Tome 10

Universal Windows Platform

EDITION 1

Olivier Dahan

Collection
ALL DOT BLOG

MVP
Microsoft® Most Valuable Professional

e-naxos

www.e-naxos.com

Formation – Audit – Conseil – Développement

XAML (UWP, WPF, Windows Phone), C#

Cross-plateforme Windows / Android / Xamarin

UX & Sound Design

ALL DOT.BLOG

TOME 10

Universal Windows Platform (UWP)

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

Olivier Dahan
odahan@gmail.com

Table des matières

Présentation

Bien qu'issu des billets et articles écrits sur Dot.Blog au fil du temps, le contenu de ce PDF a entièrement été réactualisé lors de la création du présent livre PDF en septembre 2015. Il s'agit d'une version inédite corrigée et à jour, un énorme bonus par rapport au site Dot.Blog ! Corrections du texte mais aussi des graphiques, des pourcentages des parts de marché évoquées, contrôle des liens, et même ajouts plus ou moins longs, c'est une véritable édition spéciale différente des textes originaux toujours présents dans le Blog !

Toutefois les billets n'ont pas été réécrits en raison de la fraîcheur de l'information (Windows 10 vient de sortir !) ce qui marque une différence avec les autres Tomes de ALL.DOT.BLOG qui ont parfois réclamé un travail de mise à jour assez long. Néanmoins tout ce qui est important et qui a éventuellement changé ou été précisé de façon notable a soit été réécrit soit a fait l'objet d'une note, d'un apparté ou autre ajout.

C'est donc bien plus qu'un travail de collection déjà long des billets qui vous est proposé ici, c'est une relecture totale et une révision et une correction techniquement à jour au moins de novembre Septembre 2015. Un vrai livre. Gratuit.

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

Ce Tome est dédié à Universal Windows Plateform et tout ce qui permet de comprendre et maîtriser cette nouvelle plateforme, de ses API au Design, du code aux passerelles avec les autres OS.

Ce livre est par force en évolution, cette première version jette les bases, des éditions augmentées seront proposées au fil du temps !

L'ouvrage est segmenté en quatre parties :

`La plateforme`	Tout ce qui se rapporte à UWP en tant que plateforme de développement.
`Coder`	Du code et encore du code !
`Designer`	Comment designer des applications cross-form-factors avec UWP.
`Passerelles`	Ce qui créée une passerelle entre UWP et d'autres plateformes.

La plateforme

Avant de se jeter dans le code dans de prochains articles je vous propose un petit tour d'horizon de la plateforme Windows 10 avec les Universal Apps pour comprendre les concepts principaux et les avantages de ce nouvel environnement en rupture avec le passé.

Windows 10 est là ! Licences, téléchargement, nouveautés…

Windows 10 est enfin arrivé. Les abonnés MSDN peuvent télécharger et installer… Quelle licence choisir et qu'attendre de cette version si essentielle pour Microsoft ?

Windows 10, on croit tout savoir sur lui déjà !

Tout a été dit sur la pseudo-gratuité de Windows 10 avec toutes ses petites nuances. Tout a été dit aussi sur le fait qu'il n'y aurait pas vraiment le choix de passer à Windows 10 et qu'il sera compliqué de rester dans une version antérieure encore longtemps.

On sait aussi que Windows 10 représente un virage essentiel pour Microsoft dans le sens où pour la première fois on devrait avoir un Windows en perpétuel amélioration plutôt que des versions majeures qui se succèdent (modèle à l'Apple tant loué par l'ancienne direction de MS qui malgré tout reste aux commandes dans l'ombre).

Certains ont compris que la nouveauté était ailleurs, avant on vendait des produits annexes pour faire vendre du Windows. Aujourd'hui nous avons un Windows qui devient le produit annexe, gratuit, le cadeau "Bonux", pour faire vendre autre chose : du mobile. La position de MS sur ce segment ne sera plus tenable très longtemps si les parts de marché n'évoluent pas drastiquement. Or, en tout cas vu de chez Microsoft, tout a été tenté depuis Windows Phone 7 et Surface RT. Beaucoup de produits, souvent excellents, beaucoup de travail sur les OS, bien meilleurs qu'iOS ou Android. Mais à l'arrivée toujours troisième derrière les deux autres, indéboulonnables.

Windows 10 et le concept d'Applications Universelles sont indissociables. L'un ne va pas sans l'autre. C'est Windows 10 qui va tourner sur tous les form factors et ce sont les App. Universelles qui vont donner tout son sens à Windows 10.

C'est un pari, peut-être le dernier pari de Microsoft pour rester dans la course des mobiles, après 18.000 licenciements principalement dans l'ex Nokia racheté et près de 8.000 nouveaux en ce moment toujours dans la branche téléphonie… Le plus gros plan de charrette de tous les temps pour Microsoft. Quand il n'y aura plus d'employés à virer dans la branche téléphonie, on y arrive, il n'y aura plus de mobilité signée MS tout simplement.

Il est donc crucial que Windows 10 envahisse le monde, d'où la gratuité.

Licences et téléchargement

Windows 10 a été lancé en pré-téléchargement automatique sur presque tous les PC sans que les utilisateurs le sachent. Quand tous les morceaux seront arrivés sur les machines une invite de mise à jour proposera la bascule qui sera quasi immédiate puisque tout aura été téléchargé dans l'ombre.

Il existe bien entendu toujours la méthode de l'ISO pour ceux qui ont accès aux téléchargements MSDN.

Quelle licence télécharger et installer ?

N KN VL …

Il existe plusieurs versions de Windows, la 10 n'échappe pas à la règle. Certaines nuances se comprennent d'elles-mêmes : home, éducation, pro, entreprise, tout cela ne nécessite pas d'explication.

En revanche il y a des nuances dans des petites annotations de type "N" ou "KN". Qu'est-ce que cela signifie ?

- N est fait pour le marché européen et n'inclut pas le Media Player
- KN est pour le marché coréen et n'a ni média player ni messagerie instantanée
- VL est pour les licences par volume qui utilisent MAK (Multiple Activation Key) c'est à dire une seule clé Windows pour activer plein de PC.
- Les version sans indications sont des version "full" pour le marché US ou les pays n'ayant pas les restrictions légales de l'Europe ou de la Corée.

A vous de voir donc quelle version installer, mais en général vous choisirez une version française en mode N ou N/KN.

Des versions spéciales

Derrières ces nuances qui prennent sens assez facilement Windows 10 se présente aussi sous des moutures plus exotiques que les versions précédentes, on trouve ainsi :

- Windows 10 Home 1
- Windows 10 Pro 1
- Windows 10 Education 1
- Windows 10 Enterprise 1
- Windows 10 Enterprise 2015 LTSB
- Windows 10 Features on Demand 2
- Windows 10 Language Interface Packs
- Windows 10 Symbols
- Windows 10 Symbols Debug/Checked
- Windows 10 IoT Core for MinnowBoard MAX (x86)
- Windows 10 IoT Core for Raspberry Pi 2 (ARM)

On remarque les versions ARM pour Raspberry Pi 2 ou MinnowBoard. C'est intéressant à plus d'un titre. Je n'en sais pas plus sur ces versions pour l'instant mais cela peut être une ouverture intéressante pour l'OS s'il démontre sa capacité à tourner sur autre chose qu'un PC de bureau monstrueusement équipé !

On note aussi un module "On demand" qui permet aux entreprises de mieux gérer les mises à jour en les pré-téléchargeant et en choisissant à l'avance ce qui sera déployé ou non (en tout cas c'est ce que j'en comprends pour l'instant mais nous sommes le 29-7-15, 1er jour de la véritable existence de W10 !).

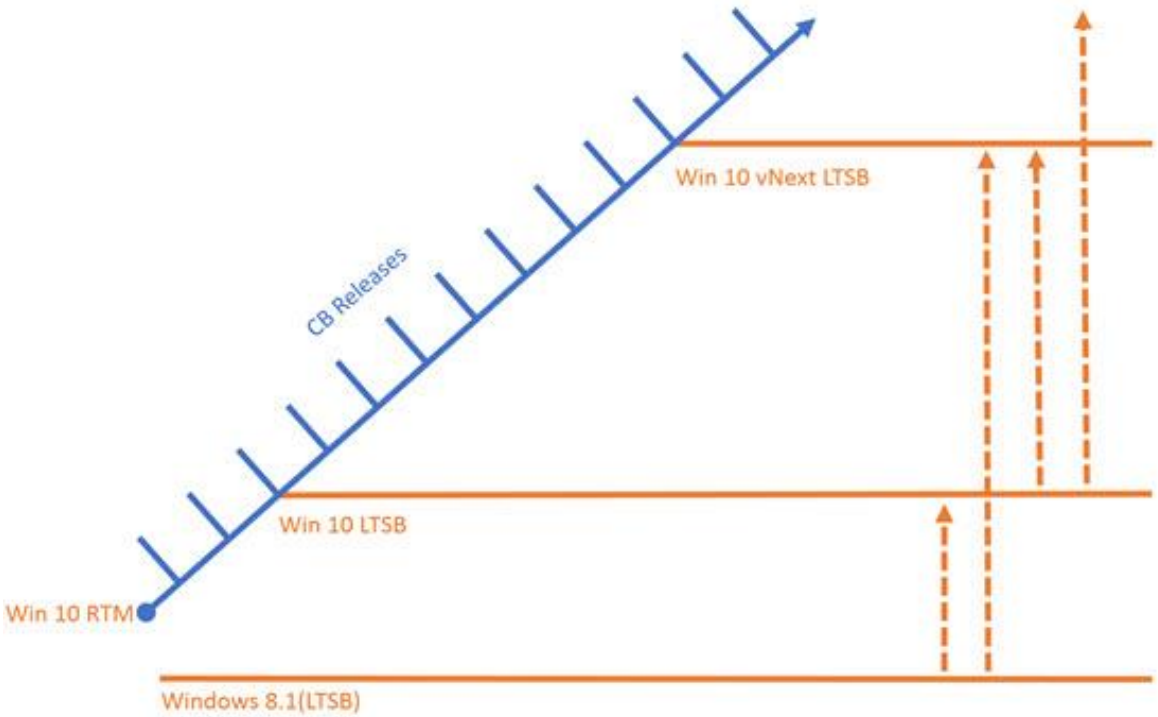
Plus bizarre est la différence entre version "LTSB" et "CB".

Long Term Service Branch et Current Branch

Windows 10 a été conçu pour être sans cesse mise à jour au lieu de marquer des points d'arrêt appelés "versions majeures" comme l'est la 10 elle-même par rapport à 8.1 ou 7 etc.

LTSB est donc un service de mise à jour ponctuel comme ce qu'on connait aujourd'hui avec les versions majeures. Seules les mises à jour de type sécurité sont assurées. Les nouvelles features n'apparaissent qu'à l'installation d'un nouveau LTSB.

CB, Current Branch, branche courante, est un système de mise à jour suivant toutes les améliorations dans le temps.



Les "CB" sont releasés régulièrement, quand les LSTB sont réleasées elles convergent progressivement vers l'état actuel des CB.

Mise à jour permanente par les CB incluant les nouvelles features, ou bien LTSB qui offre un support plus habituel avec nouvelles features par paquets en une fois mais plus longtemps après leur apparition…

Il s'agit d'un arrangement pour ne choquer personne… Microsoft vise le CB pour tous, mais en proposant un rythme plus similaire à l'ancien avec le LTSB ils veulent rassurer les entreprises et les utilisateurs plus "méfiants" quant aux nouvelles features.

C'est bien, chacun peut choisir et Microsoft a eu raison de ne pas une reprendre la logique du flingue sur la tempe de Windows 8 (du type il n'y a pas de plan B et on arrête la maintenance du reste), logique qui a été assez mal perçue à l'époque. On apprend de ses erreurs, et c'est bien de voir que MS tient compte du ressenti positif comme négatif des clients.

Qu'attendre de Windows 10 ?

On pense que techniquement un Windows 10 c'est un Windows 8.1 avec des bouts de look Windows 7, en fait beaucoup beaucoup de choses ont été refaites. Le retour du bouton démarrer, le retour d'un bureau et du fenêtrage des applications Windows + WinRT, tout cela est cosmétique. Le cœur est vraiment différent.

Ce n'est pas révolutionnaire, Microsoft à déjà fait la même chose avec Windows Vista II, pardon, Windows 7 qui a été vendu comme une nouvelle version alors qu'il ne s'agit que d'un Vista relooké (et un peu amélioré côté réactivité c'est vrai). Ici c'est tout de même assez différent. Il y a des différences majeures dans le noyau qui a été écrit pour être portable de base. Ce n'est pas un Windows 7 bricolé ni même un 8.1 relooké sinon cela ne pourrait pas tourner sur un RaspBerry Pi on se l'imagine…

C'est bien, il y a eu des erreurs, pénibles à supporter par leurs conséquences sur le marché, on entre dans une séquence de remise en phase avec les attentes des consommateurs, ne reste plus au marché que de suivre et croitre comme cela aurait du être le cas depuis Vista ou au moins avec Windows 8.

Et Windows 10 a plus d'un tour dans son sac pour tenter de séduire…

Xbox One Streaming

Si vous avez une XBox One vous pouvez streamer l'audio et la vidéo vers votre PC. En connectant le contrôleur Xbox par USB au PC vous pouvez contrôler la box… Cela permet de profiter de sa Xbox depuis un PC se trouvant dans une autre pièce par exemple. Une bonne idée.

Cortana

Causer à son PC ne me semble pas une révolution car déjà causer à sa montre ou à son smartphone ça fait un peu lapin crétin donc on ne le fait pas… J'avoue que la seule commande vocale que je trouve utile c'est celle de ma Moto 360 qui me permet de dire "appeler machin" à ma montre ce qui permet de recevoir l'appel dans mon

oreillette Bluetooth sans avoir à toucher au téléphone. Ultra utile en voiture (même si c'est devenu illégal, l'oreillette en tout cas, depuis peu. Je m'en fiche, je demanderai la météo à ma montre qui me répondra par le HP du smartphone et la police ne pourra rien dire… je suis un rebelle !).

Bon, Cortana c'est Siri ou Google Now, MS était à la traine il devait le faire, ils l'ont fait, clap clap, on est content. Il y a certainement des utilisations intéressantes à découvrir à l'usage, mais comment parler à son PC quand on écoute de la musique en permanence par exemple. Il faudra voir si cela sert vraiment. Mais on ne pourra plus dire que MS ne le fait pas. Si Android le fait, c'est une feature, si Apple le fait, c'est un must… On pense toujours comme ça chez Microsoft et ça me fait de la peine d'ailleurs ce complexe d'infériorité alors qu'il y a tant de bonnes choses chez MS et juste deux trucs à la noix chez Apple. Mais ça rapporte, et l'argent rend fou c'est bien connu…

Le retour des fenêtres

Dans un billet titré "Ne jetez pas WinRT par les fenêtres : offrez-lui en ! (la solution pour un WinRT adapté au PC)" qui date malgré tout de 2013, et illustré par Valérie histoire de fêter aussi à l'époque le 600ème billet de Dot.Blog, donc dans ce billet je plaidais déjà pour ce que Microsoft aura donc mis deux ans à faire. Peut-être que mon article à fait son chemin, allez savoir ! (je n'y crois pas c'est juste pour rire, l'ancienne équipe de direction étant tellement autiste qu'ils ne devaient surtout pas écouter toute idée contraire à celles qu'ils avaient érigées en dogmes).

En tout cas, Windows 10 répond à cette supplique : **retour du bureau et fenêtrage des applis** UWP qui peuvent marcher de conserve avec les applis WPF notamment. Ca c'est bien. Et puis avec l'article que j'ai écrit là dessus je me verrai mal ne pas saluer l'adoption de cette fonctionnalité !

Start Menu

Du coup, qui dit vrai bureau dit vrai menu Start… J'utilise avec bonheur Classic Shell depuis Windows 8.0 et j'ai zappé depuis longtemps le menu à tuile, je suis donc heureux de pouvoir me débarrasser d'outils externes pour enfin retrouver un vrai Windows… Et je remercie chaleureusement au passage le concepteur de Classic Shell sans qui ces années sous 8.0 et 8.1 auraient été un enfer. Quoi que… j'ai fini par réinstaller Classic Shell pour Windows 10 et ce n'est pas mal. Finalement je préfère… Et avec un shift-clic j'ouvre le menu original si jamais je veux voir deux ou trois tuiles bouger…

Centre de notifications

Les toasts sont les moyen le plus simple pour notifier l'utilisateur. Mais cela suppose que celui-ci soit devant son PC car les toasts ne sont pas éternels, et heureusement sinon on n'y verrait plus rien…

Windows 10 propose de regrouper toutes les notifications non lues avec les icones des applications dans un "centre de notifications". L'utilisateur peut en prendre connaissance facilement et ne loupe donc aucune information importante. Il s'agit d'une amélioration logique et pratique du concept de toast.

Edge

Comment ne pas parler de la mort de Internet Explorer tant décrié… Et de la naissance de Edge son successeur… dont ne connait pas l'avenir… On sait ce qu'on perd, etc, vous connaissez le proverbe.

Mais Edge a des atouts intéressants, notamment son orientation partage de données avec annotations, c'est une super bonne idée. Souvent on veut partager une page web avec quelqu'un mais c'est trop pénible de faire une copie d'écran d'ajouter un mot, faire un mail et tout, ou bien il faut installer des zinzins de partage et ce n'est pas pratique. Là Edge le fait tout seul et c'est vraiment bien.

Edge supporte Cortana et va très vite. Tout le monde l'attend sur ce terrain d'ailleurs.

Avec Edge c'est en fini des plugins, vraiment fini. Pour utiliser des applis Silverlight par exemple, pourtant un truc "maison" que MS aurait pu intégrer dans Edge, il faudra… installer Internet Explorer et utiliser ce dernier. Donc autant dire que personne ne le fera.

Le grand Continuum !

La vrai “killer feature” de Windows 10 c’est ça, *le grand continuum*. C’est à dire un OS identique qui va marcher sur tous les form factors et qui permet aussi de passer une tâche en cours d’une machine à l’autre. C’était promis avec Windows 8 mais visiblement c’était plus difficile à faire qu’à dire surtout avec des bras cassés comme Sinofsky, donc ça vient avec Windows 10. Mais ça vaut le coup.

Le truc c’est que cela va bien plus loin, le concept est vraiment réfléchi (ils ont eu le temps il est vrai) : en dehors de s’adapter aux tablettes, pc et smartphones, Windows 10 s’adapte de lui-même en temps réel à la configuration. Si vous avez une tablette avec écran détachable, selon que le clavier est ou non connecté l’OS offrira des possibilités adaptées. Idem pour les nouveaux smartphones Windows 10 à venir qui pourront se voir connecter un écran, une souris pourquoi et un clavier et devenir des PC avec une UX de PC et pas de smartphone. Débranchez tout et hop vous avez une UX de smartphone…

C’est génial mais on attend de voir pour juger sur pièce. Il y aura certainement plein de choses à dire à ce moment. Mais c’est prévu dans Windows 10.

Windows As a Service

Bon, on y est donc. Windows n’est plus un logiciel, c’est un service.

Tant qu’on paye, on a le service, on arrête de payer on n’a plus rien. En gros c’est ça le principe du SAAS. Un principe que personnellement je n’aime pas, que cela soit Microsoft ou d’autres qui le mettent en pratique.

Il y a certainement du bon là dedans, comme dans toute chose, mais quand le diable m’offre gentiment un verre d’eau fraiche dans le désert, j’ai tendance à me dire que ça va me couter un bras à un moment ou un autre… Ma liberté ? C’est cher, très cher pour un verre d’eau…

Universal Windows Platform (UWP)

Avec nos amis (hmmm) d’Apple, vous pouvez écrire un code pour cibler deux types de machine, celles qui supportent iOS.

Avec nos copains de Google (je les aime bien eux, je m’en méfie mais je les aime bien) vous pouvez écrire un code pour toutes les machines qui font tourner Android (avec tout de même un peu de boulot pour supporter tous les form factors !).

Avec Windows 8.x il fallait écrire 2 applications, une pour le PC et une pour le téléphone. Avec Xamarin en plus on élargissait la cible mobile mais il fallait toujours faire une version WinRT ou WPF pour le PC.

Avec Windows 10 on écrit une seule application pour la plateforme virtuelle "**Universal Windows Platform**" et on peut cibler plein de machines différentes.

La raison de cette évolution c'est qu'il n'existe qu'un seul noyau pour Windows 10 qui est ensuite décoré d'un shell et de features qui peuvent différer mais un seul noyau, un seul centre de développement pour gérer et publier les apps et un seul store pour le grand public (une version entreprise viendra plus tard).

La plateforme UWP peut être programmée en utilisant C#, C++, VB avec XAML et/ou DirectX ou alternativement en JS + HTML.

D'autres types d'applications semblent dans le futur pouvoir bénéficier d'un mode de compatibilité avec UWP et pourront ainsi tirer partie de tous les avantages de cette plateforme : les applications XP/Vista/7/8/8.1 pour le bureau Windows, les sites web Live qui tournent localement dans le moteur de Edge, plus étonnant les binaires Android (pour l'édition mobile de W10) et les apps iOS compilées avec un compilateur spécial. Mais j'en sais trop peu là dessus pour le moment, je dis ce que j'ai entendu c'est tout. On verra plus tard pour de vrais articles sur la question !

Une application qui cible UWP fonctionnera donc sur toutes les machines supportant cette plateforme, à commencer donc par W10 ce qui est un avantage décisif, à condition que Microsoft arrive à imposer ces fameuses "autres machines" qui se limitent au PC pour l'instant.

UWP sera mis à jour en permanence et Microsoft là aussi fait dans le neuf. Sous Android le foisonnement des versions pose malgré tout un problème. MS a trouvé la solution : le développeur cible UWP, pas une version de Windows. Si UWP évolue les apps suivront le mouvement sans s'en préoccuper. UWP et Windows pourront évoluer chacun de leur côté sans gêner les applications.

Il y a même une sorte de technique de programmation qui permet au développeur de savoir si telle ou telle API est supportée ou non au runtime. En écrivant un code du style :

```
var api = "Windows.Phone.UI.Input.HardwareButtons";
if (Windows.Foundation.Metadata.ApiInformation.IsTypePresent(api))
{
   Windows.Phone.UI.Input.HardwareButtons.CameraPressed
        += CameraButtonPressed;
}
```

Ici on teste l'existence de l'API qui gère le bouton hardware de l'appareil photo. Si l'API existe on s'en sert, sinon on utilise une autre stratégie. Le soft s'adapte alors automatiquement aux possibilités de la machine qui le fait tourner sans exister lui-même en 50 versions différentes.

Ce mode de programmation me semble un peu lourd (avec des tests de namespaces en mode texte, brrr), mais ne doutons pas qu'en débroussaillant tout cela dans les semaines et mois à venir nous découvrirons des moyens de rationnaliser ce genre de tests, peut-être des librairies pour aider à orchestrer la chose…

L'adaptative Layout est une autre possibilité d'UWP. Car supporter des machines très différentes avec un seul code d'UI est sportif ! De nouveaux outils dans l'OS, les API et les composants (comme le RelativePanel) vont simplifier tout cela. Le but est bien un code pour toutes les machines offrant UWP, mais un seul code d'UI aussi ! J'aurai l'occasion, forcément, de revenir plus en détail sur ces aspects axés développement. On reparlera aussi des binding compilés… ou de la possibilité en C# d'enfin pouvoir accéder à toutes les couches XAML, moteur UI.Composition et DirectX et même de mixer tout cela pour en tirer le meilleur.

Conclusion

Il y a a plein de choses intéressantes dans UWP qui rend cette plateforme bien plus achevée que WinRT. Le nouveau contexte d'un Windows 10 de nouveau fenêtré, de l'auto-adaptation à toutes les machines, etc, font que Microsoft propose *une solution globale vraiment élégante et bien pensée*.

Certains diront que c'est un peu tard. Ce n'est pas faux. Mais ce que MS propose aujourd'hui aucun de ses concurrents ne le propose.

Le vrai problème de Microsoft c'est que les clients se fichent de la beauté technique. S'ils savaient comment on programme de l'Android en découpant des images comme les sites Web qu'on faisait en 1990, s'ils savaient à quel point Cocoa est un truc frustrant, ils auraient acheté en masse du Windows Phone dont l'OS est fluide, l'affichage vectoriel, et le tooling à tomber par terre…

Ici Windows 10 offre bien plus d'avantages décisifs au développeur qu'à l'utilisateur, surtout les très nombreux qui ont zappé Windows 8… Qui utilisent va utiliser Edge et quelques tuiles dans le menu Start, est-ce assez voyant et assez excitant ? Est-ce que séduire les développeurs sera suffisant cette fois-ci ? La mise à jour gratuite permet de compenser puisqu'il n'y a pas à obtenir une décision d'achat. Espérons que cela suffise à faire comprendre aux utilisateurs que Windows n'est plus le même.

On est dans un cercle vicieux, des concurrents tiennent fermement la place. Pour les détrôner il faut plus que des avantages pour le développeur il faut des tas de softs super cool n'existant pas ailleurs. Mais pour cela il faut convaincre les développeurs qu'ils vont gagner de l'argent sur une plateforme qui hélas n'a pas décollé en plusieurs années. Sans développeur pas de soft, sans soft pas de client, sans client pas d'attrait pour le développeur.

Espérons que Microsoft saura trouver la recette magique qui fera sortir Windows de ce cercle vicieux. La mise à jour massive et gratuite est un essai intéressant favorisant les applis UWP c'est certain, donc Windows Phone et Surface. On peut dire ce qu'on veut, mais au pire Windows est aussi bien que iOS ou Android, alors dans ce cas pourquoi n'aurait-il pas le droit à sa part du gâteau ? Et comme nous savons qu'il est supérieur à tout cela, il est temps que cette injustice s'arrête. Windows 10 le sauveur ? Je ne crois pas aux contes de fées, mais raisonnablement on peut au moins souhaiter bonne chance à ce nouvel OS et aux perspectives incroyables qu'il nous offre. Puissent le Saint Client Tout Puissant le comprendre !

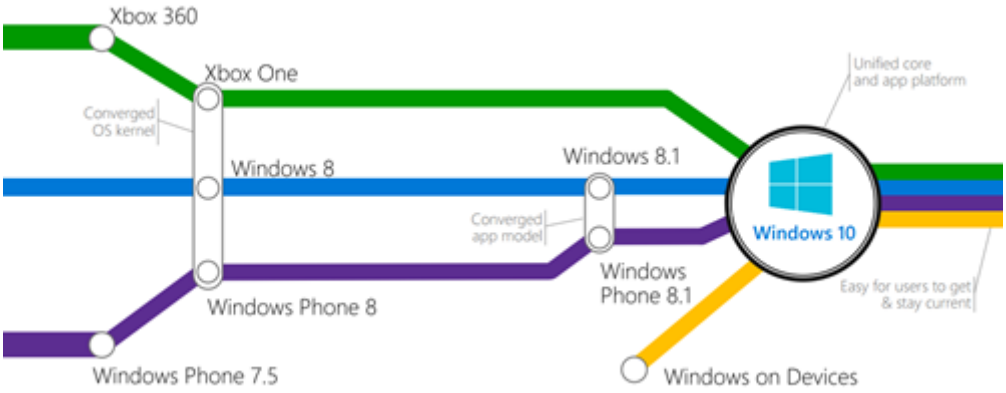
Une plateforme unifiée … de plaisir

On y est ! Je vous en avais parlé très tôt, en novembre 2011. Rappelez-vous, le fameux "our strategy with Silverlight have shifted" de Bob Muglia (qui flingua sérieusement et à raison le moral des amoureux de Silverlight). Vous remettez ? (sinon suivez ce lien où j'expliquais tout !).

A ce moment précis je vous expliquais comment je voyais les choses en développant une explication basée sur le passage chez Microsoft du rêve de l'universalité horizontale (cross-plateforme tel Silverlight) à la réalité de l'universalité verticale (cross-form-factors).

L'avenir m'a donné raison avec les annonces de Windows 8 puis 8.1 et sa convergence WinRT desktop / mobile.

La verticalité de UWP



Mais c'est désormais le présent qui enfonce le clou de cette vision qui avait quelques années d'avance sur les faits : Windows 10 réalise pleinement ce concept d'universalité verticale, à tel point même que la plateforme de développement s'appelle UWP (**Universal Windows Platform**). Même le terme et le thème de l'universalité ont été repris… (je vais demander des copyrights si ça continue !).

Bref c'est fait. Microsoft ne rêve définitivement plus, ils deviennent rationnels et nous proposent une véritable bombe atomique avec UWP : Un seul Windows conçu sur un noyau identique pour toutes les machines, de l'ARM à la télévision, du smartphone au PC. Plutôt une seule plateforme, vous allez comprendre la nuance.

Comme je le déplorais dans ce vieil article évoqué ici ce retour à la réalité n'est pas sans laisser la trace indélébile d'un rêve perdu. L'universalité horizontale, cross-plateforme, que Silverlight nous a fait effleurer était si beau… Mais un rêve est un rêve, et même cassé on doit en garder la force évocatrice comme une énergie nouvelle pour appréhender le futur et l'aimer. Aimer ce qu'on a plutôt que ce qu'on veut est le début de la sagesse parait-il…

Mais aimer UWP n'est pas un acte de désespoir ou un pis-aller. La convergence proposée par Windows 8.x était encore trop bancale, trop restrictive, elle n'était pas porteuse de rêve en elle-même. En revanche la plateforme unifiée UWP est elle porteuse d'espoir. Au moins parce qu'elle est concrète, elle existe, elle est là avec ses outils de développement. Elle rend un rêve possible, **un rêve qui est devenu aujourd'hui une obligation incontournable : le cross-form-factor**.

Après tout le cross-Platform on s'en moque, j'en ai beaucoup parlé par intérêt technique mais cela ne m'a jamais fait frissonner. C'est une obligation imposée par le marché, d'un côté Apple, de l'autre Google et un peu de Windows Phone chez Microsoft. Alors on bricole. Un bricolage peut être génial comme `MvvmCross` dont j'ai parlé longuement (plus douze vidéos d'une heure sur Youtube !) ou les fabuleuses `Xamarin.Forms`. On peut s'emballer même, pour la beauté du travail accompli. Mais c'est ce que j'appelle un plaisir négatif.

UWP, une avancée positive

Je me souviens d'une vieille blague un peu moisie je l'avoue, genre blague de récré de CM1 qui illustre en revanche à merveille ce concept, celle du type dans un asile qui en voit un autre en train de se taper la main avec un marteau et qui lui demande "*mais pourquoi vous faites ça ?*" et le fou de lui répondre "*ça me fait tellement de bien quand j'arrête !*".

Vous voyez le concept ? Xamarin, MvvmCross c'est ça. Le marché nous met sous le nez plein d'OS différents, le marteau nous tape sur la main, c'est même un "casse-tête", alors quand ces merveilleux outils apparaissent on se sent tellement soulagé que c'est *comme si le bonheur c'était ça : juste arrêter de se faire mal*… le langage populaire utilise d'ailleurs l'expression "*s'enlever une épine du pied*", même concept *d'un faux bonheur créé par l'arrêt d'une souffrance*.

Mais le vrai bonheur ce n'est pas ça. C'est partant d'un état serein et sans souffrance ajouter un quelque chose qui procure du plaisir. Ce que j'appelle donc **un plaisir positif** par opposition au *plaisir négatif* évoqué plus haut. En réalité couvrir plusieurs

OS est une simple contrainte, *sans aucune richesse ni jouissance potentielle*. En revanche pouvoir couvrir par un seul code de l'Xbox au PC en passant par les télévisions et les smartphones, ça c'est un vrai rêve, un véritable besoin car même si un seul OS existait au monde toutes ces machines différentes existeraient tout de même. Répondre à ce rêve par un tooling et une plateforme au point c'est apporter un plaisir positif, c'est ça UWP.

L'entreprise et UWP

Vous allez me dire que c'est bien joli mais qu'étant donné le marché parler d'universalité en la limitant au monde Windows qui au niveau unités mobiles est plutôt dans les cordes ça fait un peu fanboy à côté de ses pompes.

Certes. *Mais la réalité est un petit bouchon de liège sur une mer agitée*. Trimbalé par les vagues sa position change en permanence et il suffit d'un vent favorable pour qu'il aille par ici plutôt que par là…

Ce vent favorable c'est UWP bien plus d'ailleurs que Windows 10 qui n'en est qu'une émanation, un simple maillon de la chaine.

Car si vous ne l'avez pas remarqué, les unités mobiles ont envahi le monde… sauf les entreprises. Et on le comprend.

Pour le vulgum pecus le saut est facile à faire, quelques centaines d'euros et on a une bonne machine. Peu importe l'OS, après tout pour streamer de la télé réalité, écouter du rap et partager des photos de son hamburger de midi sur les réseaux sociaux l'OS n'a aucune importante. Facebook et Instagram sont les mêmes partout, et un SMS reste un bout de texte même sur un iPhone à 1000 euros…

Pour les entreprises l'affaire prend une autre tournure : il faut développer des softs spécifiques interconnectés, ça coute cher, et vu que les trente glorieuses sont finies depuis longtemps, engager un pro de Apple, un pro de Android, un pro de Windows Phone en plus des gars qui s'occupent des vieilles applis en WinForms (et qui sont techniquement souvent largués avouons-le), ce n'est pas possible tout simplement, pas plus que de fantasmer sur un type qui serait expert en tout (et pas cher en plus)…

Du coup il faudrait que les mêmes qui maintiennent du vieux WinForms puissent aussi devenir des génies du cross-plateforme et se transformer au passage en roi du Design. Là aussi il y a comme une sorte d'impossibilité pratique…

Entre deux impossibles qu'elle s'est construite toute seule l'entreprise est coincée. Pourtant il va bien falloir intégrer ces fichus zinzins mobiles dans la logique globale du SI … Et le temps presse surtout en France où le retard à ce niveau est gigantesque.

Alors UWP pourrait bien être la solution…

Former des gens connaissant WinForms donc .NET pour faire du code Universal App c'est possible. Pour les choix stratégiques d'architecture on peut ponctuellement faire appel à un spécialiste, ça ne coute pas si cher, et pour le Design on peut faire sobre et simple, il n'y a pas de client à accrocher. Tant que c'est fonctionnel c'est le principal.

Et si on a la chance d'être un peu mieux doté, alors on pourra former des équipes déjà compétentes en C#/XAML, les guider pour appliquer les bonnes pratiques de codage et de Design et ça peut même faire de très bonnes applications !

Car ce qui coute cher c'est la pluralité des OS, et elle est obligatoire uniquement pour gérer tous les form-factors. Sauf avec UWP ou cette barrière disparait et à condition de rester avec Windows mais puisqu'il couvre tous les form-factors pourquoi s'embarquer dans des culs de sac couteux ?

UWP est donc le meilleur train à prendre pour les entreprises. Au top de la modernité, mais réclamant un investissement humain quasi nul quand on a déjà des équipes de bon niveau en place. Le meilleur train car c'est aussi le dernier, dans tous les sens. C'est la dernière technologie la plus achevée, mais c'est aussi la dernière chance de rattraper le retard. Attendre mieux qu'UWP ne sert à rien, cela n'existera pas. Mieux ce serait un UWP cross-plateforme… Ca existe en ajoutant Xamarin à UWP, mais c'est le maximum. Jamais les OS ne seront harmonisés entre les concurrents et aucune chance qu'un EDI indépendant véritablement cross-plateforme émerge (en dehors des trucs java qui sentent la naphtaline). Tous les trucs de ce genre sont des leurres qui coutent cher en licences, en formations et en essais infructueux (les apps hybrides par exemple)… Rappelez-vous, l'universalité horizontale est morte avec Silverlight et avec Flash pour qui sonne aussi le glas…

UWP est donc le choix de la sagesse en restant centré sur les mêmes technologies déjà utilisées par l'entreprise, ce qui au passage assure une compatibilité sans problème entre les sous-ensembles du SI.

Reste à généraliser les tablettes Surface (mais les entreprises en utilisent déjà en raison de son support du desktop classique) et surtout les Windows Phone. Et là encore il s'agit de faire des économies et la stratégie Microsoft est bien placée : *combien coute plusieurs salariés à l'année pour maintenir du cross-plateforme dans une logique BYOD et combien coute l'achat groupé de quelques smartphones Microsoft ?*

Si je défends souvent la position du salarié il me semble qu'il faut raison garder. Il ne me parait pas raisonnable que ce soit le salarié qui décide de quel smartphone il doit se servir professionnellement. Et s'il en possède déjà un à lui, c'est son affaire. Le problème ne semble même pas traverser l'esprit des gens en ce qui concerne les PC…

Envisagerait-on des chauffeurs-livreurs imposer chacun à leur patron leur marque et leur modèle de camion préféré ? C'est ubuesque.

Avec le BYOD on tente de répondre à un faux problème. L'entreprise fonctionne sur une rationalisation du travail, ce n'est pas le buffet du Club-Med où chacun se sert comme il veut de ce qu'il veut… Faire plaisir à tout le monde c'est bien mais les entreprises ne sont pas structurées pour avoir un service informatique Club-Med…
Donc UWP est une réponse rationnelle à un problème rationnel.

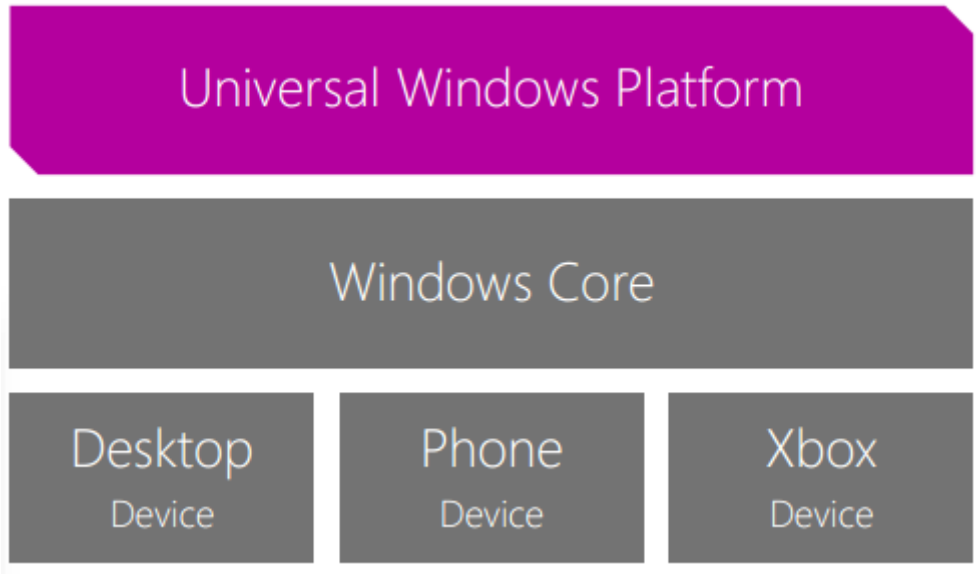
Qu'est-ce que UWP ?

Il faut considérer UWP comme une véritable plateforme physique sauf qu'elle n'est que virtuelle. Si elle présente toujours la même face aux applications (ses API), côté pile elle s'adapte à chaque hardware. En fait c'est le principe de Silverlight mais au niveau d'un OS… Silverlight pour Mac et Silverlight PC ce n'était pas le même code, mais pour les applications Silverlight c'était la même API. Microsoft pousse très loin ce concept en allant au bout du voyage, le plus loin possible, l'OS lui-même.

Les applications désormais ciblent UWP et non pas Windows 10. La clé de l'universalité est là. UWP peut évoluer à sa propre cadence indépendamment des OS spécifiques des machines.

Dans le monde Microsoft toutes les variantes actuelles commencent malgré tout à faire pas mal de hardwares différents. On le voit sur une des illustrations précédentes, on pense tout de suite à Windows sur PC ou sur smartphone mais on oublie les phablets, les petites tablettes, les grandes, les 2-en-1 (tablette / portable), les portables, Surface Hub, la Xbox, Hololens les lunettes 3D de réalité augmentée, la domotique et même IoT, le fameux "Internet of Things", l'Internet des Objets, ce qui va du bracelet pour faire du sport au Raspberry Pi 2 ou l'Arduino !

Tout cela avec la même plateforme, et comme le montre une seconde illustration :



- Un seul système d'exploitation : Un seul Windows pour toutes les machines.
- Une seule plateforme : une application fonctionne sur toutes les familles de machines.
- Un seul centre de développement : une seule soumission d'app avec un seul suivi de son flux et un seul dashboard.

- Un seul Store : un seul point d'entrée pour l'utilisateur/client, une recherche globale, monétisation centralisée …

A elle seule cette liste est un véritable conte de fée… Et il se réalise car Windows dispose d'un nouveau cœur commun qui s'interface entre la machine et les applications. C'est lui qui unifie tout en proposant un environnement d'exécution stable et commun.

Bien entendu un même Windows aurait du mal à couvrir tous les besoins de machines aussi différentes que Hololens et un Arduino, entre un smartphone d'entrée de gamme et un serveur d'entreprise.

Le noyau de Windows est donc commun mais selon le hardware on peut y ajouter des librairies spécifiques qui donnent accès aux spécificités de chaque machine.

Le développeur peut écrire un code unifié malgré ces librairies spécialisées en testant l'existence ou non des API et en s'adaptant (par exemple la présence d'un bouton hardware pour prendre un photo est piloté par une API absente sur un PC mais présente sur certains smartphones, on peut savoir à l'exécution s'il faut afficher un bouton ou si on peut utiliser le bouton hardware présent). Mais il n'y a pas non plus obligation d'utiliser les librairies spécifiques, le noyau très riche peut être utilisé pour 95% du code dans de nombreux cas.

Les outils

On retrouve bien entendu Visual Studio (2015) et le sublime Blend, rescapé de l'époque Silverlight et qui reste à mon sens l'EDI visuel le plus atypique et le plus fantastique jamais édité.

On peut développer depuis une machine équipée de Windows 10, c'est préférable pour tout faire (et bénéficier des émulateurs). On peut aussi le faire depuis Windows 8.1 et Windows Server 2012 R2. Mais dans ce dernier cas il faudra des machines physiques pour le débogue car les émulateurs ne marchent pas (mais qui voudrait rester en Windows 8.1 ???).

La fin des trucs compliqués

Pour celui qui a développé pour Windows Phone 7 et suivant comme moi, la nouvelle approche de Windows 10 me déconcerte : pourquoi n'ont-ils pas commencé par là ? ! C'est en tout cas l'une des choses que je demandais depuis longtemps car c'était un vrai frein au développement…

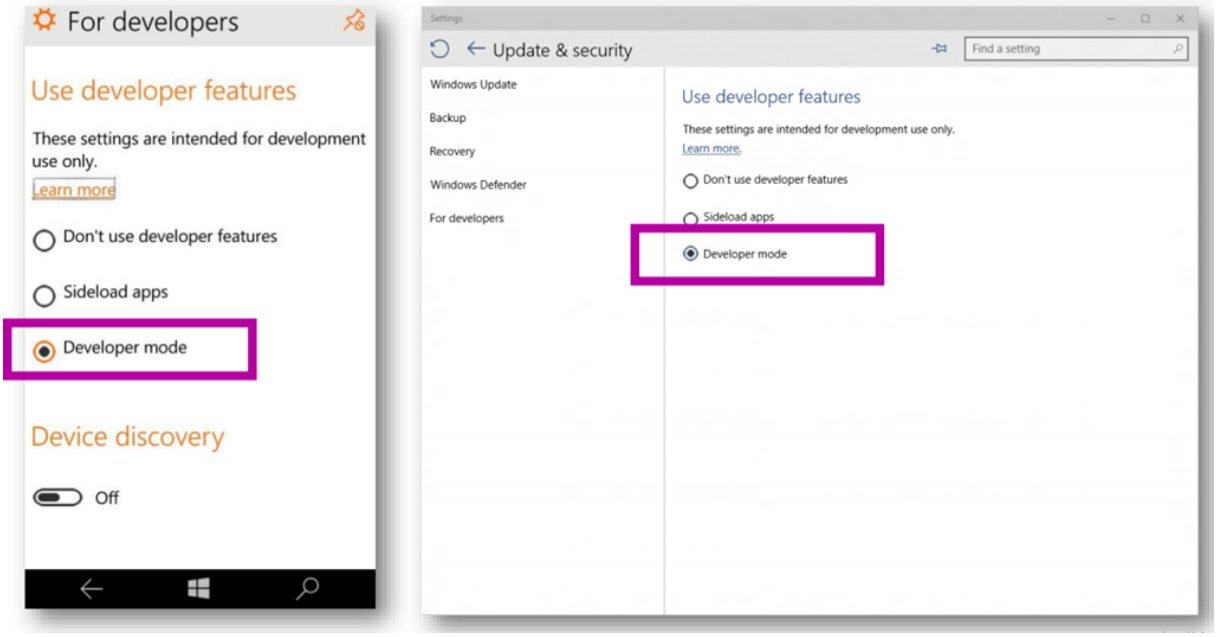
Je veux parler de l'utilisation de machines physiques pour le débogue.

Avant, mais ça c'était avant, il fallait enregistrer une machine (et pas plusieurs) via un système d'activation en ligne super bancale et complexe. A l'époque de Windows Phone 7 le service de support de Microsoft a essayé de m'aider pendant des semaines car même eux ne comprenaient pas pourquoi ça ne voulait pas marcher. Une horreur. Là où pour Android il suffisait déjà de brancher n'importe quelle machine en USB et rouler jeunesse !

Pour faire fuir les développeurs il n'y avait pas mieux que ce système tordu, complexe et capricieux.

Donc c'était avant et je suis bien content !

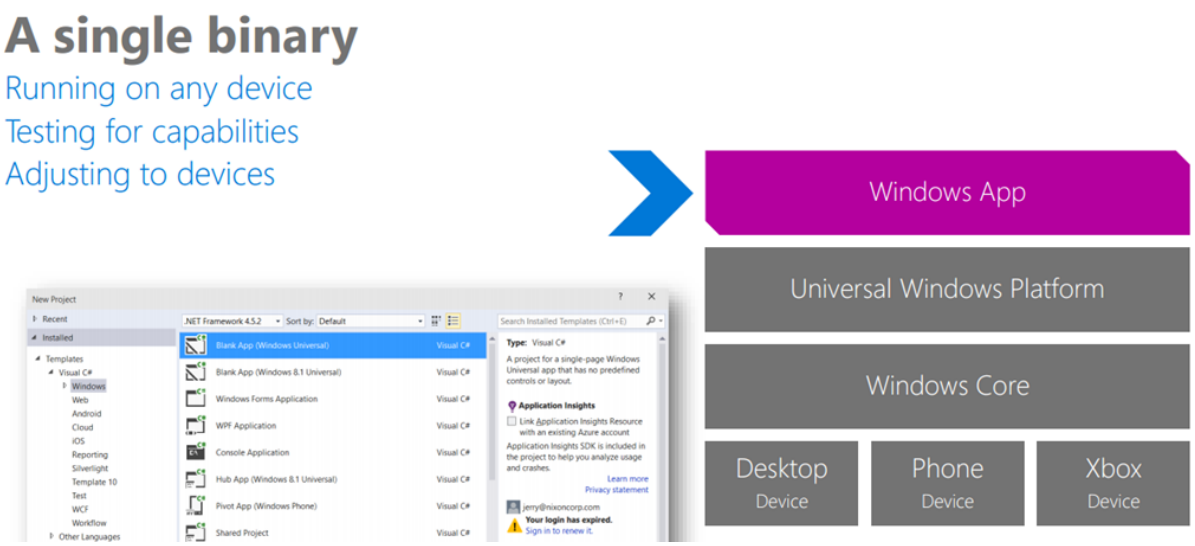
Aujourd'hui il suffit d'aller dans les bons menus d'une machine Windows 10 et dire comme pour Android qu'on débloque le mode développeur. Et c'est tout bon.



Normalement ça ne devrait pas donner l'occasion d'un paragraphe entier une telle feature, çà devrait être naturel depuis le début. Mais étant donné le saut gigantesque entre "avant" et "maintenant", ça mérite d'être dit. Le Microsoft version Nadella a l'esprit pratique et s'affranchit des trucs tarabiscotés à la Sinofsky /Ballmer qui ont fait perdre des années à tout le monde… Il y a malgré tout un vrai changement de mentalité salutaire (même si Nadella a été choisi par ceux d'avant ce qui faisait craindre le pire. Mais le pire n'est jamais certain, la preuve).

Une seule App, vraiment

Un seul binaire tourne sur toutes les machines. Pas un code compilé pour 10 cibles avec 10 binaires, non un code, un binaire, toutes les machines.



Le concept d'Adaptive App

L'adaptabilité est un concept phare de UWP, c'est l'essence même de ce dernier qui sinon n'apporterait rien. Ce fut le cas de WinRT qui en réalité n'apportait rien par rapport à .NET sauf qu'il permettait de faire des applis pour le menu à tuile dont personne n'a voulu. Techniquement WinRT était une approche intéressante mais dans la pratique pas d'intérêt (même pas mal de gênes en raison de sa nature sandboxée).

UWP marque une différence de taille par rapport à WinRT, UWP apporte quelque chose : **l'universalité mais aussi la simplicité d'un seul code même l'UI pour toutes les machines**. C'est une avancée extraordinaire. Le passage de .NET à UWP se justifie pleinement par les exigences du monde réel aujourd'hui inondé de machines très différentes les unes des autres mais étant toutes des ordinateurs qu'il faut programmer !

> *A cela .NET n'offre qu'une réponse imparfaite (Mono et Xamarin principalement). UWP est conçu pour relevé ce défi, là est son avantage incontestable.*

En réalité on chipote car du point de vue du développeur tout cela revient un peu au même. Même si quelques namespaces changent de ci ou de là, le développeur .NET

n'a pas l'impression de changer d'environnement, c'est juste un .NET un peu modifié (en apparence). Un peu comme la mort de Silverlight n'a heureusement pas impliqué celle de XAML qui se retrouve au cœur de Windows depuis la version 8.0.

On pourra d'ailleurs dire ce qu'on veut, on a longtemps reproché à Microsoft son art consommé de la rupture totale. On a par force associé certains changements comme la fin de Silverlight à cette pratique détestable. Mais en réalité depuis Vista et la sortie de WPF, *le développeur qui manipule C# et XAML a pu sans trop de travail s'adapter à tous les changements*, de Silverlight, WPF à WinRT jusqu'à UWP aujourd'hui. La critique a été assez facile pendant longtemps pour qu'on puisse se permettre de relever, a contrario, *cette exceptionnelle cohérence dans le monde Microsoft* pour le développeur depuis une dizaine d'années malgré les bouleversements technologiques vécus entre temps.

Ainsi, avec le concept d'Adaptive Apps :

- Une app s'adapte aux différentes versions de la plateforme (compilation conditionnelle, tests au runtime)
- Une app s'adapte aux différents types de machine (tests du support de certaines API par exemple)
- Une app s'adapte aux différentes résolutions (états visuels XAML spécifiques par exemple)

Les Adaptive UI sont capables en effet de gérer plusieurs type d'écran dans le même code. Tout comme l'Adaptive code peut exécuter certaines parties uniquement sur certaines machines dotées de certaines propriétés.

Il est important de comprendre ce mécanisme d'adaptabilité. C'est la réponse trouvée par Microsoft pour résoudre la quadrature du cercle : comment avoir un seul code binaire qui peut s'adapter à autant de cibles si différentes ?

On l'a vu dans des domaines beaucoup plus limité avec MvvmCross (pour les smartphones et tablettes) ce que l'adaptabilité oblige comme gymnastique. Toujours

dans le monde restreint des mobiles on voit l'effort à fournir même à l'aide des Xamarin.Forms. Alors imaginons un peu l'effort insensé pour supporter à la fois un Raspberry et un PC, des Hololens et une tablette Surface !

Certes comme je le disais *c'est l'idée de Silverlight qui est poussée au bout de sa logique* : un runtime qui présente une face identique aux applications mais qui est adapté à chaque plateforme. Sauf que pousser le concept au niveau de l'OS intellectuellement est une chose, le réaliser et le rendre "praticable" en est une autre ! UWP c'est pourtant ce pari fou qui se réalise.

L'Adaptive design & UI

Le code adaptable j'aurai l'occasion d'y revenir dans des exemples concrets, l'idée est géniale mais une fois énoncée ce n'est qu'une façon d'écrire du code ce n'est pas forcément très excitant en soi. De plus faire du code "portable" depuis Xamarin on savait grosso-modo le faire correctement. Avec UWP c'est juste un principe intégré à la plateforme elle-même. C'est plus puissant, plus propre, mais ce n'est pas une idée nouvelle.

Le véritable chalenge dans une application cross-form-factors c'est de gérer les différences phénoménales au niveau des UI !

Xamarin.Forms est une réponse possible (pour le cross-plateforme) mais limitée à un domaine précis et encore ce n'est pas parfait il faut bricoler un peu dans de nombreux cas pour faire vraiment de belles choses. La faute en revient aux OS sous-jacents qui sont préhistoriques comme iOS ou Android.

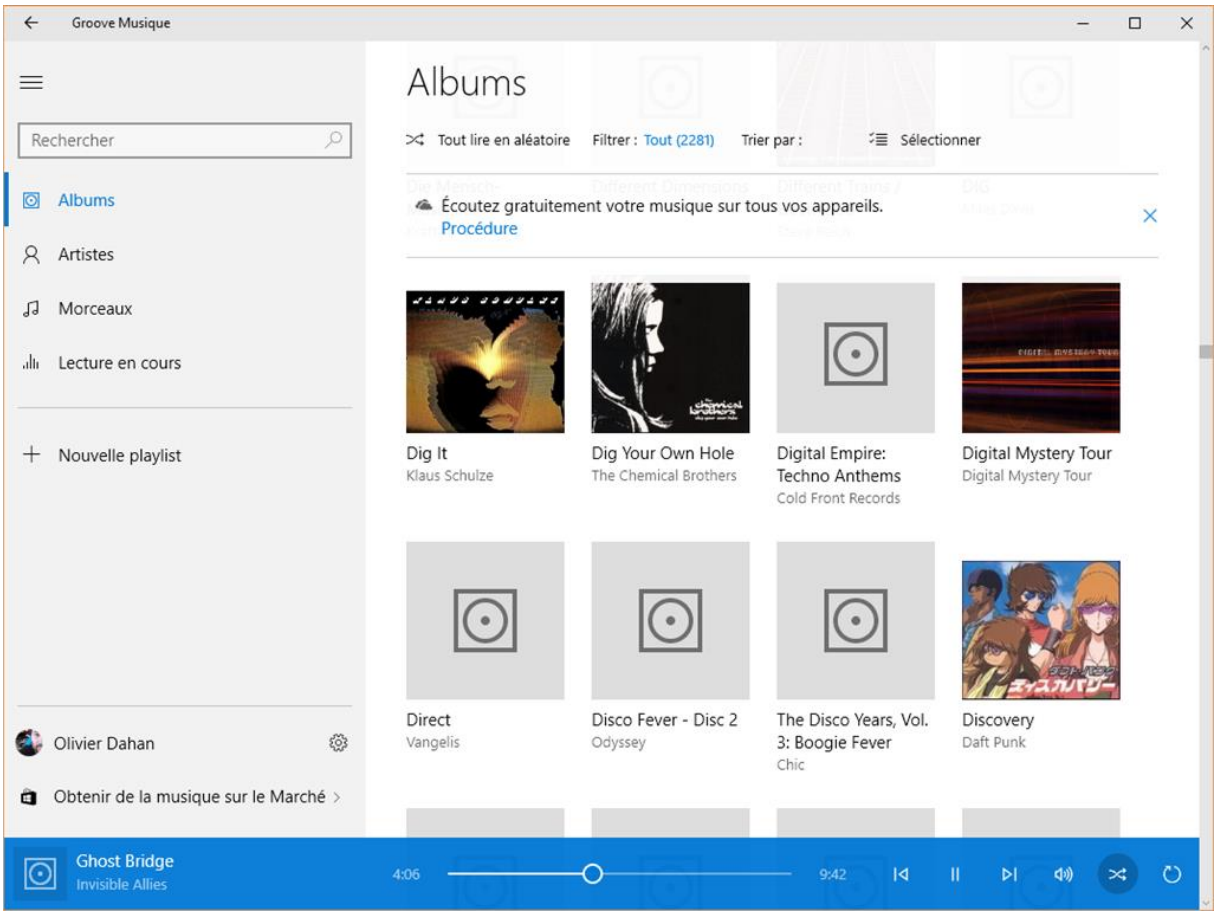
Mais sous Windows et avec XAML on travaille en **vectoriel**. *C'est à dire le mode le plus sophistiqué qui soit et qui se trouve le mieux adapté au monde actuel de l'informatique*. Une ellipse en vectoriel reste une ellipse peu importe la résolution. Sous Android il faut de tas d'images PNG différentes selon l'orientation (et oui une ellipse ce n'est pas un cercle !), la résolution, la taille écran, etc. C'est une vraie catastrophe. Et iOS ne fait pas mieux, si ça semble plus simple c'est que le monde

fasciste d'Apple ne propose qu'un seul modèle de machine à ses clients là où Android est supporté par des centaines de machines différentes.

Mais posséder XAML aussi fantastique que cela soit (je suis un adorateur fou de XAML depuis toujours) ne résout pas forcément tout. Beaucoup mais pas tout. Il faut y ajouter quelques pincées de logique bien pensée pour que cela devienne universel. Comme les *adaptive triggers* qui déclenchent des états visuels XAML selon des seuils (taille, résolution…).

Prenez Groove par exemple, le lecteur de musique de Windows 10. Si on le réduit au minimum il ressemble à cela :

Cet affichage on le soupçonne pourra fonctionner presque identiquement sur un smartphone. La mise en page en tout cas (avec peut-être des boutons plus gros). Mais si j'agrandi la fenêtre l'approchant d'un format tablette on voit alors :



La barre bleue s'est allongée, elle affiche des informations sur le disque en cours. La liste des pochettes prend maintenant une place plus importante mais apparait à gauche un système de menu facilitant le pilotage du logiciel. **Pourtant c'est le même logiciel !**

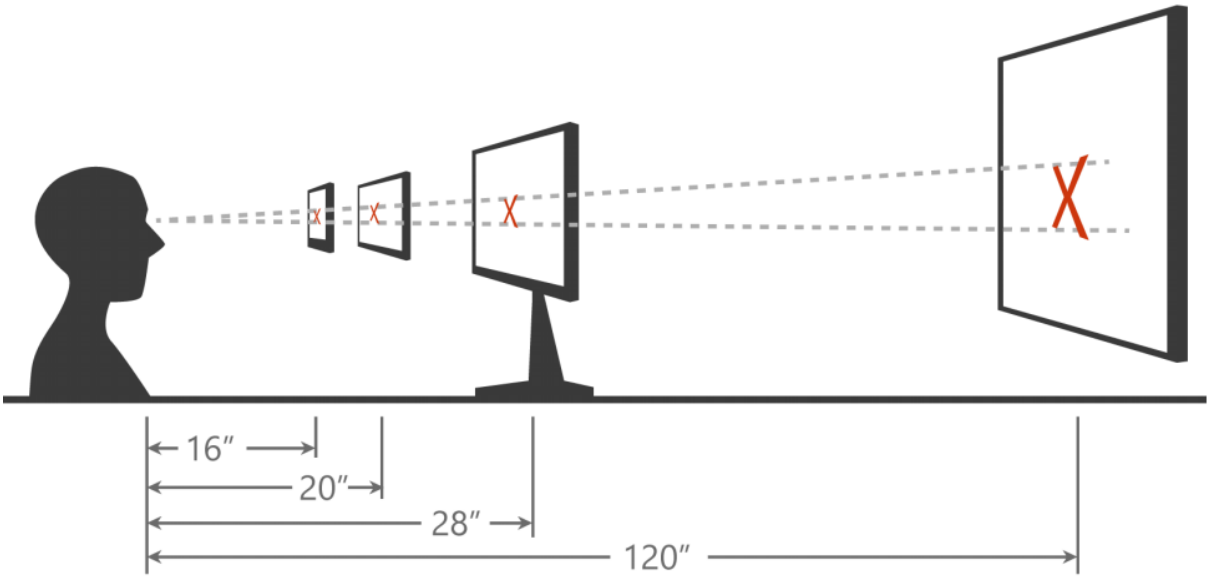
L'adaptive Design c'est ça : Prévoir une mise en page différente selon la résolution et la taille de l'écran. Le tout dans un même code XAML avec un seul exécutable.

Prenons cet autre exemple entre un smartphone et un PC portable :

La mise en page est différente, jusque dans la taille du bouton permettant de raccrocher qui est agrandi pour le smartphone (facilitant la manipulation d'une main) ce qui n'est pas le cas de la version PC adapté à un clic souris ou un touch du doigt sur Surface.

> *Adaptive Design & UI c'est penser la mise en page en constante évolution, en mouvement permanent, mais avec un seul code XAML… Et cela UWP le permet.*

XAML possède déjà une énorme souplesse, ce que UWP ajoute c'est par exemple la notion de **pixels effectifs** différents des pixels physiques. Des algorithmes ont été pensés pour que l'œil voie toujours le pixel de la même façon alors que les résolutions sont différentes mais aussi les distances d'utilisation (ce que les autres OS négligent)…



Prendre en compte les PPI c'est bien mais si on ne prend pas en compte la distance à laquelle une machine se regarde de façon habituelle cela n'est pas assez pour assurer le confort visuel de l'utilisateur.

Avec UWP on oublie les échelles, la résolution, les DPI et les centaines d'images dans des sous-répertoires, les nine-patches et autre aberrations au XXIème siècle. On travaille en pixels effectifs. Et en XAML.

Ca aussi j'en reparlerai, mais c'était important de porter votre attention sur ces améliorations décisives d'UWP en matière de Design des applications.

Franchement je suis soulagé et heureux : avec WinRT il fallait expliquer les raisons des limitations, avec UWP il faut faire découvrir des tas de possibilités nouvelles, c'est autrement plus sympa pour le blogger que je suis !

.NET natif

Impossible de finir ce tour d'horizon sans parler des bouleversements côté code même s'ils sont en partie invisibles. Les effets sont eux énormes.

Les langages managés comme C# deviennent encore plus rapides et efficaces une bonne raison à cela, **.NET devient natif au sens le plus strict du terme**.

.NET Native c'est une nouvelle génération de compilateurs dans le cloud. Les apps utilisent l'optimiseur de code C++ standard, elles possèdent un bootstrapper .NET **car il n'y a plus de runtime !** .NET devient une partie de l'exécutable tout simplement, la plateforme n'a pas besoin du framework X ou Y. Le code produit est donc *du vrai code machine* même pour un langage managé…

Les bénéfices sont nombreux et se chiffrent d'après Microsoft à une vitesse de chargement d'une app 50% plus rapide et une occupation mémoire de 14% inférieur à l'équivalent .NET classique.

Binaire machine, optimisation "optimales", pas de framework à déployer, moins de mémoire consommée, chargement plus rapide… la nouvelle plateforme a réellement de quoi séduire sur tous les tableaux.

Conclusion

Windows 10 est l'arbre qui cache la forêt UWP. Une forêt magique où tous les problèmes qui se sont entassés ces dernières années avec le foisonnement des form-factors trouvent une solution simple et élégante.

J'espère que ce petit tour d'horizon vous aura permis de saisir cette partie encore immergée de l'iceberg et que vous aurez plaisir à suivre les prochains articles qui iront plus en détail dans les possibilités concrètes de Windows 10 et UWP.

Comment se lancer dans le développent Windows 10 ?

Je vous ai présenté ces jours derniers ce qu'était le développement avec Windows 10 et plus exactement la plateforme UWP. Mais par où commencer pour développer ? …

Le choix XAML/C#

Ce n'est pas moi qui irai vous parler de javaScript ou HTML ni de TypeScript ou encore pire de C++. Mon truc à moi c'est C# et XAML.

Mais depuis le début de .NET et plus encore depuis Windows 8, tout le monde sait que Microsoft qui veut ratisser large pour son Store a largement ouvert ses portes à d'autres combinaisons de langages, aussi bien côté code que UI. Globalement toutes les combinaisons sont acceptables pour faire de bonnes applications puisque techniquement ça a été "prévu pour".

Si on connait bien JS/CSS/HTML il est vrai que la possibilité d'écrire des applis natives Windows en réutilisant ce savoir fait gagner du temps. Est-ce la meilleure façon d'aborder les choses ? Pourquoi se priver de la puissance de C#, du vectoriel unique au monde de XAML ? Je ne sais pas. Par simple ignorance je suppose. Mais on ne peut pas tout savoir ni les autres ni moi-mêmes alors développez avec ce qui vous fait plaisir, l'essentiel c'est de développer dans la joie !

En ce qui me concerne dans ces colonnes, la joie passe par C# et XAML, Visual Studio et l'extraordinaire Blend. Mais ce n'est que mon choix, vous pouvez faire comme vous préférez !

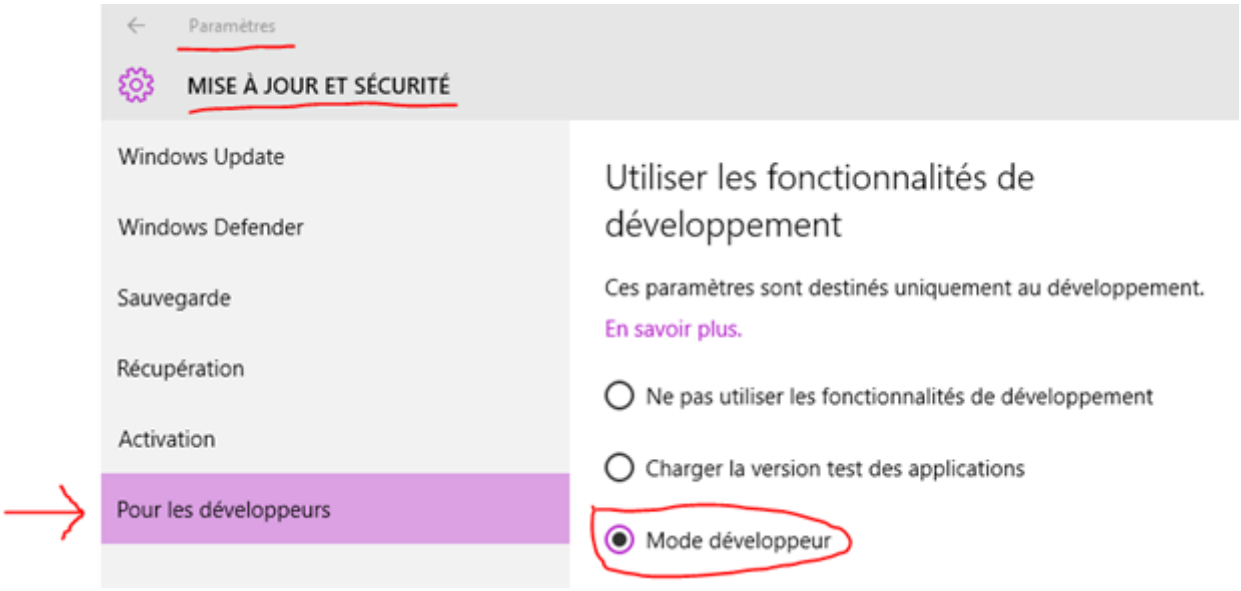
Bien commencer : le matos

Pour écrire des applications Windows 10 (et UWP) il vous faudra un minimum de choses :

- Windows 10. Même si on peut utiliser des versions antérieures certaines choses ne marcheront pas ou mal. Autant le dire tout de suite pour développer pour la nouvelle plateforme mieux vaut être en Windows 10, le reste est un mensonge.
- Visual Studio 2015 avec Blend. Bien entendu lors de l'installation ne pas oubliez de cocher les "*Universal App Development Tools*", sinon il reste possible de les installer à part en allant sur la page du SDK.

Finalement ce n'est pas grand chose, Windows 10 étant en mise à jour gratuite cela simplifie les choses.

Il faut aussi penser à basculer votre PC en mode développement, un peu comme un gros téléphone donc :



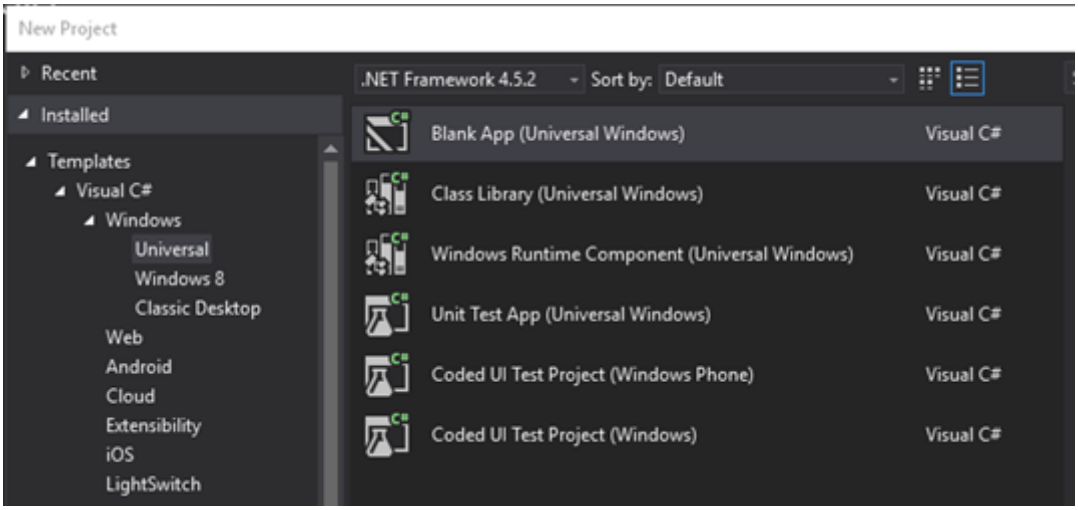
Vous obtiendrez plus de détails sur ce mode “développeur” en lisant la documentation “Activer votre appareil pour le développement”. En gros les appareils Windows 8+ et donc Windows 10 sont faits pour bloquer l’installation des applications hors Store, comme sur Android d’ailleurs et lorsqu’on développe en mode UWP forcément on fabrique des applications qui ne sont pas sur le Store. Pour éviter de se faire bloquer il faut donc activer le mode développeur. A noter que cela ne concerne que les applications UWP, si vous faites du WPF cela ne vous concerne pas. Mais ici je vais parler UWP donc c’est important !

Le célèbre Hello World !

L’incontournable…

Ouvrez Visual Studio 2015. Oui c’est beau. C’est à mi-chemin entre le look Blend de la première heure (celle de Silverlight) et le look Metro (qu’il ne faut plus appeler comme ça depuis des années je sais). Ceux qui n’aiment pas les fonds noirs seront donc déçus mais on peut changer les couleurs.

Mais ce n’est pas la beauté qui nous amène ici, donc on fait Fichier / Nouveau / Projet puis dans le dialogue qui s’affiche à gauche, Templates / Visual C# / Windows / Universal, à droite Blank App.



Les habitués le savent, j'utilise toujours les outils de dev en version US mais vous avez le droit d'utiliser la version française !

On choisit donc une application universelle "blank" (vide) en C#. Au passage tout à droite (non capturé sur l'image ci-dessus) il faut décocher la télémétrie cela ne servira pas ici.

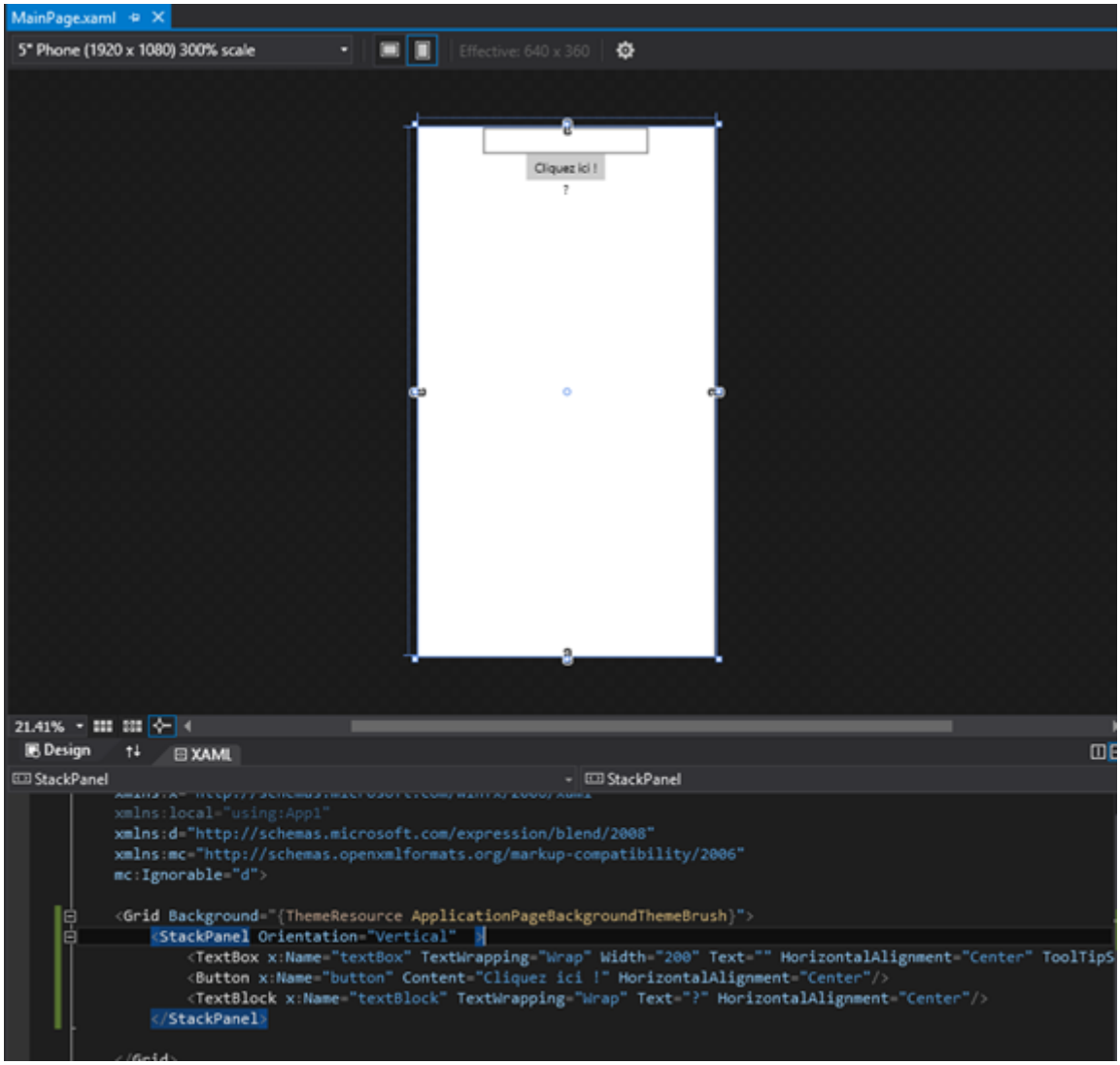
On laisse tout par défaut, même le nom de l'application "App1", c'est ça un Hellow World.

Le nouveau projet est créé. Fermons ce qui a été ouvert automatiquement et ouvrons `MainPage.Xaml`.

On arrive sur le designer qui affiche une zone rectangulaire allongée en hauteur, un format téléphone par défaut, on sent bien que tout cela, de Windows 10 gratuit à UWP n'a tout de même qu'un seul but : simplifier et forcer un peu la main pour obtenir des apps Windows Phone. Mais après tout c'est de bonne guerre et en plus de tels efforts ont été faits pour que cela ne nous complique pas la vie (UWP = un seule code et une seule UI) qu'on se pliera au jeu de bonne grâce.

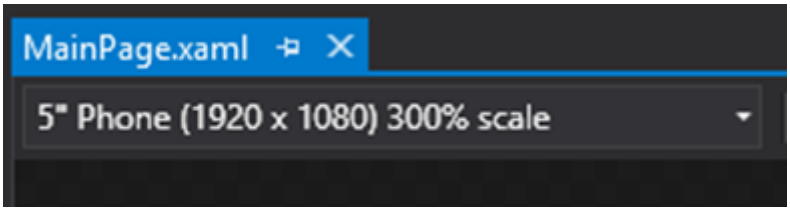
Comme toujours dans un projet "Blank" la page XAML principale ne contient rien d'autre qu'une grille.

Ajoutons un champ de saisie, un bouton et une zone de texte :



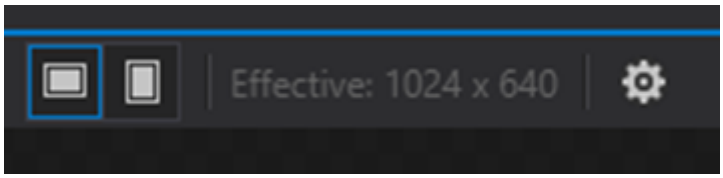
Comme le montre le code XAML j'ai placé un StackPanel vertical dans lequel j'ai mis une `TextBox`, un `Button` et un `TextBlock`, chacun étant centré horizontalement.

A noter que bien que le mode par défaut de l'affichage soit celui d'un smartphone 5", il est tout à fait possible à tout moment de switcher d'un style d'affichage à un autre :



Cliquez sur la petite flèche et vous verrez apparaitre toute une liste de form factors différents, UWP oblige, et voir instantanément votre application en mode tablette, PC, etc. C'est forcément très pratique pour visualiser en cours de travail si tout s'adapte bien quelque soit les dimensions et la résolution des cibles.

Vous remarquerez aussi juste à droite de cette zone de sélection le groupe d'icones suivant :



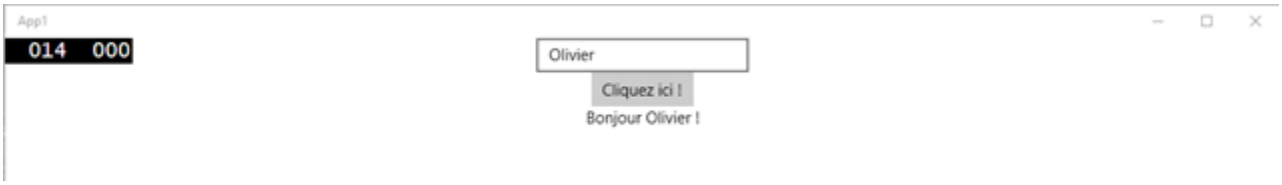
Les deux premiers permettent de passer du mode paysage au mode portrait, l'information qui suit indique qu'on travaille en 1024x640 en pixels effectifs (j'en ai parlé dans ma précédente présentation de UWP, ce sont des pixels virtuels qui permettent de développer sans se soucier de la véritable résolution). Le bouton de réglage permet de basculer dans certains modes propres à la device choisie (ici je suis passé en tablette 8") comme le mode "haut contraste" par exemple.

Comme il n'est pas le moment d'entrer dans des détails subtils on va oublier MVVM et le reste et juste programmer un gestionnaire pour le clic sur le bouton dans le code-behind. On double-clic sur le bouton et le gestionnaire est créé.. On va juste créer le texte à afficher en concaténant le prénom tapé et un petit bonjour classique :

```
namespace App1
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    3 references
    public sealed partial class MainPage : Page
    {
        0 references
        public MainPage()
        {
            this.InitializeComponent();
        }

        0 references
        private void button_Click(object sender, RoutedEventArgs e)
        {
            textBlock.Text = string.Format("Bonjour {0} !", textBox.Text);
        }
    }
}
```

Rien d'extraordinaire, un Hello Word dans toute sa splendeur donc !

Sous Windows 10 il n'y a rien de plus à faire, on tape F5 et le programme se lance…

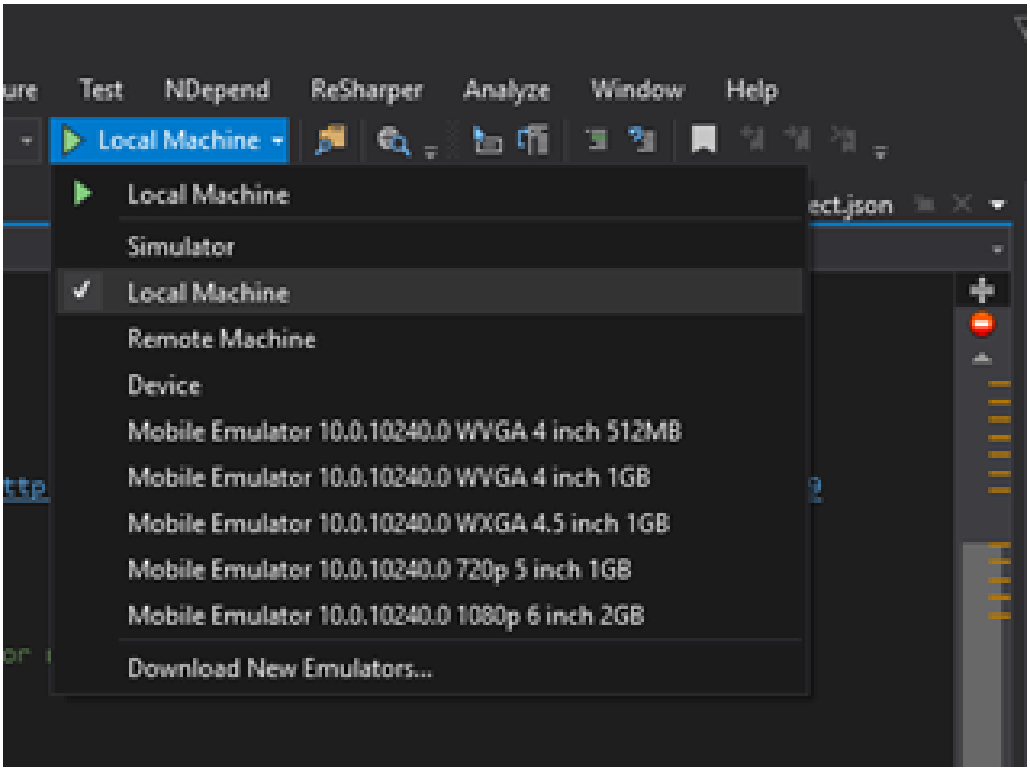


Ayant choisi un format tablette 8" en mode paysage je vous fais grâce de tout l'espace vide sous la partie utile…

Vous remarquerez pas mal de petites choses intéressantes comme les informations de débogue en surimpression à gauche, et si vous regardez VS 2015 vous verrez des graphiques se fabriquer (la télémétrie, les traces…).

On en rediscutera car VS devient une vraie tour de contrôle bourrée d'outils pour la mise au point des logiciels.

D'ailleurs quand on exécute par F5 cela utile les valeurs par défaut pour le contexte mais on peut bien entendu choisir celui-ci :



Ici on voit que par défaut c'est "local machine" qui est utilisée pour le débogue. On pourrait utiliser une machine distante, un émulateur etc…

Conclusion

Je vais m'arrêter là pour un simple Hello Word qui était le prétexte de vous présenter un peu VS 2015 et quelques unes de ses options ainsi que les liens de téléchargement indispensables.

On verra beaucoup plus de choses en détail plus tard.

Windows 10 / UWP, MVVM Light et un peu de magie !

Dans un article précédent j'ai présenté le développement sous Windows 10 et UWP avec le célèbre Hello world incontournable. Aujourd'hui on recommence l'exercice mais en MVVM !

MVVM et Windows 10 / UWP

Les applications pour Windows 10 et UWP ne sont pas si différentes de celles en WinRT, en dehors du côté universel de UWP, on pourrait même dire que c'est du WinRT. Et WinRT lui-même si on ne regarde que son API c'est un peu la même chose que du .NET en plus étendu. Même logique, espaces de noms et classes similaires sauf exceptions, on passe de l'un à l'autre dans une continuité déroutante.

Quant à XAML même s'il en a existé et existe plusieurs saveurs avec de belles différences cela reste du XAML. Quant on a compris une fois on a compris pour toujours, en tout cas jusqu'à maintenant et l'avenir semble tout aussi limpide. Enfin C# lui-même en dehors de ses améliorations constantes reste un choix de roi – roi un peu fainéant puisque là aussi quand à compris C# 1.0 on a compris pour toujours...

De fait en UWP on se retrouve à faire du C#/XAML exactement comme on en faisait en WinRT, exactement comme on en fait avec WPF et comme on en faisait avec Silverlight.

Depuis Windows 8 et WinRT il y a quelques petites choses à savoir en plus mais elles sont finalement peu nombreuses et grâce à UWP qui harmonise vraiment toutes les différences sont à apprendre un fois pour toutes les cibles.

Il résulte de cette magnifique continuité rare en informatique et malgré les ruptures d'importance dans les OS et les plateformes physiques supportées que bien évidemment tout ce qu'on a appris sur les bonnes façons de développer reste vrai dans l'essentiel.

Et en toute logique *il en va ainsi pour MVVM qui s'applique à merveille à UWP*.

Quels outils ?

Prism

J'ai fait partie du groupe qui a travaillé sur Prism pour WinRT au sein du "*Developer Guidance Advisory Council Microsoft*" il serait donc mal venu de ne pas en parler en premier… Et comme je suis honnête je vais même dire que Prism pour WinRT ne m'a pas plus convaincu que WinRT lui-même qui, dans les faits meurt avec UWP (même si derrière on reste en WinRT).

La raison est que le modèle d'application plein écran de WinRT ne m'a jamais séduit, j'ai d'ailleurs été parmi ceux qui ont publiquement demandé le retour des fenêtres pour donner un avenir aux Apps WinRT ("Ne jetez pas WinRT par les fenêtres : offrez-lui en ! (la solution pour un WinRT adapté au PC)"), ce que fait avec bonheur Windows 10.

UWP m'oblige à envisager un nouvel avenir plus prometteur à Prism pour WinRT une fois celui-ci adapté à UWP.

Brian Noyes, l'un des membres de l'équipe Prism a fait un portage de Prism WinRT vers UWP, il l'a annoncé sur son blog le 6 juillet dernier. Même s'il s'agit d'un portage "initial" selon ses termes donc imparfaitement adapté à UWP, le code est déjà utilisable et publié sur GitHub (PrismLibrary/Prism).

On notera que Prism est passé en open-source et que la version WinRT ne connaitra plus d'évolution sauf débogue. Quand je parlais de la "petite mort" de WinRT en septembre 2014 à la suite des premiers leaks de Windows 9 qui deviendrait 10 j'avais là encore eu le nez creux… WinRT et ce qui tourne autour comme les outils de développement et les librairies type Prism n'évolueront plus jamais, on passe à UWP qui est proche mais tellement différent de WinRT. Une "petite mort" le terme était donc bien choisi à l'époque.

Pour en revenir à Prism pour UWP il faudra certainement attendre la prochaine version qui est en préparation et qui sera elle parfaitement adaptée à la plateforme. L'intérêt du portage de Brian se résume à celui des rares développeurs qui ont un code WinRT à porter rapidement en UWP sans trop entrer dans les subtilités et différences entre les deux approches.

Mais gardez un œil sur Prism (et sur Dot.Blog !) car dans quelques temps la nouvelle version sera prête et là viendra le temps d'en parler plus longuement !

MVVM Light

J'ai tellement parlé de MVVM sur Dot.Blog … Et de MVVM Light aussi. Je ne peux que renvoyer le lecteur qui se poserait (encore) des questions sur cette approche de développement vers la liste de tous les articles tagués "MVVM", il découvrira une galaxie d'articles et livres qui occuperont largement ses longues soirées d'automne !

Je vais donc faire court et juste rappeler que MVVM Light est une librairie que j'ai soutenue depuis ses premières versions car j'aime la façon dont son auteur, Laurent Bugnion, a su aborder avec simplicité ce qui chez les autres était sac de nœuds (Caliburn est un bon exemple d'un tel enfer et même les premières versions de Prism). Plus loin, dès le départ Laurent a fait en sorte que MVVM Light soit "*Blendable*", utilisable au design time notamment pour fournir des données adaptées à la mise en page XAML sous Blend (ou sous VS depuis que celui-ci supporte un designer visuel digne de ce nom pour XAML). C'est essentiel quand on créé une application et trop de frameworks MVVM ignorent superbement (et à tort) cet aspect.

MVVM Light est aussi une librairie qui a su évoluer sans trop changer et on la retrouve disponible pour toutes les cibles, depuis Silverlight et WPF jusqu'à Windows Phone 7 de la première heure et donc tout naturellement jusqu'à UWP aujourd'hui.

Au moment où j'écris ces lignes MVVM Light pour UWP n'existe pas vraiment tout en existant, une sorte de paradoxe qui se comprend quand on sait que les PCL's pour Windows 8.1 peuvent s'ajouter dans un projet UWP sans aucun problème (parentée de WinRT oblige)… La version 5.1.1 actuellement disponible est donc utilisable directement (version 5.2 au moment de l'écriture de ce Tome 10).

C'est donc MVVM Light que j'utiliserai aujourd'hui pour les besoins de cet article : *Créer un (gros) Hello World MVVM*.

Rapide comparatif Prism / MVVM Light

Lorsque Prism pour UWP sera disponible il sera possible de véritablement faire un tableau des différences d'autant que des breaking changes sont annoncés, raisonner sur une adaptation rapide UWP de Prism pour WinRT ne serait pas tout à fait loyal.

Mais les différences d'approches sont assez nombreuses même si Prism pour WinRT qui est totalement différent de Prism pour WPF a été conçu presque dans l'esprit de MVVM Light à un point tel qu'au tout début du projet quand nous parlions des fonctionnalités avec l'équipe Prism j'en étais arrivé à me demander l'intérêt de créer un MVVM Light bis… Mais le projet avançant Prism WinRT a proposé des fonctionnalités véritablement propres à la plateforme, navigation, asynchronisme, et

gestion des évènements particulier du cycle de vie des Apps (notamment pour sauvegarder à temps l'état de l'App quand elle passe en fond ou qu'elle se fait décharger de la mémoire par l'OS).

De fait MVVM Light est une librairie cross-plateforme ce qui permet même de réutiliser du code Silverlight ou Windows Phone 8 en Xamarin sous UWP, c'est un gros avantage, mais qui se paye un peu par un manque d'adaptation aux spécificités de telle ou telle plateforme. Je pense qu'avec UWP Laurent pourra se permettre de personnaliser un peu plus la librairie, mais Prism pour WinRT porté rapidement pour UWP répondra mieux à certaines problématiques comme le cycle de vie des applications si cela est essentiel dans une App donnée (ce qui n'est pas toujours le cas et qui peut malgré tout se gérer avec Mvvm Light).

Et d'autres librairies ?

Il existe d'autres librairies MVVM mais pour l'instant elles restent mal ou pas adaptées du tout à UWP. On pourra citer Okra qui n'est pas mal par exemple. J'ai regardé le code et je n'aime pas le style d'écriture mais c'est un détail, l'outil semble bien pensé. Je n'ai pas encore eu le temps de faire l'essai et je vous tiendrais au courant de mes essais à moins qu'un lecteur plus rapide que moi ne l'ai déjà fait, dans ce cas qu'il n'hésite pas nous laisser ses commentaires ! Une annonce récente sur le blog de Okra laisse clairement entendre qu'une version UWP sera "bientôt" releasée, je garde donc un œil sur tout ça !

Caliburn.Micro qui a succédé depuis un moment au pachydermique Caliburn au point d'en devenir finalement le remplaçant n'a pas à ma connaissance encore été adapté à UWP mais lorsqu'il le sera c'est un framework qu'il faudra prendre en considération. Je ne suis pas du tout fan de l'approche même simplifiée de Caliburn en général, micro ou pas, mais c'est l'un des frameworks qui compte dans le paysage MVVM.

En dehors de cela il existe des embryons de librairies pour UWP mais rien de concret, mais il est vrai que cette plateforme est à peine distribuée depuis quelques semaines au travers de la mise à jour Windows 10.

Bref laissons le temps au temps… tout finira par venir prochainement et pour l'instant nous en resterons à MVVM Light !

Exemple

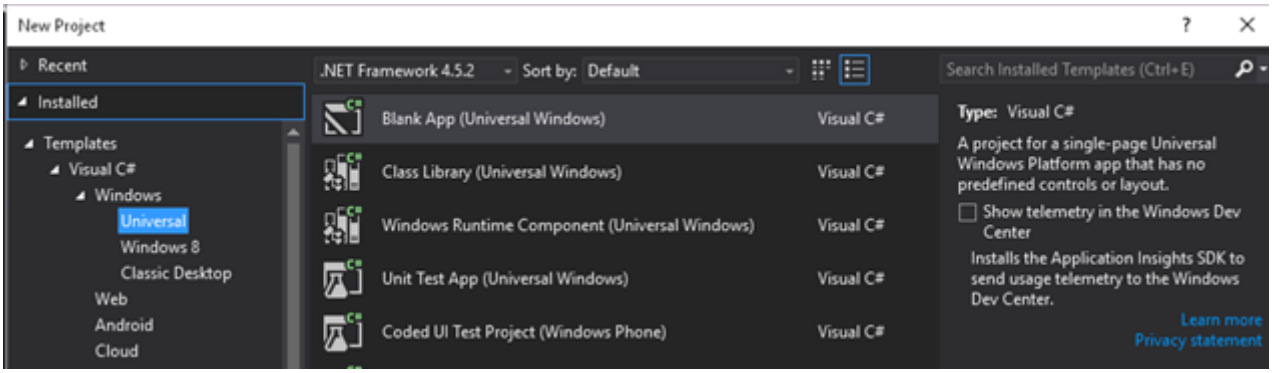
Je vais reproduire ici "à ma sauce" une application exemple publiée par Laurent Bugnion lui-même car elle contient un condensé de ce que permet Mvvm Light.

Toutefois expliquer ce code pourtant simple s'avère un exercice délicat car il y a beaucoup à dire et il ne faut pas transformer l'article en roman (ce que je fais à chaque fois ou presque malgré tout… et en me relisant je vous le dis maintenant que je connais la fin, c'est un roman !).

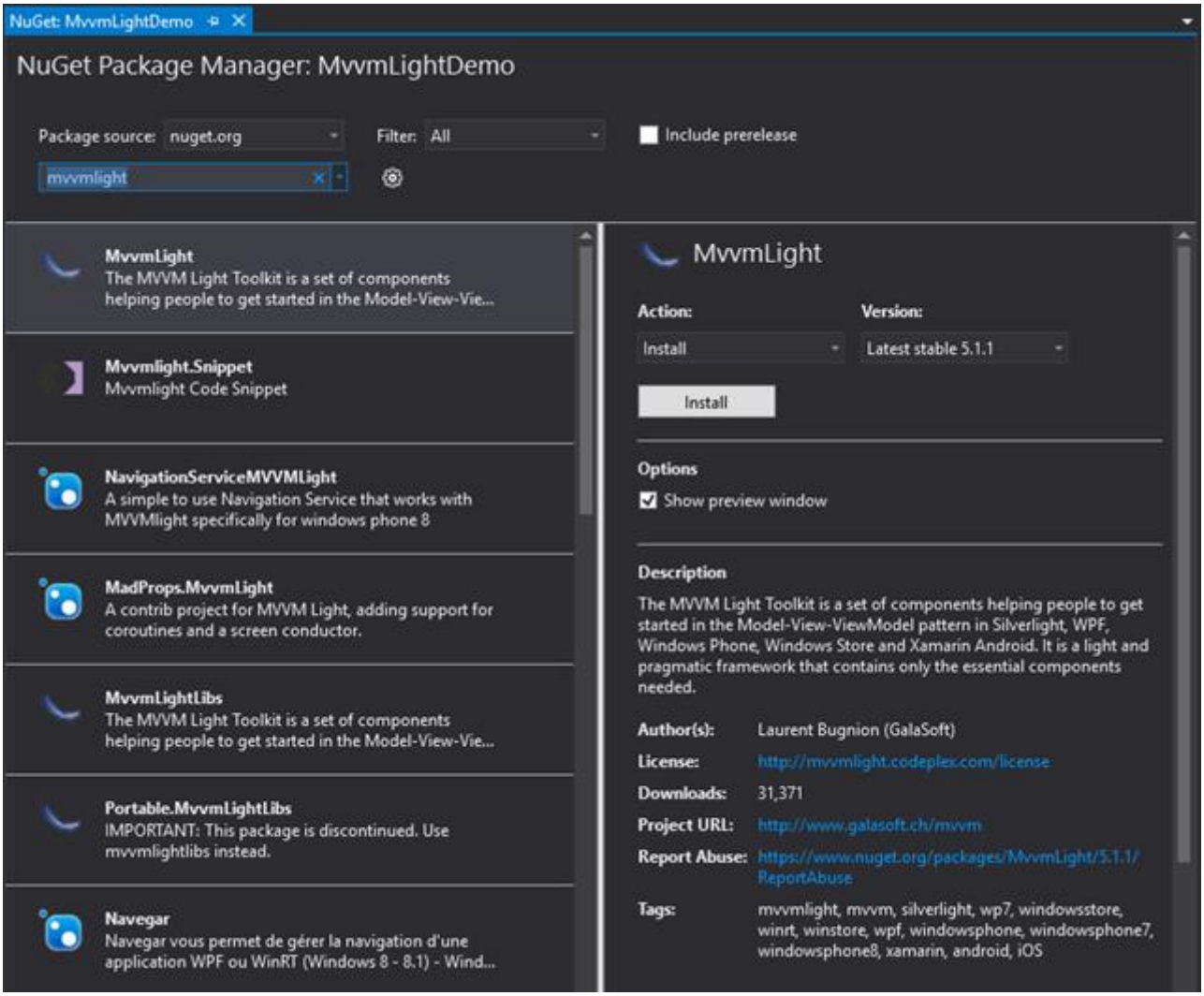
C'est parti !

Installer Mvvm Light

Pour commencer nous créons un projet via Template / Visual C# / Windows / Universal puis nous choisissons un projet vide (Blank App) comme le montre la capture ci-dessous.

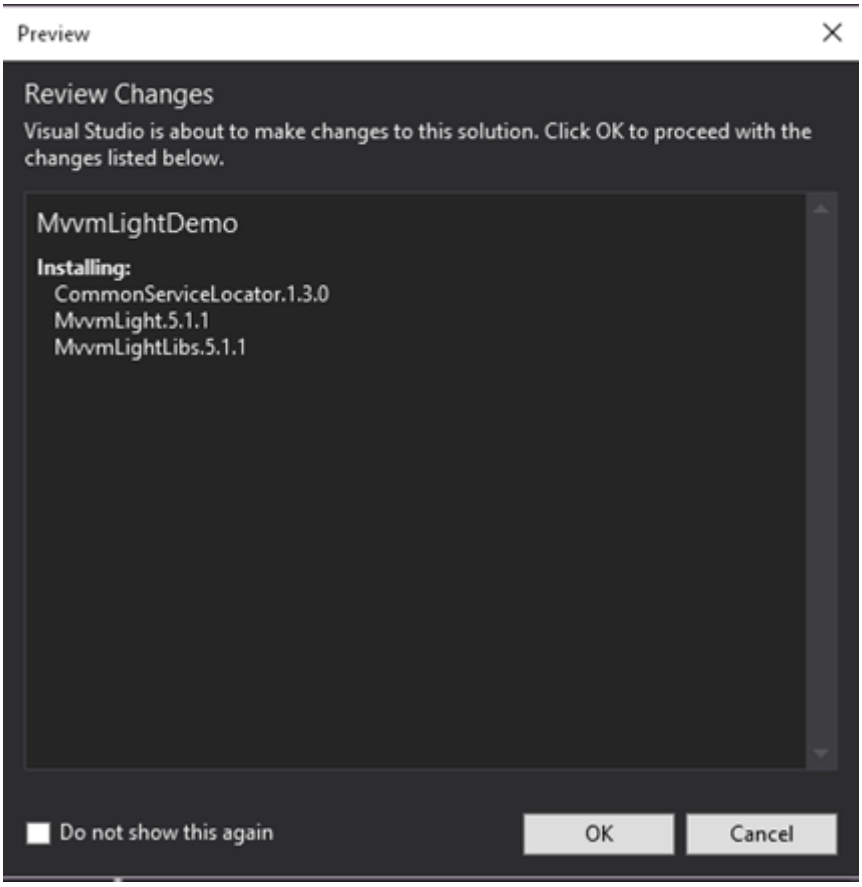


Clic droit sur Référence du projet dans l'explorateur de solution puis gestion des paquets Nuget. On remarque au passage le tout nouveau look de cet écran totalement intégré à VS.

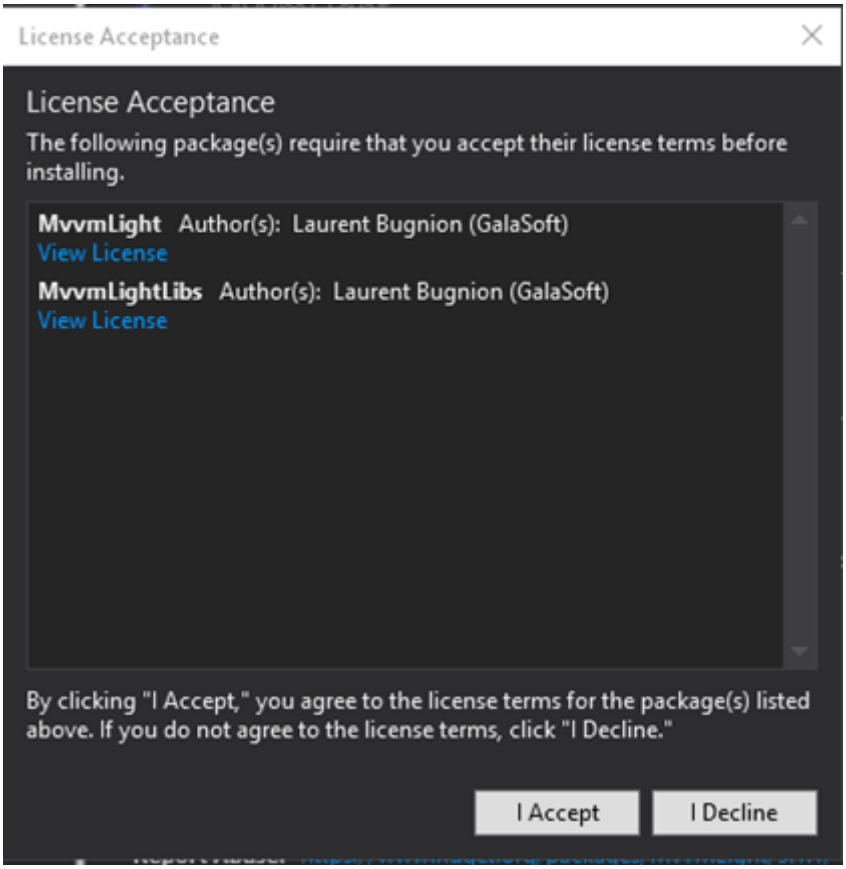


On tape dans la zone de recherche “mvvmlight” et le premier élément dans la liste de gauche est le bon. A droite les détails du paquet avec le choix de la version. La 5.1.1 est la dernière stable au moment où j’écris ces lignes, c’est elle que je vais ajouter en cliquant sur le bouton “install” à droite.

On passe alors un premier dialogue de confirmation pour l’installation des paquets :



puis un second pour la validation de la licence :



On accepte tout cela et on revient à notre application…

On s'aperçoit que si le paquet s'est bien installé le projet n'a pas été modifié comme d'habitude pour y intégrer les petites choses dont on a besoin généralement comme le *ViewModel Locator* par exemple. Donc il va falloir ajouter tout cela "à la main" mais ce n'est pas grand chose. Laurent est en train de travailler à compléter les templates et il est possible que lorsque vous lirez cet article ces manipulations ne soient plus nécessaires. Mais sinon, suivez le guide !

Modifier App.Xaml.cs

Mvvm Light met à la disposition du développeur un Dispatcher qui permet de s'assurer qu'une action sera réalisée sur le thread de l'UI. Cela est utile lorsqu'on veut modifier un élément d'UI depuis un autre thread. Ce sont des choses que j'ai déjà présentées et je n'oublie pas que nous ne réalisons qu'une sorte de Hellow World pour MVVM. Restons simples, même dans les explications !

Ce Dispatcher doit être initialisé sur le thread principal pour fonctionner correctement, c'est donc naturellement dans la classe App qu'on va ajouter en fin de méthode `OnLaunched` ce petit bout de code :

```
DispatcherHelper.Initialize();
```

Les explications sont plus longues que le code… Mais puisque nous sommes dans App et sa méthode `OnLaunched` profitons-en pour ajouter à la suite du code précédent cela :

```
Messenger.Default.Register<NotificationMessageAction<string>>(
                this,
                handleNotificationMessage);
```

Il s'agit ici de démontrer le fonctionnement de la messagerie et pour ce faire nous allons enregistrer un gestionnaire de message pour le type `NotificationMessageAction<string>`. De ce fait les messages de ce type qui seront envoyés de n'importe où dans le code aboutirons tous dans `App` et sa méthode `HandleNotificationMessage` qu'il nous reste à ajouter à la suite de la méthode `OnLaunched` par exemple pour rester grouper …

```
private static void handleNotificationMessage(NotificationMessageAction<string> message)

{

  message.Execute("Reçu et Traité par App.xaml.cs!");

}
```

Ici rien de bien intelligent il faut dire puisqu'on se limite à exécuter le delegate passé dans le message et à retourner un texte de confirmation (pur exemple sans grand intérêt fonctionnel dans la démo).

C'est tout ce qu'il y a à faire mais c'est important, au moins pour le Dispatcher, la gestion des messages est là pour la démo mais n'est pas obligatoire.

Model

Dans MVVM, il y a Model… C'est donc tout naturellement que nous allons créer un réperoire Model dans l'application et y placer les … modèles.

La "blendabilité" de Mvvm Light s'exprime notamment par une construction utilisant une interface d'accès aux données. C'est le fonctionnement de base qui est là plutôt pour montrer le principe. Dans un programme réel où les modèles sont nombreux et complexes on utilisera d'autres approches ou une version plus sophistiquées que celle que nous allons voir. Mais cet article n'est ni un cours sur MVVM, ni sur Mvvm Light. Alors passons à l'action !

Créons tout d'abord les "données" que nous allons manipuler dans la démo. Il s'agira ici uniquement d'une instance de classe véhiculant une chaine de caractère, un titre qui sera affiché par la vue. Bien entendu la réalité est bien plus sophistiquée et les données proviendront d'un réseau, d'une base de données, d'une couche DAL ou Entity Framework etc.

```
namespace MvvmLightDemo.Model

{

    public class DataItem

    {

        public string Title
```

```
        {
            get;
            private set;
        }

        public DataItem(string title)
        {
            Title = title;
        }
    }
```

Créons l'interface de gestion des données :

```
using System.Threading.Tasks;

namespace MvvmLightDemo.Model
{
    public interface IDataService
    {
        Task<DataItem> GetData();
    }
}
```

Cette interface n'expose qu'une méthode, `GetData`, qui programmation moderne fluide et réactive oblige est implémentée sous la forme d'une *méthode asynchrone*. Cela est raisonnable pour un accès à des données qui dans les faits vont solliciter disques, réseaux, middlewares, etc.

Il nous faut maintenant deux implémentations de cette interface, la première pour fournir les données "réelles" et la seconde pour le design-time afin de disposer d'une

simulation des données pour nous aider à faire la mise en page (la fameuse “blendabilité” donc).

Voici la première implémentation, le service d’accès aux données réelles :

```
using System.Threading.Tasks;


namespace MvvmLightDemo.Model
{
    public class DataService : IDataService
    {
        public Task<DataItem> GetData()
        {
            return Task.FromResult(new DataItem("Hello World - Données réelles !"));
        }
    }
}
```

Si vous avez en tête la petite série sur `Task` que je publie en ce moment vous noterez j’en suis certain l’utilisation du `FromResult` ici, mais je n’en dis pas plus ce n’est pas le sujet, juste un petit signal …

Bien entendu la classe ci-dessus retourne des données immédiates, c’est une démo. Il n’y a pas d’accès à une base de données ni rien d’autre. Donc si nous utilisions directement une instance pour nourrir notre ViewModel nous aurions des données de design affichées. Mais ce n’est qu’une conséquence de l’effet réducteur de la démo.

C’est pourquoi pour le design time nous allons implémenter une nouvelle fois l’interface :

```
using System.Threading.Tasks;

using MvvmLightDemo.Model;
```

```
namespace MvvmLightDemo.Design
{
    public class DesignDataService : IDataService
    {
        public Task<DataItem> GetData()
        {
            return Task.FromResult(new DataItem("Hello World - Données de Design !"));
        }
    }
}
```

Pour rendre lisible la structure du projet cette classe est créée dans un sous-répertoire appelé Design. C'est là qu'on mettra dans la réalité toutes les implémentations de données de design pour toutes les interfaces que nous aurons définies.

Nous avons initialisé le `Dispatcher`, ajouter un mécanisme de réception pour un certain type de message, nous avons créé les modèles, une interface et deux implémentations, l'une réelle, l'autre pour le Design, passons au ViewModel !

ViewModel

Pour pasticher une célèbre pub on pourrait dire "Le ViewModel c'est le cœur de la maison !". En effet c'est là que se joue le lien entre les modèles que nous avons définis et la vue que nous créerons ensuite. Sans le ViewModel pas de données adaptées, pas de commandes répondant à l'utilisateur, rien.

Pour les besoins de la démonstration nous allons créer un ViewModel démontrant plusieurs facettes de MVVM et de Mvvm Light. C'est un peu plus long que le code d'un Hello World pour la console en C#…

```
namespace MvvmLightDemo.ViewModel
{
```

```
public class MainViewModel : ViewModelBase

{

    public const string ClockPropertyName = "Clock";

    public const string WelcomeTitlePropertyName = "WelcomeTitle";

    private string clock = "Démarrage";

    private readonly IDataService dataService;

    private readonly INavigationService navigationService;

    private int counter;

    private RelayCommand incrementCommand;

    private RelayCommand<string> navigateCommand;

    private bool clockRunning;

    private RelayCommand sendMessageCommand;

    private RelayCommand showDialogCommand;

    private string welcomeTitle = string.Empty;



    public string Clock
    {
        get
        {
           return clock;
        }
        set
        {
            Set(ClockPropertyName, ref clock, value);
        }
    }

    public RelayCommand IncrementCommand
    {
        get
        {
            return incrementCommand ??
                (incrementCommand = new RelayCommand(
                () =>
                {
                    WelcomeTitle =
                        string.Format("Compteur de clic = {0}", ++counter);
                }));
        }
```

```
        }

        public RelayCommand<string> NavigateCommand
        {
            get
            {
                return navigateCommand ??
                       (navigateCommand =
                          new RelayCommand<string>(
                           p =>
              navigationService.NavigateTo(ViewModelLocator.SecondPageKey, p),
                           p => !string.IsNullOrEmpty(p)));
            }
        }

        public RelayCommand SendMessageCommand
        {
            get
            {
                return sendMessageCommand
                    ?? (sendMessageCommand = new RelayCommand(
                    () =>
                    {
                        Messenger.Default.Send(
                            new NotificationMessageAction<string>(
                                "Test",
                                reply =>
                                {
                                    WelcomeTitle = reply;
                                }));
                    }));
            }
        }

        public RelayCommand ShowDialogCommand
        {
            get
            {
                return showDialogCommand
                       ?? (showDialogCommand = new RelayCommand(
                           async () =>
                           {
                               var dialog =
                                ServiceLocator.Current.GetInstance<IDialogService>();
                                await dialog.ShowMessage(
                              "Hello World depuis une Universal App !", "Ca marche !");
                           }));
            }
        }


        public string WelcomeTitle
        {
            get
            {
```

```
            return welcomeTitle;
        }
        set
        {
            Set(ref welcomeTitle, value);
        }
    }


    public MainViewModel(
        IDataService dataService,
        INavigationService navigationService)
    {
        this.dataService = dataService;
        this.navigationService = navigationService;
        Task.Run(() => initialize()).Wait();
    }

    public void RunClock()
    {
        clockRunning = true;
        Task.Run(async () =>
        {
            while (clockRunning)
            {
                DispatcherHelper.CheckBeginInvokeOnUI(() =>
                {
                    Clock = DateTime.Now.ToString("HH:mm:ss");
                });
                await Task.Delay(1000);
            }
        });
    }

    public void StopClock()
    {
        clockRunning = false;
    }

    private async Task initialize()
    {
        try
        {
            var item = await dataService.GetData();
            WelcomeTitle = item.Title;
        }
        catch (Exception ex)
        {
            WelcomeTitle = ex.Message;
        }
    }
}
```

Sans trop entrer dans les détails je vais pointer toutes les petites choses intéressantes de ce code :

ViewModelBase

D'abord vous remarquerez que le `ViewModel` descend de `ViewModelBase`, une classe fournie par Mvvm Light et qui contient tout ce qu'il faut pour simplifier l'écriture d'un ViewModel, notamment l'indispensable support d'INPC (petit nom qu'on donne pour aller plus vite à l'interface `INotifyPropertyChanged`).

Clock

La prorpriété `Clock` contient une chaine qui indique l'heure courante. Plusieurs choses intéressantes sont à noter ici. La première est que cette horloge n'est pas gérée par un timer *mais par une tâche* possédant un délai. Il s'agit d'une *Promise Task* et non d'une *Delegate Task*, c'est à dire une tâche évènement n'exécutant aucun code. *Task.Delay n'est pas identique à un Thread.Sleep* de ce point de vue. Mais cela vous le savez si vous suivez la série sur les Task publiée en ce moment…

L'horloge est démarrée par la méthode `StartClock()` et arrêtée par `StopClock()`. C'est dans la première que se trouve la tâche responsable de son fonctionnement.

L'utilisation d'une tâche plutôt que d'un timer n'est pas que stylistique, la programmation asynchrone apporte des bénéfices différents. Ici encore je ne fais que pointer les choses à voir, développer à chaque fois nous entrainerait trop loin dans le cadre du présent article (d'autant que ce sont des choses que j'explique ou ai expliquées dans de nombreux billets).

Enfin on peut se demander pourquoi `StartClock()` n'est pas appelé par le constructeur du ViewModel et `StopClock()` par un évènement de sortie. D'ailleurs où est utilisé `StartClock` ? Nulle part ! En effet pas dans le code du ViewModel. Mais où alors ? Je triche un peu mais nous le verrons dans le code-behind de la `MainPage`… On peut déjà dire qu'une des raisons est l'absence dans le ViewModel de moyens simples de savoir s'il y a "sortie" ou non alors que cette information est de la connaissance du système de navigation dont la Vue est elle au courant. Mais nous pouvons si nécessaire contourner ce problème. L'autre raison est qu'il est intéressant de montrer comment la Vue peut dialoguer avec son ViewModel sans briser les principes de MVVM. Exercice de style donc.

IncrementCommand

Il s'agit d'une commande. En MVVM on utilise l'interface `ICommand` pour gérer les commandes que l'utilisateur peut donner au ViewModel. Implémenter `ICommand` est

fastidieux. C'est pourquoi Mvvm Light nous propose la classe `RelayCommand` qui simplifie la création de commandes. Elle supporte déjà `ICommand` et c'est donc comme cela que sont vues les `RelayCommand` par la Vue. Le développeur lui voit le côté `RelayCommand` ce qui est plus pratique.

Sinon `IncrementCommand` est un exemple de commande toute bête montrant comment à chaque appel un compteur est incrémenté. Le résultat est passé pour simplifier à la même zone que celle qui contient le titre au lancement. C'est un peu "bordélique" je l'accorde, mais la démo ne peut pas être un code réel. Mais ne faites pas cela dans une application de production ! Les noms et intentions doivent être clairs. La propriété `WelcomeTitle` ne peut contenir qu'un titre de bienvenue, rien d'autre. Soit il faut renommer la propriété soit il faut en créer une autre pour gérer l'affichage du compteur ce qui serait bien plus propre.

NavigateCommand

Cette commande nous permet de voir comment le service de navigation est utilisé par le ViewModel. C'est une problématique typique de MVVM, qui de la poule ou de l'œuf… qui du ViewModel ou de la Vue est en charge de la navigation ? S'agissant d'une action sur les vues ce sont ces dernières qui souvent possèdent un système de navigation proposé par la plateforme. Et c'est le cas dans UWP. Hélas ce n'est pas le code-behind de la Vue qui doit s'occuper de cela, c'est la responsabilité du ViewModel en MVVM que de répondre aux commandes de l'utilisateur et de décider d'activer telle ou telle Vue.

MvvmLight utilise depuis quelques versions un conteneur IoC qui permet de faire de l'injection de dépendances. C'est ainsi que le constructeur du ViewModel s'attend à recevoir une instance du système de navigation qui lui sera passé par le conteneur IoC au moment de son instanciation.

La navigation mise en œuvre ici active donc l'affichage d'une seconde page. Elle n'utilise pas de ViewModel (parfois cela n'est pas nécessaire) et nous l'étudierons plus loin.

SendMessageCommand

Cette commande sert à montrer l'utilisation de la messagerie MVVM proposée par Mvvm Light. Un message de type `NotificationMessageAction` est envoyé par le ViewModel sans qu'il sache qui le traitera. Le découplage est donc très fort entre demandeur et exécuteur.

Le type de message envoyé est particulier, c'est un message de notification possédant une action, un delegate. Le récepteur récupère la chaine passée dans le message (c'est l'objet principal d'un message de notification, un bout de texte envoyé comme une bouteille à la mer) mais il a aussi la responsabilité d'exécuter l'action qui est ajoutée au texte. Cela permet à l'envoyeur du message de prévoir un traitement à réaliser une fois le message traité sans que le récepteur ne sache de quoi il s'agit. Celui qui reçoit le message ne sait donc rien du code à exécuter, il n'y a pas non plus de callback qui pourraient poser des problèmes de libération de mémoire, ni d'évènement. C'est une façon originale de programmer dans un mode totalement découplé… et surtout de répondre à des problèmes bien concrets qui se posent lorsqu'on applique MVVM !

Si vous revenez beaucoup plus haut dans cet article au paragraphe traitant les modifications de `App.Xaml.cs` vous retrouver… L'enregistrement du récepteur des notifications du même type. Mvvm Light collectera le message et le distribuera à tous les récepteurs inscrits pour le type de message envoyé. Dans notre application il n'y a qu'un point de réception et c'est la classe App.

Dans notre implémentation ultra simple `App` ne s'occupe en réalité pas de la chaine de notification passée dans le message. En revanche elle exécute bien le delegate passée dans ce dernier. Et que fait-il ? Il modifie la valeur de `WelcomeTitle` en lui affectant "`reply`". D'où vient ce dernier ? C'est le paramètre chaine de l'action. Et qui donne une valeur à ce paramètre ? Celui qui appelle l'action bien entendu, donc le code de `App`, et c'est donc le texte "`Reçu et traité par App.xaml.cs !`" qui sera passé au delegate qui modifiera `WelcomeTitle` …

Ne vous inquiétez pas, au début ce n'est pas facile à piger, c'est normal. Il faut avoir en tête la totalité de touts ces mécanismes pour s'en faire une image correcte et cela ne vient pas d'un coup d'un seul comme une révélation ésotérique. On mesure d'ailleurs ici le fossé entre l'apparente simplicité d'un framework comme Mvvm Light et la complexité de son utilisation correcte…

ShowDialogCommand

Encore une autre facette de Mvvm Light et une autre problématique typique de MVVM : les dialogues.

En effet en MVVM le code se trouve dans le ViewModel, en tout cas celui qui peut être amené à réagir à une action de l'utilisateur pouvant éventuellement déboucher sur l'affichage d'un message. Dans de rares cas cela peut être aussi le Model qui initie

un dialogue, d'une part parce qu'il peut rencontrer par exemple un problème dans le traitement des données et qu'il peut être plus simple d'envoyer le message là où le problème se pose plutôt que de le faire dans tous les ViewModel qui exploitent la même source et puis parce que MVVM autorise dans les cas simple à ce que la Vue soit connectée directement au Model ce qui fait que le code utile piloté par l'utilisateur se trouve alors dans celui-ci et non dans un ViewModel.

Or les ViewModel et plus encore les Model sont des classes qui n'ont rien à voir avec l'UI et qui surtout ne doivent rien à voir avec elle !

Dès lors comment afficher un dialogue depuis un code qui n'a pas accès aux fonctions de l'UI qui le permettent ?

De même qu'il existe un service de navigation que les ViewModel peuvent manipuler alors que la navigation implique de manipuler des vues le tout en conservant un découplage fort entre Vue et ViewModel, il existe un service de dialogue qui permet à tout code, mais principalement les ViewModel, d'afficher des boites de dialogue (qu'elles attendent une réponse ou non).

La commande `ShowDialogCommand` utilise donc ce fameux service de dialogue qui est ici obtenu non pas par injection de dépendances mais en interrogeant directement le ServiceLocator ce qui met en lumière une autre façon d'obtenir des services (on aurait pu aussi le faire par injection dépendance c'est juste pour varier les exemples dans la démo). Le service de dialogue offre un moyen simple d'afficher un dialogue et d'attendre la réponse le tout sans bloquer l'application puisque tout cela est réalisé en asynchrone (vous comprenez maintenant pourquoi ces derniers temps j'insiste sur `Task` et sur l'asynchronisme, c'est "la" façon de programmer sous UWP (pas que) !).

WelcomeTitle

C'est une propriété texte dont le contenu sert un peu à tout dans cette démo. Au départ elle contient un texte de bienvenu de type Hello World mais au fil des actions elle affichera d'autres informations. Comme déjà dis une zone fourre-tout n'est jamais une bonne idée surtout quand elle porte un nom précis. Programmer en faisant en sorte que l'intention du développeur soit claire est un impératif totalement violé ici. Mais ce n'est qu'une démo et c'est assumé.

MainViewModel() et Initialize()

Le constructeur de notre ViewModel accepte plusieurs paramètres de type interface. Les interfaces rappelons-le sont les premières pierres qui ont été posées pour construire l'édifice de la programmation découplée. Bien qu'étant un concept très

ancien (Delphi, Java proposaient déjà cette notion d'interface il y a bien une vingtaine d'années !) il est toujours aussi moderne et on l'utilise le plus souvent possible dans le cadre d'une programmation moderne.

Comme dit un peu avant le constructeur récupère les instances de ces interfaces via l'injection de dépendances effectuée par le conteneur IoC proposé par Mvvm Light. Par défaut ce conteneur est très simple (mais bien suffisant pour une majorité d'applications) et il est possible d'en changer à l'initialisation de l'application (notamment grâce à l'utilisation des interfaces), par exemple pour utiliser Unity qui est beaucoup plus sophistiqué (mais bien d'autres existent comme AutoFac ou Ninject qui sont maintenus depuis des années).

En dehors de stocker les valeurs reçues, le constructeur appelle une méthode d'initialisation qui va chercher les données dans le Model. S'agissant d'une méthode async il est nécessaire de l'attendre, c'est plus clair, même si ici cela ne joue aucun rôle puisque quand les données arriveront les propriétés ad hoc seront modifiées et que cela entrainera par le jeu d'INPC la notification des contrôles XAML bindés qui se mettront alors à jour. On pourrait donc se passer du `".Wait()"` sur l'appel surtout qu'on est en fin de méthode (donc l'intérêt d'un point de retour vers le contexte est sans intérêt). En revanche s'agissant d'un constructeur on ne pourrait pas faire un `await` car il n'est possible d'avoir un constructeur asynchrone (pour des raisons évidentes, comme il ne peut y avoir de `Main()` asynchrone)…

StartClock et StopClock

Déjà évoqué plus haut ces deux méthodes servent à démarrer et arrêter l'affichage d'une horloge. On remarque donc que l'horloge est gérée par du code asynchrone et non pas par un timer et que c'est un `Task.Delay` et non un `Thread.Sleep` qui est utilisé. Ces subtilités sont décrites dans la série sur Task que je publie en parallèle de cet article.

Ces méthodes ne sont pas appelées dans le constructeur, elles le seront dans le code-behind de la `Mainpage` pour montrer comment sans briser le découplage une Vue peut éventuellement faire appel à du code du ViewModel. Ce qui est une situation rare mais d'une grande utilité dans certains cas. MVVM ayant à la base une rigidité posant certains problèmes que justement des librairies comme Mvvm Light solutionnent.

La Vue

Nous avons décrit le Model, le ViewModel, reste la Vue.

Voici d'abord à quoi la vue ressemble (sa mise en page est en mode téléphone, je reparlerai plus tard des stratégies de la Reactive UI qui permettent avec un même code XAML de s'adapter à tous les form factors) :

Rien de bien exotique, une grille, un stackpanel et des boutons ou des zones de texte et des bindings comme on sait le faire en XAML depuis longtemps ce qui donne le code XAML suivant :

```
<Page

    x:Class="MvvmLightDemo.MainPage"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:local="using:MvvmLightDemo"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d"
```

```
DataContext="{Binding Main, Source={StaticResource Locator}}">



<Grid>
    <Grid.Background>
        <LinearGradientBrush EndPoint="0.906,1.164"
                             StartPoint="0.244,-0.159">
            <GradientStop Color="#FFA20000" Offset="0" />
            <GradientStop Color="#FFCD7C10" Offset="1" />
        </LinearGradientBrush>
    </Grid.Background>

    <StackPanel Background="#FF1F1F1F"
                Width="300"
                HorizontalAlignment="Center"
                Height="450"
                VerticalAlignment="Center">
        <Button Content="Incrémenter compteur"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                Margin="0,0,0,20"
                FontSize="24"
                Command="{Binding IncrementCommand, Mode=OneWay}"
                Foreground="White" />

        <Button Content="naviguer vers page 2"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                Margin="0,0,0,20"
                FontSize="24"
                Command="{Binding NavigateCommand, Mode=OneWay}"
                CommandParameter="{Binding Text, ElementName=NavigationParameterText}"
                Foreground="White" />

        <TextBox x:Name="NavigationParameterText"
                 Text="Texte pour naviguer..."
                 Margin="0,0,0,20"
                 FontSize="24"
                 Foreground="White" />

        <Button Content="Afficher un dialogue"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                Margin="0,0,0,20"
                FontSize="24"
                Command="{Binding ShowDialogCommand, Mode=OneWay}"
                Foreground="White" />


        <Button Content="Envoyer message MVVM"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                Margin="0,0,0,20"
                FontSize="24"
```

```
                    Command="{Binding SendMessageCommand, Mode=OneWay}"
                    Foreground="White" />

            <TextBlock TextWrapping="Wrap"
                       Text="{Binding WelcomeTitle}"
                       FontFamily="Segoe UI Light"
                       FontSize="24"
                       HorizontalAlignment="Center"
                       TextAlignment="Center"
                       Foreground="White" />

            <TextBlock FontFamily="Segoe UI Light"
                       FontSize="18.667"
                       HorizontalAlignment="Center"
                       TextAlignment="Center"
                       Foreground="White"
                       Margin="23.82,20,23.18,0"
                       Text="{Binding Clock}" />
        </StackPanel>
    </Grid>
</Page>
```

Il n'y a donc rien à dire sur ce code en dehors des bindings qui implique un `DataContext` qu'on peut voir en haut. Ce morceau là est intéressant car si on le regarde bien on s'aperçoit qu'il utilise une ressource statique "`Locator`" pour se lier à sa propriété "`Main`". D'où cela vient-il ?

Mvvm Light utilise une stratégie de centralisation des ViewModel via un ViewModel Locator. On notera que cette stratégie date des premières versions de Mvvm Light pour permettre justement la blendabilité (avoir des données de design, même sous VS aujourd'hui). Or cette approche n'est pas une obligation fonctionnelle de Mvvm Light, on peut fort bien construire une application sans Locator en bindant les vues directement en dur sur les ViewModel (violation de Mvvm qui ne me semble pas si grave avec le recul et parfaitement acceptable dans certains cas) ou au contraire mettre en place un Locator plus sophistiqué. La notion de Locator est donc annexe à Mvvm Light, ce n'est pas un principe fondamental de la librairie.

C'est donc ce locator qui est utilisé ici. Mais comment est-il défini et où est créée la ressource ?

Le ViewModelLocator

Son rôle est de créer un point d'accès centralisé pour les ViewModel ce qui permet au passage de fournir des données de Design. Le découplage reste fort puisque la vue ne connait l'instance qu'au travers de son `DataContext` lié à une propriété du locator.

On pourrait aller encore plus loin et créer des interfaces pour chaque ViewModel et n'exposer dans le Locator que ces dernières. C'est beaucoup de travail pour un découplage qui dans la pratique s'avère sans grand intérêt comme je le disais plus haut.

Toutefois le ViewModel Locator ne fait tellement pas partie de Mvvm Light … qu'il faut l'écrire soi-même !

Le code du locator de notre application est le suivant :

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Ioc;
using GalaSoft.MvvmLight.Views;
using Microsoft.Practices.ServiceLocation;
using MvvmLightDemo.Model;



namespace MvvmLightDemo.ViewModel
{
    public class ViewModelLocator
    {
        public const string SecondPageKey = "SecondPage";
        static ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
            var nav = new NavigationService();
            nav.Configure(SecondPageKey, typeof(SecondPage));
            SimpleIoc.Default.Register<INavigationService>(() => nav);
            SimpleIoc.Default.Register<IDialogService, DialogService>();

            if (ViewModelBase.IsInDesignModeStatic)
            {
                SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
            }
            else
            {
                SimpleIoc.Default.Register<IDataService, DataService>();
            }

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



        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Performance",
            "CA1822:MarkMembersAsStatic",
            Justification = "This non-static member is needed for data binding purposes.")]
        public MainViewModel Main
        {
            get
            {
```

```
                return ServiceLocator.Current.GetInstance<MainViewModel>();
            }
        }
    }
}
```

On remarque l'utilisation du conteneur IoC qui permet de faire de l'injection de dépendances et qui joue aussi le rôle de cache pour les instances. Il y a beaucoup à dire sur le sujet que j'ai déjà abordé à plusieurs reprises alors passons à la suite : où est instancié ce locator ?

Pour qu'il soit visible de partout il faut l'instancier dans un code accessible comme celui de `App`. Mais il est plus intelligent de le créer en tant que ressource puisqu'il sera utilisé par du code XAML, donc ce sera dans `App.Xaml` à l'intérieur d'un dictionnaire de ressources XAML :

```
<Application

    x:Class="MvvmLightDemo.App"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:viewModel="using:MvvmLightDemo.ViewModel"

    RequestedTheme="Light">



    <Application.Resources>

    <viewModel:ViewModelLocator x:Key="Locator"  />

    </Application.Resources>



</Application>
```

Dans les ressources de l'application, donc bien un niveau central, la ressource "`Locator`" est créée par instanciation de la classe `ViewModelLocator` se trouvant dans l'alias "`viewModel`" lui-même défini plus haut comme "`using:MvvmLightDemo.ViewModel`" c'est à dire l'espace de noms du répertoire

contenant nos ViewModel et le locator (il est logique de le placer à cet endroit mais on pourrait faire autrement).

Ces déclarations permettent à la Vue de lier son `DataContext` au ViewModel principal tout en respectant le découplage fort typique de MVVM même si, *once more*, je n'ai en pratique jamais vu une Vue changer subitement de ViewModel, jamais. On pourrait donc discuter des heures de ce découplage-là qui s'impose comme règle inviolable de MVVM mais qui ne sert à rien en pratique (et complique les choses comme la présence d'un Locator dont on pourrait ainsi se passer).

Le code-behind de la Vue

On entend souvent dire qu'en MVVM il n'y a aucun code dans le code-behind de la Vue. C'est faux. En tout cas c'est partiellement faux. S'il est vrai qu'aucun code fonctionnel ne doit se trouver à cet endroit *il est en revanche parfaitement licite d'y faire apparaitre du code purement UI*, c'est même une *obligation* car on voit mal un tel code être placé dans les ViewModel qui ont interdiction de connaitre l'UI et ses namespaces.

Pour illustrer cette possibilité même si ici l'exemple n'est pas le plus parlant, le code-behind de la vue principal contient ceci :

```
using Windows.Foundation.Metadata;
using Windows.Phone.UI.Input;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
using MvvmLightDemo.ViewModel;

namespace MvvmLightDemo
{
    public sealed partial class MainPage : Page
    {
        public MainViewModel Vm => (MainViewModel)DataContext;


        public MainPage()
        {
            InitializeComponent();
            if (ApiInformation.IsTypePresent("Windows.Phone.UI.Input.HardwareButtons"))
            {
                HardwareButtons.BackPressed += onBackPressed;
            }
            Loaded += (s, e) =>
            {
                Vm.StartClock();
            };
        }
```

```
        protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
        {
            Vm.StopClock();
            base.OnNavigatingFrom(e);
        }

        private void onBackPressed(object sender, BackPressedEventArgs e)
        {
            if (!Frame.CanGoBack) return;
            e.Handled = true;
            Frame.GoBack();
        }
    }
}
```

La propriété `MainViewModel` est définie en utilisant une syntaxe très moderne, décryptée là pour la comprendre 

Le fameux code placé dans ici est celui qui initialise l'horloge affichée en bas de page principale et celui qui stoppe cette horloge. Il n'y a pas de boutons pour ces fonctions, l'horloge démarre quand la page s'affiche et doit s'arrêter dès que la page est désactivée, des informations que la Vue possède et pas le ViewModel.

C'est donc sur le `Loaded` de la Vue qu'on programme un gestionnaire qui déclenchera l'horloge. Et c'est naturellement par une surcharge de `OnNavigatingFrom` qu'on arrêtera l'horloge avant de laisser la navigation s'effectuer.

De même on gère un bouton "`Back`" dont le gestionnaire `onBackPressed` utilise les mécanismes de navigation propres aux Vues pour réaliser l'action. Cela ne pourrait pas se trouver dans le ViewModel (qui montre par ailleurs comment il est capable de naviguer aussi via le service de navigation).

Mais le code le plus étrange ici et celui que vous ne reconnaitrez certainement pas c'est celui du test :

```
if (ApiInformation.IsTypePresent("Windows.Phone.UI.Input.HardwareButtons"))
            {
                HardwareButtons.BackPressed += onBackPressed;
            }
```

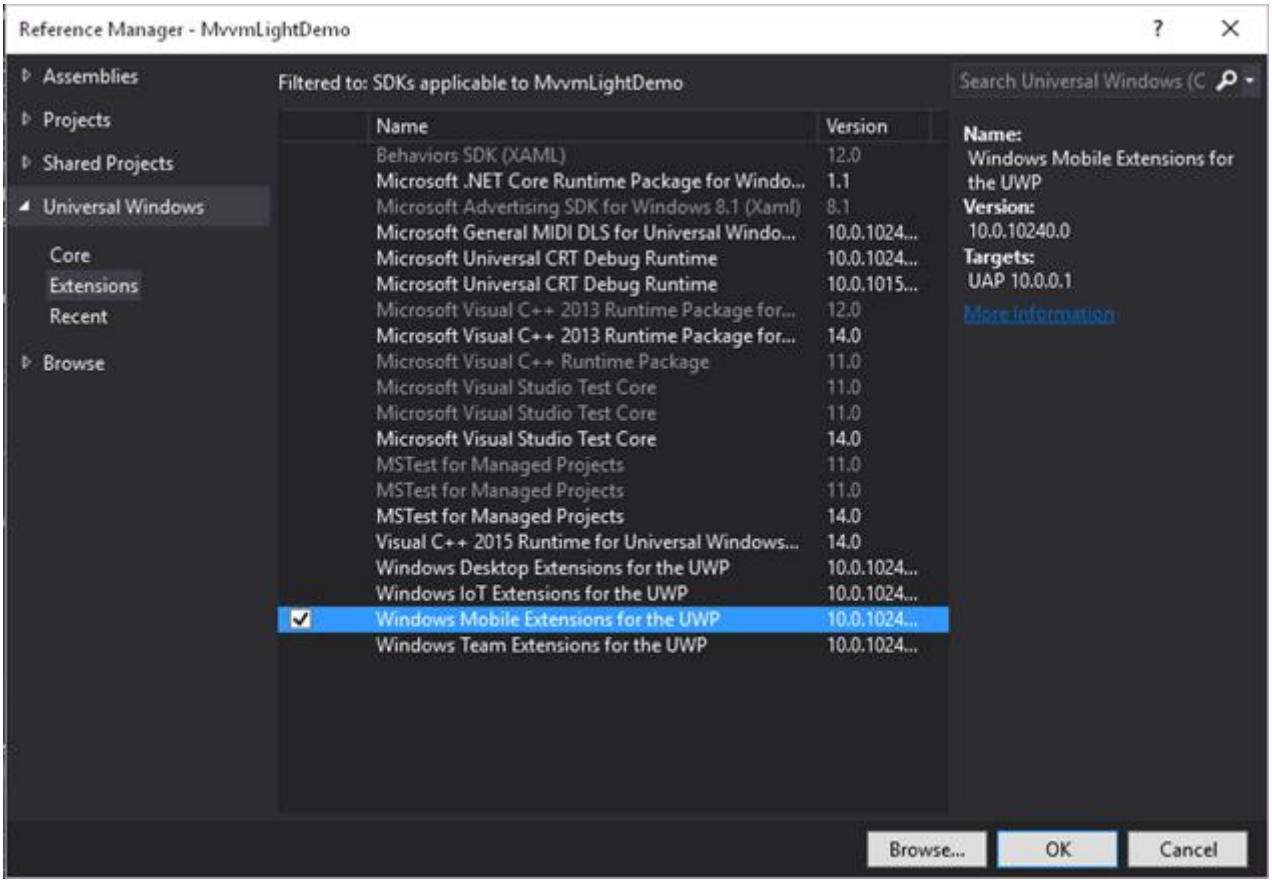
Quel est ce type que nous testons en passant un namespace complet ? Et d'où vient-il ?

On touche aussi à la façon dont UWP sur une base totalement commune à tous les form factors permet malgré tout d'utiliser des spécificités de l'une ou de l'autre. Sur un PC il n'y a pas de bouton physique "retour". En revanche sur un smartphone si (physique ou tactile c'est pareil pour nous).

Par souci d'ergonomie et de **Reactive Design**, un Design adapté à toutes les cibles qui s'adapte tout seul au runtime, nous souhaitons prendre en compte la présence d'un tel bouton quand il existe. Mais quand il existe seulement…

Cela s'effectue en deux temps.

Dans un premier temps nous allons ajouter aux références de notre projet les **extensions pour unités mobiles** :



Grâce à ses extensions, qu'elles soient implémenter ou non en réalité sur une cible donnée, nous avons la possibilité dans notre code "Universel" de nous servir de ce qu'elles offrent, comme ici l'accès au bouton "retour".

J'ai déjà longuement expliqué **l'injection de code natif** notamment dans les vidéos sur MvvmCross. Microsoft utilise ici une méthode similaire.

En revanche ce n'est pas parce que nous rendons disponible à notre App un code spécifique à un form factor que cela veut dire qu'on peut s'en servir sur n'importe quelle machine !

Bien entendu sur un PC par exemple l'utilisation directe du bouton "back" ne donnera pas "rien" mais un beau plantage *! Il faut donc tester la disponibilité de l'API avant de s'en servir.*

Et c'est ainsi qu'on comprend mieux le test en question, il s'assure que l'API "`Windows.Phone.UI.Input.HardwareButtons`" existe avant de programmer un évènement sur le bouton physique de retour. Si l'API n'existe pas, comme sur un PC, l'évènement ne sera pas connecté au bouton. Dans ce cas c'est plutôt au niveau du Design que nous ferons peut-être apparaitre une flèche de retour en haut à gauche (qui sera cachée pour un smartphone). Le code du test peut donc s'occuper de gérer ces différents aspects et c'est bien dans le code-behind de la Vue qu'il doit se trouver car tout cela est inaccessible au ViewModel, cela ne concerne que l'UI.

La vue secondaire

Pour montrer le service de navigation il fallait bien une seconde Vue… Mais elle sera simple et sans ViewModel, pourtant elle va nous permettre de voir plusieurs choses intéressantes…

Côté XAML rien de spécial, un bouton, un texte… passons rapidement :

```
<Page x:Class="MvvmLightDemo.SecondPage"

      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

      xmlns:local="using:MvvmLightDemo"

      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

      mc:Ignorable="d">



    <Grid>

        <Grid.Background>

            <LinearGradientBrush EndPoint="0.5,1"
```

```
                            StartPoint="0.5,0">
            <GradientStop Color="#FF120DE2"
                          Offset="0" />
            <GradientStop Color="#FF86FBF6"
                          Offset="1" />
        </LinearGradientBrush>
    </Grid.Background>

    <StackPanel HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <Button x:Name="GoBackButton"
                Content="Retour"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                Click="GoBackButton_Click"
                FontSize="32"
                Width="150"
                Height="60" />

        <TextBlock x:Name="DisplayText"
                   TextWrapping="Wrap"
                   FontFamily="Segoe UI"
                   FontSize="24"
                   Margin="0,20,0,0"
                   HorizontalAlignment="Center"
                   TextAlignment="Center" />
    </StackPanel>
```

```
    </Grid>

</Page>
```

Ici aussi le bouton de navigation est géré par du code-behind, exactement de la même façon que dans la Vue principale :

```
namespace MvvmLightDemo
{
    public sealed partial class SecondPage
    {
        public SecondPage()
        {
            InitializeComponent();
        }

        private void GoBackButton_Click(object sender, RoutedEventArgs e)
        {
            var nav = ServiceLocator.Current.GetInstance<INavigationService>();
            nav.GoBack();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            DisplayText.Text = e.Parameter.ToString();
            base.OnNavigatedTo(e);
        }
    }
}
```

En revanche la surcharge de `OnNavigateTo` montre une astuce intéressante pour passer des informations d'une Vue à l'autre. Le paramètre de navigation passé à la méthode contient en effet l'objet envoyé par la page principale, ici une simple chaine qui est récupérée et affichée. Mais cela pourrait être un objet complexe autant qu'un simple ID d'une fiche dont il faudrait montrer le détail par exemple.

Pour rappel nous avons sur la page principale une `TextBox` appelée `NavigationParamaterText` qui peut recevoir un texte tapé par l'utilisateur avant que celui-ci n'appelle la navigation vers la page 2. Mais il n'y a pas de propriété équivalente dans le MainViewModel. Comment ce paramètre est-il passé à la seconde Vue alors ?

D'abord disons que ce cas n'est qu'une possibilité, le paramètre pourrait fort bien être fourni par le ViewModel bien entendu. Mais regardons le code XAML du bouton de navigation :

```
<Button Content="naviguer vers page 2"

HorizontalAlignment="Stretch"

VerticalAlignment="Stretch"

Margin="0,0,0,20"

FontSize="24"

Command="{Binding NavigateCommand, Mode=OneWay}"

CommandParameter="{Binding Text, ElementName=NavigationParameterText}"

Foreground="White" />
```

Le bouton est bindé à la commande `NavigateCommand` du ViewModel mais le paramètre de cette commande est lui bindé par un *Element Binding* directement au contenu de la zone de saisie. De fait quand l'utilisateur clic sur le bouton la commande du ViewModel est invoquée et elle reçoit le texte en paramètre. Revisitons le bout de code concerné :

```
        public RelayCommand<string> NavigateCommand
        {
            get
            {
                return navigateCommand
                       ?? (navigateCommand = new RelayCommand<string>(
                           p =>
                            navigationService.NavigateTo(ViewModelLocator.SecondPageKey, p),
                           p => !string.IsNullOrEmpty(p)));
            }
        }
```

La `RelayCommand` qui est créée est une commande générique typée avec un paramètre `string`, c'est pour cela que le paramètre peut directement être utilisé dans ce format. Ensuite le code ci-dessus invoque la navigation vers la page 2 dans son delegate d'action et fixe en second paramètre la valeur du paramètre de navigation, la fameuse chaîne reçue depuis l'UI…

Ce n'est pas très compliqué mais arriver à penser tout un code en l'architecturant avec ces petites choses là en tête demande un peu d'entrainement il est vrai.

Il y a un temps pour expliquer et un autre pour voir…

Le temps des explications est terminé, non pas que j'ai tout dit, mais je pense en avoir "assez" dit pour cette petite prise en main de Mvvm Light en UWP sous Windows 10.

N'ayez crainte je reviendrais forcément sur de nombreux aspects dans les jours et semaines à venir…

Mais maintenant il faut que nous coutions le fruit de nos efforts en exécutant l'application…

Une App, Un code, Une UI, Tous les form factors !

C'est ici que la magie annoncée dans le titre de cet article fleuve s'opère… Dans un premier temps je vais choisir d'exécuter l'application dans l'émulateur Windows 10 :

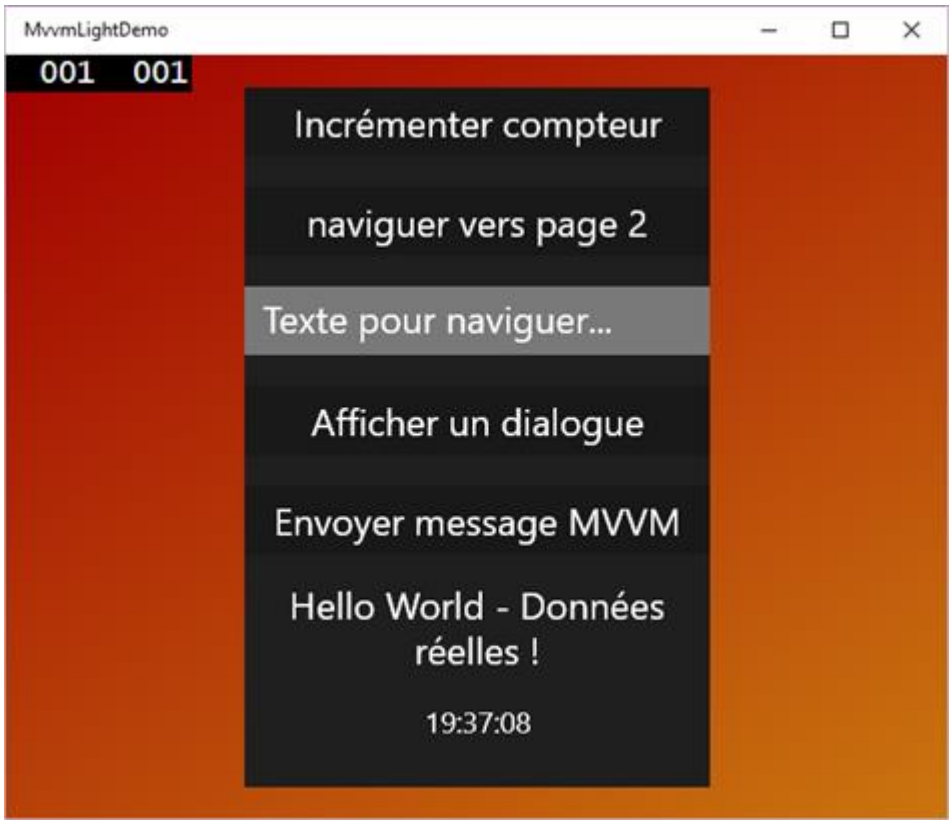
Voici la page principale, et maintenant la page secondaire :

Le bouton “retour” fonctionnera sur toutes les machines mais sur le smartphone on pourra aussi utiliser le bouton en bas à gauche. Le “*texte pour naviguer*” n’est qu’un bout de texte qu’on peut taper sur l’écran principal et qui est passé à la page 2 pour montrer les mécanismes de passage d’information entre les écrans.

On peut retourner l’écran ça marche de base sans rien faire…

Je vous fais grâce de toutes les manipulations (envoi de message, etc) car le code de l’application est joint à l’article et il sera bien plus intéressant de le faire tourner et de l’expérimenter vous-même…

Passons plutôt au côté magique, je vais maintenant lancer l'application en mode "**machine locale**" (c'est à dire déploiement sur ma machine, un PC donc) :



Comme le Design de l'application n'est pas réactif cela donne un truc moche centré. D'où la nécessite de faire un petit effort côté XAML pour faire du Reactive Design. Mais ça marche !

Conclusion

Imaginez-vous deux secondes ce que tout cela signifie ? **Un seul et unique code pour TOUS les form factors**? *UN SEUL exécutable* qui tourne sur une XBox, un PC, Surface, une phablet ou un smartphone !

Certes tout cela dans le monde Microsoft, nous sommes bien en cross-form-factor plutôt qu'en cross-plateforme, la fameuse universalité verticale qui a remplacé le fantasme d'une universalité horizontale. **Mais quel bonheur ! Microsoft répond ici à nos vrais besoins : celui de couvrir tous les form factors de façon cohérente et avec le minimum d'effort**. On attendait cela avec Windows 8 et ce n'était pas réussi, il fallait certainement un peu plus de temps. Mais avec Windows 10 le pari est tenu et gagné !

C'est une bénédiction pour tout le monde, un soulagement on revient à l'époque glorieuse *d'un Windows qui fait tout et qui couvre tous les besoins, avec un seul code*… Fini les tracas du cross-plateforme ! Windows et Microsoft sont de nouveau dans la course. Encore va-t-il falloir reconquérir tout le terrain perdu ces dernières années et occupé par des sociétés comme Google ou Apple qui ne vont pas se laisser balayer de la carte comme ça… Microsoft a eu un trou d'air, et des gros poissons sont tombés dedans et ils occupent le terrain. Difficile de refermer la nasse d'un geste et de revenir à l'hégémonie d'autant. Pourtant c'est ce qu'il faut à notre métier, de la stabilité, de la cohérence. On se moque des tablettes ou smartphones de Google ou Android, franchement. Tout ça fait la même chose… Et les produits Microsoft ne téléphonent pas moins bien ni ont de moins bons écrans. Ils proposent tous la même chose. Mais seul Microsoft nous offre un avenir aussi radieux que notre passé, le temps où un seul OS, Windows, permettait de couvrir 100% des besoins, de planifier sereinement l'avenir. C'est ça qui a fait gagner de l'argent à tout le monde et a permis à la "micro informatique" d'être ce qu'elle est, terme devenu vieux mais ô combien contemporain et même d'avenir…

Windows 10 et surtout UWP offrent enfin une réponse intelligente à un vrai problème, celui de **couvrir plusieurs form factors avec un cout minimum**. Et quand on raisonne en terme de cout et de rentabilité, c'est à dire comme une entreprise, il n'y a pas photo, l'avenir c'est Windows 10 et UWP.

Avec C#, XAML et MVVM …

UWP et cycle de vie des applications

UWP est une plateforme issue de WinRT mais elle s'en éloigne car elle est cross-form-factors et que ses Apps universelles le sont vraiment : un seul code C#, un seul code XAML, un seul exécutable, pas de dépendances à .NET, et un seul OS. Forcément il y a des petites nuances pour réaliser un tel tour de force, mais on retrouve un cycle de vie proche d'un OS mobile…

Une même UX du smartphone au PC

Le modèle de cycle de vie des applications sous UWP est calqué sur celui mis en place dans les OS de smartphones : l'utilisateur n'a pas à gérer la fin d'exécution d'un programme. Ce qui était trop voyant sur PC avec Windows 8 et ses applications plein écran est aujourd'hui plus subtil avec Windows 10 mais les bases restent semblables.

Sur les OS pour smartphone l'intention est manifestement technique bien plus qu'ergonomique. En effet, la durée de vie des batteries, les simplifications nécessaires pour en faire des machines "grand public", la faible mémoire installée, la puissance limitée des processeurs, etc, tous ces éléments ont forcé les concepteurs d'OS à imaginer un mécanisme permettant de donner **l'illusion d'une fonctionnement fluide et multitâche** là où en réalité il n'y a qu'une application d'avant-plan et nécessitée de supprimer de la mémoire certaines applications sans que cela ne gêne l'utilisateur. *Ce dernier doit en effet pouvoir revenir à l'une des applications précédentes comme on passe d'une fenêtre à l'autre sur un PC* : **sans rupture** (sans avoir l'impression que l'application vient d'être relancée en ayant perdu le contenu en cours au moment de son abandon).

Sous UWP qui se voue aussi bien aux PC qu'aux unités mobiles (tablettes, smartphones mais aussi XBox, IoT et Hololens) toutes ces motivations sont présentes. Toutefois sur PC Microsoft est revenu à la raisons avec Windows 10 qui rétablit un brai bureau et le multifenêtrage. Néanmoins ce changement cosmétique ne remet pas en cause les motivations techniques d'une gestion du cycle de vie des applications bien plus contrainte que sous les versions précédentes de Windows (de Windows 1 à 7).

En compensation à ces contraintes les développeurs peuvent tirer meilleur parti des machines en concevant une expérience utilisateur fluide qui n'affecte pas la durée de vie des batteries lorsque l'application passe en arrière-plan. Nuance sous Windows 10, minimiser une application ou en activer une autre n'a pas l'air de faire basculer les autres en mode suspendu. Certainement en raison de l'énergie et de la RAM dont un PC dispose. Il s'agit là d'une meilleure prise en charge d'un PC qui n'est donc pas qu'une grosse tablette comme certains ont voulu nous le faire croire un moment (et qui heureusement ont quitté Microsoft).

En utilisant les nouveaux évènements de cycle de vie, vos applications UWP donneront la sensation qu'elles sont toujours vivantes, même si elles ne fonctionnent réellement plus lorsqu'elles passent hors de l'écran sur un mobile ou lorsqu'elles sont arrêtées sur PC.

L'utilisateur, certainement habitué au bureau classique et à ses facilités quitte souvent une application en cours d'exécution pour faire autre chose, momentanément ou non. Sur un PC il sait qu'il retrouvera la fenêtre minimisée ou laissée en arrière-plan dans l'état où il l'a laissée. Il s'attend à ce que cela se passe de la même façon sur son smartphone ou sa tablette. Avec UWP comme il n'y a plus qu'une seule application qui peut tourner dans tous ces contextes il convient donc d'être attentif et de gérer convenablement ces situations différentes…

Quant aux nouveaux utilisateurs ou ados se fichant bien des contraintes techniques, il est évident que pour eux il est “naturel” de pouvoir revenir sur qu’on a abandonné sans avoir tout perdu du travail en cours. Ces utilisateurs là ne se posent aucune question sur cette gestion du cycle de vie des applications car elle est tout simplement ce à quoi ils s’attendent puisqu’ils n’ont de contact avec l’informatique que via les smartphones ou les tablettes la plupart du temps !

C’est ainsi que UWP depuis Windows 8 généralise sur PC un mode de gestion du cycle de vie des applications qui nous vient des unités mobiles à la fois parce que UWP fonctionne aussi sur les machines de ce type et que Microsoft a eu le courage d’innover en imposant une expérience utilisateur cohérente, fluide et transparente quel que ce soit le type d’ordinateur le faisant tourner.

Ainsi, le nouveau modèle de cycle de vie met l'accent sur les applications au premier plan pour assurer à l’utilisateur une expérience dynamique et immersive en exploitant au mieux la puissance de la machine qu’il utilise, quelle qu’elle soit.

En rupture totale avec le passé, Microsoft propose un OS qui consomme moins que les versions précédentes et qui sait s’adapter aussi bien à un Arduino lent avec peu de mémoire qu’à une bombe de bureau dotée d’un processeur 8 cœurs et de plusieurs Gigas de RAM...

Le nouveau cycle de vie des applications participe pleinement à ce nouvel objectif. Il est donc essentiel de le comprendre et de le maitriser. Je serai même tenté de comparer ce changement radical à celui que nous avons vécu lorsque Windows est passé du multitâche coopératif au multitâche préemptif. Toutefois UWP a été conçu avec tout ce qu’il faut pour gérer simplement ces nouvelles contraintes là où le passage du multitâche coopératif au préemptif fut plus douloureux pour certains développeurs...

Le cycle de vie d’une application UWP

Les choses sont simples, au départ au moins : une application UWP est à tout moment dans l’un des trois états suivants :

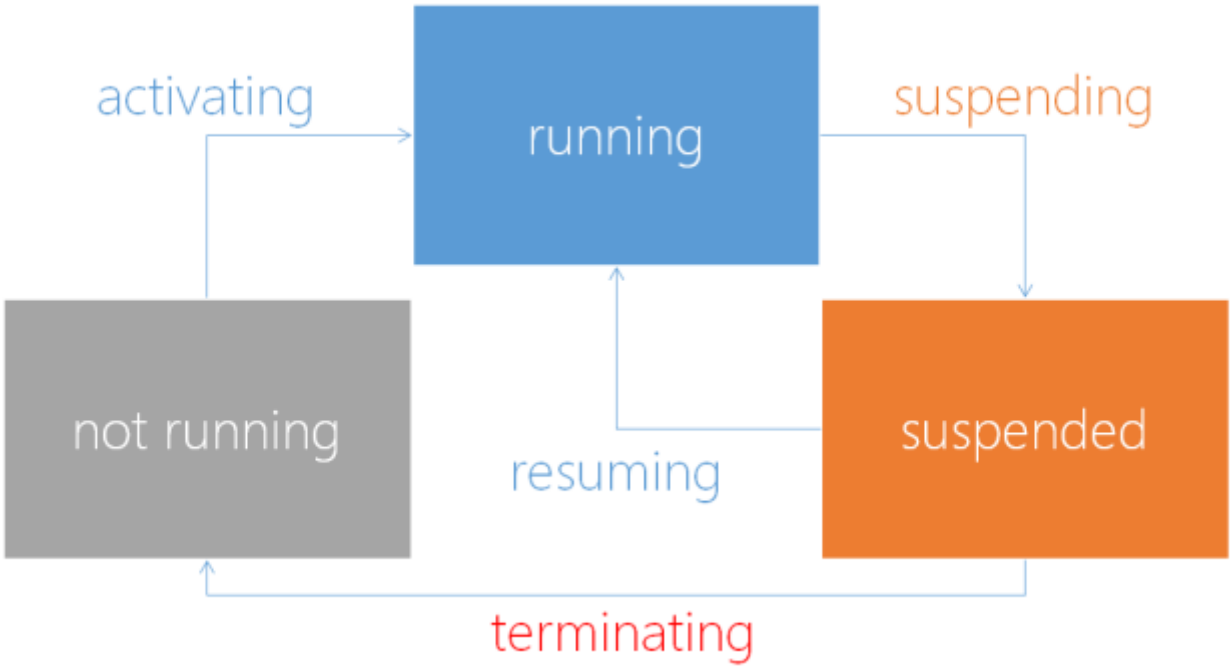
- not running (ne tournant pas)
- running (en cours d’exécution)
- suspended (suspendue)

L’état “terminé” peut être compté comme le quatrième état mais ce n’est pas vraiment un. Ca se discute et chacun pensera comme il le préfère...

Comme durant toute sa “vie” une application ne fera que passer de l’un de ces états à un autre, il existe des transitions d’état matérialisées par des évènements qu’elle peut (et doit) gérer. La mort, l’état terminé ne donnant lieu à aucun signal ni évènement. Il vient quand le grand ordonnateur l’a décidé, sans prévenir ni laisser le temps de faire autre chose. On peut d'ailleurs assimiler "not running" à l'état "terminé", dans les faits c'est la même chose, philosophiquement cela se discute mais laissons la philo de côté ☺

Ce que j’appelle la “vie” d’une application est l’espace de temps de son existence sur une machine donnée, c’est à dire entre le moment où elle installée et celui où elle est désinstallée. Entre ces deux bornes, elle ne pourra être que de l’un des trois états listés ci-dessus (plus l’état “terminé” car le grand ordonnateur de UWP est très gentil et sait faire revenir les applications du pays des morts pour qu’elles connaissent encore d’autres cycles de vie). Donc de l’intérieur une application ne peut connaitre que 3 états, mais de l’extérieur on sait qu’elle peut être dans 4 (avec l’état “terminé” donc).

Le schéma ci-dessous montre ces quatre états ainsi que les quatre évènements de transition que l’application aura à gérer (ruse : terminating n’existe pas, aucun évènement n’est envoyé, c’est pour cela que je le classe à part ainsi que son état, terminé) :



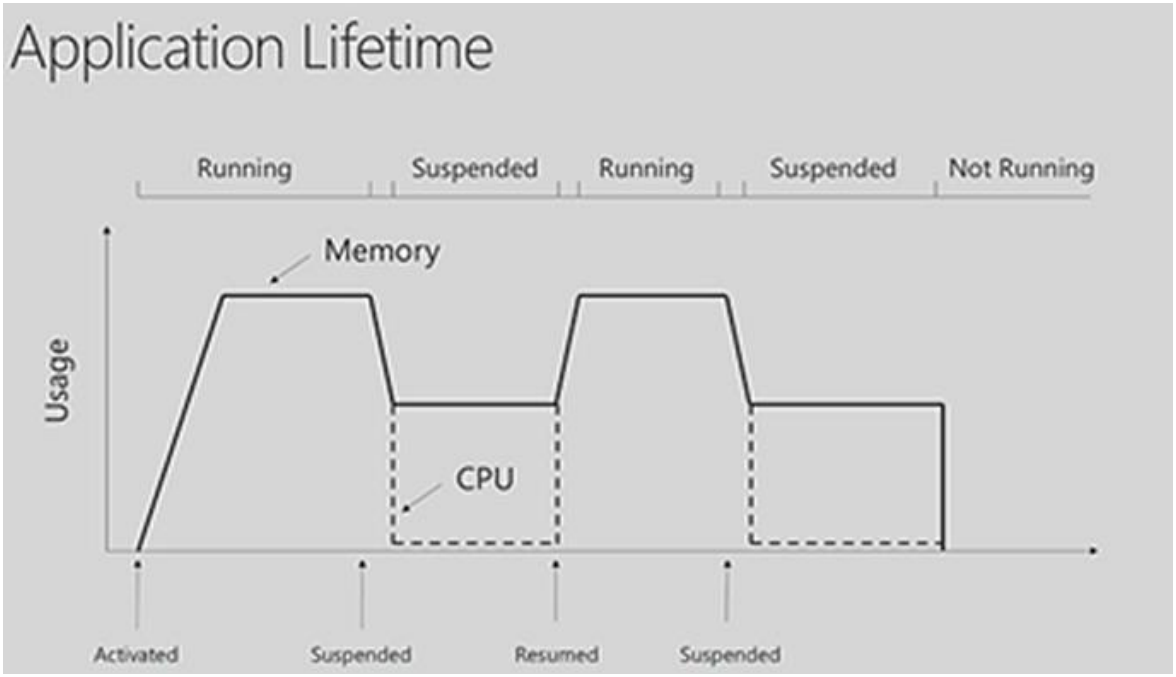
Toute application Windows 10 oscille entre ces 4 états au gré des actions de l'utilisateur qui passe d'une application à l'autre et de l'OS qui peut décider de mettre fin totalement à son activité comme de la rappeler à la vie.

Généralement une application passera son temps à basculer entre l'état "running" et "suspended" au moins sur unités mobiles. Il est donc indispensable de gérer les transitions correctement. Mais en fait l'état "suspendu" n'est pas éternel, au bout d'un moment, si l'OS a besoin de mémoire il fera passer sans autre forme de procès l'application en mode "terminé". Mais comme je le disais aucun évènement n'est déclenché à ce moment. C'est donc bien "suspendu" l'état de la dernière chance…

On peut résumer tout cela par le tableau suivant :

Raison de l'arrêt	PreviousExecutionState	Action à exécuter
Terminée par l'OS	Terminated	Restaurer la session
Fermée par l'utilisateur	ClosedByUser	Démarrage normal
Fin inopinée ou autre	NotRunning	Démarrage normal

Le graphique suivant illustre la succession des états dans le temps :



On remarque que l'état suspendu correspond à l'arrêt total de l'activité CPU de l'application alors que sa mémoire n'est pas totalement vidée. On voit aussi intervenir

en fin de graphique l'état "terminé" qui arrive après l'état suspendu et qui correspond à l'arrêt de l'activité CPU + vidage total de la mémoire.

La Suspension

En général une application UWP s'arrête de fonctionner quand l'utilisateur passe à une nouvelle application (ou retourne au menu principal). Sur PC et depuis Windows 10 le fait d'être toujours sur le bureau même dans un état minimisé laisse l'application en mode "running". C'est là une grosse différence avec Windows 8 notamment (d'un autre côté le fenêtrage avait disparait de ce dernier, ce qui lui a couté la vie d'ailleurs).

Sur mobiles Windows 10 suspend toujours une application qui n'est plus en avant-plan (évènement "suspending"). Quand l'application est suspendue son état est figé en mémoire. Elle ne peut pas tourner dans cet état qui est une sorte de "parking à application", leur mémoire subsiste mais leur moteur est arrêté. En revanche Windows 10 peut instantanément la sortir du "parking" et la réactiver lorsque l'utilisateur la fera passer à nouveau en avant-plan (évènement "resuming").

La suspension d'une application garantit notamment qu'elle ne peut plus tirer sur la batterie ce qui est essentiel sur les unités mobiles et IoT, et que l'application d'avant-plan qui arrive à sa place bénéficiera de toutes les ressources disponibles pour en assurer sa fluidité.

L'astuce est simple, mais elle donne d'excellents résultats à tel point que tous les OS mobiles fonctionnent de la sorte sans que l'un ou l'autre ne fasse preuve d'originalité à ce niveau.

Le mécanisme réel est malgré tout plus subtil. En fait quand une application perd l'avant-plan Windows 10 attend quelques secondes avant de la suspendre. Ce petit temps contribue lui aussi à assurer une meilleure fluidité de l'expérience utilisateur puisque si ce dernier revient rapidement sur l'application l'OS n'a pas eu à perdre de temps à la ranger puis à la ressortir du "parking" des applications suspendues. L'application elle-même n'ayant reçu aucun évènement particulier dans ce laps de temps n'a pas perdu de temps à enregistrer puis à relire son état.

Passé ce petit délai, Windows 10 va décider de suspendre l'application puisque l'utilisateur ne la reprend pas tout de suite. C'est à ce moment qu'elle va recevoir l'évènement suspending.

C'est un moment particulier pour l'application car elle doit utiliser cette (courte) opportunité pour sauvegarder son état sur un stockage permanent (disque, ssd, flash...). Très souvent l'application est "résumée" (réactivée) tel quel, c'est à dire depuis la mémoire de la machine et il n'y a alors rien de spécial à faire (son état n'a

pas changé puisque son image mémoire est restée intacte). Toutefois il est nécessaire de sauvegarder l'état quand l'évènement "suspending" arrive car l'application ne pas savoir à l'avance si elle va rester en mémoire ou bien si l'OS va décider à un moment donné de la supprimer totalement de la RAM pour gagner de la place. Et là, lorsque "resuming" sera reçu par l'application ce sera après avoir été rechargée depuis le stockage et il lui faudra se remettre dans l'état dans lequel l'utilisateur l'a quitté sans que celui-ci ne se doute de quoi que ce soit.

C'est une partie du mécanisme qui donne l'impression à l'utilisateur que les applications sont toujours "vivantes" peu importe combien de temps il les a laissé en arrière-plan (et même si la machine a été éteinte entre temps !).

Windows 10 laisse 5 secondes à une application pour sauvegarder son état, un peu plus (entre 5 et 10) sur les mobiles. 5 secondes après avoir notifié l'application par l'évènement "suspending" Windows 10 la "terminera" (suppression de la mémoire). Par rapport à WinRT UWP apporte un petit changement, le temps donné aux applications est un peu plus long mais dès la suspension leur priorité CPU est revue à la baisse (donnant-donnant !).

C'est là que les choses se compliquent un peu. Comme le temps imparti à la sauvegarde de l'état est limité, comme certains machines peuvent être lentes, comme il peut y avoir beaucoup de choses à sauvegarder, on comprend vite que si on veut que cette phase reste déterministe l'application doit se débrouiller pour sauvegarder son état *au fur et à mesure de son fonctionnement*, sans toutefois trop consommer en invoquant sans cesse des entrées/sorties... Jeu d'équilibriste !

Typiquement la gestion de suspending ressemble à cela :

```
public App()
{
     InitializeComponent();
     this.Suspending += new OnSuspending;
}

async protected void OnSuspending(object sender, SuspendingEventArgs args)
{
    // La suspension donne une chance à l'application de sauvegarder son état
    // Comme l'écriture disque est asynchrone nous demandons un defferal
    // qui s'assure que l'application ne sera pas terminée avant la fin
    // d'écriture sur disque.
    SuspendingDeferral deferral = args.SuspendingOperation.GetDeferral();
    // Nous utilisons SuspensionManager qui gère les données de session
    // dans un dictionnaire et le sérialise vers un fichier disque.
    SuspensionManager.SessionState["CodeClient"] = currentCustomerID;
    SuspensionManager.SessionState["Facture"] = currentInvoice;
    SuspensionManager.SessionState["Solde"] = currentCustomer.Balance;
    await SuspensionManager.SaveAsync();
```

```
    // envoi d'une notification sur la tuile (optionnel)
    Tile.SendTileUpdate(currentCustomer.Name, currentInvoice.ID, currentInvoice.Total);
    deferral.Complete();
}
```

Dans cet exemple fictif nous voyons :

- La prise en charge de l'évènement `Suspending`
- Le gestionnaire d'évènement `OnSuspending`
- L'obtention d'un "`deferral`" pour bloquer en quelque sorte le processus le temps de sauvegarder l'état de l'application (mais cela ne bloque rien, c'est juste un signal)
- L'utilisation de `SuspensionManager` qui permet de gérer des données de session
- La sauvegarde asynchrone des données (avec await)
- La mise à jour de la tuile de l'application, ce qui est optionnel
- L'indication de fin de traitement en "relâchant" le "`deferral`"
  Un "`deferral`" est un objet particulier permettant de demander à l'OS un sursis. La traduction française est d'ailleurs explicite pour une fois, puisque "deferral" se traduit par "ajournement" ou "report".
- Le `Resume` (reprise)

Quand une application est "résumée" (horrible anglicisme puisqu'on peut traduire "resume" par "reprise" mais qui permet d'appuyer sur le sens techniquement particulier qu'on donne ici au terme) elle reprend l'avant-plan dans l'état où elle était au moment où Windows l'a suspendue.

Techniquement : les données et l'état de l'application sont gardées en mémoire pendant la suspension (le fameux "parking"), ainsi quand Windows 10 la reprend (resume) elle se trouve toujours dans le même état et il n'y a pas de nécessité de restaurer les données préalablement sauvegardées.

Néanmoins, comme vous voulez que votre application donne l'impression d'être "vivante"' il sera peut-être nécessaire d'effectuer quelques rafraichissements. Par exemple si vous affichez des données provenant d'un service Web, d'une base de données distante, etc, et comme votre application peut rester longtemps en état suspendu, il sera certainement nécessaire de faire quelques requêtes pour afficher les données dans leur état actuel et non celui dans lequel elles étaient lors de la suspension...

Quand l'application sort du "parking des suspendues", elle reçoit un évènement "resuming", *c'est dans le gestionnaire de ce dernier que vous effectuerez les mises à jours nécessaires*.

Si nous reprenons l'exemple de code précédent (une gestion commerciale fictive, l'application affichant un client et une facture de ce dernier ainsi que le solde du compte client) nous comprenons que si le client ou sa facture ne risquent pas d'avoir changés il convient malgré tout :

1. de vérifier que la facture est toujours dans le même état (par exemple toujours due ou réglée),
2. que le compte client existe toujours (il peut avoir été supprimé),
3. dans tous les cas il faudra interroger le solde du compte client qui a toutes les chances d'avoir bougé (nouvelles factures émises, nouveaux règlements reçus...).

Suivant cet exemple, le code d'une reprise sera le suivant :

```
public App(){
    InitializeComponent();
    this.Resuming += new EventHandler<object>(App_Resuming);
}

async private void App_Resuming(object sender, object e)
 {
    //mise à jour des données
     currentInvoice = await IsInvoiceValid(currentInvoice.ID) ? currentInvoice : null;
     currentCustomer =
          await IsCustomerValid(currentCustomer.ID) ? currentCustomer : null;
     currentCustomer.Balance = await GetCustomerBalance(currentCustomer.ID);

    if (currentCustomer==null) customerSelection.Visible = true;
    if (currentInvoice==null && currentCustomer!=null) invoiceSelection.Visible = true;


   // notifications de la tuile
   Tile.SendTileUpdate(currentCustomer==null?"":currentCustomer.Name,
     currentInvoice==null?"":currentInvoice.ID,currentCustomer==null?
    0d:currentCustomer.Balance);

}
```

Encore une fois, cet exemple est purement fictif, ce sont les mécanismes et leurs principes qui comptent, pas l'exactitude du code.

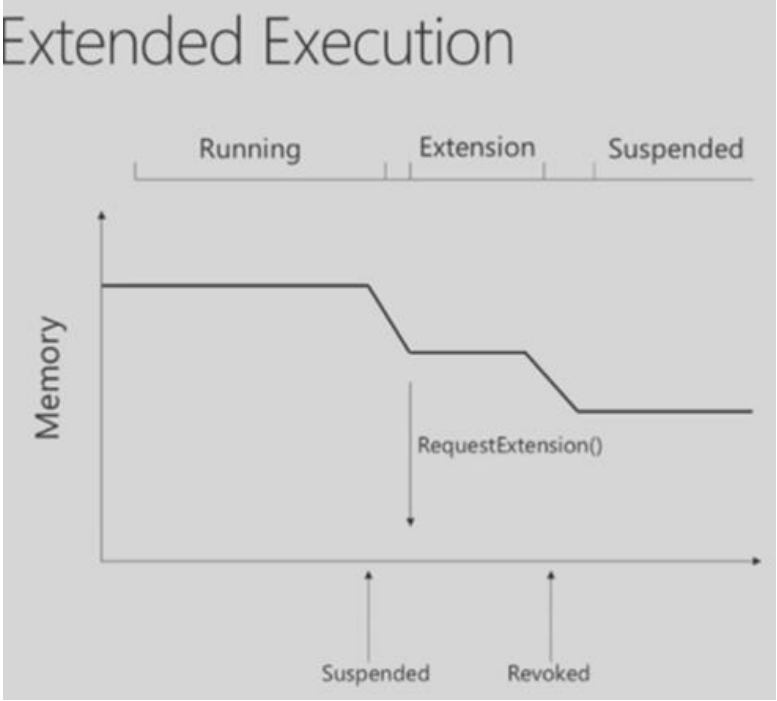
Plus de temps ?

L'utilisation du `deferral` fait partie du mécanisme normal de gestion de la suspension, sans demande d'un tel objet l'OS ne laissera aucun temps disponible pour la sauvegarde.

Mais que ce passe-t-il si dans un cas exceptionnel il y a besoin de plus de temps que 5 secondes ?

La première chose est certainement de revoir l'organisation de l'application pour faire en sorte comme je le disais déjà bien plus haut de sauvegarder en permanence des petits bouts d'information pour ne pas tout avoir à sauvegarder en une fois à la fin. Mais cela peut ne pas être applicable et même ne pas être suffisant.

Il existe dans Windows 10 un mécanisme de demande d'extension de temps qui s'ajoute à la demande de temps du deferral.



Pour cela on utilise un `ExtendedExecutionSession`, qu'on créé et manipule généralement dans un bloc using afin d'être certain de le relâcher quoi qu'il se passe.

Lorsqu'on fait une telle demande à l'OS on lui indique la raison qui peut par exemple être "sauvegarde de données, une description et un gestionnaire pour l'évènement de révocation. Puis on fait la sauvegarde et à la fin on relâche tout, extension de session et deferral.

L'Activation

Au départ les choses sont simples : l'activation de l'application n'est rien d'autre que le moment où elle lancée par Windows.

Mais nous sommes dans un environnement hautement collaboratif et intégré… De fait c'est au moment de l'activation qu'une application peut prendre connaissance du contexte de cette activation et celui-ci peut dépendre des contrats pris en charge ainsi que du mode "main view activation" ou "hosted view activation". La différence est importante puisque dans un cas l'application sera appelée "pour elle-même" alors que dans l'autre elle sera invoquée à l'intérieur d'un process dont elle n'est pas l'acteur principal (par exemple si elle est choisie comme cible ou source d'un partage de données).

Le sujet méritant de plus amples explications j'y reviendrais dans un autre billet. Pour l'instant nous resterons concentrés sur le cycle de vie, considérant l'activation sous cet unique angle de vue.

Une application peut être terminée par Windows 10 à n'importe quel moment une fois qu'elle a été suspendue. De multiples raisons peuvent entrainer l'arrêt définitif d'une application : l'utilisateur la ferme volontairement, ou se délogue, ou bien l'OS a besoin de ressources et il doit libérer de la mémoire.

Si l'utilisateur lance l'application après qu'elle ait été "terminée" elle reçoit l'évènement d'activation et Windows affiche son Splash Screen le temps qu'elle se charge. Le Splash Screen est présent dans les assets du projet créé par défaut par Visual Studio, charge bien entendu au développeur (ou plutôt au designer) de le personnaliser. Mais c'est un mécanisme désormais intégré au fonctionnement "normal" d'un logiciel après 15 ou 20 ans d'existence sous la forme d'un ajout optionnel que chacun faisait à sa manière. Windows 10 normalise ainsi de nombreuses choses, et ce côté normatif fort est à la fois un avantage certain et une contrainte à bien gérer tant pour le designer que pour le développeur. D'ailleurs la grande bataille chez les Designers notamment dans le monde Android c'est de bannir le spash screen qui aurait tendance à démontrer le manque de réactivité des applications, ce qui n'est pas faux… En tout cas Windows 10 en gère un automatiquement.

L'évènement d'activation est un point d'entrée qui permet à l'application de savoir si elle doit ou non restaurer un contexte d'utilisation et d'exécuter le code de restauration dans l'affirmative.

Les arguments de l'évènement précisent l'état précédent de l'application (que celle-ci ne peut pas deviner) notamment par la propriété PreviousExecutionState.

La valeur de cette propriété est l'une de celle prévue dans l'énumération `Windows.ApplicationModel.Activation.ApplicationExecutionState`.

C'est en partie grâce à cette valeur que l'application qui reçoit le signal d'activation peut décider si elle doit se charger avec son affichage par défaut ou bien si elle doit restaurer un état enregistré.

Le tableau ci-dessous résume les cas particuliers et les actions à entreprendre :

Raison de l'arrêt	PreviousExecutionState	Action à exécuter
Terminée par l'OS	Terminated	Restaurer la session
Fermée par l'utilisateur	ClosedByUser	Démarrage normal
Fin inopinée ou autre	NotRunning	Démarrage normal

En réalité comme on le voit, un seul cas réclame la restauration de la session de l'utilisateur : lorsque l'application a été terminée par Windows 10 après une suspension.

Si l'utilisateur a fermé l'application, ou bien si l'application a connu un arrêt inopiné (crash application ou système, ou bien si l'utilisateur n'a pas relancé l'application depuis que la session utilisateur a commencé), aucune restauration n'est nécessaire et l'application doit démarrer avec son affichage par défaut.

Il existe deux autres états possibles pour `PreviousExecutionState` : `Running` et `Suspended`. Si ces derniers peuvent être pris en compte dans des cas particuliers ils n'impliquent en revanche pas la restauration de la session puisque l'application est encore "intacte" en mémoire.

Gérer l'activation se fait comme pour les exemples précédents, seul le code de restauration est intéressant et ce dernier varie en fonction des données qui sont sauvegardées. Nous nous passerons ainsi d'un exemple fictif de code ici.

Quelques conseils

Le décor a été posé et le mécanisme décrit. Mais reste à savoir deux ou trois petits choses bien utiles pour bien gérer le cycle de vie d'une application sous Windows 10.

Démarrer sans reprise de session

Parfois il est préférable de ne pas restaurer la session utilisateur. Par exemple cela fait très longtemps que l'utilisateur n'a pas lancé votre application, "longtemps" est ici relatif et dépend de l'application et de la temporalité de son contenu, à vous de voir combien de temps cela fait exactement. Mais dans un tel cas je vous conseille d'afficher l'écran de démarrage normal et non pas les vieilles données. Cela rendra votre application plus intelligente aux yeux de l'utilisateur.

Si nous imaginons une application qui affiche des news par exemple, et si l'utilisateur n'a pas utilisé l'application depuis "longtemps", restaurer une news qui date donnera une impression négative, dans un tel cas mieux vaut ne rien afficher d'autre que l'écran de démarrage. Les exemples de ce type sont nombreux mais pas aussi évident que les news ou la météo, mais lorsqu'on écrit son application il faut y penser...

Sauvegarder les bonnes données au bon moment

Comme évoqué plus haut dans ce billet, l'évènement de suspension n'est pas un espace sans fin, Windows 10 vous donne du temps pour sauvegarder vos données mais passé un délai il "coupera le jus" à l'application et s'en sera terminé... Si l'application gère des données volumineuses ou longues à sauvegarder, il est préférable alors d'adopter une stratégie de *sauvegarde incrémentale* au fur et à mesure de l'utilisation.

Il y a deux types de données qu'une application peut manipuler : les données de session et les données utilisateurs.

Les premières sont celles qu'on sauvegarde de façon temporaire (voir les exemples de code dans le billet) pour se rappeler dans quel état est l'application (le numéro de la page d'un document qui était en cours d'affichage, le nom de ce document, le code d'une action boursière affichée, le nom de la page de l'application en cours d'affichage, etc, etc...).

Les secondes sont les données produites par l'utilisateur. Un texte en cours de rédaction, une photo, une capture vocale, un dessin... Ces données là doivent être accessible à l'utilisateur peut importe ce qu'il se passe. Elles ne sont pas sauvegardées dans les données de session mais dans l'espace dédié à l'application pour l'utilisateur en cours.

Il est donc important de bien faire la différence entre ces deux types de données et de les sauvegarder au bon moment et aux bons endroits !

Gestion des ressources externes

Si l'application a acquis des ressources externes comme un handle de fichier, une référence à un service, etc, elle doit *absolument les libérer* quand elle reçoit l'évènement Suspending. De la même façon, c'est quand elle est reprise (resume) qu'elle doit *réclamer à nouveau ces ressources*.

C'est un petit détail à ne pas oublier car il peut faire tout planter !

Ne fermez pas vos applications !

Aussi troublant que cela soit au début les applications Widows 10 ne doivent pas donner à l'utilisateur la possibilité d'être fermées. Cela était très stricte sous Windows 8 puis une petite croix de fermeture fit une apparition sous Windows 8.1 et avec le

mode fenêtré de Windows 10 cet impératif devient faux, mais sur PC uniquement ! La première fois que j'ai lancé la première bêta de Windows 8 j'ai pesté un bon moment car je ne trouvais pas la petite croix ou un procédé similaire pour fermer les applications trouvant délirant d'être obligé de faire Alt-F4 pour mettre fin à chaque appli lancée... avant de comprendre que j'avais affaire à un OS de téléphone et non plus à Windows 7  . Le bureau de Windows 10 a l'avantage de ne pas frustrer à ce point les utilisateurs d'anciennes versions ! Mais sur PC seulement. Dans tous les autres cas l'impératif reste valable : pas de fermeture d'application par l'application elle-même ni par l'utilisateur (sauf en utilisant les mécanismes offerts par l'OS).

Il faut bien comprendre, autant côté développeur qu'utilisateur, que Windows 10 gère le cycle de vie des applications lui-même et qu'il optimise la mémoire tout seul. Ni l'utilisateur ni le développeur ne doivent fermer une application dans l'espoir de gagner de la place. C'est à l'OS de se débrouiller. Le système de suspension est là pour garantir que l'UX reste excellente. Il ne faut pas tenter de lutter contre l'OS mais jouer la partie avec ce qu'il offre.

Conclusion

Avec trois évènements (Suspending, Resuming et Activated) une application Windows 10 / UWP dispose d'un mécanisme lui permettant de paraitre toujours vivante, toujours prête à fonctionner et sachant reprendre le fil de la dernière action en cours comme si elle avait toujours été en avant-plan... Ces évènements sont le reflet d'une gestion totalement nouvelle du cycle de vie des applications, en tout cas sur PC (si on omet la tentative Windows 8 qui est restée confidentielle), et garantissent, par les mécanismes sous-jacents, une utilisation fluide, dynamique et transparente des applications installées tout en préservant la puissance processeur et les ressources de la machine.

La cohérence de l'expérience utilisateur du smartphone au PC est un des avantages de Windows 10 sur l'ensemble de sa concurrence. La cohérence de son fonctionnement pour le développeur aussi.

Après un petit temps d'hésitation on finit par comprendre que ce choix est intelligent. C'est un peu la même démarche intellectuelle que celle que nous avons connu lorsque nous sommes passés de langages et d'environnement dans lesquels les destructeurs d'objet étaient essentiels à C# et son environnement managé qui prenaient en compte totalement la libération des objets. Beaucoup ont résisté, voire pesté contre cette avancée pour finir par l'adopter et la trouver totalement naturelle et normale aujourd'hui...

Il en sera pareil demain pour la gestion du cycle de vie des applications de Windows 10. On sent bien déjà que la surprise créée par Windows 8 malgré son adoption

microscopique a marqué les esprits que retrouver dans un PC un mécanisme de smartphone n'a plus l'air de surprendre grand monde. Comme quoi…

Coder

Gérer des données avec SQLite

Dès qu'une application mérite ce nom et dépasse le stade de démo elle a généralement besoin de gérer des données. UWP est une plateforme jeune et les outils et librairies tiers ne sont pas tous encore à niveau. SQLite a bonne réputation mais c'est le produit le plus mal documenté du monde… Comment s'en servir ?

Pourquoi SQLite ?

D'abord SQLite est une base de données simple possédant un petit moteur embarqué sans installation qui sied parfaitement à une majorité de développements. Toute application n'a pas besoin d'un serveur dédié avec un SQL Server dernier cri et des téraoctets de disponibles. Même en entreprise cela se rencontre fréquemment. Par habitude on préfère toujours tout centraliser sur un serveur parce que les machines faisant tourner ces logiciels sont des PC et que les serveurs sont sauvegardés. Mais ce temps est révolu, smartphones et tablettes font partie du décor. Alors on peut soit utiliser des bases dans le Cloud tel Azure, ce qui est une excellente solution, soit laisser chaque utilisateur gérer ses données qui seront simplement sauvegardées régulièrement sur un serveur. Et s'il s'agit d'applications personnelles la centralisation n'a plus aucun sens, le partage et la sauvegarde oui, mais cela se règle autrement qu'en ADO.NET avec un SQL Server derrière (les services tels que Google Drive, OneDrive, DropBox remplissent ce rôle à merveille…).

Ensuite SQLite est cross-plateforme. C'est aujourd'hui essentiel. S'il faut changer le code de gestion des données pour chaque cible visée le coût est trop important.

Enfin SQLite a très bonne réputation et est mise à jour régulièrement pour toutes les cibles. *C'est un produit vivant et c'est essentiel aussi*.

On terminera par le fait qu'elle est gratuite et simple à utiliser ce qui ne gâche rien.

Son seul point noir la documentation. Le pire étant que bien entendu le code original même un peu documenté n'est pas utilisable tel quel et qu'il faut y ajouter des librairies supplémentaires qui elles-mêmes sont encore moins bien documentées… Mais avec un peu d'acharnement et beaucoup d'essais on finit par trouver ce qui marche !

Installer l'extension UWP

Il faut d'abord se procurer l'extension UWP, le moteur écrit en mode portable. Heureusement l'équipe de SQLite a été réactive et on trouve depuis peu un setup VSIX en version finale sur le site SQLite.org, page des téléchargements :

C'est celui-là que vous devez prendre. Ou plus récent bien entendu si c'est le cas quand vous lirez cet article.

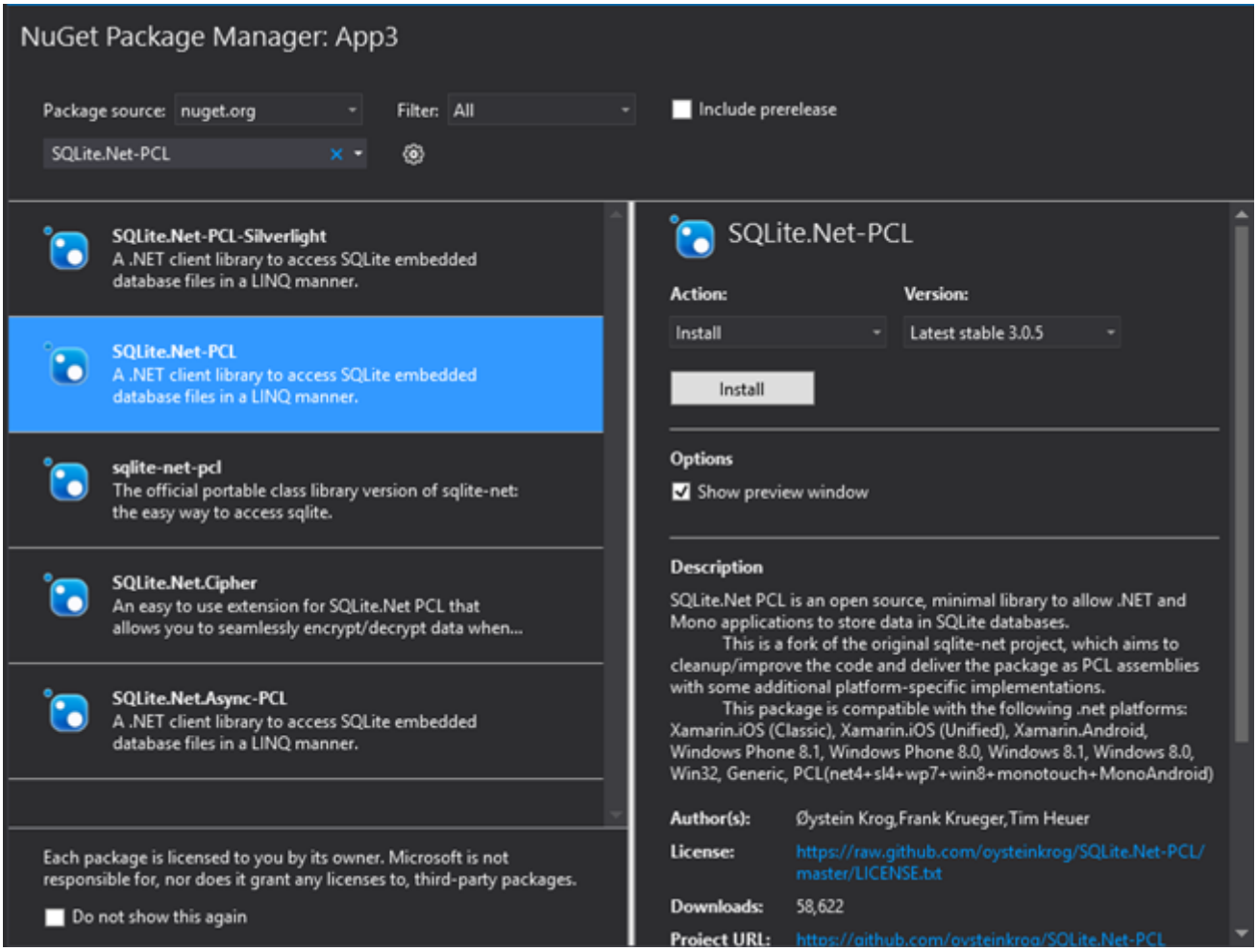
Une fois téléchargé, exécutez le setup. VS 2015 disposera désormais de l'extension UWP. Nous verrons comment l'utiliser plus loin.

C'est tout pour les préparatifs. Maintenant c'est dans chaque projet qui utilisera SQLite que le reste aura lieu.

Utiliser SQLite

Ne reste plus qu'à utilise SQLite dans un véritable projet. C'est ce que nous allons faire en créant un projet UWP "blank".

Une fois celui-ci créé, il faut ajouter un paquet Nuget. Mais pas n'importe lequel ! C'est un souk innommable dans les paquets SQLite, une vraie foire d'empoigne avec même des noms de paquets dupliqués … Et un seul fonctionnera. Saurez-vous le trouver ? Non, je rigole, je me suis assez enquiquiner je vais vous livrer la solution :



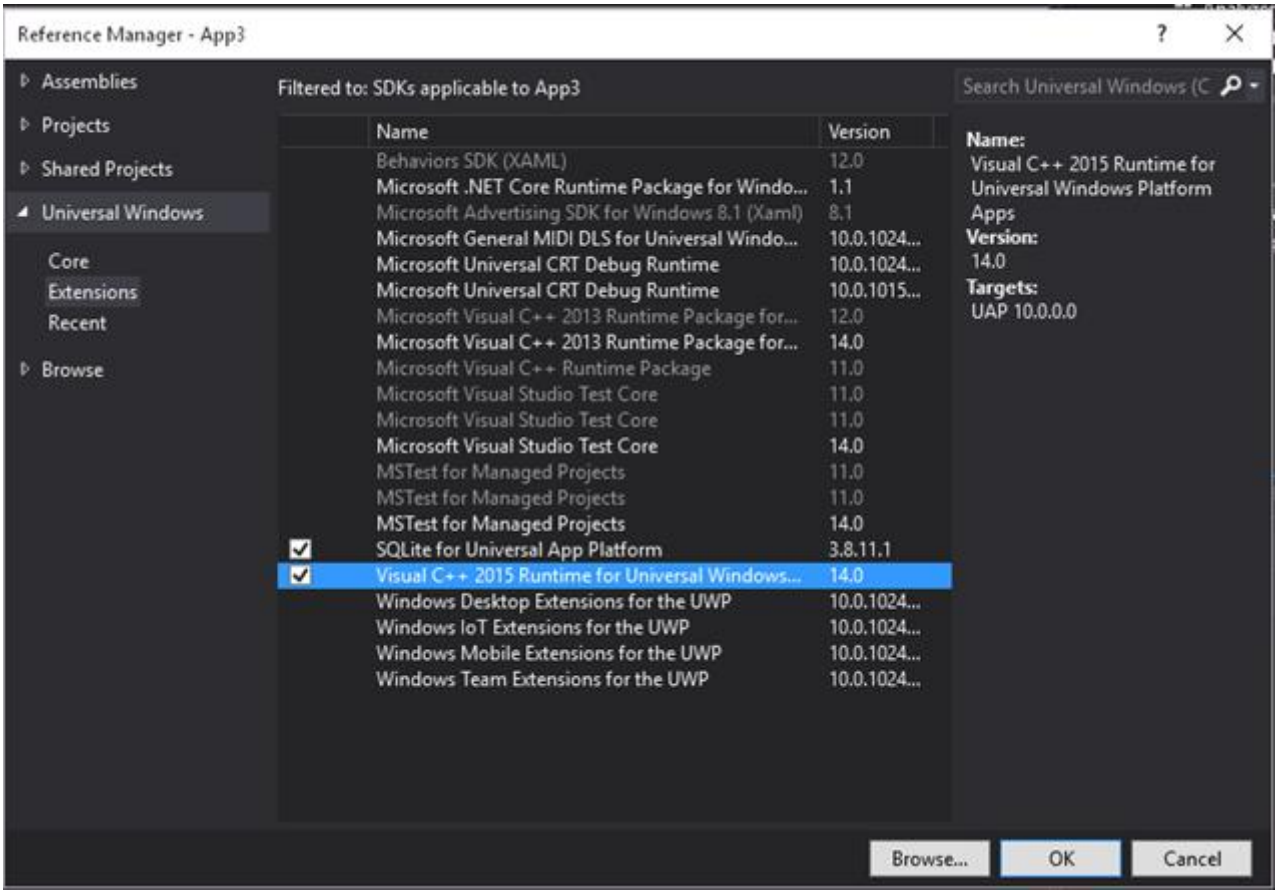
C'est celui-là. `SQLite.Net-PCL` des trois auteurs dont les noms sont inscrits à droite. Tim Heuer est bien connu des fans de feu Silverlight, gage d'une librairie bien écrite.

Cliquez sur *Install* et laisser Nuget faire son travail.

Maintenant c'est là que la fameuse extension UWP entre en scène. Toujours dans les références du projet cette fois-ci n'appelez pas le gestionnaire de paquets Nuget mais bien *l'ajout de référence*.

Dans la section (à gauche) Universal Windows, sélectionnez "*Extensions*". C'est ici que se trouve les extensions UWP qui permettent par exemple d'atteindre des fonctions spécifiques à telle ou telle autre famille de machines tout en restant portable. Dans la liste qui s'affiche au centre vous devez alors cocher l'extension "*SQLite for Universal App Platform*".

Mais ce n'est pas tout. En tout cas au moment où j'écris ces lignes il y a un avertissement du compilateur si on n'ajoute pas aussi l'extenstion "*Visual C++ 2015 Runtime for Universal Windows …*". Peut-on laisser passer le warning sans conséquence ? Dans le doute ajoutez l'extension :



Nous voici paré pour notre test !

Ca parait si simple quand je relis ces quelques explications… Si vous saviez comment il faut galérer pour trouver la combinaison gagnante !

Ne reste plus qu'à écrire un peu de code qui va nous prouver que tout marche.

Je vais tout coder dans le code-behind de `MainPage` car il n'y aura aucune fioriture. C'est le code brut de fonderie pour utiliser SQLite qui compte ici. D'abord créons une classe qui sera persistée dans SQLite :

```
public class Message
{
    public string Title { get; set; }
    public DateTime Stamp { get; set; }
}
```

Aucune astuce particulière, une classe vraiment de base. Je n'utilise pas ici certaines possibilités de SQLite comme les clés primaires, index etc. Nous verrons cela plus tard.

Maintenant il faut ouvrir une base de données dans l'espace privé de l'application, créer la table si elle n'existe pas, la remplir un peu et tenter de retrouver les données.

Tout cela sera codé “sauvagement” dans le constructeur de `MainPage` derrière le `InitializeComponents()` :

```
public MainPage()
{
    this.InitializeComponent();
    var path =
     Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path,
                  "db.sqlite");

     using (SQLite.Net.SQLiteConnection
            conn = new SQLite.Net.SQLiteConnection
            (new SQLite.Net.Platform.WinRT.SQLitePlatformWinRT(), path))
    {
        conn.CreateTable<Message>();
        conn.DeleteAll<Message>();

        for (var i = 0; i < 5; i++)
        {
            var m = new Message {Title = "Titre n° " + i,
                                 Stamp = DateTime.Now};
            conn.Insert(m);
            Task.Delay(500).Wait();
        }


        var query = from m in conn.Table<Message>()
                    orderby m.Stamp descending select m;


        var s = query.Aggregate("", (current, message) => current +
                    (message.Stamp + " " + message.Title +
                     Environment.NewLine));

        new MessageDialog(s).ShowAsync();

    }
```

La première étape consiste à créer le chemin d’accès la base de données et lui donner un nom. Le résultat est stocké dans la variable “`path`”.

Ensuite on ouvre une connexion dans un bloc “`using`” parce que c’est une bonne pratique et qu’on s’assure que le `Dispose` sera effectué dans tous les cas.

Cette connexion est créée en mode “`WinRT`”.

Ensuite le code tente de créer la table si elle n’existe pas (sans danger si elle existe). La méthode `CreateTable` utilise le type de la classe qui sera persistée. Il existe des

moyens de personnaliser les noms de table, de champs, etc. Pour le moment je me concentre sur l'efficace.

Derrière la création de la table on voir un `DeleteAll`, cela ne sert que dans ma démo pour vider la table à chaque run afin de n'avoir toujours que les 5 enregistrements de démo.

Enregistrements qui sont créés ensuite dans une boucle. Le titre du message est numéroté, le stamp utilise l'horloge classique. C'est par la méthode `Insert` de la connexion que les enregistrements sont créés.

J'ai ajouté une attente de 500 ms pour que le stamp ne soit pas le même dans chaque enregistrement sinon ça va très vite…

Nous avons créé une base de données SQLite, nous avons créé une table `Message`, nous avons créé des enregistrements… Ne reste plus qu'à se prouver que tout cela peut être récupérer dans une requête LINQ !

C'est le but de la variable "`query`" qui extrait tous les enregistrements de la table en les classant par ordre décroissant de leur stamp.

Comme j'ai dit qu'il n'y aurai pas d'UI mais qu'il faut bien afficher le résultat pour le vérifier je construis par concaténation une chaîne qui contient tous les records qui viennent d'être lus par la requête LINQ. Et j'affiche cette chaîne en utilisant un `MessageDialog` de façon très brutale là aussi (notamment je n'attends pas par `await`, je ne lis pas le résultat, etc, c'est vraiment du brutal !).

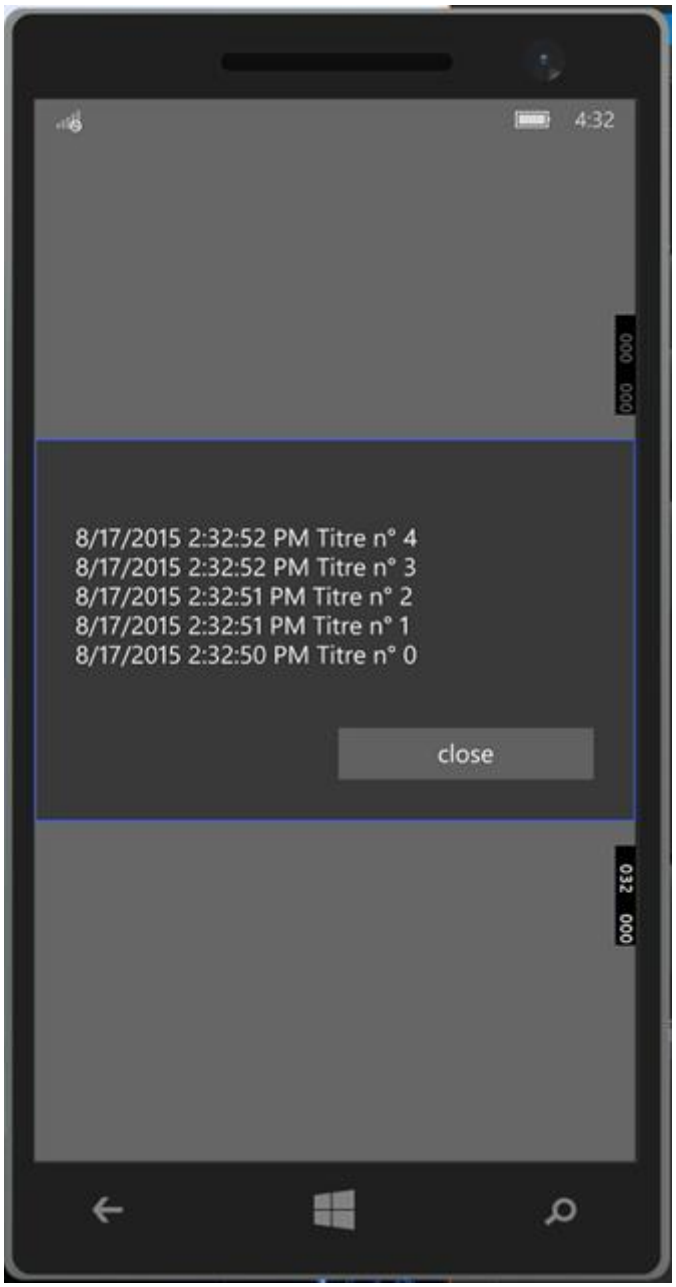
Et quand on exécute cette merveilleuse application voilà ce qu'on obtient :

Fascinant ! Aurait ajouté ce regretté M. Spock… les cinq messages sont bien listés dans l'ordre décroissant des stamps, donc dans l'ordre décroissant des numéros de titre par le fait même.

Plus ?

Oui bien sûr on peut faire plus. Même beaucoup plus. Mais là on dispose déjà du minimum vital. Créer une base, une table, la remplir et retrouver les données. C'est peu mais c'est énorme !

D'autant que normalement, juste en changeant la cible dans VS et en choisissant un Windows Phone 10 5" notre code doit fonctionner sans rien toucher… On essaye ?



C'est bluffant UWP quand même ? Non ? *Le même code, rigoureusement le même, le même exécutable, et voici une application pour PC qui tourne directement sur un smartphone* ou une Xbox…

Encore Plus ?

Maintenant que vous avez vu cela vous avez envie de développer plein de trucs, c'est trop fun. On retrouve le plaisir de Silverlight. L'envie de faire des choses. Ca faisant un moment que plus rien de neuf ne donnait cette envie chez Microsoft il faut avouer. Le cauchemar est terminé !

Pour aller plus loin il faudrait mieux connaitre les possibilités de SQLite, voir s'il est possible de gérer automatiquement des relations entre les tables, etc; Ca serait bien hein ?

Conclusion

A chaque jour suffit sa peine !

Demain je vous parlerai des SQLite extensions qui gèrent les relations entre tables et un peu plus des possibilités de base de SQLite lui-même. Gérer des données est tellement important pour la majorité des applications que cela mérite bien deux articles !

SQLite et ses extensions

Dans l'article précédent je vous expliquais comment utiliser SQLite avec UWP, il est temps d'aller un peu plus loin dans la connaissance de cette base de données et de ses possibilités…

SQLite.NET

Gérer des données c'est en fait ce à quoi sert l'informatique, science du traitement de l'information. Savoir comment on peut prendre en charge ces dernières sur une plateforme est donc essentiel. SQLite.NET est une couche qui autorise l'exploitation de bases de données portables utilisant le moteur SQLite. Légèreté, simplicité d'utilisation, fiabilité ont fait le succès de ce moteur. L'utiliser sous UWP permet d'ors

et déjà de concevoir des applications portables ambitieuses. Avec un seul code, un seul exécutable, un seul Store. Regardons de plus près ce qu'on peut en attendre…

Obtenir une connexion

Comme n'importe quelle base de données SQLite ne peut s'utiliser qu'une fois qu'on a obtenu une connexion active. Et pour l'obtenir il faut définir le chemin d'accès à la base ainsi que le nom de cette dernière.

Dans l'article précédent le chemin était construit de la sorte :

```
var path =
Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path, "db.sqlite");
```

Le stockage se fait dans `LocalFolder` de `ApplicationData`. Sur ma machine pour l'application "`App3`" ce chemin est le suivant :

```
C:\Users\Olivier\AppData\Local\Packages\8a6dbf70-b051-4496-9ab3-
e98ec06c712d_4a5vmqx2t774c\LocalState\db.sqlite
```

On voit que ce chemin se situe dans les fichiers privés de l'utilisateur. Il n'y a donc pas de partage de données entre différents utilisateurs de la même machine. Les bases de données légères et embarquées de type SQLite ne fonctionnent d'ailleurs presque jamais en mode multi-user, que cela soit simultanément ou non. Si on désire partager des données entre utilisateurs il faut opter pour un stockage dans le cloud par exemple.

Avec UWP il est possible d'utiliser des API du type

```
public static IAsyncOperation<StorageFile> GetFileFromPathAsync(string path)
```

Ce qui permet d'atteindre d'autres zones de stockage mais là encore il faut être très attentif à l'exposition des données privées et au partage des fichiers qui généralement ne peuvent être ouverts que par une application à la fois.

Une fois le path créé et le nom de la base défini il est possible d'ouvrir la connexion. Dans l'article précédent cela était fait dans une section "using" pour des raisons évidentes s'agissant de ressources non gérées (au sens *not managed* de .NET) qu'il faut veiller à libérer systématiquement. Le code était pour rappel :

```
using (SQLite.Net.SQLiteConnection conn =
  new SQLite.Net.SQLiteConnection(
        new SQLite.Net.Platform.WinRT.SQLitePlatformWinRT(), path))
           { ... }
```

La plateforme précisée est `WinRT`, il n'y a pas de choix UWP (au moins pour l'instant) car techniquement les deux mondes restent très proches et que les exigences notamment de sécurité sont fondamentalement les mêmes.

Créer des tables

La création de tables est un moment essentiel qui suit généralement l'obtention de la connexion. Cette création est non destructive et c'est pour cela qu'on la rencontre systématiquement dans la plupart des applications : on s'assure de cette façon que la table existe et si elle n'existe pas elle est créée avant toute tentative d'accès.

Dans le code exemple la table était créée de la façon suivante :

```
conn.CreateTable<Message>();
```

"`conn`" étant la connexion. La méthode `CreateTable` prend le type de la classe qui sera persistée.

La classe persistée est un POCO tout ce qu'il y a de plus classique. On peut parfaitement travailler avec ce "minimum". Toutefois nous le verrons plus loin des attributs spécifiques permettent de préciser de nombreuses options essentielles comme les clés primaires, les index, etc.

Créer et lire des données

Ce qu'on attend d'une base de données c'est de permettre les opérations CRUD. Heureusement SQLite fourni le nécessaire et de façon simple. Ainsi créer un nouvel enregistrement se fera comme suit :

```
var m = new Message {Title = "Titre " , Stamp = DateTime.Now};
conn.Insert(m);
```

En reprenant la classe `Message` de l'article précédent qui contient un titre et un date on voit comment une instance est créée de façon tout à fait classique puis comment elle est persistée par la méthode `Insert` de la connexion.

On peut ensuite accéder à toutes les données de la table :

```
conn.Table<Message>()
```

Tous les enregistrements sont retournés, ce qui peut être dangereux pour la mémoire ! En général une telle source de données est exploitée au sein d'une requête LINQ quii filtrent ce qui est remonté :

```
var query = from m in conn.Table<Message>() where m.Title="Le Titre" orderby
m.Stamp descending select m;
```

On peut bien entendu obtenir directement un enregistrement par son ID si on a eu la bonne idée d'en ajouter un dans la classe persistée…

On peut ainsi faire évoluer la classe de test `Message` en lui ajoutant une clé primaire :

```
public class Message
   {
       [PrimaryKey, AutoIncrement]
       public int Id { get; set; }
       public string Title { get; set; }
       public DateTime Stamp { get; set; }
   }
```

On commence à voir apparaitre quelques attributs spécifiques de SQLite. Ils s'obtiennent en ajoutant le `using` suivant :

```
using SQLite.Net.Attributes;
```

Dans la classe `Message` il existe désormais un champ "`Id`" de type integer qui est marqué comme étant la clé primaire. De plus nous choisissons de rendre cet entier autoincrémenté, c'est SQLite qui se chargera donc de tout. Notamment à la sortie de l'appel à la méthode `Insert` (voir plus haut) le champ "`Id`" contiendra automatiquement une valeur valide.

Grâce à cet `Id` il est possible d'obtenir maintenant un enregistrement précis :

```
var m1 = conn.Get<Message>(3);
```

"`m1`" contiendra en sortie une instance du Message ayant un `Id = 3`.

Les Attributs SQLite.NET

Comme nous l'avons entrevu plus haut il existe des attributs permettant d'agir plus finement sur le comportement de SQLite qu'avec des POCO de base.

Fixer la clé primaire est en général un préliminaire incontournable pour gérer des données efficacement, c'est le rôle l'attribut `PrimaryKey`. Mais attention, c'est SQLite, lire "light" à la fin, pas SQL Server ! Donc la clé primaire ne peut être que simple (pas de clé composée) et de type integer (d'autres types sont acceptés mais bien entendu pas en mode auto-incrémenté).

On peut décider de fixer soi-même la valeur (dans certains cas c'est plus pratique) ou bien de laisser le moteur attribuer une valeur autoincrémentée c'est le rôle de l'attribut `AutoIncrement`.

On peut ensuite agir sur les noms des colonnes qui, par défaut, sont ceux des propriétés. Avec l'attribut `Colum(name)` on peut changer le nom d'un champ et avec `Table(name)` celui de la table (qui prend le nom de la classe par défaut).

A quoi cela sert-il ? Dans le mode de fonctionnement montré ici avec SQLite.Net pas à grand chose. Mais SQLIte est une base de données qui se pilote en SQL et selon la circonstance on peut être amené à travailler directement dans ce langage. Et il peut alors être plus pratique d'avoir des noms de tables et de colonnes qu'on fixe soi-même. Sinon il n'y a pas d'intérêt.

Il est aussi possible pour les champs string de fixer une longueur maximale par l'attribut `MaxLength(int)`. La troncature n'a lieu qu'au moment d'un `Insert` ou d'un

`Update`, c'est au code de l'application de vérifier si l'utilisateur est bien limité dans ses saisies.

L'attribut `Ignore` est intéressant car il permet d'éviter le stockage d'un champ. Et pourquoi ne voudrions-nous pas stocker un champ en particulier ? D'abord s'il est d'un type que SQLite ne peut pas stocker (pas de sérialisation disponible), ou bien s'il s'agit d'une collection que SQLite ne peut pas mapper automatiquement ou encore si on a ajouté des champs calculés qui n'ont pas de raison d'être stockés.

Enfin l'attribut `Unique` permet de forcer l'unicité des valeurs d'une colonne. La clé primaire pourrait jouer ce rôle mais elle joue un autre rôle. Il s'avère souvent indispensable de fixer l'unicité par exemple d'un titre, d'un nom de société, d'une référence article… sans que ces éléments ne soient la clé primaire.

Mais il existe d'autres attributs dont le rôle peut s'avérer important, par exemple `Indexed` qui ajoute un index sur la colonne et qui accélère les recherches et les tris sur cette dernière. Autre attribut plus "sournois" mais qui peut sauver la vie dans le cas d'applications utilisant des données de plusieurs langues ou des champs avec une casse variable, "`Collation()`". Cet attribut accepte les valeurs `BINARY`, `RTRIM` et `NOCASE`. Le mode binaire est celui par défaut et toute recherche d'une chaine (par simple égalité) ne fonctionne que si les deux chaines sont rigoureusement identiques. Avec `RTRIM` les espaces de fin sont supprimés dans les comparaisons ce qui élargit le champ de l'égalité. Et avec `NOCASE` les différences de casse sont ignorées ce qui est primordial dans une application professionnelle mais qui peut se régler en amont ou bien dans les requêtes en utilisant des comparaisons ignorant la casse plutôt qu'un simple "égal". Toutefois cette dernière possibilité, simple et efficace, est à préférer si les attributs sont bien positionnés.

L'attribut `Default` permet de fixer une valeur par défaut mais avec certaines restrictions et un fonctionnement qui me semble étrange, donc jusqu'à plus ample informé je n'en conseille pas l'utilisation. Si des valeurs par défaut doivent être fixées utilisez des champs initialisés ou le constructeur de la classe.

`NotNull` est un attribut commun pour les SGBD et il permet bien entendu d'indiquer qu'un champ ne peut pas avoir de valeur nulle. Il faut alors gérer le rejet lors d'un `Insert` ou d'un `Update`. Personnellement avec ce genre de petite base de données je préfère lui en demander le minimum… Il doit y avoir des classes métiers qui gèrent les valeurs par défaut, l'interdiction des null, les longueurs maximales etc. Créer une exception ou un rejet au niveau de la base de données est trop "bas niveau" pour

l'utilisateur. Mieux vaut l'empêcher de saisir des bêtises plutôt que de le laisser faire et déléguer la remontée des erreurs à la couche d'accès aux données… D'ailleurs ce principe est vrai même si on utilise SQL Server avec 1000 utilisateurs utilisant de gros PC. Sauf que dans ce dernier cas la base pouvant être utilisée par plusieurs application il convient de la protéger par un schéma rigoureux, c'est donc d'autres raisons qui poussent alors à fixer ces paramètres au niveau SGBD. Avec SQLite je vous l'ai dit, point de partage entre applications ni d'accès multi-user, donc cet impératif n'existe pas.

CRUD

Je vous ai déjà présenté plusieurs opérations de base comme le `Get`, `l'Insert` ou `Table` qui permet de remonter tous les enregistrements ou de participer à une requête LINQ comme source de données. Il manque bien entendu la mise à jour qui se fait ainsi en utilisant la méthode Update de la connexion :

```
m1.Title = "MODIFIE";

conn.Update(m1);
```

Et bien entendu la suppression qui s'effectue de la même façon en utilisant `Delete`. `DeleteAll` permet de vider une table d'un seul coup tout comme `DropTable` permet de la supprimer définitivement de la base. Cela est très pratique pour les applications qui utilisent des tables temporaires pour le besoin d'une session de travail. On ouvre la connexion, on détruit la table et on la créée juste derrière. On est certain de disposer d'un espace de travail "propre". Attention à la gestion du tombstoning et des suspensions…

Les requêtes

On a vu que les requêtes pouvait s'écrire en utilisant LINQ ce qui permet de rester en C# de bout en bout et de bénéficier d'un contrôle syntaxique à la compilation et même lors de l'écriture du code grâce à IntelliSense et à des outils comme Resharper. Mais il est possible d'écrire aussi des requêtes SQL

C'est à cela que servent `Query<T>` et `Execute`, le premier retournant des résultats l'autre non.

```
var r = conn.Query<Message>("SELECT TITLE FROM Message WHERE Field1>300");
```

L'exemple ci-dessus retournera une liste d'objets `Message` dont le champ `Field1` est supérieur à `300`. Mais attention… Comme nous avons un `SELECT TITLE` seule cette propriété sera remontée dans les instances de `Message`, même l'`ID` restera à zéro… C'est en revanche plus efficace et lorsqu'on manipule des objets pouvant être lourds et / ou nombreux ce type de requête est bien plus rapide et économe en mémoire que son équivalent LINQ.

`Execute` s'utilise de la même façon mais sans retour de données, ce qui est donc à préférer pour des insertions ou des updates par exemple. Mais dans ce cas l'utilisation des méthodes montrées plus haut sont parfaitement adaptées et moins risquées que du SQL non contrôlé à la compilation. Néanmoins on peut avoir à mettre à jour de très nombreux enregistrements sur un champ (changer une quantité, une date…) et l'utilisation de `Execute` sera bien plus rapide et économe que la manipulation un par un d'objet via la connexion.

Requêter directement en SQL peut ainsi s'avérer crucial pour garantir de bonnes performances et une utilisation mémoire moindre.

Les Transactions

Malgré sa simplicité SQLite "a tout d'une grande" et l'un des points emblématiques qui différencient un SGBD d'une simple gestion de fichiers est certainement le support des transactions.

On retrouve donc naturellement un `BeginTransaction`, un `RollBack` et un `Commit` qui opèrent tel qu'on s'y attend.

SQLite va même plus loin puisqu'il est possible de poser des point de sauvegarde dans une transaction et d'effectuer un `Rollback` jusqu'à l'un de ces points. Il s'agit là de subtilités peu utilisées car les cas d'utilisation où cela serait justifié manquent un peu… Mais allez savoir, un jour cela pourra vous être utile !

Les méthodes moins connues

SQLite s'avère très complet et offre de nombreuses autres possibilités au développeur. Par exemple `InsertAll(IEnumerable, bool runInTransaction)` peut s'avérer particulièrement efficace pour faire des "bulk insert" le tout protégé ou non

par une transaction automatique. Le même type de méthode existe pour les updates aussi.

On trouve aussi dans le source de SQLite.Net une méthode `CreateDatabaseBackup` qui serait fort utile, mais je n'ai pas réussi à obtenir autre chose qu'un blocage infini sans exception… Le fichier est bien créé mais vide et c'est tout… Il faudra certainement attendre des mises à jour pour voir ces fonctions moins utilisées fonctionner correctement. Ce sera un plus car sauvegarder une base est généralement une bonne idée ! On notera que cette méthode bien que publique n'est pas décorée par l'attribut PublicAPI est-ce à dire qu'elle n'est pas encore prête… Il faudra attendre donc !

Plus utile il existe aussi des choses comme `Find<T>(Expression<Func<T, bool>>)` qui autorise la recherche d'un enregistrement via une expression lambda.

D'autres méthodes peuvent être découvertes dans le source de SQLite.NET mais ce qu'on attend d'une telle "mini base de données" c'est avant tout de pouvoir facilement insérer et supprimer un nombre restreint d'enregistrements et cela est largement couvert par tout ce que nous avons vu jusqu'ici.

Les SQLite-NET Extensions

Je fais partie de ceux qui pensent que pour une petite application les possibilités de SQLite.NET sont bien suffisantes dans la grande majorité des cas. Etant donnée la portabilité de UWP vers les smartphones notamment il ne me semble guère raisonnable de se lancer dans de la gestion lourde de data.

UWP n'est pas encore prêt à remplacer totalement WPF et les vraies applications de bureau classiques sur de nombreux points. Mais autant l'écart était net avec WinRT autant il est déjà plus mince avec UWP. Par exemple **Entity Framework 7** qui sera bientôt disponible (et en Open Source) supportera UWP ce qui signifie qu'on pourra bénéficier d'un puissant système de gestion de données capable de dialoguer avec les principales bases de données (à condition que des *connecteurs* soient disponibles).

En dehors de SQLite il faut malgré tout noter qu'il existe un connecteur pour MySQL qui autorise le dialogue en TCP avec un serveur. C'est là une solution intéressante et unique pour l'instant permettant de connecter une App universelle à un serveur. Mais si cela peut fonctionner sur un PC on se doute que ce n'est pas vraiment utilisable sur

smartphone ce qui fait perdre l'intérêt de la portabilité UWP. Du coup autant faire du WPF avec SQL Server ou utiliser Azure, c'est étudié pour …

Bref on l'a compris gérer des données en UWP c'est avoir conscience que cela doit fonctionner sur smartphone donc qu'il faut de la "retenue" dans le brassage des informations. Pas de big data avec UWP ! Mais parfois des applications n'ayant pas à traiter beaucoup de données en volume n'en restent pas moins sophistiquées.

Sophistiquées ? J'entends par là que bien que manipulant peu de données (par rapport à une grosse application d'entreprise) l'application peut avoir des besoins plus subtiles que l'artillerie de base CRUD. Par exemple la gestion des relations entre les tables, les un-vers-un, un-vers-plusieurs etc.

Dans un tel cas tout gérer "à la main" à partir de simple opérations de base peut rendre l'écriture de l'application assez longue et donc onéreuse.

Pour ces cas extrêmes il existe les **SQLite-NET Extensions**.

Par l'ajout de nouveaux attributs pour décorer les classes cette librairie rend automatique la gestion des relations entre instances. Ono-to-one, one-to-many, many-to-one, many-to-many sont prises en charge automatiquement ce qui forcément rend bien des services même dans des cas pas si compliqués que ça.

Voici un exemple de définition SQLite.NET classique comme nous en avons vus dans cet article :

```
public class Stock
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    [MaxLength(8)]
    public string Symbol { get; set; }
}

public class Valuation
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    [Indexed]
    public int StockId { get; set; }
    public DateTime Time { get; set; }
    public decimal Price { get; set; }
}
```

Une gestion de titres boursiers, avec une action et son historique de valeurs. Ici c'est le développeur qui gèrera la relation par des requêtes utilisant les ID.

Et voici sa version décorée par SQLite-NET Extensions :

```
public class Stock
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    [MaxLength(8)]
    public string Symbol { get; set; }


    // One to many relationship with Valuation

    [OneToMany(CascadeOperations = CascadeOperation.All)]
    public List<Valuation> Valuations { get; set; }
}

public class Valuation
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }


   [ForeignKey(typeof(Stock))]     // Specify the foreign key
    public int StockId { get; set; }
    public DateTime Time { get; set; }
    public decimal Price { get; set; }

    [ManyToOne]      // Many to one relationship with Stock
    public Stock Stock { get; set; }
}
```

On note la présence d'un attribut définissant une relation un-vers-plusieurs sur la propriété `Valuations` de l'action (`Stock`). De la même façon on voit un autre attribut dans la classe `Valuation` décorant la propriété `Stock` en Plusieurs-vers-un, l'inverse donc.

Grâce à ses définitions supplémentaires l'extension est capable de remonter des grappes de données soulageant le développeur de la gestion relationnelle de ces dernières. On peut interroger les données de façon habituelles pour n'obtenir que les entités de niveau supérieur ou bien utiliser de nouvelles méthodes qui étendent l'objet connexion de SQLite telle que `GetWithChildren` qui agit comme un `Get` mais en remplissant automatiquement les champs impliquant une relation. Dans l'exemple

de code ci-dessus cela revient à obtenir une action avec tout son historique de valeurs déjà chargé dans la propriété Valuations.

Il existe un paquet Nuget appelé SQLite.Net Extensions-PCL en version 1.3.0 au moment où j'écris ces lignes : https://www.nuget.org/packages/SQLiteNetExtensions/

Vous trouverez aussi plus de précision sur ce site qui présente les différentes facettes des extensions.

Conclusion

La boite à outils s'étoffe !

Après vous avoir présenté la plateforme UWP, la façon dont on designe les applications et comment on utilise les spécificités de XAML ou les fonctionnalités d'une famille de machines, je viens de vous donner les moyens de gérer des données même relationnelles !

Le tout avec un seul code, un seul exécutable pour toutes les machines utilisant UWP… Une même application pour un PC ou un smartphone, sans aller chercher les Hololens ou la XBox, c'est déjà un avantage énorme !

Le tout en conservant tout ce qu'on sait sur C# et XAML comme au bon vieux temps de Silverlight, n'est-ce pas magnifique ? !

Si, bien sur. Il faut juste maintenant que les DSI comprennent que leur intérêt se trouve dans UWP bien plus que dans du couteux BYOD… On a déjà dépassé les 80 millions de Windows 10 déployés et tous les smartphones Windows 8 feront tourner Windows 10 d'ici peu. La France est toujours pionnières avec 10% du parc en Windows 10 un mois après sont lancement ! La base d'utilisateurs existe déjà. Il est temps de mettre les logiciels en chantier !

Auto INPC pour UWP/UAP : Librairie gratuite Dot.Blog sur CodePlex !

Des aides comme Mvvm Light sont précieuses pour mettre en place des logiciels bien conçus suivant le pattern MVVM mais il y a toujours de la place à amélioration. Le cas de l'INPC pour les champs calculés en fait partie. Grâce à **AutoInpc** Dot.Blog vous offre une extension qui règle la question avec un simple Attribut…

INPC

INPC c'est le petit nom d'un évènement parmi tant d'autres sous .NET mais si essentiel au pattern MVVM car c'est lui qui permet au Binding XAML d'être averti d'un changement de valeur et de mettre à jour son affichage.
`INotifyPropertyChanged` n'est d'ailleurs pas un évènement mais une interface qui définit le support d'un tel évènement mais ce n'est qu'un détail rendant son implémentation et son utilisation plus souple.

Il est donc ici question d'INPC.

AutoInpc pour Mvvm Light et UWP – comment ça marche

Hier j'ai releasé la librairie AutoInpc sur CodePlex et en ai présenté l'utilité, j'avais promis d'expliquer comment ça marche. Voici donc les explications sur le code (Open Source) de cette petite extension bien pratique…

AutoInpc – Rappel

Comme expliqué hier, `AutoInpc` est une petite librairie qui vient étendre les possibilités de **Mvvm Light**, principalement dans les ViewModels mais aussi dans les Models. Elle permet en effet de façon purement déclarative *via un attribut* de gérer automatiquement l'envoi de l'évènement `PropertyChanged` par les champs calculés lorsque les champs standards dont ils dépendent sont modifiés.

Sans cette extension il faut ajouter en fin de Setter de chaque champ "maitre" un appel à `RaisePropertyChanged` pour chaque champ calculé qui en dépend.

D'une part c'est une programmation brouillon de type code spaghetti difficile à maintenir, et d'autre part cela éclate la responsabilité d'INPC à tous les champs maitres ce qui n'aide pas à clarifier le code. Pire, au bout d'un moment ce code de notification pollue tellement les Setters des champs maitres qu'on n'y comprend plus rien et que la moindre maintenance évolutive ou corrective risque d'oublier un `RaisePropertyChanged`, cause de bogues pas toujours évidents à trouver.

Bref, on place un attribut sur un champ calculé et on déclare les noms des champs maitres et c'est tout. Un peu magique.

Pour une présentation plus complète lisez l'article d'hier, il vous dira tout ce qu'il faut savoir pour utiliser AutoInpc.

Aujourd'hui c'est le "comment ça marche ?" qui nous intéresse, connaissance inutile (ou presque) pour utiliser AutoInpc mais indispensable pour qui aime coder !

Deux fichiers de code

`AutoInpc` est vraiment une petite librairie, elle ne viendra pas gonfler votre application de mégaoctets inutiles. Elle se réduit à deux fichiers de code :

- La déclaration de l'attribut
- La classe `AutoInpc` elle-même

L'attribut ComputedField

C'est lui qui sert au marquage des champs calculés. On l'utilise de la façon suivante :

```
[ComputedField(new []{"FirstName","LastName"})]

public string FullName => FirstName + " " + LastName;
```

Ici le champ FullName est décoré par l'attribut ComputedField qui déclare que le champ calculé dépend des deux champs FirstName et LastName. Lorsque l'un de ces deux champs sera modifié un évènement INPC sera lancé
via RaisePropertyChanged pour le champ calculé. De ce fait si le champ est affiché par un objet XAML, ce qui est généralement le cas des propriétés d'un ViewModel, son affichage sera bien mis à jour.

Comment est déclaré l'attribut ?

Voici le code :

```
using System;




namespace DotBlog.MvvmTools

{

    /// <summary>

    /// Defines a special attribute for computed field in an ObservableObject (Mvvm Light)

    /// </summary>

    [AttributeUsage(AttributeTargets.Property,AllowMultiple = false, Inherited = false)]
```

```
    public class ComputedFieldAttribute : Attribute

    {

        public ComputedFieldAttribute(string[] triggerFields)

        {

            TriggerFields = triggerFields;

        }



        /// <summary>

        /// List of all fields that must trigger an INPC for the given property using this
attribute.

        /// Names are unfortunately given as string so take care when renamming
properties...

        /// </summary>

        public string[] TriggerFields { get; set; }

    }

}
```

La classe `ComputedFieldAttribute` descend directement de `Attribute` fournie par `System`.

Elle est elle-même décorée par un `AttributeUsage` qui fixe les possibilités d'utilisation de l'attribut. Ici on précise que notre attribut ne sera utilisable que sur des propriétés, qu'il ne sera pas possible de cumuler plusieurs fois l'attribut sur la même propriété et qu'il ne concerne pas les champs hérités.

En dehors de ces déclarations le code de l'attribut déclare une unique propriété de type tableau de `string`, `TriggerFields`. Cette liste contiendra les noms des champs maitres. On retrouve l'initialisation de cette propriété dans le constructeur de l'attribut ce qui explique la façon dont il est appelé dans l'exemple de code précédent.

Rien de spécial à dire sur ce code donc, que du super simple. C'est ailleurs que tout se passe mais c'est l'attribut qui du point de vue du développeur est visible et fait tout le travail…

AutoInpc

C'est ici que tout le travail est effectué. Mais avant de regarder le code je vais essayer d'expliquer l'idée que ce code concrétise.

Le concept

L'idée de AutoInpc était de rendre le plus simple possible ce problème de gestion des champs calculés. J'ai toujours trouvé la façon classique de le gérer lourde et ouverte à plein d'erreurs ou oublis. Mais comment rendre cela simple ?

J'ai eu d'autres idées mais celle de l'attribut s'est vite imposée parce que c'est ce qu'il y a de plus simple pour le développeur et que j'aime surtout *la lisibilité de l'intention* que cette écriture permet. De plus cela *centralise* sur le champ calculé *la responsabilité* de déclarer de quels autres champs il dépend et j'aime que les responsabilités soient clairement identifiables dans ou près de l'objet concerné. S'il faut aller lire à l'autre bout d'un code pour en comprendre un morceau c'est du code spaghetti.

Reste un défaut à l'utilisation de l'attribut, les champs maitres passés le sont sous la forme de string. Je n'aime pas ça. On a tous galéré et Laurent le premier dans Mvvm Light pour tenter de supprimer les noms de propriété en string. Refaire la même chose me gêne. Mais alors il faudrait abandonner l'idée de l'attribut et cela ouvrait la voie à une programmation de type appel de méthodes dans le constructeur par exemple. Encore une fois la responsabilité s'éloignait de l'objet intéressé. Par exemple si un champ `FullName` constitué du prénom et du nom venait à devoir supporter aussi le titre ou autre (madame, monsieur…) le développeur devrait ne pas oublier de revenir généralement tout en haut de son ViewModel dans le constructeur pour penser à ajouter la dépendance à ce nouveau champ. Avec l'attribut c'est là, sous son nez, difficile malgré tout de l'oublier. C'est ce qui a pesé dans la balance. Comme tout choix on peut en discuter des heures, mais ce fut le mien…

Donc pour l'attribut c'était acheté, j'étais convaincu.

Mais comment s'en servir…

Comme il n'y a pas de magie hélas, il fallait un bout de code pour faire marcher l'ensemble donc une classe. Et comme il peut y avoir plusieurs ViewModels ou Models dans une application, chacun devrait posséder sa propre instance. Classe, instance, donc instanciation. Et comme il faudrait analyser l'objet dont le fonctionnement allait être étendu (le ViewModel, le Model…) il serait intelligent de

tout faire en une fois. D'où la présence d'une méthode statique `Initialize(…)` qui créée une instance de AutoInpc et qui conserve la référence vers le ViewModel (Model…) tout en retournant l'instance toute fraiche de `AutoInpc` qui pourra être conservée. Le constructeur de AutoInpc est marqué private pour en interdire l'utilisation mais attention cette construction n'en fait pas pour autant un Singleton (puisque chaque appel à Initialize créée une nouvelle instance).

Le code d'initialisation de `AutoInpc` devait être l'endroit où tout le travail est préparé : recenser les propriétés qui portent l'attribut, construire une liste des champs déclencheurs, etc, et surtout se brancher sur le `PropertyChanged` de l'objet principal pour déclencher celui des champs calculés !

Tout cela s'est avéré facile à faire.

Là où les choses se sont corsées c'est lorsqu'il a fallu déclencher l'INPC des champs calculés.

En effet, ce déclenchement ne peut pas se faire en implémentant INPC dans AutoInpc, toute instance de cet utilitaire est totalement différente de celle du ViewModel (Model…) qu'il "décore". Envoyer un signal INPC depuis l'utilitaire ne sera pas pris en compte par les bindings XAML sur l'objet principal, c'est une évidence.

Il faut donc "pirater" le `PropertyChanged` de l'objet principal pour s'en servir dans AutoInpc et *simuler l'appel* pour que tout objet abonné à l'INPC de l'objet principal ne puisse pas voir la différence avec un `RaisePropertyChanged` émis depuis ce dernier. Donc il faut notamment penser à simuler le `sender`. Mais ce n'était pas bien compliqué.

La vraie difficulté vient dans le fait que l'objet principal doit nous offrir un accès à `RaisePropertyChanged`.

Or le support se trouve dans la classe `ObservableObject` (qui sert de classe mère à `ViewModelBase` mais aussi à tout objet devant supporter INPC quand on se sert de Mvvm Light). De fait AutoInpc prévu pour fonctionner au départ sur les ViewModels fonctionne en réalité sur tout `ObservableObject` ce qui en étend encore plus l'utilité. C'est super. Mais il y a un os…

RaisePropertyChanged dans ObservableObject existe en 4 surcharges, et toutes sont protected.

L'histoire se termine ici puisque "la" méthode dont nous avons besoin pour faire marcher notre joli montage est à jamais enfouie et protégée par le langage et l'encapsulation. Bricoler le source de Mvvm Light est possible, le code est librement accessible. Mais fabriquer sa propre version divergente d'une telle librairie est vraiment une mauvaise idée. Déjà pour soi-même mais l'imposer en quelque sorte à ceux qui utiliseraient AutoInpc n'était tout simplement pas concevable.

Alors fin de l'histoire pour de vraie ?

Si tel était le cas vous ne seriez pas en train de lire le second article sur AutoInpc vous vous en doutez !

Dans l'attente d'un passage en public de `RaisePropertyChanged`, ce que Laurent ne fera vraisemblablement jamais (quoi qu'après en avoir discuté avec lui il pourrait le faire dans une prochaine version…), ou l'intégration de AutoInpc à Mvvm Light (ou d'un procédé équivalent) il faut ruser. *La ruse sous .NET à souvent pour petit nom "réflexion"…*

Dans l'initialisation de `AutoInpc` il y a donc une séquence qui pourrait être évitée mais qui pour l'instant est obligatoire et qui scanne la classe `ObservableObject` pour obtenir l'objet `MethodInfo` de la surcharge de `RaisePropertyChanged` qui nous intéresse. Dans cette version de `AutoInpc` je me suis arrêté à la version simple, l'envoi de INPC seul. Le système de broadcast des messages Mvvm Light qui existe dans certaines surcharges n'est pas pris en compte.

Une fois les informations de la méthode en la possession de `AutoInpc` l'affaire est dans le sac…

On dispose de l'instance de `l'ObservabelObject` principal et de la méthode à invoquer. Quand l'un des champs maitres change (généralement via la méthode `Set()` dans le Setter) `RaisePropertyChanged` est appelé et `AutoInpc` est au courant puisqu'il s'est lui-même abonné à `PropertyChanged` de cet objet… A ce moment il est facile de vérifier si le changement a lieu sur l'un des champs "déclencheurs" puis de répercuter l'évènement sur tous les champs calculés ayant déclaré dépendre de celui-ci. L'appel se fait via la réflexion en invoquant la méthode `RaisePropertyChanged(string)` dont nous avons conservé le `MethodInfo` en utilisant le nom du champ calculé intéressé et dans le `sender` l'instance de `l'ObservableObject` que l'évènement nous passe !

Se passer de la réflexion serait un must mais ici elle ne pénalise pas trop le fonctionnement puisque le `Methodinfo` est obtenu dans l'initialisation de `AutoInpc`, *une seule fois*. C'est ensuite cet objet qui est utilisé directement pour invoquer la méthode ce qui est très rapide. Il n'y a donc pas d'analyse via la réflexion à chaque envoi d'un signal INPC ce qui là serait un peu pénalisant (En réalité c'est un débat d'experts mais pour la majorité des applications cela n'a aucun impact visible).

Arriver à ce stade le projet était bouclé, au moins dans ma tête, mais restait une fonctionnalité que je voulais ajouter : *le mode suspendu*. Toutefois sa réalisation ne posait aucun problème spécifique. Via une simple propriété `IsSuspended` il serait possible de mettre en suspens les notifications sur les champs calculés. Pendant la suspension ces notifications seraient enregistrées en supprimant les doublons. En fin de suspension tous les évènements conservés seraient alors "rejouer". Cela est utile lorsque plusieurs champs déclencheurs sont manipulés par code par exemple. Si je change `FirstName` et `LastName` (suite à une lecture base de données par exemple), inutile d'envoyer deux fois INPC pour `FullName`. Une seule fois à la fin est suffisant… Les performances sont optimisées puisque les bindings ne sont pas sollicités inutilement et en surnombre. Cette option est un peu la cerise sur le gâteau il faut avouer mais elle a son utilité.

Donc voilà comment `AutoInpc` fonctionne et qu'elles ont été les étapes de sa construction intellectuelle avant de devenir du code sous VS 2015.

Pour tout vous avouer j'ai "codé" AutoInpc il a quelques jours alors que j'avais une insomnie et que le coq du voisin chantait trop fort et trop tôt. Dans ces cas là plutôt que de perdre mon temps à rêvasser, ce n'est pas productif, ou pire à m'énerver, ce qui éloigne à tout jamais le sommeil, et aussi étrange que cela pourrait vous paraitre, oui je l'avoue j'aime programmer de tête… Et cette nuit là j'ai programmé AutoInpc  En quelque sorte AutoInpc est le fruit d'un rêve à demi réveillé. C'est joli et poétique je trouve…

Le code

Il est disponible sur codeplex, sur autoinpc.codeplex.com. Mais puisque j'ai présenté le concept et le cheminement pour concevoir le code, il serait dommage de ne pas le commenter rapidement. Je ne passerai que sur les parties les plus intéressantes vous laissant tout de même le soin de regarder le code complet tranquillement chez vous après l'avoir téléchargé.

Initialize

```
public static AutoInpc Initialize(ObservableObject vm)
        {
            if (vm == null) return null;
            var auto = new AutoInpc();
            auto.checkAttributes(vm);
            auto.getMethodInfo();
            auto.hackInpc(vm);
            return auto;
        }
```

Comme on le voit `AutoInpc` fonctionne sur des `ObservableObject`, c'est à dire la classe mère de `ViewModelBase` dans **Mvvm Light** mais aussi celle de classes du Model car très souvent ces dernières doivent implémenter INPC et `ObservableObject` le fait très bien.

La méthode est `static` et il est impossible de créer une instance par un autre moyen. En revanche chaque appel créée une nouvelle instance, ce n'est pas un singleton.

La séquence réalisée par l'initialisation consiste :

1. A trouver les propriétés portant l'attribut `ComputedField` et à construire un dictionnaire utilisable facilement ensuite
2. A scanner `ObservableObject` par la Réflexion pour obtenir et conserver le `MethodInfo` de `RaisePropertyChanged(string)` qu'on utilisera ensuite
3. Enfin à se connecter sur le `PropertyChanged` de l'objet principal pour réagir aux changements des propriétés maitres.

L'objet `AutoInpc` ainsi initialisé est retourné à l'appelant notamment pour utiliser le mode "suspendu".

Le contrôle de l'attribut

Il s'agit ici de balayer les propriétés publiques de `l'ObservableObject` passé en référence à `Initialize()` et uniquement celles déclarée là (et non pas celle héritée) et de vérifier si elles ont été décorées ou non par notre attribut. Dans l'affirmative on obtient une instance de celui-ci et on peut alors utiliser la liste des champs pour construire un dictionnaire inversé qui sera utilisé ensuite pour savoir quand envoyer un signal INPC.

Je dis que le dictionnaire est inversé car nous obtenu le nom du champ calculé et son attribut donc la liste des champs dont il dépend, mais le dictionnaire créé n'est pas construit dans ce sens là. On repart des champs déclencheurs qui deviennent des entrées d'un dictionnaire dont la clé est le champ déclencheur et la valeur une liste de noms, ceux des champs calculés impactés. Lorsque le `PropetyChanged` de `l'ObservableObject` se déclenche il est très rapide de contrôler ce dictionnaire pour savoir s'il faut faire quelque chose. Et dans l'affirmative il est très rapide de balayer la liste des champs impactés (les champs calculés) pour envoyer un nouveau signal INPC pour chacun d'eux.

Tout ceci se traduit par le code suivant, au moins dans la partie analyse des attributs :

```
private void checkAttributes(ObservableObject vm)
      {
          var t = vm.GetType();
          var pi = t.GetProperties(BindingFlags.DeclaredOnly | BindingFlags.Instance |
                      BindingFlags.Public);

          foreach (var propertyInfo in pi)
          {
              ComputedFieldAttribute att;
              if (!tryGetAttribute(propertyInfo, out att)) continue;
              foreach (var field in att.TriggerFields)
              {
                  vm.VerifyPropertyName(field);
                  var targets = new List<string>();
                  if (triggerDefs.ContainsKey(field)) targets = triggerDefs[field];
                  else triggerDefs.Add(field, targets);
                  if (targets.IndexOf(propertyInfo.Name) == -1)
                      targets.Add(propertyInfo.Name);
              }
          }
      }
```

Le champ `triggerDefs` est le dictionnaire qui est déclaré comme suit :

```
private readonly Dictionary<string, List<string>> triggerDefs = new Dictionary<string,
List<string>>();
```

Chercher la méthode RaisePropertyChange

Il faut ensuite chercher dans l'objet principal `ObservableObject` (ou un descendant comme `ViewModelbase`) la bonne surcharge de `RaisePropertyChanged` qui

est `protected` et donc inaccessible autrement. On obtient une instance de `MethodInfo` que l'on va conserver pour les invocations plus tard.

```
private void getMethodInfo()

  {

      var t = typeof(ObservableObject);

      var methods = t.GetMethods(BindingFlags.Instance | BindingFlags.Public |
                                                          BindingFlags.NonPublic);

      simpleRaiseInpcMethod =

          (from methodInfo in methods

           where methodInfo.Name == "RaisePropertyChanged"

           let pp = methodInfo.GetParameters()

           where (pp.Length == 1) && (pp[0].ParameterType.Name == "String")

           select methodInfo).FirstOrDefault();

      if (simpleRaiseInpcMethod == null) throw
           new Exception("RaisePropertyChanged(String) method can't be found is sender.");

  }
```

La recherche est rendue plus ardue puisqu'il existe plusieurs signatures (surcharges) de la méthode qui nous intéresse et qu'il faut sélectionner la bonne…

Surveiller les changements de valeurs

Le décor est planté ne reste plus telle l'araignée dans sa toile à attendre qu'un champ de l'objet principal déclenche sans le savoir INPC dans son Setter pour lui tomber dessus !

L'abonnement à `PropertyChanged` ne mérite pas de commentaire mais voici le code de ce qui se passe quand un tel évènement est détecté :

```
private void vmPropertyChanged(object sender, PropertyChangedEventArgs e)

{

    if (!triggerDefs.ContainsKey(e.PropertyName)) return;

    if (sender==null) throw new Exception("Sender of INPC event is null.");

    var vm = sender as ObservableObject;
```

```
    if (vm == null) throw new Exception(

        string.Format("Sender of INPC event is not an ObservableObject. Property: {0}.
Sender type: {1}",e.PropertyName,sender.GetType().Name));

    foreach (var field in triggerDefs[e.PropertyName])

    {

        if (IsSuspended)

        {

            if (!waitingList.ContainsKey(field))

            {

                waitingList.Add(field, sender);

            }

            continue;

        }

        simpleRaiseInpcMethod?.Invoke(sender, new object[] {field});

    }

}
```

En réalité pour ne pas conserver une référence inutile sur l'objet principal le code retrouve l'instance de `l'ObservableObject` dans le `sender` de l'évènement. Je n'aime pas conserver des références qui créent des dépendances lorsque cela peut être évité.

Selon si le mode suspendu est actif ou non le code stockera l'évènement (le nom du champ déclencheur en fait) dans une liste ou bien invoquera grâce à l'objet `MethodInfo` obtenu lors de l'initialisation la méthode `RaisePropertyChanged` bien qu'elle soit `protected` et inaccessible. C'est une violation du principe d'encapsulation, c'est mal, mais comme on le voit si c'est utilisé en l'assumant cela permet de faire des choses impossibles…

Le stockage du `sender` avec le nom du champ pourrait paraitre superflu puisqu'il n'y a qu'un seul `sender`, `l'ObservableObject` qui a appelé l'initialisation de `l'AutoInpc`. Mais comme je ne garde pas de référence "durable" sur cet objet il faut bien le stocker temporairement (le mode suspendu n'est pas fait pour durer plus de quelques millisecondes dans les faits). Le fait de dépendre de toute façon de `l'ObservableObject` pour répondre à son `PropertyChanged` fait qu'il sera possible

dans une version à venir de finalement stocker une référence et de la réutilisr. C'est un point de détail.

Gérer le mode Suspendu

Ce n'est pas la partie la plus intéressante et je vous la laisse découvrir seul. La propriété `IsSuspended` permet à l'appelant de basculer en mode suspendu et d'en sortir. Durant la suspension on l'a vu ci-dessus les évènements sont conservés (et dédoublonnés) et lors de la sortie de la suspension la liste est "rejouée" comme une sorte de séquenceur musical. Une séquence jouée une seule fois puis vidée de ses éléments.

Plus loin

Plus loin c'est l'inconnu… C'est le futur et il dépendra du succès de ce petit morceau de code… Mais en dehors de débogue et de quelques petites améliorations il n'a pas vocation à beaucoup évoluer puisque son but est simple. Mais il ne faut jamais dire jamais ! Qui sait, par une nouvelle nuit d'insomnie peut-être me viendra-t-il en songe à demi-réveillé l'idée d'une nouvelle fonctionnalité…

Sinon vous avez la possibilité via codeplex de faire partie de l'équipe de développement et d'apporter vos rêves et votre expertise pour améliorer AutoInpc ! Je l'ai rêvé (au sens propre) et je l'ai fait, mais il est Open Source et vous pouvez l'améliorer et même le forker ! *It's up to you…*

Mvvm Light et INPC

Mvvm Light propose tout ce qu'il faut, ou presque, pour gérer le système de notification INPC. Notamment la librairie se base sur `ObservableObject` qui possède déjà les méthodes implémentées pour gérer cet évènement comme par exemple les différentes surcharges de RaisePropertyChanged.

Tout objet basé sur ObservableObject est donc potentiellement intéressé par DotBlog.AutoInpc !

Et parmi ces classes on trouve souvent des classes du Modèle mais aussi la super-star de MVVM : le **ViewModel**. La classe `ViewModelBase` de Mvvm Light descend en droite ligne d'`ObservableObject`.

Le problème des champs calculés

Entrons tout de suite dans le vif du sujet et prenons pour exemples le ViewModel le plus simple possible : Il définit deux champs, `FirstName` et `LastName` (nom et prénom) de type `string`. Grâce aux méthodes d' `ObservableObject` dont hérite `ViewModelBase` ces deux propriétés savent lever INPC lorsque leur contenu change.

Ajoutons juste un troisième champ, `FullName` qui est une propriété `string` aussi en lecture seule (pas de `Set`) et dont le Getter se résume à retourner le prénom et le nom séparés par un espace.

Ecrivons la Vue et ajoutons deux `TextBox` pour la saisie du nom et du prénom et un `TextBlock` pour l'affichage de `FullName`. On compile, on lance et là …

On a beau modifier le nom et le prénom, nos bindings ont beau être parfaits, le champ "`FullName`" ne s'affiche tout simplement pas !

Le coupable : INPC.

Les champs `FirstName` et `LastName` disposent d'un Setter et c'est dans celui-ci qu'est levé INPC. Or, `FullName` ne possède pas de Setter (et pour cause sa valeur n'est que déduite d'autres valeurs). *Mais sans Setter pas de INPC …*

Comment faire alors ?

La seule solution qui reste consiste à modifier les deux `Setters` des deux propriétés "vraies" et en fin de leurs Setters ajouter un `RaisePropertyChanged(()=>Fullname);`

Si demain on gère à l'américaine un `MiddleName`, qu'on ajoute un titre (ou formule de politesse) et que tout cela doit apparaitre dans `FullName` il faudra se rappeler de l'existence de cette propriété et ajouter dans chaque nouvelle propriété le fameux `RaisePropertyChanged` sur `FullName`. Et si demain nous créons une propriété `FullNameWithAge` il faudra ajouter encore et encore un `RaisePropertyChanged` adapté à `FirstName` et `LastName` mais aussi à la propriété `Age`. Etc… **C'est un enfer.**

Et je ne vous parle ici que d'un cas ultra simple, presque idéal, juste valable pour une démo… Dans la réalité les dépendances sont souvent plus nombreux, plus complexes, et la maintenance devient vite aléatoire. Il n'est pas rare de voir de tels champs calculés ne pas se mettre à jour correctement dans certaines circonstances à cause d'un oubli. *Car tout le problème est que ce n'est pas dans le champ calculé qu'il faut agir mais sur un nombre variables d'autres champs qui n'ont pourtant pas à connaitre le ou les champs calculés qui en dépendent !*

La solution DotBlog.AutoInpc

Il m'est apparu qu'il y avait un manque dans Mvvm Light, INPC n'y est pas traité totalement car il ne prévoit rien pour les champs calculés qui sont pourtant monnaie courante. Mais Mvvm Light est Light et c'est aussi son gros avantage. Donc plutôt que de pleurer sur ce manque, ne serait-il pas plus simple de le régler par une mini-librairie ?

C'est en tout cas ce que je me suis dit et *DotBlog.AutoInpc* en est le résultat.

Comment ça marche ?

De la façon la plus simple qu'il soit : **un simple et unique attribut** vous permet de déclarer sur le champ calculé lui-même quels sont les champs "triggers" (déclencheurs) dont il dépend.

Dans notre exemple plus haut il suffit donc de décorer le champ `FullName` par cet attribut en indiquant les champs `FirstName` et `LastName` et voilà c'est tout !

Chaque champ calculé sait de quoi il a besoin, et les autres champs n'ont pas à connaitre FullName.

Exemple

Prenons un ViewModel qui suit l'exemple cité plus haut du nom et du prénom, il déclarera donc deux propriétés de ce type :

```
public string FirstName { .... }

public string LastName { .... }
```

Puis pour le nom complet qui est le champ calculé nous écrirons le code suivant :

```
[ComputedField(new [FirstName","LastName)]

public string FullName => FirstName " " LastName;
```

L'écriture de la propriété elle-même ne doit pas vous étonner, c'est du C# 6. Ce qui compte c'est l'attribut `ComputedField` qui décore la propriété `FullName`. Sa syntaxe est on ne peut plus simple puisque qu'il suffit de passer en paramètre la liste des champs

dont dépend celui qui est calculé. Ici `FullName` dépend de `FirstName` et de `LastName` et lorsqu'un de ces deux champs sera modifié INPC sera levé automatiquement. Rien d'autre à écrire !

Un petit dessin valant mieux que…

(cliquez sur l'image pour voir l'animation sur Dot.Blog)

L'animation ci-dessus nous montre comment les modifications des champs `FirstName` et `LastName` sont répercutées sur `FullName` grâce à la seule présence de notre **Attribut** !

Mais la démo va un peu plus loin car vous apercevez deux boutons, "`Suspend AutoInpc`" et "`Resume AutoInpc`". A quoi cela sert-il ?

Le mode Suspendu

Arrivé à ce stade j'avais rempli le contrat que je m'étais fixé : ***régler de façon élégante et simple le problème des champs calculés dans les ViewModel et les objets dérivés de ObservableObject.***

Mais il manquait quelque chose.

Dans la réalité il n'est pas rare que le code lui-même *modifie simultanément plusieurs champs* qui peuvent avoir *un impact sur un ou plusieurs champs calculés*. Il est alors un peu idiot de lever autant de fois INPC alors qu'une seule fois en fin de manipulation de tous les champs serait bien plus propre et *efficace*.

Il arrive aussi que des champs calculés ne puissent être correctement rendus qu'une fois plusieurs champs ont été initialisés correctement. Tous les INPC intermédiaires causent une faute dans le ou les champs calculés jusqu'à temps qu'ils puissent être rendus correctement. Il serait plus simple de couper l'INPC automatique sur les champs calculés jusqu'à ce que tous les champs maitres soient correctement renseignés. *Mais lorsque l'état suspendu est « résumé » (annuler dirons nons) il faut que toutes les notifications qui n'ont pas été rendues le soient !*

Mieux, il faudrait que les INPC soient regroupées car si un même champ a été notifié 10 fois durant la suspension, il n'est nécessaire de le notifier qu'une seule fois lorsque l'état suspendu est stoppé…

C'est exactement à ces problèmes que répond la propriété `IsSuspended` de `AutoInpc`.

Une fois passée à `True` tous les INPC sont stockés dans une liste d'attente sans doublon. Et lorsqu'on le repasse à `False`, son état de base, tous les INPC en attente sont rendus (sans le doublons donc).

Maintenant vous pouvez mieux comprendre l'animation plus haut et le jeu des deux boutons de suspension de l'INPC…

Gratuit et Open Source

`DotBlog.AutoInpc` est une librairie gratuite, comme tout ce que j'offre ici, mais mieux, elle est **Open Source sur CodePlex** !

Rendez-vous sur http://autoinpc.codeplex.com/ pour télécharger la version 1.0 considérée comme Beta tant qu'elle n'aura pas été testée par suffisamment de personnes.

Participer ?

Oui c'est possible ! AutoInpc est pour l'instant une simple DLL UWP contenant deux fichiers sources. On peut les intégrer tel quel dans tout projet ce n'est pas compliqué. Mais ça serait bien d'en faire un paquet Nuget par exemple, voire d'améliorer ou

étendre les fonctionnalités. On peut aussi penser à des adaptations pour d'autres framework MVVM.

Si vous souhaitez m'aider dans ces petites tâches signalez-vous sur le site d'AutoInpc sur Codeplex pour entrer dans l'équipe de dev …

Comment ça marche AutoInpc ?

Je pourrais vous répondre qu'il n'y a qu'à télécharger le source sur CodePlex. Mais ce n'est pas mon habitude. Tout cela n'a d'ailleurs d'intérêt que parce que je peux écrire quelque chose sur le sujet ici. C'est une occasion de voir du code, d'en parler et de partager notre passion commune.

Donc ne vous inquiétez pas *un prochain article fera l'étude de code de AutoInpc* car il y a malgré tout deux ou trois petites choses intéressantes je pense.

Paquet Nuget pour VS 2015

Depuis l'écriture de cet article un sympathique lecteur de Dot.Blog, Marc Plessis, a eu la gentillesse de fabriquer le paquet Nuget que j'ai pu ensuite publier. Désormais vous pouvez installer AutoINPC directement dans VS 2015 via le gestionnaire de paquets ! Les sources sur CodePlex reflètent le code complet avec son paquet Nuget. Si vous voyez des améliorations à faire, n'hésitez pas à participer aussi !

Conclusion

Mvvm Light est une librairie MVVM dont j'ai toujours loué la simplicité et l'efficacité, j'y ai consacré des pages et des pages. AutoInpc règle une petite lacune de Mvvm Light et cette solution est transposable à d'autres framework car ce problème est récurrent.

Gratuit, Open Source et en plus documenté et bientôt même commenté ici, que rêver de plus ! … Que mes chers lecteurs fassent un peu de pub à ce petit projet car il intéressera tous ceux qui utilisent Mvvm Light !

Gérer le debug du cycle de vie en debug sous VS 2015

Une application UAP possède un cycle de vie bien particulier et il faut pouvoir le tester en debug comme le reste. Mais comment ?

VS 2015 cache son jeu !

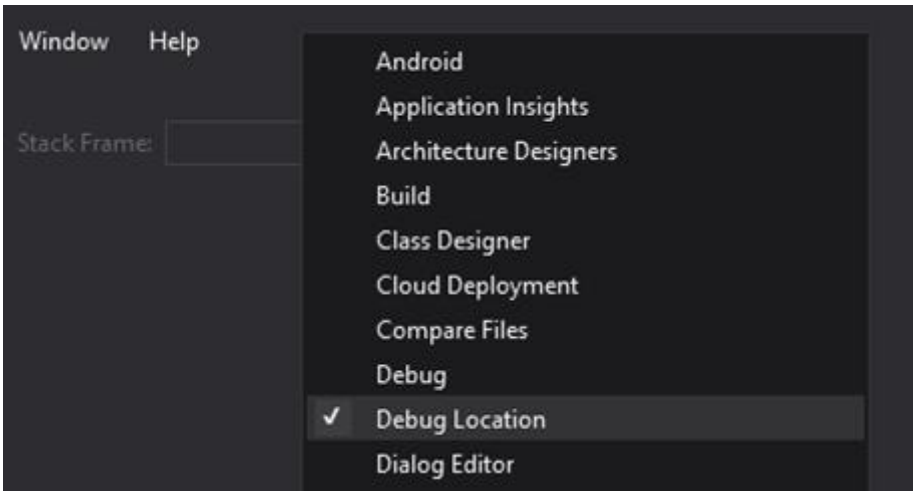
Il s'agit d'une astuce rapide donc je ne m'étendrai pas des heures : Admettons que nous sachions tout du cycle de vie et que nous ayons bien tout implémenté. Encore faut-il le tester, mais comment ?

Visual Studio 2015 cache son jeu et les options permettant de simuler l'état suspendu ou d'autres états du cycle de vie ne sont pas visibles par défaut…

Activer les options de gestion du cycle de vie

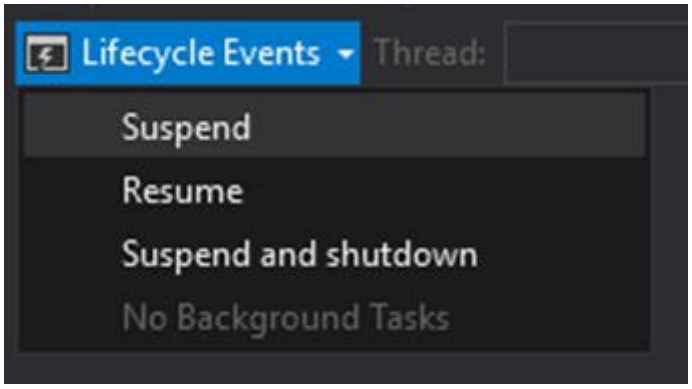
L'astuce est pourtant simple mais trouver l'endroit et la bonne entrée dans la liste est plus compliqué…

Il faut tout simplement faire un clic-droit sur la barre de menu de VS (File, Edit… pas une autre barre), une liste s'affiche alors et il suffit d'activer "**Debug Location**" ce qui n'est pas franchement parlant…



Une fois cette palette activée vous disposerez d'une barre entière permettant de contrôler / visualiser le Process, les threads, la Stack Frame etc. Mais aussi et surtout pour ce qui nous intéresse ici, un bouton appelé "**Lifecycle Events**" qui vous permettra de forcer l'application en cours de debug à subir les évènements particuliers du cycle de vie UWP qui sinon resteraient impossible à tester.

Maintenant lancez votre application UWP et le bouton sera activé, à vous de tester les différentes situations et comment votre App y réagit !



Conclusion

Gérer correctement le cycle de vie d'une application UWP est essentiel. Dans la négative l'UX sera déplaisante voire frustrante pour l'utilisateur et c'est le rejet garanti !

Encore faut-il pouvoir tester si tout a été bien géré, VS 2015 le permet mais la barre de commande étudiée pour cela est bien cachée…

Juste une astuce rapide donc.

Les exemples de code officiels UWP

Le plus efficace pour apprendre une technologie nouvelle c'est de lire des exemples de code et de se les approprier en les modifiant, en les recréant. Pour UWP il existe une ressource à ne pas manquer…

Le centre de développement

Le centre de développement des applications du Windows 10 est l'endroit où chercher des informations, des documents, du support, du code, et accéder aux communautés.

L'adresse est à book-marquer absolument : dev.windows.com/fr-fr/

Les exemples de code

On accède aussi à centaines d'exemples de code écrits par Microsoft qu'il serait bête de louper.

Il y a aussi des exemples soumis par la communauté.

Bref, tous les exemples sont à trouver soit

- sous la forme d'un fichier zip de tous les exemples à jour
- ou bien en explorant les sources sur GitHub

Le fichier de tous les exemples

Piocher parmi tous les exemples proposés est parfois très pratique et la version GitHub propose quelques sections pour rendre la navigation plus rapide. En revanche télécharger un à un tous les exemples peut devenir un enfer. Dans ce cas je vous conseille de télécharger le fichier Zip qui contient tous les exemples. Il faudra faire le ménage des langages sans intérêt, mais c'est à chacun de voir !

N'oubliez pas de faire un clic droit sur le zip reçu et de cliquer sur "débloquer" avant de décompresser. Les ressources téléchargées sur le Web et non validées de cette façon restent ensuite marquées par Windows qui s'en méfiera à chaque utilisation ! Mieux vaut les valider avant.

Conclusion

La nuit tombe de plus en plus tôt… les longues soirées d'automne s'annoncent déjà en cette fin août… La fraicheur et le vent s'installent, fini les soirées jusqu'à plus d'heure sur la plage, sauf en anorak. Vous savez maintenant comment utiliser tout ce temps libre intelligemment !

Au lit avec votre Surface, une bonne musique au casque et plein d'exemples de code, une pizza et du café sur la table de nuit, c'est pas le paradis du geek ça(*) ? :-)

Stay Tuned !

(*) Les lecteurs de Dot.Blog en couple ne doivent pas suivre ce conseil stupide à moins de vouloir précipiter une séparation…

Mvvm Light : un template de projet !

La dernière version de Mvvm Light fonctionne très bien sous UWP mais Laurent n'a pas eu le temps d'intégrer les habituels templates de projets à la librairie. Ce n'est pas très difficile à faire à la main mais on peut oublier un réglage ici ou là. Voici donc un template en attendant que les officiels (re)fassent leur apparition dans Mvvm Light !

Template MVVM Light

C'est tout bête mais les templates Visual Studio font gagner beaucoup de temps et permettent de démarrer très rapidement une application sans se poser trop de questions sur les pré-requis.

Mvvm Light n'est pas une librairie très complexe et comme je l'ai montré dans un article récent ajouter à la main les petites modifications au projet n'est pas sorcier. Encore faut-il savoir quoi changer, où le changer et quoi ajouter.

Partir d'un template est donc mille fois plus facile.

Mais pour l'instant Laurent n'a visiblement pas eu le temps d'ajouter les habituels templates au paquet Nuget de Mvvm Light 5.1.1 (la dernière quand j'écris ces lignes).

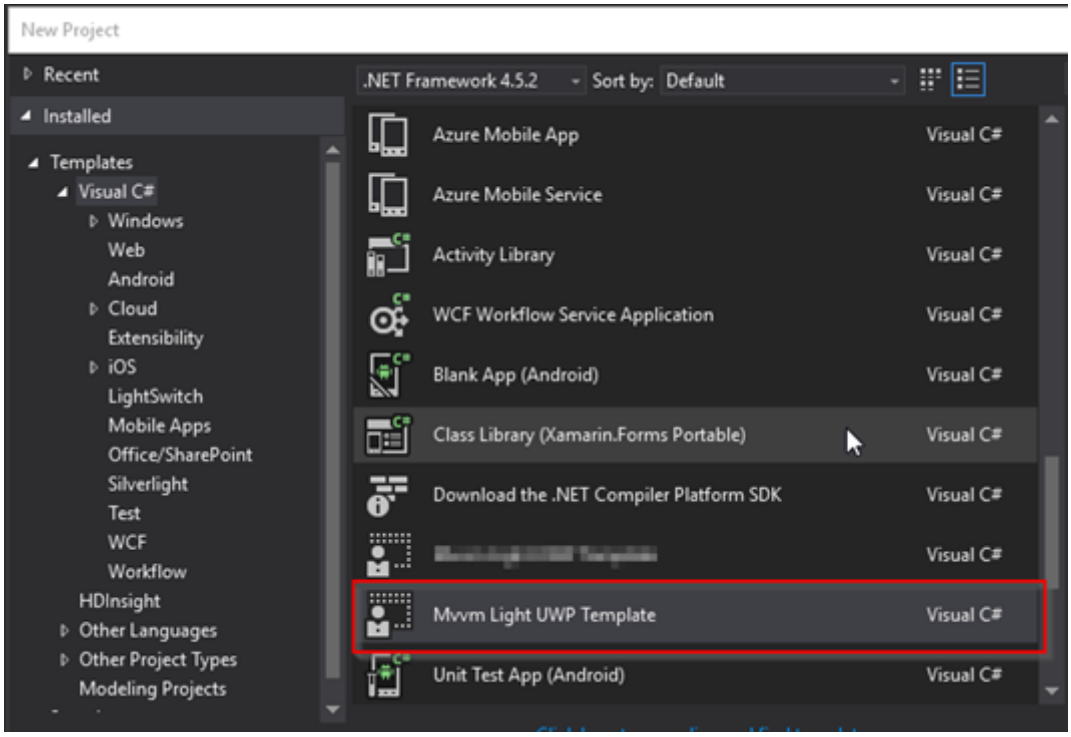
A noter : forcément les versions de Mvvm Light évoluent, la 5.2 est déjà disponible à la rédaction de ce livre PDF. Mais le template est toujours valable, il suffit de demander une mise à jour des paquets Nuget utilisés si cela n'est pas fait automatiquement. Opération rapide qui vous permettra de bénéficier de tous les avantages du template !

Un template temporaire

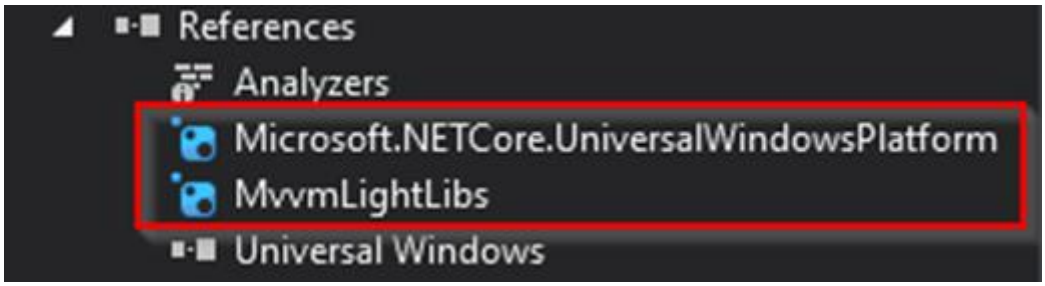
Comme je déteste les choses répétitives je me suis fais un template Mvvm Light. Et comme je suis généreux, plutôt que de le garder pour moi je vais le partager avec vous  il est là, il attend juste d'être téléchargé :

MvvmLightDotBlogTemplate

Une fois le VSIX installé dans VS 2015, faites nouveau / projet puis chercher le template suivant :



A noter une petite curiosité : lorsqu'on créée un nouveau projet avec ce template deux références ne semblent pas être ajoutées, ce sont celles entourées de rouge dans la capture suivante :



Il ne s'agit pas d'un oubli mais d'un comportement étrange de VS 2015. La première chose à faire c'est de faire un Build. Puis essayez de changer la cible qui se met parfois par défaut à ARM. Passez là à x86 ou x64, après quelques essais de ce genre les deux références apparaissent enfin et tout rentre dans l'ordre. Car tant que VS s'obstine à ne pas les afficher les références les plus simples comme celles à "`System`" sont en erreur dans le code. Alors que le Build passe sans erreur lui…

Bref, une fois les références revenues à leur place (un peu magiquement) le template est fin prêt à être utilisé !

Un Run vous le prouvera :

De même qu'en mode design vous verrez s'afficher :

Selon les habitudes de Laurent j'ai conservé un `IDataService` et deux implémentations, l'une de design et l'autre de runtime. Donc au design on dispose de données de design (le message indique "debug" c'est une petite erreur mon clavier a fourché il faut lire "design" mais refaire le VSIX pour ça était un peu contraignant…).

Bref, vous trouverez dans ce template tout ce qu'on trouve généralement dans les templates fournis habituellement avec Mvvm Light à savoir des services, un répertoire pour les ViewModel avec le MainViewModel et le `ViewModelLocator`, une `MainView` connectée au `MainViewModel` et sa seule propriété "Hello", la création de la ressource pour le locator dans `App.xaml`, l'initialisation du `DispatcherHelper`, etc.

Tout cela est déjà fait et on peut donc travailler tout de suite…

Conclusion

La création d'un template installable en mode VSIX n'est pas très compliquée et cela simplifie tellement les choses…

En attendant les templates officiels qui finiront bien par arriver un jour…

Les papiers des lecteurs : Créer un template UAP (UWP) en VSIX pour Visual Studio

Inaugurée en 2014 la rubrique "*Les papiers des lecteurs*" permet aux fidèles lecteurs de Dot.Blog de publier ici leur articles ! Aujourd'hui Philippe Matray va nous expliquer comment créer des templates VS 2015 packagés en VSIX pour en simplifier l'installation dans l'EDI !

Les papiers des lecteurs

Tout le monde n'a pas forcément l'envie ou le temps d'alimenter un blog comme Dot.Blog avec une moyenne de 100 articles par an depuis 8 ans, des livres, etc. Mais j'en suis convaincu, *tous les lecteurs ont des connaissances uniques, un savoir-faire, un truc qu'ils aiment et dont ils sont tout à fait capable de parler en quelques paragraphes…*

"Les papiers des lecteurs" est votre rubrique. Si vous avez une idée, l'envie de partager un truc, une astuce ou beaucoup plus sans pour autant vouloir créer un Blog avec tout ce que cela sous-entend, Dot.Blog vous offre la possibilité de publier vos papiers !

Comme vous le savez Dot.Blog fonctionne sans aucune publicité, je n'en tire aucun bénéfice autre que le plaisir de partager avec les autres une même passion. En publiant sur Dot.Blog *vous n'alimenterez pas les caisses d'un site à votre dépens (si si,*

ça existe mais vous en connaissez déjà les noms…), vous partagerez tout aussi gratuitement votre connaissance, juste pour le plaisir !

Inaugurée pour les TechDays 2014 avec notre "*envoyée spéciale*" Alice Barralon (Que retenir des Techdays 2014 ?), j'ai envie de donner encore plus de vie à cette rubrique en lui donnant un nom et en faisant cet appel à tous ! **Hier Alice, aujourd'hui Philippe, demain : vous ? ! Ne soyez pas timides on a tous quelque chose à partager !**

Créer des templates VSIX avec Visual Studio 2015

Un article de Philippe Matray.

UWP, bien que très prometteuse, est une plateforme jeune. Elle est la dernière-née du fastueux mariage entre C# et XAML qui a donné naissance à des technologies incroyables telles que WPF, WinRT et le regretté Silverlight parti malheureusement trop jeune.

Comme toutes les jeunes plateformes, celle-ci a besoin d'outils divers afin de rendre le développeur plus productif.

Nous avons vu que créer un template est très facile mais comment le partager avec toute son équipe ? Comment le rendre disponible à tout le monde ? Et bien, Microsoft a pensé à tout.

Visual Studio SDK

Le kit de développement Visual Studio SDK s'installe en même que Visual Studio si vous avez coché la case pendant son installation. Si ce n'est pas le cas, pas de panique, il vous suffit d'aller dans le menu « Ajouter ou Supprimer un programme », de sélectionner Visual Studio et de cliquer sur le bouton modifier pour ajouter le SDK.

Création d'un template

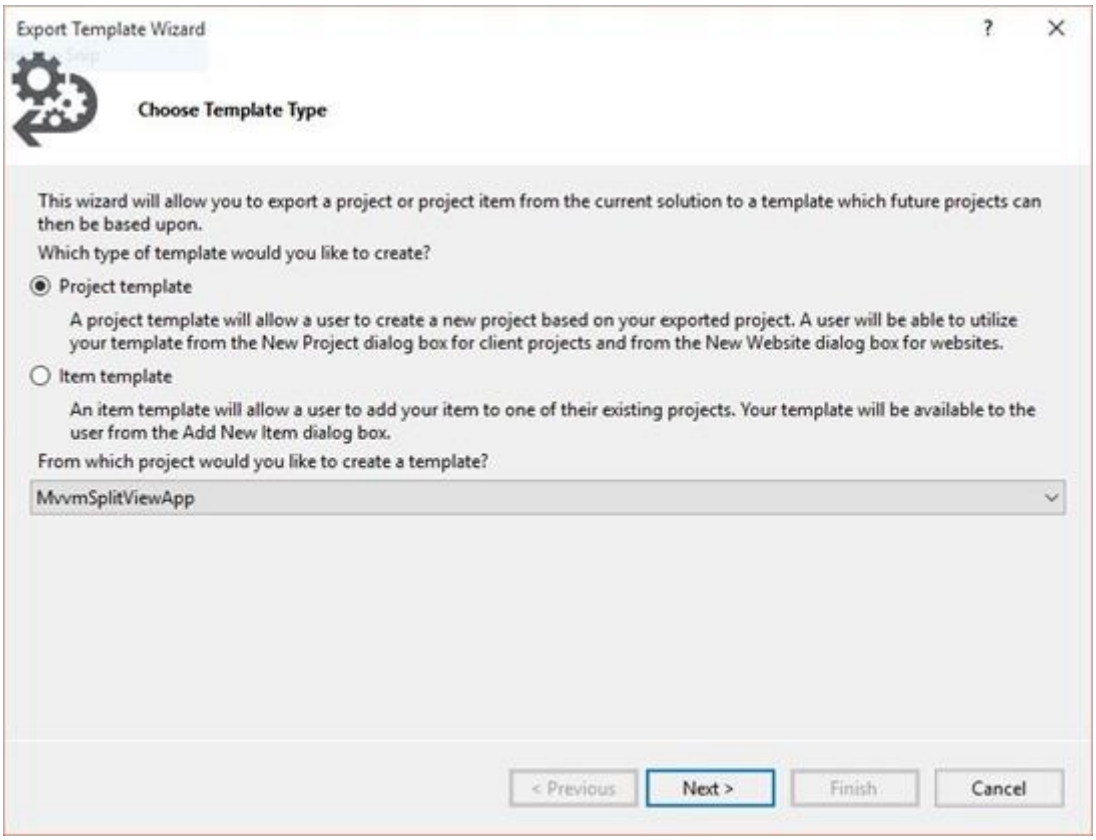
MSDN nous donne la marche à suivre.

- How to: Create Project Templates
  https://msdn.microsoft.com/en-us/library/xkh1wxd8.aspx

Commençons par ouvrir Visual Studio et créons un projet qui nous servira de modèle. Pour l'exemple, le projet que je créé est une Split View App UWP exploitant MVVM Light dans sa dernière version (5.2).

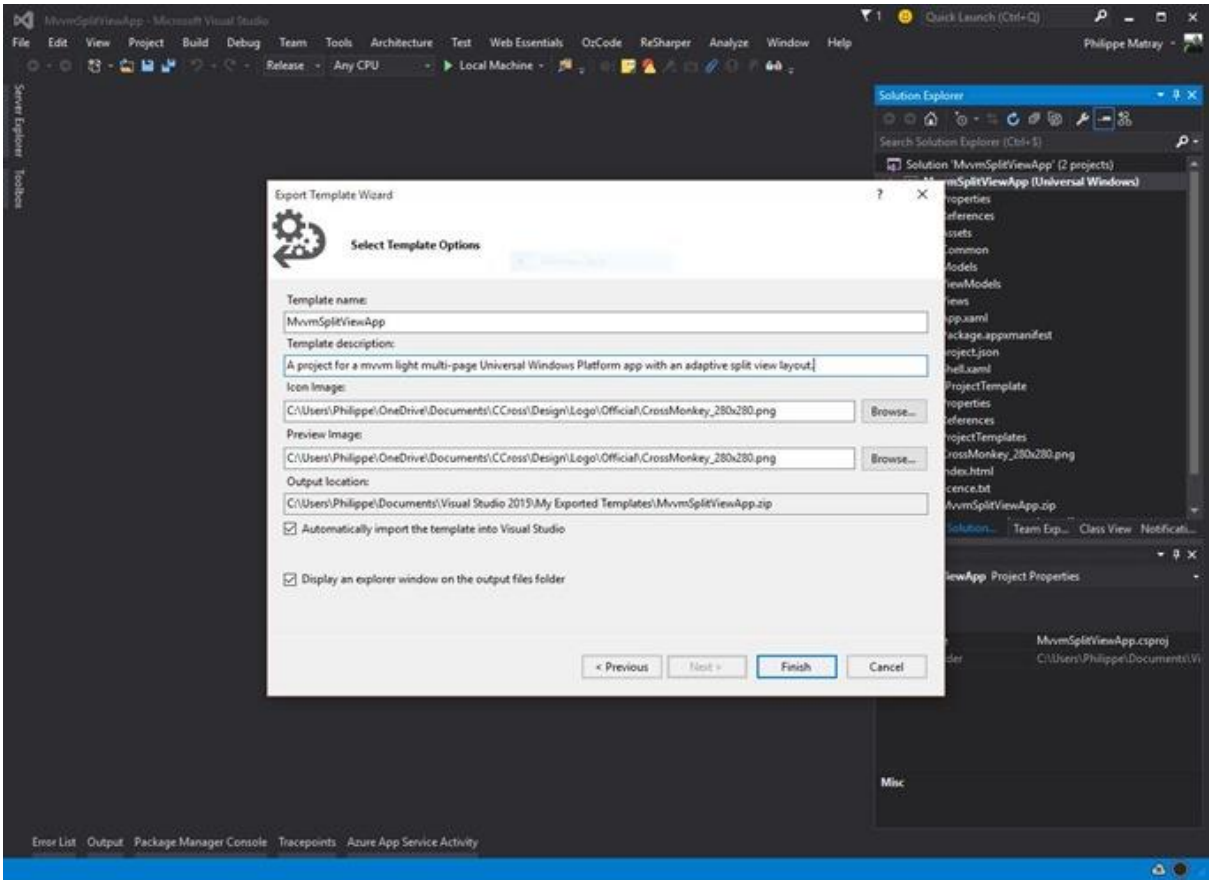
Editons-le jusqu'à ce qu'il soit propre au point d'être exportable en tant que template.

Créer un template est extrêmement simple. Une fois votre projet créé, un petit tour dans File > Export template... ouvrira le wizard suivant :



À partir de ce wizard, choisissez « Project template » et sélectionnez le projet que vous souhaitez exporter.

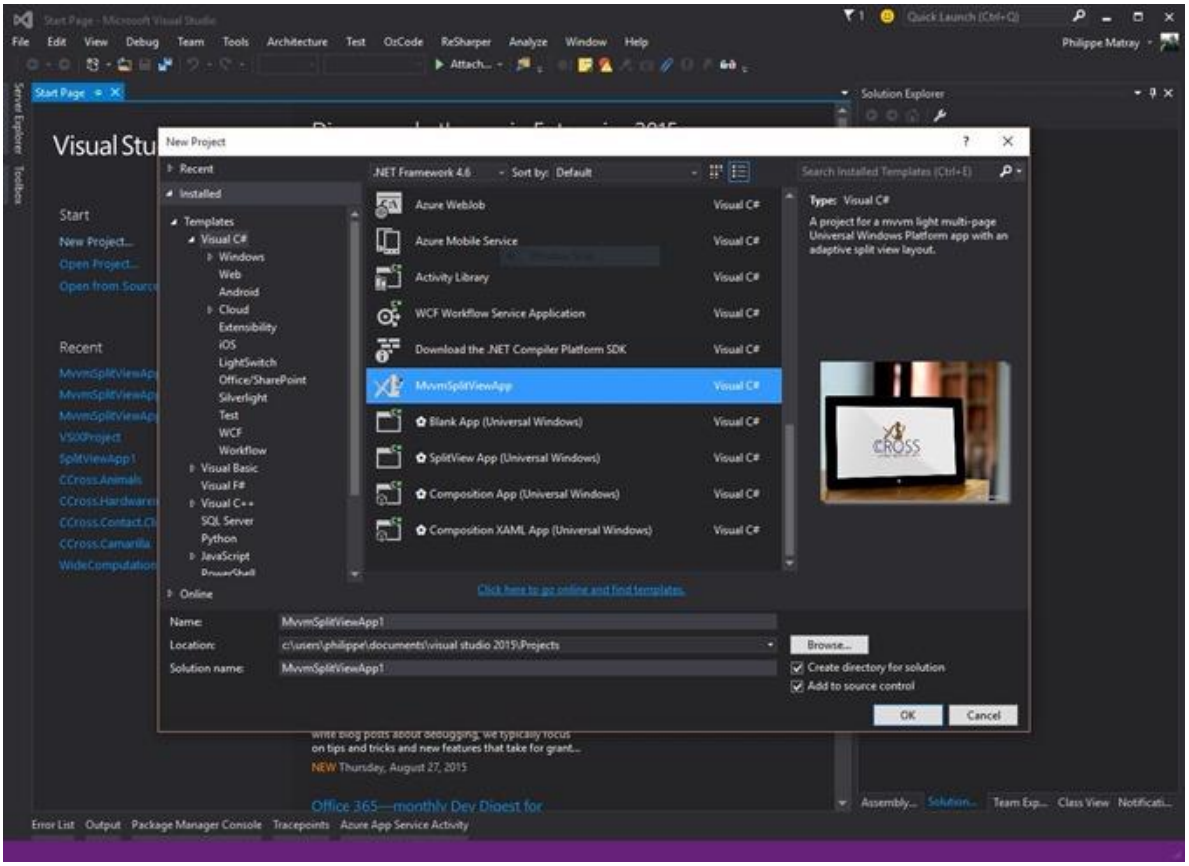
Sur la page suivante, il vous est demandé d'indiquer un nom pour votre template, une description de celui-ci, une photo pour la miniature et une seconde photo pour l'aperçu.



(Rappel : clic pour une vue à 100%)

Au terme de cette boite de dialogue, vous aurez à disposition un fichier zip qui est déjà un template exploitable. Mais on peut faire mieux et plus facilement partageable…

Gardez-le donc précieusement nous allons l'empaqueter dans un **VSIX** afin de pouvoir l'installer sur d'autres machines et/ou de le partager avec vos amis et collègues.



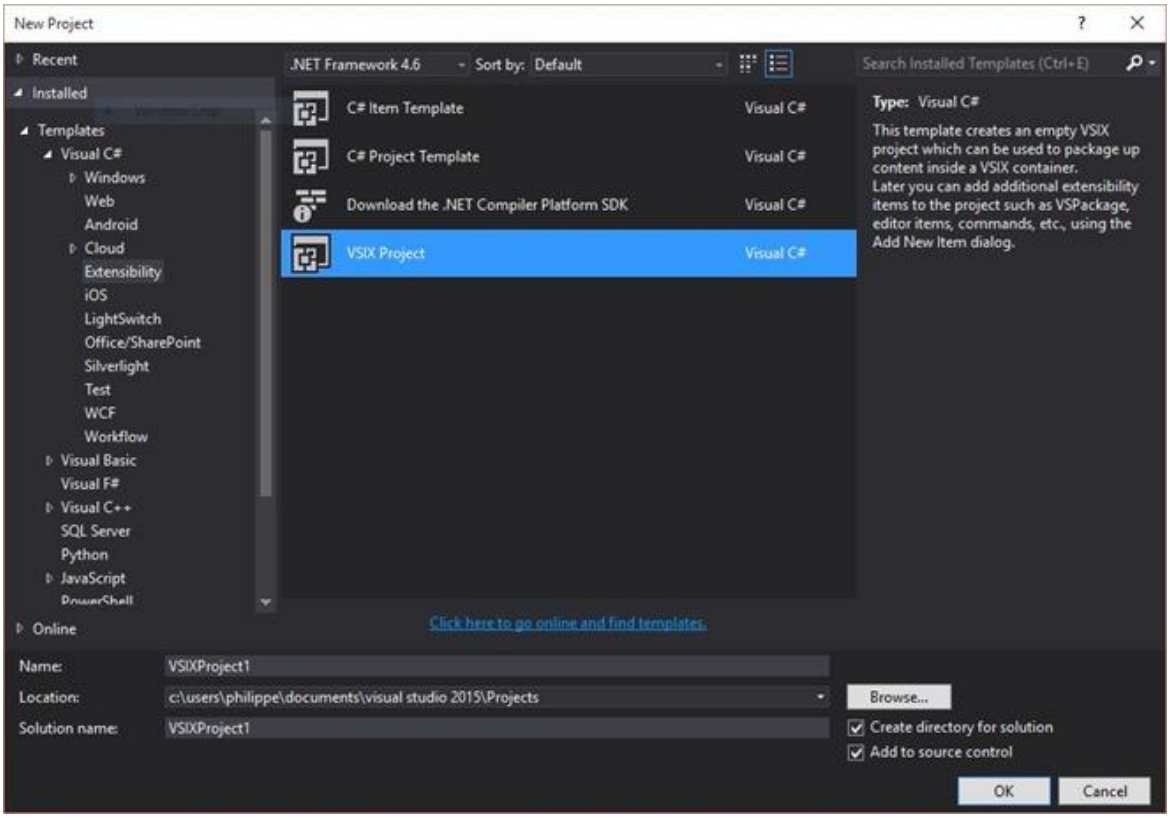
[NdlR]Il me semble bien en effet que le processus installe en même temps le template dans VS. Lorsqu'on créée ensuite le VSIX il est donc inutile d'installer à nouveau celui-ci sur la machine ayant servie à créer le template Zip.[fin de note]

Création du VSIX

Encore une fois MSDN nous mâche la besogne.

- Getting Started with the VSIX Project Template https://msdn.microsoft.com/en-us/library/dd885241.aspx

Désormais nous pouvons créer un second projet dans la solution de type VSIX Project. Ce type de projet est localisé sous Templates > Visual C# > Extensibility.



Déplacez le zip créé précédemment dans ce nouveau projet. Cliquez ensuite dessus et à la ligne « Copy to output Directory » des propriétés du fichier, sélectionnez l'option Copy Always.

Ouvrez ensuite le fichier vsixmanifest. Un designer s'ouvre et vous demande de renseigner le nom de votre extension, un identifiant unique pour celle-ci, votre nom, une description du plugin (pour faire simple, reprenez la même).

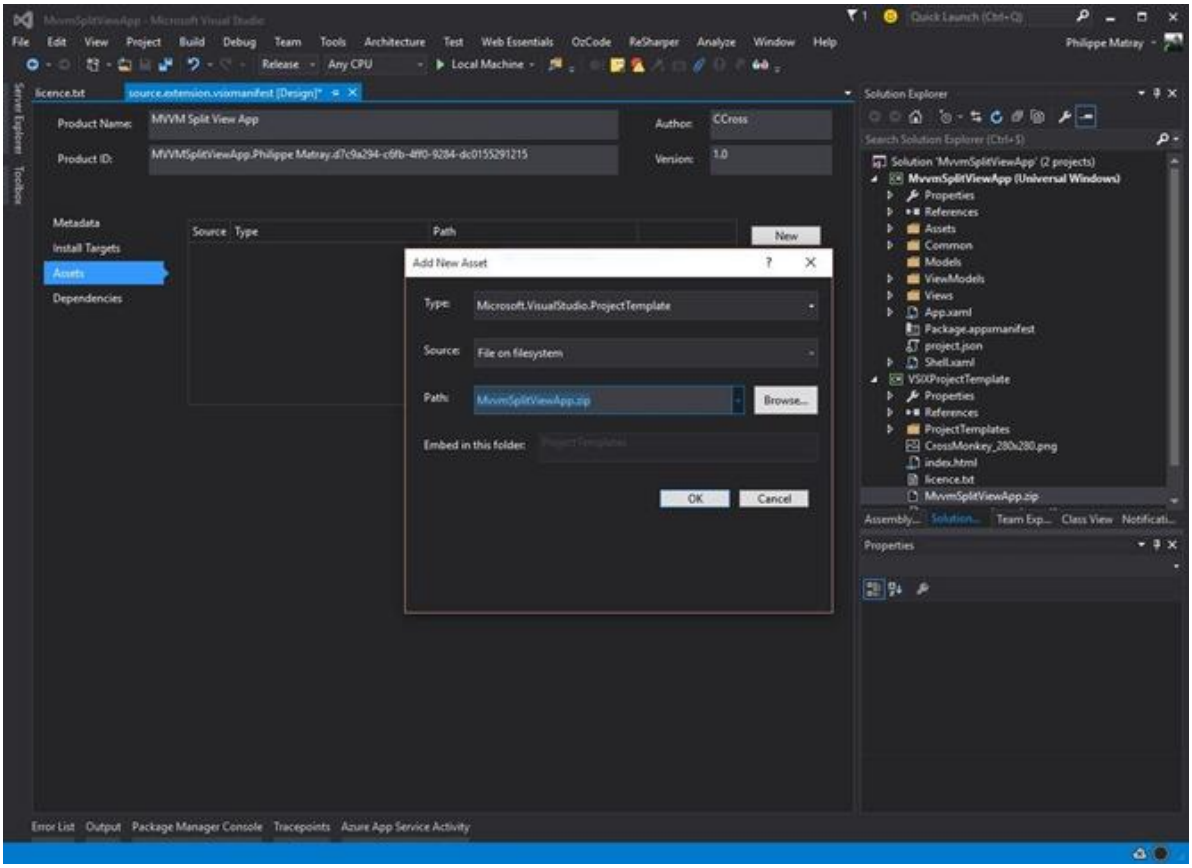
La licence s'enregistre dans un fichier texte et n'est pas obligatoire. Pour mon exemple, je me suis basé sur la MS-PL dont le texte est consultable ici.

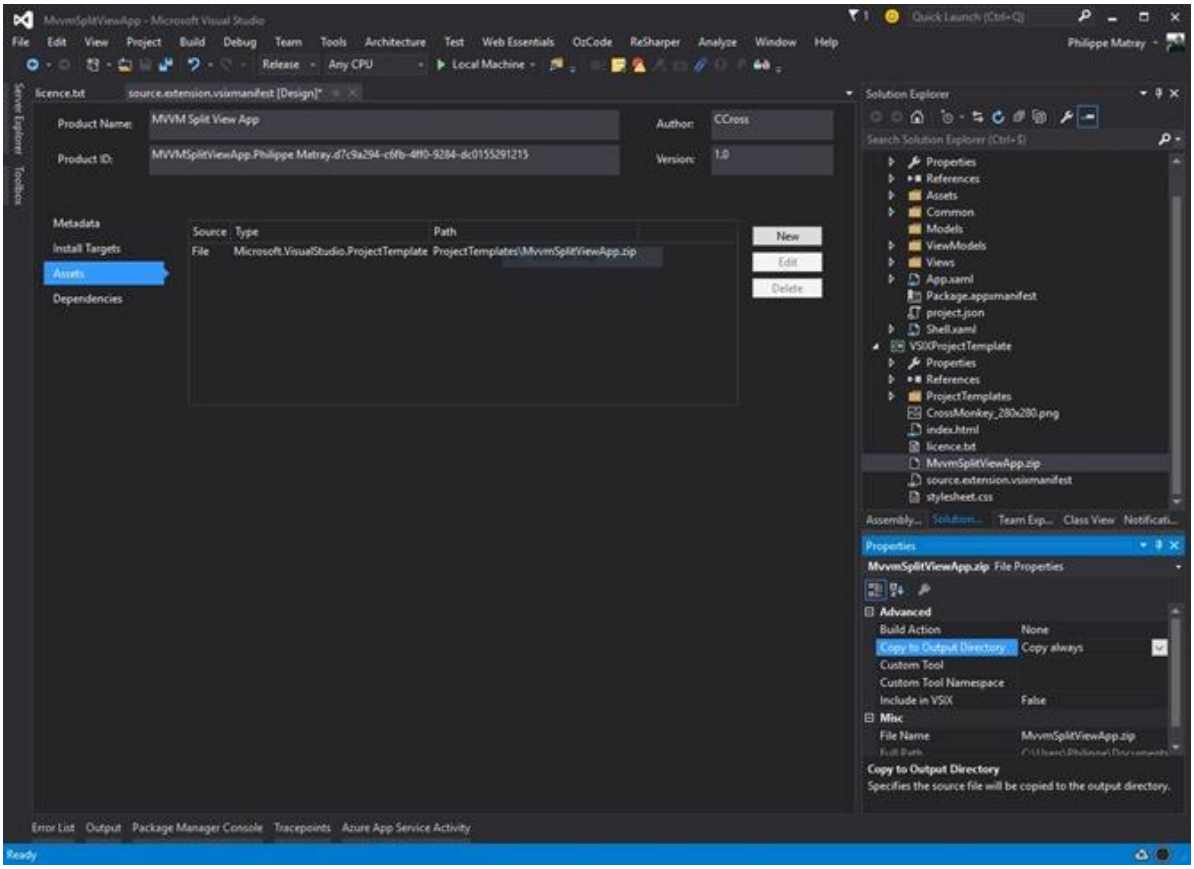
http://directory.fsf.org/wiki/License:MsPL

Enfin, dans la section Assets, ajoutez un élément de type Microsoft.VisualStudio.ProjectTemplate et ciblez le chemin du fichier zip que vous avez ajouté à ce projet.

Vous pouvez maintenant sauver et fermer le fichier vsixmanifest, compilez le projet et récupérez votre fichier .vsixdans le répertoire de build.

Ca y est, vous disposez enfin d'un fichier .vsix partageable avec vos amis et vos collègues *mais nous n'allons pas nous arrêter en si bon chemin et allons le rendre disponible sur internet*.



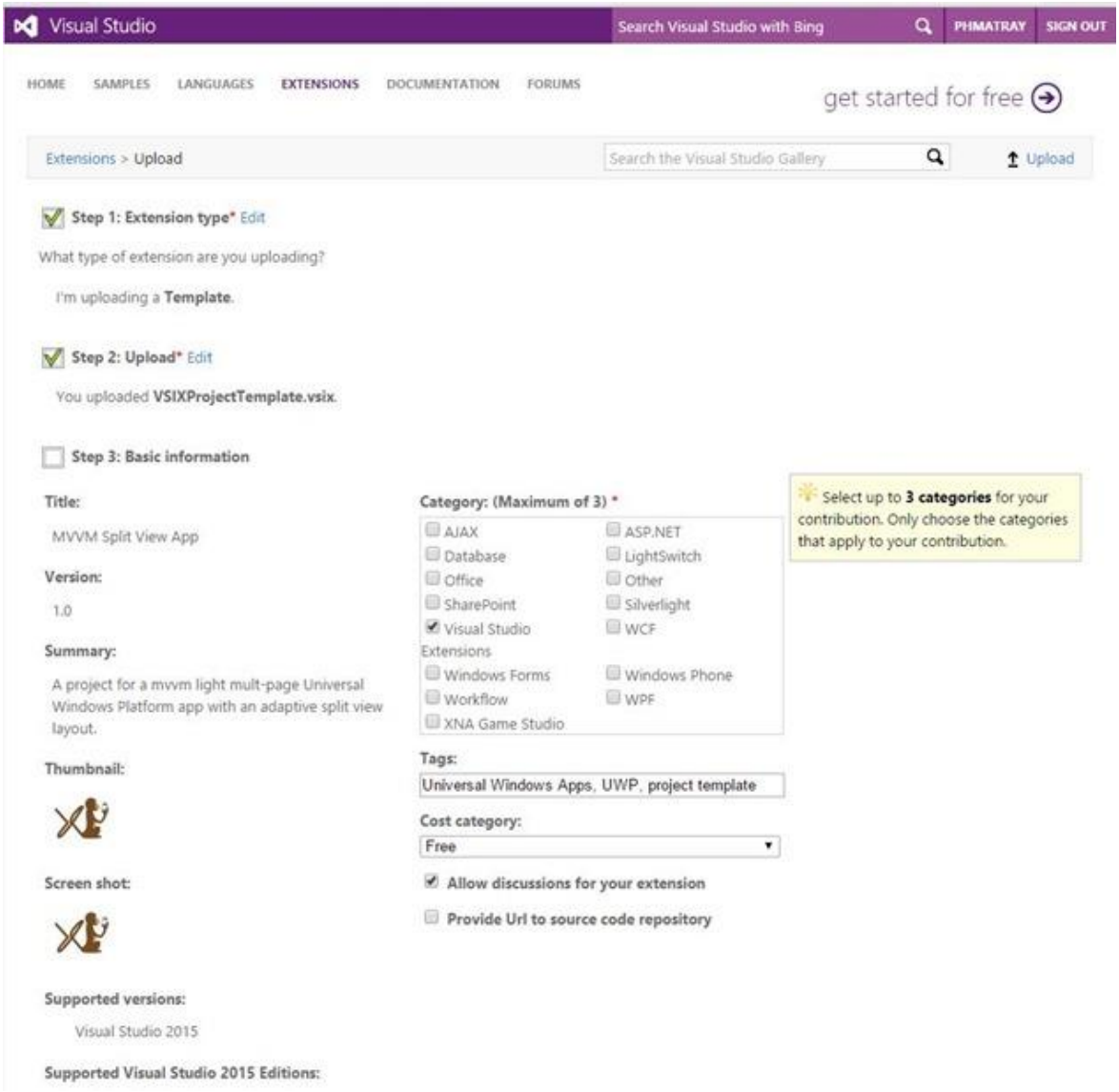


Publication sur VisualStudioGallery

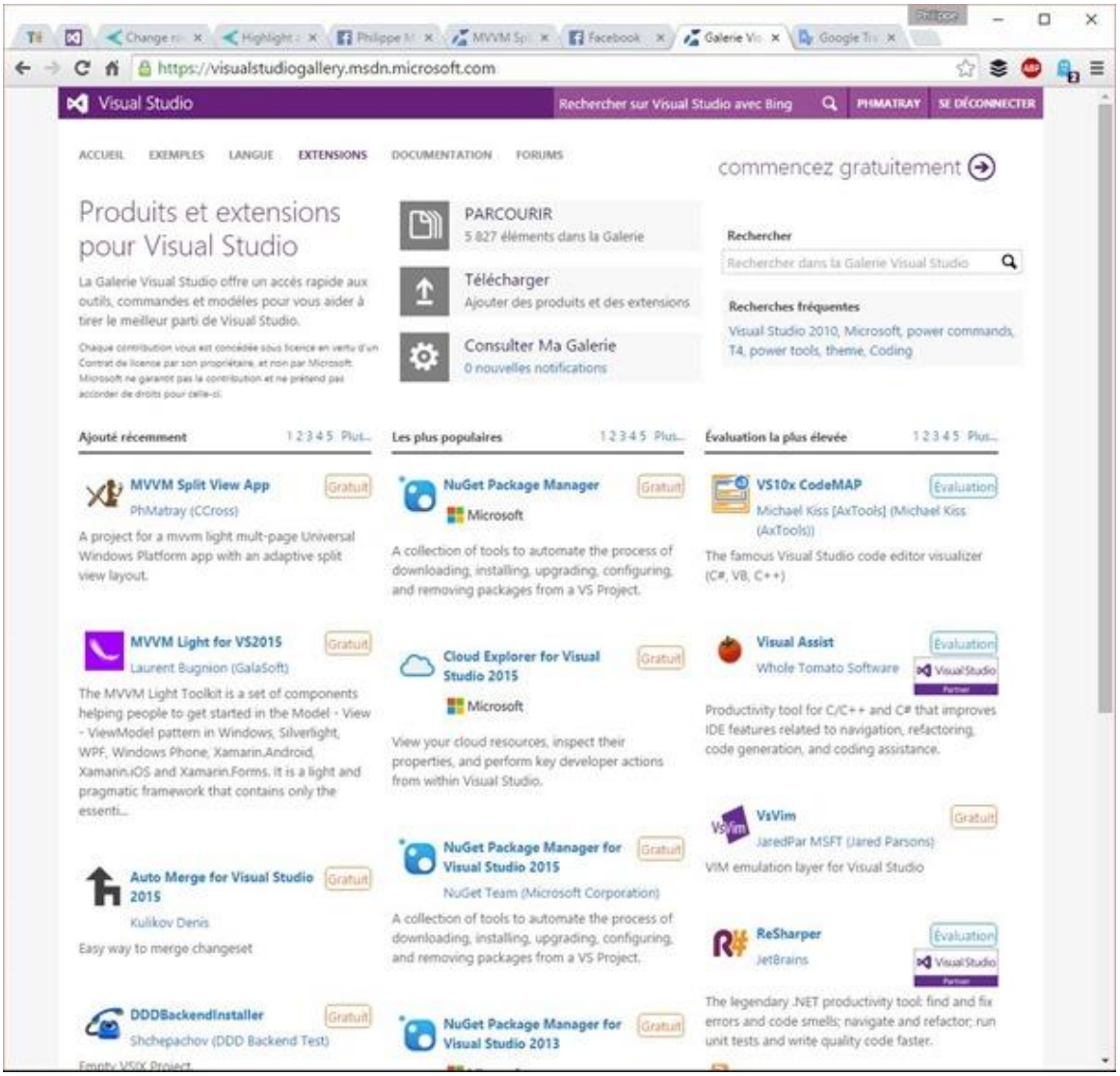
Un paquet VSIX peut être publié sur **Visual Studio Gallery**. Un bouton permettant l'upload est présent à partir de la page d'accueil.

https://visualstudiogallery.msdn.microsoft.com/

Un simple formulaire se présente à vous. Un fois votre paquet uploadé, la plupart des informations extraite de votre paquet seront renseignées.



Une fois uploadé, vous aurez la joie de constater que votre extension est directement accessible à la communauté à partir de la page d'accueil de la galerie.



Dans notre métier, développer n'est pas le seul élément plaisant. Le partage est lui aussi très enrichissant. Cela fait des années que je suis DotBlog et c'est en grande partie grâce à ses lignes que j'ai pu m'améliorer.

Bénéficier des efforts d'autrui gratuitement m'a incité à donner de mon temps à la communauté afin de rendre le travail de chacun plus agréable. VisualStudio Gallery symbolise parfaitement l'appartenance à une communauté.

Conclusion

Comme il a été vu dans un article précédent, créer un template est aisé et permet de simplifier le travail d'un développeur en lui faisant gagner du temps.

Ce gain de temps peut être étendu à toute une équipe de par la création d'un VSIX.

Enfin, le site Visual Studio Gallery permet d'offrir ce gain de temps à toute la communauté exploitant Visual Studio. N'hésitez-pas à proposer vos propres extensions, à en télécharger de nouvelles, à participer aux reviews de celles-ci. Bref faites vivre la communauté.

Bon dev et… comme dirait Olivier…

Stay Tuned !

[NdlR : je n'ai vraiment plus rien à ajouter ! Si : Merci Philippe ! Et merci d'avance à tous les lecteurs qui oseront se lancer comme lui !]

Utiliser le presse-papiers

Le presse-papiers a été la première évolution "collaborative" des OS. La première tentative d'améliorer la coopération entre applications. Si les 'Charmes' de Windows 8+ sont les lointains descendants de cette volonté de simplifier la recherche, le partage et les échanges de données, le presse-papiers n'est pas devenu obsolète pour autant ! Au contraire, c'est le B.A-BA d'une bonne UX…

Les API du presse-papiers

UWP, comme WinRT et ses milliers d'API ne pouvait pas oublier le presse-papiers… C'est pourquoi on retrouve une classe statique Clipboard dans l'espace de noms `Windows.ApplicationModel.DataTransfer`.

Cette API a sa propre philosophie et elle utilise notamment un objet messager, le DataPackage, comme vecteur des informations écrites dans le presse-papiers.

Mais regardons d'abord la classe `Clipboard` :

```
public static class Clipboard

{

    public static DataPackageView GetContent();

    public static void SetContent(DataPackage content);

    public static void Flush();

    public static void Clear();

    public static event EventHandler<object> ContentChanged;

}
```

`GetContent()` est la méthode qui bien entendu permet de récupérer le contenu actuel du presse-papiers (noté PP plus loin) et `SetContent()` celle qui permet d'écrire dans ce dernier.

La méthode `Clear()` vide le PP, `ContentChanged` est un événement qui est déclenché quand le contenu du PP change et `Flush()` est un peu plus subtile puisque cette méthode ajoute le `DataPackage` au PP tout en le libérant de son application source de telle façon à ce que le contenu partagé puisse être disponible même une fois cette dernière éteinte (supprimée de la mémoire).

La lecture et l'écriture dans le PP se font via des `DataPackage`.

Le PP de Windows 10 comme celui de ces prédécesseurs couvre bien plus que de simples données textes comme le faisait celui des premières versions de Windows… On peut donc y trouver des données de type plus vastes et correspondant mieux aux besoins des utilisateurs et des développeurs d'applications modernes.

Les types supportés vont ainsi de l'incontournable morceau de texte brut jusqu'aux fichiers en passant par les images et le texte mis en forme.

UWP s'est rangé du côté des standards, le texte formaté doit être écrit en HTML (et interprété comme tel quand il est lu depuis le PP) bien que RTF soit toujours soutenu.

Les principales utilisations

Les écritures

Placer du texte dans le PP est certainement l'opération la plus simple et la plus banale, et heureusement c'est la plus simple :

```
var dataPackage = new DataPackage();

dataPackage.SetText("blabla");

Clipboard.SetContent(dataPackage);
```

Le texte ainsi copié dans le PP est disponible au sein de l'application, des autres applications UAP mais aussi des applications standards en bureau classique (WPF par exemple).

Placer du texte formaté est à peine plus compliqué. On passe par une étape supplémentaire qui met en forme le texte HTML via le `HtmlFormatHelper` :

```
var dataPackage = new DataPackage();

var htmlContent = HtmlFormatHelper.CreateHtmlFormat("<b>blabla</b>");
```

```
    dataPackage.SetHtmlFormat(htmlContent);

    Clipboard.SetContent(dataPackage);
```

Placer une image dans le PP est tout aussi direct, du moment qu'on dispose d'une image en mémoire, sinon il faut passer par une URI.

```
    var dataPackage = new DataPackage();

    var storageFile = await StorageFile.GetFileFromApplicationUriAsync("Z:\soleil.jpg");

    dataPackage.SetBitmap(RandomAccessStreamReference.CreateFromFile(storageFile));

    Clipboard.SetContent(dataPackage);
```

Si le bitmap est déjà en mémoire on peut donc directement appeler le `SetBitmap()`.

Placer un fichier dans le PP suit un chemin à peine plus compliqué :

```
    var dataPackage = new DataPackage();

    var files = new List<StorageFile>();

    var storageFile = await StorageFile.GetFileFromApplicationUriAsync("X:\data.sql");

    files.Add(storageFile);

    dataPackage.SetStorageItems(files);

    Clipboard.SetContent(dataPackage);
```

Comme on le remarque ici l'API fonctionne en réalité sur une liste de fichiers. On peut comme dans l'exemple ci-dessus ne passer qu'un seul fichier ou répéter l'opération `GetFileFromApplicationUriAsync()` pour ajouter les fichiers à la `List<StorageFile>`.

On voit ici que toutes les API WinRT n'ont pas été réécrites pour UWP et qu'on retrouve le model asynchrone EAP et non TAP (voir les articles sur ce sujet).

Les lectures

La lecture du PP est extrêmement simple puisqu'il suffit d'écrire un code de ce genre :

```
var dataPackage = Clipboard.GetContent();
```

Il ne reste plus qu'à utiliser l'instance obtenue. Pour cela il est tout de même nécessaire de contrôler ce qu'elle contient…

le DataPackage propose une méthode `Contains()` qui permet de tester ce contenu en utilisant l'énumération `StandardDataFormat` qui offrait les valeurs suivantes sous WinRT : `Text`, `Html`, `Bitmap` et `StorageItems` auxquelles s'ajoutent `ApplicationLink`, `Uri` et `WebLink`.

Pour savoir si le PP contient du texte brut il suffit donc d'écrire :

```
if (dataPackage.Contains(StandardDataFormats.Text)) ...
```

On opère donc de la même manière pour savoir s'il s'agit d'une image, d'un fichier ou de Html. Charge à l'application de savoir quoi faire des données reçues.

Conclusion

L'API de gestion du presse-papiers n'est pas la plus exaltante d'UWP d'autant qu'elle emprunte beaucoup à celle de WinRT, c'est une évidence. *Mais elle devrait être couverte par toutes les applications, même les plus simples*. Vous savez à quel point je suis attaché au Design et à l'UX des applications. Le presse-papiers permet d'enfoncer le clou car souvent les choses les plus simples sont les plus mal supportées par les applications… Copier/Coller des données, *l'utilisateur s'attend au minimum à ce niveau zéro de l'UX qu'est le partage d'information via le PP*.

Bien entendu l'OS gère de base le Copier/Coller sur les zones de texte. Mais il ne le fait pas avec les autres types de données.

Et même en texte brut il peut être intelligent de proposer des modes de copie un peu avancés (comme copier le nom et l'adresse d'une fiche client en une seule fois au lieu d'obliger l'utilisateur qui veut récupérer ces données pour faire une lettre dans Word à copier chaque champ l'un après l'autre et jurer tout ce qu'il peut à l'encontre du @&@! de développeur qui n'a pas penser à une chose si simple…). Je ne parle pas non plus de la magie qui fera d'un simple utilisateur un fan inconditionnel et qui consiste à savoir découper les données d'autres applications de façon habile pour remplir automatiquement via un Coller toute une série de champs par exemple. C'est difficile à faire dans l'absolu mais dans le cadre d'un environnement logiciel balisé (comme c'est souvent le cas en entreprise) c'est déjà plus facile. Et là le presse-papiers peut se transformer en une arme de séduction redoutable…

Comment une Machine à états finis peut améliorer vos ViewModels et vos UI ?

Découpler la logique des états et transitions d'un ViewModel en faisant gérer les Commandes par une Machine à Etats Finis apporte un nouveau niveau d'abstraction aussi important que l'est MVVM lui-même. Êtes-vous prêt à gérer correctement le workflow de vos applications et en améliorer l'ergonomie ? …!

Le problème des commandes

Avant d'entrer dans les détails posons le problème :

Gérer des commandes en MVVM est très simple. Une commande n'est qu'une propriété de type `ICommand` du ViewModel que l'on attache (binding) à une propriété de même type d'un objet d'UI (type bouton par exemple). Lorsque l'utilisateur déclenche la commande via l'objet d'UI la propriété ICommand du ViewModel déclenche sa méthode `Execute()`.

Tout est simple et clair grâce à XAML, son binding et le pattern MVVM. Les librairies annexes comme MVVM Light simplifient encore plus la tâche et proposent en général un objet RelayCommand plus complet et plus pratique à utiliser que l'interface brute ICommand. L'implémentation d'origine a été créée par Josh Smith et on la retrouve dans de nombreuses librairies MVVM dont Prism sous ce nom ou un autre.

Mais tout cela ne change rien à la véritable question d'un logiciel bien programmé : **comment gérer proprement le CanExecute ?**

Je vois beaucoup de code dans lequel CanExecute n'est tout simplement pas géré… J'en conclue que beaucoup d'entre vous se diront "*mais pourquoi se prend-t-il la tête avec ce 'détail' ?*"

Parce que ce n'est pas un détail justement… **CanExecute reflète de façon assez fidèle** le workflow du ViewModel, ce qui est possible de faire ou non à un moment donné, donc **l'état de l'automate qu'est un logiciel**…

CanExecute est donc l'émanation programmatique de l'essence même de ce qu'est l'informatique : créer des automates, des machines de Turing. Comment cela pourrait-il être un simple "détail" ?

Plus important encore car c'est la partie visible pour l'utilisateur, la bonne gestion de CanExecute permet à l'UI de mieux le guider et donc d'améliorer l'ergonomie de vos applications.

Quand CanExecute n'est pas géré

Quand le développeur ne gère pas le CanExecute des commandes celles-ci peuvent donc s'exécuter à tout moment dans n'importe quel ordre quel que soit l'état de l'automate porte ouverte à tous les bogues. Autant dire que c'est le boxon pour être poli. A moins que le développeur ne barde en entrée les méthodes exécutées de tests divers et variés pour vérifier si le code peut ou non être exécuté justement. Ce n'est pas l'endroit où le faire, le CanExecute sert à cela… à moins de travailler par Aspect mais c'est un autre débat. *Pour l'utilisateur c'est forcément le chaos à l'arrivée après un chemin semé de doutes…*

Quand CanExecute est géré

Bien ! Vous gérez le CanExecute de vos commandes je vous en félicite ! Mais de ce que je constate le plus souvent, même dans ce cas où le développeur à fait son job, c'est qu'hélas les tests de CanExecute sont soit trop succincts soit se compliquent tellement qu'on en perd le fil. Quant à comprendre ce qui est vraiment autorisé ou non dans tel ou tel état et comment maintenir ces tests … il faudrait lire en un bloc tous les tests de tous les CanExecute et resynthétiser tout cela en un schéma. D'ailleurs quels sont les états d'un ViewModel donné ? Il est rarissime qu'un développeur envisage les choses sous cet angle et **des incohérences apparaissent vite entrainant une UX pénible pour l'utilisateur**. Le plus souvent on teste donc ce qui semble interdire une commande mais sans avoir fait l'effort de lister tous les états possibles du ViewModel… Bref même quand le CanExecute est géré on est loin de la perfection.

Il ne s'agit bien entendu pas d'académisme. La recherche d'une perfection illusoire, d'une orthodoxie tatillonne. Non, il s'agit bien d'une préoccupation majeure, un programme est une machine à états et **ne pas matérialiser clairement ces états c'est prendre de gros risques dont le plus grave de tous est de perdre l'utilisateur dans un dédale illogique**. Tout comme MVVM et le découplage fort permettent de minimiser les erreurs et les mélanges entre UI et logique, entre services et utilisateurs de services, gérer les états d'un ViewModel minimise un risque majeur, celui de pondre du code spaghetti au cœur même de la logique de l'application !

Comment gérer correctement CanExecute ?

La critique est facile, l'art est plus difficile, c'est connu. Mais vous me connaissez, plus je critique plus ma réponse est longue ! Donc comment gérer correctement CanExecute ? En gérant bien les états de l'automate qu'est le ViewModel. Certes. Et comment ?

En utilisant un outil adapté à cette situation : une **machine à états finis**.

Il y a de nombreux avantages à utiliser cette stratégie, notamment celui de créer un découplage entre d'une part la logique des états possibles et d'autre part les transitions entre ceux-ci ainsi que le code du ViewModel. Cette abstraction supplémentaire simplifie le ViewModel, le rend plus maintenable tout en rendant lisible, cohérent et facilement modifiable le workflow. Le ViewModel devient l'emplacement du code des actions, la machine à états finis le mode d'emploi des commandes. L'UI déjà découplée par MVVM tire profit de ce nouveau niveau de découplage de façon automatique grâce à la machine à états finis qui gère automatiquement le CanExecute() qui guide l'utilisateur dans sa progression. Développeur et utilisateur sont gagnants…

Qu'est-ce qu'une machine à états finis ?

Une *Finite State Machine*, ou Machine à Etats Finis modélise le comportement séquentiel d'un objet. Une telle machine possède donc un nombre fini d'états et ne peut avoir qu'un seul état à la fois. Les machines hiérarchiques autorisent la notion de sous-états et sont très utilisées car plus conformes à la complexité des mécanismes modélisés en général (l'héritage de la configuration de l'état maitre par un sous-état simplifie son paramétrage et donc celui de toute la machine).

On notera qu'on appelle Automate Fini ce type de machine mais j'utilise ici la traduction littérale de l'anglais "Machine à états finis" dans le sens d'un code spécifique (une librairie) gérant les états d'une partie d'application afin de le différencier de l'Automate Fini qu'est l'application elle-même prise comme un tout. Cette nuance est artificielle et n'a de sens que dans le contexte de cet article afin de séparer les deux choses.

Une telle machine se définie donc par :

- Des **états** généralement nommés par un adjectif indiquant une action en cours ("en train de", le "ing" à la fin d'un verbe anglais), par exemple : Listening, Waiting, Editing, Printing, etc… (on peut le faire en français aussi bien que je conseille par cohérence avec le langage de rester toujours en anglais dans les noms de champs, méthodes etc, sinon on obtient un franglais abominable je trouve).
- Des **évènements** qui sont des *conditions extérieures* comme une action utilisateur. Ces évènements sont eux aussi nommés avec des conventions diverses mais indiquant une action comme Save, Refresh ou Print ou autre

dans cet esprit. Ces évènements peuvent déclencher une sortie de l'*état courant* vers un nouvel état dit *final*.

Au final l'association de l'**état courant**, de l'**état final**, de l'**évènement** et de l'**action** forme ce qu'on appelle une***transition***.

On retrouve cette notion de machine à états finis dans toute l'informatique puisqu'elle en est la base même et c'est sans surprise que le langage de modélisation UML propose un diagramme états-transitions dans le groupe des diagrammes comportementaux par exemple. Mais sans aller chercher loin, tout le monde connait les logigrammes ou ordinogrammes qui servent à décrire un algorithme à l'aide de boites, de flèches et de losanges de prise de décision. Sans être tout à fait identiques ces diagrammes "primitifs" représentent eux aussi en quelque sorte des états et des transitions d'un automate.

Stateless State Machine

Stateless, "sans état", est un code créé par Nicholas Blumhardt qui permet facilement de coder une machine à états finis hiérarchique. Le nom est étrange mais on peut certainement l'expliquer par le fait que justement c'est cette machine qui gère les états et non plus code principal de l'application qui devient alors "sans états". Mais bon c'est une supposition et cela reste tiré par les cheveux ! Heureusement ce n'est pas le nom qui nous intéresse mais ce que fait "Stateless".

Stateless se présente sous la forme de son code original ou bien sous une forme adaptée à .NET 4.0. On trouve les deux packages dans Nuget, Stateless-4.0 étant le plus récent et celui que nous utiliserons.

Code minimaliste d'une machine à états finis

Pour mieux comprendre Stateless commençons par coder "à la main" une machine à états finis vraiment minimale. Elle ne connait que 2 états, On et Off, qu'un seul déclencheur dans la méthode Transition, la commande "on" (le reste étant compris comme la commande "off") :

```
public class BasicStateMachine
    {
        public string State { get; private set; }

         public void Transition(string state)
        {
            switch (state)
            {
```

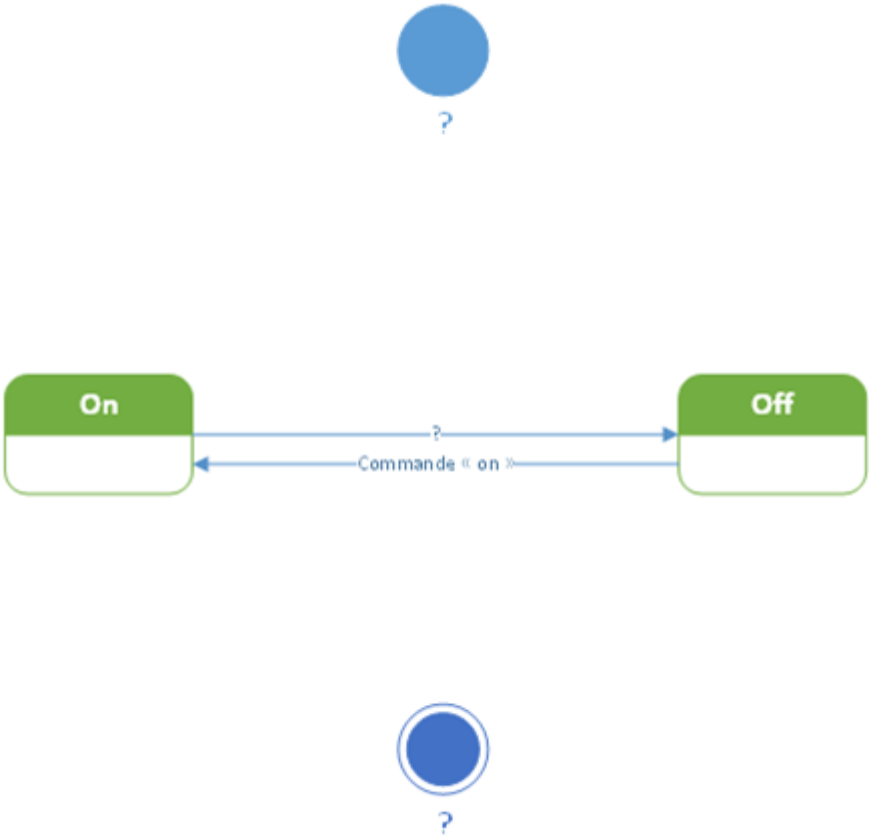
```
                case "on":
                    State = "ON";
                    OnSwitchedOn();
                    break;
                default:
                    State = "OFF";
                    OnSwitchedOff();
                    break;
            }
        }


        private void OnSwitchedOn()
        {
            // Do something
        }



        private void OnSwitchedOff()
        {
            // Stop doing something
        }
    }
```

Ce code est réellement direct et déjà contient les germes de la problématique des commandes, sujet principal de cet article ne l'oublions pas même si je suis obligé de vous présenter tout cela avant d'y arriver !

En effet, un choix a été fait ici dans l'implémentation, le switch bascule vers `On` dans le cas où la commande "`on`" est envoyée et renvoie vers l'état `Off` dans tous les autres cas. C'est un choix qu'il faut assumer. La machine doit-elle s'éteindre si je lui commande "`toto`" ou *uniquement* si je lui commande "`off`" ? un autre aspect est négligé : dans quel état se trouve l'automate au départ ?

Sans diagramme on se rend compte que même dans un cas aussi simple les problèmes se posent. Alors imaginez un peu ce qui se passe dans le code réel d'un ViewModel un peu complexe où cet aspect n'est pas même géré !!!

Schématisons un peu le code ci-dessus :



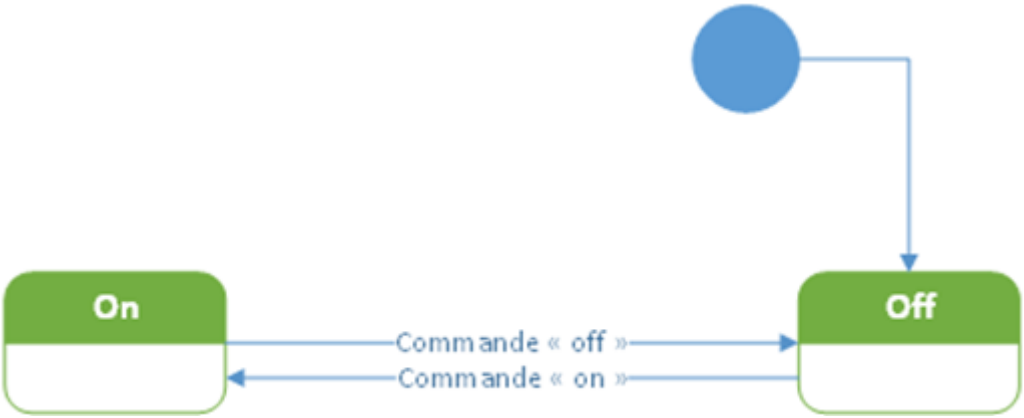
L'état initial n'est pas précisé, première erreur qui devient évidente quand on la représente. L'état final aussi n'est pas précisé, la machine est donc à fonctionnement infini, il n'y pas de sortie, ce qui n'est pas une erreur mais un choix.

Enfin on note les deux états On et Off et les transitions. La première va de On vers Off et n'est pas définie, n'importe quel évènement fera donc basculer vers Off. La seconde va de Off à On et ne fonctionne que sur l'envoi de la commande "on" ce qui est très restrictif mais qui peut s'admettre.

Bref, une fois le diagramme posé on s'aperçoit des manques et incohérences.

On notera au passage que je parle de "commandes" pour passer d'un état à l'autre ce n'est pas une terminologie tout à fait exacte (on parle d'évènements ou de déclencheurs) mais cela est à mettre en rapport avec le sujet de l'article qui concerne les commandes qui sont bien à la source des évènements qui causent les transitions. Pour approfondir toute la rigueur de la notation UML des diagrammes d'états transitions je renvoie le lecteur intéressé à la nombreuse littérature technique sur le sujet.

Lorsqu'on modélise la même machine même d'instinct on fait mieux car on voit les états, les transitions, ce qui donne plutôt cela :



Comme on visualise le diagramme on s'aperçoit qu'il faut bien un état initial, et ici il est à "Off". Ensuite s'il faut une commande "on" pour passer à "On", il faut aussi pour être rigoureux une commande "off" pour passer à "Off". Ce n'est pas innocent comme choix…

Si n'importe quoi fait basculer à Off comme dans le code de départ, cela peut être voulu, une machine outil par exemple est dangereuse, dans la précipitation d'un accident l'opérateur peut par affolement appuyer n'importe où et il est préférable dans ce contexte que cela passe à "Off", mieux vaut arrêter la machine pour rien qu'arracher un bras… Si maintenant c'est un système de circulation extracorporelle utilisé durant les pontages coronariens (qui remplace le cœur pendant qu'on le "démonte"), il est préférable que la commande "Off" soit au contraire hyper verrouillée avec même une clé physique à tourner, un code à saisir, etc.

> *Il est vraiment essentiel de comprendre que toutes ces petites choses apparemment de l'ordre du détail ont en réalité un impact gigantesque sur l'application et sa raison d'être, sa finalité, ses contraintes.*

Il n'y a pas d'état final dans l'exemple corrigé, c'est une machine à états finis mais à fonctionnement infini, pourquoi pas. On pourrait en revanche pousser plus loin le diagramme en indiquant un état "Error" si une commande différente de "on" ou "off" est envoyée ou bien représenter une boucle sur les états "On" et "Off" qui indiquerait que toute commande différente de celle attendue fait rester la machine dans son état courant.

> *Là encore rien n'est de l'ordre du détail, les questions soulevées par ces deux boites, ce point et ces deux flèches vont creuser très loin dans l'analyse fonctionnelle du logiciel, bien plus loin qu'on ne l'imagine au départ. Les diagrammes en général, même de SGBD*

sont des outils essentiels en cela que la présence ou non d'une flèche, d'une orientation, etc, aussi insignifiant que cela paraisse soulèvent des questions d'une grande profondeur dans la compréhension du mécanisme modélisé, questions qui sont rarement posées autrement.

Nous n'irons pas trop loin dans la modélisation de notre exemple et regardons simplement l'impact du diagramme corrigé succinctement sur le code :

```
public class BasicStateMachine
    {
        const string offState = "OFF";
        const string onState = "ON";
        private string state = offState;
        public string State {
                              get { return state;}
                              private set { state=value;}
                             }



        public void Transition(string state)
        {
            switch (state.ToUpper())
            {
                case "ON":
                    State = onState;
                    OnSwitchedOn();
                    break;
            case "OFF":
                    State = offState;
                    OnSwitchedOff();
                    break;
            }
        }


        private void OnSwitchedOn()
        {
            // Do something
        }

        private void OnSwitchedOff()
        {
            // Stop doing something
        }
    }
```

La machine ainsi codée possède un état initial, “Off”, et ne bascule de l’un à l’autre que si les commandes “on” et “off”, peu importe la casse, sont envoyées. Toute autre commande laisse la machine en l’état.

Au final nous avons un code plus rigoureux mais s’il fallait représenter une véritable machine à états finis et la faire évoluer il faut avouer que cela deviendrait vite fastidieux avec une telle programmation.

Stateless pour coder une machine

Avec **Stateless** la même machine se codera de la façon suivante :

```
public class StatelessStateMachine
    {
        private readonly StateMachine _stateMachine;


        public enum Trigger
        {
            TurnOn,
            TurnOff
        }

        public enum State
        {
            On,
            Off
        }

        public State Current { get { return _stateMachine.State; } }

        public StatelessStateMachine()
        {
            _stateMachine = new StateMachine(State.Off);
            _stateMachine.Configure(State.Off)
                .Permit(Trigger.TurnOn, State.On)
                .OnEntry(OnSwitchedOff);

            _stateMachine.Configure(State.On)
                .Permit(Trigger.TurnOff, State.Off)
                .OnEntry(OnSwitchedOn);
        }

        public bool Transition(Trigger trigger)
        {
            if (!_stateMachine.CanFire(trigger))
                return false;


            _stateMachine.Fire(trigger);
            return true;
```

```
        }


        private void OnSwitchedOn()
        {
            // Do something clever
        }

        private void OnSwitchedOff()
        {
            // Stop doing something clever
        }
    }
```

C'est un peu long pour si peu vous direz-vous et c'est normal, notre machine est tellement bête et simple que le code pour l'exprimer est forcément long par rapport à son utilité réelle, ce n'est qu'une sorte de Hello Word pour Stateless…

Mais si vous lisez attentivement ce code vous vous apercevrez rapidement qu'il est bien plus clair et bien plus structuré. Et surtout qu'il est bien plus souple, chaque état est défini, chaque évènement aussi, et l'API de type "fluent" qui permet de définir tout cela est aussi simple que puissante.

Modifier un état, une transition, ajouter des actions d'entrées et de sorties à une transition, etc, tout cela devient limpide.

Stateless utilise des méthodes génériques et le type des états ou des triggers est totalement libre. Ici on utilise des énumérations mais des objets plus complexes sont utilisables.

Et le rapport avec les commandes en MVVM ?

C'est en fait tout le sujet de l'article.

Maintenant vous avez une idée de ce qu'est une machine à états finis et de ce qu'est la librairie Stateless et comment elle s'utilise (au moins vous en avez une idée).

C'était un préambule nécessaire. Le "véritable" article commence ici.

"La continuité c'est maintenant !" (Enfin un slogan politique qui ne ment pas !).

Retour à la problématique des commandes

Revenons sur le problème que posent les commandes. On a bien compris qu'elles se fondent sur l'interfaceICommand (même si on utilise un RelayCommand) et que les méthodes principales sont `Execute()` qui exécute l'action, `CanExecute()` qui teste si

la commande peut s'exécuter et `CanExecuteChanged()` un évènement qui permet à l'UI de savoir si `CanExecute()` à changé de valeur.

L'intérêt d'une classe comme `RelayCommand` est de proposer non pas une interface qu'il faut implémenter à chaque fois mais une classe dont on peut créer des instances immédiatement ce qui est bien plus pratique. Autre apport de `RelayCommand`, la méthode `RaiseCanExecuteChanged()` qui déclenche l'évènement `CanExecuteChanged()`. En effet, si les possibilités d'exécution changent et bien qu'il existe un évènement *ad hoc* auquel s'abonner dans `ICommand` rien ne permet dans cette interface de forcer cet évènement et donc de prévenir l'UI que "l'exécutabilité" de la commande vient de changer.

Je n'irai pas trop loin sur ce point bien qu'essentiel car j'ai déjà écrit de très nombreux articles et livres sur MVVM et notamment sur la façon de gérer les commandes. Le lecteur intéressé saura retrouver tout cela sur Dot.Blog et dans la collection "All Dot.Blog" j'en suis certain.

Comme toujours l'enfer se loge dans les fameux détails. Le vrai chalenge n'est pas dans l'exécution de la commande, le code derrière `Execute()`, mais **dans le fait de s'assurer qu'il s'exécute quand le ViewModel est dans un état valide pour l'action donnée**.

Typiquement le code du `CanExecute()`, quand il est géré, est implémenté sous la forme d'expressions conditionnelles qui utilisent des variables locales de type `"isLoading" "isPrinting" "CanDoxxx" "objectMachin!=null"` etc…

Chaque nouvel état nécessite de coder des conditions supplémentaires qui nécessitent une évaluation, de nouvelles variables locales, etc, ce qui très rapidement devient d'une ***complexité impossible à maitriser***.

Sous XAML on dispose d'un outil qui permet de gérer les états visuels, le Visual State Manager, c'est une aide précieuse pour clarifier les choses. Mais si l'UI est découplée du code, ses états reposent sur ceux du code. Il ne peut y avoir une représentation de l'état "attente" dans le visuel XAML s'il n'y a pas un état correspondant dans le ViewModel. Tout tient donc sur la bonne gestion des états de la machine à états finis qui permet de construire un code solide et cohérent aussi bien en C# que côté UI en XAML. Une raison de plus d'adopter l'approche que je vous propose dans le présent article !

Le rôle de la machine à états finis

La machine à états finis va permettre de modélisé tous les états du ViewModel, donc ce qui est autorisé ou non de faire selon l'état en cours.

En construisant des commandes à partir du mécanisme de la machine et en laissant cette dernière décider des actions et de la gestion du `CanExecute` il sera même possible d'automatiser les réactions de l'UI puisque les objets supportant `ICommand` savent réagir au `CanExecute` en modifiant leur aspect visuel (qui peut de plus être retravaillé grâce à la grande souplesse de XAML).

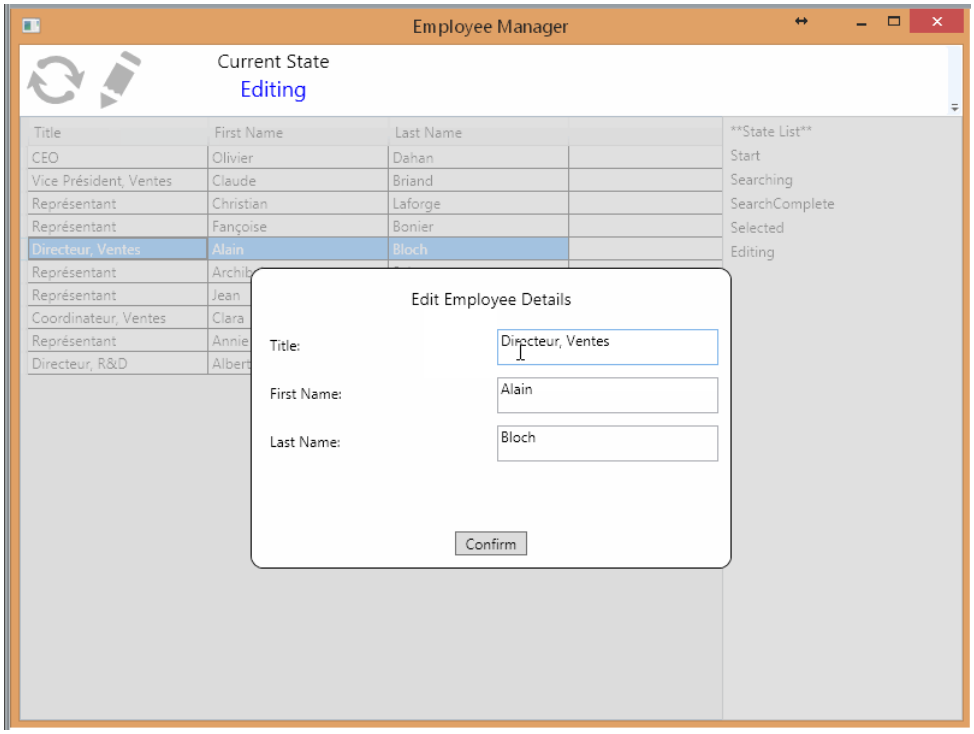
On créé ainsi un nouveau découpage intéressant : l'UI d'une part dont le rôle ne change pas, le ViewModel qui devient un magasin de code (les actions) et d'autre part la machine à états finis qui joue le chef d'orchestre afin de garantir la cohérence de l'état du ViewModel et de l'UI.

Un exemple simple

Pour tenter d'être concret nous allons partir d'un exemple simple. Une pseudo gestion du personnel qui permet de faire des recherches, d'afficher la liste des personnes filtrées et bien entendu la modification d'une fiche sélectionnée. Pour rendre tout cela encore plus évident et éviter d'avoir trop de code parasite le mode de recherche se limitera à faire un refresh de la liste et simulera un temps d'attente (pour voir l'état visuel de l'application dans son mode "busy").

Afin de comprendre les changements d'états et bien que cela sera absent d'une application réelle, nous allons adjoindre l'affichage de l'état courant ainsi qu'une liste qui va se remplir au fur et à mesure de tous les états qui se succèdent.

Regardons pour commencer l'animation ci-dessous, je la commenterai ensuite (faites un clic sur l'image pour visualiser l'animation sur le site Dot.Blog) :



En gros nous avons une application qui propose une barre de commande contenant un bouton de recherche (limité ici à un resfresh simulant une attente pour l'accès aux données) et un bouton d'édition pour modifier une fiche salarié.

Sous cette barre se trouve une liste des employés. On peut sélectionner une ligne et cliquer sur le bouton d'édition.

Cela amène un dialogue de modification qui grise le fond. De même les commandes de refresh et d'édition sont grisées et interdites. Seule la commande de confirmation de saisie fonctionne (par simplification il n'y a pas de bouton annuler ici).

Pour les besoins de la démonstration j'ai ajouté dans la barre de commande un indicateur d'état courant de la machine à états. Il est directement bindé à la propriété State de la machine. J'ai aussi ajouté une liste à droite qui montre l'évolution dans le temps des états de la machine. Certains états sont évident car stables et longs (comme Start tant qu'on ne fait rien ou Selected tant qu'une fiche est sélectionnée) et d'autres plus fugaces comme SearchCompleted. Certains états comme Searching ou Editing entraînent des modifications majeures de l'UI, l'apparition du cadre de saisie pour Editing ou celle de la petite horloge animée pour matérialiser l'attente de Searching.

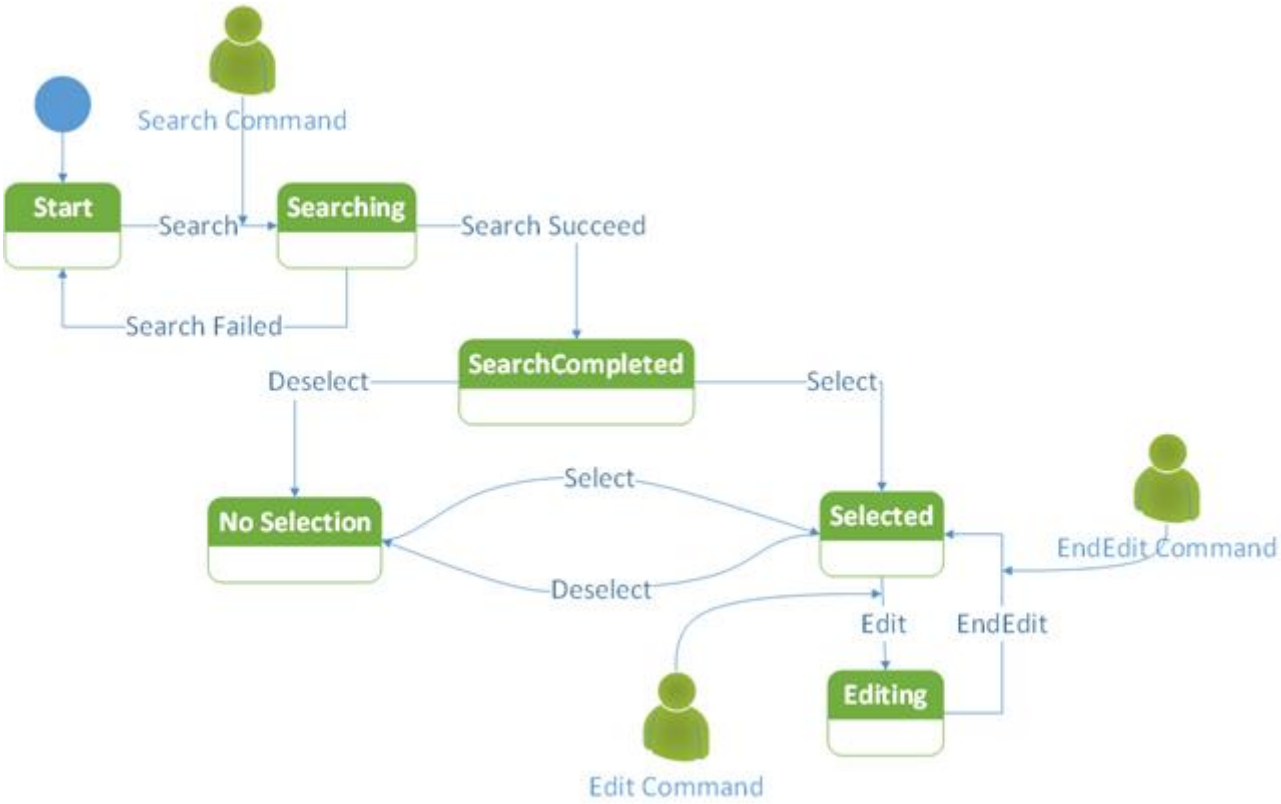
En réalité cette démonstration ne comporte aucun code pour gérer ces changements de l'UI qui sont liés directement à l'état `CanExecute` des commandes ou à l'état de la machine…

Pourtant tout est clair, sans faille possible car tout est codifié dans la machine à états !

Schématiser les états

Avant de coder la machine à états, plutôt la configurer puisque nous utilisons ici un code existant simulant une machine à états finis, il est absolument nécessaire de modéliser par un diagramme, même simple, les différents états, les évènements qu'il faut prendre en compte et les actions à entreprendre.

Je ne vais pas entrer dans le formalisme UML et sa rigueur, mais je vais utiliser un diagramme de type Etats-Transitions UML sans trop forcer sur l'académisme, l'essentiel étant de comprendre que ce schéma est indispensable même si vous le faite au crayon sur une nappe en papier au resto…



Les états de la machine sont représentés par des rectangles portant un nom blanc sur vert (`Start`, `Searching`, `SearchCompleted`…).

Les évènements ou triggers qui permettent de passer d'un état à l'autre sont matérialisés par des flèches directionnelles et sont accompagnées du nom de l'évènement (`Search` qui va de l'état `Start` à l'état `Searching` par exemple).

Les interventions de l'utilisateur au travers de l'UI sont matérialisées par un bonhomme vert pointant l'évènement concerné du diagramme. Il est accompagné du nom de la commande telle qu'elle est vue par lui ("Search Command" par exemple).

On notera que les évènements peuvent être déclenchés par l'utilisateur ou par le changement d'état d'autres automates dans l'application. Par exemple Search Succeed et Search Failed (réussite ou échec de la recherche) sont déclenchés par la partie du programme qui effectue la recherche des données.

Se rappeler : Le programme est un utilisateur comme les autres.

Comme je le disais on ne pinaillera pas sur la beauté académique du diagramme lui-même, la notation UML est plus rigoureuse et un peu plus difficile à lire que mon schéma pour celui qui ne connait pas sa syntaxe particulière. De même l'état Start n'est sémantiquement pas très futé, le démarrage est symbolisé par le cercle plein c'est lui le vrai "start", il doit pointer un état considéré comme initial (auquel on a le droit de revenir), on aurait donc pu appeler cette état Idle (au repos) plutôt que Start. Mais ne nous encombrons pas de tout cela qui est totalement accessoire ici.

En revanche nous aurions pu aller plus loin pour représenter les sous-états, ceux qui héritent de leur parent. La syntaxe UML aurait encore plus compliqué la lisibilité mais tentons malgré tout, en pensée, de visualiser ces sous-états s'ils existent :

Par exemple l'état SearchCompleted, lorsque la recherche est terminée n'est qu'un sous-état de l'état Start (qu'on aurait pu appeler Idle, "au repos" ce qui prend tout son sens ici).

De même finalement Selected est un sous-état de SearchCompleted. Il ne peut y avoir de sélection (ou de non sélection) que si la recherche est terminée.

On doit représenter cette notion de sous-états dans le diagramme car le paramétrage de la machine est plus simple et il est plus fidèle à la réalité. Certains états ne peuvent exister que si la machine est passée par un état parent. Sortir de ces sous-états est généralement limité à ce que le parent autorise.

Paramétrer la machine à états finis

Comme je l'ai expliqué au début de cet article en présentant les machines à états finis nous n'avons heureusement pas à coder nous-mêmes la machine, nous allons utiliser un code tout fait, un package Nuget dont nous avons déjà vu comment il se programme.

Mais il reste à adapter tout cela à notre exemple.

Une fois le schéma établi coder les états et les triggers ou évènements n'est pas compliqué. On pourrait utiliser des objets complexes pour les représenter, ici nous choisirons le plus simple : des énumérations.

Ainsi états et triggers seront codés de cette façon :

```
public enum States
{
  Start, Searching, SearchComplete, Selected, NoSelection, Editing
}

public enum Triggers
{
  Search, SearchFailed, SearchSucceeded, Select, DeSelect, Edit, EndEdit
}
```

Une fois que les états et triggers sont posés il ne reste plus qu'à configurer la machine.

Celle-ci est créée comme héritant de `StateMachine<State,Trigger>` et elle supportera au passage INPC pour récupérer correctement les changements d'états quand ils interviennent (ce qui permet de construire la liste des états de la démo notamment).

Ce type dont nous allons hériter est bien entendu fourni par Stateless. Et comme je le disais états et triggers sont des types génériques qui nous permettent d'utiliser ce que nous voulons.

Le code complet de la machine utilisée dans la démo est ainsi :

```
public class StateMachine : StateMachine<States, Triggers>, INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        public StateMachine(Action searchAction)
            : base(States.Start)
        {
            Configure(States.Start)
              .Permit(Triggers.Search, States.Searching);


            Configure(States.Searching)
              .OnEntry(searchAction)
              .Permit(Triggers.SearchSucceeded, States.SearchComplete)
              .Permit(Triggers.SearchFailed, States.Start)
              .Ignore(Triggers.Select)
              .Ignore(Triggers.DeSelect);
```

```
        Configure(States.SearchComplete)
          .SubstateOf(States.Start)
          .Permit(Triggers.Select, States.Selected)
          .Permit(Triggers.DeSelect, States.NoSelection);

         Configure(States.Selected)
          .SubstateOf(States.SearchComplete)
          .Permit(Triggers.DeSelect, States.NoSelection)
          .Permit(Triggers.Edit, States.Editing)
          .Ignore(Triggers.Select);


        Configure(States.NoSelection)
          .SubstateOf(States.SearchComplete)
          .Permit(Triggers.Select, States.Selected)
          .Ignore(Triggers.DeSelect);

        Configure(States.Editing)
          .Permit(Triggers.EndEdit, States.Selected);


        OnTransitioned
          (
            t =>
            {
                onPropertyChanged("State");
                CommandManager.InvalidateRequerySuggested();
            }
          );

        //used to debug commands and UI components

        OnTransitioned
          (
            t => Debug.WriteLine
              (
                "State Machine transitioned from {0} -> {1} [{2}]",
                t.Source, t.Destination, t.Trigger
              )
          );
    }

     private void onPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }

}
```

On note la présence de INPC qui n'est pas forcément indispensable ainsi que la surcharge de `OnTransitioned` qui est ajoutée pour lister les transitions dans la console de débogue. Toutefois ces "extensions" au système de paramétrage des états rendent Stateless très souple et véritablement attrayant.

La programmation de l'automate passe par une API dite "fluent", c'est à dire dont les méthodes retournent un objet modifié ce qui permet d'enchainer les appels de façon très proche d'une description fonctionnelle. La "vraie" syntaxe de LINQ marche de cette façon (celle qui utilise des méthodes et non celle qui ressemble à du SQL).

Les méthodes de la machine

Les principales méthodes disponibles sont :

- `Permit(TTrigger,TState)` qui autorise un état à faire une transition vers l'état cible en passant par le trigger;
- `Ignore(Ttrigger)` qui bloque une transition depuis l'état quand le trigger indiqué se déclenche;
- `SubstateOf(TState)` qui permet de créer un lien enfant / parent entre un état celui indiqué;
- `OnEntry(Action)` qui permet d'enregistrer une action exécutée à l'entrée de l'état;
- `OnExit(Action)` qui dans l'autre sens définit une action exécutée juste avant la sortie de l'état.

La machine passe en erreur lorsqu'un trigger est déclenché dans un état non valide ou non configuré.

Prenons en détails la configuration de l'état Searching :

```
Configure(States.Searching)
  .OnEntry(searchAction)
  .Permit(Triggers.SearchSucceeded, States.SearchComplete)
  .Permit(Triggers.SearchFailed, States.Start)
  .Ignore(Triggers.Select)
  .Ignore(Triggers.DeSelect);
```

La méthode `Configure(TState)` indique le début de la configuration de l'état passé en paramètre, ici `States.Searching`. L'appel à `OnEntry` déclenche l'action "`searchAction`" qui accède à la base de données.

Dans cet état deux seules transitions sont possibles, elles sont matérialisées par les deux méthodes `Permit`. Le premier paramètre est le trigger, le second est l'état d'arrivée. Pour l'état `Searching` ici configuré cela signifie qu'il accepte les triggers `SeachSucceeded` et `SearchFailed` qui le font transiter respectivement vers `SearchComplete` ou vers l'état d'attente `Start`.

En revanche les deux méthodes `Ignore()` sont utilisées ici pour pallier un problème de la plupart des listes WPF qui déclenchent leur `Select` lorsque la liste est bindée à la source. Nous ne voulons pas de ces transitions et les interdisons. Plus exactement elles seront donc ignorées. Ce problème ne se pose pas dans toutes les moutures de XAML et l'interdiction des états doit être comprise ici comme un illustration des possibilités de Stateless uniquement. Sous UWP il reste à vérifier la pertinence de ce qui est vrai sous WPF (qui a servi de base à cet article paru avant la sortie officielle de UWP et dont je ne pouvais donc pas parler et encore faire voir les outils de développement).

La machine propose aussi deux autres méthodes qui sont utilisées dans le ViewModel principalement :

- `void Fire(TTriger)` qui déclenche la transition associée à l'état en cours, au trigger et à l'état d'arrivé configuré. C'est comme cela que le ViewModel pilote la machine à états.
- `bool CanFire(TTriger)` qui retourne True si le trigger indiqué est autorisé dans l'état en cours.

Binder la machine aux commandes MVVM

La machine à états finis "Stateless" n'est pas conçue spécialement pour fonctionner avec MVVM ou n'importe quoi d'autre. C'est une machine à états finis, c'est tout. Elle peut servir à moult choses bien différentes. Il n'y a donc aucune relation "naturelle" entre "Stateless" et MVVM ou même avec le système de commandes `ICommand` de WPF ou UWP. *De même n'oubliez pas que les commandes ne sont ici qu'un prétexte pour parler tout cela, il n'y a pas que ICommand qui nous intéresse, la démo le montre (par la fonction Edit ou l'horloge d'attente), la machine est utile bien au-delà des commandes.*

Pourtant dans une application suivant MVVM nous avons besoin de déclarer dans le ViewModel des commandes `ICommand` qui seront bindées à des objets de l'UI, soit directement s'ils le supportent, soit en utilisant l'astuce d'un Behavior `EventToCommand` qui transforme un évènement en déclencheur d'une commande `ICommand`.

Une telle commande possède donc au minimum un `Execute()`, cela n'est pas compliqué à fournir, et un `CanExecute()` dont nous avons vu toute l'importance.

Tout l'objet de cet article et de la technique démontrée est justement d'éviter d'écrire le code de `CanExecute()` pour en confier la gestion à une machine à états finis. Or "Stateless" n'est pas conçu pour cela "out of the box".

Qu'à cela ne tienne ! Ecrivons une méthode d'extension qui permettra d'ajouter une machine la capacité de créer des commandes reliées à son fonctionnement interne :

```
public static ICommand CreateCommand<TState, TTrigger>(
  this StateMachine<TState, TTrigger> stateMachine, TTrigger trigger)
    {
      return new RelayCommand
        (
          () => stateMachine.Fire(trigger),
          () => stateMachine.CanFire(trigger)
        );
    }
```

Une nouvelle commande de type `RelayCommand` est créée en utilisant comme action le `Fire()` du trigger et le `CanFire()` de ce dernier. Nous obtenons bien une commande reliée à l'état interne de la machine.

Et c'est de cette façon que les commandes du ViewModel sont créées :

```
SearchCommand = StateMachine.CreateCommand(Triggers.Search);

EditCommand = StateMachine.CreateCommand(Triggers.Edit);

EndEditCommand = StateMachine.CreateCommand(Triggers.EndEdit);
```

Connexion à l'UI

Selon le modèle MVVM le ViewModel expose les commandes (ci-dessus) et l'UI se binde sur celles-ci. C'est ce qui se passe dans la barre de commande de notre application démo :

```
<Button ToolTip="Search" VerticalAlignment="Center"

  Style="{StaticResource ButtonStyle}"

  Command="{Binding SearchCommand}">
```

```
  <Image Source="Images\Search.png"></Image>

</Button>
```

Le lien est donc fait entre UI et Commande, et entre Commande et machine à états finis… Ce qui par transitivité relie l'UI à la machine à états donc.

De fait les boutons seront désactivés ou activés automatiquement suivant si l'état courant de la machine autorise ou non les triggers correspondants ! Zéro code ici, tout s'enchaine parfaitement.

Notre démo montre d'autres connexions entre UI et machine à états que les commandes. Par exemple vous avez du remarquer dans l'animation plus haut que lorsque le ViewModel est occupé à la recherche des données une petite horloge animée s'affiche à la place de la liste des résultats.

Ici pas de `ICommand` à connecter… Comment s'opère la magie ?

En ajoutant un convertisseur de valeur qui va traduire l'état courant de la machine en un booléen. Et pas n'importe comment, juste en passant à True si l'état est "busy" donc lorsque la machine est en mode recherche de données.

Une fois ce convertisseur créé on peut ajouter une couche à l'affichage de notre application, couche possédant la petite horloge d'attente et qui sera visible ou non selon l'état de la machine :

```
public class StateMachineVisibilityConverter : IValueConverter
  {
    public object Convert(object value, Type targetType,
      object parameter, CultureInfo culture)
    {
      string state = value != null ?
        value.ToString() : String.Empty;
      string targetState = parameter.ToString();
      return state == targetState ?
        Visibility.Visible : Visibility.Collapsed;
    }
 }
```

Le code du convertisseur est bien entendu un peu plus générique que ce que je viens d'en dire car cela serait dommage d'avoir à se répéter pour chaque état qu'on souhaiterait tester (DRY !). Il est donc conçu pour recevoir en paramètre l'état à tester. De même il ne retourne pas un booléen mais quelque chose de directement exploitable par XAML, une valeur de type `Visibility`.

L'horloge d'attente est ainsi bindée à l'état de la machine en XAML en utilisant le convertisseur avec le paramètre "Searching" qui est le nom de l'état qui doit afficher une attente :

```
<local:AnimatedGIFControl Visibility="{Binding StateMachine.State,

                          Converter={StaticResource StateMachineConverter},

                          ConverterParameter=Searching}"/>
```

"`StateMachine`" est le nom de la propriété de notre ViewModel qui expose la machine à états finis à l'extérieur. Et "`State`" est l'état courant d'une machine "Stateless". Grâce au convertisseur et à son paramètre la visibilité de l'horloge est désormais directement et automatiquement contrôlé par l'état de recherche en cours de la machine !

Pas de code superflu, pas de répétition de tests éparpillés partout et difficiles à maintenir, rien d'autre qu'une machine à états finis qui sert de point central pour contrôler tous les états du ViewModel et leur incidence sur les états visuels de l'UI… C'est beau non ?

Le même principe est appliqué à une autre grille contenant un fond gris et la fenêtre d'édition… Quand l'état bascule à Editing, cette grille ayant l'avant-plan dans le Z-Order devient visible, cachant tout le reste et interdisant au passage le clic sur ce qui est en dessous… Les boutons de la barre de commande se grisent eux aussi mais cela par le biais de `CanExecute()` déjà relié à la machine.

Conclusion

Il faut bien terminer un jour… Cela fait trois jours que je suis sur cet article l'expression prend donc ici toute sa signification !

J'espère que vous avez compris le principe et surtout les intérêts nombreux qu'il y a à suivre ce pattern de l'Automate Fini en l'appliquant à MVVM et XAML.

Non seulement les états du ViewModel deviennent clairs et maintenables, cohérents et inviolables, mais cette approche force à penser le workflow, les états et transitions du ViewModel très tôt dans le développement de celui-ci. C'est un gain énorme qui permet de lever des loups avant même de coder, de s'apercevoir que le cahier des charges ne prévoit rien dans telle ou telle situation par exemple. Car noyée dans une prose rébarbative une analyse cache souvent beaucoup de choses !

Le code exemple complet est téléchargeable en fin d'article, amusez-vous bien ! (car développer correctement avec les bonnes méthodes est un plaisir).

Le sujet est vaste alors voici quelques références intéressantes :

Tout d'abord Tarquin Vaughan-Scott à qui j'ai emprunté une partie de l'article et du code de démo qu'il avait publié dans un article en 2014 dans MSDN qui m'avait séduit et dont je voulais absolument vous parler. Mieux vaut tard que jamais ! Et j'ai réussi à faire mieux que de vous en parler, en tout cas je l'espère, en écrivant ce long article qui enrichit beaucoup l'original et le rend plus accessible à ceux d'entre-vous qui ne sont pas des fans de l'anglais…

Cet article de Bob Nystrom (en anglais) qui présente assez bien le concept de machines à états.

Josh Smith et son RelayCommand repris par MVVM Light et d'autres et qui est utilisé dans le code de démo (qui ne fait usage d'aucun framework MVVM juste du nécessaire).

MSDN et son article de présentation du système de commande de WPF.

L'article de Laurent Bugnion, créateur de MVVM Light, présentant RelayCommand et EventToCommand.

Prism pour WPF et l'implémentation de MVVM avec ce framework.

Tom Anderson et sa rapide introduction à "Stateless" dont est issu le premier exemple de l'article.

On pourrait certainement encore allonger cette liste car le sujet balaye large dès qu'on parle de MVVM, de son système de commande et lorsqu'on y ajoute les automates finis ! Mais j'ajouterai forcément la collection All.Dot.Blog dont certains tomes sont particulièrement orientés XAML ou MVVM.

Bref, il y a de quoi lire. Et comme je le dis souvent, "*si je le connais, c'est que je l'ai lu quelque part*". Croire qu'on peut générer de la connaissance pure tout seul dans son coin est soit être mégalo soit inconscient de l'importance de la communication et du partage de la connaissance. Nous ne sommes rien sans le savoir des autres. Il n'y a qu'en faisant circuler le savoir qu'on peut aider à l'émergence du progrès… Il faudrait milles vies pour tout lire sur tout, lisez déjà Dot.Blog ça pourra vous servir 

Débloquer une device pour le debug devient (enfin) un jeu d'enfant

Le vieux système de compte pour débloquer une device pour faire du debug est mort. Tout devient simple et rapide, enfin !

Déboguer sur une machine réelle était compliqué

Avec Windows Phone 7 puis les OS suivants, développer sur une machine réelle, principalement pour le débogue et les tests de performance, était un véritable enfer. J'ai pesté contre ce procédé qui obligeait à s'enregistrer, obtenir des codes d'activation, des bricolages tellement complexes que même un acharné comme moi, et après un mois passé avec le support technique de Microsoft car ça ne marchait pas avait fini par abandonner… J'ai recommencé beaucoup plus tard et ça a finalement marché. Mais c'était trop tard pour le projet que je voulais présenter à un prospect…

Que j'ai pu maudire les idiots qui avaient pensé un tel montage aberrant là où avec Android par exemple il suffisait d'activer le mode débogue par USB sur une machine pour tout de suite la connecter au PC et tester ses développements…

Mais ça c'était avant !

Tout cela datait de l'époque maudite où les commandes du navire étaient entre de bien mauvaises mains. Mais heureusement tout cela a changé et il faut rendre grâce à Nadella d'avoir totalement stoppé ces délires complexes, fermés et obtus. J'aime l'ouverture d'esprit du nouveau CEO de Microsoft et j'aime les effets de son approche pragmatique qui se manifestent de jour en jour.

Car pour activer une machine pour le débogue… Toute la procédure ubuesque inventée par des maniaques a été simplement supprimée !

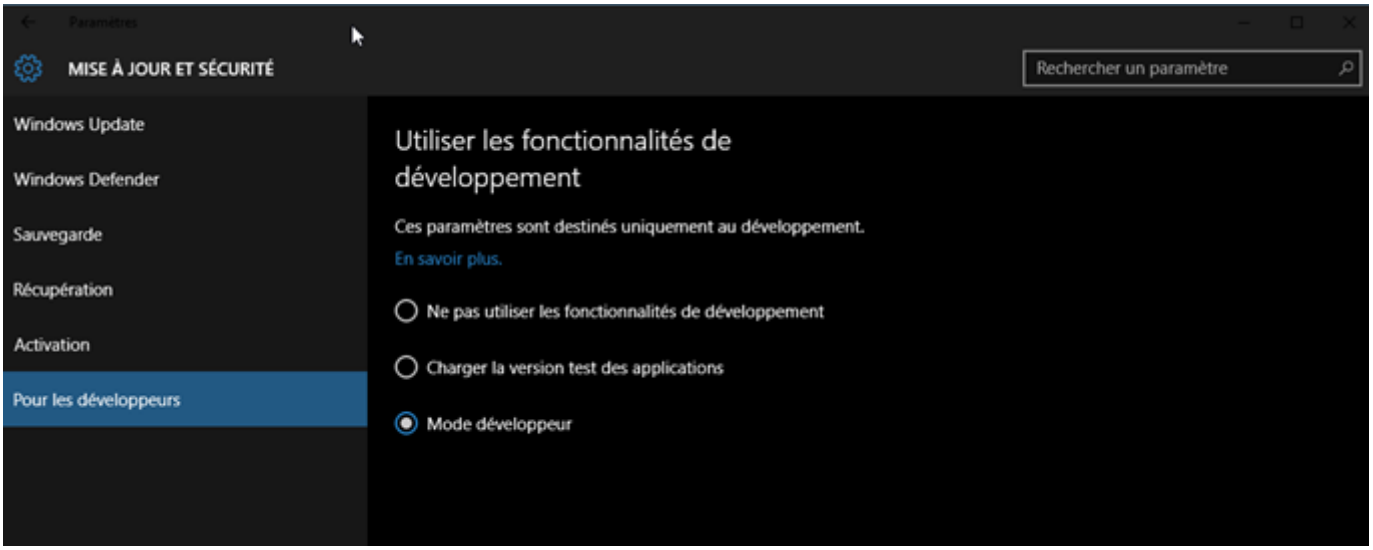
Un mode débogue immédiat, comme les autres…

Car en effet, preuve de ces changements de mentalité, Windows 10 propose désormais un mode développeur avec débogue par USB, je dirais "comme tout le monde" quoi…

Mais mieux vaut tard que jamais, on connait de grandes sociétés qui ne sont jamais revenues sur leurs décisions stupides d'enferment comme Apple pour ne citer qu'un exemple cher à mon cœur…

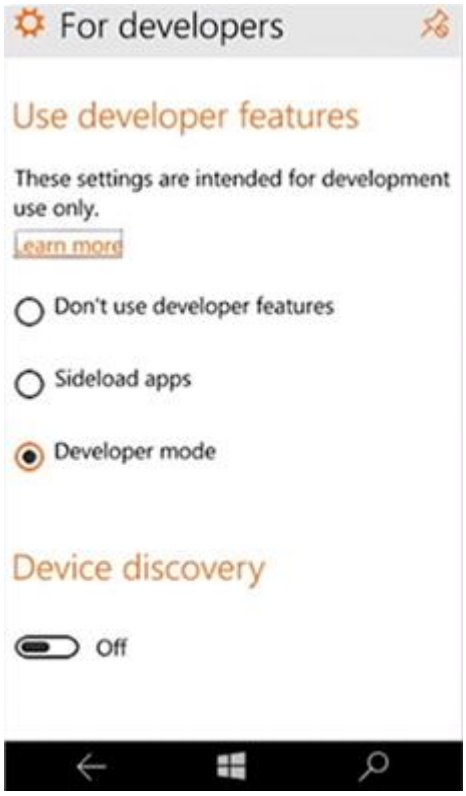
Comment faire ?

Il suffit d'accéder aux options "Mise à jour et Sécurité" du menu Paramètres et de cocher en toute simplicité le mode "Mode développeur".



Simple et efficace…

Sur un smartphone on obtient un menu assez proche :



Conclusion

Il en aura fallu du temps perdu pour en venir à une chose essentielle : faire simple pour le débogue sur machine réelle si on veut attirer les développeurs (ou ne pas faire fuir ceux qui sont là et de bonne volonté !)… L'intelligence est de nouveau aux commandes chez Microsoft, oubliez les déboires temporaires de l'ubuesque et nocive

ère Sinofsky, maintenant tout marche comme on s'y attend, logiquement, naturellement. Merci M. Nadella de restaurer enfin la santé mentale de l'entreprise, de penser ouverture plus que fermeture et simplicité plus qu'usine à gaz.

Templates de projets (blank et Compositor)

Pour l'instant le seul template de projet UWP fourni avec VS 2015 est le modèle "blank", vide donc. Je vous ai récemment proposé un template pour Mvvm Light, mais on peut trouver d'autres templates…

Des templates "pas vides"

Partir d'un template vide c'est pratique car on maitrise tout le code facilement, on doit tout faire !

Mais partir d'un modèle plus complet permet de gagner du temps… C'est pourquoi dernièrement je vous ai proposé un template Mvvm Light à installer pour créer de nouvelles Apps UWP avec ce framework en attendant que lui aussi fournisse ses habituels templates.

Mais peut-on trouver d'autres templates ?

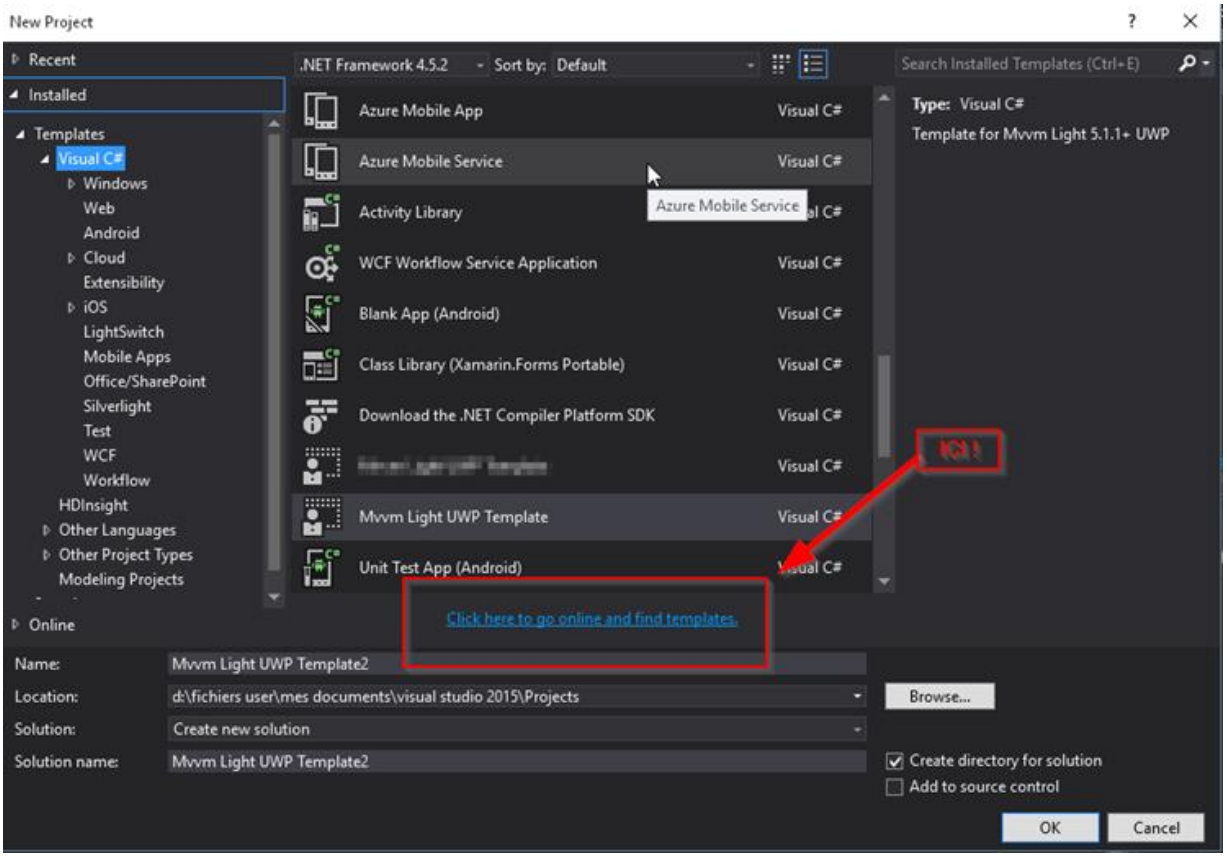
La réponse est oui vous le devinez car sinon il n'y aurait pas d'article… Mais où ?

Charger des templates UWP

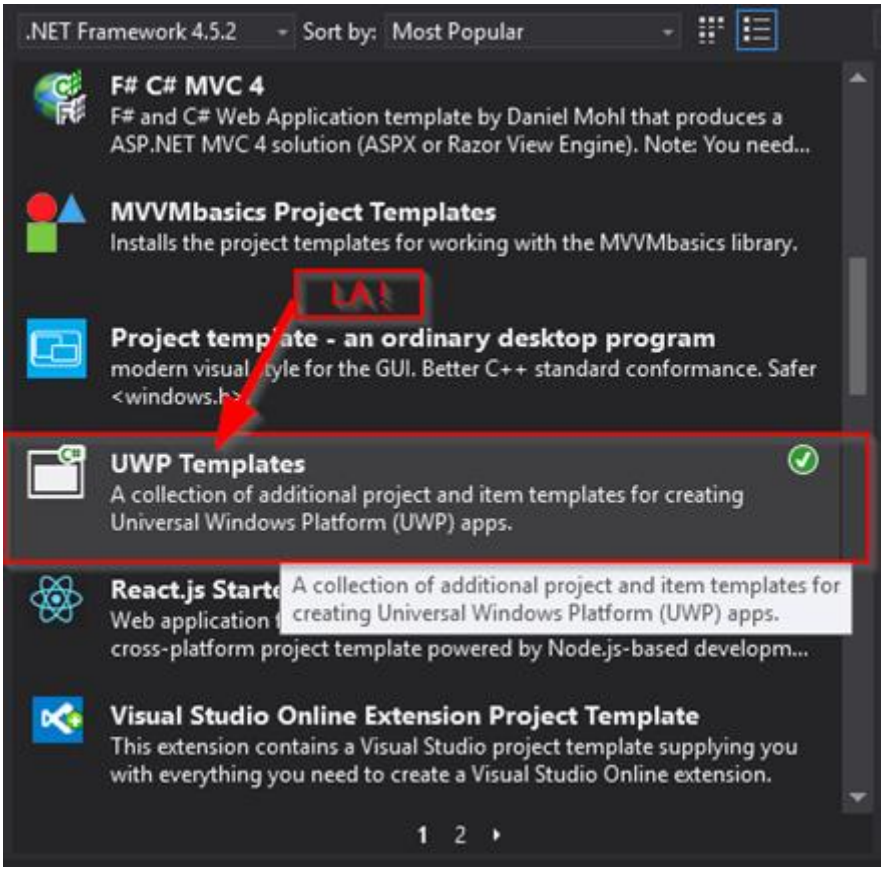
Il n'y a pas profusion de choix pour le moment et en attendant que Microsoft propose ses propres templates (on l'espère, tout comme Mvvm Light d'ailleurs !) il y a tout de même une possibilité.

Cliquez sur Fichier / Nouveau / Projet dans le menu de VS2015. Comme si vous vouliez créer un projet donc.

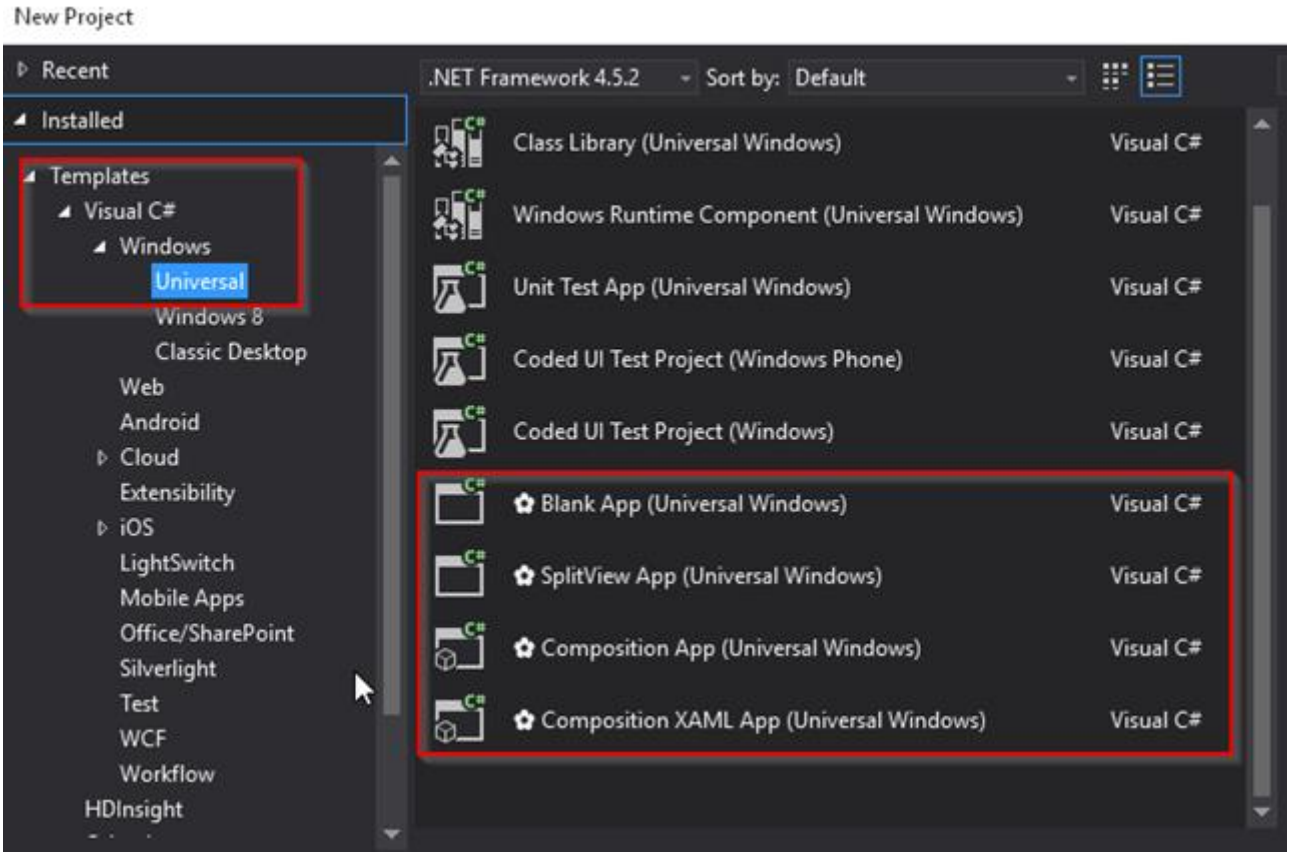
Mais regarder bien sous la liste des templates existants :



Cliquez sur le lien puis cherchez le template suivant :



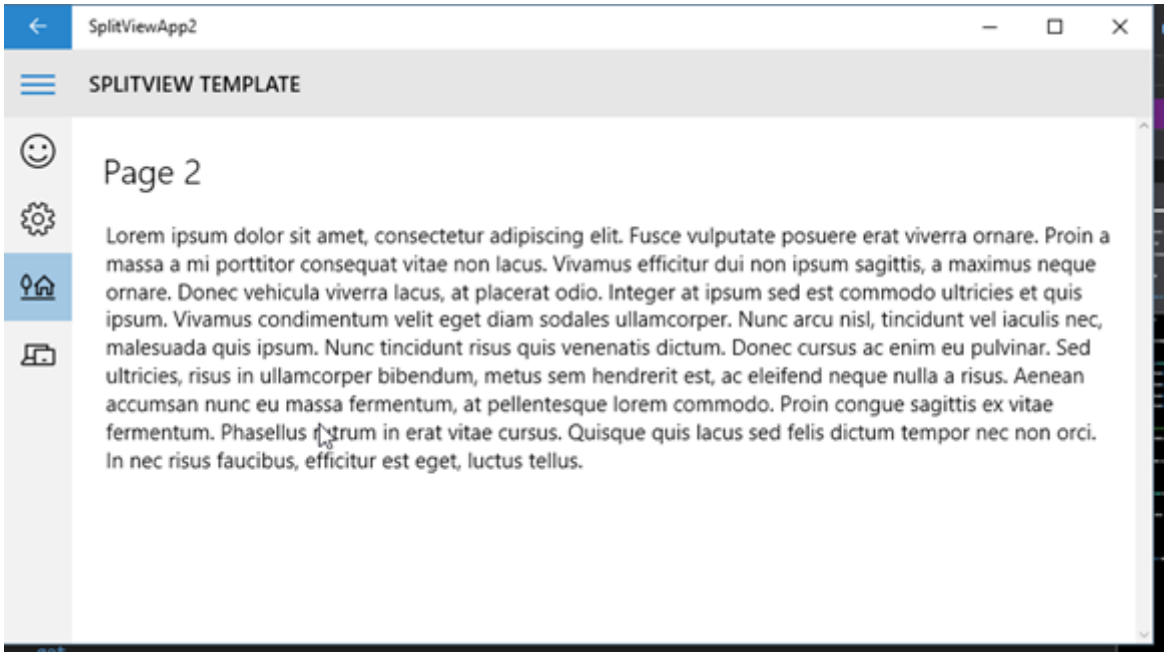
Installez cette série de templates. Sortez du dialogue une fois que c'est fait. Revenez sur Fichier / Nouveau / Projets puis dans Windows / Universal vous découvrirez la liste suivante :



Outre une nouvelle "Blank App" vous disposez maintenant d'une SplitView App, d'une Composition App et d'uneComposition XAML App.

La Blank App est comment dire … Blank. Donc comme celle de base. Passons.

La SplitView App propose un look & feel moderne et adaptatif multipage qui ressemble à cela :



Plusieurs pages exemples sont créées avec des icônes et le menu s'agrandit ou disparait ne laissant que l'icône hamburger visible selon que l'écran est large ou petit. C'est donc un bon modèle pour créer des applications adaptative au look & feel UWP.

La Surprise …

Le modèle Composition App utilise une vue créée entièrement avec le **Compositor**. Alors que la version XAML propose une mise en page XAML avec titre et une partie centrale utilisant le Compositor.

Qu'est-ce que le Compositor me direz-vous… La documentation MSDN est assez laconique mais il s'agit d'un conteneur un peu spécial capable d'effets visuels et d'animations. Le mieux est de l'essayer pour comprendre. Mais où trouver un exemple ?

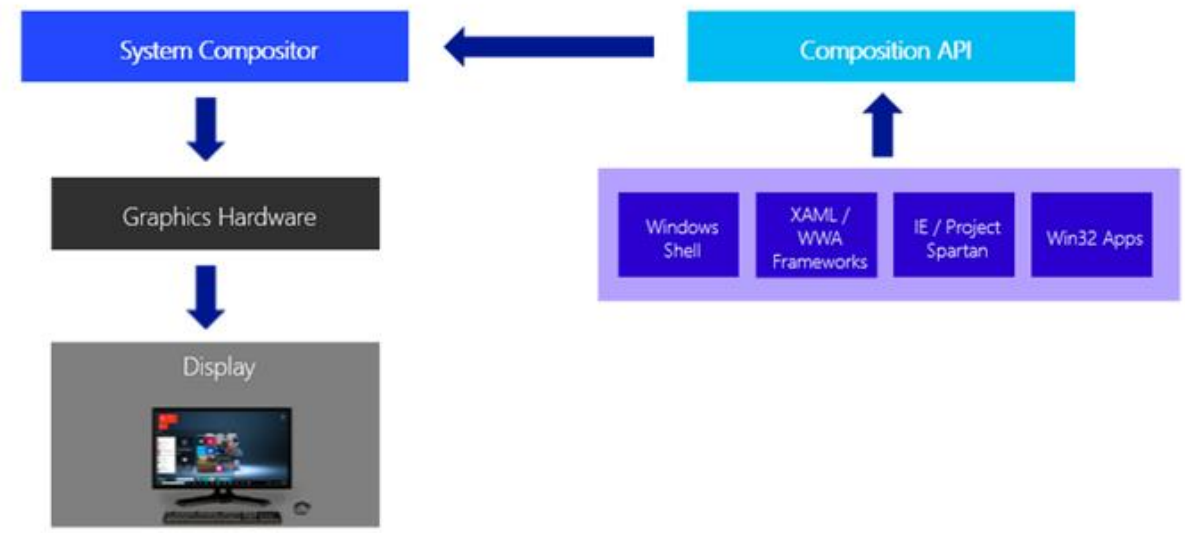
Grâce à mon flair de Pif le chien j'ai trouvé un projet GitHub officiel Microsoft qui montre quelques utilisations de la chose. Téléchargez et essayez, c'est intéressant. Et c'est ici :

https://github.com/Microsoft/composition

Si la doc est si peu bavarde c'est que le namespace lui-même (Windows.UI.Composition) est toujours dans un état de preview, c'est une nouveauté qui peut encore varier un peu d'ici sa diffusion finale. Je pense que nous sommes peu nombreux (enfin si maintenant avec vous !) à en connaitre l'existence.

Mais en cherchant un peu (sur la page en lien plus haut) on trouve aussi des émissions Channel 9 qui montrent un peu l'esprit de la bête qui est *le moteur de*

composition de Windows 10 sur lequel se greffe le reste. En accédant directement à cette API il est possible de faire des choses nouvelles et attrayantes visuellement… J'en reparlerai puisque c'est le **moteur unifié de rendu d'UWP**. Disposer d'un accès à ce dernier est génial car les effets visuels deviennent portables et ultra fluides sur toutes les cibles avec un seul code (à la différence d'animations trop lourdes en XAML ou le besoin de coder des pixels shaders peu portables).



Mais ce n'est pas le sujet d'aujourd'hui c'est juste une petite digression vous mettre en appétit !

Conclusion

La sérendipité est une merveilleuse pourvoyeuse de joies inattendues (par principe)… Vous croyez venir pour trouver des templates, mais au passage vous découvrez un monstre, rien de moins qu'une nouvelle API donnant accès au moteur de rendu graphique de UWP, donc totalement portable d'un smartphone à un PC et tous les zinzins qui supportent ou supporteront UWP comme la XBox… Incroyable non ?

VS 2015 : Astuce d'affichage (scroll en mode map)

VS 2015 est plein de surprises… certaines étaient même dans la version précédente mais on était passé à côté ! Le mode map permet de changer la scollbar sous C# ou XAML et d'avoir ainsi une visualisation miniature du code pour s'y positionner encore plus vite. Un must.

Scrollbar dans l'éditeur de code C# et XAML

Tout le monde sait ce qu'est une scrollbar, et tout le monde a bien compris qu'on dispose forcément d'une telle barre de défilement verticale lorsqu'on édite son code,

que cela soit du C# ou du XAML (ou autre d'ailleurs). Et tout le monde sait à quel point cela est tout aussi fondamentalement utile que trivial au point qu'on se demande bien ce qu'on pourrait en dire …

C'est sans compter sur l'ingéniosité de l'équipe Visual Studio !

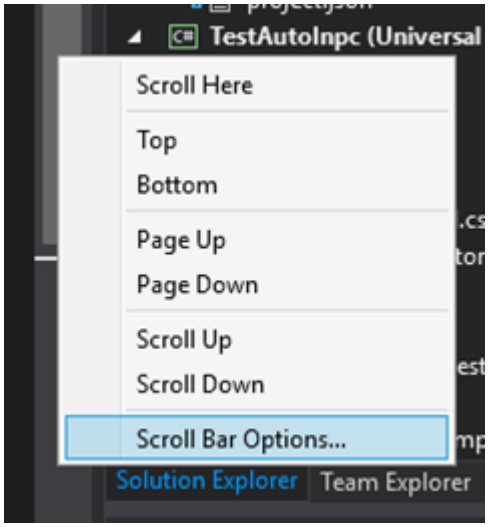
Un monde d'options cachées

Je vais vous faire découvrir un monde d'options dont vous ne soupçonniez peut-être pas l'existence…

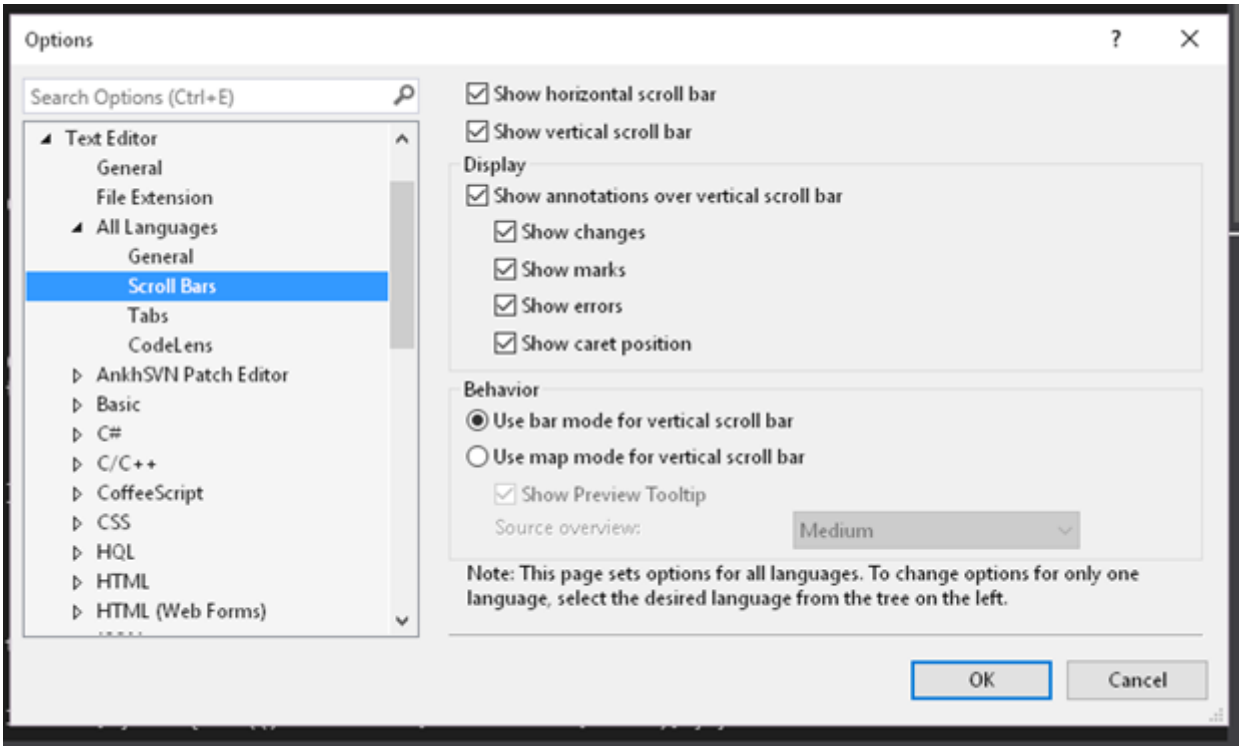
Car franchement qui aurait idée, sauf erreur de manipulation, de faire un clic-droit sur une scrollbar ? Qu'attendre de plus d'une telle barre à part qu'elle … scrolle ?

On manque parfois d'imagination mais pas les ingénieurs de l'équipe VS !

Faites un clic-droit sur la scrollbar verticale de l'éditeur de code (C#, XAML …) et vous verrez apparaitre un petit menu local :

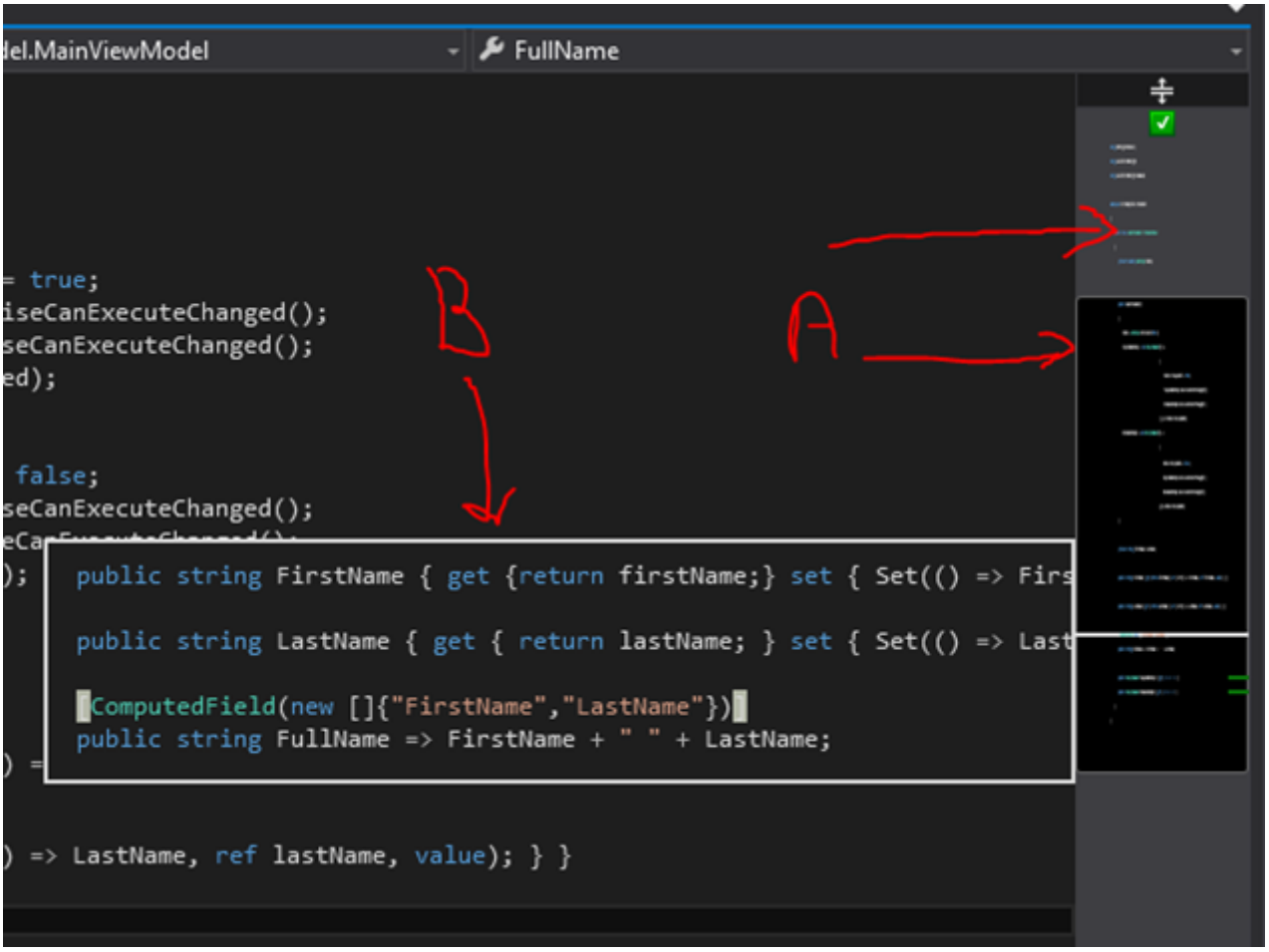


Il est déjà bien sympathique de voir qu'il y a quelques options intéressantes permettant de se déplacer très vite. Mais ce n'est pas tout. Tout en bas il y a les *Scroll Bar Options*. Cliquons dessus comme semblent nous y inviter les 3 points de suspension…



My Gosh ! C'est carrément une application de configuration qu'on découvre ici !

Et pour ce qui nous intéresse, regardons les options "All Languages / Scroll bars" qu'on peut régler aussi langage par langage si on le désire (dans ce cas on choisit d'abord ce dernier dans la liste à gauche). Dans la partie de droite, moitié inférieure sous le nom de "Behavior" (comportement) on trouve deux boutons radio, en activant "Use map mode ..." nous allons remplacer la scrollbar d'origine par une vue miniature de tout le code dans laquelle la navigation sera encore plus facile. Il est même possible de régler la taille (médium par défaut). Ce qui donne :



En “A” j’ai pointé la nouvelle scrollbar. Plus large (médium ici) elle est bien faite d’une “piste” et d’un “thumb” qu’on peut bouger. C’est toujours une scrollbar.

Toutefois elle montre la totalité du code et il devient beaucoup plus facile de s’y déplacer. On peut même cliquer sur un endroit précis pour s’y rendre.

Mais ce n’est pas tout, il est possible aussi de configurer l’apparition d’un tooltip “B” lorsqu’on laisse la souris un petit laps de temps sur la scroll bar en mode Map. Et là nous avons un agrandissement nous permettent de voir exactement quel code se trouve à cet endroit. Encore plus facile pour se déplacer !

Cela fonctionne avec C#, avec XAML, etc. Comme vous l’avez vu la boite de configuration est assez “joufflue” il y a donc matière à explorer…

Conclusion

Visual Studio est un EDI qui m’a séduit il y a longtemps. J’ai toujours été bluffé par sa qualité, sa robustesse et par les options intelligentes qu’il offrait. VS 2015 malgré des années d’aménagement ne déroge pas à la règle et continue de nous offrir toujours plus d’options.

Parfois certaines comme je le disais étaient présentent dans la version précédente mais on les avait zappées… C'est en réalité le cas du mode Map de la scrollbar qui marche aussi en VS 2013…

Comme quoi il faut cliquer partout et sur tout comme un gosse, c'est pour ça qu'ils savent si vite se servir d'une nouveauté technologique. Ils n'ont pas le poids de l'habitude qui fait que notre cerveau élague pour gagner du temps. Et on en gagne c'est vrai. Un adulte est plus productif qu'un enfant. Mais en perdant ce regard d'enfant on passe à côté de plein de choses.

Au passage cette option nous donne donc une leçon de vie, gardons notre regard d'enfant pour être plus créatifs et mieux nous adapter au changement… Et cliquez partout !

Que savez-vous de S.O.L.I.D. ?

SOLID, un principe invoqué dans de nombreux articles, dans les discussions autour de la machine à café, élevé par certain au rang saint Graal. Mais vous, qu'en connaissez-vous de SOLID ? Que seriez-vous capable d'en dire d'un peu … solide dans un entretien par exemple ? Sous UWP comme ailleurs, connaître et respecter SOLID c'est le début d'un meilleur code !

Un sigle flou, des concepts clairs

Avouons-le, de but en blanc si on vous demande de définir SOLID, d'éclater le sigle, je pari que nombre d'entre vous resterons un peu secs. Ne vous en faites pas notre métier est plein de sigles tous plus essentiels les uns que les autres et on n'en comprend que la moitié. Tout le problème c'est que si on tombe sur un fan on passe pour un inculte…

Heureusement Dot.Blog est là !

D'abord précisons que si SOLID est un sigle flou, les concepts qu'il promeut sont souvent déjà compris par la majorité des développeurs. Mais de comprendre à appliquer il y a souvent un monde. Quant à expliquer c'est encore tout un voyage !

Un but principal : rendre le code plus facile à lire et à maintenir.

Mais par quels moyens ? Que représente chaque lettre de SOLID ? Que cache chacune, qui une fois révélée, ouvre la voie vers plus d'intelligence dans le code ?

Ce sont ces concepts que je vais tenter de clarifier plus loin dans cet article.

Une dernière question : n'avez-vous jamais eu l'impression terrible en maintenant un code de jouer au Mikado ? On transpire en essayant de tirer ce petit bâton de bois sans faire tomber les autres… parfois ça marche, et parfois… Patatras ! Ou bien, plus vicieux, tout semble se passer bien on lâche la pression on pose à côté de soi le bâtonnet et d'un seul coup sans que personne ne touche à rien le tas s'affaisse et on a perdu !

Le développement ne devrait jamais être une partie de Mikado, c'est cela que nous dit SOLID !

S.O.L.I.D.

Chaque lettre de ce sigle renvoie à un concept amenant à une architecture du code de meilleure qualité. Encore faut-il entendre chacun de ces concepts derrière le sigle !

Le sigle lui-même a été forgé par C.Martin dans les années 1990 comme un moyen mnémotechnique de grouper plusieurs impératifs d'architecture menant d'un code lourd, peu aisé à maintenir vers un code à couplage faible, opérant de façon cohérente et encapsulant les besoins réels de l'entreprise de façon appropriée.

A la base on retrouve bien entendu tout ce que l'on sait sur le couplage fort ou faible, les grands principes de Programmation Object, etc. Mais tout cela est épars et Martin les a groupés pour leur donner un nouveau sens, en faire un tout lui-même cohérent de règles partageant toutes un même but à atteindre.

S comme SRP

Single Responsabilité Principle. Le *principe de responsabilité unique*.

Une classe doit absolument se concentrer sur une tâche, un but, ne prendre en charge qu'une seule responsabilité. C'est essentiel.

Prenons l'exemple d'une classe "Client" telle qu'on peut la rencontrer sous ce nom ou un autre dans de nombreux logiciels que j'ai pu auditer :

```
public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}


    public int Insert()
    {
        try
        {
            // insertion en base de données
            // retourne l'ID
```

```
            return 0;
        } catch (Exception e)

        {
            System.IO.File.WriteAllText(@"c:\temp\log.txt",e.ToString());
            return -1;
        }
    }
}
```

Quel(s) problème(s) pose cette classe ?

Elle viole une fois SRP et on peut la soupçonner de réitérer ce crime peut-être deux fois…

La première fois est évidente : dans le bloc “catch” la classe écrit dans un log. Ce n’est définitivement pas de sa responsabilité de s’occuper d’écrire des choses dans un fichier de log.

La seconde fois est soupçonnée puisque le code est juste suggéré : c’est l’accès à la base de données. S’il y a insertion via ADO.NET ou autre avec du code SQL par exemple, ou l’enregistrement dans un fichier XML ou encore une autre technique de ce genre il y aura bien une seconde violation de SRP.

Mais pas de procès d’intention, concentrons-nous sur ce qui est sûr : la violation de SRP dans le ‘’”catch”.

Quelle est la responsabilité de la classe Client ? De stocker des informations sur un Client. Pas d’écrire des fichiers de log.

Comment réparer cela ?

En respectant SRP qui ici nous impose de créer une seconde classe qui aura la responsabilité d’écrire le Log :

```
public static class Log
{
    public static void Error(Exception ex)
    {
        System.IO.File.WriteAllText(@"c:\temp\log.txt",ex.ToString());
    }
}

public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}

    public int Insert()
    {
```

```
        try
        {
            // insertion en base de données
            // retourne l'ID
            return 0;
        } catch (Exception e)
        {
            Log.Error(e);
            return -1;
        }
    }
```

Désormais la classe `Client` n'a plus la responsabilité de l'écriture dans le Log, c'est une classe uniquement dédiée à cela qui s'en charge.

Les avantages sont nombreux et évidents, comme le fait de pouvoir faire évoluer l'écriture des Logs sans avoir à toucher à toutes les classes qui comme Client s'en chargeaient directement à tort.

Si nos soupçons s'avéraient fondés, il faudrait agir de la même façon sur la partie insertion en base de données.

> *Attention : les principes d'architecture ne sont que rarement des absolus qu'on peut suivre à la lettre sans faire preuve d'un minimum de jugeote… Par exemple ici (et considérant que l'insertion des données ne pose pas de problème) vous trouverez toujours un maniaque de SRP qui vous dira que gérer les erreurs n'est pas de la responsabilité de la classe Client.*

Aurait-il tort ? Personnellement je ne le pense pas. Le code "try/catch" protège l'accès aux données. Si SRP est correctement respecté alors cet accès aux données sera effectué par un objet du DAL ou en tout cas un code spécialisé dans les accès aux données des clients. Et c'est bien entendu ce code qui devra gérer les erreurs d'insertion ou autres opérations CRUD sur les clients…

De fait le respect net de SRP oblige à complexifier un peu l'exemple :

```
public static class Log
{
    public static void Error(Exception ex)
    {
        System.IO.File.WriteAllText(@"c:\temp\log.txt",ex.ToString());
    }
}

public static class CrudClient
{
    public static int Insert(Client client)
```

```
    {
        try
        {
            //accès DAL ou autre
            return 0;
        } catch (Exception e)
        {
            Log.Error(e);
            return -1;
        }
    }
}

public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}

    public int Insert()
    {
        return CrudClient.Insert(this);
    }
}
```

Désormais la classe `Client` respecte SRP. Elle maintient les informations d'un client et sait les persister.

L'écriture dans le `Log`, la réalisation des opérations CRUD et la gestion de leurs erreurs tout cela a été transféré à d'autres classes qui n'ont qu'une seule responsabilité elles aussi (écrire un log, persister un client…).

Peut-on être encore plus tatillon ?

Oui, et on le doit.

En réalité la classe Client est une classe BOL (une classe métier), elle ne doit en aucun cas savoir comment se persister ou se réhydrater. Nous avons un embryon de solution avec notre classe CrudClient. Il suffit tout bêtement de supprimer la méthode Insert de Client pour enfin avoir un code propre qui respecte SRP !

```
void Main()
{
    var c = new Client{Name="E-Naxos"};
    CrudClient.Insert(c);
}

public static class Log
{
    public static void Error(Exception ex)
    {
        System.IO.File.WriteAllText(@"c:\temp\log.txt",ex.ToString());
```

```
    }
}

public static class CrudClient
{
    public static int Insert(Client client)
    {
        try
        {
            //accès DAL ou autre
            return 0;
        } catch (Exception e)
        {
            Log.Error(e);
            return -1;
        }
    }
}



public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}
}
```

Qui persistera le `Client` ? Le code qui en aura besoin. Comment le fera-t-il ? En invoquant `CrudClient.Insert(instanceClient);` C'est tout. C'est propre. Évolutif. Dans notre exemple ci-dessus c'est `Main()` qui créée une instance, l'initialise et demande qu'elle soit persistée.

La classe `Client` devient pure, uniquement dédiée à représenter un client. Les intentions sont désormais clarifiées.Chaque classe possède sa propre responsabilité et pourra évoluer librement sans créer de problème.

Respecter SRP est essentiel et lorsque vous regardez le petit bout de code dont je suis parti et l'architecture à laquelle j'arrive on peut comprendre pourquoi un logiciel bien écrit par un pro peut coûter plus cher qu'un code qui "semble" correct écrit par un débutant… Et à quel point il peut être difficile de justifier la différence auprès d'un profane qui ne voit que le temps consommé (le coût) et non le savoir mis en œuvre (la qualité)…

Cet exemple vous fait aussi toucher du doigt les raisons (nombreuses) qui me font considérer le *Unit Testing* comme un *cache misère dans la plupart des cas*. En effet, face à de mauvaises architectures il semble légitime de barder le code de tests unitaires. Cela évite les régressions en cas de maintenance.

Mais je suis convaincu qu'un code bien architecturé n'a pas un besoin fondamental de Unit Testing tout simplement parce qu'il est conçu pour être maintenable sans régression… On peut, et on doit à mon sens, tester des opérations de haut niveau pour s'assurer qu'elles fonctionnent toujours (*des opérations qui ont un sens pour l'utilisateur*, créditer un compte, supprimer un compte, attribuer un taux de ristourne à un compte et vérifier que les factures en prennent bien compte, etc). Mais tester chaque méthode, chaque bout de code est du temps perdu et ne peut que cacher une architecture bancale dès le départ. Et même bien testé un soft mal architecturé restera toujours mal architecturé…

O comme Open Closed Principle

Continuons avec notre classe `Client`. Supposons désormais qu'on puisse avoir à gérer plusieurs types de clients qui disposent de ristournent différentes.

Une première approche pourrait être celle-ci :

```
public enum CustomerCategory {Normal,Silver, Vip}

public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}
    public CustomerCategory Category {get; set;}

    public double GetDiscount()
    {
        if (Category==CustomerCategory.Normal) return 0d;
        else if (Category==CustomerCategory.Silver) return 5d;
        else return 10d;
    }
}
```

On remarquera que j'ai fait les choses proprement en déclarant une `Enum` plutôt que d'utiliser des entiers qui ne veulent rien dire…

Notre client possède donc une `Category` qui est initialisée par défaut à "Normal" et qui n'offre aucun discount. S'il devient un jour Silver ou VIP il obtiendra 5 ou 10 % de remise.

Qu'est-ce qui peut bien clocher ici ? SRP est respecté.

Mais ce qui ne va pas c'est la façon dont `GetDiscout()` est écrite. D'abord tous ces tests sont affreux, un switch serait peut-être plus approprié, mais là n'est pas la question.

La question c'est que la programmation objet bien faite est une programmation SANS "IF" !

Vous allez me dire, oui, c'est vrai le taux devrait être rangé dans une autre classe pour n'avoir qu'un point de modification et `GetDiscount` devrait appeler cette classe pour retourner le taux.

Beurk !

Non, je vous parle de Programmation Objet pas de passer le dimanche à jouer avec ce que vous avez acheté le samedi chez M. Bricolage !

L'Open Closed Principle nous parle d'**extensibilité**. De la facilité à étendre un code sans avoir à revisiter le code existant. Car c'est ce genre de choses qui justifient le Unit Testing… Le pompier incendiaire… Je fous le feu pour justifier mon poste de pompier !

La programmation objet est plus noble et réclame un peu plus de réflexion aussi. Plutôt que de créer un code bancal qu'il faudra tester et retester à la moindre modif tellement il tient debout par l'opération du saint esprit, est-il possible de trouver une ARCHITECTURE qui autorise les extensions sans prendre de risques ?

Oui bien entendu. Cela s'appelle l'héritage et le polymorphisme…

Dans une programmation véritablement objet il n'y a pas de IF (je pousse à l'extrême mais ici le `IF` va bien sauter !). Pourquoi ? Tout bêtement parce qu'en utilisant les principes de base de la PO et POO il n'y a le plus souvent pas besoin de ce type de construction, en tout cas au niveau architectural s'il est bien pensé.

De fait, nos clients normaux, Silver et VIP ne doivent pas se distinguer par une propriété mais par un héritage. Et le `IF` disparaît. L'avantage ? On peut ajouter demain un type de client Gold qui s'intercalerait entre Silver et VIP, ou ajouter un Premium au-dessus de VIP le tout sans avoir besoin de retester tout l'ensemble car ce qui aura été écrit ne bougera pas ! Adieu régression et Unit Testing bas niveau qui ne fait que cacher la misère…

```
public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}
    public virtual double Discount { get {return 0d;} }
}

public class ClientSilver : Client
{
    public override double Discount { get {return 5d;} }
}
```

```
public class ClientVip : Client
{
    public override double Discount { get {return 10d;} }
}
```

N'est-ce pas plus joli ainsi ?

Plus d'énumération à maintenir, plus de risque de faire une bêtise dans le `IF` lorsqu'on ajoute un type de client, etc…

Si un client est `Silver`, sont taux de ristourne est immédiat. Et si demain on ajoute un nouveau type seul le code de ce nouveau type devra être testé, pas le reste qui n'aura pas bougé. Et quand je dis testé… il faudrait juste relire la constante quoi… D'ailleurs cette constante peut être judicieusement externalisée de telle façon à ce que tous les taux de remise soient centralisés en un seul point et facile à modifier.

Ce qui donne alors :

```
public static class DiscountRate
{
    public static double Normal = 0d;
    public static double Silver = 5d;
    public static double Vip = 10d;
}

public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}
    public virtual double Discount { get {return DiscountRate.Normal;} }
}

public class ClientSilver : Client
{
    public override double Discount { get {return DiscountRate.Silver;} }
}

public class ClientVip : Client
{
    public override double Discount { get {return DiscountRate.Vip;} }
}
```

Désormais, les taux de remise étant nommés clairement et centralisés dans une classe en ayant la seule responsabilité on peut rêver à toutes les améliorations comme le stockage des remises dans une table, un fichier XML, proposer un écran de modification des taux et milles choses qui auraient été impossible avec la première écriture.

Le principe énoncé par le O de SOLID, Open Closed Principle, fait passer de l'âge de la modification à celui de l'extension.

C'est une évolution essentielle comme SRP pour construire un code de bonne qualité. Ce ne sont pas les seules règles à appliquer, mais sans elles ont peut être certain d'avoir un code bancal et difficile à maintenir même barder de Unit Testing à tous les coins de rue !

L comme Liskov

Le Liskov Subtitution Principle, *principe de substitution de Liskov*. Je dirais même de Barbara Liskov. Une femme, il faut donc le préciser dans ce monde de macho. Et même deux pour le prix d'une puisque c'est avec une copine à elle Jeannette Wing qu'elle a écrit un article dans lequel on retrouve le fameux principe. Je vous laisse lire la page Wikipédia dédié à ce dernier, c'est un peu technique mais précis, pour les exemples on va voir cela ensemble !

Supposons que nous souhaitons ajouter à notre gestion de clients la notion de prospect. Dans un premier temps et en fonction de paramètres décidés ailleurs, un prospect reçoit aussi une catégorie (selon le CA qu'il semble promettre de générer par exemple). C'est un objet de type client et il semble bien naturel de créer le prospect comme un descendant de client. Toutefois nous ne souhaitons pas que le prospect soit ajouté à la base de données, il faudra pour cela qu'il soit transformé en véritable client. Peu importe la réalité de cet exemple, ne compliquons pas trop les choses.

La première implémentation qui pourrait venir en tête (de quelqu'un qui ne connait pas SOLID !) serait quelque chose chose comme ça :

```
public static class Log
{ ... }

public static class CrudClient
{
    public static int Insert(Client client)
    {
        if (client is Prospect) return;
        try
        {
            //accès DAL ou autre
            return 0;
        } catch (Exception e)
        {
            Log.Error(e);
            return -1;
        }
```

```
    }
}

public static class DiscountRate
{
    public static double Normal = 0d;
    public static double Silver = 5d;
    public static double Vip = 10d;
    public static double Prospect = 0d;
}

public class Client
{
    public int Id {get; set;}
    public string Name {get; set;}
    public virtual double Discount { get {return DiscountRate.Normal;} }
}

public class ClientSilver : Client
{ ... }

public class ClientVip : Client
{ ... }

public class Prospect : Client
{
    public override double Discount { get {return DiscountRate.Prospect;} }
}
```

Mes premières adaptations ont été tellement sévères qu'il m'est presque impossible de violer le principe de Liskov il faudrait repartir du premier code mal écrit pour que cela soit plus évident… En effet si la méthode Insert avait été laissée dans Client j'aurais démontré par un override de cette dernière qu'en déclenchant une exception j'interdisais l'écriture en base de données. Mais comme S et O ont été respectés à la lettre, Client n'a maintenant plus la responsabilité de `Insert`. De fait je ne peux mettre en avant la construction fautive comme je le voudrais.

Mais qu'à cela ne tienne, mon code correct m'oblige à aller bidouiller la méthode Insert de `CrudClient` pour tester si l'instance est de type Prospect pour éviter l'insertion… Tordu, donc mauvais de toute façon.

Le levage d'une exception dans `Prospect` aurait malgré été plus parlante car le principe de Liskov interdit que de nouvelles exceptions soient levées par une classe héritée. Mais la page Wikipédia indiquée plus haut vous dira tout ce qu'il faut savoir sur ce sujet.

Bref, ici je pense que `Prospect` est naturellement un descendant de `Client` puisqu'il en partage presque tout à la différence de la persistance.

Mais on le voit, en considérant un Prospect comme un Client cela pose des problèmes et notre code commence à contenir des curiosités qui le rendent bancal. Bricoler la méthode Insert de `CrudClient` n'est vraiment pas une bonne solution. Cela sera difficile à maintenir dans le temps. Et puis c'est une verrue. Horrible.

Le principe de Liskov dans la pratique est d'une grande subtilité, l'exemple de violation donné par Wikipédia est parfait et le démontre bien. J'ai un `Rectangle` avec une largeur et une hauteur, je suis tenté quand je créée la classe `Carré` d'en faire un enfant de `Rectangle`. Après tout un un Carré n'est qu'un cas particulier de Rectangle. Mais dans `Carré` hauteur et largeur ne peuvent évoluer seules. Ma classe contiendra donc du code pour empêcher les variations non coordonnées de ces valeurs.

Or en disant qu'un `Carré` est un `Rectangle` je sous-entends qu'on peut utiliser un `Carré` partout où un `Rectangle` l'est. L'utilisateur s'attend à manipuler un `Rectangle` et ne comprendra pas pourquoi dans l'interface il ne peut plus modifier comme il veut la largeur et la hauteur.

C'est en réalité une fausse bonne idée cet héritage.

En réalité tout comme notre couple `Prospect` / `Client`, le couple `Carré` / `Rectangle` sont des types totalement différents. Cela ne veut pas dire qu'on renie les relations de proximité, ni même qu'un prospect est un client en puissance pas plus qu'un carré n'est qu'un cas particulier de rectangle, non, mais il faut se résoudre à admettre que leur nature est différente. Un Prospect représente un prospect et non un client et un client représente un client et non un prospect.

> *Au quotidien je vois des tas de softs (même connus) conçus sur la base d'une pure fainéantise. Les prospects sont stockés dans la table des cliens avec un flag en général. La bonne affaire !*
> *Sémantiquement c'est une hérésie, les actions autorisées sur ces objets sont différentes, ils sont de nature différentes. Ne faites pas comme tous ces fainéants, séparer clients et prospects dans vos logiciels ! Oui cela demandera un objet de plus, oui une table de plus certainement, mais bon si on commence ça on finit aussi par tout coller dans une seule grosse table… Où s'arrête cette déliquescence ? Un proverbe romain disait « quand les bornes sont dépassées il n'y a plus de limites ! » Ne dépassez pas les bornes !*

Revenons à notre exemple. Comment traduire cette réalité en supprimant la verrue ajoutée à `CrudClient` ?

Il y a bien entendu plusieurs possibilités mais l'une d'entre elles est très élégante lorsqu'on a des problèmes d'héritage : l'utilisation d'interfaces.

La solution ci-dessous peut se discuter, comme tous les points d'architecture d'ailleurs, mais elle montre comment respecter le L de SOLID sans violer le S et O…

```
void Main()
{
    var Tous = new List<IBase> ();
    Tous.Add(new Client{Name="toto"});
    Tous.Add(new ClientVip{Name="titi"});
    Tous.Add(new Prospect{Name="fifi"});
}

public static class Log
{
    public static void Error(Exception ex)
    {
        System.IO.File.WriteAllText(@"c:\temp\log.txt",ex.ToString());
    }
}

public static class CrudClient
{
    public static int Insert(Client client)
    {
        try
        {
            //accès DAL ou autre
            return client.Id;
        } catch (Exception e)
        {
            Log.Error(e);
            return -1;
        }
    }
}

public static class DiscountRate
{
    public static double Normal = 0d;
    public static double Silver = 5d;
    public static double Vip = 10d;
    public static double Prospect = 0d;
}

ublic interface IBase {
               string Name {get; set;}
                double Discount {get;}
            }

public interface IClient { int Id {get; set; } }

public class Client  : IBase, IClient
{
    public int Id {get; set;}
    public string Name {get; set;}
```

```
    public virtual double Discount { get {return DiscountRate.Normal;} }
}

public class ClientSilver : Client
{
    public override double Discount { get {return DiscountRate.Silver;} }
}

public class ClientVip : Client
{
    public override double Discount { get {return DiscountRate.Vip;} }
}

public class Prospect : IBase
{
    public string Name {get; set;}
    public double Discount { get {return DiscountRate.Prospect;} }
}
```

Le `Main()` montre qu'il est possible de manipuler à la fois des Clients et ses variantes et des Prospects. Ici une liste est constituée mélangeant client de base, client VIP et un prospect. On peut donc fournir des affichages triés, des listings etc sans problème.

On remarque que `Prospect` ne descend plus de `Client` mais ne fait qu'implémenter `IBase`, et c'est suffisant pour montrer un nom et un taux de remise…

La classe `Client` quant à elle supporte une seconde interface exposant l'ID par cohérente et pour signifier que seuls les clients sont persistés puisque c'est à ce moment là qu'ils reçoivent l'ID (en général l'ID unique de la base de données qui n'est souvent pas ce qu'ici on appellerait un "code client" qui peut aussi être unique mais qui a un sens fonctionnel et non technique comme l'ID unique de la base).

On remarque en outre que la classe `CrudClient` n'a pas changée, elle revient à ce qu'elle était, sans bricolage. Rien à faire. En effet, elle a été conçue pour persister des instances de `Client`. Or celles de `Prospect` ne peuvent être acceptées. Pas besoin de lever des exceptions ou autre magouille malsaine, la façon dont nous avons architecturé le logiciel rend tout simplement IMPOSSIBLE la confusion et donc la persistance d'un `Prospect`. Pas besoin de IF et encore moins de Unit Testing pour le comprendre. C'est le compilateur qui le refusera…

Que nous sommes loin du petit bout de code original que je vous invite à regarder à nouveau pour vous rendre compte de sa métamorphose !

I comme ISP

L'Interface Segregation Principe ou *principe de ségrégation des interfaces* vient tout naturellement compléter les principes déjà présentés.

Expliquer autrement l'ISP nous dit que les interfaces sont immuables. On n'ajoute pas après coup une méthode ou une propriété à une interface car dans ce cas là on casse tout le code qui utilise déjà l'interface. Or dans de nombreux cas, surtout en entreprise, il peut y avoir plusieurs équipes, plusieurs départements qui utilisent des DLL de base qui sont partagées. On ne sait jamais qui utilise quoi avec assurance. Modifier une interface c'est casser le code de quelqu'un d'autre avec des conséquences qu'on ne peut prévoir.

Imaginons donc maintenant que notre `IBase` doive exposer un numéro de téléphone. Modifier `IBase` est un très mauvais choix et nous venons de voir pourquoi. Alors comment le gérer ? Simplement en créant une nouvelle interface. Les interfaces se versionnent par ajout d'une nouvelle interface.

```
public interface IBase
{
    string Name {get; set;}
    double Discount {get;}
}

public interface IBaseV2 : IBase { string Phone { get; set; } }

public interface IClient { int Id {get; set; } }

public class Client  : IBase, IClient
{
    public int Id {get; set;}
    public string Name {get; set;}
    public virtual double Discount { get {return DiscountRate.Normal;} }
}

public class ClientSilver : Client {...}

public class ClientVip : Client {...}

public class Prospect : IBase, IBaseV2
{
    public string Name {get; set;}
    public double Discount { get {return DiscountRate.Prospect;} }
    public string Phone {get; set; }
}
```

L'interface `IBaseV2` est créée par héritage de `IBase`, elle ajoute la propriété `Phone`.

`Client` et ses descendants supportent toujours `IBase` alors que `Prospect` supporte `IBase` et `IBaseV2`.

Ce double support pourrait troubler, n'est-ce pas une répétition inutile ?

Non car on sait que du code existant utilise `Prospect` comme `IBase`, il ne faut pas supprimer ce support car nous casserions du code existant !

Et en ajoutant `IBaseV2` nous n'avons rien à ajouter d'autre que le champ supplémentaire, `IBaseV2` a des exigences qui sont déjà supportées par la classe qui supportait IBase. Le compilateur se débrouille très bien et n'impose bien entendu du pas de déclarer deux fois les mêmes propriétés ou méthodes ! (ce qui ne marcherait pas de toute façon).

Ici nous faisons gentiment évoluer notre code. Les nouvelles fonctionnalités utiliseront naturellement `IBaseV2` et verront les clients et prospects sous cet angle là, les anciens programmes continueront à fonctionner normalement.

Là encore les bons choix d'architecture évitent d'avoir à tester bêtement tous les méthodes de toutes les classes. Nous ne risquons pas d'introduire la moindre régression en travaillant de la sorte…

D comme DIP

Dependency Inversion Principe. Principe d'inversion de dépendance.

L'injection de dépendance est un sujet que j'ai déjà traité souvent je ne m'étendrais pas et je renverrai le lecteur intéressé à tous les articles qui abordent ce sujet comme par exemple (liste non exhaustive) :

- Windows Phone, Android et iOS & Injection de dépendances et conteneurs IoC
- MVVM, Unity, Prism, Inversion of Control…
- l'IoC avec MvvmCross, de WinRT à Android en passant par Windows Phone
- MVVM, Unity, Prism, Inversion of Control…

Dans notre exemple de code on pourrait par injection de dépendances passer au constructeur de CrudClient l'instance d'un Log, cela serait fait via un conteneur d'IoC par exemple. Le découplage serait toujours respecté. Mais tout cela ce sont les avantages déjà abordés de l'injection de dépendance, l'inversion de contrôle, etc.

Conclusion

Appliquer les principes SOLID est une base non négociable, encore faut-il comprendre chacun de ces principes !

J'espère que ce petit tour d'horizon vous permettra de mieux comprendre SOLID et de mieux architecturer votre code…

Designer

Du bon usage de RelativePanel et AdaptiveTrigger

Le RelativePanel et l'AdaptiveTrigger sont des nouveautés du XAML de UWP. Grâce à eux on positionne les éléments les uns par rapport aux autres et on réagit aux différentes résolutions. Comment s'en servir concrètement dans le cadre de l'Adaptive Design ?

Adaptive Design

Je ne m'étendrais pas des heures sur le sujet, regardez dans les billets de ces derniers mois c'est un sujet que j'ai déjà traité. Pour faire court l'adative design est une façon de concevoir les UI de telle sorte à ce qu'elles s'adaptent dynamiquement aux changements de tailles et résolutions des machines.

Ce qui était purement théorique est devenu réalité avec UWP qui offre le moyen de faire un seul code, mais aussi une seule UI, pour tous les form factors.

Il est ainsi devenu essentiel de rendre les UI réactives pour qu'elles s'adaptent instantanément tantôt à l'écran d'un PC, tantôt à celui d'un smartphone ou d'une tablette, voire d'un IoT, une XBox ou les Hololens…

Pour arriver à un tel tour de force on utilise toute la puissance du vectoriel de XAML mais aussi certaines nouveautés comme le RelativePanel.

Le RelativePanel

Si le Canvas est déconseillé depuis longtemps car trop lié aux pixels réels, la Grid a été sous Silverlight et WPF le contrôle de base par excellence pour créer des mises en page. Avec UWP la Grid reste toujours aussi utile mais le RelativePanel pourrait bien lui voler la vedette en tant qu'outil de base.

En effet les adaptions d'un StackPanel, d'un WrapPanel ou d'une Grid pour intelligentes qu'elles soient sont assez limitées en termes de mise en page. La Grid n'est finalement qu'une sorte de <table> HTML.

S'il s'agit de s'adapter à des affichages plus ou moins grand sur une seule famille de machines, comme les PC par exemple, la Grid fait des merveilles. Les lignes, les colonnes peuvent s'adapter en taille et offrir plus ou moins de place à l'information ce qui est souvent suffisant.

Mais s'il s'agit de s'adapter du PC au smartphone en passant par des tablettes ou une XBox, l'exercice de style est trop difficile en jouant uniquement sur la Grid ou les autres conteneurs. La raison en est simple : passer d'un écran de plus de 20 pouces à un autre de 4 pouces ne se fait pas seulement en rendant les colonnes et les lignes d'une Grid plus ou moins grandes… Il faut pouvoir totalement changer la disposition des contrôles, voire en cacher ou montrer certains. Le tout pour conserver une bonne lisibilité de la page en toute circonstance.

C'est là qu'entre en scène le RelativePanel qui à la base n'a rien de réactif : il permet seulement de positionner les contrôles enfants les uns par rapport aux autres de façon relative. Par exemple en indiquant que tel TextBlock sera à droite de telle TextBox… Ou en dessous, au dessus, etc.

Le RelativePanel permet un placement relatif mais pas réactif. Alors pourquoi joue-t-il un rôle si important dans cette nouvelle approche du design ?

Simplement parce qu'en changeant juste les relations entre les contrôles on modifie rapidement la mise en page. Je peux faire passer un libellé au dessus d'une zone de saisie sur un écran pas très large alors que je veux qu'il soit à sa gauche sur un écran large. On comprend bien que le RelativePanel par son mode de fonctionnement permet de changer facilement l'ordre visuel des composants bien mieux qu'une Grid qui ne peut que rendre des colonnes plus étroites ou plus larges (idem pour les lignes).

La Grid n'est pas hors jeu dans le reactive design, elle peut largement y tenir un rôle essentiel mais le RelativePanel se montre très bien adapté à ces changements de configurations visuelles. En associant le RelativePanel à d'autres conteneurs comme Grid ou StackPanel il devient possible de créer des UI très souples et facilement adaptables à de nombreux types d'écrans.

Toutefois puisque le RelativePanel n'offre qu'un algorithme de plus pour le placement de ses enfants (comme Grid ou WrapPanel) comment peut-on le rendre "réactif" ?

La réactivité réelle, celle qui déclenche "des choses" lorsque la taille de l'écran varie (ou la taille de la fenêtre de l'application), est à trouver dans un autre mécanisme, celui des AdaptiveTrigger…

AdaptiveTrigger

Pour relever le défi de l'adaptive design il ne suffit pas de disposer d'un RelativePanel comme on vient de le voir. Il faut aussi disposer d'un quelque chose de plus, un moyen d'être averti que la taille de la fenêtre est en dessous ou au dessus d'une certaine valeur pour ensuite aller modifier les relations entre contrôles dans un RelativePanel (ou d'autres conteneurs).

Le RelativePanel n'est qu'un moyen de plus de rendre le travail de design sous XAML ultra souple. Mais bien entendu on a le droit de modifier n'importe quel élément pour s'adapter à un autre form factor. Il n'y a pas même obligation d'utiliser un RelativePanel, c'est juste que cela a été conçu pour et que c'est bien pratique de créer ses UI avec ce nouveau conteneur.

Donc quels que soient les conteneurs ou contrôles utilisés, c'est grâce à l'AdaptiveTrigger qu'on va pouvoir réagir au contexte visuel et pouvoir modifier les valeurs des autres contrôles (visibilité, taille, position, … toutes les propriétés peuvent changer).

L'AdaptiveTrigger est un nouveau type de déclencheur utilisable par le VisualStateManager de XAML. Comme le RelativePanel il n'invente pas un concept, déjà existant dans XAML, mais il propose un nouvel algorithme pour élargir nos possibilités de mise en page.

Grâce à l'AdaptiveTrigger on peut tout simplement créer des états visuels dans le VisualStateManager qui dépendent de la taille écran (largeur ou hauteur). Lorsque la condition de taille est détectée par le VSM l'état visuel associé est activé automatiquement sans besoin d'aucun code C# pour cela.

On peut ainsi concevoir avec le même code XAML plusieurs configurations d'écran en un seul fichier. Grâce à Blend qui sait manipuler le VSM on passe d'un état à l'autre par un simple clic sans avoir besoin de réellement redimensionner à chaque fois son espace de travail.

> *C'est l'occasion de la dire et de le redire, Blend est l'outil pour travailler les UI, depuis les débuts de XAML et encore plus aujourd'hui. Même si le designer XAML de VS s'est beaucoup amélioré avec le temps, Blend lui est supérieur en tout point et sait manipuler des concepts absent du designer de VS, comme les états visuels ou les animations par exemple.*

Un exemple visuel

Le mieux pour bien comprendre comment tout cela fonctionne est de regarder le petit gif animé ci-dessous (une capture écran de la démo dont on verra le code plus loin) :

Forcément dans un PDF les GIF animés ne sont pas vraiment gérés (c'est dommage d'ailleurs puisque les PDF sont majoritairement utilisés pour de l'affichage écran) mais n'hésitez pas cliquer sur l'image, elle vous enverra directement sur l'image animée de Dot.Blog !

Ce que l'on voit

On voit une fenêtre d'application UWP lancée en mode Local Machine, donc sur un PC de développement.

La fenêtre est de couleur Cyan et présente trois champs :

- Le libellé "Votre Prénom :" (TextBlock)
- La zone de saisie (TextBox)
- Un libellé en bas à gauche qui indique la largeur actuelle de la fenêtre (affichage purement technique, seuls les deux autres contrôles font réellement partie de la démo).

Lorsque la fenêtre possède une largeur suffisante (au dessus de 800 pixels effectifs, inclus), le libellé se trouve devant la zone de saisie.

Lorsque la fenêtre devient plus petite, en dessous de 800 epx mais au dessus de 600 epx outre le changement de couleur du fond qui devient gris pour bien marquer visuellement la rupture dans la démo, la zone de saisie passe en-dessous du libellé.

Si on continue à réduire la fenêtre, et en dessous de 600 epx, le fond devient blanc là aussi pour marquer visuellement le passage de la frontière mais surtout le libellé disparait et à l'intérieur de la zone de saisie apparait un texte fantôme indiquant qu'il faut saisir son prénom.

Si les changements de couleur du fond ne sont là que pour vous aider à voir les franchissements de frontières, la façon dont se positionnent les contrôles et dont leurs propriétés changent sont directement liées aux mécanismes d'adaptive design présentés ici.

Ce qui se passe

Pour que notre fenêtre réagisse de la sorte il a fallu plusieurs choses :

1. avoir mis en place des états visuels pilotés par des AdaptiveTrigger
2. avoir choisi les "frontières" aussi bien leur nombre que les valeurs qui les définissent
3. avoir utilisé un RelativePanel pour positionner les contrôles de façon relative
4. avoir modifié les placements relatifs du RelativePanel dans chaque état visuel déclenché par les AdaptiveTrigger

C'est ainsi qu'on arrive à nos fins, par un savant mélange des possibilités de ces deux nouveautés de XAML sous UWP, l'AdaptiveTrigger et le RelativePanel…

Le Code

Le code XAML de la mise en page qu'on peut admirer dans la capture GIF animée est le suivant :

```
<Page

    x:Class="App4.MainPage"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:local="using:App4"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d" >
```

```
<Grid x:Name="LayoutRoot" Background="Cyan" >

    <Grid.Resources>

        <local:ActualSizePropertyProxy Element="{Binding ElementName=LayoutRoot}"
          x:Name="widthProxy" />

    </Grid.Resources>

    <VisualStateManager.VisualStateGroups>

        <VisualStateGroup x:Name="MiseEnPageGroup">

            <VisualState x:Name="LargeState">

                <VisualState.StateTriggers>

                    <AdaptiveTrigger MinWindowWidth="800" />

                </VisualState.StateTriggers>

                <VisualState.Setters>

                  <Setter Target="LayoutRoot.Background" Value="Cyan" />

                  <Setter Target="textBox.(RelativePanel.RightOf)" Value="textBlock" />

                  <Setter Target="textBox.PlaceholderText" Value="" />

                </VisualState.Setters>

            </VisualState>


            <VisualState x:Name="MoyenState">

                <VisualState.StateTriggers>

                    <AdaptiveTrigger MinWindowWidth="600" />

                </VisualState.StateTriggers>

                <VisualState.Setters>

                    <Setter Target="LayoutRoot.Background" Value="Silver" />

                    <Setter Target="textBox.(RelativePanel.Below)" Value="textBlock" />

                    <Setter Target="textBox.PlaceholderText" Value="" />

                </VisualState.Setters>
```

```
                </VisualState>

                <VisualState x:Name="PetitState">

                    <VisualState.StateTriggers>

                        <AdaptiveTrigger MinWindowWidth="0" />

                    </VisualState.StateTriggers>

                    <VisualState.Setters>

                        <Setter Target="LayoutRoot.Background" Value="White" />

                        <Setter Target="textBlock.Visibility" Value="Collapsed" />

                        <Setter Target="textBox.PlaceholderText" Value="Prénom..." />

                    </VisualState.Setters>

                </VisualState>

            </VisualStateGroup>

        </VisualStateManager.VisualStateGroups>

        <RelativePanel Padding="15">

            <TextBlock x:Name="textBlock" Text="Votre prénom :" Margin="0,5,10,0"  />

            <TextBox x:Name="textBox" TextWrapping="NoWrap" Text="" Width="150"  />

            <TextBlock x:Name="textBlock2" Text=
               "{Binding ElementName=widthProxy,Path=ActualWidthValue}"
               RelativePanel.AlignBottomWithPanel="True"
               Style="{ThemeResource TitleTextBlockStyle}" />
        </RelativePanel>
    </Grid>
</Page>
```

Ce code est très simple et je vous laisse le lire tranquillement maintenant que vous en connaissez la finalité et même le visuel en mouvement.

Reste un petit détail pour les curieux qui n'a rien à voir avec le sujet du jour mais qui mérite quelques mots :

Le `TextBlock` qui affiche la largeur du `LayoutRoot` le fait bien entendu par l'effet d'un Binding. Malheureusement et c'était déjà le cas sous Silverlight, un Binding sur `ActualWidth` ou `ActualHeight` ne marche pas, c'est "by design", aucune notification

de changement de propriété n'est envoyée. On se demande bien pourquoi un tel manque et surtout pourquoi cela n'a jamais été corrigé. Mystère !

Mais peu importe, un simple Element Binding sur `l'ActualWidth` du `LayoutRoot` (une Grid) ne marche pas.

Pour y arriver j'ai utilise une astuce que j'avoue avoir trouvée le Web il y a plusieurs années de cela. L'astuce consiste à créer un "proxy" pour les propriétés qui ne bénéficient pas d'INPC. Le code de la classe est le suivant :

```
public class ActualSizePropertyProxy : FrameworkElement, INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        public FrameworkElement Element
        {
            get { return (FrameworkElement)GetValue(ElementProperty); }
            set { SetValue(ElementProperty, value); }
        }

        public double ActualHeightValue => Element?.ActualHeight ?? 0;

        public double ActualWidthValue => Element?.ActualWidth ?? 0;

        public static readonly DependencyProperty ElementProperty =
            DependencyProperty.Register("Element",
                   typeof(FrameworkElement), typeof(ActualSizePropertyProxy),
                   new PropertyMetadata(null, onElementPropertyChanged));

        private static void onElementPropertyChanged(DependencyObject d,
                                            DependencyPropertyChangedEventArgs e)
        {
            ((ActualSizePropertyProxy)d).onElementChanged(e);

        }

        private void onElementChanged(DependencyPropertyChangedEventArgs e)
        {
            var oldElement = (FrameworkElement)e.OldValue;
            var newElement = (FrameworkElement)e.NewValue;

            newElement.SizeChanged += elementSizeChanged;
            if (oldElement != null)
            {
                oldElement.SizeChanged -= elementSizeChanged;
            }
            notifyPropChange();
        }


        private void elementSizeChanged(object sender, SizeChangedEventArgs e)
        {
            notifyPropChange();
```

```
        }


        private void notifyPropChange()
        {
            if (PropertyChanged == null) return;
            PropertyChanged(this,
                      new PropertyChangedEventArgs("ActualWidthValue"));
            PropertyChanged(this,
                      new PropertyChangedEventArgs("ActualHeightValue"));
        }
    }
```

Il n'y a rien d'autre à faire que de déclarer ce code quelque part (dans la démo je l'ai mis dans le code-behind mais ce n'est pas sa place).

Ensuite dans le code XAML il faut attacher le proxy à l'élément dont on veut surveiller la taille, ici le `LayoutRoot` :

```
<Grid x:Name="LayoutRoot" Background="Cyan" >

        <Grid.Resources>

            <local:ActualSizePropertyProxy

             Element="{Binding ElementName=LayoutRoot}"

             x:Name="widthProxy" />

        </Grid.Resources>
```

On créée la ressource qu'on binde à la `Grid` et on lui donne un nom (`widthProxy` ici) pour s'en servir plus loin.

Et c'est dans le `TextBlock` qui affiche la largeur du `LayoutRoot` qu'on retrouve un binding au proxy :

```
<TextBlock x:Name="textBlock2"

  Text="{Binding ElementName=widthProxy,Path=ActualWidthValue}"

   RelativePanel.AlignBottomWithPanel="True"

   Style="{ThemeResource TitleTextBlockStyle}" />
```

Et le tour est joué !

Dernier problème : en mode Debug sous UWP comme sous WinRT ou Windows Phone le système affiche des compteurs de frames qui ne sont pas toujours très utiles et qui surtout peuvent empêcher de voir ce qu'il y a en dessous (ce qui est le cas dans la démo de cet article, pile au mauvais endroit).

Donc dernière astuce : comment se débarrasse-t-on de cet affichage en mode debug ? En le demandant poliment, mais reste à savoir quel satané namespace et quelles classes et propriétés sont à bricoler… L'astuce est là d'ailleurs. Tout bêtement en ajoutant dans le constructeur de la page :

```
Application.Current.DebugSettings.EnableFrameRateCounter = false;
```

Il existe une autre façon de faire plus globale, dans le code de App.Xaml.cs, il y une section conditionnelle qui passe la propriété EnableFrameRateCounter à true. Supprimer cette section de code ou mettez tout simplement la valeur à false… Cette modification a l'avantage d'être valable pour toute l'application.

Conclusion

Suivre les principes de l'Adaptive Design réclame en pratique de disposer des bons outils… XAML nous les offre sous la forme d'un nouveau trigger pour le `VisualStateManager` et d'un nouveau contrôle conteneur ouvrant la porte d'une mise en page par positionnements relatifs des éléments particulièrement bien adaptée à cette gymnastique.

Au passage on règle deux petits problèmes (le binding sur la largeur de la `Grid` et l'affichage du compteur de frame) ce qui permet d'encore plus rentabiliser la lecture de cet article !

UWP fait la police dans les fontes !

Les environnements de développement modernes ne s'imposent plus via des choix techniques à destination des informaticiens mais par des choix esthétiques à destination des utilisateurs. Pour les éditeurs d'OS le succès n'est plus seulement une question de technique mais aussi et surtout d'UX ! Ces aspects sont donc fortement

règlementés autant que les API. Il en va ainsi des fontes utilisables dans vos programmes…

Design first

Lorsque je défends dans ces colonnes le concept de "*design first*" ce n'est pas uniquement par amour des belles choses, mais bien parce que l'ensemble de l'industrie informatique a basculé dans ce mode (et non cette mode !).

Désormais le look & feel fait bien plus pour l'adoption d'un OS ou d'une machine que la qualité des outils de développement. Et c'est la même chose pour les applications.

C'est pour cela qu'il faut aujourd'hui concevoir les applications en mode Design First, et ensuite s'inquiéter de l'implémentation.

Avec Microsoft nous avons l'avantage d'une société qui a un passé très marqué par la "*developer-centric business culture*", la culture d'un business centré sur les développeurs. Ce qui fait que nous disposons des meilleurs outils de développement là où les autres doivent se contenter d'outils préhistoriques.

Mais même Microsoft s'est plié à cette contrainte du Design First. La codification de "Metro" fut une étape essentielle de ce changement.

Toutefois cela ne sert à rien si les développeurs ne suivent pas le design. Car l'OS n'est plus aujourd'hui une coquille pour tout et n'importe quoi, l'éditeur *vend aussi* une UX avec son OS. Et *les applications doivent se conformer à cette UX* pour qu'elles soient cohérentes et plaisantes pour l'utilisateur final.

Ainsi, bannissez par pitié les kits qui permettent d'avoir le look Material Design de Google par exemple sur Windows 10 ! J'aime Material Design mais faire une application avec ce look sur Windows Phone serait totalement incohérent. Comme l'inverse (et il y a des launchers au look Metro pour Android !). On se doit de respecter l'OS sur lequel on se trouve. Ce qui avec UWP ne pose pas de problème puisque le même OS permet d'attaquer tous les forms factors sans être obligé de se mélanger avec d'autres OS… Le cross-form-factor remplace le cross-plateforme en quelque sorte.

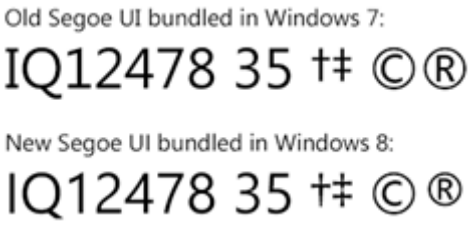
Il a donc des règles et il faut les respecter, la créativité doit s'installer dans les bornes de ces règles. Ce qui n'est pas gênant car créer sous crainte est souvent ce qui donne les œuvres les plus belles !

La police des fontes vous a à l'œil !

Ne vous inquiétez pas le GIGN ne vas pas débarquer par les fenêtres parce que vous aurez utilisé du **Comic Sans**, même si une exécution sommaire serait une punition adaptée à un tel sacrilège, un crime de lèse-design !

En revanche dans vos applications UWP Microsoft vous conseille très fortement de respecter un choix de fontes et de contextes dans lesquels l'utiliser…

La type ramp Windows 10



La *type ramp*, qu'on pourrait traduire par … pas grand chose… la "rampe des fontes" ce qui n'éclaire pas beaucoup, donc mieux vaut garder l'expression américaine… On comprend dans cette notion de "rampe" qu'il y a à la fois un choix, une succession, une progression. Et c'est bien de cela qu'il s'agit en fait : *une progression dans les tailles et les graisses selon le contexte*.

La police Windows 10 pour les UI est le Segoe UI, pas une autre. Et cette fonte se décline en différentes tailles et graisses selon le contexte précis. Ce contexte se matérialise même par un nom de style XAML directement utilisable dans vos applications. Il est fortement conseillé là aussi d'utiliser ces noms de styles plutôt que la fonte directement. UWP peut évoluer et en utilisant les styles prédéfinis vous êtres certains que votre application suivra les changements en dehors de la certitude de respecter la charte graphique UWP.

Style	Graisse	Taille	Style XAML
Header / Entête	Light	45 epx	HeaderTextBlockStyle
SubHeader / Sous Entête	Light	34	SubheaderTextBlockStyle
Title / Titre	Semilight	24	TitleTextBlockStyle
Subtitle / Sous-titre	Regular	20	SubtitleTextBlockStyle
Base	Semibold	15	BaseTextBlockStyle
Body / Corps	Regular	15	BodyTextBlockStyle
Caption / Légence	Regular	12	CaptionTextBlockStyle

On notera la taille indiquée en "**epx**", c'est à dire en "*effective pixels*" ou "pixels effectifs", des pixels invariants peu importe la taille réelle du support et sa résolution, gros avantage de UWP.

Autres fontes

Les choses ne sont pas si strictes que cela malgré tout. Et vous avez le droit d'utiliser d'autres polices que Segoe UI, mais là encore cela reste très codifié.

Windows 10 vous garantit que certaines polices seront présentes, en utiliser d'autres pourra poser des problèmes ou alourdir et ralentir votre application (téléchargement de la police notamment et mauvais affichage si ce téléchargement n'est pas possible). Il existe des polices spéciales pour les jeux de caractères étrangers non alphabétiques, je ne les listerai pas ici, la doc MSDN est là pour cela, mais il est intéressant de connaitre la liste des fontes directement exploitables en occident hors Segoe UI :

- Arial (Regular, *Italic*, **Bold**, ***Bold Italic***, **Black**)
- Calibri (Regular, *Italic*, **Bold**, ***Bold Italic***, Light, *Light Italic*)
- Cambria (Regular)
- Cambria Math (Regular)
- `Courier New (Regular, Italic, Bold, Boid italic)`
- Georgia (Regular, *italic*, **Bold**, ***Bold italic***)
- `Lucida Console (Regular)`

- Segoe MDL2 Assets (Regular) police spécifique pour les icônes d'App
- *Segoe Print (Regular)*
- SimSun (Regular)
- Times New Roman (Regular, *Italic*, **Bold**, ***Bold italic***)
- Trebuchet MS (Regular, *Italic*, **Bold**, ***Bold italic***)
- Verdana (Regular, *Italic*, **Bold**, ***Bold italic***)
- Webdings (Regular)
- Wingdings (Regular)

Sorti de ce lot de polices assurément présentes, il y a donc des risques à prendre et à calculer…

La liste ci-dessus utilise chaque fonte (en 14) pour son nom (sauf les deux de symboles à la fin), donc pour voir cette liste correctement il faut que vous ayez les polices installées chez vous. Pas de problème si vous faites tourner Windows 10, sinon certaines vous manqueront et l'affichage ne sera pas conforme.

Hors liste ?

Comment fait-on pour les polices « hors liste » ?

Deux cas se présentent :

- Il s'agit d'un choix typographique global pour l'App
- Il s'agit d'un titrage, d'un bouton particulier…

Dans le premier cas il est possible de télécharger des polices spéciales et de s'en servir temporairement dans l'application. C'est assez lourd et risqué puisque si un problème empêche le chargement ou l'installation c'est toute l'application qui aura mauvaise mine… risqué.

Dans le second cas il est plus judicieux, plutôt que d'imposer une fonte, de créer des objets graphiques bitmap, en PNG pour préserver les transparences c'est toujours mieux, et de les utiliser là où il y en a besoin. On perd la possibilité d'adapter la taille ou le texte à afficher, il faut éventuellement penser à fournir plusieurs versions de tailles différentes et pire en plusieurs langues si l'application est localisée.

Bref, aucune de ces solutions n'est particulièrement alléchante. Mieux vaut rester dans la liste… Et si on doit choisir, essayer d'utiliser une fonte téléchargée car jouer avec des dizaines de bitmaps est trop préhistorique à mon goût (et très lourd à maintenir).

Conclusion

Dans une approche design first le choix des fontes est tout sauf anodin… Connaitre les quelques règles imposées par la plateforme, utiliser les bons styles XAML, savoir quelques fontes complémentaires peuvent être utilisées, tout cela est vraiment essentiel. Rien ne sert de partir sur un design si vous ne respectez pas au départ ces impératifs. La place disponible pour votre mise en page dépendra grandement de la longueur des textes et celle-ci est directement liée à celle de la fonte utilisée et à sa nature (regarder la liste plus haut, tout est en 12 pourtant les tailles varient fortement !) .

UWP ne s'écarte pas tant que cela de Metro notamment dans l'esprit. Et Metro mettait *l'accent sur l'utilisation des mots et des fontes* dans différentes tailles. UWP reste dans cette ligne là, à vous d'en faire l'usage le plus esthétique et le plus pratique possible !

Responsive Design sous UWP

Créer des designs réactifs qui fonctionnent avec tous les form factors est un challenge. UWP met à notre disposition de nombreux outils pour nous faciliter les choses. Entre autres le `RelativePanel` et les `VisualState Triggers`…

Responsive ou Reactive Design

Peu importe le nom qu'on lui donne il s'agit toujours de la même chose : concevoir des Design réactifs. Cela n'enlève rien à toutes les choses qu'il fallait prendre en compte pour créer un bon design auparavant, cela ne fait qu'ajouter une contrainte de plus !

> *On pourra chipoter des heures la nuance entre responsive (sensible, "qui réagit bien") et reactive (réactif) qui semblent bien des synonymes une fois traduits… Entre bien réagir et être réactif le*

> *philosophe verra mille nuances, en ce qui concerne une poignée de contrôles visuels qui n'ont ni âme ni volonté propre cela simplifie malgré tout grandement le champ des interprétations possibles… Toutefois il ne faut pas confondre tout cela avec l'Adaptive Design me direz-vous ! Certes, mais une UI "adaptative" n'est rien d'autre qu'une UI "qui réagit bien" et qui est "réactive" vous répondrai-je alors !*

Cette contrainte est de taille (!) car des form factors différents il en pleut tous les jours …

Bien entendu Microsoft est le SEUL éditeur d'OS au monde à posséder déjà l'outil magique pour gérer une telle situation : un langage d'UI Vectoriel, XAML.

Qui mieux qu'un outil vectoriel peut permettre de s'adapter à toutes les tailles, toutes les résolutions ? Aucun. Ce ne sont pas les minables nine-patches ou autres combines qui nous renvoient 30 ans en arrière à la naissance du Web et des premiers sites avec collage d'images découpées en petits morceaux qui vont permettre d'écrire un démenti…

Et XAML étant vivant et bien vivant, il s'adapte aux nouvelles contraintes, facilement, logiquement, naturellement.

Par exemple dans le cadre du Responsive Design il peut s'avérer utile (et nous le verrons plus bas) de positionner les éléments de façon relative les uns par rapport aux autres. La notion de `Panel` existe déjà sous XAML, avec déjà des panels spécialisés (`StackPanel`, `WrapPanel`,…) ajouter le `RelativePanel` ne demande aucun bouleversement.

Pour aller plus loin il peut s'avérer encore plus utile de faire basculer l'affichage dans des états différents selon, par exemple, la résolution de la machine (donc au runtime). XAML possède déjà la notion de `Trigger`, de `VisualState`. Ajouter un trigger qui réagit à la résolution n'est qu'un jeu d'enfant. Et le `VisualState.Trigger` s'ajoute là aussi naturellement pour faire que XAML reste ce qu'il est depuis la première heure : le fleuron de l'intelligence en matière de création d'UI.

Ce blog n'est qu'un long cri d'amour à C# et XAML, plus de 800 articles en 7 ans dont certainement 80 à 90 % sont consacrés directement ou non à ces deux langages. UWP me donne à nouveau l'occasion de montrer leur modernité et je ne vous cacherai pas ma joie après la déception de WinRT  !

RelativePanel

C'est un Panel XAML donc une zone pouvant contenir des contrôles visuels. Ce qui différencie tous les Panels XAML les uns des autres c'est la stratégie de placement des contrôles enfants.

Le `canvas` qui est un panel aussi autorise un positionnement absolu, au pixel près. On ne l'utilise vraiment plus jamais car sa "stratégie" de placement est le niveau zéro de la stratégie tout court (obligeant pour changer de résolution de recalculer à la main tous les X et Y)… Déjà à ses débuts le canvas n'était pas très recommandé en raison de son manque de souplesse. Le `StackPanel` est déjà plus évolué et constitue un élément de base de toute mise en page XAML, il sait placer automatiquement tous ses enfants les uns derrière les autres, horizontalement ou verticalement. Le `WrapPanel` introduit plus tardivement notamment dans WPF met en page ses enfants séquentiellement de gauche à droite en sachant couper les lignes pour en créer de nouvelles selon les contraintes de largeur qui lui sont imposées. D'autres panels existent, soit fournis par XAML selon ses variantes, soit en Open Source dans des extensions de XAML, soit sous forme de contrôles commerciaux.

Il n'y a pas de limite à la sophistication d'un Panel XAML.

Le `RelativePanel` fait partie de ces panels montrant une certaine intelligence active. Il n'a pas de stratégie de placement propre comme le `StackPanel` ou le `WrapPanel` mais il offre des outils pour que le designer puisse développer un stratégie basée sur **les positions relatives des contrôles visuels les uns par rapport aux autres**.

Comme presque tous les Panel XAML le `RelativePanel` offre à ses enfants des propriétés spéciales appelées propriétés attachées.

> *Je renvoie le lecteur ignorant de ces subtilités de XAML à mes nombreux ouvrages et articles notamment le livre "XAML" dans la collection ALL.DOT.BLOG qui ne traite que de XAML et dans lequel vous trouverez des explications et des exemples de propriétés de dépendances et de propriétés attachées.*

Le `RelativePanel` offre tout un … panel de propriétés attachées permettant ainsi à ses enfants de définir leur position par rapport aux autre enfants du panel. Ces éléments doivent être nommés et désormais pour simplifier les choses XAML donne un nom par défaut systématiquement à tout contrôle posé sur la surface de design.

Ce ne fut pas toujours le cas mais l'existence même de RelativePanel rend ce changement de comportement par défaut tout à fait logique.

Le `RelativePanel` autorise ainsi une mise en page basée sur *les liens sémantiques qui relient les éléments entre eux*. Tel contrôle à vocation à être à gauche de tel autre qui lui-même n'a de sens qu'en étant au-dessus d'un troisième, etc.

En quoi le RelativePanel est-il une aide au Responsive Design ?

Le `RelativePanel` est par essence dédié au Responsive Design étant capable de réorganiser ses enfants en fonctions de règles positionnelles. Dans le cadre d'un affichage fixe un tel comportement n'aurait guère d'intérêt. Bien entendu le `RelativePanel` est exploitable pour lui-même, uniquement parce que la stratégie de placement qu'il offre est originale et puissante. Mais on sent bien qu'il prend tout son sens dès lors que l'affichage peut changer de taille au runtime.

Dans une `Grid` une fois les colonnes et les lignes définies la seule chose qui puissent s'adapter sont la largeur ou la hauteur des cases ainsi définies. Mais l'organisation visuelle est toujours la même, elle ne peut pas s'adapter. Ce qui se trouve sur une même ligne ou même colonne le restera toujours.

Le `RelativePanel` est un peu comme cela il est vrai, une fois les relations posées entre les éléments elles sont ce qu'elles sont. Oui mais… il est très difficile au runtime de modifier du tout au tout la mise en page d'une `Grid` (faire passer une cellule d'une ligne à une autre n'est pas possible, on peut changer le contenu mais cela demande du code et n'est pas tout à fait identique), alors qu'avec un `RelativePanel` il est très simple de modifier la propriété attachée d'un enfant pour lui dire ne soit plus à gauche de tel autre enfant mais positionne toi "au dessus". Par exemple sur un écran large je peux vouloir mettre les libellés devant les boites de saisie alors que sur un écran plus petit le libellé sera placé au-dessus du champ à saisir.

Ce genre de changement d'organisation spatiale deviendrait un cauchemar avec une `Grid`, c'est un jeu d'enfant avec le `RelativePanel`…

Le Responsive Design implique cette gymnastique de placements pouvant changer en fonction de l'orientation (portrait/paysage) et de la résolution (en pixels effectifs sous UWP).

En WPF à l'aide de tous les panels existants il est tout à fait possible de créer des affichages pour PC qui savent tirer parti de l'espace supplémentaire qu'offre un 26"

par rapport à un 16", mais avec du code, sinon on se limitera à agrandir ou rétrécir certaines zones. Là est bien toute l'adaptation possible.

En UWP à ces mêmes contraintes s'ajoutent celles de la résolution et de la taille qui va que quelques pouces (2 ou 3) à SurfaceHub qui fait 84 pouces en 4K HD !

Le ratio même peut changer entre un écran de smartphone en portait et un écran de Hololens…

Il résulte de tout cela que bien que le concept de `RelativePanel` pourrait s'avérer très utile même sous WPF il ne prend tout son sens que dans le cadre de UWP et du Reactive Design.

Réactif mais comment ?

Il est vrai que gérer la réactivité d'un `RelativePanel` est une autre affaire… il s'appelle d'ailleurs *Relative panel* et non *Reactive panel*. Son truc à lui c'est le positionnement relatif, pas la réactivité aux changements…

Alors comment faire ? Ecrire beaucoup de tests et de code pour modifier les propriétés des enfants d'un `RelativePanel` serait possible mais ne semble guère s'accorder avec les simplifications et la modernité de UWP…

Non, heureusement il y a mieux que ces horreurs à l'iOS ou l'Android, beaucoup mieux. Il y a les `VisualState.Triggers` !

VisualState.Trigger

Là aussi pour ne pas rendre mon exposé trop long je ne peux détailler ce qu'est un `VisualState` en XAML à ceux qui ne le savent pas déjà. Et je leur conseillerai comme tout à l'heure de se référer à mes livres ou articles qui traitent de XAML.

Pour ceux qui savent déjà, les états visuels d'une page XAML sont d'une grande souplesse puisqu'on peut définir les états que l'on veut obtenir répartis dans des groupes différenciés, par exemple le groupe `Work` qui peut avoir les états `RunningState` et `IdleState`, permettant d'adapter la vue selon que l'application travaille ou est en attente (ce n'est qu'un pur exemple). Les états de chaque groupe pouvant changer indépendamment alors qu'à l'intérieur d'un groupe un seul état peut être actif à la fois. Les animations peuvent aussi entrer dans la définition des états visuels avec des timelines, des fonctions de easing…

On connait alors dans ce cas aussi le rôle des Triggers, des “déclencheurs” qui entrainent le changement de valeur d’une propriété selon la valeur d’une autre qu’ils surveillent… En général la propriété surveillée est directement ou non un évènement comme par exemple “`MouseMove`” ou “`MouseOver`”, ce qui permettra de faire passer une zone en rouge quand elle est survolée et cela sans code C#. On peut déclencher des animations et des comportements complexes mais le principe reste le même.

Sous UWP les `Visual State Manager` a connu quelques améliorations mais la plus marquante est l’ajout d’un trigger spécial l’`AdaptiveTrigger`. Ce dernier surveille de lui-même la largeur et la hauteur de l’écran. Et lorsqu’il réagit aux bornes données on peut utiliser des `VisualStates.Setters` pour modifier les propriétés de plusieurs contrôles.

On pourrait faire la même chose avec du code C# qui surveillerait via des events le changement de taille et qui basculerait les états visuels créés en XAML avec un `GotoState()`. Mais il faut avouer que le fait que l’`AdaptiveTrigger` le fasse tout seul en XAML sans invoquer le moindre code externe est un avantage important !

AdaptiveTrigger et RelativePanel : un couple parfait !

Même s’ils ne sont pas fait pour travailler exclusivement ensemble ils ont malgré tout été pensé pour faire un couple d’enfer qui mériterait une couverture de Voici ou Gala !

En effet, une fois qu’on a compris l’avantage de chacune de ces nouveautés de XAML on commence a en entrevoir le potentiel lorsqu’elle sont couplées.

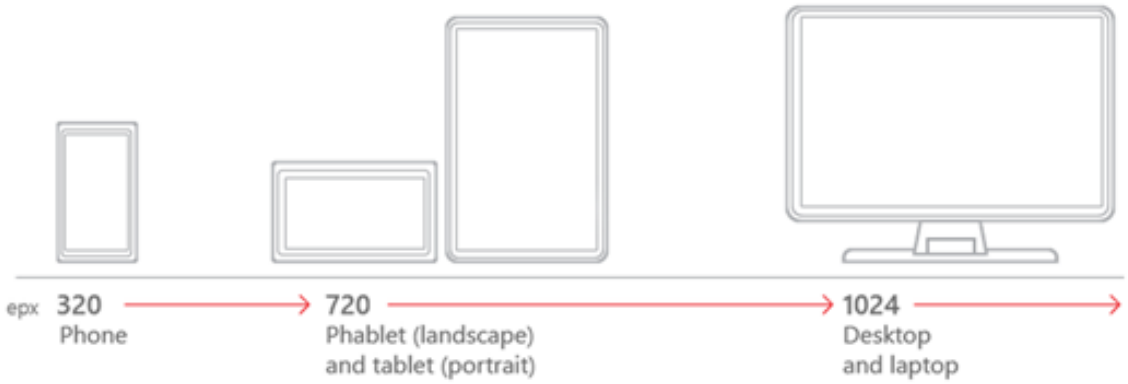
> *“Si l’écran est plus large que 750 pixels alors je mets le libellé A à gauche de la zone de saisie B. Si l’écran fait moins de 720 pixels mais plus de 400 je peux conserver ce placement en diminuant la taille du libellé qui restera lisible, mais en dessous, le libellé A doit passer au dessus de la zone de saisie B et en même temps devenir un peu plus petit…”*

Cette phrase raconte une histoire, celle d’une UI vivante et réactive… Et tout cela se réalise en XAML en combinant l’`AdaptiveTrigger` (pour détecter les changements de taille) et le `RelativePanel` (pour changer facilement l’organisation spatiale des éléments) en plus des changements classiques de propriétés qu’un Trigger peut autoriser (la taille de la fonte dans cet exemple).

Savoir s’adapter

Comment déterminer quelles sont les tailles à prendre en compte ? Microsoft définit dans UWP des familles de machine, ce qui peut donner un premier indice. Une machine de la famille *Mobile* sera forcément de plus petite taille qu'une machine de la catégorie *Desktop*. Et comme UWP raisonne en **pixels effectifs** et non en pixels réels, pas de prise de tête à avoir quand on prend en compte les résolutions… Car certains smartphone en 400 DPI par exemple peuvent avoir une taille supérieure en pixels réels à un écran de 19" en 72 DPI… Les autres OS laissent ce genre de considération de côté, à vous de vous enquiquiner…

Avec UWP tout est fait pour vous rendre la vie facile. Les tailles étant en pixels effectifs on peut facilement raisonner sur une échelle linéaire qui laisse de côté la résolution. Microsoft propose par exemple cette segmentation :



De même qu'ils indiquent avec précision les comportements qu'on peut attendre en fonction de ces tailles ce qui guide le design de façon appréciable :

Size class	small	medium	large
Width in effective pixels	320	720	1024
Typical screen size (diagonal)	4"-6"	8"	13" and wider
Typical devices	Phone	Tablet, phones with large screens	PC, laptop, Surface Hub
General recommendations	• You can make navigation and interactions easier on hand-held devices by placing navigation and command elements at the bottom of the screen so that users can easily reach them with their thumbs. • Center tab elements. • Set left and right window margins to 12 px to create a visual separation between the left and right edges of the app window. • Use 1 column/region at a time • Use an icon to represent search (don't show a search box). • Put the navigation pane in overlay mode to conserve screen space. • If you're using the master detail element, use the stacked presentation mode to save screen space.	• Make tab elements left-aligned. • Set left and right window margins to 24 px. We recommend creating a visual separation between the left and right edges of the app window. • Up to two columns/regions • Show the search box. • Put the navigation pane into docked mode so that it always shows.	• Put navigation and command elements at the top of the app window. • Make tab elements left-aligned. • Set left and right window margins to 24 px. We recommend creating a visual separation between the left and right edges of the app window. • Up to three columns/regions • Show the search box. • Put the navigation pane into docked mode so that it always shows.

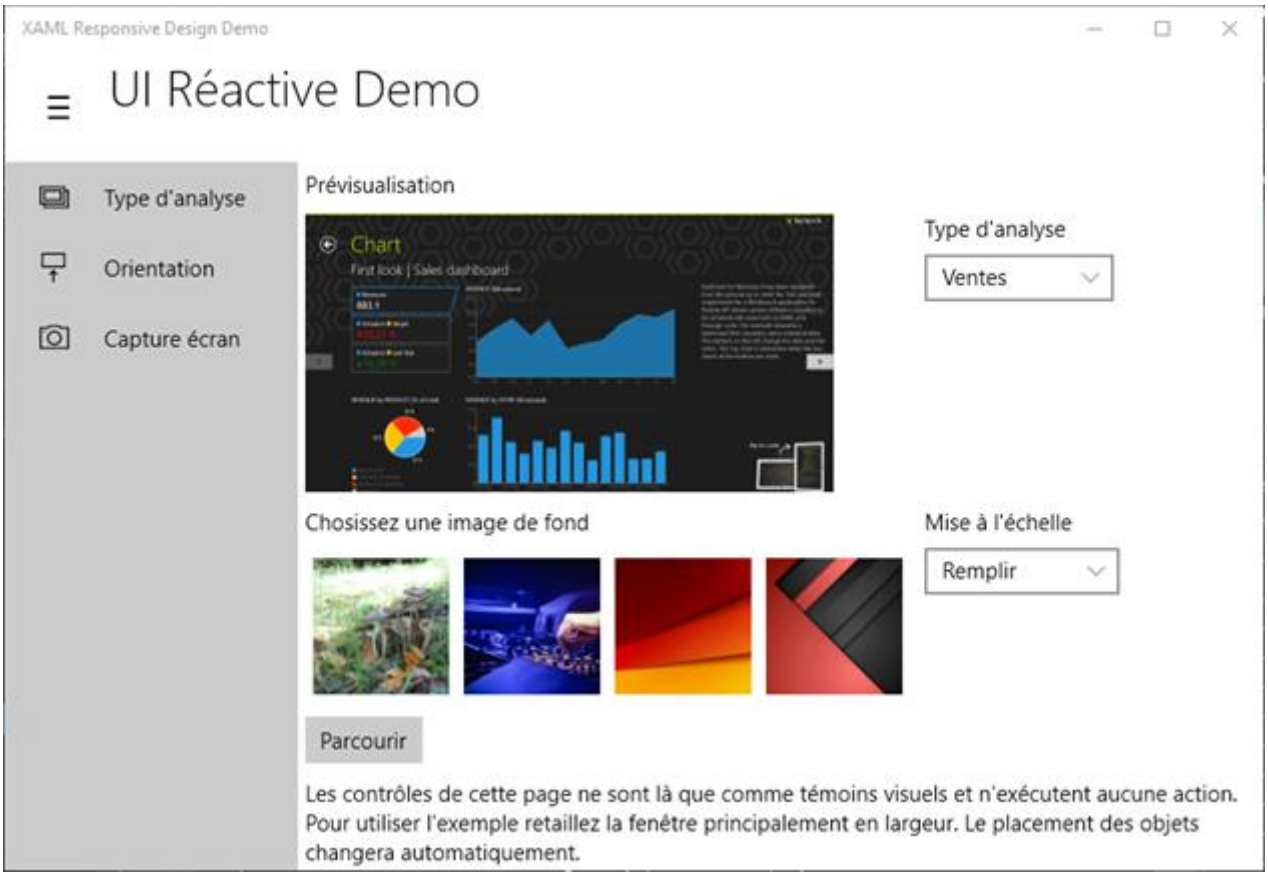
Ainsi, à l'aide ces informations il est assez facile de savoir comment programmer les `AdaptiveTriggers` pour qu'ils réagissent convenablement dans toutes les circonstances.

Créer des UI réactives pour tous les form factors avec un seul code n'a jamais été aussi facile. Même les Xamarin.Forms que j'adore ne vont pas aussi loin limitées qu'elles sont par les deux OS préhistoriques sur lesquels elles fonctionnent…

Une démo ?

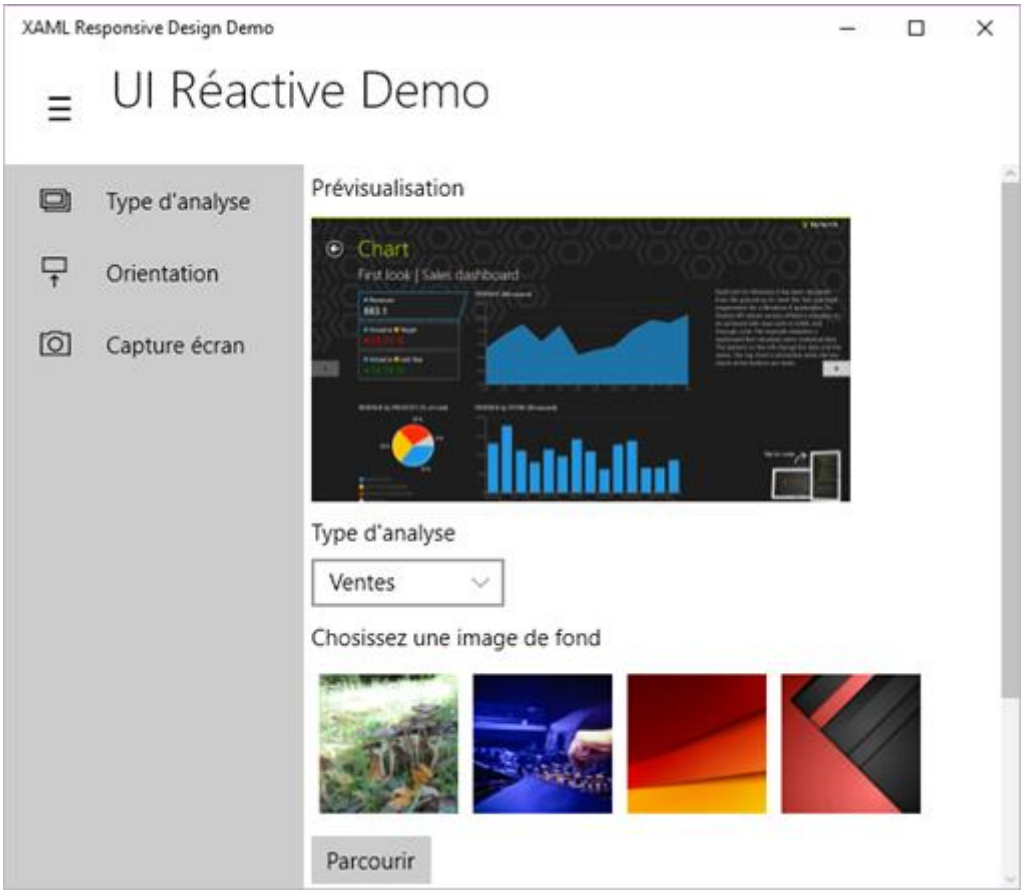
J'ai repris une démo Microsoft que j'ai un peu adaptée mais qui reste dans l'esprit de ce qu'elle montrait, le `RelativePanel` avec les `AdaptiveTriggers`… Tout juste ce dont nous avions besoin ici !

Voici comment elle apparait en mode local machine, donc desktop :



On voit un grand espace central réservé au contenu, sous une barre de titre précédée du bouton sandwich (pour le menu) et à gauche de l'écran le fameux menu déjà ouvert en grand avec icônes plus explications. C'est normal j'ai de la place, autant rendre le soft le plus simple possible à comprendre et à utiliser…

Sur le même bureau, si je diminue la largeur, à un moment donné nous obtenons :



Comme on le voit il faut scroller pour tout voir et *certains groupes de contrôles sont passés de la droite des images à sous les images*. Mais le menu est encore ouvert car la place le justifie (c'est un choix du Designer qui est supposé connaitre les utilisateurs et leurs besoins… revoir mes articles sur le design et ma vidéo Youtube sur la question…). On pourrait aussi n'afficher que la colonne des icônes pour avoir plus de place pour le contenu. Tout cela est possible, le reste est un choix de design.

Mais si j'exécute le même code sur un smartphone, disons en 5" :

Le menu s'est rétracté, il n'est visible par surimpression qu'en touchant du doigt le bouton sandwich :

Tout cela est magique car il n'y a aucun code pour le gérer en dehors du XAML de la Vue qui a elle seule peut s'adapter d'un petit smartphone à une phablette, une tablette ou un écran de PC de bureau…

Plutôt que de vous lister tout le code je préfère vous mettre en fin d'article le source de cette démo (il faut VS 2015 final et le SDK Windows 10) cela vous permettra d'étudier les subtilités XAML dans un environnement étudié pour… (Je vous conseille d'utiliser Blend qui permet de voir les états visuels, essentiels ici). Attention pour des raisons de taille j'ai supprimé dans le réperoire Assets les images utilisées, vous pouvez prendre n'importe lesquelles et les nommer de la même façon ou modifier le code XAML pour afficher les vôtres.

Conclusion

UWP est une plateforme extraordinaire, elle est dotée aussi d'outils d'une rare intelligence comme Blend ou VS, de langages modernes et d'un système de description visuel vectoriel qui fait passer tout le reste pour des antiquités.

Vous aimez vous aussi C# et XAML ? Vous allez adorez UWP ! (et votre patron aussi le devrait si vous lui expliquez bien, car UWP c'est la rationalisation des développements, la capitalisation du savoir et des équipes en place et l'accès à tous les form factors…).

Le fichier projet : cs.zip (199,9KB) (ce livre PDF possède des liens actifs, cliquez dessus !)

Le Reactive Design pour les Universal Apps

Après avoir présenter la plateforme Universal Windows Platform (UWP) et avoir montré un Hello World classique puis en MVVM il est temps de parler du Reactive Design !

Le Reactive Design : Concept, technique ou méthode ?

Le Reactive Design c'est tout cela à la fois selon comment on l'observe. C'est un concept, celui d'une UI réactive s'adaptant aux différents form factors. C'est aussi une technique, Microsoft donnant des moyens techniques pour appliquer le

Reactive Design en UWP. Et c'est aussi une méthode à suivre car le succès ne peut s'atteindre sans réflexion ni planification.

Par où commencer ?

Il existe un article fort intéressant de Microsoft à ce sujet en français. Je ne vais donc pas le reprendre ici cela n'apporterait aucune plus value. Je vous conseille donc la lecture de ce papier d'autant que pour le moment les documentations sur UWP sont assez légères, même en anglais.

"Conception dynamique des applications UWP"

Une réflexion sur les images …

Je lisais dernièrement un papier très intéressant sur le concept d'images réactives et comme cela cadre parfaitement avec le sujet c'est l'occasion de vous en parler !

Il s'agit d'une étude en plusieurs parties très complète et bien menée d'un certain Jason Grigsby sur le concept des "responsives images". Même si tout cela est envisagé sous l'angle du Web, la problématique concerne directement tous ceux qui doivent concevoir des UI "responsive" et "reactive".

Si cette série d'articles est intéressante c'est parce qu'elle est la première que je lis sur le sujet, celui des images en tant que telles et le respect de leur signifié.

Quand on parle d'UI réactive et flexibles on parle systématiquement de mise en page. Vous mettez une grille ici ou là, on vous conseille de faire des placements relatifs par là, voire de déplacer des objets ici selon la résolution (barre de commande en haut pour un PC, en bas pour un téléphone par exemple), etc.

Aussi nécessaire que soit cette approche de la mise en page et du placement des contrôles on zappe totalement le problème des images…

Et détrompez-vous, c'est bien plus sioux que de fournir une image en plusieurs tailles/résolutions !

On vous aurait menti ? Par omission oui !

L'intelligence du travail de Jason est de montrer deux choses : d'une part que les images elles-mêmes doivent être travaillées dans un mode "responsive" et d'autre part qu'il y a une sémantique à respecter, des choix à faire, un véritable travail de

directeur artistique derrière tout cela et pas seulement un bidouillage de changement de taille ou de DPI…

J'aime bien quand je lis un truc qui aborde les choses de façons "fraiche", nouvelle, qui ne redit pas les mêmes choses qu'on voit partout.

Vous forcer à réfléchir sur le fait que mêmes les images que vous utilisez méritent d'être traitées de façon bien plus sophistiquée que vous ne le pensez est à mon sens un véritable apport. Et puis vous saurez aussi quoi répondre quand on vous dira "pour les images pas de problème hein, vous passez tout en 300 DPI il vous faut 2 minutes avec Photoshop…".

Non il ne faut pas 2 minutes, non ce n'est pas un travail d'informaticien mais d'infographiste, voire donc de directeur artistique avec des choix qui doivent se justifier dans le contexte de l'information véhiculée. Qu'il s'agisse comme l'envisage Jason d'une page Web ou bien de l'UI d'une App tel que moi je transpose ici le discours.

Voici les liens vers la série originale :

- Definitions
- Img Required
- Srcset Display Density
- Srcset Width Descriptors
- Sizes
- Picture Element
- Type
- CSS Responsive Images
- Image breakpoints

Il existe une traduction française très partielle mais centrée sur l'une des parties les plus intéressantes, idem, je vous renvoie au site du traducteur :

"Introduction aux images responsives"

C’est court et cela pose le décor. Vous comprendrez assez rapidement l’enjeu et pourquoi adapter des images à différents form factors ne se résume pas **et ne peut pas se résumer** à un simple changement d’échelle avec Photoshop.

Conclusion

Le Reactive Design est un truc bien particulier qui profite à tous les développements, UWP ou non, dès lors qu’on désire offrir une UI digne de ce nom. Par exemple même si personne ne l’a envisagé dans ce contexte, reprendre le concept pour une application WPF est parfaitement justifié. Selon la taille de la fenêtre dimensionnée par l’utilisateur l’affichage ne fait pas que s’agrandir ou diminuer, il change de configuration, radicalement s’il le faut, pour offrir une UX pro.

Donc quel que soit vos objectifs de développement sur telle ou telle autre plateforme dans les semaines et mois à venir, vous pourrez tirer grand enseignement de l’esprit même du Reactive Design !

Personnaliser une Vue selon la famille de devices

UWP c'est un seul code C# et un seul code XAML pour toutes les familles de devices supportées. Mais dans certains cas il faut pouvoir personnaliser l'UI pour l'adapter plus en profondeur. UWP le permet de façon simple…

Famille de devices

Le lecteur m'excusera d'utiliser "*devices*" au lieu "d'unités" car hélas ce terme français est trop flou dans le titre d'un article. Device on voit ce que c'est, une machine, une unité mobile ou non. Mais pour les puristes de la langue française disons qu'il s'agit bien de "machines" pour faire simple.

UWP est fait pour tourner sur plein de form factors, mais ces derniers sont regroupés logiquement en familles. Vous n'êtes pas obligé de cibler toutes les machines possibles mais vous ne pouvez pas cibler une seule machine particulière, vous ne pouvez cibler au minimum qu'une famille de machines.

Ces familles sont évidentes : Desktop pour tout ce qui ressemble à un PC, Mobile pour tout ce qui est de l'ordre de la tablette, phablette et smartphone, IOT pour les objets connectés, etc.

Dans un développement "classique" on ciblera soit la famille Mobile, soit la famille Desktop ou plus intelligemment (si cela a un sens) les deux à la fois. Rajouter le support de la Xbox, des Hololens ou des IOT est une autre affaire. Par exemple ces derniers n'ont généralement pas d'UI… Soit aujourd'hui je suis en manque d'imagination soit en effet il y a un truc qui ne collerait pas à avoir une belle app de bureau qui pourrait aussi tourner sur un frigo connecté sans UI … Donc je suppose que lorsqu'il faudra cibler IOT (bracelets connectés, arduino…) le programme ne ciblera que cette famille là.

Pour tous le reste écrire un seul code C# et un seul code XAML pour toutes les familles est le gros intérêt de UWP.

Personnaliser les UI par famille

Même si je n'ai pas encore explicitement fait d'article sur la question j'en ai fait sur le Reactive Design qui explique comment on doit concevoir son code XAML de telle façon à ce qu'il s'adapte aux différents form factors sur lesquels il va tourner. Je reviendrai sur des exemples concrets dans d'autres articles pour parler par exemple du nouveau RelativePanel qui autorise le placement de ses enfants de façon relative

(à droite de, en dessous de, au dessus de, etc) ce qui autorise une mise en page automatique qui garde son sens.

Mais malgré toutes les astuces de Design, malgré la parfaite portabilité du jeu de contrôle UWP, malgré l'utilisation de Pixels Effectifs qui tiennent compte de la résolution et de la distance de vision des objets, malgré tout cela on peut se retrouver dans des cas où une mise en page distincte pour une famille donnée reste la solution la plus propre. Cela évite de créer du XAML spaghetti difficile à maintenir lorsqu'on utilise trop d'astuces au sein d'une même page.

Toucherait-on ici aux limites d'UWP ?

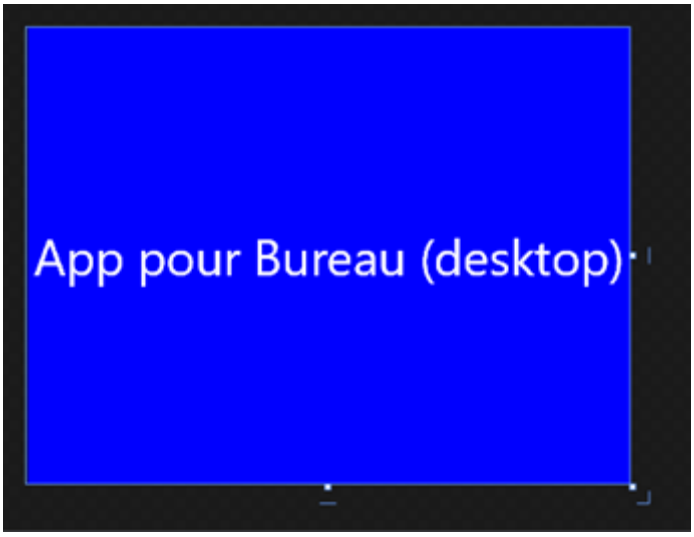
Heureusement non !

D'autres mécanismes de personnalisation ont été mis en place dans UWP pour vous permettre de pousser la personnalisation le plus loin possible sans toutefois ne jamais remettre en question le principe "d'un code, toutes les cibles".

Il existe plusieurs possibilités, celle évoquée plus haut qui consiste à utiliser les subtilités des Triggers XAML et de contrôles tel le RelativePanel, ou bien de concevoir carrément une page XAML (sans code) juste pour une famille de machine donnée. Ce mode est le moins automatique, le plus radical mais parfois le plus efficace. C'est ce mode là que je vais maintenant vous montrer. Ensuite je vous ferai voir comment on peut atteindre une haute personnalisation avec un seul code XAML…

Projet de test

Pour vous montrer tout cela je partirai d'une application UWP de type "blank" (vide) dans laquelle j'ai placé une grille bleue avec un text en 72 centré indiquant qu'on se trouve sur une appli desktop ce que nous considèrerons comme étant le mode "par défaut" de notre application. Ce qui donne un visuel de ce type pour le moment en design :

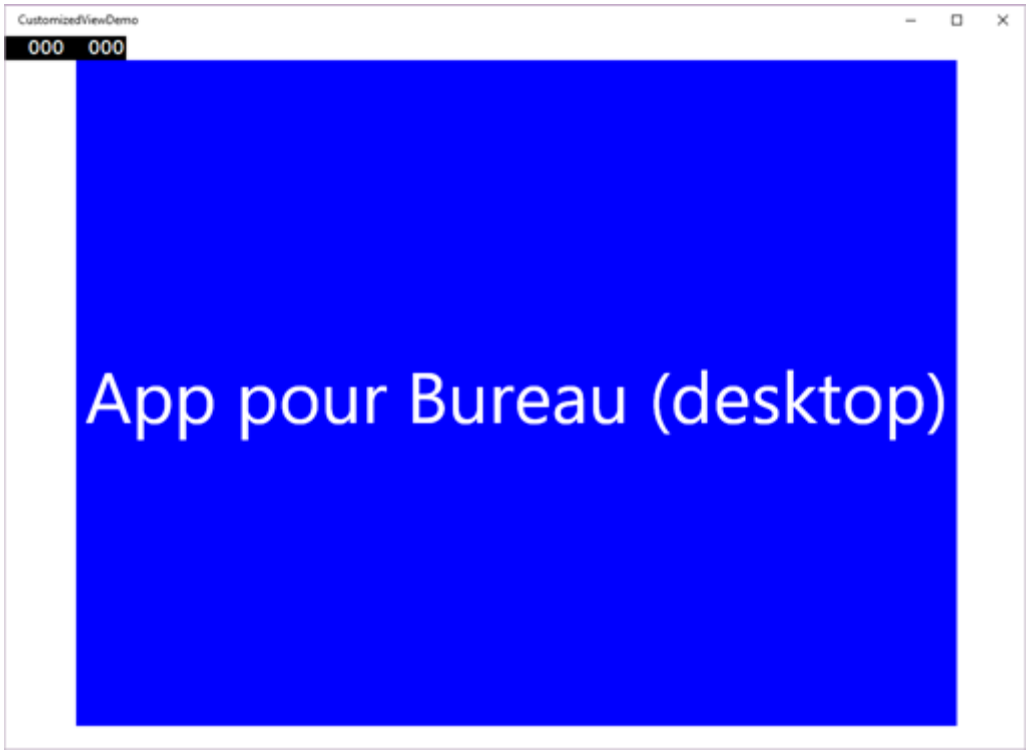


Le XAML de cette page est à rougir de honte tellement il est basique :

```
<Page

    x:Class="CustomizedViewDemo.MainPage"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:local="using:CustomizedViewDemo"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d" Height="664.227" Width="878.864">



    <Grid Background="Blue">

        <TextBlock Foreground="White" Text="App pour Bureau (desktop)" FontSize="72"

                   VerticalAlignment="Center" HorizontalAlignment="Center" />

    </Grid>

</Page>
```

Donc rien de bien exotique pour l'instant mais c'est une bonne base !

Si j'exécute le programme en mode machine locale j'obtiens la sortie suivante :



Et si j'exécute en mode Windows Phone 10 5" j'obtiens en toute logique l'horreur suivante :

On le voit bien, mais c'était à prévoir, ma mise en page pour appli desktop ne passe pas vraiment sur un smartphone… Mon code oui, mais ma mise en page non…

Il va donc falloir remédier à ce petit problème par quelques adaptations automatiques (puisque au final je n'aurai qu'un seul exe rappelons-le !).

Les sous-répertoires Familles

Une fois qu'on a éliminé les astuces XAML permettant de gérer le redimensionnement des éléments, qu'on a épuisé les placements un peu subtiles avec le `RelativePanel` il faut bien passer à la solution radicale : avoir une UI totalement différentes et adaptée à la cible. Mais toujours en gardant un seul code, un seul projet, pas un pataquès de projets distincts (comme hélas les "universal apps" de Windows 8 l'obligeaient juste pour deux cibles).

La première façon radicale de régler le problème fonctionne selon un principe déjà utilisé part d'autres OS : avoir des noms spéciaux qui permettent à l'OS de charger les bonnes ressources au runtime. Sous Android c'est un foutoir pas possible entre les résolutions, les tailles, les capacités techniques, etc, ça devient très vite un enfer. Microsoft dans UWP a choisi de se plier aussi à ce fonctionnement mais uniquement pour les cas limites et avec un peu plus d'ordre.

Microsoft a donc choisi de regrouper les machines par familles. Une même famille est présumée faire des choses similaires, ce qui n'interdit donc pas de s'adapter aux petites différences dans une famille données, mais la notion de famille marque une différence assez nette pour justifier de proposer des designs différents. Par exemple sur smartphone une appli ne proposera pas un affichage de PC. Il ne s'agit plus de seulement adapter la taille d'une grille, de jouer sur le positionnement des objets, non, il faut vraiment un design particulier pour avoir au final une super app PC et une super app smartphone, avec le même code en revanche.

Là où d'autres ont choisi un enfer de règles de nommage et de combinatoires explosives, Microsoft a donc eu l'intelligence de proposer uniquement une variante par famille. Il faut dire que les autres OS fonctionnent de façon tellement préhistorique que même un bouton doit être prévu dans toutes les tailles et résolutions possibles alors que forcément, avec le vectoriel de XAML ça élimine beaucoup de soucis… Le vectoriel est forcément l'avenir du multi form factors, le collage d'images et autres nine-patches sont une hérésie au XXIème siècle.

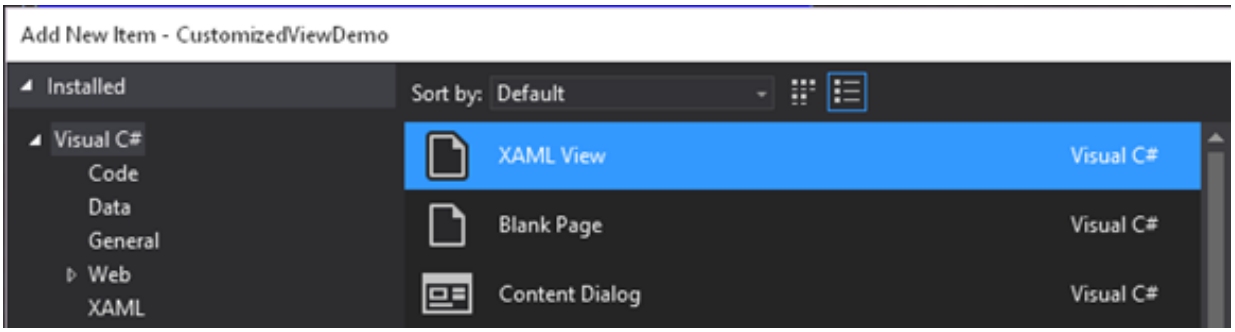
Bref le problème n'est pas la supériorité technologique des produits Microsoft, nous nous le savons, c'est d'en faire un succès auprès du public et des entreprises. Mais c'est un autre débat.

Comme nous l'avons vu plus haut notre projet est bien sympathique mais la sortie sur smartphone est une catastrophe. Nous avons donc décidé de fournir un design totalement spécifique pour les smartphones.

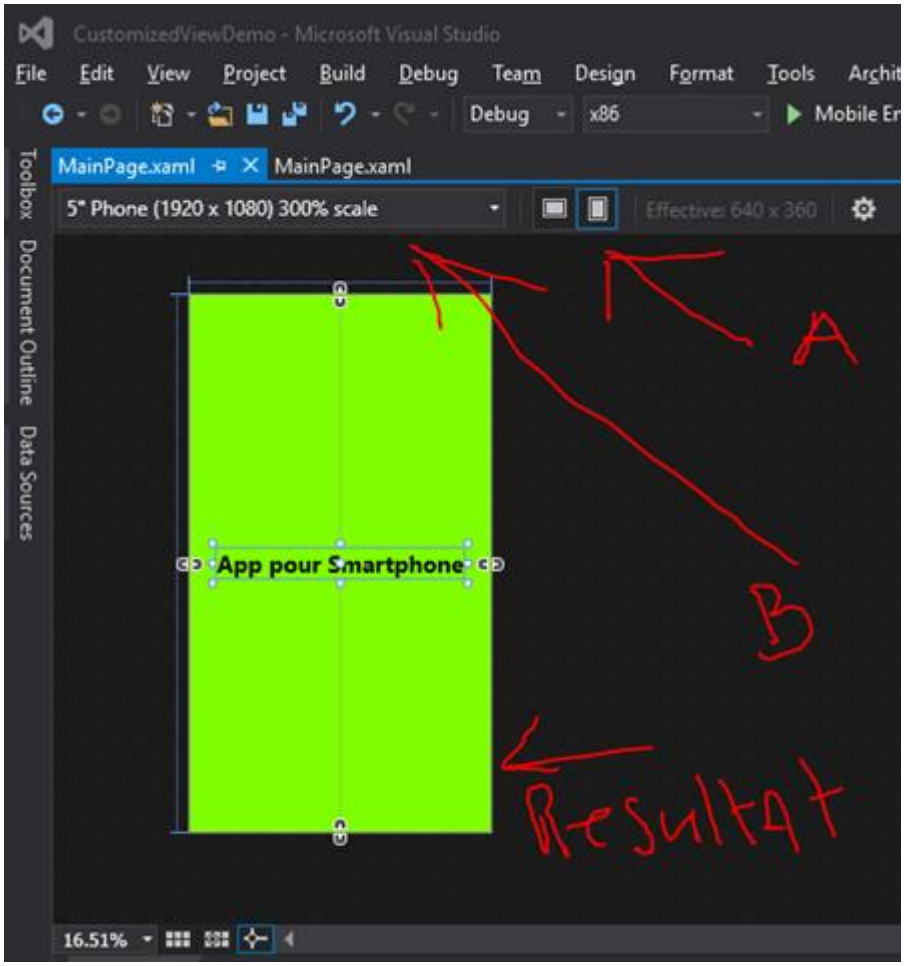
Pour le faire nous allons tout simplement créer un sous'-répertoire dans notre projet que nous allons appeler `DeviceFamilly-Mobile`. C'est simple à se rappeler. On voudrait un truc spécial pour le bureau, on créerait un `DeviceFamilly-Desktop`. Idem pour la Xbox, les hololens et le reste.

A l'intérieur de ce répertoire nous allons ajouter un fichier XAML, mais juste un fichier XAML par une page avec code-behind, et nous allons aussi l'appeler de la même façon que la Vue que nous voulons personnaliser.

Pour créer le fichier XAML seul il faut faire ajouter / nouvel item / Visual C# / Xaml View :



Pour nous aider dans la mise en page n'oubliez pas que Visual Studio propose un échantillon de plusieurs types de surface de Design. Dans le coin haut gauche on trouve un dropdown qui permet de choisir taille et résolution. Cela ne modifie pas votre code XAML mais l'affiche dans le contexte visuel que vous avez sélectionné. C'est hyper pratique ! Ici je vais choisir un affichage pour smartphone 5" :



En A on choisit l'orientation (portrait / paysage), en B la taille de la cible (résolution / taille) le résultat est une mise à la taille du code XAML de la page affichée, sans modification, et on peut donc se mettre à travailler en confiance sur ce que verra réellement l'utilisateur (ici un smartphone 5" en mode portrait).

Au passage on appréciera mes annotations façon gribouillis d'un gosse en maternelle, ça c'est du design sauvage, spontanté, une sorte de cri primal du design ☺

Pour le code XAML je ne fais que recopier celui de la `MainPage` dont je change les couleurs (fond de page et texte) ainsi bien entendu que la taille de la police qui était bien trop grande pour un si petit écran (et je la place en bold afin d'avoir une présence visuelle plus soutenue qui remplace l'effet de la taille 72 originale).

Si j'exécute mon application en mode "machine locale" (ou "simulation") j'obtiendrais toujours l'affichage bleu déjà capturé et montré plus haut. Mais si j'exécute en mode mobile 5" j'obtiens bien le résultat suivant :

Génial non ? Bien entendu la page de code-behind reste rigoureusement la même (bien que je pourrais aussi la personnaliser de la même façon) et s'il y a un ViewModel je pourrais me binder aux seuls informations et commandes que je souhaite montrer en mode smartphone.

Déconcertant de simplicité et puissant.

Les noms de fichiers par famille

On peut arriver au même résultat en renommant tout simplement les fichiers avec la même règle de nommage que celle des sous-répertoires. Par exemple pour cette Vue Xaml personnalisée j'aurais pu, au lieu de créer un sous-répertoire famille et d'y placer un fichier `MainPage.Xaml`, dupliquer le fichier `MainPage.Xaml` et le renommer `MainPage.DeviceFamily-Mobile.Xaml`.

Le résultat serait le même, rigoureusement.

En revanche n'utiliser pas les deux méthodes à la fois car là ça ne collera pas, vous créerez des ressources réellement dupliquées (deux fois `MainPage` pour le mode Mobile par exemple) et cela n'est pas plus possible qu'avant…

Les ressources aussi ? oui !

Les deux techniques sont utilisables pour les ressources aussi.

Bien entendu ici puisque j'ai deux pages XAML différentes pour la même Vue, je peux mettre une image "A" dans le mode desktop et une image "B" dans le mode smartphone même si elles ont des noms différents.

Mais imaginons que j'arrive à ne conserver qu'une page XAML pour la Vue `MainPage` mais que dans le cas d'un smartphone j'affiche une icône de smartphone et dans le cas du mode desktop un petit PC ?

Il me suffira d'avoir une image "desktop" pour mon mode par défaut et de placer une image de même nom mais avec un logo smartphone dans le sous-répertoire famille, ou bien de renommer l'image avec la seconde technique qui ne touche que le nom du fichier. Une page XAML deux images chargées contextuellement…

Les images ne sont qu'un exemple on peut ainsi disposer de tout type de ressources automatiquement adaptées au contexte (vidéo, son, …).

InitializeComponent

Ici il s'agit vraiment d'une feinte… Dès que vous avez créé un sous-répertoire ou une ressource "bis" et après avoir compilé votre code vous découvrirez en cherchant bien que le code généré pour `InitializeComponent` a légèrement changé…

Il devient ceci :

```
partial class MainPage : global::Windows.UI.Xaml.Controls.Page
   {
     [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Windows.UI.Xaml.Bui
ld.Tasks"," 14.0.0.0")]
       private bool _contentLoaded;


       /// <summary>
       /// InitializeComponent()
       /// </summary>

       [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Windows.UI.Xaml.B
uild.Tasks"," 14.0.0.0")]
       [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
       public void InitializeComponent()
       {
           this.InitializeComponent(null);
       }
```

```
        /// <summary>
        /// InitializeComponent()
        /// </summary>
        [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Windows.UI.Xaml.Build.Tasks"," 14.0.0.0")]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent(global::System.Uri resourceLocator)
        {
            if (_contentLoaded)
                return;

            _contentLoaded = true;


            if (resourceLocator == null)
            {
                resourceLocator = new global::System.Uri("ms-appx:///MainPage.xaml");
            }

            global::Windows.UI.Xaml.Application.LoadComponent(this, resourceLocator, global::Windows.UI.Xaml.Controls.Primitives.ComponentResourceLocation.Application);
        }
    }
```

Ce qu'on voit c'est qu'il existe une surcharge de la méthode qui désormais accepte une URI en paramètre.

Qu'est-ce à dire ?

Et bien que si vous ne faites rien c'est bien le mécanisme par défaut expliqué plus haut qui va se mettre en place (chargement de l'une ou l'autre version de la page Xaml selon la famille de la machine exécutant le code).

Mais puisque la surcharge existe, cela veut dire que si vous disposez de plusieurs pages alternatives pour une même page,
disons `PrincipalMainPage` et `AlternateMainPage`, vous pourrez facilement dans le constructeur de `MainPage` ajouter toute une logique pour décider de charger soit la page principale soit la page alternative (à l'intérieur d'un choix par famille, ce qui commence à faire pas mal de possibilités).

Par exemple :

```
public MainPage()
{
    if (AnalyticsInfo.VersionInfo.DeviceFamily </mark> "Windows.Mobile")
    {
        if (usePrimary)
```

```
        {
            InitializeComponent(new Uri("ms-appx:///PrincipalMainPage.xaml", UriKind.Absolute));
        }
        else
        {
            InitializeComponent(new Uri("ms-appx:///AlternateMainPage.xaml", UriKind.Absolute));
        }
    }
    else
    {
        InitializeComponent();
    }
}
```

Nous voici maintenant avec une application possédant une version Desktop par défaut parfaitement adaptée aux grands écrans de PC et possédant aussi un visuel adapté parfaitement aux unités mobiles qui peut même prévoir deux affichages différents (mobile de grande taille et mobile très petit par exemple…).

On voit que les procédés à notre disposition sont nombreux, logiques et cohérents sans pour autant rendre tout cela imbuvable. Très franchement les magouilles d'Android sur ce point précis me donnent la nausée presque autant que le trio infernal JS/CSS/Html, c'est pour dire ! Vous vous dites, tiens, depuis le début il n'a pas placé un petit truc sur Apple selon son habitude… Mais que voulez-vous que vous dise ? Dans un monde fasciste avec un seul modèle de smartphone à quoi servirait-il de s'adapter ? C'est le client qui s'adapte, pas le produit… Vous voulez vraiment que je vous dise ce que j'en pense ? Et bien non, car sur Dot.Blog on est toujours poli.

Les triggers d'état

Restez ! Ce n'est pas fini ! Il existe encore une autre façon de proposer une UI vraiment personnalisée par famille de machines sans pour autant créer des fichiers supplémentaires !

Oui, il est possible d'arriver au même résultat avec un seul fichier XAML. Parfois c'est rentable, parfois ça fait un gros fichier XAML ce qui n'est pas très maintenable, on a donc le choix des armes selon le contexte sachant que pour une Vue on peut utiliser un mécanisme et que pour une autre un second mécanisme. Je préfère les applications où le code est mis en œuvre de façon cohérente donc on choisira une approche et on s'y tiendra, mais rien ne l'oblige.

Voici un bout de code qui démontre la technique :

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">


        <VisualStateManager.VisualStateGroups>

            <VisualStateGroup >

                <VisualState x:Name="desktop">

                    <VisualState.StateTriggers>

                        <triggers:DeviceFamilyStateTrigger DeviceFamily="Desktop" />

                    </VisualState.StateTriggers>

                    <VisualState.Setters>

                        <Setter Target="greeting.Text" Value="Hello Windows Desktop!" />

                    </VisualState.Setters>

                </VisualState>

                <VisualState x:Name="mobile">

                    <VisualState.StateTriggers>

                         <triggers:DeviceFamilyStateTrigger DeviceFamily="Mobile" />

                    </VisualState.StateTriggers>

                    <VisualState.Setters>

                        <Setter Target="greeting.Text" Value="Hello Windows Mobile!" />

                    </VisualState.Setters>

                </VisualState>

                <VisualState x:Name="team">

                    <VisualState.StateTriggers>

                         <triggers:DeviceFamilyStateTrigger DeviceFamily="Team" />

                    </VisualState.StateTriggers>

                    <VisualState.Setters>

                        <Setter Target="greeting.Text" Value="Hello Windows Team!" />

                    </VisualState.Setters>
```

```
                </VisualState>
                <VisualState x:Name="iot">
                    <VisualState.StateTriggers>
                         <triggers:DeviceFamilyStateTrigger DeviceFamily="IoT" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="greeting.Text" Value="Hello Windows IoT Core!" />
                    </VisualState.Setters>
                </VisualState>
                 <VisualState x:Name="xbox">
                     <VisualState.StateTriggers>
                         <triggers:DeviceFamilyStateTrigger DeviceFamily="Xbox" />
                     </VisualState.StateTriggers>
                     <VisualState.Setters>
                         <Setter Target="greeting.Text" Value="Hello Xbox!" />
                     </VisualState.Setters>
                 </VisualState>
             </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <TextBlock x:Name="greeting" Foreground="Black"
                   Text="Unknown Platform"
                   HorizontalAlignment="Center" VerticalAlignment="Center" />
</Grid>
```

Ce code est extrait d'une démo postée sur Github, vous pouvez télécharger le projet complet pour le tester : WindowsStateTriggers.

Conclusion

Le concept de Reactive UI n'est qu'un concept. C'est quand il faut passer à la réalité, aux moyens mis à notre disposition qu'on voit si la solution proposée tient la route ou devient un enfer.

Sous Android on peut s'adapter sans problème à plein de choses, ce qui est un minimum dans un environnement qui est supporté par tant de machines différentes. Mais la solution proposée est archaïque, c'est bourrin, fastidieux. Franchement quand j'ai découvert ça au début où j'ai pratiqué Android j'ai été très déçu (les nine-patches m'ont achevé) je m'attendais à aussi moderne que peut l'être le look épuré, jeune et frais qu'on voit de l'extérieur. Et bien non. Je ne parle pas de Cocoa et Objective-C, rien que d'en parler j'ai des haut-le-cœur.

Encore une fois et quoi qu'on en pense ou dise, que le public adore ou non Windows, tout cela ne changera rien au fait que oui, toujours, Microsoft a 30 ans d'avance sur la concurrence. Ce n'est pas forcément payé par des succès fulgurants en retour, et c'est dommage, mais tout de même, que leurs produits sont fait avec intelligence…

Design adaptatif par triggers personnalisés

Je vous ai présenté les concepts de l'Adaptive Design et le rôle du nouveau trigger XAML permettant de réagir à la taille de l'écran. Mais on peut faire beaucoup plus créant ses propres triggers ! C'est toute une librairie que je vous propose de découvrir aujourd'hui…

L'AdaptiveTrigger et ses cousins

Je ne m'étendrai pas sur le sujet puisqu'un billet récent lui est consacré avec le RelativePanel. Pour faire court donc, ce trigger XAML permet de rendre un VisualState actif sans action extérieure (ni code C# ni déclenchement d'animation ou autre). Ce trigger fourni avec UWP permet de tester la hauteur ou la largeur en pixels effectifs de la fenêtre et donc de réagir automatiquement aux changements de ces valeurs en proposant généralement une mise en page différente.

Mais la taille écran n'est pas la seule à présenter un intérêt pour personnaliser une page XAML !

On peut aussi vouloir utiliser le concept même en dehors de *l'Adaptive Design*. Réagir à une propriété du ViewModel, à la Famille de device sur laquelle l'application tourne, etc…

C'est donc toute une famille de triggers qui se cache derrière l'AdaptiveTrigger !

StateTriggerBase

Derrière toute famille se cache une mère… La classe mère d'`AdaptiveTrigger` est `StateTriggerBase`. Et cette maman là est porteuse potentiellement de centaines de petits !

Le principe pour créer son propre `StateTrigger`, donc un trigger qui fait changer d'état visuel (géré par le `VisualStateManager` de XAML), consiste simplement à créer une classe enfant de `StateTriggerBase` et d'y coder ce dont on à besoin … Toutes les idées sont donc possibles.

Voici un exemple de code qui permet de créer un `DeviceFamilyTrigger`, c'est à dire un trigger qui permettra d'activer un `VisualState` automatiquement selon la famille de device Windows 10 sur laquelle le code sera exécuté :

```
public class DeviceFamilyTrigger : StateTriggerBase
{
    private string _deviceFamily;
    public string DeviceFamily
    {
        get { return _deviceFamily; }
        set
        {
            var qualifiers = Windows.ApplicationModel.Resources.Core
                .ResourceContext.GetForCurrentView().QualifierValues;
            if (qualifiers.ContainsKey("DeviceFamily"))
                SetActive(qualifiers["DeviceFamily"] == (_deviceFamily = value));
            else
                SetActive(false);
        }
    }
}
```

Trois règles :

1. L'héritage de `StateTrigger`
2. L'utilisation de `SetActive` pour activer (ou non) l'état visuel
3. Rendre publique une propriété à tester (famille, résolution..)

Tout cela est donc fort simple et ouvre la porte à de nombreuses possibilités dont vous saurez tirer le meilleur parti j'en suis convaincu.

Une librairie de StateTriggers

Un californien signant “DotMorten” a mis en ligne sur GitHub une librairie contenant de nombreux StateTriggers qu’il a écrits. Il est donc possible de profiter de ce travail offert à la communauté tout comme vous pourrez participer et ajouter vos propres créations ou forker la librairire.

On la trouve à cette adresse : https://github.com/dotMorten/WindowsStateTriggers

Triggers par Famille de device, par simple égalité de valeur, détection du passage en plein écran, du type de saisie utilisée, par expression régulière, par orientation d’écran, etc. Le choix est déjà assez vaste mais surtout, puisque c’est Open Source, cela donne un sacré coup de pouce pour écrire de nouveaux StateTriggers en disposant de nombreux codes exemples !

Conclusion

UWP vient avec de nombreuses bonnes idées qui elles-mêmes ouvrent la voie vers *plus de créativité*. C’est une plateforme *ouverte intellectuellement parlant*. Le tout se reposant sur XAML. C’est plus qu’on ne pouvait espérer pour ce langage après le passage à vide de la fin de Silverlight et les idées un peu coincées de WinRT qui portaient plus de contraintes que d’espoir.

UWP devient une plateforme aussi agréable et ouverte à l’imagination que le fut Silverlight et que l’est toujours WPF (dont l’utilité reste entière puisque tournant en mode non sandboxé et sur tous les PC n’ayant pas Windows 10).

Design UWP : des patrons gratuits pour Illustrator et PowerPoint

Design First… Je l’ai répété tant de fois ici… Dans le concret cela veut dire dessiner ses écrans avant de coder. Tester leur look, les transitions. Il y a des logiciels spécifiques pour le faire mais on peut aussi utiliser les standards du marché comme Illustrator ou PowerPoint…

Les patrons gratuits de Microsoft

En se rendant sur Design.Windows.com on découvre de nombreuses ressources pour le développement UWP. C’est un site dont je vous ai déjà parlé et un incontournable à visiter.

Si vous regardez bien en première page sur la droite on trouve un “Téléchargements de conceptions”.

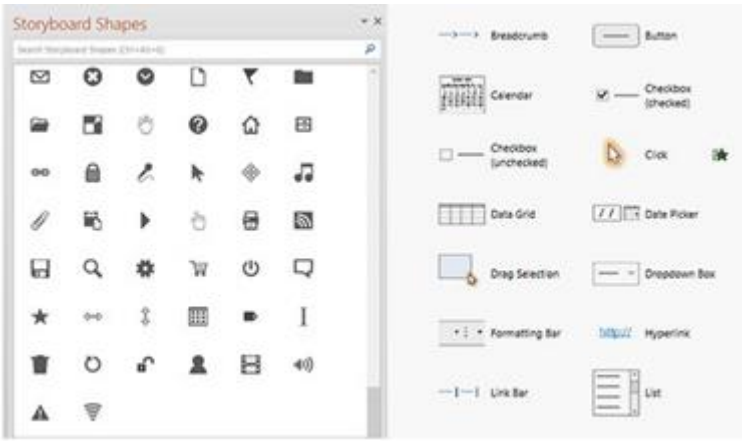
C’est mal traduit et peu parlant alors on peut passer et se jeter sur les autres liens plus aguichants (notion de base de conception, recommandations, exemples de code, téléchargements…).

Il faut tout visiter.

Car si vous cliquez sur ce lien vous obtiendrez plus que ce que vous imaginiez…

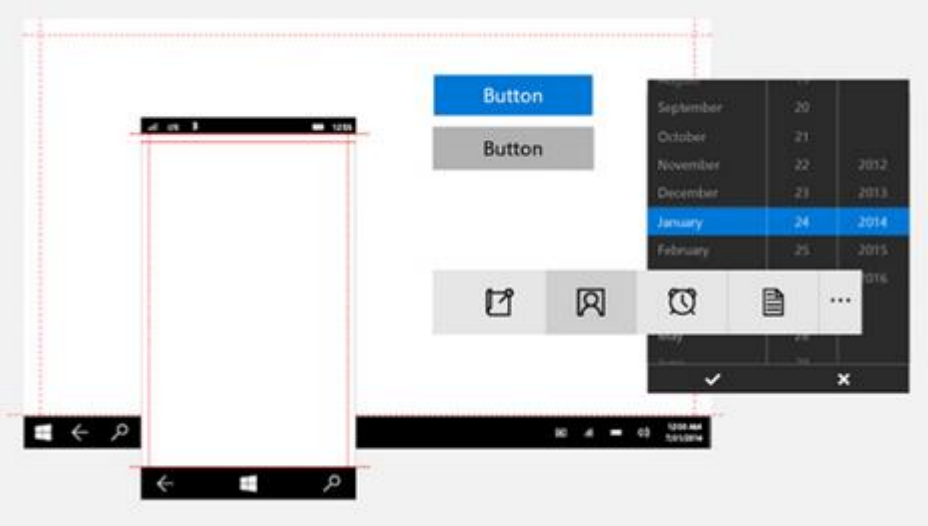
Les patrons PowerPoint

D’abord les patrons PowerPoint. Il suffit de les charger dans ce dernier pour disposer d’objets facilement utilisables pour simuler une mise en page UWP. Par rapport à la version Windows 8 tout a été mis à jour, c’est donc très utile et toute personne possédant Office peut donc participer à la mise en page d’une application. Très utile aussi lorsqu’on veut faire participer les utilisateurs afin qu’ils expriment leurs souhaits tout en goutant un peu les limites de ce qui est possible…



Les patrons Illustrator

Même principe ici mais “clientèle” différente.. Les patrons pour Illustrator se destinent aux designers qui savent ce servir de ce merveilleux logiciels vectoriel… Grâce aux patrons fournis on peut rapidement créer des ébauches d’UI qu’on peut ensuite faire évoluer vers un design final tout en restant dans le même outil.



Le profileur Illustrator

Il s'agit là d'une véritable extension pour Illustrator. Elle permet de créer automatiquement des calques sur lesquels sont notés les dimensions des objets par exemple, les couleurs utilisées, etc… Cela fonctionne avec CC 2014. Un outil très intelligent pour aider les designers à communiquer des informations essentielles aux développeurs sur leur design. Cela évite que ces derniers ne perdent leur temps à mesurer, à chercher le nom d'une fonte par exemple.

Blend sachant importer les fichiers Illustrator l'outil est encore plus pratique puisque ces informations peuvent être remontées durant l'importation. Et s'agissant de calques supplémentaires on peut les éteindre et allumer à volonté et puis les supprimer…

Conclusion

Designer des applications UWP n'est pas un jeu d'enfant, cela serait mentir. S'adapter avec un seul code à plusieurs familles de machines chacune possédant plusieurs form factors est en réalité un travail énorme qui demande des connaissances et une planification parfaite. Et il faut aussi des outils adaptés. Microsoft nous fourni ici quelques éléments intéressants qu'il serait idiot de négliger.

XAML : arbre visuel vs arbre logique sous WPF et UWP

WPF propose deux visions de l'arbre des objets visuels là où UWP ne propose de base qu'une seule approche. Mais quelle est la différence entre arbre logique et arbre visuel et comment s'en servir à la fois sous WPF et UWP ?

Arbre logique et arbre visuel

Les éléments d'une interface XAML entretiennent des liens hiérarchiques, une grille contient un StackPanel qui lui-même contient des boutons par exemple. Chaque conteneur possède des enfants visuels qui eux-mêmes peuvent avoir des enfants visuels, et ce, ad libitum (en tout cas tant que la mémoire le supporte !)

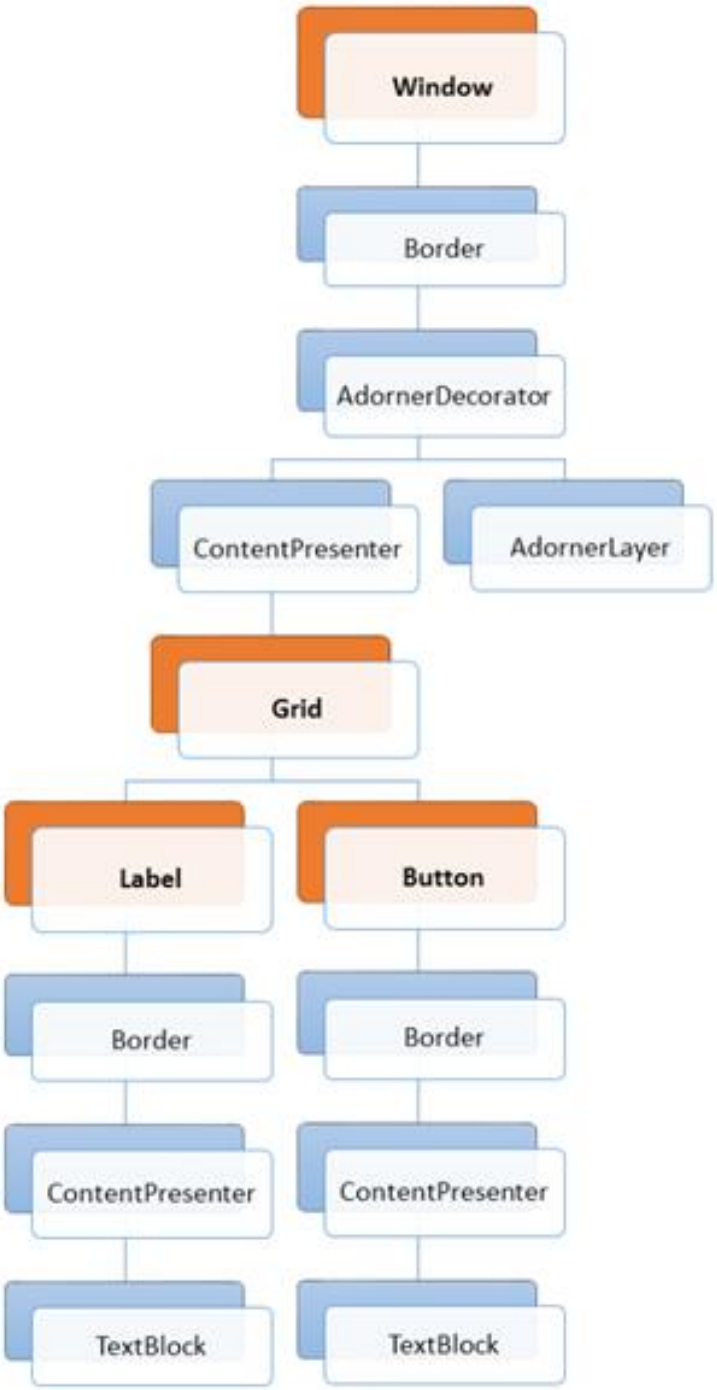
Cette organisation hiérarchisée des contrôles n'a rien d'exotique et tout le monde le comprend généralement bien.

Donc d'où vient cette nuance entre abre logique et arbre visuel ?

Prenons un exemple très simple de code XAML :

```
<Window>

    <Grid>

        <Label Content="Label" />

        <Button Content="Button" />

    </Grid>

</Window>
```

Une telle construction donne l'arbre suivant :



On voit que que le concepteur de cet affichage a utilisé une `Window` comme élément de base, c'est donc ici une page WPF, et qu'il a placé une `Grid` sur cette fenêtre et à l'intérieure de celle-ci un `Label` et un `Button`.

Mais ce que montre l'arbre de l'illustration c'est qu'il existe bien d'autres contrôles, pas tous visibles, qui constituent le véritable arbre des relations entre tous les contrôles. Ces contrôles sont généralement ce qui forme le Template de leur contrôle parent.

Ainsi un `Label` peut être templaté pour être formé d'un `Border` contenant un `ContentPresenter` qui enfin contient un `TextBlock`. De même le `Button` dispose lui aussi une hiérarchie visuelle similaire. Quant à l'objet `Window` il contient par exemple `Border` dans lequel est placé un `AdornerDecorator` etc…

On le comprend aisément il existe une différence entre l'arbre dit **Visuel** qui contient toute la hiérarchie dont les Templates et l'arbre dit **Logique** qui n'est fait que d'un Window, d'une grille et deux autres contrôles – le `Label` et le `Button`.

L'arbre Visuel c'est la totalité du schéma ci-dessus, l'arbre Logique n'est que la partie montrant les contrôles sur fond orange.

En quoi cette nuance est-elle utile ?

Je pense que vous pouvez trouver seul la réponse à cette question. Traverser l'arbre logique est plus conforme à l'idée que vous vous faites de l'UI que vous avez conçue : si vous n'avez placé qu'une grille avec un `Label` et un `Button` le tout sur une fenêtre « votre arbre des contrôles » tel que vous le concevez mentalement est l'arbre dit *Logique*. Si vous devez traverser l'arbre pour faire des traitements (rendre visible ou non certains contrôles par exemple) ce qui compte pour vous et votre code c'est de *parcourir l'arbre Logique*.

Si vous désirez atteindre les parties plus intérieures de chaque contrôle vous aurez alors besoin de balayer la totalité de l'arbre, donc l'arbre Visuel.

Rien ne sert de balayer tous les contenus de Template si on cherche uniquement à changer la couleur d'une série de boutons par exemple.

Mais la nuance entre les deux arbres n'est pas seulement qu'intellectuelle elle marque aussi une différence essentielle dans la façon dont XAML gère certaines opérations.

L'arbre Logique

C'est l'arbre du Designer pourrait-on dire, celui des contrôles de rang "supérieurs", ceux qu'on place sur la surface choisie pour créer une UI. Il décrit les relations entre les éléments et l'interface utilisateur. Cet arbre est responsable de nombreuses fonctions :

- L'héritage des valeurs des propriétés de dépendance (`DependencyProperty`)
- La résolution des références aux ressources dynamiques (`DynamicResources`)

- La recherche des noms d'éléments pour la gestion du `Binding`
- La transmission des évènements de type `RoutedEvents`

L'arbre Visuel

L'arbre Visuel peut être vu comme l'arbre "technique" celui qui dévoile toute la "plomberie" et toutes les relations sous-jacentes entre les contrôles et leurs templates. Cet arbre possède d'autres responsabilités que l'arbre logique :

- Le rendu des éléments visuels
- La propagation de l'opacité des éléments visuels
- La propagation des Layout et RenderTransforms
- La propagation de la propriété IsEnabled
- Le test de Hit

La résolution des sources relatives des Bindings (`FindAncestor`)

Traiter les arbres avec C#

Maintenant que nous savons qu'il existe deux types d'arbres et que nous en voyons l'intérêt reste à savoir comment traiter ces arbres de façon différente…

Pour cela le Framework .NET ou UWP mettent à notre disposition des classes Helper. Sous WPF, comme d'habitude, nous disposons d'un XAML complet accompagné d'un Framework qui l'est tout autant, c'est ainsi que nous pouvons utiliser aussi bien le VisualTreeHelper de `System.Windows.Media` ou bien le LogicalTreeHelper de `System.Windows`.

Sous UWP comme WinRT auquel il emprûnte énormément il faut se contenter du seul VisualTreeHelper de `Windows.UI.Xaml.Media`.

Avec WPF il est donc simple de traiter l'un ou l'autre des deux arbres puisque le Framework met à notre disposition des Helpers différenciés. Mais sous UWP il existe deux nuances : la première est l'absence de traitement de l'arbre logique, la seconde est que l'héritage des éléments visuels est légèrement différent puisque UIElement descend directement de `DependencyObject`. Sous WPF la chaîne

d'héritage est plus longue (`Object`, `DispatcherObject`, `DependencyObject`, `Visual`, et enfin `UIElement`.

Il reste néanmoins possible de simuler le fonctionnement de WPF sous UWP, mais malgré la taille gigantesque de UWP par rapport à .NET c'est à nous de rajouter du code ce qui est un peu un comble…

Pour ce faire on peut s'aider du `VisualTreeHelper de Windows.Ui.Xaml`. Et comme ce code n'est pas tout à fait pratique on peut le compléter pour le rendre plus directement utile :

```
IEnumerable<DependencyObject> GetVisualTree(DependencyObject obj)
{
    var list = new List<DependencyObject>() { obj };
    var res = obj;
    while ((res = VisualTreeHelper.GetParent(res)) != null)
        list.Add(res);
    return list.AsEnumerable().Reverse();
}
```

Si on imagine une `ListBox` posée sur notre surface de design et qu'on l'utilise comme base de notre recherche de l'arbre visuel en utilisant le code ci-dessus nous écrirons :

```
listbox.ItemsSource = GetVisualTree(listbox).Select(d => d.GetType().Name);
```

La listbox va alors se remplir de la liste de tous ces constituants (`ScrollViewer`, `Border`, `Grid`, `ScrollContentPresenter` etc).

Mais où est l'arbre logique dans tout ça ? Patience il arrive !

Sous WPF pas de question à se poser avec le Helper déjà fourni comme indiqué plus haut, mais sous UWP il faut le simuler ce qui donne le code :

```
// Simulation de WPF LogicalTreeHelper.GetParent

public static DependencyObject GetParent ( DependencyObject courant)
{
    if (courant == null )
    {
        throw new ArgumentNullException ("courant" );
    }

    FrameworkElement élément = courant as FrameworkElement ;
    if (élément ! = null )
    {
        return élément.Parent;
```

```
    }

    FrameworkContentElement element2 = courant as FrameworkContentElement ;

    si (element2 ! = null )
    {
        return element2.Parent;
    }
    return null ;
}



// Simulation du Helper WPF pour UWP

IEnumerable < DependencyObject > GetLogicalTree ( DependencyObject obj)
{
    var liste = new list < DependencyObject > () {obj};
    var RES = obj;
    while ((RES = GetParent (RES)) ! = null )
        liste.Add(RES);
    retur liste. AsEnumerable().Reverse();
}

DependencyObject GetParent ( DependencyObject obj)
{
    var s = obj as FrameworkElement ;
    if (s ! = null )
        return s.Parent;
    return null ;
}
```

Une fois `GetParent` simulé ou peut enfin écrire le `GetLogicalTree`. Et le tour est joué !

Conclusion

XAML nous offre des hiérarchies d'objets qu'on peut voir de deux façons, une logique et une visuelle. WPF et le Framework .NET donne un accès simplifié à ces deux arbres mais pas UWP. Toutefois en il est possible de simuler la fonction voire de s'en servir pour créer un code portable WPF/UWP.

La prochaine fois que vous souhaiterez parcourir l'arbre des objets d'une application XAML demandez-vous quel arbre vous devez balayer, et souvenez-vous de cet article. Bon accrobranche xamélien !

Blend pour Visual Studio 2015

Blend cet outil magnifique continue de s'adapter pour fournir l'EDI le plus puissant servant à concevoir des UI XAML. Quoi de neuf ?

Blend une petite merveille

Survivant de l'époque du bundle "Expression" où l'on trouvait le fantastique Expression Design, un mini Illustrator hélas sans suite, ou "'Expression Encoder" encodeur de vidéo pour les mobiles et le web, "Expression Blend" devenu "Blend" tout court puis "Blend for Visual Studio" est l'EDI XAML par excellence. L'outil unique pour créer des UI en gérant tous les aspects dont les *timelines* des animations entre autre.

Certes depuis "Expression Blend" l'éditeur visuel XAML de Visual Studio s'est considérablement amélioré lui aussi au point que certains pensent que cela est bien suffisant. Pour poser un bouton et faire un binding certainement. Pour ***concevoir une UI*** c'est faux.

Il semble de mes rencontres avec les gens que je forme ou les équipes de développeurs pour lesquelles j'interviens que Blend est considéré comme un "truc compliqué" et que peu de personnes le comprennent. Je trouve ça étrange, c'est limpide, puissant et surtout indispensable pour mettre en place une UI XAML, créer de nouveaux contrôles originaux.

Aujourd'hui intégré à Visual Studio (d'où son nom de "Blend for Visual Studio") le logiciel est installé avec VS, tout le monde peut donc en profiter (à l'époque du pack Expression il fallait une licence en plus).

Donc n'hésitez pas à jouer avec, surtout si vous désirez créer de vraies UI XAML et non pas des UI Windows Forms faites en XAML (ce que je vois le plus souvent et qui me déprime totalement) !

Dot.Blog est plein de bons conseils, d'articles forts longs et mêmes de livres consacrés à XAML, impossible de tout lister ici, je ne peux répéter que tout cela est à votre disposition (gratuitement et sans pub) pour comprendre comment utiliser XAML correctement. Et là Blend deviendra pour vous comme une évidence…

Nouveau Blend nouveau look

Pour un outil destiné à créer des UI modernes il semble naturel que son look soit lui aussi soigné… Et c'est le cas du nouveau Blend qui a un look sophistiqué et pur à la fois tout en améliorant son "utilisabilité" (UX).

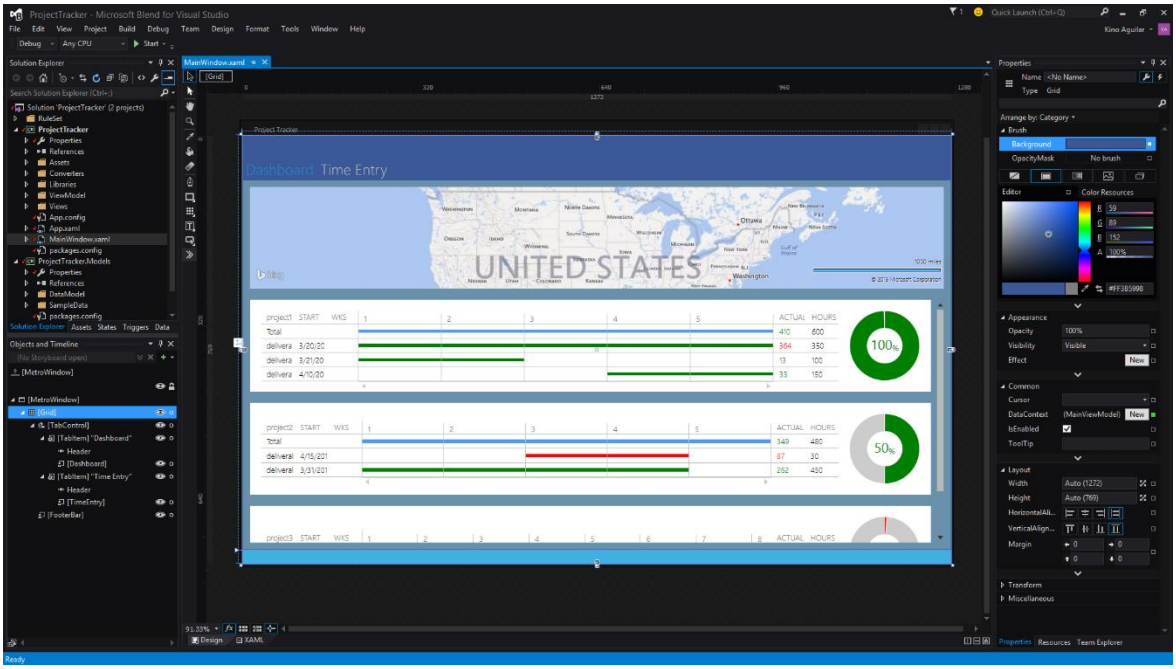
Il suit en cela le nouveau look de Visual Studio dont il est devenu dans les dernières versions un compagnon indispensable. D'ailleurs le fonctionnement Blend / VS en simultané a été largement amélioré car c'est une situation de travail courante.

Nouvelles fonctionnalités

Forcément il n'y a pas que le look qui s'améliore. Comme je le disais sa capacité à travailler en même temps que VS sur un même projet a été considérablement améliorée mais ce n'est pas tout. On trouve désormais un système IntelliSense performant pour le code XAML et des outils de debug XAML incroyables !

Une UI taillée pour le Design XAML

Le Design même de Blend est conçu pour faciliter le Design des applications XAML en donnant plus d'espace aux panneaux qui sont utilisés le plus par l'utilisateur, ce qui inclue les propriétés des objets les lignes de temps des animations (timelines). Le nouveau Blend est encore plus agréable à utiliser et surtout fournit un cadre **ultra productif** pour créer des contrôles ou des pages XAML.

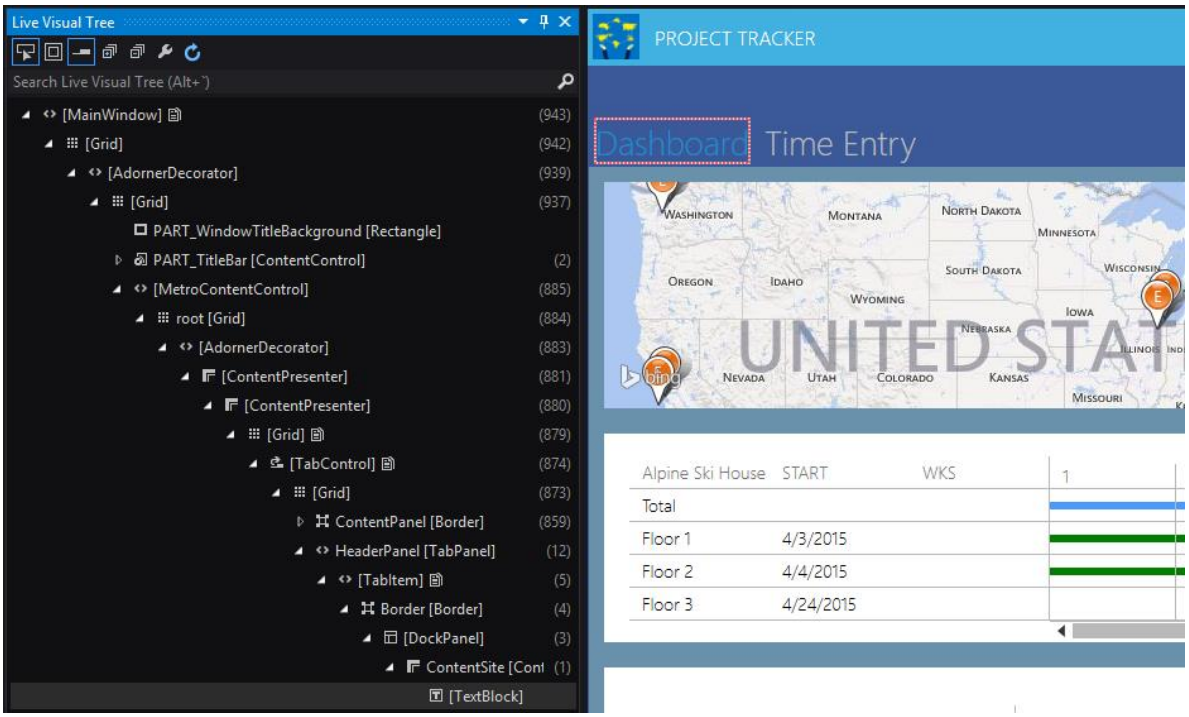


IntelliSense Roselyn

IntelliSense pour XAML c'est déjà une avancée énorme dans Blend (qui était plutôt orienté XAML que code). Mais désormais cette fonctionnalité est assurée par un nouveau service basé sur Roselyn qui est disponible dans Blend et VS.

Outils de debug

Blend intègre désormais des **outils runtime** pour inspecter XAML. Ces outils permettent d'inspecter l'arbre visuel d'une application en cours d'exécution ainsi que les propriétés des objets. C'est un must absolu lorsqu'on doit mettre au point des affichages sophistiqués. L'utilitaire "Snoop" proposé sur Codeplex perd donc de son utilité, même s'il peut rester complémentaire dans certains cas. Les nouveaux outils sont eux produit par l'équipe officielle Blend ce qui permet de penser qu'ils vont et iront bien plus loin au fil des versions.

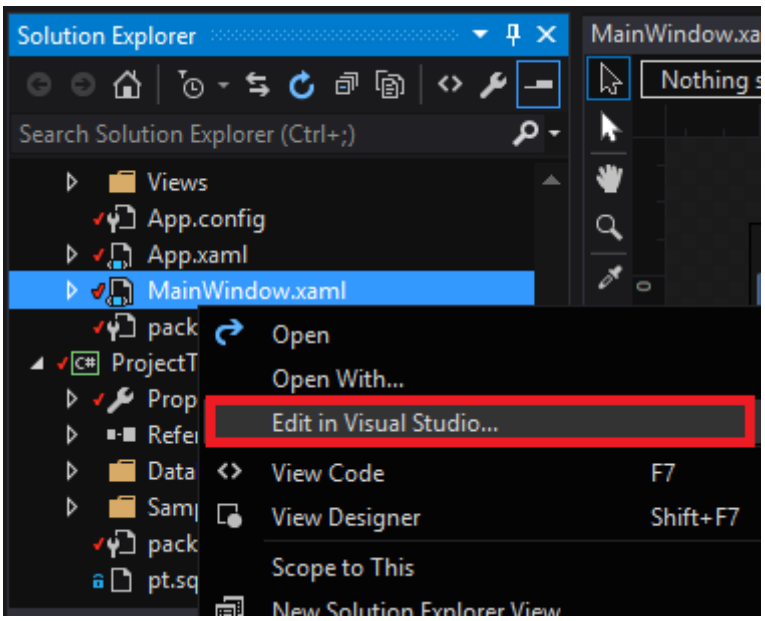


Vous pouvez en apprendre plus immédiatement sur ces nouveaux outils en lisant cet article (anglais) : introducing the UI debuggging tools for XAML.

Design dans Blend / Edit dans VS

Ce mariage Blend / VS était une obligation. Blend se concentre sur les aspects visuels et VS est un EDI unique pour la partie code. Concevoir par exemple un nouveau contrôle visuel implique d'avoir les deux logiciels ouverts en même temps. Cela était plus ou moins bien pris en charge, un switch entre VS et Blend forçait le rechargement de la page de travail en cours, mais ce n'était pas toujours fiable.

Désormais cette collaboration VS/Blend est totalement couverte grâce à des commandes plus directes comme “Design in Blend” côté VS et “Edit in Visual Studio” côté Blend. Le mécanisme de partage est aussi mieux géré ainsi que les allers-retours entre les deux EDI.



Accessibilité renforcée

Il est vrai qu’on a tendance à croire que quelqu’un qui fait des conceptions visuelles est forcément bien voyant… C’est faux… Le gout, le sens de l’UX et de l’UI n’est pas réservé à ceux qui ont 10/10 à chaque œil !

Microsoft fait beaucoup d’effort pour que ses logiciels soit accessible même aux personnes ayant des difficultés. Cet effort a été fait dans le nouveau Blend aussi. Le designer de Blend est désormais totalement couvert par le lecteur d’écran par exemple, et on peut le piloter aussi bien au clavier qu’à l’aide des outils d’accessibilité. D’ailleurs sans être malvoyant ces nouvelles options peuvent rendre de nombreux services ! Essayez-les vous “verrez” …

Conclusion

Je suis tombé amoureux de Blend en même temps que de XAML. Dès les premières versions ce logiciel m’a fasciné. Je n’ai jamais ressenti autant de plaisir que lorsque je développe un contrôle visuel, mi-Design, mi-Code, UI et C#, mélangés dans la plus belle des étreintes. Blend est le lit dans lequel cette étreinte s’épanouit… Rhââ Lovely comme disait le maitre de la BD, Marcel Gotlib !

(Pour votre éducation car certain lecteurs n'étaient pas nés en ces temps reculés, c'est une série de planches publiées dans les 70's regroupées dans trois tomes. Le titre « Rhââ lovely » provient de Frenzy d'Alfred Hitchcock. Gotlib, qui cherchait un titre à sa série, voulait éviter le cliché franchouillard et vulgaire du "*Ah ouais, c'est bon, c'est bon...*" . En allant voir Frenzy au cinéma, il retint la phrase que prononce le tueur dans le film lorsqu'il viole ses victimes).
Ce soir à l'apéro'merguez du camping vous pourrez de nouveau briller en société sans parler d'informatique, mais toujours grâce à Dot.Blog ! A la vôtre !

Passerelles

Xamarin : Le Million ! Le Million ! (et UWP)

J'ai déjà longuement parlé de Xamarin et des Xamarin.Forms avec MvvmCross ou sans, ici ou dans le cycle de 12 vidéos Youtube sur le développement cross-plateforme… Et c'est avec plaisir que nous savons maintenant que Xamarin passe le million de développeurs et annonce le support d'UWP !

Le Million !

Depuis la mise sur le marché de C# pour iOS en 2011, Xamarin a dépassé le million de développeurs ! C'est beaucoup et cela créé un écosystème qui n'a plus à pâlir devant les autres.

Des tas de nouveautés ont été annoncées et sont déjà releasées comme le support des Universal Apps de Windows 8.1 par exemple. Xamarin avance à grands pas depuis des années et leur partenariat avec Microsoft ne fait qu'accélérer les choses.

Car le meilleur reste à venir !

Support de UWP

Le mieux est encore de s'en remettre à l'annonce officielle (traduction en dessous) :

> *Availability of Private Previews of Xamarin.Forms for Windows 10 UAP*
>
> *In addition to the stable release of Windows 8.1 and Windows Phone 8.1, we're also announcing the start of a private preview of Xamarin.Forms for the Windows 10 Universal App Platform, enabling developers that build apps for all Windows platforms to share even more code. This is a very early preview and we'll be making the Xamarin.Forms for Windows 10 pre-release packages available to small batches of developers over the coming weeks. If you're interested in joining our preview program, please sign up here.*

"*Disponibilité des versions Preview de Xamarin.Forms pour Windows 10 UAP*

> *En plus des versions Windows 8.1 et Windows Phone 8.1, nous annonçons aussi le début de l'étape Preview des Xamarin.Forms pour Windows 10 Universal App Platform pour permettre aux développeurs qui construisent des applications Windows Platform de partager encore plus de code. C'est une version très précoce qui sera mise à la disposition d'un petit nombre de développeurs dans les semaines à venir. Si vous êtes intéressés inscrivez-vous (par le lien donné ci-dessus)."*

Donc Xamarin.Forms va intégrer UWP dans ses cibles ce qui signifie que nous pourrons développons une seule application pour toutes les plateformes.

Il va y avoir une sorte de compétition entre les différentes façons de faire du cross-plateforme / cross form factors… Du Xamarin.Forms qui supporte UWP, de l'UWP qui lui même supporte tous les form factors chez MS, un projet Android pur qui grâce aux passerelles MS pourra tout simplement être transformé en application UWP native automatiquement ?

Autant de routes à tester pour ne garder que celle où la visibilité est la meilleure et le péage le moins cher !

Conclusion

Le monde bouge, Microsoft bouge et Xamarin n'est pas en reste.

L'avenir s'annonce plutôt mouvementé avec des choix toujours pas faciles à faire, mais avoir le choix est déjà un bon signe de maturité du marché…

Astoria : Android sur Windows Phone 10, espoir ou danger ?

Microsoft a présenté la dernière version du projet Astoria, une bonne idée ? En tout cas techniquement c'est un joli coup !

Projet Astoria

Microsoft adore réutiliser les noms anciens. Le projet Astoria était d'une toute autre nature mais il s'agit d'une époque lointaine (c'était le nom de code de ADO.NET data service pour Silverlight en 2008) ! Aujourd'hui le projet Astoria est un système logiciel qui permet de transformer des applications Android en application native Windows Phone 10 …

C'est incroyable tellement la distance entre les deux OS est astronomique. Et pourtant ils l'ont fait !

Astoria est donc un pont entre deux mondes lointains, une façon pour Microsoft d'essayer de remplir son Store rapidement en vampirisant celui des autres… (le coup démago du HTML/JS n'ayant pas marché ce qui était à prévoir).

L'idée est futée et la réalisation technique est un joli tour de force. Dire que Microsoft invoquait des raisons techniques pour adapter Silverlight à Android ou iOS… alors qu'ils peuvent en quelques mois transformer une App Android en Windows Phone natif ! Ah la la, mauvaise foi quand tu nous tiens ! Mais n'en tenons pas rigueur à Nadella qui est un bon CEO pour MS, un type pragmatique et ouvert à qui on ne peut reprocher les fautes de son prédécesseur.

Mais tout n'est pas rose non plus au pays d'Astoria… Nombreux sont ceux qui pensent que cela va encore plus tuer le marché du développement Windows Phone. Qui voudra payer un développement spécifique faisant appel à des spécialistes Microsoft s'il suffit de faire l'application en Android puis de la porter magiquement avec un outil MS ? Le manque d'adaptation à Windows Phone ? Mais peu de gens s'intéressent à ces détails en fait.

Astoria bonne ou mauvaise idée alors ? Espoir ou Danger ?

J'aime personnellement l'esprit d'ouverture et le pragmatisme de Nadella. J'aime voir aussi qu'il tape tout azimut pour sauver le bateau du naufrage là où Ballmer et Sinofsky n'ont fait que faire des trous dans la coque en assurant l'air rigolard qu'ils avaient raison et que tout le monde avait tort…

J'entends aussi les craintes de certains MVP, d'autant mieux qu'au fond de moi je les partage "instinctivement".

Mais il faut rationnaliser.

D'une part Windows Phone plafonne à 2,7 % du marché mondial, c'est à dire que s'il ne perce pas rapidement maintenant Microsoft sera obligé d'arrêter. Ils ont déjà passé presque 7 milliards de dollars (!!!) de pertes dans leur bilan, le montant de l'achat de Nokia qui ne vaut donc plus rien… Et ils ont fait deux charrettes historiques de 18.000 et 8.000 salariés encore dernièrement principalement dans les équipes Nokia. Autant dire que l'outil de production n'existe plus et que le savoir-faire est parti avec ces plus de 26.000 personnes…

Franchement dans un tel contexte croit-on sérieusement que les spécialistes de Windows Phone en particulier ont un quelconque avenir ? Non bien sur. *En revanche dans un marché dynamisé par l'apport de nombreuses Apps il y a fort à parier qu'il y ait du boulot pour tous…*

Ayant grandi dans un milieu entre petite industrie et hôtellerie / restauration j'ai appris dès mon plus jeune âge quelques vérités qui s'acquièrent sur le terrain et pas dans les livres. Par exemple si vous décidez un jour d'ouvrir un restaurant de couscous, ne prenez pas un bail pas cher dans une petite rue même très proche d'une grande artère : installez-vous juste là à côté des autres restos à couscous. Pourquoi ? Parce que même pas cher votre petite rue isolée personne ne s'y rendra et vous coulerez rapidement. Alors que le client lui raisonne de la sorte "*tiens si on mangeait un cousous ? Allons rue machin il y en a plein on choisira !*". Et hop ! L'assurance de trouver son bonheur dans un endroit précis donnera du boulot à tous… Ce qui est valable pour le couscous l'est aussi pour les pizzérias, les restos grecs ou français *mais aussi pour les smartphones* !

Si vous devez acheter un smartphone, vous vous dites "*je vais prendre l'OS où je suis sur de trouver des Apps qui me conviennent*". Si vous êtes musicien vous prendrez iOS, les meilleurs Apps y sont et nulle part ailleurs. Seule ma vieille haine farouche et justifiée pour Apple fait que je n'ai pas d'iPhone ou d'iPad mais je le paye tous les jours en voyants d'autres compositeurs s'éclater avec des Apps qui n'existent pas sur Android et encore moins sur Windows Phone. Je le vis bien quand même ne vous inquiétez pas, *sans Apple la vie est plus fun* ! Et si vous n'avez pas ce genre de besoin, entre Android et Windows phone l'affaire est entendue de toute façon à la vue des Stores et de ce qu'ils proposent…

Il semble donc *essentiel de faire exploser le Store Windows Phone* de telle façon à ce que cette plateforme ne soit pas arrêtée dans les 12 ou 24 mois à venir, car ne ne

nous mentons pas, quand après autant d'années de présence on n'a pas atteint 15% d'un marché, on disparait.

> Ainsi, réflexe humain mais mesquin, au lieu d'avoir peur de se faire piquer un beefsteak qui n'existe pas essayons avant tout de faire de Windows Phone un succès, on discutera après pour se partager le gâteau…

A mon sens Astoria est donc porteur d'espoir plus que de danger. Le danger est déjà là, on ne risque rien de ce côté là… L'espoir en revanche c'est un papillon léger qui ne fait que passer, il faut le saisir rapidement !

Comment ça marche ?

Quand la version finalisée sera disponible il y a fort à parier que j'en parle plus en détail. Mais si vous voulez savoir tout de suite comment ça marche voici deux sources intéressantes :

Microsoft
Autant se renseigner à la source et MS a mis en place un site qui explique le projet Astoria. C'est instructif bien qu'en anglais (je dis ça pour les réfractaires à la langue de la perfide Albion !).

Le web
Astoria intéresse du monde et on le juge aux articles qui sont publiés hors du circuit officiel de Microsoft et de ses "proches". Par exemple cet article "How to install Android APK on Windows 10 Mobile" est une sorte de démo pas à pas des étapes actuellement nécessaires pour passer un APK en Windows 10 Mobile. On peut trouver d'autres articles de ce type c'est certains et même des vidéos sur Youtube vraisemblablement.

Conclusion

Le plus grand danger pour Windows Phone c'est que les choses restent telles qu'elles sont, une stagnation largement en dessous du seuil de rentabilité qui ne peut mener qu'à l'arrêt pur et simple de la plateforme.

Astoria, comme d'autres initiatives telles que UWP vont à mon avis dans le bon sens : redonner vie et couleur à un moribond dans un dernier sursaut d'espoir. Windows 10 et sa mise à jour gratuite n'est là que pour ça : tenter de rendre Windows Phone si ce

n'est incontournable au moins envisageable. La question n'étant plus "*Pourquoi développer pour Windows Phone ?*" mais plutôt "*Pourquoi pas ?*" … Microsoft est prêt à donner Windows 10 gratuitement à toute la planète dans cet unique et seul but.

Se voiler la face sur la réalité du marché, se protéger du vol d'un bien qu'on ne possède pas, tout cela est irrationnel. J'aime les choses claires, rationnelles et j'ai horreur des non-dits qui mènent au gouffre. Nadella prend de bonnes décisions alors,

Au sens réel qu'on oublie parfois :

(que) Vive Astoria ! (que) Vive Windows Phone !

Avertissements

L'ensemble de textes proposés ici est issu du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.

Les billets ont été collectés en septembre 2015 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile. Corrections, ajouts, vérifications des liens, tout cela a représenté un travail important.

Les textes originaux ont été écrits en 2015 pour cette édition spéciale UWP.

E-Naxos

E-Naxos est au départ une société éditrice de logiciels fondée par Olivier Dahan en 2001. Héritière de *Object Based System* et de *E.D.I.G.* créées plus tôt (1984 pour cette dernière) elle s'est d'abord consacrée à l'édition de logiciels tels que la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme par exemple MK Query Builder (requêteur visuel SQL).

Peu de temps après sa création E-Naxos s'est orientée vers le Conseil et l'Audit puis s'est ouverte à la Formation et au Développement au forfait. Faisant bénéficier ses clients de sa longue expérience dans la conception de logiciels robustes, de la relation client, de la connaissance des utilisateurs et de l'art, car finalement c'en est un, de concevoir des logiciels à la pointe mais maintenables dans le temps.

C#, Xaml ont été les piliers de cette nouvelle direction et Olivier a été récompensé par Microsoft pour son travail au sein de la communauté des développeurs WPF, Silverlight et Windows Plateform. 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 10) sans oublier Windows Phone.

A l'écoute du marché et offrant toujours un conseil éclairé à ses client, E-Naxos s'est aussi spécialisée dans le développement Cross-Plateforme, notamment dans le mariage des OS Microsoft avec Android, les deux incontournables du marché d'aujourd'hui et de demain.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares, surtout celles dont on peut vérifier la validitée avant de passer commande ! Plus de 820 articles, 10 Tomes All.Dot.Blog, des vidéos Youtubes, sont autant de preuves indiscutables d'une connaissance approfondie, d'un savoir faire et d'un savoir dire uniques.
Alors faites confiance à ceux qui prouvent qu'ils savent plutôt qu'à ceux qui ne font que vous le promettre…

Une demande d'information ? N'hésitez : postmaster@e-naxos.com !