Validation des Acquis et de l'Expérience

Master 2 Génie de l'Informatique Logicielle

Plateforme .NET et comparatif avec Java et C++

1.Objet

Ce mémoire propose de présenter la plateforme .NET. Nous explorerons ses possibilités au regard des grands enjeux actuels des techniques de développement.

Il ne s'agit pas d'exposer de façon exhaustive les syntaxes des langages associés à la plateforme. Nous nous focaliserons plutôt sur les fonctionnalités plébiscitées par les développeurs pour répondre à des problématiques telles que :

- l'environnement d'exécution,
- la manipulation et la persistance de données,
- les applications et services web,
- les clients lourds,
- les outils de développements.

Nous développerons par ailleurs, tout au long des fonctionnalités exposées, un comparatif avec les possibilités du monde Java et du langage C++.

La seconde moitié du mémoire concerne les technologies web, nous considérons que les exemples donnés sont hébergés par un conteneur adapté à la technologie illustrée (IIS* pour la plateforme .NET et un serveur d'application JEE, type Glassfish, pour Java).

2.Présentation de la plateforme .NET

Ses objectifs

La plateforme .Net est un composant Microsoft qui prend en charge la création et l'exécution de la nouvelle génération d'applications et de services Web. Elle est conçue pour remplir les objectifs suivants :

- Fournir un environnement cohérent de programmation orientée objet,
- Fournir un environnement d'exécution de code minimisant le déploiement de logiciel et les potentiels conflits de versions des composants applicatifs,
- Fournir un environnement qui promeut une exécution sécurisée du code y compris celui créé par un tiers d'un niveau de confiance moyen ou un tiers inconnu,
- Fournir un environnement d'exécution éliminant les problèmes de performance des environnements interprétés ou écrits en scripts,
- Fournir au développeur un environnement cohérent permettant de créer aussi bien des applications Windows que des applications Web,
- Générer toutes les communications à partir des normes d'industries pour s'assurer de l'intéropérabilité des applications .NET.

Spécifications et environnement d'exécution

En terme d'environnement d'exécution, la plateforme .NET offre un compilateur qui produit un code dit « intermédiaire » qui sera exécuté par une machine virtuelle. Il est important d'exposer les caractéristiques de ce code intermédiaire pour pouvoir exposer les briques composant la plateforme.

Le CIL*

Le CIL (Common Intermediate Language) est le code généré lors de la compilation du code source par les compilateurs .NET (conformes CLS*, voir paragraphe suivant sur la CLI*). Il est indépendant de la plateforme d'exécution et sera transformé (à la volée) en code machine lors de l'exécution par une machine virtuelle. Ce code est également appelé « Portable Executable* », il peut être décompilé afin d'être humainement lisible. Le CIL inclut des instructions pour le chargement, le stockage, l'initialisation et l'appel de méthodes sur des objets, ainsi que des instructions pour la réalisation d'opérations arithmétiques et logiques, le flux de contrôle, l'accès direct à la mémoire et la gestion des exceptions.

La CLI

La CLI (Common Language Infrastructure) est la spécification qui décrit, d'une part l'environnement d'exécution .NET et d'autre part les caractéristiques du code CIL (Common Intermediate Language). Ce code intermédiaire, produit lors de la compilation, est dit « managé », puisque son exécution sera assurée par la machine virtuelle .NET. Le CLR* (Common Language Runtime) de Microsoft est par exemple une des implémentations de la CLI.

La CLI aborde, entre autre, les aspects suivants :

- La CLS* (Common Language Specification) : Il s'agit de la spécification décrivant les caractéristiques d'un code utilisable par tous les langages .NET. La CLS décrit en réalité l'ensemble des fonctionnalités communes à tous les langages l'implémentant. Il est donc possible, au sein d'un langage respectant la CLS, de faire appel à du code écrit dans un autre langage respectant la spécification. Cette dernière définit donc les outils

(compilateurs) permettant de produire du code conforme CLS. Les compilateurs sont consommateurs conformes CLS lorsqu'ils permettent au langage d'utiliser des fonctionnalités exposées par des librairies conformes CLS. Les compilateurs sont extendeurs conformes CLS lorsqu'ils permettent aux développeurs à la fois d'utiliser et d'étendre les types définis dans les bibliothèques conformes CSL (c'est typiquement le cas de C# et de VB.NET).

- Le CTS* (Common Type System) : Il s'agit de la description des types de données utilisables par les langages respectant la spécification CLS. Le CTS prend en charge les cinq catégories de types suivantes : les classes, les interfaces, les structures, les énumérations et les déléguées.
- Les Metadata* : Ce sont les informations ajoutées dans le code CIL. Celui-ci est ainsi auto-porteur de sa propre description, ce qui lui confère son intéropérabilité. Les Metadata sont nécessaires à l'exécution du code et présentent entre autre :

    - la description de l'unité de déploiement (voir le chapitre sur les « Assembly »),
    - l'identité : nom, version, culture,
    - les types exportés,
    - les dépendances à d'autres unités de déploiements,
    - les autorisations de sécurité requises pour l'exécution,
    - les noms des classes de base, et des interfaces implémentées,
    - les membres (méthodes, champs, propriétés, événements, types imbriqués),

La CLI est donc garante de l'homogénéité et de l'intéropérabilité des langages de développement l'implémentant.

L'Assembly*

Le code CIL est stocké dans des assemblies. Un assembly est le bloc de structuration fondamental des applications .NET. Un assemblage regroupe l'ensemble des éléments nécessaires au bon fonctionnement d'une application (ou partie d'une application) : code intermédiaire, métadonnées, autorisations, ressources, images....

Le nom complet d'un assemblage (à ne pas confondre avec le nom du fichier sur le disque) contient son nom simple, son numéro de version, sa culture et sa clé publique. La clé publique est unique et est générée à partir du hachage de l'assemblage après sa compilation. En conséquence, deux assemblages avec la même clé publique sont garantis être identiques. Une clé privée peut aussi être spécifiée, elle est uniquement connue du créateur de l'assemblage et peut être utilisée pour générer le nommage fort de celui-ci. Cela garantit que l'assemblage est du même auteur lors de la compilation d'une nouvelle version.

Une assembly peut être globale au système d'exploitation ou locale à l'application qui l'utilise (dans la même arborescence de répertoires). Lorsqu'elle est globale, et donc potentiellement partagée par plusieurs applications, le GAC* (Global Assembly Cache) est chargé de les référencer. Chacune de ces assemblies y est fortement nommée (nom de l'assembly, numéro de version, langue, clé publique). Cela permet une exécution dite « side by side » ou plusieurs applications utilisent (et demandent donc le chargement en mémoire par la machine virtuelle) des versions différentes d'une assembly.

Le CLR

Le CLR (Common Language Runtime) est l'environnement d'exécution du code managé (machine virtuelle). Il gère, entre autre, la compilation JIT* (Just In Time), la mémoire, l'exécution des Threads, l'exécution et la vérification de la sécurité du code.

En ce qui concerne la sécurité, les composants managés se voient attribués divers niveaux de confiance en fonction de plusieurs facteurs, notamment leur origine (Internet, réseau d'entreprise, ordinateur local). Cela

signifie qu'un composant managé peut ou ne peut pas effectuer des opérations d'accès au fichier, au registre, au réseau ou d'autres fonctions délicates.

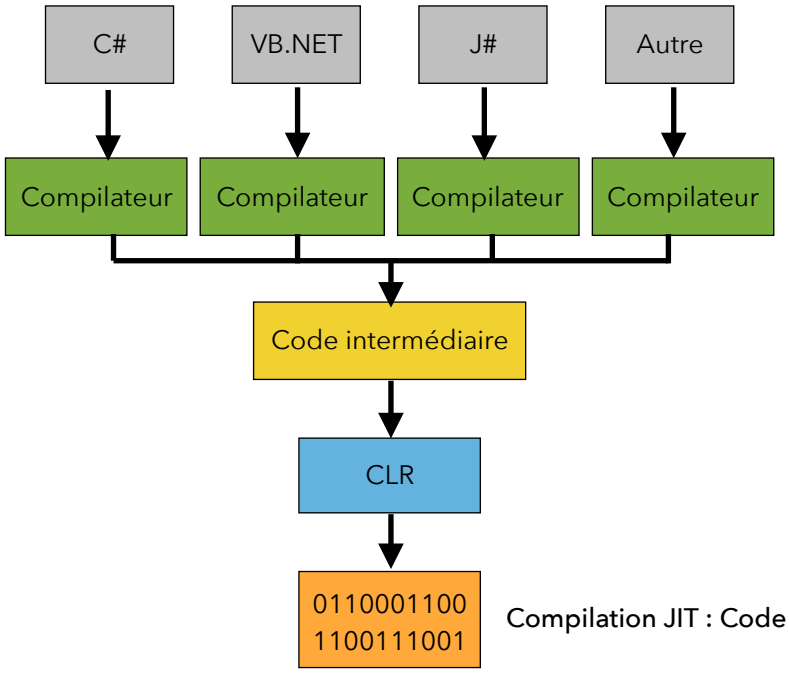
Les différents compilateurs de langages .NET génèrent du code conforme au système de type commun CTS. Cela signifie que le code managé peut consommer des instances de classe et des types de données quel que soit le langage qui les fournit. Le CLR facilite ainsi la conception de composants et d'applications dont les objets interagissent entre les langages. Les objets écrits dans des langages différents peuvent communiquer les uns avec les autres, et leurs comportements peuvent être fortement intégrés :

- les objets définis dans un composant écrit en C# peuvent tout à fait être utilisés dans une application VB.NET,
- les objets définis dans un composant écrit en C# peuvent tout à fait être étendus par des objets écrits en VB.NET.

Cette intéropérabilité est également possible grâce aux Metadata contenues dans les « Portable Executables ». Elles rendent le code auto-descriptif. Le CLR les utilise pour rechercher et charger des classes, placer des instances en mémoire, résoudre des appels de méthode, générer un code natif, appliquer la sécurité et définir les limites du contexte d'exécution.

En outre, l'environnement managé du runtime élimine un grand nombre de problèmes logiciels courants. Par exemple, le runtime traite automatiquement les références et la disposition des objets, les libérant lorsqu'ils ne sont plus utilisés. Cette gestion automatique de la mémoire résout les deux erreurs d'application les plus courantes, le manque de mémoire et les références mémoires non valides.

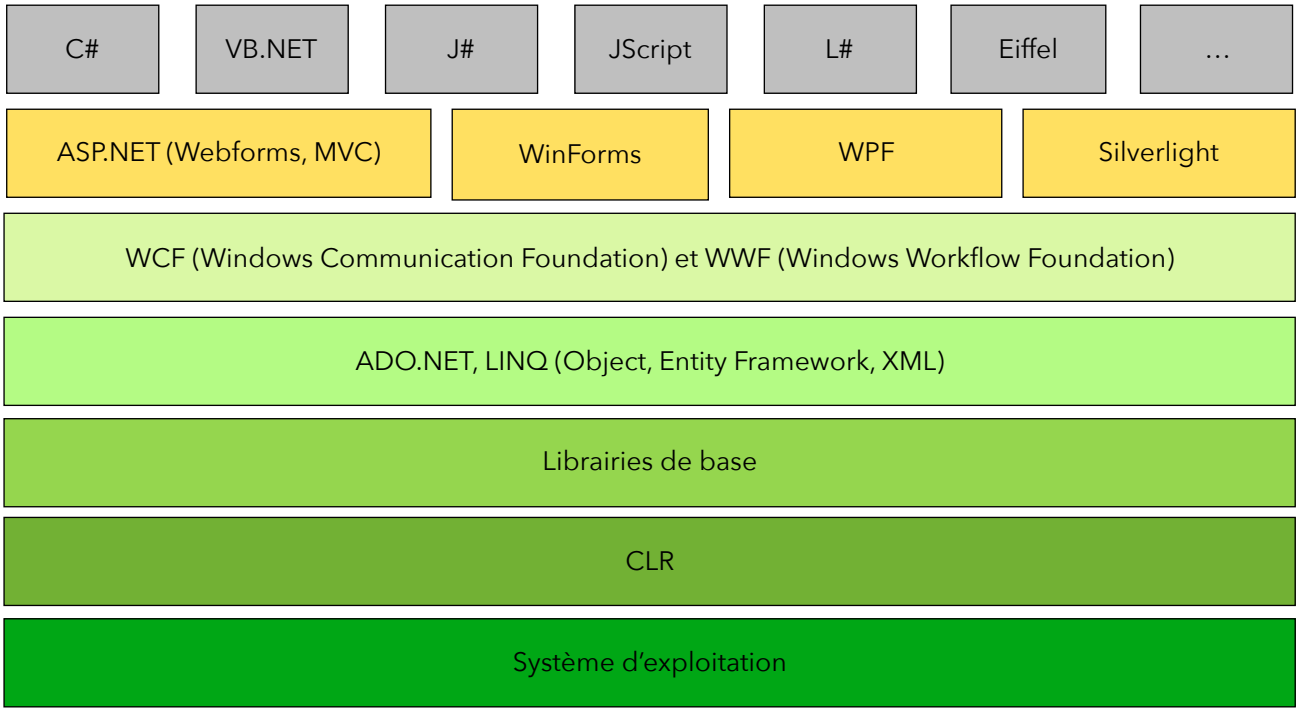
Lorsque le CLR prend en charge le code intermédiaire, il le compile à la volée « Just In Time » pour le transformer en code natif. Cela permet d'optimiser le code machine en fonction des propriétés matérielles/logicielles de l'environnement d'exécution.



Compilation JIT : Code natif optimisé

L'architecture de la plateforme

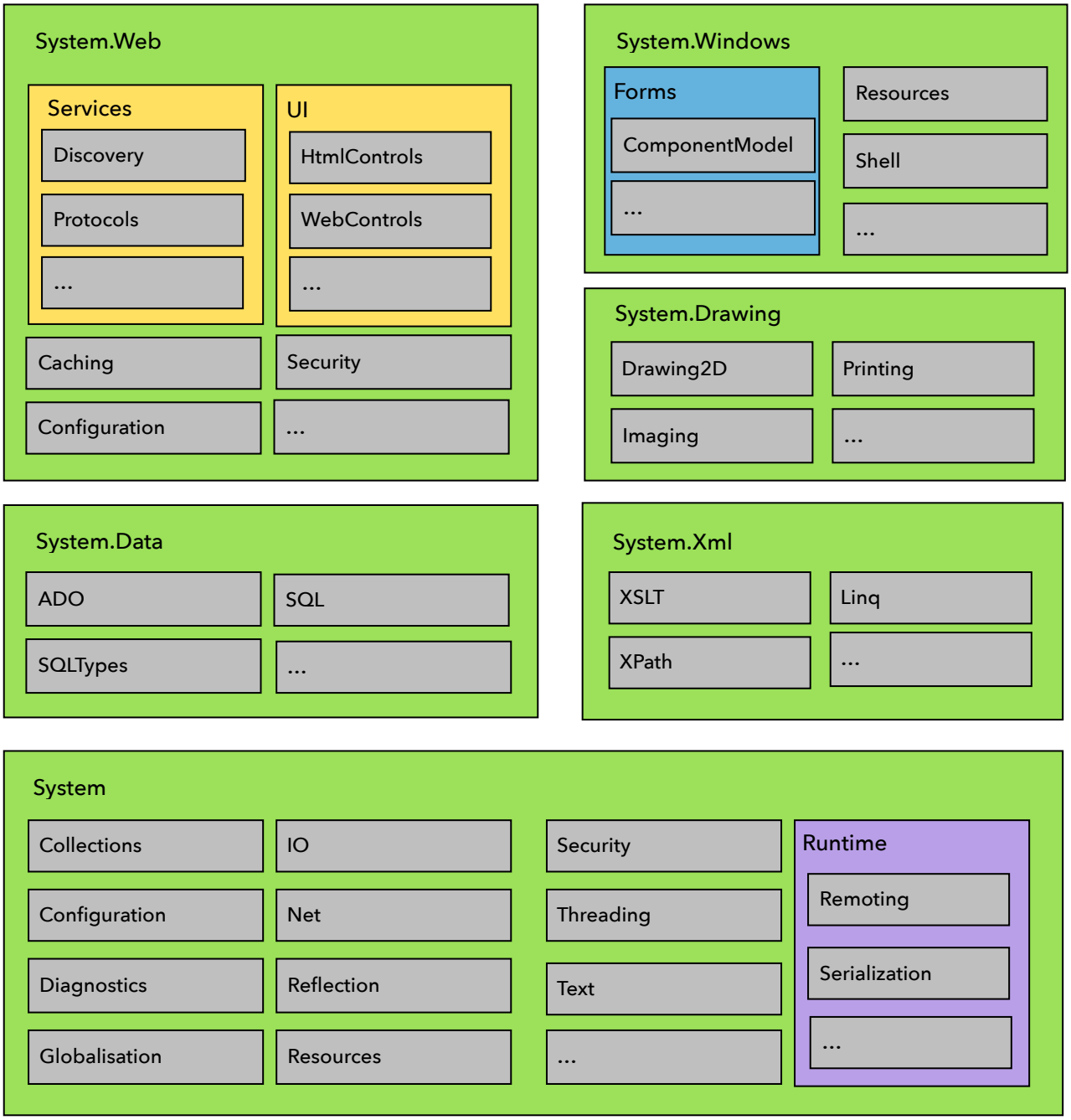
Couches techniques



- Système d'exploitation : gestion des ressources et des processus,
- Common Language Runtime : exécution du code CIL, et prise en charge de la gestion de la mémoire, des accès concurrentiels, de la sécurité,
- Librairies de base : classes mises à disposition couvrant les domaines fondamentaux tels que : I/O, collections, réseau, sécurité, multi-threading, internationalisation, mathématiques, dessin, …,
- ADO.net, LINQ (Object, Entity Framework, XML) : gestion de l'accès aux sources de données (bases de données, XML),
- WCF, WWF : mise en place d'architectures orientées services (Webservices, Workflows),
- ASP.net, WinForms, WPF et Silverlight : réalisation des interfaces utilisateurs,
- C#, VB.net, J#, JScript, L#, Eiffel, … : langages de programmations respectant la CLS et pouvant être exécutés au sein du CLR.

Librairies de base

Les classes des librairies de bases sont utilisables par tous les langages de programmation de la plateforme. Chacune de ces librairies possède son propre espace de nom unique (System.Net, System.XML, …). Ci-dessous un schéma non exhaustif de leur organisation.

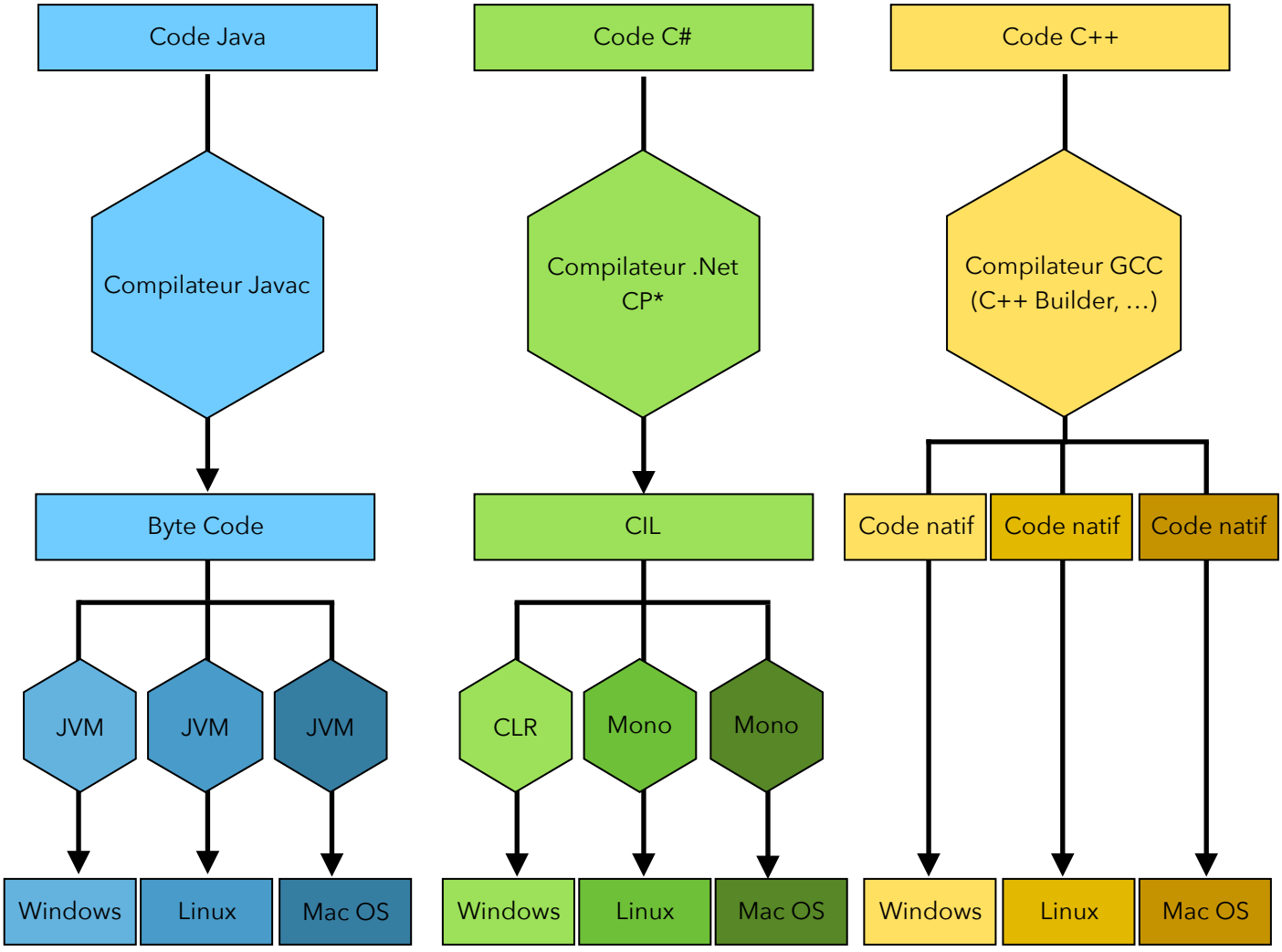

System.Web
Services
Discovery
Protocols
...
UI
HtmlControls
WebControls
...
Caching
Security
Configuration
...
System.Windows
Forms
ComponentModel
...
Resources
Shell
...
System.Drawing
Drawing2D
Printing
Imaging
...
System.Data
ADO
SQL
SQLTypes
...
System.Xml
XSLT
Linq
XPath
...
System
Collections
IO
Security
Configuration
Net
Threading
Diagnostics
Reflection
Text
Globalisation
Resources
...
Runtime
Remoting
Serialization
...

Espace de nom	Brique applicative	Quelques classe
System.*	Gestion du cycle de vie applicatif, des entrés/sorties, du multithreading, des collections d'objets, de Linq, de la réflexion…	ArrayList, IEnumerable, IQueryable, File, Thread.
System.Web.*	Gestion des requêtes HTTP, des WebServices, des contrôles ASP.NET, …	HttpRequest, HttpResponse, Checkbox, Calendar, GridView.
System.Data.*	Gestion de l'accès aux données (Entity Framework, Linq to SQL, …)	DataContext, DataSet, DataTable.
System.Xml.*	Gestion des flux XML, Manipulation de DOM, transformations XSL, accès XPath, de Linq to Xml, …	XmlDocument, XDocument, XslTransform.
System.Drawing.*	Gestion des fonctionnalités graphiques 2D, de traitement d'images, d'impression, …	Graphics, Image, Rectangle, PrintController.
System.Windows.*	Gestion des applications WPF*, les animations, les contrôles de l'interface utilisateur, la liaison de données, la conversion de type, …	Application, DragAction, Button, DataGridView, MediaPlayer, CommandManager.

3.Premiers éléments de comparaison

Nous essaierons, dans le présent chapitre, de développer un comparatif .NET/Java/C++ centré sur leurs environnements d'exécution.

Principes macroscopiques de compilation et d'exécution



Ce schéma montre « l'universalité » des codes intermédiaires Java et C#, tandis qu'il existe autant de codes compilés C++ qu'il y a de plateformes d'exécution.

Les environnements d'exécution

Caractéristique	.NET	Java	C++
Multi-langage	OUI Plus de 20 langages sont supportés. La CLS permet à n'importe qui d'écrire son propre langage .NET	OUI Au delà du Java standard, il existe de langages produisant du ByteCode pouvant être exécuté par une JVM standard. Scala, Groovy, JRuby ou encore Jython en sont des exemples. Ils peuvent même utiliser les API Java founies par le JDK.	NON La notion de machine virtuelle n'existe pas.
Multi-OS	OUI Bien que traditionnellement dédié à l'environnement Microsoft, des portages sur d'autres environnements existent. Mono supporte partiellement la CLI et permet l'exécution d'applications .NET sous Linux ou Mac Os X. Actuellement Mono implémente toutes les fonctionnalités du Framework .NET 4.5 hormis WPF, WWF et partiellement WCF et ASP.NET async tasks.	OUI Par essence Java est multiplateforme et Oracle fournit les machines virtuelles pour une grande quantité de systèmes d'exploitation et d'architectures matérielles. Plusieurs implémentations de la JVM sont par ailleurs fournies : OpenJDK et Jrockit par exemple.	NON Dans le sens où le code source C++ est compilé directement en code machine, il est donc spécifique à l'environnement de compilation. Une application développée pour Windows devra être recompilée pour fonctionner sous Linux, et même potentiellement redéveloppée en ce qui concerne les éléments spécifiques au système d'exploitation (les APIs graphiques par exemple).
Librairies	Ensemble de classes (interfaces, énumérations, ...) identifié par un espace de nom unique.	Ensemble de classes (interfaces, énumérations, ...) réunies au sein d'un package, identifié par un nom complet unique.	Ensemble de classes (interfaces, énumérations, ...) identifié par un espace de nom unique.

Gestion de la mémoire

La JVM et le CLR offrent tous les deux un mécanisme de gestion automatique de la mémoire. La mémoire est allouée en fonction du flot d'exécution de l'application. Un ramasse miette (« Garbage Collector ») est chargé de repérer les objets inutilisés, de les supprimer et de compacter la mémoire pour les objets accessibles. Le déclenchement du ramasse miette est laissé à la charge de la machine virtuelle. Il est toutefois possible de le forcer.

Cette solution affranchit le développeur de la tâche fastidieuse d'allouer/libérer la mémoire de façon programmatique, les risques de fuites mémoires sont ainsi minimisés. L'inconvénient de cette automatisation est qu'en cas de fuite mémoire, il devient nécessaire d'utiliser des outils de profiling pour les trouver.

Les algorithmes de « Garbage collector » de la JVM et du CLR sont très semblables :

- La machine virtuelle alloue un espace d'adressage contigu pour le processus qu'elle exécute, il est appelé « tas managé ». Lorsqu'une application crée le premier type référence, la mémoire est allouée pour le type à l'adresse de base du tas managé. Lorsque l'application crée l'objet suivant, la machine virtuelle lui alloue de la mémoire dans l'espace d'adressage qui suit immédiatement le premier objet. Aussi longtemps que de l'espace d'adressage est disponible, de nouveaux objets pourront être stockés selon ce principe.
- Le moteur d'optimisation du ramasse miette détermine le meilleur moment pour lancer une opération de nettoyage de la mémoire sur base des allocations effectuées. Il examine les objets inaccessibles (dont la référence n'est plus utilisée) et libère la mémoire qui leur a été allouée.
- Le ramasse miette optimise ensuite le tas managé en réorganisant la mémoire afin que tous les blocs alloués soient de nouveau contigus.

En détail, les objets créés au fur et à mesure du cycle de vie de l'application sont classés en générations. Plus un objet est vieux, plus sa génération est élevée. Les objets de génération 0 sont ainsi traités en priorité par le ramasse miette puisque, par nature, ils sont beaucoup plus volatiles.

En C++, la gestion de la mémoire est à la charge du développeur. Ainsi, un objet alloué doit nécessairement être explicitement libéré, sans quoi une fuite mémoire est induite.

Performances

Il est difficile de comparer les performances des trois langages, d'ailleurs il n'existe pas réellement de littérature à ce sujet. Java et C# ne produisent pas un code nativement exécutable par le processeur. En ce sens, nous pourrions penser que l'interprétation du code intermédiaire rend ces technologies nettement plus lentes. Contrairement à cette idée reçue, les compilateurs JIT permettent d'obtenir une excellente optimisation du code machine produit. Ce dernier est généré en fonction des caractéristiques exactes de l'environnement d'exécution. Le code C++ étant compilé une fois pour toute et pour un environnement figé, il ne prend pas en compte les subtilités de son environnement d'exécution (caractéristiques des systèmes de fichier, nombre de processeurs, type de mémoire, …).

Briques de base

Le tableau suivant liste les briques de base de la plateforme .NET et nous permet de constater qu'il existe une équivalence quasi-systématique avec la plateforme Java/JEE. Celle-ci est caractérisée par la pluralité des solutions proposées pour chacune de ces briques. La plateforme .NET apparaît clairement beaucoup plus directive dans les choix laissés aux développeurs.

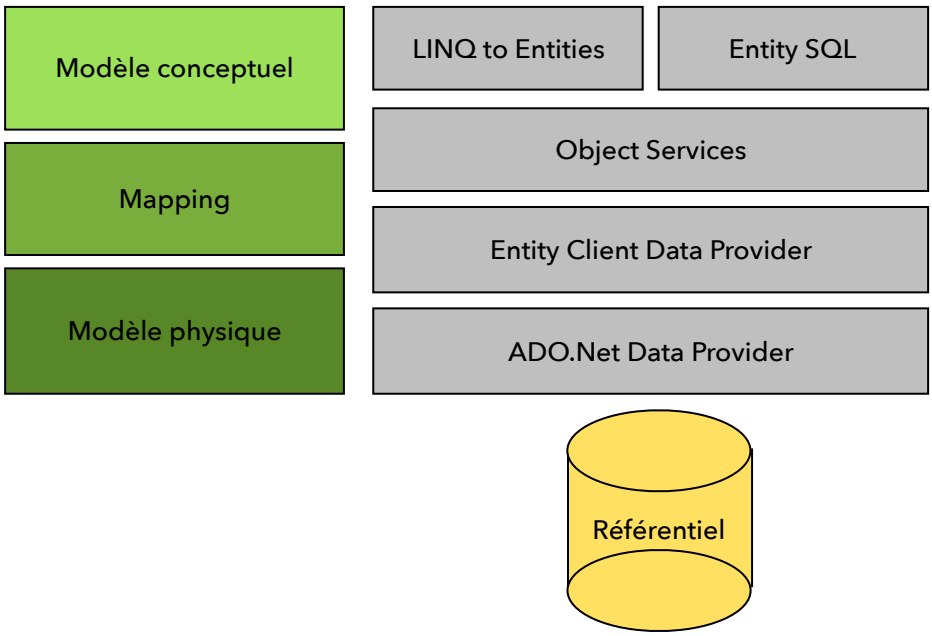
Typologie de librairies	Plateforme .NET	Java/JEE	C++
Développement Web	ASP.Net (WebForms, MVC)	De nombreuses spécifications : Servlet, JSP, JSTL, JSF, et de nombreux frameworks OpenSource (Struts, Tapestry, Spring MVC, ...).	Pas d'environnement Web natif au langage mais il existe de nombreux frameworks (WT, CppCMS, ...).
IHM	WinForms, WPF (Windows Presentation Foundation), WinRT	AWT (Abstract Window Toolkit), Swing, SWT (Standard Widget Toolkit).	Des librairies tiers telles que Qt, GTK+, wxWidgets, ...
Accès aux bases de données	ADO.Net	JDBC (Java DataBase Connectivity).	Des librairies tiers telles que Qt, ODBC, SQLApi et toutes les APIs natives fournies par les éditeurs de SGBD
Appels d'objets distants	Remoting	RMI (Remote Method Invocation), EJB (Enterprise Java Bean) Remote.	COM
Architecture Orientée Service	WCF (Windows Communication Foundation)	JAX-WS (Java API for XML Web Services), JAX-RS (Java API for RESTful Services), JMS (Java Message Service), JMX (Java Management eXtensions)	Aucune spécification ou solution native. Des librairies et conteneurs de webservices tels qu'HydraExpress ou gSOAP.
XML	Linq To XML	JAXP (Java Api for Xml Processing) pour la manipulation de documents XML, JAXB (Java Architecture for XML Binding) pour le mapping objet/xml.	Des librairies tiers telles que TinyXML, Xerces, LibXML2, ...
Sécurité	Intégré dans la plate-forme	JAAS (Java Authentication and Authorization Service), JCE (Java Cryptography Extension)	A la charge du développeur.
Interaction avec du code natif	Interactions entre code managé et code natif avec PInvoke (Platform Invocation Services)	JNI (Java Native Interface)	sans objet
Objets métiers	non	EJB (Enterprise Java Bean), Bean CDI (Contexts and Dependency Injection)	non
Mapping objet relationnel	Entity Framework, et quelques frameworks pour la plupart portés de Java (NHibernate, ...).	JPA avec plusieurs implémentations (hibernate, Toplink, ...).	Des librairies tierces comme ODB
Manipulation dynamique des objets	Librairie System.Reflection	Librairie java.lang.reflect	Pas de solution native. Des librairies tiers comme CPGF

Nous voyons clairement dans ce tableau comparatif la grande similitude entre les possibilités des plateformes .NET et Java. Chacune d'entre elles dispose d'une solution native répondant aux grandes problématiques du développement moderne. Java apparait plus ouvert et propose des multiples solutions (incluses dans la plateforme ou sous la forme de librairies tiers) tandis que .NET guide les développeurs vers une solution unique.

C++ ne couvre pas nativement autant de fonctionnalités que les autres plateformes. Il sera souvent nécessaire d'intégrer aux applicatifs bon nombre de librairies tierces. C'est la différence entre un framework et un simple langage de développement.

4.Entity Framework

Entity Framework est la solution de mapping objet/relationnel de la plateforme .NET. Il est architecturé de la façon suivante :



Les concepts :

- Modèle conceptuel : Les classes décrivant le modèle de données de l'application ainsi que leurs relations. Le modèle est « agnostique » vis à vis du référentiel de persistance.
- Modèle physique : L'accès aux données (tables, vues, procédures stockées, relations, contraintes d'intégrité) du référentiel de persistance.
- Mapping : Informations permettant d'effectuer la correspondance entre le modèle conceptuel et le modèle physique de données.

Les briques techniques :

- Linq to entities : Le langage de requêtage du modèle objet. Il retourne les entités définies par le modèle conceptuel (ce langage est étudié plus en détail dans le chapitre 6).
- Entity SQL : Un autre langage de requêtage du modèle objet, davantage orienté « SQL » dans sa syntaxe.
- Object Services : La couche responsable de la « matérialisation », c'est à dire le processus de conversion des données retournées par le « Entity Client Data Provider » en un objet du modèle conceptuel.
- Entity Client Data Provider : La couche responsable de la conversion des requêtes Linq ou Entity SQL en requêtes SQL, compréhensibles par la base de données sous-jacente. Elle communique avec ADO.Net Data provider.
- ADO.Net Data Provider : La couche responsable de la communication avec la base de données en se basant sur la technologie ADO.Net (standard dans la plateforme .Net).

Les trois approches du mapping objet relationnel

Il existe trois approches permettant de mettre en correspondance le modèle objet et le modèle physique de données : Code first, Model first et Data first.

Code first

Il s'agit de l'approche privilégiée lorsque l'on se focalise sur une méthodologie de type « model driven design* ». Elle consiste à écrire les POCO (Plain Old CLR Object, classes représentant le modèle objet) et à générer la base de données d'après cette modélisation. Dans cette approche, les annotations sont utilisées pour exposer des méta-informations permettant de superposer des notions relationnelles aux propriétés de chaque POCO. Ci-dessous un exemple définissant un « Blog » (table « InternalBlogs »), écrit par un « Owner » et possédant plusieurs « Posts » :

- La table est désignée par l'annotation « [Table] »,
- la clé primaire est spécifiée via l'annotation « [Key] »,
- les colonnes sont spécifiées via l'annotation « [Column] »,
- la jointure vers « l'Owner » est spécifiée via l'annotation « [ForeignKey] », au travers de laquelle on précise la clé étrangère,
- la jointure vers les « Posts » est spécifiée via l'annotation « [InverseProperty] », au travers de laquelle on précise la clé étrangère « InBlog » référençant l'entité « InternalBlogs » dans l'entité « Post ».

```
[Table("InternalBlogs")]
public class Blog

[Key]
public int BlogId {get;set;}

[Column("BlogDescription", TypeName="text")]
public String Description {get;set;}

[ForeignKey("OwnerId")]
public Owner Owner { get; set; }

[InverseProperty("InBlog")]
public List<Post> Posts { get; set; }
```

A défaut d'annotations, Entity Framework va générer un schéma de BDD d'après les conventions « code-first » par défaut (nommage des colonnes, construction des clés primaires/étrangères, implémentation des stratégies d'héritage).

Lors de la génération du schéma de BDD, Entity Framework peut appliquer plusieurs stratégies (suppression/création du schéma systématique, suppression/création en cas de changement du modèle objet, …). Cependant, il peut être embêtant de supprimer la base lorsque celle-ci contient déjà des données. L'outil « migration » permet de gérer automatiquement les mises à jour du schéma lors d'une modification du modèle sans aucune perte de données.

Model first

Dans cette approche, il s'agit de modéliser les entités, associations et héritages directement dans l'assistant de Visual Studio. Il est ensuite possible de générer la base de données correspondante.

Data first

Dans cette approche, la base de données est pré-existante au code et le développeur utilisera son schéma pour générer les POCO correspondants.

Le mapping est basé sur un fichier EDMX (format XML) qui décrit trois choses :

- le model physique de données,
- le modèle conceptuel (objet),
- le mapping entre les deux.

Ce fichier XML peut être généré directement d'après la BDD via visual studio, cela aura pour effet de générer pour chaque entity :

- un POCO représentant le model,
- un objet « Context » permettant d'accéder aux fonctionnalités de persistance de l'entité (que l'on peut comparer au pattern DAO).

Un « POCO proxy » est par ailleurs créé « at runtime* » (lors de l'exécution) pour chaque entité afin de compléter le code de celui-ci par des fonctionnalités liées par exemple au « lazy loading* » (chargement à la demande de la données).

Les possibilités de requêtage du modèle

Linq to entities

Cette fonctionnalité est disponible via deux syntaxes différentes : syntaxe de méthode et syntaxe de requête. Linq est présenté en détail dans le chapitre 6.

Entity SQL

Cette fonctionnalité est basée sur un langage d'expression de requête « type SQL » orienté modèle objet. Bien que la syntaxe fasse penser à une simple requête SQL, les notions manipulées sont bien les entités du modèle conceptuel. L'exemple suivant montre la recherche d'étudiants dont le nom est égal à « Bruno ».

```
string sqlString = "SELECT VALUE stu FROM SchoolDBEntities.Students AS stu WHERE stu.StudentName == 'Bruno'";

 var objctx = (ctx as IObjectContextAdapter).ObjectContext;
ObjectQuery<Student> student = objctx.CreateQuery<Student>(sqlString);
Student newStudent = student.First<Student>();
```

SQL natif

Ce dessous le code permet de lire l'identifiant de l'étudiant dont le nom est « Bruno ».

```
using (var ctx = new SchoolDBEntities())
{
  var studentId = ctx.Students.SqlQuery("Select studentid from Student where studentname='Bruno'").FirstOrDefault<Student>();
}
```

Quelques notions complémentaires

Le contexte

Comme nous l'avons vu dans les exemples précédents, les opérations de persistance avec Entity Framework sont basées sur l'existence d'un « Context ». Il représente en quelque sorte la session au sein de laquelle les opérations de lecture/création/mise à jour/suppression sont cohérentes et liées avec les opérations effectuées sur le modèle objet.

Lorsque toutes les opérations effectuées sur le modèle objet sont effectuées au sein du même « Context », on parle de mode connecté. A contrario, on parle de mode déconnecté lorsqu'après lecture du modèle objet au sein d'un contexte, ce dernier est fermé, et la persistance s'effectue dans un second créé pour l'occasion (il faut alors attacher le modèle objet au nouveau contexte).

Le mode de chargement des données

Lorsque l'on souhaite charger un objet agrégeant lui aussi des sous-objets, il est possible de spécifier à Entity Framework d'effectuer un chargement explicite des relations (mode « eager ») grâce au mot clef « include ». Considérons que la classe « Student » déclare une propriété « StudentAddress ». Ici un étudiant sera chargé avec son adresse de façon explicite :

```
using (var context = new SchoolDBEntities())
    {
       var res = (from s in context.Students.Include("StudentAddress")
                where s.StudentName == "Bruno"
                select s).FirstOrDefault<Student>();
    }
```

Il est également possible de ne charger les sous-objets que lorsqu'ils sont strictement requis (mode « lazy »). Dans l'exemple ci-dessous, on ne charge dans un premier temps que les étudiants. Le chargement de l'adresse du premier étudiant n'est effectué que lorsqu'elle est nécessaire.

```
using (var ctx = new SchoolDBEntities())
    {
      IList<Student> studList = ctx.Students.ToList<Student>();
      Student std = studList[0];

     StudentAddress add = std.StudentAddress;
    }
```

Ici, une seconde requête SQL sera générée lorsque l'on accède explicitement à l'adresse de l'étudiant. Sur des graphes d'objet à de multiples niveaux de profondeurs, cela permet de ne pas charger une quantité déraisonnable d'objets en mémoire.

Comparaison avec Java/JEE

Il convient ici de comparer .NET non seulement à Java, mais plus généralement à JEE (Java Enterprise Edition). En effet, JEE intègre la spécification JPA (Java Persistance API). Cette spécification définit :

- l'API de persistance elle-même (classes/interfaces du package javax.persistence),

- le langage de requêtage Java Persistence Query Language (JPQL),
- les annotations relationnelles.

Les APIs de persistance

Il existe plusieurs implémentations de cette spécification : Hibernate, TopLink ou encore OpenJPA. Elles respectent en général la totalité de la spécification et y ajoutent leurs propres fonctionnalités. L'intégration stricto sensu de l'API JPA dans le code applicatif permet de garantir son indépendance vis à vis d'une implémentation spécifique. On peut ainsi dire qu'il y a un équivalent théorique à Entity Framework, à savoir JPA et autant d'équivalents concrets qu'il existe d'implémentations de cette spécification.

Les trois approches de conception offertes par EF (Entity Framework) sont également couvertes par ces frameworks de persistance :

- Code first : JPA (et par conséquent les frameworks l'implémentant) fournit les annotations permettant de spécifier les caractéristiques relationnelles liant le modèle conceptuel au modèle physique. Tout comme EF, Hibernate permet de générer le schéma de BDD à partir du modèle conceptuel,
- Model first : Des outils tels qu' « Hibernate tools » permettent d'adopter une approche graphique pour définir entités et mapping,
- Data first : Hibernate permet de définir le mapping dans des fichiers XML, tout comme .Net le permet grâce aux fichiers EDMX.

Annotations et langage de requêtage

Nous l'avons vu, EF propose un ensemble d'annotations permettant de réaliser le mapping du modèle objet vers le modèle physique de données. Il existe un ensemble d'annotations équivalent dans JPA. Ci-dessous un exemple définissant une classe « Person » (table « TBL_PERSON ») possédant une « Address » :

- la table est désignée par l'annotation « @Entity »,
- la clé primaire est spécifiée via l'annotation « @ID »,
- les colonnes sont spécifiées via l'annotation « @Column »,
- la jointure vers l'adresse est spécifiée via l'annotation « @ManyToOne », au travers de laquelle on précise la clé étrangère.

```
 @Entity(name = « TBL_PERSON")
public class Person {

    @Id
    @Column(name = "PERSON_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long personID;

    @Column(name = « FIRST_NAME")
    private String firstName;

    @Column(name = « LAST_NAME")
    private String lastName;

    @Column(name = « BIRTH_DATE")
    private Date birthDate;

   @ManyToOne(optional=false)
   @JoinColumn(name="ADDRESS_ID",referencedColumnName="ADDRESS_ID")
    private Address address;
}
```

Le langage de requêtage JPQL propose une syntaxe « sql like » permettant de manipuler les entités du modèle conceptuel. Ci-dessous un exemple de requête sélectionnant les personnes s'appelant « Bruno » :

```
// create a query
Query q = em.createQuery("SELECT p FROM Person p WHERE p.firstName = :f");

// set the parameter
q.setParameter("f", "Bruno");
```

Ce langage est un équivalent à Linq to EF.

Comparaison avec C++

Des outils de mapping objet relationnel existent également en C++. Nous avons choisi de présenter ci-dessous ODB.

ODB permet, via des pragmas (équivalent des annotations), de réaliser des opérations de CRUD (Create, Read, Update, Delete) sur des bases de données MySQL, SQLite, PostGresql, Oracle et SQL Server. Le code C++ qui réalise la conversion entre les beans* persistés et leur représentation en base est automatiquement généré par le compilateur ODB. ODB permet également la génération (y compris les versions incrémentales) du schéma de base de données. Ci-dessous un exemple de mapping pour une classe « Person » :

```
#pragma db object table("people")
 class Person
 {
   ...

 private:
  friend class odb::access;
  person ();

  #pragma db id auto
  unsigned long id_;

  string firstName;
  string lastName;

  #pragma type("INT UNSIGNED")
  unsigned short age_;
 };
```

Ci-dessous un exemple de requête ODB pour sélectionner les personnes s'appelant « Bruno » et de moins de 30 ans :

```
result r (db.query<person> (query::firtName == "Bruno" && query::age < 30));

for (result::iterator i (r.begin ()); i != r.end (); ++i)
{
  cout << "Hello, " << i->first () << endl;
}
```

Il existe d'autres solutions de mapping pour C++ telles que QxORM ou LiteSQL.

Conclusion

Il existe des solutions d'ORM* (Object Relationnel Mapping) quelle que soit la plateforme de développement. Dans le cas de .NET, la solution est directement intégrée à la syntaxe du code et à l'environnement de développement, le développeur est parfaitement guidé dans sa mise en oeuvre. Par ailleurs, elle tire pleinement partie de la puissance et de la facilité d'utilisation des expressions Linq et de la génération incrémentale du schéma de base de données. Il semble donc que cette solution présente probablement le meilleure compromis fonctionnalités/facilité de mise en oeuvre.

5.Linq

Principes

LINQ (Language Integrated Query), introduit dans la version 3.5 du .NET Framework, permet de rapprocher le monde des objets et le monde des données.

Traditionnellement, les requêtes sur des données sont exprimées sous forme de chaînes simples sans vérification de type au moment de la compilation. Il est nécessaire d'apprendre un langage de requête pour chaque type de source de données : bases de données SQL, documents XML, services Web, ... Avec Linq, toute requête est exprimée dans une syntaxe universelle, native du langage .Net utilisé. Les requêtes sont effectuées sur des collections d'objets fortement typées en utilisant des mots clés du langage et des opérateurs courants. La source et le formalisme de la données d'origine sont complètement masqués. Linq est décliné en trois versions :

- Linq To Objects,
- Linq To XML,
- Linq To DataSet/Linq to SQL/Linq To Entity Framework.

Toutes les opérations de requête LINQ comportent trois actions distinctes :

- Obtenir la source de données,
- Créer la requête,
- Exécuter la requête.

Collections simples

L'exemple suivant montre comment les trois parties d'une opération de requête sont exprimées dans le code source. Il utilise un tableau d'entiers comme source de données pour des raisons pratiques, mais les mêmes concepts sont également applicables à d'autres sources de données.

```
class IntroToLINQ
{
   static void Main()
   {
      // Source de données
      int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

      // Création de la requête
      var numQuery =
         from num in numbers
         where (num % 2) == 0
         select num;

      // Exécution de la requête
      foreach (int num in numQuery)
      {
         Console.Write("{0,1} ", num);
      }
   }
}
```

La requête permet ici de sélectionner les nombres pairs dans la liste « numbers ». Etant donné que la source de données est un tableau, elle prend en charge implicitement l'interface IEnumerable<T> générique. Cela signifie qu'elle peut être interrogée avec Linq. La requête est exécutée dans l'instruction foreach et celle-ci requiert

IEnumerable ou IEnumerable<T>. Les types qui prennent en charge IEnumerable<T> ou une interface dérivée telle que l'interface générique IQueryable<T> sont des types pouvant être interrogés avec Linq.

Autres sources de données

Si les données sources ne sont pas déjà en mémoire comme type requêtable (comme dans l'exemple précédent), le fournisseur Linq doit les représenter comme telles. Par exemple, Linq To XML charge un document XML dans un type XElement requêtable :

```
XElement contacts = XElement.Load(@« c:\myContactList.xml");
```

L'exemple suivant montre comment requêter une base de données SQL Server.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// sélection des clients vivant à Londres
IQueryable<Customer> custQuery =
   from cust in db.Customers
   where cust.City == "London"
   select cust;
```

En détail

Linq ne gère pas uniquement la récupération des données. C'est également un outil puissant pour la transformation de données. En utilisant une requête Linq, il est possible d'utiliser une séquence source en entrée et la modifier de nombreuses façons pour créer une séquence de sortie de façon à :

- fusionner plusieurs séquences d'entrée dans une séquence de sortie unique avec un type nouveau,
- créer des séquences de sortie dont les éléments se composent d'une seule propriété ou de plusieurs propriétés de chaque élément de la séquence source,
- créer des séquences de sortie dont les éléments se composent des résultats des opérations effectuées sur les données sources,
- créer des séquences de sortie dans un format différent. Par exemple, il est possible de transformer des données de lignes ou de fichiers texte en XML.

Fonctionnalités de sélection, de tri, de regroupement et de jointure

Sélection

Il est possible d'appliquer les opérateurs communs du langage de programmation pour réaliser une sélection.

```
var queryCustomers =      from cust in customers
                          where cust.City == "London" || cust.City == "Paris"
                          select cust;
```

Tri

La requête précédente peut être étendue pour trier les résultats selon la propriété Name. Étant donné que Name est une chaîne, le comparateur effectue, par défaut, un tri alphabétique de A à Z.

```
var queryCustomers =
        from cust in customers
        where cust.City == "London" || cust.City == "Paris"
        orderby cust.Name ascending
        select cust;
```

Regroupement

La clause « group » permet de grouper les résultats selon une clé. Le résultat sera une liste de listes indexée par la clé de regroupement.

```
var queryCustomersByCity =
        from cust in customers
        group cust by cust.City;

foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

Jointures

Les opérations de jointure créent des associations entre les séquences qui ne sont pas modelées explicitement dans les sources de données. Il est, par exemple, possible d'effectuer une jointure pour rechercher tous les clients et distributeurs qui ont le même emplacement.

```
var innerJoinQuery =
        from cust in customers
        join dist in distributors on cust.City equals dist.City
        select new { CustomerName = cust.Name, DistributorName = dist.Name  };
```

Deux syntaxes possibles

Dans tous les exemples exposés jusqu'à présent, nous avons formulé la collecte de données sous la forme d'une syntaxe de requête. La plateforme .Net offre la possibilité de les exprimer sous la forme d'une syntaxe de méthode. Ces méthodes sont appelées « opérateurs de requête standard » et ont des noms tels que « Where », « Select », « GroupBy », « Join », « Max », « Average », etc.

```
int[] numbers = { 5, 10, 8, 3, 6, 12};

// syntaxe de requête
IEnumerable<int> querySyntax =
        from num in numbers
        where num % 2 == 0
        orderby num
        select num;

// syntaxe de méthode
IEnumerable<int> methodSyntax = numbers.Where(num => num % 2 == 0).OrderBy(n => n);
```

Ces deux syntaxes produisent un résultat identique. La seconde notation utilise des expressions lambda.

Comparaison avec Java

Il n'y a pas réellement d'équivalent à la syntaxe de requête Linq dans Java, il existe cependant un solution se rapprochant de la syntaxe de méthode. Celle-ci utilise les expressions Lambdas. Ces dernières ont été introduites dans Java 8. En complément de l'API Stream, elles fournissent donc une alternative à Linq to object. Ci-dessous un tableau comparatif de deux solutions :

Opération	Linq	Java Stream & lambdas
Rechercher les mots contenant « am » parmi : string[] names = { "Sam", "Pamela", "Dave", "Pascal", "Erik" };	List<string> filteredNames = names.Where(c => c.Contains("am")).ToList();	List<String> filteredNames = stream(names).filter(c -> c.contains("am")).collect(toList());
Ecrire « Hello » après chacun des mots d'une liste	List<string> list = new List(){ "Anders", "David", "James", "Jeff", "Joe", "Erik" }; list.Select(c => "Hello! " + c).ToList() .ForEach(c => Console.WriteLine(c));	List<String> list = asList("Anders", "David", "James", "Jeff", "Joe", « Erik"); list.stream().map(c -> "Hello! " + c).forEach(System.out::println);
Constituer une liste contenant les items suivants les trois premiers de la liste : string[] vipNames = { "Sam", "Samuel", "Samu", "Remo", "Arnold","Terry" };	var skippedList = vipNames.Skip(3).ToList();	List<String> skippedList; skippedList = stream(vipNames).substream(3).collect(toList());

Comparaison avec C++

Tout comme en Java, l'introduction des expressions lambda dans C++ V11 a permis l'émergence de librairies équivalentes à Linq To Object telles que RX (Reactive eXtensions) de Microsoft ou cpplinq.

Dans l'exemple suivant, nous constituons une liste triée de nombres pairs avec Cpplinq (opérateur >>) :

```
using namespace cpplinq;
int ints[] = {3,1,4,1,5,9,2,6,5,4};

 return from_array (ints)
    >>  where ([](int i) {return i%2 ==0;})
    >>  orderby_ascending ([](int i) {return i;})
    >>  to_vector ();
```

Conclusion

Nous n'allons pas être exhaustif ici, mais il faut retenir de ce comparatif que :

- Linq semble plus abouti, certaines fonctionnalités ne sont pas accessibles en Java ou C++ ou au prix d'une syntaxe nettement plus verbeuse et compliquée,
- Linq permet d'exprimer ces traitements sous forme d'une syntaxe de requête (Sql like), facilement compréhensible par la majorité des développeurs,
- Stream/Lambda en Java 8 présente un potentiel de traitement plus important du fait de son détachement par rapport à un traitement typé SQL, notamment grâce aux opérations de Map/Reduce,
- il existe, en Java, des APIs complémentaires aux fonctionnalités de Java 8 permettant d'obtenir des fonctionnalités équivalentes à celles de Linq (Linq To SQL/EF) avec par exemple Jinq ou jOOQ.

6.ASP.NET

Le modèle de programmation standard des applications web en ASP consistait à mixer code de présentation (HTML/CSS/Javascript) et code applicatif, interprété par le serveur (VBScript ou JScript). Sa relecture est évidemment difficile, et les faibles possibilités de conception qu'il offre en font une solution à l'évolutivité peu satisfaisante. Ce modèle est relativement comparable à la technologie JSP proposée aux développeurs à la naissance du framework JEE. ASP.NET améliore l'ASP original par adjonction de la technique dite « du code-behind ». Cette technique est mise en oeuvre lors de la conception d'un site web via des Webforms.

Les Webforms

A la différence des pages ASP classiques (qui sont interprétées), les pages ASP.NET, dites « Webforms », sont compilées en code intermédiaire à leur première exécution. Ce qui optimise les exécutions suivantes (nous retrouvons ce type de fonctionnement également pour les JSP).

Lors de la création d'une nouvelle page ASP.NET pour « monsite », deux fichiers sont en réalité générés : « mapage.aspx » et « mapage.aspx.cs ».

Tandis que « mapage.aspx » contient les langages de balisage (HTML et contrôles ASP) et de script (Javascript) :

```
<%@ Page Title="Ma Page" Language="C#" MasterPageFile="~/monsite.master" AutoEventWireup="true" CodeBehind="mapage.aspx.cs"
Inherits="MonSite._MaPage" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Ma Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:Label runat="server" id="HelloWorldLabel"></asp:Label>
   </div>
   </form>
</body>
</html>
```

« mapage.aspx.cs » contient le langage .Net (C#, VB.net, …) prenant en charge :

- l'implémentation des couches techniques (persistance, réseau, journalisation, système de fichiers, …),
- l'implémentation des règles de gestions,
- la gestion des évènements déclarés au niveau des contrôles ASP,
- l'attribution ou la lecture de valeurs aux contrôles ASP (data binding).

```
public partial class _MaPage : System.Web.UI.Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
            HelloWorldLabel.Text = "Hello, world!";
   }
}
```

La page HTML résultant de l'exécution conjointe de « mapage.aspx » et de « mapage.aspx.cs » affichera simplement le texte « Hello, world! ».

Principes de fonctionnement

L'entête du fichier « mapage.aspx » fait référence à :

- le « MasterPageFile » : désigne le « template » général des pages du site web (entête, corps et pied de page),
- le « CodeBehind » : désigne le fichier contenant le code C# qui sera exécuté lors de la compilation de la page,
- l' « Inherits » : désigne la classe dont va dériver le code compilé de « mapage.aspx ». Ici il s'agira de la classe « _MaPage » définie dans le code behind « mapage.aspx.cs ».

Le résultat de la compilation de « mapage.aspx » sera en réalité une classe qui héritera de « _MaPage », au sein de laquelle les contrôles ASP.NET auront été transformés en code intermédiaire. Ainsi contrôles ASP.NET, gestion des évènements, data binding, règles métiers, etc, feront au final partis du même code compilé. C'est pourquoi, lors de l'écriture du code behind, nous avons accès aux contrôles ASP. Dans notre exemple, le contrôle « HelloWorldLabel » est défini dans « mapage.aspx » et pour autant nous y accédons dans le code behind (HelloWorldLabel.Text = "Hello, world! »;).

« _MaPage.aspx.cs » hérite elle même de « System.ui.web.Page ». Par conséquent, il est possible de surcharger des méthodes qui sont appelées tout au long du cycle de vie de la page : « Page_Load » est automatiquement appelée lors du chargement de « mapage.aspx ».

Nous pouvons noter que le contrôle ASP.NET « HelloWorldLabel » possède une propriété « runat = server ». Celle-ci nous assure que ce contrôle sera traité par le serveur et sera donc accessible dans le code behind (dans Visual Studio, l'absence de cette propriété aurait eu pour effet de ne pas avoir accès à la complétion syntaxique).

ASP.NET repose sur un modèle de type « event driven » (piloté par les évènements). Les évènements sont le lien entre les contrôles ASP.NET (déclencheur d'évènements) et le code behind (traitement des évènements). Dans notre exemple, le chargement de « mapage.aspx » a déclenché l'évènement « Page_Load ». Cet évènement a été « capturé » par le code behind et traité par la méthode idoine.

Pour illustrer notre propos, modifions « mapage.aspx » de façon à ajouter un champs de saisie de texte et un bouton :

```
<form id="form1" runat="server">
<div>
   <asp:Label runat="server" id="HelloWorldLabel"></asp:Label>
   <br /><br />
   <asp:TextBox runat="server" id="TextInput" />
   <asp:Button runat="server" id="TextButton" text="Change the text"  onclick="TextButton_Click()" />
</div>
</form>
```

puis dans « mapage.aspx.cs » ajoutons une méthode dont la signature correspond au nom indiqué dans la propriété « onClick » du nouveau bouton :

```
protected void TextButton_Click(object sender, EventArgs e)
{
    HelloWorldLabel.Text = "Hello, " + TextInput.Text;
}
```

Dorénavant, l'appui sur le bouton « TextButton » déclenchera l'évènement « TextButton_Click ». Ce dernier correspond à l'appel de la méthode « TextButton_Click(object sender, EventArgs e) » de la classe « _MaPage » définie dans le code behind. Le texte du champ « HelloWorldLabel » sera alors valorisé avec le texte renseigné dans le champs « TextInput ».

Ce principe de fonctionnement évènementiel nous amène au concept de « PostBack ».

Le PostBack*

Le « PostBack » désigne l'action de soumission d'un formulaire HTML vers le serveur web. Les champs du formulaire sont ainsi « postés » vers le serveur (au sens HTTP du terme), exploités (règles de gestion, persistance, …) puis la page HTML est de nouveau générée.

Certains contrôles ASP.NET possèdent une propriété « AutoPostBack ». Lorsqu'elle vaut « true » le formulaire HTML construit lors de l'exécution de la page ASP.NET est du type : <FORM METHOD='POST' ACTION='mapage.aspx'>. Lors d'une action sur ce contrôle (click, changement de valeur, …) le formulaire va être soumis vers la page elle-même.

Une fonction javascript « _doPostBack() » est également générée. Son rôle est de soumettre le formulaire. Cette fonction possède des arguments qui permettent de transmettre au serveur (via des champs cachés), les caractéristiques de l'évènement déclencheur du « POST ».

Lorsqu'aucun évènement n'est précisé sur un contrôle ASP.NET qui est en « AutoPostBack » :
- la fonction « _doPostBack() est appelée »,
- cette dernière « poste » le formulaire vers le serveur,
- la page est ré-exécutée (le code behind est exécuté selon son cycle de vie standard),
- la page HTML est re-générée.

Lorsqu'un évènement est précisé sur un contrôle ASP.NET qui est en « AutoPostBack » :
- la fonction « _doPostBack() est appelée »,
- cette dernière « post » le formulaire vers le serveur,
- la page est ré-exécutée (le code behind est exécuté selon son cycle de vie standard),
- la méthode du code behind désignée par l'évènement est appelée,
- la page HTML est re-générée.

Le cycle de vie du code behind

Toute requête faite à une page passe par un certain nombre d'étapes. On appelle cela le cycle de vie d'une page. ASP.NET va ainsi enchainer un certain nombre d'étapes pour produire le résultat de l'exécution, y compris dans l'appel aux méthodes du code behind. Ces méthodes peuvent être surchargées afin de produire un comportement particulier. Par exemple, les méthodes « Page_Init », « LoadPostData », « Page_Load » ou encore « Page_PreRender » sont appelées dans cet ordre et doivent contenir le code approprié à chaque étape. On pourra, par exemple, affecter des valeurs par défaut aux contrôles ASP.NET dans la méthode « Page_Init ». A noter qu'après un PostBack, un booléen global « IsPostBack » est positionné à « true » afin d'éviter de réaffecter les valeurs par défaut aux contrôles si l'utilisateur les a modifié avant la soumission du formulaire.

Il est possible de désactiver un « PostBack » en positionnant la propriété « AutoPostBack » à « false ». Dans ce cas, un évènement généré par un contrôle ASP.NET ne déclenchera pas la soumission du formulaire vers le serveur. On pourra donc n'exécuter que du code javascript localement au navigateur par exemple.

Gestion des évènements

Lorsqu'un « PostBack » est effectué et qu'un évènement est précisé sur le contrôle à l'origine du « PostBack », ASP.NET doit déterminer la méthode du code behind à appeler lors du déroulement de son cycle de vie. Cette opération est réalisée grâce à deux champs cachés dans le formulaire HTML : « _EVENTTARGET » et « _EVENTARGUMENT » qui sont valorisés lors de l'appel à la fonction « _doPostBack() ». En effet, les contrôles font automatiquement appel à cette fonction dès lors que leur propriété « AutoPostBack » vaut « true ». Le contrôle passe en argument de cette fonction son identifiant ainsi que des éventuels arguments supplémentaires. Les deux champs sus-cités sont ensuite postés vers le serveur. ASP.NET se charge alors, en fonction de l'identifiant du contrôle, de déterminer la méthode à appeler ainsi que ses arguments.

Nous pouvons constater que le code de la fonction « _doPostBack () » est extrêmement simple :

```
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
```

Conservation de l'état de la page après un « PostBack »

HTTP est un protocole sans état. C'est à dire qu'entre deux appels successifs à une URL, le serveur n'a pas maintenu la valeur des variables utilisées. Comme nous venons de le voir, la technologie Webforms est basée sur la notion de « PostBack ». Les pages sont donc fréquemment ré-exécutées et l'enjeu est de ne pas perdre les valeurs des champs de formulaires saisies par l'utilisateur. Habituellement, il est nécessaire de mettre en place un mécanisme spécifique pour assurer cette persistance (utilisation de la session ou des cookies, recopie explicite des valeurs dans des champs cachés du formulaire). ASP.NET propose nativement une solution à cette problématique : le « ViewState ». L'idée est de sérialiser l'état de chaque contrôle ASP.NET géré par le serveur, dans un champ caché appelé « _VIEWSTATE ». Ce champ est transmis au serveur lors de chaque « PostBack ». ASP.NET est ensuite capable de restaurer dans le code behind l'état de chacun de ces contrôles. Ainsi les « Inputs » HTML de la page re-générée retrouvent leur état pré-soumission du formulaire.

Les contrôles ASP.NET possèdent une propriété « PostBackUrl » permettant de poster le formulaire vers une autre page côté serveur (pour permettre une navigation entre pages par exemple).

Il est bien sûr possible de désactiver le « ViewState » pour un contrôle ASP.NET particulier, dans le cas d'une simple consultation de données par exemple. Cela permet de ne pas alourdir le poids de la page HTML.

Data Binding*

Le rôle premier d'une page HTML est de présenter de l'information. Cette information peut provenir de différentes sources de données : objets, XML, fichiers, base de données… ASP.NET nous offre un mécanisme simple pour lier ces données et les contrôles ASP.NET chargés de les afficher : le « data binding ». Il peut être unidirectionnel (one way data binding) pour des données en lecture seule ou bidirectionnel (two ways data binding) pour des données en lecture/écriture.

Data binding unidirectionnel

Considérons le modèle suivant

```
public class Tweet
{
        public string Content { get; set; }
        public string UserID{ get; set; }

        public static List<Tweet> Tweets()
        {
                List<Tweet> tweets = new List<Tweet>();
                for (var i = 0; i < 5; i++)
                {
                        tweets.Add(new Tweet { Content = " This is a Random Post " + i, UserID = "User " + i });
                }
                return tweets;
    }
}
```

le code behind suivant :

```
public void Page_Load(Object sender, EventArgs e)
{
        var tweets = Tweet.Tweets();

        this.dropDownList.DataSource = tweets;
        this.dropDownList.DataTextField = "Content";
        this.dropDownList.DataValueField = "UserID";
        this.dropDownList.DataBind();
}
```

et le code aspx suivant :

```
<asp:DropDownList ID="dropDownList" runat="server" AutoPostBack="true" />
```

Le résultat dans un navigateur sera une liste déroulante contenant les couples valeur/texte suivants :

- valeur : « User 0 », texte affiché : « This is a Random Post 0 »,
- valeur : « User 1 », texte affiché : « This is a Random Post 1 »,
- valeur : « User 2 », texte affiché : « This is a Random Post 2 »,
- valeur : « User 3 », texte affiché : « This is a Random Post 3 »,
- valeur : « User 4 », texte affiché : « This is a Random Post 4 ».

Il est bien sûr possible de réaliser un « data binding » à partir de données persistées avec Entity Framework par exemple.

Considérons le « code behind » suivant qui charge une liste de Tweets classée par « UserID » :

```
protected void Page_Load(object sender, EventArgs e)
{
   TweetDatabaseEntities _context = new TweetDatabaseEntities();

   this.GridViewTweets.DataSource = _context.Tweets.OrderByDescending(t => t.UserID).ToList();
   this.GridViewTweets.DataBind();
}
```

et le code aspx suivant :

```
<asp:GridView ID="GridViewTweets" AutoGenerateColumns="false" runat="server" CellPadding="4" ForeColor="#333333"
   GridLines="None">
   <Columns>
      <asp:BoundField DataField="UserID" HeaderText="ID utilisateur" />
      <asp:BoundField DataField="Content" HeaderText=« Contenu du tweet" />
   </Columns>
</asp:GridView>
```

Le résultat dans un navigateur sera un tableau à deux colonnes :

ID utilisateur	Contenu du tweet
User 0	This is a Random Post 0
User 1	This is a Random Post 1
User 2	This is a Random Post 2
User 3	This is a Random Post 3
User 4	This is a Random Post 4

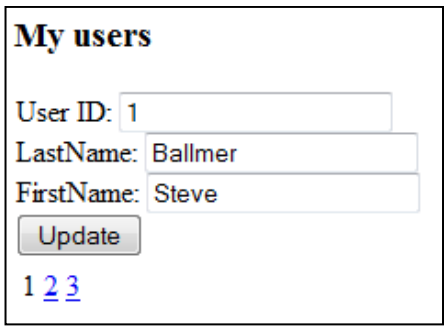
Data binding bidirectionnel

Considérons le code aspx suivant :

```
< asp:LinqDataSource ID="LinqDataSource" runat="server"
ContextTypeName="DataClassesDataContext" EntityTypeName="" EnableUpdate="true" TableName="user" />

< asp:FormView ID="FormView" runat="server"
DataKeyNames="id" DataSourceID="LinqDataSource" DefaultMode="Edit" AllowPaging="True">

< EditItemTemplate>
   User ID:
   < asp:TextBox ID="UserIdTxt" runat="server" Text='<%# Bind("id") %>' />

   < br />
   LastName:
   < asp:TextBox ID="LastNameTxt" runat="server" Text='<%# Bind("lastName") %>' />
   < br />
   FirstName:
   < asp:TextBox ID="FirstNameTxt" runat="server" Text='<%# Bind("firstName") %>' />

   < br />

   < asp:Button ID="UpdateButton" Text="Update" CommandName="Update" runat="server"/>
< /EditItemTemplate>

< /asp:FormView>
```

Il définit une source de données « LinqDataSource », liste d'objet « User » (avec trois propriétés : UserId, LastName et FirstName). Cette source de données s'appuie sur une table « user » en base de données.

Chacun des champs de la « FormView » est bindé aux propriétés des objets « User » de la source de données grâce à la syntaxe « <%# Bind(« propriété_user") %> ». Le mot clé « Bind » précise à ASP.NET le sens bidirectionnel du data binding. Ainsi le formulaire sera du type :



Les champs seront bien en lecture/écriture et le « PostBack » ira mettre à jour la table « user ».

ASP.NET AJAX*

Nous venons d'exposer le principe général de fonctionnement d'ASP.NET/Webforms. Il repose grandement sur la notion de « PostBack ». La conséquence de ce mode de fonctionnement est que l'intégralité des pages est régulièrement rechargée tandis que seule une fraction de la page doit être mise à jour. Cette problématique est traitée par ASP.NET AJAX.

AJAX (Asynchronous Javascript And XML) est une technologie standardisée basée sur l'envoi de requête « XmlHttpRequest* » vers le serveur. En retour le DOM* du document HTML est mis à jour pour ne refléter que les modifications sur les zones concernées.

ASP.NET masque cette potentielle complexité à l'aide du contrôle « asp:UpdatePanel ». Celui-ci entoure le bloc aspx concerné par la mise à jour partielle.

Considérons le code aspx suivant :

```
<form id="form1" runat="server">
    <asp:ScriptManager ID="MainScriptManager" runat="server" />
    <asp:UpdatePanel ID="pnlHelloWorld" runat="server">
      <ContentTemplate>
        <asp:Label runat="server" ID="lblHelloWorld" Text="Click the button!" />
        <br /><br />
        <asp:Button runat="server" ID="btnHelloWorld" OnClick="btnHelloWorld_Click" Text="Update label!" />
      </ContentTemplate>
    </asp:UpdatePanel>
  </form>
```

et le code behind correspondant :

```
protected void btnHelloWorld_Click(object sender, EventArgs e)
{
   lblHelloWorld.Text = "Hello, world - L'heure est actuellement : " + DateTime.Now.ToLongTimeString();
}
```

Lors du clic sur le bouton, seule la partie de la page HTML correspondant aux contrôles « btnHelloWorld » et « lblHelloWorld » sera rechargée. En réalité tout le « PostBack » est effectué, le code behind est normalement exécuté (tout son cycle de vie) mais seule la portion HTML voulue est fournie au navigateur. A noter que le contrôle « asp:ScriptManager » permet de préciser que la technologie AJAX va être utilisée.

Il est par ailleurs possible de déclencher le rafraichissement d'une portion de page via une action sur un objet ASP.NET (un bouton par exemple) ne se situant pas dans la définition de cette portion.

```
  <form id="form1" runat="server">
    <asp:ScriptManager ID="MainScriptManager" runat="server" />
    <asp:UpdatePanel runat="server" id="UpdatePanel1" updatemode="Conditional">
    <Triggers>
      <asp:AsyncPostBackTrigger controlid="UpdateButton2" eventname="Click" />
    </Triggers>
      <ContentTemplate>
        <asp:Label runat="server" id="DateTimeLabel1" />
        <asp:Button runat="server" id="UpdateButton1" onclick="UpdateButton_Click" text="Update" />
      </ContentTemplate>
    </asp:UpdatePanel>
    <asp:UpdatePanel runat="server" id="UpdatePanel1" updatemode="Conditional">
      <ContentTemplate>
        <asp:Label runat="server" id="DateTimeLabel2" />
        <asp:Button runat="server" id="UpdateButton2" onclick="UpdateButton_Click" text="Update" />
      </ContentTemplate>
    </asp:UpdatePanel>
  </form>
```

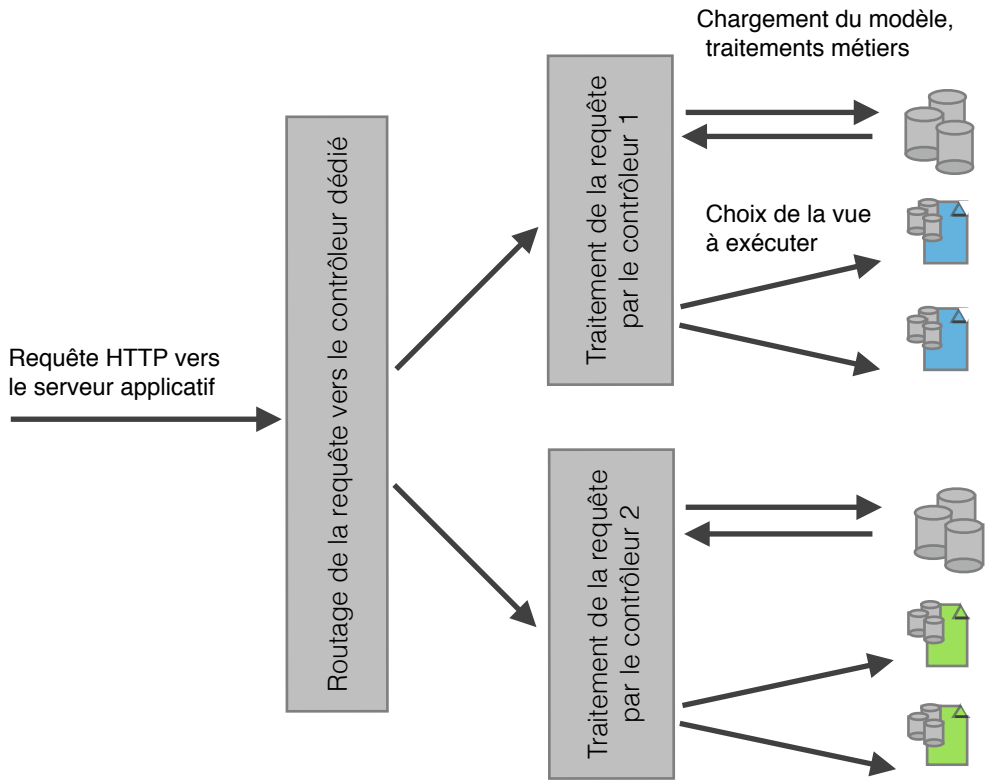
Un clic sur le bouton « UpdateButton1 » ne déclenchera que la mise au jour de la portion « UpdatePanel1 », tandis qu'un click sur le bouton « UpdateButton2 » déclenchera la mise à jour des deux portions « UpdatePanel1 » et « UpdatePanel2 ». Ceci est effectué au travers du contrôle « asp:AsyncPostBackTrigger ».

ASP.NET MVC*

ASP.NET MVC est une technologie récente, mise en avant par Microsoft. Elle permet le développement d'applications web en respectant le Design Pattern « Model View Controller », massivement utilisé par les développeurs dans d'autres environnements (notamment PHP et JEE). ASP.NET MVC permet une totale maîtrise du modèle HTTP ainsi que du code HTML généré. C'est un élément déterminant dans le cadre de sites web répondant aux contraintes du « Responsive Design » (comportements s'adaptant au type de navigateur : PC/ Smartphone/Tablette/Wearable). Le motif MVC, de par sa conception en couches applicatives, offre par ailleurs une bonne évolutivité dans le cadre de sites web conséquents. Il permet de mettre en place une conception axée sur les tests (ce que ne permet pas Webforms).

A la création d'un projet ASP.NET MVC, Visual Studio crée par défaut trois répertoires correspondant aux trois couches du Design Pattern : « Controllers », « Models » et « Views ».

Comme avec tous les frameworks permettant de mettre en place le motif MVC, la cinématique de traitement d'une requête sera la suivante :



Principes de fonctionnement

Routage des requêtes

Le fichier « App_Start/RouteConfig.cs » permet de décrire les routes. Il contient, par exemple, le code suivant :

```
routes.MapRoute(
   name: "Default",
   url: "{controller}/{action}/{id}",
   defaults: new { controller = "Client", action = "Index", id = UrlParameter.Optional }
);
```

Cette déclaration signifie qu'une URL formatée de cette façon : http://monsite/Client/Hello/Bruno sera interprétée ainsi :

- le contrôleur « ClientController » sera instancié,
- la méthode « Hello() » sera invoquée sur ce contrôleur,
- l'argument « Bruno » sera passé à cette méthode.

L'appel de l'URL http://monsite/ sera transformé par défaut en http://monsite/Client/Index/ qui sera interprété ainsi :

- le contrôleur « ClientController » sera instancié,
- la méthode « Index() » sera invoquée sur ce contrôleur,
- aucun argument ne sera passé à cette méthode.

Le contrôleur

Un contrôleur « ClientController » doit être créé dans le répertoire « Controllers ». Son code pourrait être :

```
public class ClientController : Controller
{

public String Index()
   {
      return "Accueil du site »;
   }

public String Hello(String id)
   {
      return "Hello "+id;
   }
}
```

L'appel suivant « http://monsite/Client/Hello/Patrou » produit le résultat : « Hello Bruno ». A ce stade, nous avons mis en place le C du motif MVC (à noter qu'ASP.NET MVC est compatible avec le format d'adressage REST de type URI* : http://monsite/a/b/c/…).

La vue

Dans l'exemple précédent, il n'était pas réellement question de vue, le contrôleur n'écrivant simplement qu'une chaîne de caractère dans la réponse HTTP. Dans une application Web, la vue consiste généralement en une page HTML. Nous allons modifier le code du contrôleur afin que ce dernier exécute la vue correspondant à la requête de l'utilisateur.

Modifions le code du contrôleur ainsi :

```
public ActionResult Hello(String id)
{
   return View("Hello");
}
```

puis créons un fichier « Views/Client/Hello.cshtml » qui contiendra :

```
@{
   Layout = null;
}
<!DOCTYPE html>
<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <title>Hello</title>
</head>
<body>
   <div>
      Hello Client !
   </div>
</body>
</html>
```

L'appel de la requête « http://monsite/Client/Hello/Bruno » produira dorénavant une page HTML qui affichera « Hello Client ! ». Le contrôleur charge la page située dans le sous-répertoire de « Views » ayant le même nom que lui (ici Client) et dont le nom correspondant à l'identifiant désigné par l'instruction « return View("Hello") » : « Hello.cshtml ». La vue ne consiste pas simplement en un fichier HTML, il s'agit en réalité d'un fichier pouvant également contenir du code C# (comme son extension l'indique). A ce stade nous avons mis en place le C et le V du motif MVC.

Le modèle

L'exemple précédent nous a montré comment afficher une vue, il est cependant extrêmement limité puisqu'il s'agit d'une page entièrement statique. Nous allons créer et afficher un modèle objet minimaliste dans cette celle-ci grâce à un dictionnaire de paires clé/valeur : le « ViewData ».

Créons la classe « Client » dans le répertoire « Models » :

```
public class Client
{
   public String Name { get; set; }
   public int Age { get; set; }

public static List<Client> GetClientsList()
   {
      return new List<Client>
      {
         new Client { Age = 40, Name = "Bruno"},
         new Client { Age = 35, Name = "Thibault"},
         new Client { Age = 36, Name = "Helene"},
         new Client { Age = 65, Name = "Simon"}
      };
   }
}
```

Modifions le « ClientController » ainsi :

```
public ActionResult ClientSearch(String id)
{
   ViewData["Name"] = id;
   Client client = Client.GetClientsList().FirstOrDefault(c => c.Name == id);
   if (client != null)
   {
      ViewData["Age"] = client.Age;
      return View("Client");
   }
   return View("NoClient");
}

public ActionResult ClientsList()
{
   ViewData["Clients"] = Client.GetClientsList();
   return View("ClientsList");
}
```

Une méthode permet dorénavant de lister tous les clients du modèle objet et une seconde d'afficher le détail d'un client en fonction de son identifiant. Nous devons créer trois vues correspondant aux trois cas possibles :

Views/Client/Client.cshtml :

```
<body>
   <div>
      Le client @ViewData["Name"] a @ViewData["Age"] ans.
   </div>
</body>
```

Views/Client/NoClient.cshtml :

```
<body>
   <div style="color:red">
      Le client @ViewData["Name"] n'a pas été trouvé !
   </div>
</body>
```

Views/Client/ClientsList.cshtml :

```
<body>
   <table>
      <tr>
         <th> Name </th>
         <th>Age</th>
      </tr>
      @foreach (HelloClient.Models.Client client in ViewData["Clients"] as List<HelloClient.Models.Client>)
      {
         <tr>
            <td>@client.Name </td>
            <td>@client.Age</td>
         </tr>
      }
   </table>
</body>
```

Il est désormais possible de manipuler le modèle objet dans le contrôleur et de l'afficher dans la vue, le « ViewData » permettant de faire le lien entre les deux. A noter que les instructions C# au sein de la vue son préfixées par « @ ».

L'appel à l'URL « http://monsite/Client/ClientSearch/Bruno » produira la page HTML affichant :

Le client Bruno a 40 ans.

L'appel à l'URL « http://monsite/Client/ClientSearch/Anakin » produira la page HTML affichant :

Le client Anakin n'a pas été trouvé !

L'appel à l'URL « http://monsite/Client/ClientsList » produira la page HTML affichant :

Name	Age
Bruno	40
Thibault	35
Helene	36
Simon	65

Nous avons bien mis en place les trois composantes du motif MVC.

Pour aller un peu plus loin

Les routes :

Il est bien sûr possible de déclarer plusieurs routes afin de découper l'application en domaines fonctionnels (et autant de contrôleurs). Nous pourrions modifier le fichier « App_Start/RouteConfig.cs » ainsi :

```
routes.MapRoute(
   name: "Client",
   url: « {controller}/{action}/{age}/{name} ,
   defaults: new { controller = "Client", action = "Add", age =  "32",  name = "Ethan"}
)

routes.MapRoute(
   name: "Default",
   url: "{controller}/{action}/{id}",
   defaults: new { controller = "Client", action = "Index", id = UrlParameter.Optional }
)
```

Il conviendrait d'ajouter la méthode suivante au ClientController :

```
public ActionResult Add(int age, String name)
{
   ViewData["Name"] = name;
   ViewDate["Age"] = age;
   Client client = new Client(Age = age, Name=name);

   // ici on ajoute le code de persistance par exemple

   return View("Client");
}
```

L'appel à l'URL « http://monsite/Client/Add/Luke/54 » ajoutera un nouveau client au référentiel de données de notre application puis affichera le résultat : « Le client Luke a 54 ans ». A noter qu'il est possible d'affecter des contraintes de type et de formatage sur les paramètres des URL, tout comme il est possible de préciser un nombre indéfini de paramètres.

Le modèle :

Nous avons vu qu'il était possible de passer des objets du modèle vers la vue au travers du « ViewData ». Cette méthode peut être rébarbative lorsqu'il s'agit d'afficher des objets complexes dans la vue. Il est alors préconisé de mettre en place le concept de « ViewModels » (modèle de vues). Nous pouvons tout à fait rapprocher ce concept de la notion de DTO (Data Transfert Objet) dont le but est d'agréger toutes les informations du modèle de façon adaptée à leur affichage.

Dans notre exemple, si nous souhaitons afficher le détail d'un client mais également l'heure courante et la météo dans une unique page, nous allons définir le modèle de vue suivant dans le répertoire « ViewModels » :

```
public class ClientViewModel
{
   public int Age { get; set; }
   public String Name { get; set; }
   public DateTime CurrentDate { get; set; }
   public String Weather { get; set; }
}
```

Le code de ClientController sera dorénavant :

```
public ActionResult ClientSearch(String id)
{
   ViewData["Name"] = id;
   Client client = Client.GetClientsList().FirstOrDefault(c => c.Name == id);
   if (client != null)
   {
      ClientViewModel cvm = new ClientViewModel {
            Age = client.Age,
            Name = id,
            CurrentDate = DateTime.Now,
            Weather = "It's raining !"
            };

      return View(cvm);
   }
   return View("NoClient");
}
```

Créons ensuite la vue « Views/ClientSearch.cshtml » :

```
@model HelloWorld.ViewModels.ClientViewModel
...
<ul>
   <li>nom : @Model.Name</li>
   <li>age : @Model.Age</li>
   <li>date : @Model.CurrentDate.ToString(« dd/MM/yyyy")</li>
   <li>météo : @Model.Weather</li>
</ul>
```

L'appel à l'URL « http://monsite/Client/ClientSearch/Bruno » produira la page HTML affichant :

- nom : Bruno
- age : 40 ans
- date : 04/01/2015
- météo : It's raining !

Les Helpers :

Tout comme « ASP.NET Webforms » propose des contrôles qui, une fois interprétés par le serveur, produisent du HTML, ASP.NET MVC offre une possibilité similaire avec les « Helpers ». Quelques exemples :

Helper	HTML
@Html.TextBox("nom", Model.Name)	<input id="nom" name="nom" type="text" value="Bruno" />
@Html.Label("Majeur", "Cochez si vous êtes majeur") @Html.CheckBox("Majeur", true)	<label for="Majeur">Cochez si vous êtes majeur</label> <input checked="checked" id="Majeur" name="Majeur" type="checkbox" value="true" /> <input name="Majeur" type="hidden" value="false" />
@Html.DropDownList("Clients", (SelectList)ViewData.["Clients"])	<select id="Clients" name="Clients"> <option value="1">Bruno</option> <option value="2">Thibault</option> <option value="3">Hélène</option> <option value="4">Simon</option> </select>
@{Html.BeginForm("Add", "Client");} ... @{Html.EndForm();}	<form action=« /Client/Add method="post"> ... </form>

Il en existe de nombreux et nous ne serons pas exhaustifs dans ce document.

Comparaison avec JEE

Webforms permet :

- de dissocier code de présentation et code métier,
- de conserver l'état des « inputs » HTML d'une requête à une autre,
- de faciliter l'intégration du code .NET grâce au mécanisme de « code behind » et au modèle évènementiel,
- de concevoir des pages HTML de façon concise grâce aux nombreux contrôles ASP.NET disponibles,
- de lier les données métiers et les pages HTML de présentation grâce au « data binding »,
- de créer ses propres contrôles,
- d'optimiser l'exécution des pages grâce à la compilation en code intermédiaire à la première consultation.

La comparaison de la technologie Webforms avec C++ fait peu de sens, tant ce langage n'est traditionnellement pas associé au développement Web. Nous allons donc nous focaliser sur les fonctionnalités intégrées à la plateforme JEE.

JEE intègre les spécifications suivantes :

- JSP (Java Server Pages) : technologie permettant l'écriture de code Java au sein d'une page HTML. Des balises (« Tags ») permettent l'accès aux objets Java (« beans ») et à leurs propriétés. Une page JSP est compilée à sa première exécution.
- JSTL (JSP Standard Tag Library) : technologie de balisage permettant :
  - d'exprimer des logiques de traitement (conditions, boucles),
  - de préciser des instructions de formatage de données,
  - de gérer l'internationalisation,
  - d'accéder à des sources de données SQL et XML.
- EL (Expression langage) : syntaxe permettant l'accès aux méthodes et propriétés d'objets Java placés dans un des scopes applicatifs (requête, session, application).

Ces technologies ne couvrent pas l'intégralité du spectre de fonctionnalités offertes par les Webforms. Nous nous rapprochons en fait de ce que permettait la version originale d'ASP. Afin de palier à ce déficit de fonctionnalités de haut niveau, JEE a introduit JSF* (Java Server Faces).

JSF 2.2 fait partie de la spécification JEE7. C'est une implémentation du motif de conception MVP (Model View Presenter). Dans ce motif, le « Presenter » joue le rôle de passerelle entre l'écran (la Vue) et le métier de l'application (le Modèle). Ainsi la vue est écrite sans code Java et le modèle ignore totalement qu'il est manipulé dans un contexte web. La technologie préconisée pour le rendu des pages est « facelets ». Elle permet d'une part de décrire les pages dans une syntaxe XHTML et d'autre part de développer ses propres composants réutilisables.

Principes de fonctionnement de JSF

Le « Presenter » est un POJO dont les attributs et méthodes sont associés à des éléments de la vue. Un attribut est associé à un champ de formulaire tandis qu'une méthode est liée à une action réalisée sur un contrôle HTML (de type « bouton » par exemple). Cette liaison bidirectionnelle est réalisée grâce à JSF EL (Expression Language).

A l'affichage du formulaire, JSF lit la valeur des attributs du « Presenter » et les affiche dans les champs de celui-ci. A la soumission, JSF lit les valeurs saisies dans ces champs et accède en écriture aux attributs du « Presenter ». Ce mécanisme de liaison bidirectionnelle est appelé « two ways binding ».

A la différence d'un motif MVC, au sein duquel le « Controller » est le point d'entrée des requêtes HTTP, c'est ici la « Vue » qui répond aux sollicitations de l'utilisateur en s'appuyant sur le « Presenter ».

Dans l'exemple suivant, nous allons mettre en place un formulaire permettant d'effectuer une recherche de clients.

Considérons le « Presenter » :

```
@Model // Cette annotation permet de préciser au conteneur JEE qu'il s'agit d'un « Bean managé »
public class CustormerFinderPresenter {

	private String name;
	private List<Customer> customers;

	// Considérons que les « getter » et « setter » de ces attributs sont également déclarés ici

	public void find () {
		this.customers = new CustomerDAO.find (this.name);
	}
}
```

et la vue suivante :

```
<html xmlns="http://www.w3.org/1999/xhtml"  xmlns:h="http://java.sun.com/jsf/html">

        <h:form>
                <h:inputTextValue="#{customerFinderPresenter.name}" />
                <h:commandButton actionListener="#{customerFinderPresenter.find}" action="anotherPage? faces-redirect=true"/>
        </h:form>

</html>
```

Ici, le clic sur le bouton « submit » déclenchera :
- la valorisation de l'attribut « name » du « Presenter » avec la valeur saisie dans « l'input HTML »,
- l'appel de la méthode « find » du « Presenter ».

Le résultat de la méthode « find » pourrait ensuite être exploité dans « anotherPage » avec la commande « customerFinderPresenter.customers ».

Il existe de nombreux « tags » JSF, parmi lesquels :
- « h:form », « h:inputText », « h:dataTable », … permettront d'agir sur le « two ways binding »,
- « f:view », « f:ajax », « f:event », … permettront d'agir sur le rendu de la page (internationalisation, cycle de vie de la page, …),
- « f:viewParam » permettra de remonter un paramètre de la requête HTTP vers le « Presenter ».

Tout comme le « code behind » de la technologie Webforms respecte un cycle de vie qui lui est propre, JSF permet également d'associer des évènements de chargement de la page à du code côté « Presenter ». Enfin, des annotations sur les attributs et méthodes du « Presenter » permettent d'assurer la validation des données entrantes.

JSF permet :
- de dissocier code de présentation et code métier,
- de conserver l'état des « inputs » HTML d'une requête à une autre,
- de faciliter l'intégration du code Java grâce au mécanisme de « Presenter » et au modèle évènementiel,
- de concevoir des pages HTML de façon concise grâce aux nombreux « Tags » disponibles,
- de lier les données métiers et les pages HTML de présentation grâce au « two ways binding »,
- de créer ses propres composants graphiques,
- d'optimiser l'exécution des pages grâce à la compilation en code intermédiaire à la première consultation.

Nous percevons ici la volonté de la plateforme JEE de se rapprocher des fonctionnalités offertes par ASP.NET Webforms.

En ce qui concerne le motif MVC, JEE ne spécifie pas de solution. Cependant de nombreux frameworks permettent de mettre en oeuvre ce principe. Que ce soit avec Struts ou Spring MVC le principe de fonctionnement est le suivant :
- fourniture par le framework d'une Servlet jouant le rôle d'aiguilleur de requêtes,
- déclaration d'un mapping (exploité par la Servlet d'aiguillage) décrivant les contrôleurs et vues associés à une requête,
- constitution du modèle objet en s'appuyant sur des couches de services,
- enfin, exploitation du modèle dans les vues grâce à un langage de type « EL ».

Nous retrouvons donc les grands principes de fonctionnement d'ASP.NET MVC.

7.Architecture Orientée Service

Dans un environnement .NET, la conception d'une architecture orientée service repose en partie sur l'utilisation de composants WCF (Windows Communication Foundation). D'autres solutions telles que « Remoting » (appel d'objets distants) peuvent également être mises en oeuvre. Nous nous focaliserons cependant sur WCF puisque cette technologie permet d'implémenter des services web (notamment des services REST). Cela nous fournit un point de comparaison intéressant puisqu'il s'agit d'une technologie standardisée que l'on retrouve sur beaucoup de plateformes.

Les étapes de mise en œuvre d'un service WCF sont les suivantes :

- définition du contrat : spécification des signatures des méthodes du service WCF, et du format des données à échanger,
- implémentation du contrat : implémentation du service,
- configuration du service : définition des « EndPoints » (points d'accès au service).

Le contrat de service

Le contrat de service est l'entité qui va :

- être échangée entre le serveur et le client,
- permettre au client de savoir quelles sont les méthodes proposées par le service et comment les appeler.

L'élaboration d'un contrat de service s'effectue au travers des trois métadonnées suivantes (sous la forme d'annotation des méthodes et des attributs) :

- ServiceContract : métadonnée attachée à la définition d'une classe ou d'une interface. Elle permet d'indiquer que la classe ou l'interface est un contrat de service,
- OperationContract : métadonnée attachée aux méthodes que l'on souhaite exposer au travers du service web. Il est donc possible de n'exposer au client que certaines méthodes d'une classe,
- DataMember : métadonnée attachée aux propriétés de la classe respectant le contrat de service. Ces propriétés sont passées en paramètres du service ou retournées pour celui-ci.

Considérons le contrat de service suivant :

```
namespace CustomerService
{
   [ServiceContract]
   public interface ICustomerService
   {
            [OperationContract]
            List<Customer> GetCustomerList();

            [OperationContract]
            bool AddCustomer(string Name, string Address);
   }
}
```

Il expose, par le biais des annotations « [ServiceContract] » et « [OperationContract] », un service permettant de lister des clients et d'en ajouter un.

Implémentation du service

Une fois la définition du contrat réalisée, nous devons créer une classe l'implémentant. Celle-ci n'est en aucun cas liée à la notion de service web, il s'agit d'un POCO. Elle n'est exposée sous cette forme qu'au travers des métadonnées utilisées dans l'interface qu'elle implémente.

Considérons l'implémentation suivante :

```
public class CustomerService : ICustomerService
{
	private CustomerDAO DAO; // couche de service permettant la manipulation du modèle objet

	List<Customer> GetCustomerList(){
		return DAO.FindAllCustomers();
	}

	bool AddCustomer(string Name, String Address){
		return DAO.Add(Name, Address);
	}
}
```

Puisque ce service renvoie une liste de clients, il est nécessaire de préciser quelles sont les propriétés de la classe « Customer » qui devront être sérialisées. Cela s'effectue au travers des annotations « [DataContract] » et « [DataMember] ».

Considérons la classe « Customer » :

```
[DataContract]
public class Customer
{
	[DataMember]
	private string Name {get;set;}

	[DataMember]
	private string Address {get;set;}
}
```

Configuration du service

Afin de publier le service web, il est nécessaire de le décrire dans un fichier de configuration (le fichier web.config d'une application ASP.NET par exemple). On y décrira :

- le/les « endpoints », c'est à dire les URL au travers desquelles le service sera accessible,
- le/les « behaviours », qui définissent le comportement du service côté serveur.

Dans notre exemple, la configuration pourrait être la suivante :

```
<system.serviceModel>
<services>
        <service behaviorConfiguration="CustomerServiceBehavior" name="CustomerServiceImpl">
                <endpoint address="customer" binding="wsHttpBinding" contract="CustomerService.ICustomerService">
                        <identity>
                                <dns value="localhost" />
                        </identity>
                </endpoint>
                <baseAddresses>
                        <add baseAddress="http://localhost:1701/ApplicationService"/>
                </baseAddresses>
        </service>
</services>
<behaviors>
        <serviceBehaviors>
                <behavior name="CustomerServiceBehavior">
                        <serviceMetadata httpGetEnabled="True"/>
                        <serviceDebug includeExceptionDetailInFaults="False" />
                </behavior>
        </serviceBehaviors>
</behaviors>
</system.serviceModel>
```

Une lecture rapide de ce paramétrage nous informe que le service web sera accessible via l'URL « http://localhost:1701/ApplicationService/customer » (concaténation de « baseAddress » et de « address ») et que son comportement sera notamment de ne pas envoyer les détails des exceptions levées par le service vers ses clients.

REST*

Afin de rendre le service web compatible avec REST, il convient de modifier, d'une part le contrat de service, et d'autre part son descripteur de déploiement. En effet, REST impose de spécifier les ressources (URI) identifiant les éléments de l'application, ainsi que le format des données échangées.

Le contrat de service deviendra :

```
namespace CustomerService
{
   [ServiceContract]
   public interface ICustomerService
   {
        [OperationContract]
        [WebGet(UriTemplate  = "list/")] ou [WebGet(UriTemplate  ="list/", ResponseFormat=WebMessageFormat.Json)] pour du JSON
        List<Customer> GetCustomerList();

        [OperationContract]
        [WebInvoke(UriTemplate = "add/{Name}/{Address}" )]
        bool AddCustomer(string Name, string Address);
   }
}
```

Avec REST, il ne s'agit pas, pour le client, d'appeler des méthodes « distantes » mais simplement d'avoir accès à des ressources sous forme d'URI.

L'annotation « WebGet » précise que l'appel avec une requête HTTP de type « GET » à l'URI « http://localhost:1701/ApplicationService/customer/list » donnera lieu à la production d'un contenu REST (XML ou JSON) décrivant la liste des clients.

L'annotation « WebInvoke » précise que l'appel avec une méthode HTTP de type « POST » à l'URI « http://localhost:1701/ApplicationService/customer/add/Bruno/Rouen » donnera lieu à l'ajout d'un client « Bruno » à « Rouen ».

Il reste simplement à adapter le fichier de déploiement de la façon suivante :

```
<system.serviceModel>
<services>
	<service behaviorConfiguration="CustomerServiceBehavior" name="CustomerServiceImpl">
		<endpoint address="customer" binding="webHttpBinding" contract="CustomerService.ICustomerService"
		behaviorConfiguration="CustomerServiceBehaviorRest" >
			<identity>
				<dns value="localhost" />
			</identity>
		</endpoint>
		<baseAddresses>
			<add baseAddress="http://localhost:1701/ApplicationService"/>
		</baseAddresses>
	</service>
</services>
<behaviors>
	<serviceBehaviors>
		<behavior name="CustomerServiceBehavior">
			<serviceMetadata httpGetEnabled="True"/>
			<serviceDebug includeExceptionDetailInFaults="False" />
		</behavior>
	</serviceBehaviors>
	<endpointBehaviors>
		<behavior name="CustomerServiceBehaviorRest">
			<webHttp/>
		</behavior>
	</endpointBehaviors>
</behaviors>
</system.serviceModel>
```

Comparaison avec JEE et C++

JEE

JEE intègre nativement les librairies permettant de mettre en place des architectures orientées services :

- JAX-WS : pour les services web traditionnels,
- JAX-RS : pour les services REST,
- nous pourrions également citer JMS (Java Messaging Service), EJB Remote (Enterprise Java Bean), RMI (Remote Method Invocation) ou encore JMX (Java Management eXtensions).

Pour plus de simplicité, nous nous focaliserons ici sur la mise en oeuvre d'un service REST (l'API JAX-RS n'est en fait qu'une spécification décrivant notamment des annotations. Il existe de nombreuses implémentations, parmi lesquelles Jersey, CXF ou RESTeasy).

De la même façon qu'avec la plateforme .NET dans laquelle un service REST est avant tout un POCO, avec JEE il s'agira naturellement d'un POJO.

Considérons le service REST :

```
@Path("/customer")
@Produces("Application/json")
public class CustomerService
{
	private CustomerDAO DAO; // couche de service permettant la manipulation du modèle objet

	@Path("/list")
	@GET
	List<Customer> getCustomerList() {
		return DAO.findAllCustomers();
	}

	@Path("/add/{name}/{address}")
	@POST
	bool addCustomer(@PathParam("name") String name, @PathParam("address") String address) {
		return DAO.add(name, address);
	}
}
```

Nous pouvons décrypter rapidement cette implémentation :

- L'URI de base du service REST sera : « http://monserveur/customer ». L'annotation @Path sur la classe permet de la spécifier,
- L'URI permettant de récupérer la liste des clients sera : « http://monserveur/customer/list ». L'annotation @Path sur la méthode « getCustomerList » permet de la spécifier (relativement à l'URI de base),
- L'URI permettant d'ajouter un client sera par exemple : « http://monserveur/customer/add/Bruno/Rouen ». L'annotation @Path sur la méthode « addCustomer » permet de la spécifier (relativement à l'URI de base). Les paramètres de cette URI sont précisés grâce aux mots clés {name} et {address} que l'on n'oubliera pas d'identifier en tant que paramètres de la méthode avec les annotations @PathParam,
- Les données retournées au client du service le seront au format JSON (annotation @Produces),
- Enfin, les verbes HTTP sont précisés grâce aux annotations @GET et @POST.

Nous retrouvons, aux annotations prêts, un fonctionnement identique à ce que propose la plateforme .NET.

C++

C++ n'ayant pas été conçu pour développer des applications web, il n'offre aucune solution native permettant la mise en place de services web. Au delà de la simple question d'une librairie adaptée, se pose surtout le problème de l'hébergement de tels services. Des solutions telles qu'HydraExpress (RogueWave Software) se proposent de prendre en charge ces deux problématiques. Du fait de l'aspect « propriétaire » de cet environnement, nous ne rentrerons pas dans le détail d'implémentation d'un service web en C++. Nous pouvons simplement retenir qu'il passe par l'écriture du fichier WSDL* (Web Services Description Language) décrivant le contrat de service, puis par un générateur de code produisant les classes clientes et serveurs. Il ne propose donc pas la simplicité et la flexibilité qu'offrent les plateformes .NET et JEE (les POCO/POJO n'ont aucune adhérence avec les APIs web).

8.Développement d'applications clientes

WPF (Windows Presentation Foundation) remplace les Winforms depuis le framework .NET 3.0. Il comporte tous les éléments (contrôles graphiques, librairies de base, motifs de conception) pour construire des applications clientes.

WPF propose le concept de « modèle d'application ». Ainsi, une application peut aussi bien être interprétée comme un client lourd (possédant ses propres fenêtres) ou s'afficher dans un navigateur. Ce dernier est appelé modèle d'application « XBAP » (XAML* Browser APplication). S'exécutant dans un navigateur, l'application sera contrainte par des règles de sécurité strictes et sera exécutée dans une sandbox (bac à sable).

WPF utilise d'une part le XAML (eXtensible Application Markup Language) pour décrire les interfaces graphiques et d'autre part le « code-behind » pour décrire le comportement. La liaison entre les deux s'effectue au travers d'un modèle évènementiel et d'un « data binding ».

XAML

Le XAML repose sur une syntaxe XML. Il propose de nombreux contrôles (TextBox, CheckBox, Button, RadioButton, Calendar, DatePicker, …), une gestion de layouts* (Canvas, DockPanel, Grid, …) et des librairies de manipulation d'objets 2D, 3D, d'animations et de données multimédia.

Voici un exemple minimaliste de la définition d'une fenêtre en XAML :

```
<Window x:Class="MyApplication.MyWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="EventsSample" Height="300" Width="300">
   </Grid>
</Window>
```

L'attribut « x:Class » précise quelle sera la classe de « code-behind » correspondante , ici, MyWindow :

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace MyApplication
{
   public partial class MyWindow : Window
   {
      public MyWindow()
      {
         InitializeComponent();
      }
   }
}
```

Nota bene : il s'agit d'une classe partielle puisqu'elle sera complétée à l'exécution avec la classe déduite du fichier XAML.

Modèle évènementiel

A l'image de ce qui existe en ASP.NET, avec WPF il est possible d'associer une action sur un contrôle avec une méthode du « code-behind ». Considérons le code XAML suivant :

```
<Window x:Class="MyApplication.MyWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EventsSample" Height="300" Width="300">
    <Grid Name="pnlMainGrid" MouseUp="pnlMainGrid_MouseUp" Background="LightBlue">
  </Grid>
</Window>
```

L'évènement « MouseUp » sur le composant « Grid » va déclencher l'appel de la méthode « pnlMainGrid_MouseUp » du « code-behind ».

Data Binding

WPF permet d'établir un « data binding » entre la vue et son « code-behind ».

Celui-ci s'effectue au travers de deux étapes :

- lier le contrôle XAML et la source de données,
- refléter les modifications apportées à la source de données au niveau de la vue.

L'exemple suivant permet l'ajout, la suppression et la modification d'utilisateurs dans une liste.

Considérons le XAML suivant :

```
<Window x:Class="MyAppliction.MyWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MyWindowNotification" Height="150" Width="300">
    <DockPanel Margin="10">
        <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
            <Button Name="btnAddUser" Click="btnAddUser_Click">Add user</Button>
            <Button Name="btnChangeUser" Click="btnChangeUser_Click" Margin="0,5">Change user</Button>
            <Button Name="btnDeleteUser" Click="btnDeleteUser_Click">Delete user</Button>
        </StackPanel>
        <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
    </DockPanel>
</Window>
```

Cette vue définit :

- un bouton « btnAddUser ». Un clic sur ce bouton déclenchera la méthode « btnAddUser_Click »,
- un bouton « btnChangeUser ». Un clic sur ce bouton déclenchera la méthode « btnChngeUser_Click »,
- un bouton « btnDeleteUser ». Un clic sur ce bouton déclenchera la méthode « btnDeleteUser_Click »,
- une liste « lblUsers » affichant la propriété « name » de sa source de données.

Le « code-behind » correspondant définit d'une part la fenêtre et d'autre part le modèle objet « User » :

```
namespace MyApplication
{
    public partial class MyWindow : Window
    {
        private ObservableCollection<User> users = new ObservableCollection<User>();

        public ChangeNotificationSample()
        {
            InitializeComponent();

            users.Add(new User() { Name = "Bruno Patrou" });
            users.Add(new User() { Name = "James Kirk" });

            lbUsers.ItemsSource = users;
        }

        private void btnAddUser_Click(object sender, RoutedEventArgs e)
        {
            users.Add(new User() { Name = "New user" });
        }

        private void btnChangeUser_Click(object sender, RoutedEventArgs e)
        {
            if(lbUsers.SelectedItem != null)
                (lbUsers.SelectedItem as User).Name = "Random Name";
        }

        private void btnDeleteUser_Click(object sender, RoutedEventArgs e)
        {
            if(lbUsers.SelectedItem != null)
                users.Remove(lbUsers.SelectedItem as User);
        }
    }

    public class User : INotifyPropertyChanged
    {
        private string name;
        public string Name {
            get { return this.name; }
            set
            {
                if(this.name != value)
                {
                    this.name = value;
                    this.NotifyPropertyChanged("Name");
                }
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        public void NotifyPropertyChanged(string propName)
        {
            if(this.PropertyChanged != null)
                this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
        }
    }
}
```

Le fonctionnement est le suivant :

- la source de données est la liste « users » de type « ObservableCollection ». Ce simple typage précise que tout changement qui lui est apporté sera reflété dans les composants qui l'écoutent, en l'occurence la « ListBox » (ainsi l'ajout et la suppression seront bien pris en compte). Cela ne suffit cependant pas à refléter les modifications faites sur les utilisateurs déjà présents dans « users », puisqu'ils ne sont pas « bindés » directement (c'est la liste qui l'est),
- un motif de conception de type « Observer » va donc être mis en place au travers de l'implémentation de l'interface « INotifyPropertyChanged » par la classe « User ». Un évènement sera émis à chaque changement de nom d'un utilisateur, et tous les composants qui l'écoutent seront notifiés.

Le « data binding » peut également être réalisé au travers d'un « DataContext ». Ce fonctionnement fait partie du motif de conception MVVM* (Modèle, Vue, VueModèle). Le « VueModèle » représente un POCO chargé de regrouper toutes les informations (depuis les différents objets du modèle) qui seront « bindées » sur la vue. Dans le XAML, la syntaxe « {Binding Name} » associera la valeur de l'attribut « Name » du « DataContext » au contrôle (un « TextBox » par exemple) déclaré dans la vue.

Commandes

Dans les applications clientes, il est fréquent que plusieurs éléments d'IHM (menu principal, boîte à outils, menu contextuel, …) déclenchent la même action. Le modèle évènementiel basique de WPF est peu adapté à cette contrainte. En effet, il imposerait de déclarer le même évènement à plusieurs endroits de la vue (avec des risques de duplication du « code-behind »). WPF remédie à ce problème au travers du motif de conception « Command ». Une commande permet, d'une part de centraliser l'exécution du code (méthode « Execute() »), et d'autre part, de savoir si cette action peut être exécutée (méthode « CanExecute() »). Cela permet de désactiver les éléments d'IHM associés à une commande dans le cas où l'exécution de celle-ci n'est pas autorisée. Plus de 100 commandes sont fournies avec le framework .NET.

Considérons le code XAML suivant :

```
<Window x:Class="MyApplication.MyWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UsingCommandsSample" Height="100" Width="200">
  <Window.CommandBindings>
                <CommandBinding Command="ApplicationCommands.New" Executed="NewCommand_Executed"
CanExecute="NewCommand_CanExecute" />
  </Window.CommandBindings>

  <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
    <Button Command="ApplicationCommands.New">New</Button>
  </StackPanel>
</Window>
```

Il met en évidence le mécanisme de « binding » de commande. Celui-ci permet de lier : le déclencheur de la commande (ici le bouton) aux méthodes appelées pour exécuter cette commande (ici la méthode « NewCommand_CanExecute » pour s'assurer que l'exécution est possible, et la méthode « NewCommand_Executed » pour effectuer le traitement). Ces méthodes sont déclarées dans le « code-behind » de la vue.

Le « code-behind » correspondant sera celui-ci :

```
namespace MyApplication
{
    public partial class MyWindow : Window
    {
        public UsingCommandsSample()
        {
            InitializeComponent();
        }

        private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }

        private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            MessageBox.Show("The New command was invoked");
        }
    }
}
```

Comparaison avec Java

AWT, SWING et SWT

Historiquement, la plateforme Java propose deux librairies graphiques : AWT* (Abstract Window Toolkit) et SWING*. Aucune des deux n'offre les fonctionnalités de haut niveaux que l'on peut attendre de frameworks de présentation modernes tel que WPF. Par ailleurs, étant complètement indépendantes de leur plateforme d'exécution, elles ne tirent pas partie des composants graphiques propres aux systèmes d'exploitation récents.

Dans les années 2000, IBM a initié SWT* (Standard Widget Toolkit). Cette librairie n'est pas un standard Java mais elle présente l'avantage de pouvoir bénéficier des ressources graphiques du système d'exploitation hôte de la machine virtuelle. Il y a une contrepartie à cette particularité : il existe une version de la librairie spécifique à chacun de ces systèmes d'exploitation. SWT est la librairie sur laquelle s'appuie RCP* (Rich Client Platform). Cet plateforme de développement d'applications clientes repose sur la notion de plugins, c'est son principal intérêt. Il est ainsi possible de développer des applications complètement modulaires en profitant notamment des plugins fournis par la communauté. RCP impose cependant un effort d'apprentissage assez important et ne peut prétendre à la même simplicité que WPF.

JavaFX

Depuis la version 8 de Java, JavaFX est devenu l'outil de création d'interfaces graphiques officiel de la plateforme. Il peut être utilisé pour le développement d'applications mobiles, d'applications sur poste de travail ou d'applications Web. Il s'agit déjà d'une similarité avec WPF et de la notion de modèle d'applications.

Une application JavaFX est articulée autour de la description des interfaces graphiques avec le langage FXML (syntaxe XML) et de la logique métier en code Java. Il est même possible de décrire l'apparence des composants graphiques à l'aide de feuilles de style CSS (Cascading Style Sheet). De la même façon que Microsoft propose l'outil « Blend » pour assister le développeur dans la création de IHM, Oracle propose l'outil « JavaFX Scene Builder ».

Considérons l'exemple FXML suivant :

```
<GridPane fx:controller="MyApplicationController" xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
<padding><Insets top="25" right="25" bottom="10" left="25"/></padding>
        <Label text="User Name:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
        <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
        <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="2"/>
        <PasswordField fx:id="passwordField" GridPane.columnIndex="1" GridPane.rowIndex=« 2"/>

        <HBox spacing="10" alignment="bottom_right" GridPane.columnIndex="1" GridPane.rowIndex="4">
            <Button text="Sign In"  onAction="#handleSubmitButtonAction"/>
        </HBox>

<Text fx:id="actiontarget" GridPane.columnIndex="0" GridPane.columnSpan="2" GridPane.halignment="RIGHT" GridPane.rowIndex="6"/>
</GridPane>
```

Cette interface définit :

- le nom de la classe Java qui fera office de « code-behind » : « MyApplicationController ».
- un label/champ de saisie pour « User Name »,
- un label/champ de saisie pour « Password »,
- un bouton déclenchant la méthode « handleSubmitButtonAction() » du code Java,
- un champ texte « actionTarget » qui sera utilisé pour afficher le résultat de l'appui sur le bouton.

Le code Java du controleur « MyApplicationController » :

```
public class MyApplicationController {
   @FXML private Text actiontarget;

   @FXML protected void handleSubmitButtonAction(ActionEvent event) {
      actiontarget.setText("Appui sur le bouton");
   }

}
```

Bien que simpliste, cet exemple nous montre que les concepts sont similaires à ce qu'on trouve avec WFP :

- description des interfaces dans un langage XML,
- logique métier dans un contrôleur (équivalent du « code-behind » en .NET),
- binding de données,
- modèle évènementiel,
- composants graphiques de haut niveau,
- manipulation d'objets 2D, 3D et de fichiers multimédia.

Comparaison avec C++

La création d'applications clientes fût l'un des domaines réservés de C++, du fait de sa proximité avec les spécificités graphiques des systèmes d'exploitation. L'émergence des plateformes .NET et Java ont petit à petit amoindri l'intérêt du C++ en la matière (de part une plus grande simplicité de mise en oeuvre et une meilleure portabilité).

Certaines librairies (Qt, GTK) ont cependant perduré notamment grâce à leur utilisation par les couches graphiques des systèmes Linux (KDE et Gnome principalement). Qt est d'avantage un framework complet qu'une simple librairie graphique. Elle existe sur la majorité des systèmes d'exploitation (de bureau comme mobiles).

Quelques exemples de composants de Qt :

- QtQuick : pour les composants graphiques. Description des IHM avec le langage QML,
- QtNetwork : pour la programmation réseau,
- QtOpenGL : pour l'utilisation d'OpenGL,
- QtSql : pour l'utilisation de base de données SQL,
- QtXml : pour la manipulation et la génération de fichiers XML,
- QtTest : pour effectuer des tests unitaires,
- QtXmlPatterns : pour manipuler des documents XML via XQuery et XPath.

9.Conclusion

Nous l'avons constaté tout au long de cet exposé : la plateforme .NET et la plateforme Java/JEE sont extrêmement similaires. La plupart du temps, les nouvelles fonctionnalités qui ont été apportées à l'une ont été intégrées par la suite dans l'autre. Nous notons une volonté affirmée de la part de Microsoft de proposer une solution qui guide le développeur dans ses choix d'implémentation. Java/JEE affiche une courbe d'apprentissage plus accentuée, de part la diversité des technologies disponibles. Cela requiert un effort d'intégration assez conséquent et pose la question de la pérennité des solutions choisies. Les entreprises sont donc confrontées à la question de l'orientation technologique qu'elle doivent prendre. La réponse à cette question est multiple et doit prendre en compte :

- les compétences des développeurs,
- les compétences des équipes exploitant le système d'information,
- l'infrastructure du système d'information (Microsoft/Linux, LDAP/Active Directory),
- la complexité des projets (Webforms pour des applications web simples ? JEE pour des applications web complexes devant s'intégrer dans un environnement hétérogène ?).

Le cas de C++ est différent. Ce n'est pas la technologie de prédilection pour la réalisation d'applications et de services web. Quoi qu'il en soit, il existe toutes les librairies nécessaires. Cela peut être utile pour un applicatif existant que l'on souhaiterait exposer au sein d'un système d'information. Il ne s'agit pas réellement d'un choix stratégique pour les entreprises, mais plutôt de pérenniser des applications dont le coup de re-développement serait rédhibitoire. C++ demeure une technologie adaptée aux applications bas niveau ou à la communication avec des automates dans le domaine industriel.

10. Glossaire

AJAX	Asynchronous Javascript And XML : ensemble de technologies permettant l'échange de données avec un serveur web au travers de la mise à jour de certains composants des pages.
ASP	Active Server Pages : suite de logiciels de Microsoft destinée à créer des sites web dynamiques.
Assembly	Unité de regroupement de classes .Net compilées en code intermédiaire. Permet un découpage applicatif sous forme de librairies.
At runtime	Phase du cycle de vie d'une application, au cours de laquelle celle-ci est gérée par un environnement d'exécution. Celui-ci prend notamment en charge l'instanciation des objets, la gestion des processus, la gestion de la pile mémoire, génération de code à la volée, ...
AWT	Abstract Window Toolkit : librairie graphique fournie avec J2SE.
Bean	Classe ne possédant que des attributs et leurs accesseurs.
CIL	Common Intermediate Language : code généré lors de la compilation du code source par les compilateurs .NET. Il est indépendant de la plateforme d'exécution et sera transformé en code machine lors de l'exécution par une machine virtuelle.
CLI	Common Language Infrastructure : spécification développée par Microsoft qui décrit l'environnement d'exécution de la machine virtuelle basée sur CIL.
CLR	Common Language Runtime : environnement d'exécution développé par Microsoft qui implémente la CLI. Il permet d'exécution du code managé.
CLS	Common Language Specification : spécification décrivant les caractéristiques d'un code utilisable par tous les langages .NET.
CSS	Cascading Style Sheet : syntaxe décrivant la présentation de document HTML et XML.
CTS	Common Language Types : spécification des types de données pouvant être utilisés dans les langages compatibles .NET.
Data binding	Processus de connexion entre l'interface graphique d'une application et son modèle objet (sa logique métier).
DOM	Document Object Model : interface indépendante de tout langage de programmation, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents HTML et XML.
GAC	Global Assembly Cache : stocke les assemblies spécialement destinés à être partagés entre plusieurs applications sur l'ordinateur.
IIS	Internet Information Services : logiciel serveur de services web (FTP, SMTP, HTTP, ...) de Microsoft. Il prend en charge plusieurs technologies web telles CGI, ASP, ou ASP.NET.
JIT	Just In Time : technique visant à améliorer la performance des applications exécutées par une machine virtuelle par la traduction du bytecode en code machine natif au moment de l'exécution.
JSF	Java Server Faces : technologie ayant pour but de proposer un framework qui facilite et standardise le développement d'applications web avec Java. Elle repose notamment sur le motif de conception MVP.
Layout	Mise en forme de la structure d'une IHM applicative.
Lazy loading	Terme utilisé dans le domaine du mapping objet relationnel. Il désigne la technique visant à ne charger les données uniquement que lorsqu'elles sont explicitement demandées.
.NET CP	.NET Compiler Platform : compilateur Microsoft de code compatible .NET en code intermédiaire.
Metadata	Donnée servant à définir ou décrire une autre donnée (les annotations des frameworks ORM par exemple).

Model driven design	Méthodologie visant à construire une application à partir de la définition du modèle conceptuel de données.
MVC	Model View Controler : modèle de conception destiné à répondre aux besoins des applications interactives. Il permet la séparation des problématiques liées aux différents composants au sein de leurs architectures respectives.
MVP	Model View Presenter : modèle de conception dérivé du MVC. Il garde les mêmes principes que celui-ci sauf qu'il élimine l'interaction entre la vue et le modèle puisqu'elle sera effectuée par le biais d'un objet de présentation organisant les données à afficher dans la vue.
MVVM	Model View ViewModel : patron de conception basé sur MVC et MVP visant à associer la vue avec une modèle objet qui lui est dédié. Au travers de celui-ci, il sera possible de binder des données et de gérer des évènements.
ORM	Object Relationnel Mapping : technique de programmation simulant l'utilisation d'une base de données orientée objet à partir d'une base de données relationnelle. Elle s'appuie sur la définition de correspondances entre cette base de données et les objets du langage utilisé.
POCO	Plain Old Clr Object : Bean développé dans un langage compatible .NET.
POJO	Plain Old Java Object : Bean développé en Java.
Portable Executable	Format des fichiers exécutables et des bibliothèques hébergées sur les systèmes d'exploitation Windows 32 bits et 64 bits.
PostBack	Technique visant à soumettre une page web vers le serveur puis à se recharger en retour.
RCP	Rich Client Platform : Plateforme de développement d'applications clientes basée sur la notion de plugins.
REST	REpresentational State Transfer : style d'architecture qui repose sur le protocole HTTP. Elle permet d'accéder à des ressources (via leurs URI uniques) pour procéder à diverses opérations (GET/POST/PUT/DELETE).
SWING	Librairie graphique fournie avec J2SE.
SWT	Standard Widget Toolkit : librairie graphique initiée par IBM. Il ne fait pas partie de J2SE, et est spécifique à l'environnement d'exécution.
URI	Uniform Resource Identifier : Courte chaîne de caractères identifiant une ressource sur un réseau.
WCF	Windows Communication Foundation : ensemble de technologies .NET fournissant un modèle de programmation unifiée pour construire des applications distribuées.
WPF	Windows Presentation Foundation : ensemble de technologies .NET fournissant un modèle de programmation unifiée pour construire des applications clientes.
WSDL	Web Services Description Language : grammaire XML permettant de décrire un service web.
XAML	eXtensible Application Markup Language : langage d'interface utilisateur graphique universel pour les applications web riches et les applications clientes. Sa syntaxe est basée sur XML.
XmlHttpRequest	Objet JavaScript qui permet d'obtenir des données au format XML, JSON, HTML ou encore du texte simple à l'aide de requêtes HTTP.

11. Bibliographie et sources numériques

« **.NET, Framework, ADO et services Web** » :
Jérome Gabillaud, ENI editions

« **.Net framework 4.0 site MSDN** » :
http://msdn.microsoft.com/fr-fr/library/vstudio/w0x726c2%28v=vs.100%29.aspx

« **Cpplinq** » :
https://cpplinq.codeplex.com

« **Microsoft RX** » :
http://rxcpp.codeplex.com

« **Entity Framework Tutorial** » :
http://www.entityframeworktutorial.net/what-is-entityframework.aspx

« **ODB** » :
http://en.wikipedia.org/wiki/ODB_(C%2B%2B)

« **The complete ASP.NET tutorial** » :
http://asp.net-tutorials.com

« **Apprendre ASP.NET MVC** » :
http://openclassrooms.com/courses/apprendre-asp-net-mvc/hello-world-mvc

« **Formation JEE7** » :
Orsys, octobre 2014

« **Création et consommation de services WCF** » :
http://www.dotnet-france.com/Documents/WCF/Création%20et%20consommation%20de%20services%20WCF.pdf

« **Présentation de WPF** » :
https://msdn.microsoft.com/fr-fr/library/aa970268(v=vs.110).aspx

« **JavaFX** » :
http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

« **Qt Project** » :
http://qt-project.org