www.e-naxos.com

Formation – Audit – Conseil – Développement
XAML (Windows Store, WPF, Silverlight, Windows Phone), C#
Cross-plateforme Windows / Android / iOS
UX Design

ALL DOT.BLOG

Tome 5

Développement Cross-Plateforme

Tout Dot.Blog par thème sous la forme de livres PDF gratuits !

Olivier Dahan
odahan@gmail.com

Table des matières

Introduction 11
Présentation de l'édition 2013/2014 11
Stratégie de développement Cross-Platform – Partie 1 12
Cross-platform, cross-form factor 12
My Strategy, for today 13
Une contrainte forte : la cohérence 14
Un But : Unifier ce qui est épars 15
Un principe : Concentrer la compétence 15
Les cibles visées 15
Des plateformes, pas des technologies de développement 15
Les cibles 15
Un rêve ? Non, une réalité ! 18
Et le Web ? 19
Les effets néfastes de la canicule sur un esprit surmené ? 19
La stratégie 20
Divide ut regnes ! 20
MVVM 21
Un langage et un seul 21
Une seule plateforme 22
Un seul EDI 22
Une seule version... 22
Une librairie Cross-plateform 23
Un tooling ultra simplifié 24
Explication par l'image 24
Le serveur "métier" ou la librairie PCL 26
Un noyau unique 27
Des UI spécifiques 29
Un coût maitrisé 29
Conclusion 30
MonoDroid : et si la double connaissance Android / WinRT était la clé du succès ? 31
MonoDroid c'est Mono et Mono c'est C# / .NET 32
La portabilité des applications WinRT / Android 33
Pourquoi pas Apple ? 34

L'intérêt d'un code C# et d'une plateforme .NET 35
Un cœur pour deux 35
Conclusion 36
Les vidéos de Xamarin EVOLVE 2013 en ligne 37
Platinum Sponsor 37
Xamarin, un succès croissant 38
Une réalité à intégrer : ces machines ont besoin de programmes ! 42
L'entreprise largement concernée 43
Mes clients, vos clients et vos patrons : des entreprises 44
Marier Windows et Android 44
Xamarin : en toute logique 44
EVOLVE 2013 : des conférences à voir absolument 45
Conclusion 45
Stratégie de développement Cross-Platform–Partie 2 45
Rappel sur la stratégie présentée et les outils utilisés 46
Le contexte de l'exemple 47
La preuve de l'intérêt de la stratégie du serveur métier 47
Les cibles que je vais ajouter 49
L'application 50
Réalisation 50
La solution 51
MVVMCross 51
Le projet Noyau 51
Les noyaux dupliqués 58
Conclusion partielle 59
Stratégie de développement Cross-Platform–Partie 3 60
Bref Rappel 60
La stratégie 60
L'exemple de code 61
L'étape du jour 62
Le Noyau 62
Le serveur métier 70
Une première cible : Windows "classique" en mode Console 71
Le film... 75

Une Seconde Cible : Android 79
Une troisième Cible ? 87
Conclusion 89
Stratégie de développement Cross-Platform – Bonus 90
Cross-plateforme : Xamarin Store, des composants à connaître 91
Le Xamarin Store 91
Trois OS, un seul langage 91
Le vrai cross-plateforme 92
Soutenir Windows Phone et WinRT 92
Ma sélection 93
Azure Mobile Services 93
Signature Pad 94
SQLite.NET 95
SQLCipher 96
Radial Progress 98
JSon.NET 99
ZXing.Net.Mobile 99
Xamarin Social 101
Xamarin.Mobile 101
Xamarin.Auth 101
Bar Chart 102
Conclusion 103
WP: Microsoft.Phone.Info, un namespace à découvrir... 103
Le namespace Microsoft.Phone.Info 103
La classe DeviceStatus 104
DeviceExtendedProperties 104
UserExtendedProperties 105
Conclusion 106
Cross-plateforme, stockage ou Web : Sérialisation JSON 106
Pourquoi sérialiser ? 106
Choisir le moteur de sérialisation 106
JSON.NET 107
Utiliser JSON.NET 112
Conclusion 113

Bibliothèque de code portable, Async et le reste… 113
Code portable et Async 113
Possible, mais en framework 4 114
Création d'une PCL avec Async 114
Conclusion 115
Contourner les limites des PCL par la technique d'injection de code natif (cross-plateforme) 115
Des espaces de noms non intégrés aux PCL 115
Phase 1 : Xamarin à la rescousse 116
Phase 2 : Comment appeler du code Xamarin ou .NET dans le noyau .NET CPL ? 116
Phase 3 : implémentation 116
Conclusion 118
Cross-Plateforme : MvvmCross Seminar video 118
Une présentation complète 118
Un petit coucou à e-naxos 118
Conclusion 119
Cross-plateforme : the "magical ring", une session à voir (avec code source) 119
Les anneaux de pouvoir du Seigneur des Anneaux… 119
Une Conférence TechDays 2013 119
Conclusion 120
Cross-plateforme : Android - Part 1 121
Des passerelles et des OS 121
Pourquoi Android ? 122
Répartition des OS sur la dernière année de sept-2012 à sept-2013 123
Répartition du marché des OS sur les 3 derniers mois 124
Répartition des OS Mobiles sur l'année 2012/2013 126
Répartition des OS Mobiles sur les 6 derniers mois 128
Répartition des OS Mobiles sur les 27 premiers jours de octobre 2013 129
Les Ventes 130
Que conclure de ces chiffres ? 132
Alors, pourquoi Android ? 135
Conclusion de la partie 1 136
Cross-Plateforme : Android – Part 2 – L'OS 139
Un OS est un OS (M. Lapalisse) 139
Vecteur ou bitmap ? 140

Langage 141
Pour résumer 143
Les versions d'Android 143
Conclusion 146
Cross-plateforme : Android – Part 3: Activité et cycle de vie 146
Android et les Activités (activity) 147
Le concept 148
La gestion du cycle de vie 149
Les différents états 149
Effet des touches de navigation 150
Les différentes méthodes du cycle de vie 151
Conclusion 156
Cross-plateforme : Android – Part 4 – Vues et Rotation 157
Vues et Rotation 157
Un tour par ci, un tour par là… 158
Gérer les rotations de façon déclarative 159
Les ressources de mise en page 159
Les mises en pages différentes par orientation 162
Les dessinables 164
Gérer les vues et les rotations par code 165
La création d'une vue par code 166
Détecter le changement d'orientation 167
Empêcher le redémarrage de l'Activité 169
Maintenir l'état de la vue sur un changement d'orientation 170
Conclusion 176
Cross-plateforme : Android – Part 5 – Les ressources 176
Les ressources 177
Les ressources de base 178
Créer des ressources et y accéder 178
Accéder aux ressources par programmation 179
Accéder aux ressources depuis un fichier XML 180
Les différentes ressources 180
La notion de ressources alternatives 182
Créer des ressources pour les différents types d'écran 188

Normal ? 188
Les “Density-independent pixel” 189
Les tailles écran en “dp” 190
Supporter les différentes tailles et densités 190
Localisation des strings 196
Conclusion 198
Cross-plateforme : la mise en page sous Android (de Xaml à Xml). 198
Chapitres antérieurs 198
Stratégie de développement cross-plateforme 199
Cross-plateforme : L’OS Android 199
8h de video en 12 volets MvvmCross sous WinRT / WPF / Windows Phone 8 et Android 200
L’IoC clé de voute du Cross-plateforme 200
La stratégie de mise en page sous Android 200
XML Based 201
Java-Based 201
Les attributs de mise en page 201
Les attributs utilisés couramment 202
La Taille (Size) 202
L’alignement 205
Les marges 205
Le padding 206
L’ID 206
Couleurs 207
Gestion du clic 207
Le conteneur linéaire LinearLayout 208
Définir les couleurs 212
Localisation et fichiers de valeurs 213
Deux poids deux mesures ! 214
Tout est relatif ! 215
A table ! 216
La TableRow 217
Les autres conteneurs 219
Conclusion 220
Cross-plateforme : images avec zones cliquables avec MonoDroid/Xamarin vs C#/Xaml 220

C#/XAML 221
Des avantages de la rusticité 221
N'exagérons rien non plus ! 223
Des images cliquables… 223
Xamarin Studio ou Visual Studio ? 224
Le code exemple 224
Choisir la version d'Android 225
La structure de la solution 228
L'astuce des zones cliquables 230
Le résultat 233
Le code 234
Conclusion 238
Introduction à MvvmCross V3 "Hot Tuna" (WinRT / Android) 239
MvvmCross ? 239
Hot Tuna 239
Cross-plateforme 240
Une vidéo pour comprendre 240
Conclusion 241
MvvmCross v3 Hot Tuna : les services (WinRT / Android) 241
Les services 241
MvvmCross v3 : Les listes (WinRT et Android) 241
La série N+1 242
Les listes sous WinRT/Android avec MvvmCross v3 243
Conclusion 243
MvvmCross V3 : Convertisseurs de valeur et Navigation (WinRT/Android) 243
Convertisseurs de valeur, Navigation et paramètres de navigation 243
Cross-Plateforme Windows Phone 8 / Android avec MvvmCross : Blend, données de design, Google Books API … 243
Cross-plateforme 243
Vos projets, mon savoir-faire 244
WP8, Blend, Android, MvvmCross, Xamarin, VS… 244
La vidéo 244
Le bonus 244
Conclusion 245

Cross-Plateforme Windows Phone 8 / Android avec MvvmCross : Blend, données de design, Google Books API–Partie 2 245
Partie 2 245
La vidéo 245
Le bonus 245
Conclusion 245
Cross-Plateforme Vidéo 7 : Géolocalisation et bien plus ! 246
Volet 7 246
N+1 246
La vidéo 246
Conclusion 247
Cross-plateforme Video 8 : Gérer des données SQLite sous Windows Phone et Android avec MvvmCross 247
Les données en cross-plateforme 247
La vidéo 248
Le Bonus 248
Cross-plateforme vidéo 9 : Envoi de mail avec WPF et Android 248
Cross-plateforme : quels OS et technologies ? 248
La vidéo 250
Le Bonus 250
Cross-plateforme vidéo 10 : Codez comme un Ninja ! 250
Ninja Coder 250
La vidéo 251
Conclusion 251
Cross-Plateforme 11 : Swiss, Tibet et Multibinding, Rio, le tout sous Android et Xaml 251
Réinventer le binding 251
Le cross-plateforme : l'assurance anti siège éjectable 252
La vidéo 253
Conclusion 253
Cross-Plateforme Vidéo 12 : Injection de code natif (WinRT/Android) 254
Injection de code natif 254
La vidéo 254
The End ? 255
Conclusion 255

Cross-Plateforme : L'index détaillé des 12 vidéos de formation offertes par Dot.Blog – Vidéos 1 à 6 256
8 heures de vidéos gratuites ! 256
Vidéo 1 : Introduction au framework MvvmCross v3 Hot Tuna 256
Vidéo 2 : Services et Injection de dépendance 257
Vidéo 3 : Liste bindable, plugins, ImageView, item template 257
Vidéo 4 : Convertisseurs de valeur, Commandes et Navigation 258
Vidéo 5 : Recherches de livres avec l'API Google Books 259
Vidéo 6 : Amélioration de l'application de recherche Google Books 260
Cross-plateforme : Index des Vidéos gratuites 7 à 12 261
8 heures de vidéos gratuites ! 261
Vidéo 7 : Géolocalisation, géo-encodage, messagerie 261
Vidéo 8 : Gérer des données avec SQLite 263
Vidéo 9 : Gérer le grand écart Android / WPF 263
Video 10 : Codez comme un Ninja 265
Vidéo 11 : Multibinding, Swiss Binding, Tibet Binding, Rio 265
Video 12 : Injection de code natif dans le noyau, méthode directe et création de Plugin 266
Le code source des 12 vidéos sur le cross-plateforme 268
L'IoC dans MvvmCross – Développement Cross-Plateforme en C# 268
Présentation 268
Inversion de Contrôle et Injection de Dépendance 268
De quoi s'agit-il ? 269
Les problèmes posés par l'approche « classique » 270
Ce qui force à utiliser une autre approche 270
La solution 271
Les implémentations de l'IoC 271
L'IoC dans MvvmCross 273
vNext puis V3 279
Conclusion 279
Rule them all ! L'avenir du futur – Le smartphone devient PC tout en un… 280
Un smartphone qui remplace totalement le PC 280
Tout ça pour Angry Bird et un texto à ses potes n'est-ce pas du gâchis ? 281
Ubuntu pour Android 281
La fin des PC 283

Les vidéos de présentation 284
Conclusion 284
Avertissements 286
E-Naxos 286

Introduction

Bien qu'issu des billets et articles écrits sur DotBlog au fil du temps, le contenu de ce PDF a *entièrement été réactualisé* lors de la création du présent livre PDF en octobre / novembre 2013. Il s'agit d'une version inédite corrigée et à jour, un énorme bonus par rapport au site Dot.Blog ! Un mois de travail a été consacré à la réactualisation du contenu. Corrections du texte mais aussi des graphiques, des pourcentages de part de marché évoquées, contrôle des liens, et même ajout plus ou moins longs, c'est une véritable édition spéciale différente des textes originaux toujours présents dans le Blog !

Les billets n'ont pas été réécrit, ils peuvent donc parfois présenter des anachronismes sans gravité, mais tout ce qui est important et qui a radicalement changé a été soit réécrit soit a fait l'objet d'un note, d'un apparté ou autre ajout.

C'est donc bien plus qu'un travail de collection déjà long des billets qui vous est proposé ici, c'est une relecture totale et une révision et une correction techniquement à jour au moins de novembre 2013. Un vrai livre. Gratuit.

Astuce : cliquez les titres pour aller lire sur Dot.Blog l'article original et ses commentaires ! Tous les liens Web de ce PDF sont fonctionnels, n'hésitez pas à les utiliser !

Présentation de l'édition 2013/2014

Le développement cross-plateforme est une vieille chimère… Toutefois, à force de recherches, d'essais, et grâce à l'évolution permamente du tooling, J'ai réussi à rendre ce rêve accessible à tous ! Certes je n'ai pas inventé MvvmCross, ni n'ai participé à la création de MonoTouch, mais les outils ne sont rien sans une méthode fiable et testée pour les utiliser convenablement. Assembler ce qui est épars, donner un nouveau sens à un ensemble d'outils dispersés, et rendre le tout accessible et compréhensible à tous et gratuitement, c'est un travail qui a son importance et dont j'avoue tirer une certaine fierté. Il y a beaucoup de gens intelligents qui parfois publient des choses, mais il n'y a qu'un seul Dot.Blog.

Les articles corrigés sont publiés dans un ordre logique offrant une progression cohérente en accord avec ce qu'on peut attendre d'un livre et qui ne reflète donc pas forcément l'ordre dans lequel ils ont été publiés.

La méthode de développement cross-plateforme présentée dans ce livre a été créée au départ en 2011/2012. Elle a forcément connu des améliorations au fil du temps, mais aucun changement de philosophie. Il est donc toujours aussi intéressant de parcourir cette méthode de sa génèse à ses derniers perfectionnements. D'autant que

les billets les plus anciens ont été revisités afin d'en supprimer ce qui n'a plus d'intérêt.

N'oubliez pas que la chaîne YouTube « TheDotBlog » vous propose une formation gratuite en 12 vidéos sur le cross-plateforme exploitant les techniques les plus récentes. Ces vidéos vous seront d'autant plus compréhensibles que vous aurez suivi ici le cheminement des idées ayant prévalu à la mise en place de cette stratégie.

Stratégie de développement Cross-Platform – Partie 1

Développer aujourd'hui c'est forcément développer cross-plateforme. Mieux, c'est développer cross-form factor... Android, iOS, Windows 7, Windows 8.1, sur PC, tablette, smartphone, phablettes... Un vrai casse-tête qui peut couter une fortune si on n'adopte pas dès le départ la bonne stratégie et les bons outils. C'est une stratégie opérationnelle avec ses outils que je vais vous présenter dans cette série de billets dont voici la première partie. [Note : écrit en juillet 2012 cet article a été totalement actualisé en octobre 2013, que cela concerne les chiffres cités aussi bien que le fond de la démarche proposée. Il s'agit d'un « bonus » important de cette version Livre PDF par rapport au Blog]

Cross-platform, cross-form factor

Développer c'est toujours prendre en compte les extensions et évolutions futures. Donc adopter une stratégie de développement "propre", claire, maintenable, évolutive. Cela réclame déjà beaucoup de méthode, de savoir-faire et d'expérience.

Mais développer aujourd'hui c'est bien plus que ça...
C'est aussi **prendre en compte la diversité des plateformes et des form factors**. Et là, cela devient tout de suite un numéro d'équilibriste, même pour ceux qui maitrisent les impératifs cités.

Nous avons tous nos choix, nos préférences techniques, mais nous avons tous, en professionnels, l'obligation de satisfaire la demande des utilisateurs, des clients et **d'intégrer dans nos stratégies de développement la réalité du marché**.
La réalité du marché c'est aujourd'hui par exemple 68.1% [été 2012] de parts pour Android sur le segment des smartphones en Europe [79% au second trimestre 2013]... Qui peut négliger près de **80% des utilisateurs européens de smartphone** ? La réalité du marché c'est l'IPad qui malgré une chute vertigineuse détient toujours 32.4% du marché en août 2013. Peut-on l'ignorer ? La réalité du marché ce sont des centaines de millions d'utilisateurs de Windows XP et Windows 7 sur toute la planète (34% d'XP et 46% de Windows 7 soit **80% du marché Windows en août 2013**). Qui peut se permettre de les ignorer ?

Enfin, la réalité du marché c'est Windows 8, du smartphone au PC, arrivant avec retard sur ses concurrents mais avec l'avantage d'une cohérence de gamme très séduisante. Qui peut se permettre le luxe d'ignorer WinRT qui équipe tous les nouveaux PC depuis l'annonce du 26 octobre 2012 ? Même si l'accueil frileux de WinRT a poussé Microsoft à faire une version 8.1 rapidement, et même si les parts de marché de Windows 8 sont seulement de 7.41% en août 2013, est-il possible de totalement *zapper* cet OS et de ne pas l'intégrer dans les plans de développement de logiciels qui verront le jour dans un ou deux ans et qui seront en service au moins pour 3 à 5 ans ? Au moins par cohérence avec les moyens informatiques mis en œuvre en entreprise et reposant généralement sur des OS et outils Microsoft.

Bref, **développer aujourd'hui** c'est intégrer le design, MVVM, l'évolutivité, la maintenabilité, mais aussi les différentes plateformes qui se sont imposées dans **un partage du marché durable où Windows n'est plus le seul et unique OS** car **le PC n'est plus le seul et unique type d'ordinateur** à prendre en considération.
C'est à cette réalité qu'il faut se préparer. Non, c'est à ce défi technologique qu'il faut être déjà prêt ! (avec le recul, plus d'un an après, ô combien j'avais raison de vous donner ce sage conseil !)

Comme tous les ans, j'avais rêvé des vacances les pieds en éventail. J'avais rêvé de pouvoir boucler le mixage de quelques un des dizaines de morceaux que j'ai composés ces dernières années sans avoir le temps de les terminer proprement (il y a bien quelques mix sur ma page SoundCloud mais c'est peu et pas forcément représentatif), j'avais prévu mille et une chose futiles ou non. Hélas cet été comme les autres, je me dis que je verrai ça cet hiver. Hélas pour mes rêves mais heureusement pour vous (en tout cas je l'espère !) je me suis mis en tête de formaliser ma stratégie de développement pour l'année à venir et d'écrire cette série de billets pour vous présenter une solution viable de développement cross-platform et cross-form factor !

My Strategy, for today

Ce que je vais vous présenter est le fruit de mon travail, de ma réflexion sur ce sujet et de mes tests et essais ces derniers mois de nombreux outils, frameworks, unités mobiles, etc. C'est "ma stratégie", et "pour aujourd'hui", car le marché et les technologies évoluent et peut-être que l'année prochaine je vous proposerai une autre méthode ou des aménagements … ou pas.

> *Plus d'un an après, en effet ma stratégie a légèrement évolué avec le tooling disponible. Je vous propose désormais une démarche encore plus simple reposant sur MvvmCross et Xamarin. Toutefois la méthode décrite ici reste valable même si on peut bien entendu l'améliorer en tirant profit des nouveaux outils à notre disposition.*

En tout cas cette stratégie est pérenne, c'est l'un des critères ayant présidé à son élaboration. Il y a aura peut-être mieux à un moment ou un autre, mais pour les mois à venir et les développements à lancer elle restera un très bon choix.

C'est une stratégie qui s'accorde aussi bien aux développements internes en entreprise qu'à une logique éditeur où la couverture maximale du marché est un impératif vital. Le développeur y trouvera son compte autant que le DSI devant intégrer tablettes et smartphones dans son environnement.

J'ai testé de nombreuses approches, comme par exemple ce qu'on appelle "l'hybride" ou même HTML 5. Les méthodes et outils que j'ai fini par assembler dans un tout cohérent dans la stratégie que je vais développer ici ont tous la particularité de répondre à de très nombreuses contraintes. Les outils ou méthodes que j'ai écartés ne sont pas forcément "mauvais" en eux-mêmes, mais ils ne répondent pas à toutes les contraintes que je m'étais posées.

Il s'agit donc bien de "ma" solution pour les développements à lancer "aujourd'hui", entendez par là pour les mois à venir (ce qui reste vrai un an après).

Une contrainte forte : la cohérence

Parmi ces contraintes, la plus forte était la cohérence.

Je veux dire que ma stratégie ne devait pas ressembler à un tableau de Miro ou de Picasso, où la réalité est déstructurée à l'extrême. Ici, en bon ingénieur, je voulais de l'ordre, de la logique, de la lisibilité. Je voulais un puzzle de paysage paisible où chaque pièce s'emboite avec l'autre, l'ajustement devant être parfait ou presque pour former un tout ayant un sens et de la rigueur.

Cette contrainte m'a fait écarter de nombreuses solutions tout simplement parce que la pièce ne trouvait pas sa place dans le puzzle final.

Les solutions écartées peuvent ponctuellement être utilisées avec succès par certains d'entre vous. Ma stratégie n'est pas exclusive, elle admet que d'autres solutions ponctuelles fonctionnent. Pas exclusive mais unique : elle n'accepte que ce qui entre dans le puzzle final avec harmonie. **Son originalité, son unicité c'est sa cohérence**.

Un But : Unifier ce qui est épars

Le but que je m'étais fixé est issu d'une constatation : **il n'existe aucun point commun ni passerelle entre les différentes plateformes à prendre en considération** sur le marché : iOS, Android, Linux, Windows 7, WinRT. Aucun de ces OS n'a prévu une quelconque compatibilité avec les autres. Rien. Et c'est même conçu "pour".

Unifier ce qui est épars semble un but inaccessible dans une telle condition.

Mais ce que les OS n'ont pas prévu (ni souhaité), ce lien, **cette passerelle, on peut la créer grâce à des outils**, à du soft.

Le tout étant de trouver le bon package d'outils et de méthodes qui permettent de créer ce lien immatériel unifiant des plateformes si différentes.

Un principe : Concentrer la compétence

Une autre constatation s'impose quand on a commencé à réfléchir à la question et qu'on a commencé à tester différentes choses : **aucune technologie unique existante n'est capable de régler à elle seule le problème dans sa totalité**.
La solution ne peut passer que **par l'utilisation habile de plusieurs outils et méthodes**. Il doit y en avoir un minimum dans la solution. Le foisonnement des outils, des langages, des APIs pour faire un logiciel je n'ai jamais aimé cela. On ne peut être expert en tout et forcément on s'éparpille.

Comme un Laser concentre la puissance de quelques photons inoffensifs pour en faire une arme redoutable d'efficacité et de précision, la stratégie qu'il fallait concevoir se devait de **concentrer le savoir-faire du développeur pour maximiser son efficacité et minimiser la dispersion de son expertise**.

Les cibles visées

Des plateformes, pas des technologies de développement

Les cibles visées sont des plateformes globales, je ne parle pas ici de plateforme de développement, de langages ou de technologies particulières (comme WPF ou Silverlight).

Les cibles

Comme cela a déjà été évoqué ici en divers endroits, les cibles visées par cette stratégie de la “grande unification” sont les suivantes :

- iOS (tablettes et IPhone)
- Android (idem)
- Windows XP, Vista, 7
- Windows 8.x (WinRT sur tablettes, smartphones et PC)
- Windows Phone 8.x (la version 7 est définitivement morte)
- Linux (c’est un plus, pas une obligation)

Apple

Mon désamour pour Apple n’a pas changé, mais il est aujourd’hui en tout cas indispensable de prendre en compte la réalité du phénomène IPad/IPhone. Bien que perdant pieds face à Android, Apple ne mourra pas dans les mois à venir. On pourrait discuter des heures des parts de marché que les tablettes WinRT prendront ou bien du succès des tablettes Android qui finiront peut être par éliminer Apple du marché comme les smarphones Android l’ont fait avec l’IPhone. Mais tout cela est trop spéculatif (d’autant qu’en octobre 2013 effectivement Android a enfin dépassé Apple aussi sur le marché des tablettes alors que WinRT n’a pas su séduire). J’ai préféré **prendre la réalité telle qu’elle est**. Intégrer l’IPad dans la stratégie était une obligation. De toute façon, WinRT et Android sont aussi pris en compte... La stratégie finale est tellement “englobante” qu’elle supportera tous les réajustements du marché au moins à courts et moyens termes (L’année écoulée depuis l’écriture de ses lignes et les modifications du marché montrent clairement que la stratégie proposée était très robuste puisqu’elle continue à être valide !).

Android

Android est une réalité difficile à intégrer pour certains car le phénomène est apparu très vite en venant de là où on ne l’attendait pas... Alors il faut le temps que cela “monte” au cerveau. Une fois ce cheminement effectué force est de constater qu’avec presque **80% du marché des smartphones en Europe**, Android est devenu tout simplement **incontournable**. Peu importe ce qu’on en pense, peu importe le succès à venir de Windows Phone 8 (succès qui depuis sa sortie reste modeste). Android est là pour rester. En douter serait une grossière erreur. Et un an après avoir écrit ces lignes, je crois que la réalité me donne raison…

Windows classique

Windows XP, Vista et 7 forment une triade aux pourcentages relatifs en évolution (au profit de 7) mais reste un **socle solide de plusieurs centaines de millions de PC dans le monde** qui ne va pas changer avec la sortie de Windows 8 (et qui n’a pas changé !). Et Windows 7 étant encore meilleur que XP, je parie qu’il aura la vie aussi

longue que son grand frère, notamment en entreprise. Dès lors, difficile de faire l'impasse. Mais là on enfonce des portes ouvertes.
En octobre 2013 XP et 7 représentent **80% du marché Windows** alors que Windows 8 dépasse à peine 7%. Une fois encore mes prédictions ne vous ont pas trompés.

Windows 8

Windows 8 avec WinRT a été la nouveauté de la rentrée 2012. Windows 8 est assurément un bon OS, et pour l'utiliser depuis un moment je peux assurer que l'efficacité technique est au rendez-vous. Je reste toujours circonspect concernant certains choix mais je ne doute pas un instant que la grande cohérence de la plateforme ne finisse par l'imposer. Une même API du smartphone au PC en passant par les tablettes, c'est unique sur le marché et c'est séduisant. D'autant que WinRT peut se programmer en réutilisant le savoir-faire .NET. Ne pas cibler WinRT serait idiot et dangereux. On sent bien que Microsoft a du mal à imposer cette plateforme mais qui peut en prédire l'arrêt alors même que Windows 8.1 est là pour nous dire que Microsoft (en tout l'actuelle Direction - en partance il est vrai) ne compte pas changer d'avis rapidement ?

Windows Phone

Windows Phone est une plateforme qui si elle n'a pas rencontré un succès immédiat avec la version 7 pourrait bien profiter du mouvement positif autour de Windows 8 (même si ce mouvement reste modéré à ce jour). Après tout c'est une belle plateforme, efficace, riche de nombreuses applications, ayant le même look que Modern UI (alias Metro Style) puisque c'en est la première utilisation. Certes la cassure de compatibilité du matériel intervenue entre la version 7 et la 8 a encore plus refroidit le marché. Mais selon les circonstances il sera peut-être intéressant d'intégrer Windows Phone dans la logique de développement. Pour l'entreprise il coûte moins cher d'offrir des smartphones Windows que d'engager des équipes spécialisées et coûteuses pour maintenir de l'iOS ou de l'Android. Un salarié qualifié est un investissement lourd et sans fin. L'achat d'un matériel compatible avec Windows s'amorti comptablement en très peu de temps. Selon ce que vous développez pourquoi ignorer ces clients potentiels ? Les entreprises devront bien adopter une stratégie vis-à-vis des unités mobiles et la cohérence proposée par Microsoft pourrait alors s'avérer payante. Si Windows Phone n'est pas dans ma liste de priorités, cela serait une bonne idée de l'intégrer dans la stratégie cross-plateforme que je vous propose pour toutes les raisons évoquées.

Linux

Linux, il m'a toujours amusé. Depuis sa création j'entends ses partisans m'annoncer le "grand soir" pour demain, un peu comme Lutte Ouvrière. Ce n'est jamais arrivé. Néanmoins, et j'en avais parlé dans ces colonnes, **Android c'est Linux**, une base identique à celle de Ubuntu. Et le succès du premier fait que ce dernier tente

désormais de revendiquer ses liens de sang avec l'OS de Google. La récente tentative de *crowdfunding* pour le projet Edge (qui n'était qu'une énorme pub ne coutant rien) prouve en tout cas que les velléités d'Ubuntu sont bien réelles. Leur idée de pousser un OS de plus fonctionnant avec HTML ne me convainc absolument pas, on n'en peut plus de tous ces OS ce n'est pas pour ouvrir les bras à un de plus sur le marché. En revanche le mariage intelligent entre Ubuntu et Android pour en faire des stations de travail complètes me semble plus prometteur.
Après tout, quand une majorité d'Européens possède un smartphone Android, voire une tablette du même OS pour certains, il ne reste plus qu'à leur dire la vérité : c'est du Linux... Alors pourquoi pas un PC sous Ubuntu ? Finalement vous aurez un ensemble cohérent...

Le coup sera peut-être gagnant cette fois-ci. En tout cas si un jour Linux doit se faire une place sérieuse sur le marché, c'est maintenant que cela va se jouer (2013/2014). Si Linux rate cette fantastique occasion, autant dire qu'il restera ce qu'il est depuis toujours : un OS de serveur Web pour ceux qui ne veulent pas payer une licence Windows et IIS. Même si cette plateforme n'entre pas dans les priorités de ma stratégie, l'émergence d'Ubuntu comme père spirituel d'Android dans un style Star Wars "Android, je suis ton père !" est suffisamment troublant pour se dire que si cela est possible, il serait judicieux d'intégrer un lien Linux avec les autres plateformes ciblées au départ. D'où la présence de Linux dans cette liste. Toutefois cela reste une simple possibilité que je ne couvrirai pas dans l'immédiat mais reste parfaitement envisageable dans le cadre de ma stratégie.

Un rêve ? Non, une réalité !

Quand on regarde cette liste à la Prévert on se dit que c'est un doux délire, qu'il sera impossible de trouver un moyen de créer des applications pour toutes ces plateformes à la fois sans que cela ne coute une fortune en formation et en personnel qualifié...

Et pourtant !

HTML 5 ? Non, vous n'y êtes pas.

Pas de bricolage de ce genre chez moi. Et puis HTML via un browser ce n'est pas ça sur les unités mobiles. J'ai déjà argumenté ici plusieurs fois que cette norme était dépassée par les évènements, que depuis qu'elle a forké c'est encore plus évident, et que sur les unités mobiles c'est le natif qui est roi. L'universalité visée par ma stratégie m'a forcé à rejeter (avec joie et soulagement) cette fausse bonne idée (ou vraie mauvaise idée) qu'est HTML 5. Et puis, vous l'avez remarqué et je l'ai dit : je vise des cibles en tant que plateformes globales, pas une utilisation particulière de

l'informatique que sont les browsers. L'insuccès du projet EDGE de Ubuntu prouve que le marché n'est d'ailleurs pas prêt à accueillir un OS de plus basé sur HTML. Donc exit HTML 5 comme solution universelle. Donc out de ma stratégie.

Je ne voulais pas concevoir une stratégie fondée sur des sables mouvants. Il ne s'agit pas d'un rêve, juste d'ingénierie.

Dans la partie 2 vous découvrirez l'écriture d'un projet fonctionnel tournant sur plusieurs des plateformes ciblées avec un seul code et un seul langage.
(Le suspense est insoutenable non ?)

Et le Web ?

Toutefois certains se diront que ma liste pour aussi longue qu'elle soit ne prévoit rien pour le Web.

Le Web est une cible bien à part, qui, on le voit bien à ma liste, ne compte plus que pour une parcelle dans toute cette débauche de nouvelles plateformes. Mais le Web est éternel... **Eternel mais plus universel**.

Si Microsoft a perdu la suprématie absolue dans le monde des OS avec la naissance d'iOS et Android, le Web a perdu son universalité avec le succès du natif sur ces machines.

Mais je pourrais ajouter à ma liste, par gourmandise, ASP.NET. Pour créer des applications Web professionnelles avec de vrais langages, une vraie plateforme et du HTML 3/4 pour être sûr de passer sur tous les browsers. Cela restera tout à fait compatible avec ma stratégie.

Il reste qu'il ne faut pas confondre les choses. Ma démarche cible des plateformes techniques différentes. Certes on peut voir le Web comme l'une d'entre elles. Mais du point de vue du développement ce n'est qu'une façon possible parmi d'autres d'utiliser Internet. Je ne mets pas l'accent sur le Web ici pour cette raison. Le Web n'est pas un OS, il n'est pas une plateforme définies clairement, ce n'est qu'une utilisation d'Internet via des logiciels spécifiques (les browsers). Le succès des applications natives sur unités mobiles ou même des données dans le Cloud (DropBox, SkyDrive...) nous prouvent qu'**on peut fort bien aller vers toujours plus d'importance d'Internet tout en allant vers moins d'importance pour le Web**...

Les effets néfastes de la canicule sur un esprit surmené ?

Bref je vise le monde entier, toutes les plateformes, tous les form-factors ! Façon Cortex et Minus :

> ***"Gee Brain, what do you wanna do tonight ?***
> ***The same thing we do every night Pinky, try to take over the world! "***
>
> ***("hey Brain, qu'est-ce qu'on va faire cette nuit ? – La même chose que nous faisons toutes les nuits Pinky, tenter de conquérir le monde !")***

Quand nous aurons passé cette introduction vous verrez que tout cela est possible et je vous montrerai comment le faire... Si folie il y a c'est peut-être d'avoir cherché une telle stratégie, mais dans les faits elle est on ne peut plus raisonnable et terre-à-terre.

La stratégie

Entre humour et réalité (rappelons que je suis censé être en vacances et qu'il faut bien prendre les choses avec un peu de légèreté) vous me connaissez assez maintenant pour savoir que je n'ai pas tapé toute cette tartine juste pour un joke.

Je préfèrerai siroter un martini "Shaken, not stirred !" - les amateurs de James Bond auront compris - le tout en appréciant le déhanchement de quelques créatures pulpeuses à Acapulco qu'être rivé derrière mon écran, climatisation à fond et volets fermés pour éviter que les serveurs ne grillent en ces jours de canicules. Franchement. Si, croyez-moi.

Après cette introduction essentielle sur mes motivations et le véritable sérieux qui a présidé à tout cela, après avoir précisé les cibles visées, les contraintes, le but, il est temps de présenter cette stratégie et son contenu.

Divide ut regnes !

Nicolas de Machiavel, reprenant ici une expression latine prêtée notamment à Philippe de Macédoine, savait qu'il tenait là l'une des clés du pouvoir. Diviser pour mieux régner, ça marche ! Ça n'a qu'un temps parfois, mais c'est un bon vieux truc qui marche toujours.

Nous aussi, mais en poursuivant des buts plus nobles, nous pouvons utiliser cette méthode.

Divisez pour mieux développer.

C'est avant tout séparer les tiers convenablement.

La base de ma stratégie passe par une segmentation rigoureuse du code d'une application.

- Le premier segment est un serveur Web qui centralise le code métier
- Le second segment consiste en la définition des ViewModels dans un projet portable.
- Le troisième segment ce sont les UI.

Selon l'importance des projets et leur nature, et grâce aux avancées du tooling on pourra fusionner les deux premiers segments. Quand j'ai écrit ces lignes l'année dernière certains doutes quant à l'avenir de Xamarin et le manque de framework Mvvm adapté m'obligeaient à la prudence de segmenter le code métier en le protégeant dans un serveur Web, le rendant totalement insensible aux changements de mode, d'OS, etc. Aujourd'hui on peut se diriger avec plus de confiance vers une solution totalement native. Les Portable Class Libraries de Visual Studio sont une réelle avancée qui permet d'écrire un seul code portable pour toutes les cibles visées. Il n'est donc plus indispensable de protéger le code métier sur un serveur, les PCL s'en chargent.
Toutefois la méthode proposée reste parfaitement valide et son côté « protecteur » reste parfaitement sensé. On peut juste faire plus simple dans la majorité des projets avec le nouveau tooling. Mais dans les projets entreprise (LOB) le découpage avec serveur métier reste parfaitement valide.
On suivra avec intérêt à ce sujet les 12 vidéos de présentation de MvvmCross publiées sur YouTube durant l'été 2013 (chaîne « TheDotBlog ») qui proposent une véritable formation au développement cross-plateforme utilisant les dernières technologies.

MVVM

Séparer du code de l'UI nous savons le faire depuis un moment, ça fonctionne bien si on sait trouver "sa voie" et la bonne librairie pour maitriser sa mise en œuvre.

Ici je pousserai encore plus la séparation imposée par MVVM : les ViewModels et les Models seront définis dans un projet commun que j'appellerai le "noyau" alors que les Vues seront définies dans les projets gérant les UI des différentes plateformes / form factors.

Un langage et un seul

J'aime C# et il se trouve que je l'aime aussi pour sa versatilité et sa capacité à être une base solide pour développer tout type de projets.

C'est donc lui qui sera le fil unificateur en arrière scène.

Une seule plateforme

C# n'est pas un électron libre. Derrière lui se tient aussi le framework .NET. Il sera lui aussi à la base de ma construction pour des milliers de raisons. S'il n'en fallait qu'une : il est complet et permet de tout faire.

Un seul EDI

Pourquoi s'enquiquiner avec différents EDI et langages. Qui dit C# et .NET dit Visual Studio, l'EDI le mieux fait. C'est lui qui sera utilisé de bout en bout. Pour les UI Xaml (WPF, Silverlight, WinRT) j'utiliserai aussi Expression Blend. Et pour les UI iOS et Android je tirerai partie du concepteur visuel Xamarin qui se plug dans VS.

Une seule version…

Armé de C# et de .NET il ne va guère être facile d'attaquer toutes les plateformes que j'ai listées plus haut...

... Sauf si on trouve un moyen de programmer pour iOS et Android en C#...

Et ce moyen existe.

Il s'agit de MonoTouch et MonoDroid dont j'ai déjà parlé et qui s'appellent tout simplement aujourd'hui Xamarin.IOS et Xamarin.Android. Deux petites merveilles. MonoDroid qui se plugue dans VS (avec éditeur visuel intégré désormais), MonoTouch qui s'utilise avec MonoDevelop (copie sous Mono de VS) sous Mac, le tout se programmant en C# sur une base Mono de niveau .NET ¾ (qui supporte Linq et await/async donc tout ce qui est moderne dans C#). Les dernières versions de Xamarin.IOS permettent la mise en place sur PC, le Mac restant indispensable pour faire tourner l'émulateur et donc faire du debug.

Conçus et vendus par Xamarin dont j'ai aussi parlé ici, **ces deux produits permettent de satisfaire toutes nos contraintes**, dont celle de cohérence.

Un seul EDI, un seul langage, un seul framework, et pas moins de onze cibles touchées !

IPad, IPhone, Android smartphone, Android Tablette, Silverlight, WPF, WinRT smartphone, WinRT tablette, WinRT PC, ASP.NET, et même Linux avec MonoDevelop fourni avec MonoTouch ou MonoDroid.

Onze cibles unifiées grâce à Visual Studio et le plugin Xamarin.

Cela semble régler le problème mais il n'en est rien.

En tout cas pas totalement. Dans une telle stratégie il n'y a pas que les outils qui comptent. Il y a aussi la façon de s'en servir...

On l'a vu, je vais utiliser les principes de MVVM, méthode simple et redoutablement efficace pour produire des logiciels bien architecturés.

MVVM se base sur le Binding... Or, point de Binding sous Objective-C ni le Java sous Eclipse de Android.

Déception ?

Non. MonoTouch et MonoDroid n'y peuvent rien (ils n'ont rien à voir avec MVVM), mais en revanche MVVMCross lui peut nous aider à la faire !

Une librairie Cross-plateform

Le ciment de base c'est C# et le framework .NET. Dans leur version Microsoft et leur version Mono déclinées dans MonoTouch et MonoDroid.

Avec ce ciment nous pouvons construire les murs porteurs de notre stratégie.

Mais pour le toit, il faut quelque chose capable d'englober toutes ces versions en proposant un toolkit MVVM se chargeant des différences entre plateforme.

Ce toolkit existe, c'est MVVMCross, projet Github qui **supporte MonoDroid, MonoTouch, Silverlight, WPF, Windows Phone, le mode console et WinRT**.
Bien que jeune, la librairie est fonctionnelle et déjà bien testée. Un an après (et 12 vidéos de formation évoquées plus haut), MvvmCross s'avère un framework puissant, évoluant avec les besoins et couvrant ceux-ci de manière très satisfaisante. Plus qu'un framework MVVM c'est un framework MVVM cross-plateforme gérant des plugins permettant de centraliser réellement le code métier sans avoir à le bricoler ou à l'adapter pour chaque cible.

Un tooling ultra simplifié

Finalement, c'est un tooling ultra simplifié que je vais utiliser. Pour résumer il est formé de :

- Visual Studio
- MonoDroid
- MonoTouch
- MVVMCross

Un seul EDI. Un ou deux plugins (selon les cibles que vous voulez supporter). Un seul framework MVVM.
Si vous ne supportez pas d'emblée les unités mobiles il n'y a donc que Visual Studio et la librairie MvvmCross…

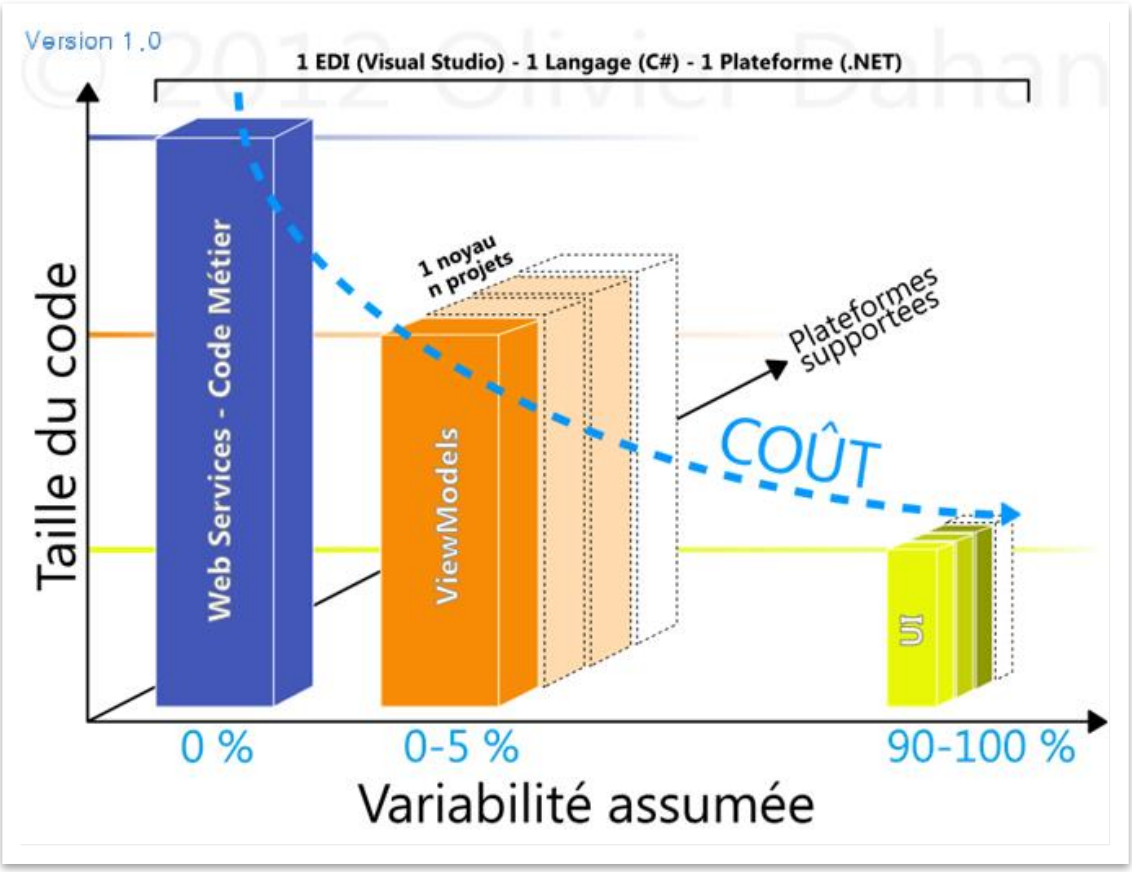
Autant dire que mon souhait de réduire au strict minimum les outils, langages et EDIs est parfaitement réalisé.

Pour développer il faut de toute façon au minimum un EDI et un langage, et au moins une librairie de type MVC ou MVVM proposant de l'IoC. Au bout du compte, je n'aurai ajouté que MonoTouch et MonoDroid pour pouvoir compiler et tester les applications destinées aux plateformes non Microsoft. Tout le reste est le minimum vital même pour un seul projet ne ciblant qu'une plateforme.

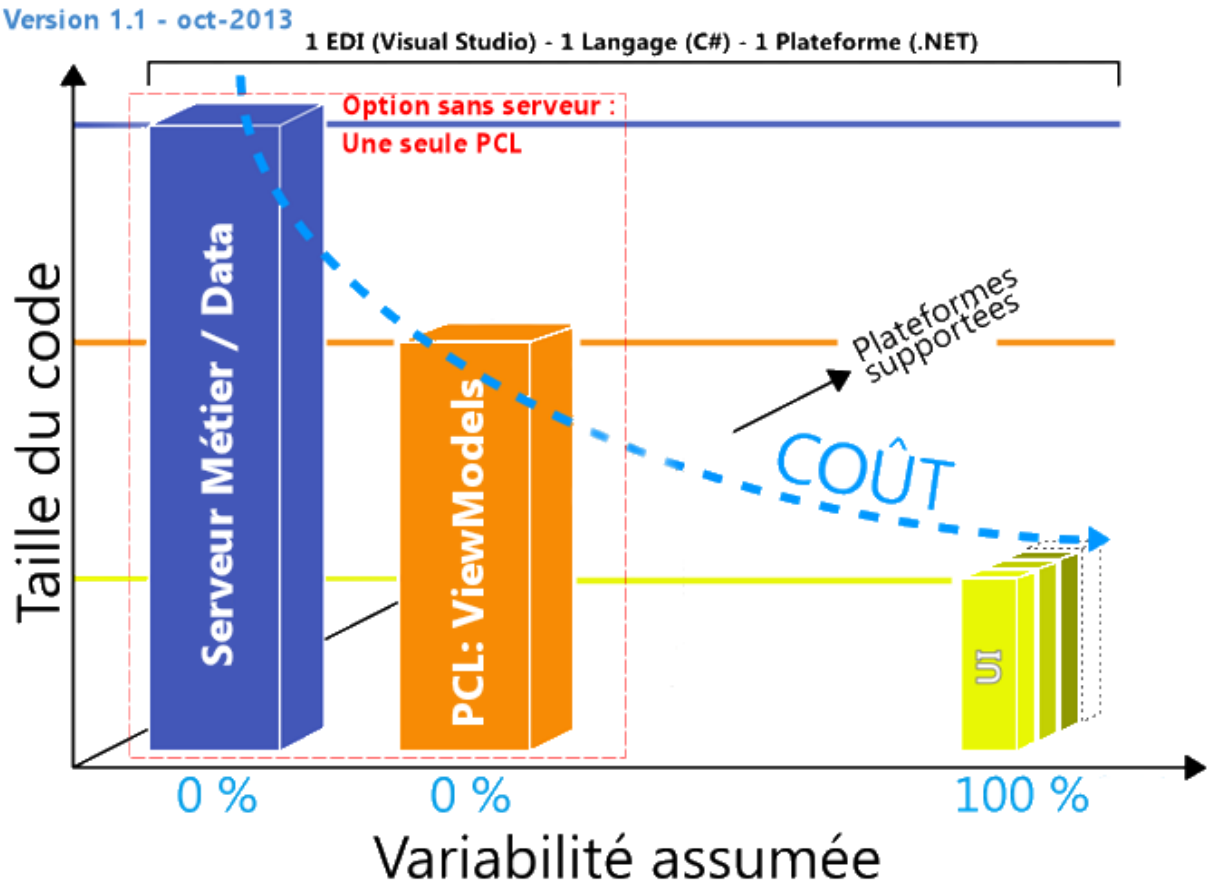
Le strict minimum donc. Pari gagné sur ce point.

Explication par l'image

Un dessin valant mille discours :



Le graphique ci-dessus présente la version 1.0 de ma stratégie cross-plateforme. La version révisée d'octobre 2013 ressemble à ce qui suit, on note la simplification extrême grâce aux PCL ainsi que la fin d'une variabilité possible (même faible) au niveau des ViewModels :



Le serveur "métier" ou la librairie PCL

Une partie essentielle de la stratégie se trouve ici : créer un serveur contenant le code métier ou le regrouper (selon la variante plus moderne de ma stratégie) dans une librairie PCL, ce qui revient au même dans le principe (protection de ce code et **zéro réécriture**). Ici je parlerai de serveur métier, sous-entendu un serveur capable de fournir des services évolués et non pas seulement des opérations CRUD sur les tables d'une base de données. Un serveur WCF Data Services est un bon exemple du type de serveur dont je veux parler même si l'interface s'effectuera plutôt en XML ou JSON via http pour une portabilité maximale (OData par exemple).
Le code placé sur ce serveur doit être le plus dense possible, il doit prendre en charge le maximum d'intelligence, de règles métiers, de fonctions de haut niveau. On appliquera exactement la même logique à la version plus moderne de la stratégie exploitant les Portable Class Librairies de VS qui elles fusionnent toute l'intelligence – sauf dans le cas où on souhaitera mixer les deux approches (applications LOB notamment).

Tout l'art d'ingénierie qui se trouvera derrière le serveur consistera à savoir effectuer un tel découpage des fonctions de haut de niveau et de savoir les exposer en créant une API logique, facile à comprendre et efficace. Par essence, et puisque nous parlons ici de créons des applications natives et non des pages Web, le serveur devra autant

que faire se peut rester “stateless”, le “stateful” est géré par les ViewModels côté client.

L’utilisation des PCL dans la version améliorée de ma stratégie permet d’éviter en grande partie ce lourd travail de création du serveur métier.

On notera que le “serveur métier” peut dans la réalité se décomposer en plusieurs applications serveur, en plusieurs services. Cela peut même être essentiel pour répartir la charge ou pour disposer de serveurs rapides au plus près des clients. Le ou les serveurs métiers peuvent même être dupliqués pour gérer une montée en charge. Les applications serveurs peuvent elles-mêmes faire appel à d’autres services locaux ou distants, il n’y a aucune limite ni contrainte imposée par la stratégie présentée. Si l’approche plus simple par les PCL évite pour la majorité des projets d’avoir à mettre en place un serveur métier, certaines applications réclameront de toute façon la présence d’un serveur et « l’intelligence » (le code métier) qui s’y trouvera bénéficiera de la puissance et de la « *scalability* » propres aux architectures serveur que le seul code PCL exécuté en natif côté client ne pourrait offrir.

Si le serveur métier n’est pas indispensable aujourd’hui il s’avère presque impossible à éviter dans les applications d’entreprise (LOB). Il reste donc nécessaire pour les projets de ce type de suivre la logique d’un serveur métier car vous le verrez au travers de l’exemple réel de la Partie 2 c’est aussi la clé de voute de la pérennité du code (même si les PCL peuvent assurer seules aujourd’hui ce rôle, l’approche serveur comme protection du code contre les changements de mode et d’OS reste valide)… Vous comprendrez mieux quand j’aborderai l’exemple concret.

On n’oubliera pas toutefois que les PCL peuvent être une alternative à la présence d’un serveur métier mais que dans de nombreux cas ce dernier devra exister pour assurer le partage des données. Y placer le maximum d’intelligence reste ainsi une approche tout à fait rationnelle que l’arrivée récente des PCL ne remet pas en cause. Si les PCL peuvent remplacer totalement le serveur métier dans des applications de type utilitaire ou des applications personnelles sans partage de données, et que cela soit dans une logique éditeur de logiciels ou non, elles ne peuvent assurer la centralisation des données, leur synchronisation ou leur simple mise à disposition (consultation simple par exemple comme un annuaire). En entreprise la nécessité d’accéder aux données s’impose en revanche pour 99% des applications. Mettre en place un serveur métier basé sur code .NET et y placer le maximum d’intelligence reste donc l’approche que je continue à préconiser dans ce cadre d’utilisation.

Un noyau unique

C'est là qu'entre en scène MVVMCross aidé des PCL. Grâce à ce framework et à cette nouvelle possibilité de Visual Studio nous allons créer un projet qui contiendra l'ensemble du code des ViewModels de l'application. Dans l'exemple de cet article que nous verrons plus loin ce projet est créé en Mono pour Android version 1.6 car lors de l'écriture de cet exemple les PCL n'existaient pas et il fallait écrire un projet noyau qu'on dupliquait (par simple copie, sans réécriture de code) pour chaque OS, on choisissait alors le profile .NET le plus restrictif. Les PCL permettent aujourd'hui de centraliser dans un projet réellement unique programmé en .NET 4 tout le code du noyau, les restrictions de .NET sont gérées automatiquement par Visual Studio qui, en fonction des cibles choisies contrôle lui-même les API .NET disponibles ou non. Aucun risque de se tromper donc.

Ce projet "**noyau**" sous forme de PCL peut être utilisé directement par tous les OS cibles. C'est là une simplification énorme par rapport à la « version 1.0 » de cette stratégie de développement qui obligeait à créer des copies du noyau pour chaque cible. Le code était le même, réintroduit d'ailleurs dans chaque projet par un lien et non par copie physique. Une seule version du code existait donc. Les PCL apportent un énorme confort en nous évitant cette gymnastique.

Au final là où nous avions dans la version 1.0 de la stratégie des projets *Noyau.Android*, *Noyau.WinRT*, *Noyau.WindowsPhone*, etc, nous n'avons plus qu'un seul projet PCL généralement appelé MonApplication.Core...
Bien sûr, autant les différents noyaux de la version 1.0 que l'unique PCL de l'approche en version 1.1 donnent naissance à des binaires différents. Mais cela est le travail de VS et des compilateurs, nous, de notre côté nous n'avons écrit qu'une seule fois le code ! Cela apparait de façon encore plus évidente dans la version 1.1 puisqu'il n'existe plus qu'un seul projet noyau.

Dans la version 1.0 on acceptait une très faible variabilité dans les noyaux (généralement de la compilation conditionnelle) car dans certains cas il n'était pas possible d'avoir vraiment le même code notamment dans certains cas où des spécificités natives étaient utilisées.

Dans la version 1.1 utilisant les PCL la stratégie ne tolère plus de variabilité dans le noyau, il est 100% portable et 100% pur. Si des variations s'imposent dans l'implémentation de telle ou telle autre fonction, cela est fait par les plugins MvvmCross qui pratique une forme d'injection de code natif totalement transparente. Je présente très largement MvvmCross dans une série de 12 vidéos déjà évoquées ici et je renvoie le lecteur vers ces dernières pour voir comment cela fonctionne. J'ai aussi écrit un article sur l'injection de code natif qu'on retrouvera sur Dot.Blog facilement.

Grâce à MVVMCross et aux PCL le code de nos ViewModels est le même pour toutes les plateformes visées, mieux **MVVMCross sait ajouter le Binding aux plateformes ne le gérant pas**. Nos classes seront développées comme pour WPF ou du Silverlight, sans aucune différence (sauf l'utilisation de MVVMCross au lieu de MVVM Light ou Jounce par exemple).

Des UI spécifiques

C'est là que les divergences apparaissent, mais seulement arrivé là, en bout de course. En utilisant MVVMCross nous allons créer autant de projets que nécessaires, un par cible visé. Ces projets vont définir les **Vues**. Chaque projet possède en référence la PCL du Noyau.
Bien sûr, ici il faudra faire du spécifique. Mais la mise en page d'une application Android, iOS ou WinRT reste quelque chose de spécifique lié à la plateforme.

Toutefois **grâce à cette stratégie nous avons repoussé au plus loin possible dans le cycle de développement la partie spécifique à chaque OS et nous avons fortement limité l'ampleur du code spécifique**.

C'est là que la stratégie est payante...

Un coût maitrisé

Bien que la liberté soit totale pour la partie serveur le plus souvent la partie métier sera codée en C# dans la PCL noyau, voire dans un serveur .NET afin d'éviter l'éparpillement des langages et plateformes, ce que justement la stratégie proposée vise expressément. Toutefois on peut appliquer la stratégie en partant d'un code Java sur un serveur Apache… rien ne l'interdit. Mais ici je resterai sur les rails de la simplification et de l'intérêt d'utiliser un seul langage, une seule plateforme, de bout en bout. Les DSI comprendront tout l'intérêt économique de cette démarche !

De fait, **l'investissement de développement le plus important va se trouver en un seul exemplaire concentré dans une PCL, voire sur le serveur métier**. **La variabilité assumée pour ce projet central est de 0%**, **il sera valable pour toutes les cibles sans aucune modification.**

Choisir la plateforme (Apache ou IIS) autant que les modalités de transport de données (XML ou JSon) et la façon de sécuriser les accès au service (où aux *n* services du serveur métier) n'a pas d'impact sur la stratégie présentée, je ne développerai pas cet aspect des choses, même s'il est très important car là où un serveur sera utilisé en plus de la PCL il y a généralement déjà une stratégie interne et le choix des serveurs physiques, de leur OS et la façon de partager les données est déjà arrêté. La stratégie

proposée s'adapte donc à toutes les situations fréquemment rencontrées en entreprise, même si, dois-je le répéter, j'en présenterai uniquement que des mises en œuvre simplifiées et cohérentes.

Enfin, nous écrivons des projets spécifiques pour les UI car les machines, les OS, les form factors l'imposent. Mais ici **ce ne sont plus que des Vues fonctionnant avec du Binding sur les ViewModels**... Il n'y a qu'un travail de mise en page, voire d'intégration entre le travail de l'infographiste/designer et celui des informaticiens (les ViewModels). Ici nous acceptons une variabilité de 100%, c'est la règle du jeu...

Mais pour certaines cibles cette variabilité sera souvent moindre. Supporter Windows Phone et WinRT permettra de réutiliser peut-être 60% du code Xaml, voire plus. Par exemple on récupèrera les définitions de style, les animations, les DataTemplates, qui, dans un projet bien écrit représentent l'essentiel du code de l'UI. On accepte aussi dans l'autre sens que dans certains cas les Vues puissent contenir du code totalement spécifique pour gérer notamment du hardware. Par exemple, une capture image sera une fonction des ViewModels exposant les commandes et les propriétés nécessaires pour passer l'image, mais la capture elle-même sera codée dans les projets contenant les Vues car chaque plateforme aura ses spécificités. Autant les grouper au même endroit et ne pas détruire l'universalité des ViewModels dans la construction présentée ici. MvvmCross V3 avec ses plugins propose une approche qui gomme totalement les différences de hardware ou des API pour les piloter. Plus le temps avance et plus la stratégie que je vous propose devient simple à mettre en œuvre !

Ainsi, le bloc le plus précieux (le code métier) à une variabilité de 0 %, seul le bloc des UI a une variabilité maximale.

En adoptant cette stratégie le coût final pour **supporter une dizaine de cibles différentes**, c'est à dire tout le marché, **reste très proche du coût de développement d'une seule version spécifique**... Le surcoût ne se trouvant que dans les cibles qu'on souhaite supporter. Mais en suivant la stratégie exposée c'est aussi à cet endroit que la stratégie a permis de limiter drastiquement les couts...

En gros, **plus le code devient spécifique, moins il y a de code à produire**. Dans l'autre sens : **plus le code est cher à produire, plus il est universel et n'est développé qu'une fois**...

Conclusion

La stratégie de développement cross-platform et cross-form factor présentée ici est une méthode qui permet de maitriser les couts tout en offrant la possibilité de couvrir toutes les cibles qu'on souhaite.

Sa mise en œuvre ne réclame qu'un seul EDI, Visual Studio, un seul langage, C#, un seul pattern, MVVM, une seule librairie pour le gérer.

Grâce à MonoTouch et MonoDroid, nous pouvons ajouter les 4 cibles les plus importantes du marché (IPhone, IPad, Android phone et Android tablette) pour le prix d'une licence de ces produits.

Grâce à MVVMCross, gratuit et open-source, nous pouvons unifier le support de toutes les plateformes aussi différentes soient-elles de Windows 8.1 à iOS en passant par WPF ou Android.

Le tout dans la cohérence, sans s'éparpiller.

Et surtout : **le tout en produisant des applications natives pour chaque cible, sans compromis technique**.

Ne reste plus, pour convaincre les éventuels sceptiques et pour éclairer au mieux ceux qui ont compris l'intérêt de cette stratégie à vous présenter une mise en œuvre complète. Il s'agira d'une démonstration fonctionnelle utilisant un service web réel développé il y a 6 ans, bien avant qu'on ne parle des nécessités évoquées ici, qui tourne sur l'un de mes serveurs en Angleterre, et plusieurs cibles tel que le Web avec Silverlight, Windows Phone, Android et même une console Windows. Cela sera bien assez pour vous montrer que ce dont je parle ici est une stratégie mature et fonctionnelle et, je l'espère, vous convaincre qu'avec C#, .NET et Visual Studio, aidé de quelques bons outils comme MonoTouch et MonoDroid, on peut aujourd'hui tout faire et conquérir le monde !

> *On notera que cette démonstration originelle montrait l'état de l'art en 2011/2012 et que bien que la stratégie n'ait pas changé d'un iota, le tooling a lui beaucoup évolué. On gardera de cette première démonstration l'idée d'un POC et on visionnera les 12 vidéos de formation au cross-plateforme produites durant l'été 2013 pour voir comment on pratique réellement aujourd'hui.*

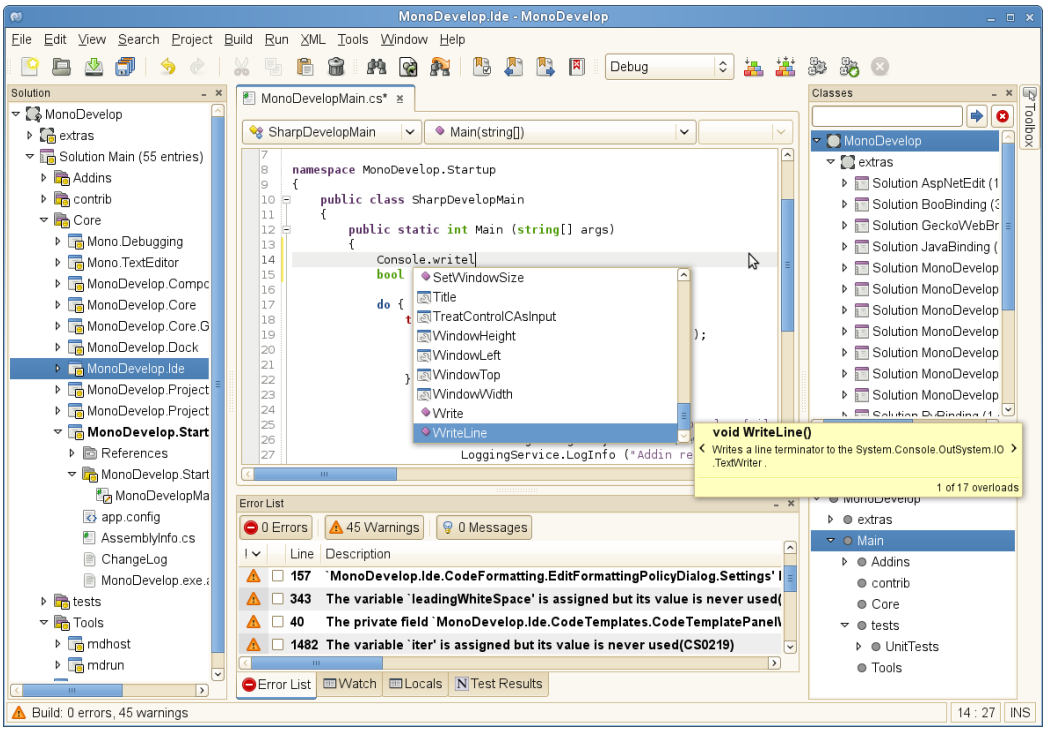
MonoDroid : et si la double connaissance Android / WinRT était la clé du succès ?

MonoDroid est cette version spéciale de Mono conçue par Xamarin pour produire des logiciels natifs **Android**. La version de base ne coute que $299 et s'installe en un clic (la démo est gratuite). **WinRT** c'est la plateforme de Windows 8 pour les

applications Modern UI. Quel rapport entre ces deux orientations totalement différentes de prime abord ?

MonoDroid c'est Mono et Mono c'est C# / .NET

MonoDroid est une solution spéciale bâtie sur Mono, la version Open Source de .NET. Spécialisée pour Android cette version de Mono est un produit commercial, frère jumeau de MonoTouch, la même chose mais pour le système **iOS** d'Apple. MonoDroid s'installe en clic, et simplifie le setup des différents SDK Android. C'est vraiment très bien fait et en quelques minutes on dispose d'un environnement de développement complet près à l'utilisation avec les émulateurs Android (vous pouvez tester vous-mêmes la version de démo pour Android ou iOS).



MonoDevelop ou VS ?

L'ensemble est fourni avec **MonoDevelop**, la "copie" Mono de Visual Studio, mais **MonoDroid s'installe en même temps dans Visual Studio** ce qui laisse à chacun le choix de son environnement. A noter tout de même qu'en version de base on ne dispose pas sous VS du concepteur visuel que propose MonoDevelop. Mais cela devient possible avec la version Business pour laquelle il faut débourser $999 (ce qui est intéressant vu le cours de l'euro). D'ailleurs certains fichiers de MonoDroid peuvent être manipulés aussi via Eclipse, l'environnement "naturel" pour Android,

voire IntelliJ IDEA, l'environnement Java créé par JetBrains, éditeur du célèbre et indispensable Resharper.
Le plus intéressant c'est bien entendu de disposer d'un environnement complet comme MonoDevelop avec le designer Visuel ou bien d'aquérir la version Business pour tout faire dans Visual Studio.

Mieux, tout cela étant du Mono au départ, donc portable, vous pouvez choisir une version Apple de MonoDevelop si vous développez sur un Mac sans perdre vos repères Visual Studio ! Cela peut intéresser des infographistes disposant d'un Mac mais ayant acquis des compétences de développement C# avec Silverlight ou WPF par exemple.

Emulateur Froyo lancé depuis MonoDevelop ou Visual Studio

La portabilité des applications WinRT / Android

Personnellement ce que je vise et ce que je conseille à mes clients c'est la **double compétence Windows 8 / Android**.
Exactement comme en 2008 je disais dans une interview que "*les Delphistes seront les cobolistes des années 2010*" et que je préconisais à tous les développeurs Delphi d'avoir la double compétence C#. Les années qui suivirent me donnèrent raison...
Windows 8 et WinRT j'en ai beaucoup parlé et je vais encore en parler car je suis convaincu techniquement par les solutions que Microsoft propose. Je reste certes critique par rapport à certains choix, mais être critique est le tout d'un expert indépendant, surtout sur un produit très jeune qui ne demande qu'à s'améliorer. Je ne suis pas un VRP Microsoft et je peux, et même je me dois d'honorer mon statut de MVP par mon indépendance intellectuelle. Et je crois en la solution Windows 8

notamment par la cohérence qu'elle propose des Smartphones aux PC en passant par les tablettes.
Toutefois c'est là qu'intervient l'indépendance d'esprit : je regarde aussi ce qui se passe autour de moi et je note la suprématie de plus en plus nette d'Android (80% de part de marché en France sur les Smartphones et sur les tablettes).

Pourquoi pas Apple ?

Apple ? Je n'ai jamais aimé leur façon de traiter les clients, et ça ne date pas d'hier mais de l'Apple IIC que j'avais acheté à sa sortie (cherchez sur Google, il y a des chances que certains lecteurs n'étaient pas encore nés à cette époque...) et d'un G3 dont j'ai hérité et que j'ai réussi à refourguer après avoir constaté que les problèmes chez Apple étaient les mêmes 20 ans après.

L'IPhone ? Un succès incontestable, mais "friable" comme un gâteau trop sec. Désormais Android a aussi ses stars comme Samsung. Et des volumes de vente qui grignotent de jour en jour la suprématie d'Apple, ce qui aujourd'hui un état de fait. En gros quand Apple vend 35 millions d'IPhone, il y a 240 millions d'unités Android qui sont vendues et quelques millions de Surface et de Windows Phones.

L'IPad ? Un autre succès incontestable mais tout aussi "soluble" qui est en train de se dissoudre dans l'acide puissant de la concurrence et du manque d'innovation qu'affiche désormais d'Apple. On ne refait pas tous les jours le coup de l'IPhone et de l'IPad, surtout quand celui qui est en l'instigateur est mort... Avec l'IPhone 5S on voit bien que certains fantasmes tombent à l'eau : Steve Jobs a bien eu raison de mourir en pleine gloire, il était sec et n'a laissé aucun projet « killer » en héritage à Apple. C'est un peu le coup des conspirationnistes et de Roswell, si les USA avaient disposé d'une technologie « alien » on se demande bien pourquoi ils se seraient fait piéger et embourbés avec des moyens conventionnels au Vietnam ou en Irak ! Le temps finit toujours par démasquer les impostures. De toute façon Apple n'a jamais su apporter de réponses séduisantes en entreprise, principale préoccupation en ce qui me concerne. Mon métier est de conseiller les entreprises, pas les mamies du Cantal aussi sympathiques soient-elles... Je ne conseillerais jamais à une entreprise de faire du BYOD surtout avec de l'Apple. C'est une hérésie, une économie de bouts de chandelle qui finit par couter une fortune (en équipes techniques formées).

Alors bien sûr, Apple existe, l'Apple mania aussi, l'Apple Store qui fait briller les yeux comme ceux d'Oncle Picsou avec les dollars qui défilent, tout cela est vrai. Mais de moins en moins face au ras-de-marée Android.

Pour tous ceux qui considèrent, à tort ou à raison, qu'Apple est incontournable, il suffit dans ce billet de remplacer "MonoDroid" par "MonoTouch" puisque **Xamarin propose les deux produits**.
Chacun fera en fonction de ses besoins et de ses préférences, je ne juge pas.

Mais en ce qui me concerne, Apple c'est no way.

L'intérêt d'un code C# et d'une plateforme .NET

Le grand intérêt de MonoDroid (et MonoTouch aussi) c'est de proposer à la fois un environnement de développement qui ne dépayse pas, donc pas de perte de temps, et à la fois un langage qu'on sait manipuler et une plateforme qu'on maitrise.

Pouvoir développer pour Android en réutilisant du code qui sert aussi à une application Tablette (ou phone ou PC) sous Windows 8 ou WPF est un sacré avantage !

Un cœur pour deux

Pouvoir développer le cœur d'une application en C# et réutiliser ce cœur pour une version WinRT, voire Silverlight et WPF, et pour une version Android est à mes yeux essentiels.

Android, grâce à la gratuité de l'OS, grâce au large choix des machines dans toutes les gammes de prix, grâce aussi, il faut l'avouer, à son succès technique (ça marche très bien) et commercial, mais aussi par la puissance de Google, est aujourd'hui une variable du marché qui ne peut plus être ignorée. Un développeur, un éditeur de logiciel, un DSI, ne peuvent plus négliger l'importance de cette plateforme et surtout la place qu'elle détient sur le marché. 80% des Smartphones sont déjà sous Android. Lorsque Microsoft occupait une telle place sur le marché des PC (ce qui est toujours le cas, ce sont les PC qui perdent leur suprématie) tout le monde était d'accord qu'il était impossible d'ignorer Windows… Pourquoi changer de logique ?

Se pose alors le problème de savoir comment centraliser le code, comment le **pérenniser**.

Choisir WinRT en entreprise est un choix d'avenir et de cohérence. Les mêmes équipes, la même plateforme, la même formation pour attaquer tous les terminaux du plus petit au PC, c'est le choix de l'intelligence. Je reste bien sûr circonspect sur le fullscreen, difficilement tenable pour les applications LOB où comparer deux clients, deux fiches articles, trois cours de bourse est un classique qui réclament… des fenêtres et pas une seule et unique surface d'affichage. Mais WinRT peut évoluer, et

même s'il n'est pas capable de couvrir 100% des besoins LOB il peut en couvrir une bonne partie avec l'avantage de la portabilité directe sur les tablettes (et à termes mais dans une moindre mesure avec Windows Phone).

Sans ignorer Android…
Et c'est là que MonoDroid permet le tour de force irréalisable autrement : pouvoir réutiliser le même code métier sous WinRT, WPF, Silverlight et Android.

(Comme expliqué plus haut on peut ajouter iOS à la liste avec MonoTouch).

Conclusion

A la différence de mon appel de 2008 qui sous-tendait une migration définitive de Delphi vers C#, ici je parle vraiment de double compétence Android / Windows 8 dans la durée.

Les deux marchés vont se côtoyer, parfois se croiser et entrer en concurrence, mais en réalité pour l'essentiel il s'agira de deux approches différentes. Windows pour l'entreprise, avec de la tablette et du Smartphone WinRT, mais aussi du BYOD avec du matériel Android.

Plus loin il va y avoir le besoin pour toute entreprise, même un site Web, même un vendeur de savons parfumés, et a fortiori un éditeur de logiciels d'assurer sa présence sous la forme d'applications natives vendeuses et efficaces. A la fois sous WinRT et sous Android (voire iOS pour certains marchés où cela est incontournable : musique assistée par ordinateur, graphismes…).

Je pense ainsi qu'il est essentiel pour tout développeur, tout éditeur, toute entreprise en générale, de cultiver la double culture WinRT et Android. Mais pas à n'importe quel prix. Pas en multipliant les équipes et les formations. Pas en utilisant des compétences rares donc chères. Non. **Tout cela peut être fait en pérennisant les compétences C# et .NET des équipes en place**.

Ce tour de magie est une réalité technique, MonoDroid (et MonoTouch) le permet aujourd'hui. Le fait que Xamarin a réussi à lever 12 millions de dollars l'année dernière et qu'ils affichent près de 500.000 développeurs aujourd'hui est un bon gage d'assurance pour le futur. Le fait que tout cela repose sur le SDK Google original est essentiel. Le fait que l'environnement de développement MonoDevelop soit du Mono, portable et Open Source est aussi une sécurité non négligeable.

La seule barrière qui restait n'existe plus, celle du tooling et des langages.

Aujourd'hui, développez pour les marchés du présent et du futur en investissant dans WinRT et Android. Mais ne restez pas scotché à une seule compétence, cela serait une erreur.

Je vais donc parler dans le futur autant de développement WinRT que de développement C# et .NET sous Android.

Bon développement multi-plateforme en C# ...

Les vidéos de Xamarin EVOLVE 2013 en ligne

Xamarin EVOLVE 2013 est une conférence dédiée au développement mobile qui s'est tenue à Austin (Texas) il y a peu de temps. Il est bon de préciser que Microsoft était "Platinum Sponsor" de cet évènement axé sur le développement iOS et Android...

Platinum Sponsor

A ceux qui pourraient penser que parler d'Android sur un blog orienté Microsoft serait une forme de défiance vis à vis de ce dernier, il est bon de rappeler qu'aux dernières conférences Techdays Microsoft Belgique a programmé une session sur le développement cross-plateforme sous Xamarin et que dernièrement au conférences Xamarin "Evolve 2013" dédiées au développement mobile sous Android et iOS Microsoft US était "Platinum Sponsor", c'est à dire sponsor "platine", le plus haut niveau de participation possible...
Même pour Microsoft, Android et Windows ne sont pas antinomiques et farouchement opposés !

Et on le comprend car dans la réalité Microsoft gagne beaucoup d'argent sur la vente de chaque terminal Android grâce à quelques brevets bien placés ! L'éditeur s'investit directement dans des conférences comme EVOLVE 2013 pour une bonne raison, ce qu'il n'arrive pas encore à gagner sur son marché propre est compensé par les ventes d'Android... D'autant que le premier lui réclame de lourds investissements non rentabilisés alors que les secondes se font sans aucun travail de sa part et sont pur bénéfice...

Parler de cross-plateforme et de développement Android c'est donc être en parfait accord avec les actes et les intérêts financiers de Microsoft !

On notera que d'ici 2017 les royalties sur Android pourraient rapporter 8.8 milliard de dollars à Microsoft... Quand on pense à la division XBox qui fait perdre 300 millions de dollars par an à l'éditeur, cela laisse rêveur et permet de comprendre

pourquoi Android n'est pas l'ennemi qu'on croit ! (lire "Microsoft could generate $8.8 billion annually from Android royalties by 2017")

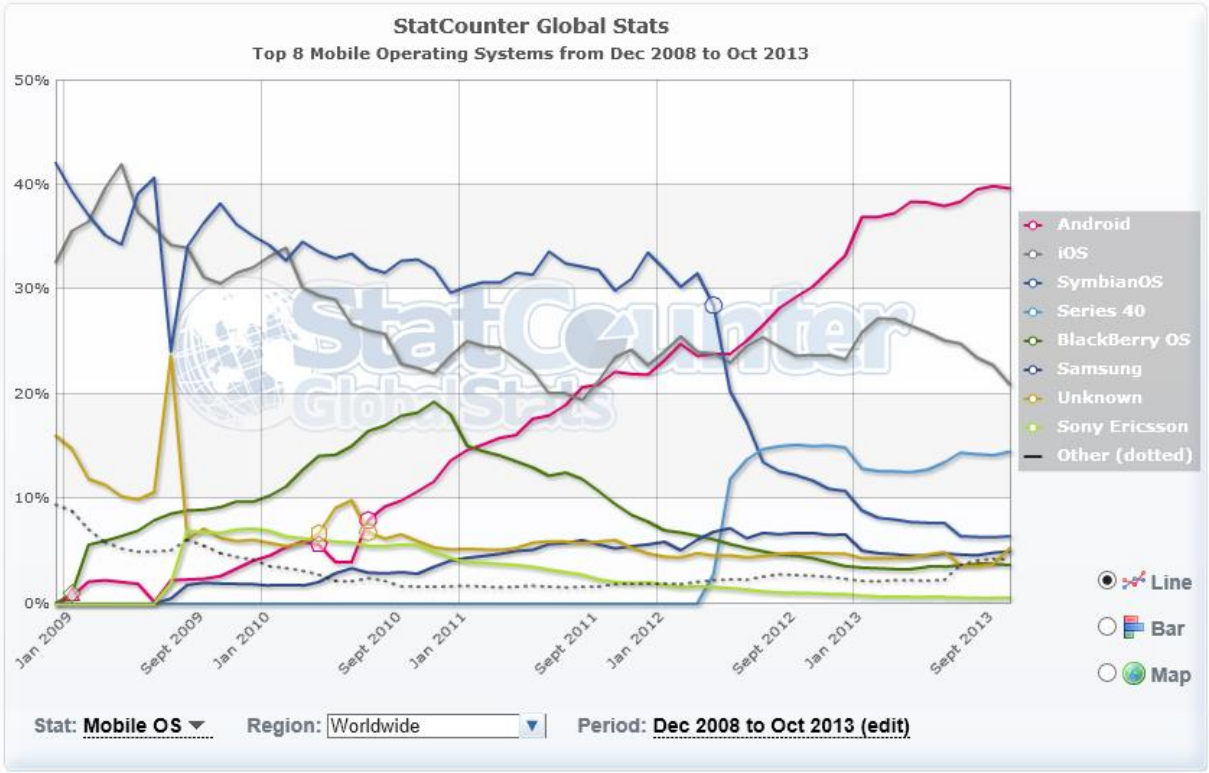
Donc si certains pouvaient voir dans cet axe de développement de Dot.Blog une forme d'éloignement de ma part, qu'ils s'en trouvent rassurés, Microsoft a une vision large et pragmatique des choses et participe directement ou indirectement au succès d'Android qui en retour gonfle ses caisses sans trop se fatiguer bien plus que ne l'ont fait à ce jour Surface RT ou la XBox !

Je ne fais qu'adopter dans ces colonnes la même approche pragmatique du marché en essayant de faire comprendre à tous les enjeux de la partie qui se joue en ce moment.

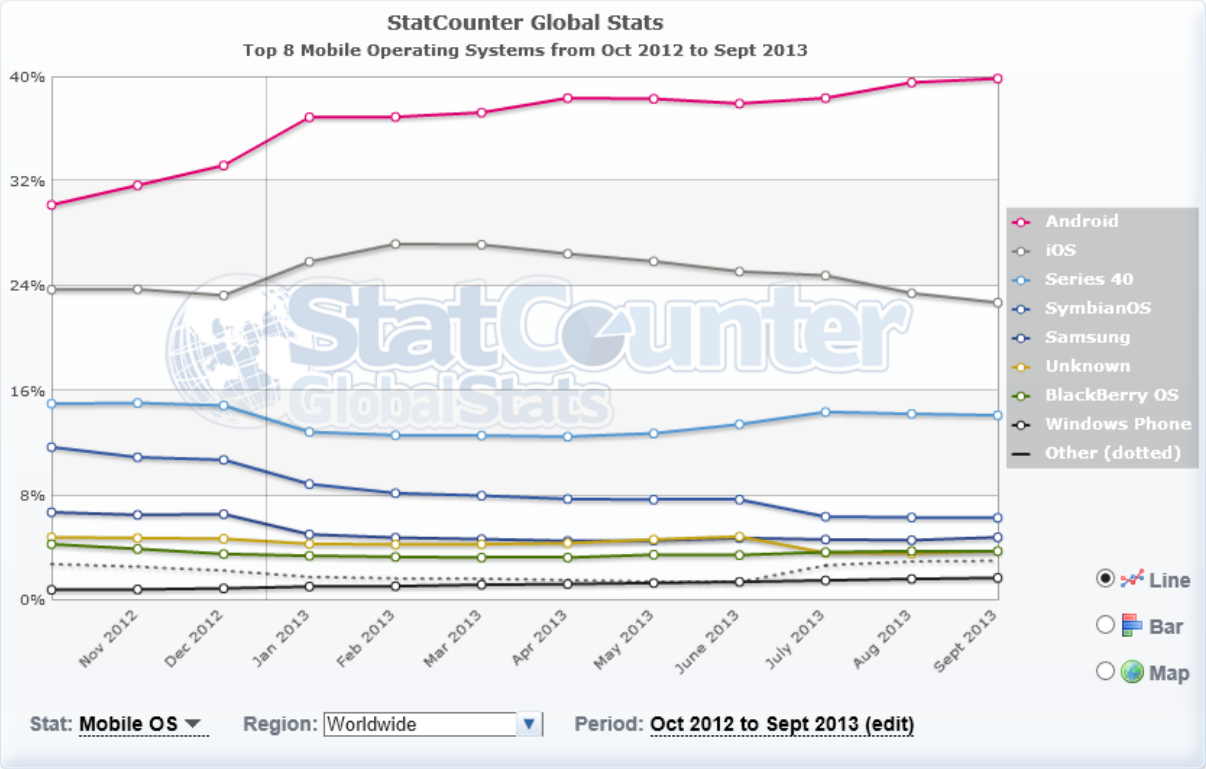
Donc Microsoft a été sponsor Platine des conférences Xamarin Evolve 2013 confirmant une fois de plus le bienfondé de la ligne éditoriale de Dot.Blog !

Xamarin, un succès croissant

Mais peut-on s'étonner de l'intérêt que les gens portent à Android et à Xamarin ? Ceux qui connaissent Xamarin vous diront que c'est une suite logique des choses. D'une part les développeurs .NET souhaitent **capitaliser leurs connaissances** pour aborder de nouveaux marchés et, d'autre part, **Android est aujourd'hui le leader mondial des OS Mobiles**, et de très très loin, c'est comme ça. Un petit chart vaut mieux qu'un long discours :



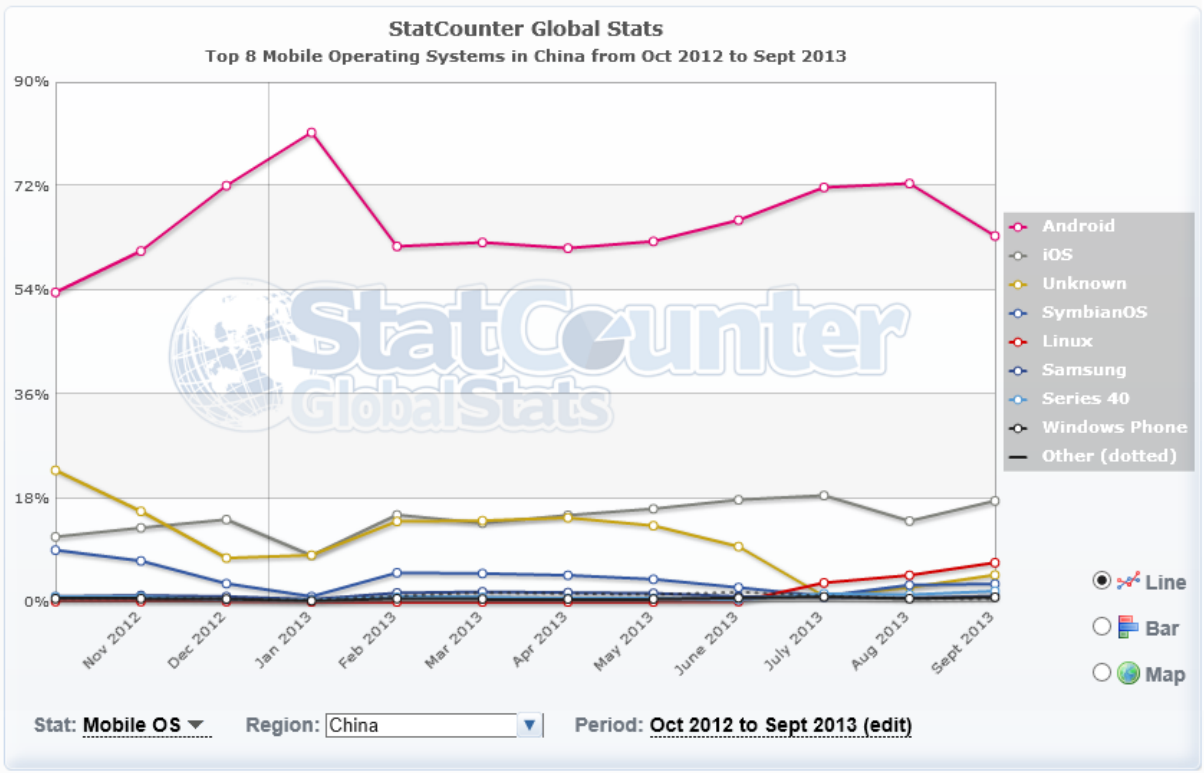
Ce graphique retrace l'évolution du marché depuis janvier 2009. La courbe mauve de Android ne laisse planer aucun doute… tout comme la pente douce mais certaine vers la chute de iOS qui marque en tout cas la fin claire et certainement définitive de son leardership. Quant Windows Phone, il est mélangé aux « autres OS » tellement il a peu d'existence pour l'instant. Il faut demander le même graphique uniquement sur l'année passée pour voir enfin apparaître une ligne dédiée à Windows Phone (qui existe depuis longtemps avec la version 7 malgré tout) :



Et encore… notre pauvre Windows Phone c'est la ligne noire tout en bas, une sorte d'électroencéphale plat (signe clinique de la mort en médecine) qui se trouve même dépassée sur la fin 2013 par un fatras inconnu « autres OS » !

Et encore, cette ligne noire mélange allégrement Windows Phone 7, 7.5, 7.8 et 8.0, des OS incompatibles entre eux (au moins au niveau Hardware entre les 7.x et la 8).

Si on voulait être tranché, on pourrait regarder le marché le plus gigantesque de la planète, la Chine, et regarder la répartition des OS mobiles :



L'énorme vague rose au sommet, ce ne sont pas les ventes du petit livre rouge de Mao, non, c'est l'hégémonie d'un OS américain : Android. Windows Phone (malgré le trucage habituel du mélange évoqué plus haut) n'apparait que tout en bas à la limite des nappes phréatiques et iOS ne semble être qu'un filet d'eau ténu coulant de façon désordonnée dans le cours d'une rivière à sec par un été caniculaire… Malgré une résistance politique historique aux USA et l'endoctrinement qui va avec, malgré un malaise historique belliqueux persistant entre la Corée (du Sud aujourd'hui) et la Chine, les chinois préfèrent en masse des machines coréennes sous OS américain… (lire "Samsung domine aussi en Chine" dans "LInformaticien.com"). Nokia qui pesait il n'y a pas si longtemps encore 30% du marché chinois n'apparait plus sur les graphiques.
Et si on veut jouer aux pronostics et se projeter dans le futur, IDC l'a fait :

Smart Connected Device Market by Product Category, Shipments, Market Share, 2012-1016 (shipments in millions)

Product Category	2016 Unit Shipments	2016 Market Share	2012 Unit Shipments	2012 Market Share	2016/2012 Growth
Desktop PC	151.0	7.2%	149.2	12.5%	1.2%
Portable PC	268.8	12.8%	205.1	17.2%	31.1%
Smartphone	1405.3	66.7%	717.5	60.1%	95.9%
Tablet	282.7	13.4%	122.3	10.2%	131.2%
Total	2107.8	100.0%	1194.0	100.0%	76.5%

Source: IDC Worldwide Quarterly Smart Connected Device Tracker, December 10, 2012.

On voit les ventes sur 2012 et leurs projections sur 2016.

Entre 2012 et 2016 le marché des smartphones connaitra une progression de 95.9% et celui des tablettes de 131.2%. Lorsqu'on sait la répartition des OS (graphiques ci-avant) on comprend facilement que tout cela a profité aux smartphones et principalement à Android et que peu de choses désormais devraient changer la donne.

Mais en 2016 IDC voit une répartition du marché encore plus tranchée; les ventes de PC de bureau ne représenteront plus que 7.2%, les portables un peu plus avec 12.8%, les tablettes dépasseront les PC portables avec 13.4%, mais surtout, les smartphones seront pourvoyeurs de 66.7% des ventes d'ordinateurs !

Une réalité à intégrer : ces machines ont besoin de programmes !

Si tous ces chiffres ne concerneraient que des modèles de voitures, on pourrait se dire qu'on entre dans des détails sans intérêt : toutes fonctionnent majoritairement à l'essence ou au diesel, parfaitement compatibles avec tous les véhicules de la planète. Du diesel russe peut faire tourner un camion américain, de l'essence algérienne peut faire rouler une berline allemande ou une citadine française…

En informatique les choses sont un peu différentes, *notre essence à nous ce sont les logiciels*. Et malheureusement pour nous les constructeurs de machines et d'OS s'arrangent pour créer des mondes cloisonnés et incompatibles entre eux. Là où les fabricants de moteurs ont compris de longue date que la compatibilité était porteuse

de stabilité et que la stabilité était bonne pour le business, les éditeurs d'OS eux pensent que diviser aide à mieux régner. Ce n'est pas forcément une mauvaise approche, c'est tout simplement un état d'esprit différent.
Et nous devons "faire avec".

Près de 67 % des ordinateurs en 2016 seront des smartphones. 14% seront des tablettes. 7% seulement seront des PC de bureau…

67 % + 14 % = 81% du marché sera mobile.

81% …

Deux OS tiendront le haut du classement, Android et iOS. Même si Microsoft conservait 100% du marché Desktop, cela ne contera donc plus que pour 7% du marché ! Et s'ils maintiennent leur 3ème place dans les smartphones, encore faudra-t-il qu'il conserve cette place dans la durée et qu'il lui donne un vrai sens (la distance avec les 2 premiers est gigantesque pour l'instant) et qu'enfin ils trouvent la voie du succès avec Surface (ce qui est compliqué car ils ont abandonné la RT très vite pour mettre sur le marché la Pro qui attire uniquement pour le bureau classique, en faisant dans les faits un simple PC portable de format réduit et non une vraie tablette au sens du marché actuel).

Toutes ces machines vont avoir besoin de développeurs et de logiciels.

L'entreprise largement concernée

Dans ce mouvement de fond il est clair que les entreprises ne pourront pas échapper à la vague. Croire le contraire est une erreur de lecture de l'histoire en marche. Même si leurs technologies de base restent et resteront classiques (SQL Server, SharePoint, Windows Server…) elles devront savoir intégrer à côté des postes fixes en Desktop toutes les technologies mobiles. Et ici ce sont Android et iOS qui dominent.

Plus loin, un nouveau type de terminal va modifier en profondeur notre rapport à l'informatique et va intéresser plus qu'on le croit les entreprises : les Google glass, sous Android bien entendu.

"Selon le cabinet 451 Research, les lunettes connectées de Google feront bientôt leur apparition dans les entreprises et pourront s'avérer utile pour les départements informatiques et les professions en extérieur qui nécessitent de garder les mains libres."
Lire "Les Google Glass s'invitent dans les entreprises" (Le monde informatique)

Mes clients, vos clients et vos patrons : des entreprises

Bien loin des Angry Birds, aussi sympathiques soient-ils (bien qu'un peu trop répétitifs), c'est à dire bien loin du marché des jeux et des gadgets "grand public", *mes clients comme votre patron appartiennent au monde des entreprises*. Ce sont ces entreprises qui nous font vivre. C'est à elle que nous vendons nos idées, notre force de travail, notre expertise.

Il est donc de la responsabilité de Dot.Blog de parler vrai aux entreprises, à leurs DSI et leurs développeurs.

Et leur dire que l'avenir c'est maintenant, que *le marché Android n'est pas une mode mais une réalité, un rouleau compresseur mondial, et que ne pas s'y intéresser dès aujourd'hui c'est prendre d'ores et déjà un retard considérable sur la concurrence…*

Marier Windows et Android

Comme je l'ai expliqué, mon but n'est pas de convertir à Android les lecteurs de Dot.Blog en les éloignant de Microsoft, au contraire. Mon dessein est de faire en sorte que face au tsunami Android beaucoup ne désertent pas le navire. Pourquoi ?

Parce que je reste convaincu de la supériorité de la technologie Microsoft dans de nombreux domaines et que les entreprises ne pourront pas se passer de cette technologie.

Mais vouloir les enfermer dans un discours de vulgaire VRP à la solde de Microsoft en excluant tout ce qui n'est pas Microsoft serait grave, en tant que Conseil je le verrai même comme une faute professionnelle… Et puis je suis un MVP, un "expert indépendant" et que cela ne prend de sens qu'en délivrant des conseils non soumis à une quelconque allégeance que même Microsoft ne réclame pas. De la fidélité oui, de l'allégeance non.

Il m'apparait ainsi essentiel de faire comprendre à la fois l'inéluctable vérité sur la position d'Android devenue leader mondial des OS mobiles et l'absolue nécessité de faire cohabiter cet OS et les machines qui le supportent avec les infrastructures, les langages, les connaissances engrangées avec les environnements Microsoft qui resteront en place pour longtemps.

Xamarin : en toute logique

Le succès des conférences EVOLVE 2013 de Xamarin, comme je le disais en introduction, n'est pas une surprise pour ceux qui connaissent le produit et qui ont conscience de la réalité du marché. Je disais même que c'était une suite logique.

Le développement proposé ici vous le prouve : Android est incontournable, le marier aux environnements Microsoft qui eux dominent pour longtemps encore en entreprise l'est tout autant, et quel produit est mieux placé pour le faire que Xamarin qui vous propose de réutiliser vos connaissances en C#, .NET et même Visual Studio ?

C'est donc bien en toute logique que cet environnement s'impose comme trait d'union entre deux OS à la base incompatibles mais que la magie de Xamarin rend "frère" au moins sur l'essentiel, le code, le framework et l'EDI.

EVOLVE 2013 : des conférences à voir absolument

A ce stade de ma démonstration, il est temps que les mots cèdent la place aux vidéos !

Car comment mieux comprendre le potentiel de Xamarin qu'en regardant les conférences EVOLVE 2013 mises en ligne ?

http://xamarin.com/evolve/2013#session-dvn2812vfp

Conclusion

Un marché énorme s'ouvre aux éditeurs, celui des unités mobiles, mais en même temps cette puissante vague n'épargne pas l'entreprise et encore moins le développeur qui se doit de maintenir ses compétences en phase avec le marché.

Faire le pont entre deux mondes, celui de Microsoft et celui de Google, c'est ce que propose Xamarin au travers d'un outil désormais très largement mature.

Faire le lien entre deux mondes qui semblent concurrents sans perdre la fidélité aux technologies Microsoft c'est que je vous propose au travers de Dot.Blog.

Le voyage n'en est qu'à ces débuts et l'avenir sera, par force, encore plus intéressant que ce qu'on peut imaginer…

Stratégie de développement Cross-Platform–Partie 2

La Partie 1 de cette série expliquait la stratégie de développement cross-platform et cross-form factor que je vous propose pour faire face à la multiplicité des

environnements à prendre en compte aujourd'hui sur le marché. Il est temps de passer à la pratique, je vous invite à suivre le guide pour une démo pas à pas de cette stratégie sur un cas réel.

> *Cette démonstration doit être vue aujourd'hui comme un POC datant de 2011/2012, pour une démonstration à jour fin 2013 on préfèrera visionner les 12 vidéos de formation cross-plateforme sur YouTube.*

Rappel sur la stratégie présentée et les outils utilisés

Nous avons vu dans la Partie 1 l'ensemble des contraintes et difficultés auxquelles un développeur ou éditeur de logiciel devait faire face aujourd'hui pour prendre en compte l'explosion des plateformes techniques et des form-factors : PC, Tablettes, Smartphones, le tout avec au minimum 4 OS : iOS, Android, Windows "classique" et Windows 8 (WinRT).
Je vous ai présenté une stratégie que j'ai élaborée après avoir testé de nombreuses solutions, outils, langages et EDI.

Pour résumer, cette stratégie tient en quelques points clés :

- La centralisation du code métier sur un serveur de type Web Service exploitable par tous les OS

- L'utilisation de la librairie MVVMCross qui propose un même socle pour les OS à couvrir et permet d'écrire un projet de base compilable pour chaque cible à partir du même code écrit une seule fois. Ce projet regroupe les ViewModels. La version récente de la stratégie exploite les PCL de Visual Studio pour simplifier la démarche, le serveur devant optionnel, tout pouvant désormais se situer dans un seul projet C# sous la forme d'une PCL. Un progrès important.

- L'écriture de projets spécifiques à chaque cible ne contenant que la définition des Vues et utilisant toutes les possibilités de la plateforme, mais s'appuyant sur MVVMCross pour l'injection de dépendances (ViewModels et autres services) et sachant ajouter le Binding (à la base du pattern MVVM) aux cibles ne le gérant pas (comme Android ou iOS).

Cette stratégie possède de nombreux intérêts :

- Un seul langage de programmation : C#
- Un seul framework : .NET
- Un seul EDI : Visual Studio (ou MonoDevelop qui en est une copie sous Mono)
- Une seule librairie MVVM cross-platform (MVVMCross)

- Concentration du savoir-faire, économie des outils et des formations
- Une grande partie du code est universelle et ne réclame aucune réécriture quelles que soient les cibles
- Plus le code est spécifique, moins il y en a à produire, la stratégie offrant une maitrise des couts optimale
- Le cœur du métier est protégé et pérennisé grâce au serveur Web ou à la PCL noyau
- La stratégie supporte des adaptations sans être remise en cause (par exemple ne pas avoir de serveur métier et tout concentrer dans le projet définissant les ViewModels).
-

Enfin, cette stratégie repose sur deux produits la rendant possible si on vise toutes les plateformes : MonoTouch et MonoDroid de Xamarin, deux plugins Visual Studio (ou MonoDevelop) permettant de développer en C# et .NET de façon native sur les unités mobiles Apple et Android.
La démarche reste parfaitement justifiée même sans cibler iOS et Android. On tirera parfaitement parti de la stratégie proposée par exemple pour développer un logiciel pour WPF et atteindre 80% du marché des PC doublé d'une version WinRT pour Windows 8 et Surface, voire triplé par une version Windows Phone.

La stratégie proposée n'implique pas le support de iOS ou Android, elle reste valide même à l'intérieur du monde Microsoft où hélas les incompatibilités existent de la même façon qu'entre les OS des différents éditeurs…

Le contexte de l'exemple

Pour concrétiser tout cela je souhaitais bien entendu écrire un exemple complet couvrant plusieurs cibles.

L'exemple devait être concret, faire quelque chose d'utile, mais devait rester assez simple pour être développé en deux ou trois jours maximum.

La preuve de l'intérêt de la stratégie du serveur métier

J'avais développé en 2008 un exemple amusant pour Silverlight, les Codes Postaux Français.

A l'époque il s'agissait avant tout de montrer les nouvelles possibilités qu'offraient Silverlight pour le développement Web, principalement le haut niveau de personnalisation de l'UI ainsi que la facilité de dialoguer avec des données distantes.

Le code source des Codes Postaux Français a été publié sur Dot.Blog en 2009.
Ce projet se basait sur un Service Web de type "asmx", les plus simples et les plus universels, servant des données en XML, tout aussi universelles.

La base de données ainsi que les recherches, et toutes les parties "intelligentes" de cette application se trouvaient donc reportées sur un "serveur métier" même si, s'agissant d'un exemple, ce "métier" et son "savoir" restaient très simples. Le principe étant lui rigoureusement le même que pour une application plus complexe.

Le service Web a été écrit en 2008. Il y a donc 6 ans. Il n'a jamais été modifié depuis et tourne depuis cette date sur mes serveurs, un coup aux USA, un autre en Angleterre, sans que les utilisateurs de l'application Silverlight ne voient la moindre différence (en dehors d'un meilleur Ping avec l'Angleterre pour les français).

Je me suis dis que reprendre cet exemple avait plusieurs vertus :

- Montrer que centraliser le savoir-faire dans un service Web était un gage de pérennité du code
- Appuyer ma démonstration cross-platform par des accès à des données distantes sur plusieurs OS ajoutait de la crédibilité
- Accessoirement cela m'évitait de créer le "serveur métier" et me permettait de me concentrer sur la partie cross-platform de l'article

A l'heure actuelle, et vous pouvez le tester (suivre les liens plus haut), il existe donc une application Silverlight (cible 1 : le Web, l'Intranet) fonctionnant depuis 6 ans de concert avec un service Web XML de type asmx ASP.NET, le serveur métier IIS sous Windows Server, qui tourne sans discontinué et qui centralise le "savoir métier" d'une gestion des codes postaux français dont la base de données (honte sur moi !) est une vieille base MS Access. C'est dire la diversité à laquelle on a affaire en même temps que la robustesse dans le temps de ces choix !

Repartir de ce service Web est une bonne façon de montrer toute l'efficacité de la stratégie proposée. Le code du service n'a jamais été conçu pour des téléphones Android, et pour cause cela n'existait pas (le 1er smartphone Android, le HTC Dream a été lancé en octobre 2008) … Pas plus que fonctionner sous WinRT ou un IPhone ni même sous Windows.

Malgré tout, ce service qui justement n'a rien d'exotique (service web XML) reste au fil du temps utilisable sans remise en cause de son code et du "savoir-faire" qu'il contient.

Les cibles que je vais ajouter

S'agissant d'une démonstration je n'allais pas acheter un Mac pour cibler iOS (l'émulateur iOS exige un Mac, alors que celui d'Android marche très bien sur PC) alors que j'ai eu un mal de chien à me débarrasser d'un G3 il y a quelques années… Un Mac Mini ne coute pas très cher (dans les 500 euros voire moins je crois) et pour ceux qui voudront cibler l'IPhone ou l'IPad c'est un investissement tout à fait justifié et raisonnable.

Il fallait tout de même choisir quelques cibles très différentes et totalement incompatibles entre elles pour montrer tout l'intérêt de la stratégie.

J'ai donc choisi les trois cibles suivantes :

- Android

- Windows Phone
- Windows console

Pour ce qui est d'Android j'utilise donc MonoDroid (Xamarin.Android) comme plugin Visual Studio. Pour Windows Phone 7.x j'utilise le SDK Microsoft qui se plogue aussi dans VS.

Le choix "Windows console" est là pour ajouter une cible très différente des deux premières et aussi pour servir de base de test à mon ensemble d'applications. MVVMCross propose en effet un mode "console" qui fait très MS-DOS et qui arrive à se connecter aux ViewModels. C'est une cible assez peu intéressante en soi, mais pour les tests c'est facile à mettre en œuvre et pour la démonstration, c'est un OS de plus et une UI qui n'a rien à voir avec les autres. On peut dire que cette cible symbolise toutes les cibles de type Windows "classique".

Je n'ai pas ajouté de cible WinRT car le développement aurait été très proche de l'exemple WP7 en Silverlight. Mais le lecteur saura transposer ce choix de cibles à toutes autres, au moins en pensée.

L'application

Donc, je vais monter une application dont le but est la consultation et l'interrogation d'une base de données des codes postaux français. La première étape, la constitution d'un "serveur métier" est déjà réalisée (voir plus haut pour le code) ainsi que la première cible (en Silverlight Web).

Je souhaite écrire un seul code C# pour toute la logique de l'application (les ViewModels) et j'accepte bien entendu l'écriture des interfaces spécifiques, que cela soit en Xaml Windows Phone ou en Axml pour Android. C'est la partie variable assumée se résumant aux Vues de l'application.

Pour ne pas tout compliquer, l'application en question fonctionnera sur un seul écran sans navigation. Ce n'est pas un billet sur le développement lui-même d'applications complexes.

Réalisation

Revue de paquetage : J'ai bien un Visual Studio (2010 pour cet exemple à l'époque) complet, à jour, j'ai bien installé MonoDroid et les SDK Android, j'ai téléchargé et posé dans un coin de mon disque les dernières sources de MVVMCross...

C'est parti !

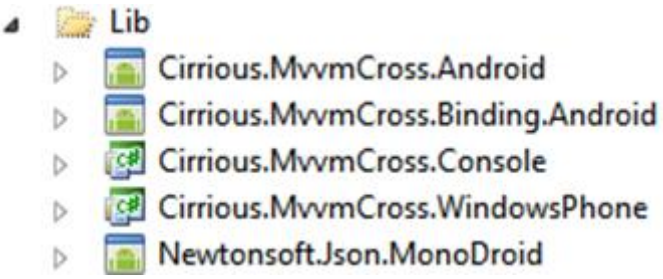
> *Aujourd'hui il suffirait de créer une PCL sous VS et d'installer MvvmCross via les packages Nuget. Comme je l'ai expliqué, rétrospectivement cette première réalisation cross-plateforme doit être considérée comme un POC ayant parfaitement joué son rôle. Pour le développement lui-même, si la ligne reste la même, les méthodes ont évolué depuis.*

La solution

Partir d'une page blanche est à la fois excitant et terrifiant... Il suffit donc de créer une nouvelle solution totalement vide. Et le voyage commence.

MVVMCross

Je commence par créer un sous-répertoire "Lib" (un répertoire de solution, pas un répertoire en dur) dans lequel j'ajoute les modules indispensables de MVVMCross (je préfère agir de cette façon plutôt que de référencer les binaires directement, j'ai ainsi l'assurance que le code de la librairie se compile correctement et en cas de problème cela sera plus facile à déboguer).



Comme on le voit, MVVMCross se présente sous la forme de plusieurs projets, chacun étant spécialisé pour une cible donnée mais exposant exactement le même fonctionnement et les mêmes APIs... C'est là que réside une partie de l'astuce !

Je n'ajoute que les projets correspondant aux cibles que je vais utiliser c'est pour cela que vous ne voyez pas dans l'image ci-dessus ni la version spéciale WinRT pour Windows 8 ni celle pour iOS.

Le projet Noyau

La stratégie exposée repose sur l'écriture d'une application "noyau" (core) qui définie l'ensemble des ViewModels. Il est tout à fait possible de se passer d'un serveur métier dans certains cas, le projet noyau est alors le début de l'histoire au lieu de se placer entre entre le serveur métier et les projets d'UI spécialisés. Certaines applications simples ne dialoguant pas avec des données externes ou ne comportant pas un gros

savoir métier peuvent parfaitement se contenter d'implémenter tout le code utile dans le "noyau".

Dans mon exemple j'ai souhaité mettre en scène la partie "serveur métier" car ce dernier fait partie intégrante de la stratégie que je propose. Mais rappelez-vous que cette stratégie est assez souple pour autoriser des adaptations.

La première question qui va très vite se poser est de savoir en quoi va être développé le noyau ?

On peut prendre n'importe quelle plateforme, l'essentiel est d'utiliser une plateforme qui supporte le minimum vital. Si on utilise un projet WinRT on sera en C#5 incompatible avec le C#4 de Mono par exemple. Si on utilise Silverlight on sera vite amené aux limites du mini framework embarqué par le plugin, ce "profile" (selon la terminologie MS) de .NET n'étant pas forcément représentatif du profile utilisé par Mono.

Bref, le plus simple, c'est de partir d'un projet Librairie de code en Android, et, pourquoi pas, une vieille version pour être tranquille. J'ai donc choisi d'écrire le projet noyau comme une librairie de code Android 1.6.

Rappel : Ces tergiversations n'ont plus de raison d'être aujourd'hui puisque le noyau est développé en un seul exemplaire à l'aide d'une Portable Class Library de Visual Studio.

Le projet noyau s'appelle CPCross.Core.Android, son espace de nom par défaut est CPCross.Core, n'oublions pas que ce projet doit être "neutre" dans le sens où il sera ensuite dupliqué pour toutes les autres cibles (ce qui n'est plus nécessaire en utilisnt une CPL). Même si cela ne changerait au fonctionnement, il serait idiot d'avoir à appeler une classe CPCross.Core.Android.MaClasse dans la version Windows Phone par exemple... Le projet est bien un projet Android parce qu'il faut bien choisir une plateforme pour implémenter le noyau, mais ce code doit être totalement transposable. Une bonne raison à cela : il n'y aura qu'un seul code de ce type ! Dans les projets "dupliqués" il n'y a aucune copie, mais des liens vers le code du projet noyau originel. Seul le fichier ".csproj" est différent car il contient les informations de la cible (OS, version de celui-ci) et que c'est grâce à ces informations que le _même_ code sera compilé en _différents_ binaires pour chaque cible.

On se rend compte que les CPL de VS ne font qu'automatiser ces manipulations les rendant totalement transparentes ce qui simplifie les choses pour le développeur aujourd'hui.

Le projet noyau référence Cirrious.MvvmCross.Android, puisque c'est une librairie de code Android 1.6. C'est de cette façon que tout le potentiel de MVVMCross sera disponible pour notre propre code.

Dans ce projet j'ajoute les répertoires ViewModels et Converters. Le premier contiendra les ... ViewModels et le second les convertisseurs éventuels (nous y reviendrons).

Le codage de l'application peut alors commencer. Dans le répertoire ViewModels j'ajoute une nouvelle classe, MainViewModel.cs selon une méthode très classique lorsqu'on suit le pattern MVVM.

Ce ViewModel descend de la classe MvxViewModel qui est fournie par MVVMCross.

Afin de ne pas rendre ce billet interminable et de ne surtout pas vous embrouiller, je ne détaillerai pas le fonctionnement de MVVMCross, seules les grandes lignes du montage du code en suivant la stratégie proposée seront évoquées (le présent livre propose des présentations plus poussées de MvvmCross, ainsi que les 12 vidéos YouTube consacrées au développement cross-plateforme).

Le ViewModel est tout ce qu'il y a de plus classique. Il expose des propriétés, il déclenche des propertyChanged quand cela est nécessaire et propose des commandes qui seront bindables à l'interface utilisateur. Rien que de très classique somme toute pour qui connait et manipule des librairies MVVM donc.

Le projet noyau va se voir ajouter une Référence Web pour pointer le service Web des codes postaux. Grâce à la compatibilité extraordinaire de MonoDroid avec le framework .NET, tout ce passe comme dans n'importe quel projet .NET... Les noms de classes générées sont rigoureusement les mêmes qu'ils le seraient dans un projet Windows.

J'ai juste rencontré un enquiquinement avec... Windows Phone. Ce dernier n'accepte pas l'ajout de Références Web (Web Reference) dans le mode "avancé" (en réalité des services Web classiques) il faut ajouter un "Service Reference", ce qui génère un nom de classe mère pour le service légèrement différent. Heureusement le reste du code est identique.

De fait, j'ai été obligé de "ruser" une fois dans le ViewModel :

```
#if WINDOWS_PHONE
        private EnaxosFrenchZipCodesSoapClient service;
```

```
#else
        private EnaxosFrenchZipCodes service;
#endif
```

Le nom de classe est différent pour Windows Phone. La variable s'appelle de la même façon et le reste du code n'y voit que du feu...

A noter que cette modification a été faite après avoir créer le projet pour Windows Phone et m'être aperçu de cette différence. Au départ bien entendu je ne l'avais pas prévu.

Aujourd'hui on utiliserait une autre stratégie pour injecter le code « natif » dans le noyau PCL afin de ne pas utiliser de compilation conditionnelle dans le noyau.

En cherchant bien on doit pouvoir forcer le nom à être identique, ne serait-ce qu'avec un refactoring et j'aurai pu éviter cette "verrue" qui me dérange. Mais ce n'est qu'une démo, le lecteur me pardonnera j'en suis certain de ne pas avoir atteint la perfection...

Le projet noyau utilise MVVMCross qui, comme de nombreux frameworks MVVM, oblige à suivre une certaine logique pour déclarer les liens entre Vues et ViewModels, pour indiquer les navigations possibles dans l'application, ou l'écran de démarrage, etc. Chaque framework a sa façon de faire, disons que MVVMCross se comporte un peu comme Jounce ou Prism.

Il faut ainsi ajouter au projet une classe qui hérite de MvxApplicationObject et qui supporte l'interface IMvxStartNavigation si on veut gérer la navigation en plus (l'application n'a qu'une page mais le gérer "comme une vraie" fait partie de la règle du jeu). Cette classe est très simple car tout est dans la classe mère ou presque :

```
using CPCross.Core.ViewModels;
using Cirrious.MvvmCross.Interfaces.ViewModels;
using Cirrious.MvvmCross.ViewModels;

namespace CPCross.Core
{
    class StartApplicationObject : MvxApplicationObject,
                                   IMvxStartNavigation
    {
        public void Start()
        {
            RequestNavigate(typeof(MainViewModel));
        }
```

```
        public bool ApplicationCanOpenBookmarks
        {
            get
            {
                return true;
            }
        }
    }
}
```

Cet objet “start” sert en fait à initialiser l’application et notamment ici la navigation vers la première page. MVVMCross ne navigue pas par les Vues, ce qui est généralement la vision classique des choses sous MVVM, mais par les ViewModels. La méthode “Start” réclame ainsi une navigation vers le type “MainViewModel”, notre seul et unique ViewModel. Ce qui déclenchera l’affichage de la page (ainsi que la recherche de la Vue correspondante et son attache dynamique, via le binding, au ViewModel).

Ce code utilise l’une des premières versions de MvvmCross. Les versions les plus récentes reprennent la même logique mais certaines classes ont changé de nom et de nombreuses simplifications aident à produire un code encore plus limpide.

Reste à définir un second objet, l’objet “application” lui-même, lui encore créé par héritage d’une classe fournie par MVVMCross :

```
using Cirrious.MvvmCross.Application;
using Cirrious.MvvmCross.ExtensionMethods;
using Cirrious.MvvmCross.Interfaces.ServiceProvider;
using Cirrious.MvvmCross.Interfaces.ViewModels;

namespace CPCross.Core
{
    public class App :
MvxApplication,IMvxServiceProducer<IMvxStartNavigation>
    {
        public App()
        {
            var startApplicationObject = new StartApplicationObject();

this.RegisterServiceInstance<IMvxStartNavigation>(startApplicationObject);
```

```
            }
        }
}
```

Il y a presque plus de “using” que de code réel...

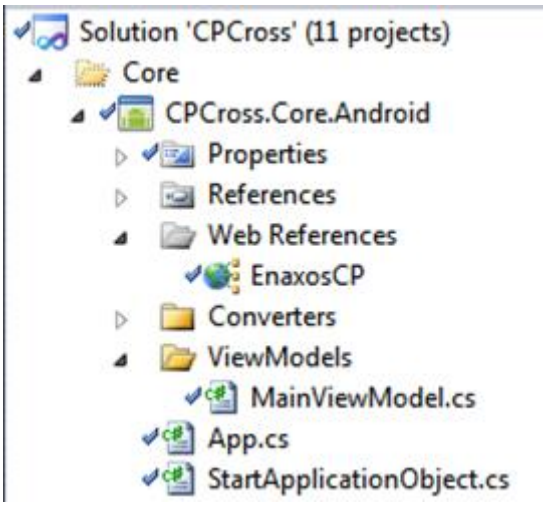
En réalité l’objet application est plus complexe que l’objet ApplicationStart que nous venons de voir. Mais dans le cas de notre code, la seule chose à faire est de créer dans le constructeur une instance de notre objet “start” puis de l’enregistrer dans le système de gestion des services (conteneur IoC).

Tout ce montage permet dans une application réelle de paramétrer finement toute la navigation et certains aspects de l’application. Ici nous n’utilisons que peu des possibilités de MVVMCross.

Et voilà...

L’application “noyau” est faite. Une compilation nous assure que tout est ok. Des tests unitaires seraient dans la réalité bienvenus pour s’assurer que le ViewModel fonctionne correctement. Je fais l’impasse sur de tels tests dans cet exemple.

Voici à quoi ressemble le projet noyau sous VS :



On retrouve la structure d’une librairie de code classique. On voit que l’icone du projet contient un petit robot Android, VS gère facilement une solution avec des plateformes différentes, c’est un vrai bonheur.

Aujourd'hui ce projet serait une PCL Visual Studio, le seul et unique projet du noyau dont l'écriture serait ainsi définitivement terminée.

On note la présence de "Web Reference", ainsi que les classes spécifiques à la mécanique de MVVMCross.

Un dernier mot sur les convertisseurs : Il s'agit ici de regrouper tous les convertisseurs qui seront utilisés dans les bindings ultérieurement, comme dans de nombreux projets Silverlight ou WPF par exemple. Même utilisation, même motivation. Mais l'écriture varie un peu car ici nous allons nous reposer sur MVVMCross.

En effet, prenons l'exemple simple de la visibilité d'un objet. Sous Xaml il s'agit d'une énumération (Vibisility avec Collapsed, Visible), mais sous Android ou iOS c'est autre chose... On retrouve dans ces environnements une propriété qui permet de dire si un objet visuel est visible ou non, ce genre de besoin est universel, mais pas les classes, les énumérations, etc, sur lesquels il repose dans sa mise en œuvre spécifique...

C'est pour cela que MVVMCross est "cross", il s'occupe de deux choses à la fois : être un framework MVVM tout à fait honorable mais aussi être un médiateur qui efface les différences entre les plateformes...

Dans mon application (dans le ViewModel), lorsque le service Web est interrogé je souhaite afficher un indicateur de type "busy" ou une progress bar en mode "indéterminé". Je verrais cela au moment de l'écriture des interfaces utilisateurs, mais je sais que j'aurai besoin d'un tel indicateur.

Le code de mon ViewModel expose une propriété booléenne IsSearching qui est à "vrai" lorsque l'application envoie une requête au serveur métier, et qui repasse à "faux" lorsque les données sont arrivées.

Je sais que sous Xaml j'aurai besoin de convertir ce booléen en Visibilty, et que sous Android il faudra faire autrement. Dilemme...

C'est là que MVVMCross va encore m'aider. Regardez le code suivant :

```
using Cirrious.MvvmCross.Converters.Visibility;

namespace CPCross.Core.Converters
{
    public class Converters
    {
```

```
        public readonly MvxVisibilityConverter Visibility = new
MvxVisibilityConverter();
    }
}
```

Je ne participe pas au concours du code le plus court, mais vous admettrez que je n'écris pas grand-chose…

Le cas de la visibilité est un tel classique que MVVMCross l'a déjà prévu. Il me suffit donc de créer une classe qui regroupera tous les convertisseurs (afin de les localiser plus facilement quand j'écrirai les UI) et de déclarer une variable convertisseur Visibility qui est une instance de MvxVisibilityConverter.

Sous Xaml (WinRT, Silverlight..) MVVMCross fournira un convertisseur retournant l'énumération correcte, sous Android il fournira la valeur attendue par la propriété Visibility (même nom, mais pas même type…).

Vous l'avez compris, l'écriture du noyau est la partie la plus essentielle de l'application après celle du serveur métier. Ce code est fait pour n'être écrit qu'une seule fois et pour être exécuté par toutes les cibles supportées. MVVMCross nous aide à gommer les différences entre les cibles pour que notre code dans les ViewModels soit universel.

Les noyaux dupliqués

Il est nécessaire maintenant de créer des "projets bis" pour les autres cibles.

Cette obligation a disparu avec l'ajout des PCL dans Visual Studio, cette étape n'a donc plus de raison d'être aujourd'hui. Je laisse ce passage car le code de l'exemple repose sur cette ancienne architecture.

En effet, la librairie de code que nous venons d'écrire, pour universelle qu'elle soit, est malgré tout un projet Android qui produira du binaire Android. Totalement inutilisable sous WinRT ou Windows Phone par exemple…

Pourtant nous nous sommes donnés du mal pour écrire un code "portable", où est l'arnaque ? Rembourseeeeez !

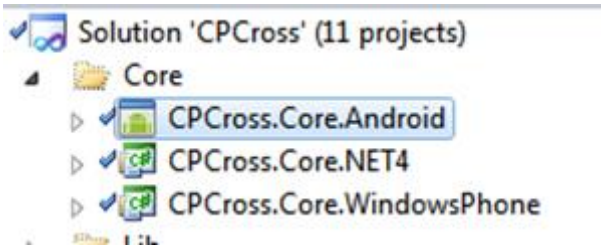
On reste calme…

Aucune arnaque en vue. Ne vous inquiétez pas. Le seul problème que nous avons c'est qu'il nous faut trouver un moyen pour compiler ce code en binaire pour chaque plateforme cible.

Et c'est facile : le chef d'orchestre qui joue les aiguilleurs c'est le fichier "csproj", le fichier projet. C'est lorsqu'on créé le projet qu'on indique quelle cible on touche.

L'astuce va tout simplement consister à créer de nouveaux projets pour chaque type de cible, projets qu'on videra de tout ce qu'il y a dedans par défaut pour le remplacer par des _liens_ vers le code du "vrai noyau", le premier.

Je vais ainsi créer un projet CPCross.Core.WindowsPhone en indiquant que je veux créer une librairie pour Windows Phone 7.1, et un projet CPCross.Core.NET4 qui sera un projet librairie de code Windows .NET 4. Ensuite, c'est le côté un peu pénible de la manipulation, je vais ajouter un à un tous les sous-répertoires en le recréant avec le même nom puis je vais ajouter des "items existants" en allant piocher dans le projet noyau original et en demandant à VS de faire un _lien_ et non une _copie_. Pas de code dupliqué. Plusieurs projets mais un seul code à écrire, cela faisait partie des contraintes de la stratégie...



Voici le résultat de cette manipulation. Le premier projet est le "vrai noyau" contenant les fichiers de code. Les deux suivants sont les projets "bis" ou "dupliqués" qui ne contiennent que des liens vers le code du premier. Chaque projet porte un nom différent avec en suffixe la cible visée, en revanche, c'est exactement le même code, donc le même espace de nom pour tous (CPCross.Core) sans aucune indication de différence.

Chaque cible doit "croire" que le code a été écrit juste pour elle et que les autres n'existent pas... C'est exactement ce que font les PCL aujourd'hui évitant la gymnastique évoquée ci-avant.

Conclusion partielle

Nous avons créé une solution Visual Studio dans laquelle nous avons ajouté la librairie MVVMCross. Nous avons ensuite écrit un projet "universel" en choisissant une cible, ici Android 1.6. Grâce à MVVMCross, MonoDroid et Visual Studio, nous avons pu écrire un premier projet définissant les ViewModels de notre (future) application.

En rusant un petit peu nous avons créé des projets "bis" pour viser les autres cibles, les autres plateformes que nous voulons supporter. En réalité ces projets bis sont des fantômes, ils ne font que contenir des liens vers le code du premier projet, il n'y a pas de code dupliqué.

Chaque projet "noyau" se compile pour produire un binaire adapté à sa cible, mais nous n'avons écrit qu'une seule version de ce code. MVVMCross nous a aidés à gommer les petites différences entre les plateformes.

La version moderne se contenterait d'un seul projet PCL qui jouerait le même rôle mais avec une mise en œuvre bien plus simple.

Nous sommes prêts à écrire les projets spécifiques à chaque plateforme ciblée...

C'est là que vous verrez enfin quelques images de l'application en action.

Une petite pause pour votre serviteur et on se retrouve dans la partie 3 !

Stratégie de développement Cross-Platform–Partie 3

Après avoir présenté la stratégie, sa motivation et ses outils dans la Partie 1, après avoir poser le décor des premiers modules communs dans la Partie 2, il est temps d'aborder la réalisation des modules spécifiques et de voir fonctionner notre applications sur différentes plateformes !

Bref Rappel

La stratégie

La Partie 1 expose une stratégie de développement cross-platform et cross form-factor basée sur l'utilisation de C#, .NET, Visual Studio, MonoDevelop, MonoTouch (pour iOS) et MonoDroid (pour Android), le tout avec le framework MVVM cross-platform MVVMCross.

Cet ensemble d'outils permet avec un seul langage de programmation, un seul EDI (pour iOS il faut toutefois utiliser MonoDevelop sous Mac), une seule plateforme

(.NET MS et Mono) d'attaquer onze cibles différentes : les smartphones et tablettes Apple, Android, Windows "classique", Windows 8 (WinRT), Windows Phone...

La stratégie se base aussi sur une **séparation particulière du code**, l'essentiel de celui-ci n'ayant besoin d'être écrit qu'une seule fois.

On trouve dans la partie immuable à 100% le serveur métier regroupant l'intelligence et le savoir-faire métier associé avec un projet « noyau », une PCL de Visual Studio qui peut souvent remplacer aussi le serveur métier.

Dans la partie la plus fluctuante ou la variabilité assumée peut atteindre 100% on trouve bien entendu les différents projets implémentant les Vues de façon spécifique pour chaque cible.

Le maitre mot de la stratégie est "**plus le code devient spécifique, moins on écrit de code**". Grâce à cette approche supporter de nombreuses cibles différentes ne coute pas très cher et se limite à la création des Vues sans remise en cause ni des ViewModels ni du code métier.

La concentration du savoir-faire du développeur avec un seul langage, un seul EDI, un seul framework (C#, VS/MonoDevelop, .NET) évite la couteuse mise en place de formations lourdes ou l'embauche de personnel qualifié par cible visée. **Tout le monde s'y retrouve, le développeur qui voit son champ d'action élargi donc son travail devenir plus intéressant, le DSI qui ne gère qu'une seule équipe et une formation minimale, l'entreprise qui diminue ses couts de réalisation et de maintenance tout en assurant sa présence sur de multiples médias et cibles.**

L'exemple de code

La Partie 2 pose le décor de l'exemple de code utilisé pour illustrer la stratégie proposée.

En repartant d'un exemple datant de 2008 (les codes postaux français) utilisant un service Web et un frontal Silverlight Web nous allons rajouter de nouvelles cibles absolument non prévues à cette époque et ce **sans remettre en cause une seule ligne de programmation du serveur métier**. Cela démontre au passage l'énorme avantage de cette stratégie qui **pérennise le code au travers des années, des modes, et des technologies changeantes**.

Après avoir créé la solution VS, y avoir ajouté les librairies MVVMCross, nous créons le ViewModel principal (l'exemple n'aura qu'une page sans navigation pour simplifier).

Ce ViewModel fait partie du projet dit "'noyau". Il est écrit une fois pour toute et va être utilisé pour créer des projets "bis" ciblant chacun une plateforme spécifique. **Il n'y a pas de duplication de code**, les projets bis sont créés en ajoutant des liens

vers le code du projet noyau original. Seul le fichier projet (csproj) diffère puisque c'est lui qui permet de sélectionner le compilateur à utiliser. *Avec l'utilisation des PCL il n'est aujourd'hui plus nécessaire de dupliquer les projets noyau, un seul projet suffit.*

Le projet noyau se voit agrémenté de quelques classes propres au fonctionnement de MVVMCross et des convertisseurs de valeurs qui seront utilisés par les Vues. On utilise ici les mécanismes de MVVMCross qui gomme les nuances entres les OS de façon à pouvoir écrire des convertisseurs qui marcheront aussi bien sous Android que sous WinRT ou Windows Phone sans rien avoir à recompiler ni modifier.

L'étape du jour

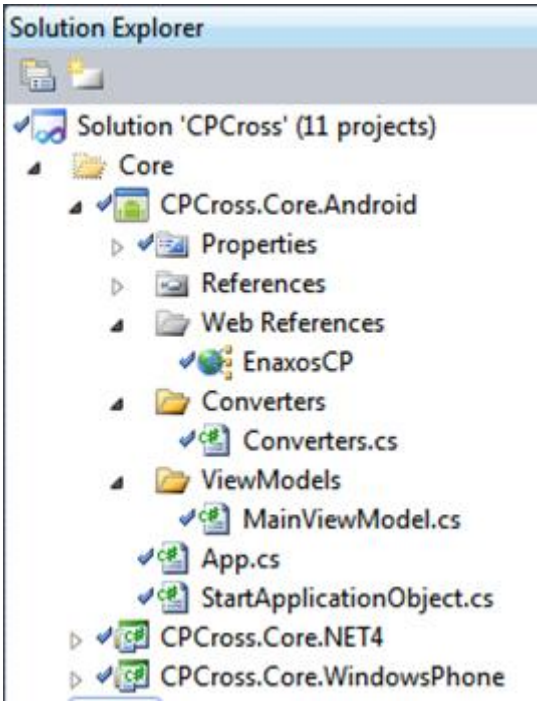
Nous possédons le serveur métier (celui des codes postaux), nous possédons le cœur de l'application écrit une seule fois en C# (le noyau). Nous allons écrire maintenant les projets spécifiques à chaque plateforme sélectionnée pour notre exemple.

Se souvenir : Il ne s'agit que d'un exemple simplifié à l'extrême pour ne pas être trop long. Le serveur métier est ici réduit à sa plus simple expression, le noyau ne définit qu'un seul ViewModel, les parties spécifiques ne couvrent que quelques cibles seulement et n'exposent qu'une seule vue. Le lecteur, j'en suis certain, saura transposer cette démonstration simplifiée à des cas plus complexes et plus réels.

A noter: Je ne détaille pas ici tout le fonctionnement de MVVMCross, j'ai prévu d'y revenir dans un article spécialement consacré à cette librairie.

Le Noyau

Je l'ai succinctement décrit dans la Partie 2. Voici son organisation :



Le projet déclare un espace de nom CPCross.Core, le projet lui-même portant en suffixe le nom de sa cible (.Android, .NET4 ou .WindowsPhone comme on le voit sur l'image ci-dessus). Avec une PCL, un seul noyau, un seul projet est nécessaire.

Le code du noyau est le suivant :

```
using System;
using System.Collections.Generic;
#if WINDOWS_PHONE
using CPCross.Core.WindowsPhone.EnaxosCP;
#else
using CPCross.Core.EnaxosCP;
#endif
using Cirrious.MvvmCross.Commands;
using Cirrious.MvvmCross.ViewModels;


namespace CPCross.Core.ViewModels
{
    public class MainViewModel : MvxViewModel
    {
#if WINDOWS_PHONE
        private EnaxosFrenchZipCodesSoapClient service;
#else
        private EnaxosFrenchZipCodes service;
#endif
        private readonly List<string> errors = new List<string>();
        private string searchField;
        private bool isSearching;
        private bool sortByZip;
        private bool searchAnyWhere;

        public MainViewModel()
        {
            // commanding
            ClearErrorsCommand = new MvxRelayCommand(clearErrors, () =>
HasErrors);
            GetCitiesByNameCommand = new MvxRelayCommand(getCitiesByName,
() => !IsSearching);
            GetCitiesByZipCommand = new MvxRelayCommand(getCitiesByZip, ()
=> !IsSearching);
```

```
            // init service
            createService();
        }

        //add error to error list
        private void addError(string errorMessage)
        {
            errors.Add(errorMessage);
            FirePropertyChanged("Errors");
            FirePropertyChanged("HasErrors");
            ClearErrorsCommand.RaiseCanExecuteChanged();
        }

        private void clearErrors()
        {
            errors.Clear();
            FirePropertyChanged("Errors");
            FirePropertyChanged("HasErrors");
            ClearErrorsCommand.RaiseCanExecuteChanged();
        }


        private void createService()
        {
            try
            {
#if WINDOWS_PHONE
                service = new EnaxosFrenchZipCodesSoapClient();
#else
                service = new EnaxosFrenchZipCodes();
#endif
                service.FullVersionCompleted +=
service_FullVersionCompleted;
                service.FullVersionAsync();
            }
            catch (Exception ex)
            {
                addError(ex.Message);
            }
        }
```

```
        private void service_FullVersionCompleted(object sender,
FullVersionCompletedEventArgs e)
        {
            if (e.Error == null)
            {
                Title = e.Result.Title;
                Version = e.Result.Version;
                Description = e.Result.Description;
                Copyrights = e.Result.Copyrights;
                WebSite = e.Result.WebSite;
                Blog = e.Result.Blog;
                MiniInfo = e.Result.Version + " " + e.Result.Copyrights;
                FirePropertyChanged("Title");
                FirePropertyChanged("Version");
                FirePropertyChanged("Description");
                FirePropertyChanged("Copyrights");
                FirePropertyChanged("WebSite");
                FirePropertyChanged("Blog");
                FirePropertyChanged("MiniInfo");
                return;
            }
            addError(e.Error.Message);
        }


        // search by name
        private void getCitiesByName()
        {
            if (!canSearch()) return;
            IsSearching = true;
            service.GetCitiesByNameCompleted +=
service_GetCitiesByNameCompleted;
            service.GetCitiesByNameAsync(searchField, searchAnyWhere,
sortByZip);
        }

        private void service_GetCitiesByNameCompleted(object sender,
GetCitiesByNameCompletedEventArgs e)
        {
            InvokeOnMainThread(() => IsSearching = false);
            service.GetCitiesByNameCompleted -=
```

```
service_GetCitiesByNameCompleted;
            if (e.Error == null)
            {
                InvokeOnMainThread(() =>
                    {
                        Result = e.Result==null ? null : new
List<CityObject>(e.Result);
                        FirePropertyChanged("Result");

                    });
                return;
            }
            InvokeOnMainThread(() =>
                {
                    addError(e.Error.Message);
                    Result = new List<CityObject>();
                    FirePropertyChanged("Result");
                });
        }


        // search by zip
        private void getCitiesByZip()
        {
            if (!canSearch()) return;
            IsSearching = true;
            service.GetCitiesByZipCompleted +=
service_GetCitiesByZipCompleted;
            service.GetCitiesByZipAsync(searchField, sortByZip);
        }

        private void service_GetCitiesByZipCompleted(object sender,
GetCitiesByZipCompletedEventArgs e)
        {
            InvokeOnMainThread(() => IsSearching = false);
            service.GetCitiesByZipCompleted -=
service_GetCitiesByZipCompleted;
            if (e.Error == null)
            {
                InvokeOnMainThread(() =>
                    {
                        Result = e.Result==null ? null : new
```

```
List<CityObject>(e.Result);
                        FirePropertyChanged("Result");
                    });
                return;
            }
            InvokeOnMainThread(() =>
                {
                    addError(e.Error.Message);
                    Result = new List<CityObject>();
                    FirePropertyChanged("Result");
                });
        }

        // internal test
        private bool canSearch()
        {
            return !IsSearching && !string.IsNullOrWhiteSpace(searchField);
        }

        // service information
        public string Title { get; private set; }
        public string Description { get; private set; }
        public string Version { get; private set; }
        public string Copyrights { get; private set; }
        public string WebSite { get; private set; }
        public string Blog { get; private set; }

        public string MiniInfo { get; private set;}

        // errors
        public List<string> Errors
        {
            get
            {
                return errors;
            }
        }

        public bool HasErrors
        {
            get
            {
```

```
            return errors.Count > 0;
        }
    }

    // search result
    public List<CityObject> Result { get; set; }

    // search field
    public string SearchField
    {
        get
        {
            return searchField;
        }
        set
        {
            if (searchField == value) return;
            searchField = value;
            FirePropertyChanged("SearchField");
        }
    }

    // result sort
    public bool SortByZip
    {
        get
        {
            return sortByZip;
        }
        set
        {
            if (sortByZip == value) return;
            sortByZip = value;
            FirePropertyChanged("SortByZip");
        }
    }

    // search mode
    public bool SearchAnyWhereInField
    {
        get
        {
```

```
                return searchAnyWhere;
            }
            set
            {
                if (searchAnyWhere == value) return;
                searchAnyWhere = value;
                FirePropertyChanged("SearchAnyWhereInField");
            }
        }

        // search state
        public bool IsSearching
        {
            get { return isSearching; }
            private set
            {
                if (isSearching == value) return;
                isSearching = value;
                FirePropertyChanged("IsSearching");
                GetCitiesByNameCommand.RaiseCanExecuteChanged();
                GetCitiesByZipCommand.RaiseCanExecuteChanged();
            }
        }

        // commanding
        public MvxRelayCommand ClearErrorsCommand { get; private set; }
        public MvxRelayCommand GetCitiesByNameCommand { get; private set; }
        public MvxRelayCommand GetCitiesByZipCommand { get; private set; }
    }
}
```

Ce code est réellement très classique pour un ViewModel suivant le pattern MVVM. On y trouve bien entendu des petites spécificités propres à MVVMCross mais quand on pratique de nombreux frameworks MVVM on s'y retrouve facilement, tous ne font que régler les mêmes problèmes de façon finalement assez proche.

Des propriétés et des commandes. Voilà ce qu'est un ViewModel dans la pratique. C'est bien ce que reflète le code ci-dessus.

Lorsque le ViewModel est instancié il fait un premier appel au service pour aller chercher les informations de base retournées par ce dernier (version, copyrights, etc).

Ensuite les recherches sont effectuées à la demande via les commandes exposées.

Le ViewModel expose même une liste des erreurs qui stocke les exceptions éventuelles. Dans notre exemple ne mettant en œuvre qu'une seule Vue sans navigation nous n'utiliserons pas ce mécanisme. Le code source étant publié en fin d'article, à vous de jouer et d'ajouter ce qu'il faut pour afficher les erreurs s'il y en a (il y a une commande ClearErrors qui peut être bindée à un Button pour effacer la liste).

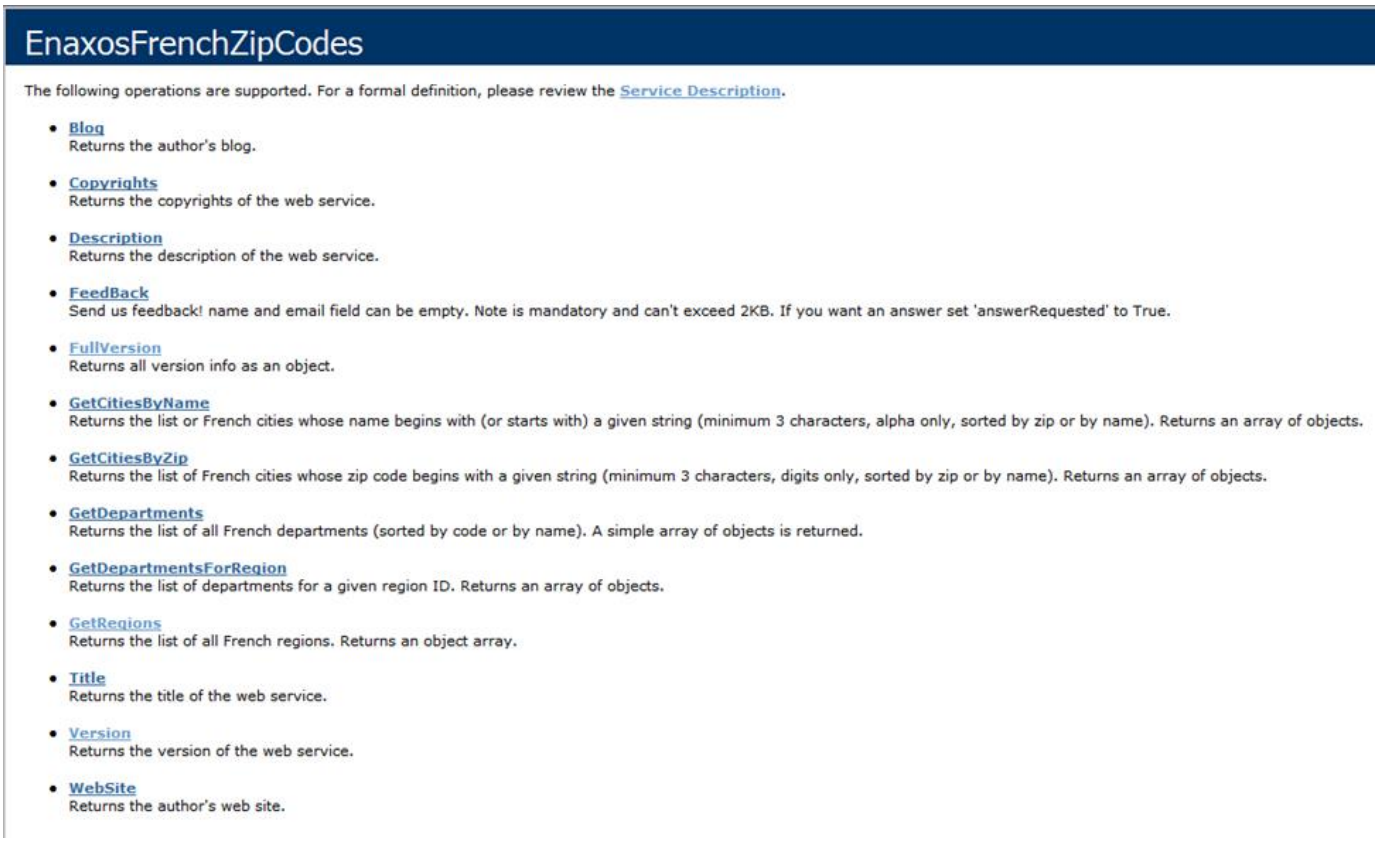
Le serveur métier

Le serveur métier est généralement un service Web (ou un ensemble de services web) le plus standard possible.

Ici j'ai repris le serveur d'une démo écrite en 2008 pour Silverlight. Il marche depuis cette époque sans aucun souci (sur une base de données Access/Jet ... c'est pour dire à quel point ce n'est pas un code moderne !).

C'est un simple '"asmx" bien loin de sophistications parfois inutiles de WCF. Sa simplicité lui donne l'avantage de l'universalité, et le fait que je puisse aujourd'hui le réutiliser sans changer une ligne de code prouve de façon éclatante que la stratégie proposée permet réellement de pérenniser le code en le protégeant des modes et des technologies clientes changeantes.

Le WDSL nous indique ceci :



S'agissant d'un exemple, ce "serveur métier" contient peu de "savoir métier" voire aucun... Mais il pourrait contenir des centaines de services sophistiqués, cela ne

changerait rien à notre stratégie. Dans un tel cas il serait d'ailleurs certainement segmenté en plusieurs services différents tournant éventuellement sur des machines différentes.

L'exemple multi-plateforme n'utilisera d'ailleurs qu'une partie des services exposés.

Une première cible : Windows "classique" en mode Console

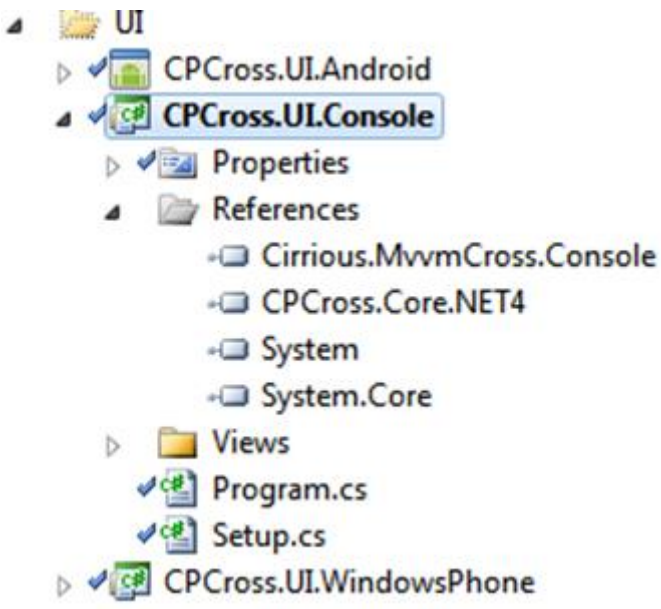
Il est temps d'ajouter une première cible "visuelle", un client spécifique.

J'ai choisi avec prudence Windows "classique" avec un mode désuet mais très pratique pour m'assurer que tout fonctionne : la console !

Certes cela fait très MS-DOS, mais cela prouve que MVVMCross couvre vraiment tous les cas de figure même les plus étranges et surtout cela m'assurait de pouvoir déboguer le noyau sans aucune complication liée à Android ou Windows Phone ou autre plateforme. Bien entendu cela montre malgré tout une première cible très différente des autres et il faut voir ici la console Win32 comme le symbole de tout ce qui peut être fait sous Windows "classique" comme WPF par exemple (Windows XP, Vista, 7, Windows 8 classic desktop).

Tous les projets d'UI portent le même nom que le projet principal avec le suffixe correspondant à leur cible. Ainsi, ce projet s'appellera CPCross.UI.Console.

Tous ajoutent en référence les librairies spécifiques de MVVMCross (ici Cirrious.MvvmCross.Console), et tous pointent le noyau.



L'image ci-dessus nous montre l'arborescence de la solution au niveau du sous-répertoire de solution "UI". On y trouve les projets d'UI dont le projet console.

Tous les projets ont la même structure. On trouve ainsi un répertoire Views qui contiendra l'unique Vue de notre application. On note aussi la présence de deux classes Program.cs et Setup.cs, propres au mode choisi (la console .NET 4) et à MVVMCross (le setup qui initialise l'application et la navigation vers la vue de départ).

La programmation de la vue en mode Console est très classique : des Console.Writeln(xxx) se trouvant dans une méthode appelée automatiquement par MVVMCross à chaque modification du ViewModel (Binding minimaliste) ce qui permet de rafraichir l'affichage, les saisies se faisant via une méthode spéciale aussi qui transforme les saisies clavier en appel aux commandes du ViewModel ou en modification des propriétés de ce dernier.

Le code cette "vue" un peu spéciale est le suivant :

```
using System;
using System.Collections.Generic;
using System.Text;
using CPCross.Core.ViewModels;
using Cirrious.MvvmCross.Console.Views;

namespace CPCross.Console.Views
{
    class MainView : MvxConsoleView<MainViewModel>
    {
        protected override void OnViewModelChanged()
        {
            base.OnViewModelChanged();
            ViewModel.PropertyChanged += (sender, args) =>
refreshDisplay();
            refreshDisplay();
        }

        public override bool HandleInput(string input)
        {
            switch (input)
            {
                case "n" :
                case "N" :
                    if (ViewModel.GetCitiesByNameCommand.CanExecute())
                        ViewModel.GetCitiesByNameCommand.Execute();
                    return true;
                case "z" :
                case "Z" :
```

```
                    if (ViewModel.GetCitiesByZipCommand.CanExecute())
                        ViewModel.GetCitiesByZipCommand.Execute();
                    return true;
                case "c" :
                case "C" :
                    if (ViewModel.Result!=null) ViewModel.Result.Clear();
                    if (ViewModel.ClearErrorsCommand.CanExecute())
ViewModel.ClearErrorsCommand.Execute();
                    refreshDisplay();
                    return true;
                case "s" :
                case "S" :
                    ViewModel.SearchAnyWhereInField = false;
                    refreshDisplay();
                    return true;
                case "a" :
                case "A" :
                    ViewModel.SearchAnyWhereInField = true;
                    refreshDisplay();
                    return true;
                default :
                    if (input.Trim().ToUpper().StartsWith("SET ") &&
input.Length > 3)
                    {
                        ViewModel.SearchField = input.Substring(3).Trim();
                        return true;
                    }
                    break;
            }
            return base.HandleInput(input);
        }

        private void refreshDisplay()
        {
            System.Console.BackgroundColor = ConsoleColor.Black;
            System.Console.ForegroundColor = ConsoleColor.White;
            System.Console.Clear();
            System.Console.ForegroundColor = ConsoleColor.Yellow;
            System.Console.WriteLine("CP-Cross / .NET 4.0 Console UI");
            System.Console.WriteLine();

            System.Console.ForegroundColor = ConsoleColor.Gray;
```

```
            System.Console.WriteLine(ViewModel.Title + " - Version: " +
ViewModel.Version);
            System.Console.WriteLine(ViewModel.Copyrights);
            System.Console.WriteLine(ViewModel.Description);
            System.Console.WriteLine();


            System.Console.ForegroundColor = ConsoleColor.White;
            System.Console.WriteLine("Search by <N>ame, by <Z>ip. <SET>
{text} to set search term. <C>lear result."+Environment.NewLine+"<S>tarts
with or <A>nywhere");
            System.Console.WriteLine();
            System.Console.ForegroundColor = ConsoleColor.Cyan;
            System.Console.WriteLine("Current Search Term :
"+(string.IsNullOrWhiteSpace(ViewModel.SearchField)?"<none>":ViewModel.Sear
chField) +" - Current option:
"+(ViewModel.SearchAnyWhereInField?"Anywhere":"Starts with"));
            System.Console.ForegroundColor = ConsoleColor.White;
            System.Console.WriteLine();

            if (ViewModel.IsSearching)
            {
                System.Console.ForegroundColor = ConsoleColor.Red;
                System.Console.WriteLine("Searching...");
                return;
            }

            if (ViewModel.Result!=null && ViewModel.Result.Count>0)
            {
                System.Console.ForegroundColor=ConsoleColor.Green;
                var limit = Math.Min(10, ViewModel.Result.Count);
                for(var i =0;i<limit;i++)
                    System.Console.WriteLine(ViewModel.Result[i].City+";
"+ViewModel.Result[i].ZipCode+"; "+ViewModel.Result[i].DepartmentName);
            }
            if (ViewModel.Result!=null && ViewModel.Result.Count>10)
                System.Console.WriteLine("... ("+ViewModel.Result.Count+"
results. 10 firsts displayed)");
            if(ViewModel.Result==null || (ViewModel.Result!=null &&
ViewModel.Result.Count==0))
            {
                System.Console.ForegroundColor=ConsoleColor.Blue;
```

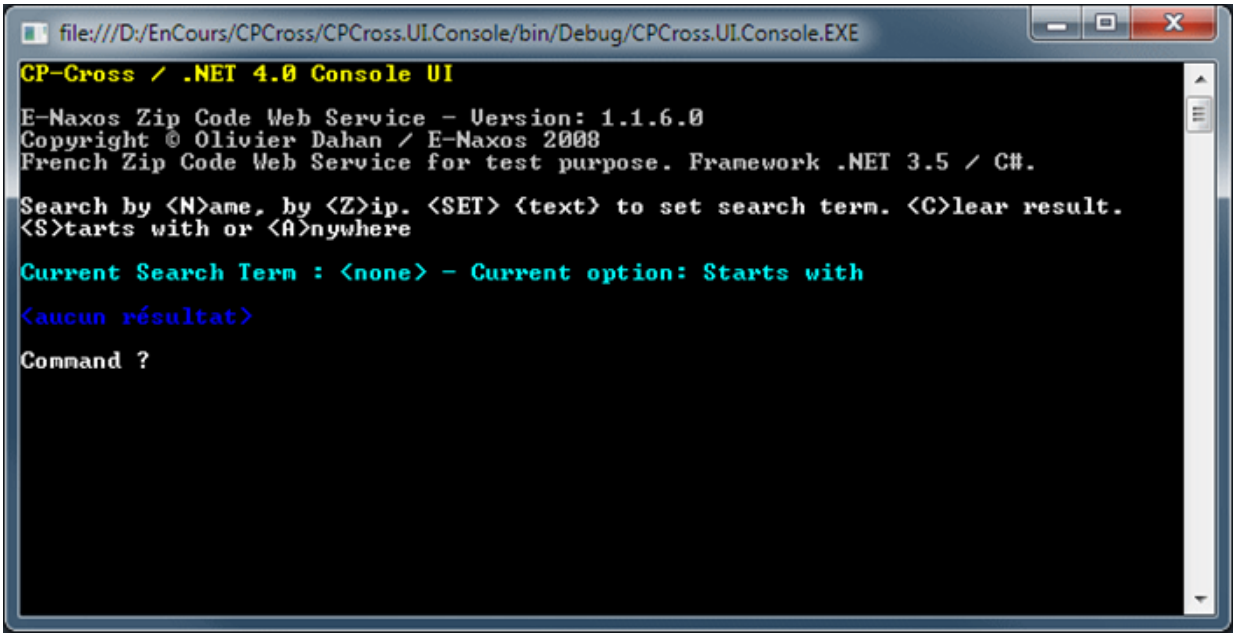
```
                System.Console.WriteLine("<aucun résultat>");
            }

            if (ViewModel.Errors.Count>0)
            {
                System.Console.ForegroundColor=ConsoleColor.DarkRed;
                foreach (var error in ViewModel.Errors)
                {
                    System.Console.WriteLine("- "+error);
                }
            }

            System.Console.WriteLine();
            System.Console.ForegroundColor=ConsoleColor.White;
            System.Console.Write("Command ? ");
            System.Console.ForegroundColor=ConsoleColor.Yellow;
        }
    }
}
```

On voit que ce code est une 'vraie' vue qui descend de MxConsoleView, ce qui permet à MVVMCross de gérer la dynamique des changements de propriétés et les saisies clavier de façon transparente. Rappelez-vous, notre ViewModel qui est derrière n'a absolument pas connaissance de cette utilisation qui en est faite...

Le couplage Vue/ViewModel est assuré par MVVMCross de façon automatique (la vue déclare dans son entête de classe qu'elle est liée à MainViewModel). Il reste possible comme avec tous les frameworks MVVM un peu sophistiqués de redéfinir les routes entre Vues et ViewModel, ici nous utilisons le comportement par défaut.

Le film...

Il est intéressant de voir tout cela en action. Quelques captures d'écran feront l'affaire et seront plus rapides à commenter qu'une capture vidéo :



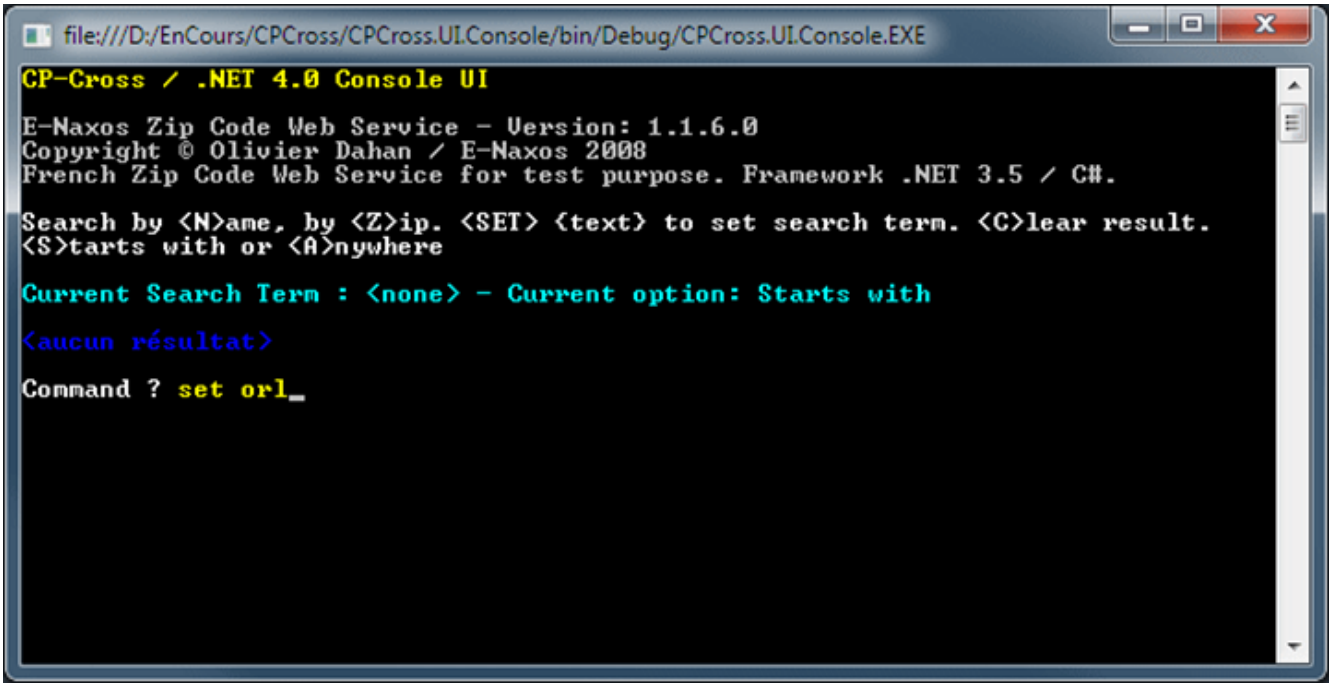
Etape 1 : l'application s'affiche. Pendant un cours instant les informations en début de page affichent les valeurs par défaut initialisées par le code (partie supprimée de la copie du code ci-dessus). Puis, une fois que le service a répondu la page se met à jour et affiche les informations du Service à jour. On note le copyright de 2008 retourné par le service Web originel.

S'agissant d'un mode console j'ai prévu un jeu de commande minimaliste : <N> pour chercher par nom, <Z> par zip (code postal), la commande SET <texte> permet de saisir la valeur à chercher. <C> pour clear (effacer le dernier résultat). Les commandes <A> et <S> permettent d'indiquer si le terme saisi doit être cherché en début de chaine uniquement ou n'importe où dans le nom (en mode Zip seul le début de chaine est contrôlé).

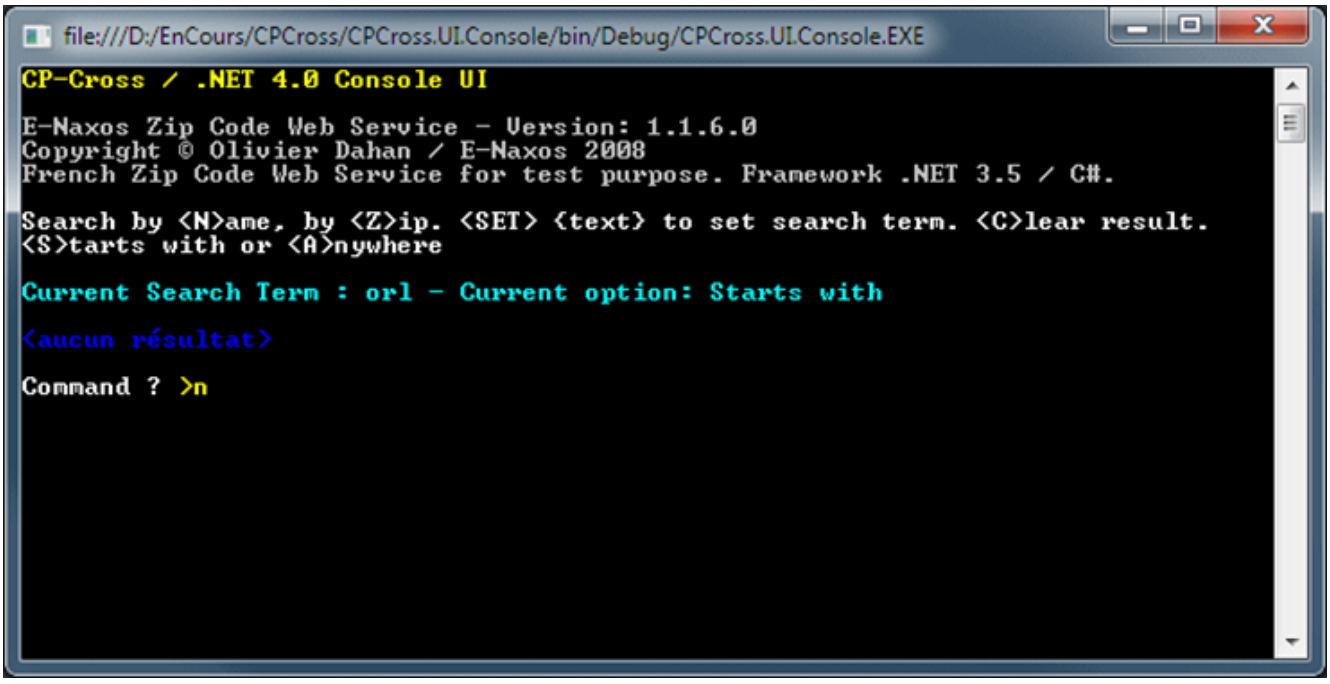
L'application affiche le terme courant (en ce moment <none>, aucun), ainsi que les options sélectionnées (recherche en début de chaine).

Il n'y a aucun résultat (pourquoi c'est en français ? une incohérence de ma part, je fais tout en anglais mais le naturel revient parfois à la sournoise !).

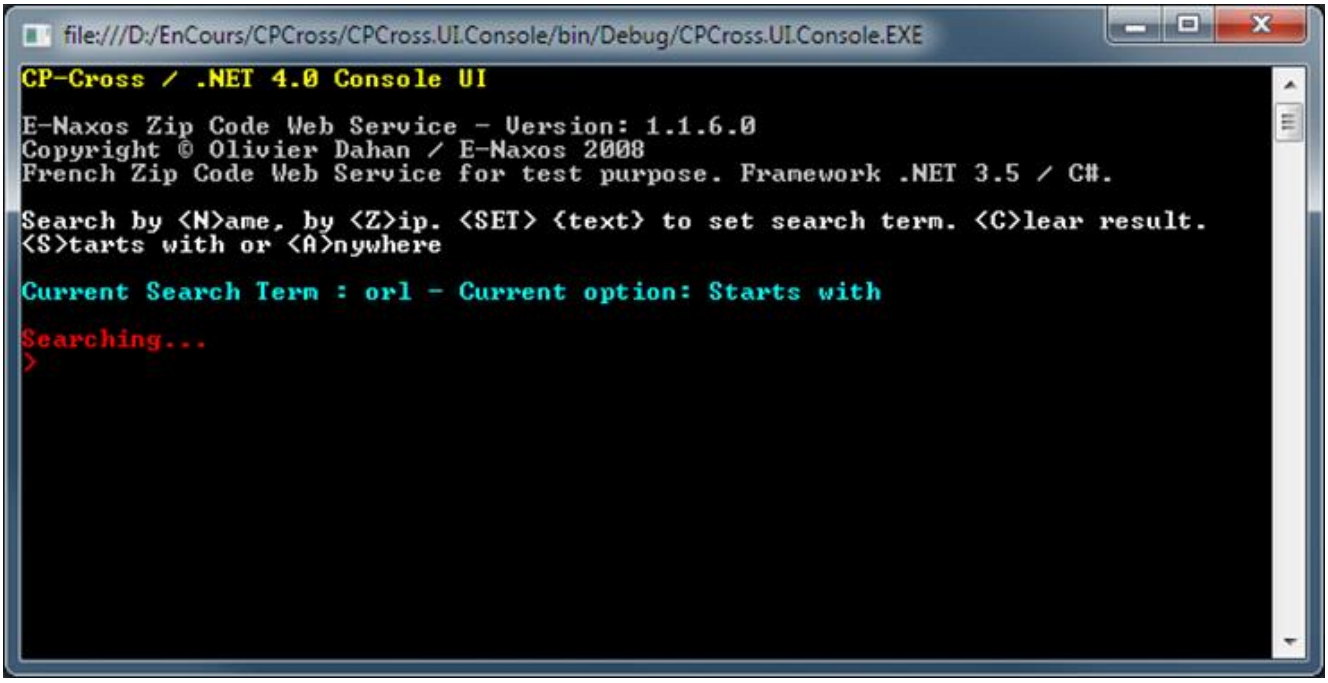
L'application est en attente de commande. Il va falloir saisir un terme à chercher.



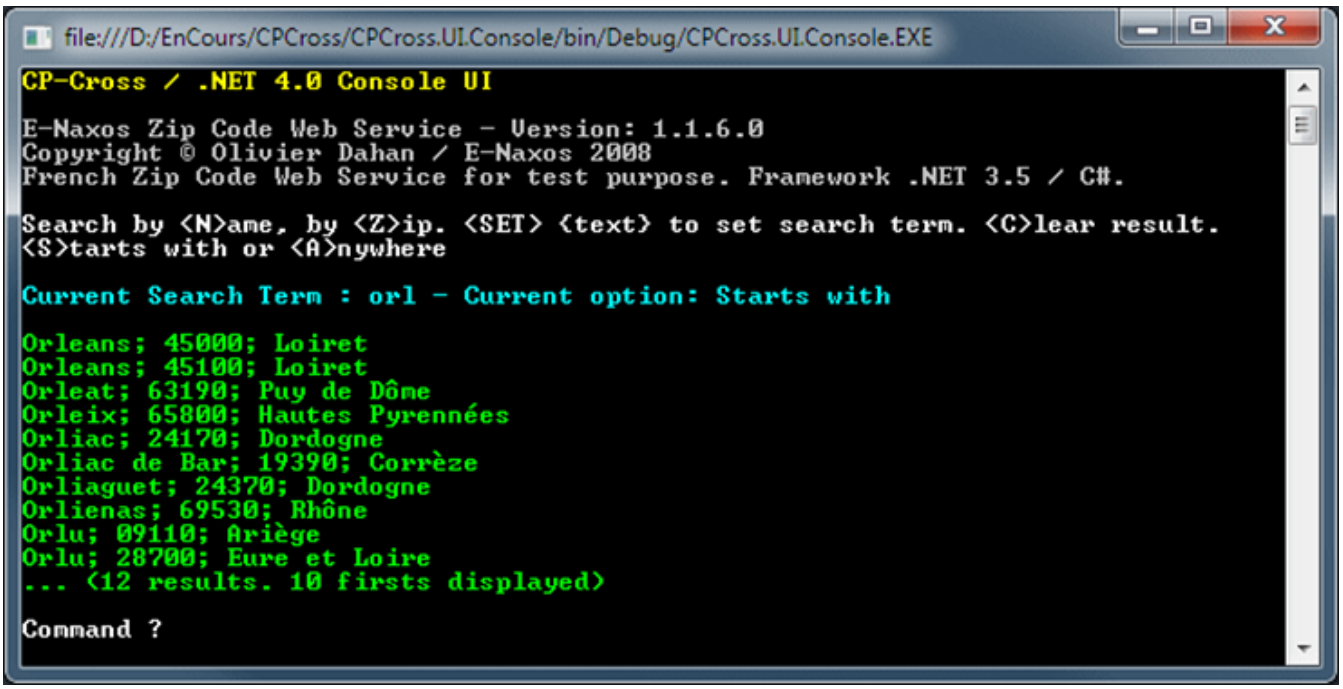
En tapant la commande “set xxxx” j’initialise la variable de recherche du ViewModel. Les lettre “orl” sont tapées.



Le terme a été validé, on voit dans la ligne turquoise que le “Current Search Term” est bien égal à “orl”. Je tape la commande “n” pour recherche par Nom.



La commande “N” vient d’être validée. Le ViewModel expose un booléen “IsSearching” qui est exploité ici pour afficher le texte rouge “Searching...”. Nous exploitons bien toutes les possibilités du ViewModel, même dans sa dynamique...



Le résultat s’affiche... Nom de la ville, code postal, département. Pour que cela tienne en une page, seuls les 10 premiers résultats sont affichés ce que nous rappelle la dernière ligne verte (en indiquant que l’application à trouvé 12 réponses).

On peut bien entendu faire la même chose avec un code postal partiel ou complet.

Ce qui est intéressant ici est bien entendu le fait d'avoir un frontal .NET 4 pour Windows classique qui symbolise tous les frontaux qu'on peut écrire pour cette cible en natif comme WPF. Rares seront les applications qui utiliseront le mode Console, j'en suis convaincu n'ayez crainte 

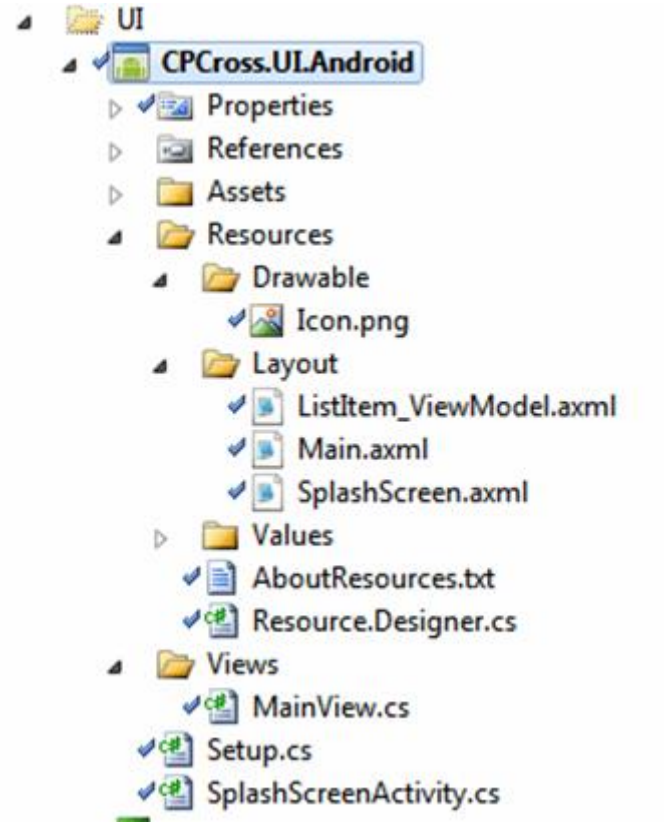
Une Seconde Cible : Android

Me voici rassurer sur le fonctionnement du "noyau", tout se déroule comme prévu. Le serveur est distant (en Angleterre) nous sommes bien dans une configuration "réelle".

Il est temps de sortir MonoDroid de sa cachette...

C'est très simple, puisqu'il est installé sur ma machine, sous Visual Studio je n'ai qu'à créer un nouveau projet... pour Android. Aussi compliqué que cela !

Voici la structure du projet :



Je n'entrerai pas dans les détails d'un projet MonoDroid ni même dans les arcanes du développement sous Android, cela s'écarterait bien trop du sujet.

On retrouve ici tout ce qui fait un projet Android de ce type : des Assets, des ressources dont notamment "Layout' qui contient les fichiers Axml qui définissent les Vues. On trouve aussi, comme sous Windows Phone, la notion de Splash Screen

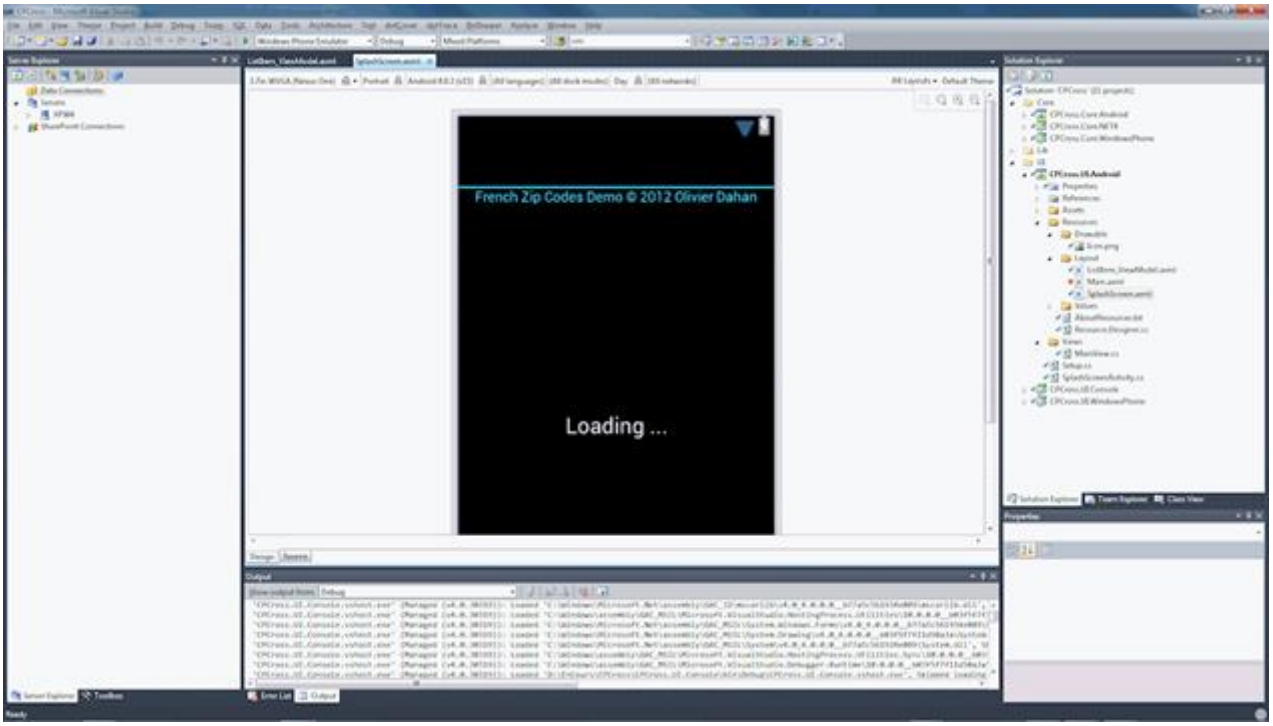
affiché automatiquement pendant le chargement de l'application ainsi que la classe "Setup" qui fait partie du mécanisme de MVVMCross.

Côté références, le projet se comporte comme le précédent : il pointe la librairie MVVMCross adaptée à la cible (Cirrious.MvvmCross.Android) ainsi que le noyau.

Le répertoire Views fait partie du mécanisme MVVMCross : les vues sont décrites par une classe que le framework peut retrouver facilement, le code ne faisant rien de spécial en général à ce niveau :

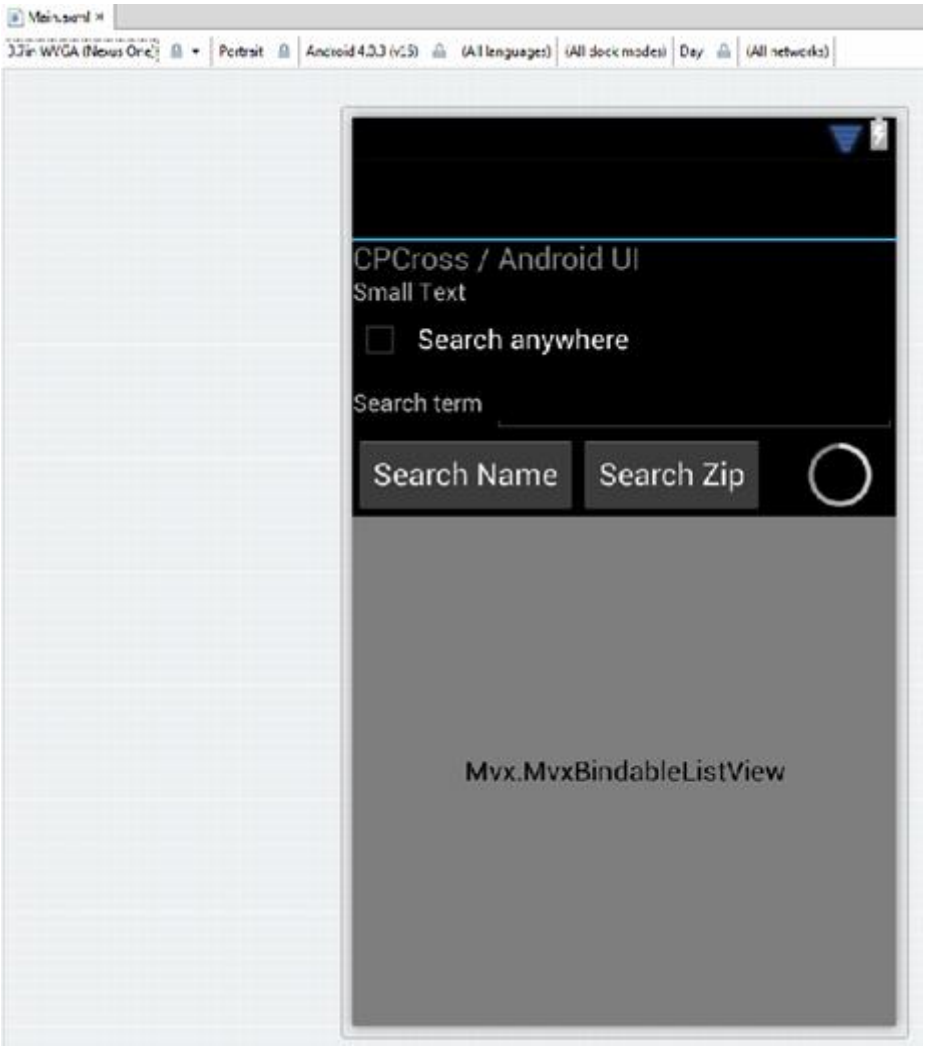
```
using Android.App;
using Cirrious.MvvmCross.Binding.Android.Views;
using CPCross.Core.ViewModels;
using Cirrious.MvvmCross.Converters.Visibility;

namespace CPCross.UI.Android.Views
{
    [Activity]
    public class MainView : MvxBindingActivityView<MainViewModel>
    {
        protected override void OnViewModelSet()
        {
            SetContentView(Resource.Layout.Main);
        }
    }
}
```

L'attribut d'activité est propre à l'OS Android (il faut voir cela comme une tâche, un processus). Le code se borne donc à spécifier la vue qu'il faut charger (Resource.Layout.Main).



Ci-dessus on voit l'écran SplashScreen en cours de conception via le designer visuel Android qui s'est intégré à Visual Studio.

Ci-dessous l'écran principal avec le même designer visuel, sous MonoDevelop (on ne peut pas voir la différence sur cette capture il faudrait voir les menus pour s'apercevoir que le look est légèrement différent de VS) :



On retrouve ici tout ce qui fait une application Android, le format téléphone (on peut choisir bien entendu n'importe quelle résolution), des messages affichés, une case à cocher, des boutons, une zone la liste résultat, un indicateur d'attente, etc.

La liste n'est pas une simple Listbox mais une liste fournie par MVVMCross pour être bindable. Il n'y a pas de Binding sous Android sauf si on ruse un peu...

D'ailleurs pour ceux qui maitrisent déjà Xaml, comprendre Axml (humm le nom est vraiment proche !) n'est pas sorcier, la preuve, voici le code de cette page :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="CPCross / Android UI"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:id="@+id/textView1"
        android:textColor="#ff808080" />
    <TextView
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
        android:text="Small Text"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txtInfo"
        android:textColor="#ffc0c0c0"
        local:MvxBind="{'Text':{'Path':'MiniInfo'}}" />
    <LinearLayout
        android:orientation="horizontal"
        android:minWidth="25px"
        android:minHeight="25px"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/linearLayout1">
        <CheckBox
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
            android:text="Search anywhere"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:id="@+id/cbAnyWhere"
            local:MvxBind="{'Checked':{'Path':'SearchAnyWhereInField'}}" />
    </LinearLayout>
    <LinearLayout
        android:orientation="horizontal"
        android:minWidth="25px"
        android:minHeight="25px"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/linearLayout2">
        <TextView
            android:text="Search term"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:id="@+id/textView2"
            android:gravity="center" />
        <EditText
```

```
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:id="@+id/edSearchTerm"
            android:layout_marginLeft="8px"
            local:MvxBind="{'Text':{'Path':'SearchField'}}" />
    </LinearLayout>
    <LinearLayout
        android:orientation="horizontal"
        android:minWidth="25px"
        android:minHeight="25px"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/linearLayout3">
        <Button
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
            android:text="Search Name"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:id="@+id/btnSearchName"
            local:MvxBind="{'Click':{'Path':'GetCitiesByNameCommand'}}" />
        <Button
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
            android:text="Search Zip"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:id="@+id/btnSearchZip"
            local:MvxBind="{'Click':{'Path':'GetCitiesByZipCommand'}}" />
        <ProgressBar
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
            android:text="Search Zip"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:id="@+id/progressBar1"
            android:indeterminate="true"
            android:indeterminateBehavior="cycle"
            android:indeterminateOnly="true"
            android:layout_gravity="center_vertical"
            android:layout_marginLeft="30px"
            local:MvxBind="{'Visibility':{'Path':'IsSearching',
'Converter':'Visibility'}}" />
    </LinearLayout>
```

```
    <Mvx.MvxBindableListView
xmlns:local="http://schemas.android.com/apk/res/CPCross.UI.Android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        local:MvxBind="{'ItemsSource':{'Path':'Result'}}"
        local:MvxItemTemplate="@layout/listitem_viewmodel" />
</LinearLayout>
```

Vous remarquerez la similitude avec Xaml. Et vous noterez l'espace de nom "local" dans une tradition identique à celle qu'on utilise sous Xaml pour référencer du code interne à l'application. Ici cela sert à accéder aux facilités, aux extensions fournies par MVVMCross pour le Binding.

La Syntaxe du Binding MVVMCross tente d'être la plus proche possible de celle de Xaml en utilisant une sérialisation JSon.

Les dernières versions de MvvmCross proposent un binding beaucoup simple encore à écrire (voir les 12 vidéos de formation sur YouTube).

Par exemple, la Checkbox est liée ainsi :

local:MvxBind="{'Checked':{'Path':'SearchAnyWhereInField'}}"

C'est à dire que la propriété Checked est liée à la propriété SearchAnyWhereInField, déclarée dans le ViewModel du projet noyau. Le ViewModel est le datacontext implicite de cette vue, MVVMCross s'en charge. Cette notion est exactement la même qu'en Xaml, une fois encore...

Plus subtile est la liaison de l'indicateur busy :

local:MvxBind="{'Visibility':{'Path':'IsSearching', 'Converter':'Visibility'}}"

Ici c'est la propriété Visibility (même nom, décidément !) de la ProgressBar qui est liée à la propriété IsSearching du ViewModel.

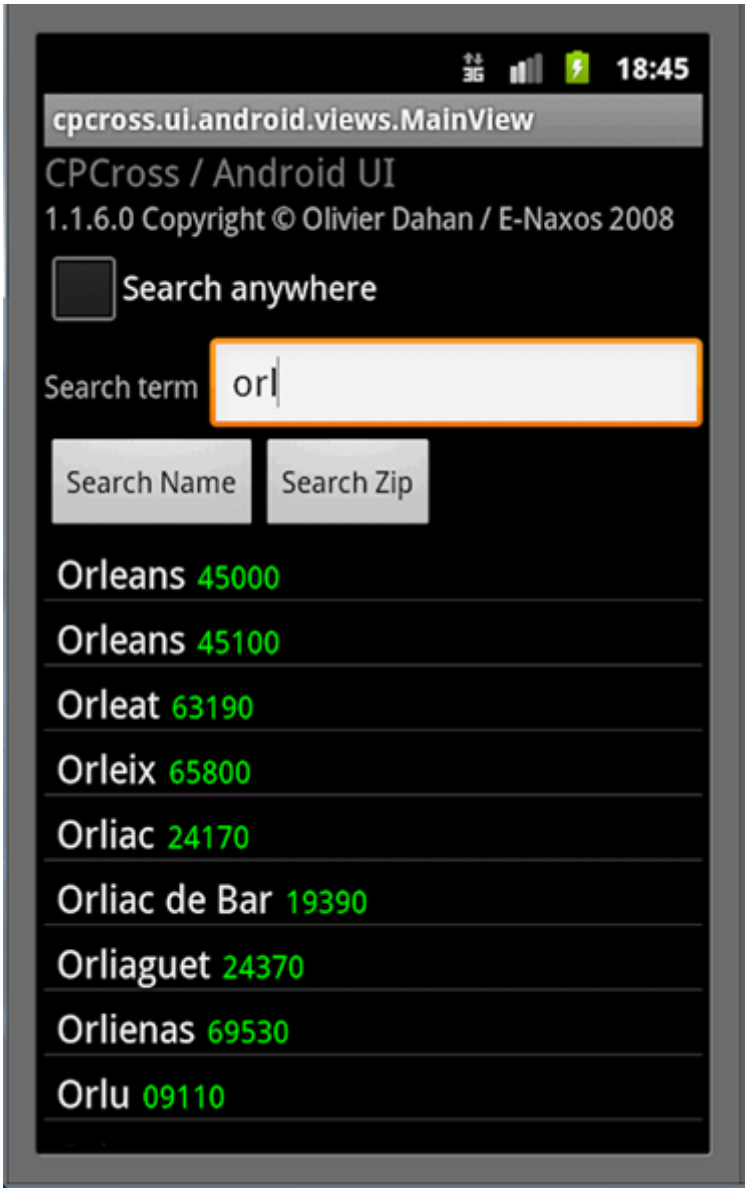
Mais rappelez-vous... Cette propriété est un booléen, et autant sous Xaml que sous Android "Visibility" est d'un autre type (différent dans les deux environnements en plus). Comment régler ce problème très courant ?

Si vous avez suivi, vous vous rappelez que le projet noyau déclare un convertisseur qui utilise les facilités de MVVMCross. Et bien c'est ici qu'il est utilisé.

Dans la version Android, notre convertisseur, écrit une seule fois, saura fournir la valeur attendue par visibility, et sous Windows Phone, que nous verrons plus loin, le même binding sur la même propriété du ViewModel utilisera le même convertisseur du noyau qui cette fois-ci retournera l'énumération attendue par Xaml ...

C'est peu de chose au final, très peu même. Mais quelle chose ! Grâce à de petites astuces de ce genre nous avons écrit un seul code pour toutes les plateformes. Et cela n'a pas de prix. Enfin, si, un cout énorme, l'écriture à partir de zéro plusieurs fois de la même application un coup en C#, un coup en Objective-C, l'autre en Java Android, etc... Si vous tentez un léger calcul le gain réalisé par l'approche que je vous propose est démoniaque ! (les bonnes âmes peuvent me verser 10% du montant économisé pour mes bonnes œuvres, je saurai les utiliser à bon escient).

Est-ce que ça marche ?



Oui ça marche !

Et bien même.

Ici l'ensemble des résultats est bien entendu accessible, on peut scroller avec le doigt si nécessaire.

La mise en page n'est pas géniale, ce n'est qu'un exemple ne l'oublions pas (parmi plein d'autres).

Les plus curieux se demandent peut-être comment la liste est mise en page ... Comme on sait le faire en Xaml en créant un DataTemplate, ici on créé une Vue spéciale qui ne fait que la mise en page d'un Item, en gros cela marche de la même façon que Xaml donc... Le StackPanel n'existe pas sous ce nom, mais on le trouve sous l'appellation LinearLayout sous Android. Il peut être en orientation verticale ou horizontale, la même chose... Dans ce LinearLayout j'ai placé deux TextBlock, heu non, pardon, deux TextView (très difficile à se rappeler aussi ...) en utilisant la même technique de Binding de MVVMCross.

Dire que cela a été compliqué à faire serait mentir. Il y a bien deux ou trois choses sur lesquelles on bute, mais grâce à Internet on trouve vite les réponses. Et surtout grâce à MonoDroid et MVVMCross qui à eux deux rendent le développement Android presque identique à du Xaml...

Une troisième Cible ?

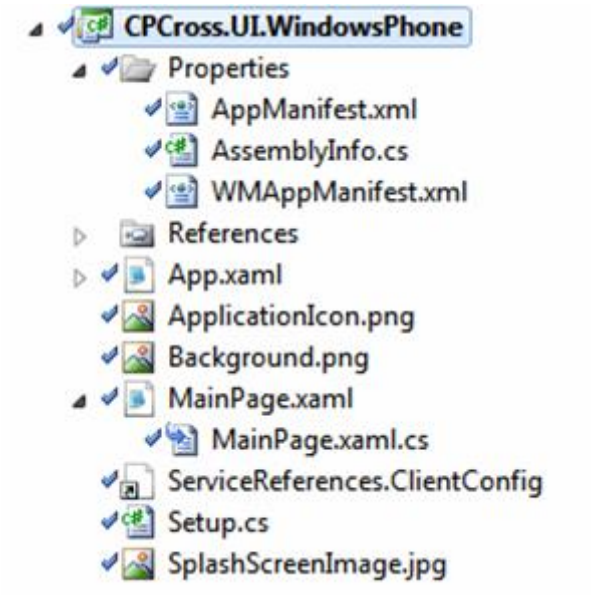
Allez, pour la route, vous en reprendrez bien une dernière ?

C'est un développement réellement cross-platform, avec un seul code écrit une fois : le serveur métier totalement universel, les ViewModels totalement portables avec MVVMCross.

Rajouter une nouvelle cible me prend juste du temps, mais j'en ai, je suis en vacances ! (même si je vois mal la différence avec d'habitude).

Je vais ainsi ajouter une version Windows Phone. J'aime bien Windows Phone.

Ajouter un projet sous VS vous savez le faire. Choisir un projet Windows Phone dans la liste (une fois le SDK installé) n'est pas très compliqué non plus.

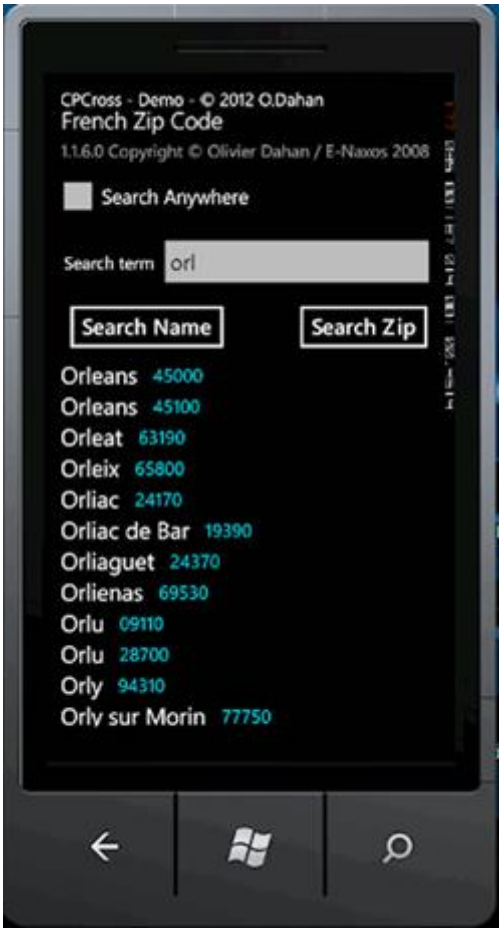


La structure est celle d'un projet Windows Phone, avec son SplashScreen, son App.xaml, sa MainPage.xaml etc.

Je passe la mise en page Xaml, le Binding qui cette fois-ci est totalement naturel, la création d'un vrai DataTemplate pour la liste des résultats, etc.

Est-ce que ça marche ?

Aussi !



C'est pas magique tout ça ?

Conclusion

Voilà... Je vous ai présenté "ma stratégie" de développement cross-platform / cross form-factor.

C'est simple, ça n'utilise qu'un seul langage de programmation, qu'un seul framework, .NET, qu'un seul EDI (VS ou MonoDevelop sa copie), une seul framework MVVM qui en plus efface les petites différences entre les plateformes, et surtout des outils géniaux comme MonoTouch ou MonoDroid.

La stratégie repose aussi sur la pérennisation du code métier dans un ou plusieurs services Web, rendant l'approche encore plus rentable et plus ouverte à toute adaptation. L'utilisation récente des PCL de Visual Studio permet le même type pérénisation du code sans les lourdeurs d'un serveur, toutefois ce dernier reste souvent indispensable notamment pour les applications LOB.

L'exemple présenté n'est qu'une simple démonstration sans fioriture, mais c'est un vrai projet multi-plateforme, multi OS, multi form factor avec un vrai service distant,

des écrans qui font un vrai travail. Il n'y aucune différence avec une application réelle autre que le contenu et la complexité de celui-ci.

L'écriture des 3 parties du billet m'a prise plus de temps que l'écriture de la solution fonctionnelle... 4 jours contre 3. Et si mon expérience de C# et Xaml est reconnue, je ne connais pas encore la même "intimité" avec Android. Et pourtant je l'ai fait (avec plus de soucis en réalité sous Windows Phone que sous Android ce qui est un comble).

Ma stratégie est simple et utilise le moins de "bidules" ou de "trucs" annexes. C'est vraiment le moyen le plus simple de faire du cross-plateforme aujourd'hui pour qui connait C# et Xaml, j'en suis convaincu.

Ca marche, aucun code n'est dupliqué, seule les parties fortement variables sont redéveloppées (les UI) et c'est normal mais peu contraignant puisqu'on travaille en MVVM et que tout est dans les ViewModels... Poser quelques Checkbox ou TextView ou TextBlock n'est franchement pas bien compliqué.

Cette approche minimise les couts, à la fois de développement et de maintenance, mais aussi, les plus chers, ceux du personnel compétent à former pour gérer autant de cibles. Ici cela se résume au strict minimum.

Bien maitriser C#, .NET et MVVM est la base non négociable. Le reste ce n'est que du Lego, normalement un vrai plaisir pour tout ingénieur qui à l'âme d'un ingénieur... C'est à dire pour celui qui aime autant les théories que les rendre tangibles, palpables en réalisant quelque chose qui marche et qui sera utile.

Espérant que ce billet en trois volet aura lui aussi été utile aux lecteurs de Dot.Blog,

A la prochaine (j'en plein de trucs en réserve!) donc...

Stay Tuned !

Le code source de la solution (sans MVVMCross), nécessite VS 2010 + derniers patches, le SDK Windows Phone, MonoDroid avec les SDK Android :

CPCross.zip

Nota: Le web service des codes postaux sert de test, vous pouvez y accéder pour tester la solution mais soyez sympa, n'en abusez pas, si le serveur venait à être gêné par le trafic généré je serai obligé de couper le web service. Merci d'avance !

Stratégie de développement Cross-Platform – Bonus

Suite à la série de billets "Stratégie de développement cross-platform", en bonus, voici le PDF de 54 pages qui sera peut être plus pratique à consulter que les pages web de Dot.Blog :

Stratégie de développement Cross Plateforme.pdf

Cross-plateforme : Xamarin Store, des composants à connaître

Continuant ma série sur le développement cross-plateforme je vous propose aujourd'hui de découvrir le Xamarin Store et de nombreux composants incroyables et souvent gratuits qui offrent des services indispensables aux applications modernes.

Le Xamarin Store

Le Xarmarin Store est un site de publication de composants et librairies dédiés au développement cross-plateforme. Beaucoup sont gratuits, certains sont payants, une grande partie se limite à iOS et Android, d'autres fonctionnent aussi sur Windows Phone, certains sont uniquement dédiés à un OS (souvent iOS car Xamarin a débuté avec MonoTouch pour l'iPhone avant de publier MonoDroid, l'histoire avec l'OS de Apple est donc un peu plus longue).
Si le store ne contient pas encore des milliers de composants (la diffusion de Xamarin 2.0 n'est pas celle d'un EDI comme Visual Studio, il faut en rester conscient), on y trouve beaucoup de choses intéressantes qui peuvent transformer une application banale en app moderne, utile et attractive.

Trois OS, un seul langage

La magie de Xamarin Studio est d'utiliser un seul langage déjà connu de tous (ou presque) dans le monde Microsoft : C# qui est standardisé ECMA. En partant du projet Mono, la version open source de C# et .NET, Xamarin a créé un environnement de développement capable de générer des applications natives pour iOS et Android depuis un PC. Le designer visuel, d'abord absent, puis seulement dans l'EDI Mono est aujourd'hui un module mâture fourni en plugin de Visual Studio et en stand-alone intégré à Xamarin Studio, l'EDI cross-plateforme.

On peut donc choisir de travailler sur un Mac ou un PC, d'utiliser Xamarin Studio ou bien Visual Studio pour créer soit des applications natives uniques soit bien entendu des applications cross-plateformes. Une seule solution Visual Studio, reconnue aussi par Xamarin Studio (on peut passer d'un EDI à l'autre sans problème, les fichiers de solutions et de projets sont les mêmes), et on peut construire une application attaquant les plateformes qu'un développeur doit aujourd'hui adresser. iOS, Android, Windows Phone, et même WPF ou Silverlight, WinRT et ASP.NET peuvent être mélangés sous Visual Studio (Xamarin Studio gère de l'ASP.NET mais pas WPF ou SL ni WinRT).

Le vrai cross-plateforme

Le “vrai” cross-plateforme n’existe pas ! Il existe deux réalités différentes sous ce terme : La même application portée à l’identique avec le même code sous plusieurs OS ou bien, ce qui est plus réaliste aujourd’hui, une application dont les modules sont répartis entre plusieurs OS et form factor. On pourrait prendre l’exemple d’une gestion commerciale dont la partie comptable tourne sur PC de bureau, la partie gestion des commandes sur tablettes le tout complété par une application Smartphone pour les clients (prise de commande, suivi des nouveautés, forums…).

Xamarin Studio permet de travailler dans ces deux cadres distincts. On peut choisir de créer une application uniquement iOS ou Android comme on peut intégrer l’un ou l’autre de ces OS (ou les deux) dans une solution plus vaste.

Pour certaines applications c’est en fait un mélange de ces deux façons de voir le cross-plateforme qui permet d’atteindre l’objectif. On peut avoir une application en WPF tournant donc sous Windows partageant du code et des donnés avec une application en ligne en ASP.NET le tout complété d’une ou plusieurs apps mobiles pour smartphones ou tablettes, chacune pouvant être portée à l’identique pour plusieurs OS. L’application elle-même n’est plus un exécutable ou une site Web, ni même une app mobile, c’est alors l’ensemble de toutes ces applications qui forme “l’Application” avec un grand “A”…

Tout cela se combine avec la stratégie de développement cross-plateforme présentée dans les chapitres précédents et largement documentée par la pratique dans le reste du présent livre.

Soutenir Windows Phone et WinRT

En réalité les applications du type que je viens de décrire bien loin de nous éloigner de Windows Phone ou même de WinRT permettent de soutenir ces environnements même si pour l’instant leur adoption reste limitée.

En effet, créer une application Windows Phone seule est un peu risqué financièrement, le retour sera par force faible vu les parts de marché actuelles de l’OS. En revanche quitte à créer la même application pour Android ou iOS, grâce à Visual Studio et Xamarin Studio il sera possible de supporter _aussi_ Windows Phone pour un coût minimum.

La démarche cross-plateforme que je vous propose depuis un bon moment déjà permet ainsi d’éviter d’éliminer Windows Phone faute de pénétration suffisante du

marché et de lui réserver une place dans les plans de développement car cela ne coûte pas très cher de le prévoir dès le départ dès lors qu'on utilise les bons outils.

WinRT est très proche de Silverlight et de WPF, technologies C#/Xaml auxquelles Dot.Blog a réservé près de 90% de son espace sur les plus de 650 billets publiés à ce jour. Si je n'en parle pas beaucoup plus c'est que toutes les techniques utilisables sous WinRT sont pour l'essentiel déjà décrites en détail par ces centaines de billets. Quant à Windows Phone, en dehors de quelques spécificités qui lui sont propres, comme WinRT il utilise C# et Xaml présentés en long et en large par Dot.Blog.

Le cross-plateforme est l'avenir du développement. C'est pour cela que je vous en parle de plus en plus. Mais qui dit cross-plateforme dit connaissance des plateformes, et c'est tout naturellement que je vous parle notamment d'Android depuis près de deux ans car c'est l'un des OS principaux à connaître à côté de ceux proposés par Microsoft.

Cela nous ramène au Xamarin Store qui propose des composants et des librairies dont un nombre assez grand est totalement cross-plateforme et fonctionne sous iOS, Android et Windows Phone à la fois !

Ma sélection

Même s'il n'y a pas encore des milliers de librairies dans le Xamarin Store il y en a de trop pour toutes les présenter.

Ainsi, je vous propose ici de découvrir les composants (ou librairies) qui m'apparaissent les plus intéressants et qui sont au moins disponibles pour iOS et Android à la fois (les OS couverts directement par Xamarin Studio).

Toute sélection est réductrice et je vous invite à consulter les autres codes proposés par le Xamarin Store.

Azure Mobile Services

On ne présente plus Azure qui permet de stocker des données dans les nuages, d'authentifier des utilisateurs et d'envoyer des notifications de send ou de push. Il existe bien entendu du code Microsoft pour se servir de ces services depuis Windows Phone ou WinRT. Sur le Xamarin Store on trouvera aussi "Azure Mobiles Services", écrit par Microsoft, pour Android et iOS !

Avec cette librairie tout devient facile sous tous les OS mobiles. Par exemple le code ci-dessous montre comment insérer une instance de la classe Item dans un stockage Azure, en une ligne de C# :

```
public class Item
{
    public int Id { get; set; }
    public string Text { get; set; }
}

Item item = new Item { Text = "Awesome item" };
      App.MobileService.GetTable<Item> ().InsertAsync (item)
        .ContinueWith (t => {
        /* success or failure */
} );
```

Bien entendu, cette librairie est gratuite.

Signature Pad

Il s'agit d'un des rares composants payants que j'ai sélectionné. Une raison bien simple : il est tout simplement unique… pas d'alternatives gratuites. De plus il n'est pas trop cher pour le service rendu (150 dollars). Ce composant permet de gérer les signatures digitales avec le doigt directement sur l'écran d'un smartphone ou d'une tablette. Le code fonctionne sous iOS et Android.

Très simple à utiliser, Signature Pad peut faire d'une application quelconque un véritable outil professionnel (faire signer un client pour une livraison, un accusé réception d'un document ou d'un objet remis en mains propres, etc).
L'un des critères de sélection que j'ai appliqué à tous les composants présentés ici : la simplicité d'utilisation. Le code qui suit vous montrera ce que simple veut dire dans ce cas :

```
 var signature = new SignaturePadView (View.Frame);
View.AddSubview (signature);

//Get the signature
var image = signature.GetImage ();
ViewController (shareController, true, null);
```

L'exemple créé une nouvelle vue puis demande tout simplement au composant de retourner l'image de la signature… (l'exemple ci-dessus est pour iOS mais reste quasi identique à celui pour Android).

SQLite.NET

Pour ceux qui ne le savent pas SQLite est le moteur de base de données intégré à Android. Toute application Android peut ainsi et très facilement créer des bases, des tables, et jongler avec les données sans se prendre la tête. Il est dommage que Microsoft n'ait pas fait un choix aussi simple pour Windows et qu'on s'emmêle un peu les pinceaux dans les différentes versions de SQL Server pour ASP.NET, pour Mobile, pour PC, etc. SQLite étant un SGBD gratuit et open on peut toutefois l'installer aussi sous Windows Phone et sur iPhone ou iPad.

Accéder à une base SQLite est possible directement depuis les API fournies par Android, mais pas aussi directement depuis Windows Phone ou iOS. D'où l'intérêt d'une surcouche portable comme SQLite.NET qui supporte aussi bien Android que iOS et Windows Phone avec le _même_ code C#. Standardiser les accès aux donnés est un énorme avantage pour tout développement cross-plateforme.
L'utilisation de cette librairie est d'une grande simplicité :

```
using SQLite;
// ...

public class Note
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public string Message { get; set; }
}

// Create our connection
string folder = Environment.GetFolderPath (Environment.SpecialFolder.Personal);
var db = new SQLiteConnection (System.IO.Path.Combine (folder, "notes.db"));
db.CreateTable<Note>();

// Insert note into the database
var note = new Note { Message = "Test Note" };
db.Insert (note);

// Show the automatically set ID and message.
Console.WriteLine ("{0}: {1}", note.Id, note.Message);
```

Créer une connexion, insérer directement une instance de classe dans une table, tout cela s'effectue en quelques lignes. On peut bien entendu travailler aussi en SQL ou utiliser la façon de faire de ADO.NET, c'est comme on le veut, selon le besoin.

SQLCipher

Travailler avec une base de données SQLite est une bonne chose, les données sont le nerf de la guerre en informatique. Mais la confidentialité peut être un pré-requis pour certaines applications notamment en entreprise. Dès lors il faut pouvoir chiffrer les données.

Cela n'est pas très difficile en soi, mais faire des appels aux API de chiffrement à chaque lecture ou insertion de données peut devenir infernal et conduire à un code lourd. Mieux vaudrait que la couche d'accès aux données le fasse toute seule…

C'est justement ce que propose SQLCipher. Cette librairie est une copie de SQLite.NET présentée plus haut, les API sont les mêmes, à la nuance près qu'un mot de passe est réclamé comme paramètre supplémentaire dans les appels de méthodes.
On peut ainsi transformer une application conçue pour SQLite.NET en une application sécurisée en changeant juste le "using" et en ajoutant le paramètre manquant.

Les données sont chiffrées avec une algorihme AES-256, ce qui assure une sécurité très élevée.

Cette librairie n'est pas gratuite mais d'une part son prix est raisonnable (150 dollars) et d'autre part il se justifie pleinement dans le cadre dans une application professionnelle.

Ici aussi le critère de simplicité est parfaitement rempli : les API sont les mêmes que SQLite.NET et interchangeables (au paramètre mot de passe près).

```
  public class Model
{
  [PrimaryKey,AutoIncrement]
  public int Id { get; set; }
  public string Content { get; set; }
}

using(var conn = new SQLiteConnection (FilePath, Password))
{
  var model = conn.Table<Model>().Where(x=> x.Id = 0)
}
```

Le code ci-dessus montre l'accès à une instance de la classe Model, en une ligne. Cela est conforme à SQLite.NET mais on remarque le paramètre supplémentaire "Password".

```
public void SqlCipherDemo ()
{
using (var connection = new
  Mono.Data.Sqlcipher.SqliteConnection(connectionString)) {
  connection.SetPassword ("secretPassword");
  connection.Open ();

  using (var command = connection.CreateCommand()) {
    var query = "select * from t1";
    command.CommandText = query;
    var reader = command.ExecuteReader ();
    while (reader.Read()) {
      //Read Values
    }
  }
  connection.Close ();
 }
}
```

Ici on utilise SQL et les méthodes d'accès typiques de AD.NET, là encore tout est similaire à SQLite.NET sauf l'appel à SetPassword sur la connexion.

Le chiffrement des données dans toutes les opérations CRUD est ainsi totalement transparent, sans risque d'erreur ce qui renforce la sécurité de l'application.

Radial Progress

Faire des opérations longues, surtout sur un smartphone ou une tablette à la puissance réduite peut s'avérer plus fréquent qu'on le pense, de même qu'accéder via la 3G à des données distantes ce qui est rarement très rapide…

Avec Radial Progress on peut donner un look plus attrayant à cette attente, et c'est toute l'UX qui s'en trouve améliorée car une UI bien faite facilite aussi la prise en main et augmente la satisfaction de l'utilisateur.
Gratuit, disponible pour iOS et Android, ce composant est très simple mais bien fait.

JSon.NET

JSON est à la mode, personnellement je n'ai jamais bien vu son avantage sur XML car sa réputation d'être moins "verbeux" a souvent été contredite par mes propres expériences. Mais la question n'est pas là, c'est à la mode… Il existe donc des tas de façon d'accéder à des données JSON mais JSON.NET est "the" librairie la plus utilisée dans le monde .NET d'autant plus qu'elle est supportée dans toutes les versions possibles : WPF, SL, WinRT, Windows Phone, et ici iOS et Android aussi.
Cette librairie est tellement bien faite que l'utiliser dépasse l'effet de mode. Il y a des possibilités réellement immenses, une gestion des cycles intelligente, des passages facilités entre JSON et XML dans les deux sens, etc.

Gratuite, puissante, très optimisée, cette couche dédiée à JSON est d'une grande utilité et tend à devenir même sur PC la norme de fait pour sérialiser les données sous .NET sans se prendre la tête ni être obligé de déclarer des DataContracts.

ZXing.Net.Mobile

Au début on pense que c'est un truc chinois "ZXing", en fait non… Quand on connait ces facétieux américains on comprend qu'il s'agit encore d'une de leur farce de langage comme ils en ont le secret, il faut prononcer "Zebra Crossing". Pourquoi des zèbres qui traversent ? Tout simplement parce que c'est le nom qu'on donne aux

passages piétons aux USA, en raison de la ressemblance avec les rayures noires et blanches des Zèbres…

Et pourquoi des passages piétons ? … L'imagination n'a pas de limite, tout simplement parce qu'un code barre ressemble lui-aussi à ce même motif de lignes noires et blanches.

Aller du zèbre africain aux codes barres en passant par la jungle urbaine, le parlé (et l'écrit) américain est plein de vie, d'images que je trouve très créatives. Car il y a aussi un message plus "subliminal" dans le nom choisi : le fameux "crossing", traverser. Ici la librairie traverse aussi les OS puisqu'elle est cross-plateforme. Joli non ?

Bref vous l'aurez compris ZXing est une librairie portable permettant de reconnaître les codes barres mais aussi les QR codes. Elle fonctionne sous iOS, Android et Windows Phone, et comble du bonheur elle est gratuite et facile à utiliser !

D'iOS à gauche à Windows Phone à droite en passant par Android au milieu…

Et voici le code, le même pour toutes les plateformes, permettant d'obtenir la valeur d'un code (barre ou QR) :

```
  public void Scan ()
{
  var scanner = new ZXing.Mobile.MobileBarcodeScanner ();
  scanner.Scan ().ContinueWith (t => {
    if (t.Result != null)
      Console.WriteLine ("Scanned Barcode: " + t.Result.Text);
  } );

}
```

J'ai testé et c'est vraiment très bien fait. On peut obtenir de nombreuses informations sur le code, comme sa nature (ISBN d'un livre, code d'un produit, etc). Il est ensuite assez facile d'aller faire une requête à Amazon ou Google pour obtenir le détail du produit. De même il est possible d'utiliser la vue par défaut ou bien de fournir une vue personnalisée pour le scan. La reconnaissance est très bonne (en tout cas sur mon SIII dont l'appareil photo et la vitesse ne sont pas représentatifs du smartphone moyen malgré tout mais sans être une fusée non plus). Il existe plusieurs applications sur le Play Store qui utilisent cette librairie, celles que j'ai testées étaient très réactives même sur un Galaxy Ace qui commence à dater (mais représente la norme en terme de puissance du parc Android actuel, que cela soit en puissance pure qu'au niveau du format écran).

Les QR codes peuvent contenir des adresses web, du texte, on peut bien entendu traiter tout cela ce qui offre des possibilités infinies.

Xamarin Social

Xamarin a produit aussi quelques librairies bien utiles et gratuites. Xamarin.Social en fait partie. Comme le nom le laisse entendre il s'agit ici de pouvoir accéder aux réseaux sociaux pour écrire directement sur un mur Facebook ou tweeter comme un pinson automatiquement…

Cette librairie est encore un peu "jeune" et ne supporte encore pas G+ ou LinkedIn, mais elle simplifie grandement les choses pour Facebook, Twitter, Pinterest, Flickr ou App.Net.

Xamarin.Social est gratuite et fonctionne sous iOS et Android.

Xamarin.Mobile

Xamarin est conçu pour coder sur des appareils mobiles, alors pourquoi cette librairie ? Tout simplement parce que justement Xamarin est cross-plateforme par essence et qu'il est bien agréable de pouvoir disposer gratuitement d'une couche unifiant les appels aux principaux services de tous les OS plutôt que de farcir le code final de nombreux appels natifs qui ne se ressemblent pas !

Xamarin.Mobile permet ainsi d'unifier par un même code l'appel à la géolocalisation, à la liste des contacts et au média picker. C'est autant de code spécifique à chaque OS qu'il n'y a plus besoin de coder…
La librairie fonctionne sous iOS, Android et Windows Phone.

Xamarin.Auth

Encore une librairie gratuite simplifiant le travail. Ici Xamarin.Auth offre un moyen simple d'authentifier un utilisateur auprès de tout service suivant la norme OAuth 1.0 ou 2.0. Elle fonctionne sous iOS et Android.
Voici un exemple d'authentification utilisant les services Facebook :

```
using Xamarin.Auth;

var auth = new OAuth2Authenticator (
    clientId: "App ID from https://developers.facebook.com/apps",
    scope: "",
    authorizeUrl: new Uri ("https://m.facebook.com/dialog/oauth/"),
    redirectUrl: new Uri ("http://www.facebook.com/connect/login_success.html"));

auth.Completed += (sender, eventArgs) => {
    DismissViewController (true, null);
    if (eventArgs.IsAuthenticated) {
        // Use eventArgs.Account to do wonderful things
    }
};

PresentViewController (auth.GetUI (), true, null);
```

C'est simple, et pratique.

Bar Chart

Il fut un temps où on disait "dites-le avec des fleurs", aujourd'hui on dirait "dites-le avec un graphique"… Une application la plus simple soit-elle se doit d'offrir des graphiques pour faire sérieux et en même être attractive. Bien entendu si cela sert aussi le fonctionnel alors c'est merveilleux !

Bar Chart est un composant gratuit tournant sous iOS et Android qui offre des graphes à barre propres et riches à la fois.

Scrolling, libellés, gestion automatique de l'orientation, Bar Chart offre gratuitement tout le nécessaire. Le code d'appel est lui aussi d'une grande simplicité.

Conclusion

Le Xamarin Store contient bien d'autres choses que je vous incite à découvrir par vous-même. J'espère avoir ici excité votre curiosité en même temps que vous avoir fait découvrir à quel point le monde Xamarin est riche en solutions qui vous simplifieront vos développements cross-plateformes.

La cohabitation des OS est une chose entendue pour longtemps, aucun n'écrasera à 100% le marché. Le BYOD s'impose comme une réalité aux entreprises. En utilisant les bons outils et les bons composants un développeur peut réaliser des prouesses …

Bon dev, et Stay Tuned !

WP: Microsoft.Phone.Info, un namespace à découvrir…

Je ne parle pas trop de Silverlight sous WP car beaucoup de choses sont identiques. Mais il est toujours d'actualité, et peut-être bien plus que lors de sa sortie, de s'intéresser à cette version particulière de Silverlight. Par exemple identifier un utilisateur ou un téléphone de façon fiable et unique est un besoin assez courant et bien spécifique à cet environnement. Analyser la mémoire utilisée est aussi quelque chose d'essentiel sur des petites machines. Ces informations et bien d'autres se trouvent dans Microsoft.Phone.Info…

Le namespace Microsoft.Phone.Info

Ce namespace donne accès à des nombreuses informations et les curieux qui y fouineront trouveront là des données et des fonctions bien utiles.

La classe DeviceStatus

Un bon exemple de cette niche d'informations est donné par la classe DeviceStatus. A partir de cette dernière on peut accéder à la consommation mémoire actuelle de l'application, ou à des statistiques bien pratiques en debug comme le pic de consommation mémoire. Le nom du fabriquant, le modèle de téléphone, la mémoire totale installée, la présence ou non d'un clavier, sont autant d'autres éléments qui peuvent s'avérer essentiels. De même que PowerSource qui permet de savoir si le téléphone est sur batterie ou branché au secteur. Il existe même un évènement indiquant le changement de cette information.

DeviceExtendedProperties

Obtenir des informations étendues sur la device elle-même est tout aussi intéressant. Même si à première vue cette classe est assez vide, elle cache des informations de premier plan.

La classe expose GetValue et TryGetValue. Toute l'astuce se trouve dans le nom qu'on passe à ces méthodes pour obtenir la valeur...

Par exemple, on peut obtenir l'ID unique du téléphone de la façon suivante :

```
var id = (byte[])
DeviceExtendedProperties.GetValue("DeviceUniqueId");
var idStr  = BitConverter.ToString(id);
```

L'id est un tableau de 20 octets, transformé en chaine cela donne quelque chose dans ce style :

"EE-7A-95-56-BE-19-56-1B-60-1C-91-C9-55-FB-32-3A-E1-7B-A5-54"

A noter que l'utilisation de cette classe affiche une demande de confirmation d'accès aux informations du téléphone. L'utilisateur doit valider cette demande. S'il refuse, l'information n'est pas accessible.

Grace à d'autres identificateurs on accède à d'autres informations, mais la classe DeviceStatus sait en retourner la plupart sans le problème du dialogue à l'utilisateur. L'identificateur unique de la machine reste donc l'intérêt principal de DeviceExtendedProperties.

UserExtendedProperties

La logique est la même : deux méthodes pour soutirer des informations un peu cachées. Ici il s'agit de l'identité de l'utilisateur.

```
var anid = UserExtendedProperties.GetValue("ANID") as string;
```

L'intérêt est ici d'obtenir une chaine de caractères identifiant l'utilisateur de façon unique. Cela peut s'avérer pratique pour soumettre un score à un jeu et que ce score soit bien accroché à la même personne quelque soit le téléphone utilisé.

ID_CAP_IDENTITY_USER doit être ajouté au manifest pour espérer obtenir une réponse. La même contrainte existe pour la classe précédente. Le manifest doit ainsi contenir :

```
<Capabilities>
  ...
  <Capability Name="ID_CAP_IDENTITY_DEVICE"/>
  <Capability Name="ID_CAP_IDENTITY_USER"/>
  ...
</Capabilities>
```

La chaine retournée est un ensemble de valeurs sous la forme "clé=valeur" . Il s'agit d'informations anonymes sur le Live ID de l'utilisateur. La chaine peut s'utiliser directement, même s'il semble préférable d'en extraire la partie efficace, le fameux ID :

```
private static readonly int ANIDLength = 32;
private static readonly int ANIDOffset = 2;

public static string GetWindowsLiveAnonymousID()
{
    string result = string.Empty;
    object anid;
    if (UserExtendedProperties.TryGetValue("ANID", out anid))
    {
        if (anid != null && anid.ToString().Length >=
                                    (ANIDLength + ANIDOffset))
        {
            result = anid.ToString().Substring(ANIDOffset, ANIDLength);
        }
    }
```

```
    return result;
}
```

Conclusion

Certaines informations ne sont pas difficiles à obtenir mais elles sont bien cachées... Le namespace Microsoft.Phone.Info en contient quelques unes, et maintenant vous savez comment y accéder !

Cross-plateforme, stockage ou Web : Sérialisation JSON

La sérialisation des données est à la programmation Objet ce que le jerrycan est à l'essence : si vous n'avez pas le premier vous ne pouvez pas transporter ni conserver le second. Or sérialiser n'est pas si simple, surtout lorsqu'on souhaite un format lisible et cross-plateforme. JSON est alors une alternative à considérer.

Pourquoi sérialiser ?

Les raisons sont infinies, mais le but général reste le même : transporter ou stocker l'état d'un objet (ou d'une grappe d'objets). C'est donc un processus qui s'entend forcément en deux étapes : la sérialisation proprement-dite et la désérialisation sans laquelle la première aurait autant d'intérêt qu'une boîte de conserve sans ouvre-boîte...

On sérialise les objets pour les transmettre à un service, on désérialise pour consommer ses objets via des services, on peut aussi se servir de la sérialisation comme d'un moyen pratique pour stocker l'état d'une page sous Windows Phone ou Windows 8 par exemple. En effet, une sérialisation de type chaîne (XML ou JSON) à l'avantage de n'être que du texte, un simple texte. Plus de références, de memory leaks, ou autres problèmes de ce type. Les objets ayant été sérialisés peuvent être supprimés de la mémoire, ils pourront être recréés plus tard. Réhydratés comme un potage instantané... De plus une chaîne de caractères cela se traite, se transmet, se stocke très facilement et cela supporte aussi avec un taux de réussite souvent impressionnant la compression (Zip par exemple).

Il existe donc des milliers de raisons de vouloir sérialiser des objets (ou de consommer des objets lyophilisés préalablement par une sérialisation).

Choisir le moteur de sérialisation

Une des forces du Framework .NET a été, dès le début, de proposer des moteurs de sérialisation efficace, à une époque où d'autres langages et environnement peinaient à le faire. La sérialisation binaire a petit à petit cédé la place à la sérialisation XML, format devenu incontournable avec l'avènement du Web et des Web services.

.NET s'acquitte toujours de cette tâche de façon efficace d'ailleurs.

Mais alors pourquoi aller chercher plus loin ?

D'une part parce que les modes changes sans véritable raison technique, ainsi JSON est un format plus "hype" que XML ces temps-ci. Vous allez me dire, "je m'en fiche". Oui, bien sur. Moi aussi. Ma la question n'est pas là. Il se trouve qu'il existe donc de plus en plus de services qui fournissent leurs données en JSON et qu'il faut bien pouvoir les interpréter pour les consommer… Realpolitik.

D'autres facteurs méritent malgré tout d'être pris en compte : notamment les capacités du moteur de sérialisation lui-même. Car tous ne sont pas égaux ! Certains, comme ceux du Framework .NET refusent les références circulaires par exemple…

Je sais que dit comme ça, les références circulaires on peut penser ne jamais en faire, mais il existe des tas de situations très simples et légitimes qui pourtant peuvent en voir surgir. Et il n'est pas "normal" d'avoir à "bricoler" les propriétés d'un objet pour que la plateforme technique puisse fonctionner. A fortiori lorsqu'on vise du cross-plateforme où la solution peut avoir à être implémentée chaque fois de façon différente.

Bien entendu les différences ne s'arrêtent pas là, certains moteurs savent gérer la sérialisation et la désérialisation des types anonymes ou des types dynamiques, d'autres non. Et les nuances de ce genre ne manquent pas !

Mieux vaut donc choisir son moteur de sérialisation correctement.

Il existe ainsi de bonnes raisons d'avoir, à un moment ou un autre, besoin d'un autre moteur de sérialisation que celui offert par .NET.

JSON.NET

C'est le moteur de sérialisation JSON peut-être le plus utilisé sous .NET, la librairie est bien faite, performante (plus que .NET en JSON en tout cas) et totalement portable entre les différentes versions de .NET. Le projet est open source sur CodePlex

(Json.net), il peut être téléchargé depuis ce dernier ou même installé dans un projet via NuGet.
Mais toutes ces raisons ne sont pas suffisante pour embarquer une librairie de plus dans une application. Il faut bien entendu que les services rendus permettent des choses dont l'application a besoin et que les moteurs offerts par .NET ne puissent pas faire.

Voici un petit récapitulatif des différences entre les différents moteurs .NET (info données par JSON.NET) :

	Json.NET	**DataContractJsonSerializer**	**JavaScriptSerializer**
Supporte JSON	Oui	Oui	Oui
Supporte BSON	Oui	Non	Non
Supporte les schémas JSON	Oui	Non	Non
Supporte .NET 2.0	Oui	Non	Non
Supporte .NET 3.5	Oui	Oui	Oui
Supporte .NET 4.0	Oui	Oui	Oui
Supporte .NET 4.5	Oui	Oui	Oui
Supporte Silverlight	Oui	Oui	Non
Supporte Windows Phone	Oui	Oui	Non
Supporte Windows 8	Oui	Oui	Non
Supporte Portable Class Library	Oui	Oui	Non
Open Source	Oui	Non	Non
Licence MIT	Oui	Non	Non
LINQ to JSON	Oui	Non	Non
Thread Safe	Oui	Oui	Oui
Syntaxe JSON XPath-like	Oui	Non	Non
Support JSON Indenté	Oui	Non	Non

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
Sérialisation efficace des dictionnaires	Oui	Non	Oui
Sérialization des dictionnaires "nonsensical"	Non	Oui	Non
Déserialise les propriétés IList, IEnumerable, ICollection, IDictionary	Oui	Non	Non
Serialise les références circulaires	Oui	Non	Non
Supporte sérialisation par référence	Oui	Non	Non
Déserialise propriétés et collection polymorphiques	Oui	Oui	Oui
Sérialise et déserialise les arrays multidimentionelles	Oui	Non	Non
Supporte inclusion des noms de type	Oui	Oui	Oui
Personalisation globale du process de sérialisation	Oui	Oui	Non
Supporte l'exclusion des null en sérialisation	Oui	Non	Non
Supporte SerializationBinder	Oui	Non	Non
Sérialisation conditionnelle	Oui	Non	Non
Numéro de ligne dans les erreurs	Oui	Oui	Non

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
Conversion XML à JSON et JSON à XML	Oui	Non	Non
Validation de schéma JSON	Oui	Non	Non
Génération de schéma JSON pour les types .NET	Oui	Non	Non
Nom de propriétés en Camel case	Oui	Non	Non
Supporte les constructeurs hors celui par défaut	Oui	Non	Non
Gestion des erreurs de sérialisations	Oui	Non	Non
Supporte la mise à jour d'un objet existant	Oui	Non	Non
Sérialisation efficace des arrays en Base64	Oui	Non	Non
Gère les NaN, Infinity, -Infinity et undefined	Oui	Non	Non
Gère les constructeurs JavaScript	Oui	Non	Non
Sérialise les objets dynamiques .NET 4.0	Oui	Non	Non
Sérializes les objets ISerializable	Oui	Non	Non
Supporte la sérialisation des enum par leur valeur texte	Oui	Non	Non
Supporte la limite de récursion JSON	Oui	Oui	Oui

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
Personnalisation des noms de propriétés par Attribut	Oui	Oui	Non
Personnalisation de l'ordre des propriétés par Attribut	Oui	Oui	Non
Personnalisation des propriétés "required" par Attribut	Oui	Oui	Non
Supporte les dates ISO8601	Oui	Non	Non
Supporte le constructeur de data JavaScript	Oui	Non	Non
Supporte les dates MS AJAX	Oui	Oui	Oui
Supporte les noms sans quote	Oui	Non	Non
Supporte JSON en mode "raw"	Oui	Non	Non
Supporte les commentaires (lecture/écriture)	Oui	Non	Non
Sérialise les type anonymes	Oui	Non	Oui
Déserialise les types anonymes	Oui	Non	Non
Sérialisation en mode Opt-in	Oui	Oui	Non
Sérialisation en mode Opt-out	Oui	Non	Oui
Sérialisation en mode champ	Oui	Oui	Non

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
Lecture/écriture efficace des streams JSON	Oui	Oui	Non
Contenu JSON avec simple ou double quote	Oui	Non	Non
Surcharge de la sérialisation d'un type	Oui	Non	Oui
Supporte OnDeserialized, OnSerializing, OnSerialized et OnDeserializing	Oui	Oui	Non
Supporte la sérialisation des champs privés	Oui	Oui	Non
Supporte l'attribut DataMember	Oui	Oui	Non
Supporte l'attribut MetdataType	Oui	Non	Non
Supporte l'attribut DefaultValue	Oui	Non	Non
Sérialise les DataSets et DataTables	Oui	Non	Non
Sérialise Entity Framework	Oui	Non	Non
Sérialise nHibernate	Oui	Non	Non
Désérialisation case-insensitive sur noms des propriétés	Oui	Non	Non
Trace	Oui	Oui	Non

Utiliser JSON.NET

Cette librairie est bien entendu documentée et son adoption assez large fait qu'on trouve facilement des exemples sur Internet, je ne vais pas copier ici toutes ces informations auxquelles je renvoie le lecteur intéressé.

Juste pour l'exemple, un objet peut se sérialisé aussi simplement que cela :

```
var JSON = JsonConvert.SerializeObject( monObjet );
```

(le résultat JSON est une string)

La désérialisation n'est pas plus compliquée.

Ensuite on entre dans les méandres des attributs de personnalisation (changer le nom d'une propriété, imposer un ordre particulier, …) voire la surcharge complète du processus de sérialisation, tout cela peut emmener loin dans les arcanes de JSON.NET car la librairie supporte un haut degré de personnalisation justement.

Conclusion

La sérialisation tout le monde connaît et s'en sert soit épisodiquement, soit fréquemment, mais souvent sans trop se poser de questions.

En vous proposant de regarder de plus près JSON.NET c'est une réflexion plus vaste que j'espère susciter : une interrogation sur les besoins réels de vos applications en ce domaine et la prise de conscience que le moteur de sérialisation ne se choisit pas au hasard en prenant le premier qui est disponible…

```
{
  "MotDeLaFin": "Stay Tuned!"
}
```

Bibliothèque de code portable, Async et le reste…

Les bibliothèques de code portable (Portable Class Libraries) introduites dans Visual Studio 2012 sont une avancée très importante dans notre monde qui hésite entre plusieurs plateformes, même au sein de la gamme Microsoft. Toutefois cibler plusieurs OS peut demander de faire des coupes sombres, comment y remédier ?

Code portable et Async

Les Portable Librairies permettent d'écrire un code unique qui peut ensuite être utilisé dans des projets ciblant des versions d'OS différents. Cela fonctionne au sein de la gamme Microsoft (par exemple écrire un code commun pour des projets WinRT et

Windows Phone), mais cela fonctionne même au-delà du monde Microsoft : c'est le principe utilisé par MvvmCross avec Xamarin pour développer un seul code tournant sous Android et Windows Phone par exemple.

C# 5 propose un nouveau mécanisme de gestion de l'asynchronisme via les modificateurs await/async. Je reviendrais plus en détail sur ce mécanisme car il est loin d'être aussi simple qu'il y parait et peut être générateur de nombreux bogues très difficiles à comprendre. Mais là n'est pas le sujet du jour.

C# 5, donc, propose await/async. Malgré les réserves indiquées ci-dessus il s'agit d'un procédé très puissant simplifiant grandement l'écriture du code. Difficile de s'en passer une fois qu'on a découvert son efficacité.

Si vous voulez cibler dans une bibliothèque de code portable à la fois Windows Phone 8 et WinRT, il n'y a aucun problème, await/async sont disponibles dans les deux personnalités de .NET puisqu'elles sont au niveau 4.5 l'une et l'autre.

Maintenant si vous désirez cibler WinRT et Windows Phone 7.1 c'en est fini de await/async car WP7 n'est pas en 4.5… Et comme une bibliothèque de code portable s'aligne sur le plus petit dénominateur commun…

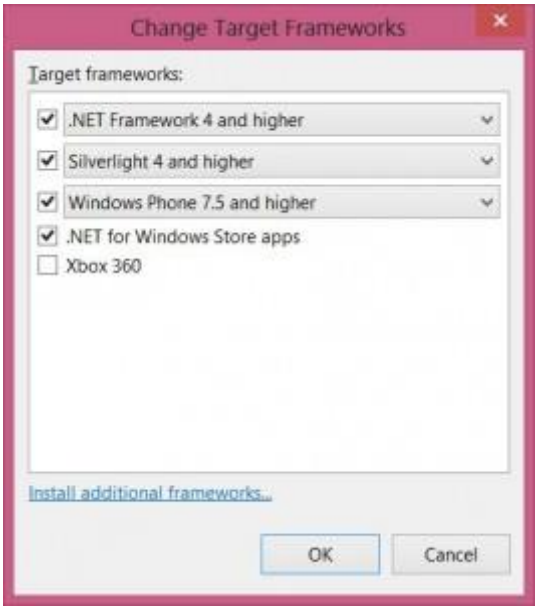
Toutefois il existe une pré-release de Async pour le framework 4 qui fonctionnait aussi avec SL4, SL5, Windows Phone 7.x et 8. De fait il est possible d'utiliser Async dans une librairie de code portable à condition d'installer cette extension (c'est un paquet Nuget).

Possible, mais en framework 4

Si vous souhaitez créer une Portable Library incluant des environnements ne supportant pas Async comme WP7 ou SL4, cela est donc possible à l'aide du paquet Nuget présenté plus haut, mais à une seule condition : cibler le framework 4 et non 4.5.

Création d'une PCL avec Async

Une fois cet impératif pris en compte, il suffit de créer une nouvelle PCL en choisissant les environnements supportés :



Pour ajouter le support à Async il faut bien entendu installer le paquet Nuget. On peut le faire directement par VS 2012 ou bien par la console :

```
Install-Package Microsoft.Bcl.Async –Pre
```

Conclusion

On aime bien pouvoir utiliser toutes les nouveautés à la fois… les Portable Class Librairies et Async par exemple. Même si cela n'est pas toujours évident il existe souvent des astuces permettant d'obtenir ce qu'on désire, la preuve…

Alors ne vous privez ni des PCL ni de Async et développez des applications portables avec le moindre effort !

Contourner les limites des PCL par la technique d'injection de code natif (cross-plateforme)

Les Portable Class Libraries de Visual Studio sont une aide appréciable pour l'écriture des projets cross-plateforme. Mais certains espaces de noms ne sont pas supportés… Comment contourner ce problème ?

Des espaces de noms non intégrés aux PCL

Il existe des espaces de noms qui ne font pas partie du noyau commun exposé par les PCL. J'ai eu dernièrement le problème pour `System.Security.Cryptography`. Au sein d'un projet cross-plateforme Android/WPF j'étais en train d'écrire une fonction de cryptage / décryptage dans le noyau PCL quand je me suis aperçu que

l'espace de noms indiqué ci-dessus n'était hélas pas intégré au jeu réduit du profil .NET de la PCL… Fâcheux…

Heureusement, ce joli code je le tenais d'un autre projet et j'avais seulement copier/coller l'unité de code. Donc pas vraiment de perte de temps. C'est maintenant que le temps allait défiler bêtement, en essayant de résoudre ce problème simple : Pas de cryptographie dans le .NET de la PCL…

Phase 1 : Xamarin à la rescousse

Le noyau CPL ne gère pas l'espace de nom dont j'ai besoin, mais est-ce que Xamarin le gère ? Car si la réponse est non, il va vraiment falloir partir sur une réécriture…

Après vérification Xamarin pour Android possède bien l'espace de nom de cryptographie et expose tout ce qu'il faut, parfaitement compatible avec le code .NET que je venais de récupérer d'un autre projet (Silverlight, c'est pour dire si tout cela est fantastiquement compatible au final !).

Ce fut un premier soulagement, car si la partie Android pouvait grâce à Xamarin faire tourner le code déjà écrit, fortes étaient les chances que le .NET ultra musclé de WPF pourrait le faire sans souci.

Phase 2 : Comment appeler du code Xamarin ou .NET dans le noyau .NET CPL ?

Je me suis immédiatement souvenu de la Vidéo 12 : Injection de code natif (WinRT/Android) que je vous ai présentée le 16 aout dernier…
Dans cette vidéo je démontre comment utiliser dans le noyau CPL du code véritablement natif et spécifique à chaque plateforme de façon simple et transparente. Au moins cela me sert déjà à moi, c'est pas mal ☺

Certes ici je n'avais pas de code "natif", juste du bon .NET bien standard !

Oui mais hélas utilisant un espace de noms non intégré au mécanisme des PCL !

L'idée m'est donc venue de traiter ce code comme du code natif…

Phase 3 : implémentation

Je vais passer les détails puisque la technique est entièrement décrite avec moult détails dans la vidéo sus-indiquée, il suffit de suivre le lien et de regarder… (et d'écouter aussi, mais en général ça coule de source quand on regarde un tutor…).

Je vais donc uniquement résumé le "plan" :

1) Déclarer une interface dans le noyau, imaginons "`ICryptoService`"' déclarant deux méthodes, l'une pour crypter, l'autre pour décrypter.

2) Dans le projet Android (ça aurait pu être le projet WPF, j'ai pris le 1er dans la liste de ma solution) j'ai créé une classe `NativeCryptoService : ICryptoSerrvice` qui implémente les deux méthodes. En réalité ici un simple copier coller du code d'un ancien projet Silverlight donc. Avec l'aide de Resharper tous les "*using*" s'ajoutent facilement.

3) Dans le `Setup.cs` (mécanisme MvvmCross) j'ai ajouté comme montré dans la vidéo un *override* de `InitializeLastChance()` méthode dans laquelle j'ai tout simplement ajouté le type de ma classe native dans le conteneur IoC en tant que singleton. A la première utilisation le conteneur créera l'instance et la conservera pour les appels futurs.

4) Dans le projet WPF (qui aurait donc pu être le projet Android, l'ordre n'a pas de sens ici) j'ai ajouté un répertoire pour le code natif et j'ai ajouté un "*existing item*". Cet item existant c'est le fichier contenant l'implémentation se trouvant dans le projet Android (étape 2). Mais en ajoutant le fichier j'ai demandé à VS de simplement *créer un lien* : de fait je n'ai aucun code répétitif, l'unité de code n'existe qu'en un seul exemplaire.

5) Dans le `setup.cs` du projet WPF j'ai ajouté le singleton comme à l'étape 3 pour le projet Android.

Et c'est tout…

Pour utiliser ce code "natif" dans la PCL je n'ai plus qu'à faire de l'injection de dépendance via les constructeurs et je récupère le singleton, ou bien si ce n'est pas dans un ViewModel ou un Service, je peux appeler directement `Mvx.Resolve()` pour récupérer le singleton.

Mon noyau est dès lors capable d'utiliser les services de cryptographie pourtant indisponibles dans une PCL… Le code natif n'existe qu'une fois dans un seul projet d'UI, il est juste référencé en lien dans le second.

Conclusion

Dans les 12 vidéos que je vous ai concoctées cet été il y a mille et une astuces présentées. En y ajoutant une pointe de créativité on peut en démultiplier l'intérêt…

Ici j'ai pu utiliser la cryptographie dans le projet noyau alors que cet espace de nom n'existe pas dans le profil .NET de la PCL. Le code est unique, non dupliqué, mais lorsque je compile la solution, le projet noyau est compilé par le compilateur Microsoft de PCL, le code de cryptographie sera compilé par le compilateur Xamarin Android dans le projet Android et le code lié (identique) dans le projet WPF sera compilé par le compilateur .NET de WPF…

C'est un joli tour de passe-passe, très simple à réaliser, et qui montre, une fois de plus, l'intérêt de travailler en cross-plateforme et d'utiliser les bons outils : Visual Studio, Xamarin et MvvmCross et surtout C# !

Cross-Plateforme : MvvmCross Seminar video

Pour continuer sur le même sujet, voici une vidéo du séminaire MvvmCross de la fin 2012. La vidéo est commentée par Stuart Lodge lui-même (concepteur de MvvmCross), mon papier et ses exemples sont présentés en fin de vidéo.

Une présentation complète

Je vous conseille le visionnage de cette vidéo très complète commentée par Stuart lui-même, donc forcément au top de l'information sur ce framework cross-plateforme.

Sur Youtube : https://www.youtube.com/watch?v=jdiu_dH3z5k

Un petit coucou à e-naxos

En fin de vidéo, Stuart présente quelques projets intéressants autour de MvvmCross et il évoque à cette occasion le gros travail de présentation que je vous ai offert il y a quelques mois (notamment "stratégie de développement cross-plateforme" en trois parties, partie 1 ici ou bien entendu dans le présent livre PDF pour une version à jour !).

Vous pouvez accéder ici à la vidéo juste au moment où ce travail est évoqué (mais regarder aussi l'heure qui
précède !) : https://www.youtube.com/watch?v=jdiu_dH3z5k#t=1h06m56s
(cette vidéo est en anglais et le son n'est pas très fort alors ouvrez bien oreilles !)

Conclusion

De Gitte en Belgique dont je présentais la vidéo ce matin à Stuart Lodge et à son clin d'oeil à mon travail nous voyons tous clairement, en Belgique, en Angleterre, en France, comment l'avenir s'écrit et quels outils, quels OS vont s'avérer indispensables pour relever le défi du cross-plateforme.
J'espère au travers de Dot.Blog contribuer à cette prise de conscience afin que chaque lecteur soit payé en retour de sa fidélité d'une vision claire du futur qui lui donnera une longueur d'avance sur les autres… Aujourd'hui d'ailleurs il ne s'agit plus d'avoir une longueur d'avance mais de ne pas prendre deux longueurs de retard !

Cross-plateforme : the "magical ring", une session à voir (avec code source)

Je parle beaucoup de cross-plateforme ces temps-ci… C'est normal, c'est l'avenir qui s'écrit comme ça… Voici une conférence à écouter avec le code source des exemples qui viendra compléter mes conseils en la matière (Xamarin, MvvmCross, etc).

Les anneaux de pouvoir du Seigneur des Anneaux…

J'aime cette comparaison que Gitte Vermeiren, une développeuse belge, fait à propos de l'ensemble VS + PCL + Xamarin + MvvmCross.
C'est en effet ce que j'essaye de faire comprendre ici en France depuis déjà longtemps…

Une Conférence TechDays 2013

Microsoft en Belgique a fait preuve de plus d'ouverture d'esprit que Microsoft France pour les TechDays 2013… Gitte a en effet eu la chance de pouvoir dérouler sa session '*Building Cross Platform Mobile Solutions*' sur le développement cross-plateforme au mois de mars dernier, mais aux TechDays belges… Microsoft France avait rejeté ma session pour les Techdays 2013 françaises, presque en tout point identique d'ailleurs, et pour cause, puisqu'elle portait exactement sur le même sujet… Mais en français. Microsoft France a-t-elle eu peur de ce que d'autres instances de Microsoft ont trouvé normal ? Mal typiquement français de l'entre-soi protecteur fermé aux idées qui ne cadrent pas avec le dogme maison ? Je n'en sais rien mais il est dommage que les français aient été privé d'une telle conférence, non pas que je pense que ma

présence spécifique eut illuminé cette manifestation, mais parce que je suis convaincu que le sujet lui-même méritait largement d'être présenté. Aider la communauté Microsoft ce n'est pas lui mentir sur un avenir 100% Microsoft, c'est l'aider à utiliser au mieux les outils Microsoft pour satisfaire les demandes du présent et du futur dans un monde où il faudra marier des smartphones et des tablettes iOS et Android avec des infrastructures et des applications Microsoft.

Bref. J'ai pu rester au lit et m'éviter des heures de répétition (une conférence bien faite est un boulot énorme qu'on ne soupçonne pas) mais je regrette avec tristesse ce manque de lucidité et d'ouverture d'esprit dont à fait preuve MS France.

Reste Dot.Blog pour ceux qui veulent rester informer de ce qui se fera (et non d'un monde parfait où tout se déroulerait tel que Microsoft le rêve).

Et dans ce monde réel, si Microsoft et ses solutions occupent une place importante, si leurs technologies sont toujours celles que je soutiens, il vient un temps où il faut sortir des discours menteurs sur l'avenir. L'avenir devra être partagé avec Android et iOS, c'est comme ça.

Savoir marier des développements pour iOS et Android dans des entreprises généralement équipées Microsoft sera le chalenge des prochains mois et prochaines années pour les DSI et leurs équipes de développement. Autant s'y préparer tout de suite. Moi je le suis, pour moi-même, pour mes lecteurs et pour les clients que je conseille. Mais vous ?

Cet avenir là, c'est ce que je répète ici, c'est ce que j'aurai bien voulu vous expliquer dans ma session TechDays 2013 qui restera dans son coin de disque dur, mais vous n'avez pas tout perdu !

D'une part ces informations se retrouvent détaillées ici au fil des billets, et la session de Gitte est visible en ligne !

La session dure une heure environ, Gitte parle en anglais (c'est toujours plus pratique qu'en Flamand ou en Allemand), et elle est accessible ici : http://goo.gl/c48l4
En fin de session Gitte donne l'adresse sur Github des sources à télécharger pour les étudier au calme.

Conclusion

Ce qu'un certain aveuglement peut arriver à étouffer ici se retrouve fatalement plus tard ailleurs … C'est la magie du Web et de la communication globale échappant au contrôle des entreprises et des Etats.

Franchement la session de Gitte est hyper bien faite, pas certain du tout que je j'aurai fait mieux alors vous y gagnez surement, surtout en anglais, c'est du bon boulot, c'est clair, ça explique tout, c'est propre et on comprend pourquoi pour Gitte cette solution que je vous explique depuis un moment est ce qu'elle appelle son "magical ring"…

Merci Gitte et merci à MS Belgique qui a eu l'intelligence d'ouvrir son espace TechDays à une information essentielle.

Bonne session, regardez et écoutez, c'est important !

Cross-plateforme : Android - Part 1

Le développement cross-plateforme dont j'ai déjà développé ici de nombreux aspects sous-entend aujourd'hui en toute logique de savoir utiliser l'OS Android. Malgré les passerelles (Xamarin, MvvmCross) il reste tout de même à comprendre "comment ça marche Android ?" et surtout de comprendre "pourquoi Android ?"

Des passerelles et des OS

Xamarin 2.0 avec son environnement de développement autonome lui-même cross-plateforme doublé de son intégration à Visual Studio est un outil fantastique pour aborder en toute sérénité les OS mobiles que sont iOS et Android. Mais cela ne fait pas tout…
D'autant que lorsqu'on parle de cross-plateforme sur un blog orienté Microsoft cela laisse supposer qu'on devra se débrouiller pour faire fonctionner avec le maximum de code commun des applications qui tourneront à la fois sur des environnements Microsoft et des environnements de type Apple ou Google.

Pour se faire nous disposons d'un autre outil extraordinaire, MvvmCross que j'ai déjà présenté très longuement (voir "Stratégie de développement Cross-Plateforme" en 3 parties ainsi que les 12 vidéos publiées cet été 2013.).
Mais tout cet outillage ne fait pas tout. MonoTouch et MonoDroid (Xamarin 2.0) et MvvmCross ne sont que des passerelles, des facilitateurs. Il faut nécessairement comprendre les OS ciblés pour faire de vrais développements cross-plateforme.

La partie Microsoft pose certainement à mes lecteurs moins de soucis que la partie Apple ou Google… Donc je m'attarderai plutôt sur ces derniers que sur le fonctionnement de WPF ou WinRT.

Plus exactement je vais vous parler d'Android.

> ***Cross-Plateforme: J'entends par ce terme deux types très différents de développement. Le premier est celui auquel tout le monde pense, faire tourner la même application basée sur un même code sur plusieurs OS. Une application de gestion de rendez-vous qui tourne à l'identique sur IPhone et Windows Phone en réutilisant une code de base identique est un exemple de ce modèle cross-plateforme. Mais j'entends par ce terme aussi une autre forme de développement, plus hybride, plus proche souvent de la réalité : une même application dont différentes parties tournent sur différents OS. Une application de gestion de rendez-vous peut offrir la gestion multi-agenda, des impressions, des statistiques dans sa version PC, et elle peut être complétée par une version Android sur smartphone, pour chaque agenda de chaque utilisateur, et par une version tablette IPad pour le chef de service qui surveille le remplissage des agendas par exemple. C'est la même application avec des fonctions communes et d'autres spécifiques. Elle utilise une partie de code commun et des ajouts particuliers selon le support, le type d'utilisateur et d'OS cible. Cela aussi c'est du cross-plateforme. Et c'est à mon avis le véritable avenir d'un mix intelligent entre tous les form factors car aucun ne supprimera réellement les autres, tous devront travailler ensemble pour offrir plus de services mieux adaptés à chaque cas d'utilisation. Ce sont donc bien ici ces deux aspects très différents du développement cross-plateforme que j'évoque quand j'utilise ce terme.***

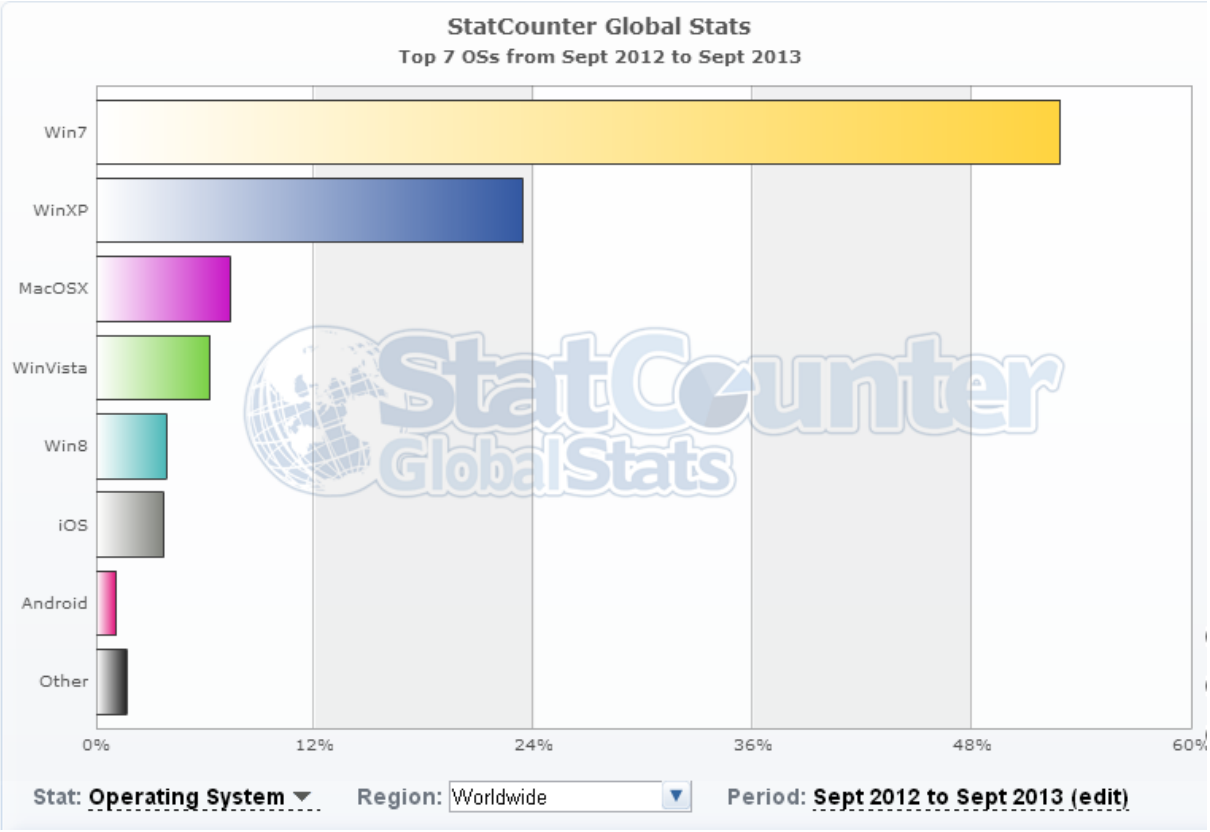
Pourquoi Android ?

Si Windows a été, et restera encore longtemps incontournable dans le monde de l'entreprise pour ses OS serveurs, sa base de données, et ses OS de bureau il s'avère qu'aujourd'hui Android occupe une même position sur tout le reste, à savoir les smartphones et les tablettes. Apple est l'éternel chalenger. D'abord face à Microsoft à l'époque "Mac ou PC ?", match gagné par le couple IBM/Microsoft, et aujourd'hui face à Google dans le match "iOS ou Android", match désormais gagné par Google.

Mais avant d'aller plus loin, il est bon de raisonner sur des bases rigoureuses et vérifiables, sur des chiffres réels issus du marché.

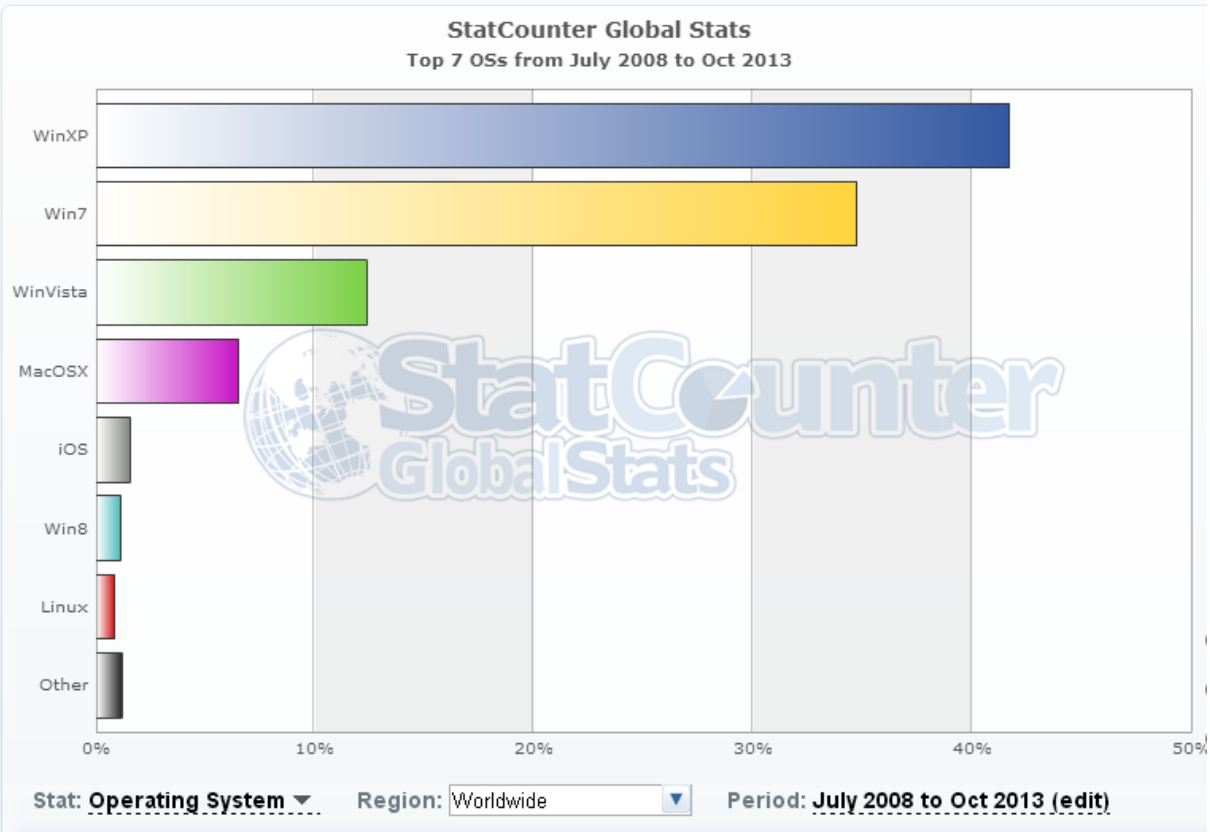
Je veux ainsi que nous partions d'une analyse réaliste du marché, rien d'autre. Ni mes sentiments à l'égard d'Apple, ni mon attachement à Microsoft, ni le fait que je ne trouve pas Android particulièrement exaltant ne doivent compter. Ce qui compte c'est le marché, et ce n'est pas moi qui le fais.
Je ne suis le VRP de personne et mon rôle n'est pas de "forcer" le destin d'un produit ou d'un autre, mais celui de conseiller clairement en fonction d'une réalité objective. Et j'essaye de m'y tenir en toute occasion, quelles que soient mes "j'aime" ou "j'aime pas" et indépendamment de toute affiliation ou rapprochement avec tel ou tel éditeur, Microsoft compris.

Répartition des OS sur la dernière année de sept-2012 à sept-2013



Ce graphique montre le Top 7 des OS dans le monde sur l'année passée. Vous pouvez obtenir les chiffres exacts sous différentes formes (fichier csv, graphique…) en vous rendant sur le site de StatCounter.
On voit clairement sur cette période la domination de Windows 7, suivi de plus en plus loin par Windows XP et même ce sacré Vista qui résiste (à peu près au même niveau que OSX) ! Tout au bout du classement, là où les parts de marché n'ont pas de prise sur les décisionnaires, on trouve en vrac iOS, Linux, Windows 8 et d'autres.

En dehors de l'inversion XP/Windows 7, la même étude en remontant depuis 2008 donne une vision encore plus nette de la domination de Windows « classique » :



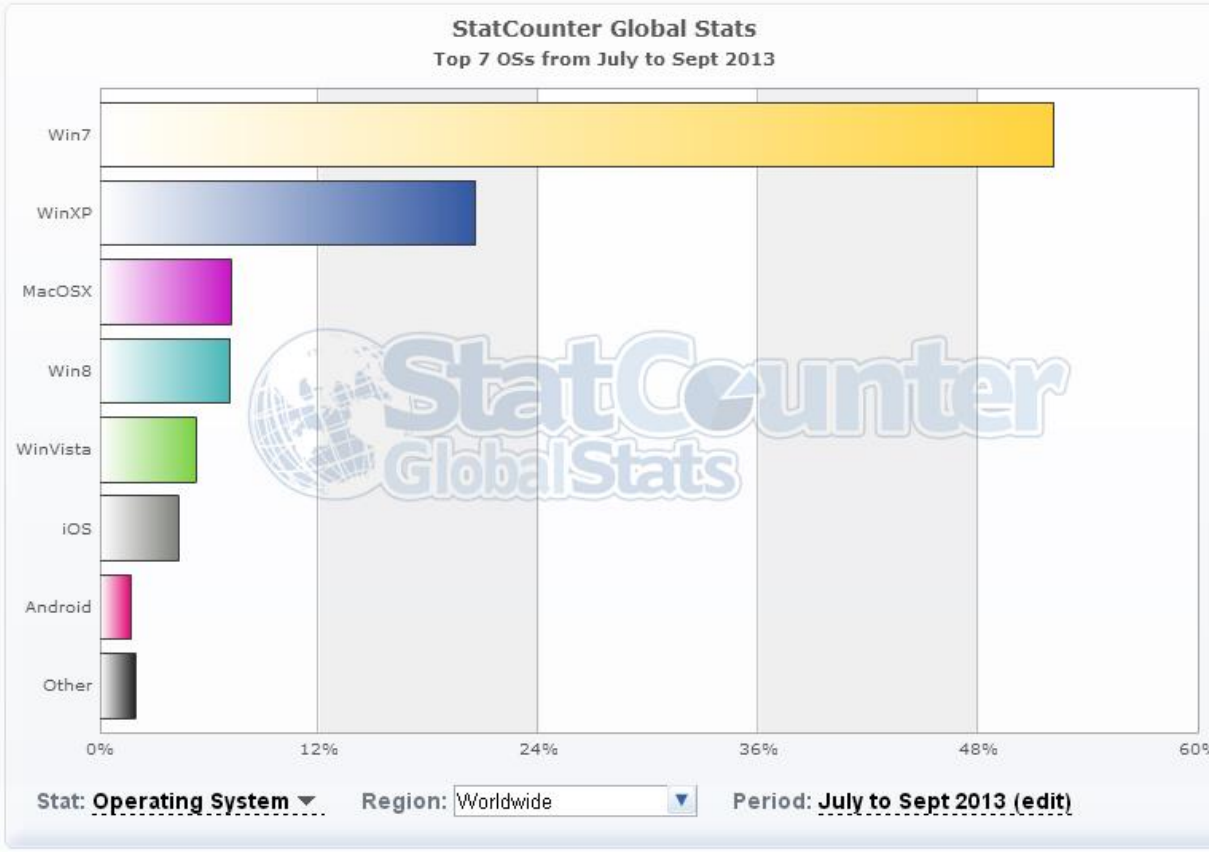
Ca c'est le marché global des OS sur les 4/5 dernières années, la répartition dans le monde des OS et de leur poids réel pour prendre des décisions (à quelques pourcents près, chaque fournisseur de statistiques ayant ses propres biais).

Mais qu'en est-il sur le marché actuel, sur les ventes et les tendances de l'instant ?

Répartition du marché des OS sur les 3 derniers mois

Bien entendu cette image est une capture à un instant donné, dans 6 mois je conseille aux lecteurs qui passeront ici de suivre le lien donné plus haut pour avoir des données à jour…

A la date d'écriture de livre PDF les trois derniers mois, de juillet à septembre 2013, donnent la répartition suivante:



Windows 7 domine très largement aujourd'hui. Windows XP ne représente plus qu'un petit quart du marché, ce qui est malgré tout énorme. Environ 75% du marché est aujourd'hui couvert par XP et 7. Des OS sachant faire tourner des applications WPF mais pas WinRT.

Mac OSX reste stable, Vista disparait petit à petit, et iOS fait un score inférieur à Windows 8 mais ce dernier mélange un peu tout, dont les PC, là où iOS ne concernent que les unités mobiles. Android sur ces trois derniers mois semble en recul, c'est un biais de cette statistique particulière. D'autres statistiques vont nous éclairer sur ce point.

En effet StatCounter analyse les visites sur le Web pour établir ses statistiques ce qui est assez juste pour se faire une idée de la répartition Mac/PC et leurs OS respectifs mais qui introduit un biais très fort pour les OS mobiles car on sait que les utilisateurs de ces derniers préfèrent largement les applications natives aux sites Web au travers d'un browser. Quand on possède des smartphones et mêmes des tablettes cela est une évidence. Les browsers mobiles sont peu ergonomiques (même un geek comme moi trouve cela pénible et le pire : compliqué à comprendre), mais d'autres raisons s'imposent immédiatement comme le fait que peu de sites offres une mise en page compatible (Dot.Blog offre un accès spécial aux smartphone par exemple, mais c'est encore rare). Tout cela contribue à diminuer la part des mobiles dans les statistiques obtenues depuis les visites sur le Web. De plus, d'autres analyses prouvent que pour

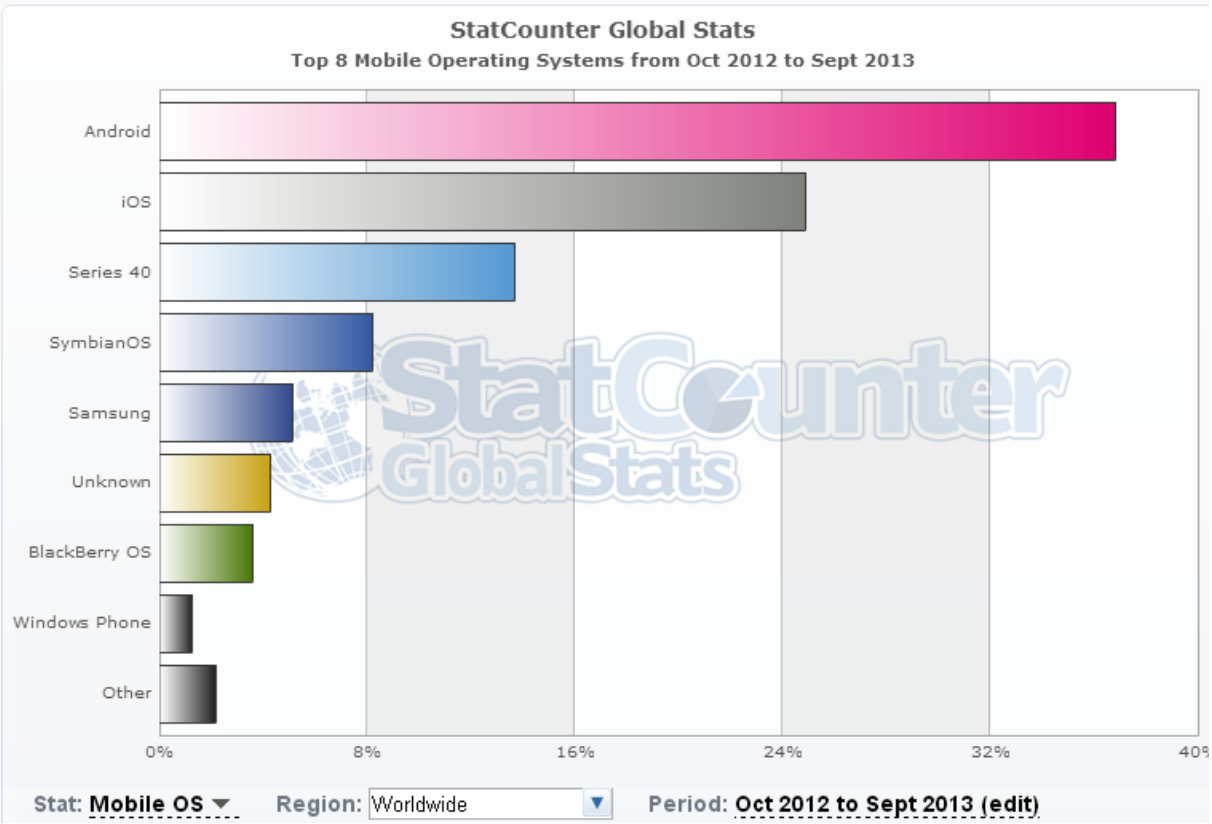
des raisons à déterminer les utilisateurs Apple utilisent beaucoup plus Internet avec un browser que ceux d'Android. Dans le graphique ci-dessus qui prend en compte tous les OS, mobiles ou non, la part des mobiles est sous-estimée et celle d'Android encore plus. Pour raisonner sur les mobiles il nous faut des statistiques où seuls les OS mobiles sont pris en compte.

Pour mieux comprendre voici d'autres statistiques (source w3schools) :

2013	Win8	Win7	Vista	NT*	WinXP	Linux	Mac	Mobile
September	10.2%	56.8%	1.6%	0.4%	13.5%	4.8%	9.3%	3.3%
August	9.6%	55.9%	1.7%	0.4%	14.7%	5.0%	9.2%	3.4%
July	9.0%	56.2%	1.8%	0.4%	15.8%	4.9%	8.7%	3.2%
June	8.6%	56.3%	2.0%	0.4%	15.4%	4.9%	9.1%	3.2%
May	7.9%	56.4%	2.1%	0.4%	15.7%	4.9%	9.7%	2.6%

On le constate, malgré le "buzz" autour des mobiles, que ces derniers ne pèsent pas encore très lourd (3,3 %) par rapport au monstre qu'est Windows et au monde des PC en général. On constate au passage que la progression de Windows 8 ne provient que de la baisse cumulée de XP, de Linux et du Mac pour les principaux. Windows 7 arrive à progresser contre Windows 8 et malgré les volumes énormes que cachent ses chiffrent (il y a eu environ 3 milliards de PC vendus et 1,5 milliard sont en utilisation) le marché du mobile a progressé lui aussi.

Répartition des OS Mobiles sur l'année 2012/2013



Sur l'année écoulée, en cumul, Android est bien largement en tête à près de 40% du marché (c'était 30% quand cet article a été écrit il y a juste quelques mois !). iOS chute inexorablement, parti de presque 100% il ne couvre plus que 25% environ du marché, ce qui reste très significatif malgré tout et qui ne peut être ignoré lorsqu'on vise certains marchés.

Symbian OS a toujours ses fans, comme d'autres. Windows Phone, même la version 7 vendue depuis plusieurs années n'apparaissait nulle part dans ce classement quand j'ai écris ces lignes, aujourd'hui, donc quelques mois plus tard WP8 apparait timidement avec le score le plus bas… On pourrait invoquer la jeunesse (relative aujourd'hui) de l'OS, mais les statistiques mélangeant WP7 et WP8 l'argument ne tient pas. Les premières statistiques à la sortie de WP7 donnaient 2% environ et incluaient les parts de « Windows Mobile » ! On assiste au même mélange mais malgré ces trucages incessants Windows Phone reste un boulet en fond de classement. Mêmes le fatras des sans nom des « autres OS » compte pour le double des parts de Windows Phone…

Voici une autre vue pour 2013, bien plus simple, d'une autre autre source :

Mobile Devices Statistics

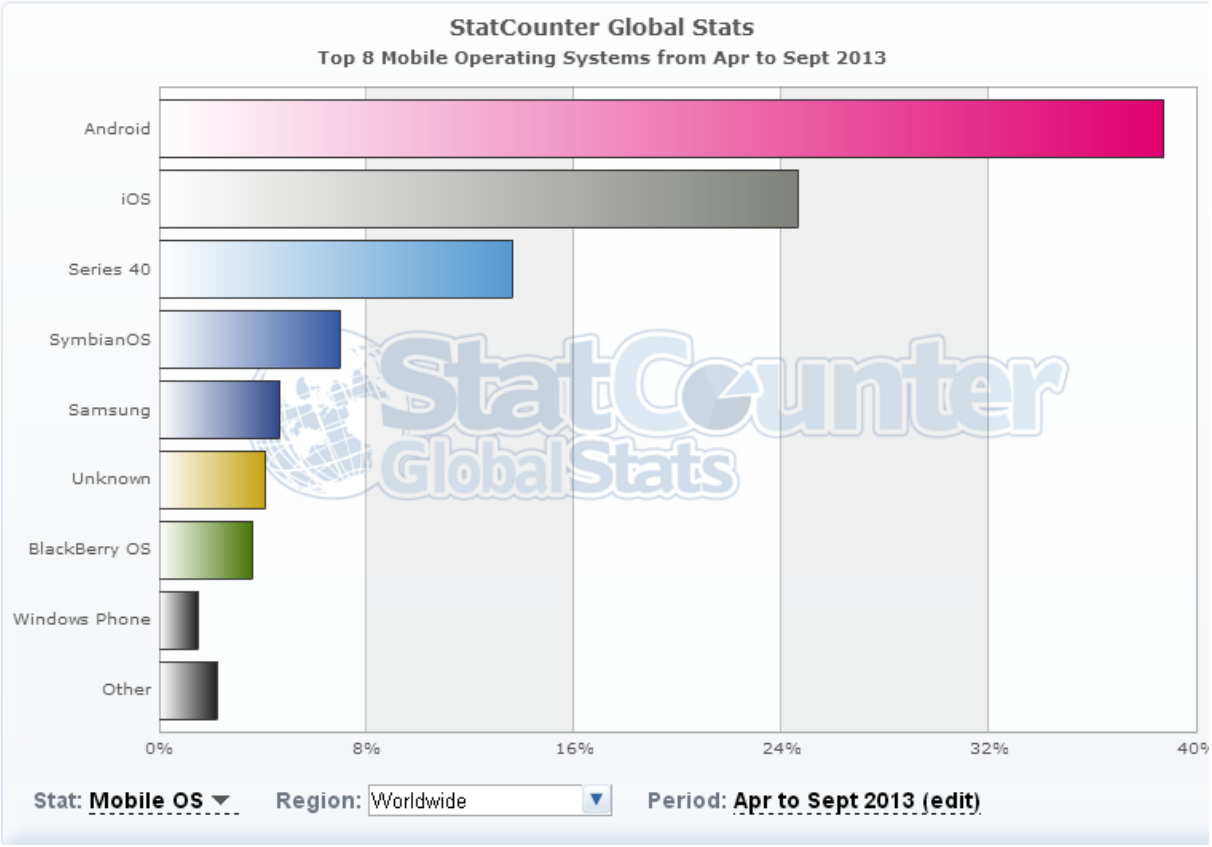
The following table is a breakdown of the Mobile numbers from our OS Platform Statistics:

2013	Total	iOS *	Android	Others
September	3.26 %	0.95 %	1.55 %	0.76 %
August	3.38 %	1.21 %	1.43 %	0.74 %
July	3.16 %	1.03 %	1.37 %	0.76 %

Dans ce découpage (qui prend en compte la part des mobiles par rapport à tous les OS) il n'existe que trois catégories : iOS, Android et "les autres". La tendance qui se confirme est celle d'une montée d'Android (1.55 % contre 0.9 au moment de l'écriture du billet original), d'une baisse importante à cette échelle d'iOS et unun maintien des « autres » à 0.76%. Connaissant la répartition des « autres OS » séparée de celle Windows Phone dans le graphique précédent (la moitié des autres OS), on peut affirmer que la présence de ce dernier reste anectodique avec une moitié de 0.76% du global des OS.

Peut-être les tendances ont-elles évolué ?

Répartition des OS Mobiles sur les 6 derniers mois

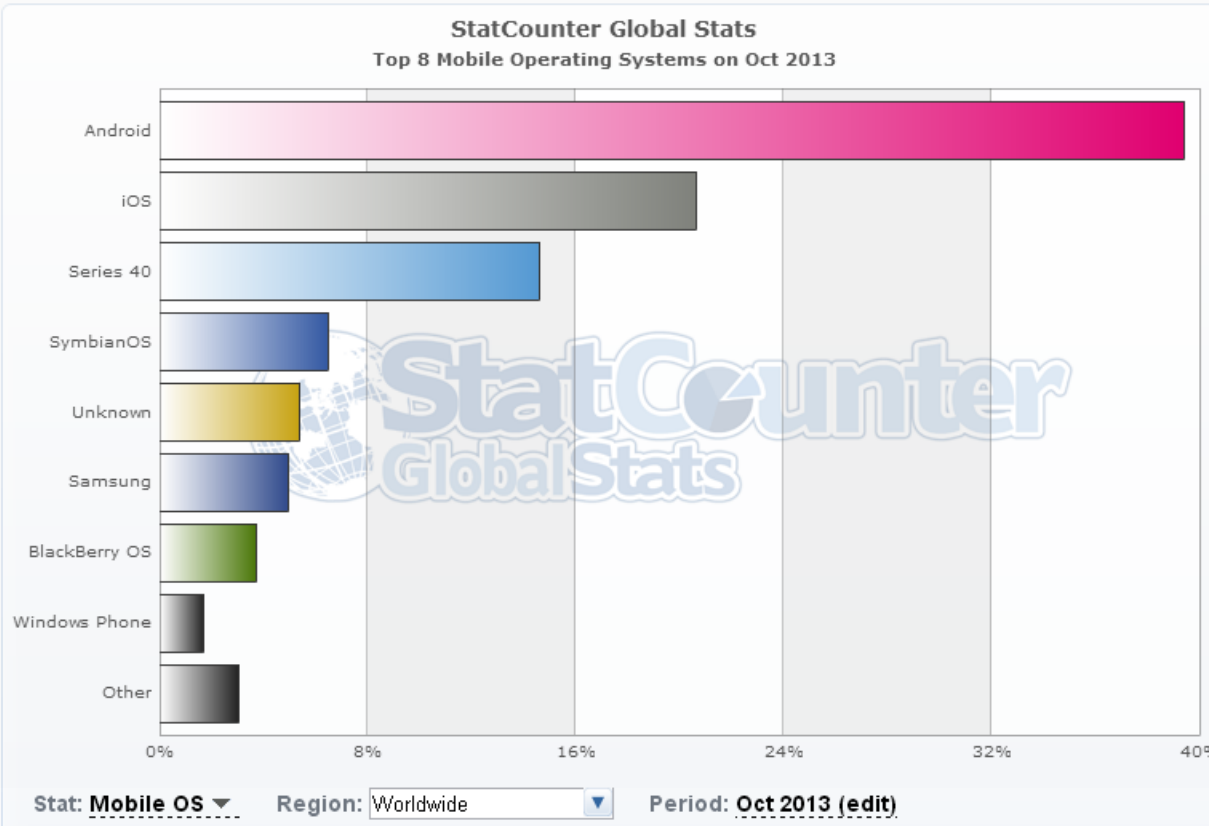


Sur les 6 derniers mois les tendances sont nettes : Android est près des 40% (contre 35% à l'écriture du billet originel, en quelques mois !), soit une progression faramineuse, et iOS se maintient à au dessus de 24 % ce qui reste toujours aussi honorable, bien qu'en baisse de 1 à 2 points par rapport à il y a quelques mois.

Windows Phone, toutes version confondues, apparait aux alentours de 2% alors qu'il représentait près de 3,6% lors de l'écriture du billet originel (selon les analystes entre 2,6 et 3,8%). C'est à dire bien en dessous même de Symbian qui flirte avec les 8% (contre 10% il y a quelques mois) bien que cet OS soit "mort" technologiquement.

Mais peut-être tout cela évolue-t-il d'une façon surprenante depuis peu ?

Répartition des OS Mobiles sur les 27 premiers jours de octobre 2013



En effet, les tendances bougent dans l'immobilité ! Android est encore plus près des 40% que sur les 3 derniers mois, indiquant une progression toujours régulière de mois en mois. iOS reste en seconde place mais en perdant presque 10%, etc, et Windows Phone reste encore bien au fond, collé contre le radiateur malgré les annonces tonitruantes de ces dernières semaines (Windows Phone serait le 3ème OS mobile… Comme quoi certains croient encore à la bonne vieille méthode du Dr Coué !).

Mais ces quelques changements ne font qu'enfoncer un clou qu'on connait déjà…

Ce qu'on voit ? Android qui atteint presque les 40%, iOS qui plonge. Forcément ce sont donc tous les autres qui trinquent de la montée d'Android, certains arrivant à se refaire sur la perte de iOS. Windows Phone (7, 7.1, 7.5, 7.8 et 8.0 tout mélangé) lui aussi reste stable… hélas. Toujours à quelques pourcents du marché. Aucune tendance à la hausse ou à la baisse visible sur les graphiques. Symbian disparait lentement mais se tient toujours légèrement en dessous des 8%.

Les Ventes

Tous ces chiffres sont intéressants mais ils partent tous d'une analyse du Web (visiteurs de sites Web) ce qui introduit un biais certain déjà évoqué mais difficilement mesurable pour être corrigé.

Il faut donc une autre source d'information basée sur d'autres méthodes, notamment les ventes de matériel. Ici aussi il existe un biais connu : les ventes livrées aux magasins (physiques ou en ligne) ne sont pas toujours égales aux achats des consommateurs. Mais les vendeurs faisant généralement bien leur métier il est rare (mais pas impossible) qu'ils achètent des quantités énormes d'un produit qui ne se vendra pas du tout (même si l'exemple des $900 millions provisionnés par Microsoft pour les méventes de Surface de première génération montrent que le biais peut être énorme pour certains produits). Il faut donc être conscient de cet écart entre vendu aux boutiques et vendu aux clients.

En prenant les chiffres de IDC on trouve pour le dernier quart 2013 :

Top Smartphone Operating Systems, Shipments, and Market Share, Q2 2013 (Units in Millions)

Operating System	2Q13 Unit Shipments	2Q13 Market Share	2Q12 Unit Shipments	2Q12 Market Share	Year-over-Year Change
Android	187.4	79.3%	108	69.1%	73.5%
iOS	31.2	13.2%	26	16.6%	20.0%
Windows Phone	8.7	3.7%	4.9	3.1%	77.6%
BlackBerry OS	6.8	2.9%	7.7	4.9%	-11.7%
Linux	1.8	0.8%	2.8	1.8%	-35.7%
Symbian	0.5	0.2%	6.5	4.2%	-92.3%
Others	N/A	0.0%	0.3	0.2%	-100.0%
Total	**236.4**	**100.0%**	**156.2**	**100.0%**	**51.3%**

Cette vision est assez différente de l'analyse du Web. Elle montre que sur les 236,4 millions de machines vendues 187,4 millions sont des Android ! La part d'Android entre fin 2012 et fin 2013 est passée de 69.1% à 79.3% du marché. Apple avec iOS a baissé de 16.6 % à 13.2 % (alors que sur l'année dernière cette chute partait de 37% pour tomber à 29.2%, la dégringolade n'en finit plus).

On pouvait se réjouir de voir Windows Phone faire un bond de 150% sur la même période… l'année dernière. Mais come je le disais alors, soit c'est un succès intergalactique soit c'est qu'en partant de zéro la moindre progression passe pour une performance, ce qui était le cas. Depuis, et seuls quelques mois ont passé, les délires de progression sont revenus à la réalité : la progression de Windows Phone est de 3.1% à 3.7% sur un an. C'est très insuffisant et montre les difficultés rencontrées par Microsoft à convaincre au-delà d'un petit cercle de convaincus qui sont déjà clients. Progresser de 0.6% en un an fait moins rêver que 150%… mais c'est plus réaliste ! Et si on compare le nombre d'unités vendues, avec 8.7 M d'unités, même iOS en chute libre reste une cible inacessible avec des 31,2 M d'unités !

Mais là n'est pas la question puisqu'aujourd'hui ce qui nous intéresse c'est de comprendre le phénomène Android et pourquoi il est essentiel pour vous de vous y intéresser de plus prêt… Les gloires et déboires des concurrents sont en dehors de notre sujet, même si pour constater la suprématie de Android nous sommes bien

obligé de constater aussi la défaite des autres OS et la tragique stagnation de Windows Phone et même des tablettes Surface.

Comme je l'ai expliqué, je ne suis pas ici pour cirer des pompes ou faire de la propagande. Mes états d'âme sont sans importance. Ma passion c'est C#/Xaml et programmer Windows Phone ou Surface est un plaisir immense. Ma mission première est de vous faire partager mes passions et mes conseils. Là je ne parle pas de mes passions, je fais du conseil. Et quand je conseille, je suis factuel, sinon je mens. J'en ai horreur et c'est bien pour cela que je suis Conseil en informatique et non Représentant…

Que conclure de ces chiffres ?

On le voit clairement, quels que soient ses idées personnelles, ses coups de cœur ou ses bêtes noires, voire même son indifférence, les tendances du marché sont évidentes.

L'analyse du Web donne une idée intéressante des rapports des forces en jeu, les chiffres des ventes d'IDC viennent écraser toute forme de relativisme…

Android n'arrête pas sa progression et domine totalement le marché. Point.

Mais après avoir été détrôné de la première place on aurait pu croire que iOS s'accrochait plutôt bien et refuserait de céder plus de terrain en se maintenant à 20/25% du marché environ, mais on s'aperçoit à l'aune des chiffres récents qu'il n'en est rien. La chutte tragique se poursuit.

Ce sont donc tous les autres OS mobiles qui pâtissent désormais de la montée d'Android. Apple à beaucoup perdu et va perdre encore certainement un moment vu l'orientation de son marché. Mais tous les autres se font dépouiller.

Parmi les perdants il y a les systèmes morts comme Symbian, ceux en train de mourir comme BlackBerry et ceux qui voudraient exister mais qui n'y arrivent pas comme Windows Phone, faute de place tout simplement.

Je suis désolé de cet état de fait. Je préfèrerais boire le champagne pour fêter les 25% de part de marché de WP8 et passer mes journées à développer des trucs en C#/Xaml pour cet OS, mais la réalité est toute autre. Et les tendances ne montrent guère d'espoir à courts ou moyens termes d'un renversement de tendance sans un renversement de stratégie de Microsoft aujourd'hui symbolisé par un changement à venir de CEO que j'appelle de mes vœux depuis deux ou trois ans… De toute façon il

y a déjà deux géants, Apple et Google, dans un lit trop petit pour deux. Il faudrait un miracle pour que Microsoft arrive à chasser l'un ou l'autre pour se trouver un coin d'oreiller… Pour l'instant il joue le rôle du tapis au pied du lit.

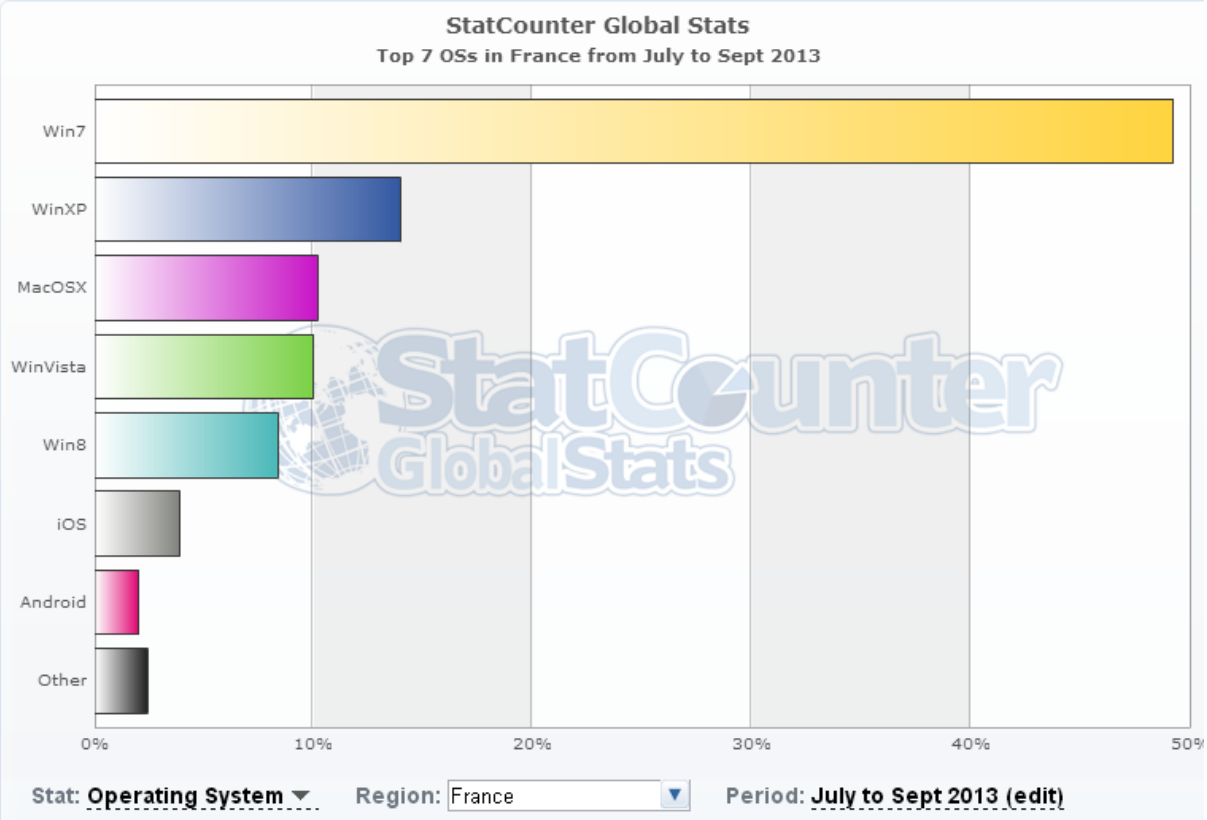
Donc quand je parle aujourd'hui de développement cross-plateforme, et en me limitant à la réalité du marché, cela implique les OS et technologies suivantes :

- Win32/Win64 pour les PC, en C#/Xaml donc WPF
- Android de Google pour les smartphones et les tablettes, avec MonoDroid/Xamarin en C#

J'exclue désormais totalement iOS.
Quant à Windows Phone ou WinRT pour Surface, éliminant iOS je ne serais pas rationnel en ne les éliminants pas aussi… C'est dur d'être rationnel !
Car je les aime bien Windows Phone et WinRT, j'aimerais leur trouver une place...
Alors un peu de subjectivité : en entreprise je suis certains qu'ils peuvent trouver cette place et simplifier la gestion des équipes de développement. C'est un bon choix pour le DSI que de fédérer sous un seul langage, une seule plateforme tous les form factors. Ce que Apple ne propose pas avec iOS et le Mac, Apple n'ayant aucune stratégie Entreprise sérieuse, je peux ainsi renverser mon conseil qui paraissait subjectif en lui offrant une assise rationnelle et factuelle !

Malgré tout, iOS ne peut être évacué pour certains développements smarphones et doit être pris en compte pour les développements tablettes. Tout dépend du produit et de la cible. Pour l'entreprise l'affaire est pliée, pour un éditeur visant le grand public c'est une option qui peut encore être étudiée. Mais la montée irrésistible d'Android, la gamme gigantesque de machines différentes pour cet OS, le choix des résolutions, des tailles, mais aussi les prix attractifs sont certainement des éléments à prendre en compte s'il faut équiper 50 représentants dans une entreprise… Si on intègre ces avantages pratiques d'Android et les chiffres des ventes, et sauf marché particulier, on aura donc intérêt, même sur tablette, à faire le choix de cet OS.

Pour rester en France, complétons ces analyses par la répartition des OS (tous, pas seulement les mobiles) sur les 3 derniers mois :



Dans notre beau pays on voit que Windows 7 domine à près de 50% le marché suivi de XP à près de 14%, talonné par Vista et Mac OSX à 10%. Windows 8 arrive ensuite à 8 % environ. C'est à dire que pour les PC, en France, il y a en gros 74 % de machines ne pouvant faire tourner des applications WinRT, et si on raisonne dans l'autre sens il existe 82% des machines pouvant exécuter une application WPF.

N'étant pas un représentant de Microsoft (ce n'est pas ce qu'on demande à un MVP qui est un "expert indépendant"), ni un employé de la marque, et même si j'en suis franchement navré pour de nombreuses raisons, je ne vois aucune raison (au sens de raisonnable) de conseiller WinRT que cela soit sur PC, tablette ou Smartphone. En tout cas aujourd'hui. Dans un avenir plus ou moins lointain, après un changement net de stratégie de Microsoft, j'espère revenir sur ce constat et pouvoir enfin faire entrer WinRT dans le "bouquet" des technologies à conseiller. Ce n'est pour l'instant pas le cas.

Cela ne signifie pas que tout ce qui vient de Microsoft est à éviter ! Ce n'est pas parce qu'une seule de leur technologie n'a pas le vent en poupe que l'ensemble des produits de la marque n'a plus aucun intérêt. Les derniers chiffrent publiés par Microsoft le prouvent d'ailleurs, c'est grâce à la confiance des Entreprises pour SQL Server, Sharepoint, Office, CRM, etc, que Microsoft gagne de l'argent. Il y a juste une découplage entre cette confiance aux produits « lourds », aux infrastructures

Microsoft et la défiance des mêmes entreprises aux grosses bêtises du discours (et des produits) « grand public » de l'éditeur.

Une erreur, un mauvais calcul n'est pas grave quand on possède comme Microsoft des dizaines de technologies toutes meilleures les unes que les autres ! A condition que la nouvelle direction à venir en prenne conscience et recentre ses efforts sur les Entreprises !

Parmi les technologies Microsoft qui restent parfaitement d'actualité et sur lesquelles il faut continuer à investir on peut lister :

- WPF, incontournable pour faire des logiciels modernes qui marchent sur 100% du parc de PC sous Windows
- Visual Studio, même pour le développement sous iOS ou Android car Xamarin 2.0 s'intègre à cet EDI le plus intelligent du marché (pour un résultat plus puissant que Xamarin Studio, malgré ses qualités).
- SQL Server, qui reste une fabuleuse base de données, rapide, puissante, capable de monter en charge et de rester stable sur des dizaines de millions d'opérations et des fichiers gigantesques,
- Windows 8 en tant qu'OS qui est rapide et stable et vraiment agréable (tant qu'on reste en bureau classique et passés les désagréments de l'update 8.1)
- Silverlight, même s'il est mort pour le Web grand public il reste l'unique outil de cette puissance sans équivalent pour créer des Intranet professionnels et sera soutenu encore dix ans,
- Office, SharePoint, Blend, …

> ***Parler de développement cross-plateforme aujourd'hui en restant cohérent par rapport au marché, cela consiste donc à créer des logiciels qui fonctionnent sous WPF côté PC et sous Android pour le reste. Les entreprises peuvent avoir certains intérêts à se tourner vers WP8 et WinRT pour des raisons de cohérence des équipes techniques.***

Alors, pourquoi Android ?

Est-ce que Android me fascine ? Honnêtement non. Android c'est un Linux 2.6 modifié pour les smartphones. Rien d'excitant ni d'exaltant outre mesure. Mais Android est une réalité incontournable, comme le fut Microsoft sur les PC dans les 25 dernières années.

Est-ce que cette réalité modifie ma “loyauté” vis-à-vis de Microsoft ? Non plus. Je constate seulement que la réalité du marché est ce qu’elle est et que pour l’instant Microsoft n’a pas su prendre une place significative sur le marché des mobiles. Je trouve cela dommage, mais je conseille mes clients et mes lecteurs en fonction de la réalité du marché et de la direction que je pense qu’il prendra. Pour l’instant aucun indicateur ne me laisse penser que Microsoft sera capable de repousser le succès d’Android. Apple tiendra encore un moment sa place de second et ce n’est pas Windows Phone 8 qui détrônera l’IPhone c’est certain. Apple et Google pesant 90% du marché mobile environ, cela laisse très peu de place à Microsoft même s’ils modifiaient radicalement leur stratégie pour revenir dans la course.

Dès lors, pour tout ce qui est mobile, et pour les raisons exposées plus haut je ne peux que conseiller d’utiliser Android sans oublier iOS ponctuellement sur certains segments de marché qui le nécessite.

Voilà pourquoi je vais vous parler d’Android et que je continuerai à parler de WPF.

Vais-je me taire au sujet de WinRT ? Non. J’aime bien WinRT. Je regrette qu’il soit mal vendu et si peu adopté, mais il n’est pas dit que Microsoft n’arrive pas à lui faire une place sur le marché. Il est trop tôt pour parler d’échec définitif. Perdre une bataille n’est pas perdre la guerre.

Mais entre garder un œil attentif sur une technologie et la conseiller il y a un monde. L’heure n’est pas à conseiller WinRT tout simplement.

Conclusion de la partie 1

Cette première partie n’était pas technique, mais elle était nécessaire. Faire des choix, donner des conseils, tout cela n’a de sens et de portée que si on explique rationnellement le pourquoi du comment.

Android n’est pas un OS qui me fascine, mais grâce à Xamarin c’est un OS que je peux programmer rapidement sous Visual Studio en C#. Et ça c’est un atout, celui de la productivité.

Android a une place sur le marché totalement incontournable, une importance aussi grande sur les mobiles que Microsoft en a sur le monde des PC et l’ascension n’est pas terminée.

Demain (fin d’année 2013 normalement), des AndroidBooks vont sortir. Je ne parle pas de ChromeBooks machines trop en avance sur leur temps car réclamant une

connexion permanente (ou presque) en tout endroit ce qu'aucun pays même le plus civilisé ne peut offrir aujourd'hui. Les AndroidBooks seront des PC équipés d'Android 5, des PC avec tout l'univers connu et aimé du grand public. Si cette sortie est gérée correctement c'est la suprématie même de Microsoft sur ce segment des PC portables qui pourraient être attaquer gravement.

On le voit, aujourd'hui Android est une force montante. Cet OS a interdit à Microsoft de se battre contre Apple et son IPhone, demain cet OS s'attaquera au cœur de la domination de Microsoft, les portables grand public. Google a acquit une suite bureautique compatible Office qui sera certainement présente sur les tablettes et AndroidBooks prochainement. Tous les indices en notre possession montrent qu'Android n'a pas encore terminé sa percée. Les mêmes indices nous prouvent que Microsoft n'a toujours pas réussi à jouer un rôle dans le monde des mobiles et que même sur le terrain des PC ils pourraient être gravement mis en danger. Ces jours derniers des analystes financiers sont allés jusqu'à conseiller de vendre les actions MS avant qu'elles ne chutent encore plus bas.

Je pense que Microsoft est à la croisée des chemins. S'enferrer dans la stratégie actuelle serait une erreur mais on ne voit pas de modification en cours ni à a venir. Microsoft est une entreprise puissante qui a les reins solides. Il faut voir cela comme une mauvaise passe. Je ne suis pas pessimiste sur l'existence même de Microsoft. C'est simplement leur suprématie sur le monde des OS qui se termine. Il suffit qu'une Direction renouvelée prenne les bonnes décisions pour de nouveau flirter avec le leadeurship j'en suis convaincu. Toutefois je ne suis pas voyant, je ne prédis pas l'avenir dans le marc de café et je m'en tiens, en bon informaticien, aux chiffres, à la réalité du marché. Et cette réalité est aujourd'hui très favorable de façon durable à Android.

Faire du cross-plateforme dès maintenant c'est simplement accepter la réalité, loin des dogmes et des clans. C'est juste admettre de voir le monde tel qu'il est et non tel qu'on le voudrait. Et cet apprentissage d'Android que je vais vous proposer dans de prochains billets, peut-être qu'un jour il ne vous servira pas juste à programmer des smartphones, mais bel et bien tous les PC qui équiperont vos entreprises…

Nous vivions depuis deux ou trois ans, depuis le "big shift" de Microsoft, dans un monde plein de doutes. Fallait-il suivre aveuglément Microsoft ou bien vendre son âme à ce diable de Jobs ? Fallait-il développer en double, en triple, en quadruple, en quintuple même! Sous iOS, Android, Win32/64, WinRT, HTML5/JS ?

Nous étions dans une angoisse dont le poids n'a cessé de s'accroitre ne pouvant plus déterminer le vrai du faux, l'intox de l'info, et ne sachant tout simplement pas le lire le futur…

Ceux de ma génération ont connu cette même angoisse lors du match Mac/PC. Fallait-il suivre Apple déjà connu ou faire confiance au "sérieux" d'IBM ? Fallait-il développer en double tous les softs ? Combien de projets ont capoté, combien ont été différés, combien se sont trompés… Et le jour où la suprématie, même à peine frétillante, d'IBM/Microsoft s'est fait sentir, tous nous avons plongé. Nous avons suivi cette voie car c'était enfin le paradis : la paix de l'âme, plus besoin de choisir. Nous avons gagné 25 ans de stabilité à ne plus connaitre les affres d'un tel questionnement. Par notre choix d'opter pour l'un plus que l'autre nous avons offert à l'ingénierie logicielle tous les progrès des 25 années passées. Si vous ne programmez pas en assembleur aujourd'hui c'est grâce à cette paix que ceux de ma génération vous ont offert.

Ces dernières années les bouleversements ont été tels que nous étions replongé dans ce cauchemar affreux. Et comme avec le progrès tout est toujours mieux qu'avant, ce n'était plus deux OS entre lesquels il fallait choisir mais trois ou quatre !

Aujourd'hui nous vivons ce même frémissement d'une stabilité si reposante à venir. Cette voie c'est celle d'Android. On ne se demande pas si on aime. Franchement le premier MS-DOS sur le premier PC était une atrocité comparée au Macintosh ! Android est-il exaltant ? Pas plus que ne l'était MS-DOS. Est-il plus puissant, plus beau ? Non. Mais il partage avec MS-DOS une chose essentielle : il nous promet une nouvelle vague de stabilité. Et seule cette stabilité permet à l'industrie du logiciel de fleurir, de s'épanouir. Nous autres concepteurs de logiciels ne sommes pas des mécaniciens. Les ingénieurs qui travaillent sur les circuits intégrés peuvent sortir de nouvelles machines tous les mois. Nous il nous faut des années parfois pour élaborer une pensée cohérente pour les programmer correctement. Sans stabilité note métier ne peut exister sérieusement.

L'industrie du logiciel dans sa totalité a besoin de stabilité, Microsoft l'a oublié en troublant le marché, en distillant la peur, en provoquant un changement brutal sans option. Android arrive à point, conquérant et aimé du public pour nous offrir la paix 25 ans après ce que Microsoft avait su faire avec Bill Gates. J'aurais préféré que Ballmer nous offre 25 ans de paix supplémentaire avec WinRT sur mobiles, c'est d'Android et de Google que vient cette paix, il faut la saisir, maintenant.

Le choix d'Android est donc un choix de la raison, celui de la stabilité, car seule la stabilité nous permet de faire notre métier de façon intelligente et de générer du business profitable à tous…

> ***"La peur est le chemin vers le côté obscur, la peur mène à la colère, la colère mène à la haine, la haine … mène à la souffrance". Maître Yoda.***

Avec Android choisissons le côté clair de la force.

Cross-Plateforme : Android – Part 2 – L'OS

Dans la partie 1 de cette série dédiée au développement cross-plateforme avec Android je vous ai présenté l'état du marché et pourquoi il faut s'intéresser à cet OS pour l'intégrer dans une logique plus vaste de développement d'applications multi-form factors. Il est temps de présenter les bases de l'OS.

Un OS est un OS (M. Lapalisse)

Je ne vais pas vous retracer tout l'historique d'Android, on s'en moque un peu, comme je ne m'étendrai pas ce qu'est un OS et à quoi cela sert. Tous les OS se ressemblent : ils font marcher un ordinateur et exposent des APIs que les applications peuvent utiliser pour leur propre fonctionnement.

Android est basé sur **Linux 2.6**. Linux tout le monde connait, c'est un OS libre, ouvert, et dont les qualités sont tout à fait honorables au point qu'il est utilisé assez largement dans le monde.
Pendant longtemps Linux a été vu comme un OS complexe, en mode ligne de commande, case-sensitive, se programmant en C, bref un truc de geek pas du tout adapté aux utilisateurs finaux. Pendant tout aussi longtemps les linuxiens nous ont promis le "grand soir", le jour où enfin les qualités de Linux seraient reconnues comme un "vrai OS" méritant les mêmes honneurs que Mac OSX ou Windows.

Pendant longtemps les utilisateurs de Windows sont restés eux imperturbables aux sirènes de Linux, raillant même cette éternelle promesse de simplicité et de succès. De même les linuxiens passaient leur temps à se moquer de Windows et de ses virus alors que Linux, lui, étaient "inviolable" alors qu'il ne s'agit que d'un problème de place sur le marché, l'absence de virus pour un OS étant plus la preuve de sa diffusion confidentielle que de sa robustesse…

Bref, tout le monde a bien rigolé pendant longtemps. Mais ça, c'était avant…

Ces temps là sont révolus. Linux n'est pas arrivé par la grande porte sous la forme d'une Red Hat ou d'un Ubuntu ou d'une autre de ses milles variantes pour PC, non, Linux est aujourd'hui entre les mains de la terre entière dans la majorité des unités mobiles (smartphones et tablettes) sous un pseudonyme : **Android**.
Quant à l'inviolabilité de Linux, le nombre grandissant d'utilisateurs et d'applications a prouvé qu'aucun OS n'était inviolable et que même Linux ou Android nécessitent d'avoir des pare-feux et des antivirus comme sous Windows.

1 partout, la balle au centre.

Une fois les mythes oubliés et les bagarres de tranchées remisées au grenier dans la malle à souvenirs, que reste-t-il ?

Un OS aussi digne que les autres, aussi imparfait que Windows ou Mac OSX, mais pas moins valeureux.

Un OS capable de s'adapter aux petites machines peu puissantes qu'ont été les premiers smartphones comme aux monstres octo-cœurs tels le Galaxy S4.

> ***Ni mieux ni moins bien qu'un autre OS, Android est aujourd'hui la preuve que le "grand soir" linuxien n'était pas un rêve fou. Ne reste plus qu'à annoncer au grand public qu'en fait ils sont des utilisateurs heureux de Linux…***

Cet OS il faut apprendre à le connaitre pour développer des applications efficaces. D'autant qu'Android n'est pas un Linux « pur » mais qu'il est doté d'une couche d'exécution Java pour le programmer. C'est un OS d'unité mobile donc présentant des similarités avec iOS ou Windows Phone notamment dans le cycle de vie des applications. Les nuances se trouvent dans le jargon et quelques APIs et plus particulièrement dans la gestion de l'affichage ou le système descriptif en XML d'Android bien que ressemblant à XAML est très différent dans sa nature.

Vecteur ou bitmap ?

Le vectoriel c'est le top. Une définition courte faite de quelques points et de courbes de Béziers et vous avez un joli dessin qui s'adapte à toutes les résolutions. XAML est un système unique, la création la plus belle de Microsoft, un truc que j'adore plus que tout. Mais XAML est vraiment unique au sens de "isolé" aussi… Les systèmes vectoriels posent le problème de déplacer la complexité non pas dans le stockage

(qui peut être énorme pour les bitmaps) mais dans le rendu qui réclame une grande puissance de calcul.

C'est pourquoi peu de sociétés ont fait le choix de créer une gestion d'UI vectorielle et encore moins pour de petites machines peu puissantes sauf Microsoft. Et lorsqu'on voit la fluidité de WPF ou même pour rester comparable celle de Silverlight sur Windows Phone 7 ou de WinRT sous Windows Phone 8 on peut jauger objectivement de la grande qualité des produits Microsoft, aucun OS connu ne sait gérer du vectoriel temps réel sur de si petites machines.

Donc la logique c'est que lorsqu'on ne dispose pas d'assez de puissance de calcul, ce qui est le cas sur un smartphone (surtout les premiers modèles) on a plutôt intérêt à se baser sur du bitmap. Mais d'un autre côté les bitmaps mangent beaucoup de place… Toutefois les premiers modèles avaient une résolution tellement faible, donc une taille d'image si petite, que cela ne posait pas de problème (les mémoires flash étaient déjà capables de stocker plusieurs giga). L'adoption d'un format compressé comme PNG peut aussi aider à diminuer l'impact du choix d'un système bitmap.

Suivant ce raisonnement implacable, Google a choisi de mettre en place une gestion d'UI non vectorielle.

Cela ne veut pas dire qu'on ne peut pas dessiner sous Android, cela signifie juste que les templates, les styles XAML, les courbes de Béziers si faciles à créer sous Blend cela n'existe pas. Tout ce qui ressemble à un dessin, sauf quelques formes de base, sont des images - PNG la plupart du temps.

La mise en page sous Android suit ainsi une logique hybride se situant entre XAML (format XML, existence des styles, conteneurs de type stackPanel, etc) et HTML (pour avoir un bouton de forme bizarre, il faut fournir une image de fond pour chaque état ayant cette forme bizarre).

Une fois qu'on a compris cette différence on a compris l'essentiel des différences entre Windows Phone et Android et j'exagère à peine.

Langage

Bon, reste une autre différence, chez Google on a choisi d'utiliser une "sorte de java" comme langage de développement. Je dis bien une "sorte de java" car en réalité un procès intenté par Oracle, gagné en première instance par Google mais relancé en appel par Oracle dernièrement tourne autour de ce fameux "Java". L'affaire est

complexe, pleine de rebondissement et n'a pas grand intérêt pour nous ici, sauf pour la petite histoire.

Dans la même situation (ou presque) Microsoft avait revu sa copie et s'était "vengé" en sortant C#, Google semble être plus têtu !

Le Java de Android fonctionne grâce à une machine virtuelle : Java Dalvik à ne pas confondre avec les Daleks ennemis jurés de Dr Who. Le code s'écrit et se teste en utilisant Eclipse. Un classique des développeurs Java. Google propose aussi son propre EDI, et JetBrains, la société ayant créée Resharper pour Visual Studio propose aussi un EDI très intéressant.
Côté langage nous disposons donc d'un standard de type Java et d'un choix d'EDI pas si mauvais, même si nous qui connaissons Visual Studio aurons naturellement une préférence pour ce dernier.

Le plus amusant dans tout ça ?

C'est qu'on s'en fiche !

On s'en fiche complètement car nous nous allons utiliser C# sur Android … Grâce à Xamarin 2.0 (anciennement MonoDroid et MonoTouch) qui nous offre les bienfaits à la fois de ce langage simple et puissant mais aussi les joies du framework .NET puisque Xamarin est basé sur le projet MONO…

On peut travailler de deux façons : soit par le biais de Xamarin Studio, une copie sous Mono de Visual Studio tout à fait correcte avec designer visuel et tout ce qu'il faut pour écrire, déboguer et tester une appli, soit par le biais de Visual Studio car Xamarin installe aussi un puglin VS. Dans ce dernier cas on bénéficie de la puissance et du confort de cet environnement (avec des plugins comme Resharper par exemple). Xamarin Studio a un avantage énorme c'est qu'étant en Mono il est cross-plateforme. On peut donc développer sur une machine Mac ou Linux et pas seulement sur PC.

Pour ce qui nous préoccupe, à savoir le développement cross-plateforme impliquant du code Microsoft, l'option Visual Studio n'est pas juste une question de confort vous l'aurez certainement compris : En utilisant le C# Xamarin dans Visual Studio nous allons pouvoir créer des Solutions VS qui contiennent à la fois des applications Android et des applications Windows ! Et tout cela va se compiler gentiment sans souci en partageant du code...

On pourra pour simplifier encore plus le partage de code agrémenter ce “montage” de la librairie “MvvmCross” dont j’ai présenté les principales caractéristiques dans un série de trois longs billets (plus 12 vidéos de 8h au total) à laquelle je renvoie le lecteur intéressé. En gros, MvvmCross est une solution MVVM un peu comme Mvvm Light ou Jounce ou Prism, mais dont le principal avantage est d’être cross-plateforme pour pouvoir partager le même code (sans recopie) entre des applications pour PC, Android et iOS.

Je ne reviendrais pas sur ces détails déjà traités ou que je traiterais plus en profondeur plus tard mais on comprend mieux maintenant où se trouve le cross-plateforme dans tout cela et la place que tient que chaque outil dans cet environnement.

Pour résumer

Android est un OS tout à fait honorable, construit sur Linux, proposant un système d’affichage bitmap plutôt que vectoriel et un langage Java.

Mais pour ce qui nous concerne, nous utiliserons Android au travers de Xamarin, c’est à dire en C# et avec le support de .NET ce qui va nous faire gagner beaucoup de temps en formation… Le tout en se reposant sur la puissance de Visual Studio et de ses éventuels plugins comme ReSharper.

Grâce à MvvmCross nous allons pouvoir fédérer le code le plus important, celui des ViewModels, dans des librairies de code portables Visual Studio. Seules les interfaces (UI) seront spécifiques aux cibles (Android, WPF, WinRT, etc). MvvmCross nous apportera une aide supplémentaire : le support du binding dans les UI Android avec une syntaxe proche de celle de XAML (En JSon dans les premières versions et désormais avec une syntaxe très simple à telle point qu’elle peut s’utiliser en XAML avec bénéfice !).

Les versions d’Android

Android évolue sans cesse. Certaines nouveautés n’étant pas forcément compatibles avec les anciens matériels et les gens conservant malgré tout leur téléphone ou leur tablette un petit moment, il y a foisonnement de versions en activité sur le marché. Mais ce foisonnement est plutôt de l’ordre du mythe entretenu par la concurrence. Dans la réalité les versions récentes sont maintenant majoritaires. Au-delà, Android étant un OS libre et ouvert, chaque fabricant de mobile peut lui ajouter sa propre “couche” maison pour offrir telle ou telle fonction spécifique permettant de se démarquer de la concurrence (Presque tous le font, d’où l’intérêt pour les puristes de la gamme Nexus proposant un Android « nu »).

C'est d'ailleurs ce qui a fait en partie le succès d'Android. Windows Phone ou iOS sont rigides, identiques partout. Chez Apple, vu qu'il n'existe qu'un modèle, cela se ne voit pas trop. Pour Microsoft je pense que ce détail n'a pas aidé à l'adoption massive de Windows Phone. Tous les téléphones sous Windows Phone avaient tendance à se ressembler fortement, trop, tous offraient les mêmes possibilités. Personne n'aime acheter un objet qui dès le départ se noie dans la masse de l'uniformisation. Chacun se sent différent et veut l'afficher à la face du monde. Pas seulement les utilisateurs, les constructeurs aussi veulent se différencier des concurrents et même les distributeurs comme Orange veulent leur petite "couche" à eux qui fait la différence avec Free ou SFR. Si Android a connu le succès c'est aussi grâce à sa souplesse face à la rigidité d'iOS et l'inflexibilité de Windows Phone qui ont déplu aux constructeurs et aux distributeurs.

C'est pourquoi je me garderai bien comme certains de fustiger Android et ses différentes versions en circulation. On entend souvent cette critique, mais elle est au contraire l'une des clés du succès… D'autant que cela est un faux débat : une application fonctionnera sur tous les téléphones Android pour peu qu'elle vise une version suffisamment basse et largement répandue, ce qui est le cas en pratique des 800.000 applications du Play Store.

Aujourd'hui il est naturel d'utiliser Android 2.2 ou 2.3 comme base pour développer bien que nous en soyons déjà à la version 4.x. Cela ne gène pas vraiment les applications et comme je le disais plus haut, cette « fragmentation » est en passe de disparaître au profit des versions récentes de l'OS. D'ailleurs en tant que MVP je me garderais bien de rigoler de cette fragmentation d'Android alors qu'en balayant devant ma porte je suis obligé de concevoir que la cassure nette et totale de compatibilité des matériels entre Windows Phone 7.5 et 8.0 est autrement plus grave qu'une petite différence dans les API's des versions d'Android en circulation…

Les améliorations des constructeurs eux-mêmes ne posent aucun problème. Par exemple pendant que Microsoft essayait (et tente toujours) de transformer les PC avec écran de plus en plus large en grosses tablettes full-screen ce qui a peu de sens, Samsung ou LG sur leurs smartphones de plus en plus grands et sur leurs tablettes offrent désormais le multi-fenêtrage pour avoir deux applications fonctionnelle en même temps, voire des fenêtres qui se recouvrent (par exemple la calculette sur un S4)… Ironie du sort, quasi paradoxe qui confirme certains choix hasardeux de Microsoft à contre courant des besoins des utilisateurs et qui explique certainement l'adoption plus que lente de WinRT et Windows 8 même sur les PC de bureau… Les couches "constructeur" n'ont pas d'impact réel sur les applications qu'on peut écrire pour Android. Elles ne font qu'apporter un menu différent ou le support du stylet par exemple. Si on ne prend pas en compte la couche Samsung ou LG pour le multi-fenêtrage l'application tournera full-screen comme la grande majorité des autres,

c'est tout. Si on vise une clientèle spéciale uniquement équipée de certains modèles comme le Galaxy S3 ou S4, on prendra en charge la fonction et on acceptera de limiter l'audience de l'application, c'est un choix assumé sans impact pour les applications "standard".

Ici encore, pour résumer, disons qu'il existe effectivement de nombreuses versions d'Android en circulation, voir même par dessus des couches constructeurs (ce qui multiplie les combinaisons) mais que cela est un élément de la réussite d'Android et que le développeur est en réalité fort peu gêné par tout cela sauf les hyper geeks qui veulent absolument utiliser les derniers gadgets (ce qui ne fait pas forcément une bonne application pour autant).

Pour le lecteur intéressé je conseille la lecture de l'historique des versions Android sur Wikipédia.

Et pour la petite histoire sachez que les versions portent toutes des noms de sucreries en suivant l'ordre alphabétique (CupCake, Donut, Eclair, Froyo, GinderBread, etc). La prochaine à venir s'appelant Kit Kat par le biais d'un accord commercial entre cet emblême de la junk food et Google. Les puristes du monde Android sont très réservés sur cette fantaisie d'ailleurs.

Il faut aussi savoir que les versions d'Android, en dehors de leur nom gourmand sont assorties d'un numéro de version de l'API qui est différent… Cela trouble un peu au départ.

Ainsi la version Gingerbread (pain d'épice) est la version 2.3, mais son API est de niveau 9. On retrouve d'ailleurs une 2.3.3 (jusqu'à la 2.3.7) toujours sous l'appellation Gingerbread 2.3 mais en API de niveau 10…

Lorsqu'on développe une application on doit choisir le niveau d'API (ce qui semble logique) ce qui n'a donc rien à voir avec la "version" d'Android ou son nom de version.

En choisissant un niveau 10 par exemple, choix assez standard à l'heure actuelle, cela signifie qu'on visera au minimum Gingerbread 2.3.3 et que l'application tournera sur toute machine équipée de cette version ou d'une suivante (2.3.5 par exemple ou 4.0). C'est ainsi qu'une application peut couvrir jusqu'à 90% du marché Android sans avoir à gérer la fameuse fragmentation évoquée plus haut, au prix, cela est évident, de s'abstenir d'utiliser des API's trop récentes (ce qui gêne très peu généralement).

Conclusion

Cette partie 2 est courte et finit de brosser le décor. Nul besoin de décortiquer l'OS, ces APIs ou de rester des heures sur Java ou Eclipse que nous n'utiliserons pas.

En revanche comprendre la nature de Android, sa provenance (Linux), ses principales différences avec notamment Windows Phone, la façon dont son nommées les versions, tout cela est pratique et sera utile à un moment ou un autre. Développer chaque point n'aurait guère d'intérêt, le Web ainsi que les librairies sont remplies de documentations et de livres pour les lecteurs qui souhaiteraient entrer dans les détails. Dans le cadre de la démarche très orientée pratique que je vous propose, tout cela est accessoire (intéressant mais pas indispensable à creuser). Mieux vaut faire court pour entrer le plus vite possible dans le vif du sujet.

Ce moment va vite arriver en réalité puisque la partie 3 va aborder le cycle de vie d'une application Android…

Cross-plateforme : Android – Part 3: Activité et cycle de vie

Les parties 1 et 2 ont planté le décor : pourquoi utiliser Android dans une solution cross-plateforme et qu'est-ce que Android. Aujourd'hui nous entrons dans le vif du sujet : la notion d'activité et son cycle de vie.

Android et les Activités (activity)

Les activités sont les éléments fondamentaux des applications Android. Elles existent sous différents états, de leur création à leur fin. Une application Android peut contenir une seule activité ou bien plusieurs. Chaque Activité peut être un point d'entrée et peut se concevoir comme une « mini application » autonome. Quand une application possède plusieurs Activités, l'une d'elle est marquée comme étant la page d'entrée par défaut.

Une activité est un concept simple ciblé sur ce que l'utilisateur peut faire, d'où son nom. De base toutes les activités interagissent donc avec l'utilisateur (il y a des exceptions) et c'est la classe `Activity` qui s'occupe de créer une fenêtre pour le que le développeur puisse y charger l'UI relative à l'activité en question.

Le plus souvent les activités sont présentées à l'utilisateur en mode plein écran, mais elles peuvent aussi apparaitre sous la forme de fenêtres flottantes ou même être incorporées à l'intérieur d'autres activités. De même une Activité à le droit de recharger son interface sans pour autant être obligée d'appeler une autre Activité.

Chaque activité est autonome et possède donc une partie code et une partie visuelle.

Le cycle de vie d'une activité commence avec son instanciation et se termine par sa destruction. Entre les deux il existe de nombreux états intermédiaires. Lorsque l'activité change d'état la méthode d'évènement du cycle de vie appropriée est appelée pour avertir l'Activité de la modification imminente de son état et lui permettre d'exécuter éventuellement du code pour s'adapter à ces changements.

On retrouve là un mécanisme propre à tous les OS mobiles comme iOS ou Windows Phone (voire même WinRT sur PC). Ce mécanisme s'est imposé partout car il permet de gérer de nombreuses applications de façon fluide sans pour autant consommer trop de puissance ni trop de mémoire. Ainsi une application qui n'a plus l'avant-plan se trouve-t-elle écartée par l'OS qui la place dans une sorte de coma. L'application a eu le temps de sauvegarder son état et lorsque l'application est rappelée par l'utilisateur l'OS la sort de ce coma et lui offre la possibilité de se "réhydrater" : l'utilisateur croit reprendre le cours de ce qu'il avait arrêté, l'UX est donc de bonne qualité, mais entre temps, pendant son "coma", l'application n'a pas consommé de ressources, ce qui ménage l'unité mobile. J'utilise ici le terme de "coma" en écho à un terme américain utilisé pour décrire cette mise en sommeil, le "tombstoning", littéralement "pierre tombale – isation".
De même je rappelle que lorsque je parle de "mobiles" je parle "d'unités mobiles" et que cela intègre aussi bien les smartphones que les tablettes et phablettes.

Le concept

Les activités sont un concept de programmation inhabituel totalement spécifique à Android mais simple à comprendre.

Dans le développement d'applications traditionnelles il y a généralement une méthode statique “main” qui est exécutée pour lancer l'application. C’est le cas en mode console sous Windows par exemple, ou même dans une application WPF.

Avec Android, cependant, les choses sont différentes; les applications Android peuvent être lancées via n'importe quelle activité enregistrée au sein d'une application, chacune jouant le rôle d’un point d’entrée “main” en quelque sorte.

Dans la pratique la plupart des applications n'ont qu'une seule activité spécifique qui est défini comme le point d'entrée unique de l'application. Toutefois, si une application se bloque ou est terminée par l'OS ce dernier peut essayer de la redémarrer sur la dernière activité ouverte ou n'importe où ailleurs dans la pile de l'activité précédente.
En outre, les activités peuvent être mises en pause par l’OS quand elles ne sont pas actives voire même être supprimées totalement de la mémoire si cette dernière vient à manquer.

Une attention particulière doit être portée pour permettre à l'application de restaurer correctement son état dans le cas où elle est redémarrée et dans le cas où son fonctionnement repose sur des données provenant des activités précédentes.

Le cycle de vie d’une activité est représenté par un ensemble de méthodes que le système d'exploitation appelle tout au long de ce cycle pour tenir l’Activité au courant de son état à venir. Ces méthodes permettent au développeur d'implémenter les fonctionnalités nécessaires pour satisfaire aux exigences de gestion de l'état et des ressources de son application.

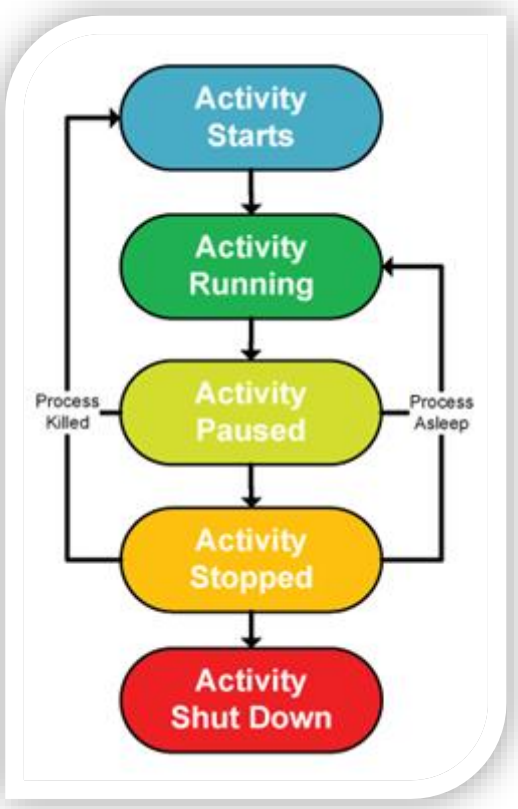
Il est extrêmement important pour le développeur de l'application d’analyser les besoins de chaque activité pour déterminer les méthodes exposées par la gestion du cycle de vie qui doivent être prises en charge. Ne pas le faire peut entrainer une instabilité de l'application, des bogues, des problèmes avec les ressources et peut-être même dans certains cas avoir une incidence sur la stabilité de l’OS (c’est un cas extrême puisque Android utilise une VM java qui ressemble au C# managé, ce procédé garantissant une bonne stabilité de l’OS même en cas de défaillance grave d’une application).

La gestion du cycle de vie

La gestion du cycle de vie des applications d'Android se matérialise pour le développeur par un ensemble de méthodes exposées par la classe `Activity`, ses méthodes fournissent un cadre de gestion des ressources. La bonne gestion du cycle de vie est l'ossature d'une application Android (comme sous iOS ou Windows Phone), c'est une colonne vertébrale d'où partent les "côtes" qui vont structurer le squelette et en assurer la cohérence.

Les différents états

Le système d'exploitation Android utilise une file de priorité pour aider à gérer les activités exécutées sur la machine. Selon l'état dans lequel se trouve une activité particulière Android lui attribue une certaine priorité au sein de l'OS. Ce système de priorité permet à Android d'identifier les activités qui ne sont plus en cours d'utilisation, ce qui lui de récupérer de la mémoire et des ressources. Le schéma suivant illustre les états par lesquels une activité peut passer au cours de sa durée de vie :



De haut en bas : Démarrage de l'activité, activité en mode de fonctionnement, activité en mode pause, activité stoppée, activitée arrêtée. Légende à gauche : processus détruit. Légende à droite : processus en sommeil.

On retrouve ici un schéma classique sur les OS mobiles qu'on peut résumer à trois groupes principaux :

Actif ou en fonctionnement

Les activités sont considérées comme "actives" ou "en cours de fonctionnement" si elles sont au premier plan qu'on appelle aussi "sommet de la pile d'activité". L'activité au premier plan est celle possédant la plus haute priorité dans la pile des activités gérée par l'OS. Elle conservera se statut particulier sauf si elle est tuée par l'OS dans des circonstances bien précises et extrêmes : si l'application tente de consommer plus de mémoire que n'en possède l'appareil par exemple car cela pourrait provoquer un blocage complet de l'interface utilisateur. Une bonne UX est parfois obligée de reposer sur des choix douloureux de type "peste ou choléra". Dans ce cas précis se trouver face à une application qui "plante" n'est pas une bonne UX mais elle est "moins pire" que de se retrouver devant une machine qui ne répond plus du tout, Android prend donc la décision de rester en vie au détriment de l'application irrespectueuse des ressources.

Mode Pause

Lorsque l'unité mobile se met en veille ou qu'une activité est encore visible mais partiellement cachée par une nouvelle activité non "full size" ou transparente, l'activité est alors considérée (et mise) en mode pause.

Lorsqu'elles sont en pause les activités sont encore "en vie". Cela signifie qu'elles conservent toutes leurs informations en mémoire et qu'elles restent attachées au gestionnaire de fenêtre. Techniquement une application en pause est considérée comme la deuxième activité prioritaire sur la pile des activités et à ce titre ne sera tuée par l'OS que si l'activité active (premier plan) réclame plus de ressources et qu'il n'y a plus d'autres moyens de lui en fournir pour qu'elle reste stable et réactive.

Mode Arrêt

Les activités qui sont complètement masquées par d'autres sont considérées comme arrêtées. Cet arrêt est soit total soit partiel si l'activité en question produit un travail d'arrière-plan. Android tente de maintenir autant qu'il le peut les informations et ressources des applications arrêtées, mais ce mode est celui qui possède la plus basse priorité dans la pile des activités de l'OS. A ce titre elles feront partie des premières activités tuées par l'OS dès qu'un besoin de ressources se fera sentir et que celles du système seront épuisées.

Effet des touches de navigation

On retrouve sous Android deux boutons importants : l'accueil et la navigation arrière (le troisième appelle les paramètres). Quel est l'effet de l'appui sur ces boutons vis-à-vis du cycle de vie des applications ?

La logique est simple : si la touche de navigation arrière est utilisée Android considère que l'activité qui vient d'être quittée n'est plus en premier plan. Ce qui est une lapalissade. De ce fait, et en considérant ce que nous avons vu plus haut, elle passe à un niveau de priorité faible et peut être tuée à tout moment pour récupérer les ressources qu'elle utilise.

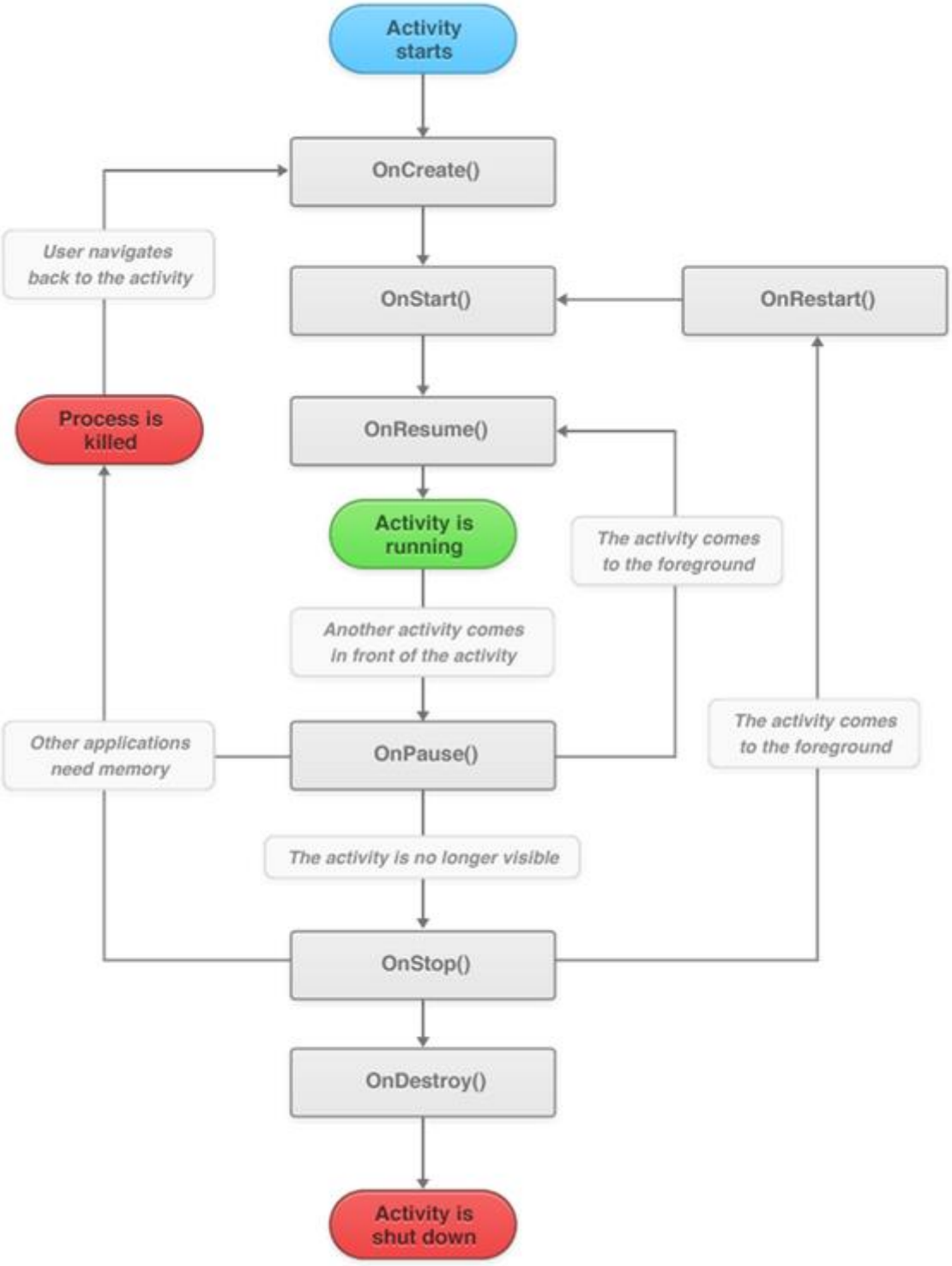
Pour mettre en œuvre le multitâche sous Android il faut ainsi utiliser la touche d'accueil qui permet par exemple de basculer entre les applications qui occupent virtuellement le premier plan. Quitter une application par la navigation arrière la condamne à courte échéance, quitter une application par la touche accueil la fait passer en mode arrière-plan. Les priorités ne sont pas les mêmes, les ressources d'une application en arrière-plan sont conservées plus longtemps ce qui offre une meilleure réactivité côté utilisateur.

Les différentes méthodes du cycle de vie

Le SDK de Android (et par extension le framework fournit par Xamarin) fournit un modèle complet pour gérer l'état des activités au sein d'une application. Lorsque l'état de l'activité est sur le point de changer celle-ci est notifiée par l'OS et les méthodes associées à ce changement dans la classe `Activity` sont alors appelées.

> Note: On remarquera qu'Android gère le cycle de vie avec une granularité très fine puisque c'est au niveau de chaque activité que le mécanisme s'applique. Une application peut avoir plusieurs activités et chacune peut donc se trouver dans un état particulier qui n'est pas global à l'application. Si les applications simples ne comportent généralement qu'une seule activité, ce n'est pas le cas des applications plus étoffées. Ces dernières profitent alors de ce mode de gestion des ressources particulièrement économe.

Voici le schéma qui illustre les différents états et leurs transitions entrainant les appels aux méthodes de la classe Activity dans le cadre de la gestion du cycle de vie de chaque activité :



La gestion de ces méthodes s'effectue tout simplement par surcharge. Toutes ne sont pas à prendre en compte dans chaque activité, comme expliqué plus haut c'est au développeur d'analyser les besoins de son application et de décider pour chaque activité de la pertinence de la prise en compte des états.

OnCreate

C'est la première méthode de gestion du cycle de vie d'une activité. Elle est appelée lorsque cette dernière est créée par l'OS. L'application doit au minimum surcharger cette méthode pour charge sa Vue principale. S'il y a d'autres besoins d'initialisations c'est bien entendu aussi à cet endroit qu'elles prendront place. L'OS ne charge pas automatique l'UI associée à une Activité comme le font Silverligth, WPF ou même Windows Forms. L'Activité à la responsabilité de charger le fichier XML de son UI quand bon lui semble, elle peut même en changer à sa guise (ce qui n'est qu'une possibilité rarement utilisée). C'est en tout cas une spécificité à retenir car dans le monde Microsoft aucune technologie ne fonctionne comme cela, pas même ASP.NET qui est pourtant très particulier comparé aux applications natives Win32/Win64.

Le paramètre passé à la méthode est de type "`Bundle`". C'est un dictionnaire qui permet de stocker et de passer des informations entre les activités. En programmation Windows on appellerait plutôt cela un "contexte". Si le paquet `Bundle` est non nul cela indique que l'activité est reprise depuis un état antérieur et qu'elle peut être ré-hydratée.

L'application peut aussi utiliser le conteneur "`Extras`" de l'objet "`Intent`" pour restaurer des données précédemment enregistrées ou des données passées entre les activités. Un `Intent` peut être assimilé en première analyse à un message système permettant de lancer une nouvelle application, d'appeler une nouvelle Activité etc.

L'extrait de code ci-dessous montre comment l'activité détecte une reprise pour se ré-hydrater via le `Bundle` mais aussi comment elle accède aux `Extras` de l'`Intent` pour récupérer d'autres données.

> Ce code est un exemple en Xamarin puisque nous n'aborderons la programmation Android qu'au travers de C# et via cet outil.

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    string intentString;
    bool intentBool;
    string extraString;
    bool extraBool;
    if (bundle != null)
    {
        //Si l'activité est reprise (resume)
        intentString = bundle.GetString("myString");
        intentBool = bundle.GetBoolean("myBool");
```

```
    }

    // Accès aux valeurs dans le Intent Extras
    extraString = Intent.GetStringExtra("myString");
    extraBool = Intent.GetBooleanExtra("myBool", false);
    // Initialisation de la vue "main" depuis les ressources de layout
    SetContentView(Resource.Layout.Main);
}
```

OnStart

Cette méthode est appelée lorsque l'activité est sur le point de devenir visible à l'utilisateur. Les activités doivent remplacer cette méthode si elles ont besoin d'effectuer des tâches spécifiques juste avant l'affichage, tels que: le rafraichissement des valeurs des vues au sein de l'activité, ou faire du ramping up de la vitesse des frames (une tâche habituelle dans la construction d'un jeu ou le nombre de FPS doit être calibré finement pour assurer la jouabilité).

OnPause

Cette méthode est appelée lorsque le système est sur le point de mettre l'activité au second plan. Les activités doivent surcharger cette méthode si elles ont besoin de valider les modifications non sauvegardées des données persistantes, de détruire ou nettoyer d'autres objets qui consomment des ressources ou qui abaissent les FPS, etc.

L'implémentation de cette méthode doit retourner le plus rapidement possible car aucune activité ne sera reprise jusqu'à ce que cette méthode retourne. L'exemple suivant illustre comment surcharger la méthode OnPause pour sauvegarder les données dans le conteneur `Extras` afin qu'elles puissent être transmises entre les activités :

```
protected override void OnPause()
{
    Intent.PutExtra("myString", "Xamarin.Android OnPause exécuté !");
    Intent.PutExtra("myBool", true);
    base.OnPause();
}
```

Windows Phone et WinRT en général imposent des limites de temps très courtes aux actions de ce type, Android est plus souple mais cela ne veut pas dire qu'il ne faut pas s'en préoccuper ! Il peut ainsi s'avérer nécessaire de mettre en place une stratégie de sauvegarde des données plus sophistiquées dans le cas où ces données sont

importantes et peuvent prendre du temps à écrire sur “disque”, par exemple en enregistrant les données modifiées au fur et à mesure de leur saisie par l’utilisateur. On appelle ce procédé « sauvegarde incrémentale » et il est vivement conseillé aussi bien sur Android que sous Windows Phone ou WinRT d’ailleurs. En adoptant ce type de sauvegarde « au fur et à mesure » le travail à fournir dans `OnPause` sera très court, voire nul.

OnResume

Cette méthode est appelée lorsque l'activité commencera à interagir avec l'utilisateur juste après avoir été dans un état de pause. Lorsque cette méthode est appelée l'activité se déplace vers le haut de la pile d'activité et elle reçoit alors les entrées de l'utilisateur. Les activités peuvent surcharger cette méthode si elles ont besoin d’exécuter des tâches après que l'activité commence à accepter les entrées utilisateur.
Android offre plus de souplesse que Windows Phone dans le cycle de vie, cela signifie aussi qu’il faut faire plus attention aux méthodes qu’on choisit pour exécuter tel ou tel code !

OnStop

Cette méthode est appelée lorsque l'activité n'est plus visible pour l'utilisateur car une autre activité a été reprise ou commencée et qu’elle recouvre visuellement celle-ci. Cela peut se produire parce que l'activité est terminée (la méthode `Finish` a été appelée) ou parce que le système est en train de détruire cette instance de l'activité pour économiser des ressources, soit parce qu'un changement d'orientation a eu lieu pour le périphérique. Ces conditions sont très différentes ! Vous pouvez les distinguer en utilisant la propriété `IsFinishing`.
Une activité doit surcharger cette méthode si elle a besoin d'effectuer des tâches spécifiques avant d'être détruite ou si elle est sur le point de lancer la construction d’une nouvelle interface utilisateur après un changement d'orientation.

OnRestart

Cette méthode est appelée après que l’activité a été arrêtée et avant d'être relancée. Cette méthode est toujours suivie par `OnStart`. Une activité doit surcharger `OnRestart` si elle a besoin d'exécuter des tâches immédiatement avant `OnStart`. Par exemple si l'activité a déjà été envoyée en arrière-plan et que `OnStop` a été appelé, mais que le processus de l'activité n'a pas encore été détruit par le système d'exploitation. Dans ce cas la méthode `OnRestart` doit être neutralisée.

Un bon exemple de ce cas de figure est quand l'utilisateur appuie sur le bouton Accueil alors qu’il utilise l'application. `OnPause` puis la méthode `OnStop` sont appelés mais l'activité n'est pas détruite. Si l'utilisateur veut restaurer l'application en utilisant le gestionnaire de tâches (ou une méthode similaire) la méthode `OnRestart` de

l'activité sera appelée par le système d'exploitation lors de la réactivation des activités.

OnDestroy

C'est la dernière méthode qui est appelée sur une activité avant qu'elle ne soit détruite. Après cette méthode l'activité sera tuée et purgée des pools de ressources de l'appareil. Le système d'exploitation va détruire définitivement les données d'état d'une activité après l'exécution de cette méthode. `OnDestroy` est un peu le dernier bar avant le désert… si une activité doit sauvegarder des données c'est le dernier emplacement pour le faire.

OnSaveInstanceState

Cette méthode est fournie par le gestionnaire de cycle de vie Android pour donner à une activité la possibilité de sauvegarder ses données, par exemple sur un changement d'orientation de l'écran (car ce changement entraîne la mort de l'Activité en cours et sa recréation). Le code suivant illustre comment les activités peuvent surcharger la méthode `OnSaveInstanceState` pour enregistrer les données en vue d'une réhydratation lorsque la méthode `OnCreate` sera appelée :

```
protected override void OnSaveInstanceState(Bundle outState)
{
    outState.PutString("myString", Xamarin.Android OnSaveInstanceState");
    outState.PutBoolean("myBool", true);
    base.OnSaveInstanceState(outState);
}
```

Conclusion

La gestion des activités et de leur cycle de vie est un point essentiel à maitriser pour développer sur Android. Le reste n'est que du code classique ou de l'affichage comme on peut en faire dans des tas d'OS. Il reste beaucoup de choses à connaitre pour être un expert Android, c'est vrai, mais une telle expertise commence par la compréhension totale et complète du cycle de vie.

Le "tombstonig" qui implique la sauvegarde et la réhydratation des applications, la gestion particulière sous Android d'être prévenu d'un changement d'orientation de l'appareil, la notion même d'activité sont des concepts de base qui structurent l'écriture du code, qui dictent ce qui est possible de faire et comment le réaliser. Puisque nous utilisons C#, le reste n'est que du code tel qu'on peut en écrire pour n'importe quelle application. Toutefois cela s'entend pour un code Android programmé directement. Nous utiliserons une stratégie cross-plateforme passant par

MvvmCross qui propose ses propres méthodes unifiées pour gérer le cycle de vie d'une application quelle que soit la cible finale (Android, iOS ou une cible Microsoft).

Le deuxième aspect fondamental à comprendre est la gestion de l'interface utilisateur puisque là, hélas, il n'y a pas d'autre solution que d'être natif… Et Android utilise sa propre logique. Nous verrons que malgré tout on retrouve des concepts habituels, qu'ils proviennent de HTML ou de XAML.

Plus on utilise d'OS plus en apprendre de nouveaux est simple car au final on s'aperçoit que tous doivent régler les mêmes problèmes et fournissent sensiblement les mêmes moyens de le faire. La gestion du cycle de vie des activités que nous venons d'appréhender dans ce billet le montre bien pour qui sait comment d'autres OS tel que Windows Phone le font. Les similitudes sont grandes. Toute l'astuce est donc de bien comprendre les différences et leurs impacts sur le code à écrire !

Cross-plateforme : Android – Part 4 – Vues et Rotation

A la suite des trois précédentes parties qui ont exposé le pourquoi du choix d'Android, les grandes lignes de l'OS et la nature du concept d'Activité, tournons notre regard sur les vues et la gestion de la rotation d'écran…

Vues et Rotation

L'optique de ces billets sur Android n'est pas de faire un cours complet sur cet OS, il existe de nombreux livres traitant du sujet, mais bien de se concentrer sur les différences fonctionnelles essentielles avec les OS Microsoft que nous connaissons et avec lesquels nous devront "marier" les applications Android.

- La partie 1 a expliqué pourquoi choisir Android
- La partie 2 a expliqué les grandes lignes de l'OS
- La partie 3 a présenté le concept de base de toute application, l'Activité

Cette partie 4 choisit de présenter la gestion des vue et de la rotation de l'écran car il s'agit d'un point essentiel faisant la différence entre du développement mobile et du développement PC. Sur ces derniers ce concept n'existe pas (la rotation). Et puis parler de rotation, c'est forcément parler des vues alors que l'inverse n'est pas forcément vrai… Et que seraient les applications sans les vues ? Pas grand chose !

Bien gérer le passage d'un mode portrait à un mode paysage et ce dans les deux sens est crucial sur un smartphone et aussi sur une tablette. C'est tout le côté pratique de

ces machines qui s'exprime ici. La rotation est le symbole de l'UX spécifique des unités mobiles en quelque sorte.

Donc d'une part la rotation des écrans est un point essentiel qui différencie le développement pour mobile, et, d'autre part, cela implique de parler des vues. Et quant on a compris ce qu'est une Activité (partie 3) et qu'on a compris ce qu'était une vue et comment gérer sa rotation (partie 4 présente) on a presque tout compris d'Android (je dis bien "presque").

Bien entendu nous nous intéresserons à ce mécanisme sous l'angle de Xamarin, c'est à dire en C# et .NET. Toutefois les vues s'expriment en format natif et n'ont pas réellement de lien avec le langage de programmation. Et tout aussi évidemment, quand je parle de "mobiles" j'entends "unités mobiles", ce qui intègre en un tout aussi bien les smartphones que les tablettes.

Un tour par ci, un tour par là…

Parce que les appareils mobiles sont par leur taille et leur format facilement manipulables dans tous les sens la rotation est une fonction intégrée dans tous les OS mobiles. Android n'y échappe pas.

Android fournit un cadre complet pour gérer la rotation au sein des applications, que l'interface utilisateur soit créée de façon déclarative en XML (nous y reviendrons plus tard) ou par le code. Dans le premier cas un mode déclaratif permet à l'application de bénéficier des automatismes d'Android, dans le second cas des actions par code sont nécessaires. Cela permet un contrôle fin à l'exécution, mais au détriment d'un travail supplémentaire pour le développeur.

Que cela soit par mode déclaratif ou par programmation toutes les applications Android doivent appliquer les mêmes techniques de gestion d'état lorsque l'appareil change d'orientation. L'utilisation de ces techniques pour gérer l'état de l'unité mobile est important parce que quand un appareil Android est en rotation le système va redémarrer l'activité courante. C'est une spécificité de l'OS. Android fait cela pour rendre plus simple le chargement de ressources alternatives telles que, principalement, des mises en page et des images conçues spécifiquement pour une orientation particulière. Quand Android redémarre l'activité celle-ci perd tout état transitoire qu'elle peut avoir stocké dans ses variables, l'instance est totalement renouvelée. Par conséquent si une activité est tributaire de certains états internes, il faut absolument les persister avant le changement d'orientation pour réhydrater la nouvelle instance. L'utilisateur ne doit en aucun cas avoir l'impression de perdre son travail en cours (surtout qu'une rotation peut être involontaire).

En outre, dans certains cas une application peut aussi choisir de se retirer du mécanisme automatique de rotation pour en prendre totalement contrôle par code et gérer elle-même quand elle doit changer d'orientation. Certaines applications de lecture de PDF ou de eBook par exemple permettent d'interdire le changement d'orientation. Quand on lit dans un fauteuil ou dans un lit il est fréquent que l'on bouge l'unité mobile sans pour autant vouloir que l'affichage change d'orientation. C'est un cas pratique où l'application a besoin de contrôler elle-même le changement d'orientation (généralement selon un choix de l'utilisateur).

Gérer les rotations de façon déclarative

Je reviendrais plus tard sur la gestion des ressources dans une application Android et comment s'appliquent les règles de nommage des répertoires, leur rôle, etc. Pour l'instant il vous suffit de savoir qu'Android gère deux types de ressources : les ressources au sens classiques (fichiers `axml` par exemple qui définissent les vues) et les "*dessinables*". Ces derniers représentent tout ce qui est image en général.

Dans un précédent billet de cette série je parlais du mode non vectoriel d'Android par opposition à XAML et des implications de cet état de fait. L'une des principales est que pour garantir la qualité d'affichage des "dessinables" il est nécessaire de les fournir dans un maximum de résolutions différentes, classées dans des répertoires dont les noms suivent des conventions comme "`drawable-mdpi`" ou "`drawable-xhdpi`", le nom du dessinable étant le même (par exemple "`icon.png`"). Les indications dans le nom du répertoire (comme le `-mdpi` ou `-xhdpi` ci-dessus) agissent comme des filtres appliqués par l'OS pour savoir quoi charger en fonction de l'orientation, de la densité en pixel, de la taille de l'écran…

En Xaml un seul contrôle vectoriel peut s'adapter à toutes les résolutions, ici il faudra penser à créer toutes les variantes possibles. WinRT dans une moindre mesure oblige aussi ce genre de gymnastique pour les icones.

Les ressources de mise en page

Par défaut les fichiers définissant les vues sous Android sont placés dans le répertoire `Resources/layout`. Ces fichiers XML (AXML) sont utilisés pour le rendu visuel des Activités. Un fichier de définition de vue est utilisé pour le mode portrait et le mode paysage si aucune mise en page spéciale n'est fournie pour gérer le mode paysage.

Si on regarde la structure d'un projet par défaut fourni par Xamarin on obtient quelque chose de ce type :

Dans ce projet qui définit une seule activité nous trouvons une seule vue dans `Resources/layout` : "`main.xml`" (ces fichiers utilisent aussi l'extension `axml` sous Visual Studio pour permettre le chargement du concepteur visuel d'Android et non pas le concepteur visuel des fichiers XML qui n'est qu'un simple traitement de texte). Quand la méthode `OnCreate` est appelée (voir la gestion du cycle de vie du billet précédent), elle effectue le rendu de la vue définie par "`main.xml`" qui, dans cet exemple, déclare un bouton.

Voici le code AXML de la vue :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<Button
    android:id="@+id/myButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
```

```
    />
</LinearLayout>
```

Les habitués de XAML ne seront pas forcément déroutés : nous avons affaire à fichier suivant la syntaxe XML qui contient des balises, éventuellement imbriquées, définissant les contrôles à afficher et leurs propriétés. Ce que nous nommons par convention en XAML le "`LayoutRoot`" (l'objet racine, généralement une `Grid`) se trouve être ici un contrôle "`LinearLayout`" (sans nom), un conteneur simple (je reviendrais sur les conteneurs ultérieurement) qui ressemble assez à ce qu'est un `StackPanel` en XAML. On voit d'ailleurs dans ses propriétés la définition de son axe de déploiement « `android:orientation="vertical"` ». On note aussi que sa hauteur et sa largeur sont en mode "`fill_parent`" qui correspond au mode "`Stretch`" de XAML. Tout cela est finalement très simple.

On remarque ensuite que cette balise `LinearLayout` imbrique une autre balise, `Button`, et là encore pas grand chose à dire. Comme sous XAML, déclarer une balise d'un type donné créé dans la vue une instance de cette dernière.

Ce qui diffère de XAML se trouve juste dans les détails de la syntaxe finalement (hors vectoriel). La façon de spécifier les propriétés est un peu différente et certaines notations sont plus "linuxienne" et moins "user friendly" qu'en XAML. Disons-le franchement, AXML est bien plus frustre que XAML. Une chose essentielle qui manque cruellement à AXML c'est le binding… Une force de .NET même depuis les Windows Forms. Il est étrange qu'une si bonne idée ne soit pas reprise partout. De base il n'y a donc pas de Binding sous Android. On pourra contourner ce problème en utilisant tout ou partie de la librairie MvvmCross qui propose le Binding dans les fichiers AXML de façon très proche de la syntaxe XAML.

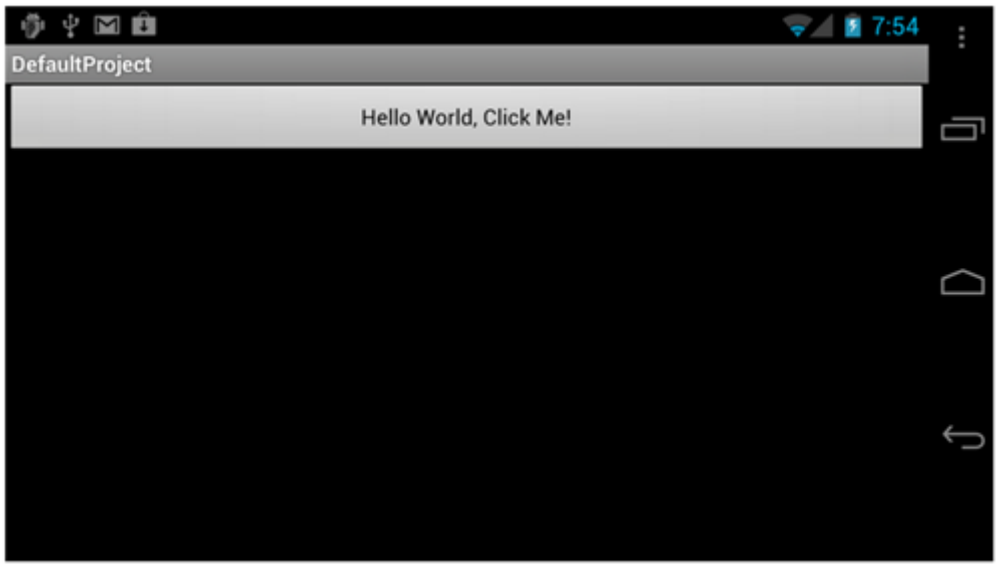
Une autre nuance : chaque contrôle possède un `ID` au lieu de porter un nom en `String`, mais ceux-ci sont des codes entiers peu faciles à manipuler, pour simplifier les choses une table de correspondance est gérée par Xamarin avec d'un côté des noms clairs et de l'autre les codes. Cette table est en fait un fichier "`Resource.Designer.cs`" qui contient des classes déclarant uniquement les noms des objets transformés en nom d'objets entiers auxquels sont affectés automatiquement un `ID` numérique. On peut ainsi nommer les objets comme en XAML, par un nom en string, Xamarin s'occupe de transformer tout cela en `ID` numérique pour Android de façon transparente. La seule contrainte est d'utiliser une syntaxe spéciale pour l'`ID` qui permet de spécifier une `String` au lieu d'un entier (une string elle-même particulière puisque c'est en réalité le nom de la ressource qui se trouvera associé à un entier), on fait ainsi référence à cette table en indiquant "`@+id/myButton`" la première partie explique à Android que nous utilisons

une table des ID plutôt que de fixer un code entier, derrière le slash se trouve le nom en clair "`myButton`".
On notera qu'il ne s'agit pas d'un exotisme de Xamarin, ce dernier se contentant de mimer un comportement rigoureusement identique en Java natif d'Android (seul le nom du fichier de ressource change un peu).

Ce sont ces petites choses qui font les différences parfois déroutantes pour qui vient de XAML dont on (en tout cas moi au moins !) ne dira jamais assez de bien. XAML est la Rolls des langages de définition d'UI. Mais je sais aussi que beaucoup de développeurs ont du mal avec XAML et Blend qu'ils trouvent trop complexes… Et c'est une majorité. Alors finalement, ces développeurs seront peut-être heureux de découvrir AXML !

Mais tout comme XAML, AXML via Xamarin offre un designer visuel qui simplifie grandement la mise en place d'une UI et qui évite d'avoir à taper du code. Même si, comme pour XAML, connaître le langage et savoir le taper directement peut s'avérer important dans certaines situations.

Pour revenir à notre exemple, si l'appareil est mis en mode paysage, comme rien n'est spécifié pour le gérer, l'Activité va être recrée et elle va réinjecter le même AXML pour construire la vue ce qui donnera un affichage de ce genre :



Ce n'est pas très joli, mais cela veut dire que par défaut, si on ne code rien de spécial, ça marche quand même…

Pour un test cela suffit, pour une application "pro" c'est totalement inacceptable bien entendu. Il va falloir travailler un peu !

Les mises en pages différentes par orientation

Le répertoire "layout" utilisé pour stocker la définition AXML de la vue est utilisé par défaut pour stocker les vues utilisées dans les deux orientations. On peut aussi renommer ce répertoire en "layout-port" si on créé aussi un répertoire "layout-land" (*port* pour portrait, et *land* pour landscape = paysage). Il est aussi possible de ne créer que "layout-land" et de laisser le répertoire "layout" sans le renommer. De cette façon une Activité peut définir simplement deux vues chacune spécifique à chaque orientation, Android saura charger celle qu'il faut selon le cas de figure.

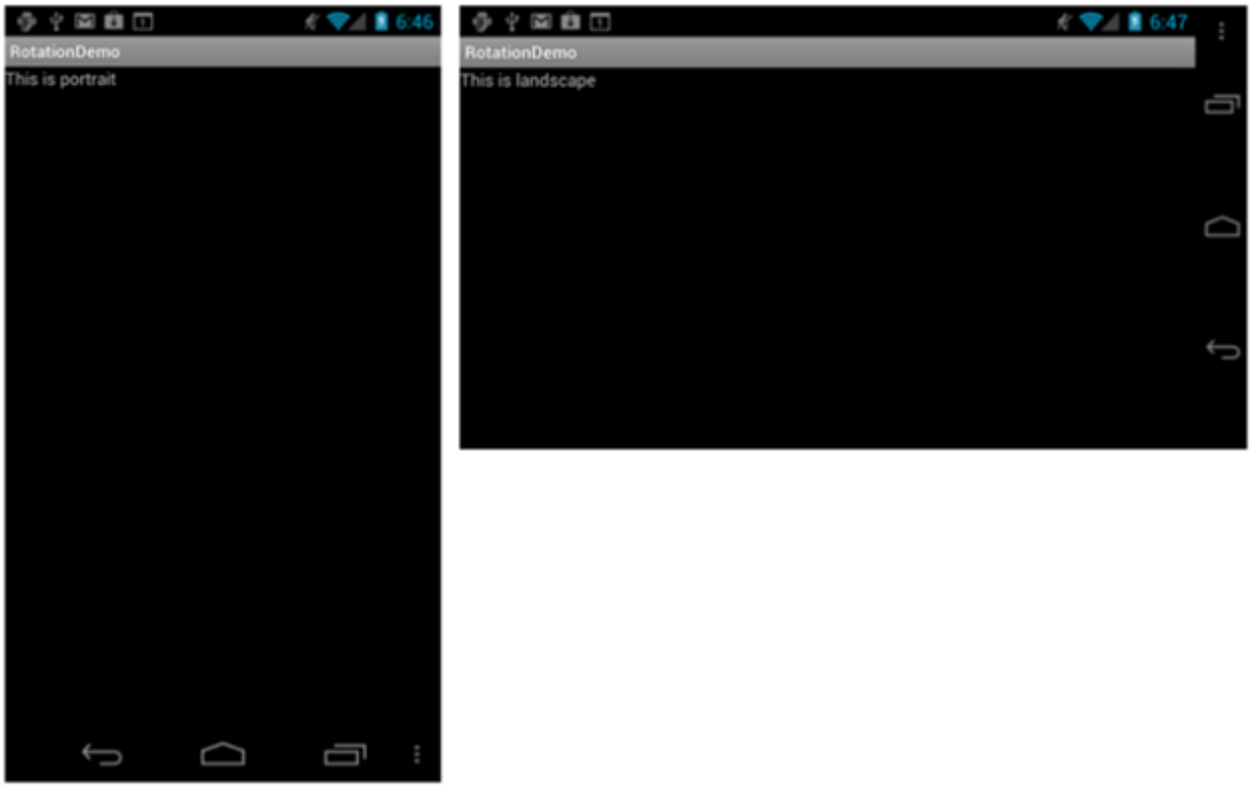
Imaginons qu'une activité définisse l'AXML suivant dans "layout-port" :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:text="This is portrait"
android:layout_height="wrap_content"
android:layout_width="fill_parent" />
</RelativeLayout>
```

Elle peut alors définir cette autre AXML dans "layout-land" :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:text="This is landscape"
android:layout_height="wrap_content"
android:layout_width="fill_parent" />
</RelativeLayout>
```

La seule différence se trouve dans la définition du TextView (un équivalent du TextBlock XAML). Le texte affiché a été modifié dans la seconde vue :



C'est simple, aucun code à gérer, juste des répertoires différents et des AXML adaptés au format… Le reste est automatiquement pris en compte par Android. Ce principe, un peu rustique (puisque basé sur une simple convention de nom de répertoires) est très efficace, facile à comprendre et très souple puisque les vues portrait et paysage peuvent être carrément différentes en tout point ou juste être des copies, ou entre les deux, une modification l'une de l'autre.

Bien gérer les orientations oblige parfois à gérer deux affichages très différents pour obtenir une UX de bonne qualité. C'est plus de travail que sur un PC dont l'écran ne tourne pas mais cela est valable pour Android autant que pour iOS ou Windows Phone…

Les dessinables

Pour tout ce qui n'est pas défini par une vue AXML, c'est à dire les "dessinables" telles que les icones et les images en général, Android qui n'est pas trop embêtant propose exactement la même gestion…

Dans ce cas particuliers les ressources sont puisées dans le répertoire `Resources/drawable` au lieu de `Resources/layout`, c'est tout.

Si on désire afficher une variante du dessinable selon l'orientation on peut utiliser la même convention et créer un répertoire "`drawable-land`" et y placer la version spécifique pour le mode portrait (le nom de fichier restant le même).

Dès que l'orientation changera, l'Activité sera rechargée et Android ira chercher les ressources (mises en page et dessinables) dans les bons sous-répertoires correspondant à l'orientation en cours. Aucun code n'est à écrire pour le développeur une fois encore.

Bien entendu cette simplicité se paye quelque part : on se retrouve avec plusieurs fichiers AXML ou images de même nom mais ayant des contenus différents ! Je trouve personnellement cela très risqué, c'est la porte ouverte à des mélanges non détectables (sauf au runtime). Il faut donc une organisation très stricte côté du Designer ou de l'infographiste et beaucoup d'attention côté développeur / intégrateur… De même là où une seul définition est utilisable dans tous les cas de figure en XAML (vectoriel oblige) il faudra un cycle de conception des images plus complexes sous Android. En général on utilisera Illustrator (ou Inkscape qui est gratuit et très puissant) pour créer des images vectorielles qu'on pourra ensuite exporter à toutes les résolutions désirées. Mais ce n'est qu'une méthode parmi d'autres.

Gérer les vues et les rotations par code

La majorité des applications peuvent se satisfaire des modes automatiques que nous venons de voir. C'est peu contraignant et suffisamment découplé pour des mises en page parfaitement adaptées.

Mais il arrive parfois que le développeur veuille gérer lui-même le changement d'orientation comme nous en avons déjà parlé.

Il existe une autre raison qui peut obliger à gérer l'orientation par code : c'est lorsque la vue elle-même est générée par code au lieu d'être définie dans un fichier AXML. En effet, dans ce dernier cas Android saura retrouver le fichier qui correspond à l'orientation, mais dans le premier cas, puisqu'aucun fichier de définition de vue n'existe, Android et ses automatismes ne peuvent rien pour vous.

Pourquoi définir une vue par code ?

Il y a des vicieux partout… Non, je plaisante. Cela peut s'avérer très intéressant pour adapter une vue très finement aux données gérées par l'application par exemple. On peut imaginer un système de gestion de fiches, les fiches pouvant être de plusieurs

catégories définies par l'utilisateur. Une catégorie de fiche définira les libellés et les zones à saisir. Un tel système est très souple et très puissant, mais dès lors il n'est plus possible à la conception du logiciel de prévoir toutes les vues possibles puisque c'est l'utilisateur qui va les définir dynamiquement par paramétrage… D'autres besoins plus simples font qu'on peut préférer une interface générée par code (par exemple saisir le détail d'une instance de classe quelconque en se basant sur ses propriétés par réflexion).

La création d'une vue par code

On retrouve là encore des principes présents dans XAML qui permet aussi de créer une vue uniquement par code. AXML et XAML permettent d'ailleurs de mixer les deux modes aussi.

Sous Android la décomposition est la suivante :

- Créer un Layout
- Modifier ses paramètres
- Créer des contrôles
- Modifier les paramètres (dont ceux de mise en page) de ces derniers
- Ajouter les contrôles au Layout créé au début
- Initialiser le conteneur de vue en lui passant le Layout

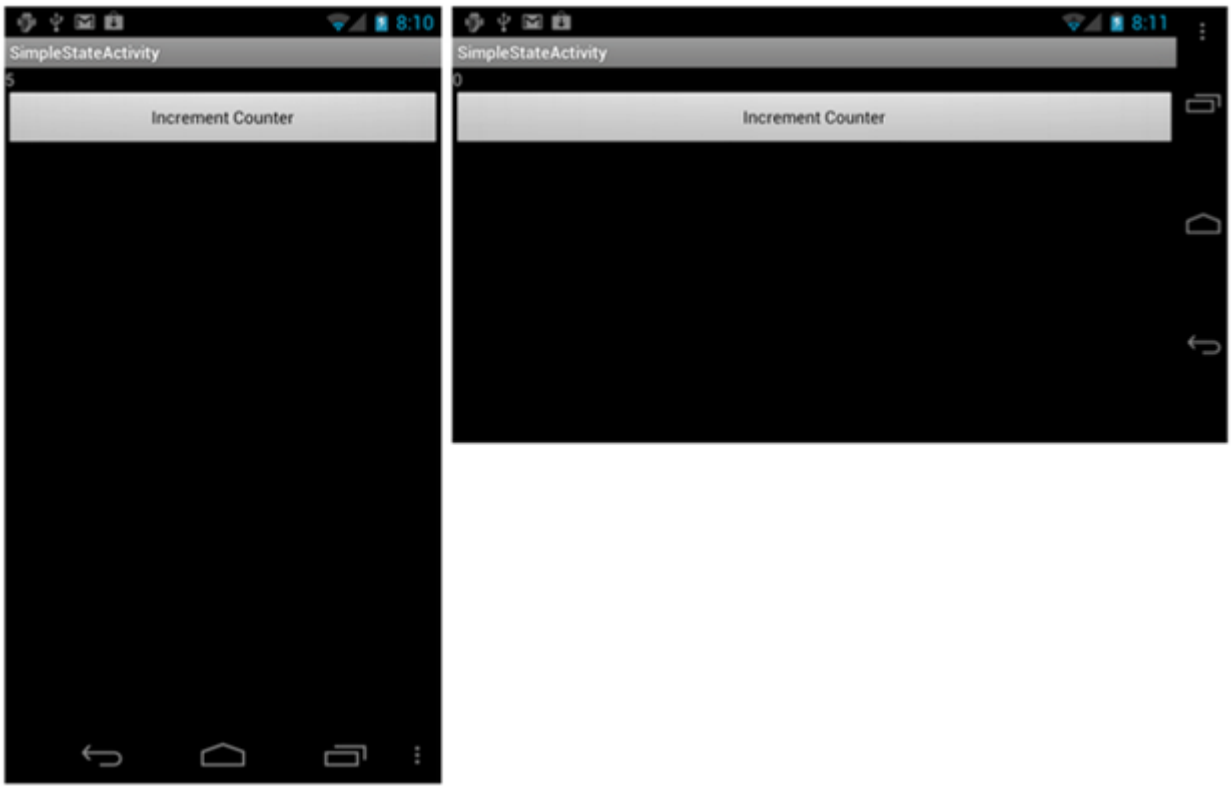
Rien que de très logique. On fait la même chose en XAML, et le principe était le même sous Windows Forms ou même sous Delphi de Borland … C'est pour dire si c'est difficile à comprendre…

Voici un exemple de vue créée par code, elle définit uniquement un `TextView` (un bout de texte) à l'intérieur d'un conteneur `RelativeLayout` (nous verrons les conteneurs plus en détail dans une prochaine partie) :

```
protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);

        // créer le "LayoutRoot", ici un conteneur RelativeLayout
        var rl = new RelativeLayout (this);

        // on modifie ses paramètres
        var layoutParams = new RelativeLayout.LayoutParams (
                ViewGroup.LayoutParams.FillParent,
```

```
                ViewGroup.LayoutParams.FillParent);
                rl.LayoutParameters = layoutParams;

        // création d'un bout de texte
        var tv = new TextView (this);

        // modification des paramètres du texte
        tv.LayoutParameters = layoutParams;
        tv.Text = "Programmatic layout";

        // On ajoute le bout texte en tant qu'enfant du layout
        rl.AddView (tv);

        // On définit la vue courante en passant le Layout et sa "grappe"
d'objets.
        SetContentView (rl);
}
```

Une telle Activité s'affichera de la façon suivante :



Détecter le changement d'orientation

C'est bien, mais nous n'avons fait que créer une vue par code au lieu d'utiliser le designer visuel ou de taper du AXML. Comme on le voit sur la capture ci-dessus Android gère ce cas simple comme celui où un seul fichier AXML est défini : le même affichage est utilisé pour le mode paysage (rien à coder pour que cela fonctionne). Cela peut suffire, mais nous voulons gérer la rotation par code.

Il faut donc détecter la rotation pour prendre des décisions notamment modifier la création de la vue.

Pour ce faire Android offre la classe `WindowManager` dont la propriété `DefaultDisplay.Rotation` permet de connaitre l'orientation actuelle.

Le code de création de la vue peut dès lors prendre connaissance de cette valeur et créer la mise en page ad hoc.

Cela devient un peu plus complexe, par force :

```
protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);

        // Création du Layout
        var rl = new RelativeLayout (this);

        // paramétrage du Layout
        var layoutParams = new RelativeLayout.LayoutParams (
                ViewGroup.LayoutParams.FillParent,
                ViewGroup.LayoutParams.FillParent);
        rl.LayoutParameters = layoutParams;

        // Obtenir l'orientation
        var surfaceOrientation = WindowManager.DefaultDisplay.Rotation;

        // création de la mise en page en fonction de l'orientation
        RelativeLayout.LayoutParams tvLayoutParams;

        if (surfaceOrientation == SurfaceOrientation.Rotation0 ||
                surfaceOrientation == SurfaceOrientation.Rotation180) {
                tvLayoutParams = new RelativeLayout.LayoutParams (
                        ViewGroup.LayoutParams.FillParent,
                        ViewGroup.LayoutParams.WrapContent);
```

```
        } else {
                tvLayoutParams = new RelativeLayout.LayoutParams (
                        ViewGroup.LayoutParams.FillParent,
                         ViewGroup.LayoutParams.WrapContent);
                tvLayoutParams.LeftMargin = 100;
                tvLayoutParams.TopMargin = 100;
        }

         // création du TextView
        var tv = new TextView (this);
        tv.LayoutParameters = tvLayoutParams;
        tv.Text = "Programmatic layout";

        // ajouter le TextView au Layout
        rl.AddView (tv);

        // Initialiser la vue depuis le Layout et sa grappe d'objets
        SetContentView (rl);
}
```

Empêcher le redémarrage de l'Activité

Dans certains cas on peut vouloir empêcher l'Activité de redémarrer sur un changement d'orientation. C'est un choix particulier dont je ne développerai pas ici le potentiel. Mais puisque le sujet est la rotation il est bon de savoir qu'on peut bloquer la réinitialisation de l'Activité.

La façon de le faire sous Xamarin en C# est simple puisqu'elle exploite la notion d'attribut. Il suffit alors de décorer la sous classe `Activity` par un attribut spécial :

```
[Activity (Label = "CodeLayoutActivity",
 ConfigurationChanges=Android.Content.PM.ConfigChanges.Orientation)]
```

Bien entendu le code de l'Activité doit être conçu pour gérer la situation : puisque l'activité ne sera pas recréer, sa méthode `OnCreate()` ne sera pas appelée… Et puisque c'est traditionnellement l'endroit où on charge ou créé la vue, cela implique de se débrouiller autrement. Cet "autrement" a heureusement été prévu, il s'agit de la méthode `OnConfigurationChanged()`. On comprend dès lors beaucoup mieux le contenu de la définition de l'attribut plus haut qui ne fait que demander à Android de déclencher cette méthode lorsque dans la configuration l'Orientation change…

A partir de là les portes de l'infini des possibles s'ouvrent au développeur… La vue sera-t-elle totalement rechargée, recrée par code, modifiée par code, ou seules quelques propriétés d'objets déjà présents dans la vue seront-elles adaptées… Tout dépend de ce qu'on a faire et pourquoi on a choisi de gérer le changement d'orientation de cette façon.

Cette méthode fonctionne pour les deux cas de figure possibles : vue chargée depuis un AXML ou vue créée par code.

Maintenir l'état de la vue sur un changement d'orientation

Par défaut l'Activité est donc redémarrée à chaque changement d'orientation. En clair, une nouvelle instance est construite ce qui implique que les états et données maintenues par l'activité en cours sont perdus.

Ce n'est clairement pas une situation acceptable dans la majorité des cas. L'utilisateur ne doit pas perdre son travail ni même les informations affichées sur un simple changement d'orientation d'autant que celui-ci, comme je le disais, peut être involontaire.

Là aussi tout est prévu.

Android fournit deux moyens différents permettant de sauvegarder ou restaurer des données durant le cycle de vie d'une application :

- Le groupe d'états (`Bundle`) qui permet de lire et écrire des paires clé/valeur
- Une instance d'objet pour y placer ce qu'on veut (ce qui peut s'avérer plus fins que de simples paires clé/valeur).

Le `Bundle` est vraiment pratique pour toutes les données "simples" qui n'utilisent pas beaucoup de mémoire, alors que l'utilisation d'un objet est très efficace pour des données plus structurées et contrôlées ou prenant du temps à être obtenue (comme l'appel à un web service ou une requête longue sur une base de données).

Le Bundle

Le `Bundle` est passé à l'Activité par Android. L'application peut alors s'en servir pour mémoriser son état en surchargeant `OnSaveInstateState()`.

Il y a d'autres façons d'exploiter le `Bundle`. Dans l'exemple qui suit l'UI est faite d'un texte et d'un bouton. Le texte affiche le nombre de fois que le bouton a été cliqué.

On souhaite que cette information ne soit pas annulée par une rotation. Un tel code pourra ressembler à cela :

```
int c;

protected override void OnCreate (Bundle bundle)
{
        base.OnCreate (bundle);

        this.SetContentView (Resource.Layout.SimpleStateView);

        var output = this.FindViewById<TextView> (Resource.Id.outputText);

        if (bundle != null)
                c = bundle.GetInt ("counter", -1);
        else
                c = -1;

        output.Text = c.ToString ();

        var incrementCounter = this.FindViewById<Button>
(Resource.Id.incrementCounter);

        incrementCounter.Click += (s,e) => {
                output.Text = (++c).ToString();
        };
}
```

Le code ci-dessus ne fait que compter les clics et les afficher. La variable "`c`" qui est utilisée appartient à l'Activité, c'est un champ de type "`int`". Lorsqu'une rotation va avoir lieu, l'Activité va être reconstruire. Le code utilise le `Bundle` pour y récupérer cet entier qui est stocké sous le nom de "`counter`".

Simple. Mais qui a stocké la valeur de "`c`" sous ce nom ? C'est la surcharge de la méthode `OnSaveInstateState()` :

```
protected override void OnSaveInstanceState (Bundle outState)
{
        outState.PutInt ("counter", c);
        base.OnSaveInstanceState (outState);
}
```

Lorsque la configuration de l'état de l'Activité nécessite une sauvegarde Android exécutera cette méthode, et c'est elle qui mémorise la valeur de "`c`" sous le nom "`counter`" (la clé) dans le `Bundle`.

C'est ce qui permet au code `OnCreate()` de l'Activité vu plus haut d'exploiter le `Bundle` qui lui est passé pour y rechercher cette entrée et récupérer la valeur de "`c`".

View State automatique

Le mécanisme que nous venons de voir est particulièrement simple et efficace. Mais il peut sembler contraignant d'avoir à sauvegarder et relire comme cela toutes les valeurs se trouvant dans l'UI.

Heureusement ce n'est pas nécessaire !

Toute fenêtre de l'UI (tout objet d'affichage) possédant un ID est en réalité automatiquement sauvegardée par le `OnSaveInstateState()`.

Ainsi, si un `EditText` (un `TextBox` XAML) est défini avec un `ID`, le texte éventuellement saisi par l'utilisateur sera automatiquement sauvegardé et rechargé lors d'un changement d'orientation et cela sans aucun besoin d'écrire du code. C'est une astuce à connaître pour ne pas écrire du code inutile.

Limitations du Bundle

Le procédé du Bundle avec le `OnSaveInsateState()` est simple et efficace comme nous l'avons vu. Mais il présente certaines limitations dont il faut avoir conscience :

- La méthode n'est pas appelée dans certains cas comme l'appui sur Home ou sur le bouton de retour arrière.

- L'objet Bundle lui-même qui est passé à la méthode n'est pas conçu pour conserver des données de grande taille comme des images par exemple. Pour les données un peu "lourdes" il est préférable de passer par `OnRetainNonConfigurationInstance()` ce que nous verrons plus bas.

- Les données du `Bundle` sont sérialisées ce qui ajoute un temps de traitement éventuellement gênant dans certains cas.

Mémoriser des données complexes

Comme on le voit, si l'application doit mémoriser des données complexes ou un peu lourdes il faut mettre en œuvre d'autres traitements que la simple utilisation du `Bundle`.

Mais cette situation aussi a été prévue…

En effet, Android a prévu une autre méthode, `OnRetainNonConfigurationInstance()`, qui peut retourner un objet. Bien entendu ce dernier doit être natif (si Xamarin nous offre .NET l'OS ne s'en préoccupe pas) et on utilisera un `Java.Lang.Object` (ou un descendant).

Il y a deux avantages majeurs à utiliser cette technique :

- L'objet retourné par la méthode `OnRetainNonConfigurationInstance()` fonctionne très bien avec des données lourdes (images ou autres) ou des données plus complexes (structurées plus finement qu'en clé/valeur)

- La méthode `OnRetainNonConfigurationInstance()` est appelée à la demande et seulement quand cela est utile, par exemple sur un changement d'orientation.

Ces avantages font que l'utilisation de `OnRetainNonConfigurationInstance()` est bien adaptée aussi aux données qui "coutent cher" à recharger : longs calculs, accès SGBD, accès Web, etc.

Imaginons par exemple une Activité qui permet de cherche des entrées sur Twitter. Les données sont longues à obtenir (comparées à des données locales), elles sont obtenues en format JSon, il faut les parser et les afficher dans une liste, etc. Gérer de telles données sous la forme de clé/valeur dans le `Bundle` n'est pas pratique du tout et selon la quantité de données obtenues le `Bundle` ne sera pas forcément à même de gérer la chose convenablement.

L'approche doit ainsi être revue en mettant scène `OnRetainNonConfigurationInstance()` :

```
public class NonConfigInstanceActivity : ListActivity
{
        protected override void OnCreate (Bundle bundle)
        {
```

```
            base.OnCreate (bundle);
            SearchTwitter ("xamarin");
        }

        public void SearchTwitter (string text)
        {
            string searchUrl = String.Format(
                    "http://search.twitter.com/search.json?" +
                    "q={0}&rpp=10&include_entities=false&" +
                    "result_type=mixed", text);

            var httpReq = (HttpWebRequest)HttpWebRequest.Create (
                    new Uri (searchUrl));
            httpReq.BeginGetResponse (
                    new AsyncCallback (ResponseCallback), httpReq);
        }

        void ResponseCallback (IAsyncResult ar)
        {
            var httpReq = (HttpWebRequest)ar.AsyncState;

            using (var httpRes =
                    (HttpWebResponse)httpReq.EndGetResponse (ar))
            {
                    ParseResults (httpRes);
            }
            }

        void ParseResults (HttpWebResponse httpRes)
        {
            var s = httpRes.GetResponseStream ();
            var j = (JsonObject)JsonObject.Load (s);

            var results = (from result in (JsonArray)j ["results"]
                    let jResult = result as JsonObject
                    select jResult ["text"].ToString ()).ToArray ();

            RunOnUiThread (() => {
                    PopulateTweetList (results);
            });
        }
```

```
        void PopulateTweetList (string[] results)
        {
                ListAdapter = new ArrayAdapter<string> (this,
                        Resource.Layout.ItemView, results);
        }
}
```

Ce code va fonctionner parfaitement, même en cas de rotation. Le seul problème est celui des performances et de la consommation de la bande passante : chaque rotation va aller chercher à nouveau les données sur Twitter… Ce n'est définitivement pas envisageable dans une application réelle.

C'est pourquoi le code suivant viendra corriger cette situation en exploitant la mémorisation d'un objet Java :

```
public class NonConfigInstanceActivity : ListActivity
{
        TweetListWrapper _savedInstance;

        protected override void OnCreate (Bundle bundle)
        {
                base.OnCreate (bundle);

                var tweetsWrapper =
                        LastNonConfigurationInstance as TweetListWrapper;

                if (tweetsWrapper != null)
                        PopulateTweetList (tweetsWrapper.Tweets);
                else
                        SearchTwitter ("xamarin");
        }

        public override Java.Lang.Object
                OnRetainNonConfigurationInstance ()
        {
                base.OnRetainNonConfigurationInstance ();
                return _savedInstance;
        }


        void PopulateTweetList (string[] results)
```

```
        {
                ListAdapter = new ArrayAdapter<string> (this,
                        Resource.Layout.ItemView, results);
                _savedInstance = new TweetListWrapper{Tweets=results};
        }
}
```

Maintenant, quand l'unité mobile est tournée et que la rotation est détectée les données affichées sont récupérées de l'objet Java (s'il existe) plutôt que déclencher un nouvel appel au Web.

Dans cet exemple les résultats sont stockés de façon simple dans un tableau de `string`. Le type `TweetListWrapper` qu'on voit plus haut est ainsi défini comme cela :

```
class TweetListWrapper : Java.Lang.Object
{
        public string[] Tweets { get; set; }
}
```

Comme on le remarque au passage Xamarin nous permet de gérer des objets natifs Java de façon quasi transparente.

Conclusion

L'approche choisie dans ce billet permet d'entrer dans le vif du sujet en abordant des spécificités "utiles" d'Android et de Xamarin. Encore une fois il n'est de toute façon pas question de transformer le blog en un énième livre sur Android en partant de zéro. L'important est de présenter l'esprit de cet OS, ces grandes similitudes et ses différences avec les OS Microsoft que nous connaissons, et vous montrer comment on peut en C# et en .NET facilement programmer des applications pour smartphones ou tablettes tournant sous Android.

Le but est bien de satisfaire une nouvelle nécessité : être en phase avec le marché dominé par Android sur les mobiles (smartphones et tablettes) et savoir intégrer ces appareils dans une logique LOB au sein d'équipements gérés sous OS Microsoft.

Cross-plateforme : Android – Part 5 – Les ressources

De l'importance d'Android sur le marché aux raisons de s'y intéresser en passant par les bases de son fonctionnement, les 4 parties précédentes ont éclairé le chemin.

Avant de passer à des réalisations concrètes, faisons un point sur une autre spécificité de l'OS, sa gestion des ressources.

Les ressources

Grâces aux ressources une application Android peut s'adapter aux différentes résolutions, aux différents form factors, aux différentes langues.

Devant le foisonnement des combinaisons de ces facteurs et puisque la gestion de l'UI n'est pas vectorielle, il faut de nombreuses ressources adaptées et forcément des conventions pour que l'OS puisse savoir comment choisir les bonnes.

Une application Android n'est que très rarement juste du code, elle s'accompagne ainsi de nombreuses ressources indispensables à son fonctionnement : images, vidéos, fichiers audio, sources xml, etc.

Tous ces fichiers sont des ressources qui sont packagées dans le fichier APK lors du processus de construction du logiciel (Build). L'APK est une sorte de Zip qui contient toute l'application et ses ressources, dans le même esprit que le fichier XAP sous Silverlight.

Quand on créée une application Android on retrouve toujours un répertoire spécial nommé "`Resources`". Il contient des sous-répertoires dont les noms suivent des conventions précises pour séparer les ressources afin que l'OS puisse trouver celles qui sont nécessaires à un moment donné (changement d'orientation, langue, résolution, etc).

On retrouve au minimum les "`drawable`" (*dessinables*) que sont les images et vidéos, les "`layout`" que sont les Vues, les "`values`" qui servent aux traductions et aux constantes de messages. D'autres sous-répertoires peuvent apparaitre selon le développement mais ceux-là sont ce qu'on appelle les ressources par défaut.

Les ressources spécialisées qu'on peut ajouter ensuite ont des noms formés en ajoutant au nom de base ("`layout`", "`drawable`" ...) une string assez courte appelée un *qualificateur* (qualifier). Par exemple les vues en mode portrait seront stockées dans "`layout-port`" et celles en paysage dans "`layout-land`".

Il y a deux voies d'accès aux ressources sous Xamarin.Android : par code ou déclarativement en utilisant une syntaxe XML particulière.

Les noms de fichiers restent toujours les mêmes, c'est leur emplacement dans l'arbre des ressources qui indiquent leur catégorie. Cela oblige à une gestion fine et ordonnée de ces fichiers puisqu'on trouvera sous le même nom des vues en mode portait ou paysage comme des bitmap de différentes résolutions voire de contenu différent.

Les ressources de base

Dans un nouveau projet Xamarin.Android on trouve les fichiers suivants dans les ressources :

- `Icon.png`, l'icône de l'application
- `Main.axml`, la Vue principale
- `Strings.xml`, pour faciliter la mise en place des traductions
- `Resource.designer.cs`, un fichier auto-généré et maintenu par Xamarin et qui tient à jour la table de correspondance entre les noms en clair des ressources utilisés par le développeur et les ID numériques que Android sait gérer. Ce fichier joue le même rôle que "`R.java`" lorsqu'on utilise Eclipse/Java pour développer.
- `AboutResources.txt`, fichier d'aide qui peut être supprimé. A lire pour s'y retrouver quand on débute.

Créer des ressources et y accéder

Créer des ressources est très faciles du point de vue programmation : il suffit d'ajouter des fichiers dans les bons répertoires… Créer réellement au sens de "produire" les ressources est un autre travail, généralement celui du designer (Vue, bitmap, vidéos) ou d'un musicien (musique ou sons en mp3). Cet aspect-là est essentiel mais ne nous intéresse pas ici.

Les fichiers doivent être marqués comme étant des ressources au niveau de l'application ce qui indique au builder comment les traiter, exactement comme sous WPF, Silverlight, WinRT, etc.

On notera qu'à la base Android ne supporte que des noms en minuscules pour les ressources. Toutefois Xamarin.Android est plus permissif et autorise l'utilisation de casses mixtes. Mais il semble plus intelligent de prendre tout de suite les bons réflexes et de n'utiliser que des noms formés selon la convention de l'OS, c'est à dire en minuscules.

Accéder aux ressources par programmation

Les ressources sont des fichiers marqués comme tels et rangés dans des répertoires particuliers de l'application. Un système automatique maintient une équivalence entre les noms des ressources et leur véritable ID numérique géré par Android. C'est le rôle de "`Resources.designer.cs`" qui définit une classe "`Resource`" ne contenant pas de code mais une série de déclaration de type "`nomRessource = xxx`" où "`xxx`" est l'`ID` entier.

Un exemple de ce fichier :

```
public partial class Resource
{
    public partial class Attribute
    {
    }
    public partial class Drawable {
        public const int Icon=0x7f020000;
    }
    public partial class Id
    {
        public const int Textview=0x7f050000;
    }
    public partial class Layout
    {
        public const int Main=0x7f030000;
    }
    public partial class String
    {
        public const int App_Name=0x7f040001;
        public const int Hello=0x7f040000;
    }
}
```

On voit que chaque ressource est déclarée à l'intérieur d'une classe imbriquée, créant ainsi une hiérarchie facilitant l'accès aux ressources d'un type ou d'une autre.

Tel que se présente le fichier exemple plus haut, l'icône de l'application "`Icon.pgn`" pourra être référencée dans le code en utilisant "`Resource.Drawable.Icon`"

La syntaxe complète étant :

```
@[<PackageName>.]Resource.<ResourceType>.<ResourceName>
```

La référence au package est optionnelle mais peut être utile lorsqu'une application complexe est constituée de plusieurs APK.

Accéder aux ressources depuis un fichier XML

A l'intérieur d'un fichier XML, qui est généralement – mais pas seulement – une Vue, il est possible d'accéder aux ressources par une syntaxe particulière, une sorte de "binding statique" :

```
@[<PackageName>:]<ResourceType>/<ResourceName>
```

L'AXML suivant montre cette syntaxe :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <ImageView android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/flag" />
</LinearLayout>
```

Le conteneur de type "`LinearLayout`" est utilisé comme base pour contenir un "`ImageView`", un contrôle affichant une image. La source (propriété "`android:src`") est fixée en utilisant la syntaxe "`@`" suivi du répertoire type de ressource ("`drawable`") et du nom de la ressource "`flag`" séparé par un slash.

Les différentes ressources

Nous avons vu les types de ressources les plus communs comme les images (*drawable*), les vues (*layout*) ou les valeurs (*values*), Android définit bien d'autre types qui seront utilisés de la même façon (définition d'un sous-répertoire).

- *animator*. Fichiers XML définissant des animations de valeurs. Cela a été introduit dans l'API de niveau 11 (Android 3.0).

- *anim*. Fichiers XML définissant des animations spéciales destinées aux Vues par exemple en cas de rotation, de changement de vue, etc.

- *color*. Fichiers XML définissant des listes d'états de couleurs. Par exemple un bouton peut avoir des couleurs différentes selon son état (appuyé, désactivé…), ces changements de couleur en fonction de l'état de l'objet sont stockés dans une liste d'états de couleurs.

- *drawable*. Toutes les ressources "dessinables", donc les graphiques au sens large (gif, png, jpeg, formes génériques définies en XML, mais aussi les "nine-patches" que nous verrons ultérieurement).

- *layout*. Tous les fichiers AXML définissant les vues.

- menu. Fichiers XML définissant les menus d'une application comme le menu des options, les menus contextuels et les sous-menus.

- *raw*. Tout fichier dans son format natif binaire. Ces fichiers ne sont pas traités par Android mais peuvent être utilisés par l'application.

- *values*. Fichiers XML définissant des couples clé/valeur généralement utilisés pour définir des messages affichés par l'application.

- *xml*. Fichiers XML quelconque pouvant être lus par l'application pour son fonctionnement (dans le même esprit que les fichiers de configuration de .NET).

Comme on le voit, Android se base beaucoup sur des conventions de nommage et des arborescences de fichiers. Cela est rudimentaire, ne coute pas très cher à gérer pour l'OS tout en offrant un niveau de souplesse suffisant au développeur.

Toutefois qui dit conventions nombreuses et rigides implique une bonne connaissance de celles-ci et une bonne organisation du travail pour ne pas commettre d'erreurs !

La notion de ressources alternatives

Le sujet a été évoqué à plusieurs reprises ici : le répertoire "`Resources`" se structure en sous-répertoires tels que ceux listés ci-dessus mais chaque sous-répertoire peut se trouver décliner à son tour en plusieurs branches qui vont permettre de différencier les ressources adaptées à une situation ou une autre.

Les principales décisions concernent les locales (localisation de l'application), la densité de l'écran, la taille de l'écran, l'orientation de l'unité mobile.

Les qualificateurs sont alors utilisés en prenant le nom du répertoire de base et en y ajoutant un slash suivi d'un code spécifique.

Pour les ressources de type "*values*" par exemple, si je souhaite créer une localisation en allemand, je placerais le fichier de ressource dans "`Resources/values-de`". Ici c'est le code pays qui est utilisé comme qualificateur.

Il est possible de créer des combinaisons, pour cela chaque qualificateur ajouté est simplement séparé du précédent par un tiret. Toutefois l'ordre d'apparition des qualificateurs doit respecter celui de la table suivante, Android n'analyse pas toutes les substitutions possibles et se fie uniquement à un ordre établi (toujours le principe de conventions rigides permettant de simplifier à l'extrême les traitements côté OS).

- *MCC et MNC*. Ce sont les Mobile Country Code et les Mobile Network Code. Les premiers indiquent le code pays et servent notamment pour les drapeaux ou autres mais pas la localisation, les seconds permettent dans un pays donné de différencier les réseaux. On peut supposer une application qui affiche un texte différent par exemple si le réseau est celui de Free ou de Orange.

- *Langage*. Code sur deux lettres suivant ISO 639-1 optionnellement suivi par un code de deux lettres de la région suivant ISO-3166-alpha-2. Par exemple "`fr`" pour la france, "`fr-rCA`" pour nos amis de l'autre côté de la "grande mare" (les canadiens francophones donc). On note la différence de casse ainsi que la présence de "`r`" pour la région.

- *taille minimum*. Introduit dans l'API 13 (Android 3.2) cela permet de spécifier la taille minimale de l'écran autorisé pour la ressource décrite. Pour limiter à une taille de 320 dp le qualificateur prendra la forme "`sw320dp`". Nous reviendrons sur les "dp" qui définissent mieux que les pixels les tailles exprimées sous Android.

- *Largeur disponible*. Même principe mais la largeur ici peut dépendre du contexte notamment de la rotation (alors que les caractéristiques écran restent les mêmes, la machine ne se changeant pas toute seule…). La forme est "`w<N>dp`". Pour limiter une ressource à une largeur écran effective de 720 dp on ajoutera le qualificateur "`w720dp`". Cela a été introduit dans Android 3.2 (API 13).

- *Hauteur disponible*. Même principe pour la hauteur disponible de l'écran. La lettre "`h`" prend la place de la lettre "`w`".

- *Taille écran*. Généralisation des tailles écran présente depuis Android 2.3 (API 9). Plus générique et moins fin que les qualificateurs évoqués ci-dessus mais actuellement permettant de couvrir 98% des machines en circulation (environ 28 % d'Android 2.3, et plus que 2.2% pour la 2.2 qu'on peut donc ignorer). Les tailles sont ici des noms fixes "`small`, `normal`, `large`" et "`xlarge`".

- *Ratio*. Permet un filtrage des ressources en fonction du ratio de l'écran plutôt que sur sa taille. Cela est couvert par l'API niveau 4 (Android 1.6) et prend les valeurs "`long`" et "`notlong`".

- *Orientation*. Beaucoup plus utilisés, les qualificateurs "`land`" (landscape, paysage) et "`port`" (portrait) permettent de définir des ressources, telles que les Vues ou des drawables en fonction de l'orientation de la machine.

- *Dock*. Permet de savoir si la machine est placée dans son dock (de bureau ou de voiture) pour celles qui savent le détecter. Les valeurs possibles sont "`car`" ou "`desk`".

- *Nuit*. Pour faire la différence en la nuit et le jour ce qui permet d'afficher des couleurs différentes (du rouge en mode nuit pour une carte des étoiles par exemple, couleur n'aveuglant pas), des images différentes, etc. Les valeurs possibles sont "`night`" et … non pas 'day' mais "`nonight`" !

- *Dpi*. Filtre sur la résolution de l'écran en DPI. Cela est important pour les drawables car une bitmap créée avec un certain nombre de pixels pourra devenir toute petite sur un écran de type Rétina ou énorme sur un écran de mauvaise résolution. Il est donc préférable d'avoir préparé plusieurs variantes selon les DPI. La valeur s'exprime en code : "`ldpi`" pour Low densité (basse densité), "`mdpi`" pour densité moyenne, "`hdpi`" pour haute densité, "`xhdpi`" pour extra haute densité, "`nodpi`" pour les ressources qui ne doivent pas être retaillées, "`tvdpi`" introduit en API 13 (Android 3.2) pour les écrans entre le

`mdpi` et le `hdpi`.

- *Type touch*. Pour filtrer en fonction du type d'écran tactile : "`notouch`" pas de tactile, "`stylus`" pour un stylet, et "`finger`" pour le tactile classique avec les doigts.

- *Clavier*. Filtrage selon le type de clavier disponible (ce qui peut changer durant le cycle de vie de l'application). "`keysexposed`" s'il y a un clavier physique ouvert, "`keshidden`" il n'y a pas de clavier physique et le clavier virtuel n'est pas ouvert, "`keyssoft`" s'il y a un clavier virtuel qui est activé.

- *Mode de saisie*. Pour filtrer selon le mode de saisie principal. "`nokeys`" s'il n'y a pas de touches hardware, "`qwerty`" s'il y a un clavier qwerty disponible, "`12key`" quand il y a un clavier 12 touches physiques présent.

- *Navigation*. Disponibilité de touches de navigation (5-directions ou type d-pad). "`navexposed`" si le système existe, "`navhidden`" s'il n'est pas disponible.

- *Navigation primaire*. Type de navigation offerte "`nonav`" aucune touche spéciale en dehors de l'écran tactile lui-même, "`dpad`" '(d-pad, pad directionnel) disponible, "`trackball`", "`wheel`".

- *Plateforme*. Enfin, il est possible de filtrer les ressources sur la version de la plateforme elle-même en spécifiant le niveau de l'API : "`v11`" pour l'API de niveau 11 par exemple (Android 3.0).

Si cette énumération peut sembler fastidieuse, elle permet de bien comprendre les problèmes qui se posent au développeur et au designer …

La liberté sous Android est telle, le foisonnement des modèles, des résolutions, etc, est tel qu'il peut s'avérer presque impossible d'écrire une application s'adaptant à toutes les possibilités. Si toutes les ressources doivent être créées pour toutes les combinaisons possibles, il faudra plus d'espace de stockage pour l'exécutable que n'en propose la plupart des unités mobiles !

Il faudra forcément faire des choix et savoir jongler avec des mises en page souples s'adaptant à toutes les situations, au moins aux plus courantes…

Le problème est moindre si on vise un type particulier de machines (cibler que les tablettes par exemple), une ou deux langues précises, etc.

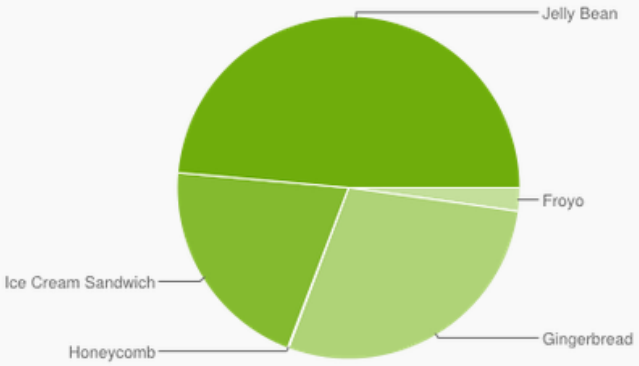
Le problème n'est pas typique d'Android, il se pose à tous les développeurs sous Windows aussi.

Il n'y a guère que chez Apple où tout cela n'a pas de sens, puisque l'acheteur n'a pas de choix…

Android utilise un mécanisme déterministe et connu pour choisir dans les ressources. Ainsi il va éliminer tout ce qui est conflit avec la configuration de la machine en premier. Il éliminera ensuite toutes les variantes qui ont des qualificateurs s'éloignant de la configuration en prenant l'ordre de la liste ci-dessus comme guide.

Pour s'aider à prendre les bonnes décisions, Google fournit aussi des outils analytiques en ligne comme les Dashboards qui permet de connaitre au jour le jour la répartition des versions d'Android dans le monde, celui des API etc.
Par exemple, à la présente date la répartition est la suivante :

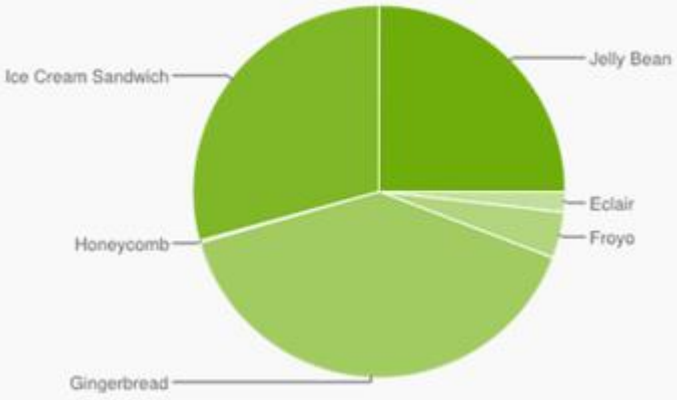
Version	Codename	API	Distribution
2.2	Froyo	8	2.2%
2.3.3 - 2.3.7	Gingerbread	10	28.5%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	20.6%
4.1.x	Jelly Bean	16	36.5%
4.2.x		17	10.6%
4.3		18	1.5%



Data collected during a 7-day period ending on October 2, 2013.
Any versions with less than 0.1% distribution are not shown.

Il est très intéressant de comparer avec la répartition qui existait il n'y a que quelques mois lorsque j'ai écrit le billet original :

Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.7%
2.2	Froyo	8	4.0%
2.3 - 2.3.2	Gingerbread	9	0.1%
2.3.3 - 2.3.7		10	39.7%
3.2	Honeycomb	13	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	29.3%
4.1.x	Jelly Bean	16	23.0%
4.2.x		17	2.0%

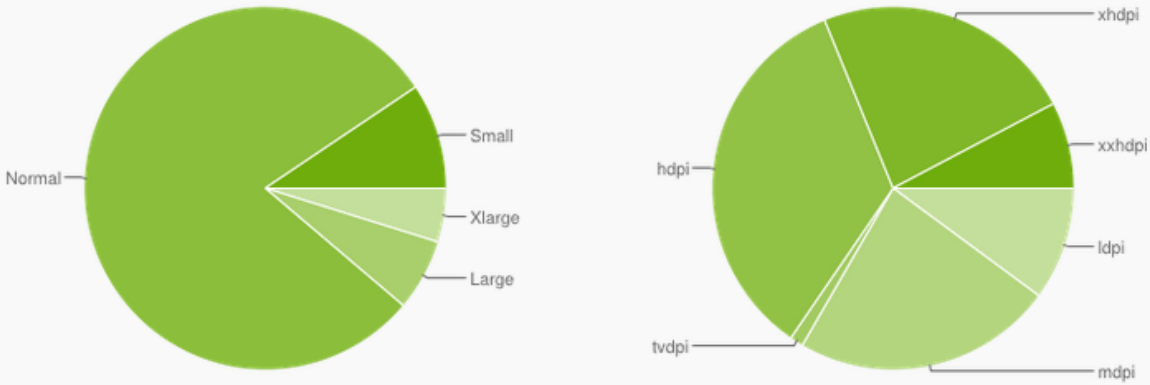


Data collected during a 14-day period ending on April 2, 2013.
Any versions with less than 0.1% distribution are not shown.

On voit sur l'image ci-dessus qu'il y a quelques mois Jelly Bean, la dernière version (Kit-Kat va bientôt sortir) n'avait que 25 % du marché, alors que le graphique précédent, à jour fin octobre 2013, montre une part de 51 %. Cela en dit long sur l'accélération constante des ventes de machines Android. La fragmentation existe toujours mais les nouveaux modèles se vendant encore plus que les précédents, en pourcentage ils écrasent les vieilles versions.

Pour les tailles écran nous obtenons le tableau suivant :

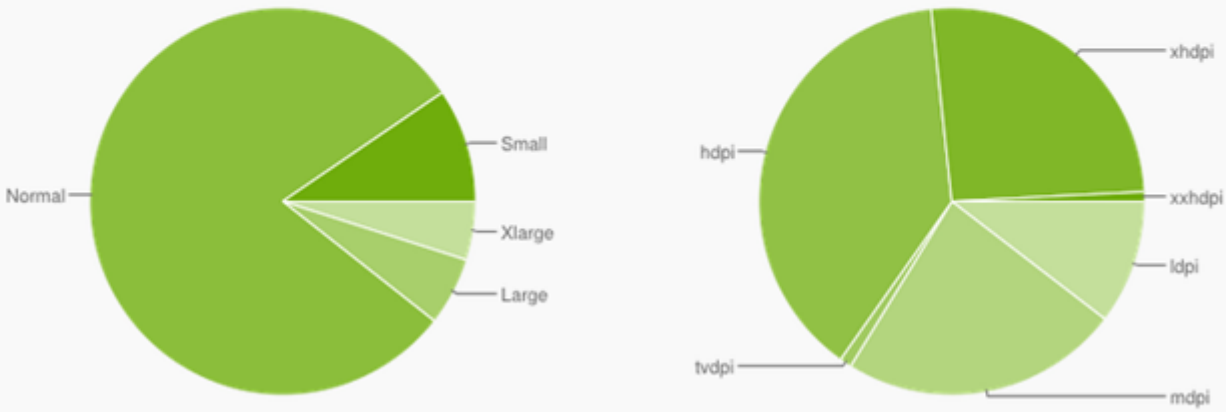
	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	9.4%						9.4%
Normal	0.1%	15.3%		33.5%	22.8%	7.7%	79.4%
Large	0.6%	3.5%	1.2%	0.5%	0.6%		6.4%
Xlarge		4.4%		0.3%	0.1%		4.8%
Total	10.1%	23.2%	1.2%	34.3%	23.5%	7.7%	



Data collected during a 7-day period ending on October 2, 2013
Any screen configurations with less than 0.1% distribution are not shown.

Graphiques qu'on peut comparer aux mêmes données début avril 2013, un écart de 7 mois pleins environ :

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	9.5%						9.5%
Normal	0.1%	16.1%		37.9%	25.0%	0.8%	79.9%
Large	0.7%	2.7%	1.0%	0.5%	0.8%		5.7%
Xlarge	0.1%	4.6%		0.1%	0.1%		4.9%
Total	10.4%	23.4%	1.0%	38.5%	25.9%	0.8%	



Data collected during a 14-day period ending on April 2, 2013
Any screen configurations with less than 0.1% distribution are not shown.

Il est intéressant de constater qu'à la différence des versions d'Android en circulation les tailles écran n'ont presque pas bougé ! La course aux phablets et aux machines ayant une grande diagonale laisse le marché dans le même état… Les gens achètent donc beaucoup de smarphone à petit écran, beaucoup moins cher, ceci expliquant cela, malgré le succès énorme des Galaxy S3, S4 et Note 2 et 3. D'autres fabricants proposent des machines à écran large (HTC One Max par exemple, ou le Sony Xperia Z) mais leurs ventent doivent être marginales. En revanche on voit une amélioration des résolutions en DPI, le xxhdpi prenant presque 8% en partant de 0.8% il y a un peu plus de six mois. N'oublions pas qu'effectivement ces comparaisons sont totalement arbitraires entre la date de révision de ce billet pour la version livre PDF et celle de son écriture originelle… En presque 7 mois, ce qui est finalement très court, le marché évolue donc très vite !

Ces informations sont sans cesse mises à jour et sont très importantes pour le développeur et le designer puisqu'elles permettent de faire des choix raisonnables en termes de niveau d'API supporté et de taille/résolution d'écran.

Créer des ressources pour les différents types d'écran

Je viens d'en parler avec les Dashboards Google, le chalenge le plus important peut-être pour le développeur et le designer est de s'adapter au foisonnement des formats disponibles (taille et résolution d'écran).

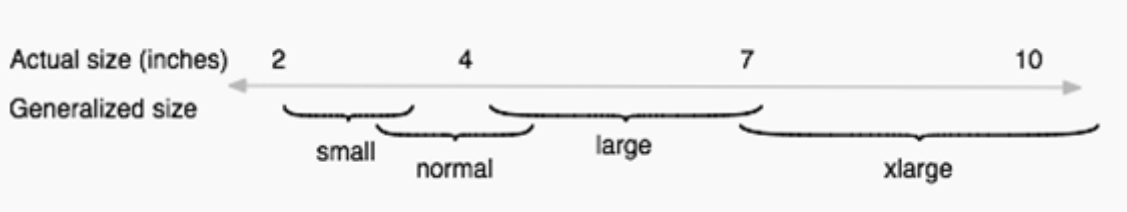
Si la majorité des utilisateurs ont un écran de taille "normale", certains ont des écrans petits (small), grands (Large) voire super grands (Xlarge). Et cela ne concerne que la taille écran, pour une même taille physique d'écran il existe de nombreuses résolutions, de "ldpi", basse densité, à xxhdpi, la super haute densité de type Rétina.

Normal ?

Qu'est que la "normalité" ? Non, ne fuyez pas, nous n'allons pas faire un devoir de philo !

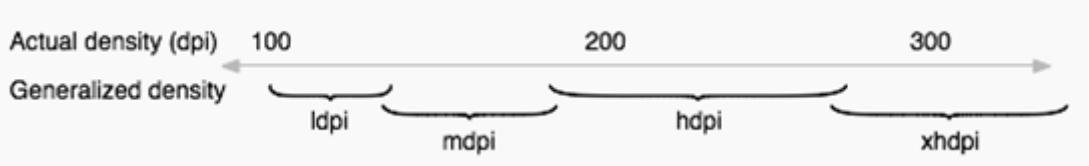
Mais on peut se demander comment est définie la "normalité" dans les tailles d'écran.

C'est un concept flou et qui n'apprend pas grand chose. C'est bien entendu sans compter sur la documentation Google qui précise ce genre de choses :



Un écran "normal" est un écran qui fait grosso-modo entre 3 et 4.5 pouces. Un pouce faisant 2.56 cm. A partir de là tout le reste se décline. Les "small" (petits) sont en dessous de 3", les grands au-dessus de 4,5" jusqu'à près de 7 pouces, disons le format "phablet" ou mini tablette, le "xlarge" commence à 7" et s'étant jusqu'au-delà de 10 pouces. Si on vise un développement smartphone il est clair qu'aujourd'hui il faut viser le support de "normal" et "large", si on vise les tablettes et les phablettes il faut cibler "large" et "xlarge". Cela limite les combinaisons malgré tout.

Les résolutions comptent beaucoup, et sont elles aussi définies par la documentation :



Le "hdpi" et le "'xhdpi" sont les plus utilisées à ce jour, c'est à dire que les résolutions les plus faibles en dessous de 200 DPI peuvent de plus en plus être ignorées.

Sous Android les tailles des objets peuvent ainsi varier grandement selon le support, une image de 100x100 pixels n'aura pas le même rendu et la même taille sur un écran très haute densité que sur un écran de 100 DPI. Cette variabilité a obligé à définir un autre système de mesure, indépendant de ces paramètres, les "DP".

Les "Density-independent pixel"

Les "DP" sont une unité de mesure virtuelle remplaçant la notion de pixel trop variable dans des environnements où la résolution des écrans est d'une grande disparité.

Cela permet notamment de fixer des règles de mise en page indépendantes de la résolution.

La relation entre pixel, densité et "dp" est la suivante :

```
px = dp * dpi/160
```

Exemple : sur un écran de 300 DPI de densité, un objet de 48 dp aura une taille de (48*300/160) soit 90 pixels.

Les tailles écran en "dp"

Les tailles écran que nous avons vu plus haut peuvent s'exprimer en "dp" plutôt qu'en pouces ce qui facilite la création de patrons pour sketcher les applications puisque toutes les tailles s'expriment en "dp" sous Android.

- *xlarge* définit ainsi des écrans qui font au minimum 960x720 dp
- *large*, 640x480 dp
- *normal*, 470x320 dp
- *small*, 426x320 dp

Avant Android 3.0 ces tailles n'étaient pas aussi clairement définies, le marché était encore flou et les avancées techniques très rapides. De fait on peut trouver de veilles unités datant d'avant Android 3.0 qui, éventuellement, pourrait rapporter une taille qui ne correspondrait pas exactement à celles indiquées ici. Cette situation est malgré tout de plus en plus rare et peut être ignorée.

Supporter les différentes tailles et densités

Le foisonnement des matériels très différents complique un peu la tâche du développeur et du designer, mais comme nous l'avons vu, dans la réalité, la majorité du marché se trouve être couvert par deux ou trois combinaisons principales, ce qui simplifie les choses.

De plus Android fait lui-même le gros du travail pour adapter un affichage à un écran donné. Il n'en reste pas moins nécessaire de lui donner un "petit coup de main" de temps en temps pour s'assurer du rendu final de ses applications.

La première des choses consiste à adopter les "dp" au lieu des "pixels". Rien que cela va permettre une adaptation plus facile des mises en page et des images. Android effectuera une mise à l'échelle de ses dernières au runtime. Mais comme vous le savez, le problème des bitmaps c'est qu'elles ne supportent pas de grandes variations de taille même avec de bons algorithmes…

Les mises à l'échelle des bitmaps ne tiennent que dans une fourchette assez faible de pourcentages, au-delà d'une certaine valeur de zoom (avant ou arrière), l'image apparaitra floutée ou pixellisée.

Pour éviter ce phénomène les environnements vectoriels comme XAML apportent une solution radicale, tout est toujours affiché à la bonne résolution. Mais sous Android qui n'est pas vectoriel il faudra fournir des ressources alternatives en se débrouillant pour couvrir les principales cibles.

Limiter l'application à des tailles écran

Lorsqu'on a fait le tour des tailles écran que l'application peut supporter il peut s'avérer assez sage de le préciser dans les paramètres (le manifeste de l'application, jouant le même rôle que sous WinRT) afin de limiter le téléchargement de l'application. Un utilisateur balayant le Store ne verra que ce que sa machine peut afficher.

C'est à chaque développeur de trancher, mais d'une façon générale je conseille plutôt d'interdire le téléchargement aux tailles non gérées proprement par l'appareil plutôt que de laisser un potentiel utilisateur charger une application qui n'est pas supportée et qui donnera l'impression d'être de mauvaise qualité. Cette frustration là créée une impression négative difficile à effacer, la frustration ne pas pouvoir télécharger n'est pas du même ordre et peut même créer l'envie…

> *N'oubliez jamais ce que j'appelle « l'effet Télé 7 Jours » en référence à ce vieux journal qui donne les programmes de la télé et qui servait déjà de bible au télespectateur des années 60. Dans ce vénérable ouvrage de mon enfance il y avait (et peut-être y-a-t-il toujours, ce journal existe encore et est en 4eme position des journaux de ce type) une rubrique « courrier des lecteurs » qui m'amusait bien plus que le reste. Elle contenait quelques lettres ou extraits du courrier que le journal recevait. On y voyait des choses comme « Merci à M. Drucker pour cette émouvante soirée sur Tino Rossi » mais la plupart des messages étaient plutôt négatifs du type « La nouvelle coiffure de Madame machin est une horreur ! » (Madame machin étant une speakerine, emploi qui n'existe plus depuis un moment à la télé). Et je m'amusais de cette correspondance toujours un poil ringarde (comme le téléspectateur moyen de l'époque, et d'aujourd'hui d'ailleurs), franchouillarde, moraliste et souvent excessive. Et je m'interrogeais (tout petit déjà j'étais agité du neurone) sur l'effet de « loupe » de ces commentaires majoritairement négatifs. Et du haut de pas grand-chose j'en avais déduit par une réflexion intense que finalement les gens mécontents s'expriment bien plus que les gens satisfaits qui ne perdent pas leur temps pour dire « tout est ok c'est bien », créant de fait une distorsion dans les appréciations.*
>
> *Depuis, j'appelle « Effet Télé 7 Jours » cette tendance à la présence toujours plus marquée des commentaires négatifs par rapport à ceux qui sont positifs. L'invention et la mondialisation*

d'Internet me permettra des années plus tard de vérifier à grande échelle la pertinence de cette réalité qui force certains à payer de faux commentaires pour tenter de « rétablir l'équilibre », voire de frauder tout simplement, mais c'est une autre histoire.

De fait, l'Effet Télé 7 Jours existent toujours, multiplier par cent milliards. Et au lieu de se cacher dans une page intérieure d'un hebdomadaire un peu ringard (de mon souvenir, les versions récentes sont peut-être tout à fait à la mode) aujourd'hui cet effet se propage sur les réseaux sociaux et les notations des applications… Vous voyez maintenant pourquoi j'ai raconté tout ça !

Il est donc préférable d'éviter l'Effet Télé 7 Jours en ne proposant son application qu'aux personnes ayant des machines sur lesquelles on est sûr que l'UX sera de bonne qualité. Vouloir balayer large c'est ouvrir grande les vannes de la critique négative qui laisse des traces indélébiles… Et qui justifia même à la base la création du métier de Community Manager qui originellement et sous d'autres noms parfois a eu pour tâche de maîtriser les avalanches médiatiques non contrôlées. Les forums ont utilisé le nom de modérateur pour ce job bien avant qu'on invente un titre plus ronflant et je connais certains gros sites qui ont utilisé cette « modération » de façon radicale en effaçant systématiquement tout ce qui déviait de la « ligne » dictée par le Boss … Et cela se pratique toujours d'ailleurs.

Mais sur un site comme le Play Store, vous ne serez jamais modérateur… Et là l'information vous échappera, vous ne pourrez pas jouer au censeur. D'où l'intérêt pour les sociétés qui en ont les moyens d'avoir un CM qui est là pour rattraper les dérapages avant qu'ils ne soient incontrolables… Mieux vaut donc prévenir que guérir, car en matière de réputation (on parle aujourd'hui pour faire plus dans le vent de e-réputation) les petites fautes peuvent laisser de grandes traces et nuirent durablement à un produit, voire à un éditeur ou une marque.

C'est ainsi dans le fichier `AndroidManifest.XML` qu'on peut jouer sur la section "`supports-screens`" pour déclarer les tailles supportées par l'application.

La conséquence est que l'application n'apparaitra dans Google Play si ce dernier est accédé depuis une machine non supportée. Si l'installation est faite "de force" par d'autres moyens, elle tournera bien entendu sans encombre, sauf que sa mise en page sera certainement peu agréable. Le manifeste joue donc un rôle de filtrage ici pour le store, mais pas pour le runtime.

Voici un exemple de déclaration vu depuis Xamarin Studio (on peut voir les mêmes données sous VS) :

Fournir des mises en page alternatives

Android tentera toujours d'afficher les vues qui sont chargées par une application en leur appliquant si besoin un redimensionnement. Cela peut suffire dans certains cas, dans d'autres il peut être plus judicieux d'adapter cette mise en page en réduisant ou éliminant certains éléments (pour les petites tailles) ou en agrandissant ou ajoutant certaines zones (pour les grandes tailles).

Depuis l'API 13 (Android 3.2) l'utilisation des tailles écran est *deprecated*, il est conseillé d'utiliser la notation "`sw<N>dp`". Si vous visez une compatibilité maximale en ce moment, vous choisirez de supporter Android 2.3 au minimum, soit l'API de niveau 10. Dans un tel cas vous utiliserez le système de tailles écran vu plus haut. Si vous décidez de cibler 3.2 et au-delà il faut alors utiliser la nouvelle notation.
La décision est un peu difficile à l'heure actuelle car la version 2.3 est un peu un pivot, 40 % de machines font tourner cette version (et les inférieures) et 60% font tourner 3.2 et au-delà. On comprend que beaucoup de développeurs préfèrent encore aujourd'hui, sauf nécessité (nouvelles API), supporter 2.3 bien que l'OS ait beaucoup évolué entre la 2.3 et les 4.x, notamment au niveau du look bien meilleur aujourd'hui. Mais rien n'interdit de mimer un look moderne sous 2.3.

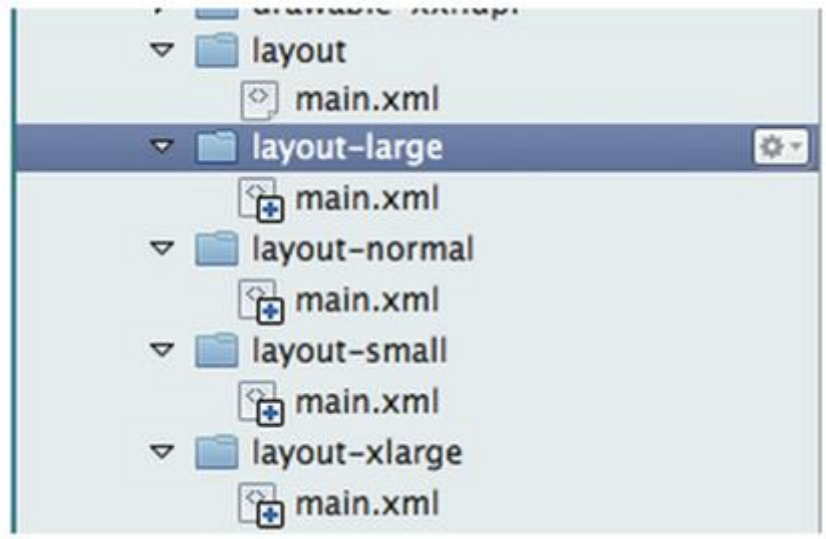
Dans l'arborescence des ressources, pour supporter un écran de 700dp de large minimum (simple exemple) il faudra fournir une vue alternative de cette façon :

On remarque la section "`layout`" par défaut avec le "`main.axml`" et la section "`layout-sw700dp`" avec un fichier "`main`" lui aussi renommé en fonction de ce filtrage.

On se rappellera qu'un téléphone typique aujourd'hui est dans les 320 dp de large, qu'une phablette de type Samsung Note 2 en 5" fait 480 dp de large, qu'une tablette

7" est dans 600 dp de large alors qu'un tablette 10" compte 720 dp minimum de large.

Si l'application cible les versions sous la 3.2 (donc les API jusqu'au niveau 12), les mises en page seront précisées en utilisant la codification "`normal`", "`large`", etc :



Si les deux procédés sont mis en place simultanément, ce qui est tout à fait possible, c'est la nouvelle notation qui prend la précédence à partir de l'API 13. Si une application veut cibler les machines avant 3.2 en même temps que celles à partir de 3.2 et qu'elle souhaite gérer plusieurs tailles d'écran elle pourra le faire en mixant les deux notations donc. Attention à une telle gestion très tatillonne qu'il faut être capable de maintenir dans le temps !

Fournir des bitmaps pour les différentes densités

Comme je l'ai déjà expliqué, Android fait une mise à l'échelle des bitmap en fonction de la résolution. Cela peut suffire mais quand l'écart est trop grand les bitmaps peuvent être floutés ou pixellisés. Dans ce cas il convient de fournir des versions adaptées aux différentes résolutions.

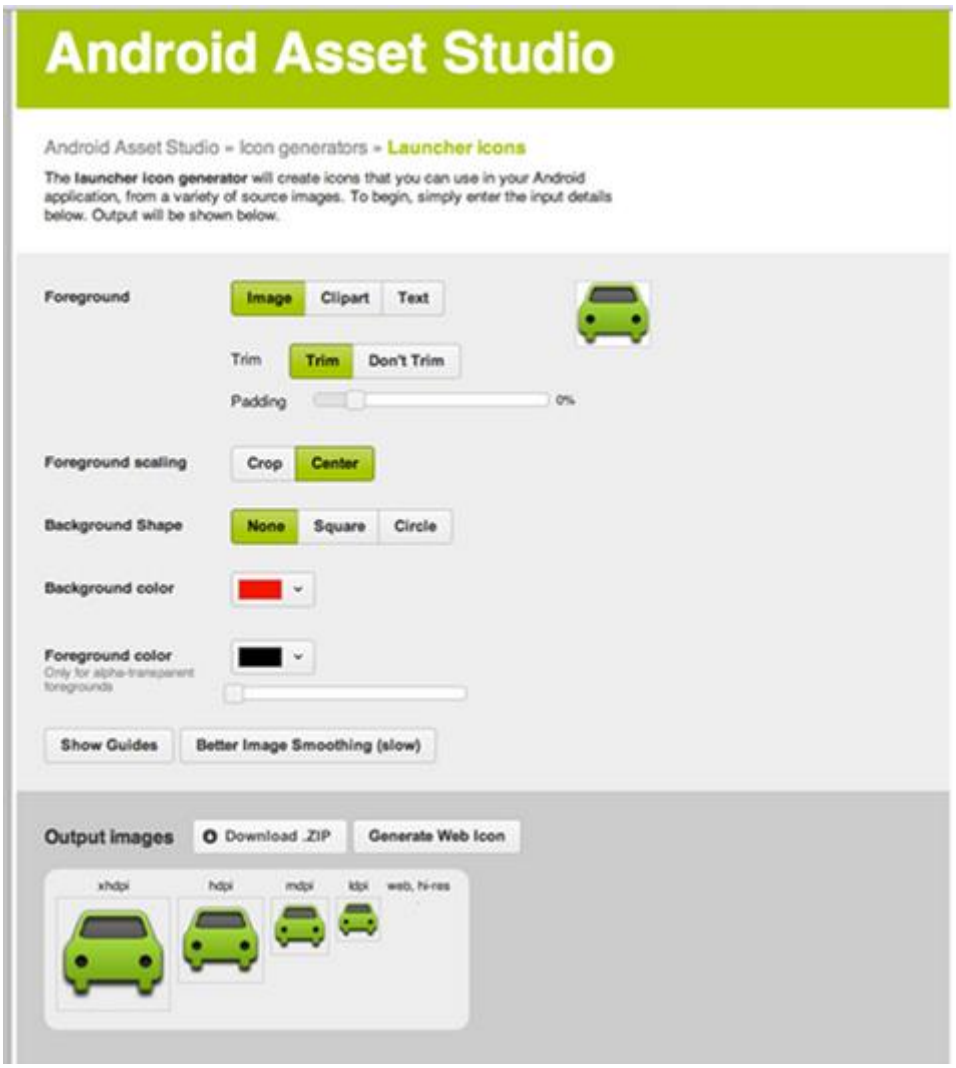
Le principe a déjà été exposé et ne présente pas de difficultés. Mais au lieu de tout faire à la main on peut aussi utiliser des outils existants…

Créer les ressources pour les densités différentes

La création des images dans les différentes résolutions est une tâche un peu ennuyeuse surtout quand on fait des adaptations de ces images en cours de développement, ce qui réclame de générer à nouveau toute la série de résolutions.

Google met à la disposition des développeurs un outil assez simple mais très pratique pour faciliter cette tâche : l'Android Asset Studio. Cet outil en ligne regroupe plusieurs utilitaires permettant de générer des icones pour la barre de lancement, ou pour la barre d'action, pour les notifications, etc. Le tout en respectant les tailles standard en générant toutes les séries nécessaires au support des différentes résolutions.

Il est possible de partir d'une image qu'on fourni, d'un symbole sélectionné dans une librairie ou d'un texte :



Une fois les réglages effectués l'outil génère la série d'images et permet de télécharger un paquet zippé.

D'autres outils communautaires existent dans le même genre.

Automatiser les tests

Tant de combinaisons possibles de tailles écran et de résolutions créent une situation un peu désespérante à première vue… "Comment vais-je pouvoir tout tester ?"

Comme je l'ai déjà indiqué, le principe est surtout de savoir se limiter et ne pas chercher à couvrir 100% du parc mondial… Ensuite il faut disposer de machines réelles pour tester. Les émulateurs ont toujours leurs limites.

Lorsqu'une application est vraiment importante (mais y en a t il qui ne le sont pas ?) on peut aussi utiliser des services spécialisés. ApKudo permet de certifier une application comme sur le store Apple ou Microsoft par exemple, avec un retour d'information sur tous les problèmes rencontrés, The Beta Family fonctionne sur le principe développeurs/testeurs, vous avez des retours de testeurs, en échange vous leur offrez une licence ou autre chose, ou bien Perfecto Mobile, eux possèdent presque toutes les machines et lancent un test en parallèle sur près de 300 unités différentes, vous obtenez des résultats clairs et vous savez exactement où ça passe et où ça ne passe pas.

Ces services permettent d'avoir une bonne idée de la fiabilité de l'installation, de l'UI, etc, et ce gratuitement ou presque.

Localisation des strings

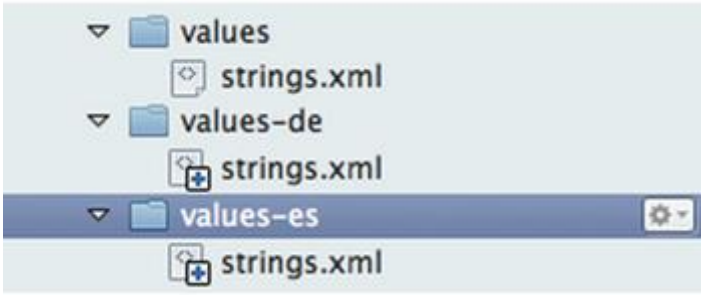
Parmi les ressources essentielles d'une application se trouvent les chaines de caractères. Une approche particulière est nécessaire dès lors que ces chaines doivent être traduites.

Il n'est alors plus question de les stocker en dur dans l'interface ou dans le code… Et comme on ne sait jamais, je conseille toujours de faire "comme si" la traduction était nécessaire dès le départ. Dans les projets directement multi-lingue la question ne se pose bien entendu pas.

Le procédé est très simple puisque dans le répertoire des ressources d'une application se trouve le sous-répertoire "`values`". C'est ici que se trouve par défaut "`strings.xml`", un fichier de définition de chaines.

En spécialisation le répertoire "`values`" en fonction du code pays (et du code région si vraiment cela est nécessaire) on pourra copier le fichier "`strings.xml`" et le traduire dans les différentes langes supportées, Android chargera les bonnes données automatiquement.

Par exemple une application supportant une langue par défaut ainsi que l'espagnol et l'allemand présentera une arborescence de ce type :



Une vue accédant aux données localisées présentera un code de ce type :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
<Button
   android:id="@+id/myButton"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
android:text="@string/hello"
   />
</LinearLayout>
```

Le bouton défini en fin de fichier utilise un texte puisé dans la ressource "`string`" qui porte le nom de "`hello`". La correspondance entre ce nom, "`hello`" et sa valeur est effectuée dans le fichier "`strings.xml`" correspondant à la langue en cours, choix effectué par Android automatiquement.

Ce mécanisme, comme le reste sous Android, est basé sur des principes simples, un peu frustres (comme des conventions de noms de répertoires) mais cela permet d'assurer des fonctions très sophistiquées sans consommer beaucoup de ressources machine. Quand on compare avec les mécanismes de localisation sous Silverlight, WPF ou WinRT on s'aperçoit que MS ne choisi par toujours le chemin le plus court pour arriver au résultat. L'astuce de Google c'est d'avoir une vision très pragmatique des besoins et d'y avoir répondu de façon très simple. Pas de quoi passer une thèse donc sur la localisation sous Android, mais à la différence d'autres OS c'est justement

très simple, robuste, et peu gourmand. Android vs Windows Phone c'est un peu l'approche russe vs celle de la Nasa… Les trucs de la Nasa sont toujours d'un niveau technique incroyable, mais quand on compare, finalement, aujourd'hui ce sont les russes qui seuls peuvent ravitailler l'ISS … La simplicité et la rusticité ont souvent du bon, même dans le hi-Tech !

Conclusion

Les ressources sont essentielles au bon fonctionnement d'une application car elle n'est pas faite que de code mais aussi de vues, d'images, d'objets multimédia.

Android offre une gestion particulière des ressources car les types d'unités faisant tourner cet OS sont d'une variété incroyable et qu'il fallait inventer des mécanismes simples consommant très peu.

Avec un bon ciblage et une bonne organisation du développement et du design il est tout à fait possible de créer des applications fonctionnant sur la majeure partie du parc actuel.

Il ne reste plus qu'à mettre tout ceci dans le chaudron à idées et à commencer le développement de vos applications cross-plateformes sous Android !

Il reste bien deux ou trois choses à voir encore, mais nous le ferons en partant d'exemples…

Cross-plateforme : la mise en page sous Android (de Xaml à Xml).

Après avoir abordé de nombreux exemples techniques de programmation sous toutes les plateformes principales (WinRT, Windows Phone 8, WPF, Android…) il est nécessaire d'en connaitre un peu plus sur chacune pour développer de vraies applications. Dot.Blog regorge d'informations sur Xaml alors ajoutons quelques informations essentielles sur Android en les rapprochant de nos connaissances Xaml…

Chapitres antérieurs

Depuis plus d'un an Dot.Blog vous prépare psychologiquement autant que techniquement à la situation présente… Un monde dans lequel le développement devient cross-plateforme car plus aucun éditeur d'OS n'est en mesure de porter à lui-seul 100% du marché et que le partage de ce dernier est beaucoup plus équilibré que par le passé où la domination quasi monopolistique de Microsoft nous avait mis à l'abri des affres du choix… On a beaucoup critiqué Microsoft sur cette période, on a

même fait des procès à Microsoft pour sanctionner ses pratiques anti-concurrentielles. Mais comme toute dictature il y avait un avantage énorme : la stabilité, et la stabilité c'est bon pour les affaires... On le voit politiquement là où les dictatures tombent c'est le chaos bien plus qu'une riante démocratie qui s'installe… Si la Liberté des Hommes ne se discute pas et qu'on peut accepter d'en payer le prix fort, dans notre humble domaine le chaos est plutôt une mauvaise chose, une bonne hégémonie simplifie tout et facilite le business. Mais ces temps sont révolus comme je vous y prépare de longue date (lire le "Big Shift" billet de 2011…). Mais je ne fais pas qu'alerter longtemps à l'avance, cette avance je l'utilise pour vous former aux parades qui vous protègeront de ces changements. La parade contre le chaos qui s'est installé s'appelle le cross-plateforme.
D'ailleurs même Microsoft vous lance un message fort dans ce sens, la suite Office 365 est conçue pour fonctionner sur toutes les plateformes ! Lorsqu'il s'agit d'endoctriner « ses » développeurs Microsoft nous crie que hors Modern UI il n'y a pas de salut, mais quand Microsoft, l'éditeur de logiciel, raisonne pour ses propres intérêts, la décision du cross-plateforme s'impose vite devant les discours de façade…

Le sujet a été traité des dizaines de fois mais quelques séries méritent votre attention si jamais vous les avez loupées (vous retrouverez dans un ordre un peu différent tous ces billets dans le présent livre PDF, les liens donnés étant ceux des textes originaux sur Dot.Blog) :

Stratégie de développement cross-plateforme

Série en 4 parties :

- Partie 1
- Partie 2
- Partie 3
- Bonus

Cross-plateforme : L'OS Android

Série en 5 parties abordant les spécificités principales d'Android :

- Cross-plateforme : Android - Part 1
- Cross-Plateforme : Android – Part 2 – L'OS
- Cross-plateforme : Android – Part 3: Activité et cycle de vie
- Cross-plateforme : Android – Part 4 – Vues et Rotation
- Cross-plateforme : Android – Part 5 – Les ressources

8h de video en 12 volets MvvmCross sous WinRT / WPF / Windows Phone 8 et Android

Sans oublier bien entendu cette série exceptionnelle de 12 vidéos pour un total de 8h en HD sur le développement cross-plateforme :

- Cross-Plateforme : L'index détaillé des 12 vidéos de formation offertes par Dot.Blog – Vidéos 1 à 6
- Cross-Plateforme : L'index détaillé des 12 vidéos de formation offertes par Dot.Blog – Vidéos 7 à 12

L'IoC clé de voute du Cross-plateforme

l'IoC avec MvvmCross, de WinRT à Android en passant par Windows Phone, PDF a télécharger.

La stratégie de mise en page sous Android

J'ai déjà parlé de mise en page sous Android mais depuis la série de vidéos (voir liens ci-dessus) je me suis rendu compte que finalement, dès lors qu'on approche le cross-plateforme avec la méthode que je décris le seul véritable problème qui se pose est celui de la mise en page sous cet OS. En effet, la stratégie que je vous propose de suivre déporte toute la complexité dans un projet noyau qui n'est que du C#. Les projets d'UI sont rudimentaires et le plus souvent ne contiennent aucun code, en dehors de la mise en page.

Travailler sous WinRT, WPF, Silverlight, Windows Phone, tout cela est facile pour ceux qui suivent Dot.Blog depuis longtemps puisque C# et Xaml en sont le sujet de prédilection. Créer un logiciel cross-plateforme ne pose donc aucun souci pour le noyau ni pour la création des projets d'UI en OS Microsoft. La seule difficulté consiste donc à réussir des mises en page sous Android (et même iOS que je n'aborde pas mais qui peut intéresser des lecteurs).

Dès lors, la seule chose cruciale à apprendre pour coder un premier projet cross-plateforme quand on suit ma méthode c'est d'être capable de faire la mise en page de l'application sous Android car cet OS est aujourd'hui aux unités mobiles ce que Microsoft a été aux PC pendant les 25 dernières années. Tout simplement.

D'où l'idée de ce billet qui regroupe les principales informations de base à propos de la stratégie de mise en page sous cet OS. Avec ces bases vous serez à même de comprendre l'essentiel pour vous lancer et mieux comprendre les informations complémentaires que vous pourrez glaner sur le Web. En tout cas c'est l'objectif !

Bien entendu si vous désirez les conseils éclairés d'un spécialiste, n'hésitez pas à me contacter, c'est mon métier…

XML Based

La première chose à savoir c'est que la mise en page d'une application Android passe par un langage de définition très proche de Xaml dans son formalisme puisqu'il s'agit d'un fichier XML. On y retrouve ainsi des balises avec des attributs et des contenus qui forment le plus souvent des arborescences décrivant la structure du document.

La grande différence avec Xaml est la nature du système d'affichage. Sous Xaml nous travaillons (et c'est unique en informatique, SVG n'étant pas à la hauteur) dans un mode totalement vectoriel autorisant aussi bien la présence de composants visuels complexes que de dessins en courbes de Béziers. Le système des templates, des styles, du bindind font de Xaml l'arme absolue de la mise en page et, peut-être par manque d'imagination, je n'arrive pas à concevoir ce qui pourrait être inventé de très différent et de beaucoup mieux.

Donc sous Android c'est très proche, formellement, mais fondamentalement différent sur le fond. On se rapproche beaucoup plus d'une mise en page de type HTML que de Xaml avec des bitmaps découpés ou retaillés. Toutefois la notion de contrôles sous Xaml se retrouve dans celle de "widget" ou de "view" sous Android et la façon d'imbriquer des conteneurs est proche de la stratégie XAML (grille, panels etc).

Java-Based

Android est un OS qui a choisi d'utiliser nativement un Java (qui normalement ne doit pas s'appeler comme ça suite à un procès lancé et gagné par Oracle). Pour le développeur C# c'est à 90% gagné puisque la syntaxe de C# est plus qu'inspirée par celle de Java. Mais c'est encore mieux quand on utilise la stratégie de développement cross-plateforme que je vous propose puisqu'elle passe par Xamarin qui nous offre un C# natif Android d'un niveau excellent (intégrant Linq, les expressions lambda, await/async…).

Toutefois le fait que le langage natif soit du Java et non un C obscure et illisible, cela permet facilement de lire les documentations qu'on trouve sur le Web et de transposer parfois par un simple copie/coller les exemples qu'on peut voir ici et là …. Très pratique donc ! Dans la pratique Android est bien plus sympathique qu'iOS.

Les attributs de mise en page

Chaque classe de Layout possède une classe interne appelée `LayoutParams` qui définit les paramètres XML généraux de mise en page. Dans un fichier XML de mise en page ce sont des attributs qu'on retrouve sous la forme sous des noms qui commencent systématiquement par `android:layout_xxxx` et qui traitent le plus fréquemment de la taille de l'objet ou de ses marges.
Voici un exemple de mise en page minimaliste XML Android :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Les classes de layout définissent bien entendu d'autres attributs en relation directe ou non avec leur mise en page, leur forme, leur positionnement, etc.

Dans le code exemple ci-dessus on peut voir la description d'une page très simple qui s'ouvre sur une balise `LinearLayout` (l'équivalent d'un `StackPanel` XAML) contenant un `TextView` (un TextBlock XAML) et un `Button` (devinez !).
Chaque objet d'UI possède des attributs, dont la poignée de paramètres généraux évoqués plus haut
comme `android:layout_width` ou `android:layout_height` (hauteur et largeur de l'objet).

Les autres attributs sont donc spécifiques à la classe (comme `text` pour le `TextView`), ces attributs ne commencent pas par "`layout_`" puisqu'ils ne concernent pas directement la mise en page.

Les attributs utilisés couramment

La Taille (Size)

`android:layout_height` et `android:layout_width` permettent de fixer la taille de l'objet (hauteur et largeur).

Les valeurs de tailles, marges et autres dimensions s'expriment de plusieurs façons sous Android. Soit en pouces (inch) en millimètres (mm) ou d'autres unités plus pratiques pour gérer les différentes résolutions et densités d'écran qu'on trouve sous Android.

Le système d'unités

	Ecrans basse résolution	Ecrans haute résolution même taille
Taille physique	1,5 pouce	1,5 pouce
Points par pouces (dpi)	160	240
Pixels (largeur*dpi)	240	360
Densité (base = 160)	1,0	1,5
Density Independant Pixels (dip ou dp ou dps)	240	240
Scale Independant pixels (sip ou sp)	dépend du paramétrage de la taille de fonte par le user	Idem

Le tableau ci-dessus montre les relations entre les différentes unités utilisées couramment pour les mises en page sur la base d'une taille physique et d'une densité choisies arbitrairement (1,5 pouce et 160/240 dpi).

Si on peut utiliser des tailles en pixels, en millimètres ou en pouces ce ne sont bien entendu pas des unités réellement utilisées car elles offrent un rendu bien trop différents selon les tailles réelles des écrans et surtout de leurs densités.

On préfère utiliser les *Density Independant Pixels*, pixels indépendant de la densité, qui permettent des mises en page ayant le même aspect quelque que soit la résolution de la machine. Les chiffres s'expriment alors suivi du suffixe "dp".

Les *Scale independant Pixels* sont moins utilisés mais peuvent être très utiles dans une stratégie se basant sur le réglage de la taille de fonte qu'a choisi l'utilisateur. On s'adapte alors non plus à la densité de l'écran mais à la taille de cette fonte. Un utilisateur ayant une mauvaise vision et réglant son appareil en mode fonte de grande taille verra alors l'application qui utilise des unités "sp" suivre le même facteur de grossissement, ce qui est indispensable si on cible certains clients/utilisateurs.

Les "dp" restent toutefois les plus utilisées. Dans le tableau ci-dessus on montre le rapport entre toutes ces unités en prenant en exemple deux écrans, l'un de faible densité, l'autre ayant une haute densité. On se place dans le cas d'un objet qui mesure 1,5 pouce (taille physique) qui sera affiché à l'identique sur les deux écrans. Sa taille physique sera donc de 1,5 pouce dans les deux cas.

Le premier écran a une densité de 160 dpi (dots per inch, points par pouce), le second 240 dpi. On est loin encore des résolutions de type Retina sur le haut de gamme Apple ou Samsung mais le tableau pourrait être étendu à l'infini, les deux exemples suffisent à comprendre le principe.

Sur le premier écran l'objet sera dessiné en utilisant 240 pixels, sur le second il faudra 360 pixels pour remplir le même pouce et demi.

Android pose comme facteur de densité la valeur de 1.0 pour le premier cas (160 dpi). L'écran de plus haute densité possède donc un facteur de 1,5 ici.

Pour couvrir le pouce et demi de notre objet, si on utilise les DIP il faudra ainsi "240dp" pour dessiner l'objet. La magie s'opère quand on regarde la taille pour l'écran haute densité: "240dp" aussi… Android se charge alors de traduire les "dp" en pixels réels pour que l'objet mesure toujours 1,5 pouce.

En mode SIP on voit qu'il est difficile ici de prévoir la taille puisque celle-ci dépend de la taille de la fonte choisie par l'utilisateur. Ce mode, comme je le disais plus haut, n'est pas le plus utilisé.

On se rappellera ainsi que les mises en pages Android se font en utilisant principalement les DIP, des mesures notées "XXXdp".

Android possédait un canvas similaire à celui de Xaml avec positionnement absolu mais il est *deprecated*. S'il ne l'est pas en Xaml c'est un conteneur qu'on utilise très peu pour les mêmes raisons de rigidité. Sous WPF ou Silverlight, et même WinRT, on préfère utiliser des `Grid` ou des `StackPanel`, et là on retrouve des équivalents presque copie-conforme (`TableLayout`, `LinearLayout`…). Les stratégies de placement se ressemblent donc beaucoup.

Les valeurs spéciales

Il existe quelques valeurs "spéciales" pour les tailles. Elles sont très souvent utilisées car elles permettent d'éviter de donner des mesures précises et donc autorisent une plus grande souplesse d'adaptation aux différents form factors :

`match_parent` : remplir totalement l'espace du conteneur parent (moins le *padding*). Renommé dans les dernières versions d'Android (s'appelait avant `fill_parent`).

`wrap_content` : utiliser la taille "naturelle" de l'objet (plus le *padding*). Par exemple un `TextView` occupera la taille nécessaire à celle du texte qu'il contient.

On peut aussi fixer une taille à zéro et utiliser l'attribut `android:layout_weight` qui permet de fixer une dimension de façon proportionnelle, un peu comme les lignes ou colonnes dans un Grid Xaml en utilisant la notation "étoile".

L'alignement

Autre attribut de base pour mettre en page un objet, outre sa taille, son alignement est essentiel.

On utilise `android:layout_gravity` pour déterminer comment l'objet est aligné dans son conteneur.

`android:gravity` ne doit pas être confondu avec le précédent attribut, ici il s'agit de fixer comment le texte ou les composants dans l'objet sont alignés.

Les valeurs possibles pour ces attributs sont de type : `top`, `bottom`, `left`, `right`, `center_vertical`, `center_horizontal`, `center`, `fill_vertical`, `fill_horizontal`, `fill`, `clip_vertical`, `clip_horizontal`.
Selon le cas les valeurs peuvent être associées avec un symbole "pipe" `|` (Alt-124).

Les marges

Encore une fois très proche de la notion de même nom sous Xaml, les marges des widgets Android servent exactement à la même chose : mettre un espace autour de l'objet. Sur un côté, deux côtés, trois ou les quatre.

Le système d'unité utilisé est le même que pour la hauteur et la largeur.

`android:layout_marginBottom`, `android:layout_marginTop`, `android:layout_marginLeft` et `android:layout_marginRight` s'utilisent selon les besoins, ensemble ou séparément.
Les unités négatives sont autorisées (par exemple `"-12.5dp"`).

Le padding

Encore une notion courante qui ne posera pas de problème aux utilisateurs de Xaml. Si la marge est l'espace qui se trouve autour du widget, le padding est celui qui se trouve entre sa bordure extérieure (rangez vos sabres laser!) et son contenu.

`android:paddingBotton`, `android:paddingTop`, `android:paddingLeft` et `android:paddingRight` s'utilisent indépendamment selon les besoins.
Les valeurs utilisent les mêmes unités que les marges, la hauteur et la largeur.

L'ID

Tous les widgets peuvent avoir un `ID`, on retrouve donc naturellement cette propriété dans la liste des propriétés communes à tous les widgets. Tout comme sous Xaml avec "`x:Name`", l'ID n'est pas obligatoire mais il est indispensable si le code veut accéder au contrôle. Avec le binding ajouté par MvvmCross les ID ne sont généralement pas utilisés, tout comme les "`x:Name`" en Xaml. Mais dans le mode de programmation natif il faut savoir qu'à la différence de Xaml la déclaration de l'ID dans le code XML ne créée par automatiquement une variable de même nom pointant le widget dans le "code-behind" de l'activité Android. Il faut alors utiliser une technique qui consiste lors de l'initialisation de l'activité à faire un `FindViewByID` pour récupérer un pointeur sur le widget. On se sert parfois de ce genre de chose en ASP.Net par exemple et sous Xaml c'est le compilateur qui s'occupe de créer automatiquement ces déclarations. Le principe est donc le même que sous Xaml mais en moins automatisé.
La déclaration d'un ID dépend de si on référence un ID existant ou bien si on souhaite déclarer l'ID lors de sa première apparition.

Pour déclarer un ID on utiliser l'attribut `android:id="@+id/btnCancel"` par exemple pour un bouton qui portera le nom de "`btnCancel`". Cette notation indique

au compilateur de créer l'ID et de la placer dans le fichier des ressources afin d'y créer un véritable ID. En effet, Android ne gère que des ID numériques en réalité.
Pour éviter d'avoir à travailler avec de tels identifiants peu pratiques, le fichier `R.java` (qu'on retrouve sous Xamarin d'une façon similaire mais sous un autre nom) contient la liste des ID "humains" déclarés dans le fichier Xml ainsi qu'une véritable valeur numérique associée. Cela permet dans le code de référencer un widget par son nom "humain" tout en accédant à son véritable ID numérique. Dans `R.java` (ou équivalent Xamarin) l'ID est donc déclaré comme une simple constante `integer` dont la valeur est choisie par le compilateur.

Pour référencer un widget sans déclarer son ID on utilise la syntaxe suivante : `"@id/btnCancel"` (si on suppose une référence au bouton `btnCancel`).
Cette syntaxe ne s'utilise qu'à l'intérieur d'un fichier Xml puisque dans le code Java ou C# on peut directement accéder à l'objet par son ID "humain".
L'intérêt de référencer un widget dans un fichier Xml s'explique notamment par la présence de certains modes de mise en page qui utilisent des positionnements relatifs (`RelativeLayout`). Dans ce cas les contrôles sont placés au-dessus, en-dessous, à gauche ou à droite relativement à un autre contrôle dont il faut donner l'ID.

Couleurs

Les couleurs sont aussi des attributs de base. On trouve le `android:background` qui peut être pour tout contrôle aussi bien une couleur qu'une référence à une image placée en ressources.

La couleur du texte, si le contrôle en affiche, se modifie avec `android:textColor`.
C'est le cas des boutons ou des `TextView` (équivalent au `TextBlock` Xaml) par exemple.
Les couleurs s'expriment indifféremment avec les formats suivants :

```
"#rrggbb", "#aarrggbb", "@color/colorName"
```

Les valeurs sont en hexadécimal. Dans le premier cas on définit une couleur pleine, dans le second cas on définit une couleur ainsi que son canal alpha pour la transparence, comme en Xaml. Dans le dernier cas on pointe vers le nom d'une couleur définie dans l'arborescence du projet au sein d'un fichier Xml.

Gestion du clic

Le clic est aussi partie intégrante des propriétés de base, tout objet peut donc recevoir et gérer un appui du doigt. Le gestionnaire se définit par `android:onClick` avec le nom d'une méthode publique dans l'activité qui prend une View comme argument et qui retourne `void`. Toutefois avec MvvmCross on utilise plutôt un `MvxBind` pour se lier à un `ICommand` dans le ViewModel.

Le conteneur linéaire LinearLayout

Ce conteneur est très utilisé comme son équivalent Xaml le `StackPanel`. Il fonctionne de la même façon, peut être horizontal ou vertical et sert à empiler des contrôles les uns derrière les autres ou les uns haut de dessus de l'autre.
En le combinant avec lui-même, tout comme en Xaml, on peut construire des mises en page dynamiques très efficacement. Un `LinearLayout` vertical contenant des `LinearLayout` horizontaux forme une sorte de tableau totalement ajustable par exemple.

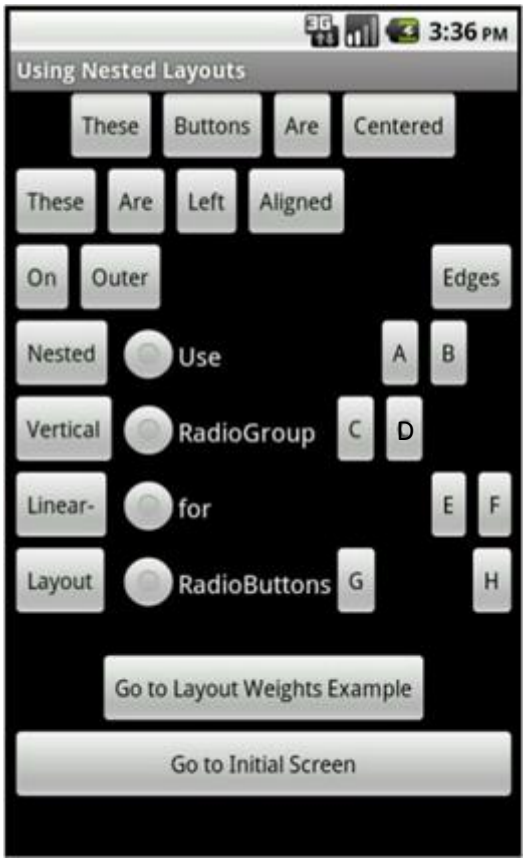
Son attribut principal est `android:orientation` qui prend les valeurs horizontal ou vertical. Le mode horizontal est la valeur par défaut (inutile de déclarer l'attribut dans ce cas).

L'attribut `android:gravity` est aussi très souvent utilisé dans un `LinearLayout` car il permet de définir comment les widgets contenus sont alignés. Les valeurs acceptées sont nombreuses : `top`, `bottom`, `left`, `right`, `center_vertical`, `center_horizontal`, `center`, `fill_vertical`, `fill_horizontal`, `fill`, `clip_vertical`, `clip_horizontal`.

La déclaration d'un conteneur linéaire ressemble à cela :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<!-- Widgets and nested layouts -->

</LinearLayout>
```

Voici un exemple de mise en page utilisant ce conteneur :

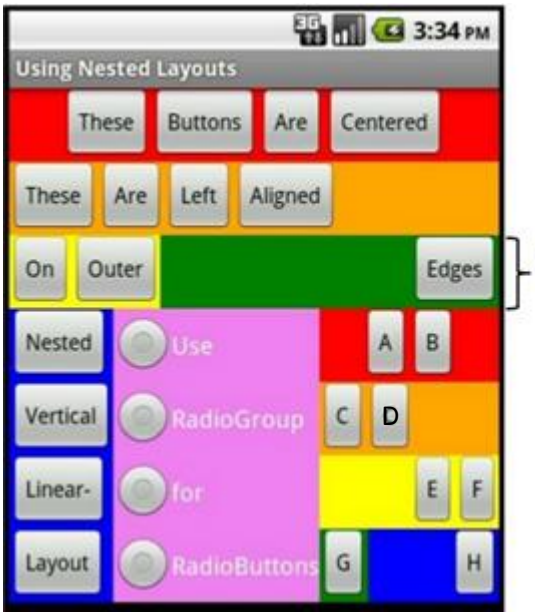


Cette mise en page s'explique par les placements suivants (les couleurs de fond sont là pour délimiter les zones des différents `LinearLayout` utilisés pour vous faciliter la lecture et n'ont aucune vocation d'exemple visuel – c'est réellement atroce ne faite jamais une UI comme celle-là et si vous le faites ne dites surtout pas que vous connaissez Dot.Blog, le département d'Etat niera avoir eu connaissance de vos agissements – et je placerai personnellement un contrat sur votre tête  !!!) :

Horizontal LinearLayout with gravity of center_horizontal.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal"
    android:background="@color/color_1">
    <Button android:text="These"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:text="Buttons"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:text="Are"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:text="Centered"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

Horizontal LinearLayout with gravity of left. Otherwise almost same as first row.



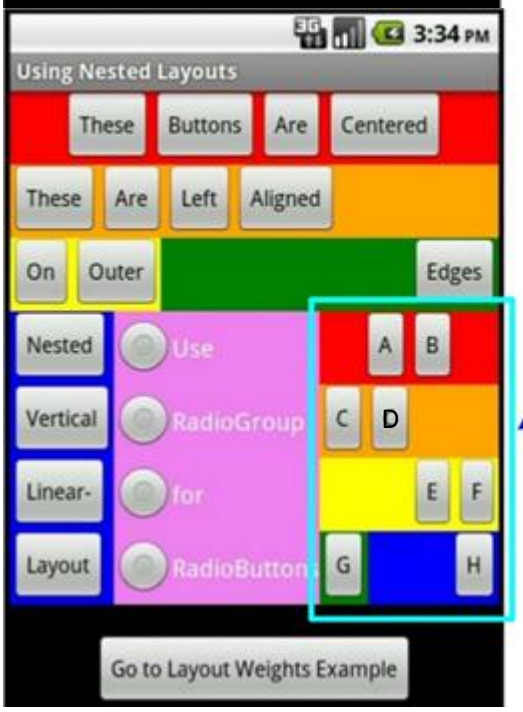
Horizontal LinearLayout.

That Layout then contains two more horizontal LinearLayouts. The first (yellow) has android:layout_width of "wrap_content" and android:gravity of "left". The second (green) has android:layout_width of "match_parent" and android:gravity of "right".

Horizontal LinearLayout.

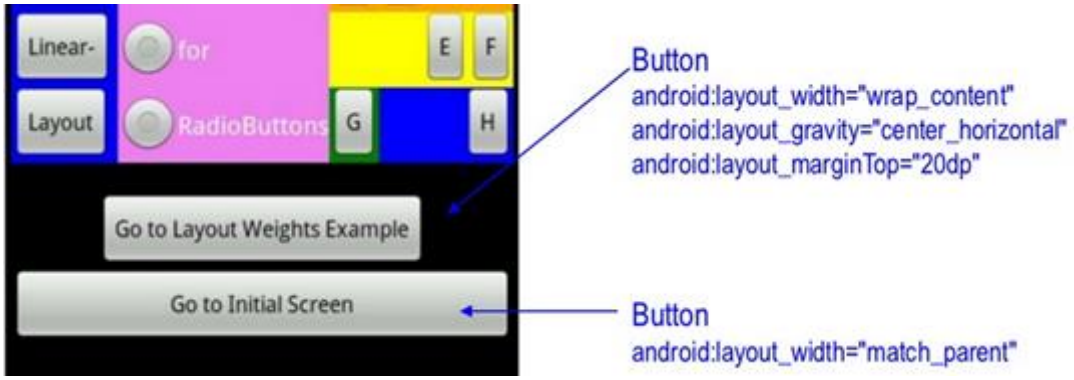
That Layout then contains three vertical nested layouts. The first (blue) is a LinearLayout with android:orientation of "vertical" and four Buttons inside. The second (violet) is a RadioGroup (similar to LinearLayout but specific to enclosing RadioButtons and making them mutually exclusive), also with android:orientation of "vertical". It has four RadioButtons inside. The third is a LinearLayout with android:orientation of "vertical" and four nested LinearLayouts inside (details on next slide).

The first two columns (nested layouts) have android:layout_width of "wrap_content", and the third has android:layout_width of "match_parent".



Vertical LinearLayout.

That Layout then contains four horizontal nested layouts. The first (red) has android:gravity of "center_horizontal". The second (orange) has android:gravity of "left". The third (yellow) has android:gravity of "right". The fourth contains two further nested horizontal LinearLayouts. The first (green) has android:layout_width of "wrap_content" and android:gravity of "left". The second (blue) has android:layout_width of "match_parent" and android:gravity of "right".



Définir les couleurs

Comme nous l'avons vu, les couleurs se définissent le plus simplement du monde par un code hexadécimal sur 6 ou huit chiffres selon qu'on intègre ou non le canal alpha.

Bien entendu cette façon de faire ne peut se comprendre que dans une démo… Toute application bien faite et donc bien designée se doit d'utiliser des feuilles de style facilement modifiable et centralisées pour assurer l'homogénéité visuelle indissociable d'une bonne UI. Chaque plateforme propose sa façon de créer des feuilles de style. Sous Xaml on utilise des templates de contrôle et des dictionnaires. Sous Android les choses marchent de façons différentes mais l'esprit reste similaire.

Tout comme on peut définir des couleurs dans un dictionnaire de ressources Xaml on peut le faire dans un fichier Xml sous Android, fichier qui sera placé dans l'arborescence des ressources du projet.

La syntaxe d'un tel fichier est fort simple :

```
<resources>
<color name="maCouleur">#rrggbb</color>
…
</resources>
```

Par convention les définitions de couleurs dans un projet Xaml se place dans un fichier appelé `colors.xml` lui-même placé dans l'arborescence `res/values/`
Les fichiers Xml de définitions des écrans peuvent référencer les ressources couleur par leur nom.

Si on définit un fichier de couleur `res/values/colors.xml` de la façon suivante :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="color_1">#ff0000</color>
    <color name="color_2">#ffa500</color>
    <color name="color_3">#ffff00</color>
    <color name="color_4">#008000</color>
    <color name="color_5">#0000ff</color>
    <color name="color_6">#ee82ee</color>
</resources>
```

On peut alors utiliser les définitions créées de cette façon dans un autre fichier Xml :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="…"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:background="@color/color_1">
        ...
     </LinearLayout>
```

Localisation et fichiers de valeurs

Les couleurs comme tout fichier de valeurs se trouvant dans le répertoire `res/values` peuvent être définies selon la valeur de la locale (le code langue) de l'utilisateur.

Il suffit pour cela de créer une copie du fichier placé dans `res/values` et de le placer dans un sous-répertoire de même niveau que `values` mais incluant le code de la locale, par exemple `res/values-fr`.

Le fichier se trouvant dans le nouveau répertoire peut ainsi être modifié et Android le chargement automatiquement en place et lieu de celui situé dans values (si la locale correspond, sinon c'est le fichier dans `values` qui est chargé par défaut).

On peut ainsi définir des fichiers `strings.xml` pour les chaînes de caractères, des fichiers de couleurs, des données localisées, etc.

Ce système fonctionnant sur des noms de répertoire est un peu “frustre” mais redoublement simple et efficace. On voit ici la différence d’approche entre un OS au départ ultra optimisé pour des petites machines peu puissantes (choisir un répertoire ou un autre est très rapide pour l’OS) et un OS plus sophistiqué comme Windows Phone mais qui réclame parfois des trésors de génie logiciel pour arriver à créer des localisations… L’approche d’Android est toutefois dangereuse et réclame une organisation sans faille des équipes… car comme tout se joue dans le nom des répertoires mais que les fichiers eux-mêmes portent exactement le même nom mais avec un contenu très différent toute erreur de manipulation, de copie ou autre peut être fatale et faire perdre un fichier bêtement écrasé par un autre… Une gestion de version est fortement conseillée !

Deux poids deux mesures !

Gérer des tableaux ou des objets dont les dimensions sont des pourcentages plutôt que des valeurs numériques précises est un besoin vieux comme Html couvert de diverses façons par des systèmes comme Xaml ou Android.

Sous Xaml la définition des `Rows` et `Columns` dans une `Grid` autorise, via une syntaxe parfois obscure pour le débutant, de définir les hauteurs et les largeurs par des pourcentages. Android propose la même chose mais en utilisant une déclaration séparée, clarifiant un peu les choses.

L’idée est donc de pouvoir assigner des nombres à l’attribut `android:weight`, le rendu étant proportionnel à la somme de ces nombres quels qu’ils soient. En général, à moins d’être maso, on se débrouille pour que la somme de ces nombres soit égale à 10 ou à 100 (le plus souvent) mais les valeurs farfelues sont aussi acceptées (comme sous Xaml) bien que déconseillées…

Ainsi, si trois objets ont une largeur respective (un poids) de `10`, `40` et `50`, la somme valant `100`, les trois objets occuperont respectivement 10%, 40% et 50% de la largeur disponible du conteneur. Mais on peut obtenir le même résultat avec les trois poids suivants : 1, 4 et 5, ou bien 4.8 19.2 et 24 (somme = 48, 4.8 = 10% de 48, 19.2 = 40% de 48 etc). Je dis ça pour illustrer le propos, n’utilisez jamais des valeurs aussi farfelues !

Le procédé est en deux étapes : la première consiste à mettre la hauteur ou la largeur à zéro et ensuite d’assigner une valeur au poids (*weight*). Le poids est global, si seule la hauteur est mise à zéro, seule la hauteur sera proportionnelle, idem pour la largeur, mais si les deux valeurs sont à zéro l’effet du poids se fera sentir sur les deux paramètres.

Voici un exemple de code utilisant ce type de mise en page :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="…"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView android:layout_width="match_parent"
              android:layout_height="0dp"
              android:layout_weight="30"
              .../>
    <TextView android:layout_width="match_parent"
              android:layout_height="0dp"
              android:layout_weight="30"
              .../>
    <TextView android:layout_width="match_parent"
              android:layout_height="0dp"
              android:layout_weight="40"
              .../>
</LinearLayout>
```

Tout est relatif !

Android propose de nombreux modes de mise en page, de conteneurs utilisant des règles différentes, tout comme Xaml. Le conteneur que nous allons maintenant voir est un peu spécial et n'a pas d'équivalent sous Xaml, c'est le conteur relatif ou `RelativeLayout`.

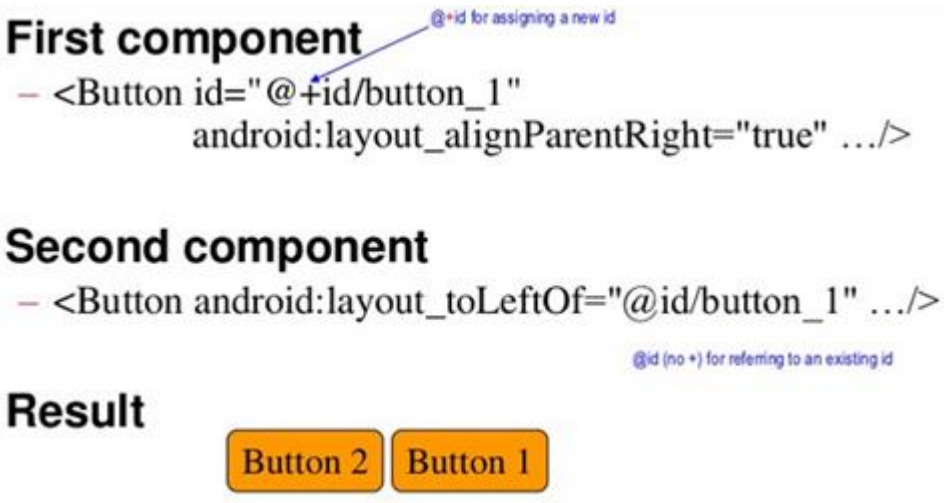
Bien entendu tous les conteneurs pouvant s'imbriquer, on peut construire des affichages très complexes mixant les effets et bénéfices de tous les modes disponibles. Savoir rester simple et ne pas hésiter à refaire un Design qui devient trop compliqué est aussi important que savoir *refactorer* un code C# qui devient confus.

L'idée derrière le conteneur relatif est qu'une fois qu'on a attribué un identifiant à au moins un widget il devient possible de placer les autres par rapport à ce dernier (au-dessus ou en-dessous par exemple). On a bien entendu le droit d'attribuer un identifiant à plusieurs widgets et de faire des placements relatifs les uns par rapports aux autres. Attention, comme je le disais, ce mode de construction peut devenir rapidement impossible à maintenir, une sorte de code spaghetti au niveau de l'UI…

Les attributs les plus importants dans une mise en page relatives sont ceux qui concernent l'alignement avec le conteneur et l'alignement avec d'autres contrôles situés dans le même conteneur.

`android:layout_alignParentBottom` (et en variante `Top`, `Left`, `Right`) ainsi que `android:layout_CenterInParent` (et `CenterHorizontal`, `CenterVertical`) concernent l'alignement par rapport au conteneur. Ces attributs sont des booléens prenant la valeur `true` ou `false`.

`android:layout_alignBottom` (et `Top`, `Left`, `Right`) ainsi que `android:layout_toLeftOf` (et `toRightOf`) ou `android:layout_above` (et `below`) concernent l'alignement d'un contrôle par rapport à une autre dans le conteneur. Ces attributs prennent tous une valeur qui est l'ID du contrôle de référence (par exemple `android:layout_alignTop="@id/id_du_widget"`)
Un placement simple de deux boutons dont l'un est la référence positionnelle donnerait cela (décomposé) :



La déclaration du premier bouton utilise un "`@+id`" pour créer l'ID "`button_1`". Ce bouton est aligné à droite dans son parent.

La déclaration du second bouton utilise un placement relatif au premier, la référence à l'ID ne contient pas de symbole "`+`" puisqu'il ne s'agit pas de créer un ID mais d'en référencer un qui existe. Le placement est de type "à la gauche de".

Le résultat visuel est montré par les deux boutons orange, le bouton 1 est à droite et le 2 est à la gauche du 1er.

A table !

On dira ce qu'on voudra mais bien souvent une bonne vieille table Html c'est bien pratique ! Surtout si elle a été un peu modernisée. C'est finalement ce qu'est une `Grid` Xaml avec ses définitions de lignes et colonnes.

Android nous propose un conteneur ayant le même but : le `TableLayout`.
L'idée est simple, placer les widgets ou des conteneurs imbriqués dans une grille (sans bordure).

Comme dans Html le nombre des colonnes et des lignes est déduit automatiquement en fonction du contenu. C'est plus souple que la Grid Xaml de ce point de vue. Les contrôles sont placés généralement dans des `TableRow`.
Les attributs les plus importants de ce conteneur :

`android:stretchColumns`. Sa valeur est constituée d'un index ou d'une liste d'index séparés par des virgules. Cet index ou cette liste d'index spécifie la colonne ou les colonnes qui doivent être stretchées au plus large si la table est rétrécie en deçà de celle de son parent. Les index sont 0-based.

`android:shrinkColumns`. Dans le même principe mais cette fois pour lister les colonnes qui doivent être rétrécies si la table est plus grande que le parent.

`android:collapseColumns`. Définit les colonnes qui disparaissent totalement. En général cette liste est manipulée codée code en fonction des choix de l'utilisateur plutôt que figée au design (une colonne invisible posée au design "en dur" est une colonne qui ne sera jamais visible, alors autant ne pas la coder…).

La TableRow

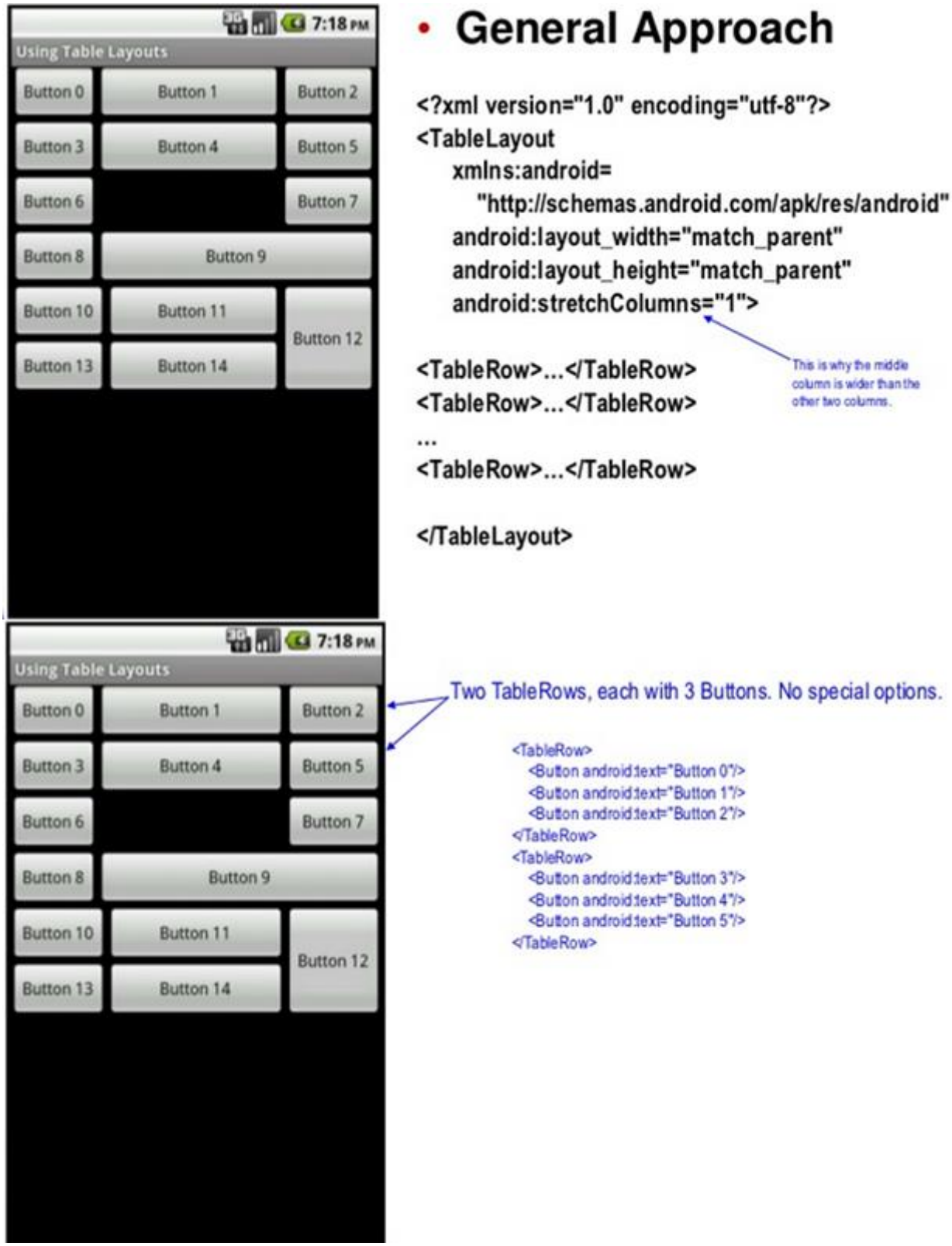
Elle sert à définir une ligne dans un `TableLayout`. Techniquement des éléments peuvent exister entre les `TableRow` mais le résultat est bien plus propre en utilisant `android:layout_span`.
Les attributs essentiels de la Row :

`android:layout_column`. Normalement une table se remplit de la gauche vers la droite en fonction de l'ordre des éléments placés dans les colonnes. En spécifiant une valeur à cet attribut on peut placer un élément dans n'importe quelle colonne.

`android:layout_span`. Indique le nombre de colonnes à regrouper de la même façon qu'un `colspan` en Html.

On notera qu'il n'y a pas d'équivalent au `rowspan` Html, il faut donc utiliser des tables imbriquées pour obtenir le même résultat.

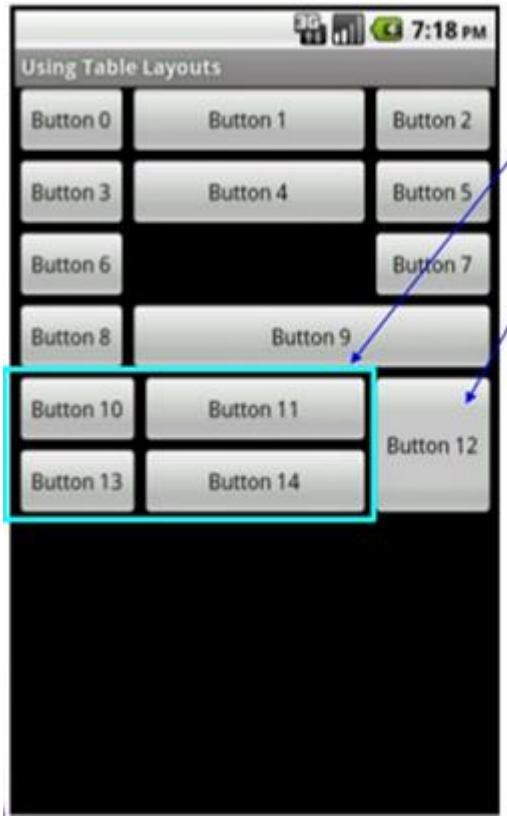
L'Approche générale est résumée ici :



Button 7 uses android:layout_column="2". So, there is no entry at all for the middle column.

Button 9 uses android:layout_span="2".

```
<TableRow>
  <Button android:text="Button 6"/>
  <Button android:text="Button 7"
          android:layout_column="2"/>
</TableRow>
<TableRow>
  <Button android:text="Button 8"/>
  <Button android:text="Button 9"
          android:layout_span="2"/>
</TableRow>
```



A nested table. Uses android:layout_span="2" so that it straddles two columns of the main table. Uses android:stretchColumns="1" so that the second column fills available space.

A Button. android:layout_height is match_parent so that it is the same height as table to its left. There is no option similar to HTML's colspan, so nested tables are needed to achieve this effect.

```
<TableRow>
  <TableLayout xmlns:android="..."
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_span="2"
          android:stretchColumns="1">
    <TableRow>
      <Button android:text="Button 10"/>
      <Button android:text="Button 11"/>
    </TableRow>
    <TableRow>
      <Button android:text="Button 13"/>
      <Button android:text="Button 14"/>
    </TableRow>
  </TableLayout>
  <Button android:text="Button 12"
          android:layout_height="match_parent"/>
</TableRow>
```

Les autres conteneurs

Même si les conteneurs que nous venons de voir sont les plus utilisés et suffisent à la majorité des applications, Android en propose d'autres qu'il faut connaître.

Le premier est l'équivalent du `Canvas` Xaml, c'est l'`AbsoluteLayout` qui utilise, comme son nom l'indique, des positions absolues. Totalement inadapté au foisonnement des résolutions et des densités, ce conteneur est aujourd'hui *deprecated*.

Le `FrameLayout` est un conteneur qui ne prend qu'un seul enfant. Il est généralement utilisé avec le `TabHost`. Sinon il est utilisé en interne par d'autres conteneurs.

Le `TabHost` est le plus intéressant des conteneurs non vus ici. C'est un contrôle avec des tabulations qui permet de passer d'une activité à une autre. Il mérite plus d'explications qui n'auraient pu tenir ici ce n'est donc pas le manque d'intérêt qui le pousse en fin d'article !

C'est le cas aussi du `ListView` et du `GridView`. Toutefois concernant le `ListView`, une sorte de `ListBox` Xaml, on préfèrera la `MvxListView` fournie par MvvmCross qui possède des propriétés bindables plus adaptées à notre stratégie de développement. La `MvxListView` a été traitée de nombreuses fois dans la série récente de vidéos (voir en introduction de cet article).

Conclusion

Cet article avait pour principal objectif de vous donner un aperçu rapide et condensé des bases de la mise en page sous Android. Il reste des milliers de choses à dire, à présenter chaque widget, ses propriétés, etc… C'est l'œuvre d'un livre plus que d'un billet de blog !

J'ai certes habitué mes lecteurs aux billets à rallonge et aux papiers dépassant les 100 pages, donc à des livres qui ne disent pas leur nom… Mais il faut bien en laisser un peu pour les prochains billets !

Cross-plateforme : images avec zones cliquables avec MonoDroid/Xamarin vs C#/Xaml

Comprendre les différences entre les OS est essentiel lorsqu'on veut créer des applications cross-plateformes. Petit exercice pratique avec Xamarin 2.0 : créer des images cliquables avec Xamarin.MonoDroid. Ceci sera le prétexte pour aborder Android et Xamarin 2.0 par un exemple réel.

C#/XAML

Je n'écrirais jamais assez de billets pour dire tout le bien que je pense de C# et de XAML, à quel point ce couple est parfait, agréable et d'une puissance à faire pleurer tous les éditeurs de langages du monde pour encore un bon moment…

Blend une UI de SF pour créer du rêve…

Faire des zones cliquables sur une image ou des fonds vectoriels en XAML est un jeu d'enfant. Avec le Visual State Manager et les Behaviors pas même besoin de code C# pour créer des visuels interactifs. Je ne ferais pas l'offense à mes lecteurs fidèles d'expliquer comment et je renvois les autres aux plus de 600 billets qui traitent majoritairement de XAML d'une façon ou d'une autre…

Donc avec C# et XAML, créer une zone cliquable sur une image est un jeu d'enfant. Parce que XAML est hyper évolué et vectoriel. C'est un véritable langage de description d'UI.

Des avantages de la rusticité

Mais il n'en va pas de même sous d'autres OS plus "rustiques" au niveau description des UI. D'ailleurs, comme me le faisait remarquer Valérie (mon infographiste) il y quelques heures, dans "rustique" il y a "russe"… Et c'est vrai que passer de Windows

Phone à Android ou iOS donne un peu l'impression de passer de la navette américaine aux bricolages ingénieux des cosmonautes russes…

Un russe a transformé sa voiture en ajoutant des chenilles. Home made.

J'ai toujours adoré l'esprit Russe, cette façon de faire aussi bien et parfois mieux avec trois fois rien. Android c'est un peu ça, même si c'est américain. Mais d'iOS ou d'Android s'il fallait dire lequel est le plus "rustique" pour un développeur vous seriez étonné du résultat…

Sous Android, et même avec Xamarin, point de vectoriel ni de databinding two-way ni rien de tout cela. XAML n'existe pas. Mais en retour : des astuces, des conventions (noms de répertoires par exemple pour les différentes capacités de l'unité mobile) et une robustesse basée sur un Linux complété d'une JVM, la célèbre Dalvik (qui sonne un peu comme Dalek, puissantes et rustiques machines que les amis de la SF sauront reconnaitre, avec un look presque… Russe !).

C'est rustique mais ça marche. Et avec un peu d'imagination, c'est même joli, fluide, et, en tout cas, cela semble séduire une masse impressionnante et toujours grandissante de gens dans le monde entier, raison principal de notre propre intérêt !

N'exagérons rien non plus !

Pour être franc je me dois de dire que j'ai forcé le trait en brossant rapidement ce portrait d'Android. Pour être juste cet OS est plein d'astuces intelligentes et a su évoluer rapidement pour prendre en charge l'explosion des form factors différents et parfois déroutants (dont les phablets) là où les autres se contentent de supporter un modèle voire deux et se tâtent encore pour savoir s'il faudrait envisager de faire de mieux… L'esprit bricoleur à la russe, Google a répondu "chiche!" à toutes les idées de tous les constructeurs et Android supporte une quantité redoutable de machines toutes différentes. Et ça marche très bien. C'est un exploit qu'il faut noter.

Alors rustique, oui peut-être dans l'esprit "russe-tic", mais évolué et performant.

Android est peut-être le Soyouz des OS, mais l'un comme l'autre s'envolent vers les étoiles alors que les autres restent au sol… Pour un ingénieur ce qui compte, c'est le résultat !

CubeSat, comme le projet PhoneSat de la Nasa, des satellites fonctionnant sous Android

Dans les trois PhoneSat's lancés par la Nasa dernièrement, trois téléphones sous Android, 2 HTC One et un Samsung Nexus S. Tous ont rempli leur mission parfaitement et une seconde vague est prévue pour bientôt. Android est peut-être finalement le plus Hi-Tech des OS mobile…

Des images cliquables…

On revient 5 minutes sur terre car notre propos était de gérer des images cliquables. On a rapidement vu qu'en XAML les choses étaient évidentes. Mais sous Android ça peut devenir plus *tricky*.

Les UI sont créées à l'aide d'un langage balisé qui a quelques ressemblances avec XAML. On y retrouve un formalisme XML avec des balises qui en contiennent d'autres, ce qui créée l'arbre visuel. A l'intérieur des balises on retrouve des noms de classes qui seront instanciées au runtime et des propriétés qui serviront à initialiser ces instances. C'est vraiment très proche.

La seule différence notable, mais elle est de taille, c'est que XAML est vectoriel alors que l'UI de Android est bitmap. XAML possède aussi le data binding et les extensions de balises, mais je me suis rendu compte que finalement beaucoup de développeurs le trouvait complexe et je ne parle pas de Blend que bien plus encore n'arrivent pas à "piger", même si je le considère comme le summum des EDI pour la conception visuelle d'une application. Après tout, le succès d'Android est peut-être qu'il est plus simple tout court et donc à comprendre aussi…

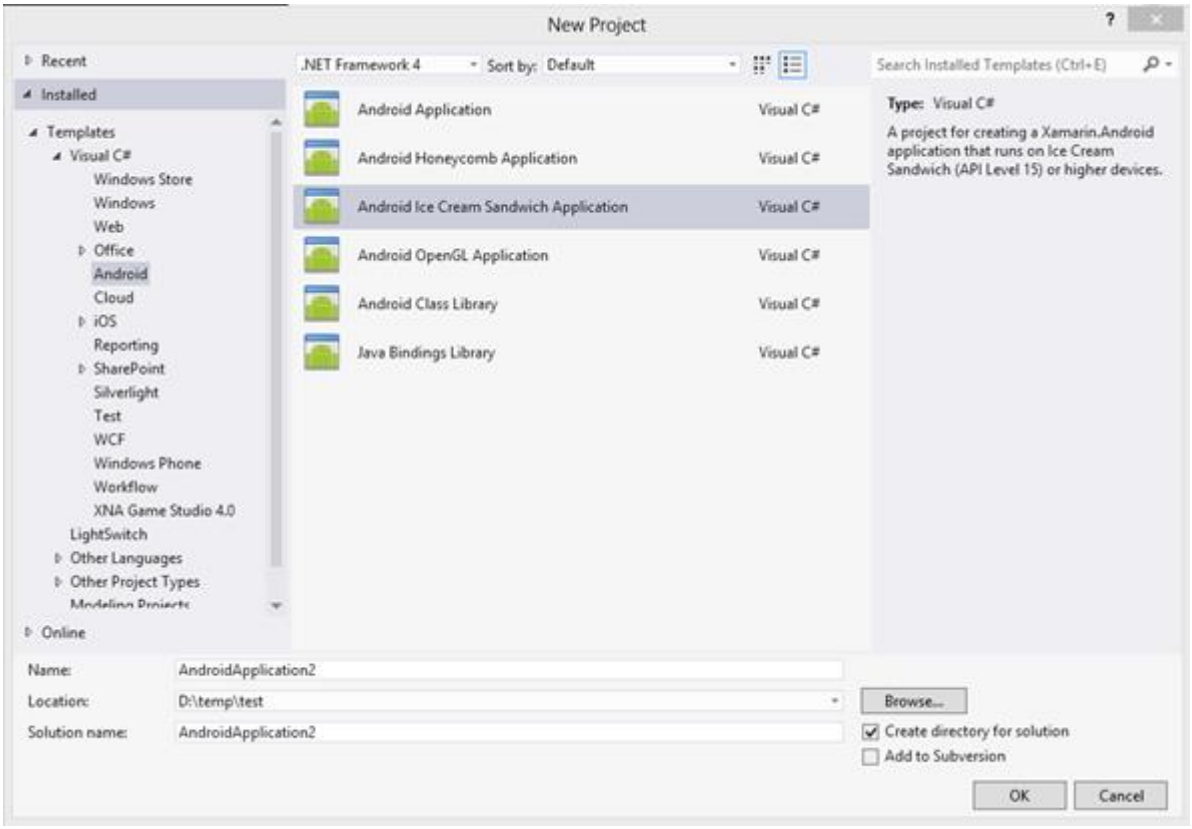
Dans un tel environnement on retrouve des techniques de mise en page plus proche de HTML que de WPF ou Silverlight. Un monde dans lequel les images jouent un rôle plus grand, où il faut fournir ces images dans toutes les résolutions supportées pour éviter la pixellisation, etc. Avantage : Android a été conçu dans un esprit minimaliste pour supporter des smartphones à la puissance vraiment réduite à sa sortie, alors quand il tourne sur des machines récentes de type S3 à 4 cœurs, c'est une bombe !

Xamarin Studio ou Visual Studio ?

Pour développer notre exemple nous avons le choix entre les deux EDI, VS étant un EDI de grande qualité dans lequel j'ai aussi ma gestion de version et ReSharper, autant le choisir. Mais les dernières versions de Xamarin Studio sont vraiment très agréables et complètes. On est loin de la … rusticité (encore elle !) de MonoDevelop qui a servi de base à cet EDI (mais qui grâce à cet héritage est cross-plateforme PC/Mac/Linux encore un avantage de l'esprit russe-tic !). Que cela soit sous XS ou VS on dispose désormais d'un éditeur visuel pour les UI. Le choix est donc vraiment ouvert, chacun fera comme il préfère (et selon sa licence de Xamarin aussi).

Le code exemple

D'abord il faut créer un nouveau projet. Je choisi de développer ici pour Ice Cream Sandwich, ICS, version 4.0 API niveau 14 parce que je le vaux bien  et qu'ici je me fiche de la compatibilité avec Gingerbread (version 2.3, API niveau 10).



Choisir la version d'Android

Vous allez me dire "c'est quoi toutes ces versions ?" … On entend d'ailleurs souvent les mauvaises langues parler de "fragmentation" d'Android, il y aurait des tas de versions dans la nature ce qui rendrait le développement presque impossible. Je vous remercie d'avoir posé la question car elle va me permettre de clarifier les choses (c'est bien d'avoir des lecteurs intelligents !).

D'une part je ferai remarquer non sans perfidie mais aussi amertume (car je le regrette vivement) que chez Microsoft il y a eu 3 versions majeures : Windows mobile, Windows Phone 7 et Windows Phone 8 et qu'aucune n'est compatible avec la précédente alors qu'une application écrite pour Android Donut de 2009 fonctionnera sur un Galaxy S4 utilisant la dernière version. Ca rend plus humble d'un seul coup quand on veut parler "fragmentation"… Et je ne parle pas des incompatibilités matérielles chez Apple où il faut racheter tous ses accessoires et câbles à chaque nouvelle version du _même_ téléphone du _même_ constructeur !

Mais un principe de droit nous dit que "la turpitude des uns n'excuse pas celle des autres". En gros si on vous prend en train de voler le dernier CD de Justin Bieber au Super U du coin, en dehors de vous coller la honte pour le restant de votre vie (à cause du choix du CD plus que du vol !), vous ne pourrez pas vous défendre en disant

que vous connaissez plein de gens qui le font aussi même si c'est vrai et même si vous pouvez le prouver. Votre délit reste un délit même si d'autres le commettent.

Je me dois donc de faire face honnêtement à la critique sans répondre à celle-ci par d'autres critiques. La réalité est que Google a compliqué inutilement les choses en donnant un nom de sucrerie à chaque nouvelle version (dans l'ordre alphabétique) ce qui est rigolo, mais qui soit aurait du être assez, soit est de trop ou aurait du se limiter au "nom de code" des versions avant leur sortie. Car ce nom de version se décline aussi en un vrai numéro de version (la 2.3, la 4.0.1 …) ce qui est une exigence technique naturelle pour suivre les versions releasées de n'importe quel soft. Mais tout cela n'était pas suffisant puisque la version n'indique pas forcément s'il y a eu des changements dans l'API, ce qui intéresse les développeurs. De fait cette dernière est aussi numérotée.

Au final on a un nom de sucrerie qui correspond à une ou plusieurs versions numérotées le tout en correspondance avec un ou plusieurs niveaux d'API.

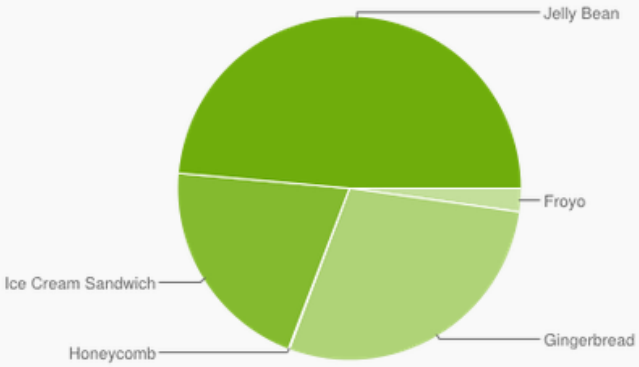
On trouve ainsi, par exemple, un Android Gingerbread (pain d'épice) qui suit en toute logique Froyo (G vient après F), le Froyo étant un yaourt glacé, et venant avant Honeycomb (H vient après G), Honeycomb étant un rayon de miel. Mais Gingerbread a connu plusieurs versions techniques qui sont numérotées : la 2.3 puis les 2.3.3 jusqu'à 2.3.7. Toutefois la plupart de ces versions n'ont pas touché aux API, sauf la 2.3.3, du coup la première release de Gingerbread, version 2.3, supportait l'API de niveau 9 alors que dès la sortie de la 2.3.3 l'API passa en niveau 10 pour y rester jusqu'à la sortie de Honeycomb…

Forcément ça embrouille un peu. *Mais en tant que développeur une seule information vous intéresse en réalité : le niveau de l'API supportée*, le reste est pur décorum.

Enfin, pour répondre complètement à la question qui était légitime, et je vous remercie encore de l'avoir posée, il faut parler vrai à propos de la fameuse fragmentation, c'est à dire la présence sur le marché d'utilisateurs ayant des machines sous plein de versions différentes ce qui serait un enfer.

La réalité est ici toute autre. Google tient à jour une analyse précise des machines en activité (avec un recul de 15 jours) qu'il est possible de consulter : le Dashboard Android. Pour les derniers 15 jours l'analyse est la suivante :

Version	Codename	API	Distribution
2.2	Froyo	8	2.2%
2.3.3 - 2.3.7	Gingerbread	10	28.5%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	20.6%
4.1.x	Jelly Bean	16	36.5%
4.2.x		17	10.6%
4.3		18	1.5%



Data collected during a 7-day period ending on October 2, 2013.

On voit qu'en fait plus de 48 % du marché est en version Jelly Bean (4.x) ou supérieure et qu'il reste 28.5% en Gingerbread (2.3). ICS qui est très récent aussi (c'est une version 4.x comme Jelly bean) occupe plus de 20%.

Si vous visez 70% du marché vous pouvez travailler directement en Ice Cream Sandwich (4.x).

Si vous désirez balayer près de 98% du marché vous devrez travailler en Gingerbread (2.3).

Au final vous devez choisir entre 2 versions d'Android selon la modernité que vous vous imposez, et malgré cela vous oscillerez entre 70 à 98% de couverture du marché. Il n'est plus possible de faire des applications Windows Phone 7.5 et de les publier sur le Store Microsoft, laissant en rade les possesseurs de Nokia 800 par exemple, je ne dirais que ça, ça calme les critiques en général…

Les différences de version d'Android sont importantes pour l'UI (beaucoup de choses ont été peaufinées dans les versions 4) mais on peut faire fonctionner de très belles applications en supportant 2.3.

Une majorité d'applications étant dans cette version justement pour couvrir le marché. Ce qui enquiquine bien entendu Google qui aimerait bien se débarrasser des versions sous la 4.0 car cela freine les développeurs dans leur utilisation des dernières nouveautés surtout sur le marché grand public. Du coup certaines applications semblent moins belles et moins ergonomiques qu'elles le devraient ce qui laisserait supposer qu'Android n'est pas au top, faisant ainsi du tort à l'image du produit. Mais la qualité d'une app dépend bien plus de son design que de la version de l'OS utilisé…

Finalement il faut se rappeler que pour l'instant (car les choses évoluent vite malgré tout) soit on vise les machines récentes, en gros 70% du marché, et on peut travailler directement en API de niveau 15, soit on vise 98% du marché et on doit alors utiliser l'API de niveau 10.

La "fragmentation" de Android est, on le voit ici, bien plus un argument d'une concurrence en difficulté qu'un véritable problème technique.

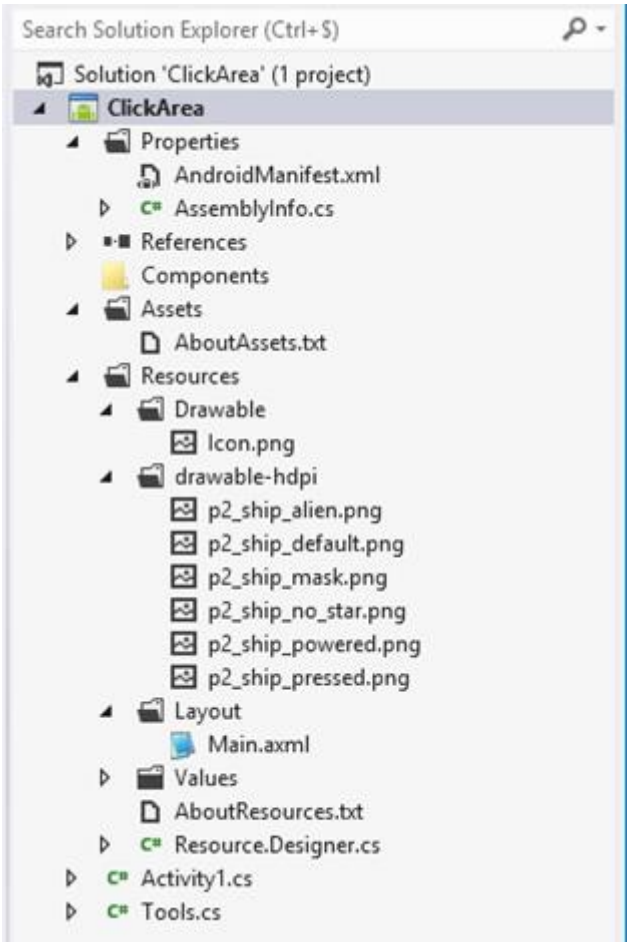
La structure de la solution

La structure d'un projet (et de sa solution) est habituelle avec ses propriétés, ses références éventuelles, ses classes, et ses ressources. On trouve bien entendu des spécificités liées à Android comme la présence d'un répertoire "`Resources`" contenant les "`Drawable`" (les 'dessinables') et les déclinaisons de ces ressources selon les densités d'écran supportées.

Ici l'image doit changer lorsqu'on cliquera sur les zones sensibles que nous allons créer, pour l'illustrer plusieurs variantes de l'image par défaut ont été réalisées. Elles sont en 800x480 et rangées dans le répertoire '`drawable-hdpi`' c'est à dire que nous les considérons comme des images visant un écran haute densité. Par notre choix intermédiaire (autant sur la taille des images que par leur classification) nous allons pouvoir couvrir sans gros problèmes toutes les autres résolutions car partant d'une situation "moyenne" les éventuels zoom (avant ou arrière) qu'Android fera automatiquement pour adapter les images n'introduiront pas de distorsion gênante.

Cette démarche est valable dans le contexte de cette démonstration mais pourrait être appliquée dans une situation réelle, c'est l'une des façons de gérer les grandes différences entre tailles et résolutions des écrans sous Android, un problème finalement bien plus délicat que le choix de la version des API.

Mais il y a tellement à dire à ce sujet que j'en resterais là pour ce billet !



Les images ont été empruntées à un article de Bill Lahti, un blogger du monde Android. Ce sont des PNG représentant une fusée dans les étoiles avec les variantes qui dépendent de la situation (feu sortant de la fusée ou non, un petit alien sorti de la fusée ou non, etc). Rien d'extraordinaire ni d'artistique, juste des images de test.

On trouve bien entendu une Activité (`Activity1.cs`) qui est l'unité d'exécution sous Android pour afficher quelque chose.

Une Activité peut être vue comme le code-behind d'une page écran. Mais ce n'est qu'une approximation. Ici ce n'est pas le code du visuel qui se charge avec l'Activité mais uniquement une instance contenant le code gérant l'affichage, l'Activité devant charger le visuel qu'elle veut, voire en changer en cours d'exécution.

L'Activité contient du code uniquement. La partie affichage est définie en XML et ce n'est qu'une description. Ici, le code visuel qui sera chargé par l'Activité se trouve dans le répertoire "`layout`" des "`Resources`" (c'est une convention qu'on doit respecter) et porte le nom de "`Main.axml`". En réalité l'extension devrait être "`xml`" mais pour charger le designer visuel spécifique dans Visual Studio notamment il

fallait adopter une extension différente (sinon c'est l'éditeur XML qui serait chargé par VS). Le fichier est donc totalement "natif" et pourrait être réutilisé en Java, à la nuance de son extension. Ce fichier de définition d'une mise en page instanciera des contrôles qui portent des noms (un ID), comme en XAML, et il sera possible d'y accéder depuis le code de l'Activité mais pas aussi directement (le nom de l'objet XML ne créée pas automatiquement une variable du même nom dans le code C#, il faut localiser l'objet graphique avec une méthode de recherche simple pour obtenir son pointeur et travailler dessus comme on le fait parfois en HTML).

Le projet contient aussi du code utilitaire, comme on en trouve dans les applications C# de tout type (ici la classe `Tools`).

J'aurais l'occasion de revenir sur certains détails d'un projet Android, ici nous avons l'essentiel pour comprendre la structure de l'exemple.

L'astuce des zones cliquables

Il est temps d'aborder le cœur de l'exercice même si, vous l'avez compris, celui-ci n'est qu'un prétexte pour présenter la création d'application sous Android avec Xamarin.

Comme nous avons pu le comprendre dès l'introduction les choses ne se présentent pas aussi simplement qu'on pourrait le faire en XAML. Mais rien n'interdit d'être astucieux.

La solution la plus simple qui vient à l'esprit serait de positionner des boutons transparents sur les zones cliquables et de gérer leur `Click`. C'est simple à comprendre mais pas forcément aussi simple à réaliser.

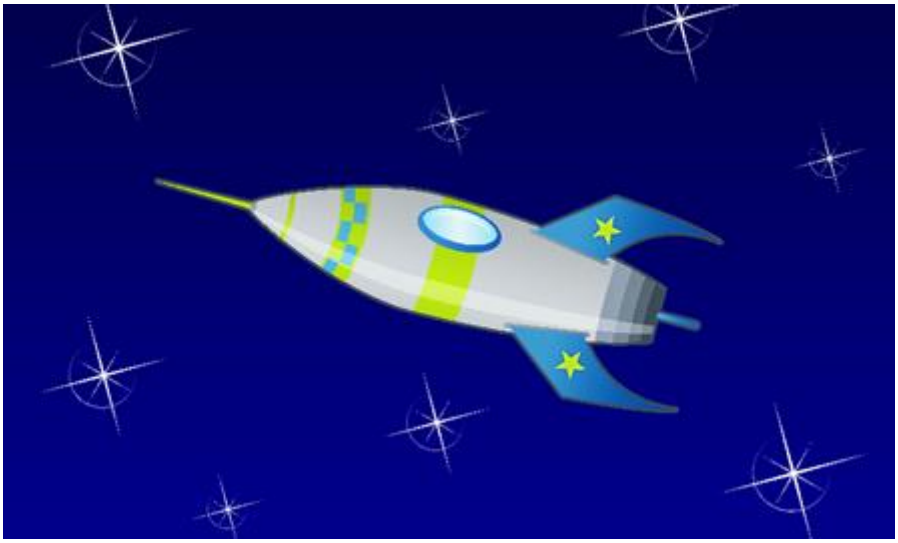
En effet le placement des boutons ne peut se faire en pixels ou avec des "margins" comme on le fait en XAML. Les outils de mise en page sont différents et la variété des écrans à supporter rend le jeu difficile. Il existe bien une sorte de "canvas" XAML sous Android, surface sur laquelle on positionne les objets par des coordonnées X,Y mais ce conteneur est déprécié et ne doit plus être utilisé. Le foisonnement des résolutions et tailles des écrans ont eu raison de cette façon trop simpliste de procéder qui réclamerait finalement beaucoup de code et de calcul pour s'adapter à toutes les situations.

Il nous faut donc trouver autre chose. L'idée reprise par de nombreux auteurs consiste à gérer deux images. La première, visible, est l'image "cliquable" (du point de vue de l'utilisateur), la seconde pourrait être appelée le "masque des régions" mais un masque se place par dessus de quelque chose alors qu'ici cette image sera placée

sous l'autre. Elle sera invisible pour l'utilisateur. On lui gardera le nom de "masque" pour simplifier en ayant conscience de cet abus de langage.

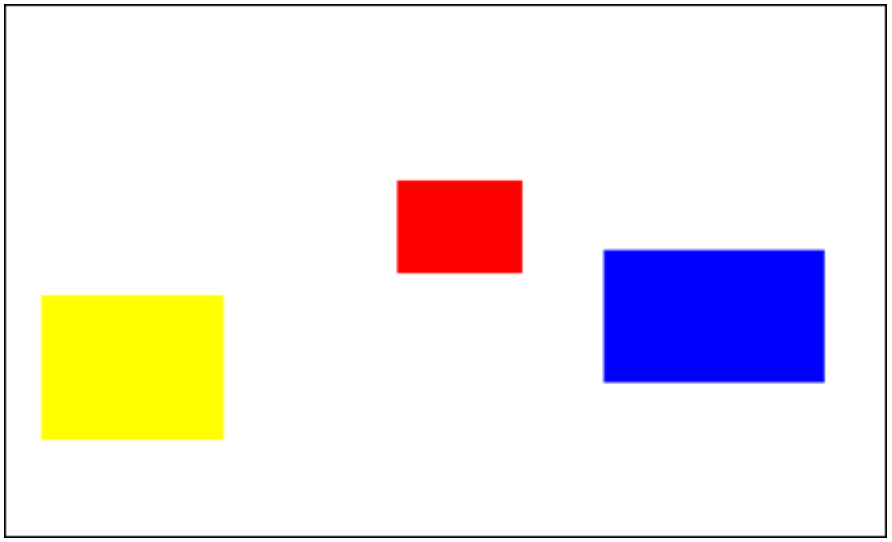
Cette image "masque des régions" consistera en une image de même taille que l'image utilisateur avec un fond uniforme blanc. Les zones seront simplement dessinées comme des formes pleines d'une couleur franche et unie : un rouge, un bleu, etc.

Pour bien comprendre regardons l'image "utilisateur" par défaut :



Trois zones cliquables sont définies, une première au niveau de la tuyère de la fusée, une seconde au niveau du hublot et une troisième à la place de l'étoile se trouvant au centre bas à gauche.

Voici à quoi ressemble l'image masque :



La bordure noire est ajoutée ici pour faire ressortir l'image et n'en fait pas partie.

Les trois zones sont définies par des rectangles de tailles différentes, ceci est arbitraire pour l'exercice même si ces zones tentent de recouvrir les trois objets (étoile, hublot et tuyère). En réalité ces formes seront parfois des rectangles, parfois des ellipses ou des cercles ou éventuellement le contour précis d'une forme de l'image principale. Vous avez bien entendu compris le principe.

On remarque que chaque forme possède une couleur et que celle-ci est différente des autres. Cela fait partie du mécanisme en permettant de savoir quelle zone a été cliquée.

La conception du masque peut s'avérer délicate mais dans une application moderne et bien designée tout (ou presque) doit passer par Photoshop et par quelqu'un qui sait s'en servir… Dans un tel cas ce n'est pas un problème. L'astuce reste abordable même à débutant, elle consiste en réalité à utiliser l'image "utilisateur" comme fond et à créer l'image masque comme un calque placé au dessus avec une transparence de calque pour voir le fond. Une fois le masque au point il suffit de copier son layer sur une nouvelle image et de la sauvegarder. Rien de sorcier.

Armés des deux images, l'image "utilisateur" et l'image "masque", comment s'en servir pour résoudre notre problème ?

La première étape consiste à placer chaque image dans un `ImageView`, un contrôle Android spécialisé dans l'affichage des images. Ces deux conteneurs d'image sont placés dans un même conteneur générique de telle façon à ce que le masque soit en dessous et non visible. L'équivalent d'un "`visibility=hidden`" en XAML et non pas

d'un "collapsed" et ce pour les mêmes raisons : dans ce dernier mode le système supprime l'image de l'arbre visuel pour gagner de la place, alors qu'en mode non visible elle reste à sa place mais sans être affichée et nous avons besoin que l'image masque soit exactement étirée et positionnée comme l'image utilisateur.

Une fois les deux images superposées il faut écouter l'évènement Touch de l'image "utilisateur". Comme tout évènement de ce type on récupère des arguments qui contiennent des informations pertinentes, notamment la position X,Y du Touch (équivalent du MouseUp ou Down) et le sens de l'action (justement Up ou Down).

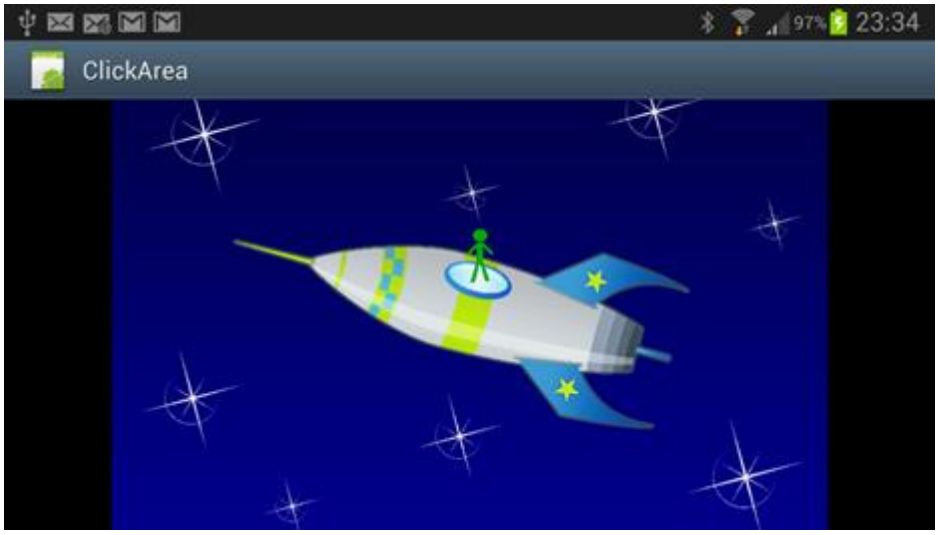
Partant de là, et puisque l'image se trouve exactement à la même position, zoomée exactement de la même façon, il nous faut récupérer le pixel se trouvant à la position indiquée mais dans l'image masque.

Ce pixel possède une couleur, et grâce à celle-ci nous savons quelle zone a été cliquée ou non ! Ensuite le programme décide de lancer les actions qu'il veut, par exemple passer à une autre page, exécuter un traitement, peu importe.

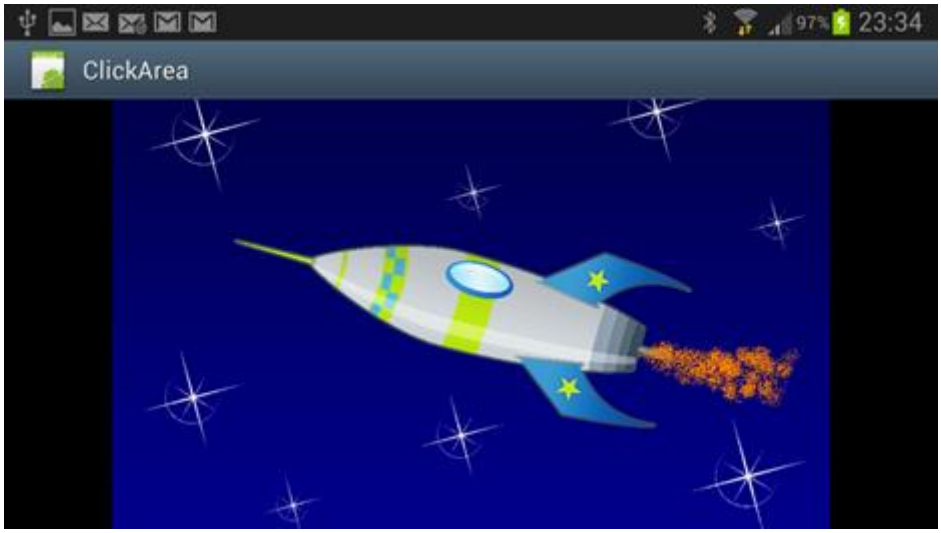
Ici nous changerons tout simplement l'image "utilisateur" pour donner l'impression que le clic allume ou éteint la fusée, fait sortir l'alien de la fusée ou non, allume ou éteint l'étoile de gauche. Nous ferons apparaitre aussi trois cercles sur les trois zones cliquables pour faire comprendre à l'utilisateur que ces zones existent (et où elles sont placées) lorsqu'il clique sur une zone non cliquable (donc dans du "blanc" dans l'image masque).

Le résultat

L'application peut être lancée dans le simulateur pour la mettre au point, mais surtout pour quelque chose de visuel il est essentiel de la tester aussi sur un vrai smartphone. Ici j'ai utilisé mon Galaxy S3 mis en mode debug relié par USB au PC. Ca marche tout de suite. Le S3 possède un gestuel particulier qui permet de faire une capture écran, ce sont donc des captures du S3 ci-dessous et non des captures du simulateur.



L'alien est sorti !



Le moteur est allumé !

L'application fonctionne aussi bien en mode portait qu'en mode paysage mais elle rend mieux dans ce dernier d'où les captures effectuées dans ce mode.

Il y a toujours quelque chose de magique dans le fait de voir s'exécuter son code sur un vrai smartphone et non dans le simulateur… Et surtout cela m'a permis de voir qu'un oubli dans ce code faisait planter l'application sur le S3 alors que cela ne réagissait pas pareil sur le simulateur (j'avais oublié de faire un `Dispose()` de l'image intermédiaire utilisée pour obtenir le pixel "cliqué" et sur le smartphone réel, ça plante très vite, mon S3 n'a pas les 16 GB de mon PC !).

Le code

Le XML du layout se présente comme suit :

Comme expliqué plus haut le layout est formé d'un conteneur générique, ici un `FrameLayout` conçu pour ne contenir généralement qu'un seul élément, ce qui nous arrange (une seule image est visible, la seconde ne pose pas de problème de mise en page puisqu'elle est cachée).

Dans ce conteneur on trouve deux `ImageView`, la première (donc la plus en dessous dans la pile visuelle) contiendra l'image masque, la seconde (qui est au dessus) contiendra l'image "utilisateur".

Les deux images sont placées de la même façon et c'est très important : la position `X,Y` de tout point de l'image utilisateur doit absolument correspondre à la même position dans le masque. L'image masque est marquée "invisible" pour ne pas être affichée toutefois elle est présente et possède son espace propre.

Le code principal de l'Activité est le suivant :

```
[Activity(Label = "ClickArea", MainLauncher = true,
     Icon = "@drawable/icon", ScreenOrientation = ScreenOrientation.Sensor)]
    public class Activity1 : Activity
    {
        private ImageView imageView;
        private ImageView imageMask;

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.Main);
            imageView = FindViewById<ImageView>(Resource.Id.ImageMain);
            imageMask = FindViewById<ImageView>(Resource.Id.ImageMask);
            if (imageView == null || imageMask == null)
            {
                Log.Debug("INIT", "images not found");
                Finish();
```

```
            return;
        }
        imageView.Touch += imageViewTouch;
        Toast.MakeText(this, "Touchez pour découvrir les zones",
           ToastLength.Long).Show();
    }
```

Deux variables sont utilisées pour localiser les deux contrôles images. Le `OnCreate()` commence par afficher l'image utilisateur par défaut puis initialise les deux variables en cherchant les contrôles dans le layout en cours.

L'évènement `Touch` de l'image principal est géré par la méthode `imageViewTouch`. En fin de `OnCreate()` un toast est affiché pour signaler à l'utilisateur qu'il peut toucher l'image pour découvrir les zones cliquables.

Rien de sorcier dans tout cela.

Le code de la gestion du Touch :

```
void imageViewTouch(object sender, View.TouchEventArgs e)
{
  if (imageView == null) return;
  if (e.Event.Action != MotionEventActions.Up) return;
  var touchColor = getHotSpotColor(e.Event.GetX(), e.Event.GetY());
  if (Tools.CloseMatch(Color.Red, touchColor))
     displayImage(Resource.Drawable.p2_ship_alien);
  else if (Tools.CloseMatch(Color.Blue, touchColor))
          displayImage(Resource.Drawable.p2_ship_powered);
  else if (Tools.CloseMatch(Color.Yellow, touchColor))
          displayImage(Resource.Drawable.p2_ship_no_star);
  else if (Tools.CloseMatch(Color.White, touchColor))
          displayImage(Resource.Drawable.p2_ship_pressed);
 }
```

Peu de subtilités ici : on contrôle l'action de Touch, la seule qui nous intéresse est le `Up` (quand le doigt est levé après avoir touché, c'est l'équivalent d'un `Mouse button up`) dans le cas contraire on sort de la méthode.

Le traitement de l'astuce des zones cliquables apparait ensuite : on obtient la couleur du point touché par l'utilisateur par la méthode `getHotSpotColor()` que nous verrons plus bas. Ensuite on teste si cette couleur est l'une de celles utilisées pour les trois zones cliquables (rouge, jaune et bleu), si la couleur est blanche (fond neutre du masque) on affiche l'image par défaut (par le biais de la méthode `displayImage` (à voir plus bas).

C'est très simple. La méthode `Tools.CloseMatch()` est conçue pour autoriser une certaine "dérive" de la couleur des pixels. Les ajustements de taille à l'écran effectués par Android peuvent en effet modifier légèrement la couleur d'un pixel. On accepte donc une tolérance sur la valeur testée.

Cette méthode est la suivante :

```
namespace ClickArea
{
    public static class Tools
    {
        public static bool CloseMatch(Color color1, Color color2,
                                       int tolerance = 25)
        {
            return !((Math.Abs(color1.A - color2.A) > tolerance) ||
                     (Math.Abs(color1.R - color2.R) > tolerance) ||
                     (Math.Abs(color1.G - color2.G) > tolerance) ||
                     (Math.Abs(color1.B - color2.B) > tolerance));
        }
    }
}
```

La méthode `getHotSpotColor()` qui permet d'obtenir la couleur du pixel en X,Y :

```
private Color getHotSpotColor(float x, float y)
```

```
{
    imageMask.DrawingCacheEnabled = true;
    try
      {
        using (var hotspots =
            Bitmap.CreateBitmap(imageMask.GetDrawingCache(false)))
        {
          if (hotspots == null)
           {
             Log.Debug("HS", "hotspots are null.");
             imageMask.DrawingCacheEnabled = false;
             return Color.Black;
           }
           return new Color(hotspots.GetPixel((int)x, (int)y));
         }
      }
      finally { imageMask.DrawingCacheEnabled = false; }
}
```

La façon d'obtenir le pixel fait intervenir le cache image de l'image masque, l'obtention d'un bitmap à partir de ce cache et ensuite la lecture de la couleur du point X,Y dans la bitmap obtenue. Il est essentiel de relâcher cette dernière d'où l'utilisation d'un bloc "using" qui effectuera un `Dispose()` automatiquement. De la même façon un `try/finally` s'assure que le mode cache du contrôle image est bien repassé à `false`.

Les raisons de cette construction s'éloignent de l'objectif du présent article, nous aurons certainement l'occasion de revenir sur ce point d'une façon ou d'une autre une prochaine fois.

Conclusion

Voir l'application fonctionner sur un vrai smartphone change la donne, n'hésitez pas à brancher en USB votre téléphone pour toujours déboguer de cette façon. Mon PC est ultra rapide, ce n'est pas une configuration "standard", et même ainsi équipé l'émulateur Android est moins rapide que l'exécution du débogue sur mon Galaxy S3.

Donc n'hésitez vraiment pas, l'émulateur n'est pas parfait comme tous les émulateurs mais surtout il est plus lent qu'un bon smartphone. Ne craignez pas de "polluer" votre téléphone, sous Android il n'y a pas de bricolage comme la Registry dont Microsoft n'arrive pas à se débarrasser : quand on désintalle une application, elle est supprimée sans laisser un million d'entrées dans des endroits sournois, Linux a toujours été plus sain de ce côté là que Windows… (Bien entendu si l'application s'amuse, avec les droits qui vont avec, à coller des fichiers de partout, il faudra les supprimer, mais ce n'est pas le cas de l'exemple étudié).

L'application que nous venons de voir n'est pas très sophistiquée, elle reste dans l'esprit de l'introduction : russe-tic mais elle a permis de parler de plein de choses et surtout de vous montrer à quel point Xamarin 2.0 est une plateforme séduisante et pratique pour développer sous Android. Elle l'est aussi pour iOS mais je ne peux pas parler de tout…

J'espère que tout cela aura excité votre curiosité et vous aura prouvé que vous aussi vous pouvez franchir le pas !

Introduction à MvvmCross V3 "Hot Tuna" (WinRT / Android)

Stuart Lodge a publié la version 3 dite "Hot Tuna" de MvvmCross. De nombreuses innovations sont présentes. Je vous propose une introduction complète en vidéo sur la création d'un projet cross-plateforme WinRT / Android.

MvvmCross ?

MvvmCross n'est pas un framework MVVM "de plus", il n'est pas comme les autres, il est unique en son genre puisqu'il permet de créer des applications cross-plateformes en suivant le modèle MVVM : WinRT, WPF, Silverlight, mais aussi Windows Phone, Android et iOS. Pour ces deux derniers il se repose sur Xarmarin (ex monoDroid et monoTouch). Toutes les adresses utiles se trouvent déjà sur Dot.Blog et dans la vidéo ci-dessous…

Hot Tuna

Hot Tuna est la troisième version majeure de MvvmCross, la précédente était appelée vNext. Elle ajoute beaucoup de choses intéressantes comme un fonctionnement par plugin pour les principales fonctions de toutes les plateformes (accès aux contacts, prendre une photo, géolocalisation, etc) ce qui permet d'écrire un code véritablement portable.

Mais ce n'est pas tout, Hot Tuna intègre un conteneur d'IoC servant à l'injection de dépendances, mais aussi un support du databinding pour iOS et Android ! Dans vNext ce support passait par une syntaxe JSON, désormais c'est le "swiss binding", une syntaxe épurée encore plus simple que celle de XAML mais tout aussi puissante.

Cross-plateforme

Comme j'ai déjà pu le dire dans ces colonnes, le cross-plateforme se présente habituellement comme l'idée qui consiste à pouvoir déployer le même code à l'identique sur plusieurs plateformes. Si c'est une des possibilités du cross-plateforme ce n'est pas celle qui a le plus d'avenir.

L'avenir est cross-plateforme mais dans un autre sens, dans celui d'entités qu'on appellera toujours "applications" mais qui seront en réalité "éclatées" en plusieurs modules : certains tournant sur PC, d'autres sur smartphone ou tablette etc. Et chaque form factor sera utilisé pour ce qu'il sait bien faire et ne fera donc pas forcément tourner le même code identique.

Mais qu'on parle de l'un ou de l'autre, le cross-plateforme est une aventure…

Sauf si on utilise les bons outils. Avec Visual Studio et C# d'un côté et Xamarin et MvvmCross de l'autre, un développeur peut contrôler toutes les plateformes existantes qui comptent sur le marché !
C'est incroyable et cette solution séduit de plus en plus de monde, et pour cause !

Une vidéo pour comprendre

Réservez-vous 45 minutes. Et suivez cette vidéo que je viens de mettre en ligne. En temps réel je développe devant vous une solution cross-plateforme (certes simple) qui fonctionne sur WinRT et Android.

MvvmCross est à l'honneur mais on y verra aussi Xamarin 2.0 à l'ouvrage, Visual Studio, l'émulateur WinRT, l'émulateur Android et quelques astuces à connaitre.

Prenez ces 45 minutes, vous verrez que cela permet de comprendre l'essentiel de l'avantage de ce "montage".

C'est une vidéo "sur le pouce", utilisant Hot Tuna dont je venais de faire le tour des nouveautés, ça pourrait aller plus vite, mais cela n'aurait pas permis pour le spectateur de reproduire tout. Là vous pouvez suivre la vidéo sur un écran, quitte à

faire des pauses, et faire la même chose sous VS à côté pour expérimenter par vous-mêmes, car c'est le but recherché. Tout est montré, pas à pas.

L'adresse directe sur Youtube en cas de problème avec la vidéo intégrée ci-dessus : http://youtu.be/OkHzOqr0zYw
Et un bonus pour les lecteurs de Dot.Blog, les sources du projet :

MvvmCross v3 "Hot Tuna" Introduction Source

Conclusion

Les vacances c'est fait pour se préparer physiquement, nerveusement et mentalement à la rentrée. La rentrée sera cross-plateforme, formez-vous maintenant avec Dot.Blog et n'hésitez pas non plus à faire appel à mes services si vous avez des projets en vue ! (je ne vis pas des 50 euros que me rapporte tous les 6 mois la petite pub Microsoft en haut à gauche sur Dot.Blog, et non ! ☺).

MvvmCross v3 Hot Tuna : les services (WinRT / Android)

Stuart Lodge a publié la version 3 dite "Hot Tuna" de MvvmCross. De nombreuses innovations sont présentes. Je vous propose ici la suite de la vidéo d'introduction publiée hier en abordant les services et l'injection de dépendances. Toujours en vidéo, en "live" et sous WinRT et Android.

Les services

Dans une application cross-plateforme comme dans une autre il est nécessaire de mettre à disposition du code un certain nombre de "services". Pour assurer le découplage de ces derniers avec le code utilisateur on peut utiliser plusieurs stratégies, celle de l'injection de dépendances dans les constructeurs a été retenue par MvvmCross (même s'il est possible de mettre en place d'autres moyens).

C'est ce procédé rendu excessivement simple par MvvmCross que je vous propose de découvrir au travers d'une vidéo d'une quinzaine de minutes aujourd'hui :

Le lien Youtube : http://youtu.be/YWU4E8d_oqQ

MvvmCross v3 : Les listes (WinRT et Android)

La série vidéo sur MvvmCross et le développement cross-plateforme continue ! Aujourd'hui le troisième volet porte sur les listes.

La série N+1

Stuart Lodge, le développeur de MvvmCross, a créé une série de vidéo en anglais appelée “N+1”, partant de N=0 à N=41, au moins pour le moment.

Chaque vidéo montre un aspect du développement cross-plateforme avec MvvmCross. Chacune montre en live la création d’une solution qui met en valeur le sujet du jour. Et chacune présente en général la mise en œuvre de l’exemple sous WinRT, Android, iOS et Windows Phone.

J’ai trouvé cette série tellement bien faite que je m’en inspire plus ou moins directement dans la série de vidéos que je vous propose en ce moment. De toute façon qui mieux que Stuart sait ce qu’il y a dans MvvmCross…

Mes choix sont légèrement différents des siens. Il tient à chaque fois à montrer le fonctionnement de MvvmCross sous le maximum de plateformes, c’est normal, c’est tout l’intérêt de MvvmCross ! Personnellement j’ai opté pour le couple Android / XAML, sachant que selon les vidéos la partie XAML est mise en valeur soit par le biais de Windows Phone, de WPF ou de WinRT.

Ce choix repose sur plusieurs raisons : d’abord cela permet de faire des vidéos sensiblement plus courtes, donc plus faciles à regarder que des pavés d’une heure ou plus. Ensuite cela permet de traiter le sujet avec plus de détails parfois quand cela l’exige. Enfin je ne dispose pas d’un Mac en ce moment, indispensable pour faire tourner l’émulateur iOS.

Mais je pourrais ajouter que le couple WinRT / Android représente le plus grand écart entre toutes les technologies présentées, mis à part iOS qui est franchement bizarre de ce point de vue. Et ce qui m’intéresse c’est de vous montrer que deux mondes aussi différents l’un de l’autre peuvent se programmer avec la même aisance et le même code commun grâce au trio Visual Studio / Xamarin 2.0 / MvvmCross.

L’avenir sera cross-plateforme, je vous l’affirme. Microsoft n’arrivera pas à devenir dominant sur les mobiles comme il l’a été sur les PC dans les 25 dernières années. Les dés sont jetés. Microsoft release en ce moment Office 365 qui est lui-même cross-plateforme !!! Ce qui serait bon pour Microsoft ne le serait pas pour nous ? Android ne représentera pas à moyen termes 100% non plus des OS mobiles. Apple baissera mais résistera encore un bon moment. Bref, le paysage qui s’étend devant nous pour les années à venir, et qui s’imposera à tous les développeurs, c’est celui du cross-plateforme.

C'est à cet avenir que Dot.Blog vous prépare… C'est ce mélange de technologies unifiées par des outils intelligents que je peux vous aider à mettre en place car vous aurez forcément de tels projets prochainement dans votre entreprise !

Les listes sous WinRT/Android avec MvvmCross v3

Le lien direct vers la vidéo Youtube : http://youtu.be/JBzj_nkeLFI
La vidéo :

Conclusion

D'autres vidéos vont suivre… Je ne suivrais pas les N+1 de Stuart exactement telles qu'elles sont faites. J'en sauterais peut-être ou en regrouperais-je d'autres, tout va dépendre, notamment du temps de disponible (on s'imagine mal à quel point produire même humblement de telles vidéos de 30 à 60 min réclame du temps, souvent une bonne demi-journée et même plus !).

MvvmCross V3 : Convertisseurs de valeur et Navigation (WinRT/Android)

La série vidéo sur MvvmCross et le développement cross-plateforme continue ! Aujourd'hui le quatrième volet porte sur les Convertisseurs de valeur et la navigation avec ou sans paramètres.

Convertisseurs de valeur, Navigation et paramètres de navigation

Le lien direct vers la vidéo Youtube (40 minutes HD) : http://youtu.be/tpL5xZHicJI

Cross-Plateforme Windows Phone 8 / Android avec MvvmCross : Blend, données de design, Google Books API …

Cinquième volet de cette exploration en vidéo du développement cross-plateforme. Aujourd'hui beaucoup de choses dans une vidéo HD de 1H ! A l'honneur : Windows Phone 8 et Blend, MvvmCross, Xamarin, Android et les API Google Books.

Cross-plateforme

Le cross-plateforme n'est ni une un mode ni une méthodologie, c'est une réalité à laquelle tous les DSI et tous les développeurs devront faire face à la rentrée et encore plus en 2014.
La prise en compte de tous les form factors et de tous les OS est une tâche ardue qui réclame plus qu'une méthode, elle réclame **un savoir-faire et des outils adaptés**.

Cette série de vidéo consacrée au développement cross-plateforme vous montre en live les bases de la stratégie que je vous propose depuis déjà plusieurs mois.

Vos projets, mon savoir-faire

Nous ne sommes pas de purs esprits savourant l'air du temps pour seul repas et se déplaçant en tapis volant mu par quelque principe magique… Le free envahit Internet, plus de 610 billets et articles gratuits et des heures de vidéo toutes aussi gratuites sur Dot.Blog, mais il n'existe toujours pas de loyer free, de pompe à essence free ou de boucher free !

Alors si vous aimez Dot.Blog et mes articles, n'oubliez pas que mon savoir-faire saura donner à vos projets l'élan qui en fera des succès ! N'hésitez pas à en parler à vos collègues, connaissances ou à votre patron !

WP8, Blend, Android, MvvmCross, Xamarin, VS…

La vidéo d'aujourd'hui traite en un peu plus d'une heure (et en HD) de très nombreuses choses. Elle reprend à la fois des thèmes abordés dans les vidéos précédentes et les augmente de nouveautés, d'astuces, de tours de main qui, je l'espère, vous démontrerons à quel point la stratégie cross-plateforme que je vous propose est le moyen le plus simple et le plus sûr d'affronter les défis de la rentrée et de l'année à venir.

Données de design sous Blend, comment étendre les types de données proposés de base, Windows Phone 8 et Android avec MvvmCross, mise en place d'un service d'accès aux Google Books API totalement cross-plateforme, mise en forme XAML des listes, il est très difficile de donner un titre à la vidéo d'aujourd'hui tellement elle balaye large et brasse de nombreuses techniques.

Le mieux ? C'est de la regarder 

La vidéo

Le Lien direct Youtube : http://youtu.be/UkhMH7FgSj4

Le bonus

Pour les lecteurs de Dot.Blog le petit bonus : le code source de la vidéo !

(La solution VS2012 nécessite l'installation du SDK Windows Phone 8 et de Xamarin 2.0 pour la partie Android)

Les sources de la solution VS Books.zip (2,5Mo)

Conclusion

Le cross-plateforme s'impose tous les jours comme le véritable avenir du développement dans un monde fractionné et tiraillé entre plusieurs éditeurs d'OS et de plateformes. Il n'y a donc rien à conclure, au contraire, c'est à une genèse que vous assistez au travers de Dot.Blog !

Cross-Plateforme Windows Phone 8 / Android avec MvvmCross : Blend, données de design, Google Books API– Partie 2

Sixième volet de cette exploration en vidéo du développement cross-plateforme. Aujourd'hui encore beaucoup de choses dans une vidéo Full HD de 30m ! En lumière : Windows Phone 8 et Android, MvvmCross, Visual Studio et Xamarin.

Partie 2

Dans cette vidéo je vous montre comment compléter le code du noyau de l'application "Books" et quelques techniques importantes de gestion de l'interface visuelle sous Windows Phone 8 et Android.

La vidéo

Le Lien direct Youtube : http://youtu.be/sMm1BtUtmmI

Le bonus

Pour les lecteurs de Dot.Blog le petit bonus : le code source de la vidéo !

(La solution VS2012 nécessite l'installation du SDK Windows Phone 8 et de Xamarin 2.0 pour la partie Android)

Books2.zip

Conclusion

La prochaine vidéo sera consacrée à la géolocalisation…

Cross-Plateforme Vidéo 7 : Géolocalisation et bien plus !

7ème volet de notre saga sur le développement cross-plateforme en vidéo. Je vous emmène à la rencontre de la géolocalisation, du GPS, de Google Maps toujours avec MvvmCross, Xamarin, Visual Studio sous Android et Windows Phone.

Volet 7

Après un rapide rappel du cycle de création des ViewModels (le modèle "CIRS"), nous entrons dans le vif du sujet : la géolocalisation. Toutefois ce ne sera qu'un prétexte pour découvrir de nombreuses techniques de développement cross-plateforme :

- La géolocalisation en mode GPS ou approximé
- Phase 1 : Longitude, Latitude et Altitude
    - Utilisation de Monitor du SDK Android pour injecter des coordonnées dans le simulateur
- Phase 2 : géoencodage ou comment récupérer des informations détaillées sur le lieu
- Ecriture d'un service REST, utilisation de Json2Csharp, écriture du service de géoencodage, le Messenger de MvvmCross pour communiquer entre le VM et la View.
- Debug sur un terminal réel (Samsung S3), de nouveau le Monitor pour obtenir des captures et des informations précises sur ce qui se passe dans la device.
- Ajout d'un appel à l'API Google Maps pour afficher la carte du lieu.
- Précisions sur le portage sous Windows Phone

Le tout en 52 minutes et en Full HD (1920x1080).

N+1

Comme vous le savez je suis (très approximativement) la série N+1 de Stuart Lodge qui me sert de trame et de prétexte à mes propres présentations. Stuart effectue systématiquement des démos sur 2 voire 3 ou 4 plateformes à la fois, c'est toujours intéressant de voir comment il fait, notamment sur iOS que je ne traite absolument pas. N'oubliez donc pas de consulter ses propres vidéos sur MvvmCross !
A titre d'information, la présente vidéo s'inspire (très librement) des N=8 et N=9 de Stuart. Les parties sur la géolocalisation sont traitées plus en profondeur par ma vidéo, celles de Stuart balayent plus large avec plus de plateformes utilisées et démontrées. Les deux sont donc à voir…

La vidéo

Le lien direct : http://youtu.be/SJHDKoQ29nU
La chaîne YouTube de Dot.Blog : http://www.youtube.com/TheDotBlog
Le Full HD réclame un visionnage plein écran sur un écran d'au moins 1920x1080p, les autres modes moins denses peuvent être moins lisibles.

Le bonus pour les lecteurs de Dot.Blog :

Code du projet Geo.zip

Conclusion

Sur ce long chemin de la découverte du développement cross-plateforme nous avons déjà posé de nombreux jalons, mais l'aventure est loin d'être terminée ! Avec près de 5h de vidéo en HD et Full HD il y a vraiment de quoi se faire une bonne idée et même se lancer. Mais les prochaines vidéos réserveront des surprises comme l'utilisation cross-plateforme de SQLite, des Picture choosers, les I/O, la gestion maître/détail, les custom controls Android et les custom controls Windows Phone, la localisation des applications, les dialogues, les vues splittées, les tabs, utiliser les fragments sous Android, et bien plus encore, le tout, toujours sous Android, Windows Phone, WinRT et peut-être même WPF.

Cross-plateforme Video 8 : Gérer des données SQLite sous Windows Phone et Android avec MvvmCross

Huitième volet de cette série de vidéos la présentation d'aujourd'hui vous propose de découvrir l'utilisation de données avec SQLite sous WP8 et Android toujours à l'aide Visual Studio, Xamarin et MvvmCross.

Les données en cross-plateforme

Gérer des données structurées de façon fiable dans un environnement cross-plateforme n'est pas une chose simple. Plusieurs contraintes viennent éliminer nombre de solutions. SQL Server par exemple qui bien qu'il existe sous diverses versions, dont des versions mobiles, ne tourne que sous OS Microsoft. D'autres solutions sont soient onéreuses, soit inapplicables pour les mêmes raisons que SQL Server.

En revanche SQLite est une base de données tout à fait honorable et parfaitement taillée pour un tel contexte. Elle tourne bien, sans consommer trop de ressources, et sait s'adapter à tous les OS.

MvvmCross v3 intègre un plugin SQLite qui en simplifie encore plus l'utilisation et l'intégration.

En 40 minutes je vous montre comment utiliser SQLite dans vos projets cross-plateformes en prenant pour exemple une petite gestion de données portée sous Windows Phone et Android.

La vidéo

Le lient direct : http://youtu.be/uXelZV41VKQ
Ma chaîne YouTube où vous retrouverez toutes les vidéos de cette série : http://www.youtube.com/TheDotBlog
La vidéo (40 minutes, HD 720p)

Le Bonus

Comme toujours le bonus pour les lecteurs de Dot.Blog : le code source du projet utilisé pour concevoir la vidéo :

DBtest.zip

Cross-plateforme vidéo 9 : Envoi de mail avec WPF et Android

Neuvième volet de cette série de vidéos la présentation d'aujourd'hui vous propose de découvrir l'envoi de mail avec Android et WPF, prétexte à l'introduction de ce dernier dans une logique cross-plateforme.

Cross-plateforme : quels OS et technologies ?

Le cross-plateforme est une technique de développement. Il s'appui sur des outils (par exemple Visual Studio, Xamarin, MvvmCross…) **mais il ne faudrait pas le réduire à un simple problème de technicien** ! Une plomberie tout juste bonne pour amuser les plombiers du XXIème siècle que sont les développeurs (du point de vue de beaucoup d'entreprises hélas)…
Le cross-plateforme est aussi, et dirais-je avant, tout un choix politique et stratégique crucial pour tous les DSI, une route à ne pas louper, un virage serré pour se glisser telle une anguille entre les mines du champ de bataille des OS, guerre qui ravage les initiatives depuis trois ans au moins. Paralysie dont il va bien falloir sortir en assumant son salaire : aller de l'avant au lieu de faire le gros dos…
Heureusement le ciel s'éclaircit. Des hordes barbares qui se sont jetées dans la bagarre seules deux aujourd'hui dominent le monde. Android pour les unités mobiles

(80% des smartphone et 67% des tablettes à la date de ce billet) et Windows pour les ordinateurs de bureau. Je parle de Win32/64 et non de WinRT.

Deux plateformes, finalement voilà ce qui reste de cette guerre dont Microsoft sort affaibli mais toujours debout, et dont Google, sorti du diable vauvert est passé de simple moteur de recherche du Web au statut de premier OS mobile sur toute la planète, anéantissant au passage l'étoile filante Apple.

Aujourd'hui et demain, les entreprises devront apprendre à gérer deux plateformes à la fois : ce bon vieux Windows en mode bureau classique et toutes ces nouvelles machines sous Android.

Ce neuvième volet de la série de vidéos sur le cross-plateforme est essentiel. *Pour une fois cette vidéo subversive, sous prétexte de technique, fait de la politique, de la stratégie d'entreprise…*
Cette neuvième vidéo vous présente un exemple de développement cross-plateforme comme un modèle de ce que vous devrez faire pour surmonter la guerre des OS, *pour que vos applications vivent et survivent au grand chamboulement qui s'est produit*.
Pour assurer une présence sur 90% du parc d'ordinateurs, mobiles ou non, le choix du couple Android/WPF s'impose à la raison. Tout le reste n'est que fantaisie (Html), modes en déclin (Apple) ou faux départs (WinRT, Windows Phone 8). Seule la réalité doit s'imposer aux décideurs, seuls les chiffres doivent guider le choix.

Et les chiffres sont têtus : Ils nous disent que 80% du marché mobile est sous Android et que 80% des entreprises utilise une version de Windows non WinRT.

Cet état de fait peut-il être chamboulé ou présage-t-il d'un partage manichéen du monde fait pour durer ?

Chacun répondra selon ses convictions, la divination est un art difficile…

Toutefois l'écart creusé par Android semble tellement gigantesque avec ses concurrents que même le géant Apple perd pieds et sa deuxième place, tant convoitées par ceux qui ont compris qu'ils ne pouvaient espérer mieux, est à des centaines de millions de kilomètres (et d'unités vendues) de la place de 1er. Quant aux troisièmes et aux suivants, ils sont à des dizaines de millions d'unités du second.

Comment croire qu'en dehors d'un miracle, une invention tranchant avec tout ce qui existe, cet ordre pourrait demain changer en un claquement de doigts ?

Personnellement je ne pense pas qu'un tel miracle arrivera. Les dés sont jetés et l'équilibre qui se forme sous nos yeux est là pour durer.

Et dans ce monde qui se stabilise, le couple WPF/Android apparait être le seul à permettre d'être présent partout, en natif, et non en mode Web dont la chute au profit des "apps" se confirme d'année en année.

Alors profitez bien de cette série unique de vidéos et particulièrement de ce neuvième volet !

En 40 minutes je vous montre comment créer un projet cross-plateforme faisant le grand écart entre un smartphone Android et une application WPF lookée en style Metro utilisant un même code métier écrit une seule fois. Une sorte de jeu de construction amusant, mais aussi une arme absolue pour conquérir les CPU du monde entier !

La vidéo

Le lient direct : http://youtu.be/uZ7vP9_qkho
Ma chaîne YouTube où vous retrouverez toutes les vidéos de cette série : http://www.youtube.com/TheDotBlog
La vidéo (40 minutes, HD 720p)

Le Bonus

Comme toujours le bonus pour les lecteurs de Dot.Blog : le code source du projet utilisé pour concevoir la vidéo :

SendMailAndroidWPF

Cross-plateforme vidéo 10 : Codez comme un Ninja !

Dixième volet de cette série de vidéos la présentation d'aujourd'hui vous propose de découvrir Ninja Coder for MvvmCross pour coder encore plus vite !

Ninja Coder

En 10 minutes je vous montre comment installer et utiliser cette extension de Visual Studio pour mettre en place encore plus rapidement des solutions cross-plateformes intégrant tous les packages Nuget MvvmCross nécessaires, les projets d'UI et le projet noyau et même le projet de test NUnit si on le désire, Tout cela de façon automatique !

La vidéo

Le lient direct : http://youtu.be/JMQw5fpcpY0
Ma chaîne YouTube où vous retrouverez toutes les vidéos de cette série : http://www.youtube.com/TheDotBlog
La vidéo (10 minutes, HD 720p)

Conclusion

Faire des vidéos à peu près propres est un travail bien long… On ne s'imagine pas en général que pour 10 minutes produites il faut plusieurs heures de labeur : préparation du sujet, répétition des démos ou création et test des projets, installation du micro (j'utilise un Zoom H2 pour une meilleure qualité du son), lancement du soft de capture, enregistrement des différents fragments. Puis vient la première phase de montage : suppression des bruits, des redites, des toussotements, des bafouillages, sur chaque segment enregistré, lissage du volume sonore, suppression du bruit résiduel, génération d'une vidéo "propre" par segment… on passe enfin au montage final : intégration du générique de début, des messages en surimpression, création des transitions, des effets de zoom, de détourage, ajout d'effets comme les flèches pour montrer un détail, ajout du générique de fin et enfin production en HD. Mais ce n'est pas fini : visionnage de test puis transfert sur Youtube, écriture du billet pendant l'upload, publication de la vidéo, récupération du code d'intégration, intégration de ce code dans le billet en cours, publication du billet ! Et j'en oublie certainement…
On notera que les génériques de début et de fin, bien que simples, ont été entièrement fait à la main, musique originale comprise (sous Live 9) !

Cross-Plateforme 11 : Swiss, Tibet et Multibinding, Rio, le tout sous Android et Xaml

Onzième volet de cette série de vidéos sur le cross-plateforme la présentation d'aujourd'hui vous propose de découvrir le multibinding, le Swiss et le Tibet binding, Rio sous Android mais aussi sous Xaml (WinRT, WPF…) !

Réinventer le binding

Le binding est une des choses les plus spectaculaires des environnements modernes. On le connaissait sous ASP.NET, sous Windows Forms, on l'a redécouvert amplifié sous Silverlight et WPF, il s'est imposé sous Xaml jusqu'à WinRT et jusqu' à donner naissance par sa seule existence à la méthode **MVVM**…
Le binding Xaml de WPF reste à ce jour le plus puissant et le plus complet. Suivi de très près par celui de Silverlight dont hélas l'avenir s'est assombri *même s'il reste une solution unique et pertinente dans de nombreux cas*. Xaml lui-même reste vigoureux,

intégré dans Windows 8 on le retrouve au cœur du développement WinRT et Windows Phone, même si ce Xaml n'est pas au niveau de celui de WPF.
Mais estimons-nous heureux, le binding n'existe absolument pas sous Android ou iOS par exemple !

Toutefois **MvvmCross**, ce framework cross-plateforme totalement *awesome* propose depuis longtemps une solution de binding pour les environnements Android et iOS. Au départ sous la forme d'une extension utilisant une syntaxe JSON, ce binding n'avait d'usage et d'intérêt que sous Android et iOS, les plateformes Xaml disposant déjà de leur propre solution.
Mais avec MvvmCross v3, Stuart Lodge **réinvente le binding** au point de faire passer celui de Xaml pour une vieillerie ! On se dit alors que c'est un comble que des environnements pauvres non pourvus de binding comme Android ou iOS finissent par bénéficier d'une solution de binding plus efficace et plus puissante que les plateformes Xaml !
Oui, ça serait vraiment dommage…

Mais c'est sans compter sans l'ingéniosité de Stuart et sans son attachement radical au cross-plateforme.**Le binding réinventé de MvvmCross est cross-plateforme et disponible sous Xaml aussi !** Et il est tellement mieux ficelé que celui de WinRT par exemple, qu'on en vient à l'utiliser pour palier les faiblesses de ce dernier…

Le cross-plateforme : l'assurance anti siège éjectable

Il y a tant de choses que j'aimerais vous expliquer, à vous et à votre hiérarchie… Déjà 11 vidéos, plus de 7h en HD, déjà des dizaines de billets sur le sujet, et je me rends bien compte que l'essentiel n'est pas de convaincre les développeurs, mais leurs responsables !

Un message simple : le cross-plateforme tel que je vous le propose ne consiste pas uniquement à créer des applications copies-conformes, des clones. Ca c'est la façon simple de faire des démos pour montrer la technique. Le cross-plateforme auquel je crois est tout autre chose : c'est un anxiolytique !

Mieux, c'est une assurance anti siège éjectable !
Je m'explique : les DSI sont frileux de nature et la conjoncture n'a jamais été aussi compliquée pour décider le lancement de la production de logiciels. WinRT, Windows 8 et Windows Phone 8 ne sont pas des succès planétaires, cela fait un peu peur de se lancer. Android n'est pas encore reconnu dans le milieu des entreprises (malgré de gros efforts de sécurisation et une suite de services entièrement dédiés aux entreprises). Apple ? avec 13% de parts de marché en chute libre, il y a de quoi avoir le trouillomètre à zéro… BlackBerry est à vendre, sa cotation sur les marchés devrait

être suspendue (si ce n'est déjà fait). Se lancer sous Linux et même si Ubuntu est une solution très attrayante, c'est risqué. Le "grand soir", les linuxiens l'annonce depuis tellement d'années qu'il finira par arriver sans qu'on y croit…

Dans un tel contexte, la frilosité des DSI est plus que compréhensible. Mais je leur dit : **n'ayez plus peur!** Avancez, lancez-vous : votre entreprise utilise depuis toujours des solutions Microsoft, lancez la production de vos logiciels sous WinRT ou WPF… *Mais avec ma méthode de cross-plateforme*… Si dans 3 ou 6 mois vous devez supporter en plus (ou à la place) Android, le cout sera marginal **car l'essentiel de l'application ne sera pas à réécrire** !
Avec la méthode que je vous propose, *vous avez le droit à l'erreur*, à l'hésitation. Faites un premier jet, et**améliorez le projet au fur et à mesure des besoins réels**.
Le cross-plateforme avec les outils et méthodes que je vous propose est une véritable garantie contre les erreurs et les ennuis qui vont avec !

En étant réactif, souple, et en pouvant se lancer tout de suite, vous pourrez aider votre entreprise à prendre de l'avance, si précieuse en période de crise. Mais sans risque de se "planter" et d'avoir à justifier un budget englouti en pure perte. Avec le cross-plateforme tel que je vous le propose, "rien ne se perd, tout se transforme" …

Vous êtes DSI ? Alors parlons de vos projets. Vous êtes développeur ? N'hésitez pas à présenter cette démarche à votre hiérarchie !

En attendant, en un peu moins de 45 minutes, je vous propose aujourd'hui de découvrir le multibinding, le swiss et le tibet binding, Rio… Bon visionnage !

La vidéo

Le lient direct : http://youtu.be/yIr3BbA0t68
Ma chaîne YouTube où vous retrouverez toutes les vidéos de cette série :http://www.youtube.com/TheDotBlog
La vidéo (10 minutes, HD 720p)

Conclusion

Le cross-plateforme n'est au départ que de la plomberie. Mais avec les bons outils et les bonnes méthodes c'est aussi une redoutable arme anti stress permettant de se lancer dans tous les développements qui ne peuvent plus attendre et tous ceux qui donneront à votre entreprise cet élan indispensable pour sortir du lot.

Cross-Plateforme Vidéo 12 : Injection de code natif (WinRT/Android)

Douzième volet de cette série de vidéos sur le cross-plateforme la présentation d'aujourd'hui vous propose de découvrir comment injecter du code natif dans les applications.

Injection de code natif

Pourquoi vouloir "injecter" du code natif au lieu de "l'utiliser" ?

Partons du début : nos applications cross-plateformes sont formées d'un noyau, une librairie PCL portable, dans laquelle aucune compilation conditionnelle n'est autorisée (parce qu'on se l'impose principalement par souci de maintenabilité).

Or puisque ce noyau n'est pas modifiable selon la compilation de la cible et qu'il est identique pour toutes celles choisies, comment peut-il utiliser un code natif qui lui est inaccessible ?

Rappelons aussi que ce noyau est l'endroit où se trouve toute la logique de l'application, le code métier. Dans notre système cross-plateforme nous n'acceptons pas d'utiliser de la logique dans les applications d'UI.

Un subtil jeu de ping-pong va devoir être mis en place…

Trois voies sont possibles : L'utilisation astucieuse d'une interface C# et du conteneur d'IoC, la création d'un plugin MvvmCross, ou le codage directe dans chaque application d'UI. On s'interdira cette dernière méthode, trop barbare et trop éloignée des standards de qualité que nous nous sommes imposés.

Reste deux approches, l'une plus rapide, l'autre réutilisable facilement.

Ce sont ces deux voies que je vous propose de découvrir dans cette session live d'une quarantaine de minutes…

La vidéo

Le lient direct : http://youtu.be/fRqOnJ1x0ug
Ma chaîne YouTube où vous retrouverez toutes les vidéos de cette série : http://www.youtube.com/TheDotBlog
La vidéo (42 minutes, HD 720p)

The End ?

Cette douzième vidéo achève ce cycle de formation sur la stratégie de développement cross-plateforme que je vous propose. Cela ne veut pas dire que je n'en parlerais plus ou qu'il n'y aura plus d'autres vidéos, mais simplement que cette série là est bouclée.

Il reste beaucoup d'autres sujets à aborder comme les contrôles customs sous les principales plateformes, le testing des ViewModels, les animations, les await/async dans l'implémentation des services et des VM, etc. Mais tout cela s'écarterait de l'objectif que je m'étais fixé pour cette "simple introduction" qui représente déjà près de 8h de vidéo HD en 12 volets (sans compter les nombreux billets qui l'ont précédée) !

Mais il faut bien en laisser pour les formations que je prodigue et surtout que je garde quelques secrets pour les développements que je vends 

Je plaisante.
Je considère que ce qui fait la différence entre deux professionnels ce ne sont pas les petits secrets qu'on finit toujours par découvrir.
La rétention d'information ne peut que berner les idiots et satisfaire les imbéciles.
Je pense que ce qui fait la différence c'est l'individu et toutes ses qualités, l'information doit être ouverte et disponible à tous. Seuls les médiocres ont besoin de cacher ce qu'ils savent pour briller.
Notre métier n'est finalement pas affaire de mémoire ou de connaissance livresque, mais de puissance de calcul et d'expérience. L'information se trouve toujours. La puissance de calcul et l'expérience pour la traiter sont des qualités plus rares.
Partagez vous aussi ce que vous savez. Vous n'en sortirez plus faible que si votre seule valeur se limite à ce savoir…

La valeur d'un développeur, d'un architecte, d'un consultant ? C'est justement ce qu'il reste quand il a dit tout ce qu'il savait…

C'est un chalenge intéressant avec soi-même dans tous les cas !

Conclusion

Cette session montre des techniques performantes et élégantes, renforçant la stratégie cross-plateforme que je vous propose sans rien sacrifier de la pureté du code, de la maintenabilité mais aussi des spécificités de chaque plateforme.

Je rappellerais que cette stratégie de développement est tellement performante qu'elle s'impose même pour le développement d'une application unique qui n'est pas vouée au départ au cross-plateforme. Cela permet de s'offrir pour pas très cher une assurance sur l'avenir car tout portage vers une nouvelle plateforme sera à la fois possible, simplifié et assez peu couteux comparativement à une réécriture totale.

Pénalités : zéro. Avantages : plein !

Cross-Plateforme : L'index détaillé des 12 vidéos de formation offertes par Dot.Blog – Vidéos 1 à 6

Pour mieux s'y retrouver voici l'index détaillé des 12 vidéos sur le cross-plateforme publiées ces dernières semaines. Les entrées d'index pointent directement sur chaque vidéo dans YouTube et permettent aussi d'un coup d'œil de voir l'étendu des sujets traités !

8 heures de vidéos gratuites !

8 heures en douze volets, de quoi largement comprendre les mécanismes d'un développement cross-plateforme efficace…

Visual Studio, Xamarin et MvvmCross sont à l'honneur, tout autant que dans chaque démonstration Android, WinRT, Windows Phone ou WPF.

Une série à ne pas louper sur la chaîne YouTube "TheDotBlog"!

Vidéo 1 : Introduction au framework MvvmCross v3 Hot Tuna

Création en direct d'une solution cross-plateforme WinRT (Windows Store/Modern UI) et Android en utilisant Visual Studio 2012, Xamarin 2.0 et MvvmCross V3.

Plateformes de démonstration : WinRT (Windows Store) et Android

Durée : 45 Minutes.

Lien direct : https://www.youtube.com/watch?v=OkHzOqr0zYw
03:50 Création solution cross-plateforme sous VS
04:50 Création du projet portable PCL (noyau)
06:40 Astuce pour afficher Monodroid et MonoTouch dans les plateformes de la PCL
10:27 Ajout de MvvmCross à la PCL
12:00 Le code de App.cs
13:19 Le ViewModel principal

15:24 RaisePropertyChanged
18:10 Le projet Windows Store
19:00 Ajout de MvvmCross
19:39 La Référence au noyau
19:50 Le fichier Setup.cs
20:56 Modification du fichier App.xml.cs
22:20 Suppression de FirstView et ajout d'une "basic page" (avec /Common)
23:46 Modifications de LayoutAwarePage.cs
24:55 Création de la vue
27:30 Exécution dans le simulateur WinRT
28:25 Contourner le manque de trigger xaml WinRT
29:25 Behavior attaché pour INPC
30:14 Exécution : comportement des bindings corrigés
32:32 Ajout de l'application Android
33:25 Ajout de la référence Nuget à MvvmCross et au noyau
34:21 Le fichier Setup.cs
36:33 La vue Android, le binding mvvmcross, le designer visuel Xamarin
39:14 Exécution de l'application Android
41:18 Récapitulation

Vidéo 2 : Services et Injection de dépendance

Mise en place d'un service, utilisation de l'injection de dépendances dans les constructeurs. Le tout cross-plateforme avec une application WinRT (Windows Store) et une application Android.

Plateformes de démonstration : WinRT (Windows Store) et Android

Durée : 15 min.

Lien direct : https://www.youtube.com/watch?v=YWU4E8d_oqQ
00:50 App.cs du noyau et déclaration automatique des services
01:58 Création du service
03:26 Utilisation du service dans le ViewModel
06:08 Modification du projet Android
07:23 Exécution de l'application Android
09:15 Modification du projet WinRT
10:00 Exécution de l'application WinRT
10:36 Récapitulation

Vidéo 3 : Liste bindable, plugins, ImageView, item template

Gérer des listes bindable avec le MvxListView, templater la liste, utiliser le MvxImageView et les plugins de cache et fichier pour afficher des images depuis une Url. Couple de démonstration : Android et WinRT.

Plateformes de démonstration : WinRT (Windows Store) et Android

Durée : 31 min.

Lien direct : https://www.youtube.com/watch?v=JBzj_nkeLFI
00:55 Création du noyau PCL
02:21 Ajout d'un service avec accès images à www.placekitten.com
05:00 Création du ViewModel
06:55 Création du projet Android
08:40 L'UI et la liste MvxListView et ses bindings
10:05 Première exécution de l'application Android
10:28 Création d'un template d'item pour la liste
14:24 Premier run avec le template
16:20 Présentation des plugins MvvmCross
17:32 Ajout des packages Nuget des plugins File et DoawnloadCache
18:18 Amélioration du template d'item avec chargement d'image via l'Url
19:54 Le MvxImageView
21:28 Exécution finale
22:12 Ajout d'un manifest Android
22:54 Création du projet WinRT (Windows Store)
24:39 Ajout d'une vue "Items Page" et ses modifications pour MvvmCross
27:25 Comment adapter une "items page" avec ses templates par défaut
29:55 Vérification du manifeste WinRT
30:05 Exécution finale de l'application templatée

Vidéo 4 : Convertisseurs de valeur, Commandes et Navigation

Episode 4 de cette présentation de MvvmCross V3 avec exemples live sous WinRT et Android. Aujourd'hui : Les convertisseurs de valeur, les commandes, la navigation et la communication de données entre ViewModels lors de la navigation.

Plateformes de démonstration : WinRT (Windows Store) et Android

Durée : 40 min.

Lien direct : https://www.youtube.com/watch?v=tpL5xZHicJI
00:53 Le projet noyau
01:50 Le projet Android

02:25 La View
03:20 Exécution de test
03:45 Les convertisseurs de valeur
04:28 Création d'un convertisseur de démonstration
07:00 Intégration du convertisseur à la vue
08:00 Exécution de test montrant l'effet du convertisseur
09:08 Ajout d'une application Windows Store WinRT
10:30 Création de la vue
12:10 Exécution de test
12:30 Ajout du convertisseur de valeur
15:39 Intégration du convertisseur dans la vue
16:08 Exécution de test montrant l'effet du convertisseur
19:14 La gestion des commandes
19:36 Ajout d'une commande au VM du noyau
22:05 Modification de la vue Android, ajout d'un bouton
23:34 Exécution Android avec la commande
24:20 Les paramètres dans les commandes
25:44 La navigation sous MvvmCross
26:42 Ajout d'un ViewModel pour la page 2 dans le noyau
28:16 Modification du ViewModel n°1 pour ajouter une commande de navigation
29:20 Création de la vue secondaire dans le projet Android
31:45 Exécution de test Android
32:40 Portage vers WinRT
34:44 Exécution application WinRT
35:20 Passage de paramètres de navigation
36:05 Création d'une classe paramètre de navigation
36:40 Modification du 1er ViewModel pour envoyer les paramètres à la page 2
37:00 Modification du 2d ViewModel pour recevoir les paramètres
37:53 Exécution Android
38:40 Transposition dans l'application WinRT (rien à faire)
39:07 Exécution WinRT

Vidéo 5 : Recherches de livres avec l'API Google Books

Utilisation des données de design Blend, mise en place d'une interrogation de la Google Books API et beaucoup d'astuces ! (Partie 1)

Plateformes de démonstration : Windows Phone 8 et Android

Durée : 1 heure

Lien Direct : https://www.youtube.com/watch?v=UkhMH7FgSj4

01:30 Création de la PCL noyau
02:36 Ajout de MvvmCross
03:15 Ajout du plugin Json
03:52 L'API Google books
04:45 Classes de désérialisation c# depuis Json avec l'aide de Json2CSharp.com
06:10 Création d'un service Simple Rest
09:00 L'interface C#
09:26 L'implémentation utilisant le plugin Json
12:00 Les classes utilisées de l'API books
13:00 Création du service spécialisé Google Books
14:30 L'interface C# IBooksService
15:19 L'implémentation BooksService
17:45 Le ViewModel
24:55 L'application UI Windows Phone 8
29:10 Exécution de test
29:57 Création de l'UI WP8 sous Blend
37:10 Exécution de test - la recherche fonctionne
38:15 Données de design sous Blend
39:04 Blend : Create Sample Data
41:25 Personnalisation des données de design de Blend (dont placekitten et lorempixel.com)
45:58 Utilisation des données de design personnalisées sous Blend
49:43 Création de l'Item Template
57:20 Exécution de test (recherche ok avec item templaté)
58:36 Le projet Android
1:01:06 Exécution de test
1:03:27 Récapitulatif

Vidéo 6 : Amélioration de l'application de recherche Google Books

Partie 2 : Amélioration du look & feel des deux applications Windows Phone 8 et Android, convertisseurs de valeur, astuces...

Plateformes de démonstration : Windows Phone 8 et Android

Durée : 32 min.

Lien Direct : https://www.youtube.com/watch?v=sMm1BtUtmmI
01:00 Améliorations sous Android et Windows Phone 8
03:54 Le noyau modifié pour gérer le busy indicator
06:11 Les Value Converters
07:15 Méthodologie : adaptation des données dans le ViewModel ou convertisseurs

de valeurs ?
08:52 L'application Android (splascreen, icone de recherche, les densités d'écran)
10:20 Le Plugin Visibility
12:22 Ressources chaînes localisables (Hint)
13:00 Exécution de test
13:54 Le designer visuel Android de Xamarin, document outline de VS
14:30 Le TableLayout
16:20 Convertisseur de valeur d'inversion booléen, binding multiple
18:00 Le principe des drawable par densité d'écran
19:20 La ProgressBar et son binding
20:11 La MvxListView et ses bindings et l'item template
21:20 Exécution de test
22:30 Le projet Windows phone 8
22:48 Le Plugin Visibility
23:07 Fix Xaml de design time du convertisseur de visibilité
25:23 Les convertisseurs de valeur
27:03 L'UI et les bindings
27:32 Coding4Fun toolkit pour améliorer l'UI
28:00 Icone par fonte symbole
29:42 Comparatif des deux applications au runtime

Cross-plateforme : Index des Vidéos gratuites 7 à 12

Pour mieux s'y retrouver voici l'index détaillé des 6 dernières vidéos sur le cross-plateforme publiées ces dernières semaines. Les entrées d'index pointent directement sur chaque vidéo dans YouTube et permettent aussi d'un coup d'œil de voir l'étendu des sujets traités ! Rappel : Le billet précédent vous propose l'index des 6 premières vidéos.

8 heures de vidéos gratuites !

8 heures en douze volets, de quoi largement comprendre les mécanismes d'un développement cross-plateforme efficace…

Visual Studio, Xamarin et MvvmCross sont à l'honneur, tout autant que dans chaque démonstration Android, WinRT, Windows Phone ou WPF.

Une série à ne pas louper sur la chaîne YouTube "TheDotBlog"!
Les index des vidéos 1 à 6 : http://www.e-naxos.com/Blog/post.aspx?id=ca963d2f-2784-4bbc-8ce8-578226a217fc

Vidéo 7 : Géolocalisation, géo-encodage, messagerie

Dans cette 7eme vidéo (52min full HD) nous abordons la création d'une application retournant des informations de géolocalisation (longitude, latitude, altitude) augmentées par des informations de geocodage (rue, adresse...) et par l'affichage du point dans Google Maps. Session montrant beaucoup de choses, le Monitor du SDK, le debug sur un smartphone réel hors émulateur, etc...

Plateformes de démonstration : CPL, Android et Windows Phone 8

Durée : 52 Minutes.

Lien direct : https://www.youtube.com/watch?v=SJHDKoQ29nU
01:45 Le cycle de vie des VM sous MVX
02:54 Construction (injection de dépendance)
03:22 Initialisation (passage des paramètres)
04:47 ReloadState (réhydratation, tombstoning)
05:24 Start
06:15 L'application de géolocalisation Android
07:50 Plugin Location
08:45 GeoLocationWatcher
11:50 Formatage de données dans le ViewModel
14:49 L'application Android
15:01 Les plugins et les packages MvvmCross
15:16 La vue
16:26 Exécution de test
16:50 L'outil android-sdk\tools : monitor.bat, trace, screenshot, injection de coordonnées gps...
19:20 Geo-encoding et Google Maps
19:45 Service Simple Rest
20:22 Service GeoCodeService
21:19 Le service Google et le résultat Json
22:05 Json2Csharp pour créer la grappe d'objets C# depuis Json
23:33 Utilisation du service d'encodage dans le ViewModel
26:40 Exécution de test
27:45 Affichage de Google Maps
29:45 Les Intents Android pour accéder aux applis internes (dialer, browser, google maps...)
31:28 Ajout de la commande ShowPosition dans le ViewModel
31:56 Commander l'exécution native dans la vue depuis le noyau via le Messenger
32:34 Le plugin Messenger de MvvmCross
32:30 Modification du ViewModel pour intégrer le Messenger
34:39 Création d'une classe message
35:27 Publication du message par le ViewModel

38:12 Récupération du service et traitement du message dans la vue Android (côté natif)
40:35 Le principe du token sur le Subscribe du message
42:28 Exécution de test
44:28 Exécution sur une device réelle (Samsung S3), utilisation du monitor Android
47:00 Récapitulation, transposition pour créer l'UI Windows Phone
49:00 Bing maps, encodage URL comme pour Google Maps

Vidéo 8 : Gérer des données avec SQLite

Présentation de la gestion des données en cross-plateforme avec Visual Studio, Xamarin et MvvmCross sous Windows Phone et Android. (HD 720p / 40 min).

Plateformes de démonstration : CPL, Android et Windows Phone 8

Durée : 40 Minutes.

Lien direct : https://www.youtube.com/watch?v=uXelZV41VKQ
00:35 Introduction
04:36 Le noyau PCL cross-plateforme
05:42 Le Modèle, l'utilisation des attributs spécifiques
07:12 Le service de données
11:27 Génération des données de test
13:05 Lorem Ipsum (adresse de l'application : http://www.e-naxos.com/slsamples/lorem/loremgen.html)
14:14 Le ViewModel
17:06 L'application Android
18:34 Le plugin SqLite
21:04 La View
23:00 Exécution de test
24:15 Persistance des données, filtrage
26:49 Récapitulatif de l'application Android
29:18 L'application Windows Phone 8
31:25 La View
32:47 Copier-coller depuis le Xml Android vers Xaml
36:34 Exécution de test
38:42 Exécution après création du DataTemplate Xaml
39:55 Récapitulatif

Vidéo 9 : Gérer le grand écart Android / WPF

Cette 9ème vidéo montre comment développer une application d'envoi de mail sous WPF et Android avec le même cœur grâce à MvvmCross. La mise en place du projet WPF utilise des techniques non vues dans les vidéos précédentes. Grand écart technologique, d'un Linux pour unité mobile (Android) à une plateforme vectorielle de bureau, cette vidéo démontre tout l'intérêt du couple Android/WPF pour couvrir 90% du parc d'ordinateurs mobiles, portable ou de bureau. Windows Phone 8 et WinRT ayant été largement couverts dans les autres vidéo, il était temps de rendre à WPF les honneurs qu'il mérite.(HD 720 / 39 Min)

Plateformes de démonstration : CPL, Android et WPF

Durée : 39 Minutes.

Lien direct : https://www.youtube.com/watch?v=uZ7vP9_qkho
01:00 Introduction
02:48 Le noyau PCL
04:00 Le ViewModel
04:53 Les services injectés (Email et Messenger)
09:37 L'application Android
12:00 DebugTrace
12:40 La vue
15:14 Gestion des erreurs sous Android
16:10 Les bindings Android avec gestion des erreurs
17:36 Exécution de test
18:59 L'application WPF
19:25 Modern UI pour WPF (pb entre MvvmCromms Model-First et <mu:i> qui est View-First)
21:30 MahApps Metro, framework identique mais seulement le visuel
22:24 Reference MahApps
22:50 Attention lors de l'intégration de MvvmCross ! (fichiers écrasés et fichiers en trop)
22:30 La MainView (MetroWindow) qui contient le look mais aussi le conteneur des "activités"
24:30 Interception de l'affichage MvvmCross pour gérer le docking dans MainWindow
25:05 Création de Program.cs pour lancer l'appli
25:33 Création de l'instance du "Presenter"
26:09 Propriété du projet : choix du nouveau point d'entrée
26:25 Setup.cs
27:00 Modifications de App.xaml.cs (start)
27:23 Le code du Presenter custom
30:00 La vue

32:15 Comparaison des similitudes dans la partie C# des vues Android et Wpf
33:57 Exécution de test
35:24 Comparaison visuelle simultanée des deux applications

Video 10 : Codez comme un Ninja

Présentation courte de l'extension Visual Studio "Ninja Coder for MvvmCross" qui permet la mise en place rapide d'une solution cross-plateforme en y intégrant tous les projets nécessaires, jusqu'aux tests NUnit si on le désire.(HD 720 / 10 min).

Plateformes de démonstration : toutes

Durée : 10 Minutes.

Lien direct : https://www.youtube.com/watch?v=JMQw5fpcpY0
00:40 Introduction Ninja Coder for MvvmCross
01:51 Télécharger le msi d'installation
02:00 Installation dans Visual Studio
02:27 Lancement de Visual Stido, Tools "Ninja coder"
02:49 Ajouter un projet MvvmCross avec Ninja coder
03:25 La solution et ses projets créés par Ninja coder
04:00 Le BaseViewModel dans le noyau
04:29 La BaseView côté Android et autres UI
05:22 Les références de MvvmCross sont intégrées (mais pas en packages Nuget)
06:45 Le projet de test Nunit et Mock
07:40 Ajout de plugins via Ninja coder
08:28 Conclusion

Vidéo 11 : Multibinding, Swiss Binding, Tibet Binding, Rio

Présentation du nouveau binding riche apporté par MvvmCross : le swiss et le tibet Binding, RIO pour concevoir des ViewModels très rapidement, le tout démontré sous Android et WinRT (Xaml). 43 min / HD 720.

Plateformes de démonstration : WinRT / Android

Durée : 43 Minutes.

Lien direct : https://www.youtube.com/watch?v=yIr3BbA0t68
00:37 Introduction
02:00 Création de la solution avec Ninja Coder
03:24 Le noyau CPL

04:34 Le ViewModel
05:48 Les convertisseurs
06:40 L'application Android
07:00 La view (mvxspinner, mvxlistview...)
07:41 Execution de démo
09:20 Binding multi propriétés, le swiss binding
10:21 La syntaxe MvvmCross du binding
10:50 Binding avec expression, Tibet binding
11:50 Convertisseurs utilisés en fonction
13:13 Exécution de démo
14:44 Multibinding, réévalutation des expressions sur INPC
17:20 Expressions numériques
18:21 Exécution de démo
19:20 Expressions chaînes
19:40 Exécution de démo
22:00 Expression de formatage
22:40 Exécution de démo
23:10 Expression IF
23:45 Exécution de démo
24:50 Résumé swiss binding et tibet binding
26:29 La documentation sur github
29:10 Rio
29:45 INC et NC
30:20 Method binding
32:35 Utilisation étendue à Xaml
33:25 Les binaires à jour sur github en attendant le package nuget
34:55 Exécution de démo de l'appli Windows Store
36:40 La view Xaml
36:55 Le mvx:Bi.ind à la place du binding Xaml
38:59 Le package spécifique pour Xaml ajouté en référence
39:40 Initialisation à la main du package BindingEx dans Setup.cs
40:00 Conclusion

Video 12 : Injection de code natif dans le noyau, méthode directe et création de Plugin

Dans ce 12ème volet consacré au développement cross-plateforme (VS2012, Xamarin et MvvmCross) est abordée l'injection de code natif selon plusieurs méthodes dont la création de plugin MvvmCross. (42min HD 720p).

Plateformes de démonstration : WinRT / Android

Durée : 42 Minutes.

Lien direct : https://www.youtube.com/watch?v=fRqOnJ1x0ug
00:38 Introduction
02:54 La solution de test
03:31 Le noyau - méthode 1 (directe, par injection de dépendance)
04:06 Création du service IScreenSize
05:00 Le ViewModel
07:14 La View Android (binding au VM)
08:36 Ajout d'une classe d'implémentation IScreenSize dans l'app Android
10:13 Modification de Setup.cs Android pour l'IoC (dans InitializelastChance)
13:20 Exécution de démo
14:00 Résumé du principe
15:04 Les différentes façons d'enregistrer le code natif dans l'e conteneur d'IoC
16:54 La même chose sous Windows Store
18:50 La Vue
19:23 Exécution de démo
19:34 Résumé de la manipulation
19:54 Comparatif des deux applications côte à côte
21:15 Seconde stratégie : création d'un plugin réutilisable
22:05 Création du folder et du noyau du plugin
23:00 Référencement du CrossCore uniquement
23:27 Création du service noyau
24:00 La classe PluginLoader du noyau
25:30 Ajout du projet android de plugin
25:50 Référencement noyau et CrossCore
26:07 Classe d'implémentation native Android
27:00 Plugins.cs dans le projet natif
27:38 Ajout projet Windows Store de plugin
28:18 Référencement noyau et CrossCore
28:50 Classe d'implémentation native Windows Store
29:38 Plugins.cs dans le projet natif
29:58 Récapitulation de la création du plugin
31:24 Ajout noyau et impl native dans le projet Android final
31:45 Ajout du Bootstrap dans le projet Android final
33:15 Modification du ViewModel noyau du projet final pour utiliser le plugin (aurait du être fait en 1er!)
34:40 Modification de la View Android pour utiliser la nouvelle propriété
35:02 Exécution de test Android
35:23 Modification du projet final Windows Store
35:30 Ajout classe Bootstrap
36:29 Modification de la View Windows Store pour afficher la propriété traitée par le

plugin
36:49 Exécution de démo
37:00 Résumé de la séquence de création et d'utilisation d'un plugin MvvmCross
39:30 L'exemple du tutor GoodVibrations

Le code source des 12 vidéos sur le cross-plateforme

Le code source utilisé dans les 12 vidéos traitant du développement cross-plateforme est disponible en téléchargement sous la forme d'un seul fichier zip. Plus de 30 projets et 10 solutions sous Visual Studio 2012 utilisant Xamarin 2.0 et MvvmCross v3 le tout sur les plateformes Windows Store (WinRT), Windows Phone 8, WPF et Android.

Le fichier (environ 16 Mo) : videomvxsource.zip

L'IoC dans MvvmCross – Développement Cross-Plateforme en C#

Présentation

MvvmCross est un jeune framework cross-plateforme dont j'ai déjà parlé sur Dot.Blog en l'intégrant à une stratégie de développement plus globale pour le développement cross-plateforme (voir la partie 1 ici : http://www.e-naxos.com/Blog/post.aspx?id=f4a7648e-b4a3-468d-8901-aa9a5b9daa2f).

La version actuelle, parfaitement utilisable, est en train d'être modifiée pour créer « vNext » qui sera disponible prochainement. Je présenterai un article complet sur le framework à cette occasion.

Pour l'instant je vous propose d'aborder un thème bien ciblé qu'on retrouve dans de nombreux frameworks : l'Inversion de Contrôle.

L'exemple sera pris dans le code de MvvmCross, comme un avant-goût de l'article complet à venir…

Inversion de Contrôle et Injection de Dépendance

MvvmCross, comme de nombreux frameworks MVVM, propose de régler certains problèmes propres aux développements modernes avec une gestion d'Inversion de Contrôle et

d'Injection de Dépendance. Il n'y a pas de lien direct entre MVVM et ces mécanismes. Ici, c'est avant tout la portabilité entre les plateformes qui est visée. Les applications utilisant MvvmCross pouvant bien entendu utiliser l'IoC pour leurs propres besoins.

La partie chargée de cette mécanique particulière se trouve dans le sous-répertoire `IoC` du framework, dans chaque projet spécifique à chaque plateforme. Pour gérer cette fonctionnalité Stuart Lodge est parti d'un framework existant, OpenNETCF.IoC (http://ioc.codeplex.com/).

Ce framework est lui-même une adaptation de SCSF (Smart Client Software Factory, http://msdn.microsoft.com/en-us/library/aa480482.aspx) et de CAB de Microsoft mais en version « light » et cross-plateforme adaptée aux unités mobiles.

OpenNETCF.IoC propose une gestion de l'IoC par le biais d'attributs, soit de constructeur, soit de champ, un peu à la façon de MEF (mais en moins automatisé que ce dernier). Il sait créer lors du runtime des liens entre des propriétés ou des constructeurs marqués (importation) et des services exposés (exportation) via le conteneur central, sorte de registre tenant à jour la liste des services disponibles.

Bien qu'utilisant une partie du code d'OpenNETCF.IoC, MvvmCross ne l'expose pas directement et propose ses propres services d'IoC (construits sur le code modifié d'OpenNETCF.IoC). D'ailleurs dans « vNext » cet emprunt s'approche de zéro, seule la partie conteneur étant conservée. Cela ne change rien aux explications ici présentées puisque MvvmCross propose sa propre gestion d'IoC qui ne change pas, peu importe le code qui l'implémente (isolation et découplage ont du bon, pour tous les types de projets !).

Même s'il peut être intéressant de regarder de plus près le fonctionnement de OpenNETCF.IoC il n'est donc pas nécessaire de connaître celui-ci pour faire de l'IoC sous MvvmCross qui expose ses propres mécanismes (même si, en interne, ceux-ci utilisent pour le moment un code emprunté à OpenNETCF.IoC…).

Le lecteur intéressé spécifiquement par OpenNETCF.IoC est invité à consulter la documentation de cette librairie que je ne détaillerai pas ici, au profit d'explications sur ce qu'est l'IoC et comment MvvmCross a choisi d'offrir ce service, ce qui sera bien plus utile au lecteur je pense.

De quoi s'agit-il ?

Le problème à résoudre est assez simple dans son principe : imaginons une classe dont le fonctionnement dépend de services (au sens le plus large du terme) ou de composants dont les implémentations réelles ne seront connues qu'au runtime. Comment réaliser cet exploit ? C'est toute la question … qui trouve sa réponse dans les principes de l'Inversion de Contrôle et l'Injection de Dépendance.

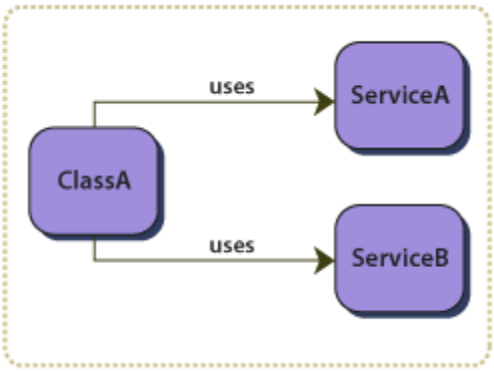


Figure 1 - L'inversion de Contrôle, l'objectif

Les problèmes posés par l'approche « classique »

Selon ce cahier des charges, dès qu'on y réfléchit un peu, on voit surgir quelques problèmes épineux si on conserve une approche « classique » :

- Pour remplacer ou mettre à jour les dépendances vous devez faire des changements dans le code de la classe (la classe A dans le petit schéma ci-dessus), cela n'est bien entendu pas ce qu'on veut (c'est même tout le contraire !).
- Les implémentations des services doivent exister et être disponibles à la compilation de la classe, ce qui n'est pas toujours possible dans les faits.
- La classe est difficile à tester de façon isolée car elle a des références « en dur » vers les services. Cela signifie que ces dépendances ne peuvent être remplacées par des mocks ou des stubs (maquette d'interface ou squelette non fonctionnel de code).
- Si on souhaite répondre au besoin, la classe utilisatrice contient alors du code répétitif pour créer, localiser et gérer ses dépendances. On tombe vite dans du code spaghetti…

Ce qui force à utiliser une autre approche

De cette mécanique de « remplacement à chaud » de code par un autre, ou tout simplement d'accès à un code inconnu lors de la compilation se dégage des besoins :

- On veut découpler les classes de leurs dépendances de telle façon à ce que ces dernières puissent être remplacées ou mises à jour sans changement dans le code des classes utilisatrices.
- On veut pouvoir construire des classes qui utilisent des implémentations inconnues ou indisponibles lors de la compilation.
- On veut pouvoir tester les classes ayant des dépendances de façon isolée, sans faire usage des dépendances.
- On souhaite découpler la responsabilité de la localisation et de la gestion des dépendances du fonctionnement des classes utilisatrices.

La solution

Dans l'idée elle est simple : il faut déléguer la fonction qui sélectionne le type de l'implémentation concrète des classes de services (les dépendances) à un composant ou une bibliothèque de code externe au code utilisateur des services et utiliser un mécanisme d'initialisation des classes qui sache « remplir les trous » (des variables) avec les bonnes instances.

Cette solution fait l'objet d'un pattern, l'Inversion de Contrôle (IoC).

Les implémentations de l'IoC

Il existe plusieurs façons d'implémenter l'IoC. Par exemple Prism en utilise deux qui sont l'Injection de Dépendance (Dependency Injection) et le Localisateur de service (Service Locator).

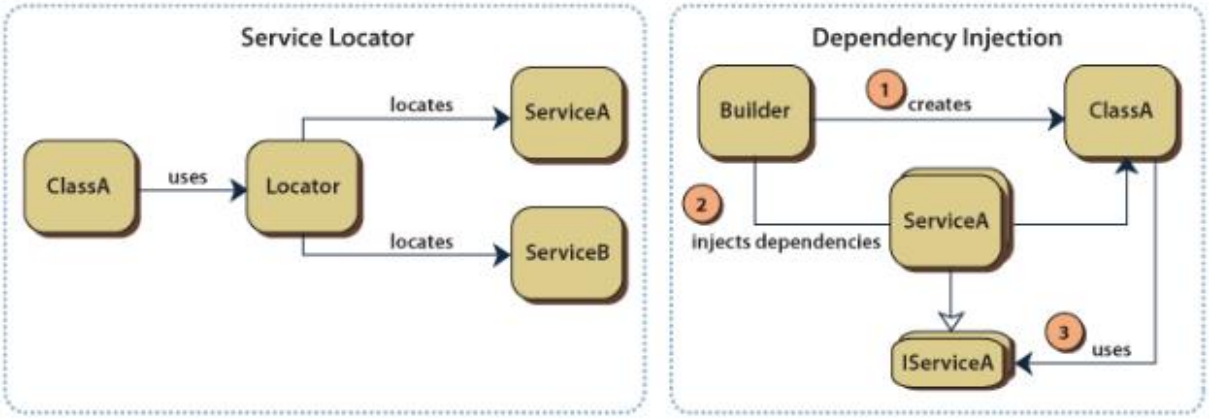


Figure 2 - Injection de Dépendance et Localisateur de Service

L'injection de Dépendance

L'injection de dépendance est une façon d'implémenter le principe de l'Inversion de contrôle, avec ses avantages et ses limitations, ce qui justifient parfois, comme Prism le fait, d'en proposer d'autres.

L'Injection de Dépendance consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant généralement sur une description (fichier de configuration ou autre). Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

C'est ce qu'on peut voir sur le schéma ci-dessus, à droite (Dependency Injection).

Il existe un « `Builder` » par lequel on passe pour créer des instances de `classA`. Ce builder ou factory recense les services proposés, comme ici le `ServiceA`, qui exposent des interfaces, ici `IServiceA`. La classe consommatrice des services (`classA`) se voit ainsi « injecter » l'implémentation concrète `ServiceA` lors de son initialisation. Elle ne connait pas cette classe concrète et ne fait qu'utiliser une variable de type `IServiceA`. C'est cette

variable que le `Builder` initialisera après avoir construit l'instance de `ClassA`. Le code de `classA` n'est donc pas lié « en dur » au code de `ServiceA`. En modifiant une configuration (dans le code ou dans un fichier) il est possible d'indiquer au `Builder` d'injecter un service `ServiceA` bis ou ter sans que cela n'impose une quelconque modification de `ClassA`.

L'inconvénient majeur de ce procédé est d'obliger la création de `ClassA` via le `Builder`. Il faut donc que le code applicatif soit pensé dès le départ en fonction de ce mode particulier d'instanciation qui peut ne pas convenir dans certains cas de figure.

Une implémentation possible de l'Injection de Dépendance se trouve par exemple dans MEF qui fait aujourd'hui partie de .NET. C'est d'ailleurs la solution retenue par le framework MVVM Jounce auquel j'ai consacré un très long article vers lequel je renvoie le lecteur intéressé.

MEF efface l'inconvénient de l'Injection de Dépendance telle que je viens de la décrire en automatisant le processus, c'est-à-dire en cachant le `Builder`. Les classes ou champs sont marqués avec des attributs (d'exportation ou d'importation) qui sont interprétés lors de leur instanciation. MEF se chargeant de tout.

MEF n'étant pas disponible sur toutes les plateformes, MvvmCross a été dans l'obligation de proposer une approche personnalisée ne se basant par sur MEF, au contraire d'un framework comme Jounce qui, en retour, n'existe que pour Silverlight…

Le localisateur de Service

Le localisateur de services est un mécanisme qui recense les dépendances (les services) et qui encapsule la logique permettant de les retrouver.

Le localisateur de services n'instancie pas les services, il ne fait que les trouver, généralement à partir d'une clé (qui permet d'éviter les références aux classes réelles et permet ainsi de modifier les implémentations réelles).

Le localisateur de services propose un procédé d'enregistrement qui liste les services disponibles ainsi qu'un service de recherche utilisé par les classes devant utiliser les dépendances et qui se chargent elles-mêmes de les instancier. Le localisateur de service peut ainsi être vu comme un catalogue de types (des classes) auquel on accède par des clés masquant le nom de ces types.

C'est ce qu'on peut voir dans la partie gauche du schéma ci-avant (Service Locator).

Le framework MVVM Light utilise une version très simplifiée du localisateur de service pour découpler les Vues des ViewModels (ce code se trouve dans le fichier `ViewModelLocator.cs`). En revanche MVVM Light ne propose pas d'Injection de Dépendance et son interprétation du localisateur de services restent assez frustre bien qu'étant souvent suffisante et ayant été construit dans l'optique du support de la blendabilité, ce que MvvmCross ne propose pas encore.

On le voit encore dans cet article comme dans les nombreux autres que j'ai écrits sur MVVM qu'hélas il n'existe pas encore d'unification des implémentations de ce pattern ni des services qui gravitent autour, chaque framework proposant « sa » vision des choses. Il ainsi

bien difficile pour le développeur qui ne les connait pas tous de faire un choix éclairé. Sauf en lisant régulièrement Dot.Blog, bien entendu !

J'en profite pour rappeler au lecteur que j'ai traité MVVM Light dans un très long article, et ici aussi je ne pourrais que renvoyer le lecteur intéressé à ce dernier (on retrouve facilement tous ces articles traitant de MVVM en faisant une recherche sur ce terme dans Dot.Blog).

L'IoC dans MvvmCross

Maintenant que les principes de l'IoC ont été exposés, il est temps d'aborder la façon dont MvvmCross aborde cette problématique.

Dans le principe nous avons vu que MvvmCross utilise un code modifié de OpenNETCF.IoC, mais que ce dernier n'est pas directement exposé au développeur, il est juste utilisé en interne. A la place MvvmCross propose d'autres procédés.

Quels sont-ils ?

Services et consommateurs

Le monde de l'IoC est très manichéen : les classes peuvent être rangées en trois catégories, celles qui proposent des services, celles qui les consomment, et … celles qui ne participent pas l'IoC !

Comme dans toutes les sociétés il existe tout de même des individus plus difficiles à classer que les autres, notamment ici les classes qui exposent des services tout en en consommant d'autres. Cela est parfaitement possible et c'est la gestion d'IoC qui s'occupe de détecter et d'éviter les boucles infinies qui pourraient se former. Par exemple grâce à l'utilisation de singletons. Mais la charge de ce contrôle de la logique de l'application reste le plus souvent au développeur.

La notion de service

Un service est proposé par une classe. Une même classe peut, éventuellement, offrir plusieurs services, mais cela n'est pas souhaitable (le remplacement d'un service entraînant en cascade celui des autres, situation qui fait perdre tout son charme à l'IoC).

En revanche plusieurs classes ne peuvent pas au même moment proposer le même service. C'est une question de logique, le gestionnaire d'IoC ne saurait pas quelle instance ou quelle classe retourner (sauf à faire un tirage au sort ce qui pourrait déboucher sur un concept intéressant de programmation régit par les lois du chaos…).

Il y a deux façons de proposer un service sous MvvmCross : soit en enregistrant une instance d'une classe précise (un singleton), soit en enregistrant un type concret.

La nuance a son importance et sa raison d'être.

Certains services sont « uniques » par nature. Par exemple on peut proposer, comme MvvmCross le fait, un service de log, ou bien le support du cycle de vie d'une application (tombstoning notamment). On comprend aisément que de tels services ne peuvent exister en même temps en plusieurs exemplaires dans la mémoire de l'application, cela n'aurait

aucun sens. Une application n'a qu'un seul cycle de vie, et, pour simplifier les choses, un système de log ne peut pas exister en différentes versions à la fois.

Autant le premier cas (cycle de vie) est une question de logique pure, autant le second (le log) est un choix assumé par MvvmCross. On pourrait supposer qu'il soit possible d'enregistrer plusieurs services de logs (une trace dans un fichier texte et un système d'envoi de messages via Internet par exemple) et qu'ils soient tous appelés sur un même appel au service de log.

Cela compliquerait inutilement l'implémentation du framework alors même que l'un de ses buts est d'être le plus léger possible puisque tournant sur unités mobiles. On comprend dès lors cette limitation de bon sens.

Ainsi ces services uniques doivent-ils être proposés par des singletons (vrais ou « simulés » par la gestion de l'IoC).

Le développeur prudent implémentera ces services sous la forme de singletons vrais, c'est-à-dire de classes suivant ce pattern. Toutefois la gestion de l'IoC est capable de reconnaitre un tel service unique et sait fournir aux consommateurs l'unique instance enregistrée. De ce fait la classe de service peut ne pas être un singleton vrai sans que cela ne soit vraiment gênant (sauf qu'il n'existe pas de sécurité interdisant réellement la création d'une autre instance par d'autres moyens, situation qu'un singleton vrai verrouille).

D'un autre côté il existe des services qui ont tout intérêt à être ponctuels ou du moins à pouvoir exister en plusieurs instances au même moment. Soit qu'ils sont utilisés dans des threads différents gérant chacun des tâches distinctes, soit pour d'autres raisons.

C'est le cas par exemple du service d'effet sonore proposé par MvvmCross. A l'extrême on peut imaginer un jeu gérant plusieurs objets sur l'écran, chacun émettant un son différent. Le service d'effet sonore ne doit pas être un singleton qui ne jouerait qu'un seul son à la fois.

MvvmCross propose de nombreux services, tous, en dehors du Log et de la gestion du cycle de vie de l'application sont de types multi-instance.

Le choix est parfois évident, comme pour les effets sonores, il l'est moins pour d'autres services comme la composition d'un numéro de téléphone ou le choix d'une image dans la bibliothèque de la machine. Ces tâches imposent un dialogue ou un fonctionnement qui ne peuvent exister qu'en un seul exemplaire au même moment.

Alors pourquoi leur donner la possibilité d'être multi-instance ?

La raison est fort simple : qui dit capacité multi-instance suppose le support du cas de zéro instance, là où le singleton oblige un minimum d'une instance…

La création de services uniques sous la forme d'instances statiques est très logique pour des services ne pouvant exister qu'en un seul exemplaire à la fois, mais hélas cela impose de créer au moins instance dès le départ, le singleton.

Or, beaucoup de services utilisent des ressources machine, la gestion des sons par exemple ou la création d'un email ou l'activation d'un appel téléphonique. Cela n'aurait ni sens ni

intérêt d'instancier toute une liste de services de ce type alors même que l'application n'en fera peut-être jamais usage et que cela va gonfler inutilement sa consommation (mémoire, CPU, électrique).

On comprend alors mieux pourquoi la majeure partie des services exposés par MvvmCross est de type « classe » et non pas « instance ». La véritable raison est donc bien plus motivée par la possibilité d'avoir zéro instance que d'en avoir plusieurs.

L'enregistrement d'un service, le concept

Je dis de type « classe » ou « instance » car c'est de cette façon que MvvmCross va enregistrer un service et faire la différence entre ceux qui peuvent exister en zéro ou plusieurs exemplaires et ceux qui existent dès le départ, et au maximum, en un seul exemplaire.

Quand je dis « classe », je dois dire plus exactement « interface » puisque l'IoC de MvvmCross se fonde sur l'utilisation d'interfaces.

En effet, rappelons-nous que tout cela n'est pas seulement un libre-service dans lequel on vient piocher mais qu'il s'agit avant tout de créer un découplage fort entre fournisseurs et consommateurs de services en utilisant les concepts propres à l'IoC. Les interfaces sont la clé de voute de ce découplage, le reste n'est qu'enrobage.

De fait, l'enregistrement d'un service s'effectue d'abord en passant le type de son interface à la méthode chargée de cette fonction.

En second lieu seulement on passera soit l'instance du singleton, soit le type de la classe d'implémentation.

L'IoC, comme je le disais bien plus haut, peut être vu comme un registre, un dictionnaire dont la clé est le type de l'interface. On revient aux principes de l'IoC exposés précédemment, par exemple au localisateur de service qui utilise des clés pour masquer les noms des classes réelles.

Pour enregistrer un service unique il faut avoir accès au premier de ceux-ci : le gestionnaire d'IoC.

MvvmCross s'initialise en créant le gestionnaire d'IoC d'une façon qui elle-même autorise ce service à être proposé en différentes implémentations (cross-plateforme donc).

Mais Qui de la poule ou de l'œuf existe en premier me demanderez-vous ?

L'œuf.

Non, je plaisante, tout le monde sait que c'est la poule.

MvvmCross utilise lors de son initialisation un autre procédé (un marquage via un attribut) respectant le concept de l'IoC qui lui permet de chercher dans l'espace de noms de l'application où se trouve le service d'IoC.

Il ne peut y en avoir qu'un seul et une exception est levée si d'aventure deux se trouvaient en présence ou si aucun ne pouvait être trouvé. Stuart Lodge a voulu ici se protéger contre

tout changement ou évolution de son propre framework et s'est appliqué à lui-même les principes qu'il a adoptés par ailleurs et qu'il propose via la gestion des IoC dans MvvmCross. Dans les conditions normales d'utilisation aucune des deux situations ne peut se présenter bien entendu. MvvmCross est fourni avec un seul service d'IoC qui fait partie du framework.

Donc il existe un premier service qui est le gestionnaire des services.

Comme tout service il faut pouvoir y accéder, ici impossible de faire appel à l'IoC puisque c'est justement lui auquel on veut accéder…

C'est au travers de méthodes d'extension que le service d'IoC va être disponible. Mais je vais y revenir.

Comme je l'ai évoqué, MvvmCross enregistre lui-même de nombreux services qui sont directement accessibles dans toutes les applications. J'ai parlé du service de log (`Trace`), de celui émettant des évènements pour gérer le cycle de vie de l'application, mais aussi du sélectionneur d'image, du composeur de numéro de téléphone, etc.

Toutefois une application peut décider d'exposer d'autres services que les ViewModels pourront exploiter comme, par exemple, un service de remontée des erreurs de fonctionnement. D'autres types de services peuvent aussi être fournis pour remplacer des briques du framework. On peut trouver dans les exemples fournis avec MvvmCross quelques utilisations de ce type.

L'enregistrement d'un service, le code

Maintenant que savons ce que sont les services, pourquoi il existe deux types d'enregistrement et comment accéder à l'IoC, il ne reste plus qu'à voir quel code permet d'effectuer dans la pratique ces enregistrements.

Le premier cas est celui du service unique. Comme le service de gestion du cycle de vie des applications. Il est enregistré comme suit dans le code de MvvmCross :

```
RegisterServiceInstance<IMvxLifetime>(new
MvxWindowsPhoneLifetimeMonitor());
```

`RegisterServiceInstance` permet d'enregistrer l'instance unique d'un service unique. L'enregistrement commence par le type de l'interface (la clé pour le dictionnaire de l'IoC), ici `IMvxLifeTime` puis est complété par un paramètre acceptant une instance de tout objet (du moment qu'il supporte l'interface déclarée).

Le second cas est celui d'un service « optionnel », ce terme convenant mieux que celui de « multi instance » puisqu'il souligne le fait que le service peut ne jamais être instancié. Prenons l'exemple du service de sélection d'image, MvvmCross l'enregistre de la façon suivante :

```
RegisterServiceType<IMvxPictureChooserTask,
MvxPictureChooserTask>();
```

`RegisterServiceType` permet cette fois-ci d'enregistrer un type et non une instance. On trouve comme précédemment en premier lieu l'interface qui est exposée, ici `IMvxPictureChooserTask`. Mais au lieu de supporter un paramètre permettant de passer l'instance du service, la méthode générique accepte un second type, celui du service (implémentation concrète), ici `MvxPictureChooserTask`. Aucun paramètre n'est passé, la déclaration générique des deux types, l'interface et la classe de service, suffisent à la méthode pour faire son travail.

La consommation des services

La consommation des services sous MvvmCross s'effectue en demandant à l'IoC de retourner une instance de celui-ci après lui avoir fourni la clé, c'est-à-dire le type de l'interface, du service.

Les méthodes permettant d'accéder aux services ne sont pas directement héritées, elles sont implémentées sous la forme de méthodes d'extension pour le type `IMvxServiceConsumer`. Le principe reste le même que pour l'enregistrement des services.

Pourquoi ce choix ?

Plusieurs possibilités étaient envisageables pour que le code de l'application puisse accéder aux services. Un moyen simple aurait été de proposer une classe statique regroupant les méthodes d'accès aux services quels qu'ils soient.

C'est finalement ce qui est fait, mais au lieu d'exposer une telle classe statique, il existe des méthodes d'extension qui lui donne accès, créant ainsi une indirection plus sophistiquée.

Toute classe qui désire consommer un service doit le demander en supportant l'interface générique `IMvxServiceConsumer<TService>`. L'héritage multiple des interfaces existant en C#, il est donc possible pour une classe consommatrice de services d'hériter autant de fois que nécessaire de `IMvxServiceConsumer` en indiquant à chaque fois le type de service consommé.

Ce choix possède toutefois un inconvénient majeur : puisque la classe consommatrice « supporte » des interfaces, elle doit les implémenter, la délégation d'implémentation d'interface n'existant pas en C#.

C'est une situation bien embêtante…

La façon de régler le problème est intéressante : il suffit que l'interface en question soit vide et de créer des méthodes d'extension qui ne sont accessibles qu'aux types supportant `IMvxServiceConsumer`…

Parmi les exemples de code fournis, on peut trouver un code de ce type :

```
public class BaseViewModel
        : MvxViewModel
        , IMvxServiceConsumer<IMvxPhoneCallTask>
        , IMvxServiceConsumer<IMvxWebBrowserTask>
        , IMvxServiceConsumer<IMvxComposeEmailTask>
           , IMvxServiceConsumer<IMvxShareTask>
```

```
        , IMvxServiceConsumer<IErrorReporter>
```

La classe `BaseViewModel`, la classe de base pour tous les ViewModels de l'application « Conférence », hérite d'abord de `MvxViewModel`, puisque c'est un ViewModel, puis elle hérite « faussement » de `IMvxServiceConsumer` autant de fois que nécessaire pour déclarer les services qu'elle désire consommer comme `IMvxPhoneCallTask` ou `IMvxComposeEmailTask`, services de base de MvvmCross, ou `IErrorReporter` qui est un service propre à cette application.

Dès lors, cette classe peut utiliser les méthodes d'extension permettant d'accéder à l'IoC. Ce qu'elle fait en exposant à ses héritiers des accès simplifiés. Voici un court extrait du code de la classe exemple :

```
        protected void ReportError(string text)
        {
            this.GetService<IErrorReporter>().ReportError(text);
        }

        protected void MakePhoneCall(string name, string number)
        {
            var task = this.GetService<IMvxPhoneCallTask>();
            task.MakePhoneCall(name, number);
        }
```

La méthode `ReportError(string text)` sera disponible à tous les ViewModels de l'application (qui hériteront de cette classe), méthode qui appelle `GetService<IErrorReporter>()` pour obtenir le service de gestion des erreurs propre à cette application. `GetService` est disponible via les méthodes d'extension pour les types supportant `IMvxServiceConsumer<TService>`.

La même chose est faite pour le service MvvmCross qui permet d'initier un appel téléphonique.

Très souvent les applications se contentent d'utiliser `GetService` dont le comportement dépend du type de service (unique ou optionnel). Si le service est unique (enregistré par son instance) c'est l'instance unique qui est retournée systématiquement, si le service est optionnel (enregistré par son type), c'est à chaque fois une nouvelle instance qui est renvoyée.

Il est donc nécessaire de porter une attention particulière à l'utilisation des services optionnels pour éviter le foisonnement des instances en mémoire…

Il existe deux autres méthodes d'extension pour les consommateurs de services :

- `Bool IsServiceAvailable<TService>()`, qui permet de savoir si un service est supporté ou non.

- `bool TryGetService<TService>`, qui permet d'éviter la levée d'une exception en cas d'impossibilité d'accès au service.

On notera que puisqu'il existe en réalité une instance statique fournissant les services de l'IoC « cachés » derrière les méthodes d'extension il est donc théoriquement possible de se passer de la déclaration des interfaces des services consommés et d'accéder directement à cette instance statique.

Et c'est en effet possible, depuis n'importe quel code il est par exemple possible d'écrire :

```
var c = MvxServiceProviderExtensions.GetService<IMvxTrace>();
            c.Trace(MvxTraceLevel.Error, "test", "coucou");
```

Il suffit de déclarer le bon "`using`" pour accéder au code des méthodes d'extension. Parmi celles-ci se trouve des variantes qui ne sont pas des méthodes d'extensions, mais de simples accès à la fameuse instance statique. Mieux, les méthodes d'extension ne font en réalité qu'appeler ces méthodes simples. Ces dernières ne faisant, *in fine*, qu'utiliser l'instance statique `MvxServiceProvider.Instance` qu'on pourrait d'ailleurs utiliser directement en se passant totalement de l'intermédiaire des méthodes d'extension.

Néanmoins je ne conseille pas d'accéder à l'IoC de cette façon, en l'absence de documentation officielle complète et dans l'attente de la sortie de « vNext », les seules indications données par les exemples supposent de déclarer les interfaces puis d'accéder à l'IoC via les méthodes d'extension.

On peut se demander pourquoi une telle indirection existe alors qu'elle ne semble pas très utile… Attendons « le coup de ménage » et les évolutions de « vNext » avant de nous prononcer d'autant que beaucoup de services proposés comme tels aujourd'hui sont, dans « vNext », proposés sous la forme de plugins, une approche plus modulaire (mais respectant toujours l'IoC présenté ici).

vNext puis V3

MvvmCross « vNext » était en cours de développement à l'écriture de ce billet, aujourd'hui c'est la V3 dite « Hot Tuna » qui la remplace avec de très nombreuses améliorations qui ont été présentées ici et dans les 12 vidéos consacrées au cross-plateforme.

Les principes de l'IoC présentés ici, et la façon de MvvmCross propose cette fonction aux applications ne changent pas.

Toutefois le code sous-jacent est devenu plus clair et beaucoup plus simple.

C'est pourquoi cet article s'arrête là pour l'instant car il serait stupide d'entrer dans les détails de la version actuelle alors même que vNext est aujourd'hui dépassé et que la V3 la remplace.

Conclusion

MvvmCross a beaucoup d'intérêt, c'est une librairie qui permet de régler en partie l'épineux problème de la portabilité des applications entre les différentes plateformes, qu'il s'agisse de celles de Microsoft ou celles d'Apple et Google.

Ceux qui me suivent régulièrement ont eu la chance de prendre ce projet à ses débuts, la version VNext est ensuite venue puis Hot Tuna proposant un framework mature et utilisable en production.

La suite logique du présent se retrouve dans les 12 vidéos présenant Hot Tuna. 8h en 12 volets pour presque tout savoir…

Rule them all ! L'avenir du futur – Le smartphone devient PC tout en un…

J'en avais parlé il y a quelques temps, l'avenir ce n'est pas de transformer des PC en grosses tablettes mais de transformer les smartphones en PC tout à fait honorables… Android et Ubuntu vont certainement le faire, les premières démos sont là…

Un smartphone qui remplace totalement le PC

L'idée est simple… Regardez votre smartphone. Il a probablement un processeur à 2 ou 4 cœurs, voire 8, de 1 à plusieurs giga de mémoire, un espace de stockage de plusieurs giga et une batterie.

Maintenant regardez votre PC : il a probablement un processeur à 2 ou 4 cœurs, de 1 à plusieurs giga de mémoire, un espace de stockage de plusieurs giga, et une batterie si c'est un portable.

Bref, en dehors du fait que votre smartphone tient dans une poche et qu'il soit capable en plus de téléphoner, d'accéder à internet sans une grosse "box", ce sont

des machines absolument identiques, ce sont des “pc”, des *personal computers*, les deux.

Tout ça pour Angry Bird et un texto à ses potes n’est-ce pas du gâchis ?

La puissance de votre smartphone ne sert finalement pas à grand chose. On se fait une vidéo, on envoie un sms à des potes, on vérifie son mail, on joue quelques minutes, et puis après ?

Au moment où il faudrait enfin **passer aux choses sérieuses**, utiliser un tableur, un traitement de texte, un logiciel un peu sophistiqué, dommage, on range son smartphone et on allume son PC.

Vous ne trouvez pas que c’est du gâchis ?

Chez Canonical (éditeur de Ubuntu) en tout cas on le pense. Mieux on a trouvé la réponse. Encore faudra-t-il qu’elle chemine et se fraye un passage dans le foisonnement des modèles, fabricants, et OS sur le marché. Car ce qui manque finalement c’est une prise de conscience des utilisateurs.

L’essai de crowdfunding du projet Edge de Ubuntu est un coup de buttoire pour aider le grand public à prendre conscience justement. Plus de 30 millions de dollars, il était évident que le projet ne serait pas bouclé. Mais il a soulevé plus de 10 millions de dollars et à attirer beaucoup de gens. C’est un signe, le premier. Il y aura d’autres essais à venir en ce sens car ce sens là, c’est tout simplement le sens de l’histoire…

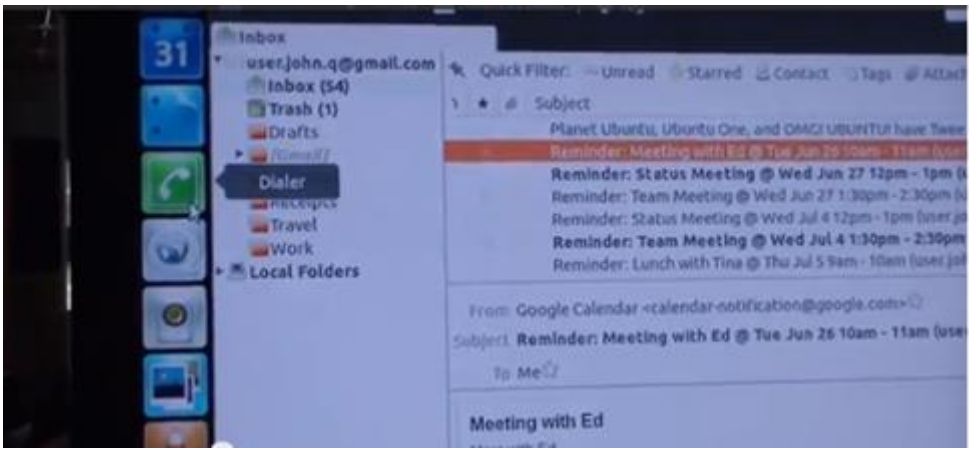
Ubuntu pour Android

La solution est pourtant simple : Ubuntu et Android sont des OS basés sur Linux (la 2.6 pour Android, une Debian pour Ubuntu), ils sont donc “mariables”. Canonical a ainsi conçu une couche logicielle qui transforme un smartphone Android en machine Linux sous Ubuntu sans supprimer la couche Android !

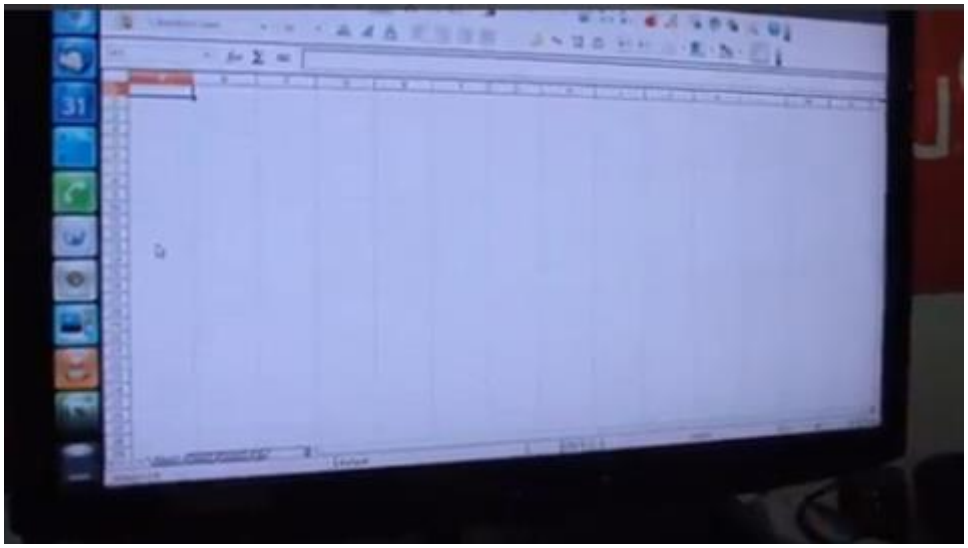
Le tout est complété d’un petit dock : on pose son smartphone dessus et l’image s’affiche sur l’écran qui y est connecté. De même le clavier et la souris reliés au dock sont opérationnels.

A partir de là tout Ubuntu est à vous ! Libre Office, voire compiler des programmes et développer, tout cela devient naturel comme disposer de la bibliothèque complète de logiciels qui tournent déjà sous Ubuntu…

Le téléphone sonne ? Pas de problème, décrochez ! Vous devez partir ? Enlevez le téléphone du dock et vous voilà avec votre smartphone Android tout à fait “normal” !



Gérer son mail, passer des coups de téléphone en mode Ubuntu, c’est possible… Comme travailler sur un tableur… Pour information en 2008 la Gendarmerie française a abandonné Windows au profit d’Ubuntu, soit environ 70.000 machines et une économie de licences de 50 millions d’euros… En 2010 les utilisateurs d’Ubuntu étaient estimés à 12 millions.



La fin des PC

Un doc, un écran de récup, un clavier et une souris. C'est tout ce dont on a besoin chez soi pour travailler. **L'ordinateur ? Vous l'avez déjà dans votre proche, c'est votre smartphone.**

Une seule machine qui concentre tout, une seule machine à acheter, plus de problèmes de "portabilité" de "compatibilité", par force, puisqu'il n'y a plus qu'un seul ordinateur ! C'est la fin du PC et des portables pour une grande partie du marché, au moins le grand public.

Dommage encore une fois que l'idée ne vienne pas de Microsoft qui décidemment depuis l'ère Sinofsky/Ballmer ne prend que des mauvaises décisions en retard sur l'avenir. J'en suis attristé, mais est-ce une raison pour nier les évidences et rejeter les bonnes idées ? Je ne le pense pas. Et puis peut-être que ces banderilles plantées par la concurrence finiront par convaincre MS de réagir vraiment…

De toute façon il y aura toujours des PC pour les "pros", ceux qui ont besoin de grosses machines de bureau, comme il y a 30 ans : les entreprises utilisaient des machines de pro (IBM 36, AS/400…) et les particuliers se contentaient d'avoir une machine à écrire, éventuellement électrique…

Demain ça sera plus cool bien entendu : un ordinateur de plus en plus puissant dans la poche de tout le monde qui se transforme en très bon PC à la maison, et de l'autre côté des machines de bureau plus complexes, plus chères aussi, pour les entreprises.

Cette époque là n'est pas de la SF : il suffit de brancher un écran et un clavier à votre smartphone, c'est déjà prévu et ça marche très bien… Reste juste à le doter d'un OS

plus complet, Ubuntu par exemple. Eux aussi sont prêts, reste à laisser le client venir. Quand il faudra dans un an ou deux rempalcer le PC portable qui sera tombé en rade, les gens rachèteront-ils une tablette même hybride comme Surface ou juste un dock et clavier bluetooth pour leur smartphone ?

Les vidéos de présentation

Le mieux c'est de voir soi-même, regardez et faites-vous votre idée. Si cela n'est pas la fin du PC ça y ressemble beaucoup en tout cas…

La vidéo de la démo est en portugais visiblement, mais c'est ce qu'on voit qui compte… Les lusitanophones seront aidés mais les autres comprendront aussi. Comme on pense que ce projet date déjà de l'été 2012 on comprend à la fois que convaincre et expliquer au grand public prendra du temps mais que tout est déjà prêt depuis longtemps…

http://www.youtube.com/watch?feature=player_embedded&v=F6_Oo-IKVUM

Une petite vidéo en US de la fin 2012 qui présente le concept, facilement compréhensible aussi :

http://www.youtube.com/watch?feature=player_embedded&v=iv1Z7bf4jXY

Conclusion

Le monde change, il change très vite. Les Smartphones sont en train de bouleverser les équilibres bien plus que les tablettes qui pour l'instant restent des jouets de gosse de riche qui ont du mal à trouver une vraie place sur le marché. C'est juste un plus grand écran pour skyper ou visionner un film. *Le Smartphone lui est déjà partout dans notre vie, il a déjà trouvé sa place, son utilité, il modifie les codes de la société, il change en profondeur notre rapport au temps et à l'espace. Et il est en train de devenir l'avenir de l'informatique.*

Mieux, le smartphone devient un PC.

Combien de temps prendra cette révolution ? Je ne suis pas voyant et je ne me risquerai pas à des pronostics de ce type. Mais à moins d'être aveugle, on sait au moins dans quelle direction les choses vont évoluer, tout simplement parce c'est "normal" : la miniaturisation nous a fait passer de l'AS/400 au PC, du PC au portable et du portable au Smartphone. Chacun à toujours remplacé le précédent puisqu'il apporte la même puissance dans moins d'espace.

Nul doute que c'est l'avenir que nous avons sous les yeux ici. Un avenir proche.

Et un avenir qui ne nous angoisse pas, car Android ou Ubuntu, grâce à Xamarin dont je vous parle en ce moment, tout cela est programmable en C# et en .NET !

Vous n'allez plus regarder votre smartphone de la même façon ce soir, j'en suis certain…

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. Le mois d'octobre et même novembre ont été consacrés à la remise en forme, à la relecture, parfois à des corrections ou ajouts importants comme le présent livre PDF sur le cross-plateforme.

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.

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 de *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 !