Tome 11

Microsoft Xamarin.Forms

Olivier Dahan

Collection ALL DOT BLOG

e-naxos

Première Edition

www.e-naxos.com

Formation – Audit – Conseil – Développement
XAML (WPF, UWP, Windows Phone), C#
Cross-plateforme Xamarin / Android / iOS
UX & Sound Design

ALLDOT.BLOG

EDITION 2016

Tome 11

1ère édition

Microsoft Xamarin.Forms

Programation Cross-Plateforme en C# et XAML

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

Olivier Dahan
odahan@gmail.com

Introduction

DOT.BLOG est un blog technique dont le format ne rend pas forcément justice au contenu. A l'origine les Blogs ont été conçus dant un esprit journalistique au sens journalier, chacun y allant de sa petite histoire, de son analyse de l'actualité, voire de sa recette de gâteau. Très vite les leaders techniques se sont appropriés ce format de diffusion bien pratique.

Hélas le format Blog, s'il est parfaitement adapté aux discussions sur l'actualité, aux banalités instantanées qui aujourd'hui d'ailleurs ont trouvées mieux avec Twitter ou Snapchat, souffre d'un gros problème : l'empilement au fur et à mesure de l'information qui la rend introuvable et difficilement consultable. Cela n'est tout simplement pas adapté à de l'information technique dont la durée de vie et l'intérêt dépassent de loin l'annonce de la dernière sex-tape de starlette en quête de reconnaissance.

Le bloggeur technique n'a pas d'autres choix que de se plier à cette instantanéité destructrice, à moins qu'il ne migre sur Twitter ou ne préfère diffuser sur Tumblr la photo de sa dernière pizza engloutie durant sa pose déjeuner… Ce qui l'oblige donc à ne plus être un bloggeur technique !

Bref, le blog c'est bien mais le Web attend une invention adatpée aux blogs techniques (les Wiki s'adaptant aux documentations mais pas aux articles).

Que faire en attendant cette révolution ?

J'ai décidé il y a quatre ans de créer la collection « ALL DOT.BLOG ».

Cette collection de livres PDF gratuits a été constituée au départ de tous les articles de DOT.BLOG repris, revus, corrigés, mis à jour et agencés dans un ordre logique correspondant mieux à celui d'un livre. Un vrai livre. Avec un sommaire, des liens cliquables et contrôlés. En PDF, donc où on peut chercher à volonté un terme et les endroits où il est utilisé. Un véritable objet de savoir véhiculant une information vivante car accessible, transportable et partageable.

Et Gratuit. Dot.Blog n'affiche aucune publicité et les articles et livres publiés sont accessibles à tous de la même façon. J'aime cette idée qu'on peut tous donner un peu de notre temps aux autres.
Certes je vends mon travail car ma générosité n'est pas si fréquente dans notre société. Il n'existe hélas pas d'épicerie, de fournisseur d'électricité ou de maisons en « freeware » … Mais admettez que 11 livres et près de 900 articles gratuits est une publicité autrement plus sympathique que du spam ! En retour pensez à moi pour vos développements ou vos formations !

Vous noterez que les billets n'ont pas été totalement réécrits, 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 à fait l'objet d'une note, d'un aparté ou autre ajout. *Il y a au final une grande différence entre le contenu du livre et les articles originaux*.

C'est donc bien plus qu'un travail de collection – déjà long – des billets qui vous est proposé ici, c'est une relecture totale, une révision et une correction techniquement à jour au mois de Mars 2016 pour cette première édition du Tome 11. Un vrai livre. Gratuit.

Astuce : tous les liens Web de ce PDF sont fonctionnels, n'hésitez pas à les utiliser !

Présentation de cette première édition 2016

Le développement cross-plateforme était une vieille chimère… Toutefois, à force de recherches, d'essais, et grâce à l'évolution permanente du tooling, des choses impossibles hier sont devenues presque banales aujourd'hui.

Dans l'édition 2014 du Tome 5 sur le développement cross-plateforme j'étais heureux de vous présenter le fruit de ce travail de recherche qui m'avait permi de vous proposer une stratégie de développement nouvelle basée sur une librairie encore inconnue, MvvmCross, en s'appuyant sur MonoDroid et MonoTouch, ancêtres de Xamarin.

En décembre 2013 lorsque j'ai bouclé l'édition 2014 jamais je n'aurais pensé qu'on pouvait aller encore plus loin aussi vite. Et tout a basculé avec les Xamarin.Forms. Enfin l'UI *aussi* devenait portable !

Imaginez un peu, C# portable pour tous les OS avec un sous-ensemble XAML portable. Incroyable mais Xamarin l'a fait. Et ce n'est que le début de l'aventure puisqu'aujourd'hui Xamarin est devenu un produit Microsoft !

C'est pourquoi j'ai décidé de créer un Tome dédié aux Xamarin Forms. Un ouvrage séparé des problématiques cross-plateformes abordées dans le Tome 5 dont les principes restent valables mais qui ne portaient pas uniquement sur Xamarin.Forms. Beaucoup de choses ont aussi changé et certaines stratégies n'ont plus de sens ou d'intérêt. *Aujourd'hui ont peut faire du cross-plateforme en restant 100% Microsoft qui va jusquà fournir un émulateur Android compatible avec Hyper-V !* Les XF méritent d'être ainsi traitées à part. Largement.

Tous les articles publiés ici le sont 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 sur Dot.Blog. Certaines modifications parfois importantes du texte font que cet ouvrage est bien un livre à part entière.

L'arrivée des Xamarin.Forms dans le giron de Microsoft rend d'une certaine façon caduque les autres approches. Nous disposons aujourd'hui d'une façon simple et directe d'aborder le cross-plateforme en C# supportée et maintenue par un éditeur à l'envergure mondiale.

Xamarin.Forms Ce qu'il faut en savoir

Xamarin 3.0 / Xamarin.Forms : entre évolution et révolution autour de l'UI

Le 28 mai 2015 Xamarin a mis sur le marché une mise à jour essentielle, la version 3.0. Xamarin.Forms est l'un des joyaux de cette version permettant d'unifier le développement des UI ! Tout commence là. Et se poursuit joyeusement avec la V4 puis le rachat de Xamarin par Microsoft mais commeçons par le début !

Xamarin 3.0 entre évolution et révolution

Il s'agit bien d'une version majeure, d'un saut important dans l'histoire de ce fantastique produit qu'est Xamarin et de sa société créatrice éponyme.

Bien loin des mises à jour plus ou moins factices des Apps mobiles qui utilisent cette ruse pour se rappeler à vous et attirer votre attention dans l'espoir de vous gaver de pub, je vous parle ici d'un colossal environnement de développement qui évolue à pas de géant.

Colossal ? Pas d'exagération ici. Xamarin c'est un ensemble fait de compilateurs C# pour iOS et Android, d'un IDE de type Visual Studio – Xamarin Studio – en plus de son intégration dans Visual Studio, d'un framework .NET portable, de tonnes de librairies de code, des éditeurs visuels spécialisés pour iOS et Android… etc, etc.

C'est incroyable tout ce qu'un tel produit contient et la masse de travail que cela représente. Miguel de Icaza, à l'origine de ce produit est, il faut bien le dire, une vraie "pointure" et même s'il n'est pas seul pour tout faire, c'est un sacré personnage.

Né en 72 à Mexico, 44 ans au compteur, il a été le meneur du projet GNOME bien connu des linuxiens et même au-delà tellement cette gestion de bureau a bouleversé la donne. Il a reçu le prix du logiciel libre pour ce travail exceptionnel. Il a créé la société Ximian spécialisée autour de GNOME, société rachetée par Novell où il devint vice-président chargé du développement…

Mais dans le monde PC on se rappellera certainement plus de lui pour avoir lancé le fabuleux MONO, cette copie conforme de .NET en Open Source fonctionnant sous Linux. En créant Xamarin il a repris le flambeau de MONO délaissé par Novell. Mais certainement encore plus fort, il a conçu MonoDroid et MonoTouch des

environnements .NET/C# pour iOS et Android aujourd'hui appelés Xamarin.Android et Xamarin.iOS.

Avec Xamarin 3.0, il pousse le raisonnement encore plus loin en offrant la portabilité totale même de l'UI !

Cette mise à jour comporte d'autres avancées qui habituellement auraient été présentées avec beaucoup d'emphase tellement elles sont importantes, mais il est vrai que l'unification des UI est en soi tellement "énorme" que cela masquera certainement tout le reste.

Designer iOS pour Visual Studio

Concevoir des applications portables c'est encore bien souvent avoir à supporter iOS. Même si ses parts de marchés ne sont plus que l'ombre de ce qu'elles furent, elles se maintiennent et on sait que le monde Apple transpire l'argent et le profit. Difficile d'ignorer iOS dans un tel contexte, en tout cas lorsqu'on cible les concierges et les bobos (même niveau intellectuel), c'est-à-dire le grand public.

Xamarin 3.0 offre un designer visuel iOS pour Visual Studio totalement bluffant. On aimerait presque avoir tout de suite un truc à développer pour iPhone histoire de jouer avec !

Mais le plus bluffant n'est pas forcément là…

Xamarin.Forms

Xamarin.Forms est une nouvelle librairie de code qui permet de construire des UI _natives_ pour iOS, Android et enfin Windows Phone depuis un code C# simple et partagé, donc unique.

L'astuce consiste bien entendu à déclarer l'UI en C# en utilisant une API riche pour ce début de plus de 40 contrôles visuels cross-plateforme qui sont traduit en leur équivalent _natif_ à la compilation sous chacun des OS supportés.

Cliquez sur l'image ci-contre pour accéder à une version haute résolution plus lisible.

Ce qui est fabuleux c'est que Xamarin.Forms s'utilise sur la base d'une page, donc il est possible de choisir, pour chaque page d'une App, d'utiliser Xamarin.Forms ou Xamarin.Android/iOS pour la mise en page. Par exemple un simple login sera développé en Xamarin.Forms pour aller plus vite, alors qu'une page plus sophistiquée nécessitant du visuel particulier pourra être créée en Xamarin.Android/iOS.

Mais cela va plus loin. Il est tout à fait possible d'insérer une vue personnalisée à l'intérieur d'une page en Xamarin.Forms… Inutile de tout coder en natif juste pour un effet ou un contrôle qui nécessite d'accéder à l'UI de façon native. Il suffit de coder ce qui est spécifique dans la page et l'intégrer dans une page Xamarin.Forms qui elle n'est à écrire qu'une fois pour les 3 plateformes cibles !

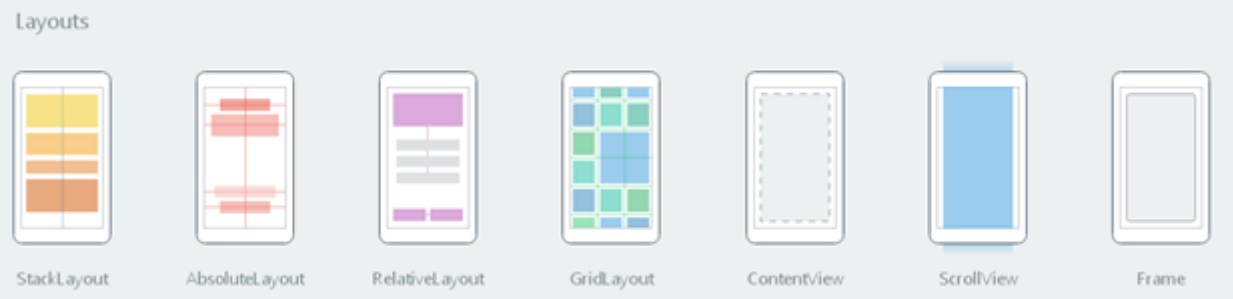
Encore mieux : au sein d'une page Xamarin.Forms il peut être nécessaire de faire appel à une fonction purement native, imagions l'accéléromètre. Pas de souci puisqu'une API de services est aussi fournie pour unifier les appels typiques de ce genre. Encore une fois : une seul code C#, trois plateformes gérées automatiquement le tout en natif ce qui est essentiel !

Les pages

Afin d'aider le développeur qui doit coder une page sans retour visuel direct, Xamarin.Forms propose des modèles faciles à imaginer pour y placer le visuel. Par exemple les pages peuvent être de simples conteneurs où on place ce qu'on veut mais aussi des vues maître/détail, des pages à onglet, etc…

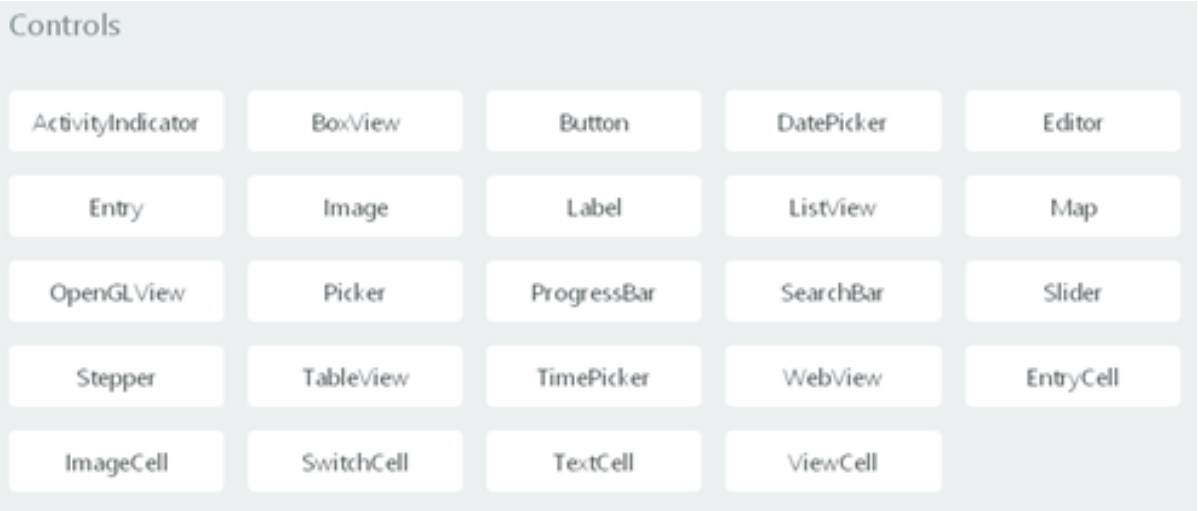
Les layouts

A un niveau plus fin on trouve aussi des conteneurs spéciaux qui aident à concevoir une UI propre et organisée en quelques instructions. De la pile de contrôle (le StackPanel en XAML) à la grille personnalisable (de type Grid XAML), le développeur dispose d'un choix suffisamment large pour couvrir immédiatement les principaux besoins.



Les contrôles

Quant aux contrôles dans cette première version de Xamarin.Forms on trouve déjà le nécessaire pour concevoir l'UI d'une majorité d'apps. L'indicateur d'activité, le bouton, le sélecteur de date, l'image, le label, … ils sont tous là, reconvertit en contrôles natifs à la compilation.



Extensibilité

Quelle que soit la richesse des Xamarin.Forms il se trouvera toujours un projet pour lequel ce qui est dans la boîte ne sera pas totalement suffisant.

Peu importe… Les Xamarin.Forms sont extensibles. Le développeur peut définir ses propres contrôles, ses propres layouts, ses propres types de cellules de grille. Ces contrôles natifs peuvent être exposés afin d'être utilisables directement dans les pages Xamarin.Forms. Il est aussi possible de sous-classer les contrôles fournis pour en créer de nouveau.

MVVM

La conception des Xamarin.Forms n'oblige pas à quitter les paradigmes essentiels à la bonne conception d'une application moderne. On retrouve donc de façon naturelle la prise en charge de l'architecture MVVM.

Binding

Comment parler de MVVM s'il n'y a pas de binding… C'est pourtant le cas des environnements primitifs comme iOS ou Android, loin de la richesse et de la sophistication de XAML. Là aussi les avancées sont considérables puisque Xamarin.Forms introduit un binding two-way pour la synchronisation entre UI et Model !

Fluidité

Le festival n'est pas terminé… Xamarin.Forms ajoute un système d'injection de dépendance avec l'assurance d'un démarrage inférieur à 10 ms. Que demander de plus ?

Messages

MVVM avec le binding c'est bien, mais avec une messagerie c'est mieux… Xamarin.Forms propose justement une telle messagerie pour assurer un couplage faible entre les différents composants des applications.

Un visuel animé

Une UI moderne utilise souvent – quand cela est pertinent et améliore l'UX – des animations de type rotation, translation, changement de taille… Les Xamarin.Forms proposent toute une panoplie d'animations de ce type pouvant être composées pour créer des effets sophistiqués.

Il existe même une API bas niveau pour appeler le mécanisme d'animation et concevoir ses propres effets personnalisés.

Bien entendu toutes les opérations sont systématiquement traduites et déléguées aux API natives de chaque plateforme.

Qu'attendre de plus ? … de pouvoir attendre qu'une animation se termine avant de passer à autre chose par exemple. Et oui, c'est possible, et de la façon la plus élégante qu'il soit puisqu'en utilisant le modèle async/await qui permet d'écrire du code d'aspect séquentiel tout en respectant la fluidité.

Peut-on faire plus fort ?

Tout en XAML !

Oui, ils l'ont fait… Même si ce n'est que le début et que les designers visuels précédents ne sont pas compatibles avec ce XAML là, il est possible de décrire une page entière à l'aide d'un sous-ensemble de XAML : vues, layouts et même bindings peuvent être décrits dans ce mode.

Bientôt un XAML portable comme .NET ? Ce n'est certainement pas le but, mais un sous-ensemble avec designer visuel pour obtenir une conception d'UI unique mais portable, certainement à termes.

Conclusion

Xamarin 3.0 c'est bien entendu encore plein d'autres choses. Mais il faut que je calme mon enthousiasme au risque de donner l'impression de faire le VRP 

Construire des applications avec du code portable c'était déjà extraordinaire. On savait étendre tout cela à plusieurs plateformes en utilisant des PCL et un framework comme MvvmCross. Mais on en restait au niveau des Models et des ViewModels. L'UI restait le dernier bastion à prendre.

Il est pris.

Mieux, certainement grâce au rapprochement entre Microsoft et Xamarin, nous avons maintenant une plateforme de développement unifiée tant pour le code que l'UI capable de supporter immédiatement les trois cibles mobiles les plus importantes du marché : iOS, Android et enfin Windows Phone.

Sans bricolage. Pour un résultat purement natif.

Si ce n'est pas une révolution, cela y ressemble beaucoup tout de même…

MVVM et les XF

Mvvm Light supporte les Xamarin.Forms

Le framework MVVM Light dont j'ai parlé de multiple fois pour ses qualités et sa simplicité a enfin sauté le pas il y a quelques temps : il est devenu cross-plateforme en supportant Xamarin !

MVVM Light un habitué des colonnes de Dot.Blog

MVVM Light de Laurent Bugnion est un produit dont je vous ai parlé cent fois et sur lequel j'écris des tonnes d'articles et un même un mini livre…

Simple et puissant mais limité à Windows…

J'ai toujours aimé ce framework car il est simple à prendre en main et en fait assez pour assurer le respect de MVVM. Sa simplicité est garante de son *apprentissabilité* ce qui me parait essentiel lorsque je le conseille à des clients dont les équipes sont hétérogènes ou fluctuantes. D'autres approches m'ont semblé dans le temps plus intéressantes comme Jounce qui ne fonctionnait qu'avec Silverlight et utilisait MEF. Mais Jounce n'a jamais été porté sur WPF ou Windows Phone.

Prism est bien entendu un fabuleux framework plutôt MVC et le mot même "MVVM" n'est arrivé que fort tard dans les dernières versions et même si son esprit à toujours été proche de MVVM on voyait bien que sa complexité n'avait pas respecté l'esprit de MVVM. Heureusement les dernières versions pour WPF ainsi que la version spéciale pour WinRT se sont attachées à respecter plus directement MVVM. Prism est un donc bon framework aujourd'hui mais il n'est pas portable. Pour WPF MVVM Light est parfait et qui se soucie désormais de WinRT…

La complexité non maîtrisée ce fut le cas de Caliburn aussi. Tellement imbuvable que son créateur a même dû créer Caliburn.Micro, qui finalement est présenté comme le remplaçant de la version non micro… Dans cette dernière forme Caliburn, micro donc, est un bon framework bourré de choses intelligentes. Mais cela reste un esprit bien particulier. Certains aimeront et ils auront finalement raison. Il faut toujours utiliser ce qu'on aime, on travaille mieux comme ça.

Donc pour une majorité de mes projets sous WPF j'ai souvent opté pour MVVM Light.

Depuis cinq ans environs quelque chose me gênait de plus en plus avec ce framework : son enfermement. Bien sur il y a eu l'intégration du support de WinRT, mais finalement ce n'était pas si compliqué à faire (et plus marketing qu'utile en pratique). Il y eut des améliorations comme l'utilisation d'un moteur d'injection de dépendance. Tout cela faisait partie des évolutions qu'on pouvait attendre d'une si bonne librairie.

Mais elle restait confinée à Windows alors que le monde s'ouvrait de plus en plus à d'autres OS obligeant à travailler en cross-plateforme…

Le cross-plateforme… 12 vidéos gratuites, un livre PDF gratuit plus celui-ci pour les XF, des tas de billets… Vous vous imaginez bien qu'on ne fait pas tout ça juste pour remplir du texte sur un blog, il y a un sens, une vision de l'avenir, une vraie motivation derrière, et cette motivation, en dehors de la passion, *c'est la réalité du marché* !

Poussé par cette dernière j'ai fini par rencontrer un produit exceptionnel, MvvmCross dont j'ai aussi parlé énormément avec beaucoup de détail puisque c'est lui qui est utilisé dans les 12 vidéos évoquées plus haut.

Ca c'était avant… Maintenant c'est aussi portable !

De la concurrence née l'émulation et la stimulation. Et MvvmCross ne pouvait rester seul au royaume du code portable.

Poussé par la même réalité et pas ses utilisateurs Laurent a fini par réagir, et c'est ainsi que la version 4.4 de MVVM Light nous a offert enfin le support de Xamarin.

Depuis cette version qui s'est améliorée (V5 actuellement) , MVVM Light offre le support de Xamarin, c'est à dire de iOS et de Android et aussi de UWP.

Dans quel esprit ?

Bon, je vais être franc, l'approche de MvvmCross est plus subtile, plus complète aussi (avec la notion de plugins par exemple). Pour l'instant MVVM Light se propose de vous laisser écrire vos ViewModel comme avant et d'utiliser le code des Activity Android pour y ajouter les bindings (par code avec AddBinding()) et les commandes (avec AddCommand()).

C'est un peu comme utiliser le code-behind sous XAML, c'est dommage de ne pas être allé aussi loin que MvvmCross en proposant un binding dans le XML Android.

Mais au fil des versions MVVM Light arrive à devenir un outil totalement portable adapté à la stratégie PCL qui est la plus utilisée pour utiliser Xamarin.Forms.

Dans les dernières versions des XF le binding du pseudo XAML utilisé rend inutile les astuces des toolkits pour combler ce manque. De fait Mvvm Light reprend sa place pour gérer MVVM et l'environnement de développement la sienne : fournir une base complète et cohérente pour le code et l'IHM. Dès lors Mvvm Light redevient tout aussi attractif qu'il peut l'être sous WPF.

Que faut-il en penser ?

C'est du MVVM Light accommodé pour supporter Xamarin.Forms. C'est très bien, c'est intéressant et pour des projets qui existent déjà et qui font utilisation de MVVM Light c'est plutôt une bonne nouvelle : pas besoin de remettre en question l'existant pour ajouter un projet Android à la solution…

Pour les nouveaux projets on s'aperçoit que les Xamarin.Forms sont si bien faites qu'on peut éventuellement se passer d'un toolkit MVVM pour appliquer le pattern.

Mais l'appui de Mvvm Light sera malgré tout un bon coup de pouce pour enrichir les commandes (RelayCommand), gérer les services (conteneur IoC), disposer d'une messagerie (Messeneger), etc…

vs MvvmCross ?

Maintenant que les peintures ont séché depuis la première release cross-plateforme MVVM Light est un excellent framework MVVM pour WinRT, Windows Phone, WPF. Il supporte désomaris Xamarin.Forms et UWP de façon assez efficace ce qui est un avantage certain. Et je suis convaincu que Laurent saura faire évoluer la bibliothèque, le plus dur était de franchir le pas du cross-plateforme.

Et MvvmCross alors ?

Ce toolkit garde son intérêt lorsqu'on doit encore écrire des UI totalement différentes autour d'un même noyau de code. Certains développements réclament en effet une personnalisation telle que viser l'universalité de l'IHM est illusoire. Les XF sont conçue pour harmoniser les IHM, MvvmCross a été conçu pour des IHM différenciées et il s'adapte parfaitement à cette situation. Les dernières versions de Xamarin.Forms viennent toutefois grignoter ce dernier avantage et je suppose, malgré tout le bien que j'en pense, que ce toolkit terminera aux oubliettes car rien n'interdit dans les XF désormais de mélanger forms XF et forms personnalisées dans une plateforme donnée.

Vs Xamarin.Forms ?

La encore le match était loin d'être gagné il y a encore peu pour MVVM Light. Les Xamarin.Forms offrent une approche plus semblable à celle de MvvmCross c'est-à-dire plus « enveloppante », plus complète. MVVM est applicable directement à peu de frais et même sans framework, et surtout, comme MvvmCross les Xamarin.Forms offrent une solution de binding avec en plus le support de XAML… Mvvm Light est très loin de tout cela et ne s'intéresse pas du tout à l'UI. Avec les avancées de chaque produiit le mariage MvvmLight / XF est redevenu plus clair, chacun ayant renforcé ses compétences, il y a une légère concurrence sur certains aspects de MVVM mais Mvvm Light va plus loin sur ce point. Je pense donc que c'est le couple gagnant pour 2016.

Conclusion

Mvvm Light est utilisable directement dans vos applications via les packages Nuget, vous pouvez aussi obtenir le source sur CodePlex.

Le plus intéressant dans tout cela c'est que bien entendu vous avez là la confirmation de ce que je me tue à vous dire depuis au moins trois ans : l'avenir est au cross-plateforme et il sera impossible d'y échapper, même pour les frameworks.

Le rapprochement Microsoft / Xamarin puis le rachat du second par le premier, l'existence de MvvmCross, la sortie des Xamarin.Forms, puis de MVVM Light portable, tout cela doit vous inciter à comprendre que l'avenir est forcément éclaté en plusieurs OS. Faire le dos rond pour que "ça passe" a eu son temps. Aujourd'hui le monde se sépare au minimum en trois OS essentiels : Windows classic (WPF) d'un côté pour les entreprises et les grosses applications (ni Google ni Apple ne peuvent avancer le moindre pion dans cet univers malgré l'accord Apple/IBM qui n'a rien donné deux ans après) et Android pour toutes les machines mobiles (Apple n'y étant plus que quantité négligeable surtout pour les d'applications business). IoS se meurt puisqu'Apple a vendu son fond de commerce, son unicité « élitiste » pour faire de la vente de masse, mais cela n'a qu'un temps. La griffe Apple ne veut pas mieux que celle de la FNAC ou Darty aujourd'hui. Les jeunes clients n'ont pas connu l'époque glorieuse, le mythe est mort, c'est un truc de vieux. Désormais ils se battent sur le même terrain que leurs propres fournisseurs comme Samsung et d'autres ! Autant dire que cette guerre est perdue d'avance. En choisissant un fabriquant chinois inconnu ils le rendent plus fort et plus célèbre. Et entre celui qui fabrique et celui qui ne fait que revendre avec son logo il arrive fatalement un moment où le premier riche de son expérience et de son savoir faire n'a plus qu'à se faire un nom et cela arrive toujours (Samsung, Huawei, Wiko…).

Ceux qui me suivent et qui écoutent mes conseils ne seront pas surpris par ce mouvement inexorable que j'annonce et explique depuis un long moment. Ceux qui prennent un peu le train en marche ou qui restaient sceptiques doivent aujourd'hui se rendre à l'évidence *: il est grand temps de se mettre à Xamarin et à Android en complément de ce qu'on sait déjà du monde Windows.*

Non pas pour supplanter ce dernier, non pas pour défier Microsoft, mais simplement pour embrasser la réalité de notre monde qui est, qu'on l'accepte ou non, cross-plateforme Windows / Android et ce pour des années et des années à venir…

Saluons le travail des loups solitaires comme Stuart Lodge et Laurent Bugnion, saluons aussi la grande intelligence d'un de Icaza qui offre désormais la portabilité iOS/Android/Windows UWP dans Xamarin. Tous ces gens ont le sens de la réalité, le feeling de ce que sera demain, c'est grâce à leur travail que nous pouvons avancer. Ce sont les premiers de cordée… Sans eux l'escalade ne se ferait pas. Tout simplement.

MVVM Light 5 est matûre et forme un couple parfait avec les XF ! Tout est prêt pour le grand changement.

MVVM Light 5 : support de Xamarin

Mvvm Light a toujours été ma préférence pour sa simplicité et son efficacité. La V5 apporte son lot de nouveautés parmi lesquelles on trouve enfin le support pour Xamarin. J'avais dit que j'y reviendrai, en voici la preuve !

Une librairie unifiée cross-plateforme

MVVM Light est désormais l'une des deux seules librairies MVVM véritablement portables sous tous les environnements C# :

- WPF 3.5, 4.0, 4.5 et 4.5.1
- Silverlight 4 et 5
- Windows Phone 7.1, 8, 8.1 Silverlight et 8.1 WinRT
- Windows Store 8 et 8.1
- Xamarin.Android
- Xamarin.iOS
- Xamarin.Forms
- Windows UWP

Il est donc possible d'utiliser la même stratégie, le même code MVVM d'un smartphone Samsung à une tablette Apple en passant Surface, Windows Phone, le Web en vectoriel avec Silverlight et sur PC aussi bien full .NET WPF qu'en mode tablette ou smartphone Surface avec UWP.

Mvvm Light 5 vs MvvmCross

Les deux approches sont radicalement différentes et intrinsèquement MvvmCross reste une de mes options préférées pour le cross-plateforme car cette librairie apporte plus que MVVM comme une syntaxe de Data Binding même sous iOS ou Android.

Toutefois avec l'arrivée des Xamarin.Forms et de composants tiers qui apparaissent pour cet environnement, travailler avec MVVM Light sur les OS mobiles prend plus de sens puisque Xamarin rend possible le binding et qu'on peut se passer de l'aide d'une librairie pour le faire.

Sous Xamarin.Forms mon choix est donc Mvvm Light.

Mvvm Light est un framework léger, opérationnel, qui a fait ses preuves. Il reste mon choix par défaut en toute circonstance. La V5 renforce encore plus cette prédilection. Toutefois selon les projets, MvvmCross peut s'avérer plus universel, plus apte à aplanir les difficultés d'un grand écart de type iOS/WPF/Android.

Quant à Prism, la version WPF a toujours été lourde malgré ses innombrables qualités et en revanche les versions WinRT puis UWP sont devenues trop limitées car trop orientée site marchand ce qui est un comble pour des applications desktop ou tablettes.

D'autres frameworks n'ont existé que pour certaines cibles, Jounce par exemple ne tourne que sur Silverlight avec MEF. C'est un framework dont j'ai dit tout le bien que j'en pensais à l'époque, mais il n'est plus d'actualité. Prism pour WinRT est de la même espèce, bien ficelé, bien pensé, mais trop limité à une seule cible déjà morte …

Reste donc en course MvvmCross et Mvvm Light V5. Tout dépend du projet, c'est une décision au cas par cas donc, l'un n'efface pas l'autre totalement en tout cas. Mais pour du développement spécifiquement sous Xamarin.Forms, objet de ce livre, Mvvm Light aura ma préférence.

Les nouveautés de la V5

Mvvm Light se transforme petit à petit et de nouvelles fonctionnalités apparaissent. Le support de Xamarin n'est pas l'un des moindres on s'en doute, mais il ne faudrait pas que l'arbre cache la forêt. Par exemple l'ajout d'un service de navigation est un must. Prendre en charge la navigation est un besoin vital dans une application, besoin que Mvvm Light ignorait depuis toujours. De même que le service de dialogue dont seul un début de commencement était géré (via le messenger avec une classe de message spécialisée).

Il s'agit là d'innovations importantes pour ce framework qui était malgré tout très en retrait sur des fonctions aussi vitales.

Dans la foulée la V5 supporte les PCL (Portable Class Library) ce qui permet de mieux l'intégrer dans une logique cross-plateforme avec partage du code au niveau binaire. C'est le mode que je préfère pour travailler sous Xamarin.Forms.

Derrière ces nouveautés se cachent d'autres détails. Par exemple les services de dialogue et de navigation sous-tendent la présence d'une Interface qui propose des implémentations pour toutes les plateformes (ou presque puisqu'il n'y a pas

d'implémentation WPF fournie pour l'instant mais rien n'interdit de l'implémenter si besoin est).

Le support des Xamarin.Forms fonctionne "out of the box" et c'est une bonne nouvelle aussi car les Xamarin.Forms que je vous ai déjà présentées sont une avancée de taille dont je reparlerai longuement ici.

Conclusion

Avec de tels changements il peut y avoir deux ou trois choses à modifier dans un code existant qu'on voudrait mettre à jour en V5, sinon il n'y a aucun problème.

S'agissant d'un travail en perpétuel chantier d'autres améliorations viendront certainement, mais les principales listées ici permettent d'ores et déjà à Mvvm Light de rester le framework MVVM de référence pour tous les développements de taille conventionnelle et d'être une base parfaitement stable pour des applications de grandes tailles.

La grande unification présentée par Microsoft n'est pas forcément celle qu'on croit. La véritable grande unification n'est pas de partager du code entre Windows Phone et Surface mais de pouvoir marier de l'iOS, de l'Android, de l'UWP avec un même code. Et cette grande unification nous vient de Xamarin et non de Microsoft et de gens comme Laurent qui avec la V5 de Mvvm Light nous offre les outils pour faire face sereinement à un marché décousu et segmenté en plusieurs OS tous incontournables. Certes Microsoft a eu le nez creux d'acquérir Xamarin et nous attendons tous maintenant les effets bénéfiques de ce rachat (meilleure cohérence, prix de XF ? …).

Mais sans Xamarin, sans MvvmCross ou Mvvm Light, nous serions coincés à choisir notre camp et peut-être à tout perdre pour avoir fait le mauvais choix. Avec eux, notre bagage C#/Xaml nous permet d'affronter ce marché distendu et multi-plateforme. Nous avons de la chance d'avoir choisi le bon langage et la bonne plateforme avec .NET … Il sera dur de nous convaincre que seul WinRT est l'avenir comme je le disais dans le Tome 5 il y a 3 ans… Tellement vrai que WinRT est passé à la trappe ! Tellement vrai que Microsoft est passé à UWP et a racheté Xamarin… Car le présent est bien plus complexe que la bonne vieille hégémonie de MS. On a raillé MS pour cette présence sans partage, et comme je l'avais prédit, nous regretterons la stabilité que nous offrait cette hégémonie… Hélas pour notre tranquillité et pour l'éditeur de Redmond malgré la grande qualité des solutions qu'il propose aujourd'hui nous ne pouvons pas tout sacrifier pour le suivre de façon exclusive car nous devons gérer des technologies de plusieurs types en même temps. J'ai connu un temps lointain où un client choisissait un logiciel et

achetait l'ordinateur qui le faisait tourner, aujourd'hui le client choisit la plateforme et veut que le logiciel fonctionne dessus, peu importe ce que cela nous coute…

Microsoft en touchant de grosses royalties sur les ventes d'appareil Android et en rachentant Xamarin place ses œufs dans tous les paniers à la fois. Que WinRT soit parti à la corbeille et que UWP soit un succès ou non ou que les 85% d'Android finissent par étouffer les Windows Phone et les Surface, Microsoft sera gagnant à tous les coups ! Nous serions bien idiots de ne pas suivre l'exemple qui nous vient de si haut en nous restreignant aux plateformes MS… Surtout que nous pouvons désormais échapper à ce carcan tout en utilisant exclusivelement des produits Microsoft !

Dans notre quête du bonheur Xamarin et la V5 de Mvvm Light nous aident à trouver la petite lumière au bout du tunnel… En rachetant Xamarin Microsoft a donné raison à cette vision des choses que j'expliquais déjà mot pour mot il y a trois ans. Heureux sont les lecteurs de Dot.Blog qui écoutent mes conseils !

MvvmLight Android & Windows Phone/UWP

Windows Phone, Android, iOS, les Xamarin.Forms et Mvvm Light portable, en voici un joli cocktail ! Comment associer tout cela pour développer des apps portables avec un code et une UI 100% unique et partagé ? C'est ce que je vous propose de voir…

Un code, une UI, trois plateformes

Grâce aux Xamarin.Forms qui ajoutent une couche UI portable mais native à Xamarin il est possible d'écrire vraiment pour la première fois un code + son UI en mode totalement portable. Mieux ce code et cette UI sont parfaitement UNIQUES sans duplication, sans `#if` sans rien. ***Un code, une UI, trois plateformes.***

Le développement pour mobiles n'a jamais été aussi clair, simple et rapide. Mais je vous ai présenté ces nouveautés au fil de leurs arrivées. De même que MVVM Light qui est devenu portable et surtout qui aujourd'hui s'adapte parfaitement aux Xamarin.Forms.

Pourquoi cet article ? Parce que toutes ces nouveautés avaient besoin de murir et de se stabiliser pour véritablement pouvoir commencer à dégager des ***stratégies de développement utilisables en production***. Les Xamarin.Forms par exemple ont connu quelques breaking changes dans le courant de l'année dernière mais aussi des améliorations essentielles. Elles arrivent désormais dans une phase mature. Mvvm Light portable a aussi connu quelques ajustements entre sa toute première version et aujourd'hui.

Bref, tout ce petit monde augurait un avenir radieux mais encore fallait-il que tout cela se stabilise. Ce nouveau plateau est atteint et on peut désormais parler production au lieu de tests pour *early adopters*.

Ménage à trois

Il en aura fallu du temps pour qu'une plateforme de développement et des outils puissants soient disponibles pour véritablement permettre ce mariage à trois entre trois plateformes que tout oppose, que tout sépare, *by design* et surtout par volonté commerciale de dresser des murs infranchissables pour rendre captif les clients autant que les éditeurs et développeurs.

> ***Nota : Quand j'ai écrit ce chapitre la plateforme Microsoft visée était Windows Phone, l'exemple reste tout aussi parlant, mais si je devais le réécrire j'utiliserai bien entendu le mode UWP intégré désormais à Xamarin.Forms depuis la version 4. Du coup on pourrait aussi faire tourner la partie Microsoft sur le PC et viser ainsi une plateforme de plus ! Donc 3 ou 4 plateformes ? Android, iOS, cela fait deux. Windows Phone cela fait trois mais cet OS est voué à disparaitre, remplacé par UWP. On reste à trois ? C'est difficile à dire car UWP est lui-même cross-plateforme et cible les smartphones, les tablettes, les PC, les Xbox, etc. Mais je m'en tiendrais à 3 plateformes, UWP comptant pour une.***

Heureusement le monde Microsoft est certainement et contre toute attente le moins fermé de tous… Apple avec iOS reste tellement hermétique qu'il faut absolument un Mac même pour lancer une émulation d'iPhone… C'est aussi pour cela que je préfère me concentrer sur les 2 seules plateformes intéressantes du marché : Windows Phone (ou UWP plutôt) car c'est le meilleur OS techniquement parlant doté du meilleur tooling, et Android car c'est le numéro 1 qui contrôle presque tout le marché et qu'il semble difficile d'ignorer le vainqueur tout comme il a été difficile d'ignorer Windows dans les 20 dernières années.

Ouverture car seulement dans le monde Microsoft a pu naitre quelque chose comme Xamarin basé sur Mono : une plateforme .NET open source tournant sur PC, Mac et Linux. Et cela n'a été possible que parce que Microsoft a ouvert dès le départ le framework .NET.

Ouverture encore car Microsoft au lieu d'ignorer ou de combattre Xamarin en a fait un allié. Et c'est une excellente chose. Pour Windows Mobile en premier. Car si cela ne coute pas plus cher de développer une app qui tourne _aussi_ sur Windows

Phone, pourquoi s'en priver ? Au lieu de prendre un SDK Google et de ne développer que pour l'OS majoritaire (ce qui est logique pour un éditeur) on peut *avec un même code créer des apps qui marchent _aussi_ sur Windows Mobile / PC et iOS*. **Il faudrait être fou pour choisir une plateforme de développement restrictive lorsqu'on a la possibilité d'écrire 3 apps en même temps avec un seul code !** Et cela est bon pour Windows Mobile comme pour UWP. Car puisqu'il ne coute rien de supporter UWP, autant choisir de le faire plutôt que d'ignorer cette plateforme appréciée par tous ses utilisateurs, ce que montre son adoption.

Ouverture encore et encore puisque Microsoft a été jusqu'à racheter Xamarin et à l'offrir avec Visual Studio (un gros cadeau vu le prix des licences Xamarins jusqu'à ce jour). Mieux, tout cela est Open Source.

Microsoft, mis à mal par dix ans de bêtise d'une direction à côté de ses pompes a loupé le coche de la mobilité. C'est un constat plutôt négatif. Mais en même temps, si on regarde la société, ses évolutions, son ouverture d'esprit, ses produits comme Azure, sa stratégie cross-plateforme, je pense sincèrement que nous vivons en ce moment les plus grandes heures de Microsoft. Jamais cette scoéité n'a été aussi à la pointe et aussi présente partout qu'aujourd'hui. Paradoxal. Espérons qu'avec le temps le marché saura reconnaitre cette excellence.

Le mobile impose les mêmes bonnes pratiques

Développer une app pour mobile c'est avant tout développer une application, un logiciel, appelez cela comme vous le voulez mais il s'agit de la même chose. Et ce qui est bon pour un logiciel WPF en mode desktop sur PC s'applique à l'identique à une app mobile.

Notamment les bonnes pratiques de développement comme la séparation des tiers, l'écriture d'un code à faible couplage, etc, tout cela reste parfaitement valable !

Pour cela il faut une méthode. MVVM en est une. Mais MVVM ne s'applique à merveille qu'au couple C#/XAML qui n'est en réalité disponible seulement que sur Windows Phone (dans le monde des mobiles) …

C'est sans compter sur Xamarin.Forms qui amène XAML (un sous-ensemble) sur iOS et sur Android ! XAML et sa syntaxe, sa logique, et son databinding qui est à la base même de MVVM.

Armé des Xamarin.Forms ne reste plus qu'à ajouter un framework comme MVVM Light pour recréer un environnement de travail semblable à celui qu'on utilise pour les "grosses" applications sur PC.

C#, XAML, le framework .NET, Xamarin, Xamarin.Forms, Mvvm Light, un ensemble unique qui recréée les conditions idéales pour développer avec le même savoir (et le même savoir-faire) sur tous les OS mobiles qui comptent.

Reste à comprendre comment utiliser toutes ces briques en même temps dans un tout cohérant pour enfin développer librement des apps *natives* pour les 3 plateformes avec un seul code et une seule UI.

C'est exactement ce que je vais vous montrer maintenant. Alors let's go !

Xamarin Studio ou Visual Studio ?

Visual Studio est malgré l'EDI le plus sophistiqué et le plus complet auquel on peut adjoindre des outils comme ReSharper par exemple. C'est aussi le seul environnement permettant de travailler sur toutes les cibles à la fois. Lorsqu'on créée une nouvelle solution cross-plateforme Xamarin.Forms seul Visual Studio sait créer outre le projet de code partagé les projets pour chaque cible. Ne serait-ce que pour supporter Windows Phone (ou UWP) et Android il est donc préférable d'utiliser Visual Studio tout simplement parce qu'il sait le faire !

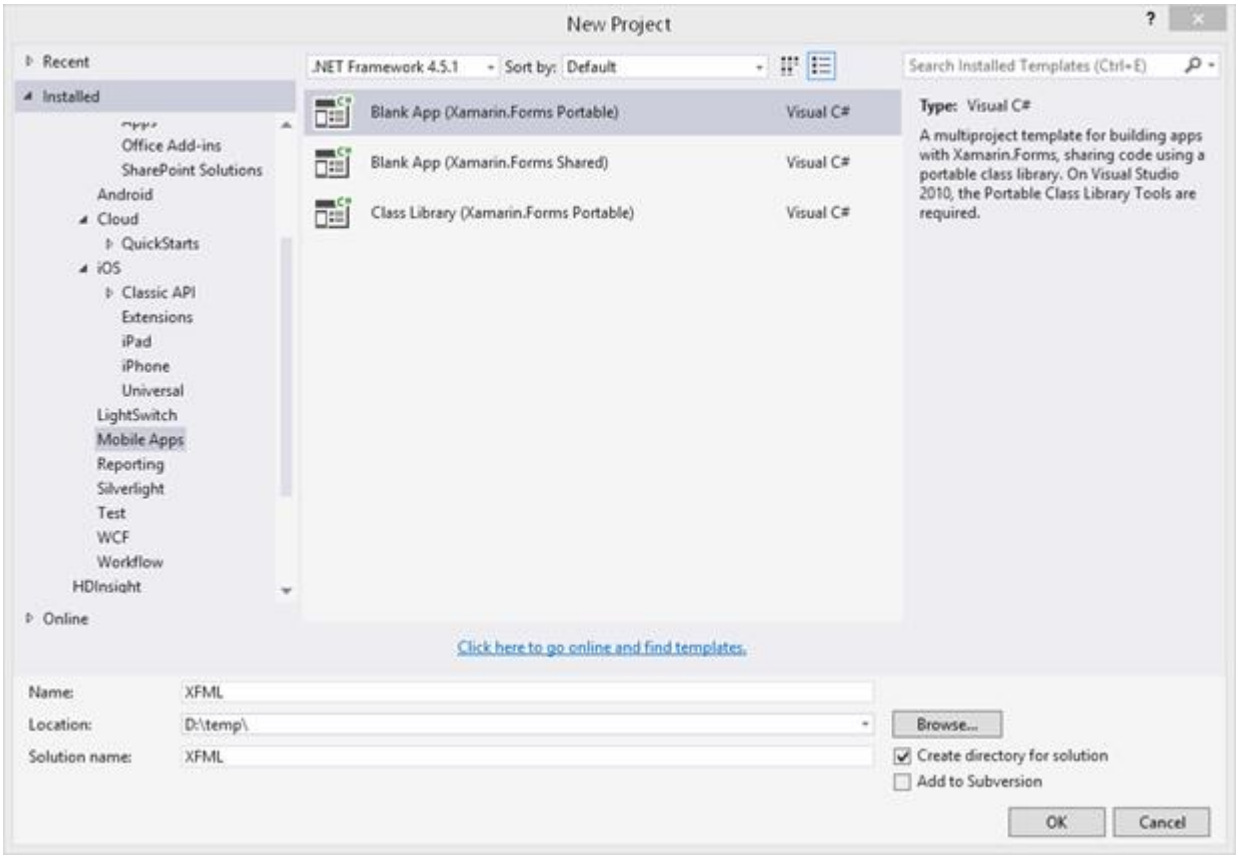
Xamarin Studio a beaucoup évolué et ressemble beaucoup à VS mais il se concentre sur ces cibles à lui : Android, Mac, Mono. Pas de projet Windows Phone notamment. Mais si on vise une app uniquement Android tout en bénéficiant de C#, XAML et du reste, alors Xamarin Studio est un très bon choix. Avec l'avantage d'être portable et d'exister sur Mac ou Linux.

Dans la suite de cet article j'utiliserai donc Visual Studio pour les raisons évoquées ici car la solution que je vais vous montrer couvre Windows Phone et Android (iOS aussi mais je n'ai pas de Mac pour l'émulateur en ce moment).

Bien entendu tout ce qui est montré ici nécessite un setup complet : Visual Studio, une machine Windows 10 pour faire tourner la partie Windows Phone (ou UWP), le SDK de ce dernier et bien sûr une licence Xamarin installée et tout le SDK Android. Cela fait beaucoup mais c'est le prix du ticket d'entrée pour faire du cross-plateforme… On notera que depuis l'intégration des XF à Visual Studio le processus d'installation de tout cela est plus « compact » et plus « naturel ».

Etape 1 – Créer la solution

Sous Visual Studio on fait comme d'habitude pour créer un nouveau projet. C'est lorsqu'arrive le dialogue de création qu'il faut choisir la bonne solution :



Il faut farfouiner dans la liste de gauche pour trouver "Mobile Apps" et c'est alors qu'à droite on trouve trois possibilités liées aux Xamarin.Forms. Soit une librairie de classes, mais cela ne nous intéresse pas ici, soit un projet vide (Blank App) mais sous deux formes possibles : PCL ou SAP. La présentation exacte dépendra bien entendu de la version de VS que vous utiliserez (ici j'utilisais un VS 2013 SP4).

PCL ou SAP ?

En mode *Shared Assets Project* il existe trois projets mobiles, un pour chaque plateforme, et un projet contenant du code partagé. Ce code est compilé par chaque application mobile. C'est un mode un peu enquiquinant car les trois plateformes n'ont pas tout à fait le même framework .NET même s'ils sont très proches. Le code partagé appartenant aux trois projets à la fois il faut faire attention à ce qu'on écrit pour éviter des incompatibilités. En revanche ce code partagé est systématiquement natif puisque lié et compilé par chaque projet cible. Il est donc possible d' écrire du code ultra spécifique pour l'une ou l'autre des cibles.

En mode PCL, le plus courant et le plus pratique, c'est Visual Studio (ou Xamarin Studio) qui s'occupe de créer une sorte de "vue" sur l'intersection des frameworks sélectionnés. De fait un tel projet partagé en mode PCL est avant tout une DLL compilée pour elle-même et validée en permanence pour "rester dans les clous" de l'ensemble d'OS choisi au départ (le "profil .net").

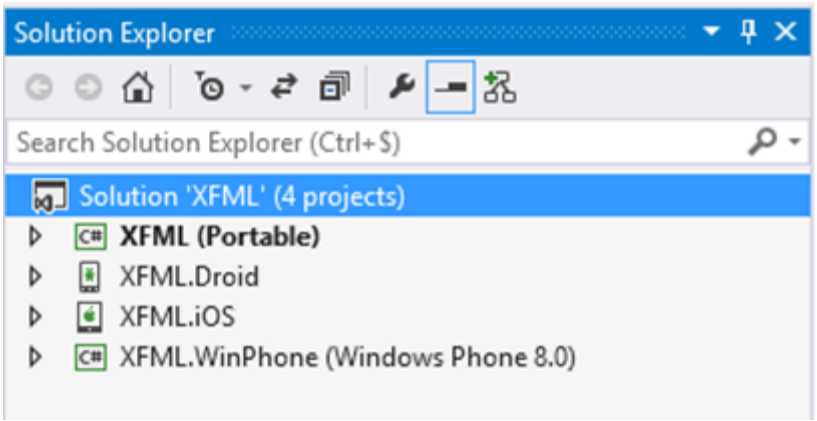
Ce mode est avantageux en cela qu'il permet d'être certain de bénéficier d'un framework totalement commun à toutes les cibles rendant l'écriture plus fiable et plus aisée. En revanche puisque la PCL est compilée pour elle-même il est impossible "out of the box" d'y écrire du code spécifique à l'une ou l'autre des plateformes cibles. Heureusement il existe des façons de contourner cette limitation et ces moyens sont plutôt plus "propres" que de farcir son code de #if. Xamarin offre un moyen simple par exemple de retourner des valeurs qui dépendent de la plateforme et ce même dans du code Xaml. Peu importe les noms de méthodes ou les astuces il s'agit en réalité du principe que je proposais dans ma vidéo sur l'injection de code natif dans un projet cross-plateforme. On peut (re)visualiser cette vidéo (qui utilise MvvmCross et Xamarin) pour comprendre le mécanisme utilisé.

Bref, pour pas mal de raisons il est donc souvent préférable d'utiliser une PCL plutôt que le mode SAP. Toutefois comme d'habitude c'est au développeur de faire son choix en fonction des contraintes du projet à développer.

Ici j'utiliserai donc un projet "Blank App (Xamarin.Forms Portable" c'est à dire en mode PCL.

La solution de base

Comme le montre la figure ci-dessous la solution créée par VS contient bien 4 projets, le projet PCL et les trois projets des plateformes supportées (Android, iOS et Windows Phone). Avec les dernières versions de VS et Xamarin.Forms la liste sera légèrement différente avec notamment l'arrivée d'un projet UWP.



Le nom de la solution qu'on retrouve dans chaque projet est *XFML* pour Xamarin.Forms et Mvvm Light. On donne les noms qu'on peut… Le projet où ce nom est tel quel est le projet commun, la PCL, les autres projets ajoutent au nom de base les extensions ".Droid" pour Android, ".iOS" pour… iOS et .WinPhone pour Windows Phone.

J'écrivais à l'origine « *On notera que pour l'instant le mécanisme utilise Windows Phone 8.0 c'est à dire en mode "Silverlight" et non du 8.1 mais cela viendra certainement à moins que Xamarin ne prépare directement le passage à Windows 10. Mais le code compilé tourne bien entendu sur Windows Phone 8.1.* ». J'avais bien deviné, les XF sont passée à UWP ce qui couvre les unités mobiles Microsoft ainsi que les PC.

Lorsque la solution se créée, comme elle ajoute le projet iOS systématiquement Xamarin qui tourne derrière réclame la connexion à la machine Mac où sera exécuté l'émulateur. On peut ignorer tout cela et supprimer à la fin du cycle le projet iOS, ce que j'ai fait.

Mettre à jour les packages

Avant de commencer à coder il est préférable de s'assurer que tous les packages installés sont à jour. Le template de solution n'utilise pas forcément les dernières versions de Xamarin.Forms notamment.

La technique est simple, il suffit de visiter les références de chaque projet (tous) de faire un clic droit et d'accéder à la gestion des paquets Nuget puis de regarder à gauche dans la rubrique "mises à jour" et d'installer toutes celles qui sont disponibles.

Mise à jour de la PCL

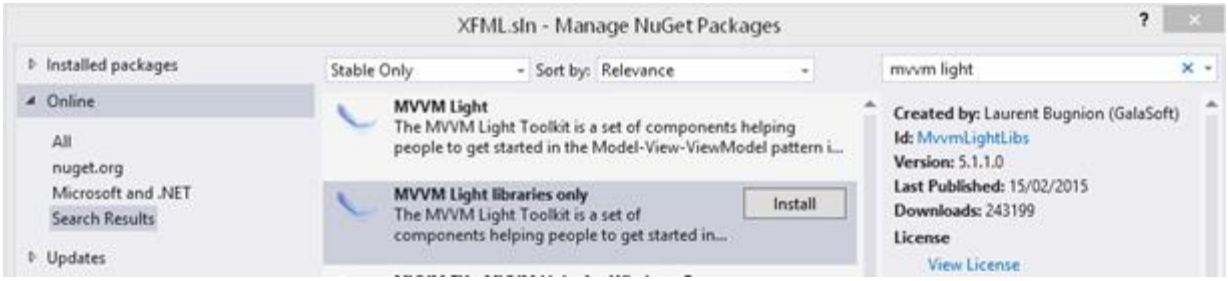
Mise à jour du projet Android

Mise à jour du projet Windows Phone

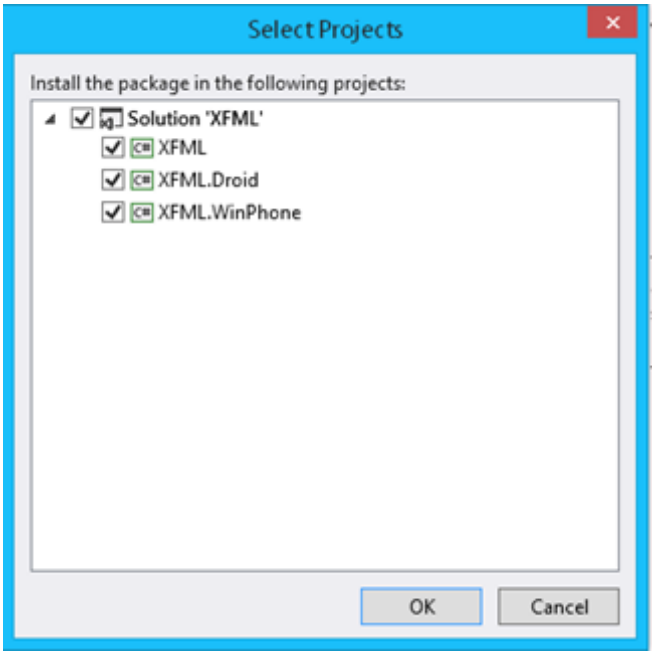
On peut faire plus simple en faisant un clic droit sur la solution et en demandant "Manage Nuget Package for Solution". Ici il est possible de faire les mêmes opérations mais pour tous les projets à la fois avec même la possibilité de sélectionner ceux qui devront être manipulés… Pourquoi faire compliqué lorsqu'il existe des astuces de ce genre !

Installer MVVM Light

C'est d'ailleurs en passant par la solution et sa gestion globale des paquet Nuget que je vais installer MVVM Light pour tous les projets à la fois.



On choisit la rubrique "online", on cherche "mvvm light" et on installe "MVVM Light Libraries only". Nous n'avons en effet pas besoin de tout le template habituellement installé, seules les librairies de base sont nécessaires (pour chaque projet donc).



Le choix des projets impactés quand on fait des opérations sur les paquets Nuget via la Solution

MVVM Light Portable

En devenant portable MVVM Light a dû s'adapter un peu. Même si ce n'est pas l'endroit pour discuter en profondeur de ces changements il faut noter au moins qu'il n'y a pas par défaut de `ViewModelLocator` par exemple.

Ce dernier n'est pas absolument nécessaire mais il permet une séparation du code que j'aime bien, nous recréerons cette classe dans quelques instants.

Il faut aussi savoir que MVVM Light Portable s'il propose toujours sa gestion d'IoC (qui peut d'ailleurs être remplacée facilement) se fonde désormais sur un code de Patterns & Practices de Microsoft : le CommonServiceLocator qui autorise une unification de l'accès aux conteneurs d'IoC. Cela permet à tous les frameworks qui utilisent cette interface de proposer différents conteneurs tout en maintenant un code unifié et similaire puisque traité par l'interface commune. En suivant le lien précédent vous accèderez au projet CodePlex de cette librairie.

Sinon pour l'essentiel MVVM Light fonctionne toujours de la même façon et surtout selon les mêmes principes pour les mêmes objectifs. Le lecteur intéressé pourra se plonger dans le livre gratuit "Méthodes & Frameworks MVVM" qu'on retrouve sur la page des livres de la collection "ALL DOT BLOG".

Etape 2 – Créer le code de l'app !

La solution est créée, les paquets Nuget de tous les projets sont à jour et nous avons installé MVVM Light dans chacun d'entre eux. La place est nette et propre et on peut se lancer dans l'écriture du code !

La page principale

Dans le projet PCL, le seul que nous toucherons dans cet article, nous créons une nouvelle page XAML : Add new item / Form XAML Page.

Cela créée une fiche XAML spécifique à Xamarin.Forms. A la fois il s'agit de quelque chose de très familier, un code XAML et son code behind en C# comme on le retrouve sous WPF, et à la fois il y a une différence de taille : il n'y a pas pour l'instant de designer visuel pour ce sous-ensemble de XAML. On peut d'ailleurs choisir de tout faire en code C# sans utiliser XAML. Mais ce dernier est bien plus clair pour spécifier une interface utilisateur que la création imbriquée d'instances de classe dans un code C#.

J'étais convaincu quand Xamarin est apparu qu'un portage XAML serait un rêve, j'oubliais que ce rêve n'est total qu'avec Blend ou Visual Studio et leur concepteur visuel ! Mais le rêve est déjà à moitié réalisé et en cross-plateforme. Nul doute que Xamarin ne s'arrêtera pas en si bon chemin surtout depuis son rachat par Microsoft et l'intégration des XF à Visual Studio. Mais pour l'instant il faut un peu de pratique WPF ou Silverlight pour concevoir toute une UI en XAML. Sinon il reste le mode C# donc.

Ici en vieux routier de XAML j'opte pour cette solution mais ce n'est qu'un choix personnel même s'il se fonde sur les qualités indéniables de XAML pour décrire une UI.

Il n'y a pas grand chose dans cette "`MainView`" (puisque tel est le nom que je lui ai donné) ! juste une `ContentPage` vide, un contrôle des Xamarin.Forms communément utilisé pour créer des pages contenant tout ce qu'on veut.

On note que l'entête de ce XAML est plus expéditif que celui d'un XAML WPF ou Silverlight ce qui permet rapidement de faire la différence.

Comme nous n'avons pas de ViewModel pour l'instant, nous laissons ce code de base en place et nous y reviendrons plus tard pour créer l'UI.

ViewModelLocator

Toute la logique d'une application se concentre dans des classes de service et dans les ViewModels lorsqu'on travaille en suivant MVVM. Les Xamarin.Forms permettent de travailler en suivant ce pattern sans obliger la présence d'un framework supplémentaire.

J'ai déjà expliqué dans un vieil article comment implémenter du code suivant MVVM sans utiliser de framework particulier pour montrer les principes fondateurs de ce pattern. Dans une petite app il est évident qu'on peut même être tenté de tout faire dans le code behind de la page XAML… après tout… Mais je ne vais pas refaire le match ! C'est un sujet éculé sur lequel j'ai tellement écrit qu'on considèrera qu'utiliser un framework comme MVVM Light, justement "light", reste la meilleure approche. Pour ceux qui resteraient dubitatif je ne peux que les renvoyer à tous mes articles sur MVVM (et ça fait de la lecture !).

Pour commencer nous allons créerons un répertoire "`ViewModel`" dans le projet PCL. Et dans ce répertoire nous allons ajouter une nouvelle classe "`MainViewModel`" puisque la vue s'appelle "`MainView`". Pour l'instant nous laissons cette classe vide.

Créons maintenant toujours dans le répertoire ViewModel la classe "`ViewModelLocator`".

C'est ici que nous allons recréer le ViewModel locator de MVVM Light qui n'apparait pas dans la version Portable. Non pas que cette indirection soit nécessaire techniquement parlant mais tout simplement parce qu'il semble toujours légitime dans l'esprit MVVM d'éviter tous les couplages forts et de proposer des points d'accès centralisés facilement identifiables pour la maintenance. Ainsi ce locator va isoler les vues de la création des instances des ViewModels. Là encore inutile de refaire le débat sur la notion de couplage faible qui est devenue l'une des bases de la programmation "moderne". Acceptons que ce soit une bonne pratique, les plus curieux pourront lire mille choses sur la question pour se faire leur opinion.

Voici le `ViewModelLocator` de notre application, recréé au plus proche de son équivalent sous MVVM Light "classique" :

```
using GalaSoft.MvvmLight.Ioc;

using Microsoft.Practices.ServiceLocation;
```

```
namespace XFML.ViewModel
{
    /// <summary>
    /// This class contains static references to all the view models in the
    /// application and provides an entry point for the bindings.
    /// </summary>
    public class ViewModelLocator
    {
        /// <summary>
        /// Initializes a new instance of the ViewModelLocator class.
        /// </summary>
        public ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

            ////if (ViewModelBase.IsInDesignModeStatic)
            ////{
            ////    // Create design time view services and models
            ////    SimpleIoc.Default.Register<IDataService, DesignDataService>();
            ////}
            ////else
            ////{
            ////    // Create run time view services and models
            ////    SimpleIoc.Default.Register<IDataService, DataService>();
            ////}

            SimpleIoc.Default.Register<MainViewModel>();
        }

        public MainViewModel Main
        {
            get
            {
                return ServiceLocator.Current.GetInstance<MainViewModel>();
```

```
            }
        }

        public static void Cleanup()
        {
            // TODO Clear the ViewModels
        }
    }
}
```

Ce qui ne change pas : l'utilisation de `SimpleIoc` pour enregistrer les ViewModels dans le conteneur et la création d'une propriété par ViewModel. L'obtention de l'instance par le biais du conteneur d'IoC est tout aussi classique tout comme la présence d'une méthode CleanUp dans laquelle on peut placer du code de nettoyage pour supprimer les ViewModels en mémoire par exemple.

Ce qui change : L'utilisation de `Microsoft.Practices.ServiceLocation` qui permet d'isoler le conteneur d'Ioc utilisé (ici `SimpleIoc`) en fournissant une interface commune non dépendante de la classe réellement instanciée. Dans le même esprit, sinon cela ne servirait à rien, c'est par le biais de ce `ServiceLocator` qu'on obtient les instances des ViewModels, SimpleIoc étant alors totalement gommé même si c'est lui qui effectue le travail. On peut ainsi envisager de changer ce conteneur par un autre à tout moment sans aucun impact sur le code.

Le ViewModel

Il est temps de revenir à notre ViewModel. Nous avions créé juste une classe vide pour pouvoir la référencer dans le `ViewModelLocator` selon le vieux principe de qui de la poule ou de l'oeuf etc..

Pour cet exemple je vais donner dans le simple, le Hello World de MVVM : un bouton et un label, quand on clique sur le bouton le label indique combien de fois on a cliqué. C'est simple mais cela permet de mettre en évidence les mécanismes de base comme la création d'un ViewModel, son héritage, les propriétés publiques avec le support de l'INPC (notification de changement), la création de commande, les bindings… Beaucoup de choses dans peu de code, c'est toujours une bonne pratique !

Voici le code de MainViewModel :

```
using GalaSoft.MvvmLight;
```

```
using GalaSoft.MvvmLight.Command;

namespace XFML.ViewModel
{
    public class MainViewModel : ViewModelBase
    {
        private int count;

        public int Count
        {
            get { return count; }
            set
            {
                if (Set(()=>Count,ref count,value))
                    RaisePropertyChanged(()=>FormattedCount);
            }
        }

        public string FormattedCount
        {
            get { return string.Format("Vous avez cliqué {0} fois", Count); }
        }

        private RelayCommand clickCommand;

        public RelayCommand ClickCommand
        {
            get
            {
                return clickCommand ??
                        (clickCommand = new RelayCommand(() => Count++));

            }
        }
```

```
    }
}
```

On notera :

- L'héritage de la classe mère `ViewModelBase`, un classique sous MVVM Light. Cela offre de nombreux services essentiels pour le support de MVVM comme l'INPC (petit non de l'interface `INotifyPropertyChanged`).
- La création d'une propriété pour le compte et une autre qui reprend la valeur entière pour la formater en chaîne. Ce qui permet de voir comment notifier le changement d'une propriété dérivée.
- La création d'une commande qui gèrera le clic sur le bouton.

Il n'y a rien d'exotique ici, que des choses classiques sous MVVM Light.

Et c'est cela qui est important : la proximité est telle avec un code "desktop" qu'on peut même envisager de reprendre des morceaux d'une application WPF ou Silverlight pour les porter presque sans effort sous Android, Windows Phone, UWP et iOS !

J'ai peut-être gardé une âme d'enfant mais je m'émerveille devant une telle portabilité, une telle identité, une qualité similaire entre ce code destiné à tourner sur un mobile et un code WPF pour gros PC de bureau. Sachant qu'ici ce code est écrit une seule fois pour tourner sur trois plateformes qui n'ont vraiment rien en commun.

Bref notre VM est maintenant complet et va fonctionner aussi bien sur un iPhone qu'un Nexus ou un Lumia et on pourrait même le réutiliser tel quel pour du UWP ou du WPF. Pas mal quand même.

Mais ça c'était déjà vrai avec Xamarin sans les Xamarin.Forms. Ce qui manquait cruellement c'était l'unification des UI justement. Faire des ViewModel portables on le faisait avec MvvmCross par exemple (et Xamarin). Passons donc à ce qui fait tout l'intérêt des Xamarin.Forms, l'unification des UI.

App.cs

Juste avant de passer au visuel, il nous reste un petit détail à régler dans le `App.cs`.

D'abord cette classe doit descendre de `Application` qui est fournie avec les Xamarin.Forms, le template doit le faire mais il faut s'en assurer.

Ensuite nous allons créer une propriété statique pour exposer le `ViewModelLocator`. La classe `App` est un bon endroit pour cela car il y a forcément une et une seule instance pour toute application Xamarin.Forms on est donc certain de pouvoir s'y référer.

Enfin dans le constructeur de `App` nous devons charger la fiche principale de l'application, notre `MainView` ici.

Ce qui donne ce code très simple mais important :

```
using Xamarin.Forms;
using XFML.ViewModel;

namespace XFML
{
    public class App : Application
    {

        private static ViewModelLocator locator;

        public static ViewModelLocator Locator
        {
            get { return locator ?? (locator = new ViewModelLocator()); }
        }
        public App()
        {
            // The root page of your application
            MainPage = new MainView();
        }

        protected override void OnStart()
        {
            // Handle when your app starts
        }

        protected override void OnSleep()
        {
```

```
            // Handle when your app sleeps
        }

        protected override void OnResume()
        {
            // Handle when your app resumes
        }
    }
}
```

Le visuel

L'application est presque terminée, reste la partie visuelle portable. Mais qui dit MVVM, qui dit MVVM Light dit forcément databinding.

Or pour l'instant si nous avons défini le ViewModel, si nous avons créé un locator pour récupérer l'instance de ce dernier nous ne savons toujours pas comment la vue va s'y rattacher.

Beaucoup de frameworks MVVM propose des solutions de routage basées soit sur un fichier de configuration, soit l'utilisation d'attributs, soit des conventions de noms ou même de l'injection de dépendance.

On peut parfaitement supporter l'une ou l'autre de ces approches mais il faut bien avouer qu'une app mobile n'est tout de même pas (en tout cas pas encore) aussi grosse que certaines applications desktop qui méritent en effet un tel niveau de sophistication. Certaines pratiques vont même jusqu'à instaurer l'utilisation d'interface décrivant le ViewModel pour que les Vues ne voient pas même l'instance de ce dernier, en tout cas qu'au travers de cette liaison faible qu'est l'interface.

Ici notre vue sait très bien à quel ViewModel elle doit se lier. le code-behind de la vue pourrait donc fort bien créer directement une instance du ViewModel et l'attribuer à son contexte. Certains pratiquent de cette façon et ce n'est finalement pas si "coupable" que ça. Dans la réalité je n'ai jamais vu une View dont on change après coup totalement la classe de ViewModel attaché. Une Vue à besoin de certaines données qu'on retrouve dans ses bindings il est donc un peu hypocrite de vouloir séparer ce qui de toute façon est lié. Quant au Unit Testing… on en parle, on en parle… et dans la réalité cela est très peu appliqué car cela coûte cher. « *Comment vous faisiez avant, vous me faisiez des softs pourris ? Heuu non bien sûr monsieur Le Client.. Et bien faites comme avant !* » Le Unit Testing

restera ainsi une simple proposition bien vite oubliée de la première version du devis !

Mais comme l'expérience nous apprend que des couplages forts finissent toujours par devenir une gêne et un frein à la maintenabilité et l'évolutivité d'un code nous avons recréé le `ViewModelLocator`. Il offre une séparation suffisante entre consommateur de ViewModels (les Views en général) et les instances des ViewModels eux-mêmes. De fait cette isolation créée par le locator semble assez étanche et faiblement couplée pour éviter de créer d'autres complications. Ainsi c'est dans le code-behind de la page XAML que nous allons instancier le ViewModel mais en passant par le `ViewModelLocator` pour conserver l'esprit d'un couplage faible.

Le code-behind de la page XAML devient alors :

```
namespace XFML
{
    public partial class MainView : ContentPage
    {
        public MainView()
        {
            InitializeComponent();
            BindingContext = App.Locator.Main;
        }
    }
}
```

Pas de quoi hurler à la lune, juste une ligne de plus sous "`InitializeComponent`". Cette ligne utilise la propriété statique de la classe `App` permettant de retrouver le `ViewModelLocator` (donc couplage faible avec ce dernier) pour obtenir enfin le `MainViewModel` au travers de la propriété Main du locator (second niveau d'indirection et de couplage faible).

Comme on peut le constater les `ContentPage` de Xamarin.Forms n'utilisent pas la propriété `DataContext` mais "`BindingContext`". C'est une cassure dans la logique XAML dont je ne vois pas l'intérêt ou l'utilité. Mais ce n'est pas très gênant une fois qu'on le sait… C'est donc le `BindingContext` qui se voit attribuer l'instance du ViewModel.

Voilà, notre page XAML est maintenant connectée à son ViewModel, avec un couplage très faible.

Le code XAML

Nous pouvons reprendre notre MainView pour lui ajouter le visuel de l'application. A savoir un bouton et un label. Nous placerons tout cela dans l'équivalent d'un StackPanel qui sera centré verticalement.

Au final le code XAML spécifique Xamarin.Forms devient le suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XFML.MainView">
  <StackLayout Orientation="Vertical" VerticalOptions="Center">
    <Button Text="Cliquez ici !"
            Command="{Binding ClickCommand}"/>
    <Label Text="{Binding FormattedCount}" HorizontalOptions="Center"  />
  </StackLayout>
</ContentPage>
```

Pour qui connait XAML tout cela est limpide. Seules les classes visuelles et leurs options sont un peu différente de WPF. N'oublions pas que c'est à cela que sert Xamarin.Forms : à créer une couche XAML de composants universels qui seront remplacés à la compilation par leurs équivalents NATIFS. Il n'y a pas de surcouche finale, aucune ruse, pas d'hybridation étrange où on planque du HTML pourri dans un webview encapsulé dans une coquille native pour tromper l'ennemi. Rien de tout cela ici. On produit du NATIF. Et pas n'importe lequel, on produit du natif PORTABLE.

Tout l'art des Xamarin.Forms c'est d'avoir créer des composants "fantômes" qui permettent d'écrire du XAML universel tout en permettant par une gestion intelligente des dénominateurs communs une compilation native utilisant des contrôles natifs pour chaque plateforme.

C'est un magnifique tour de force.

Le code XAML ci-dessus nous montre une structure habituelle, l'utilisation de contrôle ayant des noms évocateurs comme le `StackLayout` dont on comprend la similarité avec un `StackPanel` WPF par exemple. On y voit aussi une classe `Button`

pour laquelle nous n'avons pas même besoin de réfléchir au sens ainsi que du binding tout à fait classique sur les propriétés et commandes de notre ViewModel.

Bref une application C#/XAML "comme les autres".

A cette petite nuance près qu'elle va maintenant tourner sur des plateformes différentes sans réécriture ni même le moindre ajustement.

Etape 3 – Faire tourner !

Pour exécuter notre code il nous faut des émulateurs. Chaque plateforme a les siens. Microsoft utilise Hyper-V pour simuler Windows Phone, Google propose son émulateur, Intel en propose un plus rapide et même Xamarin propose un émulateur Android encore plus rapide. Microsoft en propose maintenant aussi un qui a l'avantage d'être parfaitement compatible avec Hyper-V.

Je n'aborderai même pas le cas de iOS puisque là il faut absolument un Mac en réseau pour faire tourner l'émulateur. Apple à une approche des choses qui me file de l'urticaire rien d'y penser et ce depuis toujours.

Donc munis d'un émulateur Windows Phone qui marche (ce qui est bien plus clair aujourd'hui sous UWP) et d'un émulateur Android pas trop lent nous pouvons envisager de faire tourner notre magnifique application. Il est clair que si on dispose de devices réelles c'est finalement moins compliqué et plus réaliste. Ici les émulateurs permettent des captures écran ce qui m'arrange !

iOS

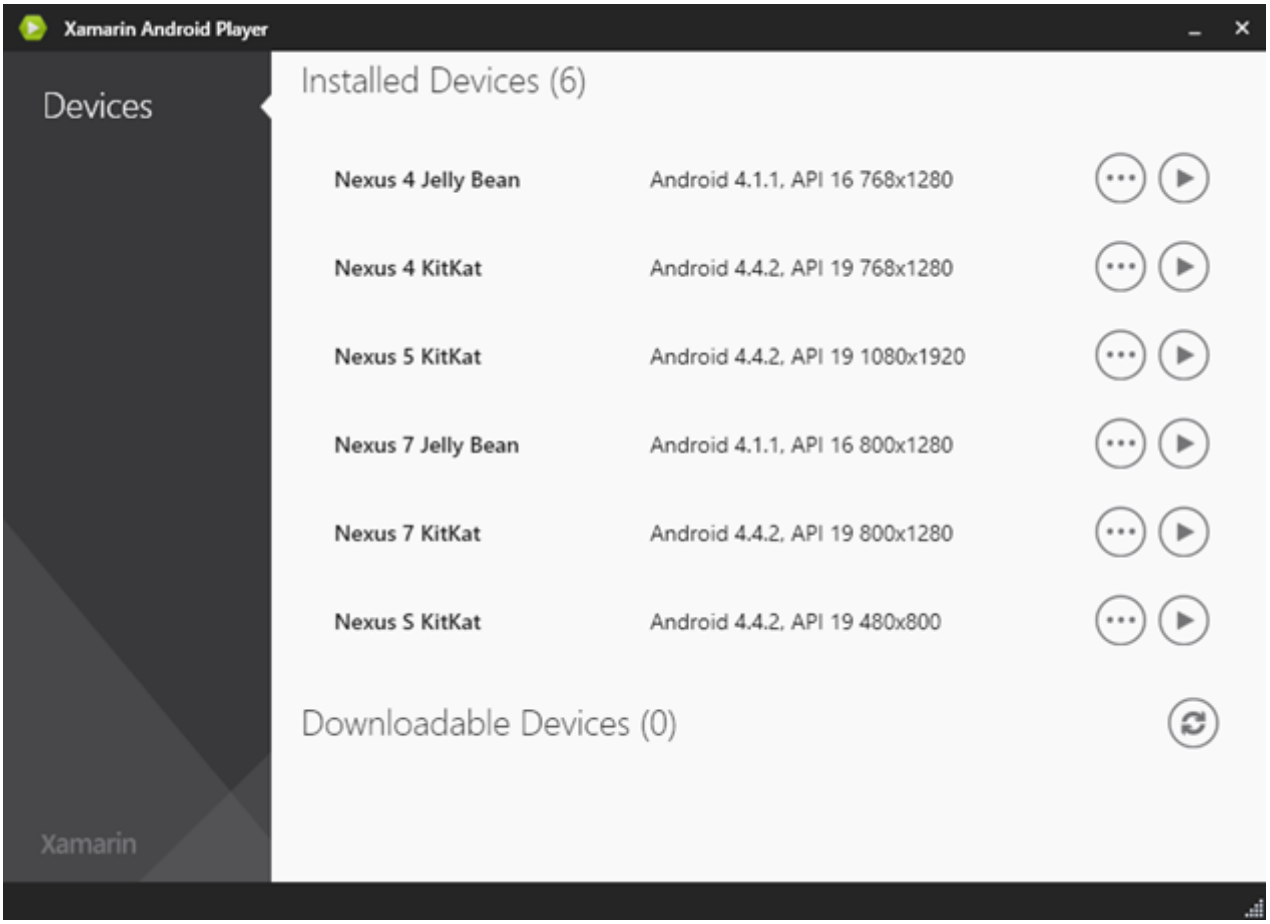
Puisque je vous dis que je n'ai plus de Mac et que je n'en veux plus ! Mais les « heureux » possesseurs d'une telle machine pourront bien entendu faire l'expérience et vérifier par eux-mêmes qu'aussi incroyable que cela puisse paraître, ça marche.

Android

Parlons plus sérieusement. Selon l'émulateur que vous choisissez le visuel peut être différent, ici je vous montrerai le Xamarin Android Player et l'émulateur Google.

Xamarin Android Player

Le choix de la device à émuler :



L'exécution sous Player :

Emulateur Google

L'exécution sous l'émulateur Google :

Un petit oubli dans le code XAML fait que le label n'était pas centré… cela est corrigé pour l'exécution suivante sous Windows Phone.

Windows Phone

Le joli splash screen Xamarin :

La même application sans aucun changement, même code, même UI, même C#, même XAML :

Conclusion

Qu'y-a-t-il de plus à ajouter…

Un code C#, un code XAML, trois plateformes couvertes.

Xamarin n'était pas gratuit, mais il l'est aujourd'hui, et c'est même un produit Microsoft. Plus aucun frein n'existe. Rappelez-vous : c'est soit trois applications dans trois langages, EDI, jeux d'API et autres totalement différents avec trois compétences différentes ou bien Microsoft Xamarin.Forms et une seule personne qui connait C#/XAML peut faire une app pour les 3 OS du marché…

Forcément l'argument est imparable, tout comme je l'espère ma démonstration.

Xamarin.Forms et Injection de dépendances

L'injection de dépendances et les conteneurs IoC (inversion de contrôle) qu'est-ce que cela a à voir avec Xamarin et les Xamarin.Forms ? Comment tout cela est-il lié ? Lisez la suite !

L'injection de dépendances

C'est un sujet dont je vous ai déjà parlé plusieurs fois et dans différentes occasions, dont le développement cross-plateforme. Mais avant d'entrer dans le vif du sujet de ce chapitrefaisons un point rapide : Qu'est-ce que l'injection de dépendances ?

Selon Wikipédia on nous dit :

L'injection de dépendances (Dependency Injection) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle. Il consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

Je ne sais pas qui a écrit cela mais je m'imagine quelqu'un qui ne sait pas ce qu'est la DI (*Dependency Injection*) et je me demande si une telle définition peut l'éclairer vraiment…

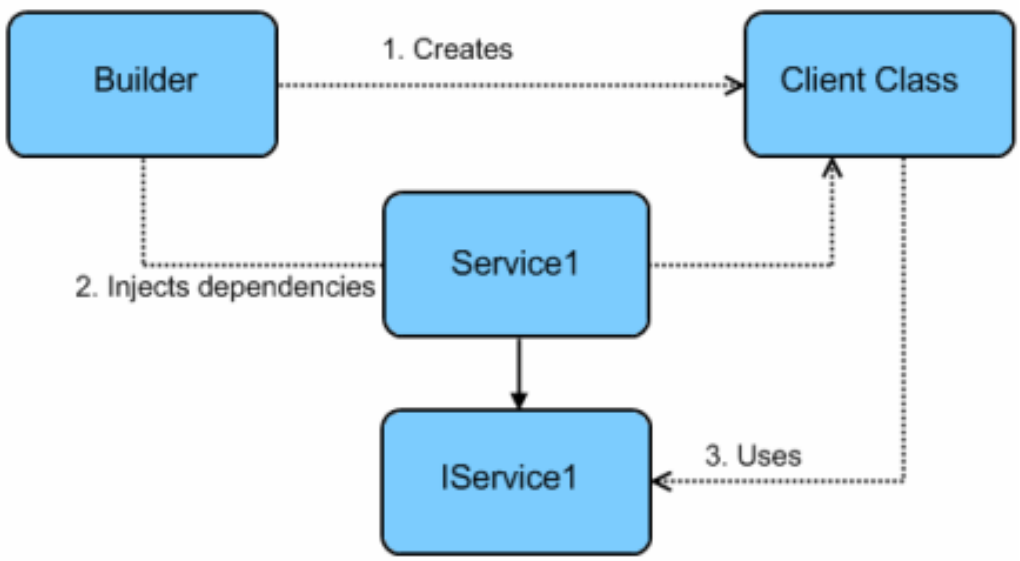
Je préfère personnellement dire :

L'injection de dépendance est un ensemble de principes de conception et de patterns qui permettent de développer un code à faible couplage.

Certes cela impose d'expliquer la notion de faible couplage, mais ce n'est pas dur non plus :

La notion de couplage faible fait référence à la façon dont les classes et les instances interagissent entre elles. Si ces classes ou instances se "connaissent" et s'appellent directement le couplage est dit fort. Si les classes et instances peuvent communiquer, s'utiliser, s'appeler sans se connaitre (par ex. sans faire un "using" de l'assemblage de ces classes) on parle de couplage faible.

On peut schématiquement représenter le concept d'injection de dépendances comme suit :



L'âme sur laquelle repose les implémentations de la DI est la **notion d'interface**. Une interface définit un *contrat*qui peut être implémenter par plusieurs classes. *Les classes clientes peuvent utiliser le contrat au travers de l'interface sans avoir besoin de connaitre la définition des classes d'implémentation*.

Le schéma ci-dessus n'est pas absolu, ce n'est qu'une façon de pratiquer l'injection de dépendances. Ici un *Builder* va créer une instance de "*Client Class*" en lui passant l'instance de *Service1*. Toutefois "*Client Class*" ne verra cette dernière que sous la forme de l'interface *IService1* qu'elle supporte.

Où se trouve le couplage faible ? Entre "*Client Class*" et "*Service1*". Comment ? grâce à "*IService1*" et au *Builder*. De fait "*Client Class*" utilisera *Service1* sans connaitre la définition de cette classe, juste en se reposant sur la connaissance de *IService1*.

Intérêts ?

Le couplage faible apporte d'abord plus de *clarté* au code. Ensuite le découplage entre clients et services permet à tout moment de proposer aux clients des *services d'un nouveau type* sans que leur code ne soit impacté. Si je définis `ILogger` une interface pour faire des Logs, et si toutes les classes de mon application ne connaissent que `ILogger` il me sera très facile d'écrire des classes comme `LogToXML`, `LogToDb`, `LogToCloud`, etc qui chacune respecteront `ILogger` mais qui stockeront ou présenteront les Logs de façon différente. *Et cela sans que jamais les classes clientes ne puissent voir ni connaitre la différence*.

C'est puissant et pratique. On pense par exemple à l'extensibilité d'une application via des *plugins*. Les plugins supportent une ou plusieurs interfaces et le code client peut utiliser ces nouveaux services sans même le savoir.

Il existe donc quelque part une sorte d'aiguillage qui lui connait, par force, les véritables instances. C'est le `Builder` dans le schéma plus haut. Mais il y a d'autres façons de coder un tel couplage faible. Notamment par l'intermédiaire de conteneurs IoC (inversion de contrôle). A proprement parler ce ne sont que de simples dictionnaires qui associent un type (celui d'une interface) à un type de service (ou une instance existante d'un tel service). Avec les classes du schéma plus haut un conteneur IoC permettrait d'enregistrer le couple instance (ou type) de `Service1` / `IService1`. Au lieu du `Builder` qui contrôle toutes les instanciations, c'est directement "*Client Class*" qui demanderait alors au conteneur IoC (un singleton) de lui fournit l'instance implémentant `IService1`. Le conteneur retournerait donc l'instance `Service1`. Mais selon les évolutions du programme, de choix paramétrables cela pourrait aussi bien être des instances de `Service2`, `Service3bis` qui seraient retournées, tant que ces classes implémentent `IService1`.

Le conteneur IoC n'est pas indispensable à la mise en œuvre de l'injection de dépendances mais dans la pratique c'est un complément incontournable pour gagner en souplesse et éviter d'écrire un code trop fastidieux. Une autre raison plus profonde est que le conteneur IoC n'est pas juste qu'un dictionnaire. Sinon autant utiliser une structure de ce type… Le conteur d'IoC est aussi capable lorsqu'il fournit une instance de la construire en lui passant les dépendances dont elle a besoin (les services qu'elle déclare consommer). C'est là qu'intervient la fameuse « injection » de dépendances…

On peut ensuite inventer des tas de façons différentes de gérer chaque étape. Par exemple on peut avoir un fichier XML de configuration qui précise les couples interface / classe d'implémentation. La configuration est lue en début d'exécution de l'application et elle construit le dictionnaire. On peut aussi supposer à l'inverse un code qui automatiquement va balayer tous les assemblages en mémoire et qui va collecter toute les classes implémentant une interface et ayant par exemple un nom se terminant par "*Service*"… Cela ne vous dit rien ? C'est comme cela que MvvmCross pratique pour gérer les services de façon automatique.

De la même façon l'utilisation des services peut se pratiquer très simplement, la classe cliente réclame au conteneur l'instance de l'interface, ou bien de façon plus sophistiquée. Par exemple avec des attributs. C'est comme cela que MEF fonctionne. On peut aussi avoir des classes clientes qui exposent un constructeur recevant en paramètre tous les services nécessaires à leur fonctionnement. Il existe alors bien quelque part un "builder" qui permet d'instancier les clients et qui

va analyser le constructeur pour passer directement en paramètres les services (en regardant dans le dictionnaire du conteneur IoC et par réflexion pour analyser le constructeur et ses paramètres). C'est ce builder qui injecte les dépendances.

Bref le principe est simple mais on peut le complexifier, le sophistiquer, l'automatiser à son gréé ce qui donne des tas de solutions possibles, des tas de frameworks plus ou moins difficiles d'ailleurs à prendre en main.

Mais au bout du compte on retrouvera toujours un conteneur IoC qui d'apparence n'est qu'un dictionnaire de couples interface / classe d'implémentation (ou instances existantes) mais qui gère aussi l'injection des dépendances. Ce qui fait la différence entre un `dictionay<TKey,TValue>` et un conteneur IoC !

Et forcément selon comment ce dictionnaire un peu spécial est géré impacte à la fois sur les possibilités plus ou moins larges de gérer l'injection de dépendances elle-même ainsi que les performances globales de l'application.

C'est pourquoi on trouve de nombreux conteneurs IoC basés sur des philosophies différentes.

Choisir l'un plutôt que l'autre dépendra donc directement des priorités qu'on se donne. On retrouvera souvent le problème de la juste balance à effectuer entre richesse fonctionnelle et temps de prise en main, complexité du code et rapidité d'exécution.

Un dernier mot sur ce que n'est pas l'injection de dépendance.

Il faut savoir désapprendre pour apprendre, savoir faire le vide des idées reçues pour en accepter d'autres plus proches de la vérité.

Dit comme ça on est en plein dans les leçons de sagesse un peu bidon d'un film d'Arts Martiaux à la sauce américaine. Van Damme sors de ce corps ! Ok. Mais ce n'est pas faux pour autant !

Il y a certaines choses sur l'injection de dépendances qui sont dites et redites et qui égarent celui qui veut comprendre ce mécanisme.

Par exemple certains présentent la DI comme une simple façon plus "branchée" d'appeler le *late binding* (ligature différée). Si la DI peut servir à faire du late binding ce n'est pas son but premier. Ce n'est qu'une utilisation possible.

D'autres encore parlent de la DI comme une technique uniquement utile pour les tests unitaires. Comme pour le late binding ce n'est pas totalement faux, mais c'est trop réducteur. En effet la DI peut simplifier les tests unitaires (en fournissant des

mocks au lieu de services réels par exemple), mais là encore ce n'est pas son but et encore moins son but unique.

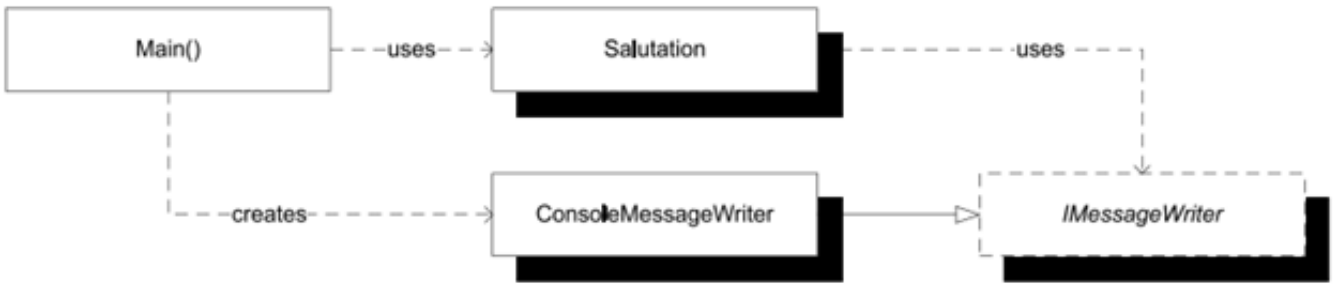
On peut aussi entendre parler de la DI comme une sorte de copie dopée du design pattern "abstract factory". Souvent même on présente la DI comme un service permettant de localiser d'autres services. Mais cela porte un autre nom, c'est le Service Locator… Mais il reste possible d'utiliser un conteneur IoC pour mettre en œuvre une abstract factory ou un service locator, c'est là que les apparences deviennent trompeuses. Prenez par exemple le `ViewModelLocator` de Mvvm Light, il utilise désormais un conteneur IoC pour mémoriser les ViewModels. C'est un locator, c'est aussi une sorte de factory, il utilise un conteur IoC donc quelque part (peut-être, car les deux ne sont pas lié !) de la DI mais cela n'a rien à voir avec cette dernière en réalité… C'est parfois un peu confusant je l'accorde et l'erreur est fréquente même chez des informaticiens confirmés.

Enfin, il semble admis comme une vérité première qu'il ne peut y avoir de DI sans un conteneur. C'est tout aussi faux. Au même titre que j'ai démontré dans plusieurs articles comment mettre en œuvre MVVM en se passant de toute librairie externe, on peut parfaitement implémenter une DI sans utiliser un conteneur IoC. Cela peut réclamer plus de code et l'intérêt n'est pas forcément flagrant mais en tout cas il n'y a pas de lien obligatoire entre DI et conteneur IoC.

Bref, la DI sert à concevoir un code à couplage faible et c'est tout. Après on peut tirer avantage de ce couplage faible pour simplifier des tests unitaires, pour créer une factory, un locator, etc…

Le Hello World de l'injection de dépendances

On comprend toujours mieux avec un exemple. En voici un dont le schéma UML reprend exactement le schéma présenté plus haut :



Il s'agit d'une application Console. La méthode `Main` joue le rôle du `Builder`. C'est ici que sont créées d'une part l'instance de la classe cliente (*Client Class* dans le premier schéma, "*Salutation*" ici) et d'autre part l'instance du service (`Service1` dans le 1er schéma, "`ConsoleMessageWrite`" ici). La classe cliente

utilisera ici aussi le service au travers d'une interface (`IService1` dans le 1er schéma, "`IMessageWriter`" ici).

Voici le code de la classe `Salutation` :

```
public class Salutation
{
    private readonly IMessageWriter writer;
    public Salutation(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }
        this.writer = writer;
    }
    public void SayHello()
    {
        this.writer.Write("Hello World!");
    }
}
```

Il s'agit de la classe cliente, celle qui utilise le service. On voit qu'elle possède un constructeur acceptant en paramètre l'instance d'une interface, `IMessageWriter`. C'est grâce à ce service que sa méthode "`SayHello`" peut afficher le message.

`SayHello()` *sait quoi écrire mais elle ne sait pas comment le faire et doit utiliser un service qui lui ne sait pas quoi écrire mais qui sait comment le faire.*

L'interface est donc définie comme cela :

```
public interface IMessageWriter
{
    void Write(string message);
}
```

Toute classe supportant `IMessageWriter` devra exposer une méthode `Write(string message)`.

Le service sera donc implémenté dans notre exemple par une classe supportant `IMessageWriter` :

```
public class ConsoleMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

La classe `ConsoleMessageWriter` définit la méthode `Write` de l'interface. Elle prend le message à afficher et se sert de la Console pour l'écrire.

On pourrait maintenant penser à une autre classe `LogMessageWriter` qui n'afficherait pas le message mais l'enregistrerait dans un fichier de Log. La classe cliente "`Salutation`" présentée plus haut n'en saurait rien, c'est le builder qui fournirait l'une ou l'autre version du service… Il y a donc un **couplage faible** entre "`Salutation`" et le service réel `ConsoleMessageWriter`. Et ce grâce à l'interface `IMessageWriter` et la présence d'un *builder* (le `Main()` ici) qui va instancier le service et le passer en paramètre à l'instance de la classe cliente…

Le `Main()` de l'exemple, c'est à dire le `Builder` du premier schéma, ressemble donc à cela :

```
private static void Main()
{
    IMessageWriter writer = new ConsoleMessageWriter();
    var salutation = new Salutation(writer);
    salutation.SayHello();
}
```

Moralité : nous venons de faire de l'injection de dépendances totale et parfaite sans l'ombre d'un framework ou d'un conteneur IoC …

Bien entendu dans la réalité les choses sont un poil plus complexe que dans ce "Hello World" de la DI… Mais les principes restent rigoureusement les mêmes. C'est cette complexification qui peut amener à utiliser des frameworks ou des conteneurs IoC. Mais rien de tout cela n'est obligatoire pour implémenter proprement de l'injection de dépendances.

Xamarin.Forms et les Xamarin.Forms.Labs

Et c'est ici que toutes les pièces du puzzle viennent s'emboiter ! Car maintenant que savons ce qu'est l'injection de dépendances et les conteneurs IoC, en ajoutant les Xamarin.Forms nous ajoutons la glu qui colle ensemble : Windows Phone, iOS, Android et même UWP (encore expérimental quand j'écris ces lignes mais déjç utilisable). Avec les Xamarin.Forms on est dans le monde du cross-plateforme et comme j'ai pu le montrer dans des articles ou des vidéos, la variabilité entre les plateformes impose la technique de l'injection de dépendances pour qui veut écrire un code unique et portable.

Par exemple comment écrire un ViewModel totalement cross-plateforme qui utilise le niveau de charge de la batterie alors que ce genre d'information est par nature native ?

Tout simplement en créant une interface qui retourne la valeur de la charge au sein du projet cross-plateforme et en mettant des classes d'implémentation natives dans chaque projet spécifique à une plateforme. Classes implémentant l'interface et enregistrant dans un conteneur IoC le couple type interface / type classe d'implémentation.

Dès lors le code unique cross-plateforme peut utiliser le service "charge de la batterie" de façon totalement naturelle, au runtime c'est le mécanisme de DI qui fournira qui à UWP, qui à Android la bonne instance de la classe native sachant retourner l'information… On retrouve ici tous les ingrédients que j'avais traité sur dans une vidéo YouTube et portant sur l'injection de code natif dans une application cross-plateforme (utilisant MvvmCross mais la problématique est générique).

Mais ce n'est qu'un exemple.

L'inversion de contrôle (IoC) est partie intégrante des Xamarin.Forms et des Xamarin.Forms.Labs. Utiliser l'injection de dépendance dans ce type d'application est naturel. Savoir utiliser pour votre propre code la puissance de ces outils vous permettra de concevoir un code de meilleure qualité.

Quand on fait appel à `DependencyService` dans une application Xamarin.Forms on utilise en fait un *service locator* intégré au framework. Ce service locator utilise *l'inversion de contrôle*. En réalité l'injection de dépendances est devenue au fil du temps l'un des patterns préférés pour mettre en œuvre l'inversion de contrôle car la DI demande à ce que tout le code respecte les mêmes règles de façon claire. Et comme la clarté est l'un des buts recherchés…

Les Xamarin.Forms.Labs, projet open source qui est au Xamarin.Forms un peu l'équivalent de ce qu'est le toolkit à WPF, apportent de nombreux compléments aux Xamarin.Forms. Parmi les ajouts marquant on trouve un mini framework MVVM très suffisant pour des applications mobiles. Son intégration aux Xamarin.Forms le rend à mon sens plus adapté que Mvvm Light même si j'aime beaucoup ce dernier et son adaptation assez récente à Xamarin.

Xamarin.Forms.Labs et l'injection de dépendance

La première chose à bien comprendre ici est que les Labs fournissent un niveau d'abstraction au-dessus du conteneur DI utilisé ce qui permet d'utiliser différents conteneurs de la même façon.

Cette généralisation est matérialisée par la classe statique `Resolver` :

```
var resolverContainer = new SimpleContainer();

Resolver.SetResolver(resolverContainer.GetResolver());
```

La première ligne créée un conteneur, ici celui fourni par défaut (`SimpleContainer`); la seconde ligne enregistrant ce conteneur comme étant celui que le `Resolver` devra utiliser. A partir de cet instant le `Resolver` est capable d'enregistrer des couples interface / type ou instance de services et bien entendu de les restituer selon les besoins des classes clientes.

L'intérêt des Labs ici est que nulle part dans votre code vous n'aurez à référencer `SimpleContainer` pour utiliser la DI. C'est un niveau d'abstraction supplémentaire, un autre niveau découplage qui participe à la conception d'un code plus maintenable.

On notera que le code ci-dessus se répète quasi à l'identique dans les projets natifs car ce sont eux qui fournissent et instancient les services natifs. Le code central et portable ne faisant usage que de la couche d'abstraction. Il s'avère que le code de `SimpleContener` est le même pour toutes les plateformes mais il pourrait en être autrement. C'est le cas par exemple pour les informations sur l'unité mobile.

Les Labs proposent notamment un objet `Device` bien pratique qui retourne des informations natives. Si l'interface du service est unique, chaque projet spécifique instancie une classe qui là est bien différente car les moyens d'accéder aux informations sont différents selon la plateforme. On retrouve ainsi un `AppleDevice.CurrentDevice`, un `AndroidDevice.CurrentDevice` etc. Chacune de ces classes implémentant l'interface `IDevice`. Par exemple cette dernière est déclarée dans `XLabs.Platform` qui est partagée par tous les projets mais

`AndroidDevice.CurrentDevice` se trouve dans l'assemblage `XLab.Current.Droid` qui est codé spécifiquement pour Android. Et c'est comme cela que le conteneur IoC et la DI deviennent un socle pour créer des applications cross-plateforme !

On notera que les Xamarin.Forms et les XLabs reprennent en réalité des mécanismes qu'on trouvait déjà dans MvvmCross que j'ai largement présenté et sur lequel je m'appuie dans la série de 12 vidéos YouTube présentant le développement cross-plateforme.

L'enregistrement de `IDevice` en utilisant le `SimpleContainer` utilisera la méthode `SetIoc()` de chaque projet natif :

```
private void SetIoc()
{
    var resolverContainer = new SimpleContainer();

    resolverContainer.Register<IDevice>(r => AndroidDevice.CurrentDevice);

    Resolver.SetResolver(resolverContainer.GetResolver());
}
```

L'exemple ci-dessus concerne l'application native Android. Pour Windows Phone c'est la seconde ligne de code qui sera modifiée pour utiliser `WindowsPhone.CurrentDevice` tout simplement (et de façon similaire donc pour iOS ou UWP).

Maintenant que nous possédons un conteneur IoC et que nous y avons enregistré l'instance du service, notre code cross-plateforme peut utiliser sans problème des informations qui lui était pourtant à jamais inaccessibles !

Imaginons un ViewModel de notre application qui a besoin de prendre connaissance des informations de la Device, il suffit maintenant d'ajouter un paramètre de type `IDevice` à son constructeur pour faire de l'injection de dépendance :

```
public class MainViewModel : XLabs.Forms.Mvvm.ViewModel
{
    private readonly IDevice _device;
    private string _message;
```

```
    public MainViewModel(IDevice device)
    {
        _device = device;
        Message = String.Format("Hello XLabs MVVM! your device is a {0}",
        device.Manufacturer);
    }

    public string Message
    {
        get { return _message; }
        set { SetProperty(ref _message, value); }
    }
}
```

Ce ViewModel très simple possède ainsi un constructeur prêt pour l'injection de dépendances, ici une instance du service couvert par l'interface `IDevice`. Ce service est stocké par le ViewModel pour son utilisation ultérieure, c'est une bonne pratique même si ici cela n'est absolument nécessaire puisque le service est utilisé immédiatement pour créer un message de bienvenue indiquant le nom du fabricant de l'unité mobile. Mais les exemples sont très réducteurs par obligation, dans la réalité conserver le service est souvent obligatoire.

La question qui se pose maintenant est de savoir comment l'instance de notre ViewModel va-t-elle pouvoir "recevoir" le fameux paramètre que nous avons ajouté à son constructeur ? C'est-à-dire à quel moment le docteur va-t-il sortir sa seringue pour faire la fameuse « injection » ?

Beaucoup de conteneurs IoC savent analyser par réflexion les différents constructeurs d'une classe pour lui fournir automatiquement les instances des services qu'elle utilise. Mais `SimpleContainer` est… comment dire … Simple ! De fait avec ce dernier il faudra explicitement indiquer au conteneur comment créer l'instance du ViewModel.

Le moyen le plus simple pour y arriver est d'utiliser la méthode `SetIoc()` de chaque plateforme. Directement après l'enregistrement du conteneur nous pouvons ajouter une ligne telle que la 3ème de ce code :

```
private void SetIoc()
{
```

```
    var resolverContainer = new SimpleContainer();

    resolverContainer.Register<IDevice>(r => AndroidDevice.CurrentDevice);
    resolverContainer.Register<MainViewModel>(r => new
MainViewModel(r.Resolve<IDevice>()));

    Resolver.SetResolver(resolverContainer.GetResolver());
}
```

La 3ème ligne enregistre pour le type `MainViewModel` une expression Lambda d'intialisation, celle-ci ne sera invoquée que si on demande au conteneur de retourner une instance de `MainViewModel` (c'est l'avantage d'utiliser une expression d'initialisation plutôt que de passer un "`new servicemachin()`" qui lui instancierait immédiatement un service peut-être jamais utilisé au runtime).

L'expression d'initialisation créée l'instance de `MainViewModel` en passant en paramètre un `Resolve<IDevice>` … et le tour est joué !

Bon, soyons honnêtes l'un des grands bénéfices d'un conteneur IoC est justement de pouvoir gérer les constructeurs et de faire l'injection de dépendances tout seul. Par défaut le `SimpleContainer` est une classe qui n'est qu'un dictionnaire de couples interface / type (ou instance). C'est bien pratique mais on peut faire mieux.

C'est pourquoi les XLabs fournissent un système d'abstraction pour `SImpleContaineur`… car justement ils permettent d'utiliser d'autre conteneurs plus sophistiqués mais avec la même interface.

XLabs et les conteneurs IoC

Comme je viens de le dire `SimpleContaineur` n'est que le conteneur par défaut. Il est simple, ne pèse presque rien et fonctionne rapidement. Mais il est simple.

Et comme je le disais plus haut, si les principes de l'injection de dépendances ne nécessitent pas obligatoirement la présence d'un conteneur IoC ce dernier permet une mise en œuvre plus souple. Toutefois comme il est possible de privilégier tel ou tel aspect il existe plusieurs "visions" de la chose d'où l'existence de multiples conteneurs.

Les XLabs proposent une notion de plugins qui permet d'en étendre le fonctionnement en mode cross-plateforme exactement comme le fait depuis longtemps MvvmCross. On dispose d'une interface partagée et utilisée par le noyau et d'autant d'implémentations spécifiques qu'on a de cibles natives. Les

XLabs sont conçus de cette façon, les Xamarin.Forms aussi, et les plugins pour XLabs ne font que reproduire d'une façon un peu fractale le même principe…

Les XLabs proposent de base certains plugins notamment pour les conteneurs IoC. On dispose ainsi de plusieurs "visions" et on choisitcelle qui correspond le mieux au projet en cours. Pour l'instant, et ce n'est déjà pas mal du tout, les XLabs proposent des plugins pour Autofac, Ninject, TinyIoC et Unity. Si on y ajoute SimpleContainer ce sont 5 conteneurs IoC différents parmi lesquels on peut faire son choix.

Mais comment faire ce choix ?

Disons-le tout de suite ce n'est pas parce qu'il y a du choix que cela justifie de se prendre la tête et qu'il faille utiliser à tout prix des conteneurs différents dans chaque projet !

Bien entendu s'il y a le choix c'est bien parce qu'il y a des nuances dans les services rendus et que c'est par l'analyse de ces derniers que votre choix devra se faire.

L'un des services rendus dont nous avons parlé plus haut et qui peut justifier de choisir un conteneur plus qu'un autre réside par exemple dans la capacité à rendre l'injection de dépendances automatique ou non.

On se rappelle l'exemple où je montrais comment enregistrer une expression lambda d'initialisation dans `SimpleContaineur` au niveau de chaque projet natif pour prendre en charge l'injection de dépendances du service `IDevice` dans le constructeur de `MainViewModel`.

Voici un excellent point de divergence entre un `SimpleContainer`, simple, et d'autres conteneurs IoC qui peuvent apporter une réponse plus automatisée à cette problématique.

Mais il existe d'autres nuances qui peuvent influer le choix d'un conteneur plus que d'un autre. La rapidité d'exécution par exemple. A ce sujet on lira avec intérêt le post de Daniel Palme qui a testé des dizaines de conteneurs. On retiendra outre les performances brutes, la richesse des fonctionnalités qu'il résume dans le tableau suivant (peu lisible, vous devrez zoomer le PDF !) :

	Performance	Configuration			Features				Environment				
Container		**Code**	**XML**	**Auto**	**Autowiring**	**Custom lifetimes**	**Interception**	**Auto diagnostics**	**.NET**	**SL**	**WP7**	**WP8**	**Win RT**
AutoFac	Average	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Caliburn.Micro	Average	Yes	No	No	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes
Catel	Average	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
DryIoc	Fast	Yes	No	No	Yes	Yes	No	No	Yes	No	No	No	No
Dynamo	Fast	Yes	No	Yes	Yes	Yes	No	No	Yes	No	No	No	No
fFastInjector	Fast	Yes	No	No	Yes	No	No	No	Yes	Yes	Yes	No	Yes
Funq	Fast	Yes	No	No	No	No	No	No	Yes	Yes	Yes	No	No
Grace	Fast	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Griffin	Fast	Yes	No	No	Yes	No	Yes	No	Yes	No	No	No	No
HaveBox	Fast	Yes	No	No	Yes	No	Yes	No	Yes	Yes	Yes	Yes	No

Hiro	Fast	Yes	No	No	Yes	No	No	No	Yes	No	No	No	No
IfInjector	Fast	Yes	No	No	Yes	No	No	No	Yes	Yes	Yes	No	Yes
LightCore	Average	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No
LightInject	Fast	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes
LinFu	Slow	Yes	No	No	Yes	No	Yes	No	Yes	No	No	No	No
Maestro	Average	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
MEF	Slow	Yes	No	No	Yes	No	No	No	Yes	Yes	No	No	Yes
MEF2	Fast	Yes	No	No	Yes	No	No	No	Yes	No	Yes	Yes	Yes
MicroSliver	Average	Yes	No	No	Yes	No	No	No	Yes	Yes	No	No	Yes
Mugen	Average	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Munq	Fast	Yes	No	No	Yes	Yes	No	No	Yes	No	Yes	No	No
Ninject	Slow	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	Yes
Petite	Fast	Yes	No	No	No	No	No	No	Yes	No	No	No	No
QuickInject	Fast	Yes	No	Yes	Yes	Yes	No	Yes	Yes	No	No	No	No
SimpleInjector	Fast	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Speedioc	Fast	Yes	No	No	No	No	No	No	Yes	No	No	No	No
Spring.NET	Very slow	No	Yes	No	Yes	No	Yes	No	Yes	No	No	No	No
Stiletto	Average	Yes	No	No	Yes	No	No	No	Yes	Yes	Yes	Yes	No
StructureMap	Average	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes
StyleMVVM	Fast	Yes	No	Yes	Yes	Yes	No	No	Yes	No	No	Yes	No
TinyIoc	Average	Yes	No	No	Yes	Yes	No	No	Yes	Yes	Yes	No	No
Unity	Average	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Windsor	Average	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No

Tous ces conteneurs ne sont pas forcément utilisables sur toutes les plateformes, je vous donne ce tableau uniquement pour que vous puissiez prendre conscience du nombreux incroyable d'implémentations proposées et donc des façons différentes de voir le concept. Pour le reste je vous renvoie à l'article cité en référence qui propose notamment des benchs entre ces différents conteneurs.

Cela permet aussi de remarquer les principales possibilités qu'on prend généralement en compte pour choisir un conteneur. Outre les performances on note par exemple l'autowiring dont je parlais, à savoir la capacité à injecter les dépendances automatiquement ou non. Et cette même possibilité peut être implémentée de façons très différentes (attributs, fichier de configuration XML, règle de nommage avec balayage automatique des assemblages, etc).

Autre critère important, la gestion personnalisée du cycle de vie des services. Lorsqu'on enregistre un service certains conteneurs permettent en effet de préciser s'il s'agira ou non d'un singleton par exemple, ou d'autres mode de gestion du cycle de vie plus complexes. Pour de petites applications simples c'est certainement une considération non primordiale, mais dans certains cas ou dans des applications plus sophistiquées cela peut devenir absolument essentiel…

Les XLabs ne fournissent pas tous les conteneurs du tableau ci-dessus, juste 4 en plus de `SimpleContainer`. Mais il s'agit juste de plugins XLabs c'est à dire d'enveloppes autour de conteneurs existants. Si on désire utiliser autre chose que le conteneur par défaut, outre l'utilisation du plugin XLabs correspondant il faudra bien entendu installer le package Nuget officiel. Ici aussi on conserve un niveau de découplage intéressant puisque les XLabs restent indépendants des versions et des évolutions des conteneurs supportés qui restent sous le contrôle de leurs créateurs respectifs.

Ce principe de couplage faible qu'on retrouve à tous les niveaux montre bien à la fois sa simplicité et la paradoxale richesse qui en découle !

Autofac

Le plugin s'appelle `XLabs.IoC.Autofac`, ce qui n'a rien de surprenant… et il faut l'installer sur toutes les plateformes, ce qui n'a rien d'étonnant non plus puisque nous savons qu'il y a toujours dans ce mécanisme une interface portable accompagnée d'une implémentation spécifique par plateforme.

Vous trouverez toutes les informations sur ce conteneur sur son site officiel autofac.org

Autofac permet d'enregistrer automatiquement par balayage du code source tous les services selon un principe qu'on retrouvait dans MvvmCross. Il permet aussi de faire automatiquement de l'injection de propriétés ou de méthodes.

Bien que ces performances ne soient pas exemplaires (ce qui n'a que fort peu d'impact en général pour des applications mobiles relativement simples il faut l'avouer) c'est un conteneur très complet et très utilisé.

A vous de voir si vous avez besoin de toutes ces sophistications ou si `SimpleContainer` est finalement suffisant…

Ninject

Ninject a été construit pour rester simple et compréhensible mais en offrant des possibilités plus étendues que `SimpleContainer`, notamment l'injection des dépendances automatisées ou en tout cas plus automatisées. Autofac est certainement plus complet sur ce point.

Le site officiel de Ninject vous en apprendra beaucoup plus que quelques lignes ici alors n'hésitez pas à le consulter !

TinyIoC

Tiny est un synonyme de Simple … On reste donc dans le principe d'un conteneur simple n'offrant pas forcément toutes les possibilités de Autofac. Même son site officiel est "tiny" puisqu'il faut se contenter d'un mini Wiki sur Github.

Toutefois l'appellation est trompeuse car TinyIoC propose des choses évoluées comme l'auto-enregistrement des services par exemple ou bien une gestion embryonnaire du cycle de vie en ayant la possibilité d'enregistrer des services multi-instances ou sous forme de singletons.

Là encore chacun fera en fonction de ses besoins, encore faut-il bien connaitre chaque conteneur pour faire un choix éclairé !

Unity

Unity est le conteneur proposé par le groupe Patterns & Practices de Microsoft. Ce n'est pas pour cela qu'il est le meilleur, comme on l'a vu le meilleur est surtout celui qui est le mieux adapté à un projet donné et que vous saurez maitriser sans problème ! Mais forcément c'est un conteneur très utilisé, bien écrit et bien débogué.

Le site officiel de Unity est riche en documentations de tout genre et en faire la lecture ici n'aurait pas d'intérêt, je vous laisse le faire vous-mêmes !

Conclusion

Je m'aperçois qu'il est temps de conclure ce chapitre… Le sujet est si vaste qu'on pourrait écrire des pages et des pages… d'ailleurs il existe des ouvrages uniquement dédiés à l'injection de dépendances.

Mais la DI et les conteneurs IoC ont déjà été abordés dans d'autres articles sur Dot.Blog ce qui m'intéressait ici était de vous présenter ces notions au service du développement cross-plateforme avec Xamarin, les Xamarin.Forms et les Xamarin.Forms.Labs (ou XLabs).

Si on comprend généralement bien ce que Xamarin peut être, si on comprend bien le principe de conteneurs IoC, il reste souvent le plus difficile : comprendre comment tout cela marche ensemble dans un tout cohérent et maitrisable !

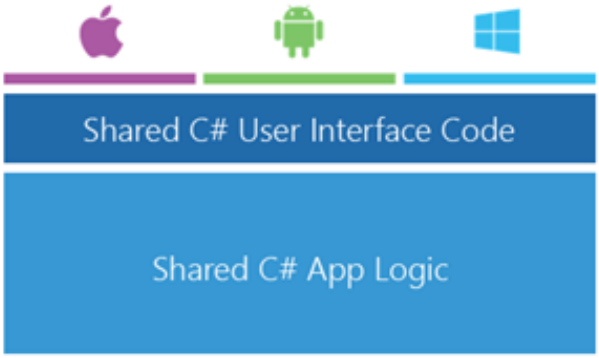
C'est souvent le but de mes articles, celui-ci comme tant d'autres (MvvmCross, Mvvm Light, …), montrer qu'en s'appuyant sur des choses simples on peut construire des logiciels puissants pour peu qu'on sache comment mélanger les ingrédients, ce qui est finalement la partie la plus difficile…

Programmer les XF

Les Bases

Dans un chapitre d'introduction "Xamarin 3.0 entre évolution et révolution" je vous parlais des Xamarin.Forms, une approche totalement cross-plateforme de l'UI. Il est temps de les voir à l'œuvre d'autant que c'estle thème de ce livre !

UI Cross-plateforme



Dans le chapitre cité en introduction je vous ai présenté les Xamarin.Forms, cette nouvelle librairie qui est fournie depuis Xamarin 3.0 (la 4.x est désormais fournie) et qui permet d'écrire des UI totalement portables sous toutes les plateformes mobiles mais pas seulement puisque le code peut tourner sur Mac et bien sûr sur PC…

C'est un pas décisif en avant ! Jusqu'à lors Xamarin apportait la compatibilité du code C# et de la plateforme .NET sous Android et iOS, c'était déjà énorme. Aujourd'hui c'est **l'UI qui devient portable** avec un même code, de Windows Phone 8 et UWP à Android en passant par iOS. Xamarin 4.0 c'est aussi la possibilité de coder en F# en plus de C# (mais pas tous les types de projets, mais Visual Studio se charge déjà d'une partie). On se rappellera d'ailleurs ma récente série de 6 billets sur la programmation fonctionnelle et F# (voir la liste des billets sur F#)

Bien entendu il existe déjà des solutions pour développer en cross-plateforme notamment en C# / Visual Studio. Je vous ai montré celle utilisant Xamarin et MvvmCross (liste des billets sur le cross-plateforme). Je ne parle pas des plateformes hybrides fonctionnant sous Html et JS qui hélas n'ont pas prouvé leur efficacité face au code natif. Mais tout le monde aujourd'hui, même des éditeurs de composants comme Telerik, proposent "leur" solution cross-plateforme qui lave plus blanc et qui sent meilleur. Une telle débauche d'effets pas toujours spéciaux s'explique par le besoin réel et de plus en plus pressant de trouver une solution au problème de la fin de l'hégémonie du PC. Satya Nadella a raison quand il parle d'ère post-PC, et il n'est pas le premier à l'évoquer d'ailleurs.

Aujourd'hui la pression monte autour des entreprises, cette pression vient de l'extérieur, du grand public qui a adopté en masse tablettes et smartphones en délaissant Windows et les PC. La pression est aussi interne : les commerciaux voudraient avoir leur catalogue de produits sur tablettes, les superviseurs aimeraient contrôler les cadences depuis leur smartphone, les clients même réclament des apps natives à la place des sites Web habituels… Toute cette pression qui urge de toute part les DSI d'agir, et toute cette angoisse de se fourvoyer qui les fait différer sans cesse les décisions… A un moment ou un autre il va falloir évacuer cette pression pour que tout n'explose pas… Passer enfin à l'action et rattraper le retard. Ne pas se tromper est gage d'une bonne gestion, mais c'est à l'extrême une apologie de l'immobilisme… Rester en C#, conserver Visual Studio, le framework .NET, tout cela évite déjà les remises en cause, et si en

plus on ajoute la portabilité des UI, alors quelles raisons pourraient encore empêcher les DSI de lâcher la pression ? …

Car si Xamarin nous permettait déjà de rester en natif tout en utilisant notre langage préféré, C#, une plateforme rodée et puissante, .NET et un EDI au-dessus du lot, Visual Studio, il restait tout de même un point gênant, même avec MvvmCross : il fallait recréer les UI pour chaque plateforme.

Certes le gain était déjà énorme, travailler en MVVM avec un code unique pour toutes les plateformes mobiles c'est ce dont nous avons besoin pour relever le défi d'un marché désormais éclaté en plusieurs OS et ce pour toujours certainement. Mais pour géant qu'était le pas le développeur se trainait toujours avec un boulet à la patte… l'UI.

Xamarin.Forms nous affranchit de ce boulet en rendant les UI tout aussi portables que le code.

Avec la V3 le procédé était jeune, les composants utilisables restaient peu nombreux et il n'existait pas de designer visuel (toujours vrai avec la 4), tout se fait par code C# ou XAML. Mais parions que tout cela va évoluer très vite surtout que déjà l'essentiel est là, productif et regroupant le nécessaire pour coder cross-plateforme des UI parfaitement utilisables. Déjà entre cet article et son intégration dans ce livre beaucoup de choses ont évolué.

Rappelons aussi que Xamarin.Forms n'est pas un procédé exclusif de type tout ou rien, dans une même application *on peut mixer sans aucun problème des fiches créées avec Xamarin.Forms et d'autres entièrement écrites en natif*. Cet aspect est essentiel car les Xamarin.Forms, à défaut de permettre pour l'instant de débrider la créativité visuelle des designers n'en reste pas moins un excellent moyen d'accélérer le développement. Par exemple on peut supposer que la page des paramètres, celle du login d'une app peuvent se satisfaire d'une mise en page propre et fonctionnelle sans pour autant nécessiter tout un processus de design couteux et sophistiqué. C'est du temps donc de l'argent gagné. Tout de suite, mais aussi pour l'avenir puisque ces forms seront maintenables sous la forme d'un code unique cross-plateforme pour toute la vie de l'app… (On peut évidemment aussi décider de les changer pour du natif plus tard si on veut !).

Tout en Xamarin.Forms, partiellement en Xamarin.Forms, pour tout de suite, pour plus tard, définitivement, temporairement… Toutes les options sont possibles et vous permettront de produire, enfin, des applications cross-plateformes dans un budget maitrisé car vous pouvez le faire avec vos connaissances actuelles, vos compétences. Le delta à apprendre existe, il ne faut pas le minimiser, mais c'est jute un delta, pas une formation à partir de zéro.

D'ailleurs cette petite série de billets sur les Xamarin.Forms finira j'en suis certain de vous convaincre…

La notion de Form

Cette notion de Form (ou Formulaire en français) est l'une des plus classiques, un des paradigmes de base du codage des UI même depuis les temps anciens. La classe mère des fiches en Delphi s'appelait déjà il y a longtemps TForm – c'était en 1995 à la sortie de Delphi 1.0 ! Plus de 20 ans que ce concept de "fiche" rode dans les langages et dirige nos écrans… De l'écran vert cathodique à l'AMOLED des smartphones haut-de-gamme, tous nécessitent depuis la nuit des temps informatique, celle des Micral, Questar et autre IBM 5250, de pouvoir afficher et saisir des données sous la forme de fiche, skeuomorphisme hérité du temps du papier et des fiches rangées dans leur Rolodex ! Les données sont le sang qui coule dans les veines de tous les ordinateurs sans qui leur existence même n'aurait plus de sens. Encore faut-il les afficher et en autoriser la saisie ou la modification !

Les Xamarin.Forms ne font que nous apporter ce concept de fiche, mais elles le font en étant cross-plateforme. C'est là que réside la nouveauté et l'intérêt.

Chaque page de l'application est ainsi représentée par une fiche, un formulaire, une Form, et l'utilisateur navigue de Form en Form. Certaines Form sont simples (login par exemple) d'autres sont plus complexes (onglets, navigation vers des pages de détail…).

Je sais, des UI cross-plateformes cela existait déjà en 2007 avec la première version de Silverlight, et avec éditeur visuel le tout en vectoriel… Mais le future avance par saccades et même parfois en reculant… pour mieux sauter ? Peut-être avec les Xamarin.Forms !

Un paradigme deux écritures

Annonçant peut-être le Saint Graal de la portabilité totalement pilotée en mode visuel, les Xamarin.Forms peuvent en effet se définir de deux façons très différentes pour un résultat visuel identique :

- Par code C#, en structurant les conteneurs, les contenus, comme on sait le faire par code en XAML ou même avec les bonnes vieilles Windows Forms;
- En code XAML : Xamarin.Forms n'est hélas pas un portage total de XAML, je vendrais mon âme au premier diable qui passe pour qu'il en soit ainsi, non mais elles supportent astucieusement un mode de définition XML avec un sous-ensemble des possibilités de XAML ce qui ouvre la voie à un prochain

éditeur visuel peut-être… Une sorte de retour de Silverlight et la boucle sera bouclée…

Nous verrons dans cette série comment utiliser ces deux modes de création de fiches même si nous allons commencer par la voie la plus simple, la plus directe : **créer une fiche en mode C# avec Xamarin.Studio**. La même chose peut être faite avec Visual Studio et c'est lui que nous utiliserons plus tard d'ailleurs. Mais il est intéressant de rappeler l'existence de Xamarin.Studio qui évolue sans cesse et surtout qui est lui-même portable sur PC, Mac et Linux…

Un premier exemple avec Xamarin Studio

Avant de nous lancer dans l'étude plus approfondie des Xamarin.Forms je pense que réaliser un premier exemple très simple permettra de mieux fixer l'ensemble du processus, comment tout s'enchaine naturellement.

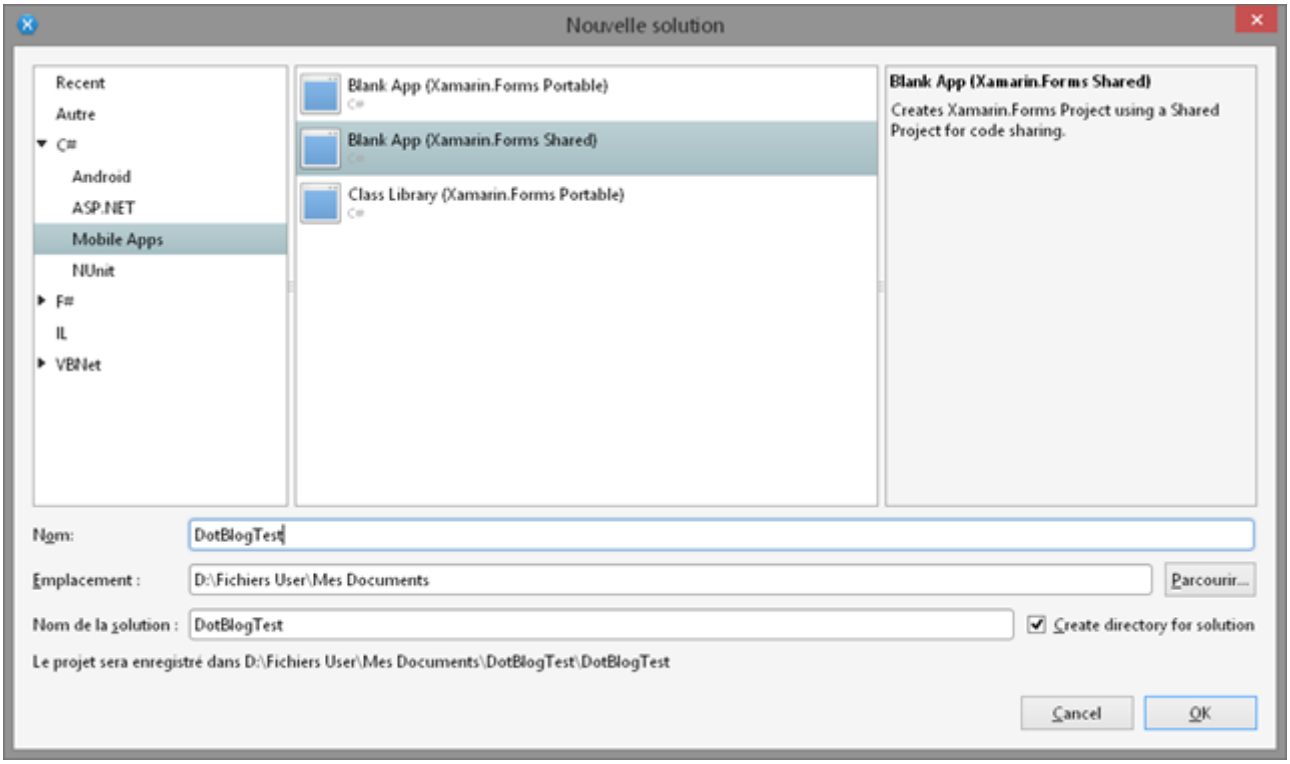
Comme je le disais plus haut je vais utiliser ici Xamarin.Studio qui a l'avantage d'être un environnement portable toujours identique sur toutes les plateformes. Vous pouvez donc écrire des applications Mac et iOS avec cet outil sans jamais voir un PC si cela vous chante. Plus tard nous utiliserons Visual Studio qui reste le meilleur EDI du marché pour PC.

Créer un projet

Fichier / Nouveau Projet, c'est un grand classique qui ne dépaysera personne… Suit une boîte de dialogue à la Visual Studio pour sélectionner le langage et le type de projet.

Ici je vais opter pour une "Blank App (Xamarin.Forms Shared)" c'est à dire une application vide, cross-plateforme, utilisant la méthode du code partagé. L'autre choix utilise les PCL que nous avons vu très souvent dans mes vidéos sur le cross-plateforme. On peut choisir l'un ou l'autre de ces deux modes, le résultat sera le même mais la façon d'y arriver diffèrera un peu, code partagé et PCL offrant chacun des possibilités légèrement différentes.

Il existe un troisième type de projet, classique lui-aussi, celui permettant de créer une bibliothèque de code (une DLL).



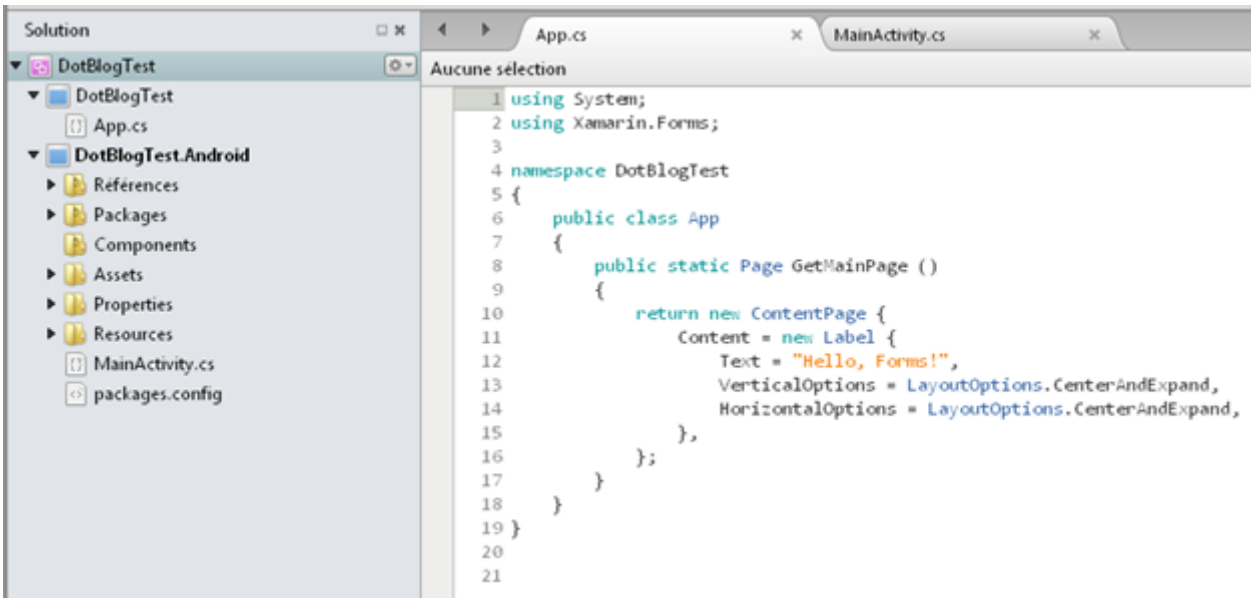
Par défaut je vais obtenir une solution contenant un projet pour le code partagé et un autre pour Android. Pourquoi un projet Android ? D'abord parce sous Xamarin.Studio Android dispose d'un système complet, Xamarin ne s'est pas amusé à refaire les éditeurs Windows Phone qui se trouvent dans Visual Studio. Donc tout naturellement c'est plutôt un projet que l'EDI est capable de gérer qui est créé par défaut.

Vient ensuite la question piège : puisque tout est portable, le code depuis longtemps et maintenant les fiches avec Xamarin.Forms, alors pourquoi donc commencer avec un projet spécialisé pour Android ? La réponse est tout aussi simple : parce que d'une part on pourra ajouter ensuite tous les projets qu'on veut, la solution de départ n'en contient qu'un seul c'est un exemple et que, d'autre part, si presque tout peut être partagé il reste parfois des personnalisations à effectuer selon la plateforme. Avoir un projet pour chacune permet ainsi de s'assurer de pouvoir faire ces petites personnalisations si nécessaire sans tout casser… Une autre raison s'impose aussi : chaque cible nécessite son propre compilateur, ses propres préférences et autorisations etc. Il faudra attendre très longtemps je pense pour que Apple, Google et Microsoft se mettent d'accord sur un langage et une plateforme interopérable ! De fait c'est Xamarin qui permet de centraliser toutes les versions et de les compiler.

Alors pourquoi un projet Android ? J'ai déjà expliqué que l'EDI Xamarin Studio ne sachant pas gérer avec la même aisance les projets Windows Phone il est normal que le projet par défaut soit en Android qui est totalement couvert par l'EDI. Oui

mais pourquoi pas de projet iOS alors ? Cette fois-ci ce n'est pas de la faute de Microsoft mais de Apple qui ne s'est jamais ouvert au monde PC. De fait compiler des projets iOS réclame d'avoir un Mac. Idem pour le débogue. On voit bien que dès lors le fameux pince-mi pince-moi à trois partenaires n'en laisse qu'un seul de véritablement portable et universel : Android tant pis pour les autres…

Bref après avoir répondu à votre saine et naturelle curiosité revenons à notre Solution… (c'est vrai, après on dira que c'est moi qui aime faire des digressions, que nenni, je ne fais que répondre par avance aux questions légitimes du lecteur !)



Le premier projet est celui du code partagé, il ne contient que "`app.cs`" qui est le code de la classe `App`, celle qui démarre et représente l'application. Par défaut elle contient une méthode `GetMainPage` qui retourne un objet `ContentPage` construit par code. Le projet Android, dans son activité principale (notion propre à Android) initialise son affichage en appelant la méthode `GetMainPage`. Si nous compilons en l'état nous aurons ainsi une application Android affichant glorieusement une seule page avec le texte "Hello, Forms!". Le principe global reste le même pour toutes les plateformes : le code de l'UI comme du reste est centralisé dans une PCL ou un projet partagé le tout étant utilisé par tous les projets spécifiques. La nuance avec ce que nous avons pu voir dans mon cycle de vidéos sur le cross-plateforme c'est qu'ici les projets spécifiques peuvent fort bien n'être que des coquilles dans lesquelles on n'intervient jamais, tout était en mode cross-plateforme le code et l'UI alors qu'en utilisant MvvmCross on code les UI en natif, seul le code est partagé.

Les deux approches ont leurs avantages. Celle que nous propose Xamarin est très directe et permet de produire vite un code unique cross-plateforme.

Et c'est énorme !

Nous allons le vérifier tout de suite en faisant évoluer un tout petit peu cette solution de base vide.

Ecrire un modèle de données

A la base de tout logiciel comme je le disais on trouve des données. Il nous en faut pour avancer. Dans l'application partagée créons un sous-répertoire "Model" et plaçons-y une nouvelle classe décrivant (sommairement) un oiseau : Nom vernaculaire, nom latin, poids moyen et description.

Nous obtenons très rapidement un code de ce type :

```
namespace DotBlogTest
{
    public class Bird
    {
        public string Name { get; set; }
        public string LatinName { get; set; }
        public int AverageWeight { get; set; }
        public string Description { get; set; }
    }
}
```

Pour cet exemple nous ferons bien entendu abstraction de toute la logique habituelle du INPC et des tests de validité des données.

Ecrire le ViewModel

Il n'est pas question de sacrifier aux bonnes habitudes. Nous simplifions les tests mais pas l'essentiel ! Nous allons ainsi ajouter un répertoire "ViewModel" au projet partagé et y placer le code du ViewModel de notre page principale.

Interlude : <musique d'ascenseur> … <ding dong ding> … <voix d'aéroport>En raison d'un incident technique les passagers à destination de Xamarin.Studio sont priés de se rendre porte Visual Studio Hall Microsoft<ding dong ding>

Je n'ai pas trop le temps de chercher à comprendre mais sous Xamarin Studio la compilation indique un problème d'espace de nom dans mon projet partagé… Et quand j'ouvre la même solution sous VS 2013 tout passe bien alors même que c'est le compilateur Xamarin qui est appelé aussi… Donc je switche sauvagement sous VS pour la fin de l'article, comme ça vous aurez vu un peu des deux dans le même projet ce qui prouve même la compatibilité totale entre les deux EDI (soyons positifs) !

Donc revenons à notre ViewModel. C'est un code rudimentaire mais fonctionnel :

```
namespace DotBlogTest.ViewModel
{
    public class BirdsViewModel
    {
        public ObservableCollection<Bird> Birds { get; private set; }

        public BirdsViewModel ()
        {
            Birds = new ObservableCollection<Bird> ();
            Birds.Add (new Bird {
                Name = "Rouge gorge",
                LatinName = "Erithacus rubecula",
                AverageWeight = 20,
                Description = "Petit passereau familier des jardins"
            });
            Birds.Add (new Bird {
                Name = "Mésange huppée",
                LatinName = "Lophophanes cristatus",
                AverageWeight = 11,
                Description = "Petit passereau de la famille des Paridés"
            });
            Birds.Add (new Bird {
                Name = "Pic vert",
                LatinName = "Picus viridis",
                AverageWeight = 200,
                Description = "Présent presque partout en Europe fait son
nid en creusant un arbre"
            });
            Birds.Add (new Bird {
                Name = "Chardonneret élégant",
                LatinName = "Carduelis carduelis",
                AverageWeight = 15,
                Description = "Oiseau partiellement migrateur à la robe
bariolée et exclusivement granivore"
            });

        }
    }
}
```

Le ViewModel est une classe "normale". Ici pas de chichi, pas de framework MVVM super sophistiqué, on fait tout à la main, et ça marche quand même ! Le lecteur fidèle n'est pas étonné puisque dans le passé je vous ai déjà expliqué comment appliquer MVVM sans aucun framework (le 9 janvier 2010 dans l'article M-V-VM avec Silverlight ce qu'on retrouve revisité dans le livre ALL.DOT.BLOG "Méthodes & Frameworks MVVM").

Ce code est ultra simple et le ViewModel ne contient qu'une seule propriété de type collection d'oiseaux (classe Bird), 95% du code restant est consacré à l'initialisation de cette collection avec quelques items de démonstration.

La page principale

Nous avons une classe Bird, le Model, nous avons une classe BirdViewModel, … le ViewModel (il y en a deux qui suivent ça fait plaisir !) qui initialise une liste d'oiseaux pour la démonstration. Ce n'est déjà pas mal mais c'est du C# pour débutant. Il nous faut afficher cette liste d'oiseaux sur tous les mobiles de la planète et pour cela nous avons besoin d'une fiche. Donc d'une page de contenu de Xamarin.Forms.

Cette page s'appelle BirdsPage sans grande originalité :

```
public class BirdsPage : ContentPage
{
  public BirdsPage ()
 {
   Title = "Oiseaux";
   var list = new ListView();
   Content = list; ...
  }
}
```

Pour l'instant ne dévoilons pas toute la mécanique pour regarder l'allure générale. Tout d'abord la classe créée descend de `ContentPage` fournie par Xamarin.Forms. C'est l'un des styles disponibles, le plus simple avec un objet conteneur qui remplit tout l'espace. Nous aurions pu choisir une `MasterDetailPage`, une `NavigationPage`, une `TabbedPage` ou une `CarouselPage` par exemple, le choix est vaste et permet de cerner la majorité des mises en page d'une application.

Ici pour faire simple c'est donc `ContentPage` qui a été choisie. Le code ne contient que le constructeur de la page car tout y est : l'initialisation du titre de la page et celle de sa propriété `Content`, le contenu unique de la page. Ici ce contenu (qui peut être n'importe quoi un peu comme un Border XAML) est une liste et plus précisément une `ListView` fournie elle aussi par Xamarin.Forms.

Cette classe, comme d'autres dans les Xamarin.Forms apporte quelque chose d'énorme : le *data binding* qui devient, de fait, cross-plateforme lui aussi.

Comme on peut se l'imaginer le but de cette classe est de fournir un support visuel de type liste, ce qui tombe bien puisque nous voulons afficher la liste des oiseaux.

Toutefois on s'aperçoit vite que mon code est un peu trop court pour être honnête… En effet créer une `ListView` c'est bien mais encore faut-il lui expliquer quoi afficher… C'est simple, nous allons créer une instance du `ViewModel` et passer à l'`ItemsSource` de la `ListView` la propriété `Birds` de ce dernier (la collection). Du MVVM assez classique bien que tout "en mode manuel".

Si nous lançons l'app tout de suite et comme s'y attendent les plus attentifs nous aurons bien une liste mais d'éléments répétant inlassablement le nom de la classe de l'objet oiseau (`Bird`) qui sera passé… Il manque à notre `ListView` un `ItemTemplate` ou un équivalent pour savoir comment afficher chaque élément de la liste.

Nous allons ainsi créer maintenant une instance de la classe `DataTemplate` pour le type "`TextCell`", une "cellule" qui ne contient que du texte sur deux lignes, une avec une police assez grosse, le texte, et une seconde ligne plus petite, le détail, mise en page assez classique dans les listes. C'est déjà tout fabriqué il suffit donc d'initialiser les deux champs (`Text` et `Detail`) en utilisant les noms des propriétés de `Bird` (`Name` et `Description`).

Enfin nous indiquons à liste que son ItemTemplate est le DataTemplate qui vient d'être créé… C'est facile, c'est, aux noms de classes qui peuvent légèrement différer, le même code que ce que nous écririons en C# pour créer "à la main" une petite page XAML avec un conteneur, une liste et un data template.

Le code complet devient donc :

```
using DotBlogTest.Model;
using DotBlogTest.ViewModel;
using Xamarin.Forms;

namespace DotBlogTest.Form
{
    public class BirdsPage : ContentPage
    {
        public BirdsPage ()
        {
            Title = "Oiseaux";
            var list = new ListView
               {ItemsSource = new BirdsViewModel().Birds};
            var cell = new DataTemplate (typeof(TextCell));
            cell.SetBinding (TextCell.TextProperty, "Name");
            cell.SetBinding (TextCell.DetailProperty, "Description");
            list.ItemTemplate = cell;
```

```
            Content = list;
        }
    }
}
```

C'est effectivement très simple. Pour l'instant notre liste n'est pas interactive mais elle fonctionne. Reste pour lancer un test à modifier la méthode `GetMainPage` de "`app.cs`" du projet partagé en supprimant le code de démonstration Xamarin et en demandant la création de notre page liste des oiseaux, `BirdsPage`. Comme nous souhaitons améliorer ensuite l'app pour ajouter l'affichage du détail nous aurons besoin du système de navigation, la page sera retournée dans une "enveloppe navigationnelle". Mais c'est tellement plus simple quand on regarde le code :

```
using System;
using DotBlogTest.Form;
using Xamarin.Forms;


namespace DotBlogTest
{
    public class App
    {
        public static Page GetMainPage ()
        {
            var birds = new BirdsPage();
            return new NavigationPage (birds);
        }
    }
}
```

J'avais dit que c'était simple, je ne mentais pas. On crée une instance de `BirdsPage` et on la retourne enveloppée dans un objet `NavigationPage`. La navigation peut être sophistiquée, ici nous n'écrirons rien d'autre que cela et utiliserons les mécanismes par défaut. Quand nous aurons ajouté la page détail il suffira à l'utilisateur d'utiliser la touche retour arrière de son appareil pour revenir à la liste. C'est aussi simple que cela.

Voici à quoi cela ressemble maintenant :

C'est déjà assez sympa puisque ça marche… et que ce code il faut le redire est uniquement du code générique qui tournera tel quel sur toutes les cibles ! D'ailleurs à aucun moment nous n'avons touché au projet Android qui sert de support à l'article et nous n'y toucherons jamais, vraiment jamais…. Tout se passe uniquement dans le projet partagé qui, comme je le présentais en début d'article pourrait aussi être un projet PCL. C'est au choix. En fin d'article je vous donne le code source de la solution, à vous d'ajouter le support Windows Phone 8.1 pour vous entrainer (c'est facile il n'y a presque rien à faire) !

Maitre / Détail

Soyons fou ! Imaginons maintenant qu'on veuille que l'utilisateur puisse choisir un oiseau de la liste et que cette action affiche une seconde page avec le détail de la fiche du volatile. C'est moi qui tape le code, alors n'hésitez pas, c'est la maison qui offre ! Ce qui me rappelle un grand principe dans notre métier applicable aussi dans la vie de tous les jours : Rien n'est impossible pour celui qui n'a pas à le faire lui-même.

Pour ajouter ce comportement maitre / détail nous aurons besoin d'une seconde page mais aussi d'ajouter quelque chose pour déclencher la navigation au moment du touch sur l'élément de la liste.

Page détail

La page détail ne sera pas différente de la page principale, nous allons choisir ici aussi une `ContentPage`. Et comme nous le ferions pour du XAML écrit en C# nous allons construire l'imbrication des éléments. Pour rester dans la sobriété nous utiliserons un *StackPanel*, enfin son équivalent Xamarin.Forms, puis nous empilerons à l'intérieur d'autres StackPanel en mode horizontal contenant un titre et un contenu. C'est vraiment de la mise en page de base, mais ça fonctionne là encore. Beaucoup de logiciels ne réclament pas plus que ça pour être utiles et adorés des utilisateurs !

Ce qui nous donne le code suivant :

```
using System;
using System.Collections.Generic;
using System.Text;
using DotBlogTest.Model;
using Xamarin.Forms;

namespace DotBlogTest.Form
{
    public class DetailPage : ContentPage
    {
        public DetailPage(Bird bird)
        {
            Title = bird.Name;
            var stack = new StackLayout {Orientation =
                                         StackOrientation.Vertical};
            stack.Children.Add(AddRow("Nom: ",bird.Name));
            stack.Children.Add(AddRow("Nom Latin: ",bird.LatinName));
            stack.Children.Add(AddRow("Poids moyen: ",
                               bird.AverageWeight.ToString()));
            stack.Children.Add(new Label{Text = bird.Description});
      //var details = new Label {Text = bird.Name + " " + bird.LatinName};
            Content = new ScrollView {Padding = 20, Content = stack};
        }

        private Layout<View> AddRow(string label, string content)
        {
```

```
            var s = new StackLayout {Orientation =
                                     StackOrientation.Horizontal};
            s.Children.Add(new Label{Text = label,TextColor = Color.Gray});
            s.Children.Add(new Label{Text = content,
                                     TextColor = Color.White});
            return s;
        }
    }
}
```

La classe `DetailPage` descend ainsi de `ContentPage` comme la page principale et seul son constructeur est utilisé pour l'initialiser. Mais cette fois-ci nous utiliserons un constructeur à un paramètre : l'instance de l'oiseau à afficher.

Le reste est juste verbeux et répétitif : on initialise le titre de la page en utilisant le nom de l'oiseau, on crée un `StackLayout` équivalent du StackPanel, et on lui ajoute des enfants en utilisant une méthode "`AddRow`" qui retourne un StackLayout horizontal avec un titre et un texte. La dernière ligne ajoutée au StackLayout est la description de la classe Bird qui prendra ainsi tout le reste de la page.

Ici nous ne prévoyons pas que le contenu dépasse la hauteur de l'écran, c'est un peu un tort car dans la réalité on ne sait pas de quelle résolution disposera l'utilisateur. Dans la page principale tout est géré par la ListView mais ici il faudrait ajouter un conteneur général de type ScrollView pour bien faire. Cela tombe bien car Xamarin.Forms possède une classe de ce nom… je vous laisse l'ajouter pour vous entraîner…

Une fois le type de page choisi, Xamarin.Forms met à notre disposition des conteneurs de mise en page différents (ci-dessous). Le StackLayout ou le ScrollView en sont une illustration. Si la page elle-même ne peut être que de l'un des types proposés, la mise en page peut être beaucoup plus complexe et les conteneurs imbriqués les uns dans les autres, comme en XAML.

Notre page secondaire est maintenant opérationnelle ne reste plus qu'à déclencher son affichage…

Détecter le touch et naviguer

Pour naviguer depuis la page principale il faut intervenir sur la ListView et capturer le touch. Ensuite il faut récupérer l'item courant, une instance de Bird, et demander enfin la navigation sur une nouvelle instance de la page secondaire en lui passant en paramètre l'oiseau sélectionné. Facile.

En fin de code de la création de la page principale nous ajoutons :

```
list.ItemTapped += (sender, args) =>
                                {
                                    var bird = args.Item as Bird;
                                    if (bird == null) return;
                                    Navigation.PushAsync(
                                           new DetailPage(bird));
                                    list.SelectedItem = null;
                                };
```

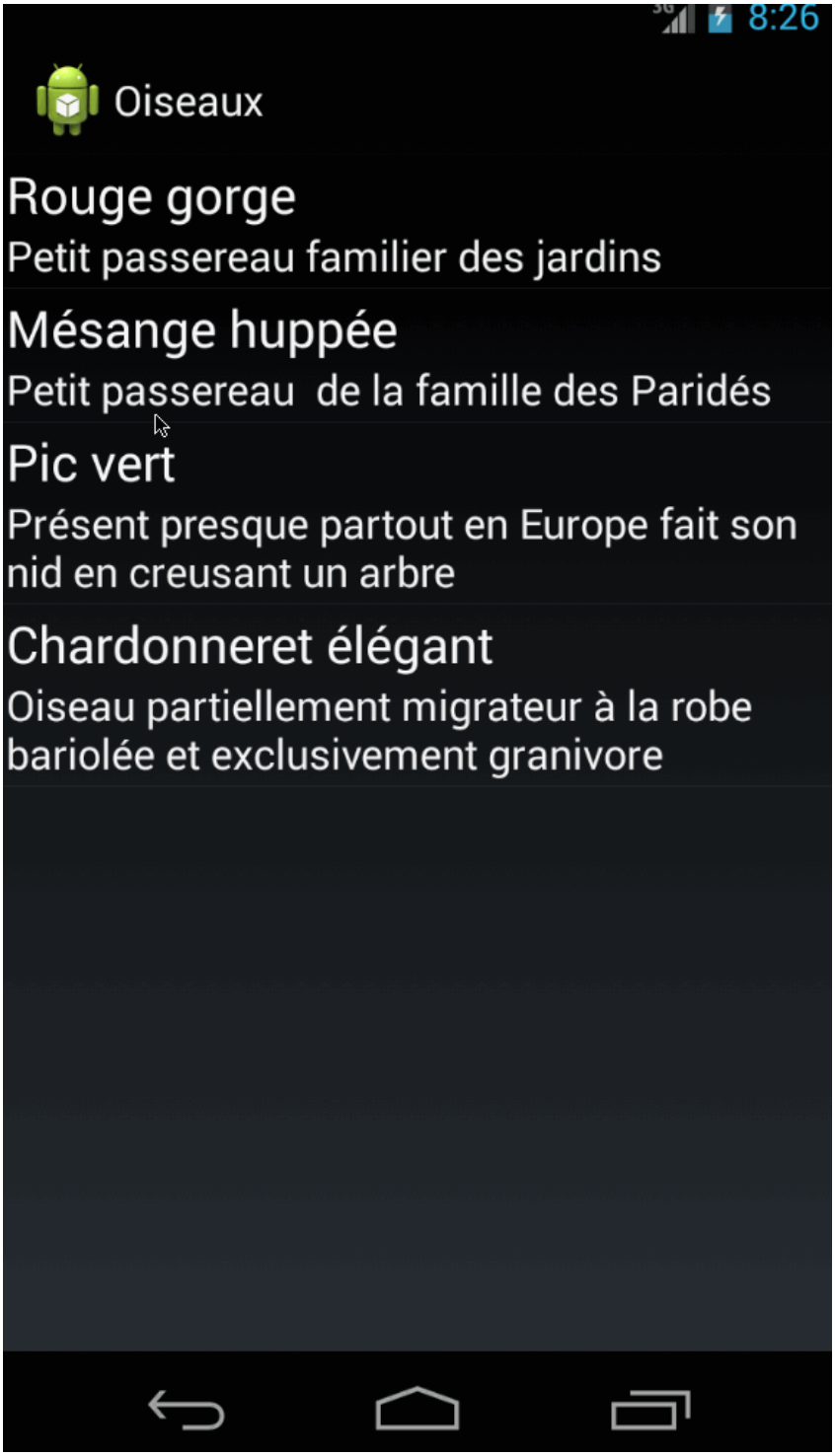
L'évènement `ItemTapped` correspond au touch ou au clic. On le définit en utilisant une expression Lambda (ceux qui ont suivi la série sur F# ces derniers jours savent mieux de quoi je veux parler !). L'expression transtype l'item retourné dans les arguments de l'évènement, si la valeur est nulle il ne se passe rien, si elle est non nulle on appelle le `PushAsync` de l'objet `Navigation` avec une nouvelle instance de la page détail dont le paramètre de création est l'item. Comme on peut l'imaginer cette action "empile" la page dans le système de navigation. Pour le confort de l'utilisateur et étant donné l'ergonomie ultra simplifiée de notre app nous annulons la sélection qu'il vient de faire en passant l'item sélectionné de la liste à null.

Let's go !

Même si c'est difficile à croire ou à réaliser, l'application est terminée et elle fonctionne… Une application totalement portable dans tous les environnements sans en changer une seule ligne de code …

Pour cet article je n'ai implémenté que l'application Android, comme je fourni le code source du projet, amusez-vous à ajouter l'application Windows Phones, Mac, iOS … Lâchez-vous c'est fait pour ça !

Le GIF ci-dessous vous montre l'application en cours d'utilisation avec la sélection d'un oiseau dans la liste et l'affichage de son détail (à voir sur Dot.Blog, sur papier forcément ça ne s'anime pas…).



Conclusion

Simple, fonctionnel, portable.

Que demande le peuple ? Rien de plus.

Que demande vos commerciaux, votre patron, vos utilisateurs ? Rien de plus.

Combien de temps pour une application totalement portable cross-plateforme avec gestion de la navigation et du maître / détail ? Quelques minutes et plusieurs heures pour l'article ! Et pour une poignée de minutes en plus on aurait pu ajouter la sauvegarde des données et la saisie par l'utilisateur. Mais nous verrons ça autrement dans d'autres articles sur les Xamarin.Forms…

Tout le monde n'est pas éditeur de jeu ou d'applications qui remplacent un site web institutionnel. Il y a aussi la foule d'entreprises, petites ou plus grandes, qui ont besoin rapidement d'intégrer les mobiles dans leur environnement. Tout le monde n'a pas vocation à placer des apps sur les Stores, au contraire, la majorité des applications d'entreprise sont réservées à un usage interne.

Avec les Xamarin.Forms la panoplie est complète pour créer rapidement des applications professionnelles pour toutes les plateformes.

Avec quelques efforts en plus on peut les designers pour leur donner un "look" et en faire des applications destinées aux clients par exemple. Et avec un peu plus de travail on peut en faire des apps commerciales placées sur les Stores…

Aujourd'hui c'est là, sur la table, c'est possible, pas très cher.

A vous de voir s'il est nécessaire d'inventer de nouvelles excuses pour ne pas vous lancer…

PS : comme promis le code de la solution (VS 2013 avec tous les patchs et bien entendu Xamarin 3.x) :

Solution Test

XAML devient Cross-Plateforme!

Dans l'introduction publiée hier je vous ai montré la simplicité et la puissance des Xamarin.Forms, mais il y a encore plus fort : elles peuvent être définies en XAML !

XAML ?

On ne présente plus XAML, langage de description graphique absolument merveilleux et vectoriel s'adaptant à toutes les résolutions et fonctionnant sur tous les PC depuis XP jusqu'à Windows 10 en passant par Windows Phone et UWP. Pendant un temps, celui de Silverlight, ce langage est même devenu universel, fonctionnant sur toutes les plateformes.

XAML suit un formalisme XML pour décrire un objet visuel, qu'il s'agisse d'une page WinRT, d'un User Control Silverlight ou WPF, ou d'un template d'affichage de données sous Windows Phone. Dans tous ces cas de figure XAML est le même, vectoriel, avec il est vrai quelques nuances de ci de là. Par exemple le XAML de

WPF supporte la 3D, c'est le seul. Silverlight pour le Web ou celui utilisé aujourd'hui pour développer sous Windows Phone n'est qu'un sous ensemble de WPF, par exemple la coercition de valeurs pour les propriétés de dépendance n'existe pas. Bref XAML est à la fois un et multiple mais quand on sait s'en servir on se retrouve toujours en terrain connu peu importe l'implémentation.

XAML c'est aussi un éditeur visuel surpuissant, Blend, anciennement Expression Blend. Depuis quelques temps le designer visuel de Visual Studio est presque du même niveau. Presque seulement. Données de test, conception de templates, etc, tout cela réclame Blend pour le faire visuellement.

Alors de quel XAML parlons-nous dans Xamarin.Forms sachant que ces dernières ne sont qu'une façade qui est compilée en natif sur chaque plateforme ? Etant donné que ni Android ni iOS n'offrent de plateforme vectorielle, le XAML dont nous parlons pourrait sembler très différents de celui auquel nous avons affaire habituellement.

En réalité le XAML des Xamarin.Forms est un sous-ensemble de XAML, comme celui de Silverlight ou Windows Phone. Ni plus ni moins. Le code s'exprime de la même façon en utilisant un jeu de contrôles particulier comme on utiliserait une librairie de chez Telerik par exemple. Bien entendu ce sous-ensemble de XAML ne possède pas les primitives graphiques qui obligeraient la présente d'un moteur vectoriel de rendu. Si on doit placer de l'imagerie on utilisera des PNG par exemple plutôt que des User Control construits graphiquement en courbes de Béziers. C'est là qu'est toute la différence.

Car pour le reste le XAML des Xamarin.Forms est en tout point semblable au XAML traditionnel, il en est vraiment très proche. Il ne lui manque que la parole. Enfin la conception visuelle je veux dire… Car en effet pour l'instant c'est un XAML qu'il faut écrire à la main.

Même si rien n'a été annoncé en ce sens par Xamarin je suppose qu'un jour prochain le support d'un designer visuel sera proposé. Cette excellente idée ne peut s'arrêter là. Ce n'est que la première version et la proximité du XAML Xamarin et de celui de Microsoft est tel que tout le monde pense tout de suite à utiliser VS ou Blend. Xamarin y pense déjà certainement aussi. D'autant que concevoir des designers visuels n'est pas une nouveauté pour eux qui ont créé ceux pour Android et iOS dans Xamarin.Studio et en plugin pour VS ! Si on fait abstraction du vectoriel justement et de tout ce que cela implique, le XAML de Xamarin.Forms n'est pas très différent du XML de description Android. Or Xamarin possède déjà un designer pour ce dernier… L'avenir pourrait être passionnant… En attendant le XAML Xamarin s'écrit à la main…

UI en XAML ou C# ?

Xamarin.Forms offre les deux approches donc. On peut comme je l'ai fait voir dans l'article précédent décrire les pages totalement en C#, c'est simple, compréhensible et on n'utilise qu'une seule compétence. Ce mode peut s'avérer parfait pour ceux qui ne se sentent pas très à l'aise avec XAML. Et puis on peut aussi, comme nous allons le voir aujourd'hui, décrire les pages en XAML.

Sachant que pour l'instant en tout cas il n'y a pas de designer visuel pour le XAML de Xamarin.Forms on peut se demander quel est l'intérêt d'écrire les UI en XAML plutôt qu'en C#.

La question est légitime. Une partie de la réponse a été donnée plus haut : pour ceux qui ne sont pas à l'aise avec XAML faire tout en C# est une option parfaitement valable. En revanche pour ceux qui connaissent XAML ils trouveront là un moyen plus moderne de travailler. Car XAML est un langage descriptif à la différence de C# qui est un langage impératif… J'ai abordé récemment dans les articles sur F# la différence d'approche entre langage fonctionnel et langage impératif ou orienté objet. On retrouve les mêmes nuances avec les langages descriptifs comme XAML comparés aux autres langages.

En XAML on décrit le visuel. On explique uniquement ce qu'on veut voir. 100% du code sert à définir l'objet visuel traité.

En C# 80% du code est mécanique et consiste à appeler des constructeurs, à passer des paramètres, à créer des listes, des objets qu'on imbrique dans d'autres. Bref à faire de la plomberie POO. 20% seulement consiste à donner une valeur à une couleur, à fixer numériquement un padding, etc.

Comme je le disais les deux approches peuvent sembler similaires car toutes les deux pour l'instant se codent à la main.

Mais entre du C# et du XAML il existe une différence essentielle, comme entre du C# et du F#.

Maintenir, c'est à dire en correctif tant qu'en évolutif, du XAML pour des objets visuels est plus intéressant que de maintenir un équivalent en C#, c'est tout simplement mieux adapté.

Alors que choisir ?

C'est bien simple, tant qu'un éditeur visuel n'existe pas pour le XAML Xamarin, ce qui marquerait un avantage décisif pour ce dernier, il y autant d'avantages à choisir XAML que C# pour décrire les UI. A vous d'utiliser l'approche qui tout bêtement vous semblera la plus agréable, la mieux adaptée à votre besoin… Je vais même aller plus loin : mixez les deux approches dans vos applications en fonction de la

nature des pages ! En effet des pages très visuelles avec des data template ou autres gagneront en lisibilité et en maintenabilité à être écrites en XAML. En revanche les pages dynamiques avec du contenu qui s'adapte aux données sera du coup bien plus simple à maintenir et à créer en utilisant directement C# !

Xamarin.Forms est un framework solide car il permet tous les "décrochages" et le mélange des approches sans mettre en péril tout l'édifice et se retrouver coincé. Une page est plus simple en C#, codez là en C#, une page est plus simple en XAML, utilisez ce dernier, une page est plus facile à coder en natif et bien codez là en pur natif, etc… On peut ainsi jongler dans une même application entre des approches de l'UI totalement différentes mais convergentes grâce à l'intelligence de l'outil.

Côte à côte

Comment mieux sentir la différence des approches XAML et C# pour coder les UI qu'en les plaçant côte à côte ?

C'est ce que je vous propose ici en reprenant l'exemple présenté dans l'article précédent et en remplaçant toutes les fiches codées en C# par des fiches identiques codées en XAML !

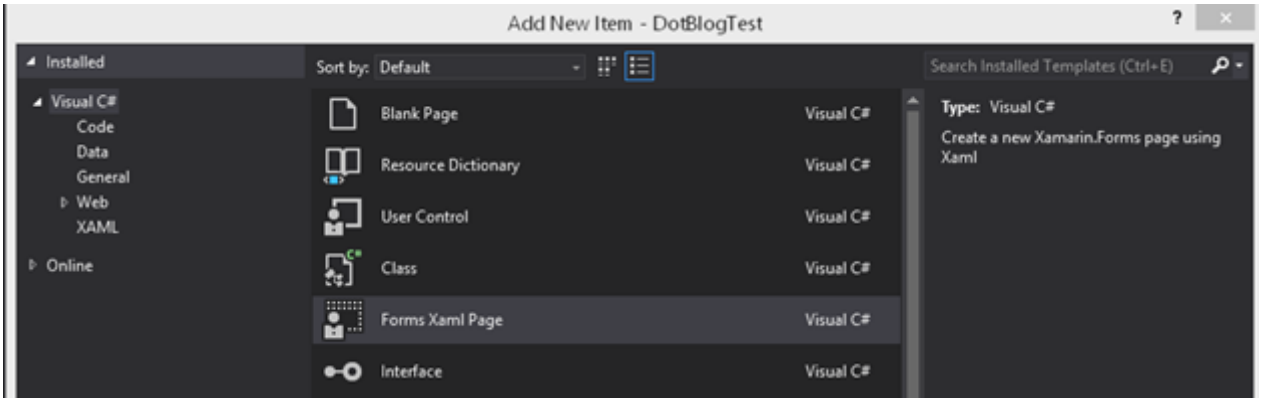
En partant d'une copie du projet d'hier (un vrai copier / coller de tout le répertoire de la solution) nous allons voir comment chaque fiche, chaque évènement, chaque binding, chaque template peut s'exprimer à l'identique en XAML.

Attention : dans cette manipulation je vais conserver une vraie copie, donc les mêmes noms de sous-répertoire, de fichiers, même les namespaces, les classes, tout cela sera identique. Seul le répertoire de plus haut niveau sera renommé avec un suffixe "XAML". Il est donc essentiel de ne pas se mélanger les pinceaux électroniques pour éviter d'effacer ou de modifier des fichiers dans la "mauvaise" solution dans le cas où vous auriez sur votre disque le projet d'hier et celui d'aujourd'hui !

La fiche principale (liste)

Dans sa version originale il s'agissait du fichier de code "`BirdsPage.cs`". Une classe de type `ContentPage` était créée avec une liste et son template plus le code de navigation sur la sélection d'un oiseau.

Dans sa version XAML, après avoir supprimé "`BirdsPage.cs`" (je pars vraiment d'un copier/coller de la solution d'hier), je rajoute dans le même sous-répertoire "`Form`" un nouvel item. Dans la liste que Visual Studio ouvre on trouve "*Form Xaml Page*" :



Je vais donc créer une nouvelle Form de Xamarin.Forms que je vais appeler de la même façon, "`BirdsPage`". Comme pour tout code XAML Visual Studio créée en réalité deux fichiers : "`BirdsPage.xaml.cs`" le code-behind, et "`BirdsPage.xaml`" le code XAML.

Bien entendu comme il n'y a pas de designer pour ce type de XAML on obtient immédiatement une exception dans le designer de VS qui s'ouvre, il suffit de le fermer et de ne garder que l'éditeur de code. Ce n'est pas très propre mais pas gênant puisque nous le savons, je l'ai dit plusieurs fois, ce XAML là ne se pratique pour l'instant que par code "à la main" et non pas visuellement.

Le contenu XAML qui vient d'être créé est le suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                        x:Class="DotBlogTest.Form.BirdsPage">
    <Label Text="{Binding MainText}" VerticalOptions="Center"
HorizontalOptions="Center" />
</ContentPage>
```

Il s'agit bien d'une `ContentPage`, comme celle de la version C#. Ce n'est que coïncidence Xamarin.Forms est très fort mais il ne lit pas encore les pensées du développeur. Cette page est presque vide puisqu'elle contient un simple `Label` dont la propriété texte est bindée à une propriété `MainText` qui proviendrait du ViewModel associé. Cela nous indique clairement que ces Forms XAML de Xamarin *supportent le data binding* ! C'est ce qui était merveilleux avec MvvmCross qui ajoutait le databinding dans les projets natifs. Ici c'est "out of the box" ou presque et cela fonctionne partout avec une seule et même syntaxe… Comme je le répète peut-être comme une supplique, il ne manque que le designer visuel pour en faire un XAML universel (j'en tremble rien que d'y penser !).

Bref tout cela est bien sympathique mais nous avons une fiche principale à coder…

En quelques secondes il est facile pour qui connait XAML de taper le code qui suit :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                        x:Class="DotBlogTest.Form.BirdsPage"
Title=”oiseaux”>
    <ListView x:Name="List" ItemSource="{Binding Birds}"
ItempTapped="OnItemsSelected">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding Name}" Detail="{Binding
Description}"/>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

Le Label de démonstration a été supprimé, à sa place j’ai placé un `ListView` qui possède un `ItemTemplate` utilisant un `TextCell` dont les propriétés sont bindées au nom et à la description de l’oiseau sélectionné. Bref exactement la même chose que le code C# original que je vous redonne ci-dessous pour comparaison :

```
public BirdsPage ()
{
    Title = "Oiseaux";
    var list = new ListView {ItemsSource = new BirdsViewModel().Birds};
    var cell = new DataTemplate (typeof(TextCell));
    cell.SetBinding (TextCell.TextProperty, "Name");
    cell.SetBinding (TextCell.DetailProperty, "Description");
    list.ItemTemplate = cell;
    list.ItemTapped += (sender, args) =>
       {
           var bird = args.Item as Bird;
           if (bird == null) return;
           Navigation.PushAsync(new DetailPage(bird));
           list.SelectedItem = null;
           };
        Content = list;
}
```

C’est bien ce que je vous disais, en mode C# on écrit beaucoup de plomberie (créations diverses, binding, etc) alors qu’en XAML on reste purement déclaratif ce qui est bien plus adapté à la description d’une vue.

Toutefois on triche un peu car il manque des petits bouts au code XAML pour faire exactement la même chose. Notamment la gestion du clic que nous avons bindée à un `OnItemSelected` qui n’existe nulle part pour l’instant …

Où placer ce code appartenant à la fiche ? Le gestionnaire de clic devrait être bindé à une commande du ViewModel mais pour l'instant je ne veux pas modifier toute l'application ce qui embrouillerait les choses. Nous ferons fi ici de l'orthodoxie MVVM… Je placerai donc ce code avec l'initialisation de la classe c'est à dire dans le code-behind. Le jeu consiste à remplacer "à l'arrache" les Forms C# par des Forms en XAML. On voit d'ailleurs que l'approche XAML aide à soulever les bonnes questions... Dans le code C# nous avions codé l'évènement comme une Lambda dans la création de l'UI ce qui ne choquait personne… sauf les plus tatillons. Mais ici en utilisant un outil de description visuelle, XAML, nous obligeons à la réflexion sur la place du code non visuel. Et même si pour cet exemple je ne le déplace pas dans le ViewModel c'est bien entendu là qu'il devrait se trouver ! Simuler à iso fonctionnalités est notre seul but dans cet article. Et c'est ce que nous faisons. En C# l'UI traitait l'évènement, en XAML aussi. Cela soulève d'ailleurs une autre question : le navigationnel doit-il être considéré comme un code "normal" ou bien comme faisant partie de l'UI. Si nous avions un XAML complet, nous serions tentés de placer un Behavior pour gérer la navigation. Donc côté UI… Mais tout cela est hors de mon propos du jour, juste quelques éléments pour faire réfléchir 

Voici donc le code-behind de la Form :

```
using DotBlogTest.Model;
using DotBlogTest.ViewModel;
using Xamarin.Forms;

namespace DotBlogTest.Form
{
    public partial class BirdsPage : ContentPage
    {
        public BirdsPage ()
        {
            InitializeComponent ();
            BindingContext = new BirdsViewModel();
        }

        public void OnItemsSelected(object sender, ItemTappedEventArgs
args)
        {
            var bird = args.Item as Bird;
                if (bird==null) return;
            Navigation.PushAsync(new DetailPage(bird));
            var listView = sender as ListView;
            if (listView != null) listView.SelectedItem = null;
        }
    }
}
```

Le constructeur appelle l'initialisation des composants, ce qui est classique pour du XAML, puis nous créons l'instance du ViewModel pour initialiser la propriété `BindingContext`. Ici aussi il y aurait beaucoup de choses à dire,

notamment sur la création systématique du VM alors que nous pourrions utiliser un cache ou mieux un conteneur d'IOC. Mais encore une fois ce sont des sujets que j'ai traités maintes fois, ici ce n'est pas le propos. Restons simples et agiles. Après tout beaucoup de petites apps mobiles pour entreprise ne réclament pas plus de complexité. La beauté du geste à un coût qui n'est pas toujours justifié.

On note donc la présence du gestionnaire de sélection d'item de la `ListView` avec un code quasiment identique à celui de l'expression Lambda de la version C#. Le code est ici tapé dans Visual Studio avec Resharper qui conseille parfois quelques aménagements. J'aime suivre ces conseils qui sont presque toujours justifiés.

Mixed !

A cette étape particulière du projet nous avons fait la moitié du chemin. Deux fiches, l'une refaite totalement en XAML, l'autre encore dans son état original en C#.

Rien d'autre que ce qui a été montré plus haut n'a été touché, ni dans le code partagé et encore moins dans le projet hôte Android. Rien.

Ayant pris soin de réutiliser le même nom de classe pour la page principale, l'application devrait fonctionner non ?

Après deux ou trois corrections mineures (j'avais oublié le titre de la page, dans le code-behind il manquait un "s" dans le nom du gestionnaire d'évènement… bref la routine !) je lance le projet et voici ce que nous voyons :

La même liste d'oiseaux, présentée de la même façon. J'ai juste modifié le titre pour signifier que l'application est différente en réalité.

Et si on clic sur un item ? Et bien ça marche et cela retourne exactement la même page que dans la version précédente. A la fois parce qu'elle n'a pas été changée et que même si elle l'avait été on ne pourrait pas voir de différence…

Conclusion

Le tricheur ! il ne va pas jusqu'au bout ! Tsss … Mais c'est pour mieux vous laissez le plaisir de le faire ! Je vous offre gracieusement et sans supplément le code de l'application, c'est fait pour ça !

Du point de vue de cet article refaire la même chose avec la fiche de détail n'aurait de toute façon qu'un intérêt très réduit.

Plus loin, vous montrer que l'application fonctionne dans ce mode mixte, une fiche en XAML l'autre en C#, est même un plus. La souplesse des Xamarin.Forms que j'ai déjà évoquée est bien là, on fait ce qu'on veut et ça marche quand même. On choisit son approche, globale ou au coup par coup selon le besoin, et ça marche. On code en XAML, ça marche. On fait du databinding, ça marche et c'est portable cross-plateforme…

C'est vrai que les Xamarin.Forms, techniquement et à défaut d'un designer visuel (j'y tiens !) ce n'est qu'une simple librairie de code comme beaucoup d'entre nous auraient pu en écrire. Rien de vraiment compliqué. Au premier coup d'œil seulement…

Car derrière chaque classe utilisée se trouve un mécanisme qui traduit tout cela en composants natifs à la compilation. C'est un système totalement cross-plateforme, code mais aussi UI qui s'utilise de façon transparente, c'est un boulot énorme !

C'est vraiment beau. Du cross-plateforme "out of the box" avec C# et XAML…

Allez Miguel, un petit effort, offre-nous un designer visuel pour les Xamarin.Forms … Maintenant que tout cela est dans le giron Microsoft ce ne devrait pas être énorme.

Les Xamarin.Forms avec XAML c'est le pied de nez d'outre tombe de Silverlight à Sinofsky ce visionnaire de pacotille… C'est une fois encore la preuve qu'on n'est pas prêt de détrôner le couple C# / XAML pour faire rapidement des applications professionnelles pour toutes les plateformes. C'était la définition de Silverlight non ?

Merci Miguel, merci de la part de Silverlight.

Ce n'est pas fini, restez encore : voici le code de la solution

FormsXaml

Support de Windows Phone

Les deux premiers volets de cette série vous ont montré les grandes lignes du cross-plateforme avec Xamarin.Forms. Nos exemples tournaient sous Android. Il est temps de les faire tourner sous Windows Phone !

Windows Phone

On le sait tous Windows Phone a eu des débuts difficiles. Si difficiles que bien que présent depuis des années il a toujours des parts de marché qui ressemblent aux

premières semaines du lancement d'un nouveau produit … Mais nous sommes là pour cela change non ?

Windows Phone est pourtant depuis sa première apparition (Windows Phone 7 sous Metro / Silverlight) le meilleur OS mobile qu'on puisse trouver. Aujourd'hui la version 8.1 est vraiment au top de ce qu'on peut attendre d'un tel OS. Peu-être pas forcément pour le grand public, sinon ça se verrait d'ailleurs, mais nous les professionnels œuvrant principalement en entreprise on le sait. D'ailleurs la version 8.1 a cédé depuis sa place à UWP, vraiment « universel » (si l'univers se réduit au monde Windows).

Le quidam ne peut savoir la beauté de C#, l'importance du vectoriel de XAML dans un monde noyé par les form factors, la puissance d'un EDI comme Visual Studio, l'incroyable qualité d'un outil de design comme Blend, la force d'un framework comme .NET … Non, le quidam de la rue ne voit que des tuiles qui ne lui plaisent pas depuis le premier jour. Et c'est vraiment dommage, surtout quand on voit que ce truc de loser a été repris et repris dans Windows Phone 8 puis 8.1 et même dans Windows 10. Car s'il savait ce qu'était Cocoa et son C minable tout autant que les bricolages odieux de type nine-patches des UI Android, il préfèrerait certainement lui aussi la pureté et la cohérence de Windows Phone ou de UWP aujourd'hui.

C'est que quelque part on lui explique mal et qu'on refuse de lui mettre un bureau classique comme tout le monde, ce que Microsoft commence à faire sur PC après l'erreur W8 et de ces fichues tuiles qui sont à la base de tous les derniers flops... Je peux toujours dire du bien de Windows Phone ici, c'est un blog que le quidam ne lit pas… néanmoins en rappelant aux professionnels que Windows Phone est tout de même ce qui se fait de mieux, même du point de vue du développement, peut-être arriverai-je à influencer un peu le quidam… Nous sommes des geeks pas des no-live donc chacun de nous connait aussi des quidams que nous pouvons convaincre !

Et d'ailleurs dans cette quête évangélique motivée par mon vrai amour de Windows Phone et non pour faire plaisir à qui que ce soit, je vais vous montrer comment on ajoute le support de Windows Phone à notre application de démo d'hier ! C'est un support Windows Phone 8.1 qui sera ajouté, d'abord parce que ce sont les machines les plus nombreuses et puis parce que Xamarin.Forms ne couvrait pas UWP qui lui-même n'était pas sorti quand j'ai écrit l'article ! Aujourd'hui on peut toujours travailler avec des projets Windows Phone 8.1 mais aussi et en plus avec des projets UWP, ce qui cible les nouveax Windows Phone mais aussi, et là ça devient vraiment séduisant, les tablettes Surface et les PC.

Attention ça va très vite !

Je vais décomposer le mouvement en le filmant à très haute vitesse car l'action va très vite, préparez-vous pour ne rien louper malgré le ralenti…

Bon, ça va être tellement court qu'avant d'y aller je veux vous parler de deux choses importantes :

L'enregistreur d'action de l'utilisateur

Au lieu de pirater SnagIt ou équivalent savez-vous que Windows, depuis la version 7, est fourni avec un "enregistreur des actions de l'utilisateur" ?

Kesako ? Un super mouchard pour tenter de piéger les bugs. Mais on peut aussi s'en servir pour enregistrer toute action dont on souhaite garder une trace, pour une démo, un article…

Bouton démarrer / lancez : psr

L'intérêt de cet utilitaire est de prendre automatiquement des screenshots de tous les écrans installés en une seule image, de montrer où l'utilisateur a cliqué, et de fournir des commentaires très précis (par exemple le nom des boîtes de dialogue, le nom des entrées de menu sélectionnées, etc).

C'est vraiment un super outil, j'espère vous avoir fait découvrir là un truc gratuit et très puissant qui vous sera très utile (on peut demander à un user de lancer PSR et de faire normalement les manips "qui plantent", il suffit ensuite de récupérer les fichiers de PSR pour "voir" ce qui a été fait. C'est donc parfait en local mais aussi pour du dépannage à distance comme si on était assis derrière l'utilisateur… Futé !).

Bien entendu, pour lever toute ambigüité, c'est dans Windows 7, 8 et 10, c'est gratuit, et cela n'a rien à voir avec l'utilitaire de capture écran, pratique mais incapable de faire le reporting précis et automatique de PSR.

Les références de projet partagés

Tel que nous y allons là, la fleur au fusil, on va vite être déçu… En effet nous avons choisi dans la solution originale d'utiliser un projet partagé pour le code commun au lieu d'une PCL. Je reparlerais des différences mais les deux sont ok.

Seulement voilà, même avec tout bien installé, il est impossible d'ajouter la référence au projet partagé dans le projet Windows Phone… L'option n'existe pas et si on tente de bricoler à la main le fichier projet ça peut tourner à la catastrophe facilement. Heureusement tout cela est réglé depuis cet article.

Il faut savoir que les projets partagés de Xamarin.Forms se basent tout simplement sur les "Projets Universels" dont je vous ai déjà parlés (voir "Template d'applications universelles : convergence WinRT et Windows Phone" d'avril dernier).

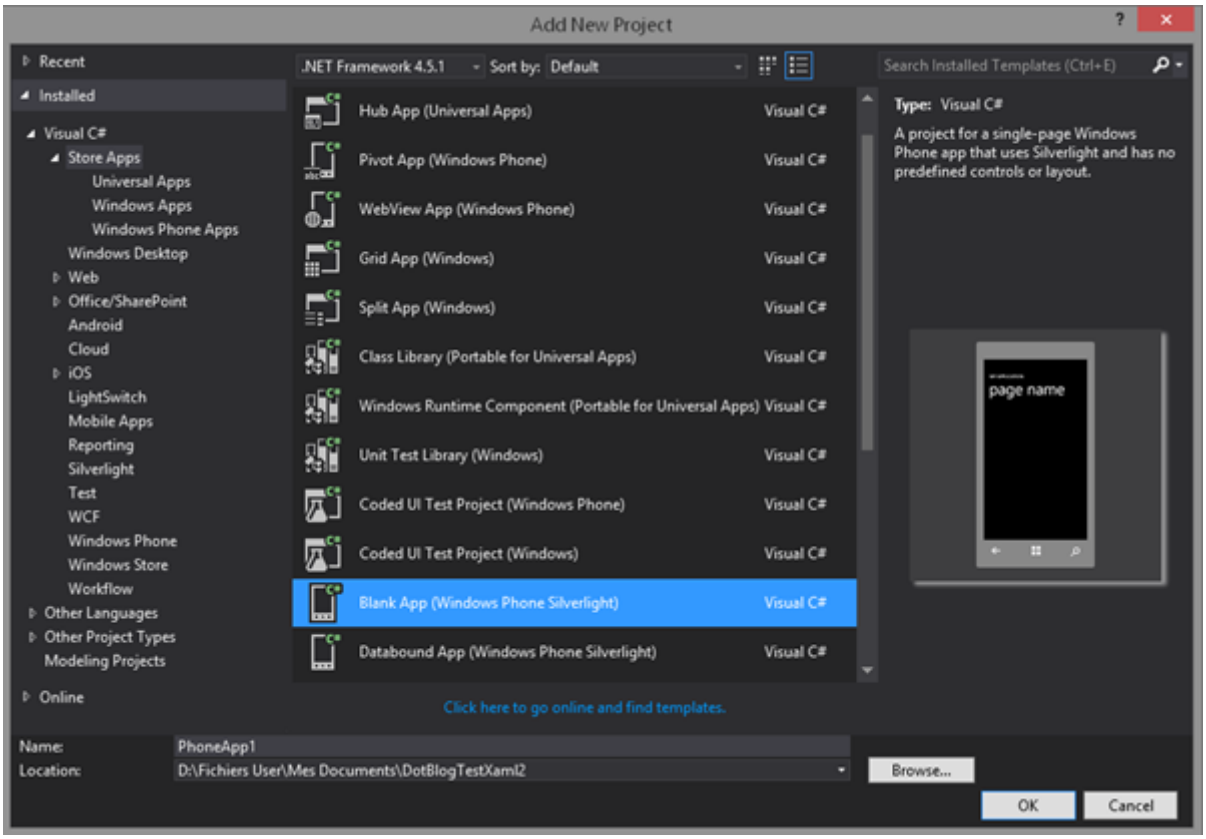
De ce fait il est possible d'ajouter la référence comme on le ferait pour un projet d'application universelle. Mais cela n'existe pas encore dans VS (maintenant oui). Mais il y a une solution : le Shared Project Reference Manager, extension gratuite pour Visual Studio qu'il suffit d'installer… Et Ô magie, une fois le projet Windows Phone ajouté à la solution il sera possible d'ajouter la référence au projet partagé comme d'importe quelle autre référence.

Vous ne pourrez pas dire que je ne vous mâche pas le travail !

Phase 1 : Ouverture de l'ancienne solution

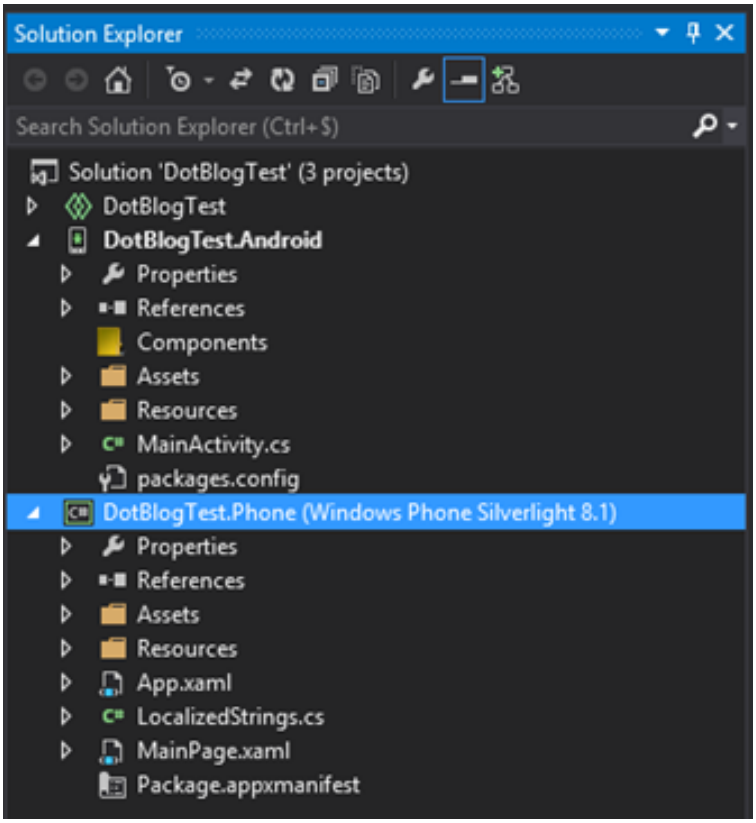
Toujours même principe, je fais un copier / coller de la solution d'hier et je renomme uniquement le répertoire de niveau supérieur. Aujourd'hui il s'appelle donc DotBlogTestXaml2.

Je me positionne sur la solution, clic droit, ajouter nouveau projet :

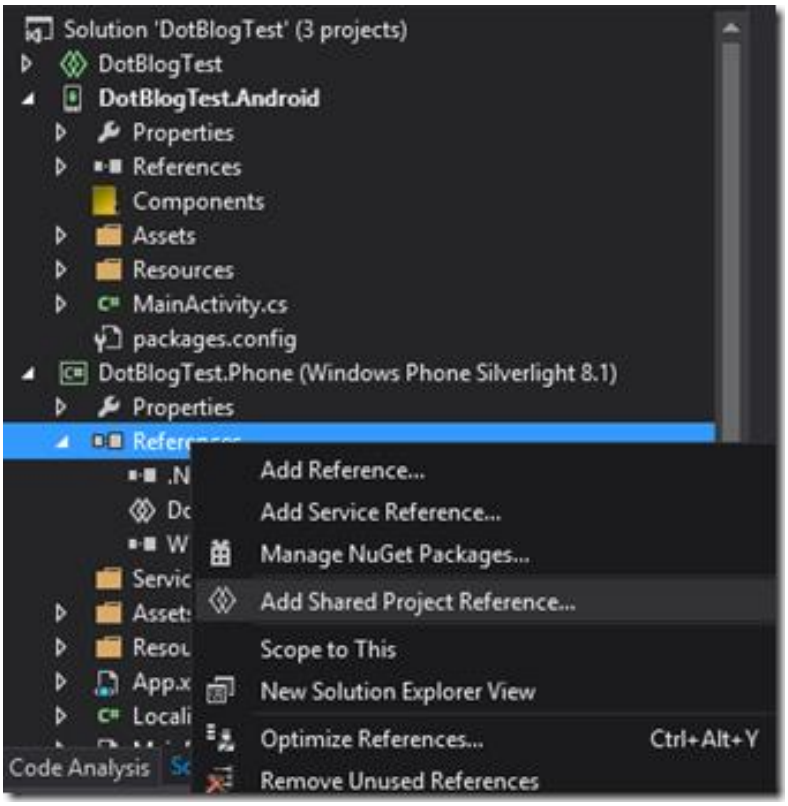


Je choisi le plus simple, "Blank app (Windows Phone Silverlight)".

Notre solution ressemble donc maintenant à cela :



Je vais placer le projet Windows Phone en projet par défaut, et ajouter la référence au projet partagé grâce au petit plugin évoqué plus haut :



La référence est bien ajoutée avec une icône reconnaissable (deux losanges verticaux entrelacés) :

Phase 2 : supprimer le code de démonstration et charger notre page

Bien entendu le projet Windows Phone par défaut contient un peu de code démo ce qui permet de le compiler immédiatement et de tester l'environnement de développement. Il faut supprimer ce code, peu de chose, uniquement dans la page principale.

Nous allons tellement faire le ménage que nous n'allons rien garder du code XAML sauf l'enveloppe de la page :

```
<phone:PhoneApplicationPage
    x:Class="DotBlogTest.Phone.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

</phone:PhoneApplicationPage>
```

C'est sur il ne reste rien, juste la `PhoneApplicationPage`.

Faisons un tour dans le code-behind et de même, retirons tout pour juste ajouter le chargement de la page liste du projet partagé mais juste avant il nous faut installer via les packages Nuget "Xamarin.Forms" :

Nous pouvons maintenant ajouter les quelques lignes nécessaires :

```
using Microsoft.Phone.Controls;
using Xamarin.Forms;

namespace DotBlogTest.Phone
{
    public partial class MainPage : PhoneApplicationPage
    {
        // Constructor
        public MainPage()
        {
            InitializeComponent();
            Forms.Init();
            Content =
DotBlogTest.App.GetMainPage().ConvertPageToUIElement(this);
        }
    }
}
```

On retrouve l'initialisation des composants, comme pour toute page XAML, puis un appel à `Forms.Init()` qui met en route les Xamarin.Forms et enfin nous fixons la propriété `Content` de la page XAML en appelant le `GetMainPage()` de notre projet partagé, le même appel que dans le projet Android (comparez vous verrez !). Toutefois ce que retourne cette méthode n'est

pas tout à fait du XAML il faut ainsi utiliser `ConvertPageToUIElement()` pour parfaire le traitement.

Phase 3 : Quelle phase 3 ?

Je ne plaisante pas, de quoi vous voulez parler ? Il n'y a pas de phase 3 puisque nous avons ajouté le projet Windows Phone et que nous avons initialisé sa `MainPage` grâce à Xamarin.Forms et à notre projet partagé.

Que voulez-vous faire de plus ?

Juste un Run pour se prouver que ça suffit ? C'est bien parce que c'est vous, en GIF animé en plus, Dot.Blog c'est vraiment la crème du top du luxe ! (ce luxe s'arrête aux limites de la technologie hélas car dans ce livre l'image n'est pas animée… retrouver l'article original non mis à jour sous Dot.Blog, ne lisez pas le texte mais regardez les animations !).

Conclusion

Je vous laisse tirer la conclusion vous-mêmes… Je suis certain que parmi vous il y en avait qui se préparaient à quelque chose d'un peu plus … rugueux. Je suis navré même en blablatant un peu, je ne peux honnêtement pas faire plus long.

Et oui, pour créer une application Windows Phone quand on dispose du projet partagé il faut en effet moins d'une minute chrono en main… Pareil pour Android, pareil pour iOS etc.

Tout le travail se trouve dans le projet partagé (ou la CPL) et c'est tout. Bien sûr que ce travail là peut être long et même délicat ! Je ne connais pas de projets qui s'écrivent facilement dès qu'ils font quelque chose d'intelligent… Mais ça c'est notre métier.

Notre métier ne consiste pas à faire de la plomberie pendant 7h pour coder deux lignes efficaces. C'est tout le contraire.

Et grâce aux Xamarin.Forms il possible de se concentrer enfin sur le code sans se prendre la tête avec la portabilité car _même_ l'UI est portable.

Vous êtes bluffé un peu j'espère… ou alors vous êtes blasé ! Moi j'ai été bluffé.

On verra bien d'autres choses sur les Xamarin.Forms ces trois premiers billets ne font qu'effleurer la surface.

PS : le code du projet Cross-Plateforme XAML

Les Labs le Tooklit des XF

Dans les trois précédents chapitres je vous ai présenté les Xamarin.Forms sous Android et Windows Phone. Il s'agit des bases et nous irons bientôt plus loin. Mais avant je voudrais vous présenter Les Xamarin.Forms.Labs, sorte d'équivalent du Toolkit pour WPF ou Silverlight mélangé avec les plugins de MvvmCross…

Xamarin.Forms

Je renvois le lecteur intéressé aux précédents articles qui expliquent et démontrent ce que sont les Xamarin.Forms. En très rapide pour ceux qui voudraient juste se faire une idée : Les Xamarin.Forms sont à l'UI ce que Xamarin est à C#, c'est à dire une plateforme de développement mobile portable mais qui ajoute la portabilité des UI.

Bien que très jeunes et sorties avec Xamarin 3.0, les XF (Xamarin.Forms) contiennent déjà l'essentiel pour concevoir des applications totalement cross-plateformes tant du point du code (ce que Xamarin faisait déjà) que de l'UI ce qui est révolutionnaire d'autant que cela peut se faire soit en code C# soit en descriptif avec un sous-ensemble de XAML (databinding two-way inclus). Les articles précédents ont démontré ces facettes.

A peine sorties, à peine disponibles les XF sont déjà des stars montantes car tous ceux qui ont pu les approcher et s'en servir, même pour faire des tests, sont tombés sous le charme… Nous tenons enfin une solution complète cross-plateforme qui n'utilise que C#, .NET et XAML !!!

Forcément cela aiguise les envies d'aller plus loin. Et c'est en cours.

Xamarin.Forms.Labs

Silvertlight ou WPF ont eu leur "Toolkit", ensemble de controls et de codes permettant d'étendre les fonctionnalités ofertes "out of the box". Devenus tout

de suite indispensables ces Toolbox font systématiquement partie de tout développement sauf très rares exception.

Il faut comprendre les Xamarin.Forms.Labs de la même façon : se plaçant au-dessus des XF les Labs offrent des fonctionnalités nouvelles, quasi indispensables, le tout en cross-plateforme…

Si je parlais d'un esprit mélangeant Toolbox XAML et Plugins de MvvmCross c'est que si on connait ces deux produits on comprend immédiatement de quoi sont faits les Labs et ce qu'ils apportent :

- D'une part comme le Toolbox XAML les Labs ajoutent des contrôles ou des services utiles
- D'autre part comme les plugins MvvmCross ils s'installent via les package Nuget et sont portable dans tous les projets cibles automatiquement.

MvvmCross que je vous ai longuement présenté (notamment avec un cycle de 12 vidéos Youtube) utilise la notion de plugin pour ajouter par exemple la sérialisation JSON, la base de données SQLite, etc. Ce qui étend les possibilités cross-plateformes du framework.

Les Xamarin.Forms.Labs agissent de la même façon : une fois installés dans chaque projet cible et dans le projet "noyau" (projet source partagé ou PCL) les Labs fournissent de nouveaux services transparents qui fonctionneront sans code supplémentaire sur toutes les plateformes. Par exemple le support de SQLite ou l'accès aux informations de la machine. Autant de choses qui doivent être faites en natif mais qui grâce aux Labs peuvent être pilotés depuis le code cross-plateforme. On retrouve la notion de framework de base et de plugins installables selon les besoins.

Les Labs sont l'extension natuelle des Xamarin.Forms. En effet, tant que Xamarin ne fournissait qu'un compilateur C# portable (avec .NET) il fallait de toute façon coder les UI à la main dans chaque projet cible. Le besoin d'un système de plugin transparent ne se voyait pas tant que ça. Trois cibles, trois projets distincts avec un peu de C# partagé éventuellement.

Puis vint MvvmCross qui a unifié tout cela en proposant même un Binding portable. Mais avec MvvmCross il faut toujours trois projets pour trois cibles et les UI sont toutes distinctes même si elles se reposent sur des ViewModels communs.

Les Xamarin.Forms en apportant une couche d'abstraction de l'UI rend cette dernière totalement portable. De fait, et comme je vous l'ai montré dans les

exemples précédents, on ne travaille plus qu'un seul code central autant pour le code habituel que pour les UI. Les projets cibles existent toujours car chacun doit être compilé en natif dans le respect de sa plateforme, mais on n'y touche pas. Juste deux lignes d'initialisation le reste étant entièrement codé en cross-plateforme !

Il est clair que cette nouvelle donne impose de pouvoir accéder à certaines particularités des OS. On peut bien entendu utiliser des #if ou bien bricoler dans les projets natifs certaines choses mais tout cela n'est pas très propre.

Le mieux serait de bénéficier comme dans MvvmCross de "plugins" portables. J'ajoute le support de JSON dans mon "noyau" et je m'en sers dans mes ViewModels. Ce même plugin est ensuite ajouté à chaque projet cible (via Nuget), je compile et chacun se retrouve avec sa version native de JSON alors que mon noyau n'en connait qu'une abstraction…

C'est exactement ce que font les Xamarin.Forms.Labs.

Que trouver dans les XF et où les trouver ?

Xamarin 3.0 est sorti l'année dernière et une vingtaine de jours à peine après cette sortie les Labs devenaient déjà une réalité avec une dizaine de contributeurs. On sent que la motivation est là ! Le temps a passé et ça s'étoffe tous les jours avec en prime la prise en charge récente de UWP !

Qu'y-a-t-il dans les XF Labs ?

Si les Forms sont jeunes, les XF le sont plus encore mais ils regorgent déjà de code précieux…

- Contrôle Calendrier,
- ExtendedTabbedPage,
- ImageButton,
- ExtendedLabel,
- ExtendedViewCell,
- ExtendedTextCell,
- AutoComplete,
- HybridWebView…

Côté services on trouve :

- Le Text to Speach
- L'accès divice (battery, senseurs, accéléromètre…)

- Les accès Phone (information réseau, passer des appels…)
- Géolocalisation
- Accès caméra (Image et video picker, prendre une photo ou une vidéo…)

Tout un bloc des XF est dédié à MVVM et même si tout n'est pas encore terminé on dispose déjà de blocs essentiels :

- ViewModelBase (support de la navigation, IsBusy, Set des propriétés, INPC)
- RelayCommand (générique avec ou sans paramètre comme dans MVVM Light)
- ViewFactory (pour lier ViewModel et View)
- IOC pour la gestion centralisée des services
- IXFormsApp (gestion des événements de l'appli comme la suspension)

En plus de son fonctionnement de base, le framework Labs accèpte lui aussi dans un jeu de poupées russes la notion de plugin ! On trouvait déjà au début du projet les plugins suivants pouvant être installés par dessus XF selon les besoins :

- Sérialisation (ServiceStackV3, ProtoBuf, JSON.NET)
- Caching (SQLLiteSimpleCache)
- Conteneur d'injection de dépendance (TinyIOC, AutoFac, Ninject, SimpleInjector)…

Pour info, en mars 2016 on trouve 2069 packages pour les Forms Labs !!!

C'est donc toute une panoplie d'outils indispensables qui est en train de se créer autour des Xamarin.Forms en faisant l'outil privilégié pour tout développement cross-plateforme désormais.

Où trouver les XF Labs ?

On trouve tout sur Nuget.org bien entendu et sur github pour le code source.

Le source de l'article : Xamarin.Forms.Labs
Labs sur Nuget : liste de tous les package à jour

Conclusion

Trop de bonheur ne nuit pas… Après l'excellente nouvelle que sont les Xamarin.Forms et leurs grandes qualités, cette librairie se trouve être suffisamment porteuse de rêve et d'espoir pour que dans la foulée naissent les Labs, un Toolkit d'ores et déjà très élaboré intégrant la notion de plugin cross-plateforme…

Comme je le dis, presque comme une incantation mystique, il ne manque plus que le designer visuel XAML et le rêve Silverlight de C#/XAML portable partout renaitra de ses cendres encore tièdes. En mieux. Plus fort, plus vigoureux, de l'iPhone à Android en passant par Windows Phone, le Mac le PC…

Les XF et les XF Labs sont les outils d'aujourd'hui et assurément de demain.

Grâce à Dot.Blog vous ne serez pas passé à côté de cette révolution, à vous d'en tirer profit !

Les outils annexes

Les données

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. Et c'est pour cela que j'en parle dans ce livre consacré aux Xamarin.Forms !

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 changent 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 sûr. 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

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
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
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

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
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
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

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
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
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

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
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
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

	Json.NET	DataContractJsonSerializer	JavaScriptSerializer
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…

Le Cloud

Nadella en a fait son cheval de bataille (“Cloud first”), mais tout le monde en fait et en parle aussi (Amazon, Google…). Et tout le monde voudrait que vous l’utilisiez. Qui ça ? Le Saint Cloud. Pas l’ancienne citée ouvrière reconvertie en parc à Bobos, non le “cloude”, le nuage américain. Mais c’est quoi ? Et pourquoi en parler ici ?

Pourquoi le Cloud dans un livre réservé aux Xamarin.Forms ?

Avant tout je me devais de préciser le pourquoi de la présence de cet article dans le présent livre consacré au développement cross-plateforme avec les XF.

Je serai bref : de nombreuses applications mobiles doivent accéder à des données centralisées (utilisateurs, configurations, données réelles, etc) et le Cloud s’avère la solution la mieux taillée pour assurer à la fois le stockage et la distribution de telles données. Comprendre ce qu’est le Cloud et les services qu’il peut offrir permet en toute logique de concevoir des applications mobiles et cross-plateformes plus efficaces et plus utiles…

Le Cloud ambigu par nature

Si les gens disent un peu n’importe quoi à propos du Cloud ce n’est pas forcément entièrement de leur faute. Le Cloud lui-même est un concept *protéiforme*, un peu flou, dématérialisé par nature et utilisable de façons très différentes. La légende veut même que nos ancêtres les gaulois avaient peur que les clouds ne leurs tombent sur la tête, c’est pour dire la méfiance ancestrale que nous avons des nuages !

> ***La minute du regretté Me Capelo : “protéiforme” ne vient pas comme d’aucuns le pensent à tort de “protéines” même si celles-ci savent se contorsionner et prendre plusieurs formes. Le mot vient du dieu grec Protée, dieu marin qui pouvait prendre toutes les formes qu’il désirait. Si le reste de l’article vous rase vous pouvez vous arrêtez là en ayant appris quelque chose. Dot.Blog pense à tout !***

Du stockage simple de données à l’hébergement d’OS et d’applications dédiées, le Cloud est vraiment aussi nébuleux qu’un nuage. Les américains sont forts pour donner des noms aux choses. Cloud veut dire nuage. C’est clair dès le départ : ça sera nébuleux. Apple ? On vous prend pour une pomme c’est dit dans le nom.

Word ? ben… ce sont des mots, pour écrire donc. Ils ont ce sens incroyable du terme commun qui devient nom de produit ou de technologie. En Français cela serait totalement impossible. Voyez-vous un produit ou un service qui s'appellerait "Nuage", "Pomme" ou "Mot" ? On a des exemples d'ailleurs. Quand on pense à l'échec de la Renault 14 juste parce que la première publicité comparait son apparence à une poire. Personne n'a voulu, même très indirectement, être pris pour une poire en France. En revanche on aime se faire prendre pour une Apple. Preuve que les français maitrisent très mal l'anglais… Et pourtant la R14 ne s'appelait pas la Renault Poire ! Alors Cloud comme Apple ou Word, c'est impensable en français.

Bref nos amis d'outre atlantique sont des farceurs. Cloud c'est un nuage et c'est par définition un truc aux contours mal définis, changeant de forme sans cesse, pouvant être ici ou là, apparaissant de nulle part et disparaissant sans crier gare (pensez que le Cloud se pratique avec une connexion Internet et regardez la qualité moyenne de celle-ci dans notre pays…).

Le Cloud suscite donc la méfiance, by design je dirais. On n'aime pas ce qui est flou, mouvant, mal défini, et on a raison c'est souvent dangereux. D'ailleurs côté danger le Cloud c'est la mort de la "privacy", qu'on traduit assez mal en français par "vie privée" car le terme américain couvre un champ sémantique bien différent en réalité même s'il englobe le concept de vie privée, mais pas que.

Frein majeur à l'adoption du Cloud par les entreprises, la "privacy". Confier ses données, ses clients, son chiffre d'affaire tout ça à un "nuage" Brrrr ! Déjà avec la NSA on sait que toute entreprise de bonne taille se fait espionner alors que ses serveurs sont locaux, alors tout déplacer pour le donner directement à une entreprise américaine…

Amazon, Google, Microsoft…. C'est vrai que ça fait froid dans le dos. Alors chacun y va de son petit truc pour vous prouver qu'il vient en ami. Dernièrement Microsoft a bravé les autorités judiciaires de son pays en refusant de communiquer des informations sur certains comptes mails stockés hors USA. Ils ont même eu une phrase frappée au coin du bon sens "vos mails n'appartiennent qu'à vous, pas à nous" (traduction rapide). On se demande où se situe la limite entre sincérité et gros coup de com' … Vous vous direz que je suis un esprit chagrin cherchant des poux partout, mais moi je pense que ça a l'air trop bien préparé jusque dans la petite phrase sortie d'un brainstorming de markéteux pour être totalement honnête… Mais bon, même si c'est pour se faire de la pub c'est bien de rappeler que nos données sont à nous et pas au marchand de Cloud… Tout est flou avec le Cloud, même les prises de position à propos du Cloud !

Résumons : côté technique le Cloud c'est nuageux et flou, côté géopolitique ce n'est pas clair (confier son savoir à un étranger), côté business c'est risqué (offrir ses secrets à un nuage), et même sur le plan citoyen ça pose problème. Je me rappelle d'un ancien directeur de la CNIL qui disait en substance dans une interview "un pays dans lequel on ne plus frauder est-il encore une démocratie ?". Bien entendu le Web et Google n'arrivent pas à retrouver la référence de cette citation gravée en revanche dans ma mémoire (mais le web est amnésique pour tout ce qui a plus de 20 ans !). Sur le Web nada. Noyée. Enterrée. Toute pensée subversive est nettoyée. J'aime aussi le joke qui fait trembler et qui dit que 1984 de Orwell n'était pas censé être un manuel d'utilisation… Quand vous donnez toutes vos photos, vos vidéos, vos lettres, vos souvenirs, votre vie à DropBox, OneDrive ou autre, votre vie vous appartient-elle encore ? Êtes-vous toujours un *citoyen libre* ? On est en droit de se le demander malgré tout.

Bref le Cloud c'est tout ça, c'est vrai. Quand on parle de C# on parle tout de suite technique, comparatif avec C++ ou Java, bouts de de code. Quand on parle du Cloud les premières choses qui viennent à l'esprit ce sont les nuages noirs et mystérieux chargés d'une puissance maléfique qui nous viennent en tête (et au dessus de la tête). La technique ne vient qu'après, très loin après.

Ceci explique certainement que le Cloud bien que s'infiltrant de gréé ou de force un peu plus chaque année n'ait pas connu un engouement plus grand que ça ce qui vaut même, revenons à l'introduction, que Satya Nadella en fasse son cheval de bataille : "Cloud first", comme "Mobile first", on sent bien que ce sont les échecs d'hier qui

sont mis en avant pour la grande bataille de demain. Nadella ne dirait pas "Excel first" ou "Office first". Ce sont des marchés déjà gagnés par Microsoft. Alors que le Cloud et le Mobile, ça reste à faire.

Donc quand on pense au Cloud on pense à tout ça. Qui n'a rien à voir avec l'outil technique extraordinaire que peut être le Cloud et le progrès qu'il représente dans le partage des données et leur délocalisation notamment.

C'est pourquoi maintenant que nous avons évacué les peurs, parlons vraiment du Cloud !

Le Cloud côté technique

Laissons derrière nous les phobies gauloises et autres craintes citoyennes et entrons dans le vif du sujet : qu'est ce que le Cloud, techniquement et non pas comme instrument de personnes mal intentionnées. Car finalement on pourrait en dire autant du marteau. Entre de mauvaises mains il se transforme en arme redoutable et mortelle. Pourtant il est en vente libre et fait le bonheur des bricoleurs du dimanche autant que la peine de leurs voisins moins matinaux…

Définition

Tentons une définition : Le Cloud est une infrastructure basée sur Internet dont les ressources, logiciels et informations sont fournies à la demande à d'autres ordinateurs ou assimilés. Comme fonctionne le réseau électrique avec ces centrales (les centres serveurs du Cloud) ses transformateurs et lignes (la plomberie Internet) et ses fils qui arrivent chez les consommateurs (la paire torsadée du téléphone).

Le Cloud computing est en fait l'aboutissement de très nombreux essais parfois anciens de fournir des ressources informatiques centralisées à des clients décentralisés. Le Minitel en France en est certainement le plus bel exemple pour une application grand public (juste un terminal, aucun stockage, toute l'information est "ailleurs" sur des ordinateurs qu'on ne voit pas et qu'on ne connait pas). Comme d'habitude dans notre beau métier on prend des trucs qui datent de 30 ans et on leur met un joli nom à la mode et c'est parti pour de longues discussions sur la "nouvelle" technologie !

On peut dégager un certain nombre de caractéristiques qui définissent assez bien le Cloud du point de vue des données, du calcul ou de l'infrastructure :

- Hébergement à distance : les services et données sont hébergés par une infrastructure distante.

- Ubiquité : Les services et données sont disponibles depuis n'importe où (sous réserve d'une connexion Internet)
- Commodification : néologisme impossible à traduire correctement ("transformation en objet" est lourd et imprécis) mais qu'on comprend facilement puisqu'il s'agit ici de transformer en un bien de consommation courant quelque chose qui ne l'est pas forcément au départ. On retrouve la notion de SaaS, logiciel comme un service par exemple. On transforme des services, des données en biens de même nature que le gaz ou l'électricité qu'on paye en fonction de sa consommation.

Pour résumer on pourrait dire que le Cloud Computing c'est l'assemblage du Software as Service (Saas) plus de la Platforme as Service (PaaS) plus l'Infrastructure as Service (IaaS).

Le SaaS c'est Office 365 : un abonnement pour utiliser le pack office en ligne, vous arrêtez de payer, vous n'avez plus Office…

Le PaaS se destine principalement aux entreprises, le Cloud fournit une plateforme de type logiciels de base, OS, etc, et l'utilisateur y exploite ses propres logiciels. Un site Web dans le Cloud peut être une des utilisations du PaaS. Les frontières sont parfois floues. Un site Web hébergé chez un fournisseur est déjà du PaaS ou non ? Oui et non. Car ce qu'on entend par PaaS notamment avec des fournisseurs comme Microsoft est un peu plus sophistiqué.

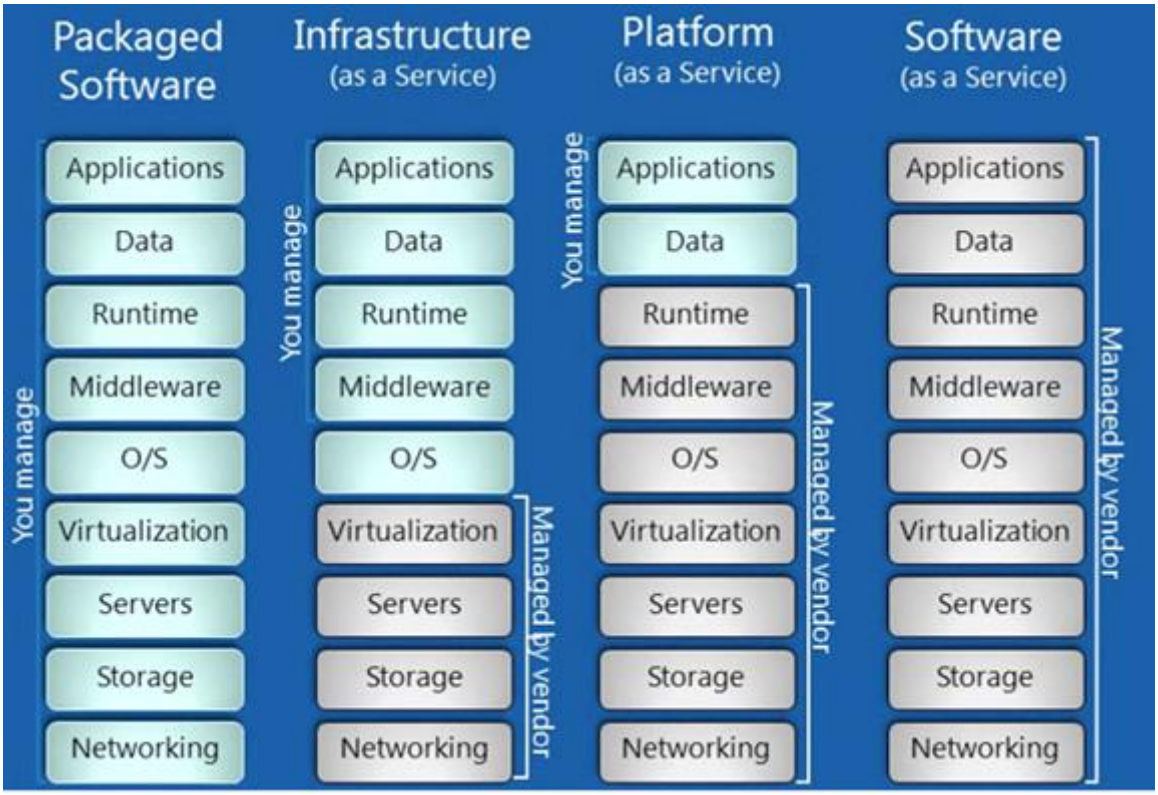
l'IaaS est la mise à disposition d'un utilisateur, entreprise en générale, d'une infrastructure distante. Le prestataire entretient les machines, leur OS, etc. C'est comme un ordinateur distant.

La différence entre IaaS et PaaS est parfois difficile à saisir et cela est normal, c'est flou c'est nuageux… par essence.

Les modèles de services dans les nuages

Il existe donc plusieurs modèles de Cloud Computing, tous partagent la notion de service, de tarification selon la consommation et plus rarement sur la base de forfaits (souvent pratiqué en revanche pour le pur stockage des données de type DropBox).

On peut résumer ces différents modèles que nous venons d'évoquer par le schéma suivant :



Les quatre colonnes de ce schéma représentent les 4 grands services qu'on peut imaginer actuellement autour des logiciels et des données.

Les lignes correspondent aux niveaux de profondeurs techniques qui doivent être maintenus par l'utilisateur / consommateur ou par le fournisseur de service.

Logiciels en boite

La colonne à gauche représente le modèle classique dit "packaged softaware" c'est à dire du logiciel vendu dans une boîte (virtuellement le plus souvent aujourd'hui). L'utilisateur / consommateur doit tout gérer lui-même, du réseau (ligne du bas) aux applications (ligne en haut) en passant par toutes les couches : stockage, serveurs, virtualisation, OS, etc.

Iaas

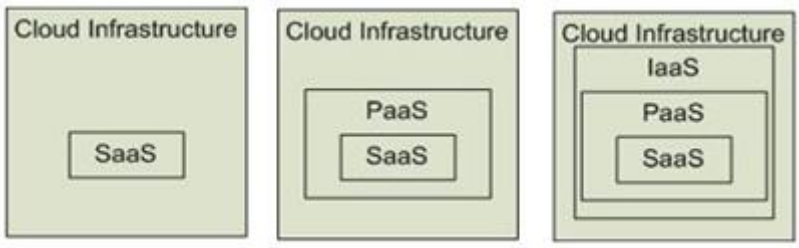
L'IaaS est la première étape du Cloud, le service minimum pourrait-on dire. Le fournisseur s'occupe de l'infrastructure, du réseau à la virtualisation. Tout le reste est à la charge de l'utilisateur, de l'OS aux applications. C'est en quelque sorte la location d'un ordinateur distant, voire d'une quantité de travail sur un même ordinateur distant partagé par plusieurs clients (ou une ferme de machines). Toutefois ce qui différencie l'IaaS de la location d'une machine partagée pour un site Web par exemple c'est qu'ici l'OS fait déjà partie des

attributions de l'utilisateur. Ce dernier peut installer l'OS qu'il désire et en faire ce qu'il veut.

PaaS

Dans le PaaS on reprend tout ce qu'offre l'IaaS plus la prise en charge par le fournisseur de l'OS, du middleware et du runtime. L'utilisateur ne s'occupe plus que d'installer des applications conçues pour la plateforme du fournisseur et de gérer ses données dans des formats compatibles avec cette dernière.

SaaS



Dernier étage de la fusée qui va dans les nuages, le SaaS représente l'ultime fantasme des éditeurs de logiciels. Il y a fort longtemps lorsqu'il était encore CEO de Microsoft, Bill Gates avait déjà annoncé que c'est vers cela qu'il voulait aller. Cela paraissait délirant car techniquement trop loin de la réalité, aujourd'hui c'est la réalité… Office 365 c'est du SaaS. On loue un accès à Office, et quand on arrête de payer on n'a plus Office. On voit ici toute la différence qui oppose le modèle classique au SaaS fantasmé et en train d'être imposé… Dans le modèle classique vous payez un logiciel, si vous ne voulez pas payer pour les nouvelles versions vous avez la possibilité de continuer à travailler avec la version achetée. Avec le Saas vous avez tout le temps la dernière version, mais dès que vous arrêtez de payer on vous coupe le robinet. Comme le gaz ou l'électricité. A une époque où on réfléchit à la meilleure manière de rendre les habitations autonomes (panneaux solaires, chauffage géothermique, récupération des eaux de pluie, éoliennes, etc) il me semble, c'est un avis personnel, que le SaaS va totalement à contre courant en recréant un mécanisme de dépendance qui, on comprend pourquoi, fait fantasmer les éditeurs de logiciels…

Personnellement je soutiens l'IaaS et le PaaS mais je suis une ferme opposant au SaaS qui est la pire chose qui puisse se faire. La dernière étape vers le dépouillement de l'utilisateur qui devient vache à lait captive. Idéologiquement et quelque soit mes sympathies techniques pour tel ou tel éditeur comme Microsoft, je me dois de m'élever et de mettre en garde contre cette dérive qu'est le SaaS (peu importe le fournisseur). Mais chacun fera en son âme et conscience, je vous livre ma pensée et même si j'espère qu'elle vous fera réagir je ne cherche pas à vous l'imposer !

Les trois grands fournisseurs

Le marché est chaud bouillant car naissant. Tout le monde se déchire mais déjà trois grands fournisseurs s'imposent sur le marché même si cela ne présage pas d'une stabilité absolue dans le futur. On l'a vu dans la téléphonie, Android sortant du diable vauvert et venant prendre la première place sous le nez de Apple et son iPhone idole des jeunes et sous la barbe de Microsoft créateur de la technologie avec le Pocket PC sorti en 2000 soit 7 à 8 ans avant la sortie de l'iPhone 1 (2007) et de Android 1.0 (2008).

Devant une telle leçon de l'histoire il faut donc rester humble sur les prévisions qu'on peut faire !

Google App Engine

C'est bien plus une interface Web pour un EDI qui permet facilement de déployer des applications Java ou Python.

Techniquement tout cela repose sur l'infrastructure Google qui est mondiale et efficace. Donc pas de problème de ce côté. C'est la spécificité de l'offre qui réduit considérablement les avantages.

Amazon Web Services

EC2 (Amazon Elastic Compute Cloud) offre une plateforme de services Web qui peut s'adapter à la montée en charge et qui bien entendu réside dans les nuages donc avec l'ubiquité généralement recherchée.

L'offre est techniquement intéressante avec un load balancing, des outils de monitoring etc de très bonne facture et une fiabilité à l'image de ce géant du Web.

Mais là encore l'offre perd de son intérêt quand on regarde l'étroitesse du couloir des possibilités mises à disposition.

Microsoft Azure

Microsoft Azure, connu aussi sous le nom de Windows Azure est peut-être la seule véritable offre sérieuse de Cloud Computing du marché. C'est à la fois du PaaS et de l'IaaS qui permet de déployer des applications et des services en profitant du l'infrastructure réseau mondiale de Microsoft et de ses data centers. On trouve des services de natures très diverses, un support de

différents outils de développement, de langages, d'outils, de frameworks, le tout incluant bien entendu les produits Microsoft mais pas seulement.

Azure s'avère un service complet et large, plutôt réservé aux entreprises on s'en doute, bénéficiant du savoir-faire Microsoft qui maitrise toute la chaine, des OS aux langages en passant par les plateformes. On bénéficie aussi d'une ouverture intéressant puisqu'on peut créer des machines virtuelles Azure qui tournent sur Linux. Ainsi il est possible de développer des applications de tout type s'adaptant à la fois aux connaissances des équipes en place et aux besoins des utilisateurs : .NET, Java, PHP, Node.js, Python ou même Ruby sont supportés.

Visual Studio contient désormais des outils facilitant le développement, le débogue et le déploiement dans Azure ce qui propulse l'offre de Microsoft bien loin devant celle des concurrents.

Il est possible en suivant le lien donné d'accéder au site d'Azure qui contient des tas d'informations que je ne vais pas recopier ici, autant puiser à la source directement (et c'est en français). Il est même possible d'essayer Azure pendant un mois gratuitement.

Ensuite la facturation est assez souple et s'adapte aux volumes, aux temps de calculs, mais bien entendu on reste dans l'esprit même du Cloud : tant que tu payes ça marche, tu ne payes plus ça s'arrête.

Mais ce problème est global à toutes les offres puisque c'est la nature même du Cloud qui est ici incriminée. Car côté offre, Microsoft Azure est de loin celle qui est la plus intéressante techniquement.

Conclusion

Le Cloud est clivant d'un point de vue éthique. Techniquement c'est un progrès. Qui se passerait aujourd'hui de l'ubiquité des données qu'offre par exemple DropBox ? A peine une photo est prise par mon smartphone qu'elle se trouve déjà dans ma DropBox. J'arrive au bureau la photo est déjà synchronisée dans mon PC. Je pars avec ma tablette dans le jardin, la photo est là aussi. C'est génial.

Mais derrière cette façade de progrès évident se cache un monstre qui peut mettre fin à toute forme de liberté individuelle, qui peut transformer tous les consommateurs en vaches à lait, qui peut espionner vos amours, vos options politiques, et tout le reste. Le risque étant le même pour un particulier que pour une entreprise, seul le type des secrets changent…

Toutefois dans les offres que le marché nous propose, Azure se dégage nettement par sa versatilité, sa capacité à supporter de nombreuses configurations, son ouverture (Linux par exemple) et par sa scalabilité qui permet de maitriser le

monstre. En choisissant des solutions Azures de type IaaS ou PaaS on reste aux commandes de ce qu'on partage et de ce qu'on confit au Cloud. On peut parfaitement crypter les données stockées pour les protéger par exemple puisque tout depuis l'OS peut être sous contrôle.

Le SaaS est lui une autre affaire, c'est Office 365 et d'autres offres que Microsoft prépare dans ce sens puisque le Cloud est devenu leur cheval de bataille. Là je suis moins chaud j'ai expliqué pourquoi. Mais pour des développements dans le Cloud, Azure reste la meilleure plateforme.

Testez-là et faites vous votre propre avis, pour le reste cela regarde la conscience et l'éthique de chacun !

Les Emulateurs

Xamarin n'arrête pas de m'épater, un dynamisme, une inventivité qu'on aimerait bien voir partout. L'une des dernières nouveautés, un simulateur Android haute performance gratuit à télécharger…

Simulateur Android

Le SDK Android est bien fait et propose déjà un simulateur permettant depuis Xamarin.Studio ou Visual Studio d'exécuter des applications fraichement compilées pour les tester.

C'est pratique, pas besoin de transférer sur un vrai smartphone pour voir qu'il y a un bug…

Le simulateur Android de Google est un excellent produit qui a la bonne idée de marcher sur PC, pas comme le simulateur Apple qui ne marche que sur un Mac (ça rime avec arnaque en effet). Le seul problème de ce simulateur est sa relative lenteur.

Heureusement il existe HAXM de Intel qui permet d'accélérer grandement les choses.

Sauf qu'il y a un petit "hic" : HAXM a besoin de VT-X les fonctions de virtualisation en gros qui sont offertes par le processeur sous le contrôle de l'UEFI (ex Bios). Cela ne serait en rien gênant si Microsoft n'avait pas eu l'idée saugrenue de créer un émulateur Windows Phone qui ne fonctionne qu'avec Hyper-V… En dehors d'un Windows 8 Pro et autres contraintes (ils ne veulent vraiment pas que des

développeurs travaillent sur WP !) Et Hyper-V utilise les fonctions de virtualisation qu'il vampirise à son seul avantage.

Au final soit on développe pour Windows Phone et adieu le développement accéléré sous Android, soit on boote sans Hyper-V (j'ai publié une astuce pour cela regardez les archives) et là on peut jouir de HAXM mais on n'a plus du tout d'émulateur Windows Phone. Pas pratique quand on développe en cross-plateforme et qu'on doit passer d'une version WP à une version Android par exemple pour vérifier qu'une modification d'un ViewModel donne bien le même résultat partout. A noter qu'on a toujours l'émulateur Surface qui lui, bizarrement sait tourner sans Hyper-V. Quel pataquès ! Et dire que Microsoft parle beaucoup d'"unification", de "convergence" mais que dans les faits ils font tout le contraire en explosant les combinaisons et les enquiquinements. Un jour peut-être ils ne finiront pas comprendre qu'il vaut mieux unifier dès le départ, ça économise temps et argent à tout le monde.

Bref avec Windows Phone et le coup de Hyper-V soit on passe son temps à rebooter entre un W8 avec Hyper-V et un autre sans, soit on se passe de l'accélération sous Android ou on se passe de Windows Phone. Une situation plus qu'inconfortable pour le développeur et couteuse par la perte de temps et de performance que cela occasionne pour celui qui l'emploie.

Quelle autre possibilité ?

Xamarin Android Player

Soyons francs Xamarin n'a pas trouvé le moyen de se passer de l'accélération matérielle ni même de couper Hyper-V lorsqu'il est activé, seul un reboot permet de changer son état (encore une idée de génie super pratique).

En revanche Xamarin a réécrit un émulateur Android plus rapide que celui de Google et parfaitement adapté au debug des applications Xamarin.Android.

Bien entendu cela tournera d'autant plus vite si Hyper-V est désactivé.

Mais même dans le cas où Hyper-V est actif l'émulateur Xamarin "Android Player" est plus performant que l'original de Google. De fait la perte de HAXM se fait moins sentir et le temps de chargement de l'émulateur par exemple est beaucoup moins long. Sa réactivité est aussi meilleure.

Ainsi, Xamarin Android Player arrive à point pour faciliter les choses, qu'on développe uniquement pour Android (et qu'on puisse utiliser toute l'accélération possible) ou qu'on développe en cross-plateforme avec Windows Phone et Hyper-V.

Une fois l'installateur téléchargé et exécuté il n'y a rien à faire de particulier en dehors de valider certains dialogues notamment ceux qui concernent l'installation de VirutalBox de Oracle. Car bien entendu il y a une feinte, Xamarin utilise cet excellent émulateur pour son Player. Vous allez me dire, s'ils avaient utilisé Hyper-V la solution serait parfaite pour ceux qui doivent switcher en plein développement entre de l'Android et du Windows Phone. Certes. Mais je suppose que développer un émulateur sous Hyper-V devait être beaucoup moins portable que pour VirtualBox qui lui tourne aussi sur Mac par exemple… (mais aussi sous Linux et Solaris).

Une fois l'ensemble installé, cela va assez vite, il suffit de lancer l'application Android Player. Elle propose un premier écran qui permet de choisir un image à lancer ainsi que la liste des images disponibles en téléchargement. Pour l'instant il

existe deux images, des Nexus 4 avec deux API Android différentes, une plus ancienne et plus consensuelle et une plus récente.

Une fois les images téléchargées, elles s'installent dans VirtualBox automatiquement, c'est vraiment bien fait tout est automatique.

Ne reste plus qu'à exécuter l'une des images comme on le voit dans la capture écran ci-dessus.

C'est beau, c'est rapide, c'est du pur Android Nexus, du bon boulot qui va simplifier le nôtre !

Des fonctions bien utiles

Bien entendu l'émulateur possède une side bar qui offre de nombreuses options bien pratique pour tester une application. On retrouve (comme on le voit sur la capture verticale à gauche) le contrôle du volume sonore, la possibilité de prendre une capture écran, les touches habituelles de Android (retour arrière, menu, commutateur de tâches).

Inévitablement on trouve aussi un bouton pour effectuer une rotation de la device afin de tester les layouts dans toutes les positions.

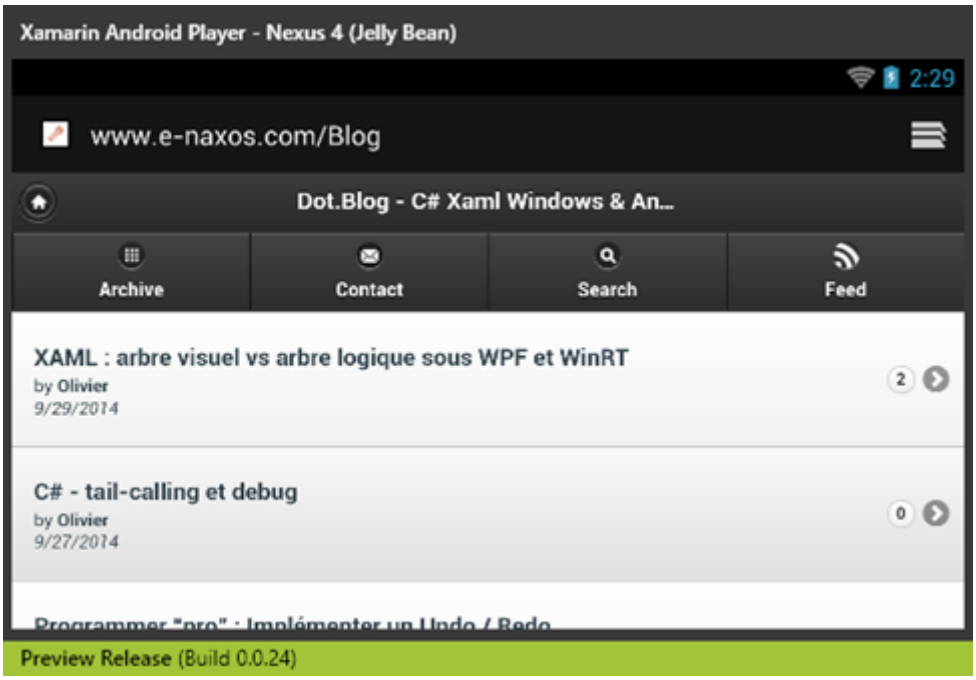
Le bouton des réglages donne accès aux informations de l'émulateur ainsi qu'à des onglets permettant de jouer sur l'état de la batterie (virtuelle) ou sur la position GPS.

Bref on dispose d'un ensemble tout à fait performant, ayant un look propre et une efficacité très satisfaisante.

La vitesse d'exécution même avec hyper-V donc sans accélération supplémentaire (VirtualBox le signal au lancement d'une image d'ailleurs) est vraiment bonne. Internet fonctionne rapidement, en tout cas pas moins vite que sur une vraie Device et c'est plutôt bien. Tester à la vitesse de l'éclair ce qui sera plus lent en réalité rend le développeur plus paresseux et il ne pense pas toujours à améliorer les performances. Avec une exécution proche de la réalité le développeur est le premier pénalisé s'il n'optimise pas son code et porte à cet aspect, généralement, plus d'attention ! C'est toujours plus enquiquinant quand on est le premier concerné, forcément.

Petit détail qui a son importance, le clavier du PC fonctionne parfaitement pour faire des saisies dans les applications. Ce n'est pas forcément le cas de tous les émulateurs, je ne citerai personne par charité… La preview reconnait le clavier Azerty mais semble avoir plus de mal avec les symboles mais pas avec les flèches

avant et arrière ce qui est très pratique là encore dans les saisies, je suppose que cela sera réglé dans la finale (sinon il reste bien entendu le clavier virtuel Android).



En mode paysage, l'affichage de la version mobile de Dot.Blog fonctionne très correctement et assez vite. Cet émulateur est vraiment bien fait.

En bricolant la position GPS j'ai réussi à le planter mais attention : c'est une Preview Release ! Elle a l'air tout à fait opérationnel et très fiable mais ce n'est pas une finale.

Où ?

Le programme se télécharge depuis le site de Xamarin depuis sa page dédiée qui vous en dira encore plus long que cette brève introduction. C'est ici : http://goo.gl/DVVoyi

Conclusion

J'admire vraiment le dynamise de Xamarin et sa capacité à innover en permanence et tout azimut. Il y a un réel potentiel créatif dans cette entreprise fondée par le créateur de Mono.

Xamarin vient de passer des accords avec IBM pour accélérer l'adoption de Android en entreprise notamment en se mariant facilement avec les équipements d'IBM. C'est une excellente nouvelle d'autant que Apple avait déjà conclu des accords similaires avec IBM.

J'aime Xamarin, j'aime leur état d'esprit conquérant, il se dégage de cette boite un vent nouveau, un parfum de winner qui donne envie de bosser, de faire des

choses. Franchement ça change d'autres ambiances plombées qu'on peut connaitre ailleurs.

C'est peut-être ça le secret du succès, avoir la patate, tout donner, et surtout : coller aux besoins des développeurs au lieu de se perdre dans des solutions dont personne ne veut. Un bel exemple à suivre en tout cas, et des supers produits que je vous conseille chaudement si vous n'êtes pas encore passé au cross-plateforme !

Le titre ne vous rappelle rien ? … Hein ? Si, l'article précédent ! Juste avec Microsoft à la place de Xamarin.

Joke ? Erreur de copier / coller ?

Nein !

En réalité Microsoft a aussi sorti son propre émulateur Android fourni avec VS 2015 ou téléchargeable à part.

Son intérêt ? C'est du Microsoft donc compatible avec l'environnement Microsoft… Avec VS mais aussi avec ce satané Hyper-V qui pose problème avec d'autres systèmes d'émulation.

Avec le rachat par Microsoft de Xamarin je ne suis pas sûr que l'émulateur de ce dernier existe encore longtemps… En revanche celui de Microsoft marche très bien surtout sur PC et avec Hyper-V je vous le conseille, c'est celui que j'utilise désormais.

Pour celui de Xamarin, l'avenir nous dira si Microsoft élargi sa stratégie cross-plateforme en conservant des outils de développement cross-plateforme. Par exemple Xamarin Studio qui copie VS est largement moins puissant que ce dernier mais il peut tourner sur un Mac… Idem pour l'émulateur Android Xamarin.

Microsoft maintiendra-t-il ces outils qui font doublons avec les siens pour être présents directement chez Apple ou bien les enterrera-t-il pour optimiser les couts ?

Nous le saurons en suivant les prochains épisodes de l'année 2016 !

Xamarin Store, des composants et des plugins XF

Je vous propose de découvrir le Xamarin Store et de nombreux composants incroyables et souvent gratuits qui offrent des services indispensables aux

applications modernes. Ayant évolué au fil du temps le XS propose aussi bien des composants natifs pour certains OS que désormais des plugins Xamarin.Forms.

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). Heureusement avec le temps et la sortie des XF on trouve désormais une section « Plugins » dédiée aux Xamarin.Forms et couvrant le plus souvent l'ensemble des plateformes (UWP qui a été ajouté dernièrement n'est pas encore couvert par tous les plugins).

Si le store ne contient pas encore des milliers de composants on y trouve beaucoup de choses intéressantes qui peuvent transformer une application banale en app moderne, utile et attractive.

Quatre 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. Un tel designer devrait bientôt voir le jour pour les Xamarin.Forms.

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, UWP et ASP.NET peuvent être mélangés sous Visual Studio (Xamarin Studio gère de l'ASP.NET mais pas WPF ou SL ni UWP).

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 factors.

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. Avec les Xamarin.Forms on peut même décider de cibler toutes les plateformes d’un seul coup !

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”…

Soutenir Windows Phone et UWP

En réalité les applications du type que je viens de décrire bien loin de nous éloigner de Windows Phone ou même de UWP 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. Le mieux étant de choisir les Xamarin.Forms et de cibler UWP qui lui-même est un ensemble d’API cross-plateforme dans le monde MS (ciblant alors les smartphones, les PC, les IoT, la Xbox etc).

La démarche cross-plateforme que j’ai longtemps proposée permettait 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. Avec les Xamarin.Forms l’effort devient quasi nul. Je suppose d’ailleurs que les intentions de Microsoft dans le rachat de Xamarin sont plutôt à chercher dans cette partie de

billard. Je vante Android et iOS mais je pousse en réalité la mise sur le marché d'Apps UWP…

UWP est très proche de Silverlight et de WPF, technologies C#/Xaml auxquelles Dot.Blog a réservé près de 90% de son espace sur les près de 900 billets (oui !) publiés à ce jour. Si je n'en parle pas beaucoup plus c'est que toutes les techniques utilisables sous UWP 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, c'est un OS qui meurt pour laisser la place à Windows 10 donc à UWP qui utilise C# et Xaml présentés en long et en large par Dot.Blog.

Le cross-plateforme est l'avenir du développement. Je l'écrivais de cette façon non ambigue il y a déjà plusieurs années. 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 dans ce livre car c'est l'un des OS principaux à connaître à côté de Windows 10 proposé par Microsoft. Il y a deux véritables géants : Microsoft dans le monde PC, Android dans le monde mobile. Si vous balayez ces deux mondes, vous en êtes le roi !

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, Windows Phone et maintenant UWP !

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 des plugins dédiés à Xamarin.Forms. Les composants spécialisés pour Android ou iOS sont tout aussi essentiels mais le présent ouvrage couvre avant tout les Xamarin.Forms …

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

Battery plugin

Accéder au hardware spécifique de chaque plateforme tout en restant universel est bien la base de l'approche des Xamarin.Forms. Bien que spécifiques de nombreuses facilités existent en réalité sur tous les mobiles. Il en va ainsi du niveau de la batterie…

C'est tout bête mais connaître le niveau de batterie peut être essentiel et cela réclame de savoir accéder à l'information dans tous les OS. Pas si simple au final et surtout un peu long.

Avec ce plugin, comme les autres, on accède à des fonctions natives de façon universelle…

https://components.xamarin.com/view/BatteryPlugin

On peut ainsi connaitre le niveau actuel de la batterie mais aussi le statut de chargement, le type de chargement et s'abonner aux évènements de la batterie.

Connectivity Plugin

Dans le même esprit le Connectivity plugin permet de prendre connaissance de l'état de la connexion de la device à l'extérieur : vitesse de la bande passante, connexion présente ou non, types de connexions disponibles, test du host, abonnement aux évènements de connectivité.

Tout cela n'a rien d'exceptionnel mais cela peut transformer une UX banale en une bonne UX… Le tout sans se plonger dans le manuel de chaque OS !

https://components.xamarin.com/view/ConnectivityPlugin

File System plugin

Pouvoir accéder au système de fichiers de façon unifiée est un gros avantage. Ce plugin ne fait donc rien d'exceptionnel techniquement mais l'avantage qu'il procure en offrant un moyen universel de gérer les fichiers est exceptionnel !

https://components.xamarin.com/view/pclstorage

Exemple d'utilisation :

```
public async Task CreateRealFileAsync()

{

    // get hold of the file system

    IFolder rootFolder = FileSystem.Current.LocalStorage;


    // create a folder, if one does not exist already

    IFolder folder = await rootFolder.CreateFolderAsync("MySubFolder",
CreationCollisionOption.OpenIfExists);


    // create a file, overwriting any existing file

    IFile file = await folder.CreateFileAsync("MyFile.txt",
CreationCollisionOption.ReplaceExisting);


    // populate the file with some text
```

```
    await file.WriteAllTextAsync("Sample Text...");
}
```

Settings Plugins

Accéder aux préférences de chaque machine de façon native est un énorme plus dans certaines applications. Le faire sans avoir à en connaitre les détails pour chaque OS est un avantage gigantesque…

https://components.xamarin.com/view/SettingsPlugin

Exemple :

```
private static ISettings AppSettings
{
  get
  {
    return CrossSettings.Current;
  }
}

private const string UserNameKey = "username_key";
private static readonly string UserNameDefault = string.Empty;

public static string UserName
{
  get { return AppSettings.GetValueOrDefault(UserNameKey, UserNameDefault);
}
  set { AppSettings.AddOrUpdateValue(UserNameKey, value); }
}
```

Messaging plugin

Envoyer des mails, des sms, passer des appels, c'est effectivement le minimum avec smartphone… Mais le faire de façon générique sans se soucier des détails de chaque plateforme il n'y a que grâce aux Xamarin.Forms et ses plugins que cela est possible !

https://components.xamarin.com/view/Xam.Plugins.Messaging

Exemple :

```
// Make Phone Call
var phoneDialer = CrossMessaging.Current.PhoneDialer;
```

```
if (phoneDialer.CanMakePhoneCall)
    phoneDialer.MakePhoneCall("+272193343499");


// Send Sms
var smsMessenger = CrossMessaging.Current.SmsMessenger;
if (smsMessenger.CanSendSms)
   smsMessenger.SendSms("+27213894839493", "Well hello there from
Xam.Messaging.Plugin");


var emailMessenger = CrossMessaging.Current.EmailMessenger;
if (emailMessenger.CanSendEmail)
{
    // Send simple e-mail to single receiver without attachments, bcc, cc
etc.
    emailMessenger.SendEmail("to.plugins@xamarin.com", "Xamarin Messaging
Plugin", "Well hello there from Xam.Messaging.Plugin");


    // Alternatively use EmailBuilder fluent interface to construct more
complex e-mail with multiple recipients, bcc, attachments etc.
    var email = new EmailMessageBuilder()
      .To("to.plugins@xamarin.com")
      .Cc("cc.plugins@xamarin.com")
      .Bcc(new[] { "bcc1.plugins@xamarin.com", "bcc2.plugins@xamarin.com"
})
      .Subject("Xamarin Messaging Plugin")
      .Body("Well hello there from Xam.Messaging.Plugin")
      .Build();


    emailMessenger.SendEmail(email);
}
```

Conclusion

Le Xamarin Store contient bien d'autres choses que je vous incite à découvrir par vous-même. Composants visuels dédiés à une plateforme spécifique, librairies de code comme JSON.NET, on trouve de véritables trésors. 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 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 …

Retour Visuel pour designer les Xamarin.Forms

Malgré les progrès essentiels des Xamarin.Forms et le rachat très récent de Xamarin par Microsoft il reste un point gênant : le pseudo-XAML utilisé par les XF ne dispose pour l'instant pas de designer visuel. Mais on peut ruser !

Du C# ou du XAML

Les Xamarin.Forms sont une couche cross-plateforme qui permettent de décrire les pages d'une application d'une telle façon qu'elles seront compilées et utilisables sur toutes les plateformes à la fois. C'est l'unification des IHM après celle du code.

Pour coder les Xamarin.Forms il existe au départ un moyen simple : C#. On créer les conteneurs et leur contenu comme on peut écrire du XAML entièrement en C# aussi.

La logique des hiérarchies de classe étant claires on peut écrire en C# un code "visuel" assez facilement lisible. Mais rien ne vaut le retour visuel !

Les Xamarin.Forms ont évolué en proposant la possibilité de décrire les pages non plus en code C# mais en code "XAML", un XAML portable donc simplifié (par exemple ne cherchez pas comment faire des courbes de Béziers, Windows traite le vectoriel pas Android ni iOS à ma connaissance).

On en arrive donc à écrire des pages XAML qui ressemblent beaucoup à du XAML standard, puisque Xamarin y a même ajouté le Binding.

Tout est là pour travailler complètement en C#/XAML et produire avec un seul code des applications cross-plateformes !

Pas de retour visuel

On peut supposer que la prochaine étape logique sera d'ajouter un designer visuel à Xamarin.Forms. Xamarin l'a fait pour Android et iOS, il n'y a aucune complexité à le faire pour les XF. D'autant que Xamarin venant d'être racheté par Microsoft, ces

derniers auront certainement à cœur de rendre l'utilisation de Xamarin encore plus intégrée à Visual Studio.

Mais pour l'instant il n'y a pas de retour visuel que l'on créé des Xamarin.Forms en C# ou en XAML.

Attention, ce retour visuel existe si vous créer explicitement une IHM Android ou même iOS, et ce sous Xamarin.Studio aussi bien que Visual Studio. Je parle donc ici d'un problème spécifique aux Xamarin.Forms.

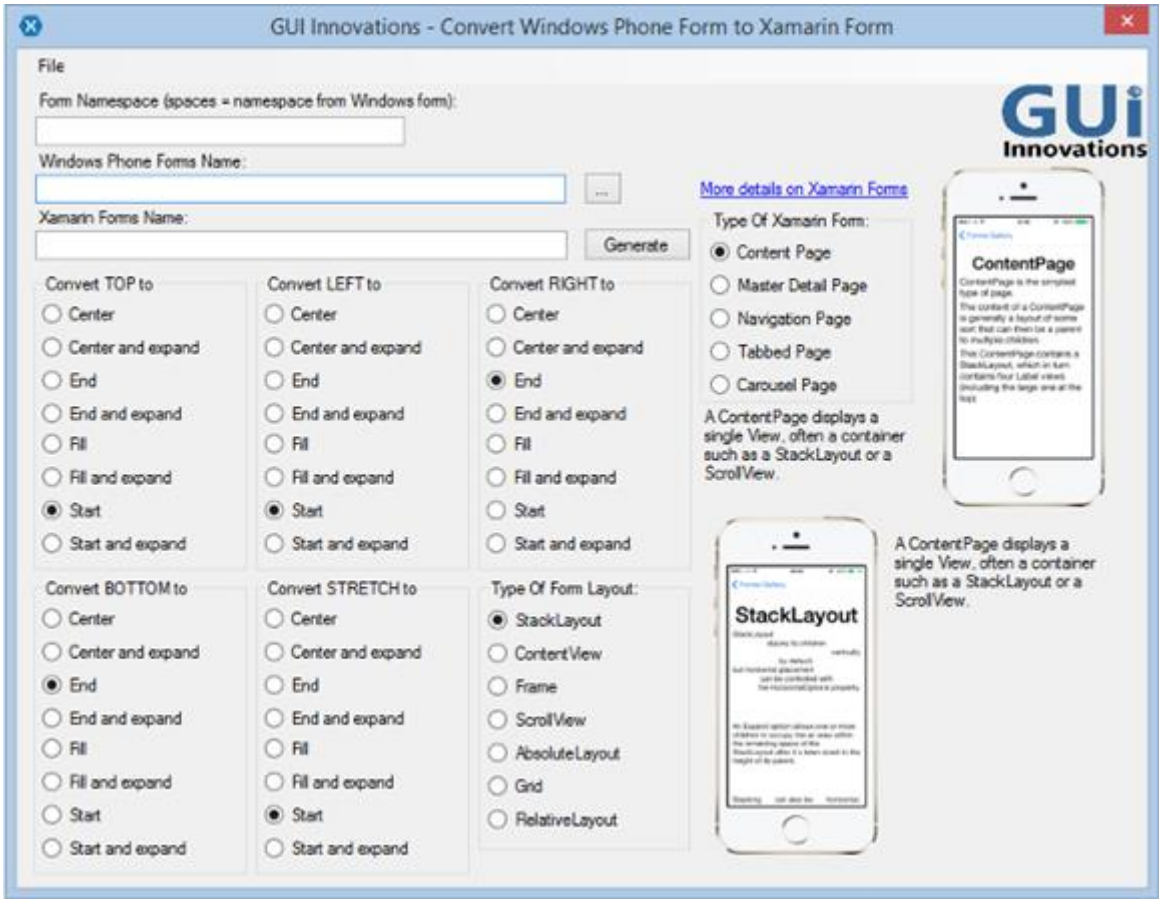
Contournement

La proximité du pseudo-XAML des XF avec le vrai XAML est telle que fournir un designer visuel sera certainement assez facile, d'où l'espoir assez grand de voir un jour cette extension apparaitre. Mais entre temps existe-t-il un moyen de "ruser" ?

Oui. Et une fois encore grâce à une âme charitable qui s'est frottée à la mise au point d'une moulinette Windows Phone vers Xamarin.Forms !

GUI-Innovation propose en effet un outil gratuit qui transforme une fiche XAML Windows Phone en Form Xamarin.Forms. Et ce en laissant certains choix à la discrétion du développeur selon le résultat qu'il souhaite obtenir.

C'est simple, encore en bêta mais fonctionnel.



Pour vous procurer ce petit outil bien pratique : http://goo.gl/zYyJgq

Conclusion

Il devient donc possible d'écrire une IHM Xamarin.Forms de façon visuelle en créant tout simplement une page Windows Phone et en la traduisant ensuite…

D'une part ce n'est pas le cycle de production rêvé et d'autre part il faudra certainement ici ou là faire quelques retouches. Mais cela est tout de même bien plus simple pour composer une page d'avoir un retour visuel au moins pour les grandes lignes.

Connaitre les OS

Windows Phone

Cet OS est voué à disparaître au profit de Windows 10 / UWP qui englobe bien plus que les smartphones puisqu'il se destine à tous les form-factors du PC au Hololens, aux IoT, tablettes etc.

Il est bien entendu assez peu raisonnable d'envisage de traiter en quelques chapitres toute l'étendue de cette plateforme. D'autant que le lecteur de Dot.Blog connait déjà bien le monde Microsoft ou bien qu'il possède déjà les clés lui permettant de creuser ses connaissances.

C'est pour cela que je n'aborderai ici que Android. Dans une révision de ce présent livre j'ajouterai peut-être une même introduction à iOS. La manipulation de ce dernier réclame malgré tout de posséder un Mac, un iPhone ou même IPad, autant d'investissements qui ne se font pas à la légère et qui supposent là encore que celui qui est concerné par ces achats connait un peu l'OS et son esprit. L'urgence me semble donc plus du côté d'Android, par sa place domination du marché, et par la simplicité de développer pour lui plus que pour tout autre.

UWP

Comme je l'ai dit plus haut UWP est une plateforme d'une grande richesse dont les tentacules s'inflitrent dans tous les form-factors avec à chaque fois quelques spécificités propres au hardware concerné. Diffile de traiter ici un tel monstre sans être réducteur. Un livre de la série All.Dot.Blog est consacrée au développement UWP j'y renvoie le lecteur intéressé.

iOS

iOS mériterait la même approche que Android mais il faut bien commencer par quelque chose et ce livre pour coller à l'actualité ne peut s'attarder sur tous les OS surtout dans sa première édition. Avec 25% de part de marché iOS ne peut pas être négligé quoi qu'on en pense. Mais pour développer il faut absolument un équipement qu'un développeur Microsoft ne possède pas forcément. Il me semble donc plus logique de parler de Android pour commencer, ce que chaque lecteur pourra expérimenter rapidement pour presque rien, plutôt que de verser dans l'élitisme de la Pomme en vous forçant avant toute chose à sortir le chéquier… Mais quoi que je pense de Apple et de ses produits, leur place sur le marché est respectable et méritera un jour de s'y arrêter plus longuement.

Android

Partie 1 – Présentation

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 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 puis les suivants 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'autres outils extraordinaires MvvmCross fut l'un des premiers et les Xamarin.Forms sont le diamant au somment de la pyramide…

Mais tout cet outillage ne fait pas tout. MonoTouch et MonoDroid (Xamarin 2.0 puis Xamarin.Droid, …) et MvvmCross ne sont que des passerelles, des facilitateurs. Même avec les Xamarin.Forms il faut nécessairement comprendre les OS ciblés pour faire de vrais développement cross-plateforme et je ne parle pas du développement de plugins cross-plateformes !

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 UWP.

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 un 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.

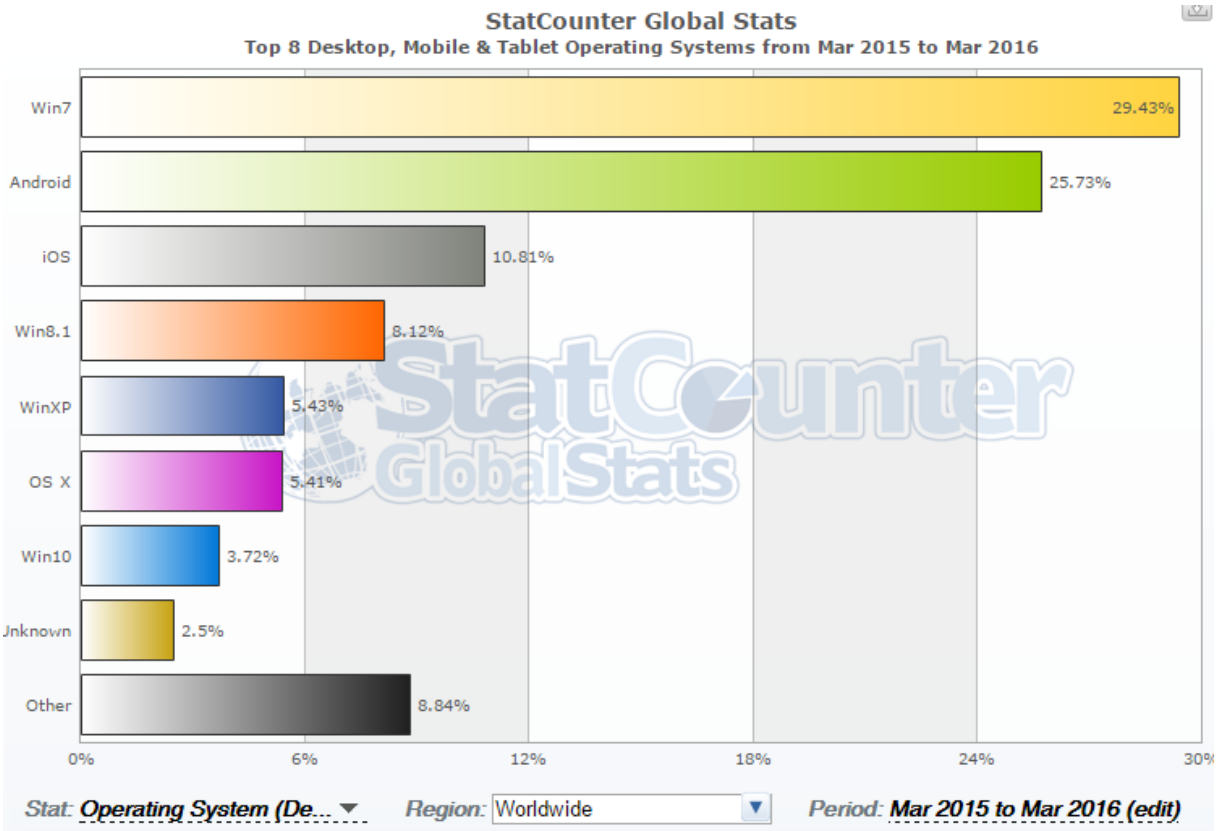
Pourquoi Android ?

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 et outils 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 de 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 fait.
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 période mars 2015-2016 desktop et mobile



Ce graphique montre le Top des OS dans le monde sur la période mars 2015-2016. Vous pouvez obtenir les chiffres exacts sous différentes formes (fichier csv, graphique…) en vous rendant sur le site de StatCounter.

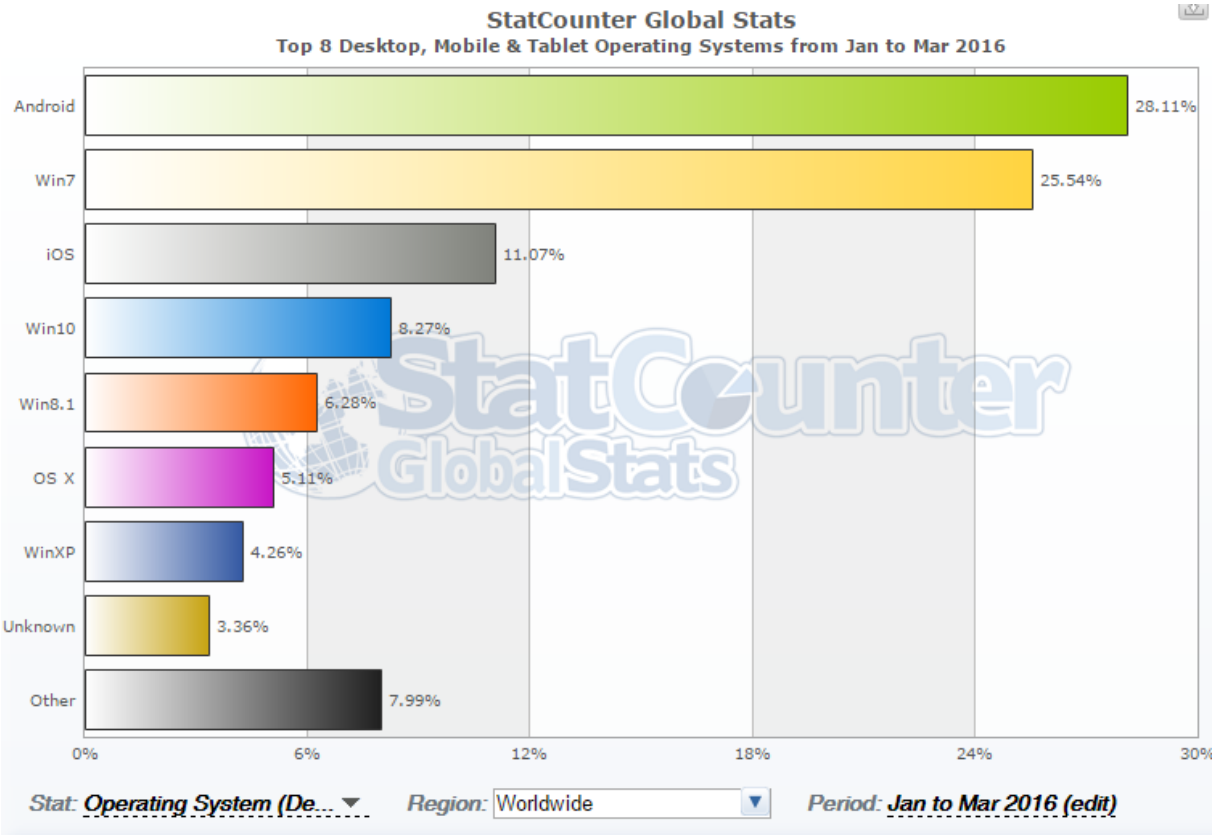
On voit clairement sur cette période la domination de Windows 7, suivi de très près par … Android et iOS. XP se traîne maintenant loin même derrière Widonws 8.1. On voit même pas Linux ou tout ces autres OS exotiques qui pensent un jour prendre le pouvoir (quoi que Android est un Linux mais vendu sans le dire, alors ça ne compte pas d'avancer masquer…).

Ca c'est le marché global des OS sur la dernière année, celle qui compte car elle pèsera lourdement sur la prochaine ! La répartition dans le monde de ces OS et leur poids réel sont là sous nos yeux pour prendre des décisions (à quelques pourcents près, chaque fournisseur de statistiques ayant ses propres biais).

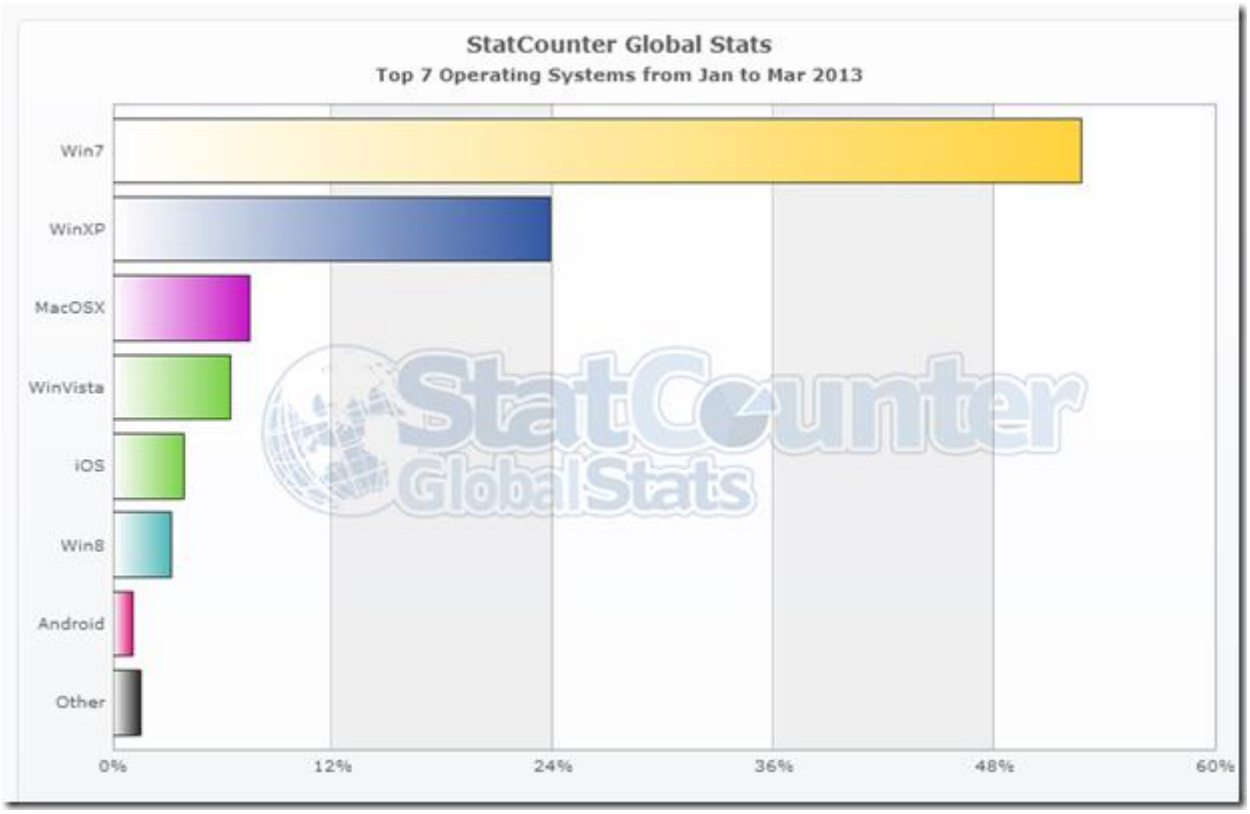
Répartition du marché des OS sur les 3 derniers mois

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 ce chapitre (avril 2016) les trois derniers mois, de janvier à mars 2016, donnent la répartition suivante:



Pour info je vous laisse ci-dessous le même graphique pour la même période en 2013… Qui l'u cru hein ?



Dans les deux dernières années le pelont de tête s'est modifié de la sorte :

- 2013 1-2-3
  - Win7
  - WinXP
  - MacOsX
  - Vista
  - iOS
  - Android

- 2016 1-2-3
  - Android loin devant
  - Win7 qui tente de rester dans la course
  - iOS qui s'accroche aux branches
  - Windows 10 qui perce timidement
  - Win8.1 qui fait de la résistance
  - OSX qui perd progressivement du terrain
  - Et le reste dont on ne donne plus les noms (par charité).

On peut aussi chercher la part des marchines non pas dans leurs ventes mais dans leur pourcentage d'utilisation du Web ce qui est une autre vue intéressante.

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

2016	Win10	Win8	Win7	Vista	NT*	WinXP	Linux	Mac	Chrome OS	Mobile
February	17.8%	15.2%	43.1%	0.4%	0.0%	2.1%	5.6%	10.4%	0.2%	5.2%
January	17.0%	15.8%	43.2%	0.4%	0.0%	2.2%	5.7%	10.2%	0.2%	5.4%

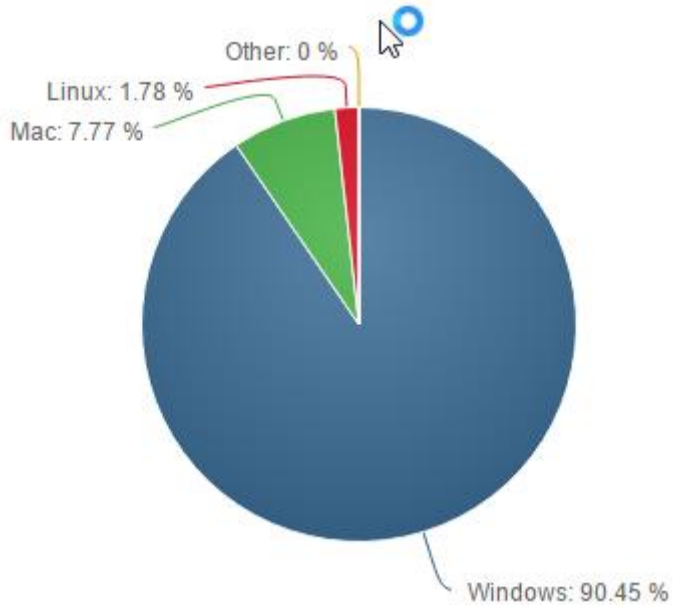
Comme pour le précédent graphique je vous laisse celui de 2013 pour comparaison ci-dessous

OS Platform Statistics

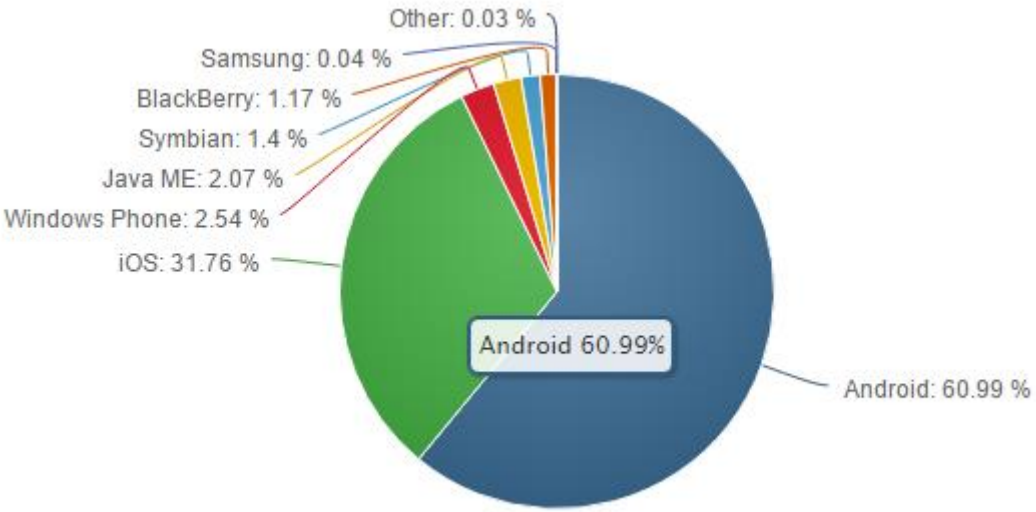
2013	Win8	Win7	Vista	NT*	WinXP	Linux	Mac	Mobile
March	6.7%	55.9%	2.4%	0.4%	17.6%	4.7%	9.5%	2.3%
February	5.7%	55.3%	2.4%	0.4%	19.1%	4.8%	9.6%	2.2%
January	4.8%	55.3%	2.6%	0.5%	19.9%	4.8%	9.3%	2.2%

On le constate, malgré le "buzz" autour des mobiles, ces derniers ne pèsent pas encore très lourd par rapport au monstre qu'est Windows et au monde des PC en général et ce malgré un doublement (de 2/3% à 5,5% environ pour les mobiles).

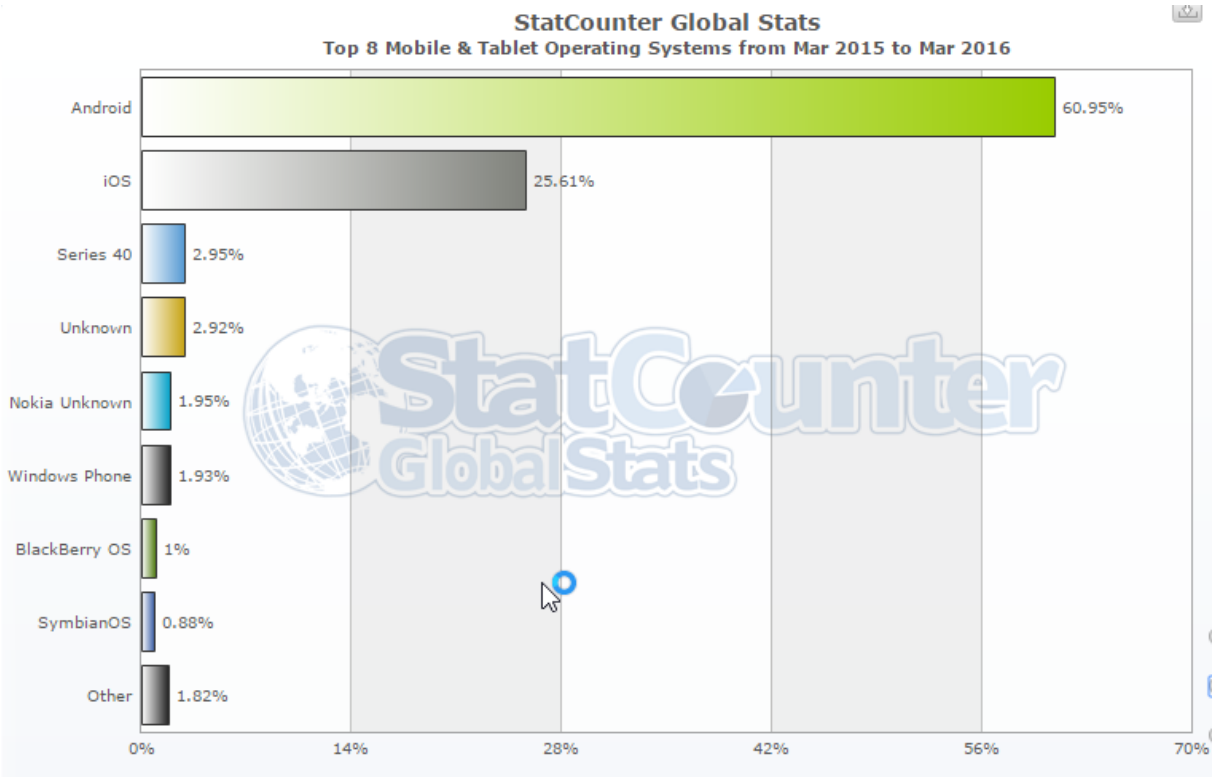
Une autre vision arrêtée à mars 2013 (source NetMarketShare) pour le marché global (desktop share) :



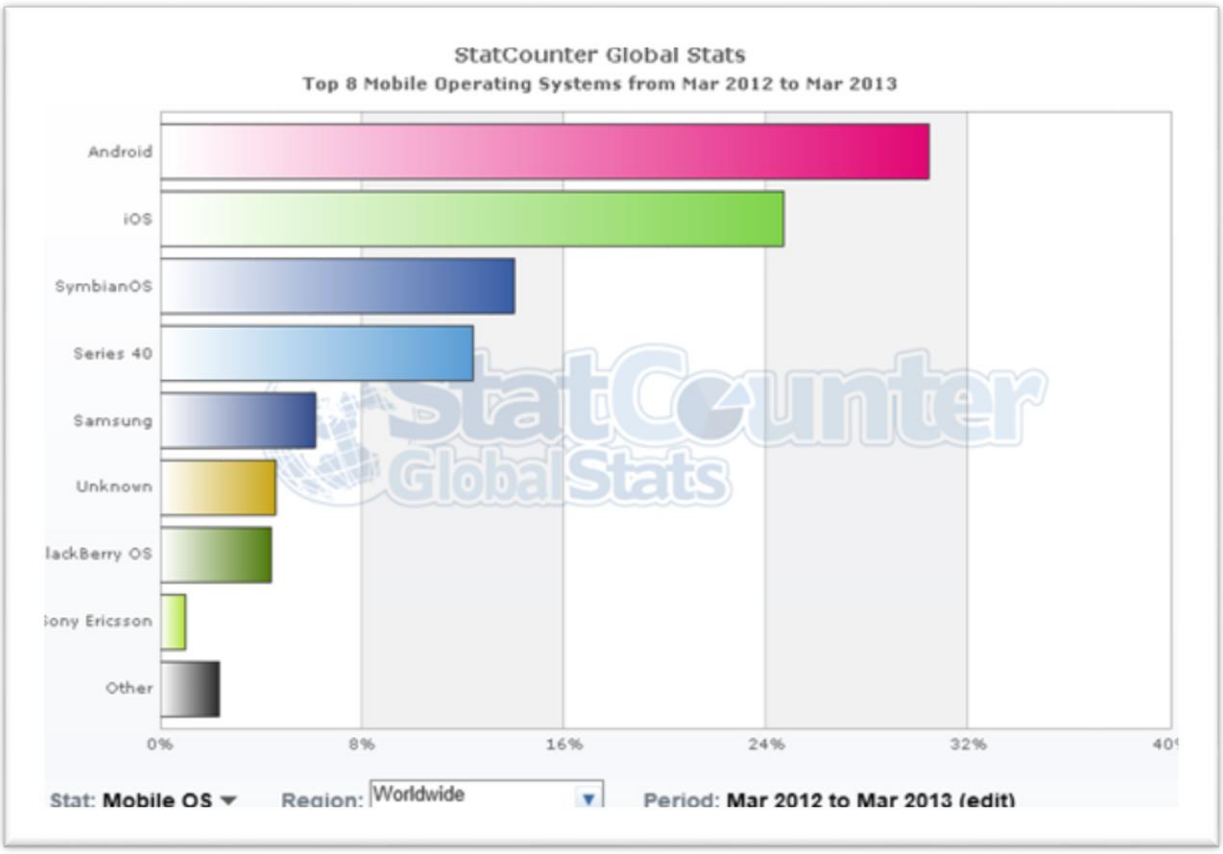
La même analyse pour les mobiles (smartphones et tablettes) :



Répartition des OS Mobiles sur l'année 2015-216



Ci-dessous la même analyse en 2013



Sur l'année écoulée, en cumul, Android est bien largement en tête à près de 61% du marché (comme 30 en 2013 !). iOS après une chute incroyable arrive désormais à se maintenir à 25% mais sans progression. Parti de presque 100% et stagner à ¼ du marché n'est pas une gloire mais ce n'est pas non plus à négliger selon les cibles visées par vos Apps !

On ne parle de tout ce qui se trouve en dessous, à un niveau non significatif et ne pouvant jouer aucun rôle sur ce marché en l'état.

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 mais difficilement mesurable pour être corrigé.

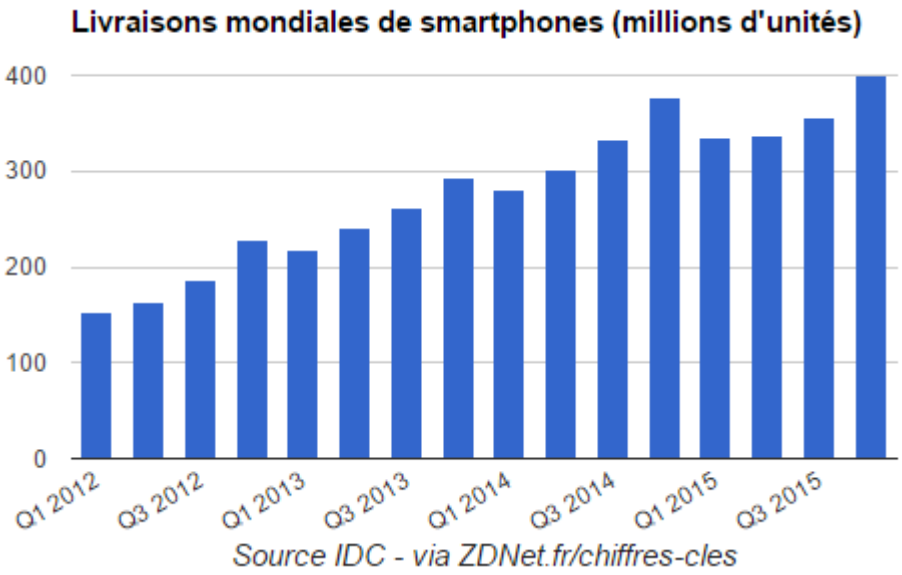
Il faut donc une autre source d'information basée sur d'autres méthodes, notamment les ventes de matériel. Ici aussi il existe un biais connu : les ventes livrées aux magasins (physiques ou en ligne) ne sont pas toujours égales aux achats des consommateurs. Mais les vendeurs faisant généralement bien leur métier il est rare (mais pas impossible) qu'ils achètent des quantités énormes d'un produit qui ne se vendra pas du tout. Mais il faut être conscient de cet écart entre vendu aux boutiques et vendu aux clients.

En prenant les chiffres de IDC on trouve pour le Q2 2015 comparé jusqu'en 2012 :

Period	Samsung	Apple	Huawei	Xiaomi	Lenovo*	Others
2015Q2	21.4%	13.9%	8.7%	5.6%	4.7%	45.7%
2014Q2	24.8%	11.6%	6.7%	4.6%	8.0%	44.3%
2013Q2	31.9%	12.9%	4.3%	1.7%	5.7%	43.6%
2012Q2	32.2%	16.6%	4.1%	1.0%	5.9%	40.2%

Source: IDC, Aug 2015

** Motorola figures have been captured under Lenovo.*



Cette vision est assez différente de l'analyse du Web. Elle montre que sur les 400 millions de machines vendues seuls 13,9 % sont de l'Apple… Le reste est pour ainsi dire que de l'Android avec SamSung (qui fait mieux que Apple à lui tout seul), Huawei, etc…

On aimerait se réjouir de voir Windows Phone faire une petit signe dans ces graphiques mais il n'en est rien, ce qui conditionne pour beaucoup le changement de stratégie de Microsoft à mon avis (rachat de Xamarin, fournir des apps sur Android et iOS etc).

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 obligés de constater aussi la défaite des autres OS et l'enterrement pas même de luxe de Windows Mobile.

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 force en jeu, les chiffres des ventes de IDC viennent écraser toute forme de relativisme…

Android n'arrête pas sa progression.

Mais après avoir été détrôné de la première place on s'aperçoit que iOS s'accroche plutôt bien et refuse de céder plus de terrain en se maintenant à 20/25% du marché environ.

Ce sont donc tous les autres OS mobiles qui pâtissent désormais de la montée d'Android. Apple à beaucoup perdu et va rester certainement stable encore un moment autour de son point actuel. En revanche 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 Windows Phone 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. 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…

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

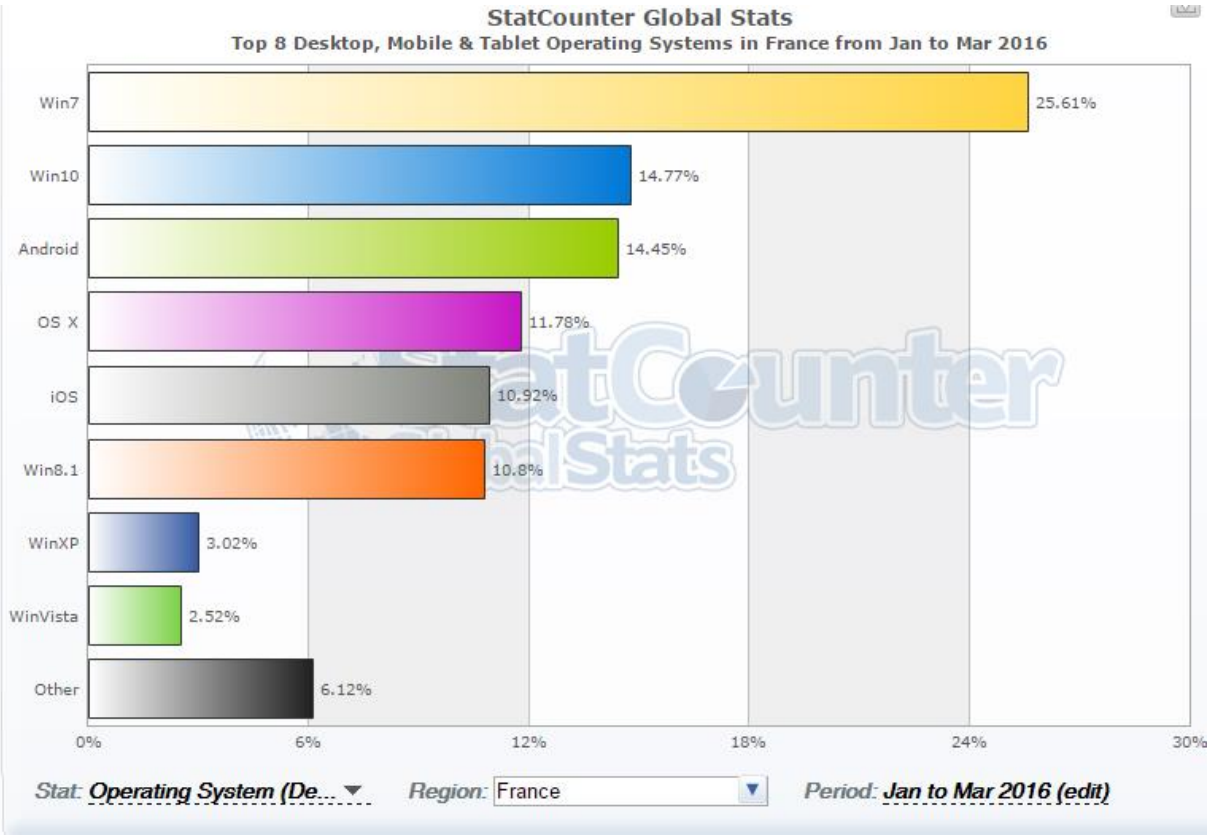
- Win32/Win64 pour les PC, en C#/Xaml donc WPF ou UWP
- Android de Google pour les smartphones, avec MonoDroid/Xamarin en C#
- Android et iOS d'Apple pour les tablettes avec MonoDroid et MonoTouch/Xamarin en C#
- Ou le choix des dieux, Xamarin.Forms qui adresse UWP (PC et Windows Mobile), Android et iOS le tout en C# et XAML !

Vous remarquerez que je n'inclue pas d'emblée l'IPhone dans cette démarche. La France est un cas à part où le marché du smartphone est aujourd'hui dominé par Android plus qu'ailleurs dans le monde. Si on développe en France pour des français, viser autre chose qu'Android est un investissement à bien mesurer car peu rentable. Néanmoins certains marchés peuvent le nécessiter. Vous remarquerez aussi que pour les tablettes j'intègre cette fois-ci l'IPad, tout simplement par sur ce form factor c'est Apple qui domine encore le marché français.

Malgré tout, iOS ne peut être évacué pour certains développements smarphones et doit être pris en compte pour les développements tablettes. Mais la montée irrésistible de 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 de 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 plus de 50% le marché suivi désormais par Windows 10 et non plus XP qui disparait petit à petit, talonné non plus par Vista à 12% et Mac OSX à 10% comme en 2013 mais bien par Android et OS X à 15 er 12 % environ !. Le paysage a beaucoup changé en 2 ans Windows 8 arrive ensuite à 10 % environ, ce qui double sa présence sur 2 ans. C'est à dire que pour les PC, en France, il y a près de 57% de machines faisant tourner Windows contre 77% il y a 2 ans …

Il y a deux ans j'en concluais que même si cela me faisait de la peine il était préférable de déconseiller WinRT. Combien j'avais raison…

Heureusement Windows 10 avec UWP prend une place importante et grâce à la nouvelle platefome VS+Xamarin développer Microsoft c'est aujourd'hui la garantie de dévleopper pour le monde entier.

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 et qui surtout échappe à la taxation de 30% du Store dont aucun grand éditeur ne veut.
- Visual Studio, même pour le développement sous iOS ou Android grâce à Xamarin intégré 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. Même si pour les mobiles SQLite est la référence (mais ne joue pas dans la même catégorie toutefois),
- Windows 10 et UWP en tant qu'OS rapide et stable et vraiment agréable s'ouvrant à un mode réellement universel (UWP)
- 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 maintenu jusqu'en 2022 mais encore faut-il convaincre les clients !
- Office, SharePoint, Blend, …

Il y a deux ans je disais que parler de développement cross-plateforme en restant cohérent par rapport au marché, cela consistait donc à créer des logiciels qui fonctionnent sous WPF côté PC et sous Android pour le reste. iOS pouvant être pris en compte pour certaines niches (médical, musique, graphistes…).

En 2016 je maintiendrais ce conseil avec quelques précisions : WPF pour toucher tous les PC du monde et surtout échapper au Store et sa taxe énorme, ensuite Xamarin.Forms bien entendu carde là découle automatiquement UWP, Android et même iOS.

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. D'autant que je constate que Microsoft a « suivi mon conseil » édicté il y a plusieurs années : favoriser le développement cross-plateforme pour rendre automatique la

production d'Apps Windows Phone, seule possibilité de sauver ces devices pour MS.

Même parler de Xamarin aujourd'hui c'est parler de Microsoft, sans avoir à faire des tournures de phrases ménageant la chèvre et le choux.

Enfin libres !

Microsoft devient raisonnable et nous offre les moyens de nous lâcher tout en restant loyal ☺

Dès lors, pour tout ce qui est mobile, et pour les raisons exposées plus haut je ne peux que conseiller d'utiliser Xamarin.Forms qui répond à TOUTES les configurations et opportunités du marché.

Voilà pourquoi je vais vous parler encore d'Android et que je continuerai à parler de UWP ou WPF.

Vais-je me taire au sujet de WinRT ? Oui. L'heure est venue de mettre un terme à cette API héritée d'un autre temps et qui n'a jamais intéressé mes lecteurs…

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 même en XAML aujourd'hui. 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.

On le voit, aujourd'hui Android est une force montante. Cet OS a interdit à Microsoft de se battre contre Apple et son IPhone en venant prendre la 2ème place, 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 veut chasser sur les terres sacrées de MS. Tous les indices en notre possession montre que Android n'a pas encore terminé sa percée. Les mêmes indices nous prouvent que Microsoft a peut-être compris la leçon et le revirement cross-plateforme dans ce contexte semble la réponse la plus appropriée.

Déjà il y a deux ans je disais :

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.

On ne pourra pas dire que je conseille dans le vide…

Le choix d'Android et UWP via les XF 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… Microsoft par le rachat de Xamarin vient de mettre fin à une guère de tranchées qui nous obligeait à faire des choix difficiles maintenant la peur, celle de faire le mauvais choix. Cette peur s'efface. Une nouvelle ère de stabilité s'ouvre, cela faisait si longtemps qu'on l'attendait !

"*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/UWP et les Xamarins.Froms choisissons le côté clair de la force.

Partie 2 – L'OS

Dans la première partie 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 de Android, on s'en moque un peu, comme 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 eux restés 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".

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 des unités mobiles (smartphones et tablettes) sous un pseudonyme : **Android**. Abandonnant au passage sa connotation C pour se tourner vers un Java que lui conteste vivement Oracle aujourd'hui. Android est donc un bric-à-brac de trucs empruntés plus ou moins légalement à droite et à gauche pour se faire du fric grâce à la pub. Aussi incroyable que cela puisse paraître ça a marché !

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 et que cela ne suffit même pas pour bloquer toutes les failles.

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.

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. C'est un OS d'unité mobile donc présentant des similarités avec iOS ou Windows Phone. Les nuances se trouvent dans le jargon et quelques APIs et plus particulièrement dans la gestion de l'affichage.

Vecteur ou bitmap ?

Le vectoriel c'est le top. Une définition courte de quelques points 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"… 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 sauf Microsoft. Et lorsqu'on voit la fluidité de WPF ou même de Silverlight sur Windows Phone 7 ou de UWP sous Windows Phone 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. D'autant que 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 de plusieurs giga).

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.

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.

Bien entendu je m'arrête ici aux vraies différences natives. Avec les Xamarin.Forms toutes ces machines se programment avec C# et XAML (une version non vectorielle) et on se pose fatalement mois de question.

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 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 . Le code s'écrit et se teste en utilisant Eclipse. Un classique des développeurs Java. Il existe d'autre IDE et depuis la version 5 Dalvik est remplacé par Android Runtime. La fameuse « fragmentation » de cet OS dont il existe des millions de machines dans le monde n'ayant que finalement peu de choses en commun…

Côté langage nous disposons donc d'un standard de type Java et d'un EDI pas si mauvais, même si nous qui connaissons Visual Studio le trouvons un peu préhistorique.

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 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 à l'origine 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 VS. Xamarin Studio est cross-plateforme on peut donc développer sur une machine Mac ou Linux et pas seulement sur PC. Avec le rachat de Xamarin par Microsoft le produit va être progressivement préinstallé dans Visual Studio il n'y aura même plus besoin de penser à ce détail.

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... Et avec l'ajout des Xamarin.Forms c'est même C# et XAML qui vont permettre de créer un seul code pour iOS, Android et UWP…

Les Xamarin.Labs et les plugions pour XF vont simplifier encore plus le partage de code et l'exploitation de code natif de façon transparente pour le développeur.

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.Forms, c'est à dire en C# / XAML 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 ou MvvmLight ou même à d'autres librairies appuyant XF nous pourrons appliquer sereinement MVVM et concevoir des applications véritablement professionnelles.

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. 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.

C'est d'ailleurs ce qui a fait en partie le succès de 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, 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 de 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 6. Cela ne gène pas vraiment les

applications. Mais du coup elles ne profitent en rien des grandes améliorations de ces dernières années et c'est dommage.

Les améliorations des constructeurs eux-mêmes ne posent aucun problème. Par exemple pendant que Microsoft essayait de transformer les PC avec écran de plus en plus large en grosses tablettes full-screen (Windows 8), Samsung ou LG sur leurs smartphones offraient le multi-fenêtrage pour avoir deux applications fonctionnelle en même temps… 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… Pour aboutir à un retour à Windows 7 sous les traits de Windows 10. Ces modifications "constructeur" n'ont pas d'impact sur les applications qu'on peut écrire pour Android. 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 S6, 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, voire même par dessus des couches constructeurs (ce qui multiplie les combinaisons) mais que cela est un élément de la réussite de 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 sucrerie en suivant l'ordre alphabétique (CupCake, Donut, Eclair, Froyo, GinderBread, etc).

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.

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).

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 sont nommées les versions, tout cela est pratique et sera utile à un moment ou un autre. Développer plus chaque point n'aurait guère d'intérêt, le Web ainsi que les librairies sont remplis 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…

Partie 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 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 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. Des extensions de Android lui-même ou des couches spécifiques à certains constructeurs possèdent la capacité de faire du multi-fenêtrage, le plus souvent comme Windows 8 le faisait (partage grossier de l'écran en deux ou trois parties maximum).

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 et UWP). Ce mécanisme s'est imposé partout car il permet de gérer de nombreuses applications de façon fluide sans pour autant consommer trop de puissance. 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.

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 du cycle de vie de cette activité pour la tenir 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.

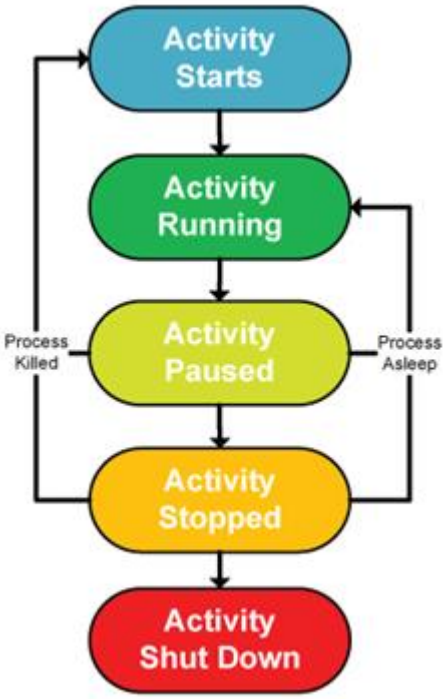
Il est extrêmement important pour le développeur de l'application d'analyser les besoins de chaque activité pour déterminer les méthodes exposées par la gestion du cycle de vie qui doivent être prises en charge. Ne pas le faire peut entrainer une instabilité de l'application, des bogues, des problèmes avec les ressources et peut-être même dans certains cas avoir une incidence sur la stabilité de l'OS.

La gestion du cycle de vie

La gestion du cycle de vie des applications de 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 piles 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.

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.

Effet des touches de navigation

On retrouve sous Android deux boutons physique ou non importants : l'accueil et la navigation arrière (le troisième appelle l'historique des apps, mais cela dépend de la version de Android). 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 à la vue de ce que nous avons étudié 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

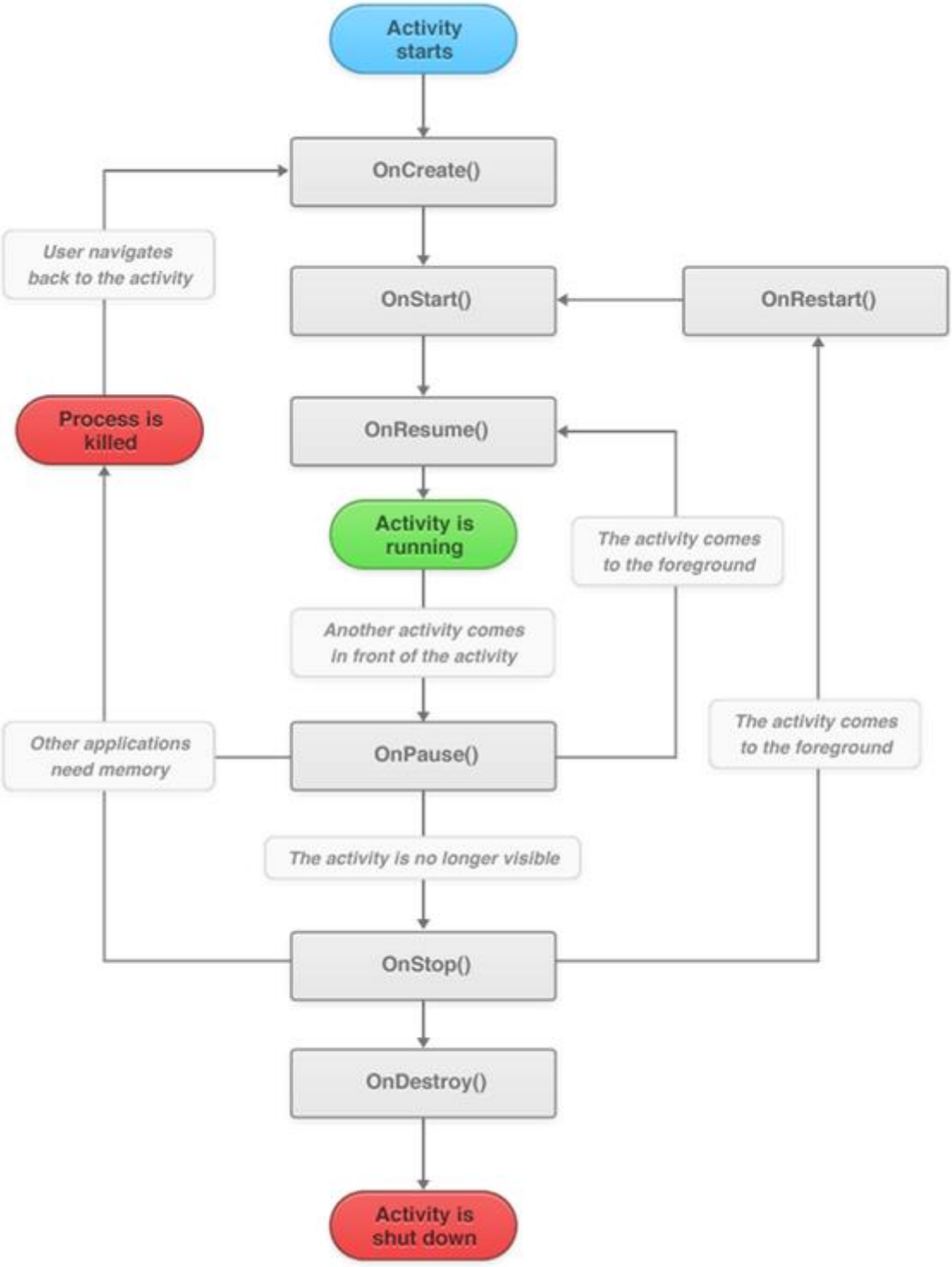
d'une application en arrière-plan sont conservées plus longtemps ce qui offre une meilleure réactivité côté utilisateur.

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 que 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.

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.

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 « classique » (non XF))

```
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 tout comme UWP 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. Ainsi 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. 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 c'est du code ou de l'affichage comme on peut en faire dans des tas d'OS. Il reste beaucoup 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 "tombstoning" 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.

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 sauf avec les Xamarin.Forms… 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 chapitre 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 !

Partie 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…

On notera que ces connaissances ne sont pas indispensables lorsqu'on travaille avec les Xamarin.Forms mais elles peuvent s'avérer essentielles pour la création d'un plugins XF par exemple. N'oublions pas non plus qu'une app écrite en XF accèpte parfaitement que certains écrans soient créés de façon classique, ce qui peut devenir obligatoire lorsqu'on souhaite exploiter certaines spécificités propres à l'OS.

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.

Cette partie 4 choisit de présenter la gestion des vues 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 de 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.

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) et les “dessinables”. Les ressources contiennent les vues, les “dessinables” tout ce qui est image en général.

Dans un précédent chapitre de cette série je parlais du mode non vectoriel de Android par opposition à XAML et des implications de ce choix. 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`").

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. En Xamarin.Forms et même en utilisant XAML on retrouve fatalement des restrictions de ce type puisque ce XAML est un noyau commun non vectoriel. La gestion des ressources peut aussi être gérée dans chaque projet natif même avec XF. Il est donc important de connaître ces conventions.

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 natif 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`). 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. 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 autorise via une notation très simple de déclarer des Bindings dans des fichiers AXML de façon très proche de la syntaxe XAML. Les Xamarin.Forms offrent de base quant à elles bien entendu un binding très complet. Il faut donc bien garder à l'esprit la différence entre le code visuel Android AXML qui de l'android avec un designer visuel spécifique pour ce format de fichier XML et le XAML de Xamarin.Forms qui est aussi un fichier XML mais suivant une syntaxe plus proche du vrai XAML… Ici je parle de code natif en Xamarin (anciennement « MonoDroïd »).

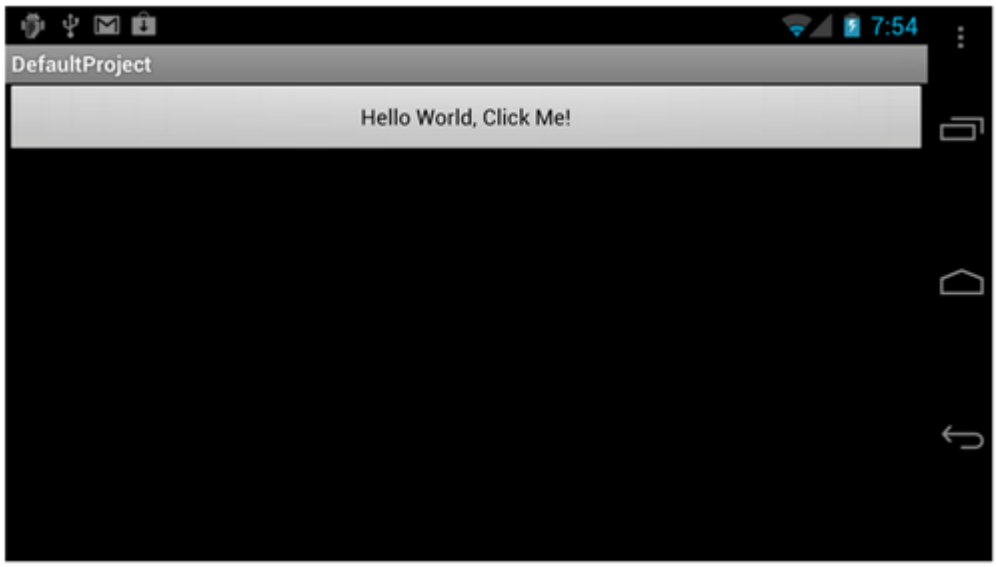
Une autre nuance : chaque contrôle possède un `ID` au lieu de porter un nom en `String`, mais ceux-ci sont des codes entiers peu faciles à manipuler, pour simplifier les choses une table de correspondance est gérée par Xamarin avec d'un côté des noms clairs et de l'autre les codes. Cette table est en 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`". Le signe plus permet d'incrément l'ID (donc de le créer) tout en l'utilisant une première fois, si on doit s'y référer ensuite dans le code XML cela se fera sans le «+ ».

Ce sont ces petits 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.

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 créée 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.

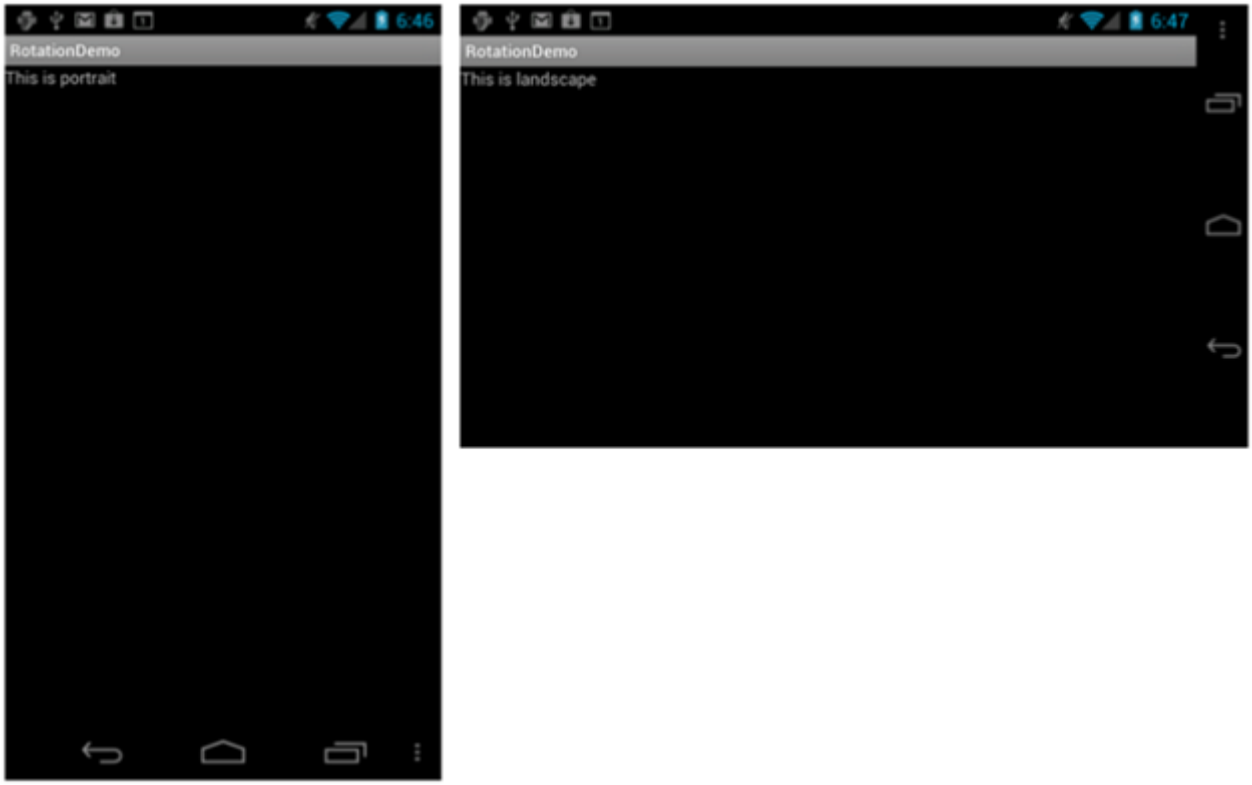
Imaginons qu'une activité définisse un l'AXML suivant dans "layout-port" :

```
<?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 (une sorte de `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…

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 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 Designer et beaucoup d'attention côté Développeur… De même là où une seule 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 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.

Le premier exemple qui me vient à l'esprit serait celui d'un lecteur de PDF par exemple. Quand je lis un tel fichier sur mon smartphone ou ma tablette je trouve que ces applications gèrent très mal le changement d'orientation car en fait elles ne le gèrent pas, elles laissent les automatismes fonctionner… C'est le cas sous Android et aussi sur Surface, pas de jaloux !

J'entends par là que lorsque je lis je ne suis pas forcément une statue, je bouge. Si je suis assis, je vais changer d'accoudoir, je vais boire, si je suis au lit j'ai les bras qui fatiguent, ça tangue un peu. Et là, pof ! ça change d'orientation… C'est enquiquinant. Sur ma tablette Android Acer il y a un petit bouton pour bloquer l'orientation c'est très pratique, mais cela n'existe pas sur toutes les unités mobiles. Sur une Samsung Note 7 il y a un réglage logiciel pour « durcir » le changement d'orientation, c'est très pratique mais encore une fois ce n'est pas une fonction de l'OS donc ne comptez que sur vous et votre code !

Un programme d'affichage de PDF bien fait devrait à mon sens gérer lui-même l'orientation en offrant à l'utilisateur trois modes : automatique, portrait forcé, paysage forcé. Ça serait tellement bien.

Je ne connais pas les 800.000 applications Android par cœur ni le million sous iOS. Donc peut-être que sous iOS ou Android il existe des lecteurs PDF qui possèdent cette fonction. Je n'ai pas encore eu la chance de tomber dessus (les stores en général sont assez mauvais question recherche, même chez Google le roi des moteurs de recherche il y aurait des efforts à fournir).

Bref, vous voyez maintenant à quoi peut servir la gestion "manuelle" de l'orientation, le mode automatique n'étant tout simplement pas une panacée satisfaisant toutes les utilisations possibles.

Mais 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 de saisie. 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…

Une seule solution : lors de la création de l'Activité, il faut interpréter les données et le fichier de paramétrage pour créer dynamiquement la fiche à présenter à l'écran.

Ce n'est qu'un exemple, et un seul suffit à prouver que cela peut donc être nécessaire.

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

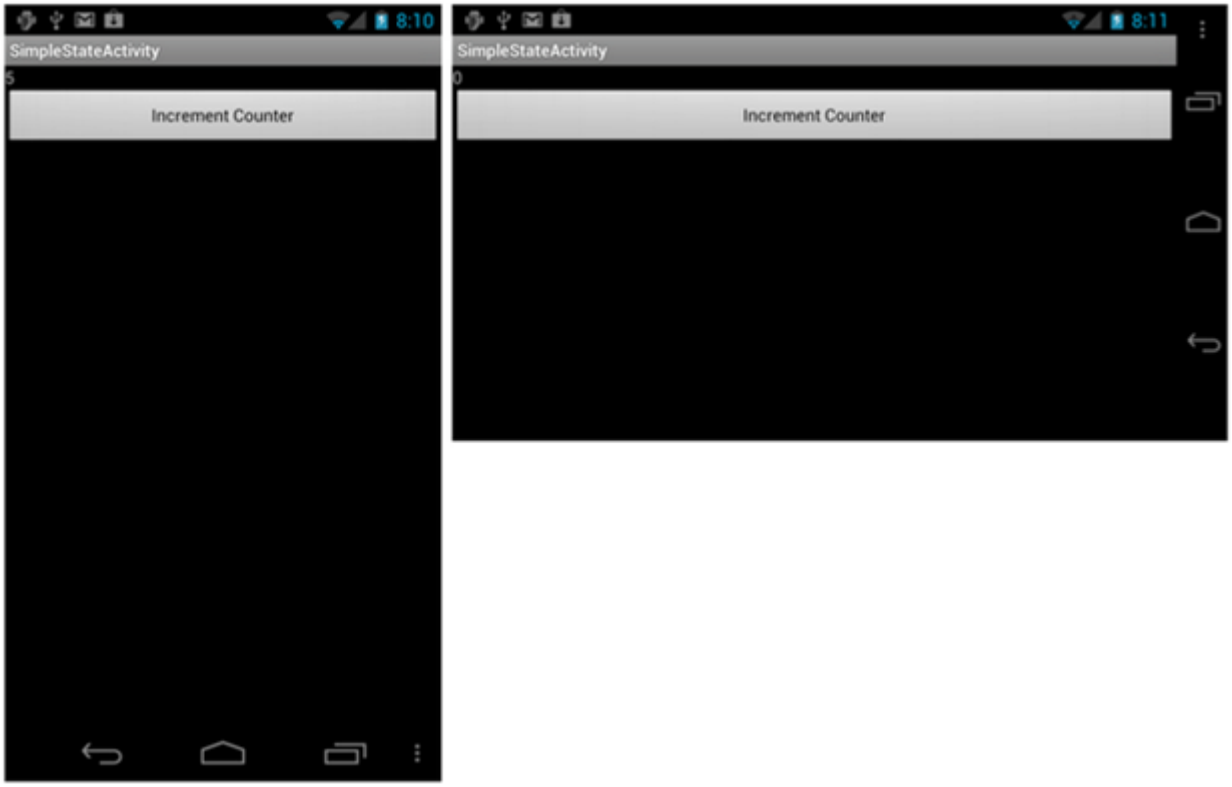
Rien que de très logique. On fait la même chose en XAML, et le principe était le même sous Windows Forms ou même sous Delphi de Borland … C'est pour dire si c'est difficile à aborder…

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 de 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 recréée, 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'infinie des possibles s'ouvre 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 particulière 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.

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" de 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 gérer sous OS Microsoft.

On notera à nouveau que je parle dans cette section de Xamarin utilisé en natif sous Android, sans la surcouche cross-plateforme Xamarin.Forms que j'aborde ailleurs dans ce livre.

La connaissance du code natif est importante pour comprendre la plateforme, comment elle fonctionne, mais aussi parce parfois les besoins spécifiques peuvent

dépasser ce qu'offre les Xamarin.Forms (au moins pour l'instant) et qu'une fiche peut se voir être développée totalement en natif même au sein d'une application en XF.

Partie 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. Pour terminer ce tour d'horizon 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).

Quand on créé 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. 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 utilisées 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 (tout comme celui des droits que vous devez posséder pour les images, vidéos, musiques, fontes etc).

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 :

```
1    public partial class Resource
2    {
3        public partial class Attribute
4        {
5        }
6        public partial class Drawable {
7            public const int Icon=0x7f020000;
8        }
9        public partial class Id
10       {
11           public const int Textview=0x7f050000;
12       }
13       public partial class Layout
14       {
15           public const int Main=0x7f030000;
16       }
17       public partial class String
18       {
19           public const int App_Name=0x7f040001;
20           public const int Hello=0x7f040000;
21       }
22   }
```

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, 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 :

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3           android:orientation="vertical"
4           android:layout_width="fill_parent"
5           android:layout_height="fill_parent">
6       <ImageView android:id="@+id/myImage"
7           android:layout_width="wrap_content"
8           android:layout_height="wrap_content"
9           android:src="@drawable/flag" />
10  </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 quelconques 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 simplifie 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 limité 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 es t "`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 99% des machines en circulation (environ 30 à 40% d'Android 2.3). 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 une 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). "`keyseposed`" 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 pose 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.

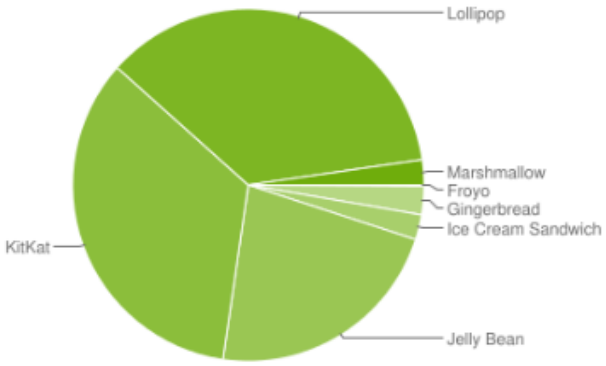
Il n'y a guère que chez Apple où tout cela n'a pas de sens, puisque l'acheteur n'a pas de choix…

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.

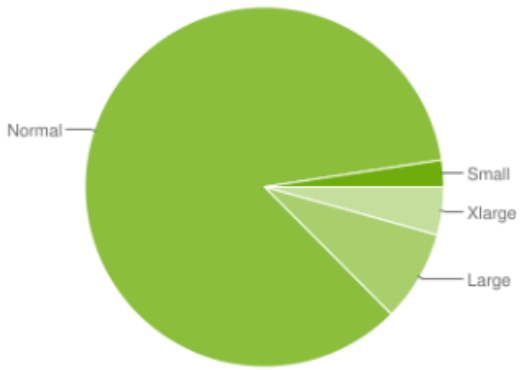
Par exemple, à la présente date la répartition est la suivante :

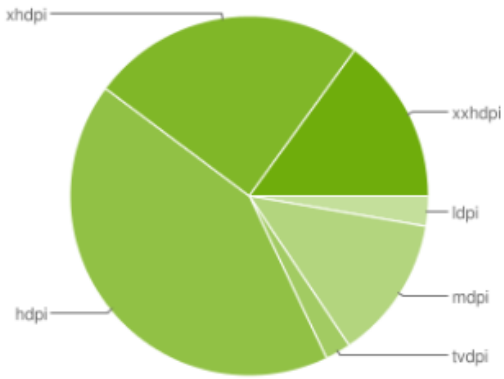
Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.3%
4.1.x	Jelly Bean	16	8.1%
4.2.x		17	11.0%
4.3		18	3.2%
4.4	KitKat	19	34.3%
5.0	Lollipop	21	16.9%
5.1		22	19.2%
6.0	Marshmallow	23	2.3%



Pour les tailles écran nous obtenons le tableau suivant :

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	2.4%						2.4%
Normal		4.9%	0.1%	41.5%	23.6%	15.0%	85.1%
Large	0.3%	4.7%	2.2%	0.5%	0.5%		8.2%
Xlarge		3.3%		0.3%	0.7%		4.3%
Total	2.7%	12.9%	2.3%	42.3%	24.8%	15.0%	





Ces informations sans cesse mises à jour sont très importantes pour le développeur et le designer puisqu'elles permettront de faire des choix raisonnables en termes de niveau d'API supporté et taille/résolution d'écran.

Créer des ressources pour les différents types d'écran

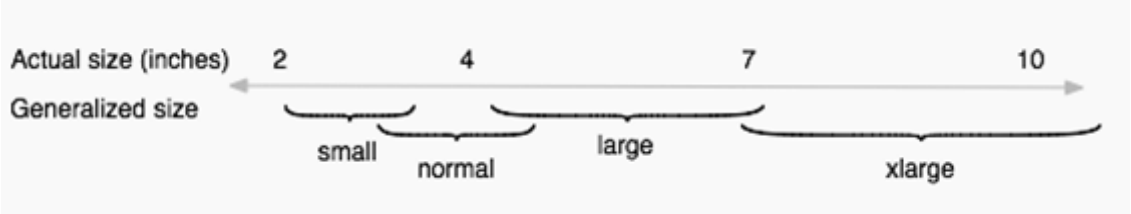
Je viens d'en parler avec les Dashboards Google, le chalenge le plus important peut-être pour le développeur et le designer est de s'adapter au foisonnement des formats disponibles (taille et résolution d'écran).

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 ?

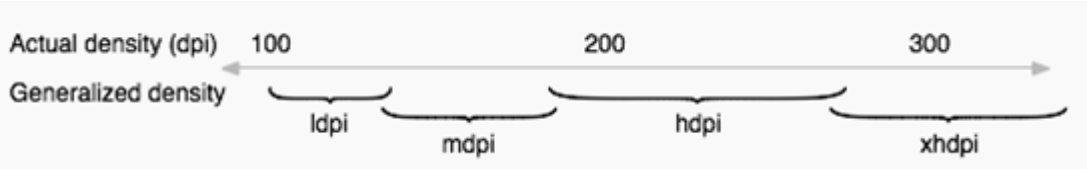
Qu'est que la "normalité" ? Non, ne fuyez pas, nous n'allons pas faire un devoir de philo ! 

Mais on peut se demander comment est défini 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.

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…

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 de moins en moins utilisée (2.6%), c'est Lollipop qui fait un peu un pivot, 60 % de machines font tourner les versions inférieures et 40% font tourner Lollipop et au-delà. On comprend que beaucoup de développeurs préfèrent encore aujourd'hui, sauf nécessité (nouvelles API), supporter 2.3 qui permet d'être certain que l'app tournera sur toutes les machines. Mais ce choix d'une API très ancienne n'est pas

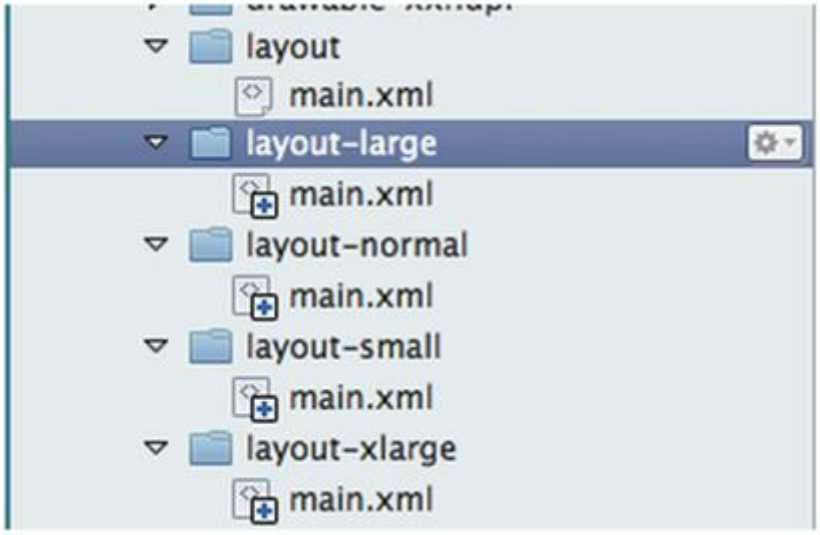
sans poser de problèmes (look, features de l'OS…). Miser sur un support minimum de Jelly Bean (4.1.x) semble plus judicieux comme compromis technique.

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 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.

Fournir des bitmaps pour les différentes densités

Comme je l'ai déjà expliqué, Android fait une mise à l'échelle des bitmaps 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.

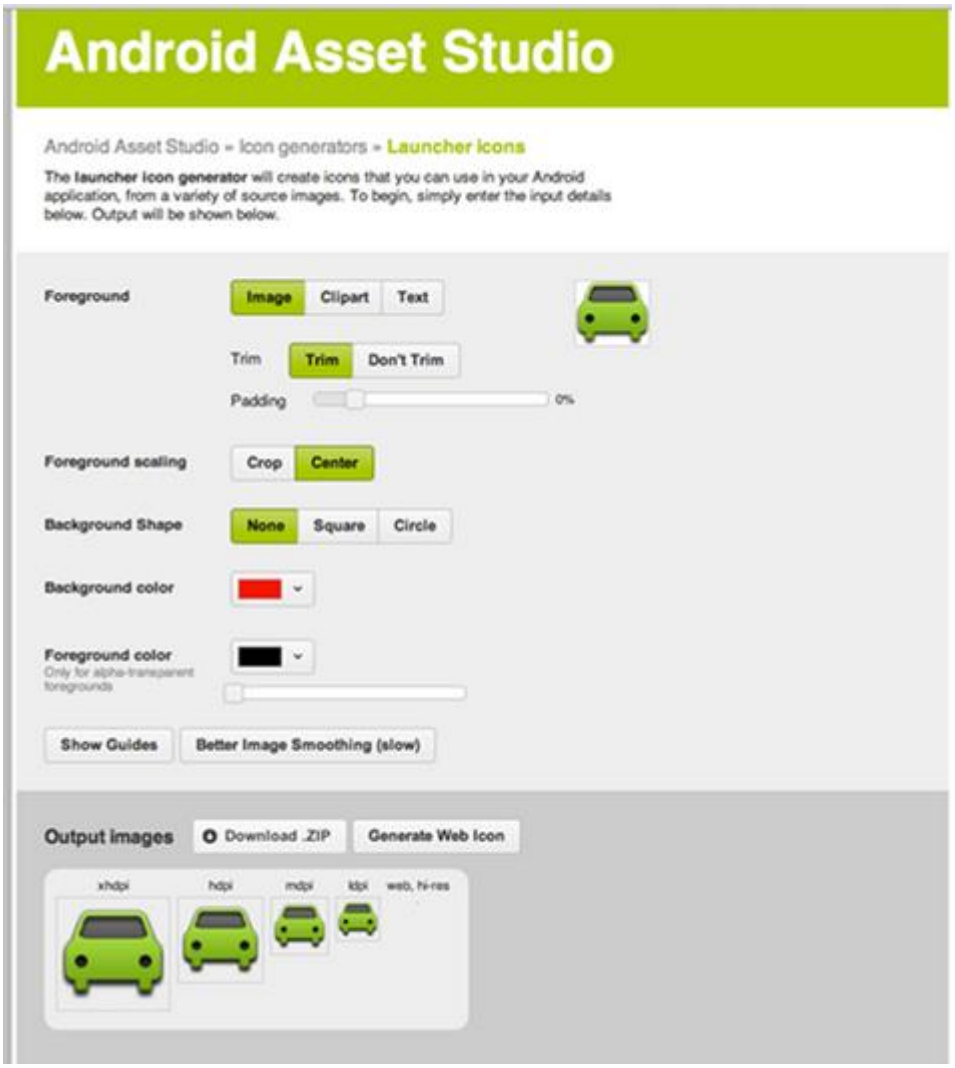
Le principe a déjà été exposé et ne présente pas de difficultés. Mais au lieu de tout faire à la main on peut aussi utiliser des outils existants…

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 donner une licence ou autre chose. Aujourd'hui il existe même certainement plus simple et plus intégré, le Xamarin Test Cloud qui propose de nombreux types de tests notamment sur des milliers de machines réelles. Autant rester dans la famille…

Ces services permettent d'avoir une bonne idée de la fiabilité de l'installation, de l'UI, etc, et ce gratuitement ou presque (comparé à la possession de centaines de devices réelles pour faire les tests !).

Localisation des strings

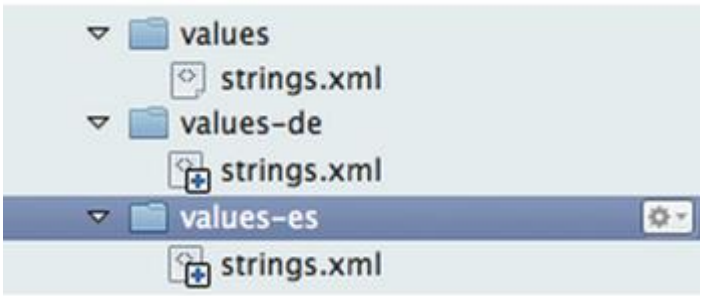
Parmi les ressources essentielles d'une application se trouvent les chaines de caractères. Une approche particulière est nécessaire dès lors que ces chaines doivent être traduites.

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 :

```
1    <?xml version="1.0" encoding="utf-8"?>
```

```
2    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3       android:orientation="vertical"
4       android:layout_width="fill_parent"
5       android:layout_height="fill_parent"
6       >
7    <Button
8       android:id="@+id/myButton"
9       android:layout_width="fill_parent"
10      android:layout_height="wrap_content"
11   android:text="@string/hello"
12      />
13   </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 UWP 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…

Avertissements

Les billets ont été collectés en février / mars 2016 pour l'édition 2015 pour les regrouper par thème et les transformer en livres PDF pour en rendre ainsi l'accès plus facile. Plusieurs semaines ont été à chaque fois consacrées à la remise en forme, à la relecture, parfois à des corrections ou ajouts importants comme le présent livre PDF sur les Xamarin.Forms.

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 sous leur forme actuelle ou intégrée même en idées sous d'autres aspects…

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 en composant pour Delphi).

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 Windows. Toutefois sa première distinction a été d'être nommé MVP *C#* en 2009. 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 UWP (Windows Store) en passant bien entendu par Xamarin.

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 avec Xamarin aujourd'hui appartenant à Microsoft.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares !