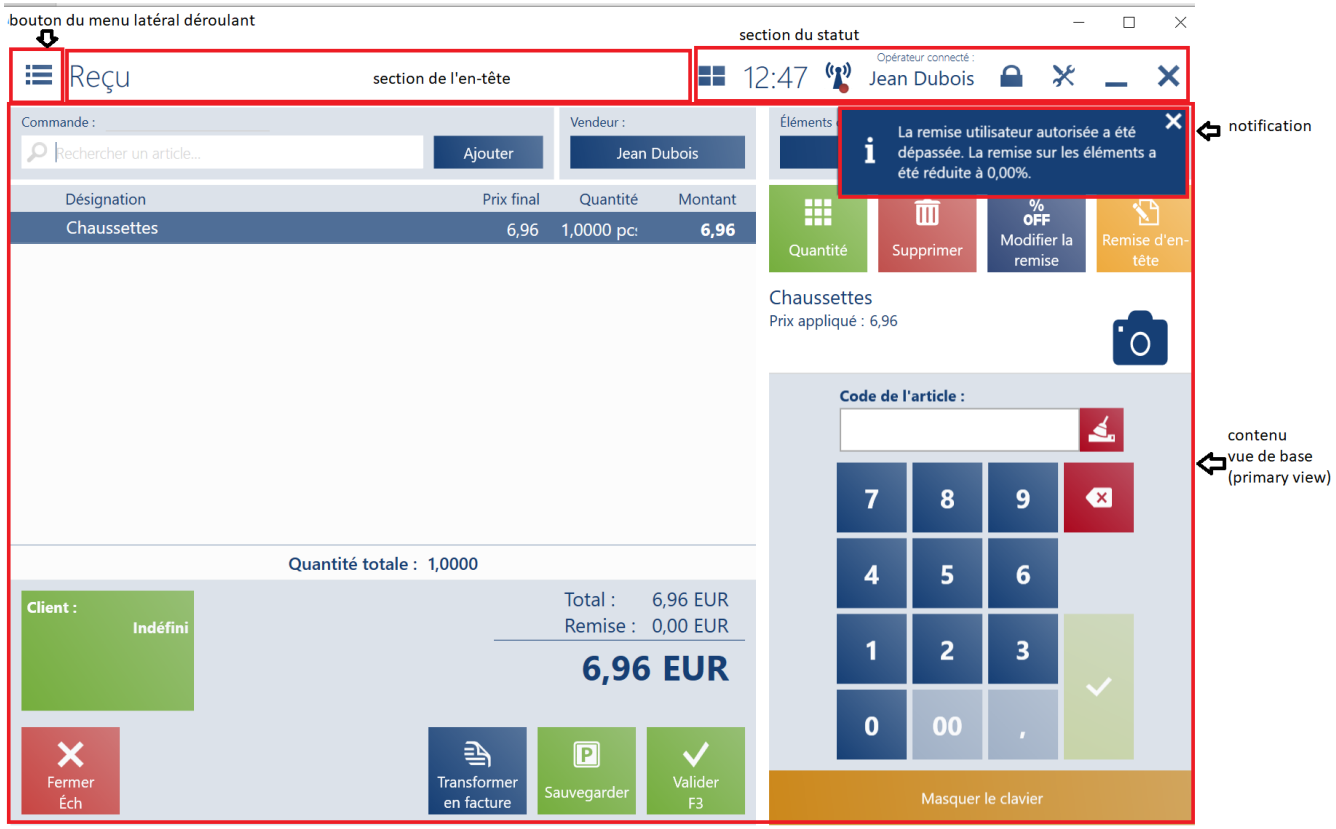
Introduction

L’application POS a été créée à l’aide de la technologie Windows Presentation Foundation. Le squelette du logiciel peut être divisée en sections. La section principale est le **contenu** – c’est la zone centrale de l’application où le contenu est affiché sous forme des vues. Le deuxième élément est la **section de l’en-tête**, inextricablement liée avec le contenu (les vues) et qui est une sorte de titre pour le contenu. À part de ces deux éléments, on distingue également la **section du statut** avec les boutons d’accès rapide (comme fermer, minimaliser) et où est affichée l’information sur l’utilisateur actuellement connecté.

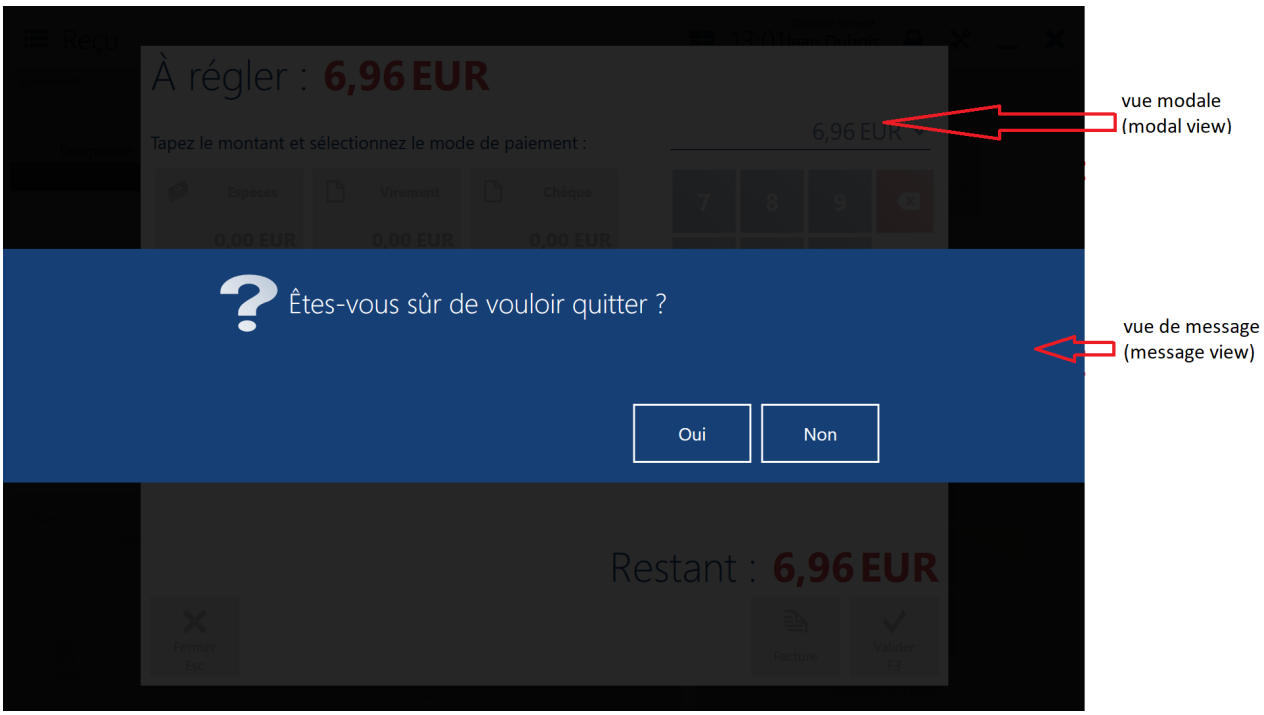


Dans la section du contenu sont affichées les vues. On distingue trois types des vues :

- vue de base (primary view)
- vue modale (modal view) – cette vue est affichée sur la vue de base. Elle cache la vue de base et empêche

l’utilisateur d’interagir avec la vue de base. En revanche, la logique n’est pas bloquée. Fermer la vue modale réouvre la vue de base.

- vue de message (message view) – cette vue est affichée sur la vue de base ou sur la vue modale. Elle cache la vue de base et empêche l’utilisateur d’interagir avec les vues en-dessous. En revanche, la logique n’est pas bloquée. Cette vue se caractérise par le fait qu’elle s’étend sur toute la largeur de l’écran et par un arrière-plan de la couleur de la police de la vue de base. Utilisation ordinaire : un message d’erreur, une question lors de la fermeture de l’application etc.



Chaque vue est créée de la même façon (voir Créer des vues). L'affichage de la vue dépend de sa façon d'ouverture (voir Naviguer entre les vues).

Créer des vues

Nouveau module

Afin de créer une nouvelle vue, il faut d'abord créer un nouveau module POS. Dans Visual Studio nous sélectionnons un nouveau projet **WPF Custom Control Library**. Ce type de projet génère automatiquement une structure indispensable et permet d'ajouter rapidement de nouveaux éléments. Au début, notre projet vide est composé du dossier **Themes** avec le fichier **Generic.xaml** et du fichier CustomControl1.cs que nous pouvons supprimer.

Dans un temps suivant, il faut créer la classe **Module.cs** qui

sera responsable pour l'enregistrement de notre module dans l'application POS. Cette classe doit hériter de la classe de base **ModuleBase** (Comarch.POS.Presentation.Core) et implémenter correctement la méthode **Initialize.** À l'intérieur de cette méthode nous allons enregistrer nos services, vues et viewmodels personnalisés, enregistrer les boutons dans le menu principal POS, enregistrer nos propres contrôles, gérer la visibilité des propriétés des contrôles dans la gestion de l'interface, élargir les conteneurs et datagrids existants dans le système.

Pour que notre vue peut être ultérieurement gérée par l'utilisateur en cours du fonctionnement de l'application POS, il faut ajouter au dossier **Themes** un nouvel élément du type Resource Dictionary (il faut cliquer avec le bouton droit de la souris et ensuite dans le menu contextuel sélectionner dans l'ordre suivant : Add, Resource Dictionary…). Nous l'appellons **ModernUI.xaml** et l'enregistrons. Dans ce fichier seront définies les propriétés par défaut des éléments de l'interface gérables (vues, contrôles) – voir Gérer la vue et ses éléments. À la fin, il faut encore enregistrer une ressource. Ceci est fait dans le constructeur de classe Module en écrivant la ligne suivante :

```
LayoutService.RegisterResources(typeof(Module));
```

(LayoutService est une classe statique présente dans l'espace Comarch.POS.Presentation.Core.Services).

Créer une nouvelle vue

La création d'une nouvelle vue doit être commencée avec la création du dossier **Views** où seront gardées toutes les vues de projet et le dossier **ViewModels** pour les viewmodels. Ensuite, nous ajoutons dans le dossier Views un nouvel élément du type **User Control (WPF)**, appelé par exemple ***OrdersView.xaml.***

Ensuite, il faut changer dans l'espace Comarch.POS.Presentation.Core le type UserControl en **View**.

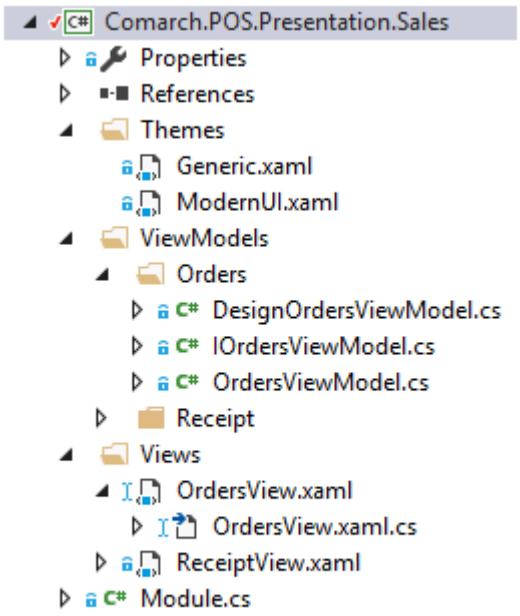
```
<core:View
x:Class="Comarch.POS.Presentation.Sales.Views.OrdersView"
          xmlns:core="clr-
namespace:Comarch.POS.Presentation.Core;assembly=Comarch.POS.P
resentation.Core"
```

Dans le code-behind nous supprimons l'hérédité d'UserControl et implémentons l'interface View requise.

La propriété **Header** est un string affiché dans la section de l'en-tête de l'application – elle est requise uniquement pour les vues qui seront ouvertes en tant que vues de base, car seules ces vues ont la présentation de l'en-tête. Dans tous les autres cas (c'est-à-dire la vue modale et la vue de message), il suffit de la paramétrer comme **String.Empty.** Il est également possible de créer son propre en-tête de section avec un contenu libre – voir **Ajouter un en-tête personnalisé**.

La propriété **HeaderLayoutID** doit en revanche contenir un nom unique de l'identifiant layout id ce qui est requis pour le fonctionnement correct de la gestion des vues par l'utilisateur POS. Si l'on souhaite que la vue ne puisse pas être modifiée, il suffit de paramétrer cette propriété comme **String.Empty.** Aux fins de cette documentation, nous paramétrons ici la valeur *« OrdersViewId ».*

Le constructeur de base requiert le transfert d'un objet du type **IViewModel**, donc nous allons maintenant créer un viewmodel. Dans le dossier **ViewModels** il faut d'abord créer un dossier appelé Orders. Ensuite, nous y ajoutons la nouvelle classe OrdersViewModel et l'interface IOrdersViewModel. L'interface doit être publique et hériter d'**IViewModel.** La classe OrdersViewModel doit être également publique et elle doit implémenter la classe de base **ViewModelBase**, ainsi que l'interface IOrdersViewModel.



En héritant de **ViewModelBase** il faut implémenter la méthode **IsTarget.** Elle est appelée lors de la navigation et permet à l'instance de viewmodel de vérifier si elle doit être activée ou non (voir Naviguer entre les vues) – à présent, nous la définissons pour qu'elle retourne la valeur true.

En outre, si l'on souhaite que la vue puisse être gérée par l'utilisateur POS, il faut ajouter une classe supplémentaire, un viewmodel appelé DesignOrdersViewModel qui héritera de **DesignViewModelBase** et implémentera l'interface IOrdersViewModel. La structure entière est présentée sur le dession à côté.

Maintenant, nous revenons au code-behind de la classe OrdersView (fichier OrdersView.xaml.cs) et nous paramétrons le constructeur pour qu'il adopte le paramètre du type IOrdersViewModel, appelé **viewModel (attention : ce nom est important et ce paramètre doit être toujours appelé comme ça !).** Ensuite, nous complétons le constructeur de base en lui transmettant notre variable viewModel.

La dernière étape consiste à enregistrer notre nouvelle vue et elle a été décrite dans le chapitre **Enregistrer des vues à naviguer et à gérer l'affichage.**

Le modèle fini de création de module avec une vue vide est disponible dans l'exemple Module simple avec une nouvelle vue vide.

Ajouter un en-tête personnalisé

Il est possible de définir pour chaque vue de base le texte dans l'en-tête qui sera affiché dans la partie supérieure de l'application POS. Par défaut, seul une chaîne de caractères définie dans la propriété **Header** de la vue peut y être emplacée. S'il est nécessaire d'avoir une structure plus complexe dans l'en-tête, une vue d'en-tête personnalisée (custom header) peut être créée.

Nous commençons à définir un en-tête personnalisé par ajouter un nouvel élément **User Control (WPF)** au dossier avec la vue. Pour améliorer la lisibilité, l'élément sera appelé comme notre vue, mais avec le préfixe Header (par exemple pour la vue OrdersView, le nom sera **OrdersViewHeader**).

À la fin, il faut revenir au code-behind de notre vue OrdersView et paramétrer dans le constructeur la propriété **CustomHeaderType** pour le type de notre en-tête personnalisé. Un DataContext indiquant le ViewModel de la vue sera automatiquement attribué à la vue d'en-tête. Dans ce cas, ça sera OrdersViewModel. Ainsi, il sera possible d'utiliser dans notre en-tête un binding direct à la propriété dans le viewmodel de la vue.

```
public partial class OrdersView
{
          public OrdersView(IOrdersViewModel viewModel) :
base(viewModel)
        {
            CustomHeaderType = typeof (OrdersViewHeader);
            InitializeComponent();
        }
…
…
```

Enregistrer des vues à naviguer et à gérer l'affichage

Après avoir créé la vue (en implémentant la logique de fonctionnement dans viewmodel et l'UI dans xaml respectivement), l'étape nécessaire pour que notre vue fonctionne correctement et de l'enregistrer. Afin de le faire, il faut ouvrir la classe **Module**, créée lors de la création du projet (voir **Nouveau module**). Nous ajoutons des lignes suivantes dans la méthode **Initialize** :

```
Register<IOrderViewModel, OrderViewModel>();
RegisterViews(new ViewStructure<OrdersView, DesignOrdersViewModel>("OrdersView", Resources.ResourceManager);
```

La première enregistre notre viewmodel dans le conteneur, grâce à ce qu'une instance de l'interface IOrdersViewModel est automatiquement implémentée dans le constructeur de vue.

Grâce à la deuxième méthode, un utilisateur POS peut enregistrer la vue en mode de gestion de son interface lors du fonctionnement de l'application. Ensuite, lorsque l'utilisateur ouvre la vue de gestion d'interface, il verra notre vue dans la liste des vues. Il pourra l'ouvrir et modifier l'affichage de tous ces éléments qui ont été définis comme gérables (pour en savoir plus voir Gestion de vue et de ses éléments). Cette méthode exige que deux paramètres soient transmis. Le premier est le nom de clé dans les ressources et le deuxième est une instance du gestionnaire de ressources. Ces paramètres permettent de présenter le nom de vue dans l'arbre des vues dans la gestion des vues de l'application POS.

Alternativement, si l'on ne veut pas enregistrer la vue en mode de gestion d'interface, au lieu de **RegisterViews** il faut utiliser la méthode **RegisterForNavigation<TView >().**

```
RegisterForNavigation<OrdersView>();
```

Ajouter une vue au menu principal

Chaque vue que l'on souhaite ouvrir en mode base peut être ajoutée au menu principal sous forme d'une mosaïque (TileButton). En ajoutant une vue au menu principal, elle sera automatiquement ajoutée au menu latéral déroulant qui peut être ouvert à partir de n'importe quelle vue de base.

Afin d'ajouter une vue précédemment créée au menu sous forme de mosaïque, il faut l'enregistrer dans la méthode **Initialize** de la classe **Module** à l'aide de la méthode **RegisterMenuTile<TView>** où TView est le nom de la classe de vue. Les paramètres d'appel sont le nom de clé dans les ressources et le gestionnaire de ressources respectivement. Facultativement, il est possible de définir le délégué canExecute. S'il est false, la mosaïque sera en gris et ne pourra pas être cliquée. Il est également possible d'ajouter un délégué aux actions qui doivent être exécutées avant l'ouverture de vue/et définir les autorisations requises pour ouvrir la vue avec la clé dans les ressources où sera enregistré le message qui apparaîtra dans la fenêtre modale dans le cas où les autorisations ne seraient pas suffisantes (pour plus d'informations sur les autorisations voir Vérification des autorisations). Aux fins de notre exemple, nous allons enregistrer la vue *OrdersView;* créée précédemment.

```
RegisterMenuTile<OrdersView>("OrdersView",
Resources.ResourceManager);
```

Après l'enregistrement de vue dans le menu principal, il sera possible de l'ouvrir en appuyant sur la mosaïque appropriée. Par défaut, la mosaïque aura l'aspect défini dans la configuration globale de l'interface POS. Afin de modifier ses propriétés visuelles, il faut ajouter des entrées appropriées dans le fichier **ModernUI.xaml** qui se trouve dans le dossier **Themes.** Par exemple, pour définir une nouvelle couleur par

défaut pour notre mosaïque ouvrant la vue OrdersView, il faut ajouter la ligne suivante :

```
<SolidColorBrush    x:Key="OrdersView.Default.Background" Color="Red" />
```

Le format de chaque clé dans ModernUI sera :

[LayoutId ou Nom du type de contrôle].Default.[Nom de propriété]

En revanche, la définition entière sera :

<[Nom du type de propriété] x:Key="[clé selon le format en-dessus]" [attributs]>[valeur]</[Nom du type de propriété>

Les attributs et la valeur ne sont pas obligatoires et ils dépendent strictement du type de propriété à laquelle nous avons à faire.

LayoutId pour la mosaïque de la vue OrdersView est défini dans le premier paramètre de la méthode **RegisterMenuTile.** Dans notre cas, LayoutId utilisé dans la clé définissant la couleur de la mosaïque est OrdersVies, car cette valeur a été définie avant, lors de l'appel de la méthode d'enregistrement de la vue sous forme de mosaïque dans le menu principal. Les propriétés, prises en charge par la gestion d'interface, dépendent du type de contrôle. Dans le cas des mosaïques c'est le contrôle **TileButton.** Une liste avec informations quel contrôle prend en charge quelles propriétés se trouve ici : Liste des propriétés prises en charge

Brancher un module dans l'application POS

Après la compilation du modèle en cours de création, la dernière étape est son démarrage dans l'environnement cible, c'est-à-dire dans l'application POS. Ceci est fait en copiant le module créé sous forme de répertoire (ou répertoires) dans

le dossier d'installation de l'application POS. Ensuite, il faut ouvrir le fichier **POS.exe.config**, y retrouver la section **<modules>** et ajouter le nouveau module à la fin de cette section selon le modèle suivant :

<module assemblyFile="[nom_module].dll" moduleType="[namespace_classe_module].Module, [namespace_classe_module]" moduleName="[nom_module]" />

S'il y a plus de répertoires, il faut enregistrer uniquement ceux dans lesquelles est implémentée la classe Module.

Naviguer entre les vues

L'application POS permet d'ouvrir un nombre illimité de vues qui sont en modes différents (vue de base, modale ou vue de message), mais à un moment donné seule une vue de chaque groupe peut être active. Les autres vues ouvertes sont définies comme inactives. À tout moment, lorsqu'une vue de base quelconque est affichée, nous pouvons appeler une fenêtre de changement des vues de base (menu de navigation) afin d'activer une autre vue qui a été ouverte auparavant et qui n'a pas été fermée (elle est devenue inactive). Le menu de navigation ne peut pas être appelé si une vue modale ou une vue de message est ouverte.

Menu de navigation

Le menu de navigation peut être appelé à l'aide du raccourci clavier CTRL+TAB ou à l'aide du bouton visible dans la section de statut à côté de l'horloge. Dans cette fenêtre, les vues de base ouvertes sont affichées sous forme des mosaïques multi couleurs avec un titre, une icône et/ou une description supplémentaire. Le titre de mosaïque peut être défini à l'aide de la propriété **Header** dans la classe View ou à l'aide de la propriété **SwitchHeader** dans le viewmodel (si l'on veut qu'il soit différent de l'en-tête de la vue). L'icône et la couleur sont définies à l'aide des styles dans le fichier ModernUI.xaml. Leur définition est effectuée de la même façon que la définition de couleur et icône d'une mosaïque enregistrée dans le menu principal. De ce fait, lorsqu'une vue a déjà été enregistrée dans le menu principal, la mosaïque dans le menu de navigation adoptera la même couleur et icône que la mosaïque dans le menu principal, à condition que la valeur de la clé de ressource (le premier argument) utilisée dans la méthode RegisterMenuTile soit identique à la valeur de la propriété HeaderLayoutId dans le code-behind de la vue. Dans notre cas, les noms sont différents, donc il sera nécessaire d'ajouter des lignes supplémentaires dans ModernUI.xaml.

Définition dans ModernUI.xaml de la couleur et de l'icône de mosaïque dans la fenêtre du menu de navigation pour la vue exemplaire **OrdersView** :

```
<SolidColorBrush    x:Key="OrdersViewId.Default.Background" Color="Red" />
<models:ImageKey    x:Key="OrdersViewId.Default.ImageKey" SvgValue="ListIcon" />
```

Il ne faut pas oublier que dans ce cas, OrderViewId est la valeur attribuée à **HeaderLayoutId** dans la classe de vue.

En outre, il est également possible d'ajouter une descirption supplémentaire à la mosaïque (comme dans la mosaïque Nouvelle vente sur la capture d'écran en dessus). Ce texte est géré par la propriété **SwitchHeader2** dans la classe de viewmodel.

Ouvrir des vues

Les vues peuvent être ouvertes en plusieurs modes (comme une vue de base, vue modale ou vue de message). L'utilisateur décide si une vue donnée doit être ouverte en tant qu'une vue indépendante des autres vues (une position séparée dans la fenêtre du menu de navigation) ou si elle doit être un enfant de la vue actuellement active (pas de nouvelles positions dans le menu de navigation). En ouvrant une vue, nous pouvons définir le paramètre IsPreviewMode (qui ouvre la vue en mode lecture seule où la plupart des contrôles modifiables est automatiquement bloqués) et un nombre illimité de nos propres paramètres.

Les vues peuvent être ouvertes à l'aide des méthodes disponibles dans la classe **ViewModelBase**, ainsi que dans le service **IViewManager.** La plus simple façon d'ouvrir une vue de base est d'appeler la méthode **OpenView <TView>()** où TView est le nom de la classe de vue dont nous voulons ouvrir ou activer, si elle est déjà ouverte. Paramètres supplémentaires :

- *isChild* (bool) – paramètre définissant si la vue en cours d'ouverture doit être un enfant de la vue actuellement ouverte,
- *parameters* (NavigationParameters) – paramètre permettant de transmettre au viewmodel de la vue en cours d'ouverture un nombre illimité de paramètres personnalisés,
- *isPreviewMode* (bool) – paramètre disponible déjà dans le constructeur de viewmodel de la vue en cours d'ouverture (contrairement au *parameters* qui n'est disponible que dans les méthodes d'initialisation de viewmodel ; pour savoir plus au sujet des méthodes d'initialisation de viewmodel, consultez le chapitre **Ordre d'appel des méthodes**). Ce paramètre sert à ouvrir la vue en mode d'aperçu (avec une fonctionnalité limitée). S'il est

défini comme *true*, la propriété **IsPreviewMode** dans la classe ViewModelBase sera définie en mode lecture seule. De plus, si un des contrôles suivants : **TextBox**, **SwitchBox**, **RadioButton**, **NumericTextBox**, **ComboBox**, **CheckBox** de l'espace **POS.Presentation.Core** sera inclus dans cette vue, il sera activé en mode lecture seule.

Afin d'ouvrir cette vue en mode de vue modale, il faut utiliser la méthode suivante :

OpenModalView<TView>() où TView est le nom de la classe de vue dont nous voulons ouvrir ou activer, si elle est déjà ouverte. Paramètres supplémentaires :

- *parameters* (NavigationParameters) – par analogie comme pour OpenView
- *isPreviewMode* (bool) – par analogie comme pour OpenView

La méthode ne possède pas de paramètre isChild, car toutes les vues modales sont ouvertes en mode hiérarchique (elles sont les enfants de la vue qui les a ouvertes). Une exception est le manque de relations entre les types de vues différents. Par exemple, si nous ouvrons une vue modale à partir de la vue de base actuelle, la dépendance parent-child ne sera pas créée.

Afin d'ouvrir une vue en mode de vue de message, il faut exécuter deux étapes. La première est définir dans xaml une vue de l'alignement horizontal :

```
<core:View
x:Class="Comarch.POS.Presentation.Sales.Views.OrdersView"
                    HorizontalAlignment="Stretch"
…
```

La deuxième étape consiste à appeler la méthode :

OpenMessageView<TView>() disponible dans le service ViewManager où TView est le nom de classe de la vue dont nous voulons ouvrir. Comme dans le cas de la vue modale, la relation parent-child est toujours présente, mais la relation

entre les différents types de vues est omise. Paramètres supplémentaires :

- *parameters* (NavigationParameters) – par analogie comme pour OpenView et pour OpenModalView,
- *isPreviewMode* (bool) – par analogie comme pour OpenView et pour OpenModalView

Si nous voulons afficher uniquement un simple message ou une question avec des boutons quelconques, il suffit d'utiliser le service **MonitService**. Il met à disposition les méthodes comme :

- ShowInformation – méthode affichant la vue de message avec un contenu quelconque et le bouton OK,
- ShowError – méthode affichant la vue de message avec le contenu d'exception et le bouton OK,
- ShowQuestion – méthode affichant la vue de message avec un contenu quelconque et les boutons décisifs OUI et NON,
- Show – méthode affichant la vue de message avec un contenu quelconque et des boutons d'actions prédéfinis (OK, OUI/NON) ou des boutons personnalisés.

Pour plus d'informations sur les messages, consultez le chapitre Messages dans l'article Notifications et messages.

Fermer des vues

Lorsque les vues sont fermées, nous disposons d'une méthode permettant de fermer la vue actuellement active et d'une méthode de fermeture d'une vue sélectionnée (avec l'option de fermer tous ses ancêtres). Comme dans le cas de l'ouverture d'une vue, lors de la fermeture d'une vue il est également possible d'ajouter des paramètres personnalisés supplémentaires qui seront transmis à la vue qui sera activée

après la fermeture de la vue actuelle.

Fermer la vue actuelle active toujours la vue à partir de laquelle la vue en cours de fermeture a été auparavant ouverte. Dans le cas où les vues sont en relation parent-child (lors de l'ourverture d'une vue où IsChild=true), fermer la vue-enfant fait revenir à la vue-parent.

Afin de fermer la vue active, il faut utiliser la méthode :

Dans le viewmodel, il faut appeler la méthode **Close()** qui appelle directement la méthode CloseView dans le ViewManager en transmettant sa vue en tant que paramètre. La méthode Close ferme toujours la vue associée au viewmodel actuel. Paramètres supplémentaires :

- *parameters* (NavigationParameters) – paramètre permettant de transmettre les informations de la vue en course de fermeture à la vue qui sera activée

Afin de fermer une vue sélectionnée (active ou inactive), il faut utiliser la méthode :

CloseView() qui est dans le service IViewManager. Paramètres supplémentaires de la méthode :

- *view* (IView) – paramètre définissant quelle vue sera fermée,
- *closeParents* (bool) – paramètre définissant si toutes les vues dépendantes dans la relation parent-child doivent être également fermées (les vues ouvertes avec le paramètre IsChild=true). La valeur par défaut est *true*.

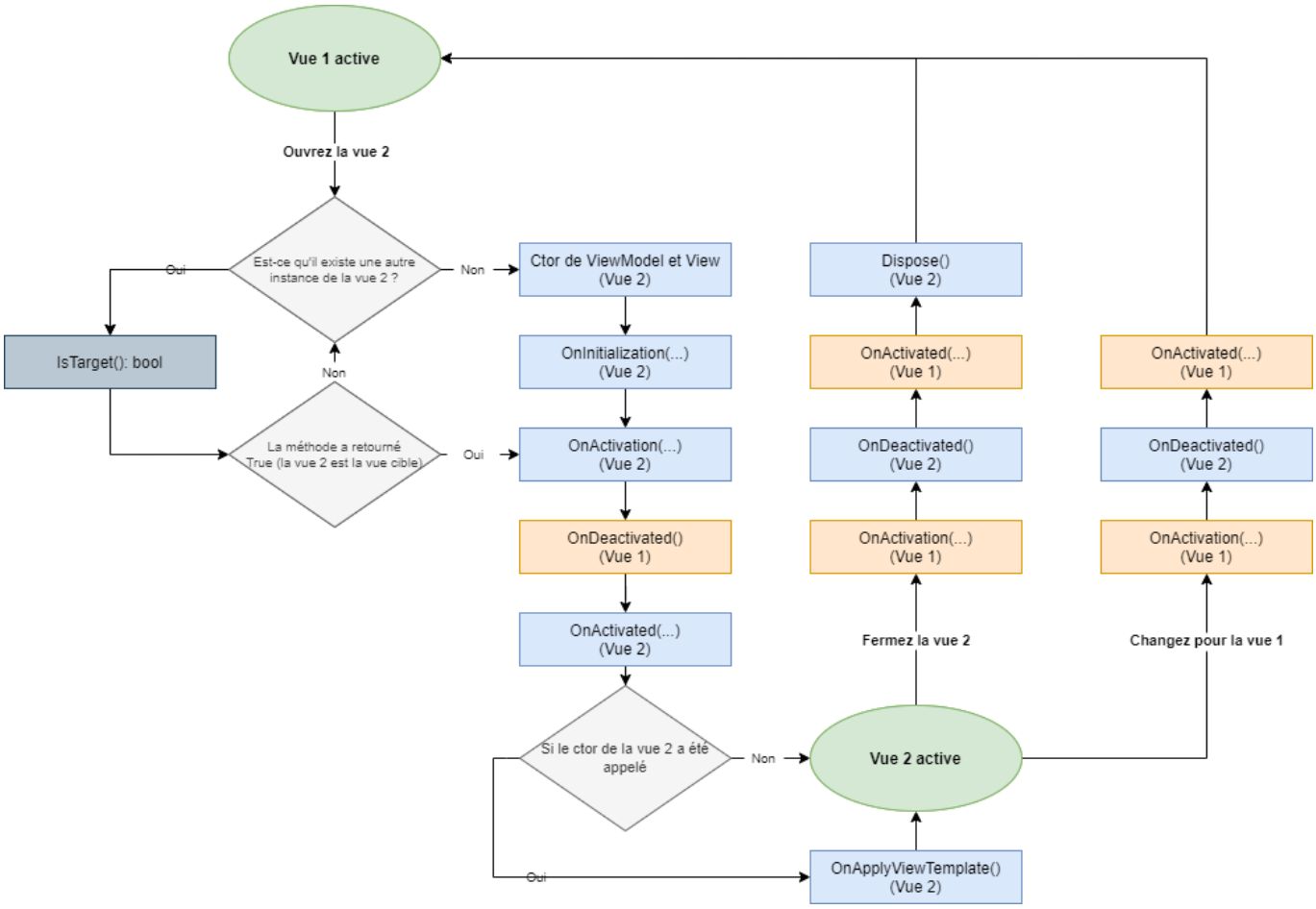
Ordre d'appel des méthodes de

navigation

Lors de la navigation entre les vues (ouverture, fermeture, basculer entre les vues actives), des méthodes spéciales sont appelées dans les viewmodels participant dans la navigation. Ces méthodes permettent d'exécuter des actions appropriées, en fonction du fait si la vue est ouverte pour la première fois, réactivée, désactivée ou fermée). Elles fournissent aussi des paramètres qui sont envoyés lors de l'appel des méthodes d'ouverture et de fermeture.

Les méthodes de navigation sont fournies par la classe de base de chaque viewmodel – ViewModelBase. Ce sont les méthodes : **OnInitialization**, **OnActivation**, **OnActivated**, **OnDeactivted**, **Dispose** et **IsTarget**. Des méthodes appropriées sont automatiquement appelées lors du processus de l'ouverture tant sur la vue à partir de laquelle nous ouvrons la nouvelle vue que la vue cible. Dans le cas du processus de la fermeture, des méthodes définies sont appelées sur la vue fermée et sur la vue qui sera activée en résultat de la fermeture de la vue actuelle. Il en va de même pour le passage d'une vue ouverte à une autre. Une question importante est quelles méthodes sont appelées lors d'une navigation particulière et dans quel ordre.

Le processus de l'ouverture, de la fermeture et du passage d'une vue à l'autre est illustré sur le schéma ci-dessous :



Description des méthodes :

- **IsTarget** (ViewModelBase)

Cette méthode est appelée lors de l'ouverture de vue, avant le constructeur de vue, sur toutes les instances des viewmodels des vues dont le type correspond à la vue en cours d'ouverture. S'il n'existe aucune instance de vue du même type que la vue en cours d'ouverture, alors une nouvelle vue sera créée. S'il existe déjà des vues ouvertes de ce type, la méthode sera appelée sur chaque viewmodel de vue dans l'ordre dans lequel elles ont été ouvertes. Si l'une d'entre elles retourne la valeur *true*, elle sera activée. En revanche, si aucune vue existante de ce type ne retourne la valeur *true*, alors une nouvelle instance sera créée.

- **OnInitialization** (ViewModelBase)

cette méthode est appelée dans le ViewModel uniquement lors de l'ouverture d'une nouvelle vue (nouvelle instance, nouvel onglet). Elle n'a pas d'accès au parent (l'appel ViewManager.GetParentViewModel(this) retourne la valeur null). Attention : dans cette méthode il ne faut pas ouvrir ni fermer les vues !

- **OnActivation** (ViewModelBase)

cette méthode est appelée à chaque ouverture de la vue ou à l'activation même. Elle n'a pas d'accès au parent (l'appel ViewManager.GetParentViewModel(this) retourne la valeur null). Attention : dans cette méthode il ne faut pas ouvrir ni fermer les vues !

- **OnActivated** (ViewModelBase)

cette méthode est appelée à chaque ouverture de vue ou à l'activation même.

- **OnDeactivated** (ViewModelBase)

cette méthode est appelée lors de la désactivation ou de la fermeture d'une vue.

- **Dispose** (ViewModelBase)

cette méthode est appelée uniquement lors de la fermeture de vue. Elle permet de libérer des ressources non-gérables par le Garbage Collector ou d'arrêter les fils-enfants.

- **OnApplyViewTemplate** (View)

Cette méthode est appelée dans le code-behind de la classe de vue, une seule fois après la création d'une instance de vue et après le chargement de toutes les composantes (contrôles) de vue.

Notifications et messages

Dans l’application POS le moyen de base de communication avec l’utilisateur sont les notifications et les messages. Les notifications permettent de transmettre une courte information qui n’empêche pas le travail d’utilisateur et elles apparaissent sous forme d’un rectangle à côté droite de l’écran, sous la section de statut. Les messages en revanche servent à transmettre des informations plus longues ou plus importantes et l’utilisateur ne peut pas les ignorer, car ils apparaissent sous forme de vue de message. Ils sont largement utilisés en cas d’erreurs.

Notifications

Chaque notification est composée du contenu affiché (un texte plus long que 6 lignes sera coupé) et de l’icône. On distingue trois types de notifications, chaque avec une icône différente :

1. Information
2. Avertissement
3. Erreur

Lorsqu’une notification est appelée, elle apparaît à droite de l’écran et disparaît au bout de trois secondes par défaut. Le temps d’affichage de notification peut être modifiée dans le fichier de configuration de l’application en modifiant la valeur de la clé **NotificationTimeout.** Dans le cas où l’utilisateur place le pointeur de la souris sur la notification, elle ne disparaît pas, mais elle sera affichée aussi longtemps que le pointeur est placé sur elle. Si lors de l’affichage d’une notification, une autre apparaît, la première notification sera deplacée d’une position vers le

bas. À un moment sur l'écran peuvent être affichées cinq notifications. Lorsqu'il y en a plus, les notifications suivantes seront spoulées et affichées après la disparition des notifications actuellement présentées.

Afin d'afficher une notification, il faut utiliser le service **INotificationService** (chaque viewmodel a l'accès à ce service par la propriété **NotificationService**) et appeler la méthode **Show(string msg, NotifyIcon icon)** où le paramètre *msg* et le contenu de message et *icon* est l'enum définissant le type de notification (c'est-à-dire son icône).

Messages

Les messages sont affichés à l'aide de la vue de message (les traits caractéristiques de cette vue ont été présentés dans le chapitre Introduction). Chaque message est composé de l'en-tête (un titre court écrit en utilisant une police plus grande que celle utilisée dans le texte du message), du texte de message (pas de limités de caractères, si le texte est plus long, une glissière apparaîtra permettant de défiler le texte), de l'icône optionnelle (un choix parmi quelques icônes prédéfinies) et d'au moins un bouton. La configuration des boutons dépend du type de message, mais elle peut être personnalisée.

Afin d'afficher un message, il faut utiliser le service **IMonitService** (chaque viewmodel a l'accès à ce service par la propriété **MonitService**) et appeler une des méthodes suivantes, en fonction des besoins :

- **ShowInformation**

Cette méthode affiche un message informatif. Icône d'information et bouton OK.

- **ShowError**

Cette méthode affiche un message d'erreur. Icône d'erreur et

bouton OK.

- **ShowQuestion**

Cette méthode affiche un message interrogatif. Icône de question et boutons : OUI, NON.

- **Show**

Cette méthode affiche un message du type personnalisé. Il est possible de définir quelle icône sera utilisée (parmi les icônes prédéfinies : information, question, avertissement, erreur ou pas d'icône), indiquer quels boutons d'action seront disponibles (OK, OUI/NON) ou définir ses propres boutons.

Services

L'application POS utilise le mécanisme des conteneurs Unity. Il permet d'accéder rapidement et facilement à tous les services enregistrés auparavant dans une classe quelconque de ViewModel et de View. Le mécanisme résout également toutes les dépendances, à condition qu'elles aussi aient été enregistrées auparavant. Un exemple serait toute vue qui est obtenu du conteneur avec l'implémentation simultanée d'un viewmodel dépendant dans son constructeur. Ceci est possible, car chaque viewmodel est auparavant enregistré dans un conteneur (voir le chapitre Enregistrer vues à naviguer et à gérer l'affichage dans l'article Créer des vues). Dans ce viewmodel sont à son tour implémentés les autres services requis qui lui sont indispensables (à condition qu'ils aient été enregistrés auparavant).

Téléchargement des services à partir du conteneur

Une instance d'un service donné peut être téléchargée de trois façons :

1. Définir une propriété publique avec l'attribut **Dependency** de l'espace **Practices.Unity** ajouté.

```
[Dependency]
public IProductsService ProductsService { get; set; }
```

2. Implémenter une instance dans le constructeur de viewmodel à l'aide d'une déclaration simple de paramètre.

```
public OrdersViewModel(IProductService productService) { … }
```

3. Appeler la méthode **Resolve** sur l'objet IUnityContainer disponible dans la classe ViewModelBase sous la propriété

```
var productsService = Container.Resolve<IProductsService>();
```

Une instance de service est créée à chaque fois au moment de son téléchargement du conteneur. Une exception est le cas d'un service enregistré comme singleton, dont l'instance est créée une seule fois, lors du premier essai du téléchargement.

Enregistrement des services personnalisés dans le conteneur

Un service peut être enregistré à tout moment, dans chaque espace où nous avons accès à l'objet IUnityContainer. Si nous souhaitons être sûrs que le service sera disponible dès le début, il faut l'enregistrer dans la classe Module. C'est le cas pour tous les viewmodels. Un service peut être enregistré sur la base de son interface. Si la classe de service n'a pas d'interface, il n'est pas obligatoire de l'enregistrer, sauf si nous voulons qu'elle soit disponible en tant que singleton.

Afin d’enregistrer un service dans la classe Module, il faut enregistrer la méthode suivante :

Register<TInterface,TClass>() où TInterface est le nom d’interface et TClass est le nom de la classe implémentant cet interface. Paramètres supplémentaires :

- *singleton* (bool) – paramètre définissant si l’instance téléchargée du conteneur doit être à chaque fois créée à nouveau ou si la même instance doit être toujours retournée. La valeur de ce paramètre est par défaut

Ci-dessous, vous trouverez un exemple d’enregistrement dans la classe Module du service de classe CustomService implémentant l’interface ICustomService en tant que singleton :

Register<ICustomService, CustomerService>(true);

Si nous tenons à ce qu’une instance du service enregistré soit immédiatement initialisée, alors que ce service contient des dépendances qui sont implémentées automatiquement, il faut vérifier si cela sera possible. Dans le cas où le service dépend d’un autre service qui existe dans un autre module, il faut vérifier qu’il a déjà été enregistré. Pour en être sûr, l’initialisation de l’instance (téléchargement du conteneur) doit être exécuté au sein de la méthode **AfterAllModulesLoaded** (qui se trouve dans ModuleBase).

Liste des services disponibles dans POS

Les services sont divisés en services de bases relatives au cœur de l’application POS et services métiers relatives à la logique du fonctionnement métier de l’application.

La liste des services de bases comprend :

- IMonitService (Comarch.POS.Presentation.Core.Services)

Service permettant d’afficher les messages (informatifs et interrogatifs) – voir le chapitre Messages dans l’article Notifications et messages

- INotificationService (Comarch.POS.Presentation.Core.Services)

Service permettant d’afficher les notifications – voir le chapitre Notifications dans l’article Notifications et messages

- IViewManager (Comarch.POS.Presentation.Core.Services)

Classe permettant de contrôler les vues (ouverture et fermeture) – voir l’article Naviguer entre les vues

- ILoggingService (Comarch.POS.Library.Logging)

Service permettant d’enregistrer les informations dans le fichier journal

- ISecurityService (Comarch.POS.BusinessLogic.Interfaces.Security)

Service responsable de l’authentification et de l’autorisation d’utilisateur POS – pour plus d’informations, voir le chapitre Vérification des autorisations

- IAutorizationService (Comarch.POS.Presentation.Base.Services)

Service permettant de valider les autorisations des utilisateurs POS connectés – pour plus d’informations, voir le chapitre Vérification des autorisations

La liste des services métiers comprend :

- IConfigurationService (Comarch.POS.Library.Settings)

Service permettant d’accéder à la configuration de l’application

- IFiscalizationService (Comarch.POS.Presentation.Fiscalization.Services)

Service de fiscalisation

- ISynchronizationService (Comarch.POS.Synchronization.Interfaces)

Service de synchronisation

- IPrintoutManager (Comarch.POS.Presentation.Core.Services)

Service d’impression

- Ainsi que tous les autres services dans l’espace Comarch.POS.BusinessLogic.Interfaces

---

Contrôles

La plupart des contrôles utilisées dans l’application POS provient de l’espace nom **Comarch.POS.Presentation.Core.Controls**. Une exception est par exemple le contrôle TextBlock qui ne possède pas son équivalent dans POS et qui est un contrôle standard fourni par la plateforme .NET. Certains contrôles particuliers créés aux fins des exigences commerciales spécifiques sont placés dans les modules comme Comarch.POS.Presentation.Products, Comarch.POS.Presentation.Customers, Comarch.POS.Presentation.Sales. Des composants sélectionnés sont décrits ci-dessous.

TextBlock

Un des contrôles de base qui permet d’afficher le texte. Il

provient de l’espace nom System.Windows.Controls.

SlideTextBlock

Un contrôle TextBlock enrichi de l’effet d’animation du texte entrant et sortant de l’écran. Chaque fois qu’une valeur est saisie dans la propriété Text résulte en ce que ce texte apparaîtra avec l’effet d’animation. Si avant un autre texte était saisi, lui et son effet d’animation disparaîtront en ce moment. Ce contrôle est par exemple utilisé sur l’écran de démarrage qui informe sur le progrès dans le démarrage de l’application.

Options de configuration disponibles :

SlideInDuration : double – temps d’animation d’entrée du texte sur l’écran, exprimé en millisecondes. Le temps par défaut est 500.

SlideOutDuration : double – temps d’animation de sortie du texte de l’écran, exprimé en millisecondes. Le temps par défaut est 500.

SlideInLength : int – distance à parcourir par le texte dans l’animation d’apparition du texte. La distance par défaut est 400.

SlideOutLength : int – distance à parcourir par le texte dans l’animation de disparition du texte. La distance par défaut est 500.

TextBox

Contrôle permettant à l’utilisateur de saisir un texte.

Propriétés sélectionnées :

Hint : string – un indice ; un texte qui sera affiché dans le contrôle s’il n’est pas complété par l’utilisateur. Le texte sera affiché en couleur définie par HintForeground. L’indice

peut être affichée en l’un des deux modes qui peuvent être configurés dans le fichier de configuration de l’application sous la clé **ClearHintOnFocus.** Sa valeur par défaut est false – le texte de l’indice disparaît lorsque l’utilisateur commence à saisir les caractères ; si la valeur est true, le texte disparaîtra immédiatement après avoir sélectionné le curseur dans le contrôle.

HintForeground : Brush – couler de l’indice.

NumericTextBox

Un contrôle TextBox permettant de saisir uniquement des valeurs numériques.

Propriétés sélectionnées :

AllowOnlyPositiveValues : bool – drapeau permettant de contrôler si le système accepte de saisir les valeurs négatives

ValueType : ValueType – paramètre permettant de définir si le système accepte uniquement les valeurs entières.

Precision : int – précision pour les valeurs à virgule flottante

MinValue et *MaxValue : object* – possibilité de définir la plage des valeurs admissibles

IsDefaultNullValue ; bool – paramètre permettant de laisser le champ vide

PasswordBox

Contrôle permettant de saisir un texte d’une façon sécure, sans possibilité d’apercevoir les caractères saisis dans l’interface.

Button (CancelButton, AcceptButton, etc.)

Le bouton est l’un des contrôles de base qui permet à l’application d’interagir avec l’utilisateur. Il permet d’effectuer une action définie au moment où l’utilisateur appuie sur cet élément. Les boutons par défaut ressemblent (car leurs dimensions réelles sont 90×80) aux carrés en couleurs différentes. Chaque bouton peut contenir un texte quelconque décrivant son fonctionnement, un raccourci clavier (permettant d’appeler une action sans cliquer) et une icône (en format vectoriel ou d’un fichier graphique, par exemple .png).

Toutes les propriétés des boutons disponibles (comme leur couleur, le texte, les dimensions, etc.) peuvent être définies livrement, néanmoins afin de rendre le travail plus facile, nous avons préparé des boutons dédiés comme le bouton de fermeture (CancelButton) ou d’enregistrement (SaveButton) qui ont des couleurs prédéfinies pour que l’utilisateur ne les définit pas à chaque fois. C’est aussi le cas du bouton de texte, du raccourci clavier et de l’icône. Toutes les variantes de bouton disponibles ont été ajoutées dans l’espace nom **Comarch.POS.Presentation.Core.Controls.Buttons**

Revue des propriétés plus importantes :

Content : object – le contenu est affiché à l’intérieur du bouton, en général c’est un texte

Orientation : Orientation – alignement du texte et de l'icône. L'alignement est par défaut vertical (Vertical) – le text est sous l'icône. Dans le cas de l'alignement horizontal l'icône sera à gauche et le texte à droite.

Width: double – largeur du bouton, par défaut 80

Height : double – hauteur du bouton, par défaut 90

Margin : Thickness – marge du bouton

InactiveVisibility : Visibility – visibilité du bouton inactif lorsque la paramètre IsEnabled=false (par exemple lorsque l'activateur de l'action retourne la valeur false)

Command : ICommand – action sous forme de commande, le biding utilisé le plus souvent pour le type DelegateCommand

DisabledState : Brush – couleur du bouton lorsqu'il est affiché comme inactif

ImageKey : ImageKey – paramètre d'une des icônes graphiques (PngValue) ou vectorielles (SvgValue) disponibles dans le motif actuellement sélectionné

ImageWidth : double – largeur de l'icône

ImageHeight : double – hauteur de l'icône

IsImageVisible : bool – visibilité de l'icône

ImageMargin : Thickness – marge de l'icône

ContentVisibility : Visibility – visibilité du texte

IsScaleContent : bool – régulation du texte excédant la zone du bouton, par défaut false

Shortcut : Shortcut – raccourci clavier remplaçant le clic

IsShortcutVisible : bool – visibilité du raccourci clavier

CheckBox

Contrôle permettant à l'utilisateur de sélectionner un des deux états logiques. Il est représenté sous forme de bouton dont la couleur change en fonction de l'état sélectionné.

Propriétés sélectionnées :

IsChecked : bool – état logique actuel, true – bouton enfoncé, false – bouton non enfoncé

CheckedStateBackground : Brush – couleur de l'arrière-plan du bouton enfoncé

CheckedStateForeground : Brush – couleur du texte dans le bouton enfoncé

DisabledStateBackground : Brush – couleur de l'arrière-plan du bouton inactif, lorsque le paramètre IsEnabled=false

DisabledStateForeground : Brush – couleur du texte dans le bouton inactif, lorsque le paramètre IsEnabled=false

Ainsi que la plupart des propriétés du contrôle Button.

SwitchBox

Ce contrôle est basé sur le contrôle CheckBox, mais la façon d'affichage est différente. Tout comme CheckBox, il peut adopter l'un des deux états (true – marqué/activé, false – non marqué/désactivé). Il ressemble à un commutateur qui peut être enrichi d'une description de l'activité réalisée par le changement de son état.

Propriétés sélectionnées :

SwitchOffContent : object – contenu qui apparaîtra lorsque l'état de commutateur est false

SwitchOnContent : object – contenu qui apparaîtra lorsque

l’état de commutateur est true

Content : object – contenu décrivant à quoi sert le commutateur (quelle tâche réalise-t-il)

IsChecked : bool – état de commutateur ; activé – true, désactivé – false

Ainsi que la plupart des propriétés du contrôle CheckBox.

NullableSwitchBox

Ce contrôle est basé sur le contrôle SwitchBox, mais la façon d’affichage est différente. Le contrôle peut adopter l’un des trois états (<true> – bouton droit marqué, <false> – bouton gauche marqué, <null> – aucun bouton marqué). L’état indéfini <null> peut être uniquement l’état de départ (s’il n’a pas été défini autrement), après avoir changé en un autre état, il n’est plus possible de revenir à l’état indéfini. Le contrôle ressemble à une pilule et il peut être enrichi d’une description de l’activité réalisée par le changement de son état.

Propriétés sélectionnées :

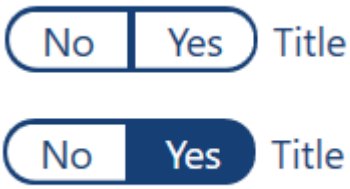
SwitchOffContent : object – contenu qui apparaîtra lorsque l’état de commutateur est false

SwitchOnContent : object – contenu qui apparaîtra lorsque l’état de commutateur est true

Content : object – contenu décrivant à quoi sert le commutateur (quelle tâche réalise-t-il)

IsChecked : bool?

Ainsi que la plupart des propriétés du contrôle CheckBox.



RadioButton

Le fonctionnement et l'aperçu de ce contrôle ressemblent au contrôle CheckBox. La différence réside dans le fait qu'en définissant plusieurs RadioButtons et en les nommant en commun dans un groupe **GroupName**, seul un contrôle de tout le groupe peut être activé à la fois.

Propriétés sélectionnées :

IsChecked : bool – état d'activation, true – contrôle activé, false – contrôle désactivé

CanUndoSelection : bool – possibilité de désactiver (paramètre IsChecked=false) le contrôle en le cliquant encore une fois

ComboBox

Contrôle permettant à l'utilisateur de sélectionner l'une des options disponibles sous forme d'une liste déroulante. Il ressemble au contrôle TextBox, mais il a une icône supplémentaire permettant de dérouler la liste des options à sélectionner.

Propriétés sélectionnées :

Hint : string – indice affiché lorsqu'aucune valeur n'a été sélectionnée dans la liste déroulante.

HintForeground : Brush – couler de l'indice.

PopupBackground : Brush – couleur de l'arrière-plan de la liste déroulante.

ItemsSource : IEnumerable – collection des options à sélectionner.

SelectedItem : object – option sélectionnée.

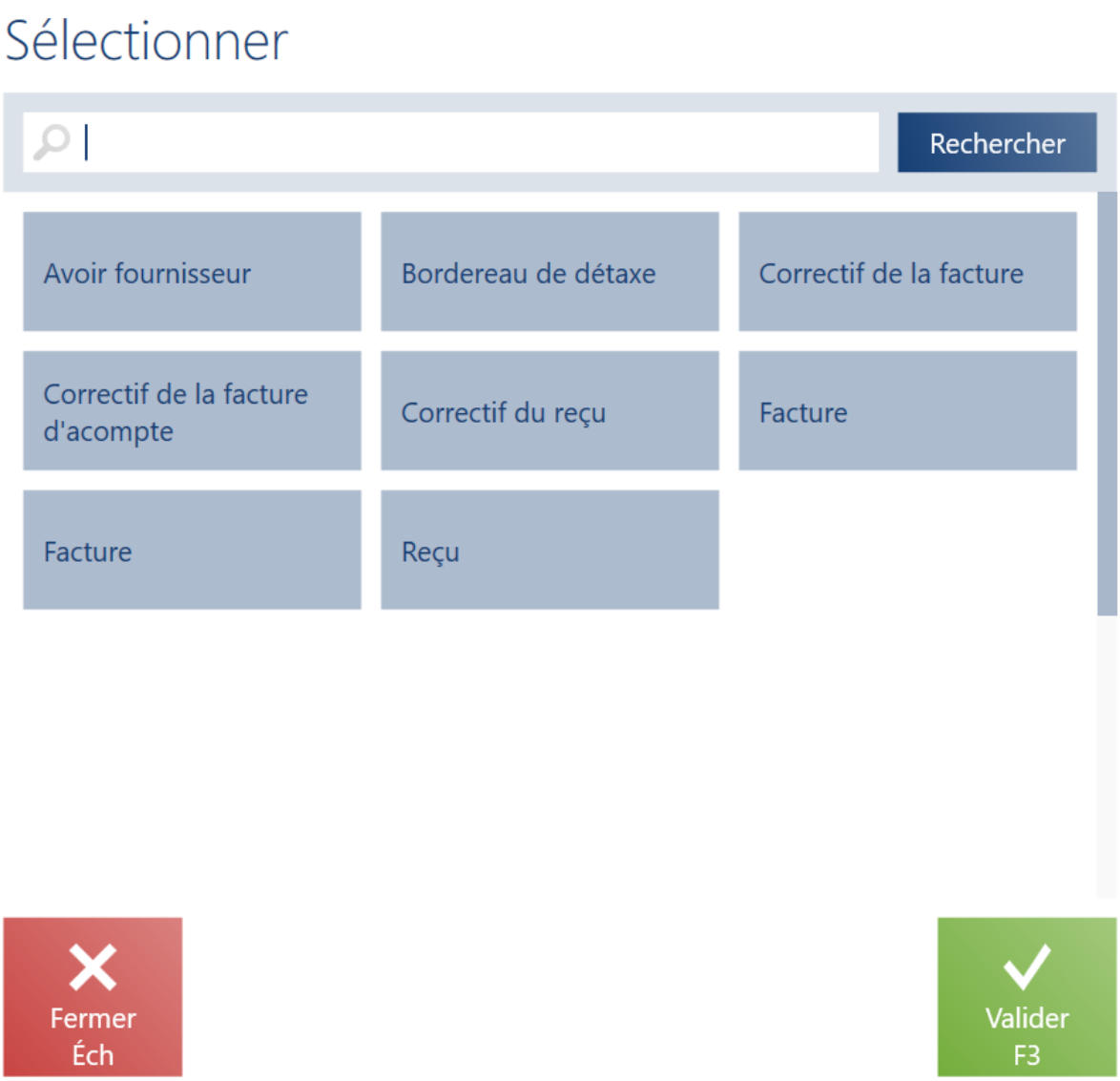
ComboBox2

Un équivalent développé du contrôle ComboBox où les options à sélectionner sont affichées non pas dans un pop-up simple, mais dans une fenêtre modale. Dans cette fenêtre les options sont par défaut affichées sous forme d'une liste des RadioButtons, mais il est également possible de créer un contenu personnalisé.

Le contrôle est composé des deux éléments :

- un élément présentant l'option sélectionnée (sous forme de TextBlock avec une description et un Button en-dessous de taille non standard)

- une fenêtre modale avec la liste des options (par défaut liste des RadioButtons)



Propriétés sélectionnées :

Source : IComboBoxSource – propriété requise, source de données – pour la vue modale par défaut il faut utiliser le type **ComboBoxSource<T>** où T est le type de données des options à sélectionner

Label : string – texte affiché en-dessus du bouton de contrôle.

LabelFontSize : double – taille de la police du label.

LabelFontWeight : FontWeight – poids de la police du label.

LabelFontStyle : FontStyle – style de la police du label.

FontSize : double – taille de la police du texte à l'intérieur du bouton.

FontWeight : FontWeight – poids de la police du texte à l'intérieur du bouton.

FontStyle : FontStyle – style de la police du texte à l'intérieur du bouton.

Il n'est pas possible de modifier directement l'arrière-plan du contrôle, la couler du texte et l'arrière-plan du bouton. Ces modifications dépendent des couleurs de base sélectionnées dans le motif.

Pour plus d'informations, voir le chapitre Exemples d'utilisation du contrôle ComboBox2 dans l'article Exemples.

TileButton

Ce contrôle est en fait un Button dont les dimensions sont différentes que les dimensions d'un Button standard. Il hérite de la classe Button et de ce fait il a toutes les propriétés d'un bouton. Il est utilisé entre autres dans le menu principal. Grâce à la méthode d'enregistrement des vues dans le menu principal appropriée il n'est pas nécessaire d'y créer ce contrôle manuellement.

ButtonSpinner

Ce contrôle, en coopération avec un autre contrôle du type input (comme le plus souvent TextBox) ajouté à son intérieur, ajoute deux boutons de contrôle supplémentaires (+ et -). Chaque enfoncement d'un de ces boutons déclenche l'événement Spin. Il peut être utilisé à contrôler le contenu du contrôle interne.

Propriétés sélectionnées :

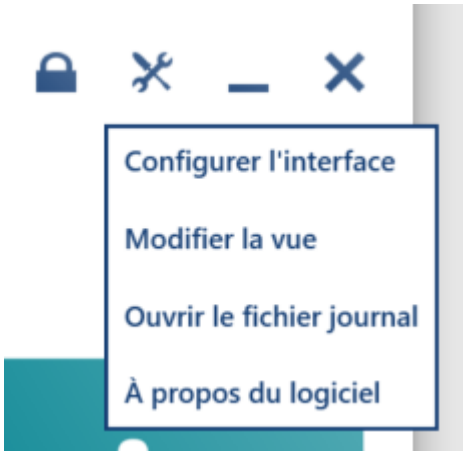
Spin : EventHandler<ButtonSpinnerArgs> – événement déclenché

après avoir cliqué un des boutons de ButtonSpinner (+ ou -).

Pour plus d'informations, voir le chapitre Exemple d'utilisation du contrôle ButtonSpinner dans l'article Exemples.

ComboBoxButton

Ce contrôle est l'extension du contrôle ComboBox, décrit en-dessus. Son fonctionnement pourtant, ressemble plutôt au fonctionnement du contrôle Button. Le contrôle ComboBox permet d'afficher le bouton sous forme d'icône qui, lorsqu'on clique dessus, déroule une liste définie d'actions. Les éléments de la liste sont définis à l'aide des éléments ComboBoxItem. Un exemple d'utilisation serait le menu de statut :



Propriétés sélectionnées :

LabelContent : object – contenu supplémentaire facultatif cliquable à gauche de l'icône.

ImageSource : Canvas – icône du bouton sous forme vectorielle.

SelectedItem : object – option sélectionnée dans la liste d'actions.

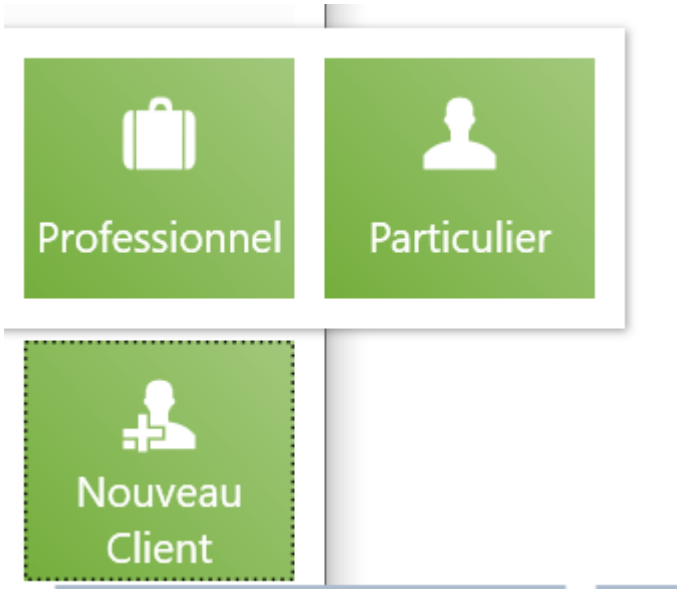
SelectedIndex : int – index de l'option sélectionnée dans la liste d'actions.

Pour plus d'informations, voir le chapitre Exemple

d'utilisation du contrôle ComboBoxButton dans l'article Exemple.

MultiButton

Ce contrôle permet d'agréger plusieurs boutons sous forme d'un seul bouton. Cliquer le bouton principal fait afficher un popup avec la liste complète de boutons. Dans le cas où la liste contiendrait un seul bouton, le popup n'apparaîtra pas, mais l'action sera immédiatement exécutée. Les boutons agrégés peuvent également être entièrement gérés dans l'application, avec la possibilité de modifier leur ordre et de les masquer (ils sont intégrés dans un conteneur gérable).



Propriétés sélectionnées :

SubButtons : List<Button> – liste définie des boutons agrégés.

Pour plus d'informations, voir le chapitre Exemple d'utilisation du contrôle MultiButton dans l'article Exemples.

ItemsContainer

Ce contrôle est basé sur le contrôle internet ItemsControl qui veut dire qu'il permet de remplacer à son intérieur d'autres contrôles d'une façon organisée, le plus souvent sous forme

d’éléments ordonnés verticalement ou horizontalement. Le contrôle a été enrichi de mécanismes permettant une gestion dynamique de son contenu défini, pendant le fonctionnement de l’application. Lors du travail ordinaire, le contrôle présente le contenu dans l’ordre défini dans xaml. Cet ordre, ainsi que la visibilité des composants définis auparavant, peut être modifié en cours du fonctionnement de l’application (un utilisateur ordinaire peut le faire) à l’aide du panneau de gestion des vues de l’application préparé à cette fin. Ces modifications sont enregistrées dans le motif actuellement sélectionné (à l’exception du motif par défaut). La possibilité de gérer les composants est contrôlée en définissant la propriété LayoutId sur le contrôle utilisé dans la vue et sur chaque élément défini à l’intérieur de celui-ci. En d’autres termes, pour que le contrôle soit gérable, tant il que ses composants doivent avoir des identifiants uniques définis. Les composants peuvent être n’importe quel contrôle, mais le plus souvent c’est un seul type, par exemple la liste de boutons. Les éléments peuvent être définis dans le xaml ou construits dans le code, sans utiliser le binding, mais par l’ajout à la collection en utilisant la méthode du contrôle **AddItem.**

Propriétés sélectionnées :

AutoWrapButtons : bool – contrôle de la possibilité d’enrouler automatiquement les boutons à l’intérieur du conteneur dans le cas où il n’y a pas assez d’espace pour les afficher et de les convertir en un bouton « Plus » du type MultiButton (les boutons excédants sont cachés dans un popup). De plus, il est possible de définir dans chaque contrôle du type Button si le bouton en question doit être enroulé (à l’aide de la propriété ItemsContainer.NoWrapButton ou directement dans le panneau de gestion des vues de POS dans l’onglet général -> ne pas agréger).

Orientation : Orientation – alignement des composants, vertical ou horizontal.

Pour plus d'informations, voir le chapitre Exemple d'utilisation du contrôle ItemsContainer dans l'article Exemples et le chapitre Gestion des éléments dans le conteneur ItemsContainer dans l'article Créer des vues gérables.

Grid

Deuxième contrôle très important dans le cadre de la gestion des vues. Il permet de construire librement une vue grâce aux colonnes et aux lignes où sont emplacés des éléments suivants. Comme dans le cas d'ItemsContainer, le contrôle Grid permet à l'utilisateur de gérer dynamiquement ses éléments pendant le fonctionnement de l'application. Les éléments définis à l'intérieur de grid peuvent être supprimés, transférés entre les cellules (créées à partir des colonnes et des lignes) de grid et même les déplacer vers les conteneurs ItemsContainer et les autres Grids (à condition que ceux-ci aient été définis auparavant dans le grid). Il est récommandé de construire sur la base de ce contrôle des vues entièrement gérables. D'abord, il faut ajouter ce contrôle dans le xaml de vue et ensuite définir dans son intérieur tous les autres contrôles requis à construire une vue. La vue elle-même n'est plus créée dans xaml, mais directement dans le panneau de gestion des vues de l'application, alors que toute la définition de vue (la mise en page des éléments) est enregistrée dans le motif.

Propriétés sélectionnées :

ColumnDefinition : string – définition du nombre et de la taille des colonnes selon le modèle : c1, c2, c3, … cx où c1, c2, c3, … cx sont les valeurs définissant les colonnes suivantes ; valeurs disponibles :

* – ajuste la largeur de colonne proportionnellement, en fonction du nombre de colonnes. Il est possible de l'utiliser avec un chiffre pour définir les autres proportions de division, par exemple 2*

Auto – ajuste la largeur de colonne automatiquement, en fonction des besoins de l'élément qu'elle contient,

[chiffre] – définir la largeur de colonne fixe, par exemple 100

Un exemple de définition d'un grid avec trois colonnes où la largeur de la première est 150, la largeur de la deuxième est en fonction des besoins de l'élément dedans et la largeur de la troisième occupe le reste de l'espace disponible :

ColumnDefinition="150,Auto,*"

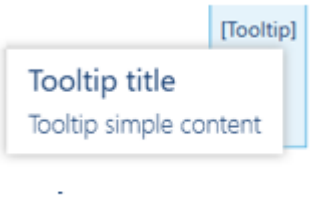
RowDefinition : string – définit le nombre et la taille des lignes selon le même modèle que dans la définition des colonnes,

Afin de définir dans quelle colonne et dans quelle ligne doit être placé un élément défini à l'intérieur du grid, il faut définir la propriété **Grid.Position** sur lui. Par exemple, si un élément du grid doit être dans la troisième colonne, deuxième ligne, il faut écrire : Grid.Position="1,2,1,1" (1 – deuxième ligne, 2 – troisième colonne, 1,1 – éléments occupera une ligne et une colonne).

Pour plus d'informations, voir le chapitre Exemple d'utilisation du contrôle Grid dans l'article Exemples et le chapitre Gestion des éléments dans un conteneur Grid dans l'article Créer des vues gérables.

Tooltip

Ce contrôle permet d'afficher les informations supplémentaires (par exemple une description détaillée des champs) sous forme d'un popup qui apparaît lorsque l'utilisateur passe la souris sur l'espace définie auparavant (Caption). Les informations détaillées peuvent contenir le titre (Title) et le contenu (Content).



Propriétés sélectionnées :

Caption : object – tout contenu (par exemple un texte ou des autres contrôles) qui lors du survol de la souris affiche un tooltip

Tite : string – titre de tooltip, affiché dans le popup

Content : string – contenu affiché dans le tooltip

TabControl et TabControlItem

Ces contrôles permettent d'afficher les données sous forme d'onglets. Il est possible de construire, en combinaison avec le contrôle de grid, une vue avec plusieurs TabControl (conteneurs d'onglets) et avec plusieurs onglets (TabControlItem), que les utilisateurs pourront librement basculer entre les TabControls de la vue au niveau du panneau de gestion des vues. Un exemple excellent d'utilisation des onglets et la vue du fiche client. Elle est composée de quatre contrôles TabControl qui permettent d'afficher les onglets comme Adresses, Attributs, Statistiques, Personnes de contact dans un n'importe quel conteneur TabControl.

Chaque conteneur peut être divisé en deux zones. La zone de la liste des RadioButtons permettant de basculer entre les onglets actifs. Si un TabControl possède uniquement un TabControlItem, alors cette zone ne sera pas affichée (en dehors le mode de gestion). La zone contenu pour afficher le contenu de l'onglet actuellement actif. La mise en page par défaut des zones (contenu en haut, boutons en bas) peut être librement modifiée.

Propriétés sélectionnées de TabControlItem:

TabContent : object – définition du contenu d'onglet

Source : TabControlItemSource – source de données de la définition du contenu d'onglet. La source de donnée par défaut est DataContext du contrôle TabControl, c'est-à-dire le viewmodel, sauf si une autre source n'a pas été définie.

La classe TabControlITemSource est une classe abstraite fournissant uniquement deux propriétés permettant de contrôler le fonctionnement des onglets. La première est le drapeau **IsLoaded** qui doit être défini comme true une fois les données ont été chargées et la deuxième est le drapeau **IsDesignMode** défini comme true dans le design viewmodel qui active le mode de gestion dans l'onglet.

Pour plus d'informations, voir le chapitre Exemple d'utilisation des contrôles TabControl et TabControlItem dans l'article Exemples.

ScrollViewer

Ce contrôle permet de défiler dans la vue le contenu excédant. Il fonctionne uniquement en mode du défilement horizontal et il est paramétré de cette façon par défaut. Afin d'utiliser le mode vertical, il faut utiliser un contrôle de l'espace .NET (System.Windows.Controls.ScrollViewer).

FieldControl et validation

Le contrôle de champ (FieldControl) sert à emballer les champs de formulaires. Il permet d'obtenir la fonctionnalité de gestion de la validation du niveau de gestion de l'interface

d’utilisateur (il est nécessaire de définir un LayoutId dans le contrôle). Son application rend obsolète le besoin d’utiliser un TexBlock/Label pour définir la description de champ.

Par exemple, si l’on veut ajouter à la vue un champ textuel que l’utilisateur devra compléter, il faut ajouter au fichier xaml :

```
<controls:FieldControl core:Layout.Id="OrdersViewTextField1"
                                    Source="{Binding Field1}"
                                     LabelText="Description du
champ textuel:">

     <controls:TextBox core:Layout.Id="FieldTestsViewTextBox2"
           Text="{Binding Field1.Data,
                UpdateSourceTrigger=PropertyChanged,
ValidatesOnDataErrors=True}"/>
</controls:FieldControl>
```

La propriété **Source** du contrôle FieldControl requiert de transmettre le type **FieldSourceBase.** Cette classe est abstraite et c’est pourquoi il faut utiliser une des classes-enfants : **FieldSource<T>**, **FieldSourceCollection<T>** où, en cas de besoin, créer une implémentation personnalisée basée sur la classe FieldSourceBase.

La classe générique **FieldSource<T>** contient une seule valeur du type T dans la propriété appelée **Data.** Elle peut être utilisée en combinaison avec les contrôles qui nécessiteront de saisir une valeur, comme le contrôle TextBox. La classe **FieldSourceCollection<T>** contient une collection d’éléments et l’élément actuellement sélectionné/marqué. La collection est emplacée dans la propriété **Items**, alors que l’élément actuellement sélectionné dans la propriété **SelectedItem.** Cette classe peut être utilisée en combinaison avec le contrôle ComboBox. Pour que la validation fonctionne dans le contrôle cible (dans ce cas TextBox), il faut ajouter dans le binding la ligne suivante : **ValidationOnDataErrors=true**.

Ensuite, dans le viewmodel de notre vue nous ajoutons la propriété Field1 du type FieldSource<string> et nous créons l'instance. Facultativement, il est possible de définir dans le constructeur la valeur par défaut qui apparaîtra dans le champ TextBox après l'ouverture de la vue.

```
public FieldSource<string> Field1 { get; set; }
…
public OrdersViewModel()
{
    Field1 = new FieldSource<string>()
            {
                Error = "message d'erreur simple"
            };
    …
}
```

Les classes FieldSource<T> et FieldSourceCollection définissent la règle par défaut de validation d'un champ. Elle consiste à vérifier que le champ n'est pas vide si la propriété Requirement a été définie dans la gestion des vues. Afin de définir une logique de validation personnalisée, il faut utiliser la méthode **SetValidationRule.**

La propriété **Error** (voir l'exemple d'utilisation en-dessus) permet de redéfinir le message d'erreur par défaut qui est affiché sous forme de ToolTip après le survol de la souris sur le champ sélectionné.

À part de ceci, la classe **FieldSourceBase** fournit à lecture les propriétés comme :

IsValid – permet de vérifier si le champ est correctement validé.

IsRequired – permet de vérifier si le champ a été marqué par l'utilisateur dans la configuration de vue comme requis.

De plus, pour forcer manuellement dans un viewmodel une vérification de validation, il faut utiliser la méthode

RaiseValidation.

Pour plus d'informations, voir l'article Exemple d'utilisation du contrôle FieldControl dans l'article Exemples.

DataGrid

Ce contrôle permet d'afficher les données complexes sous forme de tableau avec colonnes qui fonctionnent en tant qu'en-tête et avec lignes qui contiennent les données détaillées. Les informations présentées par le contrôle peuvent être groupées par tout colonne sélectionnée par l'utilisateur dans le panneau de gestion des vues. De plus, les données groupées peuvent être agrégées (par exemple sommation, moyenne ou une implémentation personnalisée). Les données pour le contrôle peuvent être téléchargées de manière asynchronique. Le contrôle prend en charge le contrôle du tri et de la lecture des données pendant le défilement (pagination) en envoyant des demandes appropriées à la source de données. Il ne tri pas lui-même les données, mais informe de quelle manière doivent-elles être triées et la source doit fournir les données triées. Pour plus d'informations sur le fonctionnement du contrôle, voir l'article Afficher un ensemble de données à l'aide du contrôle DataGrid

DatePicker et Calendar

Le contrôle Calendar affiche le calendrier et permet d'indiquer la date – l'année, le mois et le jour. Le contrôle DatePicker est un ComboBox qui, lorsqu'on clique dessus, affiche un popup avec un calendrier (Calendar). Après avoir indiqué la date, celle-ci est transférée à ComboBox.

Propriétés sélectionnées de DatePicker :

UndefinedDateText : string – si la valeur textuelle est saisie, dans l'application apparaîtra l'option d'effacer la date en cliquant sur le RadioButton dont le contenu est défini

dans cette propriété.

IsTextHidden : bool – permet de masquer l'affichage de la date sélectionnée

SelectedDate : DateTime? – date sélectionnée

DatePicker2

Un moyen alternatif de sélectionner la date est le contrôle DatePicker2. Il affiche la date sous forme de trois champs séparés. Le champ du jour (TextBox), du mois (ComboBox) et de l'année (TextBox). Les champs du contrôle peuvent être gérés indépendamment. Il est également possible de modifier l'ordre d'affichage des composants de la date dans le panneau de gestion des vues, grâce au fait que les composants sont intégrés dans le conteneur.

Propriétés sélectionnées :

SelectedDate : DateTime? – date affiché dans le contrôle

IsMonthsListEditable : bool – contrôle les modifications du combobox avec la liste des mois

IsValid : bool – drapeau définissant si la date saisie est correcte

Pour plus d'informations, voir le chapitre Exemple d'utilisation du contrôle DatePicker2 dans l'article Exemples.

AssistantControl

Ce contrôle est un contrôle métier spécialisé utilisé dans chaque pièce justificative et permettant de définir un

assistant de transaction dans la pièce, ainsi que dans la position. Son apparence et son fonctionnement ressemblent beaucoup au contrôle ComboBox2.

Propriétés sélectionnées :

Source : AssistantSource – source de données pour le contrôle, binding à l'instance AssistantSource

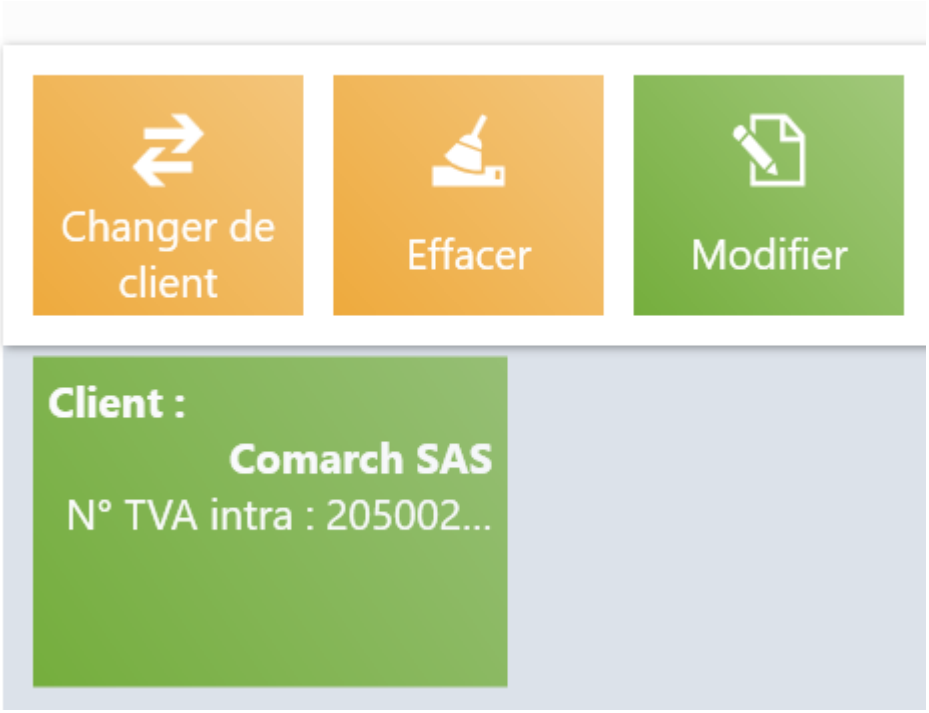
Label : string – texte affiché dans le contrôle (par exemple sur une pièce justificative : « Vendeur »).

AttributeControl

C'est un autre contrôle métier spécialisé qui prend en charge les attributs des objets métiers. Il permet de présenter les attributs et de les modifier en fonction de leur type, dans les contrôles de modification indépendants. Par exemple, un attribut du type textuel sera automatiquement affiché sous forme de contrôle TextBox et l'attribut logique sous forme de SwitchBox. Pour plus d'informations sur la gestion des attributs métiers, voir l'article Gestion des attributs

BuyerControl et RecipientControl

Ces contrôles sont des contrôles métiers à sélectionner et afficher le client et le fournisseur dans la pièce justificative. Leur structure ressemble à la structure des deux contrôles précédents. Ils sont composés des contrôles et d'une source de données joints à l'aide de binding.



Propriétés sélectionnées de BuyerControl :

Source : IBuyerSource – source de données implémentant l'interface IBuyerSource

---

Afficher un ensemble de données à l'aide du contrôle DataGrid

Le contrôle **Comarch.POS.Presentation.Core.Controls.DataGrid** est basé sur celui connu du .NET et permet d'afficher une collection de données divisée en colonnes et lignes. Le contrôle DataGrid dans POS permet en outre de télécharger les données de manière asynchrone, il prend en charge la pagination de liste (chargement des données lors du défilement de liste), modifie la façon d'effectuer le tri des données, ainsi que le mécanisme de groupement en l'enrichissant

d’agrégation des données.

Télécharger les données de manière asynchrone

Pour que les données soient téléchargées de manière asynchrone, il faut binder leur source avec la propriété **AsyncItemsSource** dans le DataGrid. La source de données doit être un objet de la classe **AsyncDataGridCollection<T>** où T est le type de données qui sera affiché dans une seule ligne. La logique responsable d’acquisition de données doit être définie dans le premier paramètre du constructeur de la classe AsyncDataGridCollection. Les données téléchargées doivent à leur tour être attribuées à la collection **Data.**

Exemple :

```
Documents = new AsyncDataGridCollection<TradeDocumentListRow>(
    (token, o) =>
    {
                    //téléchargement de données
        var documents = GetDocumentsAsync(token, o);

                     //À un moment donné, seul un fil peut
modifier une collection
        lock (Documents.DataLock)
        {
                            //pris en charge d'annulation de
téléchargement
            if (token.IsCancellationRequested)
            return null;

                             //chaque rafraîchissement de la
liste exige qu'elle soit effacée
            Documents.Data.Clear();
                                //ajout des données à la
collection qui sera lue par DataGrid
            Documents.Data.AddRange(data);
        }
```

```
    return documents;
    }, OnReceiptsOperationCompleted, loggingService);
```

L’exemple au dessus présente la définition de la propriété Documents implémentée dans le constructeur de viewmodel qui est bindé avec la propriété AsyncItemsSource du contrôle DataGrid. Le premier argument du constructeur est une logique simple de téléchargement de données (sans prendre en compte la pagination). La méthode sera automatiquement appelée de manière asynchrone directement par le contrôle de DataGrid, dès que celui-ci est initialisé. Pour décider manuellement à quel moment les données seront téléchargées, il faut modifier la valeur de la propriété **LoadDataOnDataGridInitialization=false** dans l’objet Documents. Au moment où l’on voudra télécharger les données, il faudra appeler la méthode **Documents.Refresh(false)**. Le deuxième argument du constructeur est la méthode qui sera appelée dans le fil principal (UI) juste après l’achèvement de la logique de téléchargement. Le troisième paramètre est une instance du service d’enregistrement d’erreurs ILoggingService, grâce à laquelle les exceptions potentielles retrouvées dans le fil de téléchargement seront enregistrées dans les journaux système de l’application.

Tri de données

La pagination peut être défini par défaut et modifier par l’utilisateur à tout moment en cliquant la colonne sélectionnée. Les paramètres par défaut peuvent être en revanche modifiés dans le panneau de gestion des vues. Le contrôle n’exécute pas la tâche lui-même, mais la délègue à la source de données, en lançant une méthode de récupération qui envoie uniquement des informations sur la méthode de tri souhaitée. Les informations comment trier les données téléchargées peuvent être lues dans la propriété **Sorting** dans

l’objet Documents.

Sorting : List<GridSortDescription> – liste contentant l’information selon quelle colonne les données ont été triées et de quelle façon.

Composants de la classe GridSortDescription :

PropertyName : string – nom de colonne défini dans la propriété SortMemberPath de colonne. Si le nom n’est pas défini, le nom par défaut utilisé est le nom de la propriété utilisée dans le binding.

Direction : ListSortDirection – sens de tri, peut être croissant ou décroissant.

Column : DataGridColumn – référence à la colonne

Pagination de liste

La pagination, ou en fait la lecture dynamique des données pendant le défilement de la liste, est basée sur le fait que le contrôle DataGrid, lorsque l’utilisateur approche de la fin de la liste, initie une méthode asynchrone qui va chercher les données, en informant la source qu’une autre portion sera nécessaire si elle existe. Ce mécanisme permet une prise en charge effective de grands ensembles de données, où le téléchargement de l’ensemble entier ne serait ni effectif ni efficace.

La logique de la pagination doit être implémentée dans la source. Pour savoir quelle portion de données est requis par le contrôle, il faut utiliser les propriétés **ItemsToTake** et **ItemsToSkip** qui sont emplacées dans l’objet de la classe AsyncDataGridCollection<T>.

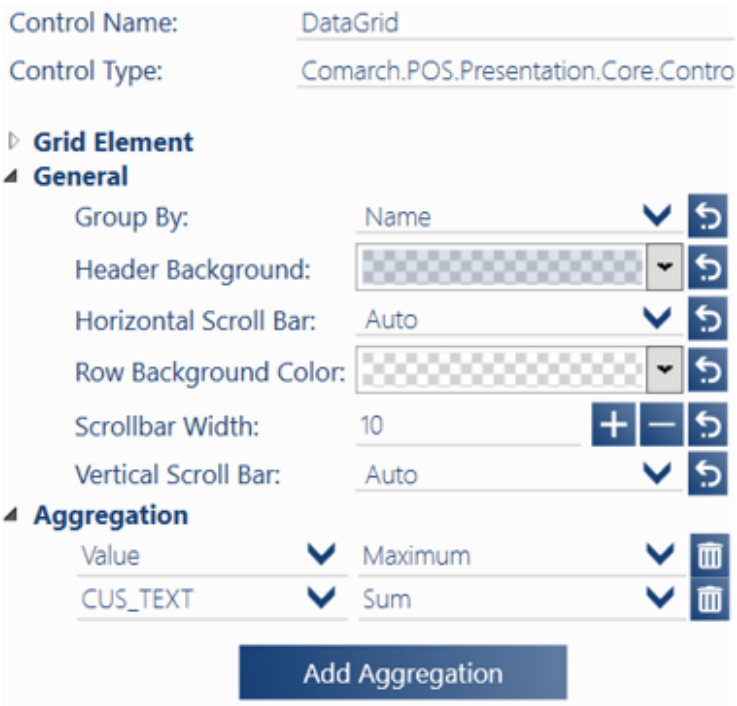
temsToTake : int – nombre de lignes des données requis par le contrôle.

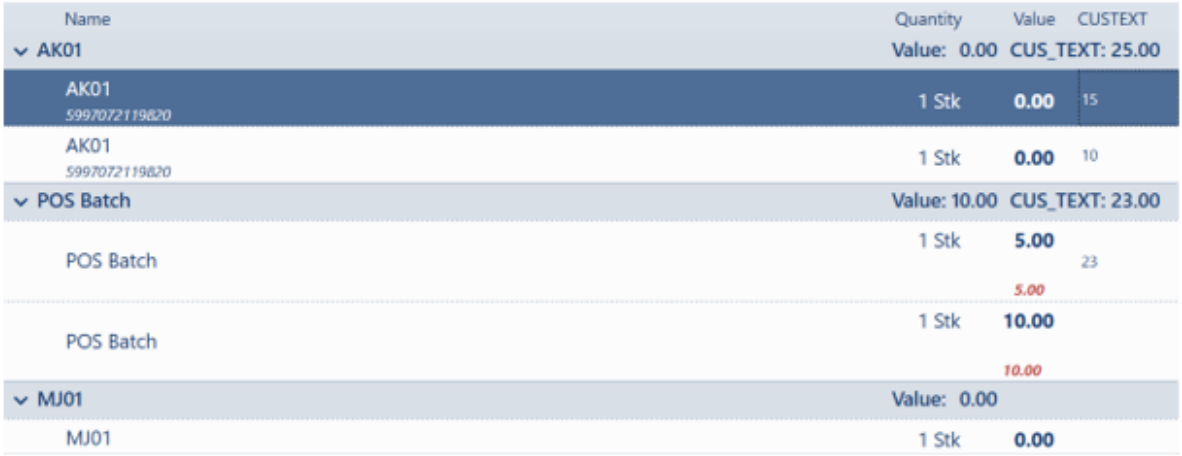
ItemsToSkip : int – nombre de lignes des données qui doivent être omis en comptant à partir du début d'ensemble (ce nombre représente le nombre de lignes qui ont déjà été téléchargées).

Groupement et agrégation

Le contrôle DataGrid prend en charge le mécanisme de groupement et d'agrégation des valeurs des entités de la collection présentée par le contrôle. Activer la fonction du groupement désactive le mécanisme de la pagination de données. Afin d'activer le groupement, il faut définir sur le contrôle la propriété **IsGroupingEnabled="True"**. Le groupement est entièrement gérable dans la vue de gestion UI et il permet de grouper et agréger selon les critères définis auparavant. Pour qu'une propriété puisse être groupée, elle doit être marquée avec l'attribut **[AllowGroupBy]** (de l'espace nom : Comarch.POS.Core). Facultativement, l'attribut permet de définir le chemin du fichier ressource et le postfixe de clé avec traduction. L'attribut par défaut est **Properties.Resources** et chaque clé est le nom de la propriété plus le postfixe **Header.** Par exemple, la clé pour la propriété Name sera NameHeader.

L'agrégation est possible uniquement selon toutes les propriétés portant la marque de l'attribut AllowGroupBy. Les moyens d'agrégation suivants sont disponibles dans la version standard de l'application : **Somme, Moyen, Maximum et Minimum.** Ces moyens fonctionnent uniquement sur les propriétés du type numérique ou string qui est converti en type numérique.





Extensions d’agrégation

Afin d’ajouter un moyen d’agrégation personnalisé, il faut créer une nouvelle classe d’implémentation **IAggregationType** (espace nom : Comarch.WPF.Controls.Aggregation). Il faut implémenter dans cette classe le mécanisme d’agrégation dans la propriété Function.

Supposons que la nouvelle agrégation personnalisée soit réalisée uniquement sur les chiffres, au lieu d’implémenter l’interface il est possible d’hériter de la classe **AggregationNumberType.** Ensuite, la méthode **Aggregate** peut être surchargée sans se soucier de la vérification des types des

données agrégées.

À la fin, la classe créée doit être enregistrée dans le système comme le nouveau moyen d'agrégation. Dans la classe module il faut appeler la méthode **RegisterDataGridGroupAggregation**, transmettre le nom de la classe et définir la clé et le ressource à partir duquel seront téléchargées les traductions pour le nouveau moyen d'agrégation.

Pour plus d'informations, voir le chapitre Exemple d'utilisation du moyen d'agrégation de données personnalisé dans DataGrid dans l'article Exemples.

Groupement des attributs

Le groupement et l'agrégation selon les attributs fonctionnent automatiquement et n'exigent pas d'implémentation supplémentaire. Il est important que l'entité d'élément contenant les attributs possède les propriétés :

- **Attributes** du type répértoire Dictionary<string,AttributeEntity>
- **AttributeRows** du type AttributesDictionary.

Schémas de couleur et police

L'interface de l'application POS a été créée de la façon qui permet de garder la cohérence de la mise en page de toutes les vues et la facilité de gestion. C'est pourquoi nous avons définis les noms des couleurs qui peuvent être utilisées lors de la création/modification d'une vue.

Les noms des couleurs définis sont divisés en deux groupes.

Les couleurs gérées directement par l'utilisateur dans l'interface POS et les couleurs gérées indirectement, qui ne peuvent pas être modifiées directement par l'utilisateur, mais qui sont définies automatiquement à la base des couleurs gérées directement.

Couleurs gérées directement

- **ThemeColor** – couleur de base du motif, utilisée comme :
  - couleur du texte dans les vues de base et modales,
  - couleur de l'arrière-plan dans la vue de message,
  - couleur de l'arrière-plan des boutons dans la vue de message,
  - couleur de l'arrière-plan du bouton Rechercher/Ajouter par exemple dans la liste des ventes, des reçus, des produits,
  - couleur de l'arrière-plan de la position marquée dans le contrôle ComboBox,
  - couleur de l'arrière-plan des notifications,
  - couleur de rediobutton et de checkbox sélectionné/marqué
- **ThemeBackground** – couleur de base du motif, utilisée comme :
  - couleur de l'arrière-plan de la vue de base et modale,
  - couleur du texte et des icônes vectorielles des boutons (sans menu latéral),
  - couleur du texte et des icônes vectorielles dans la vue de message,
  - couleur du texte et de l'encadrement des boutons dans la vue de message,
  - couleur du texte dans le bouton Rechercher/Ajouter,
  - couleur du texte et de l'icône vectorielle de la notification,
  - couleur du texte de radiobutton et de checkbox sélectionné/marqué

Couleurs gérées indirectement

- **ThemeColor2** – couleur de base (ThemeColor) avec la transparence au niveau de 85%, utilisée comme :
    - couleur de l’en-tête des listes (DataGrid)
- **ThemeColor3** – couleur de base (ThemeColor) avec la transparence au niveau de 25%, utilisée comme :
    - couleur de sélection d’une position dans la liste
- **ThemeColor4** – couleur de base (ThemeColor) avec la transparence au niveau de 65%, utilisée comme :
    - couleur de sélection d’une ligne dans la liste après le survol de la souris,
    - couleur de sélection de l’encadrement d’un bouton composé uniquement d’une icône vectorielle (HeaderButton) après le survol de la souris,
    - couleur de sélection d’une position dans le menu latéral après le survol de la souris,
- **HintColor** – couleur de l’indice dans le contrôle TextBox
- **Background** – couleur fixe avec une transparence définie – #1ADFE5E5, utilisée comme :
    - couleur de l’arrière-plan d’une liste

Utilisation des couleurs dans le code

Dans tout extrait de code xaml, les couleurs sont utilisées via la commande DynamicResource, par exemple :

```
<TextBox Foreground="{DynamicResource ThemeColor}" />
```

Dans la plupart des cas, l’exemple ci-dessus n’a aucune raison d’être, car les paramètres prédéfinis rendent une configuration supplémentaire de la propriété Foreground redondante.

Police

La police a été également définie pour l'application entière et elle est gérée au niveau de l'interface d'utilisateur POS. Pour la plupart des contrôles, il n'est pas nécessaire de la définir en plus. Une exception serait par exemple la définition des colonnes de liste où il ne faut pas oublier de définir la police de la façon suivante :

```
<DataGridTextColumn FontFamily="{DynamicResource FontFamily}" />
```

Gestion de vue et de ses éléments

Les éléments d'interface (les contrôles) peuvent être modifiés en chageant les couleurs gérées directement – la couleur du motif (ThemeColor) et la couleur d'arrière-plan (ThemeBackground), dans le panneau **Configuration d'interface**, globalement, tous les contrôles à la fois, dans le panneau **Éléments globaux** ou localement, indépendamment par vue, dans le panneau **Gestion des vues.** La modification d'un n'importe quel paramètre de contrôle localement a une priorité plus haute qu'une modification globale.

Layout.Id

Le fonctionnement de la gestion des vues dépend de quelques facteurs. Le premier est d'enregistrer correctement la vue (utilisation de la méthode **RegisterViews** de la classe **ModuleBase**) et de créer un view-model approprié pour le mode design (DesignViewModel), le deuxième est de marquer les

contrôles avec un attribut approprié.

Cet attribut est **Layout.Id** qui doit être unique dans l'ensemble de l'application POS. C'est une condition indispensable pour que le contrôle soit géré indépendamment. Si plusieurs contrôles ont le même Id, alors la modification d'une propriété entraînera la modification des autres contrôles.

Définir les valeurs par défaut des propriétés gérables

Les valeurs par défaut des contrôles peuvent être définies de plusieurs façons, en fonction des besoins.

1. Directement comme les attributs du contrôle défini dans xaml

<TextBlock Foreground="Red"/>

Il ne sera pas possible de gérer le contrôle ci-dessus, car l'attribut **Layout.Id** n'a pas été défini pour lui.

Cette méthode peut être également utilisée avec les Schémas de couleur et police :

<TextBlock Foreground="{DynamicResource ThemeColor}"/>

Le contrôle sera géré uniquement en modifiant le motif (la couleur de la police sera modifiée).

Pour qu'un contrôle soit gérable globalement dans la configuration globale d'interface, il faut l'enregistrer à l'aide de la méthode **RegisterControl** de la classe **ModuleBase.**

Pour qu'un contrôle soit gérable localement par vue où elle a été utilisée (gestion des vues), cette vue doit être enregistrée correctement (à l'aide de la méthode **RegisterViews**) et le contrôle doit avoir l'attribut **Layout.Id.**

En cas d’utilisation de Layout.Id, il ne faut pas paramétrer les propriétés par défaut du contrôle directement sur lui. Si par exemple nous définissons la valeur par défaut de la propriété Foreground avant de définir Layout.Id, cette propriété sera ignorée. En revanche, si nous définissons la propriété Foreground après Layout.Id, elle ne sera pas gérable (sa valeur sera toujours remplacée par la valeur définie directement).

Une définition directe des attributs doit être effectuée uniquement pour les propriétés non-gérables comme celles qui sont indispensables au fonctionnement correct du contrôle (par exemple Binding).

2. Style global ou local de contrôle (en utilisant x:Key)

```
<core:View.Resources>
        <Style TargetType="TextBlock" BasedOn="{StaticResource
{x:Type TextBlock}}">
            <Setter Property="Foreground" Value="Red"/>
                        <Setter  Property="Background"
Value="{DynamicResource ThemeBackground}"/>
        </Style>
</core:View.Resources>
..
<TextBlock core:Layout.Id="TextBlockLayoutId" />
```

L’exemple ci-dessus illustre le réglage global du style de couleur de la police et de la couleur d’arrière-plan pour tous les contrôles TextBlock utilisés dans la vue <core:View> actuelle.

```
<core:View.Resources>
        <Style x:Key="TextBlockStyle"
               TargetType="TextBlock" BasedOn="{StaticResource
{x:Type TextBlock}}">
            <Setter Property="Foreground" Value="Red"/>
                        <Setter  Property="Background"
Value="{DynamicResource ThemeBackground}"/>
        </Style>
</core:View.Resources>
```

…
<TextBlock Style="{StaticResource TextBlockStyle}" core:Layout.Id="TextBlockLayoutId" />

La modification ci-dessus présente comment définir à l'aide des styles uniquement le contrôle sélectionné.

3. Attribut par défaut défini dans ModerUI.xaml selon la définition <type_d_attribut x:Key="[LayoutId].Default.[Nom_du_type]">[valeur par défaut]</type_d_attribut>

Fichier *ModernUI.xml*

<SolidColorBrush x:Key="TextBlockLayoutId.Default.Foreground" Color="Red"/>

Fichier de vue

<TextBlock core:Layout.Id="TextBlockLayoutId" />

Pour que l'exemple ci-dessus fonctionne correctement, il est requis d'enregistrer les ressources ModernUI.xaml du module dans la classe enregistrant le module (voir le chapitre Nouveau module dans l'article Créer des vues). Dans le constructeur nous appelons :

LayoutService.RegisterResources(typeof(Module));

Les limites de cette approche lient dans l'impossibilité de définir les valeurs dynamiques et de grouper les valeurs qui se répétent sous un nom commun. Par exemple, l'utilisation suivante du code est incorrecte :

<SolidColorBrush x:Key="TextBlockLayoutId.Default.Foreground"
Color="{DynamicResource ThemeColor}"/>

Dans ce cas le type Color (System.Windows.Media.Color) et ThemeColor(SolidColorBrush) ne correspondent pas !

Il n'est pas recommandé d'utiliser cette méthode. Une exception est la définition des paramètres par défaut des

colonnes de DataGrid.

Liste des propriétés prises en charge

Les tableaux ci-dessous contiennent un catalogue des propriétés qui sont gérées dans le panneau de configuration d'interface pour les contrôles individuels disponibles dans POS.

Propriété	Nom
Propriété prise en charge [nom:type]	Nom dans le panneau de gestion

Framework Element

System.Windows.**FrameworkElement**

Propriété	Nom
Width : double	Largeur
Height : double	Hauteur
Margin : Thickness	Marge
HorizontalAlignment : HorizontalAlignment	Alignement horizontal
VerticalAlignment : VerticalAlignment	Alignement vertical
MaxWidth : double	Largeur maximale
MaxHeight : double	Hauteur maximale
Grid.Position : string	Emplacement

Control

System.Windows.Controls.**Control**

Propriété	Nom
Background : Brush	Arrière-plan
Foreground : Brush	Couleur du texte
FontSize : double	Taille de la police
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police
Padding : Thickness	Remplissage
Grid.Position : string	Emplacement

Grid

System.Windows.Controls.**Grid**

Propriété	Nom
Background : Brush	Arrière-plan
Margin : Thickness	Marge
Visibility: Visibility	Visibilité
Width : double	Largeur
Height : double	Hauteur
Grid.Position : string	Position

Comarch.POS.Presentation.Core.Controls.**Grid**

: System.Windows.Controls.Grid

Propriété	Nom

ColumnDefinition : string	Colonnes
RowDefinition : string	Lignes

Border

System.Windows.Controls.**Border**

Propriété	Nom
Visibility : Visibility	Visibilité
Background : Brush	Arrière-plan

ScrollViewer

System.Windows.Controls.**ScrollViewer**

: System.Windows.Controls.Control

Propriété	Nom
VerticalScrollBarVisibility : ScrollBarVisibility	Barre de défilement vertical
HorizontalSchrollBarVisibility : ScrollBarVisibility	Barre de défilement horizontal
Width: double	Largeur
Height : double	Hauteur

Comarch.POS.Presentation.Core.Controls.**ScrollViewer**

: System.Windows.Controls.ScrollViewer

Propriété	Nom

Separator

System.Windows.Controls.**Separator**

: System.Windows.FrameworkElement

Propriété	Nom
Background : Brush	Arrière-plan

TextBlock

System.Windows.Controls.**TextBlock**

: System.Windows.FrameworkElement

Propriété	Nom
Background : Brush	Arrière-plan
Foreground : Brush	Couleur du texte
FontSize : double	Taille de la police
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police
Padding : Thickness	Remplissage
TextAlignment : TextAlignment	Alignement du texte
Visibility : Visibility	Visibilité
TextWrapping : TextWrapping	Renvoi à la ligne

DoubleColorTextBlock

Comarch.POS.Presentation.Core.Controls.**DoubleColorTextBlock**

: System.Windows.FrameworkElement

Propriété	Nom
FirstForeground : Brush	Couleur du texte 1
SecondForeground : Brush	Couleur du texte 2
Background : Brush	Arrière-plan
FontSize : double	Taille de la police
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police
Padding : Thickness	Remplissage
Visibility : Visibility	Visibilité

ErrorTextBlock

Comarch.POS.Presentation.Core.Controls.**ErrorTextBlock**

: System.Windows.Controls.TextBlock

Propriété	Nom

TextBox

Comarch.WPF.Controls.**TextBox**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	Nom
BorderThickness : Thickness	Encadrement
BorderBrush : Brush	Couleur de l’encadrement

FocusedBorderBrush : Brush	Couleur de l’encadrement (focus)
ErrorColor : Brush	Couleur d’erreur
HintForeground : Brush	Couleur d’indice
Hint: string	Indice

Comarch.POS.Presentation.Core.Controls.**TextBox**

: Comarch.WPF.Controls.TextBox

Propriété	**Nom**

Underline

Comarch.WPF.Controls.**Underline**

: System.Windows.FrameworkElement

Propriété	**Nom**
Stroke : Brush	Couleur
StrokeThickness : Thickness	Épaisseur

Comarch.POS.Presentation.Core.Controls.**Underline**

: Comarch.WPF.Controls.Underline

Propriété	**Nom**

ColumnDefinition

System.Windows.Controls.**ColumnDefinition**

Propriété	Nom
Width : double	Largeur

RowDefinition

System.Windows.Controls.**RowDefinition**

Propriété	Nom
Height : double	Hauteur

DataGridColumn

System.Windows.Controls.**DataGridColumn**

Propriété	Nom
MaxWidth : double	Largeur maximale
MinWidth : double	Hauteur maximale
Width : DataGridLength	Largeur
SortDirection : ListSortDirection	Trier
Visibility : Visibility	Visibilité
HorizontalAlignment : HorizontalAlignment	Alignement horizontal

DataGridTemplateColumn

System.Windows.Controls.**DataGridTemplateColumn**

: System.Windows.Controls.DataGridColumn

Propriété	Nom

DataGridTextColumn

System.Windows.Controls.**DataGridTextColumn**

: System.Windows.Controls.DataGridColumn

Propriété	Nom
FontSize : double	Taille de la police
Foreground : Brush	Couleur du texte
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police

DataGridCell

System.Windows.Controls.**DataGridCell**

Propriété	Nom
Margin : Thickness	Marge
HorizontalAlignment : HorizontalAlignment	Alignement horizontal

DataGridColumnHeader

System.Windows.Controls.Primitives.**DataGridColumnHeader**

Propriété	Nom

Margin : Thickness	Marge
HorizontalContentAlignment : HorizontalAlignment	Alignement horizontal du contenu

DataGrid

Comarch.WPF.Controls.**DataGrid**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	**Nom**

Comarch.POS.Presentation.Core.Controls.**DataGrid**

Propriété	**Nom**
RowBackground : Brush	Arrière-plan de ligne
HeaderBackground : Brush	Arrière-plan de l’en-tête
ScrollBarWidth : double	Largeur de la barre de défilement
VerticalScrollBarVisibility : Visibility	Barre de défilement vertical
HorizontalScrollBarVisibility : Visibility	Barre de défilement horizontal
GroupBy : DataGridGroup	Trier par
IsVirtualizingWhenGrouping : bool	Virtualiser lignes
ScrollUnit : ScrollUnit	Unité de glissière
Grid.Position : string	Position

ItemsContainer

Comarch.POS.Presentation.Core.Controls.**ItemsContainer**

: System.Windows.FrameworkElement

Propriété	Nom
Orientation : Orientation	Orientation
Padding : Thickness	Remplissage
Background : Brush	Arrière-plan
MoreMultiButtonHeight : double	Hauteur
MoreMultiButtonWidth : double	Largeur
MoreMultiButtonImageHeight : double	Hauteur d’icône
MoreMultiButtonImageWidth : double	Largeur d’icône
MoreMultiButtonMargin : Thickness	Marge
MoreMultiButtonImageMargin : Thickness	Marge d’icône
MoreMultiButtonFontSize : double	Taille de la police
MoreMultiButtonIsImageVisible : bool	Afficher icône

Expander

Comarch.POS.Presentation.Core.Controls.**Expander**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	Nom
HeaderBackground : Brush	Arrière-plan de l’en-tête
HeaderPadding : Thickness	Remplissage de l’en-tête

Image

System.Windows.Controls.**Image**

: System.Windows.FrameworkElement

Propriété	Nom
Stretch : Stretch	Étendre
StretchDirection : StretchDirection	Sens d’étirement

Comarch.WPF.Controls.**Image**

Propriété	Nom
HorizontalAlignment : HorizontalAlignment	Alignement horizontal
HorizontalContentAlignment : HorizontalAlignment	Alignement horizontal du contenu
VerticalAlignment : VerticalAlignment	Alignement vertical
VerticalContentAlignment : VerticalAlignment	Alignement vertical du contenu

Comarch.POS.Presentation.Core.Controls.**Image**

: System.Windows.FrameworkElement

Propriété	Nom
DefaultImageKey : ImageKey	Icône par défaut

BundleImage

Comarch.POS.Presentation.Core.Controls.**BundleImage**

Propriété	Nom
IconForeground : Brush	Couleur
IconMargin : Thickness	Marge
IconImageKey : ImageKey	Icône
Width : double	Largeur
Height : double	Hauteur
PopupMaxWidth : double	Hauteur maximale
PopupMinWidth : double	Hauteur minimale

Button

Comarch.POS.Presentation.Core.Controls.**Button**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Autres sur la base de Button, par exemple :

Comarch.POS.Presentation.Core.Controls.AcceptButton

Comarch.POS.Presentation.Core.Controls.CancelButton

Comarch.POS.Presentation.Core.Controls.SelectButton

Comarch.POS.Presentation.Core.Controls.CleanButton

Comarch.POS.Presentation.Core.Controls.TileButton

Comarch.POS.Presentation.Core.Controls.PaymentTypeTile

Comarch.POS.Presentation.Core.Controls.PrintLabelButton

Comarch.POS.Presentation.Core.Controls.ShowItemsVariantsButton

Propriété	**Nom**
ImageKey : ImageKey	Icône
ImageMargin : Thickness	Marge d’icône
ImageWidth : double	Largeur d’icône
ImageHeight : double	Hauteur d’icône
ImageHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ImageVerticalAlignment : VerticalAlignment	Alignement vertical
IsImageVisible : bool	Afficher l’icône
ShortcutMargin : Thickness	Marge
ShortcutWidth : double	Largeur
ShortcutHeight : double	Hauteur
ShortcutHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ShortcutVerticalAlignment : VerticalAlignment	Alignement vertical
IsShortcutVisible : bool	Afficher le raccourci
Shortcut : Shortcut	Raccourci clavier

IsScaledShortcut : bool	Mise à l’échelle du contenu
ContentMargin : Thickness	Marge
ContentWidth : double	Largeur
ContentHeight : double	Hauteur
ContentHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ContentVerticalAlignment : VerticalAligment	Alignement vertical
ContentVisibility : Visibility	Visibilité
IsScaledContent : bool	Mise à l’échelle du contenu
Orientation : Orientation	Orientation
ItemsContainer.NoWrapButton : bool	Ne pas agréger

RadioButton

Comarch.POS.Presentation.Core.Controls.**RadioButton**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	**Nom**
CheckedStateBackground : Brush	Arrière-plan d’élément sélectionné
CheckedStateForeground : Brush	Texte d’élément sélectionné
ImageKey : ImageKey	Icône
ImageMargin : Thickness	Marge d’icône

ImageWidth : double	Largeur d’icône
ImageHeight : double	Hauteur d’icône
ImageHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ImageVerticalAlignment : VerticalAlignment	Alignement vertical
IsImageVisible : Visibility	Afficher l’icône
ContentMargin : Thickness	Marge
ContentWidth : double	Largeur
ContentHeight : double	Hauteur
ContentHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ContentVerticalAlignment : VerticalAlignment	Alignement vertical
ContentVisibility : Visibility	Visibilité
IsScaledContent : bool	Mise à l’échelle du contenu
Orientation : Orientation	Orientation

FieldControl

Comarch.POS.Presentation.Core.Controls.**FieldControl**

: System.Windows.FrameworkElement

Propriété	**Nom**
Orientation : Orientation	Orientation
LabelMargin : Thickness	Marge
LabelWidth : double	Largeur

LabelHeight : double	Hauteur
LabelFontSize : double	Taille de la police
LabelFontStyle : FontStyle	Style de la police
LabelFontWeight : FontWeight	Poids de la police
LabelForeground : Brush	Couleur du texte
LabelForegroundError : Brush	Couleur en absence de validation
LabelHorizontalAlignment: HorizontalAlignment	Alignement horizontal
LabelVertivalAlignment: VerticalAlignment	Alignement vertical
ContentIsRequired : bool	Requis

CheckBox

Comarch.POS.Presentation.Core.Controls.**CheckBox**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	**Nom**
CheckedStateBackground : Brush	Arrière-plan d’élément sélectionné
CheckedStateForeground : Brush	Texte d’élément sélectionné
DisabledStateBackground : Brush	Arrière-plan d’élément inactif
DisabledCheckedStateBackground : Brush	Arrière-plan d’élément inactif sélectionné
ImageKey : ImageKey	Icône

ImageMargin : Thickness	Marge d’icône
ImageWidth : double	Largeur d’icône
ImageHeight : double	Hauteur d’icône
ImageHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ImageVerticalAlignment : VerticalAlignment	Alignement vertical
IsImageVisible : Visibility	Afficher l’icône
ContentMargin : Thickness	Marge
ContentWidth : double	Largeur
ContentHeight : double	Hauteur
ContentHorizontalAlignment : HorizontalAlignment	Alignement horizontal
ContentVerticalAlignment : VerticalAlignment	Alignement vertical
ContentVisibility : Visibility	Visibilité
IsScaledContent : bool	Mise à l’échelle du contenu
Orientation : Orientation	Orientation

ComboBox

Comarch.WPF.Controls.**ComboBox**

: System.Windows.FrameworkElement,

System.Windows.Controls.Control

Propriété	**Nom**
HorizontalContentAlignment : HorizontalAlignment	Alignement horizontal du contenu
PopupBackground : Brush	Arrière-plan

FocusedBorderBrush : Brush	Couleur de l’encadrement (focus)
ErrorColor : Brush	Couleur d’erreur

Comarch.POS.Presentation.Core.Controls.**ComboBox**

: Comarch.WPF.Controls.TextBox

Propriété	Nom

ComboBox2

Comarch.POS.Presentation.Core.Controls. **ComboBox2**

Propriété	Nom
LabelFontSize : double	Taille de la police
LabelFontStyle : FontStyle	Style de la police
LabelFontWeight : FontWeight	Poids de la police
Shortcut : Shortcut	Raccourci clavier
FontSize : double	Taille de la police
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police
Visibility : Visibility	Visibilité

AutoCompleteComboBox

Comarch.POS.Presentation.Core.Controls.**AutoCompleteComboBox**

: System.Windows.Controls.Control

Propriété	Nom
HintForeground : Brush	Couleur d’indice
Hint: string	Indice
FocusedBorderBrush : Brush	Couleur de l’encadrement (focus)
ErrorColor : Brush	Couleur d’erreur

SwitchBox

Comarch.POS.Presentation.Core.Controls.**SwitchBox**

: System.Windows.Controls.Control

Propriété	Nom
VerticalContentAlignment : VerticalAlignment	Alignement vertical du contenu
Margin : Thickness	Marge

SearchBox

Comarch.POS.Presentation.Core.Controls.**SearchBox**

Propriété	Nom
FontSize : double	Taille de la police
Foreground : Brush	Arrière-plan
HintForeground : Brush	Couleur d’indice
Margin : Thickness	Marge
Hint: string	Indice

DatePicker

Comarch.POS.Presentation.Core.Controls.**DatePicker**

: System.Windows.Controls.Control

Propriété	Nom
FocusedBorderBrush : Brush	Couleur de l’encadrement (focus)
ErrorColor : Brush	Couleur d’erreur
BorderBrush : Brush	Couleur de l’encadrement
Visibility : Visibility	Visibilité

ButtonSpinner

Comarch.POS.Presentation.Core.Controls.**ButtonSpinner**

Propriété	Nom
ButtonImageWidth : double	Largeur
ButtonImageHeight : double	Hauteur
ButtonWidth : double	Largeur du bouton
ButtonHeight : double	Hauteur du bouton

FilterItemsControl

Comarch.POS.Presentation.Core.Controls.**FilterItemsControl**

Propriété	Nom
MaxFilterItemsPerRow : int	Nombre de filtres maximal
Visibility : Visibility	**Visibilité**

Grid.Position : string	Position

SearchBoxFilter

Comarch.POS.Presentation.Core.Controls.**SearchBoxFilter**

: Comarch.POS.Presentation.Core.Controls.ComboBox2

Propriété	Nom
DefaultFilter : string	Valeur de filtre par défaut

StockTile

Comarch.POS.Presentation.Core.Controls.**StockTile**

Propriété	Nom
Background : Brush	Arrière-plan
IsCodeVisible : bool	Afficher le code d’entrepôt
WarehouseMargin : Thickness	Marge
WarehouseCodeFontSize : double	Taille de la police du code
WarehouseNameFontSize : double	Taille de la police de la désignation
StocksFontSize : double	Taille de la police
StocksMargin : Thickness	Marge

SetValueNumbersKeyboard

Comarch.WPF.Controls.**SetValueNumbersKeyboard**

Propriété	Nom
Visibility : Visibility	Visibilité

AttributeControl

Comarch.POS.Presentation.Core.Controls.**AttributeControl**

: System.Windows.Controls.Control

Propriété	Nom

AssistantControl

Comarch.POS.Presentation.Core.Controls.**AssistantControl**

Propriété	Nom
LabelFontSize : double	Taille de la police
LabelFontStyle : FontStyle	Style de la police
LabelFontWeight : FontWeight	Poids de la police
FontSize : double	Taille de la police
FontWeight : FontWeight	Poids de la police
FontStyle : FontStyle	Style de la police
Shortcut : Shortcut	Raccourci clavier
Visibility : Visibility	Visibilité
Grid.Position : string	Position

DocumentKeypad

Comarch.POS.Presentation.Core.Controls.**DocumentKeypad**

Propriété	Nom
HeaderFontSize : double	
KeypadHeaderColor : Brush	

Créer des vues gérables

Dans l'article Créer des vues nous avons décrit les étapes de base indispensables pour créer le squelette d'une nouvelle vue. Nous avons également montré comment enregistrer une telle vue, encore vide, pour qu'elle soit gérable lors du fonctionnement de l'application (chapitre Enregistrer vues à naviguer et à gérer l'affichage dans l'article Créer de vues).

Chaque élément (contrôle) marqué avec l'identifiant unique LayoutId devient automatiquement gérable. Leur gestion c'est dans la plupart des cas une possibilité de manipuler leurs certaines propriétés comme la couleur d'arrière-plan, du texte, le formatage de la police, les marges, la largeur, la hauteur etc. Tout cela a été également décrit dans le chapitre précédent.

Les éléments gérables reçoivent une caractéristique supplémentaire au moment où ils sont déclarés à l'intérieur d'un des conteneurs disponibles dans l'application POS : **Grid** et **ItemsContainer** (de l'espace Comarch.POS.Presentation.Core.Controls). Cette caractéristique permet de déterminer si et où un élément doit être placé dans un conteneur.

Gestion des éléments dans le conteneur ItemsContainer

L’ajout de contrôles à l’intérieur du conteneur ItemsContainer entraîne leur présentation par défaut dans l’ordre où ils ont été déclarés. La particularité du conteneur fait que les éléments sont positionnés par défaut horizontalement ou verticalement, en fonction de la propriété **Orientation** (qui peut être également gérable) définie sur le conteneur. Si le conteneur possède le LayoutId unique, il devient gérable ce qui permet à l’utilisateur de manipuler ses éléments (tous les éléments doivent également posséder ses identifiants LayoutId uniques). Lorsque l’utilisateur active le mode de gestion de la vue contenant ce conteneur, il pourra, à l’aide de la souris, attraper n’importe quel contrôle à son intérieur et le faire glisser en modifiant l’ordre des éléments intérieurs du conteneur ou même supprimer l’élément du conteneur ou l’ajouter à nouveau.

Les éléments déclarés à l’intérieur du conteneur ItemsContainer ne peuvent exister que dans ce conteneur. Cela veut dire que l’utilisateur ne peut pas ajouter le contrôle à un autre conteneur. Cette restriction est enlevée par un conteneur du type Grid.

Exemple :

```
<StackPanel>
                                <controls:ItemsContainer
core:Layout.Id="ExampleView.ItemsContainer1">
          <TextBlock core:Layout.Id="ExampleView.TextBlock1"/>
          <Button core:Layout.Id="ExampleView.Button1"/>
      </controls:ItemsContainer>
                                <controls:ItemsContainer
core:Layout.Id="ExampleView.ItemsContainer2">
          <TextBlock core:Layout.Id="ExampleView.TextBlock2"/>
          <Button core:Layout.Id="ExampleView.Button2"/>
      </controls:ItemsContainer>
  <StackPanel>
```

Dans l'exemple ci-dessus nous voyons une vue composée de deux conteneurs gérables *ExampleView.ItemsContainer1* et *ExampleView.ItemsContainer2.* Les éléments déclarés à l'intérieur de ces conteneurs seront affichés par défaut dans l'ordre dans lequel ils sont enregistrés.

Après avoir activé le mode de gestion de cette vue, l'utilisateur pourra modifier l'ordre des éléments. Il pourra supprimer un élément/des éléments du conteneur ou les y réinsérer (s'ils ont été supprimés avant). En revanche, il ne pourra pas transférer, par exemple l'élément *ExampleView.TestBlock1* du conteneur *ItemsContainer1* vers le conteneur *ItemsContainer2.* Le transfert des éléments entre les conteneurs ne sera pas possible.

Gestion des éléments dans le conteneur Grid

Un autre conteneur qui permet de manipuler ses éléments est Grid. Ce conteneur nécessite de définir une grille des colonnes et des lignes où il sera possible d'emplacer des éléments. Cette grille peut être définie par défaut dans xaml ou elle peut être gérée par l'utilisateur (l'utilisateur peut modifier le nombre de colonnes et de lignes). Par défaut, chaque élément ajouté au Grid ne sera pas affiché jusqu'à ce que l'utilisateur fasse glisser en mode de gestion l'élément dans la cellule sélectionnée du conteneur.

Il est possible de modifier l'affichage par défaut des composants pour que cela fonctionne comme dans le cas du conteneur ItemsContainer. Dans le cas où la propriété **DefaultShowChildren=true**, tous les éléments seront par défaut affichés dans la vue.

Comme dans le cas du conteneur du type ItemsContainer, chaque élément défini dans le conteneur Grid peut être supprimé/ajouté au conteneur. En outre, si un autre conteneur (Grid ou ItemsContainer) existe à l'intérieur du conteneur,

les éléments peuvent être ajoutés directement dans le conteneur imbriqué. Grâce à cela, il est possible de transférer par exemple les contrôles entre les conteneurs ItemsContainer différents.

Exemple :

```
<controls:Grid core:Layout.Id="ExampleView.BaseGrid">
      <TextBlock core:Layout.Id="ExampleView.TextBlock1"/>
      <Button core:Layout.Id="ExampleView.Button1"/>
      <TextBlock core:Layout.Id="ExampleView.TextBlock2"/>
      <Button core:Layout.Id="ExampleView.Button2"/>

                                  <controls:ItemsContainer
core:Layout.Id="ExampleView.ItemsContainer1">
                                  <controls:ItemsContainer
core:Layout.Id="ExampleView.ItemsContainer2">
  </controls:Grid>
```

Dans l’exemple ci-dessus, tous les contrôles ont été définis directement dans le Grid. Maintenant, lors du premier démarrage de cette vue, l’utilisateur verra une vue vide. Cela est dû au fait que, par défaut, le Grid n’affiche pas les éléments. En revanche, en mode de gestion il sera possible de définir la grille du Grid (ses colonnes et lignes) ainsi que de faire glisser dans le Grid chaque contrôle défini. Il sera également possible de transférer, par exemple, le contrôle *ExampleView.TextBlock1* directement dans le conteneur *ExampleView.ItemsContainer1* ou *ExampleView.ItemsContainer2*, après avoir ajouté au préalable ces conteneurs dans le Grid.

La grille par défaut du Grid peut être également définie dans le xaml à l’aide de la propriété **ColumnDefinition** et **RowDefinition.** Le nombre souhaité de colonnes ou de lignes est déterminé en saisissant autant de valeurs (nombre spécifique, * ou Auto) que le nombre de colonnes/lignes souhaité, séparées par des virgules.

Ainsi, par exemple, pour définir cinq colonnes, dont la première aura une largeur de 100, la deuxième sera

automatiquement sélectionnée et les autres obtiendront proportionnellement l'espace restant, la propriété **ColumnDefinition** doit avoir la valeur : **„100,Auto,*,*,*”**.

Il est également possible de définir, à l'aide de la propriété **Grid.Position,** l'emplacement par défaut des éléments dans une cellule appropriée du Grid. La valeur est définie à l'aide de quatre chiffres séparés par des virgules qui signifie le numéro de ligne (numérotation à partir de 0), le numéro de colonne (numérotation à partir de 0), le nombre de lignes (au moins 1) et le nombre de colonnes respectivement, qui doit être occupées par le contrôle.

Exemple :

```
<controls:Grid    core:Layout.Id="ExampleView.BaseGrid"
DefaultShowChildren="True">
       <controls:Grid.Style>
                    <Style TargetType="controls:Grid"
BasedOn="{StaticResource {x:Type controls:Grid}}">
                   <Setter Property="ColumnDefinition"
Value="*,*,*" />
                <Setter Property="RowDefinition" Value="*,*,*"
/>
           </Style>
       </controls:Grid.Style>

         <TextBlock core:Layout.Id="ExampleView.TextBlock1"
Text="Hello">
           <TextBlock.Style>
                        <Style TargetType="TextBlock"
BasedOn="{StaticResource {x:Type TextBlock}}">
                   <Setter Property="controls:Grid.Position"
Value="1,1,1,1" />
                </Style>
           </TextBlock.Style>
       </TextBlock>

    </controls:Grid>
```

Maintenant, la vue est composée du Grid pour lequel nous avons

activé l’affichage des éléments par défaut et pour lequel nous avons définis trois colonnes de la même largeur (largeur définie dynamiquement, en fonction de la largeur de la vue) et trois lignes de la même hauteur (hauteur définie dynamiquement comme dans le cas de la largeur). Le conteneur contient un élément TextBlock dont la position par défaut dans le conteneur a été défini comme la cellule centrale.

Extensions des vues existantes

Des éléments supplémentaires peuvent être ajoutés à tout vue existante dans POS. Une extension peut être l’ajout d’une nouvelle colonne au datagrid, la suppression d’une colonne existante ou l’ajout d’un n’importe quel contrôle dans l’espace préparé auparavant. Il est également possible d’appeler les logiciels des entreprises tierces en ajoutant simplement le contrôle du bouton sous lequel sera caché la logique initialisant un autre processus.

Ajouter un nouvel élément au conteneur existant

Grâce aux extensions, il est possible d’ajouter tout contrôle à la vue existante sélectionnée, mais uniquement dans les lieux préparés auparavant. Il n’y a pas de limite du nombre des éléments ajoutés, mais ils ne peuvent être placés que dans les conteneurs préparés à l’extension (ItemsContainer et Grid).

Afin d’ajouter un contrôle à la vue existante, il faut commencer par déterminer si la vue est gérable et si elle

contient le conteneur approprié où il sera possible de placer le nouvel élément. À ces fins, il faut ouvrir dans l'application POS la vue de la gestion d'interface. Ensuite, sélectionner de la liste des vues déroulante la vue pour laquelle nous allons créer une extension. Ensuite, il faut appuyer sur la barre Éléments et sélectionner à partir de la liste des conteneurs déroulante dans quel endroit de vue sera placé nouvel élément. Après avoir sélectionné un conteneur, il faut enregistrer son nom, car c'est un identifiant global qui sera indispensable à l'étape suivante de la création d'extension.

Si, lors de la création de notre propre vue, nous souhaitons la rendre extensible, il faut préparer l'espace pour cette éventualité en ajoutant un ou plusieurs contrôles de conteneur ou en construisant la vue à l'aide du contrôle Grid (Comarch.POS.Presentation.Core.Controls), en n'oubliant pas de leur donner des identifiants LayoutId uniques (pour plus de détails, voir l'article Gestion de vue et de ses éléments).

Pour ajouter un contrôle à la vue, il faut d'abord créer un nouveau module (voir le chapitre Nouveau module dans l'article Créer des vues) ou, s'il a déjà été créé, il faut aller au corps de la méthode **Initialize()** dans la classe Module. Afin d'étendre la vue avec un nouveau contrôle, il faut utiliser la méthode

AddButtonToContainer – en cas d'ajout de bouton ou

AddElementToContainer<TFrameworkElement> – en cas d'ajout de tout contrôle du type FrameworkElement

Les paramètres requis pour les deux méthodes sont :

- *containerLayoutId* (string) – identifiant du conteneur auquel sera ajouté un contrôle,
- *buttonLayoutId / elementLayoutId* (string) – identifiant unique du nouveau contrôle (chaque contrôle doit avoir dans le conteneur un identifiant unique),

- *styleKey* (string) – nom de clé facultatif dans le fichier ModernUI.xaml où sera défini le style du contrôle
- *buttonViewModelFunc / elementViewModelFunc* (Func<IViewModel, FrameworkElementViewModel>) – paramètre facultatif permettant de créer un ViewModel local pour le contrôle. Dans le ViewModel nous pourrons définir la logique avec laquelle le contrôle pourra se binder (à l'aide de style).

Par analogie, pour ajouter un élément au Grid, il faut appeler :

AddElementToGrid<TFrameworkElement> – les paramètres sont les mêmes que pour les éléments ajoutés à un conteneur

Exemple :

Nous voulons ajouter à la vue du reçu un bouton qui, lorsqu'il sera cliqué, fait afficher une notification avec la valeur du document.

Le nom du conteneur auquel le bouton sera ajouté est DocumentViewRightButtonsContainer. Dans la classe **Module** du nouveau module d'extension, dans la méthode **Initialize**, nous ajoutons la ligne :

```
AddButtonToContainer("DocumentViewRightButtonsContainer", "ExtensionButton1", "ButtonStyle", ButtonViewModelFunc);
```

où l'**ExtensionButton1** est le nom unique (identifiant LayoutId) du nouveau bouton, le **ButtonStyle** est le nom de la clé avec le style pour ce bouton et le **ButtonViewModelFunc** est la méthode qui retourne le ViewModel local où sera implémentée la logique appelant la notification avec le contenu approprié.

```
private FrameworkElementViewModel ButtonViewModelFunc(IViewModel viewModel)
{
    return new ButtonViewModel(viewModel, ViewManager,
```

```
Container);
}

public class ButtonViewModel : FrameworkElementViewModel
{
    public DelegateCommand ExtensionButtonCommand { get; set;
}

    private readonly IDocumentViewModel _documentViewModel;
    private readonly INotificationService _notifyService;

    public ButtonViewModel(IViewModel viewModel, IViewManager
viewManager, IUnityContainer container) : base(viewModel,
viewManager)
    {
        if (viewModel.IsDesignMode)
            return;

                                        _notifyService    =
container.Resolve<INotificationService>();
        _documentViewModel = (DocumentViewModel)viewModel;
                               ExtensionButtonCommand=new
DelegateCommand(ExtensionButtonAction);
    }

    private void ExtensionButtonAction()
    {
            _notifyService.Show($"Montant de document :
{_documentViewModel.Document.Value}", NotifyIcon.Information);
    }
}
```

Dans le fichier ModernUI.xaml de notre module, nous ajoutons le style pour le nouveau bouton en y définissant le contenu du bouton et nous relions l'action de clic à la commande associée à la méthode **ExtensionButtonAction.**

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentati
on"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                                        xmlns:buttons="clr-
```

```
namespace:Comarch.POS.Presentation.Core.Controls.Buttons;assembly=Comarch.POS.Presentation.Core">

<Style x:Key="ButtonStyle" TargetType="buttons:Button"
       BasedOn="{StaticResource {x:Type buttons:Button}}">
    <Setter Property="Content" Value="Pokaż wartość" />
        <Setter  Property="Command"  Value="{Binding ExtensionButtonCommand}" />
</Style>
```

Le code entier de l'exemple est disponible dans le chapitre Ajouter un contrôle au conteneur d'une vue existante dans l'article Exemples.

Ajouter une colonne à un DataGrid existant

La façon la plus simple d'ajouter une nouvelle colonne à un datagrid existant de la vue de document (par exemple du reçu/de la facture, de la commande client etc.) est assigner dans le système ERP un attribut à un élément de document (pour plus de détails, voir l'article Gestion des attributs). En revanche, si nous ne voulons pas que la nouvelle colonne soit un attribut, il faut procéder comme dans le cas de l'ajout des contrôles aux conteneurs. Afin d'étendre une liste DataGrid existante, il faut connaître son identifiant unique. Pour le connaître, il faut ouvrir la vue contenant le datagrid en mode de gestion des vues, le cliquer et trouver son LayoutId dans la rubrique Propriétés. Ensuite, nous accédons au contrôle à l'aide de la méthode **RegisterDataGridExtension** de la classe ModuleBase et nous implémentons l'ajout d'une nouvelle colonne dans la collection. Les paramètres de cette méthode sont :

- *dataGridLayoutId* (string) – identifiant du layoutId du datagrid étendu,
- *action* (Action<DataGrid, IViewModel, bool>) – délégué à la méthode qui sera appelée lors de la création du contrôle de datagrid

Exemple 1.

Nous voulons ajouter une colonne à la liste de la vue d'un nouveau reçu qui affichera l'information si la position ajoutée ne dépasse pas le montant défini.

L'identifiant Layout Id de la liste dans un reçu c'est ReceiptDocumentViewDataGrid. Dans la méthode Initialize de la classe Module nous ajoutons :

```
RegisterDataGridExtension("ReceiptDocumentViewDataGrid", DataGridNewColumn);
```

Ensuite, nous implémentons la méthode DataGridNewColumn :

```
private void DataGridNewColumn(DataGrid dataGrid, IViewModel viewModel, bool isDesignMode)
{
    var column = new DataGridTextColumn
    {
        Header = "Dépasse 100 ?",
             Binding = new Binding {Converter = new ValidateConverter()}
    };

                                        Layout.SetId(column, "DocumentViewDataGridExtendedColumn1");
    dataGrid.Columns.Add(column);
}
```

La valeur du paramètre *isDesignMode* change en *true*, lorsque la vue contenant ce datagrid est ouverte en mode de gestion des vues. Ensuite, nous ajoutons la classe de convertisseur ValidateConverter qui contiendra la logique retournant la valeur appropriée pour chaque cellule dans la colonne ajoutée :

```
internal class ValidateConverter : IValueConverter
{
     public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
```

```
        var row = value as TradeDocumentItemRow;

        if (row!=null)
        {
            return row.Price > 100 ? "OUI" : "NON";
        }

        //en cas de positions reprises
        return "Ne concerne pas";
    }
   …
}
```

L’exemple ci-dessus suppose que toutes les informations requises à afficher les valeurs sont disponibles dans l’entité de ligne. Dans le cas où la logique commerciale pour la nouvelle colonne doit être également étendue, il faut d’abord télécharger les données indispensables pour la nouvelle colonne. Les données téléchargées peuvent être stockées dans une propriété publique spécialement préparée disponible dans chaque viewmodel – **CustomDataDictionary**. C’est une propriété du type répertoire (string, object) à laquelle nous pouvons faire appel dans la colonne définie à l’aide de binding.

Exemple 2.

Nous ajoutons une nouvelle colonne dans la vue du reçu qui affichera le nom du tarif du produit ajouté à la liste. L’entité de produit (*IDocumentItemRow*) ne contient que l’identifiant du tarif (*PriceListId*), mais n’a pas de son nom.

Nous commençons par télécharger la liste complète des tarifs et l’enregistrer dans le répertoire **CustomDataDictionary**. Il suffit de télécharger les tarifs une seule fois au début, par exemple lors de l’ouverture de vue. À ces fins, nous pouvons utiliser les extension points pour nous brancher sur la méthode après l’initialisation – **AfterOnInitialization**, ou par l’héritage de la classe DocumentViewModel et la surcharge de

la méthode **OnInitialization.** Il est également possible de télécharger la liste des tarifs lors du branchement avec la nouvelle colonne, c'est-à-dire dans l'action de la méthode **RegisterDataGridExtension**. Aux fins de cet exemple, nous allons sélectionner cette dernière méthode.

Dans la méthode Initialize de la classe Module nous faisons appel :

```
RegisterDataGridExtension("ReceiptDocumentViewDataGrid", DataGridNewColumnWithCustomBL);
```

Ensuite, nous implémentons la méthode DataGridNewColumnWithCustomBL

```
private void DataGridNewColumnWithCustomBL(DataGrid dataGrid, IViewModel viewModel, bool isDesignMode)
{
       if (viewModel is CustomDocumentViewModel vm)
       {
                  //fill custom dictionary with dictionary of data for custom column (priceListId => name)
                  vm.CustomDataDictionary.Add(CustomColumnTest, new Dictionary<int, string>
                 {
                     {1, "premier" },
                     {2, "deuxième" }
                 });

                  //after initial price changed refresh custom column binding
                 vm.AfterSetInitialPrice += () => { vm.OnPropertyChanged(nameof(vm.CustomDataDictionary));
                 };
       }

       var column = new DataGridTextColumn
       {
          Header = "Price list name",
          Binding = new MultiBinding
          {
```

```
            Converter = new CustomMultiConverter(),
            Bindings =
            {
              new Binding(),
              new Binding
              {
                                    RelativeSource = new
RelativeSource(RelativeSourceMode.FindAncestor,
typeof(DocumentView), 1),
                  Path                =                new
PropertyPath($"DocumentViewModel.CustomDataDictionary[{CustomC
olumnTest}]")
              }
            }
          }
      };

    dataGrid.Columns.Add(column);
}
```

Dans la première partie de la méthode, nous simulons le téléchargement des tarifs en complétant le répertoire CustomDataDictionary avec un répertoire avec deux valeurs (id du tarif, nom du tarif). Ceci est fait exprès, car CustomDataDictionary pourrait être utilisé à stocker également des autres informations (par exemple pour une autre logique commerciale). La clé *CustomColumnTest* est un champ du type const string défini dans la classe Module, contenant un nom unique grâce auquel il sera possible d'identifier notre collection de données (tarifs).

```
public const string CustomColumnTest = "CustomColumnTest";
```

Dans la seconde partie de la méthode, nous créons une colonne avec multibinding et un convertisseur. Le multibinding définit le binding à l'entité de ligne actuelle, ainsi qu'au répertoire avec les tarifs contenus dans le répertoire CustomDataDictionary sous la clé *CustomColumnTest*. Dans le convertisseur, par contre, nous obtenons les deux objets afin

de pouvoir retourner le nom de la liste de prix sur la base de l'id contenu dans l'entité et du nom contenu dans le dictionnaire.

```
internal class CustomMultiConverter : IMultiValueConverter
    {
           public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)
        {
                    var documentItemRow = values[0] as IDocumentItemRow;
             var dictionary = values[1] as Dictionary<int, string>;

             //if item has price list id then show price list name (when initial price changes, price list id nulls)
              if (documentItemRow?.PriceListId.HasValue ?? false)
            {
                string name = null;
                                                                if (dictionary?.TryGetValue(documentItemRow.PriceListId.Value, out name) ?? false)
                {
                    return name;
                }
            }

            return null;
        }
...
    }
```

L'exemple complet contient encore le branchement sur la méthode SetInitialPrice dans lequel est appelé la demande d'actualiser le binding avec CustomDataDictionary, car après la modification du prix initial, le prix affiché n'est plus le prix du tarif (la propriété PriceListId est désormais null) et la nouvelle colonne avec le nom ne doit plus l'afficher.

Le code complet des exemples est disponible dans le chapitre Ajouter une colonne à DataGrid dans une vue existante dans l'article Exemples.

Accès à un élément existant

Il est également possible d'avoir accès aux propriétées de chaque contrôle existant qui a un layoutId défini. À cette fin, il faut utiliser dans la classe Module la méthode **AttachToFrameworkElement.** Les paramètres de cette méthode sont analogues à ceux de la méthode RegisterDataGridExtension.

Ajouter des éléments à la zone de statut

La zone de statut se caractérise par le fait que les éléments y emplacés sont disponibles durant le fonctionnement de l'application, indépendamment des vues ouvertes. L'accès à cette zone peut se faire de tout vue de base. Afin d'ajouter à la zone de statut un contrôle avec une logique personnalisée, il faut appeler la méthode **AddElementToStatusBar<TFrameworkElement>** dans la méthode **Initialize** de la classe **Module.** Les arguments de la méthode sont :

- elementLayoutId (string) – identifiant unique du contrôle ajouté,
- styleKey (string) – nom de clé dans le fichier ModernUI.xaml où sera défini le style du contrôle
- elementViewModelFunc (Func<IStatusBar,StatusBarEementBase>) – délégué à la méthode qui sera appelée lors de la création de contrôle

Pour voir un exemple d'implémentation, consultez le chapitre Exemple d'extension de la zone de statut dans l'article Exemples.

Extensions de la gestion d'interface

Les options d'extensibilité dont nous avons parlées jusqu'à présent concernent uniquement les contrôles intérieurs de POS. En cas de nécessité d'ajouter un nouveau contrôle gérable, nous pouvons utiliser le mécanisme d'extensibilité de la gestion d'interface. Ce mécanisme permet d'enregistrer de nouveaux types de contrôles et de définir lesquelles de propriétés seront gérables. À part de cela, il est également possible de rendre gérables les propriétés des contrôles existants et de masquer la gestion des propriétés sélectionnées.

Masquer les propriétés pour édition

Si une propriété est masquée, elle est supprimée de la gestion d'interface. L'utilisateur ayant accès à la gestion d'interface ne pourra pas modifier sa valeur. La valeur du contrôle dont la propriété sera masquée sera définie par défaut pour la valeur définie dans les styles. Pourtant, si la valeur par défaut a été définie dans le fichier **ModernUI.xaml**, elle sera ignorée (les différences dans la définition des valeurs par défaut ont été décrites dans le chapitre Définir les valeurs par défaut des propriétés gérables dans l'article Gestion de vues et de ses éléments).

Nous pouvons masquer des propriétés spécifiques sur un contrôle spécifique identifié de manière unique par son Layout Id. La méthode à masquer les propriétés est la méthode **DisablePropertiesForLayoutElement** dans la classe **PropertiesManager**. Paramètres de cette méthode :

- *layoutId* (string) – paramètre qui est l'identifiant du contrôle dont la propriété on veut masquer.
- *properties* (params DependencyProperty[]) – une ou plusieurs propriétés qui seront masquées pour un contrôle donné.

Exemple du masquage de la propriété Icône (ImageKey) de la mosaïque Nouvelle vente (layout id NewSalesDocument) :

```
PropertiesManager.Instance.DisablePropertiesForLayoutElement("
NewSalesDocument", TileButton.ImageKeyProperty);
```

Attention
Dans le cas présenté ci-dessus, l'icône disparaîtra de la mosaïque Nouvelle vente, car la valeur par défaut de la propriété ImageKey a été définie dans le fichier ModernUI.xaml.
Exemple du masquage de la propriété Hauteur (Height) du bouton Valider (AcceptButton) de la configuration globale :

```
PropertiesManager.Instance.DisablePropertiesForLayoutElement("
AcceptButton", FrameworkElement.HeightProperty);
```

Ajouter des nouvelles propriétés à modifier

Dans le cas où un contrôle existant ou un tout neuf contrôle a des propriétés personnalisées définies et nous voulons qu'elles soient gérables dans la gestion d'interface, il faut les enregistrer. Cela peut se faire de deux manières, selon que la propriété doit être visible uniquement pour un contrôle à un endroit spécifique ou globalement pour toutes les occurrences de ce contrôle.

Dans le premier cas (enregistrement visible uniquement pour un contrôle à un endroit spécifique), il faut utiliser la méthode **AddPropertiesForLayoutElement** de la classe **PropertiesManager**. Paramètres de la méthode :

- *layoutId* (string) – identifiant unique de contrôle (Layout Id),
- *properties* (params DependencyProperty[]) – propriétés de contrôle qui seront gérables

Dans le second cas (enregistrement visible globalement pour toutes les occurrences de contrôle), il faut utiliser la méthode **RegisterControlProperties** de la classe **PropertiesManager**. Paramètres de la méthode :

- *controlType* (Type) – type de contrôle,
- *properties* (params DependencyProperty[]) – propriétés qui seront visible pour le contrôle du type indiqué
- *baseTypeProperties* (Type) – type de contrôle de base à partir de laquelle seront téléchargées les propriétés enregistrées auparavant et utilisées à enregistrer les propriétés de ce contrôle

Gestion des attributs

L'application POS prend en charge les attributs à valeur unique synchronisés à partir du système ERP. Types d'attributs pris en charge : texte, nombre, répertoire, valeur logique, liste, date. Pour que la classe d'attribut donnée soit synchronisée vers un point de vente POS, elle doit être marquée dans le système ERP (Standard) à afficher et/ou à modifier dans la section Retail POS. Les attributs peuvent être affichés sous forme des colonnes dans la liste (contrôle datagrid) ou sous forme des contrôles (générés en fonction du type de classe d'attribut, par exemple pour la classe du type texte ça sera le TextBox, alors que pour le type répertoire ça sera le ComboBox).

Ajouter une nouvelle classe

d’attribut à une vue existante dans POS

Dans l’application POS, chaque vue qui prend en charge les attributs est directement associé à un objet du système ERP. Par exemple, la vue Liste des clients affiche les attributs dans la liste sous forme des colonnes. Il est possible d’afficher les attributs avec leurs valeurs attribuées uniquement à l’objet Client dans le système ERP. Dans le cas de la vue du reçu, la liste (datagrid) affiche les classes d’attributs associées uniquement à l’objet Élément de reçu (REC), alors que les attributs affichés dans le coin inférieur droit sous forme des contrôles sont associés à l’objet Reçu (REC).

Liste des vues prenant en charge l’attribut avec les objets du système ERP associés

Vue POS	Élément de vue	Entité métier dans le système ERP
Nouvelle vente (DocumentView)	Liste des positions	Élément du reçu (REC)
	Document	Reçu (REC)
Aperçu du reçu (DocumentPreviewView)	Liste des positions	Élément du reçu (REC)
	Document	Reçu (REC)
Attributs du reçu (vente rapide) (DocumentAttributesView)	Document	Reçu (REC)
Détails de position dans le reçu (vente rapide) (DocumentItemPropertiesView)	Document	Élément du reçu (REC)
Facture (DocumentView)	Liste des positions	Élément de la facture client (FC)
	Document	Facture client (FC)
Aperçu de la facture (DocumentPreviewView)	Liste des positions	Élément de la facture client (FC)
	Document	Facture client (FC)

Attributs de la facture (vente rapide) (DocumentAttributesView)	Document	Facture client (FC)
Détails de position dans la facture (vente rapide) (DocumentItemPropertiesView)	Document	Élément de la facture client (FC)
Correctif manuel du reçu Correctif manuel de la quantité du reçu – échange (ManualExchangeView)	Liste des positions	Élément du correctif manuel de la quantité du reçu (CQR)
	Document	Correctif manuel de la quantité du reçu (CQR)
Aperçu du correctif manuel de la quantité du reçu (ManualCorrectionPreviewView)	Liste des positions	Élément du correctif manuel de la quantité du reçu (CQR)
	Document	Correctif manuel de la quantité du reçu (CQR)
Correctif manuel de la quantité de la facture Correctif manuel de la quantité de la facture – échange (ManualExchangeView)	Liste des positions	Élément du correctif manuel de la quantité de la facture (CQFC)
	Document	Correctif manuel de la quantité de la facture (CQFC)
Aperçu du correctif manuel de la quantité de la facture (ManualCorrectionPreviewView)	Liste des positions	Élément du correctif manuel de la quantité de la facture (CQFC)
	Document	Correctif manuel de la quantité de la facture (CQFC)
Correctif du reçu (ExchangeView)	Liste des positions	Élément du correctif de la quantité du reçu
	Document	Correctif de la quantité du reçu

Aperçu du correctif du reçu (CorrectionPreviewView)	Liste des positions	Élément du correctif de la quantité du reçu
	Document	Correctif de la quantité du reçu
Correctif de la facture (ExchangeView)	Liste des positions	Élément du correctif de la quantité de la facture
	Document	Correctif de la quantité de la facture
Aperçu du correctif de la facture (CorrectionPreviewView)	Liste des positions	Élément du correctif de la quantité de la facture (CQFC)
	Document	Correctif de la quantité de la facture (CQFC)
Facture d’acompte (AdvanceInvoiceView)	Document	Facture d’acompte
Aperçu de la facture d’acompte (AdvanceInvoicePreviewView)	Document	Facture d’acompte
Correctif de la facture d’acompte (AdvanceInvoiceCorrectionView)	Document	Correctif de la facture d’acompte
Aperçu du correctif de la facture d’acompte (AdvanceInvoiceCorrectionPreviewView)	Document	Correctif de la facture d’acompte
Bordereau de détaxe (TaxFreeView)	Liste des positions	Élément du bordereau de détaxe (BD)
	Document	Bordereau de détaxe (BD)
Aperçu du bordereau de détaxe (TaxFreePreviewView)	Liste des positions	Élément du bordereau de détaxe (BD)
	Document	Bordereau de détaxe (BD)

Liste des ventes (DocumentsListView)	Liste des positions	Reçu (REC), Facture client (FC), Facture d’acompte, Correctif de la quantité du reçu (CQR), Correctif de la quantité de la facture (CQFC), Correctif manuel de la quantité du reçu (CQR), Correctif manuel de la quantité de la facture (CQFC), Correctif de la facture d’acompte, Bordereau de détaxe (BD)
Clients (CustomesListView)	Liste des positions	Client
Ajouter/Modifier le particulier (CustomerView)	Document	Client
Ajouter/Modifier le professionnel (BusinessCustomerView)	Document	Client
Détails du client (CustomerDetailsView)	Document	Client
Détails du professionnel (BusinessCustomerDetailsView)	Document	Client
Nouvelle commande (SalesOrderView)	Liste des positions	Élément de la commande client (COMC)
	Document	Commande client (COMC)
Aperçu de la commande client (SalesOrderPreviewView)	Liste des positions	Élément de la commande client (COMC)
	Document	Commande client (COMC)

Préparation de la commande (SalesOrderPreparationView)	Liste des positions	Élément de la commande client (COMC)
Nouveau devis (SalesQuoteView)	Liste des positions	Élément de la commande client (DC)
	Document	Commande client (DC)
Aperçu du devis client (SalesQuotePreviewView)	Liste des positions	Élément de la commande client (DC)
	Document	Commande client (DC)
Commandes et devis client (SalesOrdersListView)	Liste des positions	Commande client (COMC), Devis client (DC)
Réclamation (ComplaintView)	Liste des positions	Élément de la réclamation de vente (REVC)
	Document	Réclamation de vente (REVC)
Réclamations (ComplaintsListView)	Liste des positions	Réclamation de vente (REVC)
Document de caisse (DC/RC) (CashDocumentView)	Document	Transaction caisse/banque
Dépôt/Retrait Coffre-fort (VaultInflowOutflowView)	Document	Transaction caisse/banque
Documents de caisse (CashDocumentsListView)	Liste des positions	Transaction caisse/banque
Aperçu du document de sortie de stock ME- (WarehouseDocumentPreviewView)	Liste des positions	Élément du mouvement d’entrepôt (ME-)
	Document	Mouvement d’entrepôt (ME-)

Nouveau document de sortie du stock (NewWarehouseDocumentView)	Liste des positions	Élément du mouvement d’entrepôt (ME-)
	Document	Mouvement d’entrepôt (ME-)
Document de sortie du stock (WarehouseDocumentView)	Liste des positions	Élément du mouvement d’entrepôt (ME-)
	Document	Mouvement d’entrepôt (ME-)
Document de réception PRR (ReceivingAndDeliveryReportView)	Document	Protocole de réception (PRR)
Documents d’entrepôt (WarehouseDocumentsListView)	Liste des positions	Mouvement d’entrepôt (ME-), Protocole de réception (PRR), Commande fournisseur (COMF), Bon de réception (REC)
Réception d’un bordereau d’expédition (DeliveryNoteView)	Document	Réception d’un bordereau d’expédition
Aperçu du bordereau d’expédition (DeliveryNotePreviewView)	Document	Réception d’un bordereau d’expédition
Réception de la livraison ME- (WarehouseDocumentsToReceiptListView)	Liste des positions	Mouvement d’entrepôt (ME-), Protocole de réception (PRR), Commande fournisseur (COMF), Bon de réception (REC)
Bon de réception (PurchaseOrderReceptionView)	Liste des positions	Élément du bon de réception (REC)
	Document	Bon de réception (REC)

Commande fournisseur (PurchaseOrderView)	Liste des positions	Élément de la commande fournisseur (COMF)
	Document	Commande fournisseur (COMF)
Correctifs des ressources (StockCorrectionsListView)	Liste des positions	Profit (PROF), Perte (PERT)
Profit/Perte (NewInternalReceiptOrReleaseView)	Liste des positions	Elément du profit (PROF), élément de la perte (PERT)
	Document	Profit (PROF), Perte (PERT)
Aperçu du Profit/ de la Perte (InternalReceiptOrReleasePreviewView)	Liste des positions	Elément du profit (PROF), élément de la perte (PERT)
	Document	Profit (PROF), Perte (PERT)
Commande interne (InternalOrdersListView)	Liste des positions	Commande interne (COMI)
Nouveau colis (GeneratedWarehouseDocumentView)	Liste des positions	Elément du mouvement d’entrepôt (ME-)
	Document	Mouvement d’entrepôt (ME-)
Transferts manuels (ManualMovementsListView)	Liste des positions	Elément du mouvement d’entrepôt (ME-)
Transfert manuel (NewManualMovementWarehouseDocumentView)	Liste des positions	Elément du mouvement d’entrepôt (ME-)
	Document	Mouvement d’entrepôt (ME-)
Mouvements internes (InternalMovementsListView)	Liste des positions	Elément du mouvement d’entrepôt (ME-)
Mouvement interne (NewInternalWarehouseDocumentView)	Liste des positions	Mouvement d’entrepôt (ME-)

Inventaire (InventoryCountView)	Document	Inventaire
Liste de comptage (InventoryCountListView)	Document	Liste de comptage
Commande interne créée (CreatedInternalOrdersListView)	Liste des positions	Commande interne (COMI)
Nouvelle commande interne (NewInternalOrderView)	Document	Commande interne (COMI)
Aperçu de la commande interne (CreatedInternalOrderPreviewView)	Document	Commande interne (COMI)
Liste des articles	Liste des positions	Article

Pour ajouter à POS la classe d'attribut sélectionnée, il faut la marquer comme Retail POS (aperçu et/ou modification). L'option d'aperçu permettra à l'attribut de fonctionner en mode lecture seule dans l'application POS. En revanche, si l'option modification est sélectionnée, la valeur de l'attribut pourra être modifiée et après l'enregistrement cette information sera transférée au système ERP. Ensuite, il faut lier la classe d'attribut avec le type d'objet approprié qui est pris en charge par POS (voir le tableau ci-dessus). Après avoir ajouté et synchronisé les réglages, les nouveaux attributs apparaîtront dans l'application POS sur la vue appropriée et l'utilisateur POS pourra les gérer à partir du niveau de la configuration d'interface.

Par exemple, nous ajoutons une nouvelle classe d'attribut X qui est du type texte et nous la marquons comme modifiable dans la rubrique Retail POS. Ensuite, nous la lions avec l'objet Client dans le système ERP. Après la synchronisation des données, nous démarrons l'application POS et nous ouvrons la configuration de la vue Ajouter un client. Dans la configuration, nous cochons le conteneur approprié (**CustomerItemsContainer**) qui permet l'affichage des attributs. Ensuite, il faut sélectionner dans la liste disponible à

droite le nouvel attribut X et le faire glisser dans l'endroit souhaité sur la vue.

Ajouter une nouvelle classe d'attribut à la nouvelle vue qui ne la prend pas en charge

La prise en charge des attributs peut être également ajoutée aux vues personnalisées créées dans le cadre de l'extension. Les attributs peuvent être affichés sous forme des colonnes générées dynamiquement dans la liste (contrôle DataGrid) ou sous forme des contrôles indépendants (le type de contrôle sera différent en fonction du type de classe d'attribut).

Attributs sous forme des colonnes dans la liste DataGrid

Afin d'implémenter dans les listes les attributs sous forme des colonnes générées dynamiquement, il faut dans un premier temps implémenter l'interface **IAttributable** pour l'entité de données. Cette interface fournit trois propriétés indispensables à la prise en charge correcte des attributs dans les listes. Les deux premières propriétés : **Id** et **ObjectType** doit être définies conformément à l'entité. L'Id est l'identifiant de l'entité et ObjectType est le type d'entité. La troisième propriété **Attributes** est un répertoire qui doit être rempli lors du téléchargement asynchrone des données pour la liste. Pour remplir ce répertoire, il faut utiliser la méthode **FillAttributesForList** qui est dans le service **IAttributesService**. Avant de pouvoir afficher les données du répertoire, la liste doit d'abord générer des colonnes supplémentaires. Pour que cela soit possible, lors du premier téléchargement des données (IsInitialization=true) il faut définir la propriété **AttributeClasses** dans la classe **AsyncDataGridCollection**. Pour définir cette propriété il faut

utiliser la méthode **GetAttributesClasses** du service **IAttributesService**. Cette méthode adopte deux paramètres : le premier permet de définir quelles classes d'attributs doivent être affichées sur une liste donnée et le second filtre quelles classes seront visibles pour des groupes donnés. Dans le cas où le second paramètre est null, toutes les classes des groupes sélectionnés dans le premier paramètre sont téléchargées. L'implémentation doit être réalisée d'une façon qui permet la gestion du filtrage des classes d'attributs en mode design de vue. De ce fait, cette méthode doit être appelée dans le DesignViewModel d'une vue donnée avec le second paramètre définit sur null. Ceci permet de télécharger toutes les classes et de générer toutes les colonnes possibles (cachées par défaut). En revanche, dans le ViewModel de la vue, il faut transmettre au second paramètre la propriété **VisibileAttributesClassesList** qui contiendra une liste des classes d'attributs définie en mode design par l'utilisateur. Dans le cas de la méthode FillAttributesForList qui remplit le répertoire Attributes, il faut transmettre deux paramètres. Le premier c'est une liste des entités (une entité doit implémenter IAttribuable) et le second c'est une liste des identifiants des classes d'attributs dans laquelle il faut compléter chaque entité avec les valeurs de ces classes d'attributs. Aux fins d'optimisation et pour ne pas télécharger trop d'informations, le second paramètre est défini en transmettant la propriété **VisibleAttributesClassesList**.

Le tri des colonnes qui ont été générées pour les attributs n'est pas activé.

Les colonnes générées pour les attributs sont par défaut cachées. Afin de les afficher, il faut implémenter en mode design le téléchargement de tous les attributs possibles pour une liste donnée, pour que l'utilisateur puisse sélectionner cette colonne et définir sa visibilité, ainsi que ses autres propriétés.

Dans le cas où la liste est en mode lecture seule, les attributs sont affichés sous forme de texte. En revanche, en mode d'édition les attributs sont générés sous forme des contrôles appropriés, en fonction du type de données. Pour un type logique ça sera le CheckBox, pour une liste et pour un répertoire le ComboBox et pour les autres types le TextBox.

Un exemple complet d'implémentation peut être consulté dans le chapitre Vue d'un document commercial avec la gestion des attributs dans l'article Exemples.

Attributs comme contrôles indépendants

Une façon alternative d'affichage des attributs dans les vues créées est de les implémenter sous forme des contrôles dynamiquement générés qui dépend du type de classe d'attribut. Ces contrôles permettent également de modifier les valeurs des attributs. Leur implémentation est composée de trois étapes. La première étape consiste à télécharger les attributs et les afficher sous forme de contrôles visibles dans le conteneur défini. La seconde à implémenter la validation des valeurs dans les attributs. La troisième étape c'est l'enregistrement des modifications apportées aux valeurs des attributs. Un exemple complet d'implémentation peut être consulté dans le chapitre Vue d'un document commercial avec la gestion des attributs dans l'article Exemples.

Vérification des autorisations

Authentification d’utilisateur

Les utilisateurs POS sont authentifiés à l’aide du service ISecurityService. Il contient les méthodes permettant la connexion d’utilisateur au système, la déconnexion, le verrouillage de l’écran et la vérification si l’utilisateur a des autorisations requises.

Méthodes du service ISecurityService :

- **SignIn(string login, SecureString password)**
  Cette méthode permet à l’utilisateur de se connecter au système.
- **SignOut()**
  Cette méthode déconnecte l’utilisateur. En conséquence, toutes les vues ouvertes auparavant sont fermées et l’application revient à la vue de connexion.
- **Lock()**
  Cette méthode bloque l’écran. Si elle est appelée, la vue de connexion est ouverte et il n’est pas possible d’afficher les autres vues ouvertes jusqu’à ce que l’identité de l’utilisateur ne soit vérifiée en fournissant le mot de passe.

Autorisation d’utilisateur

Chaque utilisateur connecté dans POS peut avoir ou ne pas avoir les autorisations aux endroits de l’application définis auparavant. Les autorisations sont définies dans le système ERP par le biais de l’attribution des autorisations aux actions et objets métiers à un groupe d’utilisateurs. Afin de vérifier si l’utilisateur connecté a des autorisations requises, il faut utiliser le service **IAuthorizationService** et sa méthode **ValidatePermissions** ou, si nous sommes dans la classe de ViewModel, appeler directement la méthode élargissant avec le même nom. Appeler la méthode fait vérifier les autorisations. Si la vérification échoue, l’application ouvre une vue modale où il est possible de sélectionner

l’utilisateur pour lequel le processus de vérification sera relancé (à condition qu’un nom d’utilisateur et un mot de passe appropriés soient fournis). Ceci ne change pas l’utilisateur connecté, mais fait passer l’étape de vérification des autorisations. Paramètres de la méthode :

- *accessDeniedMessage* (string) – texte qui doit apparaître lorsque l’utilisateur n’a pas d’autorisations vérifiées,
- *autorization* (IAuthorization) – autorisation que nous voulons vérifier,
- *successAction* (Action) – action qui doit être exécutée lorsque la vérification se termine avec succès,
- *cancelAction* (Action) – action facultative qui doit être exécutée lorsque l’utilisateur annule la possibilité d’élever les autorisations en se connectant à un autre compte lors de la vérification des autorisations,
- *login* (string) – nom d’utilisateur facultatif pour lequel les autorisations seront vérifiées,
- *password* (SecureString) (par défaut null) – mot de passe de l’utilisateur pour lequel les autorisations seront vérifiées (requis si le nom d’utilisateur est saisi)
- *logByCard* (bool) (par défaut false) – définit si l’utilisateur s’est connecté à l’aide d’une carte magnétique

Exemple :

Nous vérifions si l’utilisateur connecté est autorisé à ajouter un reçu :

```
this.ValidatePermissions(« Pas d'autorisations à émettre un reçu",
Authorization.Check.To(PermissionName.Receipt).WithLevels(PermissionLevel.Add),
                () =>
                {
                      NotificationService.Show("Autorisations vérfiées avec succès", NotifyIcon.Information);
                });
```

Gestion des exceptions

Chaque exception non réglée dans POS sera représentée sous forme d'une vue de message et enregistrée dans le fichier du journal d'événements. La fenêtre se compose du titre d'erreur et de son contenu. Si l'exception est du type System.Exception ou hérite de lui, le titre d'erreur sera le type d'exception et le contenu sera son Message. L'information enregistrée dans le journal d'événements contiendra le type d'exception, le contenu Message et la pile d'appel. Cette solution n'est pas trop douée, car l'utilisateur ne doit pas s'interroger quand il voit des titres bizarres comme : NullReferenceException.

Une meilleure solution est d'utiliser la classe **Comarch.POS.Library.Erros.RetailException**. Cette classe hérite de System.Exception et introduit deux propriétés supplémentaires qui sont par défaut prédéfinies. C'est la propriété **UITitle** utilisée à afficher le titre d'erreur et **UIMessage** utilisée à afficher le contenu plus adapté aux besoins de l'utilisateur. Le contenu est bien sûr localisé dans les langues pris en charge par POS. Comme dans le cas précédent, le type d'exception, son contenu original et la pile d'appel sont tous enregistrés dans le journal d'événements. Pour définir un titre et/ou un contenu personnalisé, il faut créer un nouveau type d'exception héritant de la classe RetailException et surcharger les propriétés appropriées.

Dans l'application POS est définie une dizaine des types dérivés du RetailException. Le tablau ci-dessus présente une liste exemplaire incomplète avec le contenu d'UITitle et UIMessage pour la langue polonaise.

Type d’exception	UITitle	UIMessage
RetailException	Erreur système	Une erreur inconnue s’est produite
RetailSecurityException	Accès refusé	Une erreur inconnue s’est produite
RetailVoucherException	Gestion des bons d’achat	Une erreur inconnue s’est produite
RetailVoucherBlockedException	Gestion des bons d’achat	Le bon d’achat {0} est bloqué.

Extensions de la synchronisation des données avec DataService

Ajouter et actualiser les données

Action du côte de Comarch ERP Standard

La base de Comarch ERP Standard contient la procédure **POS.ExportCustomObjects**. Cette fonction doit être remplacée de façon à ce qu’elle récupère les données qui doivent être envoyées à un point de vente particulier depuis la base de données et les renvoie au format XML.

Les paramètres ci-dessous sont passés à la procédure pour limiter et personnaliser l’ensemble des données envoyées à un point de vente POS particulier :

- @rowVersion (bigint) – valeur permettant d'effectuer une synchronisation différentielle. C'est la valeur qui se trouve dans XML généré par cette procédure lors de la dernière synchronisation réussie (dans l'attribut **RowVersion**).
- @companyUnitId (int) – Id de centre où un point de vente POS a été défini (CompanyStructure.CompanyUnits)
- @pointOfSaleId (int) – Id de point de vente POS (Synchronization.PointsOfSales)
- @languageId (int) – Id de langage de données (Dictionaries.Languages)

Lors de l'exportation des données des colonnes du type **bit** et **datetime**, il faut utiliser les fonctions **POS.GetBitString** et **POS.GetDatetimeString**.

Action du côté de POS

Dans la base POS existe la procédure **Synchronization.ImportCustomObjects**. Elle doit être remplacée de façon à ce qu'elle mette à jour les tableaux dans la base de données POS sur la base des données XML reçues de Comarch ERP Standard – celles générées par la procédure **POS.ExportCustomObjects**.

Exemple

L'exemple présente la synchronisation différentielle des données de deux tableaux – l'exportation de la base Comarch ERP Standard et l'importation dans la base POS :

- Dic_PaymentForms -> Configuration.CustomPaymentForms
- Dic_Country -> Configuration.CustomCountries

a) Procédure d'exportation

```
ALTER PROCEDURE [POS].[ExportCustomObjects]
    @rowVersion bigint,
    @companyUnitId int,
    @pointOfSaleId int,
```

```
    @languageId int
AS
BEGIN
                                    SET NOCOUNT ON;
    declare @dbts bigint = cast(@@DBTS as bigint)
    select
            @dbts as [@RowVersion],

        (select
            pf.Id                       as [@Id],
            pf.Name             as [@Name],
            pf.CategoryId               as [@Type],
            POS.GetBitString(pf.Active)
                        as [@IsActive]
        from SecDictionaries.Dic_PaymentForms pf
        where pf.Timestamp > @rowVersion
        for xml path('PaymentForm'), type),
        (select
                        c.Id            as [@Id],
            c.Code              as [@Code],
            c.Name              as [@Name],
            POS.GetBitString(c.Active)
                    as [@IsActive]
        from dbo.Dic_Country c
        where c.Timestamp > @rowVersion
        for xml path('Country'), type)
    for xml path('CustomObjects')
END
```

b) Procédure d’importation

```
ALTER PROCEDURE [Synchronization].[ImportCustomObjects]
    @XML xml
AS
BEGIN
    SET NOCOUNT ON;
    -- Countries --
    select
        doc.col.value('@Id', 'int') Id,
        doc.col.value('@Code', 'nvarchar(50)') Code,
        doc.col.value('@Name', 'nvarchar(100)') Name,
        doc.col.value('@IsActive', 'bit') IsActive
```

```
    into #Countries
      from @XML.nodes('/DataFromERP/CustomObjects/Country')
doc(col)

update pos
    set
        Code = erp.Code,
        Name = erp.Name,
        IsActive = erp.IsActive
    from Configuration.CustomCountries pos
        join #Countries erp on erp.Id = pos.Id

    insert into Configuration.CustomCountries
    (
        Id,
        Code,
        Name,
        IsActive
    )
    select
        Id,
        Code,
        Name,
        IsActive
    from #Countries erp
             where   not   exists   (select   1   from
Configuration.CustomCountries where Id = erp.Id)

    -- Payment forms --
    select
        doc.col.value('@Id', 'int') Id,
        doc.col.value('@Name', 'nvarchar(50)') Name,
        doc.col.value('@Type', 'tinyint') [Type],
        doc.col.value('@IsActive', 'bit') IsActive
    into #PaymentForms
     from @XML.nodes('/DataFromERP/CustomObjects/PaymentForm')
doc(col)

    update pos
    set
        Name = erp.Name,
```

```
        Type = erp.Type,
        IsActive = erp.IsActive
    from Configuration.CustomPaymentForms pos
        join #PaymentForms erp on erp.Id = pos.Id

    insert into Configuration.CustomPaymentForms
    (
        Id,
        Name,
        Type,
        IsActive
    )
    select
        Id,
        Name,
        Type,
        IsActive
    from #PaymentForms erp
            where not exists (select 1 from
Configuration.CustomPaymentForms where Id = erp.Id)
END
```

Gestion des répertoires universels

Mise à jour de la procédure d’exportation

Dans la clause **WHERE** de la procédure **POS.ExportGenericDirectories** il faut inclure **InternalName** de répertoire qui doit être également synchronisé.

Dans la clause **WHERE** de la procédure **POS.ExportGenericDirectoryValues** il faut inclure **InternalName** de répertoire qui doit être également synchronisé.

Clés étrangères pour le schéma Standard

Pour les tableaux synchronisés

La disponibilité des objets, les autorisations etc. peuvent, au niveau de la ligne d'un tableau donné, affecter les données qui seront synchronisées. Un exemple serait les Clients, groupes clients, entrepôts, registres, modes de paiement etc.

Afin d'éviter une évaluation multiple, il faut utiliser le tableau **POS.SentObjects** pour savoir quelles données doivent être synchronisées dans les tableaux dérivées (sur la base de FC).

Exemple : Exportation des définitions d'impôts associés à un client

```
(select
        ven.Id as [@Id],
        ven.ActivityId as [@ActivityId],
        ven.VendorId as [@VendorId]
         from Implementations.VendorActivityConnectionsEcoTax
ven
          inner join Implementations.ActivityEcoTax ac on
ven.ActivityId = ac.Id
         inner join Implementations.SettingsEcoTax sett on
ac.Id = sett.CompanyActivityId and sett.CompanyId =
@companyUnitId
         inner join POS.SentObjects so on so.ObjectId =
ven.VendorId and so.SyncTypeId = 14 and so.POSId =
@pointOfSaleId
        where ven.Timestamp > @rowVersion
                                                  for xml
path('VendorActivityConnectionsEcoTax'), type)
```

La valeur du type d'objet synchronisé peut être consultée dans le tableau **POS.SyncTypes.**

Pour les tableaux non-synchronisés

L'utilisateur doit lui-même implémenter le processus de synchronisation, comme si le tableau provenait d'une extension.

Suppression des données

Actions du côté de Comarch ERP Standard

1. Il faut ajouter dans le tableau **DeletionTypes** l'entrée avec « Id >= 1000 » et le nom unique du type des objets supprimés.
2. Dans le déclencheur AFTER DELETE du tableau à partir duquel la suppression des données doit être synchronisée avec POS, ajoutez des entrées concernant les objets supprimés à la table POS.DeletedObjects. Colonnes à compléter :
    - **DeletionTypeId** – identifiant du type défini dans le point a)
    - **Ident** – identifiant d'objet supprimé. Un identifiant peut être un nombre (int), un GUID (uniqueidentifier), nvarchar ou un ensemble de valeurs séparées par le caractère « | », par exemple : „3428|654". Voir aussi le point concernant les actions du côté de POS (colonnes IdentColumnName, IdentColumnType).
    - **POSId** – identifiant POS (Synchronization.PointsOfSales) – il faut le compléter, si l'objet doit être supprimé d'un point de vente POS particulier ou laisser NULL s'il doit être supprimé de tous les points de vente.

Actions du côte de POS

Ajoutez au tableau **Synchronization.DeletionTypes** une ligne définissant le moyen de gestion par le mécanisme de synchronisation d'information sur la suppression d'objets provenant d'ERP.

Il existe deux modes de suppression : **automatique** et **personnalisé**.

- En mode **automatique**, le mécanisme de synchronisation supprimera automatiquement les données du tableau indiqué en identifiant les lignes à partir des noms des colonnes (les valeurs doivent être complétées dans les colonnes TableName, IdentColumnName, IdentColumnType, ainsi que la valeur NULL dans la colonne CustomProcName). Ce mode est utilisé pour supprimer dans POS presque tous les types d'objets supprimables dans l'ERP (voir les entrées standards dans le tableau Synchronization.DeletionTypes).
- Le mode **personnalisé** exige que la procédure prenant en charge la suppression des objets du type donné (la colonne CustomProcName doit être complétée) soit indiquée. Ce mode est utilisé lorsque la condition de suppression est plus complèxe et il n'est pas possible d'appliquer le mécanisme automatiue ou lorsqu'il faut effectuer des opérations supplémentaires lors de la suppression.

Le tableau **Synchronization.DeletionTypes** contient les colonnes suivantes :

- **DelType** – nom comme dans la base Standard dans le tableau **DeletionTypes.**
- **Order** – nombre définissant l'ordre de suppression des types d'objets.
- **TableName** [uniquement en mode automatique] – nom du tableau dont les objets doivent être automatiquement supprimés dans le cadre du type donné DelType.
- **IdentColumnName** [uniquement en mode automatique] – noms des colonnes (séparés par le caractère « | ») selon lesquels se fait l'identification automatique des lignes supprimées, par exemple : „Id", „GUID", „PriceTypeId|CustomerGroupId". Leur quantité et ordre doivent correspondre à la valeur saisie dans la base Standard dans la colonne POS.DeletedObjects.Ident.
- **IdentColumnType** [uniquement en mode automatique] – types

des colonnes définis comme IdentColumnName dans la même quantité et ordre, par exemple „int”, „uniqueidentifier”, „int|int”. Dans le cas de nvarchar(x), il faut saisir « nvarchar ».

- **CustomProcName** [uniquement en mode personnalisé] – nom de procédure responsable pour la suppression des objets d’un type donné. Cette colonne doit utiliser les données du tableau temporaire #DeletedObjects. Voir les procédures existantes Synchronization.DeleteCustomerPriceTypes, Synchronization.DeleteWarehouseDocuments.

Extension de la synchronisation des données avec POS – option 1

Exemple d’une procédure d’exportation

```
IF  EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[Synchronization].[GetCustomData]') AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
DROP FUNCTION [Synchronization].[GetCustomData]
GO

CREATE FUNCTION [Synchronization].[GetCustomData]
(
    @syncType int,
    @documentId int
)
RETURNS XML
```

```
AS
BEGIN

    declare @data XML;
                        set      @data      =      (select
[Implementations].[GetSpecificData](@syncType, @documentId)
                       for xml path('SpecificElements'),
root('CustomData'), type)
    return @data

END
GO

IF   EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[Implementations].[GetSpecificData]') AND type in
(N'FN', N'IF', N'TF', N'FS', N'FT'))
DROP FUNCTION [Implementations].[GetSpecificData]
GO

CREATE FUNCTION [Implementations].[GetSpecificData]
(
    @syncType int,
    @documentId int
)
RETURNS XML
AS
BEGIN

    declare @specifics XML;

    set @specifics = (select
                    el.OrdinalNumber as [@OrdinalNumber],
el.SpecificCode          as [@SpecificCode],
                    el. SpecificTypeId   as [@SpecificTypeId]
                    from ExtensionSchema.SpecificDataTable el
                    inner join Documents.TradeDocuments doc on
el.DocumentId = doc.Id
                      where el.DocumentId = @documentId and
@syncType = 45
                    for xml path('row'))
    return @specifics
```

```
END
GO
```

Exportation des données de POS

Il est possible d'ajouter des données personnalisées aux objets créés dans POS et synchronisés avec le système ERP. À ces fins, il faut remplacer la fonction **Synchronization.GetCustomData** dans la base POS. Cette fonction retourne **XML** et accepte le type d'objet synchronisé (**int**), ainsi que son identifiant (**int**) comme les arguments. Elle est démarrée séparément pour chaque objet qui doit être envoyé au système ERP.

Importation du côté de DataService

L'importation des données s'effectue dans le code C#. Chaque objet traité possède la propriété **CustomData** du type **XElement**. Les données doivent être désérialisées et traitées par l'utilisateur.

Les informations utiles peuvent être téléchargées de la classe statique **WebServiceHelper**. Il est possible d'obtenir les informations sur l'instance de POS pour chaque appel de la méthode du contrat de DataService :

– code de POS

– GUID de POS

– code de profile

– version

Exemple d'importation sur

DataService

```
[DataServiceBusinessModule]
public static class Module
{
   [MethodInitializer]
   public static void Initialize()
   {
                              var    customerService    =
IoC.Container.Resolve<IDataCustomerExtensionPointService>();
          customerService.AfterSaveCustomerEvent  +=
CustomerServiceEx_AfterSaveCustomerEvent;
   }
                        private       static       void
CustomerServiceEx_AfterSaveCustomerEvent(object   sender,
DTOResultEventArgs<Queue.DTO.CustomerDTO, string, int> e)
   {
          Console.WriteLine("{0}:  {1}",  e.Argument,
e.EntityRow.CustomData.Name);
      var xe = e.EntityRow.CustomData; // XElement
   }
}
```

Extension de la synchronisation des données avec POS – option 2

Exportation sur POS

Il est possible de synchroniser les données non-associées aux objets créés dans POS et synchronisés avec le système ERP. À ces fins, il faut utiliser la procédure

Synchronization.ExportCustoms. Il ne faut pas oublier que la structure dans le corps de la procédure **<Customs><Custom>** doit être conservée, mais la structure sous le nœud **<Custom>** peut être librement modifiée.

Il est souhaitable que la structure des tableaux à partir desquels nous sélectionnons les données à télécharger soit sous forme d'arbre et que le tableau principal contient les colonnes suivantes :

- GUID – assure l'identification de l'objet entre POS et ERP
- Id – à la relation dans la structure
- Type – si nous voulons différencier les données sur DataService, d'autant plus s'il existe des extensions individuelles
- WasSentToERP – pour filtrer les données qui ont déjà été envoyées

Marquer les données envoyées

Il est possible d'utiliser une méthode standarde du marquage des objets envoyés pour marquer les objets personnalisés.

À ces fins, nous surchargeons la méthode du service **ISynchronizationRepository** (cette zone n'a pas de points de branchement)

```
public class SynchronizationRepositoryExt : SynchronizationRepository
{
   public override void MarkIfWasSentToERP(string xml)
   {
      base.MarkIfWasSentToERP(xml);

      using (var context = new POSDBContext())
      {
context.ExecuteProcedure("Synchronization.MarkSentCustomData", xml);
```

```
      }
   }
}
```

À l’intérieur, nous pouvons par exemple appeler une procédure qui marquera les objets envoyés.

```
CREATE PROCEDURE [Synchronization].[MarkSentCustomData]
    @p0 xml
AS
BEGIN
    UPDATE CustomSchema.CustomRootTable
    SET WasSentToERP = 1
                    Where     GUID     in     (SELECT
xmlData.Col.value('@GUID','varchar(max)')       FROM
@p0.nodes('/Customs/Custom') xmlData(Col))
END
GO
```

Exemple d’exportation

```
CREATE PROCEDURE [Synchronization].[ExportCustoms]
AS
BEGIN
    SET NOCOUNT ON;

    select
        pad.GUID                        as [@GUID],
        pad.Type as [@Type],
        pad.Data1               as [@Data1],
          Synchronization.GetDatetimeString(pad.ChangeDate)
as [@ChangeDate],
        (
            select
td.Number                       as [@NumberString],
td.Status                               as [@Status],
td.Value                                as [@Value]
                                                    from
CustomSchema.Table1 td                          where pad.Id =
td.RootId
            for xml path('Table1'), root('Tables1'), type
```

```
        ),
        (
            select
ti.ToPayNet                             as [@ToPayNet],
                ti.Points                                   as
[@Points],
ti.ToPay                                as [@ToPay]
            from                        CustomSchema.Table2 ti
where
pad.Id = ti.RootId
            for xml path('Table2'), root('Tables2'), type
        )

    from CustomSchema.RootData pad
    where pad.WasSentToERP = 0

    for xml path('Custom'), root('Customs')
END
GO
```

Importation du côté de DataService

L’importation des données se fait en utilisant le service **IDataCustomService** et en surchargeant la méthode **SaveCustom.** L’argument pour la méthode est chaque ligne **Custom** sous forme d’objet XElement**.**

Pour prendre en charge plusieurs extensions, il faut utiliser les points de branchement pour **DataService.**

Snippet pour l’importation

```
[DataServiceBusinessModule]
public static class Module
{
   [MethodInitializer]
   public static void Initialize()
   {
                            var    dataCustomService    =
IoC.Container.Resolve<IDataCustomExtensionPointService>();
```

```
            dataCustomService.OnSaveCustomEvent  += DataCustomService_OnSaveCustomEvent;
   }
    private static void DataCustomService_OnSaveCustomEvent (object sender, XEEventArgs e)
   {
   //désérialisation + enregistrement des données
   }
}
```

Actions de DataService personnalisées

Le contrat de DataService comporte deux méthodes permettant d'appeler une action universelle :

```
byte[] CustomGet(string operationCode, byte[] args)
void CustomExecute(string operationCode, byte[] args)
```

Tout comme le type retourné (uniquement **CustomGet**), l'argument est un tableau d'octets pour qu'il soit possible d'envoyer et de recevoir n'importe quelle structure.

Appel sur POS

Les méthodes pour appeler les actions universelles se trouvent dans le service **ISynchronizationService**. Il suffit d'implémenter dans son propre module une instance du service mentionné ci-dessus et appeler l'opération souhaitée.

Gestion sur DataService

Pour brancher la prise en charge d'une action universelle, il faut enregistrer sa gestion dans un module d'extension.

Attention
La classe dotée d’un attribut doit être statique

```
[DataServiceBusinessModule]
public static class Module
{
   [MethodInitializer]
   public static void Initialize()
   {
                          var    customOpsService    =
IoC.Container.Resolve<ICustomOperationsService>();
           customOpsService.RegisterCustomGet("my_op",
MyCustomGet);
   }
   private static byte[] MyCustomGet(byte[] data)
   {
     //code
   }
}
```

Points de branchement (POS extension points)

Le module **Comarch.POS.ExtensionPoints** permet d’étendre **à plusieurs reprises** la même méthode de viewmodel. Il est ainsi possible de créer plusieurs extensions indépendantes qui modifient le fonctionnement de la même fonction native.

Architecture de branchements

Un viewmodel héritier supplémentaire, ainsi qu’un service métier lui dédié ont été créés pour le viewmodel étendu. Par exemple :

```
public   class   DocumentViewModelExtensionPoint   :
```

```
DocumentViewModel
public class DocumentViewModelExtensionPointService :
PrintingViewModelNavigationExtensionPointService<IDocumentView
Model>,
  IDocumentViewModelExtensionPointService
  IDocumentViewModelExtensionPointInternalService
```

Chaque méthode étendue de viewmodel possède deux événements « à brancher » dans le service : Before et After. Une exception sont les activateurs, c'est-à-dire les méthodes qui retournent le drapeau boolean. Un seul événement existe pour eux.

En élargissant une méthode particulière de viewmodel nous accédons aux paramètres de l'appel donné, ainsi qu'à l'instance du viewmodel donné. Le but est de partager le contexte entier de l'opération.

Attention
L'événement Before<nom_de_méthode> permet de signaler l'arrêt d'appel d'un appel standard. Il ne faut pas oublier qu'il est possible que dans un environnement avec plusieurs extensions, plusieurs extensions seront branchées à un seul **événement**. Chaque appel suivant sera basé sur les résultats de l'appel précédent. Il ne faut pas présumer qu'au début d'appel la valeur Cancel sera définie sur **false**. En outre, il faut présumer que l'ordre d'appel des extensions est non-déterministe.

Exemple :

```
[Dependency]
public       IPaymentViewModelExtensionPointService
PaymentViewModelExtensionPointService
{ get; set; }

public override void Initialize()
{
PaymentViewModelExtensionPointService.BeforeAddToPaymentFormEv
ent +=
```

```
_paymentViewModelExtensionPointService_BeforeAddToPaymentFormE
vent;
}

private                                                    void
_paymentViewModelExtensionPointService_BeforeAddToPaymentFormE
vent(object                                              sender,
EntityViewModelCancelEventArgs<IPaymentViewModel,
PaymentFormRow> e)
{
            if (e.Cancel) // gestion d'état d'origine
                return;

            // code
        }
```

Appeler une méthode native de ViewModel

Il peut arriver que l’on veuille, par extension, faire dépendre l’appel d’une opération native de la décision de l’opérateur. Les messages sur POS bloquent la gestion et son traitement peut être continué uniquement dans un handler de retour de la boîte de dialogue avec le message.

```
private                                                    void
_documentViewModelExtensionPointService_BeforeProductAddEvent(
object                                                   sender,
EntityViewModelCancelEventArgs<IDocumentViewModel, LotRow> e)
{

   e.Cancel = true;

    MonitService.ShowQuestion("Voulez-vous ajouter le produit
?", (s, m) => {
       if (m.MonitResult == MonitResult.Yes) {
           var serv = DocumentViewModelExtensionPointService.
GetViewModelActions(e.ViewModel);
           serv.ProductAdd(e.EntityRow);
```

```
        }
    });
}
```

La méthode GetViewModelActions retourne les instances qui permettent d’appeler les méthodes natives du viewmodel de base en omettant l’extensibilité.

Branchements disponibles

En général, chaque extension de viewmodel permet de modifier les méthodes de navigation :

- OnActivated() – BeforeOnActivatedEvent, AfterOnActivatedEvent
- OnInitialization() – Before OnInitializationEvent, AfterOnInitializationEvent
- OnDeactivated() – BeforeOnDeactivatedEvent, AfterOnDeactivatedEvent

Viewmodels qui prennent en charge le processus d’impression ont de plus les extensions des méthodes :

- Print() – BeforePrintEvent, AfterPrintEvent
- CanPrint() – CanPrintEvent

Exemple d’implémentation

Ci-dessus, vous trouverez l’implémentation d’une extension finie qui fera afficher deux messages au moment d’enregistrement d’un document commercial – un message avant l’enregistrement et l’autre tout de suite après l’enregistrement.

```
using Microsoft.Practices.Unity;
using
Comarch.POS.ExtensionPoints.Presentation.Sales.ViewModels;
using Comarch.POS.Presentation.Core.Services;
using Comarch.POS.Presentation.Core;
```

```
public class Module : ModuleBase
{
   private readonly IUnityContainer _container;
   [Dependency]
   public IMonitService MonitService { get; set; }

   [Dependency]
       public  IDocumentViewModelExtensionPointService
DocumentViewModelExtensionPointService { get; set; }
   public Module(IUnityContainer container) : base(container)
   {
      _container = container;
   }

   public override void Initialize()
   {
       DocumentViewModelExtensionPointService.BeforeSaveEvent
+= _documentViewModelExtensionPointService_BeforeSaveEvent;
      DocumentViewModelExtensionPointService.AfterSaveEvent +=
_documentViewModelExtensionPointService_AfterSaveEvent;
   }

                                    private           void
_documentViewModelExtensionPointService_AfterSaveEvent(object
sender, GenerateViewModelEventArgs<IDocumentViewModel> e) {
      MonitService.ShowInformation("Extension: after save");
   }

                                    private           void
_documentViewModelExtensionPointService_BeforeSaveEvent(object
sender,
GenerateEntityViewModelCancelEventArgs<IDocumentViewModel>  e)
{
      if (e.Cancel)
         return;

      MonitService.ShowInformation("Extension: before save");
   }
}
```

Modules supplémentaires et leurs extensions

Les modules supplémentaires sont intégrés avec la logique de POS à l'aide des **événements.** À ces fins, des interfaces spéciales qui permettent la communication mentionnée ci-dessus ont été créées dans POS. Chaque interface possède son équivalent « interne ». L'interface de base définit les méthodes du service à communiquer avec les services externes. Le service « interne » définit les événements par le biais desquels s'effectue la communication.

L'interface de base est utilisée sur les viewmodel dans POS.

L'interface interne est utilisée dans les modules externes.

À présent, les interfaces suivantes existent :

IDeviceEventService et **IDeviceEventInternalService** (gestion du tiroir)

IDocumentEventService et **IDocumentEventInternalService** (fiscalisation des documents)

IFiscalPrinterEventService et **IFiscalPrinterEventInternalService** (impressions pour paiement électroniques sur l'imprimante fiscale)

IPaymentEventService et **IPaymentEventInternalService** (paiements électroniques)

ISessionEventService et **ISessionEventInternalService** (rapports fiscaux et rapports des paiements électroniques)

Module fiscal

Il existe deux interfaces de base qui permettent d'ajuster le fonctionnement du module fiscal aux besoins de l'utilisateur.

IFiscalizationService – contient toutes les méthodes participant dans la communication avec l'imprimante fiscale. En outre, il est également possible de contrôler les méthodes qui préparent les données à la fiscalisation (par exemple les éléments de document, paiements, adresse sur la facture)

ItemCustomizationService – permet de modifier tout champ qui est envoyé à l'imprimante fiscale.

Un aspect important de l'extension du module fiscal est le fait que la classe de base Module doit hériter de la classe Module du module fiscal et non pas de la classe **Comarch.POS.Presentation.Core**. Nous pouvons alors surcharger les méthodes supplémentaires (**RegisterServices**, **TriggerEventBinding**, **RegisterViewModels**, **RegisterViews**, **AddContainerElements**)

Modification du contrôleur Comarch.B2.Printer2

En outre, en cas de besoin il est possible, en héritant de la classe **PrinterManager**, surcharger chaque méthode.

Il ne faut pas oublier que la classe héritière doit également hériter de l'interface **IPrinterService.** Ceci est dû au fait que les contrôleurs sont chargés dynamiquement et l'instanciation se fait sur la base de l'interface appropriée.

```
public class MyPrinterManager : PrinterManager, IPrinterService { … }
```

Impression de document personnalisé

Il est possible d'imprimer un document personnalisé. Sur un viewmodel approprié il faut utiliser la méthode**PrintCustomDocument** du service **IDocumentEventService.**

Ensuite, vous pouvez soit

– en héritant de la classe **FiscalizationService** appeler sur l'instance interne **IPrinterService** une méthode d'impression des lignes en mode non-fiscal ((NonFiscalOpen, NonFiscalLinePrint, NonFiscalClose)

soit

– écrire un driver personnalisé (par exemple sur la base de Comarch.B2.Printer2), surcharger la méthode **PrintCustomDocument** et, en héritant de la classe **FiscalizationService,** surcharger la méthode **PrintCustomDocument**

Drapeaux supplémentaires

La propriété **FiscalParams** qui stocke les informations supplémentaires pour le module fiscal a été ajoutée dans la classe **TradeDocument.**

Autres – Bons d'achat

Protocole de communication pour les bons d'achat.

Méthodes Rest utilisées, contrat : Comarch.B2.DataService.Contracts.dll

VoucherEntity[] GetInternalVouchers(string numer)

La méthode doit retourner une liste actuelle des bons d’achat conformes au numéro indiqué (d’habitude ce n’est qu’un bon d’achat).

VoucherResult UpdateVouchers(VoucherEntity[] vouchers);

La méthode doit exécuter l’action d’ajout/d’activation/de désactivation/ de mise à jour des bons d’achat particuliers qui ont été transmis en tant que paramètres.

L’activité à réaliser pour des bons individuels dépend des données dans l’entité VoucherEntity. Données importantes transmises par POS :

VoucherEntity	
Id : int	Id de bon d’achat (0 s’il doit être créé)
TypeId : VoucherKindEnum	Type de bon d’achat (Unknown, InternalSold, InternalReleased, External, GiftCard)
SortId : int	Genre de bon d’achat
CurrencyId : int	Devise de bon d’achat
IsActive : bool	Statut de bon d’achat True – bon interne (vendu, émis ou carte) à utiliser (désactiver) ou, en cas de carte, à mettre à jour le statut Amount
Number : string	Numéro de bon d’achat
Amount : decimal	Montant de bon d’achat

Action prévue en fonction de la configuration des paramètres

VoucherEntity transmis pour prendre en charge les bons d’achat (DataService) à l’aide de la méthode **UpdateVouchers** :

Id	SortId	IsActive	Action
–	External	–	**Utilisation d’un bon externe** Enregistré dans la base comme utilisé et inactif dont le montant a été défini dans **Amount**, la devise dans **CurrencyId** et le numéro dans **Number**
0	InternalReleased	–	**Génération d’un bon d’achat interne émis** Enregistré dans la base comme actif et non-utilisé dont le montant a été défini dans **Amount**, lu numéro dans **Number** et la devise dans **CurrencyId**
>0	InternalReleased InternalSold GiftCard	true	**Utilisation de bon ou mise à jour du montant de carte cadeau**
>0	InternalReleased	false	**Activation d’un bon existant émis intérieurement**

Bool IsExternalVoucherExists(string numer, int sortId)

La méthode doit vérifier si un bon externe avec le nom indiqué et du type indiqué existe déjà dans la base.

Exemples

Une solution Visual Studio avec une liste complète des exemples divisés en projets séparés complète le sujet des extensions dans POS. Pour activer les exemples, vous pouvez construire toute la solution, copier les fichiers de résultat dans le dossier d'installation de POS et enregistrer un seul module initialisant **POSUsageExample.dll** (un guide d'enregistrement se trouve dans le fichier READEME.txt de la solution). Une alternative est de construire chaque projet avec une bibliothèque individuelle (en fonction de l'exemple que vous voulez analyser) et enregistrer uniquement cette bibliothèque dans l'application POS. Les projets sont divisés en trois catégories : exemples d'utilisation des contrôles, exemples des vues entières et exemples d'extension des vues existantes.

Exemples d'utilisation des contrôles POS

Exemple d'utilisation du contrôle ComboBox2

Exemples d'utilisation du contrôle avec une vue modale d'une liste de choix standard et avec une présentation personnalisée. Disponibles dans le projet **ComboBox2Example.**

Exemple d'utilisation du contrôle ButtonSpinner

Exemple d'utilisation du contrôle avec un TextBox pour contrôler les valeurs numériques saisies par l'utilisateur. Disponible dans le projet **ButtonSpinnerExample.**

Exemple d'utilisation du contrôle ComboBoxButton

Disponible dans le projet **ComboBoxExample.**

Exemple d’utilisation du contrôle MultiButton

Exemple d’utilisation du contrôle avec sa gestion complète dans l’application. Disponible dans le projet **MultiButtonExample.**

Exemple d’utilisation du contrôle ItemsContainer

Exemples d’utilisation du contrôle avec sa gestion complète dans l’application. Le premier avec une définition des éléments dans xaml et le second avec un contenu dynamique construit dans le code de manière asynchrone. Disponible dans le projet **ItemsContainerExample.**

Exemple d’utilisation du contrôle Grid

Exemple de construction d’une vue parfaitement gérable sur la base de Grid. Disponible dans le projet **GridExample.**

Exemple d’utilisation du contrôle FieldControl

Exemples d’utilisation du contrôle avec gestion complète et prise en charge de la validation. Disponible dans le projet **FieldControlExample.**

Exemple d’utilisation des contrôles TabControl et TabControlItem

Exemples d’utilisation des contrôles TabControl et TabControlItem pour créer des onglets sur une vue. Disponible dans le projet **TabControlExample.**

Exemple d’utilisation du contrôle DatePicker2

Un exemple d’utilisation du contrôle avec la validation se trouve dans le projet **DatePicker2Example.**

Exemples de la création des vues

Module simple avec une nouvelle vue vide

L’exemple montre comment créer des modules d’extension pour POS. Disponible dans le projet **EmptyViewExample.** Il comporte une classe Module permettant d’enregistrer le module et une vue vide (SimpleView, SimpleViewModel), ainsi que le mode de gestion de l’interface (DesignSimpleViewModel). La vue a été enregistrée sous forme d’une mosaïque dans le menu principal de l’application POS.

Vue typique de liste des ventes

L’exemple montre comment construire une vue typique avec une liste téléchargeant des données de manière asynchrone, prenant en charge le tri et la pagination, avec un moteur de recherche et des filtres. Cette vue contient une classe Module (responsable pour l’enregistrement du module d’extension et de la vue sous forme d’une mosaïque dans le menu principal de l’application POS), des classes de la vue et du viewmodel de la liste – SimpleListView, SimpleListViewModel i DesignSimpleListViewModel. Exemple disponible dans le projet **DataGridCompleteExample.**

Vue typique de document commercial

L’exemple montre comment construire une vue typique de document commercial contenant un DataGrid et un moteur de recherche SearchBox. Disponible dans le projet **DocumentExample.**

Vue d’un document commercial avec la gestion des attributs

Un exemple de construction de la vue de document commercial enrichi de la gestion des attributs pour la liste et sous forme des contrôles générés dynamiquement dans le conteneur de vue. Disponible dans le projet **DocumentAttributesExample.**

Exemples d’extensions des vues existantes dans POS

Ajouter un contrôle au conteneur de vue existante

Le projet **ControlExtensionsExamples** contient un exemple présenté dans ce document et un autre qui montre comment ajouter des boutons au conteneur ItemsContainer, ainsi que sur le Grid de la vue existante créée dans le même projet.

Ajouter une colonne DataGrid sur une vue existante

L’exemple complet de l’extension des colonnes de DataGrid a été présenté dans le projet **DataGridColumnExtensionExample**.

Exemple d’implémentation du moyen d’agrégation de données personnalisé dans DataGrid

Une implémentation exemplaire d’agrégation sous forme d’une médiane a été présentée dans le projet **DataGridAggregationExample.**

Exemple d’extension d’une zone de statut

Le projet **StartBarExtensionExample** ajoute deux boutons à la zone de statut. Un qui est un raccourci pour ouvrir un nouveau document commercial et l’autre utilisant le contrôle ComboBoxButton.