REF : SENT-TEC

BDO Sentinel

Technique (Audit Architecture)

Architecture

****

Description de l’architecture

Version – 2.0

Table des matières

[A. Présentation 5](#_Toc475203134)

[A.1. But du document 5](#_Toc475203135)

[A.2. Description du projet 5](#_Toc475203136)

[B. SOLUTION TECHNIQUE 7](#_Toc475203137)

[B.1. Choix effectués 7](#_Toc475203138)

[B.2. Solution 8](#_Toc475203139)

[B.2.a. SHELL 9](#_Toc475203140)

[B.2.b. MODULES 9](#_Toc475203141)

[B.2.c. INFRASTRUCTURE 9](#_Toc475203142)

[B.2.d. SERVICES 10](#_Toc475203143)

[B.2.e. Référencement entre projets 10](#_Toc475203144)

[B.3. PRISM 12](#_Toc475203145)

[B.3.a. Initialisation 12](#_Toc475203146)

[B.3.b. Modularité 13](#_Toc475203147)

[B.3.c. MVVM 13](#_Toc475203148)

[B.3.d. Interface utilisateur 13](#_Toc475203149)

[B.3.e. Navigation 13](#_Toc475203150)

[B.3.f. Communication 13](#_Toc475203151)

[B.4. Accès aux données 13](#_Toc475203152)

[A. Schéma de l’architecture 14](#_Toc475203153)

[A.1. Vue générale 14](#_Toc475203154)

[A.1. Shell (Interface utilisateur) 14](#_Toc475203155)

[A.2. Les modules 15](#_Toc475203156)

[A.3. Les Services 15](#_Toc475203157)

[A.3.a. ScenarioService 16](#_Toc475203158)

[A.4. Les projets Infrastructure 16](#_Toc475203159)

[A.4.a. Projet DataAccess 16](#_Toc475203160)

[1. Projet Service.DataAccess.Interfaces 17](#_Toc475203161)

[2. Projet Sentinel.DataAccess 17](#_Toc475203162)

[A.4.b. Projet Common 18](#_Toc475203163)

[A.5. Connexion 19](#_Toc475203164)

[A.6. Exécution d’un événement (Scenario) 20](#_Toc475203165)

[B. Environnement Infrastructure 22](#_Toc475203166)

[B.1. Outils de développement 22](#_Toc475203167)

[B.1.a. SQL Server 2012 22](#_Toc475203168)

[C. Documentation technique 22](#_Toc475203169)

[C.1. Organisation d’un appel DB 22](#_Toc475203170)

Gestion des versions

|  |  |  |  |
| --- | --- | --- | --- |
| Numéro | Emetteur | Description | Date |
| V1.0 | MBU | Rédaction première version du document | 16/02/2017 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Création du document par Michaël Buisson, le 16/02/2017 10:00

Dernière sauvegarde effectuée par Michael Buisson, le 16/02/2017 10:46

# Présentation

## But du document

Ce document représente une base de travail pour un audit externe de l’architecture logicielle mise en place.

Par audit, nous entendons la validation « théorique » de la solution proposée face aux « Best Practices ».

Nous présentons ici les principes retenus pour la mise en place de notre projet et de son architecture.

Nous n’entrerons pas dans les détails « métiers » mais des extraits de code seront repris pour illustrer les méthodes appliquées pour l’implémentation de la solution.

**Ce que ce document n’est pas :**

Un document détaillé reprenant la description des besoins métiers et amenant à la décision de certains choix techniques.

Il n’est pas non plus un document complet sur l’architecture logicielle et matérielle appliquée à chaque processus.

## Vue globale du projet

BDO Luxembourg souhaite remplacer une application existante, développée en Access, permettant aux utilisateurs de notamment réaliser un suivi de tâches liées à leur métier, de renseigner des données et d’effectuer diverses recherches et contrôles.

Ce nouveau produit (nom de code Sentinel) devra assister l’utilisateur au cours des divers processus, être simple, graphique et convivial, et paramétrable aux besoins du métier.

Il sera du type client lourd avec une base de données relationnelle SQL server. Son déploiement devra se faire de façon aisée sur les postes clients et la première installation ne devra pas dépasser la ½ journée.

Les performances (temps d’affichage, de réaction, d’exécution de l’application) seront primordiales.

La solution à développer peut être découpée suivant 8 grands axes (piliers).

Suivi des

tâches

Ecrans

standards

Ecrans

dynamiques

Scenarii

Contrôle

croisé

Générateur

de

requêtes

Dashboards

Sécurité

applicative

* **Suivi des tâches** : Le suivi a pour objectif de fournir à l’utilisateur une trame à suivre afin de compléter certains processus.
* **Ecrans standards** : Il s’agit des écrans à structure fixe. La structure de ces écrans est définie dans le cahier des charges et n’évoluera pas en fonction du paramétrage applicatif de l’administrateur fonctionnel.
* **Ecrans dynamiques** : Les écrans dynamiques sont des écrans à structures définies par l’administrateur fonctionnel. Le logiciel proposera une interface permettant à l’utilisateur de définir des écrans. La définition de la structure de ces écrans sera stockée en base de données.
* **Scenarii** : Un scenario correspond à l’exécution par le système d’une ou plusieurs actions paramétrées en fonction d’un événement déclenché dans l’application.
* **Contrôle croisé** : Le contrôle croisé consiste à vérifier la présence d’une donnée de la solution, dans une source de données externe.
* **Générateur de requêtes** : Le générateur de requête va permettre d’offrir à l’utilisateur une interface d’analyse des données.
* **Dashboard** : Un tableau de bord consiste en une représentation graphique des données retournées par une requête.
* **Sécurité applicative** : Le logiciel pourra gérer des données sensibles qu’il convient de protéger au mieux.

Ces piliers seront accompagnés de vues qui pourront interagir entre elles sans toutefois être forcément fortement liées.

La présentation des informations sera faite dans des zones, pour certaines redimensionnables, sur la base du « design » suivant :



Menus



Bloc 1

(Gauche)



Bloc 2

(Centre)



Bloc 3

(Droite)



Barre d’état



Barre de titre

Certaines vues seront affichées dans un bloc fixe alors que d’autres pourront l’être indifféremment dans un des blocs.

La fenêtre principale pourra conduire à l’ouverture de multiples fenêtres annexes ayant le même design.



# SOLUTION TECHNIQUE

## Choix effectués

Pour répondre aux différents besoins de l’application (desktop, modularité, design moderne, dynamisme des vues à afficher, navigation, communication entre les vues…), il a été décidé de s’orienter vers la technologie WPF en utilisant le framework PRISM et le pattern MVVM.

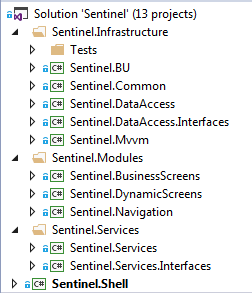
Version des outils

|  |  |
| --- | --- |
| Outils | Version |
| Visual Studio | 2015 |
| Team Foundation Services | 2010 |
| SQL server (standard edition) | 2012 |
| WPF | 4.5.2 |
|  |  |

|  |  |
| --- | --- |
| Librairies NuGet | Version |
| EntityFramework | 6.1.3 |
| EntityFramework.CodeTemplates.CSharp | 6.1.3 |
| Prism.Core | 6.2.0 |
| Prism.Unity | 6.2.0 |
| Prism.Wpf | 6.2.0 |
| CommonServiceLocator | 1.3.0 |
| Unity | 4.0.1 |
| T4 Editor | 2.3.0 |

## Organisation du code

Voici le code tel qu’organisé actuellement dans Visual Studio.



On peut distinguer 4 grands groupes :

* **Infrastructure**: Regroupe des fonctionnalités techniques (helpers, moteurs, accès données, …).
* **Modules** : Regroupement des vues et logiques de l’interface utilisateur.
* **Services** : Ensemble de fonctionnalités transverses à l’application.
* **Shell** : Interface utilisateur.

Liens entre ces groupes

MODULES

SHELL

INFRASTRUCTURE

SERVICES

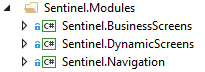
### SHELL



Point d’entrée de l’application, il initialise les différents composants nécessaires.

Le shell fournit également le socle de l’interface graphique (fenêtre principale MainWindow) et ses différentes régions.

### MODULES



**Business screens**

Traitement des écrans métier standards.

Ils seront dispatchés dans les 3 blocs de la fenêtre principale.

**DynamicScreens**

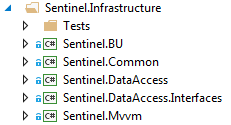
Traitement des écrans dynamiques (paramétrés spécifiquement pour le métier).

Ils seront dispatchés dans les 3 blocs de la fenêtre principale.

**Navigation**

Traitement des différents écrans de menu affiché dans la zone « Menu » de la fenêtre principale.

### INFRASTRUCTURE



**BU**

Logique métier et moteurs des différents processus.

**Common**

Regroupe un ensemble d’objets utiles pour la communication entre les modules et également toutes les énumérations, constantes, helpers et autres ressources statiques de l’application.

**DataAccess** / **DataAccess.Interfaces**

Centralise toutes les classes en lien avec la base de données (entités, opérations etc…)

### SERVICES



**Services / Services.Interfaces**

Ensemble de fonctionnalités transverses à toute l’application (logs, sécurité, audit, intermédiaire vers la base de données, etc…).

Ils sont injectés dans les modules via le principe des conteneurs « Unity ».

### Référencement entre projets



## PRISM

### Initialisation

* Bootstrapper

Dans le projet Shell nous créons la classe « BootStrapper.cs » qui va initier l’initialisation de l’application (l’appel est effectué depuis le code behind de App.xaml).



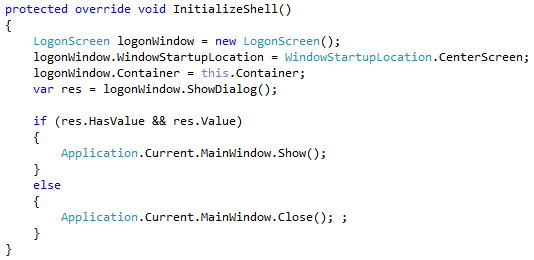
L’héritage de « UnityBootStrapper » va permettre l’utilisation de conteneurs « Unity » pour l’injection de dépendance.

* Récupération de l’instance de la fenêtre principale dans le conteneur.

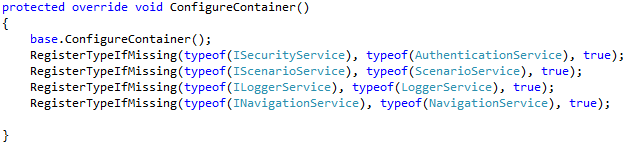


* Initialisation de la fenêtre principale.

On y intègre la notion d’écran d’authentification préalable à l’affichage.

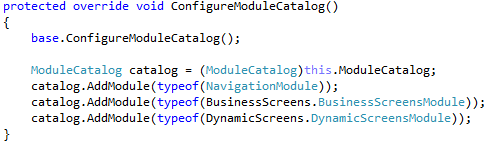


* Configuration du conteneur en procédant notamment au chargement de tous les services utiles à l’application via leur interface.



* Configuration du catalogue des modules nécessaires à l’application.

*Actuellement les modules sont défini dans la classe directement, il n’est pas exclu qu’ils le soient dans un fichier de configuration annexe pour plus de flexibilité.*



### Modularité

Chaque module de l’application est représenté par un projet contenant :

* Le point d’entrée et définition de ce module
* L’ensemble des vues inhérentes au module

Le point d’entrée est une classe centrale qui doit implémenter l’interface « PRISM » IModule.



C’est cette classe qui est ajoutée au catalogue des modules (cf. chapitre précédent).

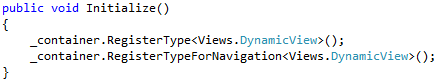
* Construction du module

On demande à « PRISM » de nous fournir le conteneur « Unity » transverse à l’application ainsi que le gestionnaire des régions (cf. points suivants).

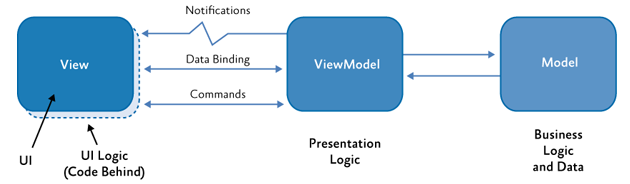


* Initialisation du module.

On charge les vues utilisées dans le conteneur et on les enregistre comme pouvant être affichées dans les différentes régions de l’application (principe de navigation).



### MVVM



* VIEW

Le système de vue est présent à la fois dans le Shell et dans les menus.

Le référencement du ViewModel par la Vue est automatiquement résolu grâce à la commande suivante dans le fichier XAML :



* VIEWMODEL

Tous les ViewModel liés à des modules héritent de la classe « SentinelViewModel » qui fournit des commandes et méthodes communes.





Les classes ViewModel échangent avec les Vues grâce à des propriétés et commandes qui sont « Bindés » sur les Vues.

Ces classes héritent de la classe abstraite « PRISM » BindableBase qui fournit une implémentation de l’interface InotifyPropertyChanged pour le binding des propriétés.



* MODEL

La partie Model est représentée par toutes les classes qui peuvent être impliquées dans le traitement et la validation des données. (Common, Services, Entités, DataAccess,…).

* BINDING

Comme énoncé ci-avant, il peut être effectué via des propriétés.

Toutes ces propriétés appellent l’implémentation de la méthode SetProperty (provenant de BindableBase)



Lorsque la propriété est du type collection, celle-ci doit être du type ObservableCollection<T>. Ce type a la particularité d’implémenter l’interface INotifyCollectionChanged qui permet de notifier la vue que le contenu de la liste a été modifié (ajout et suppression d’éléments, modification d’une valeur).



La source du binding est par défaut le datacontext de la Vue qui est dans notre cas le ViewModel.

On résout la propriété à binder dans la Vue via le mot clé « Path ».



Si l’on souhaite afficher le contenu d’une collection on peut utiliser des contrôles de type ItemsControl. Dans ce cas, le ViewModel ne sera plus la source du binding qu’il faut redéfinir via l’ItemsSource.



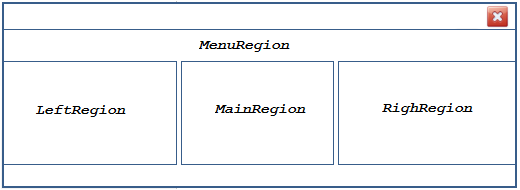
Dans un ItemsControl, il faut définir un modèle d’affichage des données. Pour chaque répétition du modèle, le bidning se fera sur l’élément courant de la collection.

S’il s’agit d’un objet, on précisera la propriété de cet objet via le mot clé « Path » comme décrit ci-avant.

### Interface utilisateur et navigation

* Définition des régions

L’interface est composée de 4 régions



Elles sont définies dans le code XAML de la fenêtre principale



* RegionManager

Afin de gérer les régions et la navigation avec PRISM, il faut utiliser une instance de la classe RegionManager fournie par PRISM.

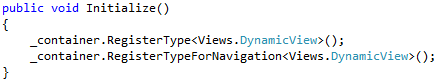
Elle peut être obtenue, dans le XAML, via la commande ci-avant énoncée et dans les modules via l’injection de dépendance.



* “View-based” navigation

La navigation à l’intérieur de l’application sera faite par le remplacement de Vues dans les différentes régions (principe de « view-based navigation » dans PRISM).

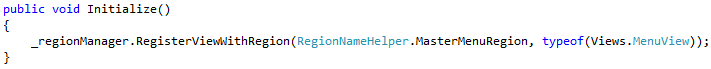
Dans chaque module, lorsque l’on charge les Vues dans le conteneur, on les enregistre également comme étant utilisées dans la navigation.



On utilise deux méthodes pour gérer l’affichage des vues dans les régions.

Le principe du « View discovery » qui va placer automatiquement la Vue dans une région lorsqu’elle est créée.

Cette définition est placée dans l’initialisation du module.



La méthode « View Injection » où l’on gère cet affichage dans le code au besoin.



Pour libérer une région de sa vue sans la remplacer, on peut utiliser



### Communication

* Event Aggregator

Ce service de PRISM permet la communication entre les différents modules de l’application, via un mécanisme de publication / souscription a des événements, sans que ceux-ci soient fortement liés.

Il peut être obtenu via l’injection de dépendance



Pour la création d’un événement spécifique à notre application, on crée une classe ayant pour classe de base « PubSubEvent<TPayLoad> » (où TPayLoad est le type du paramètre que l’on passera via l’événement).





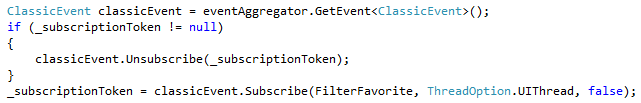
La publication de l’événement s’effectue en récupérant celui-ci et en le publiant via le service EventAggregator.





La souscription s’effectue via la méthode Subscribe de notre événement que l’on récupère au préalable.



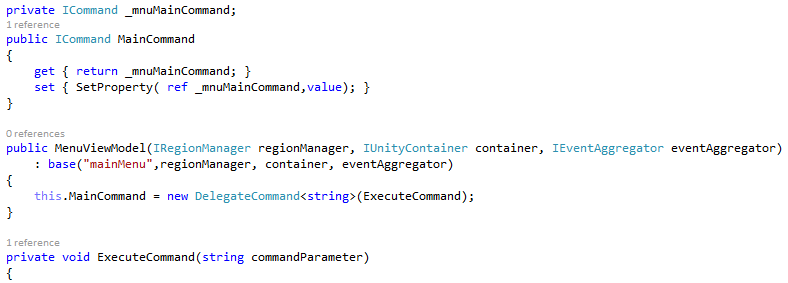


Dans notre code, on garde une trace de la souscription via son token. Par sécurité, annule potentielle souscription existante avant de créer la nouvelle.

L’utilisation de l’option ThreadOption.UIThread autorise le souscripteur à mettre à jour directement les éléments de l’interface utilisateur.

* Delegate command

On l’utilise pour la communication entre le ViewModel et la Vue

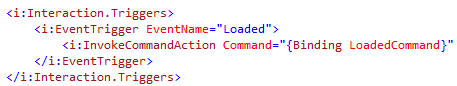


Dans la Vue

Si le control implémente l’interface ICommandSource

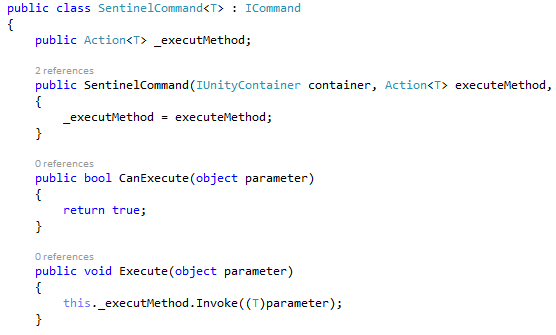


Sinon on utilise les « interaction triggers »



* Custom command



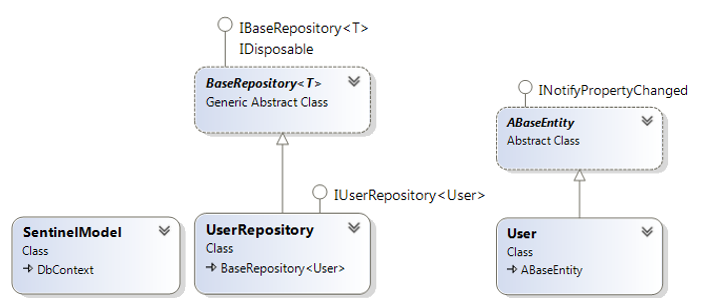


## Accès aux données

Pour la couche d’accès aux données, le choix s’est porté sur l’ORM Entity Framework.

L’approche DBFirst est utilisée et va générer automatiquement les composants (entités, repository, contexte) en fonction des tables de la base de données, via notamment des templates T4 .

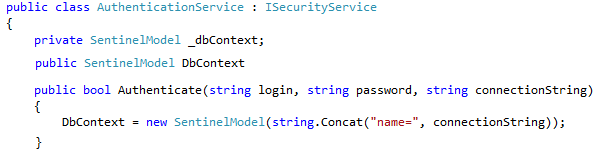
Le pattern repository est mis en place pour créer une couche d’abstraction entre les accès aux données et la logique business.



**SentinelModel** est la classe « contexte » qui expose les données de la DB sous forme de collection d’entités et définie les SP utilisées pour les opérations CRUD.



C’est le service d’authentification, premier instancié et injecté dans toute l’application, qui va fournir le contexte.



**Les entités** héritent toutes de la classe ABaseEntity qui implémente l’interface INotifyPropertyChanged afin de pouvoir utiliser le principe du binding sur leurs propriétés.

**Les classes Repository** sont créées pour chaque entité. Elles fournissent les opérations possibles sur les entités et utilisent le contexte pour interroger la base de données.



