pour bénéficier de la totalité du show !

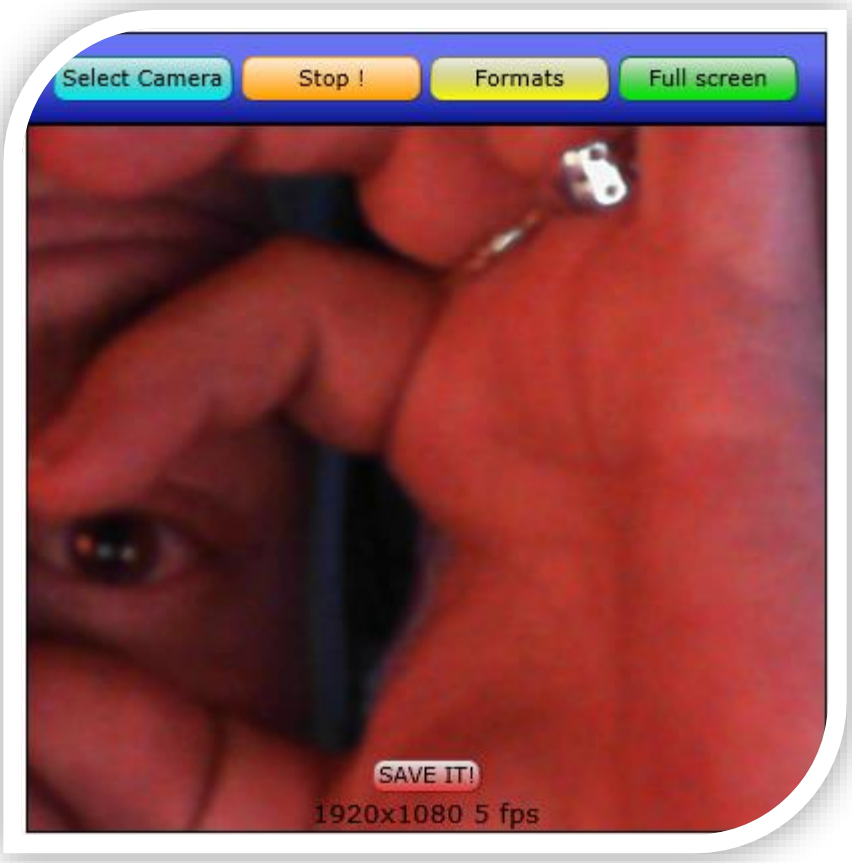


Figure 38 - Prendre des photos

Quelques explications et scenarii à tester.

L'application est plus intéressante en mode plein écran, commencez par cliquer sur le bouton vert "`Full Screen`".

Cliquez sur le bouton bleu "`Select Camera`". Un dialogue vous avertira que le grand méchant que je suis veut prendre possession de votre Webcam. Comme l'application est purement locale, n'hésitez pas à répondre par l'affirmative, promis je n'ai pas mis

d'espion, vous pouvez tester l'application même en slip. Ce qui ne m'intéresse pas d'ailleurs (sauf si vous êtes une jolie blonde, là je regrette mon honnêteté ☺). A noter que la sélection de la caméra peut être entérinée pour cette application en cochant la checkbox du dialogue; cela évite d'avoir à répondre sans cesse à la question. Notons aussi que c'est à ce niveau qu'une application réelle pourrait laisser le choix de la caméra à utiliser si plusieurs étaient détectées. Dans la démo j'ai optée pour celle que vous avez indiquée comme "défaut" dans les réglages de votre PC ou Mac.

Cliquez sur le bouton jaune "`Formats`", cela affichera un dialogue (qui arrivera de la gauche) contenant une listbox avec tous les formats reconnus par votre WebCam. Nous verrons plus loin que par une petite requête Linq le programme va choisir automatiquement la configuration qui offre la meilleure résolution (on pourrait faire le contraire, bien entendu).

Maintenant cliquez sur le bouton orange "`Capture`".

Si tout va bien, et selon comment est réglée votre Webcam, vous devriez voir apparaitre une tête que vous connaissez bien : la vôtre !

Au passage, le bouton "`Capture`" s'est transformé en bouton "`Stop`" pour arrêter la capture. De même le texte tout en bas de l'écran rappelle la résolution qui a été choisie par l'application. Enfin un petit bouton rouge centré en bas vous propose "`Shoot it!`".

Lorsque vous aurez recoiffé votre mèche (ou remis un coup de rouge à lèvre), prenez la pause et cliquez sur ce dernier bouton. Clic-clac ! (je fais le bruit dans ce texte car je n'ai pas ajouté de bruitage à l'application). Le bouton rouge se transforme en "`SAVE IT!`"

Cliquez dessus et sélectionner un répertoire où stocker ce merveilleux cliché. (Mes documents, mes images, au hasard).

Allez vérifier dans le répertoire en question, votre photo s'y trouve bien ! Merveilleux non ?

Le code

Plusieurs choses sont à dire sur ce code. Je vais essayer d'être clair sur chaque fonction.

Sélection de la caméra

Pour sélectionner la caméra j'ai fait une méthode qui retourne un objet `VideoCaptureDevice` :

```
1: private VideoCaptureDevice getCamera()
2: {
3:   if (!CaptureDeviceConfiguration.AllowedDeviceAccess && !
4:               CaptureDeviceConfiguration.RequestDeviceAccess())
5:   return null;
6:   return CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
7: }
```

Code 31 - Sélection de la caméra

Le test sert à éviter de reposer la question que Silverlight vous pose la première fois et ce dans le cas où vous avez coché la checkbox. Ce choix est mémorisé par le plugin et cela évite d'ennuyer l'utilisateur s'il a déjà autorisé l'application à utiliser la caméra.

Si tout est ok, la méthode retourne la device de capture video par défaut.

Obtenir les formats

Voici le code :

```
1: private void btnFormats_Click(object sender, RoutedEventArgs e)
2: {
3:   if (currentCamera == null) currentCamera = getCamera();
4:   if (currentCamera == null) return;
5:   lbFormats.ItemsSource =
6:        (from VideoFormat f in
               currentCamera.SupportedFormats select f).ToList();
7:   lbFormats.ItemTemplate = Helper.GenerateFrom(typeof(VideoFormat));
8:   OpenFormatsAnim.Begin();
9:  }
```

Code 32 - Enumérer les formats supportés par la caméra

Plusieurs "feintes" ici. Je passe la logique qui vérifie si la caméra a déjà été sélectionnée pour arriver à l'affichage des formats : une requête Linq toute simple qui remonte la liste de tous les formats supportés. La feinte la plus intéressante est la ligne 7 où j'attribue à `ItemTemplate` de la `listbox` un mystérieux "`Helper.GenerateFrom(..)`". J'y reviens dans deux secondes. La séquence est bouclée par l'appel à l'animation qui fait apparaître la listbox.

DataTemplate automatique

J'avais commencé le projet sous VS, donc pas terrible pour le templating, et vu la simplicité du code il n'y a pas non plus de datasource pratique pour écrire le `DataTemplate` sous Blend. Bref, je suis parti en code, ma punition c'est d'y rester… Mais autant s'en sortir de façon souple, originale et réutilisable.

Regardons la classe Helper :

```
 1: public static class Helper
 2:     {
 3:         public static DataTemplate GenerateFrom(Type type)
 4:         {
 5:             var pf = type.GetProperties();
 6:             var sb = new StringBuilder();
 7:             sb.Append(
 8:
   "<DataTemplate xmlns=\"http://schemas.microsoft.com/client/2007\">");
 9:             sb.Append(
10:
  "<Border BorderBrush=\"#FF001170\" BorderThickness=\"1\" CornerRadius=
11:             \"5\" RenderTransformOrigin=\"0.5,0.5\">");
12:             sb.Append("<StackPanel>");
13:             foreach (var fieldInfo in pf)
14:             {
15:               sb.Append("<StackPanel Orientation=\"Horizontal\" >");
16:               sb.Append(
17:               "<TextBlock Text=\"" + fieldInfo.Name +
                  "\" Margin=\"1\" />");
18:                 sb.Append(
19:                       "<TextBlock Text=\"{Binding " + fieldInfo.Name
                          + "}\" Margin=\"1\" />");
20:                 sb.Append("</StackPanel>");
21:             }
22:             sb.Append("</StackPanel>");
23:             sb.Append("</Border>");
24:             sb.Append("</DataTemplate>");
25:
26:             return (DataTemplate)XamlReader.Load(sb.ToString());
27:         }
28:     }
```

Code 33 - Générateur de DataTemplate

Cette classe contient une méthode `GenerateFrom` qui prend un type en paramètre. Et elle retourne un `DataTemplate`.

La ruse consiste ici à construire dans un `StringBuilder` toute la syntaxe Xaml d'un `DataTemplate` totalement adapté au type passé en paramètre.

D'abord un `Border` entoure l'ensemble, ensuite un `StackPanel` vertical contiendra la liste des propriétés publiques, enfin, chaque propriété est enchâssée dans un `StackPanel` Horizontal qui contient un premier `TextBlock` affichant le nom de la propriété ainsi qu'un second dont la propriété `Text` est bindée au nom de cette même propriété.

La réflexion est utilisée pour itérer sur l'ensemble des propriétés publiques.

En fin de séquence, avec l'aide d'un `XamlReader` je transforme la chaine de caractères Xaml construire par code en un véritable objet transtypé correctement en `Datatemplate`.

De la génération dynamique de Xaml au runtime rapide et pratique. On peut imaginer plus sophistiqué, réutiliser une sorte de modèle pour les balises de présentation créé sous Blend afin de récupérer un code Xaml moins "brut" niveau look.

N'empêche, cette petite séquence génère des `DataTemplate` à la demande, au runtime, pour n'importe quelle classe. A conserver dans un coin, ça vous servira peut-être.

La capture

Passons à la vidéo. D'abord il faut savoir que les vidéos s'utilisent comme des brosses, et une brosse ça sert à peindre. Donc notre application est en réalité remplie par un simple `Rectangle`. C'est lui qui sera "peint" par la brosse video que nous créerons.

Le choix du format

```
1: var format = (from VideoFormat f in currentCamera.SupportedFormats
2:                 orderby f.PixelWidth * f.PixelHeight descending
3:                 select f).FirstOrDefault<VideoFormat>();
4:  if (format != null) currentCamera.DesiredFormat = format;
```

Code 34 - Choix du format de capture

Une autre requête Linq nous sert ici à sélectionner automatiquement le format offrant la meilleure résolution. On pourrait bien entendu prendre en compte la profondeur en bits de l'image, ou d'autres critères ou bien les traiter autrement (prendre au contraire l'image la plus petite si c'est pour créer un photomaton pour une fiche client par exemple).

Lancer la capture

```
1: currentSource = new CaptureSource();
2: currentSource.VideoCaptureDevice = currentCamera;
3: if ((currentSource.State == CaptureState.Started)
4:         || (currentSource.State == CaptureState.Failed)) return;
5:
6: var videoBrush = new VideoBrush { Stretch = Stretch.None };
7: videoBrush.SetSource(currentSource);
8: rectVideo.Fill = videoBrush; // this is a rectangle
9: currentSource.Start();
```

Code 35 - Lancer la capture

Il faut obtenir un nouvel objet `CaptureSource` puis lui affecter la device de capture sélectionnée.

Quelques tests permettent d'éviter les problèmes - si la capture est déjà commencée ou si la source n'a pu acquérir la device, on annule la séquence par un return. Un vrai programme tenterait d'être plus user friendly en affichant des messages et en essayant de proposer une solution.

Ensuite on crée une `VideoBrush` à laquelle on assigne comme source la `CaptureSource` initialisée au-dessus. Puis on remplit le rectangle, notre toile de ciné en quelque sorte, en attribuant la brosse à sa propriété `Fill`. Et on démarre la capture. Et votre bobine ébahie apparaît à l'écran ☺

On note qu'en utilisant le même procédé on peut remplir des lettres ou des formes avec une video. Ce n'est pas un truc très courant, mais ça peut donner un certain style à un écran d'accueil par exemple.

Prendre une photo

```
   1: private void btnShoot_Click(object sender, RoutedEventArgs e)
   2: {
   3:   if (btnShoot.Tag == null) // no image to save
   4:    {
   5:      if (currentSource == null) return;
   6:      if (currentSource.State != CaptureState.Started) return;
   7:      currentSource.CaptureImageCompleted +=
currentSource_CaptureImageCompleted;
   8:      currentSource.CaptureImageAsync();
   9:      return;
  10:    }
  11:    var sd =
  12:          new SaveFileDialog { DefaultExt = "jpg", Filter = "Jpeg
Images|*.jpg" };
  13:    if (sd.ShowDialog() != true) return;
  14:    try
  15:    {
  16:       using (var fs = (Stream)sd.OpenFile())
  17:        {
  18:          fs.Write((byte[])btnShoot.Tag, 0,
((byte[])btnShoot.Tag).Length);
  19:          fs.Close();
  20:          btnShoot.Tag = null;
  21:          btnShoot.Content = "Shoot it!";
  22:         }
  23:    }
  24:    catch (Exception)
  25:    {... }
  26:
  27: }
```

Code 36 - Prendre une photo

Le bouton de prise de vue sert à deux choses comme nous l'avons vu. Au départ il déclenche le processus d'enregistrement de la vue. Comme ce processus est asynchrone on ne peut pas lancer le dialogue `SaveFile` à la suite puisque la photo ne sera pas encore arrivée.

En revanche dans le code recevant la photo, on place cette dernière dans le `Tag` du bouton. Son titre devient "`Save It!`" et quand on clique dessus, puisque le `Tag` n'est

pas nul il va remplir la fonction de sauvegarde du fichier. Ici il est possible d'afficher le dialogue `SafeFile` car l'appel à ce dernier est bien intégré dans une méthode déclenchée directement par l'utilisateur.

Une application réelle utiliserait plutôt une liste de photos qui s'allongerait à chaque prise et présenterait certainement un bandeau de tous les clichés. C'est depuis ce bandeau que l'utilisateur pourrait choisir quelles photos sauvegarder ou supprimer. Cela serait plus propre. Ma démo ne va pas jusqu'à là bien entendu, d'où la "combine" du `Tag`, un peu spartiate mais qui marche.

De WriteableBitmap à Jpeg

Je vous ai un peu menti… mea culpa, mais c'est pour la bonne cause, pour ne pas encombrer trop le discours.

En réalité dans la gestion de l'événement `CaptureImageComplet` ce n'est pas directement le résultat que je place dans le `Tag` du bouton, car ce résultat est un `WriteableBitmap`, pas un jpeg.

De plus Silverlight ne sait pas encoder du jpeg :-(

Alors il y a encore une feinte !

Ce qui est stocké dans le `Tag` du bouton est une image jpeg correctement encodée. Mais par quelle magie ? … L'utilisation d'une librairie externe.

Il en existe plusieurs, plus ou moins complètes, voire payantes pour certaines.

Ici j'ai choisi `JFCore`, qui est un encoder JPEG simple et rapide. Pour faire tourner le code de l'exemple il vous faudra télécharger JFCore à cette adresse : JFCore.

Attention c'est l'adresse d'un Trunk Subversion, il vous faudra Turtoise ou Ankh ou un outil Subversion pour récupérer le paquet. Attention n°2 : le code est écrit pour SL 2, quand on le charge dans VS il y a une étape de traduction mais tout se passe bien, le code est fiable et passe sans problème. N'oubliez pas de faire passer les projets de la solution en mode Silverlight 5 et .NET 4.5.

Il existe une autre librairie du même genre pour encoder en PNG : Dynamic image generation in Silverlight. A vous de tester.

Le code magique est donc celui qui transforme la `WriteableBitmap` en Jpeg, une fois compris qu'on utilise ici un encodeur à ajouter (JFCore) :

```
   1: void currentSource_CaptureImageCompleted(object sender,
                                    CaptureImageCompletedEventArgs e)
   2:         {
   3:             btnShoot.Tag = null;
   4:             if (e.Result == null) return;
   5:             var width = e.Result.PixelWidth;
   6:             var height = e.Result.PixelHeight;
   7:             const int bands = 3;
   8:             var raster = new byte[bands][,];
   9:
  10:             //Convert the Image to pass into FJCore
  11:
//Code From http://stackoverflow.com/questions/1139200/using-fjcore-to-
encode-silverlight-writeablebitmap
  12:             for (int i = 0; i < bands; i++)
  13:             { raster[i] = new byte[width, height]; }
  14:
  15:             for (int row = 0; row < height; row++)
  16:             {
  17:                 for (int column = 0; column < width; column++)
  18:                 {
  19:                     int pixel = e.Result.Pixels[width * row +
                                                    column];
  20:                     raster[0][column, row] = (byte)(pixel >> 16);
  21:                     raster[1][column, row] = (byte)(pixel >> 8);
  22:                     raster[2][column, row] = (byte)pixel;
  23:                 }
  24:             }
  25:             var model = new ColorModel { colorspace = ColorSpace.RGB
};
  26:             var img = new FluxJpeg.Core.Image(model, raster);
  27:
  28:             //Encode the Image as a JPEG
  29:             var stream = new MemoryStream();
  30:             var encoder =
                  new FluxJpeg.Core.Encoder.JpegEncoder(img, 100, stream);
  31:             encoder.Encode();
  32:
  33:             //Back to the start
  34:             stream.Seek(0, SeekOrigin.Begin);
  35:
```

```
36:             //Get the Bytes and write them to the stream
37:             var binaryData = new Byte[stream.Length];
38:             long bytesRead = stream.Read(binaryData, 0,
                                        (int)stream.Length);
39:
40:             btnShoot.Content = "SAVE IT!";
41:             btnShoot.Tag = binaryData;
42:         }
43:     }
```

Code 37 - Capture image

Je vous laisse regarder (en sautant les parties purement cosmétiques gérant les boutons).

Conclusion

Jouer avec les webcams, prendre des clichés, les enregistrer en Jpeg, tout cela n'est pas si compliqué, mais réclame la connaissance de quelques astuces. J'espère que ce billet vous aura apporté le minimum vital pour commencer à jouer avec cette technologie qui, bien utilisée, peut largement agrémenter des tas de logiciels (comme l'intégration de la photo d'un produit ou d'un client ou d'un salarié directement en quelques clics). Fabriquer un "trombinoscope" qui permet à chacun en appelant l'application de se prendre en photo avec sa webcam au lieu de payer un photographe qui loupera la moitié des gens (partis en extérieur) le jour de son passage (couteux) est un exemple. Vous en trouverez bien d'autres j'en suis certain !

WCF Service pour Silverlight

Le Framework .NET, au travers de Visual Studio, nous offre la possibilité de créer de nombreux types de services distants : des Web services classiques, des WCF services et des "Silverlight-enabled WCF services". Quelle différence y-a-t-il entre ces différents "services" et pourquoi choisir les services particuliers WCF pour Silverlight ?

Web Service classique vs. WCF Service

La première question importante est de savoir quelles sont les différences entre ces deux modes de services distants.

Web Service classique

Le Web service classique est un mécanisme de transfert de données aujourd'hui parfaitement standardisé et largement utilisé quelle que soit la plateforme. Utilisant un support XML pour les données, n'importe quelle plateforme peut consommer un Web service de n'importe quelle autre plateforme. La fantaisie la plus grande qu'on puisse rencontrer est l'adoption de JSON au lieu de XML, ce qui n'entraîne rien d'autre que d'utiliser la bonne librairie.

"L'invention" des Web services fut un bond énorme en avant vers un Internet collaboratif, plus ouvert, et surtout ouvert aux applications qui deviennent des utilisateurs comme les autres du réseau. Les humains peuvent utiliser le Web avec ses pages HTML (peu importe la technique qui sert à les produire) et ses plugins riches (Flash, Silverlight). Les applications deviennent des acteurs du réseau en pouvant elles aussi communiquer, échanger des informations dans un format qui leur être propre et plus facile à interpréter qu'une page HTML.

Avantages

Créer des Web Services "classiques" aujourd'hui, et même sous .NET, a tout son sens car seuls les services se présentant de cette façon sont "universellement" reconnus et utilisables.

Si vous devez créer des services exploitables par des applications tierces, des plateformes différentes, alors il faut utiliser les services classiques.

De plus créer des Web services classiques sous .NET est d'une facilité déconcertante.

Inconvénients

Cette universalité a un prix : Les données pouvant être échangées ou transmises sont limitées à des types sérialisables de façon tout aussi universelle. Les objets complexes .NET ne peuvent pas être transmis directement. Pour des applications .NET de bout en bout, c'est un gros inconvénient et une lourdeur inutile à supporter.

De plus, il existe des variantes, des approches différentes voire même des standards parallèles. Par exemple JSON est un format qui s'oppose au XML classique pour la représentation des données. Néanmoins Silverlight sait comprendre JSON, il reste donc possible d'exploiter ce format particulier de "Web service classique" sans aucun souci côté application SL.

Syntaxe et mise en œuvre

Elle est d'une légèreté sans pareil. Un blogger comparait les services Web "classiques" à un Cesna 150 et les services WCF à un 747. On fait plus de choses et plus fines avec dernier mais l'apprentissage est aussi beaucoup plus difficile…

Pour faire un service Web classique le bloc-notes est suffisant. Un simple fichier avec l'extension `ASMX` et des méthodes décorées avec l'attribut `[WebMethod]` et hop! Le service est immédiatement utilisable…

La mise en œuvre est tout aussi déroutante puisqu'il suffit que le fichier `ASMX` soit accessible sur un serveur IIS et ça marche tout seul.

De nombreux articles ont détaillé la conception des services Web classiques, et ce depuis des années, je renvoie ainsi à Bing le lecteur qui souhaite retrouver les articles français qui lui permettront de mettre en place facilement de tels services.

Services WCF

WCF (*Windows Communication Foundation*) est une brique essentielle du Framework .NET qui fédère de très nombreuses techniques de communication qui existaient auparavant et qui offre une API unifiée et objectivée. En ce sens WCF "annule et remplace" tout ce qui se faisait jusqu'à lors (le graphique ci-dessous montre ces techniques).

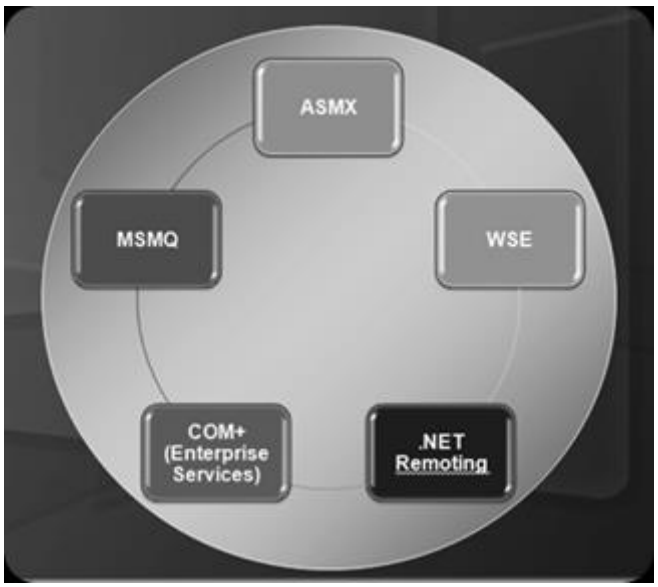


Figure 39 - Les techniques remplacées par WCF

On voit tout en haut du cercle "ASMX". Oui, WCF est conçu pour remplacer totalement les Web Services classiques tels que ceux discutés plus haut. Mais WCF remplace aussi :

MSMQ. Microsoft Message Queuing. Protocole essentiellement basé sur les messages et permettant à des applications consommatrices de recevoir des messages depuis des applications productrices. Ajouté dans Windows 95, MSMQ est toujours disponible sous Windows 7/8 et a même été ajouté à Windows CE depuis la version 3.0.

WSE. Web Services Enhancements. Un toolkit Microsoft pour .NET permettant d'exploiter les dernières spécifications des WS classiques. Releasé pour le Framework 1.0 jusqu'à WSE 3.0 pour VS 2005. Depuis, toutes les nouvelles fonctionnalités sont intégrées dans WCF.

COM+ (Entreprise Services). Ancêtre du Remoting .NET. Première technique Win32 de communication fiable entre applications locales et distantes sous Windows. Efficace et puissant, COM+ était malgré tout une techno complexe et peu aisée à mettre en œuvre.

.NET Remoting. Une technologie assez simple et très puissante permettant de faire communiquer des applications .NET de façon objet. Successeur de COM+ sous .NET, .NET Remoting est typiquement un outil dit "3 tiers" ou "n-tiers" dans la lignée de ce que Borland appelait à l'époque DataSnap, issu lui-même de MIDAS. Java avait aussi son J2EE dans le même esprit.

Comme on le voit ici, WCF est un effort énorme et louable de Microsoft de mettre enfin de l'ordre dans un foisonnement de technologies toutes plus ou moins délicates à manipuler. C'est, entre autres choses, ce qui m'a fait aimer Microsoft depuis sa période .NET : un changement radical d'état d'esprit visant plus d'homogénéité et de cohérence dans les outils de développement et considérant enfin le développeur comme un "client" respectable à qui il faut offrir de beaux outils et de belles plateformes pour le séduire. L'éclatement des techniques et plateformes depuis l'ère Sinofsky ne donne pas cette même sensation de cohérence, de choix judicieux, de confiance en l'avenir. Espérons que le prochain CEO saura résister à l'énorme pression du comité qui cherche un clone pour poursuivre ce qui est en cours et qu'il saura redynamiser le paysage de l'offre globale de l'éditeur en inspirant un vision nouvelle à laquelle nous aurons envie de croire pour nous investir.

Les apports de WCF

WCF n'est pas qu'un agrégat de technologies variées. Il ne s'agit pas de mettre un nom unique sur des API qui resteraient spécifiques ou teintées de leurs origines. WCF est une technologie neuve, à part entière. Si elle remplace de nombreuses autres solutions c'est parce qu'elle les a "réinventées" dans un tout complet. De fait WCF n'est pas que la somme des technologies qu'elle remplace, elle apporte beaucoup de choses en plus de ce qu'elle remplace.

Par exemple, WCF est généralement de 25 à 50% plus rapide que les équivalents de la génération précédente pour les mêmes tâches. WCF est taillé pour être "scalable" et peut prendre en compte des configurations lourdes multiprocesseurs. Les serveurs WCF peuvent être réalisés, au choix, sous la forme de code hébergé sous IIS mais aussi sous forme de Services Windows ou même de simples applications consoles ou autres.

WCF offre un mode de programmation par attribut clair, une configuration souple par fichier XML. WCF offre aussi un `DataContractSerializer` qui autorise la sérialisation d'objets .NET complexes ce qui est un avantage énorme sur les services Web classiques. WCF peut utiliser toute une palette de canaux de transmission pour véhiculer ses messages : HTTP, TCP, MSMQ, Named Pipe, etc.

Avantages

Remplace un fatras de technologies disparates et pas forcément agréables à utiliser. S'inscrit dans la "logique .NET" et "l'esprit .NET". Offre toutes les fonctionnalités pour réaliser des choses simples comme des montages complexes (assure donc la possibilité d'une monté en charge maîtrisée). Autorise deux applications .NET, même de nature différente, à dialoguer et à échanger des objets .NET complexes.

Inconvénients

Il y en a peu, mais un mérite d'être indiqué par honnêteté : WCF, par la surface qu'elle couvre, est une technologie "à tiroir" où il est possible de se perdre un peu. Maîtriser WCF réclame un investissement plus important que faire un Web Service classique, c'est une évidence, surtout si faire du Web Service est l'utilisation principale qu'on fera de WCF.

Néanmoins WCF est totalement intégré à .NET, sa manipulation est facilité par Visual Studio, et lorsqu'il s'agit de programmer des applications Silverlight, purement .NET,

la question du choix entre service classique ou WCF se pose à peine, c'est WCF qu'il faut préférer.

Syntaxe et mise en œuvre

Le côté tentaculaire de WCF fait qu'il serait peu sérieux de vouloir résumer en deux lignes sa syntaxe et sa mise en œuvre comme je l'ai fait plus haut pour les services classiques. Il existe des livres entièrement consacrés à WCF, si vous devez vous orienter vers du 3-tiers (ex .NET Remoting) par exemple, je vous conseille vivement d'investir dans cette littérature !

"Silverlight-Enabled" WCF Services

Quel serait donc ce troisième choix de services WCF "Silverlight-enabled" ?

C'est assez simple, il s'agit juste d'une facilité de développement, de mécanismes de configuration gérés automatiquement par Visual Studio pour rendre le service WCF utilisable facilement avec Silverlight sans trop s'encombrer de toutes les spécificités de WCF. On peut voir cela comme une sorte de « template » mais pas comme une technologie parallèle ou spécifique.

Par exemple, le fichier `Web.Config` de l'application ASP.NET hébergeant le service est modifié de telle sorte que le canal de communication du service soit `basicHttpBinding`, c'est à dire un mode HTTP puisque les applications Silverlight sont des applications Web qui passent par HTTP. Un autre réglage concerne la compatibilité avec ASP, puisque le service est intégré à une application Web ASP.NET. Le paramètre `aspNetCompatibilityEnabled` est ainsi initialisé à `True`.

C'est finalement peu de choses, mais du coup cela facilite grandement l'intégration d'un service dans une application Silverlight.

Conclusion

Silverlight permet la consommation de services distants de plusieurs natures. Du service Web classique au service WCF.

Choisir la technologie du service distant réclame de comprendre les avantages et les inconvénients de chacune des possibilités.

La complexité de WCF tente certains développeurs à préférer les services classiques, beaucoup plus simples à maîtriser à et mettre en œuvre. C'est toutefois se priver de

la richesse de WCF et d'un dialogue objet .NET très supérieur qualitativement à ce que peut offrir un service classique.

Visual Studio, en offrant un mode "Silverlight-enabled" pour les services WCF simplifie le paramétrage et la prise en main. Il suffit juste d'ajouter des méthodes au service… De plus, en choisissant les services WCF vous laissez la porte ouverte à des extensions, des améliorations qui seraient impossibles à réaliser avec des services classiques ce qui obligerait à "casser" le code existant. Mieux vaut partir du bon pied et bien chaussé. Les tongs c'est rapide à mettre, mais ce n'est pas fait non plus pour aller très loin…

Produire et Utiliser des données OData en Atom avec WCF et Silverlight

OData (Open Data Protocol) est un protocole web pour l'interrogation et la mise à jour des données basé sur les technologies web comme HTTP, AtomPub et JSON. OData peut servir à exposer et à accéder aux données de plusieurs types de sources comme les bases de données, les systèmes de fichiers et des sites Web. OData peut être exposé par une grande variété de technologies comme .Net mais aussi Java et Ruby. Côté client OData est accessible par .Net, Silverlight, WP7, JavaScript, PHP, Ruby et Objective-C pour n'e citer que quelques-uns d'entre eux. Cela rend OData très attrayant pour des solutions où les données doivent être accessibles par plusieurs clients de plateformes différentes, dont Silverlight…

Qu'est-ce que OData ?

C'est avant tout un protocole de partage de données basé sur Atom et AtomPub. C'est une "norme" publique publiée par Microsoft sous License OSP (Microsoft Open Specification Promise), ce qui garantit à qui veut s'en servir qu'il ne risque aucune poursuite pour violation de brevet.

Microsoft fournit plusieurs SDK pour utiliser OData avec PHP, Java, JavaScript, WebOS, .Net et IPhone.

OData vs XML

La première question qui se pose à celui qui ne connait pas encore OData est de savoir quelle différence il peut y avoir entre ce standard et un simple service Web

produisant du XML. Après tout ces derniers fonctionnent bien depuis des années, que propose de plus OData ?

La réponse est simple : cela n'a rien à voir 

Un service Web en XML permet certes de fournir des données, voire des méthodes pour les manipuler, mais virtuellement un service Web basé sur XML peut offrir n'importe quel type de service.

Un service web basé sur OData, comme son nom le laisse supposer, est un service très orienté données. Donc proposant de base des méthodes (au sens de moyens) directement adaptées à la manipulation de celles-ci.

S'il fallait faire une comparaison ce n'est pas entre un service web XML et un service web OData qu'il faudrait la faire : OData ressemble en fait bien plus à une surcouche de manipulation de données comme les WCF RIA Services par exemple.

Personnellement je continue à préférer les WCF RIA Services car ils sont beaucoup plus puissants. A mon sens, et en l'état des librairies fournies ainsi que de la norme elle-même OData n'a d'intérêt réel que pour exposer des données lisibles par de nombreux tiers qu'on ne maîtrise pas forcément. Par exemple une grande entreprise pourrait fournir à un accès en lecture seule à son catalogue sous forme de service OData, chacun pouvant créer son propre logiciel d'accès. Une bibliothèque municipale ou nationale, l'INSEE, et autres services d'Etat pourraient tirer parti d'OData : manipuler beaucoup de données en accès lecture c'est son avantage principal.

Lorsqu'il s'agit d'implémenter toute la logique d'une application LOB, OData réserve des surprises (mauvaises), une faiblesse « congénitale » lié à son mode d'interaction par URI largement surpassée par les WF RIA Services.

OData n'en reste pas moins une technologie très intéressante et gageons qu'avec le temps les faiblesses seront gommées.

Pas d'opposition OData/XML, mais une complémentarité

En réalité, OData est une surcouche qui se base sur d'autres standards pour travailler. Notamment, les données peuvent transiter soit en XML très classique, soit dans d'autres formats tels que JSon ou Atom.

On peut donc très bien avoir un service OData utilisant XML.

JSon est aujourd'hui plus "à la mode", question de gout et de facilité de parsing (JSon passe pour moins verbeux, c'est peut-être vrai, mais je n'ai pas vu de grandes différences avec XML dans beaucoup de cas, en revanche il est plus facilement lisible par un humain que XML).

Quant à Atom c'est au départ un format basé sur XML spécialisé pour les flux RSS.

OData et REST

OData exploite l'architecture REST plutôt que SOAP. REST a été inventé par Roy Thomas dans sa thèse "A*rchitectural Styles and the Design of Network-based Software Architecture*". C'est une architecture, donc un ensemble de règles et contraintes. REST est l'acronyme de "*Representational State Transfer*".

C'est une architecture orientée ressource. En fait toute information de base sous REST est appelée une ressource.

REST utilise une couche de communication HTTP ce qui, comme les services Web, lui permet de "passer" à peu près partout. REST utilise les verbes `GET`, `PUT`, `POST`, `DELETE`, ... pour permettre la manipulation des ressources.

Un simple service Web XML utilisant SOAP est donc très différent du point de vue architectural d'un service de données OData. Le premier est très générique, plutôt basé sur un échange de messages entre clients et serveurs, alors que OData est fortement orienté données et permet directement de les manipuler en proposant des moyens clairement définis pour ce faire.

C'est aussi une limitation forte de OData qui se présente comme une passerelle CRUD sur des ressources, ce qui est limitatif, là où un service Web ou les RIA Services exposent aussi des méthodes.

Exposer des données OData

Avant d'utiliser des données via OData encore faut-il trouver une source ! Comme on n'est jamais si bien servi que par soi-même nous allons rapidement mettre en œuvre un serveur OData de test.

Même si notre but est la consommation de services OData dans Silverlight, la production des données est une phase essentielle et intéressante qu'il serait dommage de sauter...

Créer un serveur de données OData

Pour les besoins de cet article je vais créer un nouveau projet application Web sous Visual Studio. Le plus simple possible. Pas d'astuce ici.

Je vais ajouter à ce projet une base de données. Pour la démo j'ai opté pour une base au format SQL CE 4. Il suffit de poser le fichier dans `App_Data` et l'affaire est (presque) jouée. Le Server Explorer de VS propose alors la connexion à la base de donnée dans "*Data Connections*" (j'utilise toujours VS en anglais, je pense que vous retrouverez facilement les équivalents dans la version traduite).

La base de test est typique d'une application LOB : des clients, des commandes, etc. Peu importe, je ne vais utiliser qu'une table ici, celle des clients.

J'ai ajouté ensuite un ADO.NET Entity Model et j'ai choisi de le créer à partir d'une base existante. Le dialogue qui suit propose directement la connexion à notre base de test.

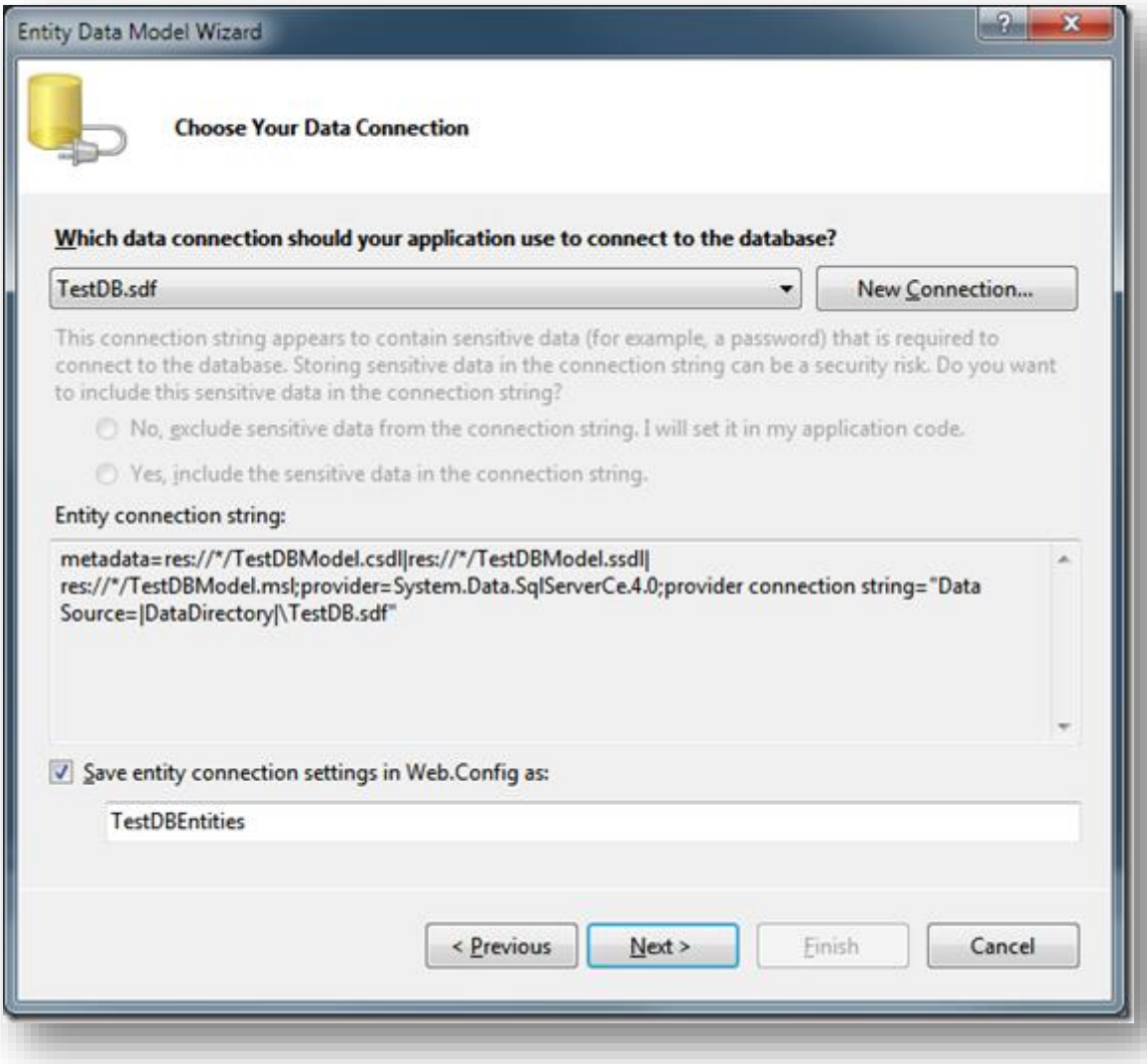


Figure 40 - Créer un Entity Model depuis une connexion SGBD

En quelques clics j'ai pu créer un modèle complet de la base (inutile dans cet exemple je l'avoue, au final je vais tout retirer sauf la table `Customer` !).

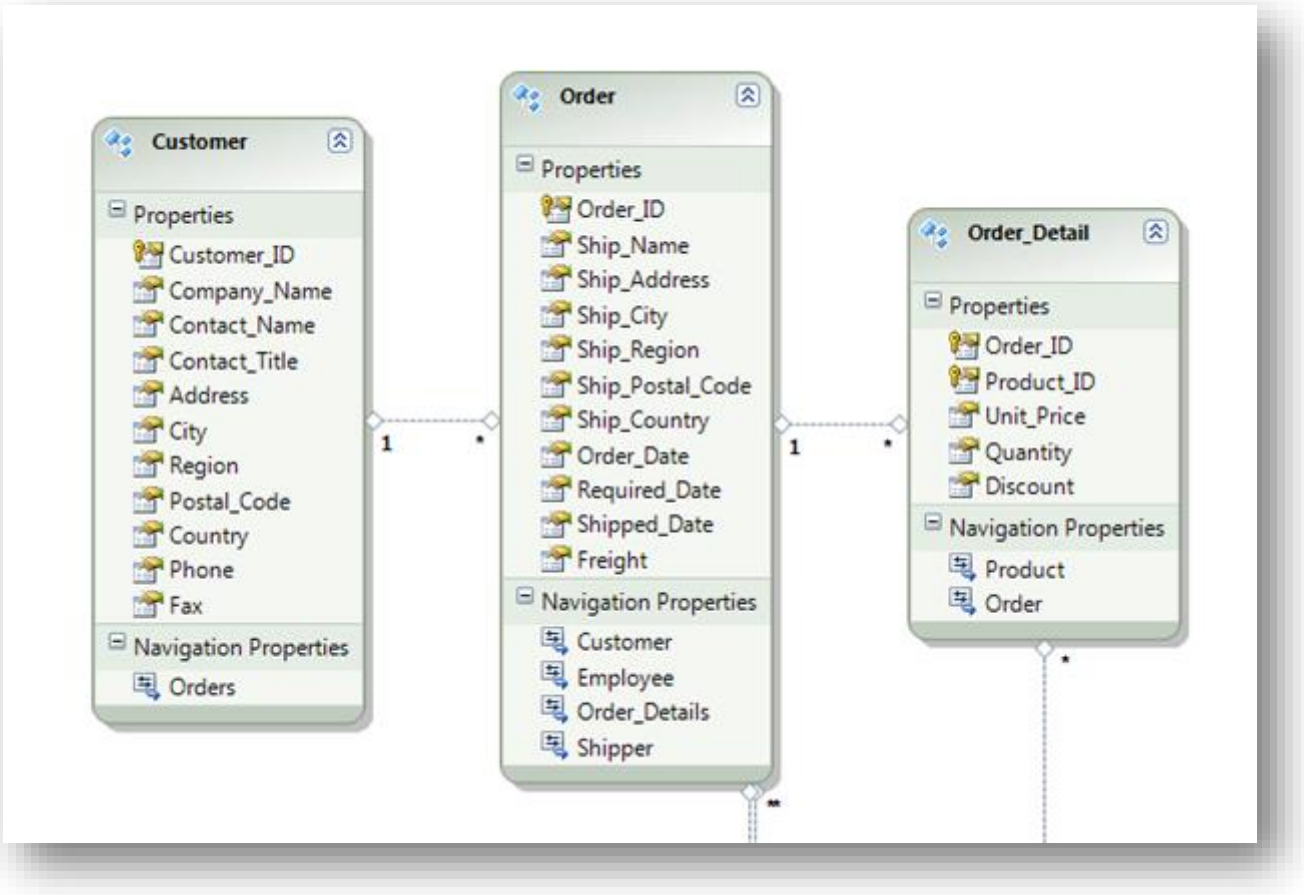


Figure 41 - Un modèle d'entités générées automatiquement

Je n'utiliserai que la table `Customer` pour l'exemple.

Pour que l'application Web fonctionne si elle est déployée sur un serveur il ne faut pas oublier d'ajouter les binaires de SQL CE (ce qui n'est pas nécessaire avec une base SQL Server "normale" puisqu'elle est censée être installée sur le serveur). L'astuce consiste ici à faire un clic-droit sur le nom de l'application Web et de choisir l'entrée "*Add deployable dependencies*" (ajouter les dépendances déployables). Un dialogue apparait permettant de choisir d'intégrer SQL server Compact. Dès qu'on valide le répertoire "References" de l'application se charge de tous les binaires nécessaire au fonctionnement autonome de SQL CE (qui doit être au préalable installé sur votre machine de développement cela va sans dire).

Ne reste plus qu'à exposer ces données en OData !

- Mais comment ? On a lu tout ça pour le savoir !

Pas d'affolement c'est très simple puisque rien ne change ou presque avec un service RIA Services...

J'ai donc ajouté un nouvel item à mon application ASP.NET : un *WCF Data Service*.

Par défaut ce service est vide est ne sait pas quel type de données gérer :

```
using System.Web;

namespace OdataDemo
{
    public class TestDBService : DataService< /* TODO: put your data source class name here */ >
    {
        // This method is called only once to initialize service-wide policies.
        public static void InitializeService(IDataServiceConfiguration config)
        {
            // TODO: set rules to indicate which entity sets and service operations are visible, updatable, etc.
            // Examples:
            // config.SetEntitySetAccessRule("MyEntityset", EntitySetRights.AllRead);
            // config.SetServiceOperationAccessRule("MyServiceOperation", ServiceOperationRights.All);

        }
    }
}
```

Code 38 - Mise en place d'un WCF Data Service

On voit le souligné rouge invitant à indiquer le type de la source. Ici ce seront les entités de notre modèle Entity Framework.

Par défaut le service n'expose aucune données, c'est au développeur de préciser les règles d'accès.

Si l'application est lancée et qu'on accède au service ce dernier retournera le code suivant :

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
-<service xmlns="http://www.w3.org/2007/app" xmlns:app="http://www.w3.org/2007/app"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xml:base="http://localhost:2116/TestDBService.svc/">
  -<workspace> <atom:title>Default</atom:title> </workspace> </service>
```

Dans la méthode `InitializeService` qu'on voit dans la capture ci-dessus je vais ajouter ces règles mais avant tout il y a une astuce à connaître : comme on le voit sur la capture le paramètre "`config`" passé à `InitializeService` est de type `IDataServiceConfiguration`, or cette interface n'expose pas l'une des propriétés dont nous avons besoin pour fixer le protocole... Petit bug ? Je ne saurai dire. En tout cas la feinte consiste à modifier le type du paramètre et à utiliser directement `DataServiceConfiguration` au lieu de l'interface.

Maintenant nous pouvons exposer la table `Customers` et indiquer le protocole à utiliser :

```
namespace OdataDemo
{
    public class TestDBService : DataService<TestDBEntities>
    {
        // This method is called only once to
        // initialize service-wide policies.
        public static void InitializeService(
                                DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("Customers",
                                            EntitySetRights.AllRead);
            config.UseVerboseErrors = true;
            config.DataServiceBehavior.MaxProtocolVersion =
                                        DataServiceProtocolVersion.V2;
        }
    }
}
```

Code 39 - Exposer une ressource OData

Si nous relançons le service, nous voyons apparaitre la ressource "`Customers`". Et si nous accédons à cette dernière (en ajoutant "`/Customers`" au nom du service) Internet Explorer croyant voir un flux RSS sous Atom affichera une série de billets vides (la table `Customers` n'a pas de propriétés identiques à celle d'un flux RSS standard...). En revanche on voir 91 entrées qui correspondent aux 91clients de la base de test...

Rassurez-vous, avec Chrome c'est encore pire, il demande d'installer une extension pour lire les flux RSS.

En demandant de voir le code source de la page nous pouvons voir les données présentées sous forme d'un flux Atom :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xml:base="http://localhost:2116/TestDBService.svc/"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
      xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Customers</title>
  <id>http://localhost:2116/TestDBService.svc/Customers</id>
  <updated>2012-08-04T20:43:32Z</updated>
  <link rel="self" title="Customers" href="Customers" />
  <entry>
    <id>http://localhost:2116/TestDBService.svc/Customers('ALFKI')</id>
    <title type="text"></title>
    <updated>2012-08-04T20:43:32Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Customer" href="Customers('ALFKI')" />
    <category term="TestDBModel.Customer"
         scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:Customer_ID>ALFKI</d:Customer_ID>
        <d:Company_Name>Alfreds Futterkiste</d:Company_Name>
        <d:Contact_Name>Maria Anders</d:Contact_Name>
        <d:Contact_Title>Sales Representative</d:Contact_Title>
        <d:Address>Obere Str. 57</d:Address>
        <d:City>Berlin</d:City>
        <d:Region m:null="true" />
        <d:Postal_Code>12209</d:Postal_Code>
        <d:Country>Germany</d:Country>
        <d:Phone>030-0074321</d:Phone>
        <d:Fax>030-0076545</d:Fax>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://localhost:2116/TestDBService.svc/Customers('ANATR')</id>
    <title type="text"></title>
    <updated>2012-08-04T20:43:32Z</updated>
    <author>
      <name />
```

```
    </author>
    <link rel="edit" title="Customer" href="Customers('ANATR')" />
    <category term="TestDBModel.Customer"
           scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:Customer_ID>ANATR</d:Customer_ID>
        <d:Company_Name>Ana Trujillo Emparedados y helados</d:Company_Name>
        <d:Contact_Name>Ana Trujillo</d:Contact_Name>
        <d:Contact_Title>Owner</d:Contact_Title>
        <d:Address>Avda. de la Constitución 2222</d:Address>
        <d:City>México D.F.</d:City>
        <d:Region m:null="true" />
        <d:Postal_Code>05021</d:Postal_Code>
        <d:Country>Mexico</d:Country>
        <d:Phone>(5) 555-4729</d:Phone>
        <d:Fax>(5) 555-3745</d:Fax>
      </m:properties>
    </content>
  </entry>
  <entry>
...
```

Code 40 - Code XML du flux de données OData

Bref, ici nous savons que le service fonctionne et qu'il répond.

Contrôler les données transmises

Donner l'accès, même en lecture seule (ce que la gestion des droits peut faire dans l'initialisation du service), n'est pas toujours suffisant. Il peut être nécessaire de filtrer ou contrôler les données avant de les servir au client. De nombreuses situations réclament un tel filtrage. La première est de limiter la taille du set retourné. La seconde peut être de cacher certaines données à certains utilisateurs (gestion de rôle débouchant sur des accès partiels aux informations par exemple), etc.

Dans notre exemple nous publions la table "`Customers`" et nous souhaitons limiter l'accès aux clients (ceux de la base de données) se trouvant en France.

Il suffit de détourner les requêtes automatiques du service OData pour y insérer le filtre. Cela s'effectue de la façon suivante :

```
[QueryInterceptor("Customers")]
public Expression<Func<Customer,bool>> OnQueryCustomers()
{
    return c => string.Compare(c.Country, "France",
                    StringComparison.InvariantCultureIgnoreCase) == 0;
}
```

Code 41 - Filtrer les données OData au niveau du serveur

L'attribut permet d'indiquer quelle ressource on souhaite filtrer, le code de la méthode créé le filtrage (ici sur le nom du pays).

C'est vraiment très simple non ?

Il est vrai que je n'aime pas trop le modèle de programmation par « effet de bord ». Il n'y a pas ici de lien direct entre la requête et son résultat, il faut un attribut qui, on le suppose, va faire correctement son travail en insérant notre code dans le déroulement d'un autre code dont nous ne savons rien. C'est pratique, mais le côté magique fait craindre une perte de maîtrise du flux de traitement des données. Dans une démo cela ne pose aucun problème, tout est merveilleux … mais dans la réalité cette approche « lointaine » avec les processus fournissant réellement les données peut s'avérer gênante voire paralysante (pas de maîtrise du flux de données donc).

Encore faut-il maintenant consommer ces données côté Silverlight !

Le client Silverlight

Après avoir ajouté une application Silverlight à la solution en cours, il faut ajouter la référence au service :

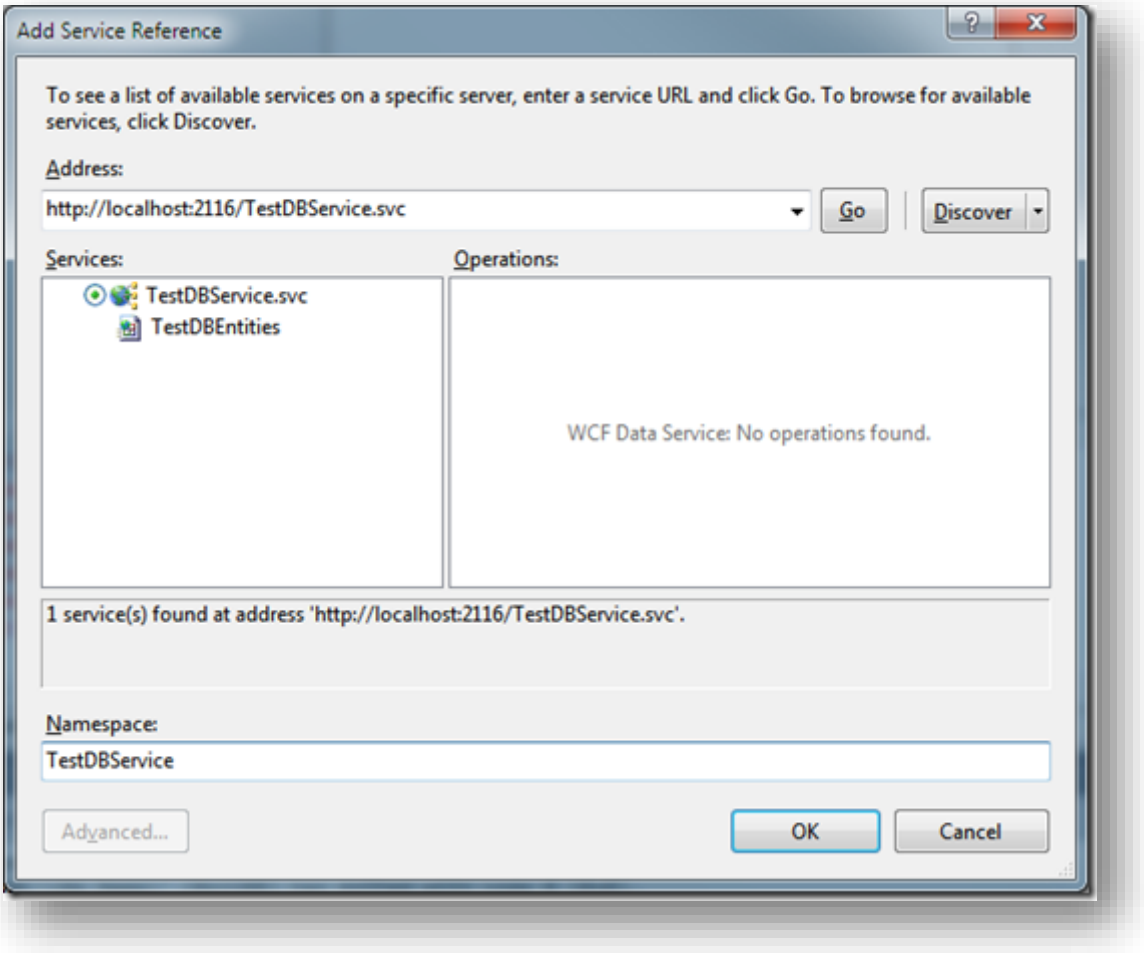


Figure 42 - Ajouter le service OData au projet Silverlight

Le reste est automatique puisque tout ce qu'il faut pour accéder au service est généré dans l'application Silverlight.

Visuellement l'application se limite à une simple `DataGrid` et un bouton "`Load`". Quand le bouton est cliqué les données sont chargées depuis le service distant et affichées dans la grille.

Le code Xaml de la MainPage est le suivant :

```
<UserControl x:Class="Client.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="800" d:DesignWidth="1024"
     xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk">

    <Grid x:Name="LayoutRoot" Background="White">
        <sdk:DataGrid AutoGenerateColumns="True" Height="585"
                HorizontalAlignment="Left" Margin="12,12,0,0" Name="dataGrid1"
                VerticalAlignment="Top" Width="1000" />
        <Button Content="Load" Height="23" HorizontalAlignment="Left"
            Margin="937,614,0,0" Name="button1" VerticalAlignment="Top"
            Width="75" Click="button1_Click" />
    </Grid>
</UserControl>
```

XAML 11 - La fiche de test OData

Je ne vous mentais pas, c'est vraiment minimaliste !

Quant au code C# le voici :

```
public partial class MainPage : UserControl
   {
       public MainPage()
       {
           InitializeComponent();
       }

       private DataServiceCollection<TestDBService.Customer> customers;


       private void LoadCustomers()
       {
           var url = "http://localhost:2116/TestDBService.svc/";
           var uri = new Uri(url, UriKind.Absolute);
           var service = new TestDBService.TestDBEntities(uri);
           customers = new DataServiceCollection<Customer>(service);
           var query = service.Customers;
           customers.LoadCompleted +=
                  new EventHandler<LoadCompletedEventArgs>(customers_LoadCompleted);
           customers.LoadAsync(query);
       }

       void customers_LoadCompleted(object sender, LoadCompletedEventArgs e)
       {
           if (e.Error!=null)
           {
               MessageBox.Show(e.Error.Message);
               return;
           }
           dataGrid1.ItemsSource = customers;
       }

       private void button1_Click(object sender, RoutedEventArgs e)
       {
           LoadCustomers();
       }
   }
```

Code 42 - Accéder aux données OData

On voit qu'ici aussi il n'y a pas de quoi s'affoler...

Une variable de type `DataServiceCollection` typée selon la classe "`Customer`" est créée. Elle contiendra les données retournées.

Le clic sur le bouton déclenche la méthode `LoadCustomers()` dont le code est facile à suivre :

L'Uri est créée et une instance du service l'est aussi en lui passant l'Uri en paramètre. Ensuite c'est une instance de la `DataServiceCollection` qui est créée. Une requête est créée comme étant l'ensemble des données de la ressource `Customers` (filtrée par le serveur, mais ça le client ne peut ni le savoir ni le changer). Nous aurions pu créer une requête LINQ plus complexe, mais ce n'est pas le sujet de cet article. Donc autant rester simple.

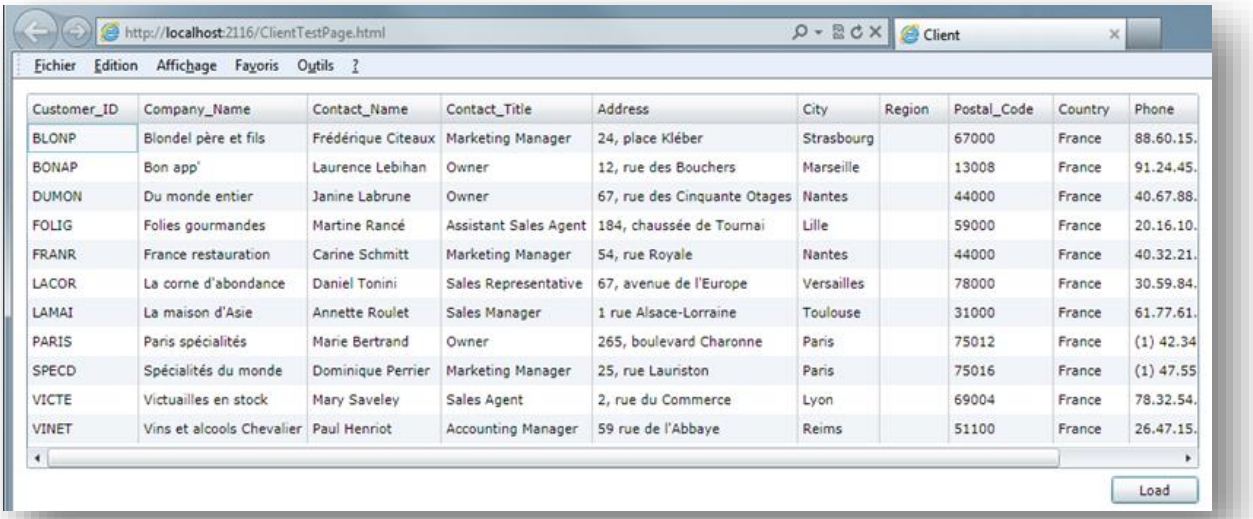
Puis le "`LoadCompleted`" de la `DataServiceCollection` est pointé vers la méthode qui accusera réception des données quand elles arriveront (c'est de l'asynchrone, ne l'oublions pas... Avec les nouveaux mots clé `await` et `async` de la version actuelle de C# cette programmation serait beaucoup plus limpide).

Enfin, la demande est faite à la collection de charger le résultat de la requête. A partir de là le service est interrogé, le serveur va répondre, traiter la requête, le code de filtrage que nous avons ajouté limitera la réponse aux clients se trouvant en France, et la collection de données sera renvoyée à Silverlight.

C'est là que Silverlight et tout son code simplifiant le travail attrapera les données dans ses petits bras musclés, moulinera tout cela, et déclenchera la méthode customers_LoadCompleted. Nous n'ajoutons que le strict minimum vital : tester si il y a ou non une erreur. Dans l'affirmative nous affichons le message d'erreur, dans la négative l'`ItemsSource` de la grille de données est initialisée pour pointer la `DataServiceCollection`.

Ce n'est qu'un code de démonstration et non un exemple pour une application de production.

En relançant l'application Web (dont nous avons positionné la page de démarrage à la page test de l'application Silverlight et non plus sur le service) nous obtenons, après un clic sur le bouton "`Load`", l'affichage suivant :



Customer_ID	Company_Name	Contact_Name	Contact_Title	Address	City	Region	Postal_Code	Country	Phone
BLONP	Blondel père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg		67000	France	88.60.15.
BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille		13008	France	91.24.45.
DUMON	Du monde entier	Janine Labrune	Owner	67, rue des Cinquante Otages	Nantes		44000	France	40.67.88.
FOLIG	Folies gourmandes	Martine Rancé	Assistant Sales Agent	184, chaussée de Tournai	Lille		59000	France	20.16.10.
FRANR	France restauration	Carine Schmitt	Marketing Manager	54, rue Royale	Nantes		44000	France	40.32.21.
LACOR	La corne d'abondance	Daniel Tonini	Sales Representative	67, avenue de l'Europe	Versailles		78000	France	30.59.84.
LAMAI	La maison d'Asie	Annette Roulet	Sales Manager	1 rue Alsace-Lorraine	Toulouse		31000	France	61.77.61.
PARIS	Paris spécialités	Marie Bertrand	Owner	265, boulevard Charonne	Paris		75012	France	(1) 42.34
SPECD	Spécialités du monde	Dominique Perrier	Marketing Manager	25, rue Lauriston	Paris		75016	France	(1) 47.55
VICTE	Victuailles en stock	Mary Saveley	Sales Agent	2, rue du Commerce	Lyon		69004	France	78.32.54.
VINET	Vins et alcools Chevalier	Paul Henriot	Accounting Manager	59 rue de l'Abbaye	Reims		51100	France	26.47.15.

Figure 43 - L'affichage des données OData par l'application Silverlight

C'est du brut de fonderie, mais ça marche. Nous avons requêté et reçu des données distantes produites par un service OData...

Bien entendu, selon le même scénario et puisque nous avons mis la table `Customers` en mode d'accès read/write (au niveau de l'initialisation du service dans le code côté serveur) nous pourrions extrapoler jusqu'à gérer la modification des données, la `DataGrid` pouvant être directement utilisée dans ce sens.

Ce n'est qu'un chapitre et non un livre en soi, il faut bien que je mette une limite...

L'aspect sécurité

D'autant qu'il faut dire un mot sur la sécurité.

Il semble en effet bien imprudent d'ouvrir comme cela les vannes de ses précieuses données sans visiblement qu'aucun mécanisme d'identification du client ne soit en place.

Parfois cela est voulu : publication de données, de statistiques en lecture seule, d'un catalogue de produits, d'un flux d'informations etc.

Souvent en entreprise ce n'est pas du tout souhaitable. L'accès aux données doit être contrôlé.

Dans un tel cas il est nécessaire de pouvoir identifier chaque client avant d'accepter sa requête. Il existe plusieurs solutions, la sécurisation des services WCF est un sujet délicat et pointu.

C'est pourquoi plutôt que de bâcler un exemple peu réaliste je souhaitais surtout ici appuyer sur la nécessité de sécuriser ses services de données. A vous de vous plonger dans les méandres de WCF et de sa sécurité, il existe des tonnes de choses sur ce sujet qui sortirait totalement du cadre de ce billet.

Il y a bien sur des méthodes simples, comme passer des informations d'identification par le header de la requête HTTP entre le client et le serveur. Cela peut suffire pour des données peu vitales ou un service publié en intranet (le client envoie le nom Windows de l'utilisateur et le serveur vérifie dans une liste que cet utilisateur est autorisé par exemple). Pour une véritable sécurisation de niveau professionnelle d'un service en lecture/écriture publié sur le Web, c'est une autre affaire. Consulter un expert en sécurité est alors le meilleur conseil qu'un expert Silverlight peut vous donner !

Conclusion

Nous avons créé ici un serveur de données OData ainsi qu'un client Silverlight sachant les consommer et les présenter à l'utilisateur.

C'est peu... mais c'est énorme !

Le but de cet article n'est pas d'être un cours, ni sur OData, ni rien d'autre. Il a juste pour objectif de vous montrer à quel point la création de données OData est simple tout autant que de consommer de telles données.

Si je vous ai donné l'eau à la bouche, si j'ai réussi à faire en sorte qu'OData n'est plus aussi mystérieux pour vous, alors je serai déjà pleinement satisfait, je ne visais rien d'autre !

La solution complète Visual Studio 2010 (avec la base de test SQL CE 4) est téléchargeable ici :

OdataDemo.zip

Intégrer Bing Maps dans vos applications Silverlight

Intégrer un outil aussi puissant que Bing Map peut transformer une application banale en une application extraordinaire… La puissance d'un système géographique mondial, la fascination que produit les zooms (à tel point que même les journaux télévisés utilisent le principe de plus en plus souvent pour localiser un événement) tout cela dans vos applications Silverlight c'est possible ! On pourrait croire que c'est compliqué, même pas…

L'exemple live

Ci-dessous se trouve une application Silverlight qui "embarque" un espace Bing Maps tout aussi utilisable que si vous alliez sur le site de Bing Maps. Pour le fun (et la démo) j'ai pointé directement un endroit … révolutionnaire. J'ai ajouté un pushpin pour bien marquer l'endroit et si vous passez le curseur sur le pushpin vous pourrez même lire en surimpression dans un phylactère un texte qui donne un indice de plus sur le lieu !

Une fois que vous aurez deviné de quel endroit il s'agit, essayer les boutons en bas, notamment celui qui permet d'afficher la toolbar de navigation. Et amusez-vous avec Bing Map comme sur le site original. Mais rappelez-vous : tout cela se passe à l'intérieur d'une application Silverlight écrite en quelques lignes ! Regardez aussi en bas à droite les coordonnées géographiques exactes de l'endroit où se promène votre souris.

Bien que la technique démontrée soit toujours valable, la clé temporaire de test pour le développement utilisée pour créer cet exemple n'est plus valable (alors que le site gérant les clés indique le contraire). Voilà un mauvais choix car l'exemple est cassé et Bing nous indique que les informations de crédit du compte ne sont pas valables. Dommage pour Microsoft, de telles pingreries coutent souvent cher en image de marque. Ici c'est un bel exemple vendeur qui ne … vend plus puisqu'il est cassé…

Les lecteurs pourront toutefois utiliser le code source fourni et obtenir une clé de développement pour faire tourner l'exemple. On remarquera qu'en dehors du bandeau indiquant le problème de licence l'exemple reste malgré tout utilisable (le phylactère marche, la position montrée est bien celle choisie, les checkbox font bien apparaitre les éléments de l'UI, etc).

Figure 44 - Un bel exemple (mais cassé !)

Je veux le faire aussi !

Je vous comprends, c'est tellement classe ! … Enfin dans sa version qui marche car là, c''est moyennement excitant je peux le comprendre.

Télécharger le kit

D'abord il faut télécharger le kit de développement pour Silverlight. Un simple setup automatique qui créera un répertoire dans *Program Files* contenant les dll nécessaires à ajouter à vos projets ainsi qu'un CHM d'aide.

Première étape donc : le Bing Maps Silverlight Control SDK

Obtenir une clé de développement

La fameuse clé !

Pour que le plugin Bing Maps fonctionne il faut lui donner une clé de développement et un nom d'application. Si vous avez un Windows Live ID c'est très rapide. Lorsque j'ai créé l'exemple vous aviez le droit à 5 clés. Désormais c'est trois seulement. Chaque clé est associée à une application alors ne les grillez pas toutes ! (Vous

remarquerez ma générosité, j'en ai grillé une pour l'appli exemple de ce billet, autant d'abnégation est à souligner, surtout qu'aujourd'hui on a le droit à 3 clés et non plus 5 et qu'en plus l'exemple ne marche plus !).

Si vous n'avez pas de Windows Live ID bien… il faudra en créer un. De toute façon c'est un compte que tout utilisateur d'outils Microsoft se doit d'avoir car c'est un moyen d'identification utilisé partout dans les sites Microsoft, des forums aux abonnements MSDN et même récemment pour l'activation de VS.

Bref, en vous rendant sur le Bing Maps Portal utilisez la fonction "Create a Bing Maps account"

Une fois le compte créé, vous pourrez créer des clés, jusqu'à 3 donc. Une clé s'obtient en indiquant un nom d'application ainsi que l'URL de cette dernière. Vous devez connaître l'URL avant de demander la clé (un détail important).

Passons aux choses sérieuses

Vous avez installé le SDK, vous avez obtenu une clé pour votre application, tout est ok. Comment ajouter une Map dans l'application ?

Une Map tout de suite !

J'ai presque honte de faire un tutor sur ça…

Ajouter en référence à votre projet les deux DLL fournies avec le SDK. Il s'agit de `Microsoft.Maps.MapControl.Common.dll` et `Microsoft.Maps.MapControl.dll`. Le répertoire s'appelle `Bing Maps Silverlight Control` et se trouve dans `Programmes` (ou `Programmes (x86)` si vous êtes en 64 bits).

Sous Blend, parce que c'est plus cool et plus facile, allez dans l'onglet *Assets*, cliquez sur le nœud *Location* et cliquez sur la DLL des contrôles Maps que vous venez d'ajouter en référence. Vous verrez un ensemble de nouveaux contrôles.

Choisissez le contrôle `Map` et posez-le sur la surface de design dans votre `LayoutRoot`.

Ne reste plus qu'à saisir votre clé et le nom de l'appli dans les propriétés du contrôle ou directement dans la balise Xaml et c'est tout : F5 et ça marche !

En trois clics l'affaire est donc dans le sac… Impressionnant non ?

Les améliorations

Après, tout est question de destination de l'application. Peut-être voudrez-vous ajouter des images géolocalisées, des pushpins comme je l'ai fait dans l'exemple, réagir aux mouvements de souris (comme je l'ai fait pour afficher les coordonnées), peut-être voudrez-vous construire votre propre barre de navigation, afficher ou cacher l'échelle, ajouter des polygones (pour marquer un bâtiment par exemple), des textes, etc, etc…

Les possibilités sont très nombreuses et faciles à mettre en œuvre (certaines réclameraient néanmoins des exemples, hélas absents du SDK, et le CHM n'est pas toujours suffisant).

Par exemple la mise en route de l'application de démo ressemble à cela :

```
 1: private void MainPage_Loaded(object sender, RoutedEventArgs e)
 2: {
 3:     VisualStateManager.GoToState(this, "NormalState", false);
 4:     BingMap.Center = new
          Microsoft.Maps.MapControl.Location(48.85318, 2.36911, 120);
 5:     BingMap.Culture = "fr-FR";
 6:     BingMap.ZoomLevel = 19;
 7:     var pin = new Microsoft.Maps.MapControl.Pushpin
 8:                     { Location = BingMap.Center, Content = "!" };
 9:     pin.MouseEnter += pin_MouseEnter;
10:     pin.MouseLeave += pin_MouseLeave;
11:     pin.Effect = new System.Windows.Media.Effects.DropShadowEffect();
12:     BingMap.Children.Add(pin);
13:     BingMap.MouseMove += BingMap_MouseMove;
14: }
```

Code 43 - Personnalisé l'affichage Bing Map

Le VSM sert uniquement à montrer ou cacher le phylactère. Vous noterez que sa position est fixe je n'ai pas compliqué les choses en faisant exactement pointer la pointe de la bulle sur le pushpin, ce n'est qu'une démo ;-)

En ligne 4 je positionne la Map pour qu'elle soit centrée sur un point précis (La place de la Bastille pour nos amis belges, canadiens, africains du sud comme du nord et autres amis lointains qui lisent aussi beaucoup ce blog et qui ne savent pas forcément que c'est le symbole de la révolution française de 1789 et que "ça ira, ça ira" était une chanson révolutionnaire de cette époque).

En ligne 5 j'ai "tenté" d'indiquer la culture que Bing Maps est sensé utiliser. Ça reste en US tout le temps, une fausse manip, un oubli de ma part, un bug, un manque de doc ? Si l'un d'entre vous à la réponse les commentaires du billet sont là pour ça !

En ligne 6 je fixe le niveau de zoom exact.

En lignes 7 et 8 je créée un pushpin. C'est facile il suffit de lui donner une localisation (longitude, latitude et altitude). On peut même afficher un contenu (un point d'exclamation ici).

En ligne 9 et 10 j'attrape les événements de `mouse enter` / `leave` du pushpin, ce qui commandera le changement d'état de l'application via le VSM pour afficher ou cacher la bulle.

En ligne 11 j'en profite pour appliquer un effet (un drop shadow) au pushpin.

En ligne 12 j'ajoute le pushpin aux enfants visuels de la Map.

Rien de bien extraordinaire donc.

Le mouseMove de la Map est lui aussi géré pour afficher les coordonnées souris :

```
1: private void BingMap_MouseMove(object sender, MouseEventArgs e)
2:         {
3:             var p =
                BingMap.ViewportPointToLocation(e.GetPosition(BingMap));
4:             txtLocation.Text = "Lat. " +
5:             p.Latitude.ToString("#.#####") + " Long. " +
6:             p.Longitude.ToString("#.#####");
7:         }
```

Code 44 - Récupération de la position de la souris

Il suffit de convertir la position de la souris en une localisation avec `ViewportPointToLocation`. Il existe une méthode pour faire l'inverse. Il est donc très facile de synchroniser des éléments Silverlight par dessus une Map par exemple (ce qui réclame dans un sens ou dans l'autre, de convertir une position x,y Silverlight en une localisation géographique).

Conclusion

Rien ne sert d'en faire trop, vous avez compris le principe. On peut faire des choses simples ou des choses très sophistiquées, c'est juste une question d'imagination. Et aussi d'un peu de temps pour expérimenter.

Bing Maps dans vos applications Silverlight n'est en tout cas ni un rêve, ni une chose complexe. Encore une fois, on retrouve une sale habitude de Microsoft qui complique inutilement les procédures d'enregistrement, de clés, etc, ce qui au final ruine par exemple la belle démo de ce billet. Régler ces problèmes administratifs est plus complexe et moins fiable que les outils créés par Microsoft qui eux sont souvent d'une exceptionnelle qualité. Des équipes de développement au top, une Direction qui fait de mauvais choix, un classique désormais chez Microsoft. C'est vraiment dommage.

Les sources du projet (attention j'ai supprimé ma clé personnelle, l'application ne fonctionnera pas sans, enfin si mais un message vous invitera à utiliser une clé valide. De même l'exécution depuis VS n'est pas apprécié par Bing Map qui marche en revanche très bien lancé depuis Blend – en réalité c'est l'adresse de l'appli utilisant `localhost` dans Blend qui passe, alors que l'adresse complète de VS n'est pas reconnue comme valable pour la clé enregistrée) : SLBingMap.zip (62,13 kb)

Chargement Dynamique de DLL

Le pendant d'un Framework réduit au strict minimum pour être caché dans un plugin est qu'une application Silverlight devient vite un peu grassouillette dès qu'elle commence à faire autre chose qu'afficher deux carrés animés pour une démo… Pour compenser la maigreur du plugin il faut en effet ajouter de nombreuses DLL au projet qui enfle alors très vite. Et qui dit taille qui grossit dit téléchargement long. Et pour une application Web le pire des ennemis, la pire tare qui fait que l'utilisateur zappe avant même d'avoir vu quelque chose c'est bien le temps de chargement, directement lié à la taille de l'application. Les bonnes pratiques (mais aussi le simple bon sens et le sens du commerce) impliquent de réduire la taille des applications pour que le cœur ne soit pas plus gros qu'une image jpeg. Mais le reste ? Il suffit de le télécharger selon les besoins ! Comment ? C'est ce que nous allons voir…

Halte aux applications joufflues !

Comme je le disais en introduction il faut être très attentif à la taille des applications Silverlight qui a tendance à devenir très vite importante dès lors qu'il s'agit d'une vraie application faisant quelque chose d'utile.

Rappelons que Silverlight est avant tout un mode de conception conçu pour le Web. Et que le Web c'est très particulier, c'est un *esprit* très différent du desktop. J'aimerais insister là-dessus, mais ce n'est pas le sujet du billet.

Donc rappelez-vous d'une chose : Silverlight c'est du Web, et pour le Web les applications doivent être **légères**. Légères graphiquement (séduisantes, aérées), légères fonctionnellement (on ne refait pas une grosse appli de gestion en Silverlight c'est stupide, sauf en Intranet câblé en gigabit), légères en taille.

Pour conserver une taille raisonnable à une application Silverlight il y a plusieurs choses à respecter en amont (la légèreté évoquée ci-avant) et plusieurs techniques à appliquer en aval.

Par exemple si l'application utilise des ressources nombreuses ou encombrantes (images, vidéos, musiques…) elles doivent être téléchargées en asynchrone et non pas être placées dans le fichier Xap.

Autre exemple, ne pas tout faire dans une même application. Si le fonctionnel à couvrir est assez gros, mieux vaut créer des applications différentes qui s'appellent les unes les autres ou sont appelées via un menu fixe.

On tirera aussi grand avantage à utiliser la mise en cache des assemblages (option qui réclame quelques manipulations que nous verrons dans un prochain billet).

On peut aussi utiliser MEF et le lazy loading, désormais disponible sous Silverlight. C'est un excellent choix d'architecture, et j'envisage aussi un prochain billet sur le sujet.

Mais on peut ponctuellement et sans impliquer l'utilisation d'un Framework de type MEF tirer avantage de quelques lignes de codes pour charger, à la demande, des assemblages stockés en divers endroits sur le Web. C'est ce que nous allons voir maintenant.

L'exemple live

Quelques mots sur l'exemple (vous ne pourrez y jouer qu'une fois ou bien il faudra faire un F5 dans le browser pour recharger la page) :

Figure 45 - Une jolie application...

Le but

Nous avons une application principale qui sait faire beaucoup de choses. Tellement de choses qu'il semble plus logique, pour diminuer la taille de l'application principale, de placer les écrans, les modules et autres classes annexes dans des fichiers qui ne seraient chargés qu'à la demande (de l'application ou de l'utilisateur) et en asynchrone.

La démo

L'application principale est un écran d'accueil avec un bouton permettant d'instancier deux `UserControl`. Pour simplifier ces derniers ne sont que des images vectorielles (une fraise et un citron repris des exemples d'Expression Design). Ces `UserControl` sont placés dans une DLL externe qui ne fait pas partie du Xap de l'application que vous voyez en ce moment (en vous rendant sur le billet original en cliquant sur le titre de ce chapitre).

Lorsque vous cliquerez sur le bouton ce dernier va disparaitre et un compteur affichant un pourcentage de progression le remplacera (il s'agit juste d'un texte dans

une fonte très grande). Arrivé à 100% de chargement une fraise et un citron seront affichés.

Selon l'état du serveur, du réseau et de votre connexion, vous aurez, ou non, le temps de voir défiler le pourcentage d'avancement. Parfois même, si quelque chose ne va pas, la fraise et le citron ne s'afficheront pas, et le pourcentage deviendra le message d'alerte "`Failed!`" (Raté !). Si cela arrive, rafraichissez la page par F5 et tentez une nouvelle fois votre chance… Une application réelle prendrait en charge autrement cette situation, bien entendu.

Maintenant à vous de jouer : cliquez !

Bon, on ne joue pas des heures avec ce genre de démo surtout que finalement elle montre le résultat mais pas le "comment".

Comment ça marche ?

Toute la question est là. Comment s'opère la magie et qu'y gagne-t-on ?

On gagne gros !

Répondons à cette dernière question : L'application chargée par ce billet est un Xap qui fait un peu moins de 5 Ko (5082 octets pour être précis, soit 4,96 Ko). La fraise et le citron étant des vecteurs convertis en PNG avec une résolution très satisfaisante ils ne "rentreraient" pas dans un si petit panier vous vous en doutez… En revanche l'affichage de l'application est instantané. Le lecteur qui ne souhaiterait pas cliquer sur le bouton n'aurait donc téléchargé que 4,96 Ko.

Les deux `UserControl` qui contiennent les images sont eux placés dans une DLL (une librairie de contrôles Silverlight) qui pèse exactement 65 Ko (66 560 octets).

Le gain est donc évident même sur un exemple aussi petit : au lieu de près de 70 ko, l'application principale est chargée et opérationnelle avec moins de 5 Ko. Soit 7 % seulement de l'application complète, près de 93 % de téléchargement en moins ! C'est gigantesque !

Appliquée sur plusieurs DLL exposant des `UserControl` (ou des classes, des pages, des dialogues…) beaucoup plus nombreux et beaucoup plus riches, donc plus réels, et même si la "carcasse" (le shell) accueillant tout cela grossirait un peu c'est certain, cette technique permet un gain phénoménal et une amélioration très sensible de l'expérience utilisateur.

La technique

Commençons par créer une nouvelle solution avec un projet de type "*Silverlight Class Library*". Cela produira une DLL. A l'intérieur de cette dernière nous allons créer plusieurs classes. Pour l'exemple vu ici il s'agit de deux `UserControl` affichant chacun une image PNG placée en ressource de la DLL. La première classe s'appelle `Strawberry` et la seconde `Lemon`.

C'est fini pour la partie DLL externe … Compilez, en mode Release c'est mieux, prenez la DLL se trouvant dans `bin\release` et placez là sur votre serveur, dans un répertoire de votre choix. Je passe le fait que ce répertoire doit être accessible en lecture depuis l'extérieur (voir la configuration IIS) et le fait que le serveur doit posséder dans sa racine un fichier `ClientAccessPolicy.xml` autorisant les applications Silverlight à venir piocher dans le répertoire en question. Vous trouverez un exemple de ce fichier dans les sources du projet de démo.

Ne reste plus qu'à créer le shell, la station d'accueil en quelque sorte.

Dans la même solution, ajoutez cette fois-ci une application Silverlight classique (qui produira un Xap donc). La démo fournie est un simple carré de 400x400 avec un texte en bas et un bouton qui déclenche la partie intéressante. Comment ce bouton est placé, comment il s'efface, comment et quand apparaît le pourcentage d'avancement, tout cela est sans grand intérêt ici et le code source fourni vous en dira plus long que des phrases.

Ce qui compte c'est ce qui se cache derrière le clic du bouton.

```
1: var dnl = new WebClient();
2: dnl.OpenReadCompleted += dnl_OpenReadCompleted;
3: dnl.DownloadProgressChanged += dnl_DownloadProgressChanged;
4: dnl.OpenReadAsync(new Uri(
5:  "http://www.e-naxos.com/slsamples/sldyn/SLDynShape.dll"),
6:   UriKind.Absolute);
```

Code 45 - Déclencher un chargement asynchrone d'un module

D'abord nous créons une instance de `WebClient`. Ensuite nous programmons deux événements : `OpenReadCompleted`, le plus important, c'est lui qui sera appelé en fin de téléchargement, et `DownloadProgressChanged`, celui qui permet d'afficher le pourcentage d'avancement ce qui est purement cosmétique (une application réelle pourrait éviter cet affichage si elle lance le téléchargement "en douce" dès que

l'application principale est chargée par exemple, masquant ainsi le temps de chargement).

Enfin, nous appelons la méthode `OpenReadAsync` en lui passant en paramètre l'Uri complète de la dll. Je publie de bonne grâce le code source complet avec l'adresse réelle de l'exemple, alors pour vos essais, soyez sympa utilisez votre serveur au lieu d'appeler ma dll fraise-citron ! Ma bande passante vous remercie d'avance :-)

L'appel à `OpenReadAsync` déclenche le téléchargement de la ressource de façon asynchrone, c'est à dire que l'application peut continuer à faire autre chose. Par exemple la démo affiche le pourcentage de progression du téléchargement, ce qui est une activité comme une autre et qui démontre que l'application principale n'est pas bloquée ce qui est très important pour l'expérience utilisateur (la fluidité, éviter les blocages de l'interface sont mêmes des b.a.-ba en la matière).

Passons sous silence l'affichage du pourcentage, pour arriver directement au second moment fort : le gestionnaire de l'événement `OpenReadCompleted` :

```
1: void dnl_OpenReadCompleted(object sender, OpenReadCompletedEventArgs
e)
2: {
3:             txtPercent.Visibility = Visibility.Collapsed;
4:             try
5:             {
6:                 AssemblyPart assemblyPart = new AssemblyPart();
7:                 extLib = assemblyPart.Load(e.Result);
8:
9:                 var control = (UserControl)
                                    extLib.CreateInstance
10:                                 ("SLDynShape.Strawberry");
11:                control.HorizontalAlignment =
                                HorizontalAlignment.Left;
12:                control.VerticalAlignment = VerticalAlignment.Top;
13:                control.Margin = new Thickness(10, 10, 0, 0);
14:                gridBase.Children.Add(control);
15:
16:                control = (UserControl) extLib.CreateInstance
17:                                                ("SLDynShape.Lemon");
18:                control.HorizontalAlignment =
                                HorizontalAlignment.Right;
19:                control.VerticalAlignment = VerticalAlignment.Bottom;
20:                control.Margin=new Thickness(0,0,10,40);
21:                gridBase.Children.Add(control);
22:            } catch
23:            {
24:                txtPercent.Visibility = Visibility.Visible;
25:                txtPercent.Text = "Failed!";
26:            }
27:        }
```

Code 46 - Réception du module téléchargé et activation

Les choses importantes se jouent en quelques lignes :

Ligne 6 : Nous créons une instance de la classe `AssemblyPart`.

Ligne 7 : nous initialisons la variable `extLib` en demandant à `l'AssemblyPart` de se charger depuis "`e.Result`" c'est à dire de transformer le tas d'octets qui est arrivé en un assemblage .NET fonctionnel.

La variable `extLib` est déclarée comme suit :

```
1: private Assembly extLib;
```

Dans l'exemple elle pourrait être totalement locale à la méthode. Mais la réalité on préfèrera comme cela est montré conserver un pointeur sur l'instance de l'assemblage qui vient d'être chargé. Il sera ainsi possible à tout moment d'y faire référence et d'instancier les classes qu'ils offrent.

Ligne 9 : Nous créons une instance de la fraise en utilisant `CreateInstance` de l'assemblage `extLib` et en passant en paramètre le nom complet de la classe désirée.

Le reste n'est que présentation (positionnement de l'UserControl, ajout à l'arbre visuel, création du citron de la même façon, etc).

Conclusion

Avec trois lignes de code efficace il est donc possible de gagner 93 % du temps de téléchargement d'une application, voire plus. Avec une pointe de ruse en plus on peut même arriver à totalement masquer le chargement si l'utilisateur est occupé par autre chose.

Le chargement étant asynchrone l'application n'est jamais bloquée. En cas de réussite du téléchargement elle peut offrir de nouveaux services, en cas d'échec elle continue à fonctionner normalement (même si certaines fonctions provenant des DLL externes ne sont pas accessibles).

En créant un fichier XML décrivant les DLL externes et leur Uri, fichier qui serait téléchargé en premier par l'application, il est même possible d'ajouter facilement des fonctions à une application Silverlight sans tout recompiler, pendant même que des utilisateurs s'en servent (ceux ayant déjà chargé le XML ne verront les nouveaux modules qu'au prochain accès mais ne seront pas perturbés par la mise à jour).

On voit ici le bénéfice en termes de maintenance, de coût et de temps de celle-ci, qu'elle soit évolutive ou corrective. Avec de l'imagination on peut utiliser le système pour offrir certains modules et pas d'autres selon le profil utilisateur (location de logiciel, achat de fonctions à la demande, etc).
Bref, il s'agit d'une poignée de lignes de code en or.
A vous d'en tirer profit !

Accéder à l'IP du client

Accéder à l'IP du client est parfois nécessaire, Silverlight étant une technologie côté client tournant dans une sandbox cette information ne lui est pas accessible. Dans un précédent billet (non publié dans ce livre) je vous ai montré la façon classique qui via un service Web permet d'obtenir l'IP du client. Aujourd'hui je vous propose une astuce d'un style très différent.

La technique du Web Service

Dans le billet "Silverlight : accéder à l'IP du client" je vous montrais comment créer un Web Service très simple qui permet de retourner à l'application Silverlight l'IP de l'utilisateur.

Cette technique fonctionne très bien même si le petit exemple en début de l'article ne tourne plus...

... Et il ne tourne plus parce que le serveur utilisé en face ne répond plus.

Il ne répond plus parce que je pointais le serveur d'un autre site qui offrait ce service et qui ne l'offre plus.

Il ne l'offre plus car, par force, cela devait générer pas mal de trafic au fur et à mesure que le service était connu...

Et on en arrive à ce qui est gênant avec cette technique : il faut un serveur, un service, et ce dernier consomme de la bande passante alors même qu'on pourrait connaitre l'information d'une autre façon ...

La technique de la page Aspx

Une application Silverlight peut être intégrée dans une page Html, ce qui se pratique souvent, ou bien dans une page ASP.NET, ce que Visual Studio vous propose lorsqu'on intègre l'application SL dans un site Web.

Bien entendu, si la page qui héberge l'application est de type Html, il n'y a pas grand-chose à faire que d'en revenir à la technique du service Web montré dans le billet évoqué plus haut. Mais en dehors de ASP.NET cette page peut aussi avoir été générée avec des langages comme PHP et dans ce cas le même type d'astuce que celle que je vais vous montrer peut s'appliquer.

Si la page est de type ASP.NET (ou technique similaire) il y a une approche très différente de celle du Web Service qui peut être utilisée.

En effet, une page ASPX n'est jamais qu'un "modèle" qui est balayé par ASP.NET, en conjonction avec de l'éventuel code-behind, pour générer, au final, un document HTML qui sera transmis au client. Tout cela implique que le client a fait un appel à votre serveur et que sur ce dernier une application a répondu.

Du coup, si on y réfléchi bien, un "service", au sens large du terme, a déjà été activé avant que le plugin Silverlight n'ait activé votre application. Ce qui se comprend quand on prend conscience que la page HTML est d'abord retournée au butineur et que seulement ensuite celui-ci en interprète le contenu pour l'afficher (et gérer les balises comme `<object>` utilisées pour instancier des objets comme on le fait pour le plugin Silverlight).

N'y aurait-il pas moyen, au lieu de créer un Web Service qui sera appelé ensuite par l'application, d'utiliser ce premier appel à la page ASP.NET ?

Si, bien sûr.

Première étape : configurer le plugin

Comme vous le savez, le plugin Silverlight s'active par une balise `Object` qui contient des sous-balises `<param>` pour paramétrer l'appel à l'application (le fichier Xap), la taille de la région réservée dans la page, la couleur du fond, etc.

Parmi ces balises de paramétrage il y en a une qui n'apparait pas par défaut et qui autorise le passage de paramètres à l'application Silverlight elle-même.

Puisque le code HTML de la page est généré côté serveur par une sorte de service, la page ASPX et le Framework ASP.NET, on peut récupérer l'IP de l'appelant comme dans toute application serveur et en profiter pour générer un bout de HTML qui contiendra cette valeur, passée comme un paramètre dans la balise `Object` qui instancie l'application Silverlight...

Regardons de plus près cette astuce en étudiant le code d'une balise `<object>` :

```
<object data="data:application/x-Silverlight-2," type="application/x-
silverlight-2"
          width="100%" height="100%">
  <param name="source" value="MonAppli.xap" />";
  <param name="onError" value="onSilverlightError" />
  <param name="background" value="white" />
  <param name="minRuntimeVersion" value="4.0.50826.0" />
  <param name="autoUpgrade" value="true" />

  <param name="initParams"
         value=
         "userIp=<%Response.Write(Request.Params["REMOTE_ADDR"]);%>"
   />

  <!-- code si le plugin SL n'est pas installé -->
</object>
```

Code 47 - Paramètres du plugin Silverlight

Ce qui nous intéresse ici se trouve à la fin, la balise `<param>` pour le paramètre "`initParams`". Sa "`value`" est constituée normalement d'une suite "`paramètre=valeur`" séparés par des virgules. On peut y placer ce qu'on veut du moment qu'on respecte cette syntaxe.

Ici, nous avons passé un paramètre "`userIp`" qui est égal à ... une incrustation C# ! Cela marcherait avec du VB de la même façon. Et cette incrustation utilise `Request.Params["REMOTE_ADDR"]` pour récupérer l'adresse IP de l'appelant. Puis elle utilise `Response.Write` pour écrire cette valeur directement dans le flux HTML, donc après le signe égal du paramètre.

Au final le client reçoit un HTML bien formé contenant une balise `Object` qui instancie votre application SL à laquelle est passée la série de paramètres contenu dans "`initParams`", donc la valeur en clair de l'IP de l'appelant.

Dans un second temps le browser va interpréter la page pour l'afficher, il tombera sur la balise `Object` et passera de façon toute naturelle les sous-balises `<param>` dont "`initParams`" qui contient l'IP destinée à l'application SL... Rusé, is'nt it ? !

Seconde étape : récupérer l'IP

Bon, la page ASPX a joué le rôle d'un Web Service pour récupérer l'IP du client, et l'inscrire "en dur" dans la page HTML retournée, comme si elle en faisait partie depuis toujours...

Reste à récupérer l'IP, c'est à dire le paramètre "`userIp`" destiné à l'application Silverlight et passé dans "`initParams`".

C'est dans `App.Xaml` que les choses se passent :

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    var userIp = e.InitParams["userIp"];
    this.RootVisual = new MainPage();
    ((MainPage) RootVisual).UserIp = userIp;
}
```

Code 48 - Lire un paramètre custom du plugin Silverlight

La variable `userIp` récupère le paramètre "`userIp`", c'est aussi direct et simple que cela.

Ensuite le `RootVisual` est instancié, et nous lui passons enfin la fameuse adresse IP pour qu'il en fasse ce qu'il veut. Pour ce faire nous avons juste ajouté une propriété publique à la page Silverlight, une propriété de type `string` qui se nomme `UserIp` (ce qui est très original vous l'admettrez !).

Après, ce que l'application Silverlight fera de cette IP est une autre affaire, la question était "comment récupérer l'IP du client dans une application Silverlight", pas "que faire de l'IP client"...

On peut bien entendu stocker la valeur dans une variable statique, un service de l'application ou de toute autre façon qui conviendra à l'architecture de votre application, le stockage dans une propriété de la page principale de l'application n'est qu'un simple exemple.

Conclusion

C'est une petite astuce, mais si l'application Silverlight a besoin uniquement de l'IP, il semble un peu lourd de créer un Web Service juste pour ça, et cette astuce permet de l'éviter. La même technique peut d'ailleurs être utilisée pour toutes les informations que le serveur peut recueillir sur l'appelant.

Comme on le constate elle se met en œuvre de base avec trois uniques lignes de code, l'une dans le fichier ASPX, les deux autres côté Silverlight (la propriété et la récupération du paramètre).

C'est facile, cela évite l'écriture d'un service, son paramétrage et sa maintenance. Bref, cela rend moins lourd la récupération de l'information. A condition d'héberger l'application SL dans une page ASP.NET.

On notera qu'à côté de la création d'un serveur Web et à côté de la génération de paramètres du plugin Silverlight il existe encore une troisième voie qui consiste à générer des variables HTML cachées. Depuis Silverlight il suffit d'interroger la page HTML hôte pour lire ces champs cachés. C'est une option intéressante (surtout si on veut transmettre beaucoup d'informations différentes).

Compression des fichiers Silverlight Xap

Moins un fichier Xap est gros, plus il est rapide à télécharger et moins cela prend de bande passante. Une sorte de lapalissade. Mais peut-on améliorer la compression des fichiers Xap ?

La réponse dépend de quelle version on parle…

Avec les versions 1 à 3 il est possible de gagner environ 10 à 30 % en "rezipant" les Xap (qui ne sont que des fichiers Zip avec une autre extension). On peut utiliser le compresseur gratuit 7-Zip et même concevoir un petit batch qu'on ajoute dans les événements Post Build du projet pour rendre l'opération automatique.

Mais depuis Silverlight 4 cette astuce ne sert plus à grand-chose. Microsoft a largement amélioré la compression et d'après mes tests avec 7-Zip, Izarc et WinRar, on retombe exactement sur les mêmes tailles que celles produites par la compression native de Silverlight.

Bien entendu on parle de format Zip, en compressant un Xap avec WinRar en mode Rar on obtient effectivement un gain… inutile puisque le plugin ne peut lire que des Xap compressés au format Zip :-)

En revanche d'autres techniques sont aujourd'hui bien plus efficaces pour réduire le temps de téléchargement d'une application Silverlight et ce sont elles qu'il faut utiliser puisque la "feinte" du "rezipping" n'apporte plus rien :

Le chargement dynamique des Xap (voir mon billet Silverlight 4 : Chargement Dynamique de DLL) qui est simplissime à mettre en œuvre,

MEF ((http://www.codeplex.com/MEF) sachant que MEF est intégré à .NET 4.0 et SL 4+ (mais le site contient pas mal de lien vers des exemples et reste intéressant),

Ou bien utiliser le cache des librairies, technique introduite dans SL3 (un PDF en français qui explique la technique : Le cache de librairies en Silverlight 3).

Conclusion

Perdre son temps à mettre en place des ruses pour "rezipper" les Xap est inutile depuis Silverlight 4, néanmoins il existe d'autres stratégies pour améliorer l'expérience utilisateur, notamment le temps d'activation du premier écran de l'application et l'économie de téléchargement des parties non utilisées du logiciel. Rangez 7-Zip et WinRar, laissez Silverlight se débrouiller pour la compression, mais travaillez plus l'architecture de vos applications !

Silverlight et le mode fullscreen

Le mode FullScreen de Silverlight est bien pratique mais possède quelques limitations qui peuvent agacer comme la perte du plein écran si l'utilisateur clique sur une autre application. Or Silverlight n'est pas si restrictif, encore faut-il le programmer correctement.

Silverlight et la Sécurité

J'ai mis un "S" majuscule à Sécurité car dans Silverlight, la Sécurité ce n'est pas rien ! Microsoft a vraiment eu une frousse terrible que SL puisse être utilisé pour créer un malware, ce qui aurait ruiné son lancement. De fait, et de façon un peu paranoïaque il faut l'avouer, de nombreuses choses sont interdites sous Silverlight. Je ne vous en dresserai pas la liste, elle est parfois déprimante, tant de puissance bridée, tant d'idées de softs géniaux tuées dans l'œuf lorsqu'on s'aperçoit que tel ou tel petit "détail" n'est pas réalisable (un simple Ping, télécharger une image ou une page d'un site Web qui n'a pas de fichier police, etc).

Silverlight en est à sa version 5 et effectivement aucun malware n'a été écrit. C'est une réussite totale pour les gars qui ont bouclé la sécurité du plugin, rien à dire, « *mais je reste convaincu que beaucoup de ces limites vont finir par peser si elles ne sont pas assouplies. Une aide à un bon lancement, mais un frein à l'expansion, le dosage devrait être revu au risque de tomber dans le piège de cette alternative…* ». C'est ainsi que je m'exprimais quand j'ai écrit ce billet, Silverlight 4 était à peine sorti et on ne savait pas encore que Microsoft l'arrêterait deux ans plus tard. L'un des arguments de Microsoft a consisté à affirmer que le plugin n'était pas « si utilisé que

çà » sur le Web grand public. Quand je me relis, je me dis que ma clairvoyance qui n'a rien de magique, juste du bon sens, a manqué cruellement à la direction de Microsoft depuis quelques années… Silverlight n'a pas été assez utilisé en Web grand public car Microsoft a interdit de nombreuses choses au nom de la sécurité, au nom de la non concurrence avec Windows Phone 7, au nom de plein d'âneries qui ont tué Silverlight bien plus que le refus de Jobs de voir Flash sur iOS…

Bref, parmi les choses évidentes qui pourtant ne passent pas, il y a le mode plein écran (full screen). C'est bête, mais c'est utile le plein écran.

Or, Silverlight propose bien un mode full screen. Mais hélas la saisie y est interdite (encore une raison de Sécurité, justifiée mais bon, à la fin ça agace un peu il faut l'avouer. Toujours cette impression de puissance bridée qui créée la frustration). Il est vrai que dans la version 5 la possibilité de faire des saisies a été ajoutée, mais c'était 4 versions trop tard et juste au moment où MS arrêtait SL !

Pire, lorsque l'application est en plein écran, malheur à l'utilisateur qui ose cliquer sur une autre application (vérifier un mail qui vient d'arriver par exemple) : Booom ! Le bel affichage plein écran se transforme en un petit cadre enchâssé dans le browser. Même l'utilisateur ça l'agace, et pourtant c'est "pour son bien" ! Même avec l'assouplissement de la V5 le mode fullscreen de Silverlight reste très limité.

Un compromis qu'on aimerait voir plus souvent…

Ok, il faut faire attention à la sécurité. Mais quand cela coupe les ailes d'un produit fantastique comme Silverlight on ne peut en rester là. Il faut faire des compromis. A ce jour Microsoft en a fait très peu et n'en fera donc plus, noyant un peu le poisson avec le mode Out Of Browser (OOB) qui lui permet plus de choses, mais qui, hélas, est vu comme une simple application desktop très bridée par les clients qui, quand ils veulent du Web, ne veulent pas entendre parler de soft à installer… Encore une erreur stratégique. L'OOB de SL n'a servi à rien et tout car ce qui est intéressant de faire dans ce mode a lui aussi été bridé par une sécurité délirante imposant par exemple l'installation d'un couteux certificat, ce qui ne peut se faire en plus qu'en bricolant la Registry à la main ! Si ce n'est pas de l'assassinat d'un produit alors il faut totalement changer la définition de plein de mots dans le dictionnaire… Personne ne semble l'avoir compris dans l'équipe (par ailleurs excellente) de Silverlight ou plus réaliste ils l'ont compris mais ils n'ont rien pu faire contre. C'est dommage. En fait nous le savons bien, la team Silverlight n'y était pour rien, les blocages venait de Sinofsky qui détestait C#, XAML et .NET. Il aura causé un tort considérable à Microsoft

(notamment avec Windows 8 qui s'est très peu vendu et dont on traine aujourd'hui le lourd fardeau qui oblige Ballmer à partir, c'est la moindre des choses, et qui force MS à manger son chapeau en parlant de futures versions avec bureau classique et vrai menu démarrer…. Combien d'années perdues, de business envolé ?).

Du coup, on se retrouve avec quelques issues de secours mais uniquement valables en OOB, un mode dont les clients ne veulent pas… On est bien avancé !

Heureusement, il y a des exceptions : Le fameux mode fullscreen.

Il est en effet possible d'empêcher que le carrosse ne se transforme en citrouille dès que l'utilisateur ira cliquer en dehors de l'application.

Comme il s'agit de "Sécurité", cela va être un ménage à trois : d'un côté Silverlight et son conseil supérieur de la sécurité qui nous offre du bout du clavier un paramètre à modifier, de l'autre le développeur qui devra utiliser ce paramètre, et enfin, le troisième oreiller dans ce lit déjà trop petit : l'utilisateur qui devra valider un dialogue ! Mais au bout du compte on y arrive.

Le code

Je parlerai plus d'astuce que de code, puisqu'il s'agit juste de basculer un paramètre :

```
1: Application.Current.Host.Content.FullScreenOptions =
2:     System.Windows.Interop.
                    FullScreenOptions.StaysFullScreenWhenUnfocused;
```

Il va falloir maintenant basculer en mode plein écran - attention "Sécurité" oblige, cela ne peut pas être fait automatiquement, uniquement dans le traitement d'un clic initié par l'utilisateur, certains avaient prévu que l'empreinte génétique de l'utilisateur soit aussi comparée à la liste des utilisateurs autorisés à s'autoriser ce genre de chose, heureusement la technologie ne le permettait pas encore ! Ouf ! On l'a échappé belle !

Enfin, l'utilisateur voit l'application en plein écran… Quelle lutte déjà. Et en plus cet idiot va cliquer sur sa boite mail pour voir le dernier message qui vient d'arriver…

Que n'a-t-il pas fait là ! Sans notre petite ruse, c'est cuit, fini, l'application Silverlight se rétracte dans son trou comme un escargot effrayé par une feuille de batavia à l'air patibulaire (mais presque)…

Mais heureusement, grâce à la feinte exposée plus haut, l'utilisateur qui voulait juste voir un mail sera arrêté dans son élan par un dialogue aussi moche qu'inquiétant :

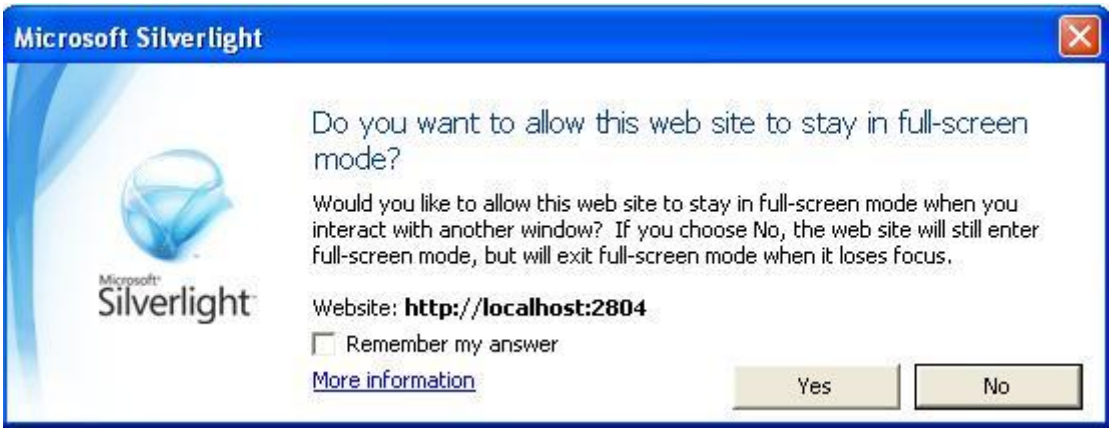


Figure 46- Le dialogue utilisateur du full-screen

Il aura le droit à la version française quand même (normalement) …

S'il clique sur oui (Yes) il pourra enfin lire son mail sans que Silverlight ne rétrécisse.

Arrrghhh, gaspp ! Malheureux ! Il a oublié de cocher la case "se rappeler de ma réponse" (*remember my answer*). Le prochain coup qu'il voudra cliquer ailleurs il se reprendra à la face le même dialogue.

Les plus courageux qui auront lu le message et qui auront coché la case en question seront enfin tranquilles ! Ils savoureront une application Silverlight plein écran qui ne se transforme pas citrouille si on clique ailleurs (mais qui ne peut pas servir à saisir la moindre chose, Sé-Cu-Ri-Té ! enfin si avec SL5 mais du bout du clavier, faut pas taper n'importe où sinon ça pète !).

Ah !! Tranquilles ? Qu'ils croient ! Car … "Sécurité" oblige, ce choix n'est valable que pour une seule application. « *Si demain Silverlight était un franc et vrai succès, c'est donc des dizaines de fois que l'utilisateur devrait répondre à la même question, pour chaque application (au lieu d'un réglage global par défaut, éventuellement affiné site par site ensuite)…* » Je laisse encore un passage original car il parle de lui-même. SL ne deviendrai jamais un franc et vrai succès, d'abord parce qu'il a été stoppé en plein élan, et bien entendu parce que MS n'a jamais ouvert la cage permettant à ce bel oiseau de s'envoler… Un acharnement qui aura payé, aucun malware n'a été écrit en SL. Mais plus grand-chose d'autre ne le sera non plus…

A l'inverse du motto de la Nasa, je crois que le slogan secret fut "*Success is not an option*" et tout a été fait pour respecter ce pacte, belle efficacité. Dommage.

Mais certains se coucheront l'âme sereine chez Microsoft : aucun malware n'aura été écrit sous Silverlight. *Quitte à ce que le produit n'atteigne jamais les sommets qu'il mérite largement pourtant…* Il ne les atteindra plus, aujourd'hui nous le savons.

Imprimer avec Silverlight

Silverlight 4 a introduit un mécanisme permettant d'invoquer l'impression côté client même en mode Web simple (par opposition au mode Out-of-Browser). Comment en tirer parti ?

Print Screen

L'utilisation la plus basique qu'on peut envisager est celle d'un *print screen* (copie d'écran), ce n'est pas fantastique mais c'est déjà pas mal puisque cela n'existait pas avant SL 4 !

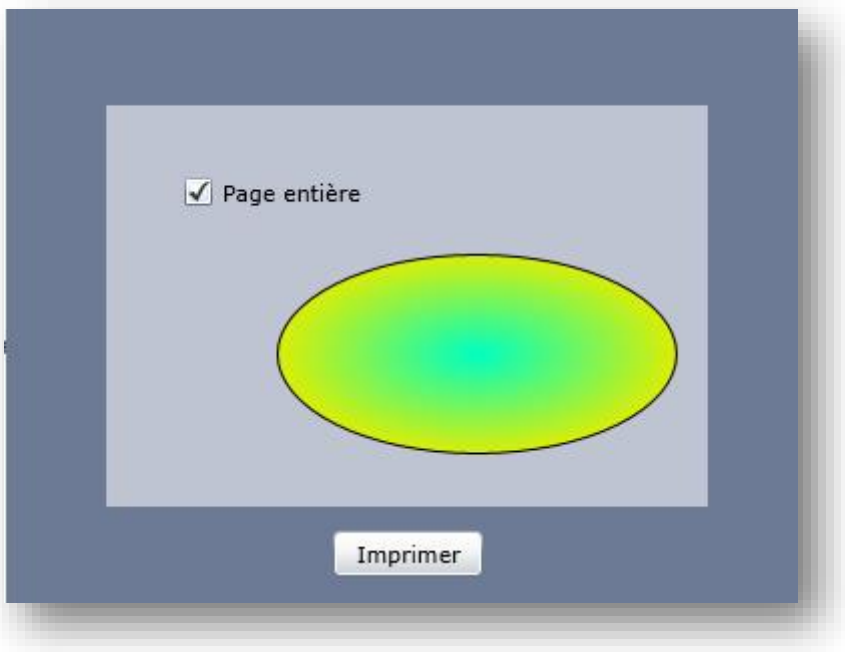


Figure 47 - Page exemple pour l'impression

Regardez l'exemple ci-dessus. Si vous cliquez sur le bouton "Imprimer" vous verrez apparaitre le dialogue d'impression classique de Windows (ou du Mac) vous permettant de sélectionner une imprimante. Pour les tests, utilisez une imprimante PDF ou XPS, cela évite de consommer bêtement du papier… (L'exemple live

fonctionne sur Dot.Blog, cliquez sur le titre de ce chapitre pour accéder au billet originel).

Vous avez forcément remarqué la case à cocher “Page entière” et vous avez peut-être même déjà joué avec. Sinon, testez de nouveau une impression mais en décochant la case.

Dans le premier cas toute la page Silverlight est imprimée, dans l’autre seul le cadre central l’est (la grande différence est que le bouton *Imprimer* n’apparait plus).

Comment ça marche ?

Prenez une application Silverlight, placez-y un bouton pour imprimer et voici à quoi doit ressembler le code pour effectuer un Print Screen comme dans l’exemple ci-dessus :

```
 1: private void btnPrint_Click(object sender, RoutedEventArgs e)
 2:         {
 3:             var doc = new PrintDocument();
 4:             var wholePage = cbWholePage.IsChecked;
 5:
 6:             doc.PrintPage += (s, ee) => ee.PageVisual =
 7:                  (wholePage ?? false)
                     ? (UIElement) LayoutRoot : centralDoc;
 8:
 9:             doc.Print("Silverlight 4 Screen to Paper demo");
10:         }
```

Code 49 - Imprimer avec Silverlight

Comme on le remarque c’est court… Mais ça ne fait pas grand-chose non plus !

Tout tient à la classe `PrintDocument` dont on créé une instance et à qui on explique ce qu’il faut imprimer en programmant son évènement `PrintPage`.

Dans l’exemple ci-dessus cette programmation s’effectue vie une expression Lambda qui s’occupe de passer soit l’objet `LayoutRoot` (page entière), soit le canvas central (page partielle). A qui est-ce passé ? A la propriété `PageVisual` qui provient des arguments de l’évènement.

Enfin, on appelle la méthode `Print` à laquelle on peut passer un texte qui sera visible dans le gestionnaire d'impression (certains pilotes d'imprimante récupèrent aussi ce texte pour créer le nom de fichier : pilote PDF par exemple).

C'est tout ?

Out-of-the-box comme on dit, oui, c'est après tout… Bien entendu on peut ruser un peu sur les marges, on peut imprimer quelque chose de plus long que la simple page en cours. Mais le principe est là : on créé un document d'impression, on se branche sur son évènement d'impression et là on fournit un *arbre visuel* à imprimer. L'appel à `Print` déclenche le processus.

C'est rustique, tout en bitmap (donc *ramivore*, et *cpuvore*), je dirais même que c'est moyenâgeux, n'ayons pas peur des mots.

Disons que c'est comme pour le son ou les bitmaps, MS nous livre ici les prémices d'un début de ce qui pourrait devenir une voie intéressante avec 1 an de travail en plus …

Mais ne boudons pas l'arrivée des impressions, même simplistes, dans Silverlight. Cela permet de s'affranchir de solutions plus lourdes (serveurs d'impressions) et des round-trips entre le client et le serveur. Là tout est fabriqué sur place, tout chaud prêt à consommer, sur PC et sur Mac.

Des Extensions ?

Il est vrai qu'on est très loin des besoins d'une application de gestion, pourtant domaine qui semble être le créneau porteur de Silverlight (Flash est mieux fichu pour le son et les graphiques et fait parfaitement des pubs ou des sites sapin de Noël). Petite incohérence dans le ciblage ou tout simplement problème de timing pour l'équipe de SL ? Je préfère envisager la seconde solution que la première qui serait plus grave de conséquences.

J'espère donc, comme pour le `WriteableBitmap`, que l'équipe de SL saura nous fournir prochainement des couches un peu plus haut niveau, donc utilisables.

Un générateur d'état minimaliste

En attendant, un projet s'est ouvert sur CodePlex : http://silverlightreporting.codeplex.com/

Ce mini projet est vraiment un balbutiement de générateur d'état, mais une fois encore ne boudons pas cette offrande (c'est gratuit et en code source) car elle peut fort bien servir de point de départ à quelque chose de plus ambitieux (ne pas confondre avec un projet de nom proche qui n'a jamais décollé).

Ici on peut définir entête et pied de page, il y a une gestion de la pagination et les "bandes" peuvent avoir une talle variable. Voici un exemple de sortie :

Employee Review List Page 1

Name and Address	Performance Rating	New Salary	Bonus
Tolley, Jesse 335 Silver Way Able Church, VT 28090 Please begin naming animal sounds for the codebase. Relaxed attitude Not performing to expectations.	8	$74,033.00	$7,169.00
Foor, Amy 107 Water Way Fallston, VT 17290 You had better continue coding noises in the kitchen. Loyal. Lorem ipsum dolar set.	5	$65,502.00	$2,945.00
Papa, Alexis 108 Foddergard Ave Hodge Church, DC 45849 Please stop churning pens for your codebase. I've forgetten why I keep you employed. You're no Tim Heuer!	4	$64,414.00	$3,612.00
Foor, Bill 604 Hill Ave Arington, CT 30300 Management all agrees that you eliminate eliminating animal sounds in the cubicle. You are weak old man. Loyal.	5	$61,636.00	$4,492.00
Angel, Shawn 49 Studio St Gambriway, VT 63576 Management all agrees that you stop coding noises in the cube. Do or do not. There is no try. Does good work.	6	$51,487.00	$5,225.00
Brown, Marcus 615 Mayapple St Croftown, DE 35944 Your coworkers request that you eliminate giving noises on your bathroom. You're no Tim Heuer! Loyal.	6	$59,485.00	$4,694.00
Tolley, Kevin 691 Mayapple Ave Crofway, MA 66712 Management all agrees that you continue randomizing animal sounds on your codebase. Not performing to expectations. I've forgetten why I keep you employed.	5	$63,497.00	$4,284.00
Sune, Pete 346 Woodland Ave Ableway, PA 18587 You had better stop eliminating pens for your cube. You're no Tim Heuer! I've forgetten why I keep you employed.	8	$46,471.00	$6,638.00
Badly, Shawn 304 Silver St Fallston, PA 34310 You had better eliminate naming animal sounds in the cube. Should have stuck to C++. Too big to fail.	8	$56,954.00	$6,061.00
Badly, Sean 1238 Mayapple Circle Hilltown, NH 28286 Please begin making noises in the cubicle. Would like to have on all my teams. Should have stuck to C++.	7	$72,900.00	$3,533.00
Foor, Sean 495 Water St Hillton, DE 45166 You had better stop coding pens for the code. Please shower daily. Lorem ipsum dolar set.	7	$69,232.00	$5,711.00
Crimson, Bill 557 Elm Ave Falls Church, MA 46097 Management all agrees that you eliminate randomizing reviews in the cube. Should have stuck to C++. You're no Tim Heuer!	5	$54,803.00	$2,777.00
Badly, Fred 15 Sandy View St Fallston, MA 49326 Your coworkers request that you stop making feedback in the kitchen. Your weaknesses are just overuse of your strengths. You need to work on 'modulation.' Too big to fail.	8	$55,730.00	$6,895.00

Page 1 of 3

Figure 48 - Exemple d'impression

Conclusion

Niveau reporting, local j'entends, il y a tout à inventer ! Si vous avez une bonne idée vous deviendrez peut-être riche car c'est vraiment un module qui manque pour tout ce qui est application de gestion.

Le petit projet indiqué plus haut ne peut pas réellement servir de base sérieuse car ce qui est imprimé n'est qu'un arbre visuel qui occupe déjà de la place en mémoire et qui, de plus, est transformé en bitmap. Autant dire que pour 2 à 5 pages ça marche, mais qu'il ne semble guère possible de pousser le concept pour créer un véritable générateur d'état.

Il existe des serveurs d'impressions vendus comme des composants Silverlight. Ne vous laissez pas avoir, il s'agit pour l'essentiel de vieilles solutions pour Java ou ASP.NET, uniquement en mode serveur, réadaptées avec plus ou moins de bonheur à Silverlight. Toutes les solutions que j'ai pu tester obligeaient un téléchargement plus lourd qu'une bonne application à elle seule. Inutilisable ou presque.

Bref, pas de pessimisme, mais du réalisme : Silverlight nous laisse sur notre faim et l'unique petit projet un peu évolué ne semble pas pouvoir aller bien plus loin. Il faudra certainement encore un peu de temps pour que nous voyions apparaitre de vraies solutions d'impression locales sous Silverlight.

Du Son pour Silverlight !

Paradoxalement le son et l'image sont les parents pauvres de Silverlight. Certes il y a le `MediaElement` ou la classe `WriteableBitmap`, mais on est encore loin du compte…

Son et Image sont les mamelles d'une application Web réussie

Silverlight est un outil fantastique mais quand on prend un peu de recul et qu'on y regarde de plus près tout ce qui concerne le son et l'image est d'un niveau assez faible. Comment contourner ces problèmes et pourquoi tant insister sur ce qui n'est finalement pour beaucoup que "gadgetisation" des applications ?

C'est que les mêmes disaient exactement la même chose lors de l'apparition des écrans couleurs, le mode CGA des premiers IBM PC dans les années 80, alors que personne aujourd'hui n'accepterait de travailler sur un écran vert… Ne pas savoir faire la différence entre simple mode et évolution inéluctable est gênant dans notre métier résolument tourné vers l'avenir…

L'ajout de `WriteableImage` dans Silverlight pour manipuler des bitmaps est presque un joke, il faut tout programmer, c'est vraiment du brut de chez brut. Inutilisable sans y consacrer des heures de développement supplémentaires. Mais l'image n'est pas mon propos aujourd'hui. C'est le son.

Car son et image sont les mamelles d'une application réussie… Regardez les belles applications faites en Flash et vous verrez de nombreux effets visuels très riches, très réussis. Bien entendu le Design général est essentiel, bien entendu les fonctionnalités sont la base même d'une bonne (ou mauvaise) application. Mais un bon Design servant de bonnes fonctionnalités n'est pas suffisant dans notre monde d'image et de mouvement. Il faut aussi que "ça jette", que l'œil soit attiré, que l'oreille se tende, qu'il y ait une **sensation d'immersion** dans l'univers créé par l'application. ***Bref, que l'utilisateur du 21ème siècle qui est un zappeur fou impatient ait envie de voir la suite (et d'acheter le produit) !***

Et cela réclame des effets visuels et du son à la hauteur (ce qui veut dire **bien réalisés et bien dosés**, je ne parle pas de passer à fond une musique à la noix dès que le site s'ouvre…). Supprimez les effets sonores et visuels d'Avatar ou Elysium, il ne reste que deux lignes de scénarios… Retirez le son surround et la 3D temps réel de la majorité des jeux modernes et il ne restera même pas la moitié de l'intérêt d'un Pac-Man ou d'un Asteroid ! **Il existe aussi de magnifiques réussites où tous les avantages techniques sont utilisés à bon escient**. C'est notre monde, tel qu'il est. Il y a les rêveurs qui voient le monde tel qu'ils souhaiteraient qu'il soit (et qui sont donc à côté de la plaque en permanence) et les réalistes qui, quel que soit leur opinion sur ce monde dirigent leurs actions en tenant compte de sa réalité. Faites partie de ces derniers, si ce n'est dans votre vie privée dans laquelle je ne veux pas m'immiscer, mais au moins quand vous développez des applications.

Je vous conseille ainsi de faire un tour sur le site de *Template Monster* : http://www.templatemonster.com/ ils vendent des templates de sites pour Flash et aussi pour Silverlight. Même si le style est souvent un peu trop clinquant, arbre de Noël, et un peu emprunt du mauvais goût typiquement américain (notamment musique de fond à la noix dont je parlais plus haut), tout est réalisé avec un niveau de professionnalisme qu'on aimerait (moi en tout cas!) voir plus souvent…

Même les templates les plus ringards ou les plus mièvres sont ficelés aux petits oignons. Introductions fluides et accrocheuses, sonorisation des clics, effets bien synchros et bien choisis, il y a mille idées à reprendre de ces templates et une bonne leçon de professionnalisme à tirer d'une visite approfondie de ce qu'il propose. Le

paradoxe américain… Un goût pas toujours très fin, mais une capacité hors du commun à bien réaliser les choses.

Sur *Template Monster* regardez certaines transitions visuelles des modèles Flash et essayez de les refaire sous Silverlight… Vous comprendrez alors qu'il reste du pain sur la planche pour l'équipe de Silverlight (les effets bitmaps sont encore trop simplistes dans ce dernier).

Silverlight n'en est pas moins un magnifique produit se reposant sur le Framework .NET, offrant des outils professionnels comme les WCF RIA Services par exemple. Heureusement car *c'est ce qui fait de Silverlight une solution crédible pour de nombreuses applications orientées business.*

Le paradoxe de Silverlight c'est d'avoir été conçu pour le grand public en ayant oublié de le doter d'un traitement du son et des bitmaps indispensable dans un tel cadre. Il a eu plus de succès en entreprise mais là, pour faire du LOB il lui manque des choses essentielles comme un moteur d'impression à la hauteur. Le produit aurait pu être un succès énorme sans ces grossières erreurs d'approche.

Mais venons-en au son.

Un "moteur" de mobylette

Concernant l'image, l'arrivée de `WriteableBitmap` est une bonne chose mais c'est trop peu. Mais en ce qui concerne le son, le moteur fourni (entièrement programmable, sans aucune extension) est plutôt celui d'une mobylette que d'une moto de compétition…

Pas de Boucle

Le premier point négatif est l'absence de mode boucle (*loop*). Impossible de faire jouer un même son en boucle : musique de fond par exemple. Pour un site c'est souvent inutile (voire nuisible) mais pour un jeu, un utilitaire un peu looké ou un site publicitaire cela peut être essentiel.

Grande latence

Un paramètre permet de jouer sur la latence mais ce n'est pas suffisant pour assurer une bonne réactivité. De plus le décodage des mp3 ou wma laisse un "blanc" qui n'est pas éliminé par le décodeur. Si on y ajoute la lenteur du moteur actuel, boucler un son court est impossible et laisse largement entendre un silence. Impossible donc de simplement boucler quelques millisecondes de bruit blanc pour simuler le bruit du

vent par exemple. Ni même de fabriquer des ambiances sonores évolutives à partir d'échantillons.

Pas d'effets

Aucun effet disponible. Pas de réverbération, ni d'écho, pas de contrôle du pitch (hauteur du son ~ vitesse de lecture), encore moins de time-stretching ni aucune fonction un peu évoluée.

Si tout doit être préparé à l'avance en studio et qu'on doit utiliser 100 échantillons différents là où un seul suffirait, c'est toute l'application qui sera ralentie lors de son téléchargement (sans parler de la complexité et du coût d'un tel travail qui impose que tout soit figé sauf à retourner en studio pour refaire les effets).

Peu de décodeurs

A l'heure actuelle `MediaElement` ne sait lire que quelques formats. Le moteur sous-jacent permet dans l'absolu de traiter tous les formats, encore faut-il écrire beaucoup de code. Il manque des briques élémentaires plus sophistiquées et directement utilisables.

Les solutions actuelles

Pour l'instant il n'existe rien de convaincant, ni en provenance de Microsoft ni même de la communauté pourtant riche de millions de développeurs C#. Mais il existe tout de même quelques tentatives qui méritent d'être relevées et qui peuvent, dans certains cas, vous aidez à pallier certains manques de l'implémentation actuelle de Silverlight. Le but ici n'étant pas de se contenter d'un constat sévère mais bien d'avancer positivement des solutions pour dépasser les limitations actuelles.

Contrer le gap audio pour faire de boucles

C'est le premier point noir. Crée des boucles est essentiel. D'abord parce que cela permet d'utiliser des échantillons de petite taille et d'accélérer le téléchargement de l'application, ensuite parce que cela ouvre la voie vers de nombreuses utilisations pratiques (fond ou ambiance sonore, effets audio s'adaptant en longueur aux événements visuels, etc).

Hélas les formats supportés par `MediaElement` possèdent leurs propres problèmes. Par exemple les Mp3 standard possèdent tous un blanc en tête de fichier. Si on ajoute à cela que `MediaElement` doit remplir un buffer interne avant de jouer un son, les difficultés s'accumulent.

Le meilleur moyen de supprimer les gaps sonores est soit de gérer son propre décodeur et d'éliminer le gap, soit d'utiliser un format "gapless", sans blanc, ou bien de trouver une astuce pour sauter le gap en début de fichier (et parfois aussi celui se trouvant à la fin).

Il est possible d'encoder les fichiers avec un outil gapless comme WinLame. Il sait traiter les Mp3 et Ogg Vorbis (et quelques autres formats). Il est toujours en bêta et ne semble plus évoluer depuis 2010 mais il possède déjà un bon niveau de maturité. Sur des sons courts la qualité reste malgré tout médiocre.

Silverlight propose malgré tout et en dernier ressort un accès à la machinerie interne. Il est ainsi possible d'écrire son propre décodeur. Larry Olson a écrit il y a un moment les Managed Media Helpers qui ont inspiré beaucoup d'autres auteurs. Techniquement l'astuce consiste à écrire une sorte de codec qui fournit le flux au `MediaElement`. Ce flux peut provenir du décodage d'un fichier existant (Mp3 par exemple) ou bien être créé *ex-nihilo* et cela devient un synthétiseur… Le code d'Olson ne fait que traiter les frames Mp3 envoyant le flux efficace vers le décodeur du `MediaElement`, ce qui ne règle pas le problème des boucles puisqu'il reste impossible de pointer n'importe où dans le flux. Mais vu le peu de choses que Microsoft a fourni sur le moteur interne, ce projet a valeur de documentation pratique qui peut s'avérer indispensable pour comprendre comment créer votre propre décodeur.

Un autre développeur, DDtMM, a écrit une librairie Seamless Mp4 qui traite donc le format Mp4 (uniquement la partie sonore du format) et qui propose le mixage de plusieurs sons et boucles avec un résultat correct. Toutefois cela implique un encodage Mp4 peu pratique au lieu d'exploiter des formats plus classiques comme Mp3 ou Wav. La librairie n'est pas parfaite mais pour avoir tester ses différentes évolutions, cela est malgré tout exploitable. Le projet est fourni avec les sources, et là aussi cela peut constituer une bonne documentation sur la façon d'écrire des décodeurs audio sous Silverlight.

On trouve aussi Saluse Media Kit, un décodeur qui traite les Mp3. C'est une option intéressante même si les sons courts semblent poser là aussi des problèmes. Mais, one more time, cela fait un exemple de plus traitant un format classique qui peut servir de base à l'écriture de votre propre décodeur. A noter que le Saluse Media kit va un peu plus loin que les autres en offrant la possibilité d'ajouter des effets (écho, panoramique, changement de pitch). Le résultat n'est pas forcément à tomber par terre (niveau qualité sonore) mais c'est un bon début puisque la notion d'effet et de traitement de flux audio numérique y sont abordées.

Atsushi Ena est à la base d'un autre projet, MoonVorbis qui possède de nombreux atouts mais reste focalisé sur le format Ogg Vorbis et qui s'adresse, comme le nom le laisse deviner, à MoonLight, la version Mono de Silverlight. Le code est malgré tout une bonne base pour qui voudrait l'adapter (l'auteur a testé la compatibilité avec Silverlight 3, il faut donc voir comment cela se comporte aujourd'hui). C'est une des rares librairies que je n'ai pas encore eu le temps de tester, donc si vous le faites n'hésitez pas à venir en parler !

Contrer la latence

La latence est l'ennemi du son numérique, en tout cas de sa réactivité. Les "claviers" le savent bien, pendant longtemps le problème des "soft synths", des synthétiseurs numériques simulés sur ordinateur, a été la latence introduite par les cartes son et par le manque de célérité des anciens processeurs. Or, une latence aussi basse que 50 ou 100 millisecondes, si elle peut paraître très faible, voire inaudible pour un profane, est, je peux vous l'assurer, absolument insupportable pour le musicien pendant qu'il joue ! Par défaut la latence des flux du `MediaElement` est de … 1000 ms !

La latence du `MediaElement` est énorme, mais les décodeurs que l'on peut écrire en se basant sur `MediaStreamSource` offrent plus de souplesse. On dispose notamment de la propriété `AudioBufferLength` réglée donc par défaut à 1000 ms (1 seconde). Selon votre hardware, il est possible de réduire cette valeur jusqu'à 50 ms sans trop de problème, même parfois en dessous. Mais pour faire un soft synth c'est encore beaucoup. Descendre à 15 ms est presque parfait mais cela engendre sur la plupart des machines des "glitches" et une distorsion du son. On arrive là aux limites du matériel. Pour les musiciens qui utilisent des soft synths il y a un passage obligé : une carte son professionnelle, généralement externe et utilisant Firewire (IEEE 1394) plutôt que de l'USB. Avec du matériel de ce type on peut descendre à 3 à 4 ms ce qui rend le jeu presque aussi naturel que sur un instrument hardware. Mais on s'éloigne des possibilités s'offrant à Silverlight, d'une part parce qu'en mode Web il ne peut accéder aux pilotes hardware et ne pourrait donc pas utiliser les pilotes ASIO éventuellement présents sur la machine hôte, et que parce que justement, en tant qu'application Web on ne sait pas à l'avance quelles seront les performances de la machine de l'utilisateur et qu'il vaut mieux s'en tenir à des valeurs moyennes plutôt que de tabler sur les performances d'un monstre multicœur overclocké…

En se contentant de 50 ms on ne peut pas espérer écrire de véritables soft synths en Silverlight, mais cela n'est pas grave puisqu'un soft synth s'utilise au sein d'un autre logiciel plus complexe assurant le mixage, les effets spéciaux, et que ce cadre

d'utilisation n'est absolument pas celui de Silverlight. En revanche on peut développer des tas de petites applications sympathiques en Silverlight "orientées son" : simulateur de batterie numérique, mini séquenceur, etc… **Il y a donc malgré tout un univers à explorer, techniquement accessible à Silverlight**, pour qui met les mains dans le cambouis et s'attaque à la réalisation d'un décodeur spécialisé. Pour rappel, un décodeur Silverlight peut fort bien décoder un flux existant (Mp3, Mp4, Ogg, etc) mais avec un peu d'astuce il peut aussi créer des sons et devenir ainsi un soft synth…

D'autres références

Pour ceux qui sont intéressés par le sujet, il faut bien entendu se rendre sur MSDN pour lire la documentation de MediaStreamSource à la base de tous les espoirs d'un son meilleur pour Silverlight.

On trouvera aussi une introduction par Tim Heuer dans un billet de 2008, ainsi qu'une série d'articles sur MSDN en trois parties.

Peter Brown a tenté d'écrire un soft synth en Silverlight. Le produit final n'est pas vraiment utilisable en raison de petites imperfections mais en revanche c'est un exemple de code intéressant pour qui veut se lancer dans ce type de programmation.

Moins mélodieux mais plus technique, le célèbre Petzold qui fut l'idole de toute une génération de programmeurs et qui s'est converti de longue date à C# a produit quelques exemples dont un petit softsynth avec utilisation du clavier pour jouer des notes.

Il aussi KSynth et KSynth-b de Mike Hodnick dont la page semble hélas ne plus exister sur le site. Même le lien vers la version online ne fonctionne plus. En revanche on peut encore accéder au trunk de son système de gestion de version ici. En cherchant bien on trouve le code source de Ksynth sur Google code.

Dans un genre encore plus abouti il y a Pluto Synth mais je n'ai pas réussi à trouver le code source… Mais même sur mon PC plutôt gonflé ça devient peu réactif, ça consomme à fond mais il s'agit d'un véritable séquenceur Midi 16 pistes avec soft synth intégré et divers sons suivant la norme GS. De loin la réalisation la plus aboutie et une belle performance de développement.

Enfin, il existe SilverSynth sur CodePlex qui réintègre KSynth, certainement pas un soft synth utilisable tel quel mais pouvant fournir le code de base pour réaliser quelque

chose de plus fonctionnel. C'est en tout cas peut-être la librairie dont il faut partir pour créer la sienne. Elle contient des bouts de code de Petzold et d'autres auteurs et il existe plusieurs exemples. La version KSynth-b contient une classe gérant les échantillons Wav mais c'est vraiment le début d'un commencement… Cette librairie répond ainsi plus aux besoins d'un soft synth que du traitement des Waves (ou autres signaux sonores), décidément enfants pauvres jusqu'au bout…

Peu ou rien sur le traitement des signaux eux-mêmes donc (analyse graphiques, effets…).

N'oubliez pas Google Code, l'outil de recherche de code de Google qui est très utile. On peut grâce à lui trouver d'autres exemples en C# (ou d'autres langages) comme les transformées de Fourier (base de l'analyse graphique multibande, de la mise en place de filtres et même de détection de beat, etc).

Conclusion

Silverlight contient les germes de nombreuses améliorations indispensables concernant le traitement de l'image autant que celui du son. Hélas, pour l'instant il ne s'agit que de germes et non de features utilisables out of the box.

Concernant le son on voit qu'il existe des tentatives plus ou moins abouties de faire pousser la petite graine et obtenir quelque chose d'utilisable.

Le chemin reste malgré tout encore assez long pour une solution à la hauteur des besoins, c'est à dire pour quelque chose qui soit au minimum aussi utilisable que Flash…

Dans les dernières versions de Silverlight Microsoft a ajouté la possibilité d'accéder aux librairies XNA pour la 3D et le son. On y trouve beaucoup de choses intéressantes mais d'une part XNA est un projet arrêté et d'autre part le véritable intérêt est de pouvoir faire du son en Silverlight sans entrer dans les méandres de librairies tierces fermées. Comme toujours sur Dot.Blog chaque sujet est d'abord l'occasion d'apprendre à faire soi-même les choses, pas à utiliser des solutions opaques toutes faites.

Personnaliser un thème du toolkit Silverlight

Les thèmes fournis avec le toolkit Silverlight (ou en version WPF) sont bien pratiques pour obtenir rapidement un visuel propre sans investir des heures pour créer un

ensemble de styles. Toutefois on ressent souvent le besoin de les personnaliser, de changer un petit détail. Hélas, autant l'utilisation des DLL de thèmes sont pratiques autant elles sont hermétiques à toute modification... Regardons comment personnaliser un thème du toolkit et surmontons ce mur qui n'est finalement pas infranchissable.

Les thèmes

Il existe deux moyens d'appliquer un style dans Silverlight, soit en utilisant les styles implicites soit en utilisant les styles explicites. Lapalisse n'aurait pas pu trouver plus évident !

La différence essentielle entre ces deux formes de styles tient dans leur effet : un style implicite s'applique automatiquement à toute une classe de contrôle sans avoir besoin de le préciser au niveau de chacun d'eux. Un style explicite nécessite une déclaration qui l'est tout autant dans le contrôle qui doit recevoir le dit style.

Cette nuance se traduit non pas par une syntaxe différente mais plutôt par une sorte de ruse : les style explicites doivent avoir un nom (`x:Key`) puisqu'ils appartiennent à un dictionnaire de styles (traduit côté code par un `dictionnaire<>` qui n'accepte donc pas les clés nulles, ni les clés redondantes).

La ruse tient dans le fait que les style implicites ne doivent pas déclarer de nom (`x:Key`), en apparente violation avec les règles de gestion des dictionnaires de styles. Mais laissons cette "plomberie" et ses mystères aux concepteurs de Silverlight ce n'est pas l'objet de ce billet. Ce qui nous intéresse ici est de savoir faire la différence entre ces deux formes de styles et de savoir les déclarer.

Un thème est donc tout simplement un ensemble de styles implicites contenu généralement dans un seul fichier XAML. Un thème permet d'obtenir un visuel cohérent tout au long d'une application.

Le toolkit de Silverlight fourni plusieurs thèmes qui ne sont pas forcément merveilleux, sauf le thème Expression Dark car il reprend le look de Expression Blend logiciel dont vous savez mon amour inconditionnel... Merveilleux ou non, ces thèmes offrent un visuel cohérent rapidement et permettent d'échapper au look XP de base qui est assez atroce tout en évitant l'investissement énorme que peut représenter la création d'un thème ex-nihilo.

Motivation

Les thèmes du toolkit sont fournis en version DLL et en version source XAML. Si on désire d'emblée créer un thème personnalisé on peut bien entendu utiliser le source XAML. Mais souvent il arrive qu'on soit parti sur la version DLL, plus rapide à mettre en œuvre mais qu'on veuille, à un moment donné du développement, personnaliser un élément...

Un exemple qui m'est arrivé et qui peut motiver une telle personnalisation : Dans un logiciel pour lequel moi et mon infographiste avions conçu un look simple et épuré j'avais utilisé le thème Expression Dark qui se mariait bien avec le design qui avait été créé. L'utilisation d'un thème permettait ici (et c'est une méthode réutilisable par tous) d'avoir à la fois un look très original puisque créé pour le client sans avoir à définir un style pour chaque contrôle (boutons, sliders, listbox...). Justement, parlons des listbox. C'est au moment de créer le DataTemplate pour l'une d'entre elles que le problème s'est posé. La largeur du DataTemplate ne s'adapte pas à la largeur de la listbox mais à celle du contenu. C'est un piège classique que j'aurai du anticiper (mea culpa est) d'autant plus que j'avais écrit un billet sur le sujet l'année dernière (Largeur de DataTemplate et Listbox). Faut-il être distrait (et écrire tellement de choses !) pour tomber dans un tel panneau... Heureusement grâce à Dot.Blog (qui me sert souvent d'aide-mémoire !) j'ai retrouvé facilement la solution. Elle consiste à ajouter quelques lignes de XAML à la Listbox pour fixer un style qui modifie l'alignement horizontal de l'ItemContainer.

Or, cette astuce, en fixant un style local à la `Listbox` interdit l'application du style implicite du thème du toolkit ! C'est tout le thème de la listbox qui tombe ainsi à l'eau. Deux solutions, soit réinventer le thème Expression Black de la `Listbox`, soit customiser le thème du toolkit.

On en revient au sujet du présent billet ...

Personnaliser un thème du Toolkit

Pour passer d'un thème du toolkit à une version personnalisée de ce dernier encore faut-il avant toute autre chose passer du thème automatique en DLL au même thème en version Xaml modifiable sans faire tomber toute l'application.

Première étape : de la DLL au source Xaml du thème

Quelques étapes vont être nécessaires pour passer de l'un à l'autre. Selon le nombre de pages utilisant la DLL la manip sera plus ou moins longue mais en général l'ensemble de la modification se fait assez vite, surtout en suivant le guide :

Phase 1 : Localiser le source Xaml du thème

Il faut dans un premier temps localiser le code source Xaml du thème que vous utilisez dans l'application. Je vais prendre l'exemple du thème Expression Black dans le reste du billet. Ce code source se trouve sur ma machine dans :

```
C:\Program Files (x86)\Microsoft
SDKs\Silverlight\v4.0\Toolkit\Apr10\Themes\Xaml
```

(Exemple sous Windows 7 / 64 bits, le chemin peut être légèrement différent si vous utiliser un autre OS).

Phase 2 : copie de la source Xaml

Une fois le fichier repérer, copiez-le dans le répertoire `Assets` ou `Skins` de votre application Silverlight (le répertoire que vous avez réservé pour les dictionnaires de styles).

Je vous conseille de renommer le fichier pour éviter toute confusion avec l'original, d'autant que nous faisons cette manip en vue de le modifier.

Phase 3 : suppression de la DLL

Comme le projet utilisait en référence la DLL du thème Expression Dark, il va être nécessaire de supprimer cette dernière. A la place il faut ajouter une référence à la DLL "`Theming`" du Toolkit (`System.Windows.Controls.Theming`).

Phase 4 : correction de l'espace de noms

Dans chaque Vue de votre application utilisant la DLL il va être nécessaire de modifier la déclaration des namespaces pour ajouter "`Theming`" :

```
xmlns:theming="clr-amespace:System.Windows.Controls.Theming;
assembly=System.Windows.Controls.Theming.Toolkit"
```

Phase 4 : Correction des balises

La DLL originale n'est plus référencée, le code source Xaml du thème a été rapatrié, le namespace du `Theming` du Toolkit a été ajouté à chaque Vue. Reste à supprimer la balise de l'ancien thème et à la remplacer par "`theming:Theme`".

L'ancien code ressemblait à cela :

```
<toolkit:ExpressionDarkTheme >
        <Grid x:Name="LayoutRoot">
            ... ... ...
        </Grid>
</toolkit:ExpressionDarkTheme>
Le nouveau sera le suivant :
<theming:Theme>
        <Grid x:Name="LayoutRoot">
           ... ... ...
        </Grid>
</theming:Theme>
```

Phase 5 : Incorporer le nouveau dictionnaire

Ajouter le fichier Xaml du thème dans le projet n'est pas suffisant… Encore faut-il le référencer dans la liste des dictionnaires de ressources gérées dans App.Xaml :

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Skins/MainSkin.xaml"/>
        <ResourceDictionary Source="Skins/MyExpressionDark.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Phase 6 : La peur !

Lancez la construction de votre projet et faites-vous peur !

Tout est cassé et vous vous prenez une volée de bois vert de la part du compilateur. Des erreurs par kilos, voire par tonnes.

La cause première de la plupart des erreurs que vous verrez est simple : le fichier Xaml de ressource du thème qui a été ajouté au projet fait référence à des tas d'espaces de noms, notamment ceux du Toolkit. Il est fort probable que votre projet ne les références pas tous à cet instant précis de votre développement.

Globalement deux solutions s'offrent à vous : Ajouter l'ensemble des espaces de noms manquant dans les références du projet ou bien supprimer tous les styles du thème qui posent problème (et que vous n'utilisez pas pour le moment sinon votre projet aurait déjà la référence de leur espace de nom).

Personnellement je vous déconseille la seconde approche. Qui vous dit que dans deux jours ou deux mois vous n'aurez pas besoin d'ajouter tel ou tel autre contrôle ? Si vous avez supprimé les définitions de style dans le thème il faudra les recréer. Pour l'instant je vous conseille donc uniquement d'ajouter toutes les références qui manquent.

Bien entendu en ajoutant toutes les DLL du Toolkit à votre projet celui-ci risque d'enfler un peu ! Ce n'est pas forcément souhaitable, un gros fichier Xap est toujours un handicap. Mais laissez les choses en l'état et pensez à faire le ménage une fois la V 1.0 de votre application sera prête à être releasée. C'est plus sage.

Une fois tous les espaces de noms référencés correctement dans le projet ce dernier devrait se compiler à nouveau correctement.

A noter une petite curiosité dans le thème que j'ai utilisé dans l'exemple : en début de manœuvre je vous ai dit qu'il fallait déréférencer la DLL du thème puisque nous allions utiliser le code source Xaml. C'est vrai. Mais pour raison étrange le code source XAML référence lui-même la DLL ce qui dans la dernière étape oblige à la remettre en place. Bien entendu j'ai tenté de la mettre en commentaire trouvant la situation ridicule... Plantage à l'exécution. En référençant la DLL tout est rentré dans l'ordre. Je n'ai pas d'explication immédiate, je vous tiendrai au courant si je trouve (le temps de trouver) !

Pour connaitre la liste des espaces de noms nécessaires, ouvrez tout simplement dans Visual Studio le fichier thème Xaml que vous avez ajouté. En début de fichier se trouve toutes les références utilisées. Normalement les références manquantes sont soulignées d'une ligne ondulée d'erreur. Il suffit d'ajouter les références utilisées par l'ensemble de ces lignes plutôt que de farfouiller partout quel espace de nom est manquant (je pense que cela fonctionne dans VS de base et que ce n'est pas une aide de Resharper, outil indispensable que j'utilise depuis si longtemps qu'il m'est impossible de vous dire si un VS "nu" supporte ou non telle ou telle commande...).

Phase 7 : Pause café !

C'est fini pour la première partie, la plus dure. Il est temps de faire un petit Commit vers Subversion et d'aller se fumer un café (en prenant soin de se couvrir en conséquence, l'été est encore loin !). Les plus sobres iront boire un verre d'eau du robinet et s'abstiendront (avec raison) d'allumer une cigarette...

Seconde étape : La personnalisation proprement

Ca y est ? Vous êtes prêt ? Alors c'est bon, il ne vous reste plus qu'à modifier les styles du thème puisque désormais ils sont intégrés en code source dans votre projet (utilisez Blend de préférence).

Le billet s'arrête là... Vous amener jusqu'à ce point était son but, maintenant à vous de jouer !

Conclusion

Utiliser les DLL des thèmes est autrement plus pratique que d'utiliser les sources Xaml. Nous l'avons vu, la DLL sait se passer de tout un tas de référence que le code Xaml oblige à ajouter au projet. Moins simple à manier, plus lourd côté référence, la version source n'a pas que des avantages, mais elle laisse la porte ouverte à toutes les personnalisations du thème et cela peut être crucial.

Que votre motivation soit purement esthétique ou technique (comme je l'expliquais en début de billet dans mon cas), il arrivera fatalement un jour où vous aurez besoin de personnaliser un thème du toolkit. Ce jour-là, il vous suffira de retrouver ce billet et d'avoir une pensée pour votre serviteur grâce à qui vous pourrez aller tranquillement en pause-café au lieu de rester scotcher devant votre PC.

Lire un fichier en ressource sous Silverlight

Je ne sais pas pour vous, mais en tout cas à chaque fois que je dois intégrer un fichier dans une application Silverlight je suis obligé de réfléchir un peu. Faut-il mettre le fichier sur le server dans `ClientBin`, faut-il l'intégrer dans Visual Studio au projet et dans ce cas en mode `Content`, en mode `Resource` ? Et ensuite ? Trop de possibilités embrouillent toujours... Faisons le point !

ClientBin

Placer un fichier sur le serveur, généralement sous `/ClientBin` (mais ce n'est pas une obligation) est une bonne solution si la ressource est un peu "joufflue". Cela évite de grossir le fichier xap et simplifie aussi la mise à jour de la dite ressource (juste une copie de la nouvelle version sur le serveur).

Selon ce qu'on utilise pour visualiser / traiter la ressource, on utilisera par exemple un `WebClient` pour la télécharger.

Les exemples ne manquent pas, alors passons à la suite...

Content ou Resource ?

Quel casse-tête ! Comment choisir ?

D'abord il faut savoir que les deux solutions seront du même type, c'est à dire que la ressource en question sera placée dans le `xap` et qu'elle viendra l'alourdir. Cela tranche avec la solution précédente et fait une grande différence. Mais bien souvent, pour des fichiers de taille restreinte, choisir de placer la ressource dans le xap simplifie beaucoup le déploiement et accélère l'accès à la ressource : si le xap est lancé c'est qu'il est arrivé, et avec lui la ressource en question. Elle sera donc disponible immédiatement comme un fichier local, ce qui évite aussi la gestion asynchrone du `WebClient`.

Mais alors, `Content` ou `Resource` ?

J'y viens. Dans le cas du mode `Content` (qu'on choisit dans les propriétés de la ressource sous VS), le fichier sera copié dans le xap, à côté de la DLL de l'application (et des autres assemblages éventuels). Si vous changez l'extension du xap en "`zip`", vous pourrez ressortir facilement la ressource, voire la changer facilement.

Si vous décidez d'utiliser le mode `Resource`, le fichier sera stocké selon un mode classique dans les ressources de l'application. S'agissant d'une DLL (une application Silverlight n'a pas de "exe") on retrouvera ainsi la ressource dans la section `Resource` de la DLL. Plus difficile à extraire et à mettre à jour sans une recompilation. Pour voir la ressource, il faudra extraire la DLL du xap puis utiliser un outil comme Reflector.

Bon, alors, quel mode ?

Franchement, s'il est facile de trancher entre ressource externe (sur le serveur) ou interne, édicter une règle absolue pour orienter le choix entre le mode Resource ou Content est plus délicat.

Disons que le mode `Resource` est une sorte d'obligation (pour simplifier les choses) s'il s'agit d'une ressource utilisée par une DLL qui n'est pas la DLL principale de l'application. Par exemple vous avez dans votre application une seconde DLL qui contient l'accès aux données et celle-ci est capable de fournir des données de test en mode conception. Ces données de test sont stockées dans un fichier XML. Si vous le placez en mode `Content`, il faudra copier le fichier dans le xap, ce qui peut être oublié puisqu'on travaille principalement sur la DLL de l'application (les DLL utilitaires étant généralement créées et déboguer avant). Dans un tel cas je vous conseille de placer la ressource en mode `Resource`. Le fichier XML sera intégré à la DLL d'accès aux données, le fichier ne sera jamais visible et seule la DLL des données sera déployée. C'est plus propre, moins sujet à oubli.

Mais en dehors de ce cas de figure j'avoue que proposer une réponse toute faite n'est pas évidente. Alors tentons de résumer :

Le mode ressource : Le fichier est intégré à la DLL. Cela est utilisable aussi bien pour le projet principal que pour des librairies placées à l'intérieur ou à l'extérieur du package (le xap). De cette façon vous êtes certains que le ressource sera disponible immédiatement pour l'application. Ce mode est parfait à condition de ne pas avoir à mettre à jour la ressource après la compilation. L'autre avantage est la simplification du déploiement et la diminution des dépendances puisque la ressource est cachée dans sa DLL hôte.

Le mode Content : Le fichier est inclus dans le xap mais pas à l'intérieur d'une DLL. Cela est plus pratique si la même ressource est utilisée par plusieurs DLL du même xap par exemple (et évite ainsi sa duplication). Le fichier n'est pas compilé avec une DLL mais il faut savoir qu'il n'est pas juste ajouté dans le xap, il existe des références codées dans les métadonnées de ce dernier et on ne peut pas faire non plus

n'importe quoi... La principale raison d'utiliser le mode `Content` est donc de pouvoir mettre à jour la ressource après compilation et / ou de partager une même ressource entre plusieurs DLL du même xap.

Regardons comment accéder à un fichier intégré à l'application selon les deux modes.

Mode Content

La première chose à faire, quel que soit le mode, est de placer le fichier dans le projet. Je vous conseille de séparer les choses proprement en créant des sous-répertoires depuis VS (un répertoire pour les sons, un autre pour les images, un autre pour les textes...).

Une fois le fichier ajouté au projet il faut cliquer dessus dans l'explorateur de projet et regarder la fenêtre des propriétés. Là, il faut changer la propriété Build Action pour la mettre à "Content".

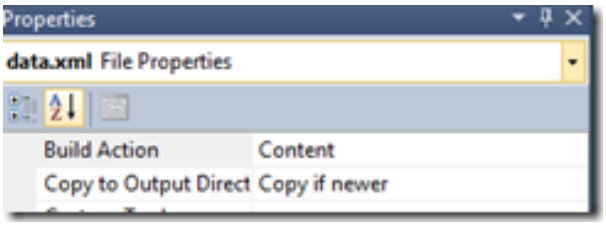


Figure 49 - Mode Content

Si nous prenons l'exemple d'un fichier XML placé directement dans le projet (sans créer un sous répertoire) l'ouverture peut se faire directement avec un `XDocument` ou un `XElement` :

```
XElement myElement = XElement.Load("TestData.xml");
```

Un fichier en mode `Content` se comporte donc comme un fichier "*local*" (en supposant le xap comme un petit univers recréant une sorte de mini disque dur). Si le fichier est dans un sous-répertoire on utilisera bien entendu ce dernier dans le nom pour le charger. Vous pourrez inspecter le fichier xap et vous verrez le sous-répertoire dans le zip/xap.

Mode Resource

Le début de l'histoire est le même. C'est le `Build Action` qui change et qu'il faut positionner à `Resource`.

Prenons ici l'exemple d'un fichier texte "`MonTexte.txt`" placé dans le projet sans sous-répertoire dédié et que nous souhaitons lire ligne par ligne.

L'accès est un peu plus "vicieux" qu'en mode `Content`. D'abord parce qu'ici on ne peut passer que par des flux (`Stream`) et que ceux-ci ne sont accessibles que via des URI. Doublement vicieux donc. Mais cela est triplement vicieux même ! Car la dite URI se doit d'avoir un format un peu spécial sinon jamais le fichier ne sera chargé...

Voici l'URI qu'il faut construire pour notre fichier texte :

```
Uri uri = new Uri("MonAppli;component/MonTexte.txt", UriKind.Relative)
```

On notera pour commencer que la chaîne de l'URI commence par le namespace de l'application (dans cet exemple "`MonAppli`"). Ce namespace est suivi d'un point-virgule.

Suit le mot "`component`" et enfin le nom du fichier. Il est précédé d'un slash car c'est en fait "`component`" qui est toujours suivi d'un slash... Si le fichier avait été placé dans le sous-répertoire *data* du projet, on trouverait donc "...`component/data/montexte.txt`".

Enfin, l'`UriKind` est en mode relatif (par rapport à la racine du projet dans son package).

Avoir une URI c'est bien joli. Et ça rime. Mais qu'en faire ?

Il nous faut maintenant obtenir et ouvrir un flux :

```
StreamResourceInfo streamInfo = Application.GetResourceStream(uri);
```

En réalité ce que nous obtenons par `GetResourceStream` n'est pas un `Stream` mais un `StreamResourceInfo`. Le stream lui-même est une propriété de ce dernier.

Il est donc nécessaire d'accéder au flux du `StreamResourceInfo` pour enfin accéder au contenu fichier ressource, mais avant cela il aura été indispensable d'obtenir un lecteur de flux (`StreamReader`) !

Ce qui donne :

```
if (null != streamInfo)
   {
      Stream stream = streamInfo.Stream;
      StreamReader sr = new StreamReader(stream);
      ...
```

Seulement maintenant nous pouvons, via le lecteur de flux, lire ligne à ligne notre texte :

```
...
      string line = String.Empty;
       while ((line = sr.ReadLine()) != null)
       {
          // travail sur "line"
       }
}
```

Vous noterez au passage la petite ruse syntaxique de C# qui permet d'utiliser le résultat d'un test sur une assignation en une seule opération (`line` est lue par `ReadLine` puis elle est comparée à `null`, ce qui marque la fin de fichier).

Et voilà... Un peu plus "tordu" mais comme nous l'avons vu plus haut la mise en ressource d'un fichier peut s'avérer plus intéressante que la mise en contenu ou même en ressource serveur.

Conclusion

Chacun fait s'qui lui plait-plait-plait.... disait la chanson. C'est à peu près ça. A vous de juger quelle méthode est la meilleure selon le projet.

Mais maintenant j'espère que vous saurez pourquoi vous choisissez une méthode plus que l'autre et que vous saurez vous en sortir pour accéder à vos ressources !

Silverlight : gérer une base de données locale

Tout le monde le sait, Silverlight ne gère pas de base de données, en tout cas directement. C'est une technologie "client" et si données il y a elles sont stockées côté serveur. La "glue" pour faire fonctionner l'ensemble s'appelle WCF, quelle qu'en soit la mouture. Or bien souvent les applications Silverlight tireraient profit d'une base de données purement locale. Il existe certaines solutions payantes, mais je vais vous présenter une solution simple, gratuite et open source : Silverlight Database (un nom vraiment dur à retenir !

Utiliser l'Isolated Storage

La première idée qui vient en tête est bien entendu d'utiliser l'Isolated Storage (IS) pour stocker la base de données. Mais ensuite ? Silverlight ne propose aucune des interfaces ni classes permettant de se connecter à moteur de base de données. Et il n'existe pas de solution out-of-the-box de "mini moteur de base de données" dans Silverlight.

Le cas de l'Out Of Browser (OOB)

Les applications Silverlight s'entendent aujourd'hui aussi bien en mode Web (le mode "classique") qu'en mode OOB, nécessitant l'installation sur la machine cliente. Ce dernier mode offre au développeur bien plus de possibilités mais il ne permet pas plus de se connecter à un SGBD local. Le cas de Silverlight sur Windows Phone 7 est un peu particulier mais il s'agit en réalité d'une mode OOB spécifique proposant des possibilités identiques (notamment dans la relation avec le hardware client et le code natif de celui-ci) mais aussi affichant les mêmes lacunes quant aux bases de données locales.

Je m'intéresserai ici aux applications Silverlight "classiques", Web. Mais la solution exposée peut fort bien être utilisée dans des applications OOB, sous Windows, Mac ou Windows Phone.

Les solutions payantes

Je me dois de les citer car certains d'entre vous auront peut-être besoin de plus de puissance que la solution que je vais exposer. Mais je dois avouer que je n'ai fait que survoler ces produits sans les tester à fond et je vous laisse vous faire votre avis tout seul...

Siaqodb

Elle se trouve là : http://siaqodb.com/

Il s'agit d'une base de données embarquée qui, sur le papier (je n'ai pas testé comme je le disais plus haut), semble offrir des possibilités assez riches. De $186 à $1822.50 selon les licences, c'est à la fois pas cher et fort couteux, tout dépend de l'usage qu'on doit en faire !

Perst

Elle se trouve ici : http://www.mcobject.com/perst

Il s'agit d'un produit commercial se voulant open source. Je dois être resté vieux jeu, mais pour moi de l'open source on doit pouvoir accéder au code source, et majoritairement cela implique la gratuité. Mais ici rien de tout cela. Je n'ai pas trouvé les sources (c'est de l'open closed...) et c'est payant, sauf la démo (of course). Le produit affiche des caractéristiques intéressantes, reste à le tester et à payer la licence (et à voir le fameux source qui est opened), $495 par développeur pour du SL classique, $395 pour du SL sur Windows Phone.

Effiproz SL

Elle est ici : http://www.effiproz.com/product_sl.aspx

Là encore le produit indique qu'il sait tout faire, reste donc à le comparer aux deux autres pour se forger un avis. A $490 la licence de base par développeur pour la version Silverlight à $950 pour la version qui marche partout (Windows Phone 7, Asp.NET, Mono, MonoTouch, Compact Framework), avec des intermédiaires aux prix différents sans raison technique (les autres le font aussi) selon la version spécifique que vous désirez acquérir. Par exemple pourquoi $390 pour WP7 et $490 pour SL classique ? Ils ont tous copiés la même grille tarifaire à peu de choses près, mais on se demande bien pourquoi ces différences. Techniquement, once more, à vous de vous faire une idée.

Le vrai gratuit et open source

Il n'y a pas grand-chose... le sujet ne semble pas passionner les développeurs de SourceForge, CodePlex ou autres, il est vrai que c'est moins "excitant" de prime abord que de lancer le centième projet de jeu hypra-cool-qui-tue ou de grosse-lib-qui-déchire-sa-mère, mais dont on ne voit jamais rien d'autre que les versions alpha sans release !

Pourtant les bases de données c'est un sujet passionnant. J'ai écrit de gros articles sur la question il y a un moment déjà et je me rends compte que des années après il y a toujours autant de méconnaissance des mécanismes et des fondements de ces applications particulières. Elles sont donc utilisées à tort et à travers, et même de travers le plus souvent : aucune normalisation ("c'est quoi la normalisation ?"), absence de cohérence dans les schémas, relations incohérentes dans le meilleur des cas et inexistantes le plus souvent, index placés n'importe comment, pas de clé primaire, ou clés farfelues, mauvaise utilisation (voire aucune utilisation) des transactions, etc, etc... En plus de 25 ans de métier je croyais sincèrement qu'un jour la "culture base de données" finirait par passer tellement cet outil est devenu commun, mais il faut bien constater, hélas, que seul un développeur sur 10 sait de quoi il s'agit...

Bref, j'arrête de faire mon vieux ronchon, et je reviens au sujet. Sorry pour la digression, mais ça m'énerve cette méconnaissance d'un des piliers de la réalisation d'un logiciel professionnel, et pas chez les bricolos du dimanche ou les amateurs, mais bien chez des gens qui prétendent avoir un diplôme d'ingénieur... C'est bon j'arrête  (mais ça m'énerve... ok! ok!).

Donc les bases données embarquées, light, locales, appelez-les comme vous voudrez, ça ne courre par les arcanes du Web. Surtout pour Silverlight.

Redéfinir le besoin

Quand je parle de base de données locales ou light ou embarquées, je parle de solutions extrêmement simples n'ajoutant pas 10 Mo de téléchargement à l'application et servant uniquement à persister des informations, donc des objets, de façon simple, afin de les retrouver tout aussi simplement à la prochaine exécution de l'application.

Je ne parle donc pas ici de "vraies" SGBD relationnel complexes supportant les transactions ni d'autres options luxueuses de type procédures stockées.

Il convient ainsi de redéfinir sérieusement le besoin. Quand on parle de base de données locale pour une application Silverlight classique (voir OOB), on évoque plus une zone de stockage qu'un vrai moteur SGBD. Une zone de stockage, il y en a une, elle s'appelle l'Isolated Storage, mais ce n'est qu'un File System dans une sandbox. On peut certes y ranger des données, mais ce n'est pas très structuré.

On a besoin en réalité d'une surcouche qui serait capable de persister des objets d'une façon intelligente, de telle sorte qu'on puisse retrouver ces objets plus tard. Et

encore s'agit-il de petits groupes d'objets. Vouloir stocker des milliers ou des millions d'enregistrements dans l'IS serait pure folie. Dans un tel cas il est évident qu'il faut utiliser WCF avec un serveur de données en face.

Ainsi précisé, le besoin se résume à un système capable de lire et d'écrire des objets regroupés dans des tables. Si le système en question est capable de compresser les données, voire de les crypter, cela serait encore mieux (rappelons que l'IS n'est pas un espace invincible, c'est juste un répertoire bien caché dans les méandres de Windows mais qu'il est possible d'atteindre et de corrompre ou simplement de lire).

Silverlight Database

C'est un petit projet, pas récent, et pas mis à jour depuis un moment, qu'on peut trouver sur CodePlex ici : http://silverdb.codeplex.com/

Il implémente une mini base de données dans l'Isolated Storage pour Silverlight, c'est ce que nous cherchons. Les sources sont disponibles et sont légères. C'est aussi l'un de nos critères. La base de données sait persister des objets et les relire dans des tables, cela nous convient aussi. Mieux, les données peuvent être codées (sécurité) et compressées (gain de place dans l'IS).

Le projet est même complété d'un service WCF totalement transparent qui permet de stocker ou d'obtenir des "bases de données" sur ou depuis un serveur. Cela peut être intéressant pour créer des sortes de catalogues, pas trop gros à télécharger justement, et ne contenant que les données utiles. On peut aussi envisager de sauvegarder ainsi les bases locales sur un serveur pour les centraliser.

Bref, les fonctionnalités de cette librairie semblent correspondent au besoin (une fois celui-ci correctement redéfini comme nous l'avons fait plus haut).

Compatibilité ascendante

Bien que les dernières sources datent du 26 septembre 2009 (il faut télécharger celles-ci et non pas la version 1.0, donc il faut aller dans l'onglet "source code" et prendre les dernières sources), le projet passe très bien l'upgrade vers Silverlight 4 et .NET 4. A cela une raison : un code simple (une leçon à retenir pour vos projets !).

Les fonctionnalités

Comme je le disais notre besoin est simple, par force et par logique, sinon il faut mettre en place un serveur avec un vrai SGBD. Face à nos besoins rudimentaires, on retrouve donc un nombre restreint de fonctions assurant le minimum vital. J'aime le minimalisme, vous devez commencer à le comprendre au travers de mes billets ! Les grosses usines à gaz, les frameworks réclamant six mois de formation, tout cela je le rejette avec force, c'est l'une des causes des prises de pieds dans le tapis sournois du développement... Moins les librairies sont nombreuses et moins elles sont grosses meilleur est le résultat.

Le mieux pour vous rendre compte étant bien entendu que vous téléchargiez le projet et que vous regardiez le code source ainsi que le code exemple. Mais pour vous donner un aperçu voici quelques extraits qui vous aiderons à comprendre l'esprit de la librairie :

Créer une base de données

```
if (Database.CreateDatabase("TestDatabase") != null) ... // c'est ok
```

Supprimer une base de données

```
Database.DeleteDatabase("test");
```

Créer une table

Supposons la classe suivante :

```
public class Person
   {
       public string FirstName { get; set; }
       public string LastName { get; set; }
       public DateTime? BirthDate { get; set; }
       public decimal Salary { get; set; }
   }
```

On créer une table des personnes dans une base de données de cette façon :

```
Database db = Database.CreateDatabase("test");
db.CreateTable<Person>();
if (db.Table<Person>() != null)
    {
        db.Table<Person>().Add(NewPerson());
        if (db.Table<Person>().Count == 1)
            {
              // base et table ok et une instance de Person ajoutée.
            }
        else
            {
              // erreur à gérer
            }
    }
    else
    {
        // erreur à gérer (IS plein par exemple)
    }
```

Créer une base cryptée, créer une table, persister des instances, sauvegarder le tout

```
Database db = Database.CreateDatabase("test", "password");
db.CreateTable<Person>();

if (db.Table<Person>() != null)
    {
       for (int i = 0; i < 20; i++)
          db.Table<Person>().Add(NewRandomPerson()); // ajout de 20
instances
       db.Save(); // sauvegarde de la base
    }
 else // gérer erreur
```

Supprimer un enregistrement

```
db.Table<Person>().Remove(person2);
```

Utiliser Linq

```
Database db = Database.CreateDatabase("test");
db.CreateTable<Person>();

Person person1 = new Person() { FirstName = "John ", ... };
```

```
Person person2 = new Person() { FirstName = "Robin ", ... };
Person person3 = new Person() { FirstName = "Joan ", ... };

db.Table<Person>().Add(person1);
db.Table<Person>().Add(person2);
db.Table<Person>().Add(person3);

var query = (from onePerson in db.Table<Person>()
             where onePerson.Salary == 2
             select onePerson);

foreach (var p in query) ...
```

Etc...

Enfin pour être honnête, le "etc..." se résume à deux ou trois autres combinaisons et à la partie WCF automatisée qui permet d'envoyer, de recevoir ou de mettre à jour une base de données avec un serveur distant (`GetDatabase`, `PutDatabase`, `UpdateDatabase`, encore des trucs compliqués à retenir !). SL Database supporte même une forme de Lazy Loading pour ne pas encombrer la mémoire inutilement. Le luxe !

Conclusion

Pourquoi payer plus cher ?

Si on reste cohérent, une base de données locale pour Silverlight ne doit rien faire de plus, sinon cela signifie qu'on a besoin de mettre en place un vrai circuit de gestion de données avec serveur distant, WCF Ria Services par exemple ou d'autres solutions (Web services) du même type.

Certes, avec "Silverlight Database" on reste dans le rudimentaire. Mais grâce à Linq To Object travailler sur des données prend tout de suite un air de SGBD même sur de simples listes... La séparation des données par classe (les tables) est pratique, l'encapsulation de tout cela dans un fichier unique est pratique, la compression et le cryptage sont de vrais plus dans le contexte de l'Isolated Storage, bref, minimaliste mais bien proportionné aux besoins.

Les données sont en mémoire, ne jamais l'oublier, jusqu'au "save" en tout cas. Mais même avec WCF Ria Services il en va de même.

Simple. Forcément performant (c'est en RAM), sécurisé et peu gourmand en place disque. L'option de transfert de bases avec un serveur est une cerise le gâteau...

En effet, pourquoi payer plus cher...

Silverlight : L'Isolated Storage en pratique

Je vous ai déjà parlé en détail de l'Isolated Storage de Silverlight dans le billet "Silverlight : Isolated Storage et stockage des préférences utilisateur", je voudrais vous présenter aujourd'hui quelques classes "helper" pour tirer parti rapidement des nombreuses possibilités de l'IS dans vos applications.

L'Isolated Storage

Je ne reviendrai pas sur les détails déjà exposés dans le billet cité plus haut auquel je renvoie le lecteur désireux d'en savoir plus sur ce "stockage isolé". Mais en deux mots rappelons que l'Isolated Storage est une zone de stockage locale (côté client) dédié à chaque application Silverlight. L'application n'a pas besoin de droits particuliers pour y accéder. L'I.S. a aussi l'avantage de pouvoir globalisé pour un nom de domaine.

C'est à dire que vous pouvez stocker des données spécifiques à l'application en cours ou bien spécifique à tout un domaine, par exemple www.e-naxos.com. Dans ce cas toutes les applications Silverlight qui proviendront de ce domaine pourront partager (lecture / écriture) les données stockées.

On peut utiliser cet espace "domaine" pour stocker les préférences utilisateur qui seront valable pour toutes les applications du domaine (choix de la langue, préférence de couleurs, etc.).

L'I.S. est limité en taille par défaut, et si une application désire plus d'espace elle doit le demander à l'utilisateur qui peut refuser.

Autre aspect : l'I.S. bien que "caché" dans les profondeurs des arborescences de l'hôte est tout à fait accessible, il est donc fortement déconseillé d'y stocker des valeurs de type mot de passe, chaine de connexion ou autres données sensibles (en tout cas sans les crypter).

SI le stockage des préférences de l'utilisateur est une fonction principale de l'I.S. il ne faut pas oublier d'autres utilisations comme le stockage de données en attente de synchronisation avec un serveur par exemple, l'exploitation de l'espace I.S. comme

d'un cache pour des données distantes longues à télécharger (images, fichiers de données, voire DLL ou Xap annexes d'une application modulaire...).

La gestion de l'espace

Parmi les fonctions de base indispensable à une bonne gestion de l'I.S. il y a celles qui gèrent l'espace disponible.

La classe helper suivante permet ainsi d'obtenir l'espace libre dans l'I.S. de l'application en cours, l'espace actuellement utilisé et de faire une demande d'extension de taille à l'utilisateur.

```
// Classe      : IsManager
// Description : Manages Isolated Storage space
// Version     : 1.0
// Dev.        : Olivier DAHAN - www.e-naxos.com
// ---------------------------------------------


using System.IO.IsolatedStorage;
using GalaSoft.MvvmLight;

namespace Utilities
{
    /// <summary>
    /// Simple Isolated Storage Manager.
    /// Mainly to manage IS space.
    /// </summary>
    public static class IsManager
    {
        /// <summary>
        /// Gets the free space.
        /// </summary>
        /// <returns></returns>
        public static long GetFreeSpace()
        {
            if (ViewModelBase.IsInDesignModeStatic) return 0L;
            using (var store =
                 IsolatedStorageFile.GetUserStoreForApplication())
            {
                return store.AvailableFreeSpace;
            }
        }

        /// <summary>
        /// Gets the used space.
        /// </summary>
        /// <returns></returns>
        public static long GetUsedSpace()
        {
            if (ViewModelBase.IsInDesignModeStatic) return 0L;
            using (var store =
                   IsolatedStorageFile.GetUserStoreForApplication())
```

```
            {
                return store.UsedSize;
            }
        }

        /// <summary>
        /// Extend IS quota.
        /// </summary>
        /// <param name="bytes">The bytes.</param>
        /// <returns></returns>
        public static bool ExtendTo(long bytes)
        {
            if (ViewModelBase.IsInDesignModeStatic) return false;
            using (var store =
                   IsolatedStorageFile.GetUserStoreForApplication())
            {
                return store.IncreaseQuotaTo(bytes);
            }
        }

    }
}
```

On notera que la classe proposée est statique, comme ses méthodes qui s'utilisent donc directement. L'unité de toutes les méthodes est l'octet.

Les paramètres utilisateurs

Une fois l'espace géré, l'utilisation la plus fréquente de l'I.S. est le stockage des préférences utilisateurs. Je vous propose ici une petite classe qui implémente le pattern Singleton et autorise la lecture et l'écriture de tout paramètre.

C'est un singleton et non une classe statique pour une raison simple : le binding Xaml ne fonctionne pas sur les statiques... Il faut ainsi une instance pour faire un binding sous Silverlight. La classe créée une instance d'elle-même accessible depuis un point d'entré connu (le singleton) ce qui permet de binder directement les champs d'une Vue sur la classe `ISParameter` (qui est alors un Modèle. Sous MVVM la liaison directe Vue/Modèle sans passer par un ViewModel – Modèle de Vue – est parfaitement licite).

```
namespace Utilities
{
    /// <summary>
    /// Isolated Storage Manager for user's parameters
    /// </summary>
    public class ISParameters
    {
        #region Singleton + init
        private static readonly ISParameters instance = new ISParameters();
        /// <summary>
        /// Gets the instance.
        /// </summary>
        /// <value>The instance.</value>
        public static ISParameters Instance { get { return instance; } }
        private ISParameters()
        {
            // init defaut values
            InitDefaultParameterValues();
        }

        /// <summary>
        /// Inits the default parameter values.
        /// </summary>
        public void InitDefaultParameterValues(bool factoryReset=false)
        {
            if (factoryReset)
            {
                if (userSettings.Contains("PARAM1"))
                            userSettings.Remove("PARAM1");
            }

            if (!userSettings.Contains("PARAM1"))
                userSettings.Add("PARAM1", TimeSpan.FromSeconds(30));

            userSettings.Save();
        }

        #endregion

        #region fields
```

```
        private readonly IsolatedStorageSettings userSettings =
                            IsolatedStorageSettings.ApplicationSettings;
        #endregion

        #region public methods
        /// <summary>
        /// Gets the parameter.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="name">The name.</param>
        /// <returns></returns>
        public T GetParameter<T>(string name)
        {
            try
            {
                return userSettings.Contains(name)
                            ? (T) userSettings[name]
                            : default(T);
            } catch(Exception e)
            {
                ISLogger.Log(e.Message,LogLevel.Error);
                return default(T);
            }
        }

        /// <summary>
        /// Stores the parameter.
        /// </summary>
        /// <param name="name">The name.</param>
        /// <param name="value">The value.</param>
        public void StoreParameter(string name, object value)
        {
            try
            {
                if (userSettings.Contains(name))
                    userSettings[name] = value;
                else userSettings.Add(name, value);
                userSettings.Save();
            }
            catch(Exception e)
            {
```

```
                ISLogger.Log(e.Message, LogLevel.Error);
            }
        }
        #endregion
    }
}
```

Le principe d'utilisation de la classe est le suivant : on utilise "`StoreParameter`" pour stocker la valeur d'un paramètre dans l'I.S., on utilise la méthode "`GetParameter`" pour lire un paramètre. Cette dernière est générique ce qui permet de récupérer le paramètre directement dans son type natif (il faut le connaitre mais c'est généralement le cas).

Lorsqu'on gère des paramètres dans une application il y a forcément des valeurs par défaut. C'est la raison d'être de la méthode "`InitDefaultParemeterValues`" qui est appelée par le constructeur.

Il est aussi nécessaire de permettre à l'utilisateur de remettre les paramètres à leur valeur par défaut. C'est pourquoi cette même méthode est publique et peut ainsi être invoquée à tout moment pour rétablir l'ensemble des paramètres à leur valeur "usine". Le paramètre "`factoryReset`" est là pour forcer la remise à défaut justement (sinon la procédure ne fait qu'écrire les valeurs par défaut lorsqu'elles n'existent pas).

A noter : il faut modifier manuellement cette méthode pour y ajouter tous les paramètres de l'application ainsi que leur valeur par défaut. Le code ne comporte qu'un exemple pour le paramètre "`PARAM1`" qui serait de type `TimeSpan`. Il suffit de recopier ce code autant de fois que nécessaire et de l'adapter à chaque paramètre.

Cette contrainte à un avantage : centraliser en un seul point la liste de tous les paramètres...

Enfin, on remarquera que les exceptions sont gérées et font appel à une mystérieuse classe `ISLogger`. Vous pouvez supprimer cette gestion si vous le désirez, sinon passons à la troisième classe qui se trouve être la fameuse `ISLogger`...

La gestion des Logs

Gérer des logs est le propre de toute application pour en simplifier le debug. On pourrait intégrer quelque chose comme Log4Net, mais il n'y a pas de version Silverlight. Certes on peut utiliser Clog (voir sur Codeplex) pour envoyer les messages au serveur via WCF. Mais si l'application plante, logger des messages en utilisant un

mécanisme aussi complexe a peu de chance de fonctionner à tous les coups (par exemple si justement ce sont les connexions réseaux qui ne marchent pas !).

Dans mes applications je résous le problème de deux façons : en utilisant un logger simplifié qui stocke les informations dans l'I.S. et un autre mécanisme qui utilise les connexions de l'application. Par exemple WCF Ria Service ou un simple Web Service selon les cas. Je créé alors un Logger global qui s'occupe de stocker l'erreur en local en premier puis qui tente de l'envoyer au serveur. Comme cela les erreurs sont reportées au serveur si c'est possible, mais si cela ne l'est pas je dispose d'une trace utilisable sur chaque client. Le fichier de log local peut être présenté à l'écran ou bien sauvegardé ailleurs, mais il ne s'agit plus que de présentation sous Silverlight.

La partie qui nous intéresse ici se concentre sur le Log local stocké dans l'I.S.

Rien ne sert de faire compliqué. On pourra toutefois sophistiquer un peu plus la classe suivante pour lui permettre de gérer les `InnerException` s'il y en a. Cela peut être pratique. J'ai préféré vous présenter la version simple, à vous de l'améliorer selon vos besoins (idem pour le formatage du fichier de log qui est ici un simple fichier texte, on peut concevoir des variantes en XML par exemple).

Notre petit logger a besoin d'une énumération qui indique la gravité du message :

```
namespace Utilities
{
    /// <summary>
    /// Log Level
    /// </summary>
    public enum LogLevel
    {
        /// <summary>
        /// Information (lowest level)
        /// </summary>
        Info = 1,
        /// <summary>
        /// Warning (perhaps a problem)
        /// </summary>
        Warning = 2,
        /// <summary>
        /// Error (exceptions not stopping the application)
        /// </summary>
        Error = 3,
        /// <summary>
        /// Fatal errors (exceptions stopping the application, should be
log to the server if possible)
        /// </summary>
        Fatal = 4
    }
```

Le code qui suit n'exploite pas réellement de filtrage sur le niveau de log, à vous de compliquer les choses selon vos besoins !

```
using System;
using System.Collections.Generic;
using System.IO.IsolatedStorage;
using System.IO;
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Messaging;

namespace Utilities
{

    /// <summary>
    /// Simple logging class using Isolated Storage to keep track of
exceptions and important messages.
    /// </summary>
    public static class ISLogger
    {

        private const string LOGNAME = "TenorLight.log";

        /// <summary>
        /// Logs the specified message.
        /// </summary>
        /// <param name="message">The message.</param>
        /// <param name="logLevel">The log level.</param>
        public static void Log(string message, LogLevel logLevel)
        {
            if (IsManager.GetFreeSpace() < (message.Length * 3))
            {
                Messenger.Default.Send(new NotificationMessage("ISFULL"));
                return;
            }
            if (!ViewModelBase.IsInDesignModeStatic)
                try
                {
                    using (var store =
                       IsolatedStorageFile.GetUserStoreForApplication())
                    {
                        using (Stream stream =
                              new IsolatedStorageFileStream(
                                      LOGNAME, FileMode.Append,
                                      FileAccess.Write, store))
```

```
                    {
                        var writer = new StreamWriter(stream);
                        switch (logLevel)
                        {
                            case LogLevel.Info:
                                writer.Write(String.Format(
                                        "{0:u} [INFO] {1}{2}",
                                         DateTime.Now, message,
                                         Environment.NewLine));
                                break;
                            case LogLevel.Warning:
                                writer.Write(String.Format(
                                "{0:u} [WARNING] {1}{2}", DateTime.Now,
                                 message, Environment.NewLine));
                                break;
                            case LogLevel.Error:
                                writer.Write(String.Format(
                                  "{0:u} [ERROR] {1}{2}", DateTime.Now,
                                  message, Environment.NewLine));
                                break;
                            case LogLevel.Fatal:
                                writer.Write(String.Format(
                                 "{0:u} [FATAL] {1}{2}", DateTime.Now,
                                 message, Environment.NewLine));
                                break;
                            default:
                                break;
                        }
                        writer.Close();
                    }
                }
            }
            catch (Exception e)
            {
                Messenger.Default.Send(
                 new NotificationMessage<Exception>(e, "LOGERROR"));
            }
        }

        /// <summary>
        /// Gets the log.
```

```
        /// </summary>
        /// <returns></returns>
        public static List<string> GetLog()
        {
            var li = new List<string>();
            if (!ViewModelBase.IsInDesignModeStatic)
                try
                {
                    using (var store =
                         IsolatedStorageFile.GetUserStoreForApplication())
                    {
                        using (
                            var stream =
                               new IsolatedStorageFileStream(LOGNAME,
                                  FileMode.OpenOrCreate, FileAccess.Read,
                                   store))
                        {
                            var reader = new StreamReader(stream);
                            string s;
                            while ((s = reader.ReadLine()) != null)
                                        li.Add(s);
                            reader.Close();
                        }
                    }
                }
                catch (Exception e)
                {
                    Messenger.Default.Send(new
                       NotificationMessage<Exception>(e,"LOGERROR"));
                }
            return li;
        }

        /// <summary>
        /// Clears the log.
        /// </summary>
        /// <returns></returns>
        public static bool ClearLog()
        {
            if (!ViewModelBase.IsInDesignModeStatic)
                try
```

```
                {
                    using (var store =
                     IsolatedStorageFile.GetUserStoreForApplication())
                    {
                        if (store.FileExists(LOGNAME))
                                    store.DeleteFile(LOGNAME);
                        return true;
                    }
                }
                catch
                {
                    return false;
                }
            return false;
        }

        /// <summary>
        /// Lines the count.
        /// </summary>
        /// <returns></returns>
        public static int LineCount()
        {
            var log = GetLog();
            return log.Count;
        }

    }
}
```

Le logger s'utilise de la façon la plus simple, c'est une classe statique. Pour logger un message on utilise "`ISLogger.Log(monmessage,leniveau)`".

Quelques méthodes utiles complètent la classe : `LineCount()` qui retourne le nombre de lignes actuellement dans le log, `ClearLog()` qui efface le log, et `GetLog()` qui retourne le log sous la forme d'une grande chaine de caractères.

Avec ça on dispose d'une base pratique sur laquelle on peut "broder" pour l'adapter à toutes les situations.

Une première amélioration serait d'ajouter un lock pour éviter les collisions dans une application multi thread.

Vous remarquerez que certaines méthodes utilisent la messagerie MVVM Light en cas d'exception. Comme nous sommes dans la classe `Logger` la question se pose de savoir qui "logue les logs du loggers" ? !

Dans cet exemple les exceptions gérées par le logger lui-même sont simplement retransmises à l'application via une notification, un message MVVM Light. Les Vues peuvent intercepter ces messages et afficher une boite de dialogue ou ajouter le message à une liste ou autre mécanisme afin que l'utilisateur soit averti.

Si vous n'utilisez pas MVVM Light vous pouvez supprimer les appels à `Messenger` bien entendu. Il faut alors décider si vous laisserez les blocs `try/catch` vide (pas toujours très malin de cacher les erreurs) ou bien si vous substituerez les appels à `Messenger` par un autre mécanisme propre à votre application ou bien encore si vous déciderez de supprimer les blocs `try/catch` pour laisser les exceptions remonter vers l'application (ce qui permet d'en prendre connaissance).

Le stockage de fichiers

Enfin, l'I.S. peut être utilisé comme un cache local par exemple pour éviter à l'utilisateur d'avoir à supporter les attentes du téléchargement d'éléments externes lorsque ceux-ci ont déjà été chargés une fois (images utilisées par l'application mais non intégrées au XAP par exemple).

La classe statique suivante propose deux méthodes très simples : l'une pour stocker des données dans un fichier, l'autre pour les lire.

L'exemple traite les données sous la forme d'une grande chaine de caractères. Bien entendu c'est un exemple, vous pouvez gérer des arrays de Bytes par exemple pour lire et stocker des fichiers binaires (jpeg, png, wav, mp3 ...).

Il suffit d'adapter les deux méthodes selon vos besoins. Dans la version présentée on peut lire et écrire des fichiers texte simplement (txt, Xml, Xaml...).

```
using System.IO.IsolatedStorage;
using System.IO;

namespace Utilities
{
    public static class ISHelper
    {
        public static void StoreIsolatedFile(string path, string data)
        {
            var iso = IsolatedStorageFile.GetUserStoreForApplication();
            using (var isoStream = new IsolatedStorageFileStream(path,
                                FileMode.Create, iso))
            {
                using (var writer = new StreamWriter(isoStream))
                {
                    writer.Write(data);
                }
            }
        }

        public static string ReadIsolatedFile(string path)
        {
            var iso = IsolatedStorageFile.GetUserStoreForApplication();
            using (var isoStream = new IsolatedStorageFileStream(path,
                     System.IO.FileMode.Open, iso))
            {
                using (var reader = new StreamReader(isoStream))
                {
                    return reader.ReadToEnd();
                }
            }
        }
    }
}
```

Conclusion

Rien de bien savant aujourd'hui, du pratique et de l'efficace, qui ne fait pas grossir inconsidérément la taille de votre application Silverlight !

Il est toujours possible de gérer tout cela plus finement, mais un XAP qui n'en fait pas trop est un XAP qui reste petit, et donc qui se télécharge vite, ce que l'utilisateur apprécie toujours !

Quand browsers et proxies jouent à cache-cache avec vos Xap…

Lorsqu'on travaille avec Silverlight on peut facilement devenir fou à cause d'un proxy ou d'un browser dont le cache est mal géré. En vous resservant un vieux Xap sans vous le dire, vous avez l'impression que votre session de debug ne sert à rien ou bien vos utilisateurs vous décrivent des bugs pourtant réglés depuis un moment… Dans tous les cas un proxy (ou un browser) est en train de jouer à cache-cache avec vos Xap et vos nerfs !

Vider moi ce cache !

Le bon réflexe est de vider le cache du browser. Si le problème ne vient pas d'un proxy sur la route entre le serveur et le client cela va arranger les choses. Mais d'une part c'est fastidieux et, d'autre part, c'est encore pire s'il faut l'expliquer à un utilisateur.

De plus cela doit être recommencé à chaque changement dans le Xap, autant dire qu'en développement cela peut s'avérer très lourd ! Heureusement le problème ne se pose pas avec IE dans ses versions récentes. Mais il arrive qu'on debug aussi avec d'autres browsers.

Bref, vider le cache du butineur, cela peut être utile, mais c'est à la fois lourd et pas toujours efficace.

Tromper les caches

Voilà une alternative plus intéressante. C'est un peu comme passer à fond devant un radar mais avec de fausses plaques. Le radar est toujours là, mais il ne sait pas que c'est "vous". Bien entendu ce jeu là est interdit. Tromper les caches des browsers et des proxies est une opération largement moins risquée ! Le principe est le même, passer à fond devant le contrôleur sans qu'il vous reconnaisse. Mais sans les ennuis avec la police.

Tromper, c'est mal, mais ça peut aussi être utile, nécessaire voire même héroïque.

Nécessité faisant loi, nous allons donc tromper les caches !

Mais comment tromper un cache ?

Tromper un cache c'est facile direz-vous, un cache c'est bête.

Oui, mais par nature un système qui contrôle, radar, cache ou autre, plus c'est bête, plus c'est dur à tromper justement ! Car ça marche de façon automatique, sans se poser de question.

On peut embrouiller l'esprit de celui qui par intelligence pratique le doute métaphysique, mais avec un cache, vous pouvez toujours tenter une question du type "l'existentialisme est-il un humanisme ?" ... vous attendrez longtemps la réponse ! Il va rester concentrer sur son job le bougre (essayez avec un radar aussi, vous verrez bien).

De fait, les caches, de browsers ou de proxies, sont des butors obstinés. On les fabrique pour ça d'ailleurs. Mais certains ne sont pas exempts de défauts. Et votre mission, si vous l'acceptez, sera de faire passer un vieux Xap pour un Xap tout neuf ne devant pas être caché.

En réalité non : vous voulez que les Xap neufs ne passent pas pour de vieux Xap justement, tout l'inverse ! A moins que cela ne revienne au même ici ? ... A méditer  !

Quelques balises de plus

Le plugin Silverlight s'instancie généralement dans une balise `<object>` même sous Asp.Net (bien qu'il existe un contrôle plus direct). De même, le plugin peut être placé dans une page Html classique mais il est souvent préférable de le placer dans une page ASPX pour la souplesse que cela autorise.

Justement ici nous supposerons que le plugin est instancié par une balise <object> dans une page ASPX ! C'est inouï les coïncidences parfois (comme les phrases avec deux ï) !

Une balise classique

Voici à quoi ressemble une form Asp.Net avec balise `<object>` classique pour instancier une application Silverlight via son plugin :

```
<form id="form1" runat="server" style="height:100%">
   <div id="silverlightControlHost">
     <object data="data:application/x-silverlight-2," type="application/x-
silverlight-2"
                width="100%" height="100%">
           <param name="source"
value="ClientBin/SilverlightApplication2.xap"/>
           <param name="onError" value="onSilverlightError" />
           <param name="background" value="white" />
           <param name="minRuntimeVersion" value="4.0.50826.0" />
           <param name="autoUpgrade" value="true" />
           <a
href="http://go.microsoft.com/fwlink/?LinkID=149156&v=4.0.50826.0"
                 style="text-decoration:none">
                           <img
src="http://go.microsoft.com/fwlink/?LinkId=161376"
                           alt="Get Microsoft Silverlight" style="border-
style:none"/>
           </a>
      </object>
      <iframe id="_sl_historyFrame"
style="visibility:hidden;height:0px;width:0px;border:0px">
      </iframe>
  </div>
</form>
```

Deux situations

Il y a deux situations particulières à gérer : le debug et l'exploitation. On peut vouloir mettre en œuvre l'astuce dans l'un ou l'autre des cas ou dans les deux.

A vous de décider et surtout de savoir si cela est justifié dans votre cas précis.

Une seule approche

Elle consiste à passer un paramètre "bidon" dans le nom du Xap, paramètre ayant à chaque fois une valeur différente.

Pour ce faire nous allons utiliser la possibilité d'intégrer du C# directement dans le code HTML d'une page Asp.Net, code C# qui modifiera l'écriture du code HTML, agissant ainsi comme un préprocesseur très sophistiqué. En réalité c'est le mode de programmation Asp.Net des "débuts", avant le "code behind". A la mode PHP donc. J'ai horreur de ça, mais il faut avouer que là, précisément, c'est utile.

Dans l'exemple de code plus haut, on voit que la balise déclarant l'application Silverlight utilise une série de sous balises `<param>` permettant de fixer les paramètres qui vont être transmis au plugin. Parmi ceux-ci se trouve le paramètre ayant le nom de "`source`" et fixant le nom du fichier Xap à charger.

C'est cette balise là que nous allons substituer par un bout de code C# comme celui-ci :

```
<%
    const string strSourceFile = @"ClientBin/AppliSL.xap";
    string param;
    if (System.Diagnostics.Debugger.IsAttached)
    param = "<param name=\"source\" value=\"" + strSourceFile + "?"
                                        + DateTime.Now.Ticks + "\" />";
    else
    {
       var xappath = HttpContext.Current.Server.MapPath(@"") + @"\" +
strSourceFile;
       param = "<param name=\"source\" value=\"" + strSourceFile +
"?ignore="
                + System.IO.File.GetLastWriteTime(xappath) + "\" />";
    }
    Response.Write(param);
%>
```

... Mais deux façons de faire

En effet, vous remarquerez que suivant que le mode Debug est reconnu ou pas la même astuce est utilisée mais avec deux techniques légèrement différentes.

D'abord il est intéressant de faire la séparation entre debug et mode d'exploitation normal. On peut vouloir utiliser l'astuce "anti cache" dans un cas et pas dans l'autre. Avec le code tel qu'il est écrit il est très facile de le faire.

Vous noterez au passage que le code ci-dessus remplace totalement la balise `<param>` "`source`" puisque son rôle est justement de la réécrire totalement (le `Response.Write()` à la fin).

Ensuite, on voit qu'en cas de debug on ajoute "`?`" plus les ticks de l'horloge au nom du Xap alors que dans mode normal on utilise une technique plus complexe.

Pourquoi cette différence ?

Dans le cas du debug ce qui compte c'est de berner le cache en lui faisant croire que le Xap est différent à chaque fois (ce qui est généralement vrai puisque justement on est en train de le tripoter). Il s'agit ici de régler un problème de cache local dans le browser. Ce problème n'est pas systématique et arrive avec certains browsers. Vous n'êtes pas obligés d'implémenter l'astuce, il suffit de ne pas ajouter les ticks derrière le nom du Xap...

Dans le second cas nous nous plaçons dans l'optique d'une utilisation en exploitation. Il y a donc un serveur avec votre Xap quelque part, et ailleurs un utilisateur avec son browser. Entre les deux il y a la terre... parcourue de câbles, de fibres optiques, survolées par des satellites le tout saupoudré de machines diverses et variées et pourquoi pas de proxies.

Les caches sont utiles : ils diminuent les temps de chargement en rapprochant les ressources du point où elles sont consommées. Tromper les caches aveuglément pénalisera vos applications en obligeant un chargement complet depuis vos serveurs.

L'astuce simplette des ticks de l'horloge parfaitement adaptée pour le debug serait contreproductive en exploitation. Il faut laisser les caches cacher.

Sauf qu'on veut s'assurer que si le Xap a été modifié c'est bien la dernière version qui arrive dans le browser de l'utilisateur et non une version plus ancienne...

Pour régler le problème le code C# qui réécrit la balise `<param>` utilise une feinte plus intelligente que les ticks de l'horloge : la date de dernière écriture du Xap est récupérée et utilisée pour former le fameux paramètre "bidon" placé derrière le nom du fichier Xap.

Ainsi, le nom du *Xap + paramètre "bidon"* sera bien différent à chaque fois que le Xap aura été mis à jour, mais le reste du temps le fichier pourra continuer à être caché sans dégradation de performances.

Pour tester l'astuce il suffit de regarder le source de la page Web contenant l'application Silverlight en fonctionnement. Vous pourrez alors constater que le paramètre fixant le nom du fichier Xap comporte un "`?xxxxxx`" en fin de nom de fichier, les "`xxxxx`" étant soit un très grand nombre (les ticks en mode debug), soit "`ignore=<date>`" dans le mode "production".

On notera pour finir qu'en debug on passe juste un nombre alors qu'en mode d'exploitation on fait vraiment "semblant" jusqu'au bout en créant un paramètre

nommé (“ignore”) afin que les caches l’analysent correctement pensant à un véritable paramètre d’application comme cela se pratique communément dans les URL.

Conclusion

Il y a parfois des problèmes qui semblent insolubles car on n’a pas la “main” sur l’élément ou les éléments en cause. Ici le cache d'u browser ou la présence d’un proxy ... approximatif dans la chaine entre serveur et client. Toutefois, comme dans l’éternel jeu du gendarme et du voleur, il y a toujours de la place pour l’inventivité. Aucun système de contrôle ne peut tout avoir prévu. C’est une lueur d’espoir dans un monde de plus en plus régenté par des machines, qu’il s’agisse de radars, de caméra de vidéosurveillance ou de simples proxies...

MEF et Silverlight 4+ [Applications modulaires]

Code Source

Cet article est accompagné du code source complet des exemples. Si vous le recevez sans ces derniers il s’agit d’une copie de seconde main. Téléchargez l’original sur www.e-naxos.com.

Décompressez le fichier Zip dans un répertoire de votre choix. Attention les projets nécessitent Silverlight 4 et Visual Studio 2010 au minimum. Le dernier exemple utilise le toolkit MVVM Light de Laurent Bugnion.

Projets fournis

Exemple1 « `HelloWord` » , Importation de propriété et de champs

Exemple2 « `MultipleModules` », Importation de plusieurs modules

Exemple3 « `TypedMetaData` », Métadonnées fortement typées

Exemple4 « `CustomAttribute` », Attribut d’exportation personnalisé

Exemple5 « `DynamicXap` », Chargement dynamique et MVVM

Préambule

MEF, **Managed Extensibility Framework**.

> *MEF est un Framework managé (sous CLR donc) permettant d'améliorer l'extensibilité des applications.*

MEF est une technologie qui fut longtemps un projet séparé, disponible pour WPF et Silverlight sur CodePlex. Arrivé à maturité, MEF a été introduit dans Silverlight 4 (et dans le Framework .NET 4.0). Il devient donc une brique essentielle de l'édifice.

Donc tout le monde connaît l'acronyme, tout le monde l'a déjà lu, entendu, prononcé. Mais combien de lecteurs à cet instant précis et sans se référer à une documentation ou des tutoriaux pourraient construire une application utilisant MEF ?

Certainement très peu car si le mot est connu, et le principe vaguement aussi, peu de développeurs connaissent suffisamment bien la façon d'utiliser MEF pour ne serait-ce qu'envisager de l'intégrer dans leurs applications, tout de suite, sans y réfléchir.

Et c'est dommage. D'autant que l'utilisation de MEF peut soulever d'autres questions comme celle du Lazy loading, ou l'utilisation du cache d'assemblage en conjonction avec MEF, ou encore comment marier MEF et un framework MVVM ou encore plus réaliste, comment offrir des modules MEF différents selon les profils des utilisateurs ? Questions qui réclament un peu d'expérience avec la bête pour pouvoir y répondre.

Il est vrai que la fourniture indépendante de MEF pendant longtemps, avec des variations notables qui ont fait que telle démo ne tournait plus ou que tel tutoriel était devenu caduque n'a pas arrangé les choses. On s'y intéresse un jour, on fait des exemples, et quand on en a besoin, plus rien ne marche avec la dernière version. C'est le côté agaçant de ces produits finis mais en cours d'élaboration, diffusés officiellement mais pas encore intégrés au Framework. Le prix à payer pour savoir à l'avance ce qui sortira un jour, mais un savoir remis en cause à chaque release. Le seul bénéfice est que ce long processus permet aux équipes de MS d'affiner le produit et de l'intégrer au Framework qu'une fois une certaine maturité atteinte. Nous sommes gagnants à l'arrivée, avec un Framework riche et solide. Mais l'entre-deux n'est pas sans poser de problèmes, je le concède volontiers (par exemple l'un de mes billets écrit en 2008 sur MEF n'est plus directement utilisable avec le MEF officiel intégré à SL4).

L'autre aspect négatif de cette fourniture diluée dans le temps est qu'en réalité la version finale intégrée au Framework est une « vieille nouveauté ». J'ai évoqué plus haut mon billet de 2008 sur MEF… 2008 ça fait longtemps ! Qui sera intéressé par un nouvel article sur MEF alors que mon vieux billet, comme la majorité des autres, est sur le Web de longue date et y est bien référencé avec le temps, alors même que plus aucun ne donne de conseils fiables pour la version d'aujourd'hui ? Ceux qui tenteraient maintenant d'utiliser MEF en en lisant ce qu'ils trouveraient facilement sur le Web concluraient rapidement que ce n'est pas « stable » ou que « ça ne marche pas ». Même la documentation sur CodePlex n'est pas vraiment à jour…

Il n'y a pas de choix parfait et Microsoft fait des efforts louables pour faire évoluer la plateforme sans la polluer avec des nouveautés instables et immatures. Pour cela il faut bien sortir des versions intermédiaires, déconnectées du Framework en se laissant la possibilité de faire des « breaking changes » justement pour atteindre la maturité minimale qui honore tant le Framework. Mais il y a des dommages collatéraux comme ceux que je viens d'évoquer…

Bref, une application complète est complexe. Elle doit marier plusieurs technologies pour atteindre ses objectifs : MEF, WCF Ria Services, Linq to Entities, Linq to Object, Linq to Xml, MVVM, Entity Framework, Injection de dépendance, Framework de navigation… Chacune de ces techniques ou technologies sont parfois difficiles à maîtriser en elles-mêmes, alors que dire de la difficulté à les marier ensemble dans un tout cohérent sans tomber dans le code spaghetti !

Le présent article n'a pas vocation à répondre directement à cette dernière interrogation, il faudrait plusieurs livres pour en venir à bout certainement. Puis un livre sur ces livres, une prise en main. Puis un tutoriel sur le livre sur les livres… Au final on serait de nouveau noyé. Car tel l'Océan, on ne peut pas décider de maîtriser Silverlight en s'entraînant dans sa baignoire puis dans une piscine, puis un lac, etc… Un vrai marin de haute mer se forme en haute mer. Et seuls ceux qui y survivent peuvent dire qu'ils sont marins.

Des technologies comme Silverlight sont tellement vastes (en intégrant toutes les technologies satellites) qu'à la fois on est bien obligé de l'aborder par petits bouts, tout en ne pouvant considérer y comprendre vraiment quelque chose qu'une fois qu'on a réellement navigué sur cet Océan.

Rien de désespérant ou de paradoxal dans ce constat, marin de haute mer est un métier, une passion, et cela se mérite. La sécurité absolue, la connaissance parfaite n'existent pas, la prise de risque existe toujours. C'est ce qui en fait toute l'exaltation et toute la valeur légendaire de ceux qui rentrent au port sains et saufs…

MEF – Le besoin

Nous avons vécu sans MEF pendant des décennies, qu'est-ce qui pourrait bien nous forcer à l'utiliser aujourd'hui ? Quel ou plutôt quels besoins nous forcent-ils à nous y intéresser ?

Nos logiciels prennent du poids… Ils sont de plus en plus « riches » de fonctionnalités diverses et variées, chacune devenant de plus en plus sophistiquée. Au final l'ensemble devient difficilement maintenable et pèse trop lourd pour une utilisation via l'Internet. Même en mode desktop, avoir un logiciel qui se charge vite et répond tout de suite est un challenge dès qu'il est obligé de charger et d'initialiser des tonnes de code.

Deux problèmes se posent alors : Celui du poids de l'ensemble et de son temps de chargement, et celui de la modularité. La maintenabilité étant un besoin tellement basique qu'il est presque déplacé de le citer encore, sauf d'admettre que cet objectif n'est pas la plus grande réussite des informaticiens, en général…

Qu'un logiciel puisse se charger rapidement est essentiel pour l'UX (l'Expérience Utilisateur). La vivacité, le fait d'être « *responsive* », de répondre immédiatement aux sollicitations de l'utilisateur fait la différence entre un bon logiciel bien conçu et un logiciel lambda.

Mais plus loin, le foisonnement des fonctionnalités impose de les *modulariser*. Pour rendre la maintenance plus simple, certes, mais aussi pour offrir « à la volée » de nouveaux modules sans avoir à modifier, recompiler, redéployer toute l'application ; pour limiter les modules chargés simultanément ; pour offrir à l'utilisateur les modules dont il a besoin (selon son profil, la licence qu'il a achetée, etc…).

Dans l'idéal on part d'une situation de ce type :

Figure 50 - L'application idéale

La gentille Carlita ????, après avoir bien travaillé, release la version 1.0 de sa superbe application. Mais très vite la situation devient la suivante :

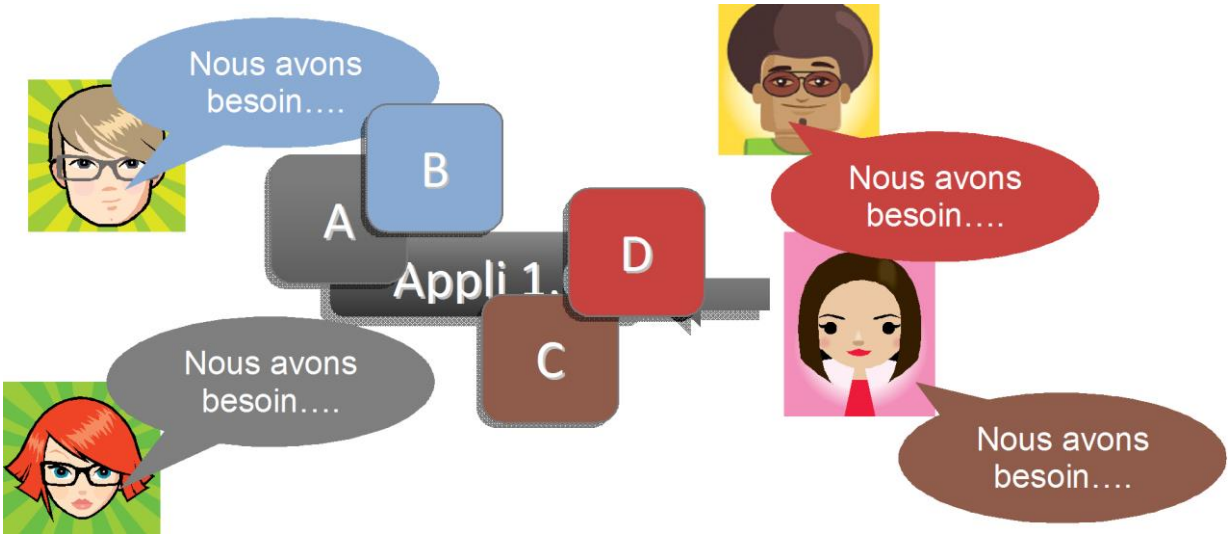


Figure 51 - Dans la vraie vie…

Autant les utilisateurs que les informaticiens s'aperçoivent à l'usage que telle ou telle fonctionnalité manque cruellement, que telle ou telle autre devrait être modifiée, etc… La pauvre Carlita se retrouve face à une situation explosive, les releases se succèdent pour supporter les « verrues » A, B, C, D, … Bientôt l'alphabet ne suffira plus !

MEF est une solution simple qui permet **de gérer la monté en puissance d'une application** et qui **accepte avec tolérance les extensions incrémentales**. C'est un bénéfice immédiat pour :

- Le client

- Le développeur et son équipe

Certes on savait comment charger des fichiers XAP dynamiquement avant MEF. Mais il ne s'agit que d'une des phases d'une gestion dynamique et modulaire : comment connaître la liste des modules disponibles (le *discovery* service), comment la filtrer selon le profil utilisateur, comment gérer les dépendances entre les modules (quel module doit aussi charger tel autre pour pouvoir fonctionner) ? … Bien des problèmes que le simple chargement dynamique d'un XAP ne saurait résoudre à lui seul.

> *Il faut d'ailleurs noter que si, lorsqu'on évoque MEF, surtout avec Silverlight, on pense au chargement dynamique de XAP, cela n'est qu'une utilisation spéciale. MEF fonctionne de base avec un seul XAP en lui permettant de bénéficier de cette même modularité simplificatrice.*

Pour gérer toutes les facettes de cette modularité il faut en réalité une couche logicielle adaptée. Tout comme nous utilisons désormais des Frameworks MVVM (MVVM Light dont j'ai parlé il y a quelques temps dans de gros articles, ou Caliburn.Micro dont je parlerai bientôt, ou Prism), il convient d'utiliser un Framework pour gérer « l'extensibilité » des applications.

Ce Framework existe, c'est MEF.

Forcément les petites démonstrations de base sont toujours simples et idylliques, se basant sur le principe faussement évident, qu'une fois qu'on a compris les petites démonstrations simples on est capable de faire une vraie application… Aussi vrai que jouer avec un bateau en plastique dans votre baignoire vous prépare à barrer un trois-mâts au cap Horn !

Mais il faut bien prendre le problème par un bout, commencer doucement.

C'est pourquoi nous passerons ici aussi par l'étape de la baignoire avec le petit canard jaune qu'on s'amuse à faire voguer sur la mousse… Mais nous tenterons d'aller un peu plus loin, sans aller jusqu'au cap Horn ! Et nous aborderons les rivages de problèmes plus réalistes, donc plus complexes.

Larguez les amarres et hissez la Grand-Voile moussaillons, nous levons l'ancre !

Du lego, pas un puzzle !

MEF permet de construire des applications modulaires dont les services rendus peuvent apparaitre ou disparaitre au fur et à mesure de son existence sans remettre en cause la totalité de l'édifice.

De ce point de vue l'application peut être vue comme une boîte de Lego : une série de briques élémentaires, s'enchaînant les unes ou autres, pouvant dépendre les unes des autres, et créant un tout cohérant mais non figé.

Tout le contraire d'un puzzle ! Et pourtant c'est dans cet état que se retrouvent nombre d'applications dès qu'elles ont évolué u peu… Des tas de pièces spécifiques, s'emboîtant ici et surtout pas là, avec des trous laissés par les pièces manquantes, et dont la seule évolution possible, à termes, est de finir éparpillées sur le tapis après une bonne crise de nerf...

Figure 52 - Une application est faite de "parties"

L'ajout de fonctionnalités, le changement de certaines est, en réalité, partie prenante de la construction d'un logiciel. Mais si cela n'a pas été prévu d'emblée, on se retrouve face à un puzzle. Morcelé. Tiraillé par des pièces non adaptées rentrées en force comme un enfant face à un puzzle trop complexe. Les pièces ne sont pas interchangeables, elles ont été conçues pour un emplacement, une utilisation unique et spécifique. Il n'y a plus de maintenance possible sans casser l'ensemble, sans remettre en cause l'allure général du puzzle.

En adoptant MEF dès la création d'une application, tout comme un Framework MVVM, on s'évite de se retrouver un jour face à un immense puzzle mal agencé, un peu gondolé par les pièces entrées en force aux mauvais endroits…

MEF ne s'intègre pas « plus tard » ou « si… » mais tout de suite. Même s'il reste possible de « MEFiser » une application existante, cela n'est valable que dans les belles démonstrations pour les PDC ou sur Channel 9. Dans la vraie vie, s'y prendre trop tard… c'est trop tard !

Pensez Lego et non puzzle, utilisez MEF systématiquement…

MEF et MVVM

Ces deux-là n'ont pas été conçus pour fonctionner ensemble. Cela ne signifie pas qu'ils sont incompatibles mais seulement que rien n'a été prévu d'un côté comme de l'autre pour faciliter le mariage.

On peut même sentir une certaine confusion au départ entre ces deux approches pourtant différentes car leurs avantages sont présentés presque de la même manière : meilleure maintenabilité, couplage faible entre les modules, etc… Autant d'arguments qui sont ceux de MVVM comme de MEF.

Y a-t-il compétition entre les deux approches ?

Non. La maintenabilité de MVVM provient de sa séparation entre code et interface. Celle de MEF s'applique entre tous les modules d'une application. Le couplage faible de MVVM s'applique là aussi entre le code et l'UI, entre la Vue et le Modèle de Vue principalement. Le couplage faible de MEF est plus global et concerne tous les modules d'une application. MVVM met en place une communication par message pour dialoguer entre modules. MEF créée un lien fort mais virtuel par le jeu des importations et des exportations qui peuvent traverser les XAP.

Si les deux techniques tentent d'aboutir à des résultats proches, elles ne se situent pas au même niveau d'implémentation et elles empreintes des chemins si différents qu'au bout du compte elles en deviennent complémentaires et que leurs bénéfices s'ajoutent au lieu de se superposer.

Reste toutefois à bien comprendre comment mixer ces deux approches dans une même application pour en tirer profit. Ce que nous verrons plus loin.

Principes de base : Exportation, Importation, Composition

Les principes de MEF sont très simples :

- *Exporter* consiste à marquer un code par un attribut spécifique. Dès lors ce code pourra être vu comme l'une des briques de l'ensemble.
- *Importer* consiste, par un marquage de même type, à indiquer quels variables sont en demande d'informations en provenance des briques exportées.
- *Composer* est une opération qui demande à MEF, une fois le catalogue des briques et des demandeurs connu, de faire la jonction entre eux, de satisfaire les demandeurs en leur fournissant les données des briques exportées.

Tout cela peut avoir lieu au sein d'un même XAP, entre diverses DLL le composant, ou bien, entre le XAP « maître » et des XAP satellites.

MEF fonctionne aussi avec WPF et Windows Phone 7.

Le gain le plus grand pour une application Silverlight est bien entendu la séparation en plusieurs XAP qui ne seront téléchargés qu'en fonction des besoins de l'utilisateur.

L'exportation

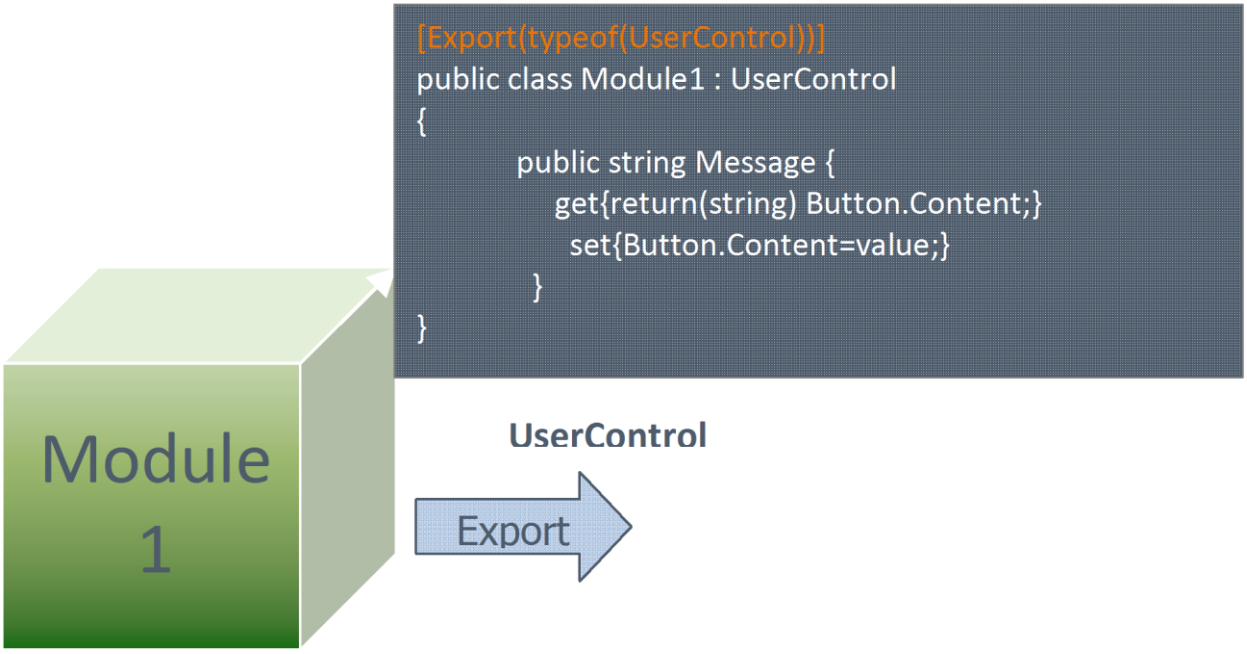


Figure 53 - Le principe d'exportation

Le module (ou partie) qu'on souhaite « publier » pour le rendre disponible à l'application est un code « normal », ici un `UserControl`. L'exportation s'effectue très facilement en ajoutant un attribut `[Export]` directement sur la classe. Il existe des variantes de cet attribut permettant de fixer des conditions plus subtiles que le seul nom de la classe (le type) ce qui est le comportement par défaut.

On remarque que le module de l'exemple ci-dessus possède une propriété publique « `Message` » tout à fait standard. En réalité, en dehors l'attribut d'exportation, le code du module ne change pas, on écrit les choses comme s'il n'y avait pas MEF. C'est l'un de ses grands avantages, MEF est très peu intrusif.

Mais cela n'interdit pas à notre module d'être lui-même consommateur d'autres parties…

L'importation

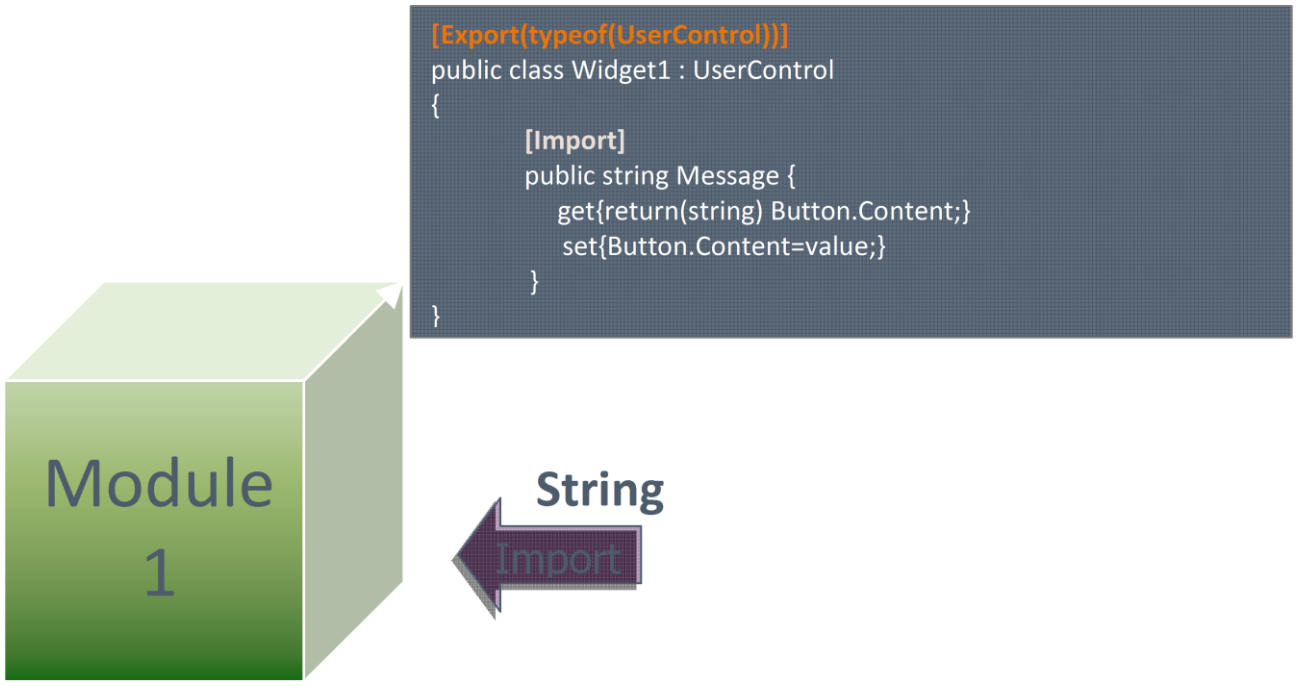


Figure 54 - le principe d'importation

Dans cet exemple, la propriété « `Message` », qui est parfaitement « normale » au départ, se trouve décorée d'un simple attribut `[Import]`. Cela signifie que la valeur initiale du message sera délivrée par une autre partie du logiciel.

Notre module est ainsi une partie exportée, disponible pour le grand jeu de Lego, mais il est lui-même consommateur d'autres parties. Il existe de fait une dépendance avec un autre module, mais ce dernier est inconnu. On sait seulement qu'il fournira une `String` pour « nourrir » la propriété. L'appareillement entre les « exportateurs » et les « importateurs » est effectué par MEF, sachant, comme on vient de le voir, qu'un module peut appartenir aux deux catégories à la fois.

Le découplage entre les parties est extrêmement fort (totale méconnaissance des autres parties) alors même que les interactions entre elles peuvent être nombreuses.

Sans MEF ces codes seraient interdépendants, auraient des références croisées les uns vers les autres et seraient impossibles à réutiliser ailleurs ou dans d'autres conditions sans casser ces dépendances. Avec MEF on conserve la souplesse (et la nécessité fonctionnelle) des dépendances mais celles-ci sont résolues « ailleurs », « autrement » sans créer de liens forts entre les parties.

Pour l'instant on constate que l'importation n'indique rien de plus que l'attribut Export. C'est-à-dire que les mécanismes par défaut de MEF seront utilisés pour gérer l'appareillement. Ce comportement par défaut repose sur le type des objets.

Dans le cas du module lui-même, l'exportation s'effectue via le type `UserControl`. Ce qui est très générique aussi mais qui peut à la rigueur être suffisant : chaque DLL ou XAP peut être construit pour ne proposer que des modules sous la forme de `UserControl`. Pas de risque qu'un `UserControl` construit pour l'interface par exemple puisse se trouver parmi les modules : un tel composant ne sera bien évidemment pas marqué par l'attribut Export…

En revanche, pour la propriété de type `string`, les choses sont différentes. Des chaînes de caractères une application peut en exporter des dizaines. En tout cas si elle choisit d'utiliser ce mécanisme pour initialiser certaines chaînes il y a fort à parier qu'il y en ait plus d'une. L'attribut `Import` utilisé sur la propriété n'est alors vraiment pas suffisant pour permettre à MEF de savoir de quelle chaîne en particulier il est question.

Pour ce faire, l'attribut d'importation supporte qu'on lui passe le nom d'un contrat. Ce contrat est un nom, une chaîne de caractères et sera réutilisé par le module qui exporte la valeur. Le code de l'attribut devient alors :

```
[Import("Accueil.Message")]
      public string Message {…
```

Ainsi marquée, la propriété `Message` ne sera « nourrie » que par une exportation réutilisant le même nom de contrat.

La composition

C'est l'étape cruciale dans le processus MEF. Après avoir collecté les exportateurs et les importateurs, MEF doit les connecter, vérifier les dépendances, charger les modules annexes, etc…

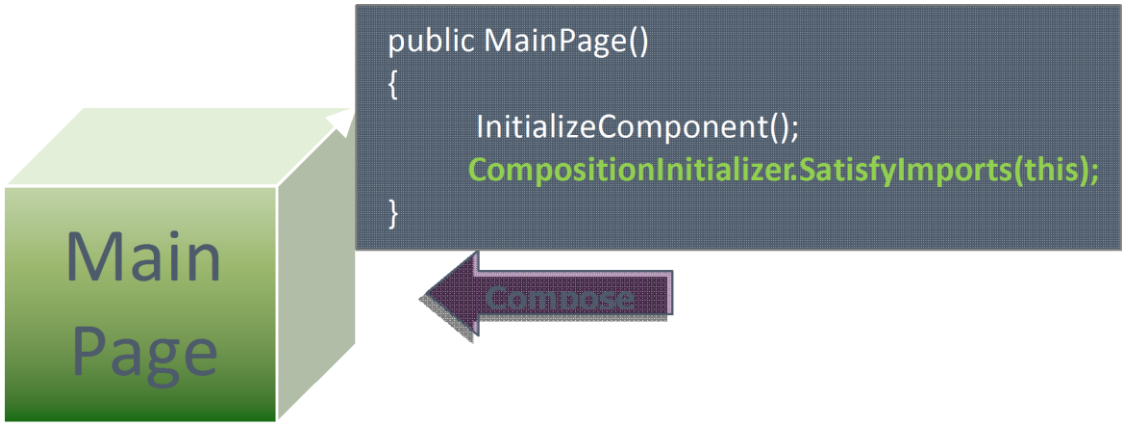


Figure 55 - le principe de composition

En général, mais nous verrons que cela n'est pas systématique, c'est à la page principale de l'application, le « *shell* » (la coquille qui gère tous les modules), que revient la charge de lancer la composition de MEF. L'appel, comme on le voit sur l'extrait de code ci-dessus, est explicite « `SatisfyImports(this)` », c'est-à-dire « satisfais les importations », le paramètre « `this` » prenant comme base l'instance courante : c'est un peu le début du fil d'Ariane que MEF remontera pour satisfaire toutes les importations et uniquement celles qui ont un intérêt pour l'instance passée en paramètre.

L'esprit de MEF

Il est essentiel de comprendre l'esprit de MEF et ce que son utilisation entraîne. **MEF n'est pas qu'une sorte de boite à outils permettant de gérer des plugins**. Cela serait terriblement réducteur.

> *MEF est un framework d'**injection de dépendances** (qui est une forme d'**inversion de contrôle**).*

L'injection de dépendances consiste à éviter le couplage fort qui existe entre des classes qui s'utilisent les unes les autres.

Les modules affichables d'une application sont de ce type. Mais l'analogie avec la gestion de plugin fait perdre toute l'altitude nécessaire qu'il faut prendre pour voir que l'injection de dépendances va bien plus loin.

Elle impose un style, une sorte de pattern comme peut l'être MVVM.

L'application doit être conçue de façon totalement modulaire, pour les affichages, éventuellement, la découverte de plugins, pourquoi pas, mais **surtout au niveau de son architecture** interne par la mise en place notamment de classes de « services » qui exportent ces derniers sans savoir quelles autres classes vont les utiliser.

La modularité ne concerne donc pas seulement les pages ou les widgets affichés, elle est bien plus profonde et plus essentielle, elle touche à l'architecture même de l'application.

Tout ce qui est visuel est facile à démontrer selon le principe bien connu qu'un dessin vaut mille mots. De fait, toutes les démonstrations, et mes exemples n'échappent pas à la règle, utilisent plutôt des effets visibles (comme l'affichage de widgets) que de sombres calculs modularisés dans des classes de services…

Encore une fois, le travail de celui qui explique, par le choix des analogies qu'il fait, par le choix des moyens utilisés a tendance à « briser » la créativité du lecteur (ou de celui qui est formé). Engageant ce dernier sur des voies « évidentes », qui, tels les charriots romains ayant laissé des ornières dans la pierre des voies, force inconsciemment ou involontairement à suivre ensuite ces voies tracées lors de l'apprentissage. Montrer MEF par l'utilisation de widgets créée de telles ornières d'autant plus dangereuses qu'elles sont invisibles… C'est pour cela qu'ici je souhaite insister sur le véritable esprit de MEF, sa véritable finalité et toute la puissance qu'il libère. MEF n'est pas une gestion de plugin. MEF est un framework d'injection de dépendances qui force à penser la modularisation de toute l'architecture d'un logiciel.

J'espère, par cet avertissement, avoir corrigé, au moins un peu, l'effet pervers de ces voies invisibles que vont être les démonstrations à venir, vous évitant de les suivre aveuglément sans avoir même conscience de la beauté de tous les autres chemins qui s'offrent à vous…

Les métadonnées

Ce que nous venons de voir de MEF est l'essentiel. Le strict minimum pour faire fonctionner la librairie à l'intérieur d'un même XAP avec des importations très simples.

On conçoit aisément que dans la réalité il soit nécessaire d'échanger plus d'informations pour gérer les modules importés.

Par exemple, dans une application de type *dashboard* (tableau de bord), certains modules doivent peut-être, par vocation, être plutôt affichés en haut ou en bas de l'écran, à droite, au centre … Ce n'est bien entendu qu'un exemple d'information qu'un module peut avoir à communiquer sur lui-même pour en faciliter son utilisation. Dans un autre contexte d'autres informations pertinentes du même genre peuvent être nécessaires pour gérer convenablement chaque module.

MEF supporte la notion de métadonnées.

Exportation de métadonnées

Les métadonnées se précisent sous la forme d'un attribut qu'on ajoute à celui de l'exportation.

Pour suivre notre exemple de position d'affichage (qui, je le redis, n'est qu'un exemple et n'est pas forcément d'une pertinence absolue), le module pourra s'exporter de la façon suivante :

```
[ExportMetadata("Location",Location.Top)]
[Export(typeof(UserControl))]
public class Module1 : UserControl
```

Ici, "`Location`" est une énumération déclarée dans le code et autorisant, parmi d'autres, la valeur « `Top` ».

Importation de métadonnées

Exporter des métadonnées est une chose, encore faut-il pouvoir les récupérer !

C'est au moment de l'importation qu'on utilisera le type `Lazy<T>` qui autorise une syntaxe plus complète permettant d'indiquer le type des métadonnées :

```
[Export(typeof(UserControl))]
public class MainPage: UserControl
{
      [ImportMany(typeof(UserControl))]
      public IEnumerable<Lazy<UserControl, IWidgetMetadata>
      {    get;set;    }
}
```

Le code ci-dessus montre l'importation de multiples modules via l'attribut `ImportMany`, la propriété d'importation est ainsi en toute logique une liste, ou plutôt

un `IEnumerable`. C'est dans cette déclaration qu'on utilise le type `Lazy<T>` auquel on passe le type du module attendu, ce type étant complété d'une interface définissant les métadonnées.

Les métadonnées personnalisées

Plutôt que la déclaration d'exportation que nous avons vue plus haut (un attribut `ExportMetaData` suivi d'un attribut `Export`) il est plus pratique dans de nombreux projets de créer son propre attribut d'exportation.

L'écriture est simplifiée et l'ensemble des informations importantes est fixé dans un attribut totalement dédié.

Par exemple, le code suivant :

```
[ExportMetadata("Location",Location.Top)]
[Export(typeof(UserControl))]
public class Module1 : UserControl
```

Pourra se résumer à :

```
[ModuleExport(Location = Location.Top)]
public class Module1 : UserControl
```

Ici, `ModuleExport` est un attribut créé par héritage de l'attribut `Export`, il intègre directement les métadonnées spécifiques à cette exportation précise.

Contrôler les exportations via des attributs personnalisés n'est pas une obligation, mais, comme on le voit sur cet exemple, cela rend le code globalement plus clair, les intentions étant visibles. Alors qu'une fausse manipulation, une erreur de copier/coller peut faire « sauter » la ligne de métadonnées de la première formulation, l'attribut personnalisé ne peut être partiel. Il existe ou est oublié, mais cela se voit, et les métadonnées ont une valeur spécifique ou bien leurs valeurs par défaut. Au-delà de la stylistique c'est bien une robustesse accrue qu'offrent les attributs personnalisés.

Point intermédiaire

Nous venons de voir les grands principes de MEF. Le décor est planté mais cela reste malgré tout théorique et ne couvre pas toutes les facettes de MEF. Difficile de se lancer avec seulement ce que je viens de dire.

L'essentiel ici est d'avoir bien compris le principe général de MEF et comment il se met en œuvre dans une application.

La section suivante, par des exemples de code, va nous permettre de préciser les choses et de voir fonctionner MEF.

Comme il ne s'agit pas de reprendre in extenso la documentation de MEF, je vais tenter de balayer les principales utilisations qui réclament un peu de savoir-faire. Comme par exemple la gestion des catalogues dynamiques sous Silverlight, le mariage MEF et MVVM Light, etc…

Pour information la documentation complète de MEF se trouve ici :

http://mef.codeplex.com/wikipage?title=Guide&referringTitle=Home

Exemples pratiques

Planter le décor est nécessaire pour savoir de quoi on parle, mais vient un moment où les principes généraux et la documentation doivent céder la place à l'action, à du code qui montre comment faire.

Il reste beaucoup de facettes de MEF à présenter, et c'est au travers d'exemples concrets que nous allons progresser.

Le Hello World de MEF

Tradition oblige, voici un « hello world ! » avec MEF.

Le contexte est simple : un seul XAP, pas de MVVM. L'application n'est pas hébergée dans une application Web.

Juste un bouton qui affiche un message, ce message étant fourni par un module.

Les namespaces

Pour utiliser MEF dans une application Silverlight nous avons besoin d'ajouter deux références :

- `System.ComponentModel.Composition`
- `System.ComponentModel.Composition.Initialization`

Il faut aussi ajouter les `using` correspondants dans les unités de code concernées, bien entendu.

La seconde unité n'est utilisée que par le code qui initialisera MEF. La première est utilisée systématiquement.

Le visuel

L'application n'affiche qu'un bouton invitant à le cliquer, c'est sobre, c'est beau, c'est limpide, ça donne ça :

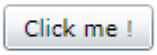


Figure 56 - Hello MEF !

La pureté du Design ne doit pas vous égarer, je sais, on resterait des heures à regarder une mise en page aussi parfaite… Mais il est temps de cliquer sur le bouton pour obtenir ça :

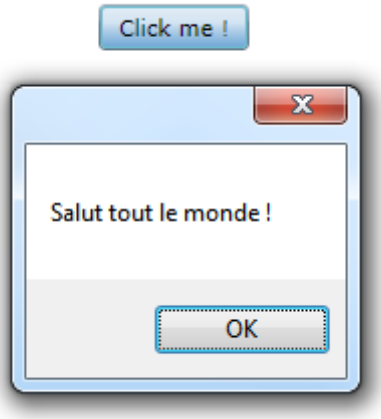


Figure 57 - Hello MEF (clic)

Je sais, au début ça fait un choc. Mais passé l'émotion bien naturelle, regardons le code si vous le voulez bien.

Le code

Premièrement j'ai créé un projet Silverlight (4.0 mais cela marche avec les versions suivantes de la même façon), sans le faire héberger dans un projet Web. Juste le minimum.

```
Dans la MainPage par défaut créée dans le projet par VS, j'ai juste ajouté,
côté XAML, un simple bouton :
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
<Button Content="Click me !"
   Height="23" HorizontalAlignment="Center" Name="btnHello"
   VerticalAlignment="Center" Width="75" Click="btnHello_Click" />
    </Grid>
</UserControl>
```

Il y en a plus long de déclarations préliminaires que de code utile puisque ce dernier se résume à une `Grid`, le `LayoutRoot` par défaut, et une balise `Button`.

Côté C# nous trouvons dans la `MainPage` le gestionnaire annoncé par le code XAML :

```
private void btnHello_Click(object sender, RoutedEventArgs e)
{ MessageBox.Show(Message); }
```

On comprend immédiatement au regard de la complexité de ce code pourquoi je suis MVP ☺

Ce code affiche le contenu de la propriété `Message` qui elle est définie de la sorte :

```
[Import("HelloWord.Message")]
public string Message { get; set; }
```

Enfin, MEF pointe le bout de son nez ! Je passe sur la déclaration de la propriété pour atteindre l'attribut : `Import`. Passé en paramètre : le nom du contrat. Ce nom est arbitraire mais devra correspondre à une exportation utilisant le même nom sur un type identique (`string`).

C'est tout pour la `MainPage`. Enfin presque. Mais progressons dans l'ordre. Nous venons de voir le code de la page principale de l'application, le « *shell* » en quelque sorte, celui qui consomme des modules. Ici le module est une instance unique d'une classe très simple (un `string`).

Regardons d'où vient ce fameux message.

```
namespace SilverlightApplication1
{
    public class MessagePart1
    {
        [Export("HelloWord.Message")]
        public string Hello = "Salut tout le monde !";
    }
}
```

La classe `MessagePart1` est une classe comme les autres, un peu vide certes, mais « normale ». Sans aucun signe ostentatoire pouvant choquer ou dérouter le développeur standard dans ses convictions intimes ou techniques.

Elle expose une chaîne de caractères (ce qui n'est pas très orthodoxe je le concède). Cette chaîne est décorée par l'attribut `Export` qui reprend le nom du contrat utilisé dans la `MainPage`.

On remarque ainsi que le nom de la chaîne elle-même qui est exportée n'a aucune importance puisque le contrat est nommé et que c'est ce nom qui conditionnera le travail de MEF.

C'est tout ?

Oui. Enfin il reste une chose à faire pour que cela fonctionne. Devinez-vous quoi ?

Dans la `MainPage` nous avons importé un « module » (rudimentaire, une chaîne, mais quelle que soit la classe le principe est le même), nous avons ensuite créé une classe exportant le « module » (la propriété `Hello` de `MessagePart1`).

Il manque la *glue*. L'action qui va coller les morceaux ensemble pour que cela fonctionne : la phase de **composition**.

Revenons au code de la `MainPage` et regardons son constructeur :

```
public MainPage()
        {
            InitializeComponent();
            CompositionInitializer.SatisfyImports(this);
        }
```

La seconde ligne effectue ce travail de composition.

C'est vraiment tout ?

Cette fois-ci oui.

N'est-ce pas déconcertant de simplicité ?

Alors allons un cran plus loin avec l'exemple suivant.

Gestion de plusieurs modules et métadonnées

Nous allons accélérer le pas et voir d'une part comment utiliser MEF pour gérer des « vrais » modules (dans le style d'une gestion de plugins) tout en étudiant au passage les métadonnées.

Pour l'instant les modules resteront rudimentaires, c'est le principe qui compte, le tout toujours limité à des classes faisant partie du même XAP.

Nous allons voir que les métadonnées peuvent s'utiliser soit directement, soit de façon fortement typées, soit par le biais d'un attribut personnalisé.

Commençons par le plus simple…

Le visuel

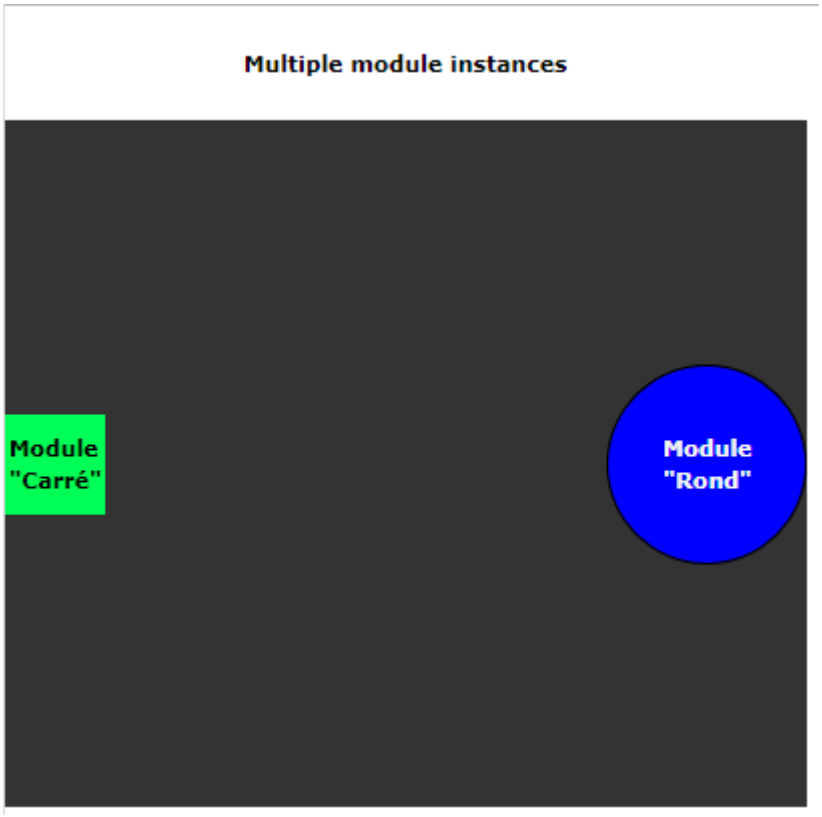


Figure 58 - Modules multiples

Je sais, le visuel de cet exemple est, bien plus encore que le précédent, un sommet artistique que seul un Designer confirmé pouvait créer. Un fond gris, un titre, un rond bleu et un carré vert, Miro et Picasso en seraient … verts de jalousie (ou bleus de peur !). Au moins vous ne serez pas perturbé par une débauche d'effets spéciaux !

Le principe de fonctionnement

Nous simulons ici un *dashboard*, c'est-à-dire un tableau de bord, un style d'application très répandu (gestion de portefeuille boursier, surveillance de processus industriels…). Nous disposons ainsi d'un *shell* (la `MainPage`) offrant une surface d'affichage (un `DockPanel` ici) pour les différents modules.

Chaque module propose la visualisation (et éventuellement la saisie) de données différentes. Notre application propose pour l'instant deux modules. Le premier est matérialisé par un carré vert de 50x50, identifié comme tel par un texte, et un rond bleu de 100x100, identifié identiquement. Ces modules d'exemple ne font absolument rien, mais ils le font bien.

Comme la figure ci-dessus le laisse voir, le module Carré Vert est *docké* à gauche, le module Rond Bleu étant *docké* à droite.

S'agissant d'une application modulaire utilisant MEF, vous avez compris que le Carré Vert et le Rond Bleu sont deux modules exportés et reconnus automatiquement par le *shell*. Nous allons voir comment.

Vous remarquerez que les modules sont *dockés* selon un ordre précis. Ce sera l'affaire des métadonnées de fixer ce genre de chose. Nous le verrons plus loin (tout en rappelant que ce n'est qu'un exemple et que fixer la position d'affichage dans un module n'est pas une excellente idée, sauf cas particulier).

Enfin, une chose que vous ne pouvez pas voir directement, chaque module est un `UserControl` différent.

Nous avons ici le schéma classique d'une application *dashboard*. Seuls les modules sont simplifiés, mais en reprenant le code de l'exemple et en complétant les `UserControl` nous obtiendrions une application réelle. Il n'y a donc de simplification que dans le contenu des modules, pas dans le fonctionnement de l'application ni dans l'utilisation de MEF.

Le lecteur notera que j'ai choisi d'utiliser un `DockPanel` pour servir de conteneur, cela est parfaitement arbitraire, l'application fonctionnerait de la même façon avec une `Grid`, un `Canvas` ou un `WrapPanel` par exemple. La seule différence concernerait les métadonnées qui dans l'exemple retourne une indication de *dockage* propre au `DockPanel`, il faudrait adapter cette information de positionnement pour un autre type de conteneur. Les modules pourraient aussi avoir des positions attribuées par le *shell*, ce qui semble plus plausible dans la réalité. La position d'affichage n'est qu'un exemple de métadonnées. Dans une application réelle les métadonnées peuvent retourner toutes sortes d'informations : nom du module, GUID, liste d'objets, etc… C'est vous qui décidez du contenu des métadonnées qui, de façon interne, sont représentées par un dictionnaire d'objets (donc une clé au format `string`, et une valeur de type `object`).

Le Code

Passons à l'essentiel : le code.

Les principes de base démontrés dans l'exemple précédent restent parfaitement valides :

- Exportation
- Importation
- Composition

La façon de réaliser chacune de ces étapes comporte juste quelques variations.

La MainPage

Son code XAML est très simple, comme je le disais plus haut : un `DockPanel` dans une `Grid` et un `TextBlock` pour afficher le titre :

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:toolkit=http://schemas.microsoft.com/winfx/2006/xaml/presentation/too
lkit
    x:Class="SilverlightApplication1.MainPage"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400" Width="400" Height="400">

    <Grid x:Name="LayoutRoot" Background="White">
      <Grid.RowDefinitions>
            <RowDefinition Height="0.143*"/>
            <RowDefinition Height="0.857*"/>
      </Grid.RowDefinitions>

      <toolkit:DockPanel x:Name="MainDock" LastChildFill="False"
                                      Grid.Row="1"
Background="#FF333333"/>
      <TextBlock TextWrapping="Wrap" d:LayoutOverrides="Width, Height"
                  HorizontalAlignment="Center" VerticalAlignment="Center"
                  FontWeight="Bold">
                  <Run Text="Multiple"/><Run Text=" module
instances"/></TextBlock>

    </Grid>
</UserControl>
```

La surface d'affichage des modules sera le `DockPanel` qui se nomme `MainDock`.

Le code C# de MainPage

Comme dans l'exemple précédent, la `MainPage` est le *shell*, le réceptacle qui gère les modules. Il doit ainsi dans son constructeur appeler le moteur de composition de MEF puis il doit agir sur les modules pour les afficher (il pourrait les traiter d'une autre façon, ou plus tard). Pour cela il doit proposer une propriété publique qui recevra la liste des modules. La seule différence avec le premier exemple est que ce dernier gérait une importation d'instance unique (une chaîne de caractères) alors que le présent exemple va gérer une collection d'objets importés.

```
namespace SilverlightApplication1
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
            CompositionInitializer.SatisfyImports(this);
            displayModules();

        }

        [ImportMany]
        public Lazy<UserControl,IDictionary<string, object>>[] Modules {
get; set; }

        private void displayModules()
        {
            foreach (var control in Modules)
            {
                var dock = (control.Metadata.ContainsKey("Location"))
                                    ? (Dock) control.Metadata["Location"]
                                    : Dock.Bottom;

                control.Value.SetValue(DockPanel.DockProperty,dock);
               MainDock.Children.Add(control.Value);
            }
        }
    }
}
```

Le constructeur ne présente aucune différence avec l'exemple précédent en dehors du fait qu'il appelle une méthode `displayModules` dont le rôle sera d'afficher chaque module.

La première différence notable se trouve dans la propriété importée. La propriété `Modules` est d'abord décorée avec l'attribut `ImportMany` au lieu de `Import`.

La signification semble évidente : `ImportMany` importe « *many* », plusieurs, modules, là où `Import` n'importe qu'une seule instance.

Il est donc naturel que la propriété décorée, `Modules`, soit un simple `IEnumerable<T>`. Cela serait en fait le cas si nous ne gérions pas de métadonnées. Mais comme nous souhaitons récupérer de telles métadonnées le type de `Modules` est `Lazy<T,M>`. Où le type `T` représente le type des objets importés (ici `UserControl`) et `M` le type des métadonnées (un dictionnaire générique `string` / `object` comme indiqué plus haut). Ici, `Modules` est déclarée comme étant une `Array` d'objets `Lazy<T,M>`.

Mais il s'agit là d'une simple tournure, il serait possible de déclarer Modules de cette façon :

```
public IEnumerable<Lazy<UserControl,IDictionary<string, object>>> Modules
{get; set;}
```

On retrouve alors `IEnumerable<>`. Cela ne fait aucune différence en réalité, c'est juste un choix stylistique.

La méthode `displayModules()` est plus intéressante puisqu'elle va nous permettre de récupérer les fameux modules et leurs métadonnées qui permettront de les positionner dans la surface d'affichage (le `DockPanel`).

`displayModules()` ne contient qu'une boucle `foreach` sur la propriété `Modules`. Elle va ainsi énumérer l'un après l'autre tous les modules qui auront été découverts par MEF.

La boucle se divise en trois étapes :

- Récupération des métadonnées pour obtenir l'information de placement du module,
- Modification de l'emplacement du `UserControl` en conséquence,
- Ajout du module à la surface d'affichage.

Les éléments de `Modules` ne sont pas réellement des `UserControl`, ce sont des `Lazy<T,M>`, rappelez-vous. De fait cette classe encapsule chaque `UserControl` importé qui est disponible au travers de sa propriété `Value`. `Lazy<T,M>` offre une autre propriété au sens immédiat : `MetaData`.

Grâce à cette dernière nous pouvons récupérer l'information de position que chaque module a exporté (nous allons voir bientôt comment).

Une fois l'emplacement connu, et la propriété `Dock` du `UserControl` fixée, ce dernier est ajouté à l'arbre visuel du `DockPanel`.

Le code des modules

Nous n'étudierons que le code d'un seul module sachant que le second est fait exactement pareil (en dehors de sa forme) et que vous disposez du code source avec l'article.

Au hasard… ce sera le Rond Bleu.

Tout d'abord il s'agit d'un `UserControl`. Dans un autre type d'application cela pourrait être n'importe quoi, pas forcément quelque chose qui s'affiche. On peut très bien modulariser du pur code (des méthodes de calcul, des effets visuels ou sonores, des sons, des collections d'images, du code envoyant des emails par différentes méthodes, …). Une application *navigationnelle* pourra, par exemple, exporter des instances de `Page`, offrant ainsi à l'utilisateur un site composé selon ses besoins ou sa licence.

Bref, ici c'est un `UserControl`. Pour le Rond Bleu sa taille est fixée à 100x100 pixels avec un `TextBlock` centré indiquant le nom du module. Tout cela est très simple et se reproduit en quelques secondes. Ce n'est pas du côté de XAML qu'il faut chercher les astuces de cet article.

Le code C# du contrôle est le suivant :

```
namespace SilverlightApplication1
{
    [Export(typeof(UserControl))]
    [ExportMetadata("Location", Dock.Right)]
    public partial class RoundModule : UserControl
      {
            public RoundModule()
            {
                  // Required to initialize variables
                  InitializeComponent();
            }
      }
}
```

Le `UserControl` est construit « normalement ». Une fois encore, MEF n'impose rien au niveau des objets exportés. Si c'est un descendant de `Page` ou de `UserControl`, on le programme sans se soucier de MEF, idem s'il s'agit d'une classe de service.

MEF ne se fait voir qu'en décorant la classe `RoundModule` (le `UserControl`).

Tout d'abord il y a l'attribut d'exportation. Ici nous devons préciser le type. Un objet peut apparaître sous sa propre classe (en général sa classe mère si on gère plusieurs instances comme dans cet exemple) ou bien sous la forme d'une *Interface*. Cette dernière option est largement préférable dans la réalité. Travailler par contrat est un découplage de plus, et une sécurité intéressante (pas d'accès aux champs non pris en compte par l'Interface) et une garantie de maintenabilité plus grande.

Ainsi, l'attribut `Export` utilise ici le type `UserControl` mais pourrait très bien préciser le type d'une interface particulière dans le cas de plugins réels.

La vraie différence avec l'exemple précédent se situe dans le second attribut `ExportMetaData`. Il ne s'agit pas d'une obligation, comme je le disais au début de cet exemple, j'ai choisi d'ajouter la gestion des métadonnées pour éviter de multiplier de trop le nombre des projets. Celui que nous étudions en ce moment montre deux choses distinctes : la gestion des instances multiples et celle des métadonnées.

Sans les métadonnées nous ne pourrions pas récupérer l'information de placement utilisée par le `DockPanel`, ce serait au *shell* de décider où placer les modules. De même nous n'utiliserions pas `Lazy<T,M>` pour déclarer l'importation mais simplement

un `IEnumerable<UserControl>`. Enfin, nous n'aurions pas, naturellement, d'attribut `ExportMetaData` dans les modules…

Mais nous en avons un ici. En fait nous pourrions en avoir autant que nous voulons, un pour chaque métadonnée exportée. Dans cet exemple nous n'exportons qu'une seule information, le *docking*. Ainsi nous ne trouvons qu'un seul attribut `ExportMetaData`.

Il prend en argument deux valeurs. La première est la clé. C'est-à-dire le nom de la métadonnée (ici `Location`, *position* en anglais). La seconde est la valeur (ici `Dock.Right`, dans le second module la valeur est `Dock.Left`, c'est la seule chose qui change en dehors de la couleur et de la forme).

Rappelez-vous que les métadonnées sont gérées par un `Dictionary<string,object>`. On comprend mieux la raison d'être des deux paramètres, ce sont les deux paramètres de la méthode `Add()` d'un dictionnaire générique de ce type.

Point Intermédiaire

Cet exemple est plus proche de la réalité tout en étant une abstraction qui ne doit pas vous faire perdre de vue que presque tous les choix effectués sont purement arbitraires. Le danger d'un exemple est d'orienter inconsciemment le lecteur, de bloquer sa créativité et son imagination en lui soumettant des voies toutes tracées.

Ici, il faut se rappeler que la gestion des métadonnées n'est pas une obligation et n'a rien à voir avec celle des instances multiples de modules. Il faut avoir conscience que les modules sont des `UserControl` uniquement parce que cela permet d'avoir un résultat visuel facilitant la compréhension mais qu'il pourrait s'agir de code pur n'affichant rien. Il faut se méfier aussi de l'utilisation directe qui est faite des modules dans le constructeur du *shell*, on pourrait très bien utiliser les modules à un autre moment sans les afficher tout de suite. Enfin, concernant la gestion des métadonnées il faut se souvenir qu'il s'agit d'un simple dictionnaire et qu'on peut y mettre ce qu'on veut. Le choix d'indiquer une option de positionnement à l'écran permet de voir (au sens visuel) le résultat ce qui est pratique pour une démonstration, dans une application réelle le positionnement des modules serait plutôt géré par le *shell* et les métadonnées reflèteraient d'autres informations plus pertinentes pour le fonctionnement de l'ensemble (nom du module pour l'afficher dans un menu par exemple, icône de la fonction représentée…).

Les métadonnées sont d'une grande importance dans beaucoup de situations réelles. Nous avons vu comment les gérer de façon simple. Cela fonctionne efficacement

mais ces données ne sont pas typées. Dans une gestion de modules réelle il est préférable de typer ces informations.

Il existe deux façons de typer les métadonnées :

- En déclarant une interface
- En utilisant un attribut d'exportation personnalisé.

Voyons maintenant chacune de ces options.

Métadonnées fortement typées

Dans l'exemple précédent nous avons mis en évidence l'utilisation des métadonnées. Notre exemple n'utilisait qu'une seule métadonnée (le *docking*) l'exploitant au travers du dictionnaire fourni par défaut.

Cette approche est directe et ne réclame pas de code supplémentaire. Toutefois dans un logiciel bien conçu laisser des zones de « non typage » présente toujours un risque.

Il serait bien plus judicieux d'utiliser des métadonnées fortement typées plutôt qu'un simple dictionnaire d'objets.

MEF supporte cette possibilité avec très peu de code. Il suffit de créer une interface regroupant les métadonnées sous la forme propriétés en lecture seule et d'utiliser cette interface dans la déclaration de l'importation. Lors de l'exploitation du champ `MetaData` de l'objet `Lazy<T,M>` (un module retourné par MEF) cette propriété apparaitra alors comme étant du type de l'interface souhaitée. L'accès aux valeurs est alors plus direct (par le nom des propriétés de l'interface) et totalement « *safe* » du point de vue du typage.

Pour illustrer cette façon de faire, je vous propose de reprendre l'exemple précédent en le modifiant de la façon suivante :

- Ajout d'une seconde métadonnée (la couleur de fond de l'objet)
- Création d'une interface contenant les deux propriétés (*docking* et couleur)
- Modification de l'importation (pour spécifier l'interface)
- Modification de la méthode `displayModules()` (utilisation de l'interface pour fixer *docking* et couleur de chaque module affiché).

Ces modifications sont légères mais laissent apparaitre deux difficultés qui n'ont rien à voir avec MEF. J'ai hésité avant de les traiter ici, c'est hors sujet, mais d'un autre

côté le code source seul ne permettrait pas forcément de comprendre pourquoi j'ai fait tel ou choix d'implémentation.

Le lecteur m'excusera donc la petite digression qui va suivre et qui ne concerne pas MEF mais qui, j'en suis convaincu, l'intéressera tout autant.

Digression

Vous êtes prévenu, si seul MEF, rien que MEF vous intéresse vous pouvez sauter ce passage…

Binder le Background d'un UserControl à l'un de ses éléments

En réalité le problème est plus vaste et concerne toutes les propriétés qu'offre un `UserControl`. Vous allez comprendre rapidement.

Prenons le « module Rond Bleu ». Puisque maintenant la couleur de l'objet peut être fixée par une métadonnée (ce qui sera détaillé plus loin) nous devons pouvoir changer la couleur `Background` du `UserControl` et que ce changement soit réellement appliqué au cercle.

Ce cercle est une `Ellipse` qui possède une propriété `Fill`.

La classe `UserControl` possède une propriété `Background`.

Ce que nous voulons c'est tout bêtement relier les deux. De telle sorte qu'une modification de la propriété Background du `UserControl` change le `Fill` de `l'Ellipse`.

C'est simple. Légitime. Cela se résout par un simple binding direz-vous.

Hélas Silverlight ne supporte pas toutes les subtilités du binding relatif de WPF. De fait il n'est pas possible d'utiliser la syntaxe complète (notamment avec `FindAncestor` sur le type `UserControl`) que nous pourrions mettre en œuvre directement sous WPF.

Certains petits futés pensent avoir trouvé la solution en exploitant *l'Element Binding*, en reliant la propriété `Fill` de `l'Ellipse` à la propriété `Background` du `UserControl`. Cela fonctionne. Mais, trois fois hélas l'astuce ne fonctionne que si la cible est nommée. Ce qui oblige à ajouter un `x:Name` au `UserControl`.

Le premier problème qui se pose est que ce nom est codé en dur dans le XAML du contrôle. Que se passe-t-il si le nom est changé par code ? Le binding sera cassé. Mais cela n'est pas le plus gênant. Cette solution interdit tout simplement d'utiliser

deux fois le même `UserControl`, la création de la seconde instance retournera une erreur (nom dupliqué).

Heureusement il reste une troisième possibilité : effectuer le binding dans le constructeur du `UserControl`, à cet endroit nous avons accès à « `this` » qui représente l'instance du `UserControl` et nous n'avons donc pas besoin d'un nom pour créer *l'Element Binding* entre la propriété de `l'Ellipse` et celle de son hôte (le `UserControl`)…

Cela explique pourquoi les constructeurs de deux modules ont été modifiés pour intégrer chacun un binding par code.

Pour le Carré vert (qui va changer de couleur donc), la couleur du fond est celui de sa `Grid LayoutRoot`, le binding est donc celui-ci :

```
LayoutRoot.SetBinding(Panel.BackgroundProperty, new Binding { Source =
this, Path = new PropertyPath("Background") });
```

La propriété `Background` de la `Grid`, qui provient de son ancêtre `Panel`, est bindée à la propriété `Background` de la source « `this` », donc de l'instance du `UserControl`.

Pour le module Rond Bleu (qui lui aussi changera de couleur) le binding est le suivant :

```
Circle.SetBinding(Shape.FillProperty, new Binding {Source = this, Path =
new PropertyPath("Background")});
```

Ici, c'est la propriété `Fill` de l'Ellipse (qui s'appelle `Circle`), propriété héritée de `Shape`, qui est bindée à la propriété `Background` de la source « `this` », donc l'instance du `UserControl`.

Cette approche est à demi satisfaisante mais elle permet d'éviter l'incohérence de la quatrième solution que certains retiennent : tout simplement ajouter une nouvelle propriété de dépendance au `UserControl` et ignorer superbement la propriété `Background` originale… C'est sûr, ça marche. Mais d'une part le code d'une propriété de dépendance est plus lourd que le binding montré ci-dessus, et d'autre part laisser une propriété `Background` qui ne fonctionne pas pour en créer un doublon (*`BackgroundBis`*, *`BackgroundVrai`* ? quel nom lui donner ?) m'apparait une hérésie fonctionnelle et stylistique.

Donc, j'opte pour le binding dans le constructeur. Une variante plus fiable pour des objets plus long à s'initialiser que nos petits modules consiste à placer le binding non pas dans le constructeur mais dans le `Loaded` de l'élément concerné (la `Grid` ou `l'Ellipse` de nos modules).

Bien entendu le même problème se poserait si nous voulions utiliser une autre propriété que `Background` du `UserControl`. La solution serait aussi la même.

Colors n'est pas une énumération

Satanées couleurs ! On pensait s'en être sorti avec le binding de la couleur de fond et voilà que nous tombons sur un autre problème !

Silverlight expose une classe `Colors` qui retourne des couleurs par leur nom. Cela est bien pratique.

Comme nous voulons passer la couleur de fond des modules dans leurs métadonnées, il semblerait évident d'écrire par exemple un code de ce genre :

```
[ExportMetaData("Background",Colors.Cyan)]
```

Tristement, nous sommes bien obligés de constater que le compilateur n'en veut pas… Un argument d'un attribut ne peut être qu'une constante, un `typeof` ou quelques autres possibilités, mais en aucun cas le *nom d'une propriété*.

En effet, `Colors` n'est pas une énumération… IntelliSense en nous donnant la liste des couleurs lorsqu'on tape « `Colors.` » nous fait oublier qu'ici il nous donne *la liste des propriétés statiques de la classe* `Colors`. Du coup `Colors.Cyan` est le nom d'une propriété statique et en aucun cas un item d'une énumération…

Une telle écriture est donc interdite, il va falloir trouver autre chose pour passer la couleur dans les métadonnées.

Pour faire simple nous passerons une chaîne de caractères indiquant le nom de la couleur. La version longue consisterait à déclarer une énumération de toutes les couleurs puis de convertir chaque élément dans sa bonne valeur ARGB. J'ai opté pour quelque chose de plus court ici.

C'est pourquoi vous trouverez le code suivant dans la `MainPage` :

```
private Color getThisColor(string colorString)
{
   var colorType = (typeof(Colors));
   if (colorType.GetProperty(colorString) != null)
    { var o =
              colorType.InvokeMember(colorString,
                                     BindingFlags.GetProperty,
                                     null, null, null);
      if (o != null) return (Color)o;
    }
   return Colors.Black;
 }
```

Cette séquence prend le nom d'une couleur (*case sensitive*) et via la réflexion recherche la propriété qui le porte dans `Colors` et se sert du résultat pour retourner la couleur de type `Color`.

L'avantage de ce code est qu'il tient en quelques lignes. La solution de l'énumération serait à mettre en œuvre dans une vraie application pour définir notamment toutes les couleurs qu'on retrouve dans WPF (alors que `Colors` sous Silverlight ne contient que très peu de couleurs).

Fin des digressions

Vous savez maintenant pourquoi le code source fourni avec cet article utilise un binding dans les constructeurs des modules et pourquoi le paramètre couleur est passé dans les métadonnées d'exportation sous la forme d'une chaîne de caractères…

Revenons à nos moutons !

Métadonnées via une Interface

Comme expliqué juste avant les petites digressions, nous allons procéder de la sorte :

- Ajout d'une seconde métadonnée (la couleur de fond de l'objet)
- Création d'une interface contenant les deux propriétés (*docking* et couleur)
- Modification de l'importation (pour spécifier l'interface)
- Modification de la méthode `displayModules()` (utilisation de l'interface pour fixer *docking* et couleur de chaque module affiché).

Ajout de la seconde métadonnée

L'ajout d'une seconde métadonnée est purement déclaratif dans les exports, pour chaque module nous aurons ainsi les déclarations suivantes :

Pour le Rond :

```
    [Export(typeof(UserControl))]
    [ExportMetadata("Location", Dock.Right)]
    [ExportMetadata("Background", "Purple")]
    public partial class RoundModule : UserControl
```

Pour le Carré:

```
    [Export(typeof(UserControl))]
    [ExportMetadata("Location",Dock.Left)]
    [ExportMetadata("Background", "Yellow")]
    public partial class SquareModule : UserControl
```

> *Je ne sais pas si cela provient de* `Resharper` *(excellent add-on dont je ne pourrais me passer pour coder) ou si VS sait le faire aussi par défaut, mais il se peut que la seconde déclaration de* `ExportMedata` *vous indique une alerte du type « attribut dupliqué ». Il suffit d'ignorer cet avertissement. Ici, nous multiplions les* `ExportMetadata` *par nécessité : un attribut par métadonnée.*

Création de l'interface des métadonnées

Cette interface contient toutes les métadonnées utilisées. Elle est constituée d'une suite de propriétés portant le même nom que les métadonnées et ne disposant que d'un accesseur en lecture. Voici l'interface `IModule` créée pour notre exemple :

```
namespace SilverlightApplication1.Contract
{
    public interface IModule
    {
        Dock Location { get; }
        string Background { get; }
    }
}
```

Modifier l'importation

La séquence d'importation, au lieu d'indiquer un dictionnaire générique doit maintenant préciser le nom de l'interface (cela se passe dans le code de `MainPage`) :

```
[ImportMany]
public IEnumerable<Lazy<UserControl, IModule>> Modules { get; set; }
```

Modification de la séquence d'affichage

C'est ici qu'on peut voir tout l'intérêt de l'interface et de son typage (toujours dans `MainPage`) :

```
        private void displayModules()
        {
            foreach (var control in Modules)
            {
                var dock = control.Metadata.Location;
                var colorName = control.Metadata.Background;
                var color = getThisColor(colorName);
                control.Value.SetValue(DockPanel.DockProperty, dock);
                control.Value.Background = new SolidColorBrush(color);
                MainDock.Children.Add(control.Value);
            }
        }
```

La propriété `MetaData` de l'objet module est automatiquement typée en tant que `IModule`, notre interface. Récupérer les données est donc direct et fortement typé (la chaîne est bien une `string`, le *docking* est bien représenté par une valeur de l'énumération `Dock`, tout cela sans transtypage depuis le type `object`).

Filtrage des métadonnées

L'utilisation d'une interface créée une sorte de « vue » sur les métadonnées. Et cette vue peut être considérée comme un *filtre* : seuls les modules exportés remplissant toutes les métadonnées de l'interface seront considérés comme valides et seront importés.

Il est essentiel de se rappeler de cet effet de bord. Soit pour s'en servir (mais c'est une voie douteuse que je vous incite à éviter), soit pour éviter que certains modules ne

s'exportent pas (créant un bogue qui peut être difficile à trouver si on ne pense pas à ce filtrage implicite).

Pour éviter cette situation il existe un attribut à utiliser dans la déclaration de l'interface : `DefaultValueAttribute`. Sur la base de l'interface de notre exemple voici comment celle-ci pourrait être décorée :

```
namespace SilverlightApplication1.Contract
{
    public interface IModule
    {
        Dock Location { get; }
        [DefaultValue("Red")]
        string Background { get; }
    }
}
```

Par l'ajout de cet attribut, si un module ne spécifie pas de valeur pour la métadonnée `Background`, celle-ci prendra la valeur « `Red` » et ne sera pas considérée comme manquante, de fait le module ne sera pas filtré (donc ne sera pas exclu) lors de son importation. En revanche, l'oubli de la métadonnée `Location` exclura le module (je n'ai pas placé de valeur par défaut).

Point intermédiaire

Nous avons vu comment utiliser des métadonnées. Nous venons de voir comment rendre cette utilisation plus orthodoxe en typant ces informations via une interface et comment définir des valeurs par défaut sur cette dernière.

Mais il reste quelque chose d'assez peu pratique : la multiplication des attributs `ExportMetaData`. Cela soulève quelques problèmes :

- Le code est long surtout si on utilise de nombreuses métadonnées;
- Il est facile d'oublier une ligne de métadonnée, rien ne l'indiquera ni ne nous avertira du problème ;
- L'oubli d'une métadonnée filtrera le module qui ne sera plus reconnu et donc ne sera plus exporté ; ou, si une valeur par défaut a été spécifiée, c'est cette valeur peut-être pas appropriée qui sera utilisée ;
- Plus c'est long à écrire, plus on utilise copier/coller, et plus on introduit de bogues (après le « coller » il est facile d'oublier de modifier l'une des lignes) ;

- Enfin, ce n'est pas vraiment élégant.

Ces critiques peuvent paraître accessoires comparées aux nombreux avantages que les métadonnées confèrent à MEF.

Bien que secondaires, ces considérations nous poussent à vouloir quelque chose de plus clair, plus élégant et offrant moins de prise aux bogues et oublis éventuels.

Il existe une solution. Elle consiste à créer son propre attribut d'exportation.

C'est ce qu'on nous allons étudier à la section suivante.

Attribut d'exportation personnalisé

Rappelons la progression jusqu'à maintenant :

1) Un import simple d'une chaîne (le message du premier exemple de code)
2) Importation de multiples instances avec ajout de métadonnées
3) Amélioration des métadonnées avec un typage fort via une interface.

L'étape de simplification ultime consiste donc à faire en sorte de fusionner en un seul attribut tous les attributs d'exportation utilisés : `Export` et `ExportMetaData`.

Notre code sera propre, clair, les intentions seront visibles, les accès aux métadonnées typées, et nous disposerons alors d'une base saine mimant en tout point ce qui doit être fait dans une application réelle.

La création d'un attribut d'exportation est vraiment simple une fois qu'on a franchi les étapes précédentes. Cet attribut réemploie l'interface déjà définie mais de façon implicite, via les noms des propriétés qu'il expose.

Voici le code pour mieux comprendre :

```
namespace SilverlightApplication1.Contract
{
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class ModuleAttribute : ExportAttribute
    {
        public ModuleAttribute() : base(typeof(UserControl)) { }

        public Dock Location { get; set; }
        public string Background { get; set; }

    }
}
```

Le code est vraiment court. La classe créée hérite de `ExportAttribute` et elle décorée par deux attributs : `MetadataAttribute` qui indique que cet attribut d'exportation gère aussi les métadonnées (optionnel), et `AttributeUsage` qui explique à MEF quels types d'éléments de code peuvent être décorés par le nouvel attribut. Dans ce code j'ai indiqué `AttributeTargets.Class` car les modules sur lesquels j'utiliserai l'attribut sont des classes. On pourrait autoriser l'utilisation du nouvel attribut sur des méthodes, des propriétés, etc… Le paramètre `AllowMultiple` est initialisé à `false` car tel qu'il est conçu notre nouvel attribut ne doit s'utiliser qu'une fois par classe.

Le corps de la classe se résume à publier un constructeur qui ne fait qu'appeler celui de base en passant le *type des objets décorés* puis à définir les propriétés correspondant aux métadonnées.

Le constructeur appelle celui de sa classe parent (`ExportAttribute`) en lui passant le type `UserControl`, car nos modules sont exportés sous cette forme. Rappelez-vous qu'il est possible d'exporter les modules sous la forme d'une interface commune (cas plus réaliste pour des plugins), ce serait alors le type de cette interface qui serait indiqué ici. Une erreur classique (et qui fait planter l'application) consiste à passer le type de l'interface des métadonnées au lieu du type du module exporté. Méfiez-vous !

Les propriétés portent des noms rigoureusement identiques à ceux utilisés dans l'interface de métadonnées. Le type doit lui aussi être respecté.

Notre application est désormais dotée d'une classe d'exportation personnalisée, il ne reste plus qu'à s'en servir…

Pour le Carré cela donnera :

```
[Module(Location = Dock.Left, Background = "Yellow")]
public partial class SquareModule : UserControl
```

Pour le Rond:

```
[Module(Location = Dock.Right, Background = "Purple")]
public partial class RoundModule : UserControl
```

Rien d'autre ne change. Nous avons juste rendu le code plus lisible. L'utilisation d'un attribut personnalisé permet aussi d'afficher clairement *l'intention*. Ici l'attribut s'appelle `ModuleAtrribute` (ce que C# permet de tronquer en `Module` tout court, sans le suffixe). A la seule lecture de ce code nous comprenons que l'objet marqué est un « Module » (ce qui a un sens dans notre application).

N'oubliez-pas en effet, que MEF permet d'exporter et d'importer de nombreuses choses au sein d'une même application ! Il peut exister des modules, des services (code d'exportation de données en CSV, en XML…), des messages, etc…

Si tout ce petit monde est marqué du seul attribut standard « `Export` » il sera difficile de comprendre ultérieurement pourquoi quelque chose ne marche pas (un message exporté comme un module par exemple). L'utilisation d'attributs d'exportation spécialisés évite ces confusions et facilite la maintenance du code tout en le rendant plus clair.

Point intermédiaire

Arrivé à ce point de l'article vous commencez certainement à avoir une bonne idée de ce qu'est MEF. Même si sur le fond cela est certainement vrai (ou il faut que j'arrête d'écrire des articles !) l'impression est peut-être trompeuse car il reste beaucoup de choses à dire sur MEF et son utilisation « en vrai ».

Hélas tout ne pourra pas être couvert dans cet article qui dépasse pourtant les 70 pages. N'oubliez pas la documentation de MEF !

Organiser les déclarations

Une chose qui n'a pas été dite à propos de tous ces éléments c'est comment on doit les organiser dans une application.

Et c'est un point important si on veut conserver une unité à l'ensemble et ne pas se prendre les pieds dans le tapis.

En effet, une application réelle proposera vraisemblablement bien plus de modules que les exemples des pages précédentes, bien plus d'exportations, bien plus d'importations en divers endroits.

Pour que l'application soit maintenable, il est ainsi conseillé de créer et de déployer un assemblage à part qui ne contient que les contrats (généralement des interfaces, voire des classes abstraites). Je parle ici des contrats sous lesquels les modules (ou autres) apparaissent, pas des métadonnées. Tous les modules, toutes les parties, tous les XAP éventuels formant l'application feront référence à cet assemblage.

On garantit ainsi l'unicité et l'uniformité indispensable de ces déclarations.

Cela peut sembler plus lourd, mais dans une application réelle utilisant intensivement MEF, donc de nature hautement modulaire, vous verrez vite que ce conseil est payant.

Il est aussi conseillé de définir dans un autre (ou plusieurs autres) assemblage(s) toutes les interfaces de métadonnées ainsi que les attributs d'exportation personnalisés.

Une telle organisation s'avère de plus bien adaptée au travail en équipe.

Dans la suite de l'article je vais tenter de répondre à d'autres questions pratiques sur l'utilisation de MEF. Mais faisons un point sur certains aspects abordés jusqu'ici.

Les différentes exportations

Avec MEF tout tourne autour des exportations : c'est le point de départ. Ce qui peut être exporté, et comment cela est exporté conditionne ce qui pourra être importé et comment cela pourra être exploité.

Les exemples que nous avons vus jusqu'ici portent sur l'exportation de classes visuelles (des `UserControl`), le premier exemple portait sur une propriété (un message). Pour bien comprendre les possibilités de MEF il semble important de bien saisir quelles sont les possibilités de l'exportation.

Exportation de parties composables

Une exportation de partie composable (*Composable Part level export*) est une action réalisée lorsque une partie composable s'exporte elle-même. C'est le cas des `UserControl` des exemples précédents. Dans ce cas la classe est simplement décorée d'un attribut `Export`.

Exemple :

```
[Export]
public class UnePartieComposable {  ... }
```

Exportation de propriétés

Une classe peut exporter une ou plusieurs propriétés sans être elle-même une partie composable et exploitable comme telle. C'est le cas du premier exemple où un champ de type `string` est exporté (le message). Il s'agit, pour être tout à fait exact, d'un cas non pas identique mais similaire puisque nous parlons ici d'exportation de propriétés et non de champs. Mais, nous l'avons vu, cela est possible et fonctionne de fait de la même façon.

Exemple d'exportation et d'importation d'une propriété :

```
public class Configuration
  {
    [Export("Timeout")]
    public int Timeout
    {
      get {return int.Parse(ConfigurationManager.AppSettings["Timeout"]);}
    }
  }
  [Export]
  public class UsesTimeout
  {
    [Import("Timeout")]
    public int Timeout { get; set; }
  }
```

La valeur entière `Timeout` de la classe `Configuration` est exportée sous le nom de contrat "`TimeOut`" alors que sa classe mère (`Configuration`) n'est pas composable. La valeur est ensuite importée par la classe `UsesTimeout` qui utilise l'entier en déclarant

une propriété décorée par l'attribut `Import` utilisant le même nom de contrat. La classe mère (`UsesTimeout`) participe, en outre, au jeu de composition puisqu'elle est marquée par `Export`, ce qui n'est qu'une possibilité et non une obligation.

Exportation de méthodes

C'est un cas que nous n'avons pas vu dans cet article car je préfère le principe d'exportation d'une classe de services, quitte à ce qu'elle ne contienne qu'une seule méthode, que l'exportation d'une ou plusieurs méthodes d'une classe qui ne serait pas composable.

Ce jeu qui consisterait à « piocher » ici et là des petits morceaux ne me plait guère et présente le risque important de tomber rapidement dans du code spaghetti…

Mais si je peux m'autoriser quelques conseils, je ne suis pas un censeur, et comme l'exportation des propriétés ou des champs (que je déconseille pour les mêmes raisons), je me dois de vous parler de l'exportation des méthodes. Il serait bien prétentieux de penser que si je ne vois aucune bonne justification à la chose personne sur terre n'en trouvera jamais une ! Et si ce cas se présente à vous, il faut savoir que cela est possible.

Une partie peut donc en effet exporter des méthodes sans être elle-même une « partie composable ».

Une chose à savoir : en raison d'une limitation du Framework (selon la documentation de MEF d'où sont tirés les morceaux de code de cette section) l'exportation des méthodes est limitée à celles ne possédant pas plus de 4 arguments.

Exemple :

```
public class MessageSender
  {
    [Export(typeof(Action<string>))]
    public void Send(string message)
    {   Console.WriteLine(message);    }
  }
  [Export]
  public class Processor
  {
    [Import(typeof(Action<string>))]
    public Action<string> MessageSender { get; set; }

    public void Send()
    {  MessageSender("Processed");   }
  }
```

Dans cet exemple, la classe `MessageSender` exporte la méthode `Send` qui prend en paramètre un message. L'exportation est typée comme une `Action<string>`.

La classe `Processor` fait une importation de la méthode sous la forme d'une propriété qui est utilisée ensuite par sa propre méthode `Send`.

On notera que l'exportation qui est faite ici par l'utilisation d'un type peut se faire aussi par un simple nom de contrat (comme dans notre premier exemple avec la chaîne de caractère `Message`). Les signatures doivent bien entendu rester parfaitement compatibles.

Les exportations héritables

MEF supporte un mode d'exportation très particulier, les exportations *héritables*. Il s'agit en fait de pouvoir décorer une classe ou une interface dont l'exportation est automatiquement héritée par les classes qui les implémentent.

Cela est très intéressant quand on souhaite rendre MEF totalement *transparent*.

Imaginons une interface `ILogger` proposant une méthode `Log(string message)`. Les développeurs vont pouvoir créer plusieurs classes implémentant cette interface. L'une fera des logs dans un fichier texte, l'autre en XML, l'autre via le réseau avec TCP, etc. Toutes ces classes implémenteront `ILogger`. On reste dans de la programmation habituelle, classique, *sans aucune référence à MEF*.

Imaginons maintenant, presque à leur insu pourrait-on dire, qu'on souhaite que toutes les classes implémentant `ILogger` puissent participer à une composition. Le programme principal devra être à même de découvrir toutes les versions de `ILogger` afin que l'utilisateur puisse choisir celui qui lui convient par exemple.

Chaque classe implémentant `ILogger` ne connaît rien de MEF, les développeurs qui les ont écrites non plus.

Mais en décorant l'interface `ILogger` de l'attribut `InheritedExport`, comme par magie, toutes les classes implémentant `ILogger` seront visibles par MEF et pourront être composées… Pratique non ?

Voici un exemple :

```
[InheritedExport]
public interface ILogger {  void Log(string message); }
public class Logger : ILogger {  public void Log(string message); }
```

Ce code reprend ce que je viens de dire : l'interface `ILogger` définit un contrat constitué d'une seule méthode, `Log`, qui permet de logger un message.

L'interface elle-même est « MEF-aware », celui qui l'a écrite connaît MEF.

Mais regardons la classe `Logger` définie juste après : elle ne fait qu'implémenter `ILogger`, rien de plus. Pourtant cette classe pourra participer « sans le savoir » à une composition MEF basée sur l'interface `ILogger`, comme si `Export(typeof(ILogger))` la décorait…

Un procédé original qui peut même permettre, après coup, et pour peu qu'on travaille déjà avec des interfaces, de « MEFiser » et de modulariser une application existante.

Public ou Private ?

MEF supporte la découverte de parties publiques et non publiques. Cela est automatique et ne nécessite aucune action particulière. Mais en revanche il faut savoir que ce mécanisme n'est pas supporté dans les environnements à *confiance partielle*, comme peut l'être Silverlight… Sous Silverlight on prendra comme règle que ne peuvent être composées que des parties (au sens large : classes, propriétés, méthodes…) qui sont publiques.

Cela semble finalement naturel et logique.

Mais il faut se méfier des évidences… Par exemple une application pourrait fort bien vouloir utiliser MEF pour composer des parties totalement `private`, sans vouloir violer aucun principe de la programmation objet. Sous Silverlight cela n'est pas possible. Composable = publique. A se rappeler donc…

Lazy<T> et le chargement différé

Dans les exemples que nous avons vu j'ai volontairement introduit la gestion des métadonnées pour grouper les thèmes afin de conserver une taille raisonnable à cet article (ce qui n'est déjà plus le cas, mais j'aurai essayé !).

Dans ces exemples, il a été montré que pour gérer les métadonnées, et surtout pouvoir les récupérer, il fallait définir les importations avec le type `Lazy<T,M>,` ou `T` est le type de l'objet importé et `M` le type des métadonnées.

Cela est vrai (heureusement). Pour récupérer les métadonnées, typées ou non, il faut utiliser `Lazy<T,M>` car seule cette déclaration permet de spécifier que les métadonnées doivent être gérées et est capable de les retourner en encapsulant les parties dans un objet incluant une propriété `Metadata`. On a vu d'ailleurs que dans ce cas l'objet exporté lui-même est accessible via la propriété `Value` et non pas directement.

Si tout cela est bien vrai je vous ai caché une partie de la vérité. Il est temps de l'avouer : cette technique pour récupérer les métadonnées n'est qu'une astuce. Officielle certes, mais une astuce tout de même car il n'existe pas d'autres moyens d'obtenir ces données spéciales dans MEF.

C'est une astuce car comme son nom le laisse supposer depuis le départ, `Lazy<T>` ou `Lazy<T,M>`, les deux existent, ont surtout été conçues pour permettre le *chargement tardif* des objets d'une composition. On peut ainsi utiliser `Lazy<T>` si on souhaite faire du chargement tardif sans pour autant utiliser de métadonnées.

Lazy veut dire paresseux en anglais. Le *Lazy loading* consiste à charger des objets « paresseusement » c'est-à-dire tout le contraire de « rapidement ».

Dans le jeu des compositions de MEF il faut prendre en compte le temps de découverte des parties puis de leur initialisation sachant que MEF gère les dépendances et qu'une partie composable peut réclamer, par un effet de bande, l'initialisation de nombreuses autres parties liées par des dépendances.

Une application « moderne » est avant tout une application qui se charge et qui réagit vite. Sur un PC, mais aussi, il faut en tenir compte aujourd'hui, sur des machines beaucoup moins puissantes que sont les Smartphones et les Tablettes… Et avec Silverlight il faut aussi *prendre en compte la lenteur d'Internet*.

Une application utilisant MEF ne le fera pas pour un ou deux modules. Une application modulaire se basant sur MEF le fera certainement pour des « tas » de modules et parties, parfois réparties sur plusieurs XAP qui devront être téléchargés via Internet (ou un Intranet dans le meilleur des cas).

Or, que se passe-t-il si une application tente de découvrir et de charger toutes les parties composables ? Cela va être long, voire très long. Tout le contraire de ce qu'on cherche à obtenir.

Pour cette raison le type `Lazy<T>` ou `Lazy<T,M>` quand on spécifie les métadonnées, est avant tout conçu pour *différer le chargement des parties composables*. La découverte est effectuée au plus vite, mais le chargement est mis en attente.

Jusqu'à quand ?

Jusqu'à ce que la propriété `Value` soit lue…

Pour faire du *Lazy Loading* avec MEF, il suffit donc de remplacer le type de la partie composable par un `Lazy<T>` dans la déclaration de l'importation.

Ainsi l'hypothétique déclaration :

```
[Import]
public IMessageSender Sender { get; set; }
```

passera en mode *Lazy Loading* en écrivant :

```
[Import]
 public Lazy<IMessageSender> Sender { get; set; }
```

Ceci est valable pour les importations simples (comme ci-dessus) autant que pour les importations multiples, avec, ou comme ici, sans métadonnées.

Chargement différé immédiat

Derrière ce titre en forme de paradoxe se cache une stratégie assez souvent utilisée : elle consiste à proposer un Shell le plus léger possible et, dès que celui-ci est affiché, à démarrer une tâche de fond qui télécharge tout (ou partie) des modules à utiliser. Le temps que l'utilisateur saisisse son login par exemple, qu'une première page soit affichée, l'application peut avoir le temps de télécharger tous les modules complémentaires…

Ici on utilise bien un chargement différé ou tardif, mais en lançant ce chargement dès le début de l'exécution du programme.

L'utilisateur dispose ainsi d'une application qui se charge très vite et le temps de chargement des modules est masqué astucieusement.

Au fur et à mesure que les modules arrivent et sont découverts, des entrées de menu peuvent s'ajouter, des boutons de navigation apparaître, etc.

C'est une stratégie intéressante qui semble malgré tout éloignée du but du *Lazy Loading*. En réalité ici on vise principalement le chargement rapide du logiciel, c'est tout. Les modules sont tous chargés en mémoire systématiquement. Certaines applications profitent mieux de la modularisation de cette façon. Comme toujours, les décisions doivent être prises projet par projet, en fonction des impératifs propres à chacun.

Je ne traiterai pas cet exemple ici, faute de place, mais voici un code exemple issu de la documentation CodePlex de MEF qui montre de façon assez simple la stratégie :

```
public class App : Application {
  public App() {
    this.Startup += this.Application_Startup;
    this.Exit += this.Application_Exit;
    this.UnhandledException += this.Application_UnhandledException;

    InitializeComponent();
  }

  private void Application_Startup(object sender, StartupEventArgs e)
  {
    var catalog = new AggregateCatalog();
    catalog.Catalogs.Add(CreateCatalog("Modules/Admin"));
    catalog.Catalogs.Add(CreateCatalog("Modules/Reporting"));
    catalog.Catalogs.Add(CreateCatalog("Modules/Forecasting"));

    CompositionHost.Initialize(new DeploymentCatalog(), catalog);
    CompositionInitializer.SatisfyImports(this)

    RootVisual = new MainPage();
  }

  private DeploymentCatalog CreateCatalog(string uri) {
    var catalog = new DeploymentCatalog(uri);
    catalog.DownloadCompleted += (s,e) => DownloadCompleted();
    catalog.DownloadAsync();
    return catalog;
  }

  private void DownloadCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error != null) {
      MessageBox.Show(e.Error.Message);
    }
  }

  private Lazy<IModule, IModuleMetadata>[] _modules;

  [ImportMany(AllowRecomposition=true))]
  public Lazy<IModule, IModuleMetadata>[] Modules {
    get{return _modules;}
```

```
    set{
      _modules = value;
      ShowModules()
  }

  private void ShowModules() {
    //logic to show the modules
  }

}
```

MEF, MVVM et Chargement Dynamique de XAP

Jusqu'à maintenant nous avons vogué par vent arrière sur une mer calme. Ici on se retrouve au Vendée Globe (donc en solitaire, sans escale et sans assistance), lors du passage du cap Horn par gros temps…

En effet, nous allons voir comment créer une application modulaire utilisant MEF capable de charger dynamiquement des modules au *runtime* suite à une action utilisateur (depuis un module découvert par MEF), les modules d'extension se trouvant dans un XAP secondaire situé sur le serveur, le tout en respectant la pattern MVVM et en limitant la taille de chaque exécutable. Pire, dirais-je, nous allons ajouter une gestion de catalogue de modules permettant de charger d'autres parties en fonction du profil de l'utilisateur !

Ceux qui survivront auront le droit à mes chaleureuses félicitations ☺

Même si j'ai bien chargé la barque, le projet que nous allons étudier reste malgré tout un simple exemple. La réalité est toujours plus complexe et soulève toujours mille nouvelles questions. La sagesse impose de garder cette évidence à l'esprit.

Mon but n'est d'ailleurs pas d'écrire un livre sur MEF ni de me substituer à la documentation du Framework. Je souhaite seulement vous donner le coup de pouce nécessaire et suffisant pour vous permettre de prendre le large tout seul. Certains préfèrent aux courses dangereuses en solitaire la sécurité des croisières en compagnie d'un capitaine expérimenté qui saura les conduire à bon port, j'en profite ainsi pour rappeler à ces derniers que je vends mes services ☺

Le projet

Le projet reprend l'esprit des derniers exemples de code de cet article : il y a un *shell*, la vue principale, qui permet d'afficher des widgets (des gadgets, des parties, des modules, tout cela revenant au même).

Le dernier exemple présenté permettait, via les métadonnées MEF, d'indiquer la position de l'affichage de chaque module. Nous reprendrons cette idée en jouant ici sur deux conteneurs (une partie haute et une partie basse) et ajouterons aux métadonnées d'autres informations comme le nom du widget afin qu'il puisse être affiché dans une `Listbox` qui présentera ainsi tous les modules chargés.

L'autre aspect que notre projet devra couvrir est le chargement dynamique des modules. De fait les « widgets » seront chargés et découverts de plusieurs façons différentes :

- Un premier widget sera découvert dans l'application principale et sera immédiatement affiché dans le shell.
- Ce widget comporte un bouton permettant de charger à la demande d'autres widgets se trouvant dans un XAP secondaire sur le serveur.
- Un profil utilisateur simplifié sera affiché dans la vue principale. Le changement de profil entrainant le chargement de nouveaux modules adaptés à ce profil, via un catalogue XML se trouvant sur le serveur.

Cela représente trois types de chargement et de découverte des modules. Certains sont internes à l'application principale, d'autres proviennent de XAP externes. Le chargement des modules externes étant lié soit à une action utilisateur, soit à une décision de l'application.

Si nous nous arrêtions là cela ferait déjà beaucoup… Mais il manquerait une autre approche couramment utilisée : La découverte de widgets par le biais d'un catalogue sur le serveur.

En effet, la gestion de catalogue de MEF est parfaite pour WPF. De base MEF sait retrouver tous les modules d'un sous-répertoire donné par exemple. Mais rappelez-vous que Silverlight ne peut pas accéder aux répertoires du serveur. Il est donc impossible d'utilise ces mécanismes fournis avec MEF. Il va être nécessaire d'implémenter notre propre gestion de catalogue.

Une petite séquence de captures d'écran va éclaircir tout cela avant de passer à l'étude du code.

Le projet en action

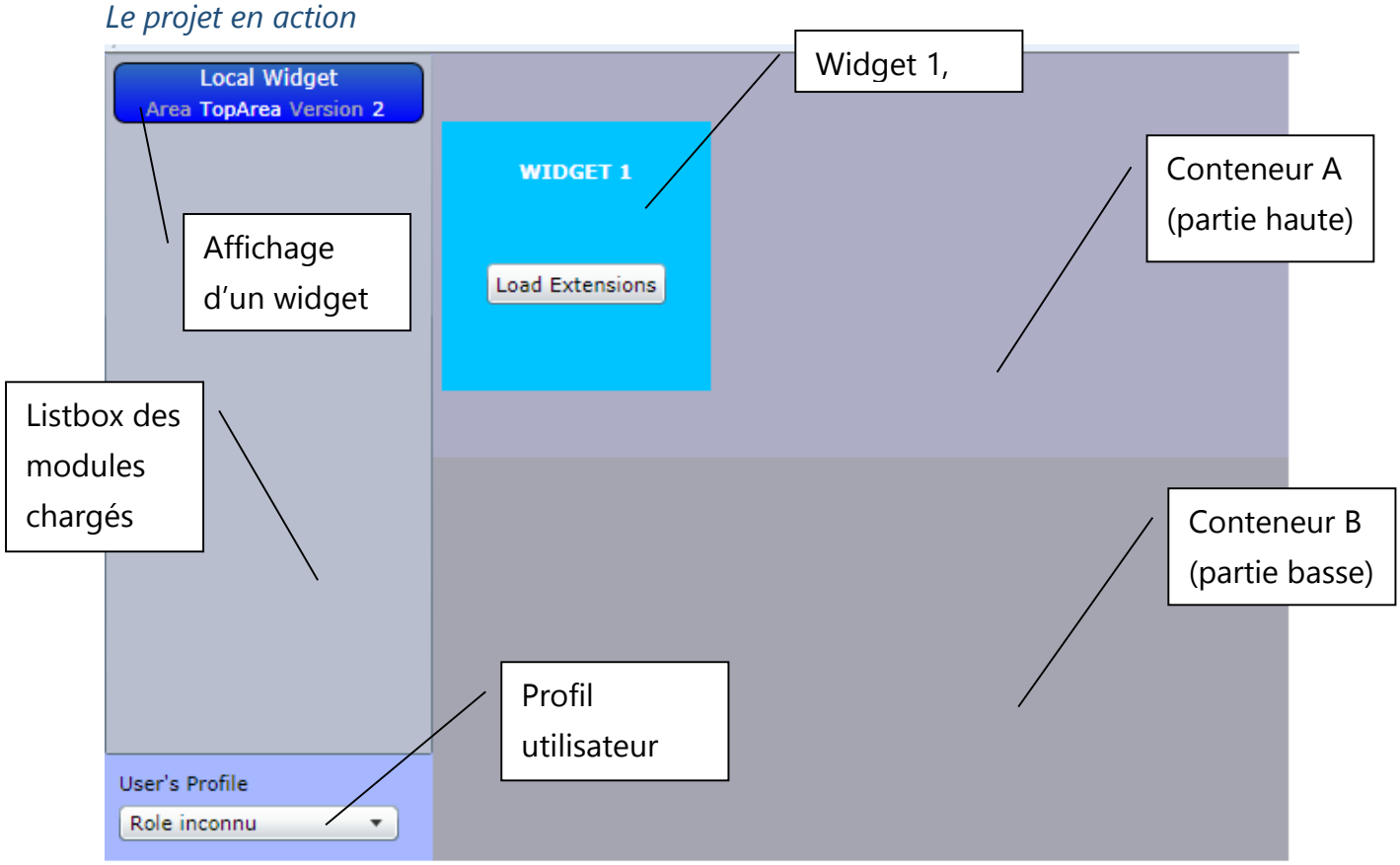


Figure 59 - DynamicXAP - Le Shell au lancement

Comme le montre la capture (figure 10), le shell se présente sous la forme de deux parties verticales, à gauche la `Listbox` affichant les modules chargés et en bas le profil utilisateur, à droite l'affichage des widgets coupé en deux horizontalement, avec un conteneur A en partie supérieure et un conteneur B en partie inférieure.

Ce sont les métadonnées des widgets qui indiquent la position d'affichage (en haut ou en bas) ainsi que le nom qui apparaît dans la `Listbox`.

Chaque module chargé est ainsi affiché dans la `Listbox` et présente les informations suivantes :

- Le nom du widget (tel que précisé dans ses métadonnées)
- Le nom de la zone où il est affiché (`TopArea` ou `BottomArea`)
- La version du widget (information factice uniquement pour ajouter quelques métadonnées de plus).

Vous noterez que dès le lancement de l'application le shell présente un premier widget. Ce dernier existe dans le XAP principal. Il est découvert automatiquement par le mécanisme d'initialisation de MEF. Ce widget présente la particularité d'offrir un bouton « `Load Extensions` » (charger les extensions). Un clic sur ce bouton entraîne

le chargement des widgets contenus dans le XAP « `extensions.XAP` », ce dernier est donc téléchargé depuis le serveur puis « dépiauté » par MEF pour y découvrir les modules.

Cliquons sur le bouton :

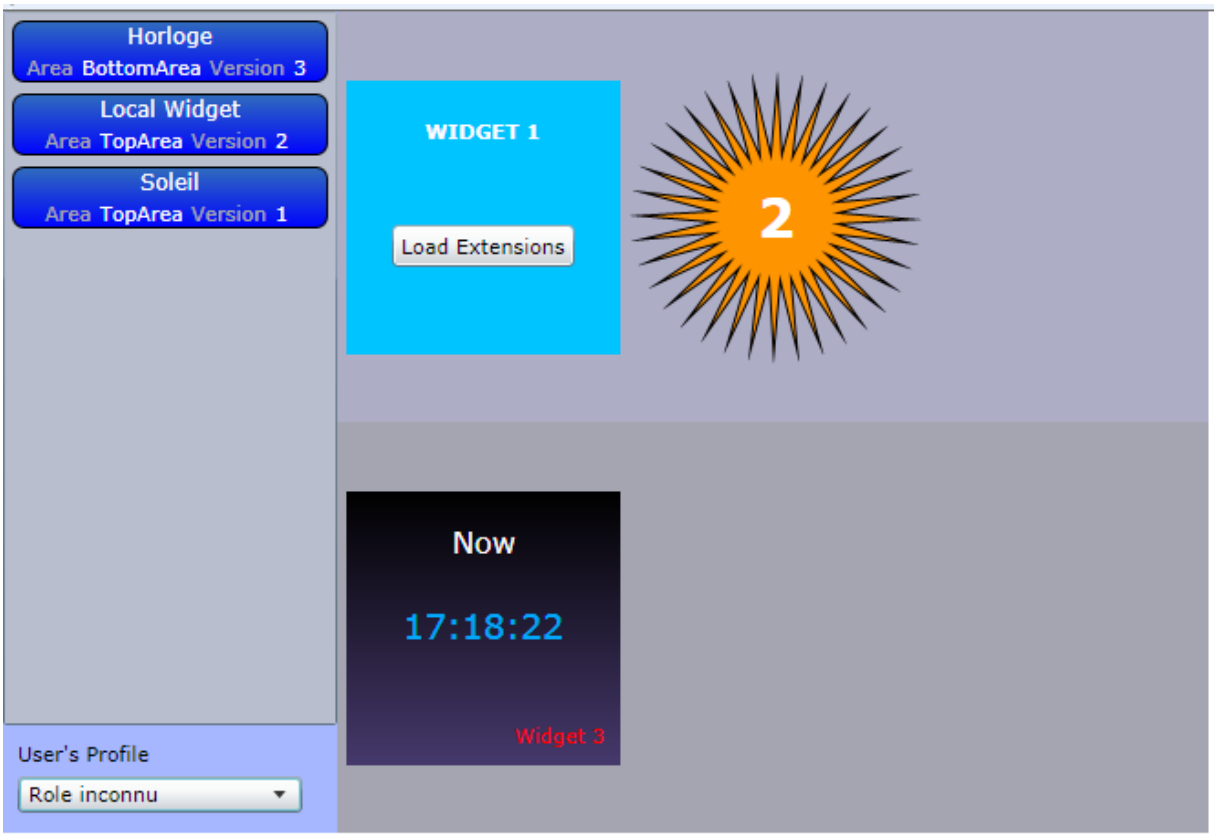


Figure 60 - DynamicXAP - Chargement des extensions

La capture ci-dessus nous montre le shell après que nous avons[3] cliqué sur le bouton « `Load Extensions` ».

La `Listbox` affiche désormais trois widgets (classés par ordre alphabétique de leur nom). Les trois widgets (le premier ainsi que les deux nouveaux) sont répartis entre le conteneur A et le conteneur B selon les indications portées par leurs métadonnées respectives (ce qu'on peut vérifier dans la `Listbox`).

Pour l'instant la partie gérant le profil utilisateur indique « *Rôle inconnu* ». Je concède volontiers que si je fais beaucoup d'efforts lors de l'écriture d'un article comme le

[3] C'est bien l'indicatif qui s'emploie après « après que », même si le subjonctif, condamné par les grammairiens, est d'usage de plus en plus fréquent. Le subjonctif ne s'emploie qu'après « avant que ». D'où la confusion ? C'était la petite minute de « Maitre Capello » qui nous quitté cette année en mars à 88 ans…

présent, je n'hésite pas mêler anglais, français et franglais dans les applications exemples… *Mea culpa est*.

Cliquons sur la `ComboBox` du rôle et choisissons le rôle « *Administrateur* » :

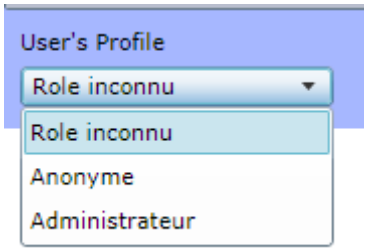


Figure 61 - DynamicXAP - Le rôle utilisateur

L'affichage devient :

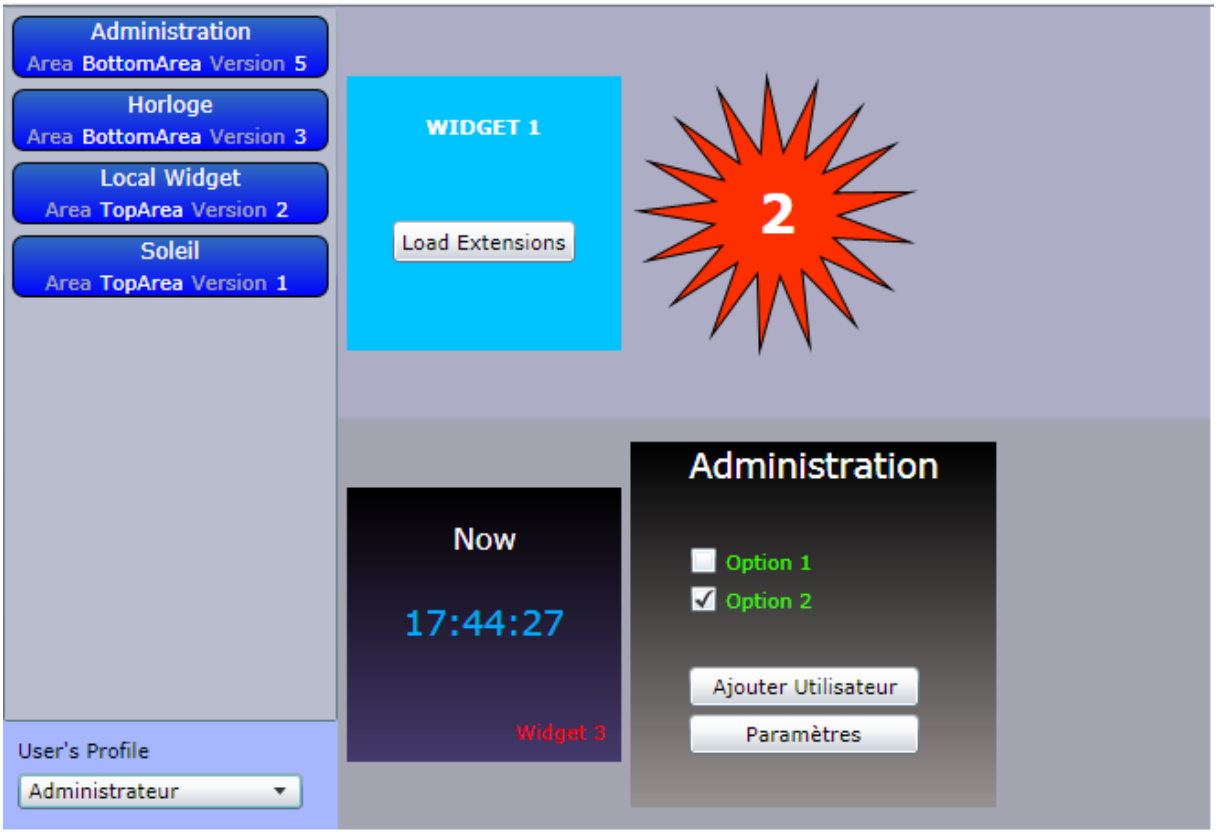


Figure 62 - DynamicXAP - Rôle Administrateur

La sélection du rôle Administrateur a déclenché l'affichage du widget d'administration (totalement fictif).

Si nous avions choisi le rôle « *Anonyme* », l'affichage serait :

Figure 63 - DynamicXAP - Rôle Anonyme

Le widget « `Anonymous Gadget` » est désormais affiché dans le conteneur A (puisque ses métadonnées précisent l'emplacement « `TopArea` ».

Bien entendu, le rôle de l'utilisateur serait géré en interne suite à un login dans une application réelle. Ici le choix est laissé dans l'interface pour simplifier la démonstration. S'il s'agit d'un autre type de sélection, comme l'affichage de certaines palettes d'outils par exemple, le choix d'une `ComboBox` se révèlerait mieux approprié.

De même, si vous testez directement l'application (ce que je vous incite à faire !) vous verrez rapidement que le choix d'un rôle n'annule pas la situation précédente et qu'au final on se retrouve avec les modules de tous les rôles affichés. Dans la réalité il faudrait ajouter une séquence de « nettoyage » qui cacherait les modules devenus inutiles.

Mais l'application est déjà bien assez complexe comme cela…

Ce que nous n'étudierons pas

Dès qu'une application grossit un peu elle doit faire face à de nombreux petits soucis accessoires qu'il faut bien gérer. Bien que centrée sur MEF mon application de démonstration n'échappe pas à la règle.

Elle contient ainsi une certaine quantité de code qui n'est pas directement lié à ce que je veux faire voir, code que je ne commenterai pas ici.

Néanmoins il m'a semblé important de vous indiquer les quelques « astuces » utilisées :

ListBox

Lorsqu'on fournit un `DataTemplate` à une `ListBox`, un phénomène fâcheux apparait : chaque instance du `DataTemplate` s'adapte à la taille de son contenu ce qui donne des items de tailles différentes, et ce n'est pas très beau. A cela une raison, l'un des conteneurs par défaut à son placement horizontal en mode « `Left` » ou lieu de « `Stretch` ». Il faut templater la `ListBox` pour modifier ce comportement. Voir mon billet : « *Largeur de DataTemplate et Listbox* » (http://www.e-naxos.com/Blog/post.aspx?id=ee204aa8-caa0-4211-92a4-9595fc20b712).

ComboBox

La `ComboBox` affiche une énumération (les différents rôles gérés par l'application), or les énumérations ne sont pas des collections et le Framework de Silverlight ne contient pas la méthode `GetValues` de WPF pour obtenir une liste des valeurs possibles.

De même, les valeurs internes de l'énumération ne sont pas forcément celles qu'on souhaiterait afficher.

La première chose à rappeler est que les énumérations sont parfaites pour matérialiser les états internes d'une application, mais qu'elles ne devraient jamais être utilisées dans les interfaces. Exposer directement des états internes dans l'UI créé un couplage code / UI non souhaitable et complique le travail de localisation. Mais parfois, comme dans l'exemple proposé, cela peut sembler pratique.

Ce dernier aspect est discutable lorsqu'on voit la quantité de code à ajouter (trois classes au moins plus le code XAML) pour finalement utiliser une énumération dans une `ComboBox`.

Regardez comment cela est fait, c'est toujours instructif, mais évitez d'utiliser cette astuce dans vos applications.

Voir mes billets suivants autour du même sujet :

« Conversion d'énumérations générique et localisation » (http://www.e-naxos.com/Blog/post.aspx?id=f197deab-5f45-4674-894d-a0d64c35ce8a)

Silverlight : « `Enum.GetValues` qui n'existe pas, binding et autres considérations » (http://www.e-naxos.com/Blog/post.aspx?id=8d8f4a13-698b-4666-b68e-d8fe9a7d0f1e)

Par où commencer ?

C'est la question qui se pose maintenant.

Par exemple, faut-il créer un projet avec le modèle fourni par le toolkit MVVM Light ou plutôt partir d'une solution vierge et ajouter MVVM par petites touches ?

On peut aussi se demander si MEF ne peut pas à lui tout seul remplacer un toolkit MVVM, les vues et les modèles de vue pouvant être connectés via des Import / Export sans l'aide de l'indirection d'un service locator comme MVVM Light le fait.

L'injection de dépendances façon Prism ne fait-elle pas un doublon avec l'esprit même de MEF ?

Et Unity ? Ça peut servir où cela serait mélanger trop de choses proches ensemble ?

Comment répartir le code entre combien de projets ? Cela aussi est important.

Et puis, faut-il utiliser MVVM Light, Jounce, Prism ou Caliburn.Micro ?

Toutes ces questions ramènent à réussir l'ordonnancement des différentes couches et à choisir correctement chaque librairie.

Prism, MEF, Jounce, Unity, MVVM Light, Caliburn ?

Ce n'est pas le sujet de cet article mais comment entrer dans le vif de ce même sujet si on n'a pas une idée, au moins vague, des différences principales entre ces toolkits ou frameworks alors même que nous allons en utiliser au moins deux ?

Alors en quelques mots :

MVVM Light

C'est un toolkit qui se focalise uniquement sur l'application du pattern MVVM. Il fournit un système de messagerie et quelques classes de base (pour les ViewModels par exemple) et un service Locator simplifié permettant de créer un découplage entre

Views et ViewModels. L'intérêt de MVVM Light est de ne faire que MVVM sans vouloir s'occuper de tous les problèmes possibles et d'être conçu pour assurer la « *blendability* » c'est-à-dire la capacité de disposer de données pour faciliter le design sous Blend, point sur lesquels les autres font l'impasse. MVVM Light, par son ancienneté (relative) utilise un style moins « moderne » que d'autres librairies comme Jounce par exemple. La Blendabilité est assurée par des mécanismes de code pur, là où Jounce utilise plus astucieusement les nouvelles possibilités de Blend (comme la déclaration de données de Design).

Prism

Il ne s'intéresse pas directement à MVVM mais fournit la plupart des outils permettant de mettre en œuvre le pattern et ces cousines tel MVC. Toutefois il s'occupe aussi d'autres questions (comme la gestion des commandes et des modules) et se destine plus à la gestion des applications basées sur des modules visuels avec, par exemple, la notion de régions dans lesquelles les modules viennent s'afficher. C'est plus un guide qu'un toolkit mais étant fourni depuis le début aussi sous la forme de code, Prism est devenu au fil du temps un toolkit à part entière tout en se refusant de l'être. Position ambiguë. Il est riche, bien pensé, mais complexe et il faut donc du temps pour le maîtriser.

MEF

C'est aujourd'hui une partie du Framework Silverlight et de .NET 4.0. Cela lui donne un statut différent, c'est un morceau du Framework tout simplement. MEF vise la modularité comme Unity mais favorise un modèle d'extensibilité « externe », via des modules extérieurs à l'application, de type plugin, même si l'approche « intérieure » est possible et même la plus simple. Il sait découvrir et charger des modules depuis des répertoires, il sait extraire des modules d'un XAP. Assez simple de prime abord, comme vous pouvez le constater au fil de cet article, ce n'est pas forcément simpliste non plus… Reste que MEF est une partie du framework et qu'il faut savoir s'en servir, un peu comme LINQ dans un autre domaine.

Unity

Cette librairie vise aussi à la modularité mais dans un esprit plus « statique » et s'utilise plus comme une technologie « intérieure », c'est-à-dire que personne ne voit, ne sait ni n'a besoin de savoir qu'une application utilise Unity. Alors que MEF se destine plutôt à la « croissance extérieure » comme stratégie. Unity est avant tout un toolkit d'injection de dépendances ce qui en fait un allié des projets utilisant

beaucoup de tests unitaires ou de mocks. Les ressemblances avec MEF sont toutefois nombreuses (MEF est aussi basé sur l'injection de dépendances).

Jounce

Jounce est un nouveau toolkit pour Silverlight (là où les autres s'appliquent aussi à WPF et à WP7 généralement). Son but est de fournir des blocs de base pour l'écriture d'application de gestion modulaires qui utilisent les patterns MVVM et MEF. Jounce s'inspire ouvertement de Prism et de Caliburn.Micro avec lesquels il peut s'utiliser tout en pouvant s'en passer. Un peu ambigu aussi, car Jounce n'était au départ qu'une librairie « personnelle » de l'auteur qui se défendait de publier un « framework de plus », tout en le faisant… Mais c'est une librairie très intéressante à plus d'un titre. La version 1.0 qui vient d'être releasée en fait un concurrent sérieux de MVVM Light en apportant une note de « fraicheur » dans le traitement des problèmes. Jounce est tellement intéressant que je vais lui consacrer un prochain article.

Caliburn.Micro

Caliburn était presque plus complexe que Prism. Son auteur est reparti de zéro en reprenant l'essentiel. Caliburn.Micro n'est pas qu'une version simplifiée de Caliburn, c'est une nouvelle version qui remplace totalement Caliburn. Ce toolkit est un mélange de différentes solutions pour Silverlight, WPF et WP7 permettant de simplifier l'écriture des applications suivant MVVM, MVP ou même MVC. Son approche est différente des autres et contient beaucoup de bonnes idées. La nouvelle version « Micro » le rend certainement plus envisageable que l'ancienne, par trop complexe à mon goût (et même à celui de son auteur !).

L'heure du choix

En voici une belle brochette de toolkits, de quoi écrire des articles jusqu'à ma lointaine retraite !

Mais il faut faire des choix, c'est toujours difficile.

D'un autre côté, dites-vous qu'un monde où le choix n'existe pas n'est que très rarement un endroit où il faut bon vivre…

Personnellement j'évite de conseiller Prism pour sa complexité. Seules des équipes vraiment motivées, soudées, homogènes et bien formées peuvent s'en servir, cas de figure qui n'est pas le plus courant dans la réalité.

J'ai écarté Caliburn pour les mêmes raisons. La version Micro remet cette décision en cause et je conseille au lecteur de tester cette nouvelle mouture. J'ai hésité longtemps d'ailleurs entre Caliburn.Micro et Jounce pour le thème de mon prochain article. Même si mon choix se porte plutôt sur Jounce, que cela ne vous empêche pas de vous faire votre propre idée sur Caliburn.Micro !

Unity est séduisant. Ce toolkit et utilisé par Prism pour faire de l'injection de dépendance. Mais MEF est intégré au Framework aujourd'hui. Autant maîtriser et utiliser ce qui est fourni dans la boîte avant d'aller chercher plus loin. Surtout que si les ressemblances sont nombreuses les nuances font, à mon sens, pencher la balance vers MEF.

En conclusion, le modèle le plus simple à comprendre et à mettre en œuvre aujourd'hui, même s'il ne fait pas tout (surtout parce qu'il ne fait pas tout !) consiste à marier MEF et MVVM Light.

Ce n'est pas la panacée, les autres toolkits sont pleins de bonnes idées aussi et de services absents de cette association. Mais par expérience je préfère conseiller à mes clients des toolkits assez faciles à apprendre, quitte ponctuellement à emprunter un morceau de code à d'autres toolkit (ils sont tous open source) plutôt que de les lancer sur des « monstres ». Lorsque leur équipe évolue, qu'untel s'en va, que deux nouveaux arrivent, il ne faut pas six mois pour avoir de nouveau une équipe opérationnelle. C'est à mon sens un des critères essentiels.

De fait, l'exemple qui va suivre utilisera MEF + MVVM Light. Chacun pourra adapter selon ses goûts et ses besoins. La présence de MVVM Light est tellement … légère que lui substituer un autre framework plus complexe ne posera pas de problème à ceux qui en maîtrisent déjà un.

> *Ayant traité de MVVM et MVVM Light dans de très longs articles, j'ai choisi de conserver MVVM Light ici principalement pour éviter de « mettre la charrue avant les bœufs ». Comme on le verra, le mariage MVVM Light +MEF n'est pas idéal. Jounce que j'aborderai dans un prochain article a l'avantage d'être construit d'emblée comme un toolkit MVVM se reposant sur MEF. En toute logique c'est cette solution là que nous devrions choisir. Mais dans cet article présentant MEF, je préfère utiliser MVVM Light sur lequel le lecteur pourra trouver toutes les explications dans mes précédents papiers alors que je n'ai pas encore abordé Jounce… Cela va venir, mais un article après l'autre !*

L'architecture globale

Le partage des tâches

Les toolkits et frameworks étant choisis, il faut décider de l'organisation générale et de la séparation des tâches.

Par exemple, MVVM Light offre un locator permettant de lier les vues aux ViewModels. Utilisant des champs statiques cette classe est la clé de voute de la *blendability* de MVVM Light. Le locator assure aussi le découplage entre les vues et l'implémentation de leurs ViewModels.

Conserver le locator de MVVM Light n'aurait pas de sens puisque dans ce cas il en serait terminé de la découverte dynamique de vues que nous allons mettre en œuvre. En revanche le découplage des vues et des ViewModels est essentiel, mais cela MEF le permet très bien puisqu'il gère l'injection de dépendances naturellement. Donc nous n'utiliserons pas le locator de MVVM Light.

Cependant, MEF ne propose pas de procédé de communication comme l'`EventAggregator` de Prism. Ni même de service de gestion des commandes. Mais cela existe dans MVVM Light (qui offre une messagerie pour les communications et la classe `RelayCommand` pour les commandes).

De même MVVM Light offre une classe de base pour les ViewModels qui supportent `INotifyPropertyChanged`. MEF ne se charge pas de cet aspect.

Finalement, le couple MEF + MVVM Light se révèle un choix intéressant. Deux toolkits assez simples, car concentrés sur un problème précis chacun, qui se marient bien puisque ne se recouvrant pas ou très peu.

L'organisation de la solution

La solution Visual Studio se décompose en 6 projets :

- `Contract`, un projet de type Silverlight Class Library, il contient la définition des contrats
- Shell, l'application principale Silverlight
- `Shell.Web`, l'application Web qui héberge la précédente ainsi que tous les XAP et le catalogue XML
- Extensions, le premier XAP d'extensions dynamiques. Projet Silverlight classique sans `MainPage` ni `App.Xaml`.
- `Extensions2`, le second XAP d'extensions dynamiques. Construit comme le précédent et contenant les modules pour le profil utilisateur « *anonyme* »

- `AdminExtensions`, troisième fichier XAP d'extensions dynamiques. Construit comme les deux précédents et contenant les modules pour le profil « `Admin` »

Le projet Silverlight principal (*Shell*) est créé comme une application Silverlight classique accompagnée de son projet Web d'hébergement (`Shell.Web`).

Le projet Web n'offre aucun service particulier, il sert juste de centralisateur via son répertoire `ClientBin` qui héberge tous les XAP et le catalogue. Il propose aussi une page de test html permettant d'exécuter l'application Shell. Au final le projet pourrait être déployé sur un serveur Linux, cela ne poserait aucun problème. Ni ASP.NET, ni aucun service Web ou WCF n'étant utilisé (ce qui réclamerait un serveur IIS).

Dans la pratique c'est l'application `Shell` accompagnée de `Shell.Web` qui ont été créées en premier. Le projet `Contract` a été écrit en créant le `Widget1` intégré à l'application `Shell`. Mais cela s'explique par le fait que j'ai construit l'exemple « à la volée » sans écrire d'analyse préalable. Dans un contexte réel le premier projet écrit serait `Contract` qui définit l'ensemble des contrats des modules.

C'est d'ailleurs par-là que nous allons commencer la visite du code.

Le code de l'exemple

Il est bien entendu hors de question de lister ici le contenu C# et XAML des six projets qui constituent la solution… cela serait vraiment fastidieux.

Les contrats

Dans une application de ce type il est indispensable de fixer les contrats dès le départ (quitte à faire quelques aménagements en cours de développement). Les contrats cela regroupe aussi bien les classes de base, les classes abstraites que les interfaces partagées par les modules, les interfaces fixant les métadonnées utilisées avec MEF ou les attributs MEF d'exportation personnalisés. Selon la taille du projet il peut devenir judicieux de séparer au moins en deux packages les contrats, d'un côté les contrats métiers ou fonctionnels et de l'autre les contrats spécifiques à MEF. Ici tout est regroupé dans le projet `Contract`.

C'est un projet de type *Silverlight Class Library*. Il référence l'unité `System.ComponentModel.Composition`.

> *Il est important de noter que les unités référencées, comme celles de MEF, sont déjà déployées par le projet principal* `Shell`*, de fait, pour diminuer la taille des XAP satellites ces unités sont marquées « **Copie Locale = False** » dans VS. La*

modularité qu'offre MEF diminue la taille de l'application principale, mais aussi celle des projets contenant les modules, chaque centaine de Ko gagnée rend au final l'application plus réactive et moins consommatrice de ressources réseau.

IWidgetBehavior

Cette interface définit le comportement commun de tous les widgets. Dans une gestion modulaire il est important de *penser par contrat*. Le `Shell` ne peut pas, et ne doit surtout pas, connaître les classes de chaque module. *Il doit pouvoir manipuler les modules de façon générique*.

S'il existe plusieurs types de modules (ce qui est souvent le cas : modules visuels, modules de services…) il existe plusieurs interfaces de ce type.

Le code de l'interface est :

```
namespace Contract
{
    public interface IWidgetBehavior
    {
        bool IsActive { get; set; }
        bool IsSelected { get; set; }
        string WidgetName { get; set; }
        WidgetArea Area { get; set; }
        int Version { get; set; }
    }
}
```

Ce contrat définit des propriétés, pour la plupart « factices » uniquement là pour l'exemple. Dans la réalité il contiendrait aussi des méthodes permettant d'agir sur les modules ou leur demander d'effectuer les tâches pour lesquels ils sont conçus, voire d'énumérer les commandes de menu qu'ils contiennent et qui devraient être ajoutées au menu principal du `Shell`. Il n'y a pas de limite, tout dépend de l'application et de ce que doivent faire les modules.

Les propriétés `IsActive` et `IsSelected` ne servent pas dans l'exemple.

La propriété `WidgetName` retourne le nom du module tel qu'il doit se présenter à l'utilisateur (dans la `ListBox`). Il pourrait être issu directement du module, dans notre exemple il sera alimenté après coup par le `Shell` en fonction des métadonnées des widgets qui définissent déjà le nom à afficher.

La propriété `Area` est elle aussi un « repiquage » des métadonnées. Elle indique l'emplacement où le module doit être affiché (en haut ou en bas).

La propriété `Version`, elle aussi reprise des métadonnées et représente la version du module. Il est intéressant de pouvoir gérer cette information dans une application modulaire. Les parties pouvant venir de plusieurs XAP, ces derniers pouvant être dans le cache du browser ou d'un proxy, en cas de problème chez un utilisateur cette information de version peut s'avérer indispensable pour le débogage. Ici ce n'est qu'un entier, dans la réalité on utilisera une version complète exprimée en 3 ou 4 groupes (ex : `2.3.58.599`).

WidgetInfo

Cette classe est une implémentation rudimentaire de l'interface précédente. Elle sera utilisée pour créer la liste affichée par la `Listbox`. Pourquoi cette redondance puisque les modules contiennent cette information et que les métadonnées fixent les principales informations de leur côté ?

En réalité, la liste des modules qui sera créée par MEF, comme dans les derniers exemples, sera une liste de `Lazy<T,M>`. Le champ `Value` pointera vers le `UserControl` définissant le widget. La lecture de `Value` entraîne le chargement du module (puisque nous sommes en mode *Lazy Loading*). Le module, une fois instancié appartient à la collection créée par MEF. Puisqu'il s'agit de modules visuels, ils seront affichés par le `Shell`. Chaque module étant un `UserControl`, la parenté visuelle de ce dernier sera donc fixée par le `Shell` qui l'ajoutera à l'un des deux conteneurs.

Si j'utilise maintenant la liste des modules retournée par MEF pour nourrir la `ListBox`, le `UserControl` contenu dans `Value` va être « capturé » par cette dernière. Or il appartient déjà à l'arbre visuel. Un objet ne peut pas être deux fois dans l'arbre visuel à des endroits différents.

Pour simplifier les choses et éviter ce problème, une liste de `WidgetInfo` sera créée par le `Shell` lorsque la liste des modules sera complétée par MEF. Cette liste, totalement déconnectée des widgets eux-mêmes pourra alors être affichée dans la `ListBox` sans que se pose le problème évoqué.

Ensuite, les widgets exposent, nous l'avons vu, un certain nombre d'informations (voire de méthodes) via l'interface `IWidgetBehavior`. L'implémentation de cette interface peut s'envisager de deux façons : soit chaque widget implémente l'interface directement, soit, surtout lorsque qu'il s'agit d'une liste de propriétés, chaque widget

peut se contenter de créer un champ qui, lui, est une instance d'une classe qui implémente déjà tout le contrat, ce qui simplifie grandement les choses.

La classe `WidgetInfo` peut être utilisée de cette façon. L'un des widget que nous verrons s'en sert de cette façon pour la démonstration.

Le code de `WidgetInfo` :

```
namespace Contract
{
    public class WidgetInfo : IWidgetBehavior
    {
        public WidgetArea Area { get; set; }
        public string WidgetName { get; set; }
        public bool IsSelected { get; set; }
        public bool IsActive { get; set; }
        public int Version { get; set; }
    }
}
```

Une simple implémentation de `IWidgetBehavior` donc.

WidgetArea

Puisque chaque widget indique l'endroit où il doit être affiché, cette information sera stockée sous la forme d'une valeur puisée d'une énumération :

```
namespace Contract
{
    public enum WidgetArea
    {
        TopArea,
        BottomArea,
    }
}
```

Deux possibilités : le conteneur du haut ou celui du bas.

IWidgetMetaData

Nous touchons au premier élément « purement » MEF : l'interface définissant les métadonnées fortement typées.

```
namespace Contract
{
    public interface IWidgetMetadata
    {
        WidgetArea Area { get; }
        string Name { get; }
        int Version { get; }
    }
}
```

Zone d'affichage (`Area`), nom « *user friendly* » du widget (`Name`) et version du widget. Les métadonnées de notre exemple restent très limitées.

WidgetExportAttribute

Second élément purement MEF du projet `Contract`, `WidgetExportAttribute` est l'attribut d'exportation personnalisé qui sera utilisé pour décorer les `UserControls` utilisés comme des modules.

```
namespace Contract
{
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class ExportWidgetAttribute : ExportAttribute
    {
        public ExportWidgetAttribute()
            : base(typeof(UserControl)) { }

        public WidgetArea Area { get; set; }

        public string Name { get; set; }

        public int Version { get; set; }

    }
}
```

L'attribut est une classe héritant de `ExportAttribute` et décorée d'attributs MEF.

`MetadataAttribute` indique que l'attribut d'exportation transporte aussi des métadonnées typées, cela n'est pas obligatoire mais semble plus judicieux. Dès lors qu'on décide de gérer des métadonnées MEF, il semble cohérent de les intégrer à l'attribut d'exportation ce qui simplifie leur déclaration.

`AttributeUsage` fixe l'utilisation qui sera faite de l'attribut notamment en limitant les éléments qu'il sera à même de décorer. Ici j'ai fixé `AttributeTargets.Class`, ce qui signifie que l'attribut ne pourra servir qu'à exporter des classes (et non des propriétés ou des assemblages par exemple). `AllowMultiple` est à false parce que cet attribut ne peut être utilisé qu'une seule fois par classe qu'il décore.

Le constructeur ne fait qu'appeler le constructeur hérité en passant le type de `UserControl`. C'est sous cette classe que seront exportés les widgets. Il pourrait s'agir d'une classe mère s'ils descendaient tous d'une telle classe, ou bien, le plus souvent, il s'agira de l'interface définie pour fixer le comportement des widget. Dans notre projet il serait ainsi possible de faire apparaitre tous les widgets sous la forme de l'interface `IWidgetBehavior` (qu'il faudrait agrémenter d'une propriété retournant l'instance du `UserControl` pour pouvoir l'ajouter à l'arbre visuel).

Le reste du code est formé des propriétés qui reprennent les valeurs des métadonnées.

Le Shell

C'est l'application centrale, celle qui va recenser et afficher les modules. C'est une application Silverlight classique. Lors de sa création j'ai validé la création du projet Web correspondant (`Shell.Web`) qui servira de *shell* lui aussi mais de *shell technique* (sans rapport avec MEF) en regroupant tous les XAP dans son répertoire `ClientBin` et qui contiendra le catalogue de module que nous verrons plus loin.

Le `Shell` suit le pattern MVVM. Il utilise pour cela MVVM Light et référence l'unité `GalaSoft.MvvmLight.SL4`. Le projet n'est pas créé à partir du modèle fourni avec MVVM Light. L'unité est référencée, c'est tout, nous verrons plus loin les endroits précis où MVVM Light est exploité.

De la même façon le Shell référence l'unité principale de MEF plus l'unité spécifique à l'initialisation de MEF. La copie locale est laissée à `true`, ce seront les seules copies nécessaires de ces unités. Les autres XAP utilisant MEF n'auront besoin que d'une référence sans copie locale.

Ces unités sont :

- `System.ComponentModel.Composition`
- `System.ComponentModel.Composition.Initialization`

Le Shell contient une `MainPage` et un `App.Xaml` comme tout projet Silverlight de base. Il possède aussi trois sous-répertoires :

- `Helpers`, des unités de code utilitaires
- `ViewModels`, les ViewModels (de `MainPage` et de `Widget1`)
- `Widgets`, qui contient les widgets du XAP principal (une seul, `Widget1`)

Le `Shell` référence aussi le projet `Contract`. Ici aussi la copie locale est à `true` mais tous les autres XAP utilisant `Contract` n'auront besoin que d'ajouter la référence (avec copie locale à `false`).

IDeploymentService

L'une des particularités de l'exemple en cours est de permettre le chargement de modules depuis des XAP secondaires stockés sur le serveur.

Comme je l'ai déjà expliqué, MEF dispose de plusieurs services de découverte mais ils utilisent tous des procédés non exploitables sous Silverlight comme la recherche dans des répertoires. En mode OOB et avec *elevated trust* une application Silverlight peut éventuellement se servir de ces méthodes (je ne l'ai pas testé), mais une application OOB c'est plus du WPF que du Silverlight et, en tout cas ce n'est pas le Silverlight pour le Web dont je parle dans cet article. Et dans ce cas de tels accès directs à des répertoires ne marchent pas.

Pour Silverlight (Web) Microsoft a ajouté un système d'initialisation de MEF particulier, celui utilisé dans la première série d'exemples de cet article, et qui se borne à recenser les modules dans le XAP en cours.

C'est très bien, mais pas suffisant pour charger des modules externes, ce qui est l'un des avantages de MEF.

Etonnamment MS ne fournit pas, au moins pour l'instant, de méthode intégrée à MEF pour charger des XAP externes facilement.

Il est donc nécessaire de développer sa propre solution.

MEF utilise la notion de catalogue, voire de catalogues agrégés, pour créer ses compositions. Lorsqu'on utilise MEF comme je l'ai fait depuis le début on n'a pas à se focaliser sur ce genre de choses. Mais pour écrire un chargeur de XAP qui s'intègre à MEF, il le faut… Heureusement le site CodePlex de MEF nous donne quelques voies

(voir http://mef.codeplex.com/wikipage?title=DeploymentCatalog) que nous pouvons améliorer.

Mais pour commencer il est préférable de définir une interface qui masquera l'implémentation qui pourra ainsi évoluer au fil du temps.

Cette interface c'est `IDeploymentService` :

```
namespace Shell
{
    public interface IDeploymentService
    {

        void AddXap(string relativeUri,
                    Action<AsyncCompletedEventArgs> completedAction);
        void RemoveXap(string uri);
    }
}
```

L'interface définit un contrat de deux méthodes, `AddXap` pour ajouter un XAP au catalogue de MEF en cours et `RemoveXap` qui permet d'enlever un XAP du catalogue.

On remarque que `AddXap` prend en paramètre une chaîne de caractères indiquant l'URI (relative) du XAP. Lorsque tous les XAP sont stockés ensemble dans le répertoire `ClientBin` de l'application serveur, il suffit de passer le nom du XAP (ex : « `extensions.xap` »). Pour `RemoveXap` il faut utiliser la même chaîne que celle passé à `AddXap`.

Enfin, on note la présence d'un second paramètre à la méthode `AddXap`. Il s'agit d'une action de type `Action<AsyncCompletedEventArgs>` qui sera appelée automatiquement par le service de chargement une fois le XAP chargé et les modules découverts et instanciés. La signature nous rappelle que l'opération est de type asynchrone, le chargement du XAP n'est donc pas bloquant.

La question qui peut venir à l'esprit maintenant est « *comment fait-on pour connaître à l'avance le nom du XAP à charger puisque justement nous sommes en train de créer un système modulaire qui peut avoir à découvrir de nouvelles sources ?* ».

C'est une bonne question, et la réponse est simple : de base cela n'est pas possible. Une autre réponse plus satisfaisante est : certes, mais nous allons voir plus loin

comment le faire en gérant un catalogue au format XML stocké sur le serveur.
Réponse plus rassurante ☺

DeploymentCatalogService

Cette classe implémente le service de chargement de XAP défini par l'interface `IDeploymentService`.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using GalaSoft.MvvmLight.Messaging;

namespace Shell
{
    [Export(typeof(IDeploymentService))]
    public class DeploymentCatalogService : IDeploymentService
    {
        private static AggregateCatalog aggregateCatalog;

        readonly Dictionary<string, DeploymentCatalog> _catalogs;

        public DeploymentCatalogService()
        {
            _catalogs = new Dictionary<string, DeploymentCatalog>();
        }

        public static void Initialize()
        {
            aggregateCatalog = new AggregateCatalog();
            aggregateCatalog.Catalogs.Add(new DeploymentCatalog());
            CompositionHost.Initialize(aggregateCatalog);
        }

        public void AddXap(string uri,
                Action<AsyncCompletedEventArgs> completedAction = null)
        {
            DeploymentCatalog catalog;
            if (!_catalogs.TryGetValue(uri, out catalog))
            {
                catalog = new DeploymentCatalog(uri);
                if (completedAction != null)
                    catalog.DownloadCompleted += (s, e) =>
completedAction(e);
                else
                    catalog.DownloadCompleted += catalog_DownloadCompleted;
```

```
                catalog.DownloadAsync();
                _catalogs[uri] = catalog;
            }
            aggregateCatalog.Catalogs.Add(catalog);
        }

        static void catalog_DownloadCompleted(object sender,
                                                AsyncCompletedEventArgs
e)
        {
            if (e.Error != null)
            {
                Messenger.Default.Send<NotificationMessage<Exception>>(
                      new
NotificationMessage<Exception>(e.Error,"Exception"));
                //throw new Exception(e.Error.Message, e.Error);
            }
        }

        public void RemoveXap(string uri)
        {
            DeploymentCatalog catalog;
            if (_catalogs.TryGetValue(uri, out catalog))
            {
                aggregateCatalog.Catalogs.Remove(catalog);
            }
        }
    }
}
```

Il n'est pas évident de commenter ce code sans être obligé d'entrer dans les détails de fonctionnement de MEF (ce pourquoi la documentation de MEF est faite). Mais, de très haut, disons que la classe gère un catalogue agrégé (un catalogue qui peut contenir d'autres catalogues). Elle utilise les fonctions de base de MEF (comme `DeploymentCatalog`) pour charger et analyser le XAP, considéré comme un catalogue, et ajouter ce dernier au catalogue agrégé si tout a fonctionné.

L'action passée en paramètre à `AddXap` est exécutée en fin de traitement, qu'il y ait erreur ou non. Si aucune action n'est fournie (on peut passer `null`) le code fournit sa

propre méthode de fin de traitement qui, en l'état, ne fait que détecter s'il y a eu erreur et, dans l'affirmative, lève une exception.

Dans cette version, lever une exception s'adapte assez mal à la situation. Quant à fournir à chaque appel de `AddXap` le code d'une action cela ne serait pas très pratique. J'ai choisi d'utiliser la messagerie de MVVM Light pour diffuser le message d'erreur.

Toute classe de l'application souhaitant être avertie qu'une erreur s'est produite peut ainsi s'abonner à ce message et le traiter comme elle le souhaite : pas d'action à écrire, aucun couplage avec aucun code existant, adaptabilité à tout code futur garantie.

Pour l'application `Shell`, c'est la vue principale `MainPage` qui s'occupe de récupérer le message et qui l'affiche tout simplement dans une boîte de dialogue, responsabilité conforme à MVVM. Une application réelle réagira de façon plus sophistiquée en gérant un log, en proposant une solution à l'utilisateur (ce qui est une bonne idée pour améliorer l'UX).

Dernier point sur `DeploymentCatalogService` : il faut l'initialiser. Cela s'effectue au plus tôt dans l'application, l'appel à la méthode `Intialize()` (qui est statique) est donc logiquement placée dans `App.Xaml` sur l'évènement `StartUp` avant l'assignation de `RootVisual` (qui pourrait fort bien être un module découvert par MEF) :

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            DeploymentCatalogService.Initialize();
            this.RootVisual = new MainPage();
        }
```

Pour terminer sur cette classe, vous aurez remarqué qu'elle est exportée (elle est décorée par un attribut `Export` de MEF). Cela signifie que MEF recensera la classe, en créera une instance qui pourra être importée ailleurs par d'autres modules qui auront alors accès aux fonctions de chargement d'un XAP externe. C'est ce qui sera fait, nous le verrons plus loin.

MainPage

Il s'agit de la page principale de l'application, le véritable « *shell* » visuel.

Son code XAML peut être étudié depuis le code source fourni. Voici le code-behind :

```
namespace Shell
{
    public partial class MainPage : UserControl,
IPartImportsSatisfiedNotification
    {
        public MainPage()
        {
            InitializeComponent();
            CompositionInitializer.SatisfyImports(this);
            DataContext = ViewModel;
            Messenger.Default.Register<NotificationMessage>(this,
                                            delegate { displayWidgets();
});

Messenger.Default.Register<NotificationMessage<Exception>>(this,
              (n) => {if (n.Notification == "Exception")
showException(n.Content); });
        }


        [Import]
        public MainPageViewModel ViewModel { get; set; }

        public void OnImportsSatisfied()
        {
            displayWidgets();
        }

        private void displayWidgets()
        {
            MainContainer.Children.Clear();
            SecondaryContainer.Children.Clear();
            Dispatcher.BeginInvoke(() =>
               { try
                   {
                     foreach (var widget in ViewModel.Widgets)
                      {
                        if (widget.Metadata.Area == WidgetArea.TopArea)

MainContainer.Children.Add(widget.Value);
                        else SecondaryContainer.Children.Add(widget.Value);
```

```
                    }
                }
            catch (Exception e)
                { showException(e); }
            });
        }

        /* exceptions */

        private void showException(Exception exception)
        { Dispatcher.BeginInvoke(() =>
          MessageBox.Show("Exception:" + Environment.NewLine +
exception.Message)); }
    }
}
```

La première chose à noter est l'importation du ViewModel. C'est MEF qui réalisera la connexion entre la Vue et son ViewModel. Nous n'utilisons donc pas le locator de MVVM Light pour cette opération. C'est un point essentiel dans la façon de « marier » MEF et un toolkit MVVM. Le découplage est assuré par MEF, inutile d'utiliser d'autres procédés.

Ensuite, on peut voir que le `DataContext` de la Vue est assignée dans le constructeur du code-behind. Il est initialisé en utilisant le ViewModel importé par MEF.

On notera aussi le support par la classe de l'interface `IPartImportsSatisfiedNotification` qui appartient à MEF. Elle définit un contrat très simple constitué d'une seule méthode `OnImportsStatisfied` qui est appelée lorsque les importations ont été satisfaites (qu'il y ait erreur ou non, ce qu'on peut savoir par l'argument de la méthode).

En supportant cette interface, notre Vue peut prendre en charge l'affichage des modules. La *recomposition* étant permise (grâce à `ImportMany(AllowRecomposition = true` dans le ViewModel), à chaque fois que de nouveaux modules seront ajoutés la méthode sera appelée, permettant à l'application de recréer l'interface visuelle en accord avec l'ensemble des modules découverts.

La Vue contient en outre du code non lié à MEF comme l'enregistrement dans son constructeur de deux notifications utilisant la messagerie de MVVM Light. La première notification demande à la vue de rafraîchir son affichage, la seconde est

plus générique : tout code provoquant une exception peut, au lieu de laisser l'exception suivre son cours, la stopper et la transmettre via une notification. La Vue principale tient ici son rôle centralisateur (celui de *shell*) et affichera ainsi les messages d'erreur peu importe leur provenance.

Dans une application réelle cette séquence sera améliorée, avec par exemple le log de l'erreur, voire l'envoi d'un message via un Service Web ou autre afin que le serveur puisse monitorer les erreurs des postes clients.

Les affichages sont insérés dans des appels `BeginInvoke` de la classe `Dispatcher` qui s'assure que ces modifications du visuel de l'application sont réalisées par le thread principal. En effet, certains évènements (traitement des exceptions, demande de rafraîchissement de l'affichage) peuvent être initiés par des threads secondaires qui n'ont pas le droit d'accéder à l'interface visuelle. Des tests supplémentaires pourraient être effectués pour savoir si l'appel à `BeginInvoke` est nécessaire ou non. Dans cet exemple c'est majoritairement le cas et le temps d'affichage n'est pas critique. Il est généralement plus efficace de tester la nécessité d'un appel à `BeginInvoke` et l'éviter s'il n'est pas nécessaire. Tout dépend du contexte et c'est donc à vous de décider en fonction de ce que fait votre application.

Terminons sur un aspect essentiel lié à MEF : La Vue principale, en tant que classe, n'est pas exportée. C'est le niveau le plus haut, le *shell*. Cette particularité lui permet d'appeler dans son constructeur `CompositionInitializer.SatisfyImports(this)`, ce qui déclenche la satisfaction des importations. Cet appel ne peut pas être effectué par une classe qui est elle-même exportée. Ce qui interdit par exemple de placer ce code fonctionnel dans le ViewModel (posant d'ailleurs un petit problème quand à l'application stricte de MVVM qui voudrait que cela soit le ViewModel de la `MainPage` qui se charge de cet appel, ce qu'il ne peut pas faire étant lui-même exporté).

Les Bindings de la MainPage

Sans publier la totalité du XAML de cette page, retenons le binding de la `Listbox` :

```
<ListBox ItemsSource="{Binding WidgetsInfo}" …
```

`WidgetsInfo` est une propriété de type collection du ViewModel. C'est la liste des informations de chaque module affiché.

Le code XAML comporte en outre la définition d'un `DataTemplate` permettant de mettre en forme ces données.

D'autres définitions et bindings concernent la `ComboBox` du profil utilisateur. Pour ne pas tout mélanger nous verrons cela plus loin.

MainPageViewModel

Il s'agit du ViewModel de la page principale. La classe est exportée par un attribut MEF `Export`, ce qui permet à la Vue de récupérer son ViewModel automatiquement comme nous l'avons vu plus haut.

L'importation et l'exportation du ViewModel est ici réalisée directement sur le type de ce dernier. Cela est discutable puisque la Vue est ainsi obligée de connaître la classe du ViewModel créant un couplage qu'on cherche généralement à éviter, de surcroît avec MVVM.

Dans la réalité, je n'ai jamais rencontré de situation où, en cours de développement d'une application ou même après, d'un seul coup on se met à écrire un nouveau ViewModel ayant un nom de classe différent et qu'on souhaite lié à une Vue existante. C'est une situation purement théorique. C'est en cela d'ailleurs que certaines applications strictes du pattern MVVM me paraissent très artificielles, voire néfastes. D'où, d'ailleurs, mon billet « Faut-il brûler la pattern MVVM ? » (http://www.e-naxos.com/Blog/post.aspx?id=64f3750f-dca5-487d-af3c-2e9d033e4b3f) que je vous invite à lire, le sujet débordant le cadre de cet article.

Je considère en effet qu'interdire à la Vue de connaître son ViewModel impose trop de contraintes au regard des faibles avantages purement théoriques de ce découplage. Une Vue est conçue pour un ViewModel et réciproquement, c'est la réalité. Rares sont les applications re-routant une Vue vers un autre ViewModel (ou réciproquement). Face à cette « interdiction » de MVVM qui apparaît purement artificielle, je préfère rester simple et conserver un code maintenable, quitte à violer la pattern (ce qui me fait penser que MVVM n'est pas un vrai pattern, mais simplement un principe intéressant qu'il ne faut surtout pas prendre au pied de lettre comme on le ferait avec une vraie pattern de type Gang Of Four).

Revenons au ViewModel de la `MainPage`… et concentrons-nous sur ses fonctions vitales :

```
    [Export]
    public class MainPageViewModel : ViewModelBase,
IPartImportsSatisfiedNotification
    {


        [ImportMany(AllowRecomposition = true)]
        public Lazy<UserControl, IWidgetMetadata>[] Widgets { get; set; }

        public IEnumerable<WidgetInfo> WidgetsInfo { get; set; }

        public void OnImportsSatisfied()
        {
            var l = new List<WidgetInfo>();
            foreach (var widget in Widgets)
            {
                var w = new WidgetInfo
                            {
                                    Area = widget.Metadata.Area,
                                    WidgetName = widget.Metadata.Name,
                                    IsSelected = false,
                                    IsActive = true,
                                    Version = widget.Metadata.Version
                            };

                l.Add(w);
            }
            WidgetsInfo = l.OrderBy(x => x.WidgetName).ToList();
            RaisePropertyChanged("WidgetsInfo");
            Messenger.Default.Send(new
NotificationMessage("UpdateDisplay"));
        }
…
```

Tout d'abord la classe est décorée par `Export`. C'est comme cela qu'elle peut être importée par la Vue. Comme mentionné plus haut, l'exportation et l'importation sont effectuées en se basant sur le type du ViewModel ce qui impose que la vue connaisse cette classe. Cela n'est pas si grave dans la réalité, mais est en désaccord avec l'orthodoxie MVVM. Comme expliqué plus haut, je l'assume.

Pour rester conforme au pattern MVVM il faudrait exporter les modules sous un nom de contrat de type Interface ou classe mère. Ici par exemple j'utilise `ViewModelBase` comme classe mère pour les ViewModel. Cette classe est issue de MVVM Light et offre quelques services comme le support de `INotifyPropertyChanged`. On pourrait fort bien exporter tous les ViewModel sous ce type (en ajoutant un nom de contrat MEF). On pourrait aussi définir une interface commune à tous les ViewModels, la faire supporter par chacun d'eux et les exporter sous le type de cette Interface. Dans ce cas la Vue récupèrerait un ViewModel typé faiblement (une interface ou une classe mère) et ne pourrait pas s'en servir autrement que pour initialiser son DataContext.

Cette façon de faire est un bon moyen de rester dans l'orthodoxie MVVM, qui de toute façon, fait que les interactions entre la Vue et son ViewModel sont basées uniquement le Databinding. La Vue ne doit normalement pas avoir, dans son code-behind, à appeler directement du code de son ViewModel.

Même si cette approche me semble inutilement rigide, c'est un compromis qui pourra satisfaire ceux qui veulent coller au plus près de MVVM.

D'autres librairies orientées MVVM utilisent la notion de bootstrap décrivant les associations entre Vue et ViewModel ou d'autres procédés de ce genre. Le choix étant assez large, chacun fera en fonction de ce qui se rapproche le mieux de sa façon de voir les choses et d'interpréter MVVM.

L'ensemble des widgets est importé par les lignes suivantes selon un principe déjà étudié dans cet article :

```
[ImportMany(AllowRecomposition = true)]
public Lazy<UserControl, IWidgetMetadata>[] Widgets { get; set; }
```

Le ViewModel propose aussi la propriété suivante :

```
public IEnumerable<WidgetInfo> WidgetsInfo { get; set; }
```

C'est elle qui est liée par binding à `ItemsSource` de la `Listbox` de la Vue qui affiche la liste des modules chargés.

Le ViewModel supporte aussi l'interface MEF `IPartImportsSatisfiedNotification`, ce qui lui permet de fabriquer la liste `WidgetsInfo` à chaque fois que de nouveaux modules sont ajoutés.

D'autres fonctions sont prises en compte par le ViewModel mais nous les couvrirons plus loin dans l'article.

Le Widget1

C'est le premier module de notre application. Le seul qui se trouve défini dans le XAP principal. Ce n'est pas une obligation, cela permet uniquement d'illustrer cette possibilité. Le XAP principal contient très souvent des modules élémentaires indispensables au fonctionnement du `Shell` comme un dialogue de paramétrage par exemple (qui lui-même peut intégrer des modules variables, chaque module ayant des paramètres venant « s'imbriquer » automatiquement dans le dialogue général des paramètres).

Les modules (ou widgets, parties…) lorsqu'ils sont visuels peuvent bien entendu suivre le pattern MVVM et disposer de leur propre ViewModel, lui aussi découvert par le jeu des exportations et importations. C'est le cas du `Widget1` dont l'unique commande est gérée par un ViewModel à titre de démonstration.

Visuel et binding

Visuellement, `Widget1` se présente comme cela (sous VS) :

Figure 64 - Widget1 - Visuel

Un fond bleu, un titre figé (qui n'est pas issu des métadonnées) et un bouton. C'est ce dernier le plus important puisqu'il va déclencher le chargement volontaire d'un catalogue d'extensions.

Nous y reviendrons plus loin.

Le bouton est lié par binding à une commande exposée par le ViewModel du widget (via le `DataContext` initialisé par la Vue, voir le code-behind à la section suivante) :

```
<Button Content="Load Extensions" Height="23" HorizontalAlignment="Center"
Name="button1" VerticalAlignment="Bottom" Width="100"
VerticalContentAlignment="Bottom" Margin="25,0,25,48"
Command="{Binding LoadExtensions}"/>
```

Le code-behind

Le code-behind du `UserControl` « `Widget1` » commence ainsi :

```
namespace Shell
{

    [ExportWidget(Area = WidgetArea.TopArea, Name = "Local Widget", Version
= 2)]
    public partial class Widget1 : UserControl, IWidgetBehavior,

INotifyPropertyChanged
    {
        public Widget1()
        {
            InitializeComponent();
        }

        [Import]
        public Widget1ViewModel ViewModel
        {
            get { return (Widget1ViewModel)DataContext; }
            set { DataContext = value; }
        }
…
```

On note la présence de notre attribut d'exportation personnalisé `ExportWidget` qui permet à la fois de marquer la classe comme étant exportée dans MEF et de fournir les métadonnées.

Le ViewModel est importé comme celui de la `MainPage`. Petite variante pour le `DataContext` : c'est le constructeur qui dans l'exemple précédent récupérait le ViewModel pour initialiser le `DataContext` alors qu'ici c'est la propriété `ViewModel` qui se charge de lire et d'écrire le DataContext.

Cette seconde approche est à utiliser partout. L'initialisation du `DataContext` n'est effectuée qu'une fois la propriété correctement importée par MEF (ce qui déclenche le setter) donc une fois la classe correctement construite. Dans le code de la `MainPage` on peut se demander comment la propriété `ViewModel` peut déjà être accessible dans le constructeur alors que pour l'initialiser MEF est bien obligé de créer l'instance (donc d'exécuter son constructeur). Qui de la poule ou de l'œuf ? C'est tout simple : la Vue principale n'est pas exportée… et c'est dans son constructeur qu'est appelée la méthode de composition de MEF. Tout naturellement, à la ligne suivant cet appel la propriété `ViewModel` est correctement initialisée… Cela n'est utilisable que dans la Vue principale, c'est pourquoi le `Widget1` montre la façon plus classique de procéder (et qu'on pourrait appliquer à la Vue Principale par souci d'homogénéité d'ailleurs).

Le problème de l'interface métier

Les widgets, comme nous l'avons vu, supportent l'interface métier `IWidgetBehavior`. Cette dernière ne sert pas à grand-chose dans l'exemple mais sa présence simule un cas réel : les plugins (ou widgets) sont le plus souvent pilotés via une interface. Et comme nous allons le voir, dans le contexte MEF + MVVM cela n'est pas forcément évident. D'où l'intérêt d'aborder ce sujet.

> *A noter qu'il faut comprendre ici « interface métier » au sens le plus large du terme. Selon ce qu'on mettra dans cette interface elle sera réellement une interface métier ou une simple interface. La nuance peut être importante car du code métier, avec MEF, aurait plutôt tendance à être dans des classes de services importées par les ViewModels par exemple. Vouloir placer des interfaces purement métier sur les widgets eux-mêmes est certainement un problème artificiel qui ne sert que les besoins de la démonstration. Dans la réalité l'architecture intègrera le code métier (et les interfaces correspondantes) dans une approche différente. Toutefois, même s'il ne s'agit pas de code métier « pur », le problème de piloter les widgets via des interfaces reste entier, ce qui justifie d'aborder la question.*

Dans notre exemple cette interface ne fixe que des propriétés mais il est évident que dans un cas réel elle devrait aussi proposer des méthodes permettant d'interagir avec les widgets. Cette interaction est très étendue et dépend du projet. Peut-être les widgets ont-ils besoin d'être synchronisés entre eux (ID d'une fiche client, article ?) ou bien doivent-ils être capables de retourner la liste des commandes qu'ils supportent pour qu'elles s'intègrent au menu du Shell, et bien d'autres choses encore…

Si le principe des interfaces et leur utilité n'ont pas besoin d'être présentés, ce qui nous interroge c'est de savoir « qui » va supporter cette interface ?

En effet, les widgets sont exportés en tant que `UserControl`. Ils sont importés par le `Shell` qui les affiche. Le `Shell` reçoit une énumération de widgets, soit directement des `UserControl` soit, comme dans cet exemple, des `Lazy<T,M>` qui possèdent une propriété `Value` retournant le `UserControl` ce qui revient au même au final.

Or, ces widgets sont des éléments d'interface visuelle. Il pourrait d'ailleurs s'agir de descendants de `Page` plutôt que de `UserControl` dans un système navigationnel. Ou tout autre type pour des services mais là le problème que je vais vous exposer ne se poserait pas.

Un module de service c'est une classe, du code. Sous MVVM ou non cela ne change rien. Sous MEF en revanche ce code sera plutôt placé dans des classes de services qui seront exportées pour être utilisées partout où cela est nécessaire.

Mais un module qui est affichable, un `UserControl` ou une `Page`, sous MVVM c'est une Vue. Et qui dit Vue dit deux choses : d'une part la Vue ne gère rien elle-même et d'autre part il existe un ViewModel qui se charge justement de cette gestion.

Si nous exportons bien les ViewModels qui sont ainsi importés par leurs Vues, et si nous exportons bien aussi les Vues (les `UserControl`) qui sont ainsi recensées et affichées par le `Shell`, ce dernier ne connait pas les ViewModels… Il pourrait les connaître au travers des Vues mais cela ne serait vraiment pas orthodoxe.

Si un module expose une interface (métier ou non), selon MVVM c'est le ViewModel qui devrait implémenter celle-ci, la Vue ne pouvant pas posséder de code métier ou fonctionnel, juste du code lié à l'affichage.

Toutefois, si nous suivons le pattern, le `Shell` n'aura pas accès à l'interface pour piloter les modules puisqu'il voit les Vues et non les ViewModels…

Et si nous implémentons l'interface au niveau de la Vue nous dérogeons à l'une des règles principales de MVVM.

Cornélien !

Dans une vision purement MVVM la solution consiste à implémenter l'interface au niveau des ViewModels et de baser tous les échanges entres les widgets et entre les widgets et le `Shell` sur des messages.

C'est une solution « propre » mais elle fait intervenir des messages asynchrones qui sont malgré tout très pénibles à gérer. Pour obtenir une solution plus utilisable il faudrait utiliser un Workflow **Jounce** qui offre un procédé très intéressant pour sérialiser les processus asynchrones (Pour plus d'information voir mon billet « Silverlight: Sérialiser les tâches asynchrones » - http://www.e-naxos.com/Blog/post.aspx?id=1429db2d-e248-4968-8356-403cdd11aede).

Une telle implémentation dépasserait le cadre de l'exemple en cours, et mélanger MVVM Light et Jounce ne serait pas judicieux, on opte pour l'un ou pour l'autre. Ici je n'aborderai pas la communication entre les widgets ou entre ces derniers et le shell. Mais c'est aussi ce qui fait la différence entre un article, même aussi long que peut-être le présent, et une véritable application qui demande des mois de travail !

Dans l'exemple en cours ce sont ainsi les Vues qui supportent l'interface « métier ». De fait l'interface est accessible au `Shell` puisqu'il possède la liste des modules créés par MEF et que ces modules sont les Vues. Cela permet de régler ponctuellement le problème mais il ne s'agit pas d'une solution digne d'un environnement de production, il faut en avoir conscience.

Ainsi, la vue doit gérer `INotifyPropertyChanged` pour signaler les modifications des valeurs des propriétés de l'interface, de même elle doit implémenter les propriétés, ce qui l'oblige à se comporter comme un ViewModel et qui n'est vraiment pas satisfaisant en dehors de cette démonstration. C'est pourquoi on trouve dans la Vue de `Widget1` le code suivant :

```
#region IWidgetBehavior Members

        private bool isActive;
        public bool IsActive
        {
            get { return isActive; }
            set
            {
                if (isActive == value) return;
                isActive = value;
                doPropertyChanged("IsActive");
            }
        }

        private bool isSelected;
        public bool IsSelected
        {
            get { return isSelected; }
            set
            {
                if (isSelected == value) return;
                isSelected = value;
                doPropertyChanged("IsSelected");
            }
        }

        private string widgetName;
        public string WidgetName
        {
            get { return widgetName; }
            set
            {
                if (widgetName == value) return;
                widgetName = value;
                doPropertyChanged("WidgetName");
            }
        }

        private WidgetArea area;
        public WidgetArea Area
        {
```

```
        get { return area; }
        set
        {
            if (area == value) return;
            area = value;
            doPropertyChanged("Area");
        }
    }


    private int version;
    public int Version
    {
        get { return version; }
        set
        {
            if (version == value) return;
            version = value;
            doPropertyChanged("Version");
        }
    }

    #endregion



    #region INotifyPropertyChanged Members

    private void doPropertyChanged(string property)
    {
        var p = PropertyChanged;
        if (p == null) return;
        p(this, new PropertyChangedEventArgs(property));
    }

    public event PropertyChangedEventHandler PropertyChanged;

    #endregion
```

Widget1ViewModel

Cette classe est le ViewModel du widget. La seule fonction exposée par le ViewModel est la prise en compte de la commande de chargement.

```
namespace Shell
{
    [Export]
    public class Widget1ViewModel : ViewModelBase
    {
        public Widget1ViewModel()
        {
            LoadExtensions = new RelayCommand(
              () => Service.AddXap("extensions.xap",
                                   args =>
                                       {
                                         if (args.Error != null)
                                         MessageBox.Show(
                                                   "Error loading
extensions"
                                                   + Environment.NewLine
                                                   + args.Error.Message);
                                       }));
        }

        [Import(typeof(IDeploymentService))]
        public IDeploymentService Service { get; set; }

        public ICommand LoadExtensions { get; private set; }

    }
}
```

La commande est exposée comme un `ICommand`, l'interface Silverlight gérée par les boutons. Toutefois la commande `LoadExtensions` est créée en partant de la classe `RelayCommand` de MVVM Light. Cette classe évite la création d'une classe implémentant `ICommand`. L'action à réaliser est enregistrée comme une expression Lambda (qui en contient une autre, la réponse à l'appel de `AddXap()`).

On remarque que le service de déploiement est importé (suivant son interface `IDeploymentService`), c'est par le biais de ce dernier que la commande va pouvoir ajouter l'extension « `extensions.xap` » via la méthode `AddXap()` du service.

La commande donne l'ordre au service de déploiement de charger le XAP en question et gère la réponse pour pister une éventuelle erreur. Si une exception est retournée par le service, la commande se contente de la transmettre à qui voudra bien la gérer, sous la forme d'un message MVVM Light.

On sait que le Vue principale du `Shell` s'est abonnée aux messages de notification de ce type et qu'elle affichera une boite de dialogue montrant le contenu de l'erreur. Mais cette connaissance n'est pas nécessaire.

> *A noter : lorsqu'on construit une application utilisant la messagerie MVVM il est intéressant de nommer toutes les notifications. Pour ce faire il est important de déclarer une énumération qui reprend le nom de chaque message. On utilise alors l'énumération (avec `ToString()`) comme texte de la notification. Le premier avantage est d'éviter qu'une faute de copie n'interdise le traitement du message, grâce à l'énumération le nom de la notification est toujours le même. Le second avantage est qu'il est plus facile de retrouver tous les morceaux de code qui font usage de tel ou tel message (un outil comme Resharper avec sa fonction 'Find Usage' est particulièrement utile dans un tel cas). Cela permet de vérifier que tous les messages émis sont bien reçus et traités au moins une fois par un récepteur.*

Le ViewModel n'est qu'un adaptateur pour la Vue, les traitements doivent être délégués au BOL ou à des services. C'est ce qui est fait ici pour le chargement du XAP.

Le service est importé via MEF, ce qui signifie que, quelque part, il existe une instance de ce service qui a été exportée… Revenez quelques pages en arrière, à la section abordant le `Shell`, et vous retrouvez l'interface `IDeploymentService` (page 326) ainsi que son implémentation qui est exportée.

Le chargement dynamique

Le mécanisme complet du chargement dynamique est le suivant :

1. `Widget1` est découvert et affiché par le `Shell`.
2. Ce module propose un bouton pour charger d'autres extensions. Pour ce faire il utilise le service de déploiement préalablement exporté.

3. Le service de déploiement va charger le XAP, l'analyser et ajouter au catalogue agrégé qu'il gère les nouveaux modules.
4. En supportant l'interface `IPartImportsSatisfiedNotification`, le ViewModel de la Vue principale est averti quand l'opération est terminée.
5. Il déclenche alors sa séquence de rafraîchissement de l'UI
6. L'utilisateur voit apparaître les nouveaux modules.

Le lecteur peut revoir la séquence visuelle simplifiée dans les captures d'écran de la section Le projet en action, page 310.

Les premières extensions

Les extensions chargées par le `widget1` se situent dans un projet à part, `Extensions`, Il s'agit d'une application Silverlight standard dont nous avons supprimé le fichier `App.Xaml` et la `MainPage`.

Ce projet ajoute deux références (avec copie locale = false) :

- `System.ComponentModel.Composition`
- Et `Contract`.

La première référence concerne MEF, la seconde notre projet regroupant les contrats, dont l'attribut d'exportation MEF personnalisé.

En dehors de cela le projet contient deux `UserControl`

- `Widget2`
- `Widget3`

`Widget2` se présente sous cette forme :

Figure 65 - Widget2 de Extensions.XAP

Une animation est lancée au chargement (l'étoile est animée et change de couleur)

`Widget3` se présente comme suit :

Figure 66 - Widget3 de Extensions.XAP

Il affiche l'heure (le `TextBlock` central) via un `DispatcherTimer` initialisé par la vue.

Ces deux widgets n'offrant qu'un comportement purement visuel il n'y a aucune raison de leur adjoindre un ViewModel. Vous n'en trouverez donc pas dans le projet `Extensions`.

Chaque Vue (chaque `UserControl`) est exporté de la façon suivante :

```
[ExportWidget(Area = WidgetArea.TopArea, Name = "Soleil", Version = 1)]
    public partial class Widget2 : UserControl, IWidgetBehavior,
INotifyPropertyChanged
```

et

```
[ExportWidget(Area = WidgetArea.BottomArea, Name = "Horloge", Version = 3)]
    public partial class Widget3 : UserControl, IWidgetBehavior,
INotifyPropertyChanged
```

On y retrouve le problématique support de l'interface `IWidgetBehavior`.

En dehors de ces quelques déclarations le projet `Extensions` ne contient rien d'autre. Il a été ajouté au projet `Shell.Web` (Propriétés / Applications Silverlight) afin que son XAP soit automatiquement placé dans le `ClientBin` de ce dernier (j'ai refusé en revanche l'ajout d'une page de test, ce qui ne sert pas à grand-chose ici).

Catalogue de modules

J'ai indiqué que cet exemple montrerait aussi comment gérer un catalogue de modules. Il est temps de passer à l'examen de ce problème.

Problème ? Oui, car en effet MEF ne sait pas gérer automatiquement le recensement des extensions sur un serveur depuis Silverlight. A cela une bonne raison : la sécurité très fermée de Silverlight ne l'autorise pas, que cela soit pour MEF ou qui que ce soit d'autre.

Dès lors, puisqu'il n'est pas possible de lire le contenu d'un répertoire distant, les outils de MEF permettant de cataloguer automatiquement les extensions sur disques ne sont plus utilisables.

On pourra regretter que Microsoft n'ait pas su faire une exception pour MEF, ce dernier faisant partie du Framework ce qui aurait tout simplifié. Mais créer des exceptions c'est aussi prendre le risque de créer des failles de sécurité…

Ainsi, nous nous retrouvons avec MEF, capable de découvrir des extensions, ce qui en fait en grande partie son intérêt, mais uniquement sous WPF. Sous Silverlight, de base, seuls les modules exportés et importés à l'intérieur du XAP principal sont gérés.

Nous avons vu que l'ajout d'un service de déploiement nous permettait de charger des XAP externes et de résoudre ce problème. Mais en partie seulement : les XAP chargés le sont nominativement, comme « `extensions.xap` ».

S'il faut connaître à l'avance le nom des XAP avant de les charger, il n'y a plus vraiment de « modularité » au sens plugin du terme.

Mais ne vous désespérez pas !

Tout comme nous avons su ajouter le service de déploiement pour charger des XAP externes, je vais vous montrer comment implémenter une solution tout à fait satisfaisante pour le problème de la découverte de nouveaux modules.

En réalité la solution est plutôt simple : Dans le répertoire `ClientBin` (ou l'un de ses sous-répertoire, à vous de voir) nous plaçons le fichier `Catalog.xml`.

Ce fichier se borne à énumérer tous les modules disponibles.

Comme Silverlight, avec le `WebClient`, permet facilement de télécharger un fichier qui provient de son propre serveur (ou d'un autre en suivant les règles d'un fichier de police à placer dans la racine du serveur), l'application peut récupérer le fichier XML et le traiter.

Lorsque les développeurs mettent à disposition un nouveau module il leur suffit d'ajouter une entrée dans le fichier XML situé sur le serveur. C'est une modification simple, légère et centralisée.

Le catalogue XML peut ainsi se borner à une simple liste de noms de fichiers XAP. Toutefois il est fréquent que les utilisateurs ne voient qu'une partie des modules. Par exemple en fonction de la licence qu'ils ont acquise ou bien en fonction d'un rôle (déterminé suivant le login). On peut penser à tout autre type de filtrage du même genre, tous reviennent à la même solution : il faut ajouter une ou plusieurs informations dans le catalogue pour que l'application puisse filtrer ce qu'elle peut (ou doit) charger.

Pour notre exemple j'ai choisi un filtrage sur le rôle de l'utilisateur. Comme je n'allais pas ajouter toute une gestion d'utilisateurs réelle pour cela, j'ai juste placé une `ComboBox` en bas à gauche du `Shell`. Par défaut le rôle est inconnu. Dès qu'on sélectionne un nouveau rôle (`Anonyme` ou `Administrateur`) le mécanisme de découverte, filtrage et chargement des nouveaux modules s'enclenche.

S'agissant d'une démo que j'ai voulu garder assez simple, le ménage n'est pas effectué lorsqu'on sélectionne un nouveau rôle. Les modules s'ajoutant aux précédents. Mais dans la réalité l'utilisateur ne pourrait pas choisir son rôle, c'est la gestion des utilisateurs, suite à un login valide, qui attribuerait un rôle fixe le temps de la session.

Je passerai sur la gestion de la `ComboBox`, ses deux bindings au ViewModel du `Shell` pour gérer à la fois la liste des rôles et le changement d'item sélectionné. Ce qui compte c'est ce qui se passe lorsqu'un rôle est sélectionné.

Chargement du catalogue

Lorsqu'on sélectionne un rôle dans la Vue, la propriété `CurrentUserRole` du ViewModel de la `MainPage` du `Shell` est mise à jour. Cette propriété est déclarée comme suit :

```
        private UserRole currentUserRole = UserRole.Unknown;
        public UserRole CurrentUserRole
        {
            get { return currentUserRole; }
            set
            {
                if (currentUserRole == value) return;
                currentUserRole = value;
                RaisePropertyChanged("CurrentUserRole");
                LoadModules(currentUserRole);
            }
        }
```

On note que le setter entraîne, outre la notification classique de changement de valeur, l'appel à `LoadModules(role)`.

C'est cette dernière qui va se charger de rapatrier le fichier XML, de l'analyser et de décider des modules à charger.

Pour utiliser le service de déploiement, le ViewModel déclare une importation de celui-ci :

```
        [Import(typeof(IDeploymentService))]
        public IDeploymentService Service { get; set; }
```

Vient ensuite le code de `LoadModules(role)` :

```
        private void LoadModules(UserRole userRole)
        {
            var wc = new WebClient();
            wc.OpenReadCompleted += (s, e) =>
            {
                if (e.Error!=null)
                {
                    Messenger.Default.Send(new
NotificationMessage<Exception>
                                                    (e.Error,"Exception"));
                    return;
                }
                var streamInfo = e.Result;

                var xElement = XElement.Load(streamInfo);

                var modulesList = from m in xElement.Elements("ModuleInfo")
                                  select m;
                if (!modulesList.Any()) return;
                foreach (var module in modulesList)
                {
                    var roleAttribute = module.Attribute("UserRole");
                    if (roleAttribute == null) continue;
                    if
(string.Compare(roleAttribute.Value,userRole.ToString(),

StringComparison.InvariantCultureIgnoreCase)!=0)
                        continue; // user role mismatch, no module.

                    var moduleAttribute = module.Attribute("XapFilename");
                    if (moduleAttribute != null)
                    {
                        var moduleName = moduleAttribute.Value;

                        Service.AddXap(moduleName,null);
                    }
                }
            };
            wc.OpenReadAsync(new Uri("Catalog.xml", UriKind.Relative));
        }
```

Le principe reste assez simple, décomposons-le :

Un `WebClient` est déclaré. Son évènement `OnReadCompleted` se voit assigner une expression Lambda. L'ouverture asynchrone du catalogue XML est déclenchée par `OpenReadAsync`.

Cet appel finira par invoquer l'expression Lambda définie précédemment.

Le gestionnaire de `OnReadCompleted` commence par vérifier qu'il n'y a pas d'erreur. S'il y en a une, elle est transmise via la messagerie MVVM Light selon un principe que j'ai déjà expliqué.

S'il n'y a pas d'erreur Linq to XML est utilisé par analyser le fichier catalogue reçu. Les tests sont rudimentaires, je ne compare ici que la stricte égalité entre le nom du rôle de l'utilisateur géré par l'application et celui indiqué dans le catalogue. Il est évident que dans la réalité cela serait un peu plus sophistiqué. Mais le principe resterait le même.

Ensuite c'est facile, si l'entrée du catalogue correspond au filtre, un appel est fait au service de déploiement pour qu'il charge le XAP dont le nom a été récupéré dans le catalogue.

Ce chargement entraînera la notification déjà gérée par le ViewModel du `Shell`, notification qui permet à ce dernier de rafraîchir l'affichage, et donc de présenter les nouveaux modules ainsi découverts…

Le fichier `Catalog.xml` ressemble à cela :

```
<ModulesInfos>
  <ModuleInfo XapFilename="Extensions2.xap" UserRole="Anonymous" />
  <ModuleInfo XapFilename="AdminExtensions.xap" UserRole="Admin"/>
</ModulesInfos>
```

Il est bien évident que vous pouvez le personnaliser à souhait selon les besoins de votre application. La seule information véritablement indispensable est le nom du fichier XAP.

Et d'où viennent ces deux autres XAP d'ailleurs ?

Les extensions supplémentaires

Pour montrer la fonction de découverte dynamique de modules via un catalogue XML encore fallait-il disposer d'extensions…

Il m'a donc fallu rajouter deux autres fichiers XAP à la solution. L'un chargé pour le rôle « `Anonyme` » et l'autre pour le rôle « `Administrateur` ».

J'ai repris la même « recette » que celle expliquée pour l'extension « `extensions.xap` ».

`Extensions2.XAP` ne contient qu'un widget, `Widget4`, qui se présente comme cela :

Figure 67 - Widget4

`AdminExtensions.XAP` ne contient aussi qu'un seul widget, `Widget5`, dont l'aspect est un peu plus travaillé pour faire illusion :

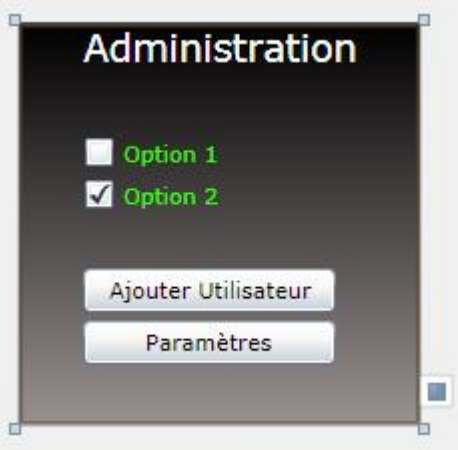


Figure 68 - Widget5

Aucune des commandes affichées n'est fonctionnelle, il s'agissait juste de donner un petit look « administration » à ce widget.

Il est évident que chaque XAP supplémentaire pourrait définir de nombreux widgets. Le catalogue XML pourrait aussi définir plusieurs XAP différents à charger pour un même rôle. Tout cela fait la différence entre une démonstration et une application réelle…

Conclusion

Il est temps de conclure. Arrivé ici et après toutes ces dizaines de pages j'ai l'impression de n'avoir qu'effleuré le sujet… Mais je pense aussi vous avoir donné les armes pour comprendre MEF et surtout pour l'utiliser dans de vraies applications en vous ayant aidé à résoudre certains problèmes délicats comme le chargement dynamique ou la découverte de nouveaux modules. D'autres problèmes sont soulevés et ne sont pas forcément traités, mais cela reflète la réalité de telles architectures d'une part, et celle d'un simple article qui doit bien s'arrêter quelque part…

Merci aux lecteurs qui auront atteint cette ligne de conclusion. Comme je le disais, ceux qui arriveront au bout vivants auront le droit à mes chaleureuses félicitations. Vous les méritez largement ☺

Validation des données sous Silverlight

Lorsqu'une nouvelle version de Silverlight sort on parle souvent des ajouts les plus visibles. Par exemple la `PathListBox` avait fait l'objet de nombreux tutors sur le Web mais personne ne s'en sert... C'est fun à regarder, mais presque inutile. Alors que la validation des données, c'est moins rigolo, mais c'est d'une absolue nécessité !

IDataErrorInfo

La validation des données est un élément important de toute application. Si vous avez écrit du code pour valider des données dans des applications Silverlight avant la sortie de la version 4, vous avez probablement remarqué qu'il n'existait pas de moyens simples de s'assurer que les données sont saisies correctement ni que les erreurs soient affichées en cohérence avec les contrôles de saisie.

La technique alors classique utilisée pour la validation des données consistait à lever des exceptions dans les blocs d'accesseur de propriété (`Set`) si les données étaient jugées non valides. Les contrôles de saisie liés à une propriété pouvaient ainsi être informés de l'exception de validation de données en définissant `ValidatesOnExceptions` à true dans le Binding.

Bien que cette technique fonctionne toujours et ne soit pas si mauvaise, il y a de nouvelles options disponibles depuis Silverlight 4 qui peuvent être utilisées sans avoir recours à la levée d'exceptions.

Et bizarrement, certainement par l'habitude prise avec la technique précédente, les nouvelles possibilités de Silverlight me semblent très peu utilisées, alors même que nous sommes passés à la version 5 depuis un petit moment.

Pourtant SL4 a introduit des moyens de validation plus sophistiqués. Une bonne raison de parler de `IDataErrorInfo` aujourd'hui !

(On notera que d'autres interfaces comme `INotifyDataErrorInfo` ont été rendues disponibles en même temps. Elles ne seront pas vues dans ce billet pour tenter de lui garder une taille de billet de blog !).

Voici la définition de `IDataErrorInfo` :

```
public interface IDataErrorInfo
{
    string this[string columnName] { get; }
    string Error { get; }
}
```

On ne peut pas dire que cela soit bien lourd ni compliqué à comprendre...

Les membres de IDataErrorInfo

L'interface `IDataErrorInfo` ne contient que deux membres, une propriété `Error` qui permet de retourner un message d'erreur global concernant l'objet validé et une propriété indexée par défaut qui peut être utilisée pour écrire la logique de validation pour les différentes propriétés au sein de l'objet.

Les interfaces sont toujours simples... par définition elles ne contiennent pas de code, ce qui implique que c'est au développeur de respecter le contrat en écrivant ce qu'il convient pour supporter ce dernier.

Implémenter IDataErrorInfo

On implémente généralement l'interface `IDataErrorInfo` directement sur une entité côté client (dans votre projet Silverlight donc) puisque le procédé de validation proposé n'est "visible" que si l'entité (ses propriétés) est liée en Binding à un contrôle Xaml.

Le squelette d'une telle implémentation sur une classe fictive "Personne" est le suivant :

```
public class Personne : INotifyPropertyChanged, IDataErrorInfo
{
    public string this[string propertyName]
    {
        get
        {
           ….
        }
    }

    public string Error
    {
        get { …. }
    }
}
```

Assez souvent la propriété Error n'est pas utilisée, il suffit donc de retourner systématiquement null :

```
public string Error
{
    get { return null; }
}
```

Mais on peut bien entendu s'en servir pour retourner une information sur l'état global de l'entité validée :

```
string _Errors;
const string _ErrorsText = "Les données de la Personne sont invalides.";

public string Error
{
    get { return _Errors; }
}
```

La majeure partie du travail de validation est effectuée dans la propriété indexée par défaut de l'interface.

Elle est appelée par les contrôles liés aux propriétés de l'objet (par Binding) pour voir si une propriété liée particulière est valide ou non.

La classe `Personne` fictive ci-dessus possède deux propriétés fondamentales que sont le nom et l'âge qui doivent être validés. Cela peut se faire dans la propriété indexée ajoutée à la classe suite au support de l'interface `IDataErrorInfo`, comme le montre l'exemple qui suit :

```
public string this[string propertyName]
{
    get
    {
        _Errors = null;
        switch(propertyName)
        {
            case "Nom":
                if (string.IsNullOrEmpty(Nom))
                {
                    _Errors = _ErrorsText;
                    return "Le nom ne peut être vide !";
                }
                break;
            case "Age":
                if (Age < 1)
                {
                    _Errors = _ErrorsText;
                    return "L'age doit être plus grand que zéro !";
                }
                break;
        }
        return null;
    }
}
```

Ici, les opérations de validation effectuées sont très simples. Dans la réalité les tests peuvent être aussi sophistiqués que nécessaire.

La clé du procédé consiste à utiliser un Switch sur le nom de la propriété pour valider chaque propriété de l'entité. On retourne null s'il n'y a aucun problème, ou bien une chaine précisant l'erreur dans le cas contraire.

Je préconise toujours la création de méthodes privées pour valider chaque propriété. Le Switch du getter de la propriété indexée ne doit faire qu'appeler ces méthodes et ne pas s'emmêler les pinceaux dans du code de validation proprement dit. Dans l'exemple ci-dessus cela passe encore, dans du code réel, cela devient très vite du code spaghetti !

On peut même se passer du Switch dans le getter, c'est ma préférence. Le getter ne fait qu'appeler une méthode qui elle contient le Switch et qui appelle elle-même les méthodes de validation de chaque propriété. Un code bien architecturé est toujours, un jour ou l'autre, un choix dont on se félicite !

Retour visuel sous Xaml

Valider c'est bien... Ne pas lever d'exception pour le faire c'est encore mieux. Mais encore faut-il que cela puisse se voir côté client donc en Xaml !

C'est dans le Binding des propriétés liées à l'entité que se jouera la détection de l'erreur via IDataErrorInfo. Prenons le cas d'une TextBox liée à la propriété Nom de la Personne, on écrira un code comme le suivant :

```
<TextBox Text="{Binding Nom,Mode=TwoWay,ValidatesOnDataErrors=true}"  />
```

La capture suivante montre l'effet de la validation sur l'objet `TextBox` :

Figure 69 - Valider des données

Faiblesse

L'inconvénient de l'interface `IDataErrorInfo`, c'est qu'elle ne fournit pas un moyen d'effectuer la validation de façon asynchrone. Dans certaines situations vous devrez peut-être vous connecter à un serveur pour vérifier qu'un ID utilisateur ou qu'une adresse mail sont uniques au sein du système, ou qu'un numéro de compte de plan comptable correspond bien à un code existant stocké dans la base de données. IDataErrorInfo ne supporte pas ce type de scénario directement.

Dans un prochain billet je vous parlerai de l'interface `INotifyDataErrorInfo` qui prend en charge la validation de données asynchrone...

Conclusion

Valider les entités par le simple support de `IDataErrorInfo` est un procédé que je qualifierai de “non violent” comparativement à la levée d’une exception. Et je préfère le code pacifié que le code warrior qui pète dans tous les sens avec des exceptions.

Toutefois `IDataErrorInfo` n’est pas une panacée.

D’abord elle ne gère pas, comme je l’ai dit plus haut, les cas de validations asynchrones qui sont de plus en plus souvent nécessaires dans les applications LOB.

Ensuite elle pose un problème de choix. Car `IDataErrorInfo` n’a absolument aucun effet en elle-même... Ce n’est que lorsqu’une propriété est liée par un Binding (bien programmé de plus) que l’effet de la validation se voit.

Or, quand on implémente une classe d’entité, peut-on toujours savoir à l’avance si la classe sera utilisée uniquement dans un Binding Xaml ou bien si elle risque d’être utilisée en interne par du code '”normal” ? Dans ce dernier cas seules les exceptions peuvent garantir une véritable validation.

De ce fait, `IDataErrorInfo` ne peut concerner que des entités vouées au seul Binding avec une UI Xaml. Et finalement ce genre d’entité est très rare...

Mais la gestion de cette interface en Xaml est pratique, simple, et le retour visuel peut être templaté pour l’améliorer.

Dilemme...

C’est pourquoi je préconise dans un code réel d’implémenter à la fois `IDataErrorInfo` et une levée d’exception. Une propriété statique de la classe de

l'entité permet de dire si on veut utiliser l'un ou l'autre des procédés. On peut compliquer encore un peu en rendant ce choix propre à chaque instance. Une instance Bindée en Xaml utilisera le procédé de l'interface, et une instance de la même classe utilisée dans du code préfèrera activer la levée d'exception.

Je pense que la popularité de `IDataErrorInfo` n'a jamais été très grande à cause de cette ambigüité : elle sert à valider des objets, mais seulement vis à vis de Xaml, utilisez la même entité par code et aucune validation ne semble être faite. Pour que cela fonctionne il faut, comme je l'explique ci-dessus mettre en œuvre un code autrement plus complexe. Et les gens n'aiment pas écrire trop de code, et ils ont raison, seuls les paresseux sont de bons développeurs... les fous du clavier sont des dangers à éliminer de son équipe (ou à transférer dans l'équipe de testing !).

On notera aussi que `IDataErrorInfo` impose le support d'une propriété indexée par défaut. C# ne gérant pas les indexeurs nommés comme le faisait Delphi son grand frère (et c'est vraiment dommage) cela peut poser de sérieux problèmes si le code des entités utilise déjà un indexeur par défaut...

Fausse bonne idée que `IDataErrorInfo` ? Peut-être. Ou pas. A vous de voir selon le contexte !

Silverlight 5 : exemples de code des nouveautés

Silverlight 5 propose des nouveautés intéressantes, en voici quelques-unes extraites d'une communication de Microsoft pouvant être rendue publique.

Ancestor RelativeSource Binding

Ancestor RelativeSource Binding enables a DataTemplate to bind to a property on the control that contains it, equivalent to FindAncestor and AncestorType, AncestorLevel bindings in WPF.

```
<UserControl x:Class="WpfApplication1.UserControl1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ContentControl Tag="SomeValue">
        <HeaderdContentControl>
            <HeaderedContentControl.Header>
                <TextBlock Text="{Binding Tag, RelativeSource=
{RelativeSource,AncestorType=ContentControl, AncestorLevel=2}}" />
            </HeaderedContentControl.Header>
            <Button>Click Me!</Button>
        </HeaderdContentControl>
    </ContentControl>
</UserControl>
```

Implicit DataTemplates

This feature enables the following capabilities:

ContentPresenter DataTemplates can be selected based upon the content type.

Implicit definition of DataTemplates

Dynamically update the ContentPresenter DataTemplate when its content changes

Enable proper scoping of DataTemplates

```
<DataTemplate DataType="local:Book">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="60" />
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="20" />
                </Grid.ColumnDefinitions>
                <Rectangle Fill="Transparent" Grid.Column="2"/>
                <Image Source="{Binding ImageSource}"
HorizontalAlignment="Left" Grid.RowSpan="2" Width="60"/>
                <TextBlock Text="{Binding Title}" Grid.Row="0"
Grid.Column="1" />
                <Grid Grid.Row="1" Grid.Column="1">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="{Binding Author, StringFormat='by
{0}'}"/>
                    <Button Content="Edit" Margin="4,0" Grid.Column="2"
  Height="26" VerticalAlignment="Top"
                        Style="{StaticResource StandardButtonStyle}"
                        Command="{Binding DataContext.EditBookCommand,
                        RelativeSource={RelativeSource FindAncestor,
AncestorType=UserControl}}"
                        IsEnabled="{Binding IsSelected,
                             RelativeSource={RelativeSource FindAncestor,
AncestorType=ListBoxItem}}"
                        />
                </Grid>
            </Grid>
        </DataTemplate>

        <DataTemplate DataType="local:ClearanceBook">
            <Grid>
                <Grid.RowDefinitions>
```

```
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="60" />
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="20" />
                </Grid.ColumnDefinitions>
                <Rectangle Fill="Red" Grid.Column="2"/>
                <Image Source="{Binding ImageSource}"
HorizontalAlignment="Left" Grid.RowSpan="2" Width="60"/>
                <TextBlock Text="{Binding Title}" Grid.Row="0"
Grid.Column="1" />
                <Grid Grid.Row="2" Grid.Column="1">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="{Binding Author, StringFormat='by
{0}'}"/>
                    <ComboBox VerticalAlignment="Top" Margin="4,0"
   Width="90" Grid.Column="1"
   SelectedItem="{Binding Deal, Mode=TwoWay}"
                        ItemTemplate="{StaticResource DealItemTemplate}"
   IsEnabled="{Binding IsSelected,
       RelativeSource={RelativeSource FindAncestor,
AncestorType=ListBoxItem}}"
                            ItemsSource="{Binding DataContext.Deals,
                                  RelativeSource={RelativeSource
Mode=FindAncestor,
                                      AncestorType=UserControl}}">
                    </ComboBox>
                    <Button Content="Edit" Margin="4,0" Grid.Column="2"
   Height="26" VerticalAlignment="Top"
                        Style="{StaticResource StandardButtonStyle}"
                        Command="{Binding DataContext.EditBookCommand,
                               RelativeSource={RelativeSource
FindAncestor,
AncestorType=UserControl}}"
                         IsEnabled="{Binding IsSelected,
```

```
                                RelativeSource={RelativeSource
FindAncestor,
AncestorType=ListBoxItem}}"
                        />
                </Grid>
            </Grid>
        </DataTemplate>

        <DataTemplate DataType="local:BestSellerBook">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="60" />
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="20" />
                </Grid.ColumnDefinitions>
                <Rectangle Fill="Yellow" Grid.Column="2"/>
                <Image Source="{Binding ImageSource}"
HorizontalAlignment="Left"
Grid.RowSpan="2" Width="60"/>
                <TextBlock Text="{Binding Title}" Grid.Row="0"
Grid.Column="1" />
                <Grid Grid.Row="1" Grid.Column="1">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="{Binding Author, StringFormat='by
{0}'}"/>
                    <TextBlock Text="   BEST SELLER" FontWeight="Bold"
Grid.Column="1"/>
                    <Button Content="Edit" Margin="4,0" Grid.Column="2"
Height="26"
   VerticalAlignment="Top"
                         Style="{StaticResource StandardButtonStyle}"
                         Command="{Binding DataContext.EditBookCommand,
                             RelativeSource={RelativeSource FindAncestor,
```

```
AncestorType=UserControl}}"
                              IsEnabled="{Binding IsSelected,
                                     RelativeSource={RelativeSource
FindAncestor,
AncestorType=ListBoxItem}}"
                          />
                </Grid>
            </Grid>
        </DataTemplate>
```

ClickCount

Enables Multi-click input on left and right mouse button.

```
private void OnMouseDownClickCount(object sender, MouseButtonEventArgs e)
{
    // Checks the number of clicks.
    if (e.ClickCount == 1)
    {
        // Single Click occurred.
    }

    if (e.ClickCount == 2)
    {
        // Double Click occurred.
    }

    if (e.ClickCount >= 3)
    {
        // Triple Click occurred.
    }
}
```

Binding on Style Setter

Binding in style setters allows bindings to be used within styles to reference other properties.

```
<Style x:Key="TextBlockStyle2" TargetType="TextBlock">
    <Setter Property="FontFamily"
        Value="/SL5;Component/Fonts/Fonts/.zip#Segoe UI/>
    <Setter Property="FontSize" Value="0,3,0,0"/>
        <Setter Property="Foreground" Value=
"{Binding Source={StaticResource SysColors},Path=ControltextBrush}"/>
</Style>
```

Trusted applications in-browser

Silverlight 5 enables trusted applications to run in-browser for the first time. The Beta includes support for two features: Elevated In-Browser via Group Policy & In-Browser HTML support w/Group Policy.

```
<Deployment.OutOfBrowserSettings>
    ::
    <OutOfBrowserSettings.SecuritySettings>
        <SecuritySettings ElevatedPermissions="Required" />
    <OutOfBrowserSettings.SecuritySettings>
    ::
</Deployment.OutOfBrowserSettings>
```

Additional information

The Xap must be signed and interest must be present in the Trusted Publishers Certificate store.

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Silverlight\ : add the following DWORD key "AllowElevatedTrustAppsInBrowser"` with 0x00000001.

AllowInstallOfElevatedTrustApps

With respect to trusted in-browser apps, disabling the AllowInstallOfElevatedApps regkey will disable install trusted apps but not prevent enabled apps to consume trusted functionality in-browser. Specifically, if an app has been enabled to run trusted in-browser but the AllowInstallOfElevatedTrustApps has been disabled, then the app will run as a trusted app in browser but not be installable.

AllowLaunchOfElevatedTrustApps

With respect to trusted in-browser apps, disabling the AllowLaunchOfElevatedApps regkey will disable execution trust feature consumption of any application (in-browser or OOB). Specifically, if an app has been enabled to run trusted in-browser

but the AllowLaunchOfElevatedTrustApps has been disabled, then the app will run as a partial trust app in browser.

Realtime Sound (low-latency Audio)

Enables pre-loading of an audio source for precision timing of playback. Multiple playback instances are supported through creation of SoundEffectInstance objects.

```
using Microsoft.Xna.Framework.Audio;
...


var streamInfo = Application.GetResourceStream(new Uri("Windows_44100_16_Mono.wav",
UriKind.RelativeOrAbsolute));
SoundEffect soundEffect = SoundEffect.FromStream(streamInfo.Stream);
soundEffect.Play(); // Default playback mode
soundEffect.Play(0.8f, -0.5f, 1.0f); // Explicit volume, pitch, pan

// Using explicitly created SoundEffectInstance
{
      SoundEffectInstance inst = soundEffect.CreateInstance();
      inst.Volume = 0.85f;
      inst.Play();
}
```

Variable Speed Playback (“Trick Play”)

This API enables development of fundamental player experiences such as fast-forward and rewind and variable speed playback at common speeds (1.2x, 1.4x) – a very common scenario for enterprise training content. The MediaElement.Rate property supports the values -8.0, -4.0, 0.5, 1, 4.0, 8.0. Note : AudioPitch correction is not present in the Beta but will be added for the final release.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
      media.PlaybackRate = 2.0;
}
```

Known issue

Setting the PlaybackRate property in XAML resets the value to 1.0 before the media is opened.

Linked Text Containers

Enables RichTextBox controls to be associated and text to overflow from one into the next. multiple RichTextBoxOverflows can be chained (from an initial RichTextBox) to automatically spread text across a pre-defined free-form layout or multiple column layout.

```
    <StackPanel Width="400">
        <RichTextBox x:Name="MasterRTB" Width="50" Height="50"
OverflowContentTarget={Binding ElementName="overflowContainer"}>
            <Paragraph>
                There is noooooooooo way all of this content is going to fit into a RTB
that is 50x50!
            </Paragraph>
        </RichTextBox>
        </RichTextBoxOverflow x:Name="overflowContainer">
    </StackPanel>
```

Text Tracking & Leading

Tracking/leading set more precisely how far apart each character is for extra creative control, Adding finer grained control over the spacing between letters in words in the RichTextbox control.

```
    <TextBlock FontSize="12" CharacterSpacing="300">
The intercharacter spacing of this text has been loosened.
    </TextBlock>
```

Custom Markup Extensions

Custom markup extensions allow you to run custom code from XAML. Markup extensions allow code to be run at XAML parse time for both properties and event handlers, enabling cutting-edge MVVM support.

First, put this namespace in your XAML

xmlns:local="clr-namespace:CMEInvokeMethodsSL"

Next add some Xaml

```
<Grid x:Name="LayoutRoot">
        <Canvas>
            <TextBox Text="Some Text"
                     GotFocus="{local:MethodInvoke Path=TextChanged}"
                     Width="102" Canvas.Left="35" Canvas.Top="42" />
            <ComboBox Height="34" Name="cbOptions"
                      SelectionChanged="{local:MethodInvoke
Path=OptionSelected}"
                      Width="120" Canvas.Left="143" Canvas.Top="42">
                <ComboBoxItem Content="A" />
                <ComboBoxItem Content="B" />
                <ComboBoxItem Content="C" />
                <ComboBoxItem Content="D" />
            </ComboBox>
            <Button Name="InvokeButton"
                    Click="{local:MethodInvoke Path=SaveData}"
                    Content="Invoke Button"  Margin="128,108,119,115"
Canvas.Left="141" Canvas.Top="-64" />
        </Canvas>
    </Grid>
```

Add the following class to your code. This implements a custom markup extension using the IMarkupExtension interface.

```
using System;
using System.Reflection;
using System.Windows;
using System.Windows.Markup;
using System.Xaml;

namespace BookShelf
{
    public class MethodInvokeExtension : IMarkupExtension<object>
    {
        public string Method { get; set; }

        private object _target;
        private MethodInfo _targetMethod;

        public object ProvideValue(IServiceProvider serviceProvider)
        {
            try
            {
                Delegate del = null;

                // IProvideValueTarget - Provides information about the
target object
                IProvideValueTarget targetProvider =
                serviceProvider.GetService(typeof(IProvideValueTarget))
            as IProvideValueTarget;
                // IXamlTypeResolver - Handles types that use xml prefixes
                IXamlTypeResolver xamlResolver =
            serviceProvider.GetService(typeof(IXamlTypeResolver))
            as IXamlTypeResolver;
                // IRootObjectProvider -
           //Provides access to the root object (the Framework Element)
                IRootObjectProvider rootObject =
            serviceProvider.GetService(typeof(IRootObjectProvider))
            as IRootObjectProvider;

                if (targetProvider != null)
                {
                    // Get the target element (ex: ListBox)
                    var targetObject = targetProvider.TargetObject;
                    // Get the target element's event info
```

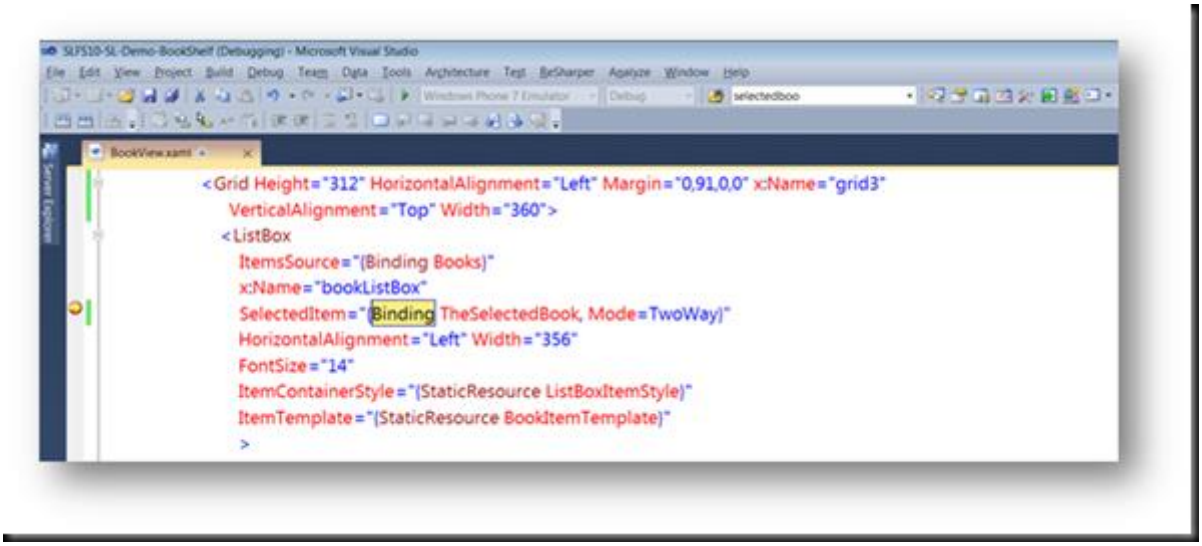
```
                    EventInfo targetEvent = targetProvider.TargetProperty
as EventInfo;

                    if (targetEvent != null)
                    {
                        // Get the event's parameters and types for the
target element
                        ParameterInfo[] pars =
targetEvent.GetAddMethod().GetParameters();
                        Type delegateType = pars[0].ParameterType;
                        MethodInfo invoke =
delegateType.GetMethod("Invoke");
                        pars = invoke.GetParameters();

                        // Create the function call (to invoke the event
handler)
                        Type customType = typeof(MethodInvokeExtension);
                        var nonGenericMethod =
customType.GetMethod("PrivateHandlerGeneric");
                      MethodInfo genericMethod =
            nonGenericMethod.MakeGenericMethod(pars[1].ParameterType);

                        // Grab the root FrameworkElement
                        if (rootObject != null)
                        {
                            FrameworkElement rootObjFE =
rootObject.RootObject
                as FrameworkElement;

                            if (rootObjFE != null)
                            {
                                // Grab the FrameworkElement's DataContext
                //and the method name exposed by it
                                _target = rootObjFE.DataContext;
                                _targetMethod =
_target.GetType().GetMethod(Method);
                                // Make sure the FE's DataContext has the
Method name
                //or get out.
                                if (_target == null) return null;
                                if (_targetMethod == null) return null;
```

```
                                // Create the event handler and attach it
from
                //the target element to the DataContext's method name
                                del = Delegate.CreateDelegate(
                delegateType, this, genericMethod);
                            }
                        }
                    }
                }

                return del;
            }
            catch (Exception ex)
            {
                string innerex = ex.InnerException.ToString();
                return null;
            }
        }

        // Invoke the generic handler
        public void PrivateHandlerGeneric<T>(object sender, T e)
        {
            _targetMethod.Invoke(_target, null);
        }
    }
}
```

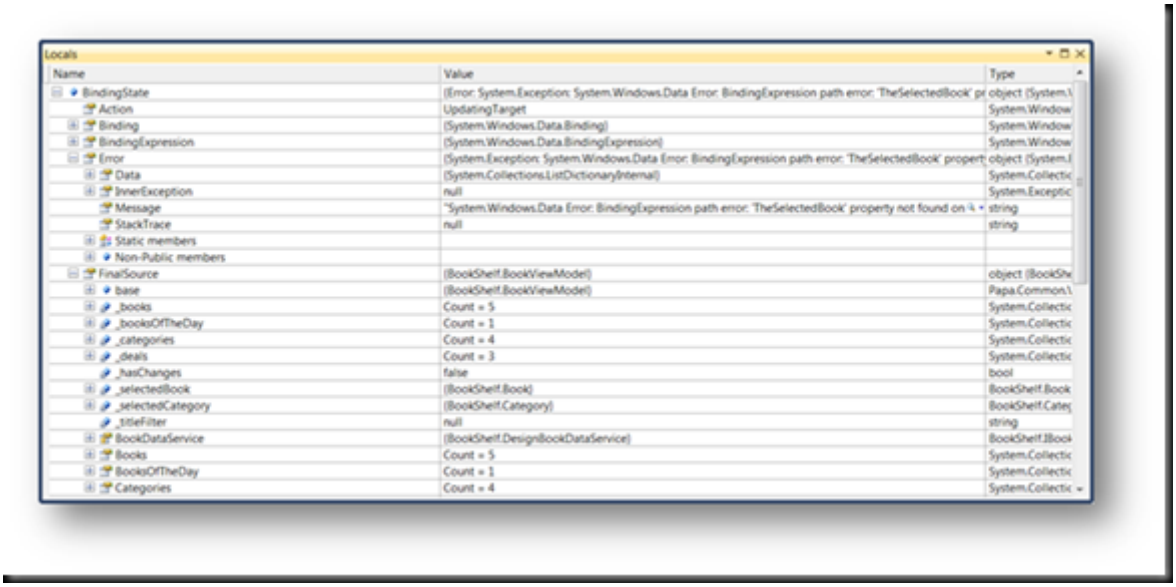
XAML Binding Debugging

Data binding debugging allows breakpoints to be set in XAML binding expressions so developers can step through binding issues, examine the Locals window, and create conditional breakpoints.

Here you can see Visual Studio stopped due to a binding error:



And the Locals window to assist with debugging the binding:



3D Graphics API

Silverlight 5 now has a built-in XNA 3D graphics API to enables you to build rich, GPU accelerated visualizations and rich user experiences. This includes a new 3D drawing surface control as well as immediate mode GPU access, shaders and effects for 2D and 3D use.

Additional Silverlight 5 Features Included in this Beta

- Hardware video decode for H.264 playback.
- Multi-core background JIT support for improved startup performance.
- ComboBox type ahead with text searching.
- Full keyboard support in full-screen for trusted in-browser applications, enables richer kiosk and media viewing applications in-browser.
- Default filename in SaveFileDialog – Specify a default filename when you launch the SaveFileDialog.
- Unrestricted filesystem access – trusted applications can Read write to files in any directory on the filesystem.
- Improved Graphics stack – The graphics stack has been re-architected to bring over improvements from WP7, such as Independent Animations.
- Performance optimizations –
- XAML Parser performance optimizations.
- Network Latency optimizations.
- Text layout performance improvements.

- Hardware acceleration is enabled in windowless mode with Internet Explorer 9.

Silverlight 5 Features not included in this Beta

The Beta is a step towards the final release of Silverlight 5. The full feature set planned for Silverlight 5 was announced at the Firestarter event in December 2011 and also includes the following features not shipped in this Beta:

- Support for Postscript vector printing enables users to create reports and documents, including the ability to create a virtual print view different from what is shown on the screen.
- Improved power awareness prevents the screen saver from being shown while watching video and allows the computer to sleep when video is not active.
- Remote control support, allowing users to control media playback
- DRM advancements that allow seamless switching between DRM media sources.
- Enhanced OpenType support.
- Fluid user interface enables smoother animation within the UI. Inter-Layout Transitions allow developers to specify animations to apply when elements are added, removed or re-ordered within a layout. This provides smoother user experiences when, for example, items are inserted into a list.
- Text clarity is improved with Pixel Snapping.
- The DataContextChanged event is being introduced.
- WS-Trust support: Security Assertion Markup Language authentication token.
- 64 bit support
- COM Interop for trusted applications running in-browser.
- Calling unmanaged code using P/Invoke from trusted applications in and out of browser.
- The Pivot viewer control enables a visual way to rapidly sort through large collections of graphical data, for example a catalog of movies represented by images of dvd covers, using intuitive transitions to show the effects of filters on the result set.

Legal Notice

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Microsoft, Expression, Silverlight, the Silverlight logo, Windows Server, the Windows logo, Windows Media, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Conclusion

J'ai laissé le bla-bla légal à la fin car il faut toujours garder en tête que SL 5 est encore en bêta et que bien entendu tout ce qui est dit ici peut changer d'ici la sortie de la version finale.

Silverlight 5 : Les nouveautés

Silverlight 5 est disponible depuis dimanche dernier. Cette version est très particulière à plus d'un titre dans le contexte de doute qui avait envahi la communauté SL depuis

quelques mois. Ainsi, loin d'être mort, Silverlight démontre encore une fois sa puissance au travers d'évolutions qui n'ont pas été implémentées uniquement pour le plaisir de créer une nouvelle release...

Des innovations majeures

La version 5 de Silverlight n'est pas juste une release de plus qui aurait été bâclée pour sortir à tout prix une dernière version. Nous allons le voir, les nouveautés ont demandé beaucoup de travail et un tel investissement s'inscrit forcément dans la durée. Il aurait été bien plus simple pour Microsoft de n'implémenter que quelques artifices de ci de là et de stopper les features les plus couteuses.

Or ce n'est pas le cas. La sortie même de SL 5, prévue en Novembre a été décalée légèrement sur Décembre pour permettre la fourniture d'une release parfaitement testée et fonctionnelle.

Certaines nouvelles features démontrent certes un recentrage sur Windows, mais SL 5 fonctionne toujours sur Mac. Et tous les browsers supportés jusqu'à maintenant le sont toujours. Cela balaye les bruits qui ont couru sur une version qui ne marcherait que sur IE par exemple. Mais en même temps certaines fonctionnalités ne visent que Windows, c'est vrai. Le rêve du "plugin universel et portable everywhere" se transforme petit à petit en une solution qui donne l'avantage à Microsoft et à son OS, j'en ai largement parlé dans de précédents billets.

Mais après tout Apple fait-il le moindre effort pour vendre OSX sur les PC ? Nenni. Google fait-il le moindre effort pour faire tourner les applications Android sur PC ou sur Mac ou même IPad ? Nenni.

Il serait donc un peu "gonflé" de déduire du recentrage de Silverlight sur Windows quoi que ce soit d'autre qu'une simple réponse "du berger à la bergère", d'autant que ce produit tourne toujours sur Mac... Microsoft fait toujours et encore exception sur le marché, offrant une portabilité qu'aucun autre éditeur d'importance n'offre aujourd'hui et en tout cas surtout pas Apple ni Google.

Il est donc sage de remettre tous les débats récents sur Silverlight dans un contexte plus réaliste et de constater que Silverlight est bien vivant et reste une solution portable PC / Mac. Le recentrage Windows de certaines nouvelles fonctionnalités n'est que bonne guerre dans un monde qui a été verrouillé de longue date par des systèmes propriétaires totalement fermés comme ceux d'Apple qu'on acclame bêtement par ailleurs.

Les critiques visant systématiquement Microsoft et sa fermeture font véritablement rire doucement lorsqu'on voit à quel point un Steve Jobs peut être encensé alors même qu'il a été le chantre des systèmes propriétaires et fermés ! Deux poids deux mesures que toute personne dotée d'un véritable cerveau (et non pas d'une "boite" à aduler le premier pseudo gourou venu) ne peut sérieusement pas accepter.

Bref, Silverlight 5 est là et bien là, chargé de nouveautés, toujours cross-plateforme et cross-browser.

Qu'il donne quelques nouveaux avantages à la plateforme Windows semble être bien plus un constat réaliste face au comportement conservateur et fermé des concurrents qu'un réel tournant prémédité. Microsoft ne peut être le seul éditeur à tout offrir gratuitement aux autres plateformes quand ces dernières font tout pour rester fermées et rendre le client captif. Gentil Microsoft, mais pas idiot.

Les nouveautés de SL 5 sont nombreuses et certaines réclament encore du temps en investigation pour les creuser. Pour ce billet je me limiterai donc à vous fournir un panorama des nouveautés officiellement présentes dans la version finale. J'aurai l'occasion de revenir plus en détail sur certaines choses dans de prochains billets.

Les nouveautés

Support amélioré des Média

Audio avec latence faible : la nouvelle API SoundEffect permet enfin de jouer des boucles "propres", d'avoir des effets sonores sans latence (avec réglage du pitch, du panoramique et du volume, possibilité de mettre en cache les sons...).

Playback avec variation de vitesse : Les vidéo peuvent être jouées à des vitesses différentes et supportent le rembobinage et l'avance rapide. Jusqu'à 2 fois la vitesse normale le système automatique de correction de pitch permet même de visualiser en accéléré tout en conservant un audio inchangé (mais plus rapide).

Amélioration des performances en H.264 sur les vidéos non protégées.

Gestion des DRM avec rotation de clé / LiveTV permettant la lecture des sources protégées par système de licences tournantes.

Protection des contenus en interdisant la lecture des média protégés par des applications non autorisées.

Support amélioré du texte

Tracking et Leading : réglage fin de l'interlignage et de l'espacement des caractères.

Conteneurs liés : Autorise un texte à s'écouler sur plusieurs conteneur avec mise en page automatique pour créer du multi colonage ou d'autres effets.

Support des fontes OpenType et du texte en mode "pixel snap" pour une meilleure lisibilité. Les fontes OpenType sont supportées totalement, c'est à dire avec leurs différents styles, les ligatures de caractères, les petites capitales, le positionnement en indice et exposant...

Impression vectorielle PostScript : nouveau mécanisme d'impression bien moins gourmand en mémoire que le système bitmap introduit dans SL 4 (mais nécessite une imprimante dotée d'un pilote PostScript).

Amélioration des performances du moteur de rendu.

Support pour applications LOB avancées

PivotViewer : nouveau composant introduit désormais dans le SDK conçu pour l'affichage dynamique des collections.

ClickCount : Support du multi click, principalement utile pour supporter le double-clic.

ListBox et ComboBox : Positionnement automatique de la liste en fonction d'un texte tapé au clavier.

Ancestor RelativeSource binding : extension du Binding pour se lier à une propriété d'un contrôle parent (existait en WPF).

DataTemplates implicites : Les DataTemplates peuvent être définis implicitement.

DataContextChanged : nouvel évènement qui faisait défaut.

Ajout du support de PropertyChanged au trigger via l'énumération UpdateSourceTrigger.

Dialogue Save / Open file : Possibilité de spécifier un nom de fichier par défaut et un répertoire par défaut.

Debug du DataBinding : possibilité de mettre des points d'arrêt sur les DataBinding XAML pour examiner les locales et déboguer les data bindings.

Extensions Xaml personnalisées : Exécuter du code personnalisé au moment du parsing XAML.

Binding dans les Styles : Les styles supportent désormais le binding.

Amélioration des performances

Parser Xaml : Amélioration des performances pour le parsing des UserControl et des dictionnaires de ressources.

Latence réseau diminuée : Amélioration jusqu'à 90% du temps de réponse dans les scénarios utilisant ClientHttpWebRequest.

Accélération matérielle pour le rendering sous IE9 en mode windowless : SL5 utilise maintenant la nouvelle API SurfacePresenter pour le rendu en windowless sous IE9.

JIT Multi-cœur : Les applications SL démarrent plus vite en profitant des machines à plusieurs cœurs.

Support des browsers 64 bit.

Améliorations graphiques

Amélioration de la pile graphique : réorganisation de la pile de rendu graphique pour ajouter des features comme les animations indépendantes (tournant dans des threads séparés).

3D : utilisation de l'API XNA pour créer des affichages 3D animés ou non, basé sur l'accélération GPU.

Extensions du mode "Trusted"

Ces améliorations concernent principalement les applications in-browser en y ajoutant des possibilités hier réservées à l'OOB. Cela ne fonctionne (pour la plupart) qu'avec des applications signées par un certificat et l'ajout d'une entrée dans la base de registre Group Policy. Bien que très contraignantes techniquement, ces sécurités permettent d'envisager des applications plus complexes, uniquement en mode Web sans installation. La cible visée est clairement l'Intranet pour des applications LOB, ce qui n'écarte pas les applications Web spécifiques (par exemple logiciel médical, application de Comptabilité ou Gestion dans le Cloud très sécurisées...).

Multi fenêtrage Windows : possibilité de créer des fenêtres secondaires Windows non modales et pouvant se positionner sur tous les écrans disponibles. Possibilité de personnaliser totalement l'aspect (par défaut look Chrome de Windows). Disponible en mode droit élevés (sans obligation de certificat).

Full-Trust in-browser : Avoir la puissance d'une application OOB en mode in-browser, avec les obligations de sécurité listées plus haut.

Accès au système de fichier sans limite : Lecture / Ecriture de tout répertoire depuis une application en full-trust.

P/Invoke : Autorise une application SL à utiliser du code natif Windows sur la machine cliente. Permet même d'embarquer des DLL natives et de les exécuter.

Conclusion

Comme on le voit, de très nombreuses choses ont été ajoutées et certaines d'importance comme la 3D, la gestion correcte du son et des bouclages, le support d'OpenType et de toutes ses possibilités typographiques, etc...

Certains ajouts comme le mode full-trust en in-browser sont clairement destinés à la conception d'applications LOB en entreprise. L'ajout de P/Invoke, des fenêtres Windows, etc., sont autant d'indices de ce nouvel axe vers lequel Microsoft pousse Silverlight. C'était d'ailleurs un simple constat, Silverlight a été essentiellement utilisé en entreprise pour créer des applications LOB et assez peu pour faire des applications Web grand public.

Nous sommes bien dans la fin du rêve du "plugin universel" avec un recentrage évident de Silverlight en tant qu'outil Intranet pour les applications LOB en entreprise tournant sous Windows.

Certes le support Mac existe toujours, mais les nouveautés réservées à Windows sont un signe clair de ce nouvel axe de développement de SL. Et finalement cela ne gênera que peu de monde puisque, justement, c'est ainsi que nous utilisions tous SL majoritairement...

Beaucoup de nouvelles features (malgré les bêta auxquelles j'ai eu accès sans pouvoir en parler, NDA oblige) restent assez nébuleuses et manquent cruellement de documentation et d'exemples. Par exemple l'impression vectorielle ou le P/Invoke sur un DLL native embarquée dans le Xap.

La partie 3D est, regrettablement, totalement incompatible avec celle de WPF et se calque sur XNA. L'avantage est une plus grande vitesse d'exécution et des possibilités largement documentées dans les docs XNA pour la XBox. La portabilité facilitée des jeux, grands utilisateurs de 3D aujourd'hui, font de ce choix un choix réaliste et efficace. Encore une fois Microsoft fait preuve d'un sens des réalités louables, même

si la 3D de Silverlight se limite à Windows et n'a rien à voir avec la 3D vectorielle de WPF.

Silverlight 5 cristallise tous les bouleversements en cours et à venir : la fin de certains rêves mais l'affirmation d'un ancrage dans la réalité des besoins des utilisateurs, le tout sur fond de repli sur Windows. La naissance des 3 grands blocs qui vont s'affronter (voir l'un de mes précédents billets) est en route...

Pour ce qui est des nouvelles possibilités dont la mise en œuvre est encore un peu floue, cela va se décanter et je ne manquerai pas de les illustrer dès que j'aurai pu les éclaircir !

Vive Xaml, Vive C# et Vive Silverlight !

Les extensions de balises Xaml de Silverlight 5, vous connaissez ?

Vous avez certainement dû lire quelque chose à propos des "*custom markup extensions*" de Silverlight 5... si, en cherchant bien... Mais que vous en est-il resté ? Sauriez-vous utiliser cette possibilité (qu'on retrouve dans WPF) pour faire quelque chose d'utile, pour vous sortir d'une impasse ? Il y a de grandes chances que la réponse soit non. Alors suivez-moi, vous allez voir, c'est une feature vraiment intéressante...

Custom Markup Extensions

Une ligne parmi les nouveautés de Silverlight 5, une ligne oubliée pour une version éclipsée par les problèmes de communications de Microsoft autour de Silverlight... Lire à ce propos "Silverlight 5 : Hommage à un oublié..." que j'avais écrit début juillet.

Forcément ça fait beaucoup d'oublis pour une pauvre feature.

Et pourtant ! Les "custom markup extensions" sont fantastiques !

Il s'agit ni plus ni moins de pouvoir créer ses propres extensions à Xaml.

C'est à dire faire des choses qui étaient impossibles avant ou bien très complexes et réclamant beaucoup de code ou d'astuces difficilement maintenable.

Les possibilités sont infinies. On peut s'en servir pour gérer un système de localisation complet (concernant toute ressource et pas seulement du texte), on peut s'en servir

pour automatiser des réglages, pour exécuter du code en Xaml, récupérer des valeurs qu'un Binding ne sait pas retrouver, etc...

C'est vraiment dommage que ces extensions personnalisées de Xaml n'aient pas rencontré le succès qu'elles méritent.

Heureusement il y Dot.Blog 

Les bases

Nous utilisons souvent les extensions de balisage fournies avec Xaml comme celles concernant le Binding, l'accès aux ressources dynamiques, le Template binding etc. On utilise ces extensions de balisage avec les accolades dites "américaines" {}. Avec Silverlight 5, nous pouvons maintenant écrire nos propres Extensions de balisage personnalisées, et c'est une avancée de taille, même si, comme je le disais, elle est passée un peu inaperçu.

Pour créer une extension personnalisée il faut :

Créer une classe héritant de "`MarkupExtension`" (ou bien implémenter l'interface `IMarkupExtension`).

Le nom de la classe doit avoir le suffixe "`Extension`"

`IMarkupExtension` expose une méthode, `ProvideValue`. C'est cette méthode qu'il faut fournir pour retourner une valeur, celle qui sera exploitée par Xaml.

Il est tout à fait possible (et souvent souhaitable) de passer des paramètres à notre classe d'extension qui ont pour reflet les propriétés publiques de cette dernière.

On peut obtenir aussi le nom et les propriétés du contrôle qui est lié grâce à `ServiceProvider`, passé par défaut en paramètre à la méthode `ProvideValue`.

Comme on le comprend ici, il ne s'agit pas de pouvoir ajouter des balises à Xaml mais de pouvoir personnaliser une valeur de propriété en lui attribuant un équivalent de la syntaxe du Binding.

En première approximation, on peut dire que les extensions personnalisées sont une "sorte de" Binding avec une syntaxe similaire. Mais "une sorte de" seulement. Le Binding effectue des opérations bien spécifiques (liaison entre deux propriétés de deux instances), les extensions personnalisées permettent tout ce qui possible dans la limite de la syntaxe imposée. Exécuter du code, récupérer des valeurs, activer des

fonctions, traduire des mots, des ressources, bref, tout ce qu'on peut vouloir faire en générant une valeur qui sera attribuée à une propriété en Xaml.

Mise en œuvre simplifiée

Afin de concrétiser tout cela sans se lancer dans un code trop ambitieux, je vais prendre un exemple minimaliste, donc un peu stupide, mais c'est le mécanisme général que je souhaite d'abord vous montrer afin d'effacer les craintes éventuelles sur la complexité de cette nouvelle fonctionnalité.

Je vais créer une extension qui sait concaténer deux chaines de caractères et qui supporte l'indication d'un séparateur.

Voici le code de cette extension :

```
using System;
using System.Windows.Markup; // nécessaire pour les extensions

namespace markupA
{
    public class ConcatenationExtension : MarkupExtension
    {
        public string Str1 { get; set;}
        public string Str2 { get; set;}
        public string Separator { get; set;}

        public ConcatenationExtension()
        {
            Separator = " ";
        }

        public override object ProvideValue(IServiceProvider
                                             serviceProvider)
        {
            var a = Str1 ?? string.Empty;
            var b = Str2 ?? string.Empty;
            var s = string.IsNullOrWhiteSpace(Str1) ||
                    string.IsNullOrWhiteSpace(Str2)
                            ? string.Empty
                            : Separator;
            return Convert.ToString(a + s + b);

        }
    }
}
```

C'est très court... Ça ne fait rien d'extraordinaire je sais, mais tout de même...

Dans la `MainPage.xaml` je vais poser un `TextBlock` et je vais utiliser mon extension pour attribuer à la valeur `Text` de ce dernier le résultat d'une concaténation de deux chaines en utilisant un séparateur fixé librement.

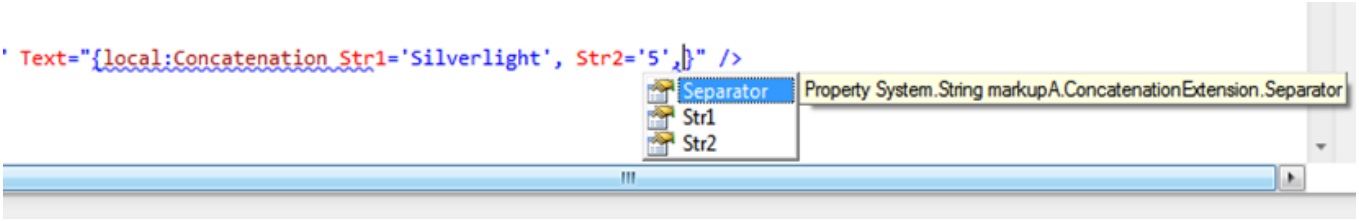


Figure 70 - Extension de balises

Comme on peut le voir, Visual Studio reconnait mon extension et l'IntelliSense sait même me proposer le nom des propriétés de celle-ci, ce n'est pas magique ça ? !

Bon... il y a malgré tout quelques "bricoles" qui ne sont pas tout à fait au point sous VS 2010. Par exemple vous voyez sur la capture ci-dessus qu'il y a un souligné ondulé sous le nom de mon extension. VS me dit que j'utilise une classe comme une extension mais qu'elle devrait descendre de MarkupExtension. C'est bien ce qui est fait, le code un peu plus haut le prouve... Bug. Mais pas gênant.

Deuxième déception, VS semble incapable d'afficher la valeur au design. C'est dommage.

En revanche, quand on compile tout va bien et quand on exécute cela marche comme prévu, c'est déjà beaucoup !

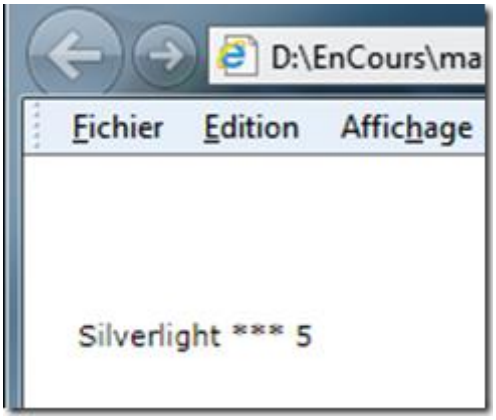


Figure 71 - Effet d'une balise personnalisée

Je n'ai pas testé sous Blend car Blend 5 n'est officiellement pas sorti et que la version bêta mise à disposition à la sortie de SL 5 est arrivée en fin de validité. Il y a peut-être des choses nouvelles que j'aurai pu utiliser mais si elles sont sous NDA je ne pourrais même pas vous dire qu'elles existent (difficile de dire sans dire tout en disant sans le dire).

Bref, ça marche comme prévu et c'est toujours comme ça avec la plateforme .NET, des bugs il y en a, mais ils ne sont bloquants que de façon rarissime. Quand on utilise

beaucoup de logiciels, d'IDE et de langages différents, parfois couteux, on sait à quel point un tel niveau de qualité est exceptionnel en informatique, alors je ne m'attarderais pas sur ces petits dysfonctionnements.

A noter : La classe utilise le suffixe Extension, mais le lecteur attentif aura certainement remarqué qu'en Xaml ce suffixe disparait et ne doit pas être utilisé. C'est un peu comme pour les propriétés de dépendances, on créée ajoute le suffixe "Property" côté code C# mais on ne l'écrit pas en Xaml.

Un peu plus sophistiqué, simuler "'Static" de WPF

Comme vous le savez, sous Silverlight les ressources sont Dynamiques, on ne dispose pas du marqueur "`Static`" de WPF (ou alors j'ai loupé moi aussi une nouveauté de SL 5 !).

Quoi qu'il en soit cela fait un bon exercice pour créer une extension qui simule "`Static`" ! Et c'est ce que nous allons faire maintenant.

Regardons tout de suite le code, déjà un peu plus travaillé :

```
using System;
using System.Reflection;
using System.Windows.Markup;

namespace markupA
{
    public class StaticExtension : MarkupExtension
    {
        public string TypeName { get; set; }
        public string PropertyName { get; set; }

        public override object ProvideValue(IServiceProvider
                                             serviceProvider)
        {
            object returnValue = null;
            var t = Type.GetType(TypeName);

            if (t != null)
            {
                var p = t.GetProperty(PropertyName, BindingFlags.Static
                            | BindingFlags.Public);

                var v = p.GetValue(null, null);
                if (v is IConvertible)
                {
                    var ipvt =
                   serviceProvider.GetService(typeof(IProvideValueTarget))
                   as IProvideValueTarget;

                    if (ipvt != null)
                    {
                        var pi = (PropertyInfo)ipvt.TargetProperty;

                        returnValue = Convert.ChangeType(v,
                                              pi.PropertyType, null);
                    }
                }
                else
                {
                    returnValue = v;
                }
```

```
            }
            return (returnValue);
        }
    }
}
```

Le principe reste bien entendu rigoureusement le même. Les paramètres sont des propriétés publiques de la classe qui dérive de `MarkupExtension` bien entendu.

Ici, le but du jeu est de passer le nom d'un type et celui d'une de ces propriétés pour générer la valeur.

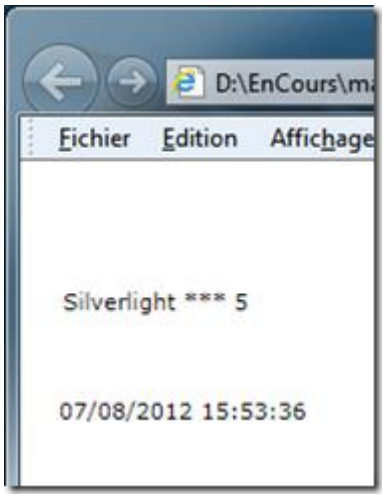
On retrouve donc un peu de Réflexion dans ce code.

Comme ce n'est qu'un exemple, le code est largement perfectible, par exemple ne tester que `IConvertible` sur la propriété est vraiment insuffisant, mais ce n'est qu'un exemple.

Côté Xaml nous pouvons écrire quelque chose de ce type :

```
"Top" Text="{local:Concatenation Str1='Silverlight', Str2='5',Separator=' ***
="Top" Text="{local:Static TypeName=System.DateTime,PropertyName=Now}"/>
```

Bien entendu les petits soucis évoqués plus haut existent toujours mais ils n'empêchent pas IntelliSense de fonctionner et encore moins l'application de faire ce qu'on attend d'elle :



Conclusion

Faire le tour des possibilités des extensions personnalisées est impossible, c'est infini...

L'idée était plutôt de vous rappeler que cela existe, et de vous montrer à quel point cela est simple à mettre en œuvre.

Je suis certains que vous trouverez mille façons de vous en servir.

Lorem Ipsum generator : un autre exemple Silverlight

Et un de plus dans la galerie ! :-)

http://www.e-naxos.com/slsamples/lorem/loremgen.html

"Lorem ipsum…" vous connaissez tous ce texte en pseudo-latin que tous les infographistes utilisent pour tester la mise en page de leurs documents, de la carte de visite aux sites web, de la plaquette commerciale sur papier glacé jusqu'à, bien entendu, des applications Silverlight.

Il existe beaucoup d'extraits de ce fameux texte. Je vous propose "ma" version, en Silverlight.

Figure 72 - Lorem Ipsum

Fonctionnalités

Génération

LIG (Lorem Ipsum Generator) est capable de générer des Paragraphes, des Phrases et des Mots de façon aléatoire.

Une phrase est constituée de mots, il est possible de choisir une fourchette, par exemple entre 3 et 8 mots. Le tirage est aléatoire pour chaque phrase.

Un paragraphe est constitué de phrases, il est ici aussi possible de choisir une fourchette, par entre un paragraphe fait de 5 à 8 phrases.

Qu'il s'agisse de paragraphes, de phrases ou de mots, la "chose" générée le sera en plusieurs exemplaires (tous différents) et ce nombre est fixé par une fourchette.

Les radio boutons en bas à gauche permettent de choisir le mode de fonctionnement.

Une case à cocher sous le texte permet de forcer la première phrase (en mode Paragraphes) à sa valeur la plus connue, le célèbre "Lorem ipsum …".

Visualisation et statistiques

Le texte généré est immédiatement visualisé à l'écran. De plus, LIG fournit des statistiques simples mais importantes sur ce dernier : nombre de symboles et nombre de mots.

Récupération du texte

Le plus simple est de cliquer sur le bouton "Copy". Une confirmation est demandée par mesure de sécurité. Il suffit ensuite de coller ce texte où vous en avez besoin…

Bouton Info

Il affiche une page d'information sur l'application

Installation Out-Of-Browser (OOB)

LIG supporte le mode de fonctionne OOB, en cliquant sur le bouton "Install" (en haut à droite, il s'illumine quand on passe la souris dessus) l'application LIG sera téléchargée depuis le Web et installée sur votre machine. Selon votre choix une icône peut être placée sur le bureau ainsi qu'une entrée dans le menu Démarrer (ou l'équivalent sous Mac).

Lors de la première installation l'application sera ouverte immédiatement, le site web sera toujours à l'écran aussi. Fermez ce dernier. Vous disposez maintenant d'une

application locale totalement indépendante… Bien entendu LIG n'a alors plus besoin de connexion Internet pour fonctionner et **durant son utilisation ni il n'envoie ni il ne reçoit des données depuis Internet. C'est totalement safe et sans espionnage**.

Conclusion

Du coup, me privant d'une remontée d'info automatique pour respecter la liberté de chacun de ne pas être pisté par la moindre application, je remercie par avance les utilisateurs de LIG de bien vouloir me laisser un mot pour me dire s'ils apprécient ou non l'application.

Le code source sera mis à disposition d'ici quelques temps, alors… Stay Tuned !

Le code source de "Lorem Ipsum Generator"

"Lorem Ipsum Generator" est un générateur on-line gratuit de textes, phrases, mots, lignes du fameux "Lorem Ipsum" utilisé par tous les infographistes et typographes du monde (au moins occidental) pour simuler la mise en page de textes réels sans que le "faux texte" ne puisse attirer l'œil dans une phase où seul le design compte.

Le logiciel Silverlight

J'ai publié en 2010 une version Silverlight très "lookée" d'un tel générateur qui est d'ailleurs toujours utilisable (pour la création de vos mise en pages WinRT, Silverlight, WPF ou même HTML) à l'adresse suivante : http://www.e-naxos.com/slsamples/lorem/loremgen.html

Le code source

J'avais promis de publier le code, me réservant la possibilité d'en faire un article. Et puis le temps a passé… Emporté par les tourbillons du fils de Gaïa et d'Ouranos, le bien nommé Kronos, j'ai oublié de le faire.

A la demande insistante de certains lecteurs je me fais violence en cette heure tardive (presque une heure du matin) pour fabriquer un zip depuis mon repository subversion afin de récompenser la fidélité et la patience de ces derniers.

Le projet que je livre ici n'est pas commenté, mais vous y trouverez facilement ce que vous cherchez. Il est possible que certaines librairies utilisées à l'époque ne soit pas incluses (je pense notamment aux effets visuels avec des shaders).

Tout ce qui pourrait manquer se retrouvera sur Internet j'en suis certain.

Bonne lecture du code, et si vous en faites quelque chose d'autre, prévenez-moi, ça m'intéresse de voir comment un projet peut être repensé.

Lorem Ipsum Generator Source code zippé

Silverlight 5 – Fuite mémoire avec les images (une solution)

Utiliser Silverlight 5 n'est pas forcément une option puisque même si vous êtes content de la version 3 ou 4 et que vos logiciels marchent bien sous ces versions vos utilisateurs ont forcément reçus une mise à jour vers la version 5 qui ruinera peut-être ces applications qui pourtant tournaient correctement jusqu'à lors…

Bogue fantôme ?

Le code incriminé est très simple et se retrouve dans de nombreuses applications Silverlight dès lors qu'elles manipulent des images.

Les premières informations sur ce bogue ne sont pas si récentes puisque déjà en 2009, Jeff Prosise indiquait avoir rencontré le problème et fournissait des exemples (repris ici). Puis les choses semblent s'être tassées étrangement. Mais la diffusion de SL 5 a relancé le débat avec des utilisateurs rapportant le même genre de problème (out of memory) dans des applications manipulant des images. Il y a deux jours sur la liste de diffusion "Silverlight List" (liste réservées aux personnes sous NDA) quelques MVP ont rapporté des ennuis similaires arrivés à leurs clients ou contacts.

Bogue fantôme ? Fluctuations quantique du vide ? Je ne saurai le dire. Quoi qu'il en soit le nombre de personnes rapportant ce problème est reparti à la hausse depuis la sortie de SL 5.

Un code simple

Comme je le disais, le code permettant de mettre le bogue en évidence est d'une grande simplicité et peut se retrouver sous une forme ou autre dans toute application qui manipule des images.

```
private Image CreateThumbnailImage(Stream stream, int width)
{
    BitmapImage bi = new BitmapImage();
    bi.SetSource(stream);

    double cx = width;
    double cy = bi.PixelHeight * (cx / bi.PixelWidth);

    Image image = new Image();
    image.Width = cx;
    image.Height = cy;
    image.Source = bi;

    return image;
}
```

Quand je vous dis que c'est un code simple, c'est vraiment simple. Cet exemple est une méthode qui créée une miniature d'une certaine largeur à partir d'un flux (stream) contenant une image. Peu importe d'où vient le flux (téléchargement internet par exemple ou lecture disque).

Le problème ?

Il est tout simple lui aussi : `out of memory`.

Simple comme une lame de rasoir, tranchant comme une guillotine toute neuve, incontournable comme un éléphant coincé entre les murs d'une rue trop étroite. L'application plante sans autre forme de procès.

L'exemple porte sur la création de miniatures car c'est un cas où les choses tourneront vite au vinaigre. Avec deux ou trois images manipulées par l'application le bogue passera inaperçu, mais en balayant tout un répertoire pour créer des miniatures la mémoire va rapidement exploser !

La classe BitmapImage nous rappelle que nous manipulons des images toujours plus grosses dont nous avons oublié la taille réelle… Un Jpeg de 2 Mo ne pèse pas 2 Mo, c'est son image disque compressée qui fait ce poids. Mais pour l'afficher ou la traiter il faudra bien en reconstituer chaque pixel en mémoire… et c'est là que BitmapImage

nous rappelle que ces 2 Mo tout à fait raisonnables prendront peut-être 40 à 50 Mo en RAM !

Cela n'est malgré tout pas grand-chose après tout. D'abord les PC modernes disposent de plusieurs Go de mémoire et .NET et son Garbage Collector font le ménage de façon efficace.

C'est vrai.

Encore faut-il que le code de l'application soit bien écrit et qu'il n'y ait pas de bogue dans le Framework. Si la première condition n'est pas toujours réalisées, même les meilleurs font des erreurs, il en va de même avec la seconde condition : le Framework lui-même malgré sa qualité n'est pas exempt de bogues.

Si vous balayez tout un répertoire de Jpeg de bonne taille (dans les 2 Mo) et que vous afficher les miniatures en utilisant le code plus haut, au bout d'une vingtaine d'images le code plantera en out of memory. Un examen de la mémoire consommée vous montrera peut-être un chiffre délirant dépassant 1 à 2 Go !

Quand on pense à une vingtaine de miniatures en 100x100 environ, une trentaine de Ko, on voit mal d'où peuvent provenir ces Go consommés !

Pourtant, en y réfléchissant on s'aperçoit que le composant Image créé par le code exemple référence le BitmapImage original. Ce dernier n'est pas collecté par le GC. Et quand on a compris qu'une image Jpeg de 2 Mo pesait en réalité 40/50 Mo en mémoire, on comprend vite comment au bout d'une vingtaine d'images on arrive et même dépasse le Go de RAM…

Une solution ?

Il existe un moyen d'éviter le problème. Il complique le code, par force :

```
private Image CreateThumbnailImage(Stream stream, int width)
{
    BitmapImage bi = new BitmapImage();
    bi.SetSource(stream);

    double cx = width;
    double cy = bi.PixelHeight * (cx / bi.PixelWidth);

    Image image = new Image();
    image.Source = bi;

    WriteableBitmap wb = new WriteableBitmap((int)cx, (int)cy);
    ScaleTransform transform = new ScaleTransform();
    transform.ScaleX = cx / bi.PixelWidth;
    transform.ScaleY = cy / bi.PixelHeight;
    wb.Render(image, transform);
    wb.Invalidate();

    Image thumbnail = new Image();
    thumbnail.Width = cx;
    thumbnail.Height = cy;
    thumbnail.Source = wb;
    return thumbnail;
}
```

L'idée qui se cache derrière ces lignes est que plutôt d'assigner une grosse `BitmapImage` à une petite Image on passe par une `WriteableBitmap` avec une `ScaleTransform` pour créer la miniature. On assigne enfin cette miniature à l'image retournée. Dans cette version le `BitmapImage` n'est plus lié à l'objet Image retourné, et quand on sort du scope de la méthode le GC peut faire son boulot (au moment où il le décidera).

Ca plante toujours !

Hmm… Et oui.

Dans les deux conditions évoquées plus haut concernant la qualité du code, nous n'avons fait que résoudre la condition 1 : écrire un code plus intelligent et moins naïf.

Reste la condition 2 : un éventuel bogue du Framework lui-même.

Et hélas, cette seconde condition se réalise dans ce cas pourtant simple.

Le véritable problème est que `WriteableBitmap`, pour une raison qui échappe à la raison, conserve une référence sur l'objet XAML source ! Aucune méthode, aucune propriété de cette classe ne permettent de relâcher cette ressource.

L'objet `WriteableBitmap` conserve ainsi une référence sur l'objet Image qui lui-même référence la `BitmapImage`. Aucune de ces instances n'est donc supprimée de la mémoire par le GC.

Point final ?

Non, pas tout à fait.

Enfin la solution…

Avec de l'imagination, des tests, et une bonne connaissance du Framework certains ont découvert que la référence retenue par la `WriteableImage` vers l'objet XAML source peut être évitée dans le cas, subtile, où la méthode `Render` n'est pas utilisée. Mais cela est impossible, il faut bien faire un `Render` pour appliquer la transformation. La solution consiste alors à utiliser une seconde `WriteableImage` qui sera créée non plus avec `Render` mais par une vilaine boucle `For` qui copiera un à un les pixels de la première vers la seconde. Et c'est cette seconde `WriteableImage` qui sera utilisée comme source de l'objet Image retourné…

Dès lors tout le fatras intermédiaire (`BitmapImage`, `WriteableImage` avec référence sur la source, etc), tout cela ne sera plus référencé par personne et en sortant du scope de la méthode pourra être supprimé de la RAM !

Le code final devient alors :

```
private Image CreateThumbnailImage(Stream stream, int width)

{
    BitmapImage bi = new BitmapImage();
    bi.SetSource(stream);

    double cx = width;
    double cy = bi.PixelHeight * (cx / bi.PixelWidth);

    Image image = new Image();
    image.Source = bi;

    WriteableBitmap wb1 = new WriteableBitmap((int)cx, (int)cy);
    ScaleTransform transform = new ScaleTransform();
    transform.ScaleX = cx / bi.PixelWidth;
    transform.ScaleY = cy / bi.PixelHeight;
    wb1.Render(image, transform);
    wb1.Invalidate();

    WriteableBitmap wb2 = new WriteableBitmap((int)cx, (int)cy);
    for (int i = 0; i < wb2.Pixels.Length; i++)
        wb2.Pixels[i] = wb1.Pixels[i];
    wb2.Invalidate();

    Image thumbnail = new Image();
    thumbnail.Width = cx;
    thumbnail.Height = cy;
    thumbnail.Source = wb2;
    return thumbnail;
}
```

Forcément cela devient beaucoup plus tordu et infiniment moins simple et moins naïf que le code exemple donné en début de billet. C'est le prix à payer pour réunir les deux conditions d'un code qui passe : écrire du bon code pas naïf, et contourner le bogue du Framework.

C'est lourd, bien plus lent que la première version, mais au lieu de planter dans les premières images, votre application pourra créer des milliers de miniatures sans souci.

Conclusion

Ce bogue ne touche pas que les applications qui créent des miniatures, vous avez bien compris que cela n'est qu'un exemple. Toute application qui manipule des images peut rencontrer le problème d'une façon ou d'une autre. L'exemple des miniatures n'est que l'une de ces façons…

Alors si vous avez des applications Silverlight qui plantent en Out Of Memory sans raison alors qu'elles ne font rien de bien compliqué avec des images, vous avez peut-être ici la clé de vos problèmes, et un bout de code dont vous pouvez vous inspirer pour régler le problème !

Encoder en ANSI ou autre avec Silverlight

Encoder des données texte avec Silverlight est parfois problématique car le runtime ne supporte que l'Unicode (UTF-16) et UTF8. Souvent on a besoin d'autres encodages, comme par exemple ANSI. Que faire ? Utiliser un générateur !

La classe Encoding

C'est la classe de base des encodeurs .NET.
Sous Framework complet on dispose de plusieurs encodeurs comme le fameux ANSI évoqué en introduction.

Hélas, le profil .NET de Silverlight est par force limité (sinon il serait aussi gros que le framework complet). Et il y a des coupes sombres un peu partout. Cela a été fait avec doigté et beaucoup d'intelligence, mais le problème c'est que parfois certaines choses manquent cruellement.

La stratégie la plus fréquente consiste à écrire le "morceau" de .NET qui manque. Comme un projet Silverlight n'a pas besoin de tout le framework, cette réécriture des parties manquantes se limite le plus souvent à quelques classes. La taille du XAP augmente un peu pour compenser celle du plugin, mais c'est une stratégie globalement payante pour tout le monde : Microsoft peut distribuer un plugin assez léger bien que très complet, le développeur ajoute ce qui lui manque ce qui n'arrive jamais à faire la taille du framework complet malgré tout.

Reste que réécrire certaines parties n'est pas toujours simple. Soit parce que cela est juste fastidieux, soit parce que cela ne marche pas vraiment (limiter les valeurs d'une

propriété de dépendance par exemple, le code supprimé ne peut pas être “réinjecté” là où il le faudrait), soit parce que cela est particulièrement ardu.

Dans tous les cas c’est une perte de temps qu’on aimerait éviter.

Grâce au partage sur le Web on finit toujours par trouver quelque chose, au moins un squelette de code qu’on peut triturer pour l’adapter à ses besoins.

Mais parfois on a des surprises encore meilleures !

Un générateur de classes dérivant de Encoding

C’est encore le plus simple, bien entendu… Plutôt que de bricoler un bout de code, utiliser un générateur est le summum du gain de temps !

Et bien cela existe…

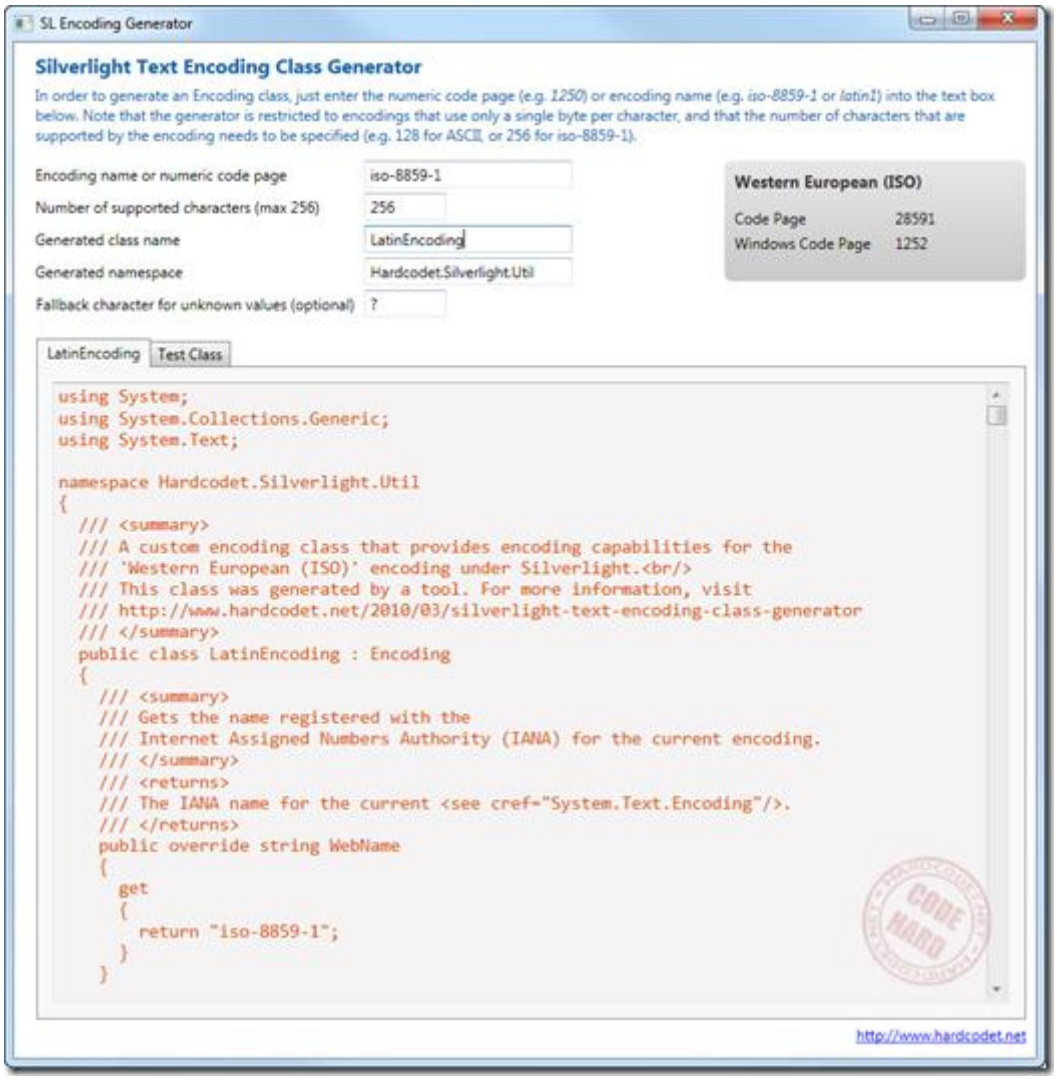


Figure 73 - générateur d'encodeur

Philipp Sumi a écrit un tel générateur pour créer des encodeurs de texte pour tous les types possibles, dont ANSI.

Cet utilitaire est fournir en code source et en code compilé pour aller encore plus vite.

Que demander de mieux ?

Rien. Dire Merci à Philipp, c'est déjà pas mal

Le code Source

Le code Compilé

L'entrée de blog de Philipp Sumi

Conclusion

Grâce à la générosité de nombreux bloggers dans le monde, développer est souvent plus simple, plus facile que cela ne le fut à une époque où nous n'avions pas Google, ni Internet, juste quelques BBS américain à 500F l'heure de communication avec les USA sur un modem 1200b et un pauvre manuel du langage souvent mal traduit…

Epoque de pionniers, époque terminée, quels chanceux vous êtes !

Avertissements

L'ensemble de textes proposés ici est issu du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.

Les billets ont été collectés en septembre 2013 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile. Les mois d'octobre et novembre ont été consacrés à la remise en forme, à la relecture, parfois à des corrections ou ajouts importants comme le livre PDF sur le cross-plateforme par exemple (TOME 5).

Les textes originaux ont été écrits entre 2007 et 2013, six longues années de présence de Dot.Blog sur le Web, lui-même suivant ses illustres prédécesseurs comme le Delphi Stargate qui était dédié au langage Delphi dans les années 90/2000.

Ce recueil peut parfois poser le problème de parler au futur de choses qui appartiennent au passé… Mais l'exactitude technique et l'à propos des informations véhiculées par tous ces billets n'a pas de temps, tant que les technologies évoquées existeront …

Le lecteur excusera ces anachronismes de surface et prendra plaisir j'en suis certain à se concentrer sur le fond.

E-Naxos

E-Naxos est au départ une société éditrice de logiciels fondée par Olivier Dahan en 2001. Héritière d'Object *Based System* et de *E.D.I.G.* créées plus tôt (1984 pour cette dernière) elle s'est d'abord consacrée à l'édition de logiciels tels que la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme par exemple MK Query Builder (requêteur visuel SQL).

Peu de temps après sa création E-Naxos s'est orientée vers le Conseil et l'Audit puis s'est ouverte à la Formation et au Développement au forfait. Faisant bénéficier ses clients de sa longue expérience dans la conception de logiciels robustes, de la relation client, de la connaissance des utilisateurs et de l'art, car finalement c'en est un, de concevoir des logiciels à la pointe mais maintenables dans le temps.

C#, Xaml ont été les piliers de cette nouvelle direction et Olivier a été récompensé par Microsoft pour son travail au sein de la communauté des développeurs WPF et Silverlight. Toutefois sa première distinction a été d'être nommé MVP C#. On ne construit pas de beaux logiciels sans bien connaître le langage…

Aujourd'hui E-Naxos continue à proposer ses services de Conseil, Audit, Formation et Développement, toutes ces activités étant centrées autour des outils et langages Microsoft, de WPF à WinRT (Windows Store) en passant par Silverlight et Windows Phone.

A l'écoute du marché et offrant toujours un conseil éclairé à ses client, E-Naxos s'est aussi spécialisée dans le développement Cross-Plateforme, notamment dans le mariage des OS Microsoft avec Android, les deux incontournables du marché d'aujourd'hui et de demain.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares !