« Notre métier connait sans cesse des rebondissements.

Aimer l'informatique c'est aimer le changement.

Qui aurait pu prévoir il y a quelques années que de presque 100% de parts de marché l'iPhone serait réduit à un îlot de résistance seul à tenir face à un simple moteur de recherche qui occuperait plus de 84% de ce même marché avec un Linux modifié ? ... »

Avec les Xamarin.Forms Microsoft nous offre le moyen le plus rapide de rentabiliser les compétences .NET en entreprise pour développer avec un seul code et une seule UI des applications ciblant tous les mobiles du marché et les PC. C'est une solution unique, gratuite et pacificatrice en ce qu'elle évite d'avoir à choisir et de risquer de se tromper.

C'est à la découverte du développement avec les Xamarin.Forms que l'auteur, Microsoft MVP, vous entraîne tout au long du présent ouvrage.

C#, XAML, MVVM, les Xamarin.Forms, vous aurez accès à de nombreux exemples fonctionnant à la fois sous Android, iOS et UWP (smartphones, tablettes et PC).

Un livre Ecrit par **Olivier Dahan**.

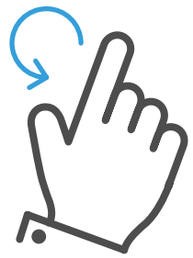
Éditions E-NAXOS, Paris - www.e-naxos.com

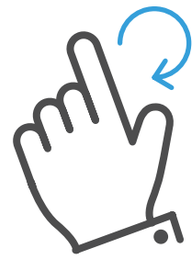
Développez en natif sous .NET pour Android, iOS, Universal Windows (UWP) avec Microsoft Xamarin.Forms

Développez en natif sous .NET pour **Android, iOS et Universal Windows (UWP)** avec **Microsoft Xamarin.Forms**

Olivier Dahan
Microsoft MVP

Olivier Dahan
Microsoft MVP





« Développez en natif .NET pour Android, iOS & UWP avec Xamarin Forms »

www.e-naxos.com\Livre.aspx

Un livre écrit par Olivier Dahan (odahan@gmail.com) et publié par e-naxos (www.e-naxos.com)

Extrait de L'Ouvrage – Chapitre 13

Téléchargez le code source du livre à cette adresse : http://www.e-naxos.com/Livre.aspx

Chapitre 13 – Animations, Transitions et Styles 386
Les animations fonctionnelles 386
Les ViewExtensions 389
La classe Animation 397
Les AnimationExtensions 400
Application exemple 400
Styles & Templates : *à lire dans le livre !*
Les Effets *à lire dans le livre !*
Pour résumer *à lire dans le livre !*

TOUS DROITS RESERVÉS. TOUTE DISTRIBUTION, DUPLICATION, UTILISATION, DE QUELQUE NATURE QU'ELLE SOIT ET QUELS QUE SOIENT LES PROCEDÉS UTILISÉS EST INTERDITE SANS L'AUTORISATION ÉCRITE DE L'AUTEUR. CE CHAPITRE OFFERT NE PEUT ÊTRE UTILISÉ QUE PAR VOUS POUR VOTRE USAGE PERSONNEL. NE LE COPIEZ PAS, NE LE TRANSFÉREZ PAS (aiguillez les personnes éventuellement intéressées vers le lien officiel où vous avez téléchargé le fichier).

Sommaire du Livre

Introduction 1

L'auteur 3

Chapitre 1 – Présentation des XForms 5

L'acquisition par Microsoft 5

Un mythe devient réalité 5

Xamarin 3.0 entre évolution et révolution 6

Designers visuels iOS et Android pour Visual Studio 7

Les XForms 7

Les pages 8

Extensibilité 10

MVVM 10

Binding 10

Fluidité 10

Messages 10

Un visuel animé 10

Tout en XAML ! 11

Xamarin 4.x La continuité 11

Quelles cibles choisir ? 12

Pour résumer 14

Créer une App 14

Chapitre 2 – Les bases 31

L'environnement 33

Les émulateurs 33

De quoi est fait une application XForms ? 34

Architecture du code 36

Espaces de noms 38

Initialisation des XForms 39

Symboles natifs et profil .NET 41

Gérer les spécificités des plateformes 42

Le positionnement des objets 46

Couleurs et thèmes 48

Les fontes et le texte 50

Les unités de mesure 55

Pour résumer 59

Chapitre 3 – XAML avec les XForms 60

Introduction 60

UI en C# ou en XAML ? 61

Les bases de XAML 62

Les extensions 65

Pour résumer 74

Chapitre 4 – Le Binding 75

Le Binding Xaml, Ange ou Démon ? 75
Le Binding 75
Définition 75
Utilisations 76
Schéma du principe 77
Déclaration 77
Les modes de binding 80
Hiérarchie de valeur 83
Règles de précédences 84
La notion de BindingContext 86
Les convertisseurs de valeur 86
Les dangers du Binding 95
Les syntaxes du Binding 99
Les propriétés de dépendance 107
Pour résumer 117

Chapitre 5 – Triggers et Behaviors 118
Les Behaviors 118
Créer un Behavior 120
Exemple : valider la numéricité d'une saisie 120
Behaviors avec propriétés 123
Attached Behaviors ou Bindable Behaviors 129
Autres exemples 132
Les Triggers 133
Pour résumer 145

Chapitre 6 – MVVM 146
Model-View-ViewModel aujourd'hui 146
Diviser pour mieux régner 149
Les instruments de MVVM 150
INotifyPropertyChanged 151
Les Commandes 156
Inversion de Contrôle 162
Injection de dépendances et Localisateur de Services 163
Les Messages MVVM 168
Pour résumer 175

Chapitre 7 – Mvvm Light 176
Mvvm Light, MvvmCross et Caliburn.Micro 176
Les domaines couverts 178
Utiliser Mvvm Light 180
Pour résumer 198

Chapitre 8 - Les Pages 199
NagigationPage 200
ContentPage 201
Les page Maître / Détail 201

Les pages à onglets 207
La Datapage 216
Les pages et les dialogues modaux 217
Pour résumer 219

Chapitre 9 - Les Layouts 220
Le StackLayout 222
L'AbsoluteLayout 226
Le RelativeLayout 237
La Grid 245
La ScrollView 253
Layout personnalisé 255
Pour résumer 263
View et ses descendants 264
Les vues simples 267
La SearchBar 271
Entry et Editor 272
Le Picker 273
La ListView 281
Les Cells 293
La TableView 298
La CarouselView 301
L'intégration de contrôles natifs 302
Pour résumer 302

Chapitre 11 – La Navigation 304
Naviguer 304
Naviguer en MVVM 305
Navigations horizontale et verticale 307
Navigation MVVM 309
Naviguer avec Mvvm Light 310
La navigation par lien profond 326
Pour résumer 328

Chapitre 12 – Fichiers et Données 329
PCL Storage 329
Gérer le cycle de vie et les données d'état 346
Base de données 356
Pour résumer 384

Chapitre 13 – Animations, Transitions et Styles 386
Les animations fonctionnelles 386
Les ViewExtensions 389
La classe Animation 397
Les AnimationExtensions 400
Application exemple 400
Styles & Templates 405

Les Effets 412
Pour résumer 414
Le mot de la fin 415

(Index page 416)

Chapitre 13 – Animations, Transitions et Styles

Bien que le XAML des XForms ne fournisse pas de notions tel que le `StoryBoard` toute une infrastructure de gestion des animations est disponible. Certains exemples de ce livre ont utilisé de telles techniques mais sans les expliciter.

Si on ne doit jamais abuser des animations, a fortiori sur de petits écrans, et si elles doivent avant tout servir l'UX sans transformer l'application en « sapin de Noël » elles peuvent ajouter ce petit supplément visuel qui fait la différence.

Les OS mobiles proposent déjà certaines animations pour les transitions de page ou le swipe dans une liste, ce que les XForms proposent c'est d'ajouter des animations à des contrôles visuels hors ces automatismes.

Dans ce livre je vous ai présenté le `ClickLabel`, un Label amélioré qui gère le clic (ou le tap). Ce contrôle personnalisé fonctionne parfaitement mais lorsque l'utilisateur le sollicite il ne se passe rien visuellement. Ce manque de retour visuel est gênant, ou pris dans l'autre sens ajouter un tel retour améliorerait les sensations de l'utilisateur et lui permettrait de savoir que son geste a été pris en compte évitant une anxiété même légère et souvent non consciente. C'est ce genre d'animations qui rendent l'UX meilleure. C'est du design, c'est-à-dire l'étude de ce qui permet de rendre intelligible une UI et de guider, rassurer aussi, l'utilisateur.

Souvent les animations sont à tort classées dans la catégorie « gadget » ce qui laisse sous-entendre une certaine inutilité, une perte de temps, un coût non justifié. Or l'UX n'est en rien un gadget ni un luxe, c'est juste ce qui fait qu'une application sera plaisante à utiliser … ou pas…

L'infrastructure d'animation des XForms n'est pas intégrée à XAML, elle se pilote par code. Mais comme l'a montré le chapitre « Triggers et Behaviors » il est tout à fait possible d'encapsuler des comportements complexes, dont les animations, dans ces déclencheurs ou comportements qui eux se manipulent en XAML. Un effet visuel souvent utilisé pourra ainsi être codé sous la forme d'un déclencheur d'action ou d'un Behavior pour être réutilisé directement en XAML et participer à une programmation totalement déclarative de l'UI.

Les animations fonctionnelles

Il est intéressant de connaître le concept d'animation fonctionnelle lorsqu'on crée le design d'une application.

Une animation fonctionnelle est une animation subtile que l'on place dans une application pour servir un but. A la différence des animations de type dessin animé ou jeux sur écran elles ont un rôle fonctionnel et une utilité clairement établie et logique. Leur but est de servir l'UX, le design global de l'application et font partie de la panoplie du créateur d'UI.

Si l'art et le design était des sciences exactes il serait facile de valider le côté fonctionnel d'une animation sans aucun doute possible. La réalité est plus nuancée et parfois il faut se questionner longuement pour trancher si telle ou telle animation sert véritablement le design et l'UX. Mais lorsqu'on doute c'est déjà que cela n'est pas évident… Et il est préférable de supprimer tout ce qui est de trop.

Lorsqu'on teste une partie d'un visuel on ne fait que ça et on peut aussi se laisser entraîner par l'aspect agréable d'une animation et un peu d'autosatisfaction aussi. Or il ne faut jamais oublier le contexte dans lequel l'utilisateur percevra l'animation. Si cela lui fait perdre du temps dans le processus complet il s'agacera rapidement. Il est donc essentiel d'avoir en permanence le recul nécessaire pour juger de l'à propos d'un embellissement graphique quel qu'il soit mais plus encore d'une animation qui par nature est visuellement invasive.

Les animations fonctionnelles jouent aussi un rôle de guide. De bonnes animations peuvent simplifier la prise en main d'une application et éviter de fournir une page d'aide énorme. Les Apps sur mobile ont définitivement enterré le concept de « manuel d'utilisation » qui coutait cher à produire et que personne ne lisait. Mais cela n'est pas juste une économie ! L'effort à fournir en compensation est plus important peut-être et il se déplace dans le design qui doit être auto-explicatif.

Par exemple imaginons un bouton qui ouvre un panneau. Lorsque l'utilisateur clic sur le bouton le panneau s'ouvre. Et quand il clique à nouveau il se ferme. Si tout cela est instantané l'utilisateur aura du mal à associer ses actions car dans la vie réelle rien n'est instantané et qu'en tant qu'être humain nous vivons dans le réel et non dans les circuits d'un ordinateur, sauf dans les films de science-fiction. Si j'ouvre une porte même violemment cela prend un certain temps. Ce n'est pas immédiat. Si je passe devant un détecteur et que les portes coulissantes de l'hôtel s'ouvrent ce n'est pas une transition sèche et immédiate. Le fait de voir ce qui se passe nous aide à comprendre le monde dans lequel nous évoluons. Et lorsque nous ne comprenons plus un stress apparait. Eviter de stresser vos utilisateurs…

Ainsi pour le bouton et le panneau si ce dernier se déploie par une animation et se rétracte dans un mouvement logique et naturel l'utilisateur n'aura pas besoin d'intellectualiser son interaction avec le programme pour comprendre ce qu'il fait. Il le sentira naturellement, instinctivement, car cela aura suivi le processus naturel du monde réel. Ce processus est inconscient mais il existe. Et il crée le plaisir d'utiliser une App plus qu'une autre, ou le déplaisir.

Dans ce cas l'animation du panneau sera bien fonctionnelle et servira un but clair visant à une meilleure UX.

Tout cela se rapproche un peu de la *Cognitive Friction* décrite par Alan Cooper père du Visual Basic avant qu'il ne le cède à Microsoft. Sujet qu'il a abordé il y a déjà longtemps dans son livre que je vous conseille ardemment si vous pouvez vous le procurer « *The Inmates are running the Asylum* », qu'on pourrait traduire par « les fous gèrent l'asile ». J'en ai parlé dans l'une de mes conférences consacrées au design dont une variante enregistrée est visible sur ma page YouTube. Une bonne animation supprime une possible Cognitive Friction ou plus positivement encore ajoute de la compréhension instinctive qui l'évite.

Une animation fonctionnelle peut aussi servir à placer des *hints* visuels pour aider ou guider l'utilisateur, tout comme elles peuvent accompagner visuellement en dirigeant l'œil vers ce qui est important à un moment donné. Le son peut aussi jouer un rôle important mais en général les utilisateurs n'aiment pas le côté trop intrusif d'une application qui se fait trop entendre par des bips ou des musiques ! Tout repose donc sur le visuel mais avec souplesse et subtilité.

Autre animation fonctionnelle : le retour visuel. C'est ce qui sera exploité à la section suivante pour ajouter au `ClickLabel` un retour visuel lorsqu'il est sollicité. Il est très important que l'utilisateur puisse être rassuré sur la suite qui est donnée à son interaction avec l'application.

On peut aussi citer les animations qui avertissent de l'état du système, changement de couleur de fond, mise en mode « éteint » de certains contrôles qui ne peuvent momentanément plus être utilisé (le fameux `CanExecute` des `ICommand` si important pour ces raisons).

Il y a donc une nécessité absolue de coller à la réalité, à la souplesse organique des mouvements réels qui ne sont jamais instantanés, au temps nécessaire même court qu'il y a pour que quelque chose se passe. Il ne s'agit en aucun cas de plonger dans le skeuomorphisme qui a fait son temps. Coller à la réalité est devenu plus fin, plus évanescent et se situe dans le comportement, le mouvement, et non plus dans l'apparence des choses.

Trois grands principes doivent être retenus lorsqu'on parle d'animation et de design pour l'informatique :

La « physicalité » Les objets virtuels doivent être ressentis comme réels, ils doivent avoir une masse, une consistance, de telle façon à ce que l'être humain habitué au réel puisse anticiper ses interactions comme il le fait sans cesse avec le monde réel. Il faut aussi qu'il obtienne un retour qui lui fasse ressentir l'objet virtuel comme un objet réel. Il doit pouvoir intégrer le fonctionnement de l'application à son système de perception et non pas devoir s'adapter à un monde irréel ne répondant pas aux lois usuelles de la physique. Bien entendu cela s'entend à la suite de tout ce que je viens de dire, notamment qu'il ne s'agit en aucun cas de skeuomorphisme.

L'orientation visuelle Les animations notamment celles de transition doivent être des agents intercesseurs entre les pages, les différents états de l'application. L'utilisateur doit pouvoir suivre visuellement le flux de cette dernière sans être déconcerté. Ces animations doivent aussi servir de guide, d'aide à la compréhension du processus en cours.

La personnalité L'être humain est particulièrement doué pour reconnaître des patterns, tellement que lorsqu'il n'y en a pas il s'en invente (comme reconnaitre des formes animales dans les nuages – la *paréidolie*). L'aspect et le mouvement sont très rapidement caractérisés par le cerveau. Cela peut être utilisé pour donner une personnalité à l'application. Choisir des animations très rapides donnera par exemple une sensation de vivacité et de jeunesse, des animations plus douces un sentiment de quiétude, d'assurance, etc. En jouant sur l'aspect et le mouvement une application peut devenir reconnaissable et transmettre aussi une impression sur ce qu'elle est.

Si Microsoft a dû réviser son approche maladroite de Windows 8 et son mode de fonctionnement qui faisait d'un PC une grosse tablette pour rétablir le bureau classique dans Windows 10 ce n'est pas sans raison, ni même sans des grincements de dents et un changement de l'équipe dirigeante, ce qui n'est pas rien pour une telle entreprise… C'est une leçon à retenir, on ne clame pas contre

les utilisateurs qu'un design est génial. **Un bon design est un design que les utilisateurs aiment**. S'ils ne l'aiment pas, peu importe ce que vous en pensez, c'est un mauvais design…

Il ne faut pas hurler en sautillant sur une scène « développeurs, développeurs, développeurs ! » comme le fit Ballmer mais plutôt « users, users, users ! ». Que cela soit votre devise quand vous créez une App !

Les ViewExtensions

Les `ViewExtensions` sont comme le nom le semble supposer des extensions pour les Vues. Extension étant à comprendre au sens C# du terme, c'est-à-dire celui des méthodes d'extension. Ces méthodes dédiées aux animations s'appliquent ainsi à toutes les views au sens XForms, c'est-à-dire à tous les éléments visuels comme le bouton ou le label.

La classe ViewExtensions définit de nombreuses animations « prêtes à l'emploi ». C'est la façon la plus rapide et la plus simple de créer des animations. D'autres moyens existent et sont décrits plus loin dans ce chapitre.

Les méthodes offertes sont les suivantes (par ordre alphabétique) :

- `CancelAnimation`, pour stopper une animation en cours
- `FadeTo` pour animer l'opacité d'un contrôle
- `LayoutTo` anime le rectangle définissant les limites du contrôle
- `RelRotateTo` pour faire tourner le contrôle
- `RelScaleTo` pour animer la taille du contrôle
- `RotateTo` pour animer la rotation du contrôle
- `RotateXTo` anime un skew sur l'axe Y
- `RotateXTo` anime un skew sur l'axe X
- `ScaleTo` pour animer un changement de taille
- `TranslateTo` pour animer une translation sur X et Y

Certaines animations sont en double mais avec le préfixe `Rel` comme Relatif. Cela s'avère très utile lorsqu'une animation est lancée plusieurs fois de suite et qu'il n'est pas agréable visuellement qu'elle reparte systématiquement du départ. Souvent on préfère que l'animation débute là où la précédente s'est arrêtée. Dans ce cas les animations relatives viennent au secours du développeur en prenant en charge non plus une transformation absolue et rigide d'un point fixe à un autre point fixe mais en partant de l'état actuel du contrôle et en appliquant une animation relative à cet état. On peut par exemple faire une animation de la rotation par `RotateTo`, il s'agira d'une transformation impérative. En utilisant `RelRotateTo` il sera en revanche possible, partant de la rotation actuelle du contrôle, d'animer un ajout ou une soustraction d'un nombre de degrés quelconque. Le contrôle ne repartira pas de sa position de départ mais poursuivra son mouvement depuis son état en cours.

Les extensions d'animation retournent des `Task<bool>`, cela signifie qu'elles sont asynchrones et peuvent être attendues par le code sans bloquer le thread principal. Seule la méthode d'annulation `CancelAnimation` est immédiate.

De la même façon et sauf cette méthode particulière toutes les animations prennent trois paramètres, le premier est de nature variable mais sert toujours à préciser la quantité à animer (opacité, taille…), le second sert à préciser la durée en millisecondes, et le troisième permet d'ajouter un easing, c'est à un dire une fonction qui modifie la vitesse de l'animation dans le temps. On parle souvent d'effets *organiques*, c'est-à-dire d'animations qui ne sont pas purement linéaires dans le temps ce qui donne un rendu trop mécanique. Le côté organique ou naturel de ces transformations permet par exemple un amorti en fin de course, ou une accélération, un rebondissement, un effet élastique, etc. Cela existe dans le XAML standard et fonctionne dans les XForms selon le même principe. Sauf le premier, les deux paramètres de vitesse et de easing sont optionnels, des valeurs par défaut étant fournies.

Il n'y a pas de règle générale, tout dépend du contexte, mais le plus souvent pour des animations courtes et simples seuls les deux premiers paramètres sont fixés. On essayera malgré tout de choisir une fonction d'easing adaptée car cela change réellement la perception de la fluidité de l'UI. Une belle animation est souple et cette souplesse ne peut s'obtenir que par un easing bien choisi.

Les animations, comme la voix ou les sons, sont particulièrement difficiles à démontrer dans un livre on le comprend facilement. Faire dix captures d'écran de la rotation d'un label pour tenter de capter quelques-unes de ses positions intermédiaires ne me semble véritablement ni utile ni pédagogique. Je fais confiance à votre imagination qui est certainement capable de comprendre et de visualiser ce qu'est une rotation ou un changement de taille !

Essayons plutôt de rendre le `ClickLabel` plus agréable visuellement par l'ajout d'une animation.

Ce contrôle a été créé pour démontrer l'utilisation des propriétés de dépendances au chapitre traitant du Binding.

Pour lui ajouter une animation au moment du clic (ou tap) il faut avant tout détecter cet évènement or le `ClickLabel` a été pensé de façon à gérer une `ICommand` qui est directement transmise au `TapGestureRecognizer` qui décore le `Label`. C'est la classe de reconnaissance du tap qui est en charge d'appeler la commande. Et cela ne nous arrange pas ici.

En réalité il va falloir repenser une bonne partie du code pour gérer le tap et invoquer la `ICommand` afin de faire précéder cette action d'une animation.

De fait une nouvelle classe va être créée, le `AnimatedTapLabel`. Le terme *Tap* est plus conforme à l'esprit mobile que le *clic* et puisqu'il est animé c'est une bonne chose que de le dire. On se félicite toujours plus tard d'avoir donné des noms explicites aux classes en général et aux contrôles en particulier !

Voici le code du nouveau contrôle :

```
using System;
using System.Windows.Input;
using Xamarin.Forms;
```

```
namespace FilesAndData.Controls
{
    public class AnimatedTapLabel : Label
    {

        public static readonly BindableProperty CommandProperty =
            BindableProperty.Create(
                "Command",                  // nom de la propriété
                typeof(ICommand),           // type retourné
                typeof(AnimatedTapLabel),   // type déclarant
                null,                       // valeur par défaut
                BindingMode.OneWay,         // binding mode
                null,                     // validation de la valeur
                (bindable, value, newValue) =>
                {
                    if (bindable == null) return;
                    var l = (AnimatedTapLabel)bindable;
                    l.updateIsEnabled();
                    if (value != null) ((ICommand)value)
                         .CanExecuteChanged -= l.canExecuteChanged;
                    if (newValue != null) ((ICommand)newValue)
                         .CanExecuteChanged += l.canExecuteChanged;
                }
                );

        private void canExecuteChanged(object sender, EventArgs e)
        {
            updateIsEnabled();
        }


        private void updateIsEnabled()
        {
            IsEnabled = Command != null &&
                        Command.CanExecute(CommandParameter);
        }


        public static readonly BindableProperty
            CommandParameterProperty =
            BindableProperty.Create(
                "CommandParameter",         // nom de la propriété
                typeof(object),             // type retourné
                typeof(AnimatedTapLabel),   // type déclarant
                null,                     // validation de la valeur
                BindingMode.OneWay,
                null,
                (bindable, value, newValue) =>
                {
                    if (bindable == null) return;
                    var l = (AnimatedTapLabel)bindable;
                    l.updateIsEnabled();
                }
                );
```

```
public static readonly BindableProperty
  NumberOfTapsRequiredProperty =
    BindableProperty.Create(
        "NumberOfTapsRequired",
        typeof(int),
        typeof(AnimatedTapLabel),
        1,
        BindingMode.OneWay,
        (bindable, value) =>     // validation de la valeur
        {
            var tc = (int)value;
            return tc > 0 && tc < 3;
        },
        (bindable, value, newValue) =>  // property changed
        {
            if (bindable == null) return;
            var l = (AnimatedTapLabel)bindable;
            l.tap.NumberOfTapsRequired = (int)newValue;
        }
        );

public ICommand Command
{
    get { return (ICommand)GetValue(CommandProperty); }
    set { SetValue(CommandProperty, value); }
}

public object CommandParameter
{
    get { return GetValue(CommandParameterProperty); }
    set { SetValue(CommandParameterProperty, value); }
}

public int NumberOfTapsRequired
{
 get { return (int)GetValue(NumberOfTapsRequiredProperty); }
 set { SetValue(NumberOfTapsRequiredProperty, value); }
}

private readonly TapGestureRecognizer tap =
                                new TapGestureRecognizer();

public AnimatedTapLabel()
{
    tap.Tapped += tapTapped;
    GestureRecognizers.Clear();
    GestureRecognizers.Add(tap);
}

private async void tapTapped(object sender, EventArgs e)
{
    await this.RotateTo(360d, 150, Easing.CubicOut)
        .ContinueWith(
        t => Device.BeginInvokeOnMainThread(
                                        () => Rotation = 0d));
    if (Command == null) return;
    if (!Command.CanExecute(CommandParameter)) return;
```

```
            Command.Execute(CommandParameter);
        }
    }
}
```

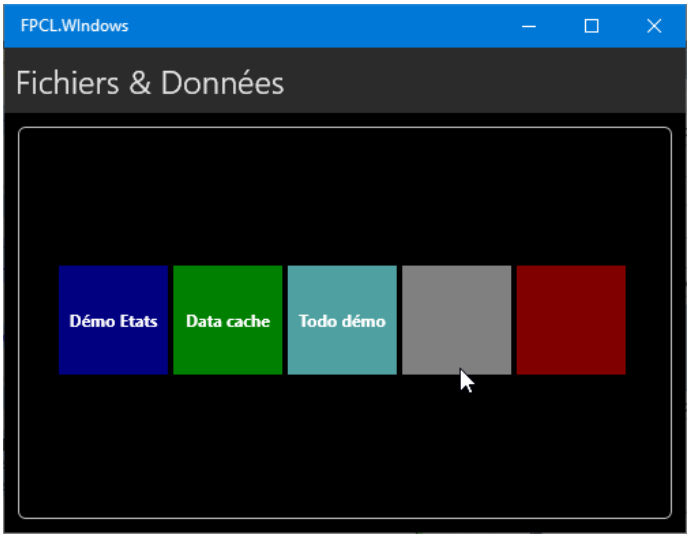
Comme on peut le constater si les propriétés sont identiques à celles du **ClickLabel** (en dehors du nombre de tap qui est ajouté) la logique a radicalement changé.

Le code s'est alourdi puisqu'au lieu de transférer la commande au **TapGestureRecognizer** il faut désormais intercepter l'événement **Tapped** de ce dernier et jouer l'animation avant d'invoquer l'exécution de la commande.

Je vous laisse étudier le code pour rester dans le sujet de cette section, les animations. Le code utile traitant celle du **AnimatedTapLabel** se situe tout à la fin dans la méthode **tapTapped**.

On y voit l'animation de rotation appelée en mode attente par **await**. Elle suivie d'une continuation de tâche qui remet l'angle de rotation à zéro et ce en utilisant l'aide de **BeginInvokeOnMainThread** car ici on manipule directement une propriété de l'objet. Aucun thread ne peut changer les propriétés d'un objet d'interface en dehors du thread principal. Sans cette invocation particulière le contrôle provoquera au mieux une exception au pire un plantage difficile à comprendre selon les plateformes.

L'attente de l'animation s'explique par le fait que le contrôle est utilisé comme élément de menu dans le menu principal de l'App de démonstration. Or si l'animation est jouée en parallèle de la navigation (donc sans attente) on ne la verra que peu ou pas. Cette dernière est très courte (150 millisecondes) et ce petit délai avant de naviguer n'est pas gênant d'autant qu'il apporte un retour visuel à l'utilisateur.



Le menu principal de l'application de démonstration est constitué de 5 **AnimatedTapLabel**, seuls les trois premiers sont utilisés, les deux restants servant à tester le nouveau contrôle lorsqu'il n'y a pas de commande donc pas de navigation. Le label gris sans texte a été paramétré pour ne fonctionner que sur un double tap. Il faut donc double-cliquer sous UWP desktop ou double-taper

sur mobile pour le voir s'animer. Ce que vous ne manquerez pas de faire grâce au code complet du livre qui est là pour vous faciliter l'expérimentation !

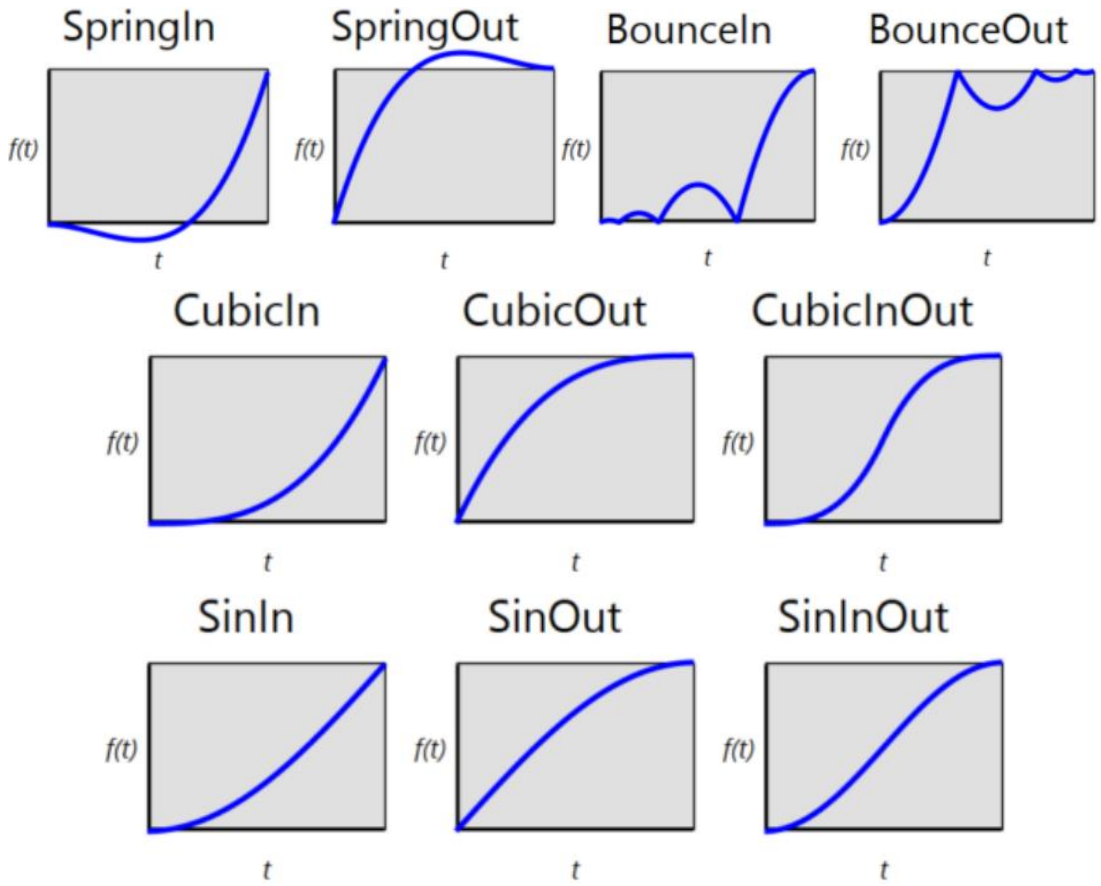
L'animation qui a été choisie est donc une rotation sur 360 degrés. Un fading n'aurait pas été assez visible et une translation n'aurait pas eu d'effet heureux dans le conteneur utilisé. Mais on pourrait très bien conjuguer des petites translations rapides pour simuler un tremblement par exemple ou bien s'appuyer sur un easing de type élastique (*Bounce* ou *Spring*) pour faire vibrer le contrôle.

On notera que la rotation s'appuie sur `AnchorX` et `AnchorY` définis par défaut au centre du contrôle et non manipulés dans l'exemple puisque cela convient parfaitement. Mais en changeant ces valeurs on modifie le centre de rotation pour obtenir un effet très différent.

Il ne faut pas oublier que les animations ne détachent pas l'objet animé de son contexte, il reste régi notamment par ses options de positionnement et par l'algorithme de placement de son conteneur. En fonction de tous ces régalages l'animation peut apparaitre de façon différente.

C'est un easing de type `CubicOut` qui est utilisé par le contrôle, ce qui ici et vu la rapidité de l'animation ne joue pas beaucoup il faut l'avouer.

Les différents types de easing se comprennent mieux lorsqu'on regarde leurs courbes de réponse. Les petits graphiques (figures ci-dessous) montrent l'évolution de la fonction de easing dans le temps. C'est encore la meilleure façon de comprendre l'effet de chacun.



En général les fonctions de easing fournies suffisent largement, mais il est possible de fournir ses propres formules mathématiques à une animation. Il s'agit alors d'un calcul retournant une valeur en fonction du temps par exemple :

```
button.ScaleTo(scale, 1000, new Easing(t => (int)(5 * t) / 5.0));
```

Vous pouvez visualiser et tester d'autres types de easing sur l'excellent site *http://robertpenner.com/easing/* qui ne brille pas par la sophistication de sa page d'accueil mais par son contenu ! Les exemples sont donnés pour de nombreux langages. Les animations sont bien faites et permettent de voir l'effet de chaque courbe présentée.

Sur GitHub, *https://github.com/danro/jquery-easing/blob/master/jquery.easing.js*, on trouve une longue liste de définitions de easings avec leurs calculs facilement transposables en C# qui est cousin de Java et de JavaScript pour ce qui est de la syntaxe.

On n'oubliera pas non plus que si les animations peuvent être attendues par un **`await`** elles peuvent se succéder automatiquement par un **`ContinueWith`** comme le fait l'exemple pour créer des effets complexes. Il est même possible de ne pas les attendre pour jouer simultanément un **`FadeTo`** et un **`RotateTo`** par exemple. Dans ce cas on peut attendre l'ensemble des animations jouées en parallèle avec les méthodes de **`Task`** comme **`WhenAll`**, **`WhenAny`** etc.

Il y a vraiment tout un monde à explorer pour ceux qui aiment jongler avec les fonctions mathématiques ! Pour les autres on trouve suffisamment de choses sur Internet pour varier ses animations.

L'exemple précédent montrait comment animer un bouton pour donner un retour visuel à l'utilisateur, mais on peut aussi se servir des animations de base pour créer des effets de transition.

L'exemple de la *Todo List* du chapitre précédent possède une transition d'entrée avec un fade-in du plus bel effet. Le code-behind de la **`ContentPage`** contient ceci :

```
protected override void OnAppearing()
{
    base.OnAppearing();
    List.Opacity = 0d;
    List.FadeTo(1, 2000);
}
```

C'est la liste qui est l'élément principal de la page qui est animée et non pas la page elle-même cela ne fonctionne pas. Il faut animer le conteneur principal de la page, une grille, un **`StackLayout`** ou ici directement le contrôle qui occupe tout l'écran. On peut le fait uniquement sur une partie des contrôles de la page, voire avec des timings différents pour créer un effet de « fleuraison ».

Là aussi choisir un easing adapté peut rendre l'effet encore plus convaincant (mon exemple ci-dessus n'en utilise aucun juste par esprit de contradiction et surtout parce que l'animation linéaire « fonctionne » bien ici).

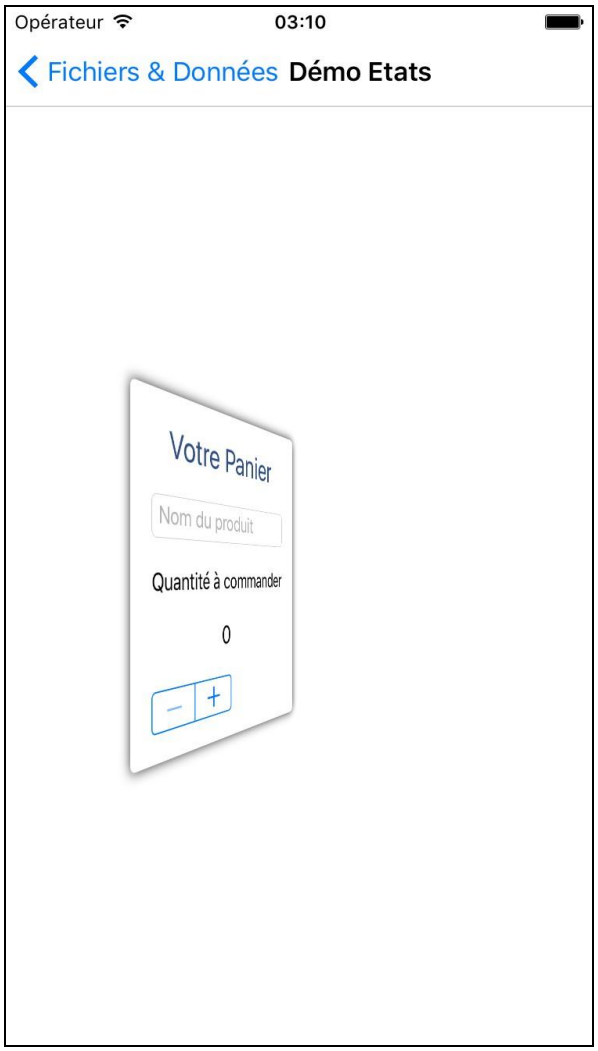
Un autre type d'animation servant à attirer l'œil de l'utilisateur sur les éléments importants de l'UI se trouve dans la page principale de l'application, celle qui affiche le menu (les **AnimatedTapLabel** carrés). Les labels s'affichent progressivement dans un léger clignotement (dû au easing) qui forme une sorte de vague qui traverse l'ensemble :

```
protected override void OnAppearing()
{
    base.OnAppearing();
    var r = new Random();
    foreach (var child in MainContainer.Children)
    {
        if (!(child is AnimatedTapLabel)) continue;
        child.Opacity = 0d;
        child.FadeTo(1d, 1000 + (uint) r.Next(1000),
                         Easing.BounceOut);
    }
}
```

Les animations de **ViewExtensions** bien que peu nombreuses et assez basiques peuvent ainsi être utilisées pour créer des effets sophistiqués auquel on ne pense pas tout de suite. La rotation, en jouant sur les ancres notamment, peut servir à créer des sortes d'effet 3D comme une porte qui s'ouvre et se ferme. Cet effet est utilisé sur le premier exemple de l'application dont le code-behind contient :

```
protected override async void OnAppearing()
{
    base.OnAppearing();
    MainFrame.AnchorX = 0;
    MainFrame.RotationY = 90;
    await MainFrame.RotateYTo(0, 1500, Easing.CubicOut);
    MainFrame.AnchorX = 0.5d;
}
```

La capture sous Android montre l'animation de type « porte » utilisée comme transition d'entrée d'une page. On remarque que l'effet est rendu avec une illusion de perspective tout à fait intéressante.



La classe Animation

Les **ViewExtensions** ne sont pas le seul moyen de créer des animations avec les XForms. Il existe d'autres biais pour animer toutes sortes de propriétés. Et c'est le rôle de la classe **Animation** dont la signature du constructeur est la suivante :

```
public Animation (Action<double> callback,
                  double start = 0.0f,
                  double end = 1.0f,
                  Easing easing = null,
                  Action finished = null)
```

Il s'agit ici du constructeur avec paramètres qui permet une initialisation rapide mais il existe aussi un constructeur sans paramètre lorsqu'on doit décider des paramètres ultérieurement, notamment pour la définition de multiples animations simultanées.

Bien entendu on ne peut animer que des valeurs continues de type numérique, animer une propriété **string** n'aurait pas beaucoup de sens par exemple (sauf quelques exceptions dont je reparlerai plus bas). **Animation** prend ainsi une valeur de début (**start**) et de fin (**end**) et crée l'animation de l'une à l'autre. La valeur calculée est passée à callback qui utilise la valeur produite.

Il n'y a donc pas d'animation directe, **Animation** n'est finalement qu'un générateur de valeurs. Le plus souvent bien entendu cela sera pour animer la propriété d'un contrôle. Lorsque l'animation est terminée **finished** est appelé.

*Dire que la documentation actuelle est laconique serait un doux euphémisme… Il faut jouer longtemps avec cette classe pour comprendre le principe adopté par ses développeurs. Et il faut comprendre la classe **Animation** comme un moyen de créer des animations simultanées, en dehors de ce contexte les autres procédés sont plus simples. On peut même associer cette classe à un Storyboard comme on en trouve sous UWP.*

Animation offre la possibilités de gérer une liste d'animation enfants qui peuvent démarrer en même temps et rester synchronisée. C'est d'ailleurs son principal intérêt. On trouve ainsi les méthodes **Add**, **Insert** et **WithConcurrent** qui toutes permettent d'ajouter d'autres instances de **Animation** à la liste des enfants d'une animation.

Quelque chose manque non ? Une animation sans durée cela est étrange… En fait c'est la méthode **Commit** qui démarre l'animation qui possède ce paramètre ainsi que de nombreux autres parfois redondant (même ceux du **Commit** ont la priorité) :

```
Commit(IAnimatable owner,
       string name,
       uint rate = 16,
       uint length = 250,
       Easing easing = null,
       Action<double, bool> finished = null,
       Func<bool> repeat = null);
```

Le premier paramètre est de type **IAnimatable** qui est implémenté par tout objet pouvant être animé et la seule classe qui l'implémente est **VisualElement**, à la base de tous les contrôles visuels des XForms. L'interface définit deux méthodes **BatchBegin** et **BatchCommit**. Mais c'est l'animation qui gère les appels, on ne fait que passer l'objet à animer et ne pouvant faire autrement que descendre de **VisualElement** il supportera donc l'interface.

On retrouve dans Commit une durée (**length**), une fréquence d'échantillonnage (**rate**) qui se précise en millisecondes entre deux pulsations, le easing et deux autres callbacks. L'un étant appelé à la fin de l'animation l'autre devant retourner une valeur booléenne indiquant si l'animation doit être répétée ou non.

Le nom de l'animation permet de la caractériser ce qui est utile en conjonction de la méthode **AnimationExtensions** qui permet en autres choses de savoir si une animation donnée est en cours et éventuellement de l'annuler.

Certains paramètres se comportent de façon légèrement différente selon les plateformes on s'en serait douté. Il est donc essentiel une fois encore d'expérimenter.

Pour bien fixer les idées, **Animation** est principalement utilisée pour gérer un ensemble d'animations simultanées portant sur le même objet visuel. On peut bien entendu ne lancer

qu'une seule animation par ce biais mais cela donne une construction étrange et bien moins pratique que les autres façons de faire.

Voici un exemple de code qui permet de mieux visualiser ce principe de **storyboard** qui se cache derrière **Animation** :

```
var width = 800;
var storyboard = new Animation ();
var rotation = new Animation (callback: d => label.Rotation = d,
                              start:    label.Rotation,
                              end:      label.Rotation + 360,
                              easing:   Easing.SinIn);


var trans1 = new Animation (callback:d =>label.TranslationX = d,
                            start:    0,
                            end:      width,
                            easing:   Easing.BounceOut);

var trans2 = new Animation (callback:d =>label.TranslationX = d,
                            start:    -width,
                            end:      0,
                            easing:   Easing.SpringIn);

storyboard.Add (0, 1, rotation);
storyboard.Add (0, 0.5, trans1);
storyboard.Add (0.5, 1, trans2);

storyboard.Commit(label, "anim", length: 1000);
```

On comprend mieux ici ce concept de *Storyboard* et le pourquoi du nom de la méthode **Commit**. On crée une instance de **Animation**, on y ajoute autant d'animations que nécessaire puis on valide par **Commit** comme on ferait un *commit* sur une transaction de base de données. Ce qui déclenche l'animation.

Les AnimationExtensions

Comme les **ViewExtensions** cette classe définit des méthodes d'extension utilisables directement sur des vues. Mais pour être consistante avec la méthode **Commit** de **Animation** l'argument principal n'est pas une vue mais un **IAnimatable**, ce qui revient au même comme je l'ai fait remarquer plus haut.

Les méthodes définies par ces extensions sont des surcharges de la méthode **Animate** dont la version la plus étendue à la signature suivante :

```
public static void Animate<T>(this IAnimatable self,
                              string name,
                              Func<double, T> transform,
                              Action<T> callback,
                              uint rate = 16,
                              uint length = 250,
                              Easing easing = null,
```

```
                                    Action<T, bool> finished = null,
                                    Func<bool> repeat = null);
```

Si le système d'animation des XForms se complique un peu en réalité vous n'aurez besoin que de cette seule méthode dans la majorité des cas (ceux des animation simples). Raison pour laquelle je suis passé assez rapidement sur la classe Animation et ses animations enfants mais aussi parce que j'y reviendrai par des exemples plus parlants.

`Animate` reprend les principaux arguments déjà rencontrés comme le nom, l'action à réaliser, la durée, la fréquence, le easing etc.

La fonction `transform` est en revanche nouvelle dans cet ensemble et elle est précieuse. Elle permet d'animer des propriétés qui ne sont pas de type `double`. La fonction passe un `double` qui évolue entre 0 et 1 (ou autres) et retourne une valeur du type à animer. Charge à la fonction de créer une correspondance entre le `double` et la propriété. Par exemple pour animer une couleur une telle fonction ferait le lien entre une valeur comprise entre 0 et 1 et une couleur. La méthode de conversion dépend entièrement de ce que le développeur veut faire. On peut ainsi animer une énumération, une liste d'items, une date, n'importe quoi du moment que l'on est capable d'écrire une fonction de transformation entre un `double` et le type en question.

L'autre méthode définit par les extensions est `AnimationIsRunning` qui permet de savoir si une animation (précisée par son nom) tourne sur un `VisualElement` donné.

Application exemple

Le premier exemple que je vous propose de regarder concerne l'utilisation de la classe `Animation` car son fonctionnement peut dérouter. La documentation quasi inexistante ne donne que peu d'exemples et même le livre de Charles Petzold publié gratuitement par Microsoft Press et qui est supposé servir de documentation plus étaillée s'embrouille un peu et guide vers de mauvaises directions.

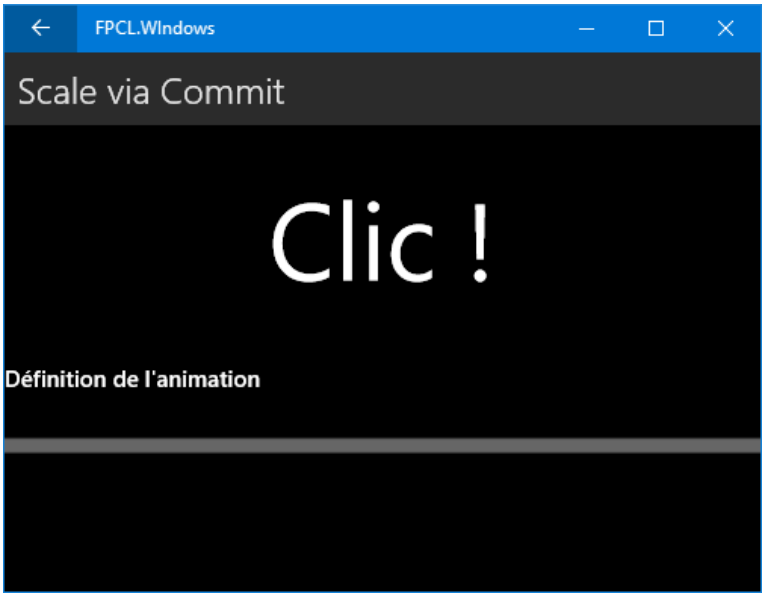
Dans un premier temps je vais vous montrer une animation très simple, un changement d'échelle appliqué à un bouton. L'animation débutera sur le clic de ce dernier.

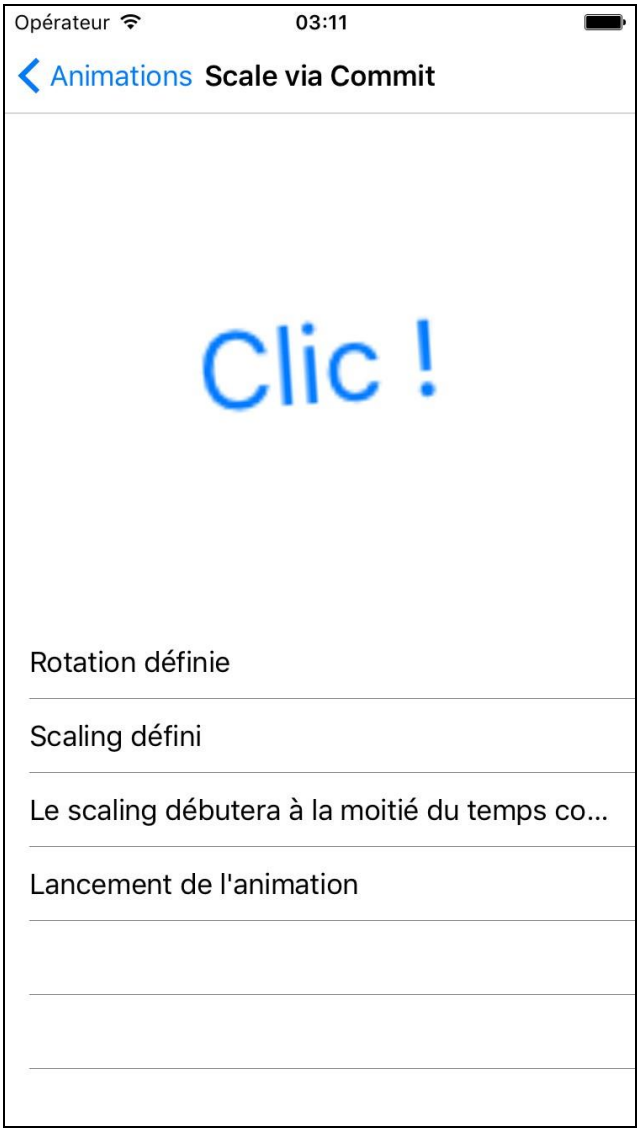
Pour suivre certains événements une `ListView` sera affichée en dessous.

Visuellement le résultat est montré par les deux captures, l'une sous Android l'autre sous UWP desktop.

Les deux captures ont été figées pendant l'animation. La `ListView` affiche le message « définition de l'animation » que vous retrouverez plus loin dans le code C#. Le bouton a déjà atteint une bonne taille mais comme on le constate il ne perturbe pas la mise en page. Celle-ci est réalisée avec une Grid définissant deux colonnes de même taille en automatique (avec l'étoile).

Dans la capture UWP le trait gris qui déborde sur la ListView est le contour du bouton dont on ne voit plus que le texte et cette ligne. A la différence de Android le bouton UWP est « vide », il ne possède qu'un contour et encore celui-ci ne s'affiche-t-il que lorsque la souris le survole. L'effet est moins évident que sous Android.





La dernière capture sous iOS montre le bouton en train de pivoter et de grossir. Le bouton n'a pas de bordure et seul le texte s'anime. Les trois plateformes nous offrent presque toutes les

combinaisons : bouton avec fond, bouton avec bordure, bouton sans rien… Mais malgré ces différences notre code unique XForms fonctionne dans tous les cas !

La page XAML a pour code :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FilesAndData.Views.A1View"
             Title="Scale via Commit">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="*"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0" Text="Clic !" Clicked="buttonClicked"
            HeightRequest="60" WidthRequest="150"
            VerticalOptions="Center" HorizontalOptions="Center"/>
    <ListView x:Name="ListView" Grid.Row="1" />
  </Grid>
</ContentPage>
```

Cette démonstration ne faisant intervenir que des éléments de l'UI elle ne possède pas de ViewModel, tout est effectué dans le code-behind :

```
using System;
using System.Collections.ObjectModel;
using Xamarin.Forms;

namespace FilesAndData.Views
{
    public partial class A1View : ContentPage
    {
        public A1View()
        {
            InitializeComponent();
            Trace.Add("Démo animation");
            ListView.ItemsSource = Trace;
        }

        private ObservableCollection<string> trace =
                                new ObservableCollection<string>();

        public ObservableCollection<string> Trace => trace;

        private void buttonClicked(object sender, EventArgs e)
        {
            Trace.Clear();
            animate((VisualElement)sender);
        }

        private void animate(VisualElement obj)
        {
            Trace.Add("Définition de l'animation");
            var anim = new Animation(
```

```
                (d => obj.Scale = d),
                1,
                6,
                Easing.SinIn,
                () =>
                {
                    Trace.Add("Terminé");
                }
                );

            anim.Commit(
                obj,
                "Anim1",
                16,
                1500,
                Easing.SinIn,
                (d, b) =>
                {
                    Trace.Add($"Terminé. Finale: {d} Annulé: {b}");
                    obj.Scale = 1;
                },
                () =>
                {
                    Trace.Add("Répétition (non)");
                    return false;
                }
                );
        }
    }
}
```

Passons sur la gestion de la **Trace** qui ne fait qu'ajouter des lignes à l'**ObservableCollection** servant de source à la **ListView**.

L'animation est définie en créant une nouvelle instance de **Animation** puis en faisant un **Commit**. Certains paramètres semblent définis deux fois. Et au final on comprend mal la véritable vocation de la classe **Animation**.

Lorsqu'on a joué un peu plus avec tout cela et étudié le code source décompilé des XForms on s'aperçoit que l'intention des développeurs n'était pas de faire ce que je viens de vous montrer mais plutôt ceci qui redéfini autrement la méthode **animate** de l'exemple ci-dessus :

```
        private void animate(VisualElement obj)
        {
            var anim = new Animation();

            var rotation = new Animation(d => obj.Rotation = d,
                                         0, 360, Easing.SinInOut,
                () =>
                {
                    Trace.Add("Rotation terminée.");
                    obj.Rotation = 0;
                });
```

```
        Trace.Add("Rotation définie");

        var scaling = new Animation(d => obj.Scale = d,
            1,6,Easing.CubicIn, () =>
                                {
                                   Trace.Add("Scaling terminé");
                                   obj.Scale = 1;
                                });

        Trace.Add("Scaling défini");

        anim.Add(0,1,rotation);
        anim.Add(0.5,1,scaling);

        Trace.Add(
         "Le scaling débutera à la moitié du temps complet.");

        Trace.Add("Lancement de l'animation");
        anim.Commit(obj,"Anim1",16,1500);

    }
```

Je ne peux pas dire pour vous qui lirez ce livre ou tout s'enchaîne, mais pour moi tout a changé quand j'ai compris ce que montre ce code (les deux versions coexistent dans le code source fourni avec livre).

Une animation est créée sans aucun paramètre. Puis deux autres animations sont définies plus finement, l'une faisant une rotation, l'autre un changement d'échelle. Les deux animations sont ajoutées à la première qui joue le rôle de *storyboard*. Et enfin le `Commit` valide l'ensemble et lance l'animation.

A aucun moment des paramètres semblent être définis en double. Tout s'éclaire …

J'ai ainsi découvert que la méthode `Add` de `Animation` qui permet d'ajouter les méthodes filles possèdent deux paramètres de type `double` bien étranges à première vue. Si l'on considère que la durée de l'animation (fixée dans le `Commit`) vaut « 1 », alors ces deux paramètres permettent de définir quand commence et quand s'arrête une animation fille. Dans l'exemple de code `scaling` débute à `0.5` et se termine à `1` alors que `rotation` commence à `0` pour se terminer aussi à `1`. Cela signifie que `scaling` commercera à la moitié du temps total de l'animation. Très subtil !

L'effet visuellement est assez intéressant puisque la rotation va débuter toute seule puis à moitié chemin des 1500 millisecondes elle va se compliquer en ajoutant le changement d'échelle, les deux mouvements se conjuguant. La rotation sera plus lente puisque s'étalant sur la durée totale, le changement d'échelle sera plus rapide puisque s'exécutant sur la seconde moitié du temps total.

Si `Animation` semblait étrange au départ, si la documentation n'en dit une fois que l'on a compris le véritable esprit de cette classe elle devient merveilleusement limpide et particulièrement utile pour créer des animations complexes qui vous attireront la jalousie de vos pairs et l'admiration de vos clients. Enfin pas forcément, mais au moins vous aurez essayé !

Découvrez la suite du chapitre ainsi que tous les autres dans le livre ! www.e-naxos.com\Livre.aspx

Nota : la mise en page de cet extrait ne correspond pas à celle utilisée par le livre