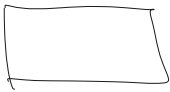
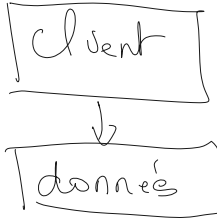
Architectures en couches

1 niveau

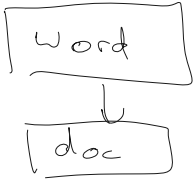


2 Niveaux

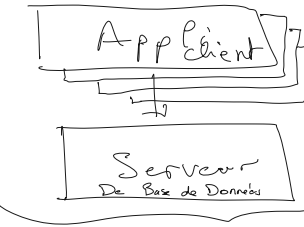


2 Niveaux (Tier)

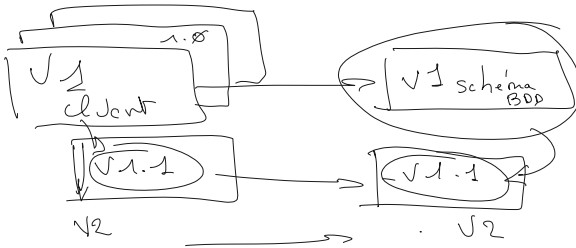
Local



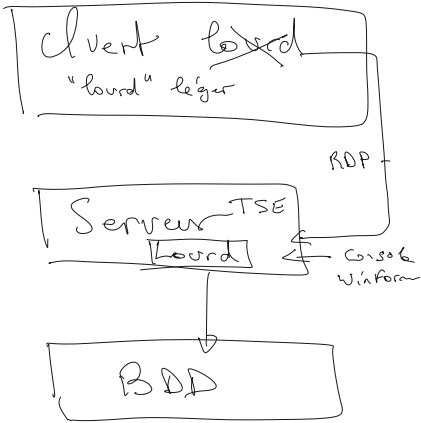
Réseau



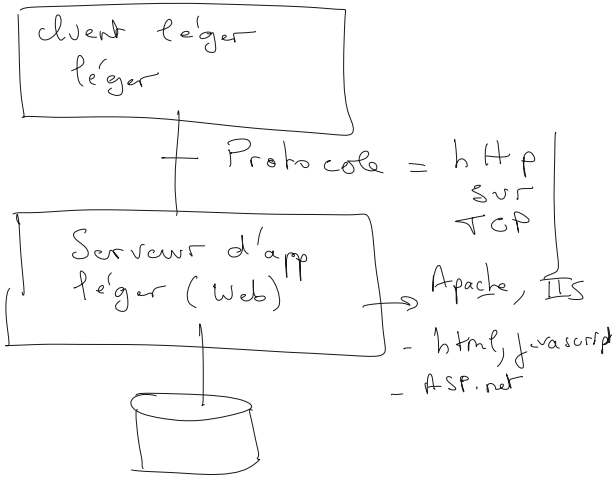
Client/Serveur
Client Lourd

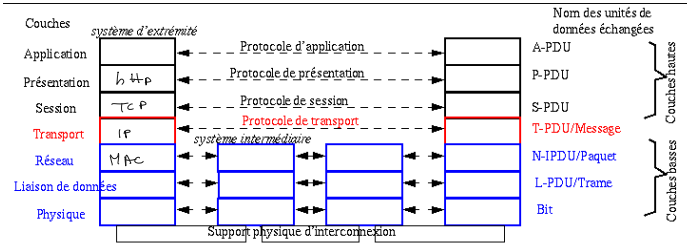


Architecture 3 Tier



3 Niveaux (Tier) Web



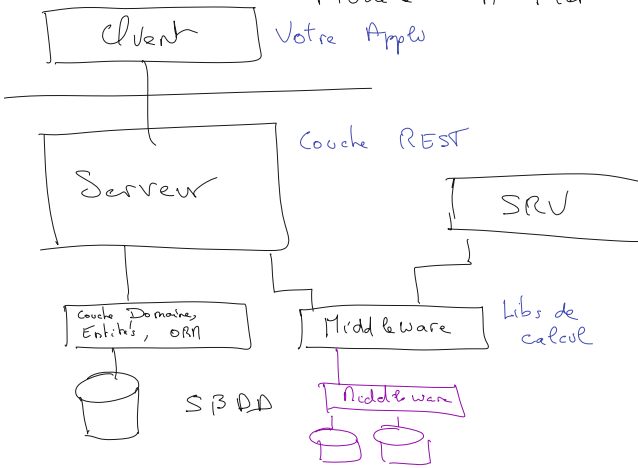


HTTP (web), SMTP, POP, FTP, SSH — RDP

TCP - UDP

IP (Transport) — autre = IPX, Netbeui, Appletalk

Modèle n-Tiers



"Middleware" = logiciel au milieu

Continuous Integration

CI

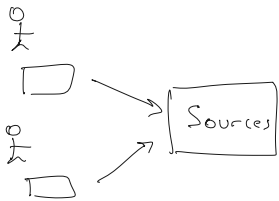
Delivery

CD

Scenarios : CI/CD

---

déploiement traditionnel :



"Packager"
Installer → (wix)

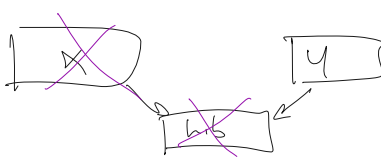
Projet NSI

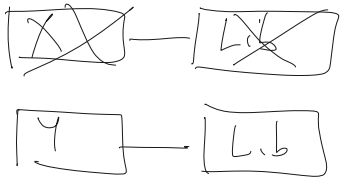
MicroSoft Installer → fiabiliser Windows

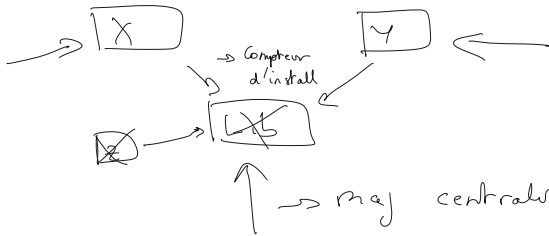
msi → purement MS -

→ Installer proprement

& retirer







Composants centralisés

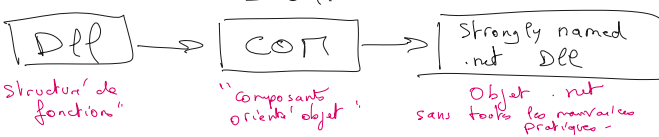
Windows : %Win\System32

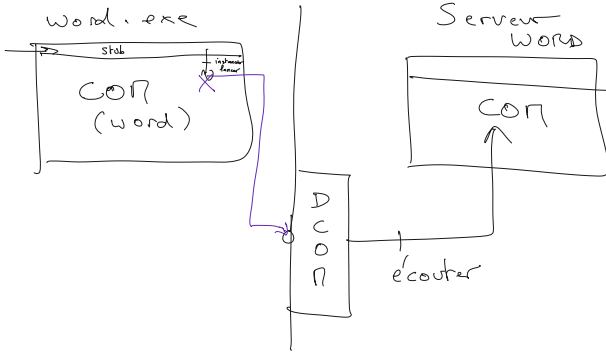
"Dll Hell"

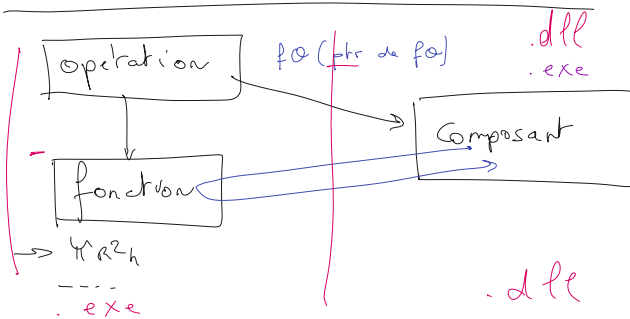
Enfer des dlls

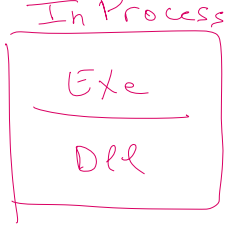
DCOM?

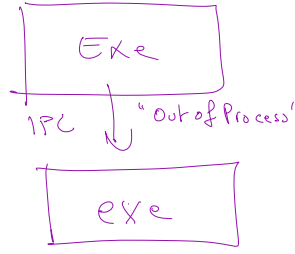
GAC

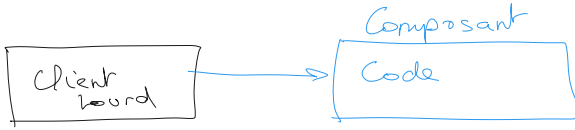



word.exe
Stub
CoM
(word)
DCOM
Serveur
WORD
CoM
écouter


operation
fct (ptr de fct)
.dll
.exe
Composant
fonction
.exe
.dll


In Process
Exe
Dll


Exe
IPC
"Out of Process"
exe



"Composant" → Terme générique

→ structure → DLL "de fonctions"

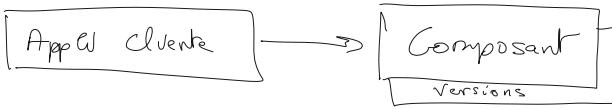
→ O. objet → Comp. COM DLL

---

① DLL locale → côte à côte avec son "client" → dans le même dossier

② DLL globale → System32

Modèle



Physique

Versions

Majeur . mineur . revision . build number

.build → incr automatique (via l'env.)

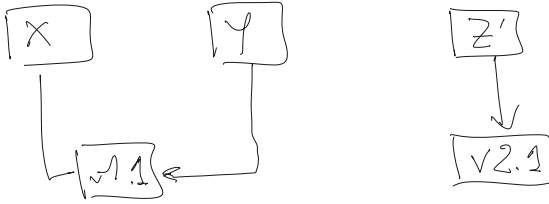
revision → incr sur retouches, ajouts pour la version en cours.

mineur → incr sur ajouts de fct, corrections, compatibilité ascendante garantie

→ Idéal → 1 seul version majeure d'un composant sur un système

⇒ n'avoir qu'une version de son composant !

. Majeur → incr. sur impossibilité de garantir la compatibilité ascendante



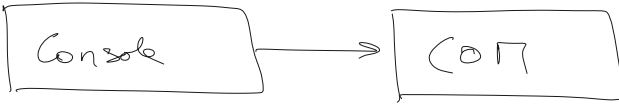
Apps clientes :

C:\PF\<Editeur>\<produit>\<version majeur>

ex : c:\PF\DPT\Esprit\1\ binaires pour la v1
\2\ binaires pour la v2

Composants

C:\PF\Common Files\<Editeur>\<produit>\<Majeur>



→ eval Version

→ Base de Reg

→ VBS

Dll de fonction → Recherche par emplacement sur le Disque.

Comp COM → doit être inscrit dans la Reg DB

HKCR\CLSID → Class ID

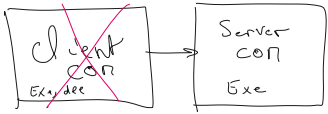
<GUID> globally Unique ID du Composant

Local Server32 → Exe, DCOM

InProc Server32 → Dll, in Process

Composants COM

Autonome, .exe
→ Compatible DCOM



→ chargement unique
→ Risque de fuites

Hébergé
"In Prouss"
DLL

→ chargement multiple
→ propre

CLSID → ID Technique

ProgID → ID Convivial (Nom)
ex: Word.Application.15

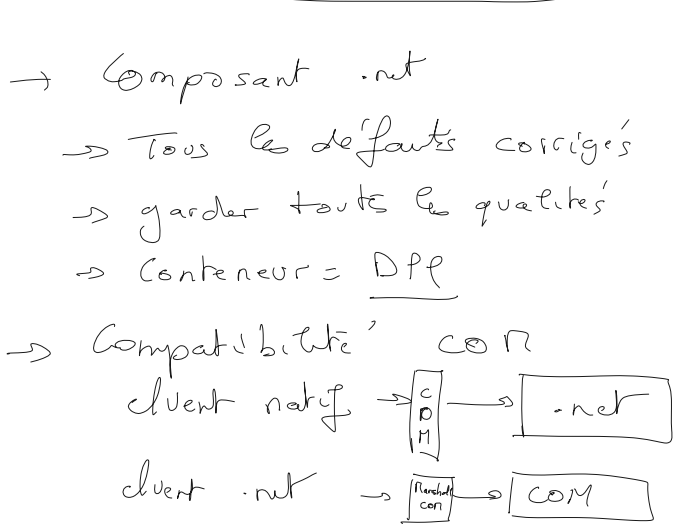
HKCR\<ProgID>
→ description
CurVer → Pointe ProgID
"Version en cours"
CLSID → Id CLS du ProgID

Exercice concernant les composants COM :
Il existe une Dll de type COM qui permet de se connecter aux bases de données.
Son ProgID utilisé dans les scripts, est "ADODB.Connection"

Listez :

1. Son CLSID
2. Son Chemin
3. Est-ce un composant COM ou DCOM ? (Inprocess ?)
4. Si possible : sa version = ?
5. Y a t'il plusieurs versions autres que celle par défaut ?

Affichez vos réponses dans le canal global de la formation



1) Dll Simple .net

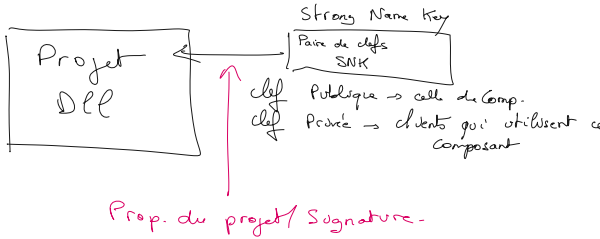
→ Créer une Dll

→ classes publiques sont accessibles en dehors du Composant

→ La Dll sera côte à côte

---

2) Créer un Composant .net Global



Enregistrement dans le GAC (Global Assembly Cache) :

- Créer un projet Dll
- Propriétés du projet : signer l'assembly
- Compiler le projet client (exe) - le recompiler
- inscrire la dll dans le GAC : gacutil /i <dll>
- Vérifier : gacutil /l <NomDuComposant>
- supprimer : gacutil /u <NomDuComposant>
- S'assurer que la Dll ne soit pas côte à côte (isolée) de l'exe.

.net Framework

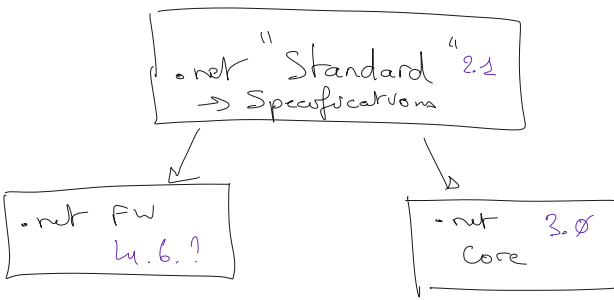
→ Seulement sous Windows

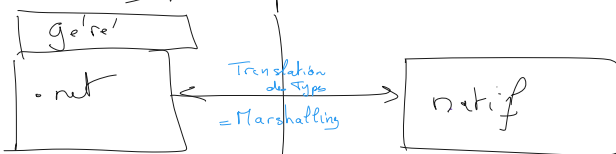
- Le Framework le plus complet

.net Core

→ Windows, Linux, MacOS

→ Sous-ensemble du Framework (≈ 90%)





domaines totalement ≠

ex: Int32
→ Instance de classe

int (C++)
→ 4, 8 octets
Big Endian, Little Endian

(1)

natif ⟶ .net

1) Fabriquer Dll de fonctions

2) Enregistrer Comp .net en tant que Comp. COM

---

(2)

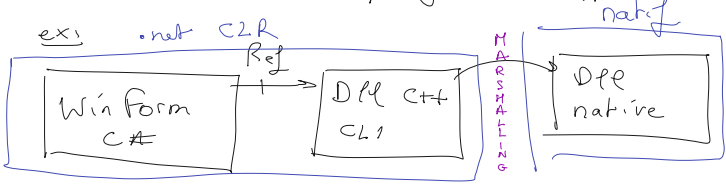
.net ⟶ code natif

(1) Dll de fonctions

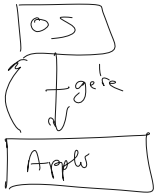
a) consiste à Appeler directement le code → Marshalling doit être implémenté manuellement → Très lourd LOURD

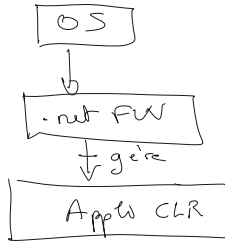
---

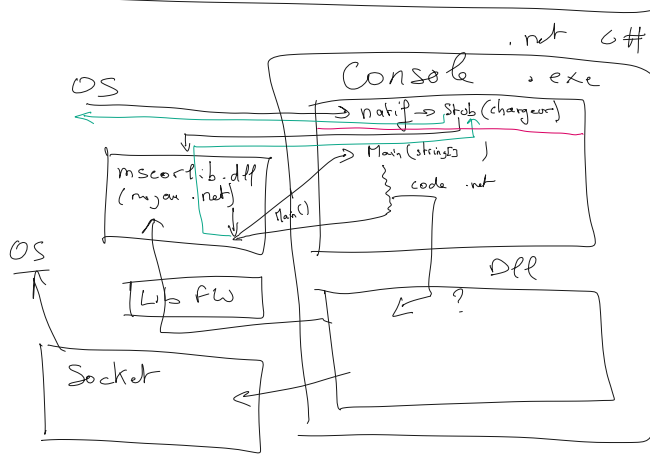
b) Fabriquer une interface en C++/CLI qui fasse les appels



natif :


OS
gère
Appli


OS
.net FW
gère
Appli CLR


.net C#
Console .exe
OS
natif -> Stub (charge.cor)
Main (string[])
mscorlib.dll
(mgan .nat)
code .nat
Main()
OS
Dll
LIB FW
?
Socket

Consolidation connaissance architecture .Net/natif

```
class Personne
{
    string nom;
    int age;

    string SeDecrire()
    {
        return "Bonjour je suis "+nom;
    }
}

*/

#include <string.h>
#include <windows.h>

struct Personne {
    int age;
    char* nom;
};

typedef Personne* PPersonne;

char* PersonneSeDecrire( PPersonne self )   // self equivaut au this en c++
{
    char presentation[1024];

    strcpy(presentation, "Bonjour je suis ");
    strcat(presentation, self->nom);    // "équivalent  a this->nom
}



// côté appelant :
void main()
{
    Personne jean;
    jean.nom = new char[1024];
    strcpy(jean.nom, "Jean");

    // appel de l'équivalent d'une méthode :

    PersonneSeDecrire(&jean);   // appel explicitement de l'équivalent this par l'appelant
    // equivalent en c++ :
    // jean->SeDecrire();
}
```

Exercice de conception UML par le code et par le modèle :

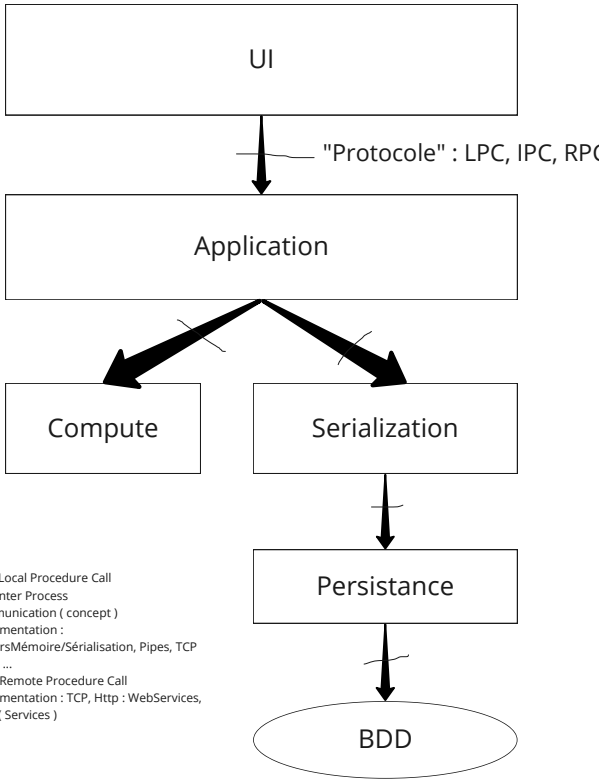
- Installez la fonctionnalité de conception de classes (class designer)
- Créez un projet vide
- Ajoutez un code avec héritage de classes et agrégation de membres complexes (exemple : une voiture a un moteur)
- Affichez ce code dans le diagramme de classes
- Modifiez le diagramme de classe graphiquement :
    - Ajoutez une nouvelle classe
    - Intégrez de nouveaux membres
        - Testez un membre a relation n (tableau ou liste) vers un élément complexe : exemple une voiture a 4 roues
    - Consultez le code généré.

Composants .Net en tant que composants COM :

- .Net Core :
  - https://docs.microsoft.com/en-us/dotnet/core/native-interop/expose-components-to-com
- .Net Framework :
  - https://docs.microsoft.com/en-us/dotnet/framework/interop/exposing-dotnet-components-to-com
- Obligations de conception :
  - https://docs.microsoft.com/en-us/dotnet/standard/native-interop/qualify-net-types-for-interoperation
- Exemple de code :
  - https://docs.microsoft.com/en-us/dotnet/framework/interop/com-interop-sample-com-client-and-net-server
- Tuto C# :
  - https://aakinshin.net/posts/wrap-cs-in-com/

UML

Design Pattern



LPC : Local Procedure Call
IPC : Inter Process Communication (concept)
Implémentation : FichiersMémoire/Sérialisation, Pipes, TCP Local, ...
RPC : Remote Procedure Call
Implémentation : TCP, Http : WebServices, REST (Services)

Frameworks

Exemple : ORM : nHibernate -> LINQ To SQL -> Entity Framework

- Objectif(s) du framework
- Réputation
- Pérénité
- Difficulté d'apprentissage et de mise en oeuvre

Patterns (d'architecture)

Frameworks en face

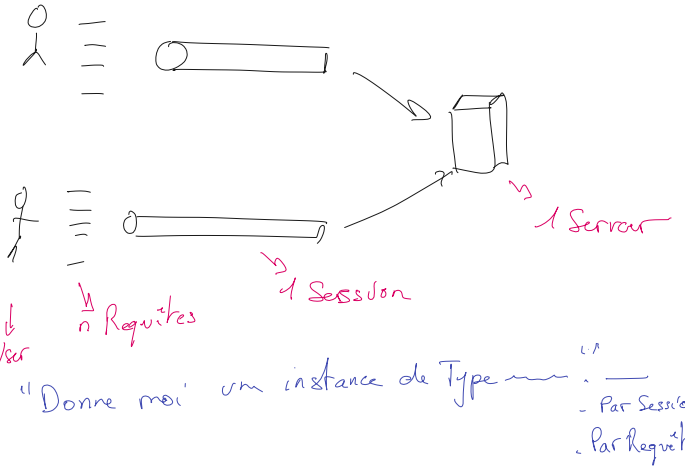
Pour Microsoft : Généralement ceux fournis par Visual Studio (ou le .Net Framework)

Design Pattern Injection de Dependances MS :

https://docs.microsoft.com/fr-fr/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

Principes

- Amont -> Enregistrement des "Services"
  Association de type d'Interface à type de classe ILogger <- MyLogger
- Dans le code -> demander l'instance pour un type donné



SingleToN : Single To Numerous (Toujours le même pour n)

Racine de la documentation de l'Injection de dépendances en .Net : https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.extensions.dependencyinjection.iservicecollection?view=dotnet-plat-ext-3.1

Besoin fonctionnel

→ Cahier de charge fonctionnel

→ Liste de tout à mettre en œuvre

→ choix d'architecture.

- Application
  - Lourd
  - Léger (Web)
- Flux (ETL) + Simple, - recommandé
- Business Process (Workflow, SOA)

Appli Lourde

→ Architecture physique :

→ Mono client (1 niveau)

→ Client Server (2 Niveaux)

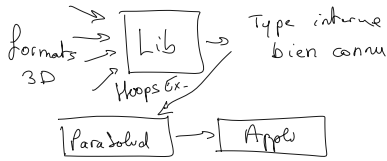
→ API externe (Services (web))

→ Persistance des donnée

→ Fichiers → cas spécifique, (mime) types bien connus - STL STEP IGES Solidworks Parasolid

→ Base de données -

→ (Service externe)

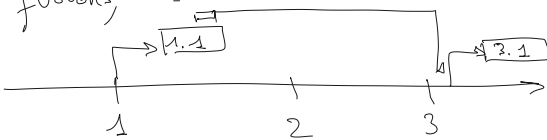


→ Gestion de Versions

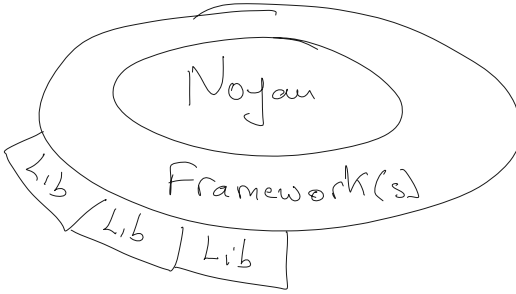
- Travail en équipe → backend (Git)

(3 Repos)

→ Connaissance en gestion → branches, fusions, ...



→ Intégration des ≠ Composants



Composants → DLL (signed ?)

→ Types ? (fQ, COM, .net)

→ enregistrement.

Bonne pratique dossiers d'install

C:\PF\ CF\

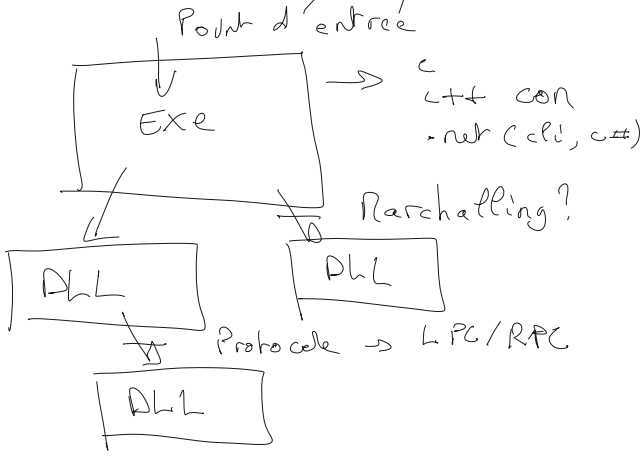
<Editeur>/<Produit>/<majeur>

\bin

\conf

\data

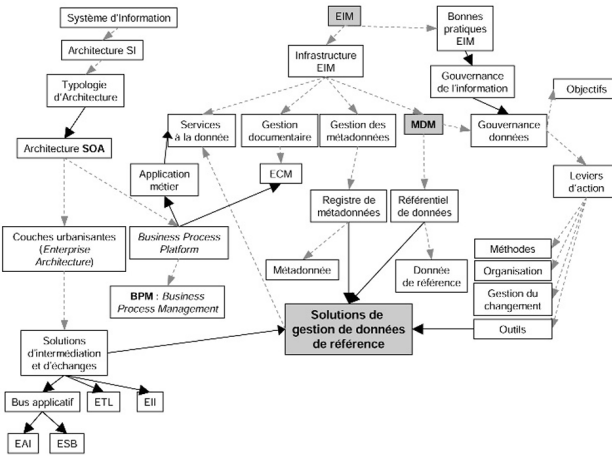
Par Appli :



" SI " → gouvernance

" MDM " Master Data Management

Gestion des données maîtres : Master Data Management



- Frameworks
- Libs

} Externe => "s'y plier"

Architecture interne :

Choix technologiques : - plateforme (Win / Linux / MacOS, ...)

- Langage : Java, Python, C++, .net C#

→ C++ natif → historique (+)
→ performant (+)
→ complexe, moins productif (-)

→ .net CS → nouvelle technos (+)
→ productif (évols / temps)
→ fiable (prod bugs moindre)

→ Scripting ? ~~VBA~~ → COM

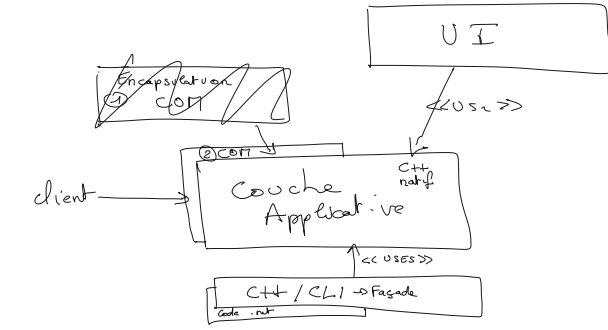
nouvelle techno

.net — C++

Powershell ← Modules

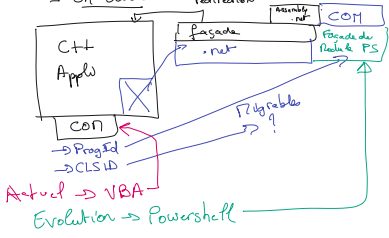
NSDOS → CMD → VBS → Powershell

≈ 2005



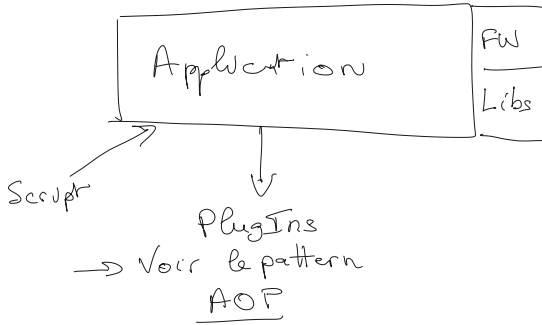
Transition de technologie

- Directe
- En douceur redirection



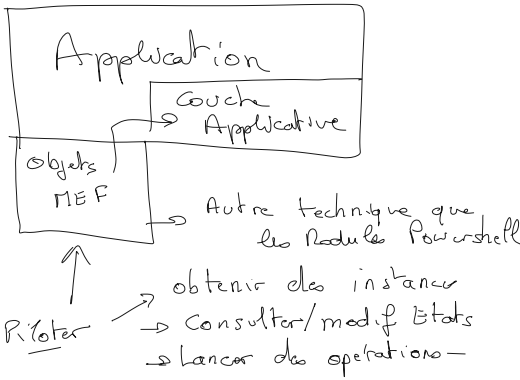
Forcer les Guid en .Net : https://docs.microsoft.com/fr-fr/dotnet/api/system.runtime.interopservices.guidattribute?view=netcore-3.1

Ecrire un module PowerShell en .Net : https://www.powershellmagazine.com/2014/03/18/writing-a-powershell-module-in-c-part-1-the-basics/



2 tech → Plug COM
→ Extension MEF .net

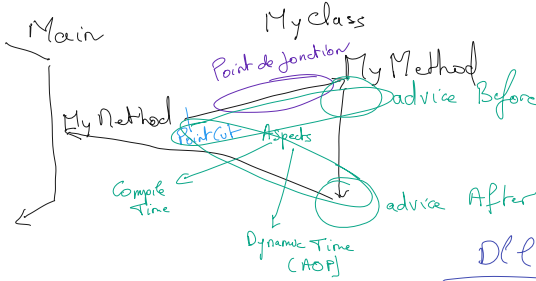
https://docs.microsoft.com/fr-fr/dotnet/framework/mef/

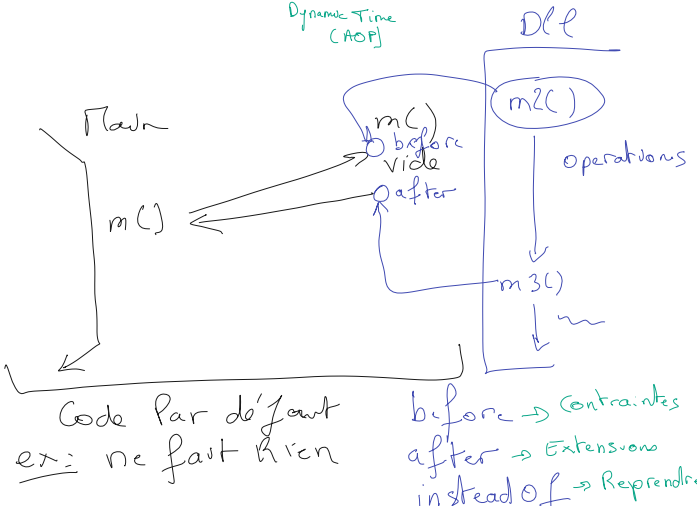


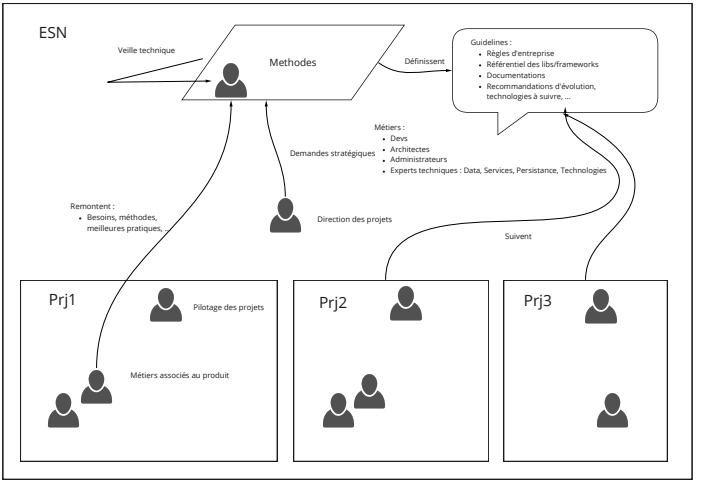
AOP : Aspect Oriented Programming/Pattern

Pas de Framework Microsoft implémentant ce pattern
Permet la modularité : Ajout/Suppression de plugins (modules dynamique)
pour étandre fonctionnellement notre application
Les plugin sont développables par les clients.

Article de Blog officiel :
https://docs.microsoft.com/fr-fr/archive/msdn-magazine/2014/february/aspect-oriented-programming-aspect-oriented-programming-with-the-realproxy-class







- Architecture patterns :
  - MV* : MVC, MVP, MVVM (front)
  - ORM : back, persistance (également nommé DAL : Data Access Layer)
  - Extensibilité : AOP
  - Souplesse : IoC
- **Cultivez-vous en consultant les design patterns du Gof, et GRASP**

Qualité logicielle

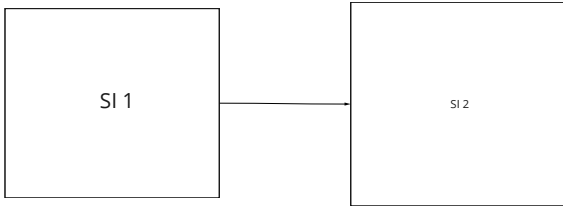
Gestion de type TDD (Test Driven Development)

- Test first

Types de tests :

- Unitaires
- Intégration
- Fonctionnels
- UI : pilotage
- Charge

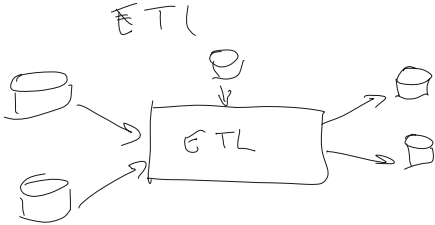
Gestion des flux

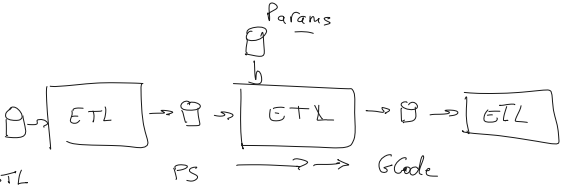


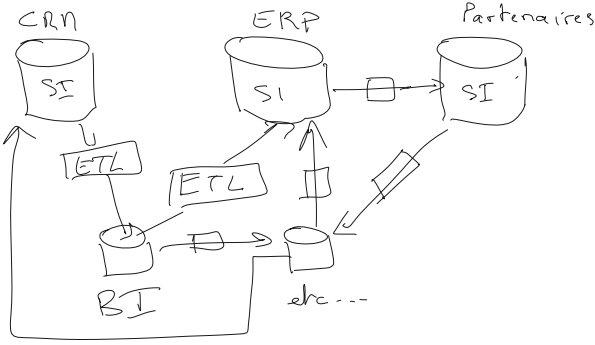
SI : Système d'Information
→ Données
→ Process

Décentralisé, Simple, défauts à Terme

"ETL" : Extract Transform Load

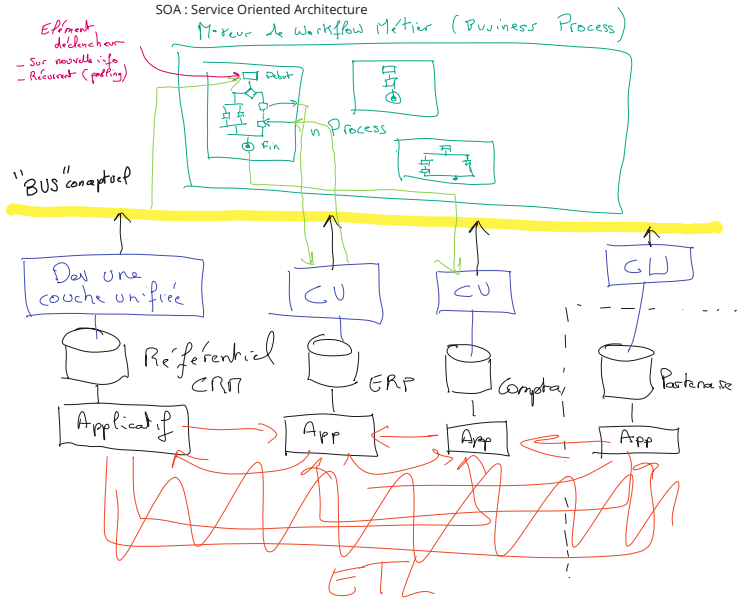






- "Plein" de flux
- Utilisant des stockages hétérogènes (fichiers, bases de données de tel et tel éditeur, flux réseaux, ...)
- Technologies de transformation différentes (Process développé en interne, script bash, script SQL type Insert select, ETL éditeur : SSIS Microsoft, Talend for DI - Data Integration, ...)
- Multiplication des flux et technologies complexe à maintenir, et risque de défaut concernant la qualité des données.

SOA : Service Oriented Architecture



Couche unifiée: Encapsulation, surcouche d'accès à la donnée
→ Recommandé, parfois nécessaire

Approche ETL :

- Simple techniquement
- Fonctionne en Batch (en lot) : tout les autant, je tire, traite et renvoie un paquet d'éléments

Approche type Business Process (BPM : Business Process Management) BPEL : Business Process Enterprise Language en XML

- Déclaration de process d'entreprise : Que dois-je faire quand un nouvelle élément de type.... (ex : commande, un incident, une demande de transfert, un appel, un mail, un message.... arrive)
- Une instance du process sera initiée pour chaque nouvel élément
  - ex Paquet de 1000 mails => 1000 instance de ce process
  - Avantage : chaque process est dans une étape particulière du modèle
  - chaque instance a sa durée de vie, état, ... **peut durer des années !**

SOA : Concevoir des surcouches au dessus de nos systèmes pour pouvoir être attaquées via des protocoles standards (Anciennement : WebServices, actuellement : REST)
ESB : Enterprise Service Bus : Système de workflow qui se connecte aux référentiels (via une architecture SOA) ou directement si possible, de façon a instancier des process d'entreprise sur évènement déclencheur.

WebService : Technologie utilisant des protocoles standards et open pour interopérer entre des SI hétérogènes : ex : SAP avec IBM zOS et Windows et Serveur Web Linux
Utilisation de format XML sur http
REST : variante, évolution des WebServices (trop lourds) orientée autour du Web pour échanger des informations et executer des opérations.
Formats différents : JSon (primaire), Yml, XML, CSV, binaire....
Avec un protocole http
Compatible téléphonie mobile limitée en puissance et en énergie.

Solutions :

- BizTalk de Microsoft
- Talend for ESB (Enterprise Service Bus)

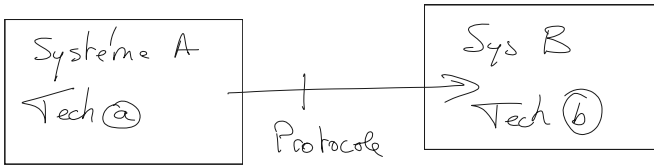
Type bonnement

Moteur de Workflow pour développeur :

- .Net 3 : apporté C#3 , 3 Frameworks : WCF - Windows Communication Foundation, WPF (Windows Presentation Foundation), WF (Workflow Foundation)

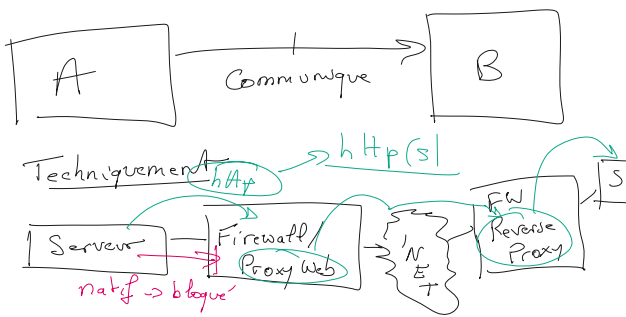
En tant que Dev , si vous souhaitez implémenter votre propre moteur de workflow

- Editeur graphique, gestion des état, persistance, modèle de service windows, etc...
- -> WF



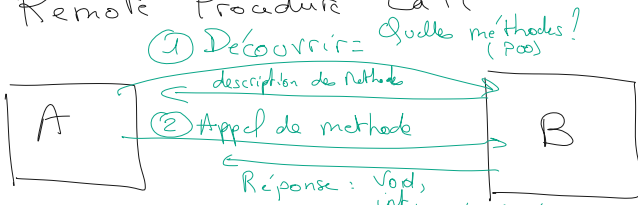
Protocole :

- Synchrone ou asynchrone
- Fire And Forget ou Requête Réponse
- Uni ou Bi Directionnel
- natif et performant ou Texte Interprétable
- protocole spécifique ou Open



Objectif : Faire du RPC

Remote Procedure Call



A Consomme des Services de B

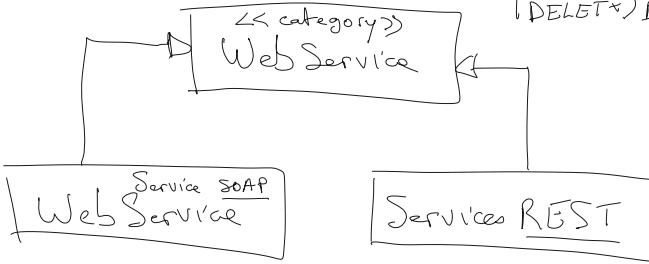
B expose des Service (web)

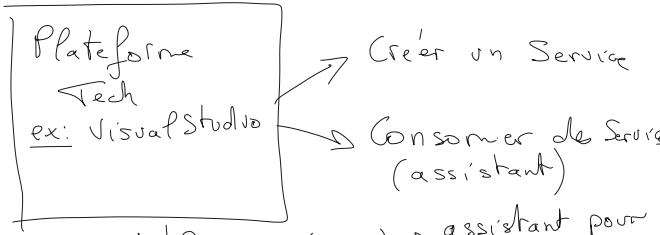
"WebService" →

- Services via le web de manière générale (pas de protocole en particulier)
- C'est aussi une implémentation qui s'appelle "WebService" en particulier (http+xml SOAP)

Services REST sont des WebServices qui utilisent http + verbes

GET	C
POST	R
PUT	U
DELETE	D





_ Service REST

Serveur →/ Embarquer dans ASP.net MVC WebAPI (pure appli web)

→/ Exposition de Service WCF au format REST

Consommer → Appel direct via un Client Http et désérialisation _

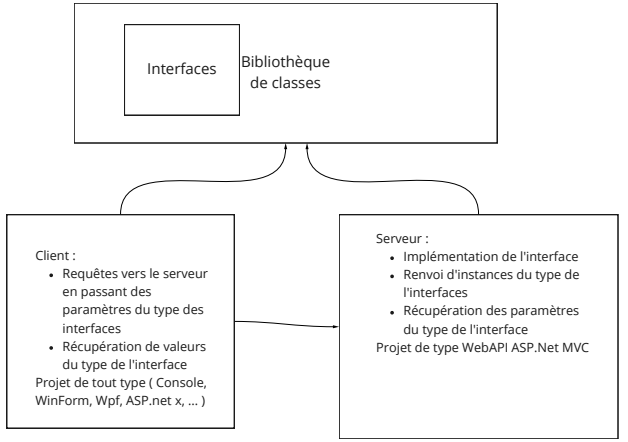
```
static HttpClient client = new HttpClient();
HttpResponseMessage response = await client.PostAsJsonAsync(
        "api/products", product);
product = await response.Content.ReadAsAsync<Product>(); // désérialiser la
réponse en tant qu'objet

Product product = null;
        HttpResponseMessage response = await client.GetAsync(path);
        if (response.IsSuccessStatusCode)
        {
            product = await response.Content.ReadAsAsync<Product>();
        }
        return product;
```

Mise en oeuvre des WebServices :

- Créer un WebService :
  - Une classe qui héberge des méthodes
  - Des méthodes qui sont publiées sur le Web (formation SOAP : Http+XML) et qui font des opérations (ex : additions, soustraction, ...)
- Créer un client de WebService :
  - Un programme qui se connecte au WebService et demande a effectuer l'opération
  - Le client aura un "Proxy" -> Une classe locale au client qui ressemble à la Classe sur le serveur

Mise en oeuvre des Services REST :

- Créer un projet du Type WebAPI (hébergé sous IIS ou Kestrel en .Net core)
  - Ajouter une classe de services
  - Publier des méthodes d'opérations
- Créer une couche de service à une application lourde (Console, WinForm, WPF, Service Windows) avec WCF (.Net 3.0)
  - Définition du service : Interface, et classe d'implémentation simple
  - Descripteur de "mise en avant" du service : synchrone/asynchrone, natif/SOAP/REST, port utilisé, ...
- Créer un client REST :
  - Qui se connecte à cette classe, et lance l'opération (Identique au ServiceWeb : Calcul d'une somme...)



Client WebAPI Console qui effectue un Appel vers une WebAPI REST, passant un type complexe, et recevant un type complexe en retour, asynchrone.

```
static async Task RunAsync()
{
    // Calcul de superficie :
    HttpClient client = new HttpClient();
    string path = "http://localhost:64657/api/Formes";
    string baseadd = "http://localhost:64657";
    //HttpResponseMessage response = await client.GetAsync(path, );   // serialiser le paramètre vers le serveur
    client.BaseAddress = new Uri(baseadd);
    HttpResponseMessage response = await client.PostAsJsonAsync("api/Formes", (new Rectangle() { Hauteur = 2, Largeur = 3 }));
    if (response.IsSuccessStatusCode)
    {
        var res = await response.Content.ReadAsAsync<IResultatGenerique>();
        Console.WriteLine(res.ToString());
    }
    else
        Console.WriteLine(response.StatusCode);

        /*
        HttpClient client = new HttpClient();
        string path = "http://localhost:64657/api/Values";

        HttpResponseMessage response = await client.GetAsync(path);

        if (response.IsSuccessStatusCode)
        {
            var res = await response.Content.ReadAsAsync<IEnumerable<string>>();
            foreach( var e in res )
                Console.WriteLine(e);
        }
        else
            Console.WriteLine("!ok");
        */
    }
}
```

```
namespace WebAppFW.Controllers
{
    0 références
    public class FormesController : ApiController
    {
        // GET api/values
        /*
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }*/

        // GET api/values/5
        0 références
        public IResultatGenerique Post(Rectangle forme )
        {
            return new ResultatSuperficie() { Resultat = forme.Hauteur * forme.Largeur } ;
        }

        /*
        // POST api/values
        public void Post([FromBody] string value)
        {
        }

        // PUT api/values/5
        public void Put(int id, [FromBody] string value)
        {
```

Gestion de versions avec les composants COM :
ProgID : ex : Word.Application

HKCR\<ProgID> : version par défaut
CurVer : <ProdID>.<version>
CLSID : <ClassID> pour la version par défaut
HKCR\<ProgID>.<version>
CLSID : <ClassID> pour cette version

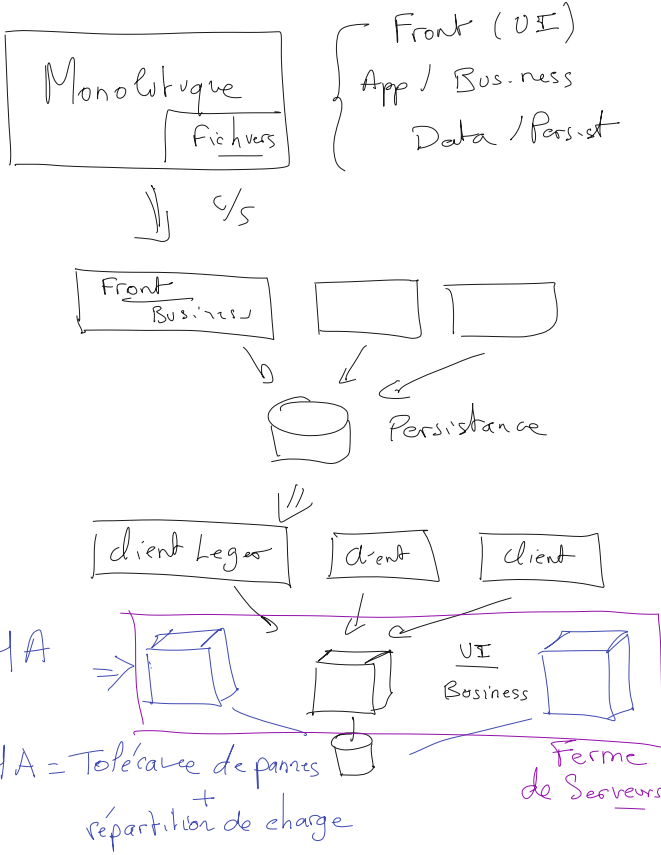
HKCR\CLSID\
CLSID : Données du composant pour sa version
dont
InProcServer32 : <path> vers le composant

Stocker des versions différentes du composant :
1) C:\PF\....
composant.<version>.dll

2) C:\PF\...\<version>\composant.dll

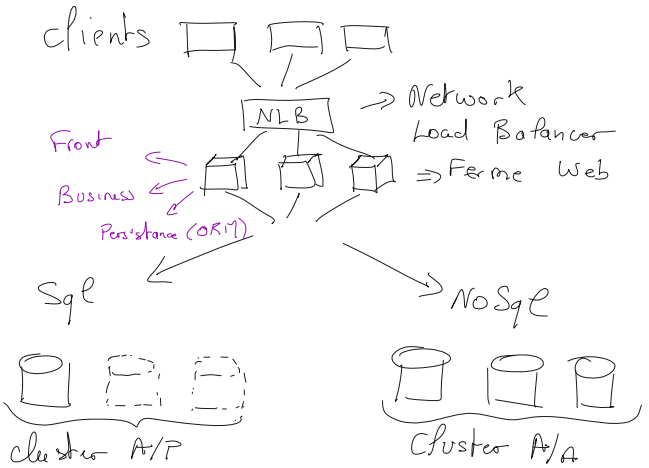
ex 1) :
c:\pf\editeur\produit\compute.1.dll
compute.2.dll

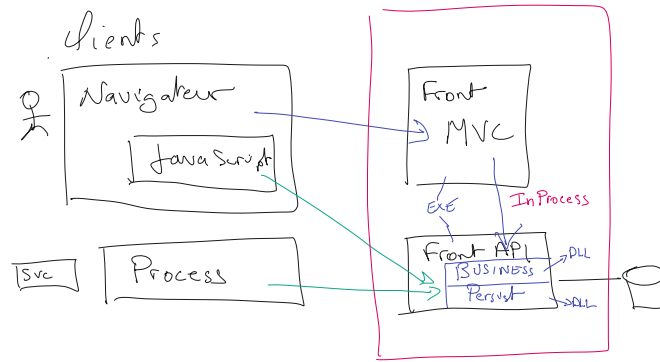
2)
c:\pf\editeur\produit\1\compute.dll
c:\pf\editeur\produit\2\compute.dll

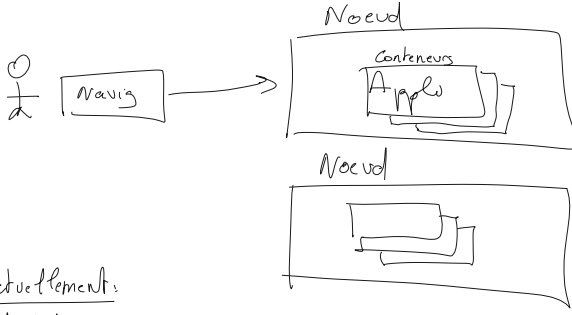

Monolithique
Fichiers
Front (UI)
App / Business
Data / Persist
C/S
Front
Business
Persistance
client Leger
client
client
HA
UI
Business
HA = Tolérance de pannes
+
répartition de charge
Ferme
de Serveurs

Ferme/Farm : Tous les serveurs sont homologues et n'ont pas besoin de communiquer entre eux.
Cluster : Ensemble de machines qui ont connaissance de leurs homologues
Actif/Passif : un seul noeud (serveur) est up, les autres sont tièdes(warm) -> que la tolérance de panne, pas de répartition de charge.
Actif/Actif : un maximum de noeuds sont up pour fournir la HA

Serveurs Web : mode ferme
Serveurs de BDD relationnels : Cluster actif/passif (Cohérence des données)
Serveurs de BDD NoSql : Cluster actif/actif (non cohérence)

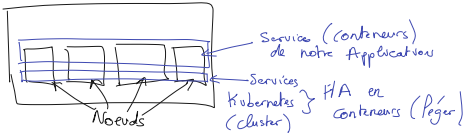


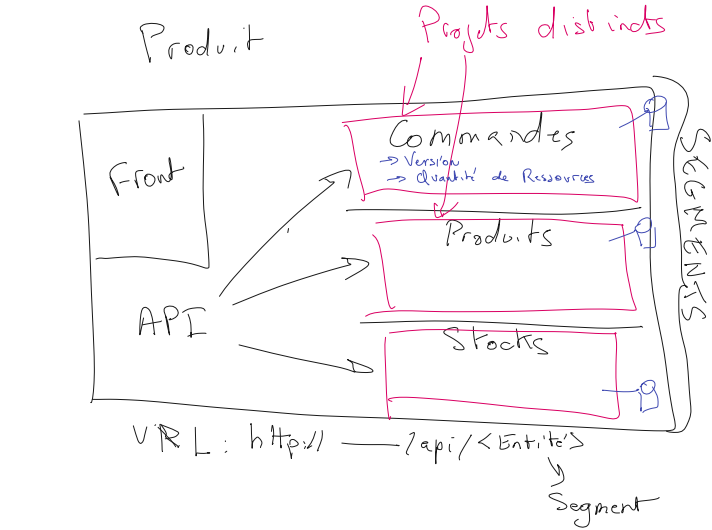


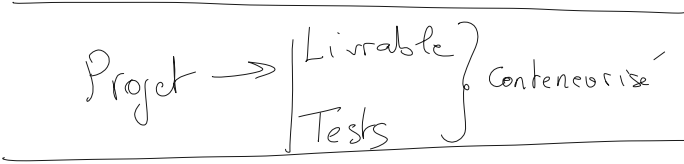


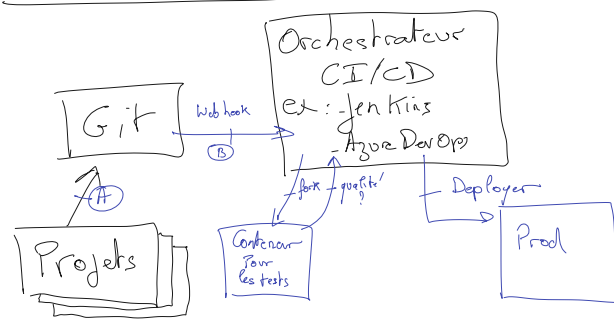
Actuellement :

Data Center




Produit
Projets distincts
Front
Commandes
→ Version
→ Quantité de Ressources
Produits
Stocks
API
SEGMENTS
URL : http:// —— /api/<Entité>
Segment


Projet
Livrable
Tests
Conteneurisé


Orchestrateur
CI/CD
ex : - Jenkins
- Azure DevOps
Git
Web hook
B
A
Projets
fork + qualité
Conteneur pour les tests
Deployer
Prod

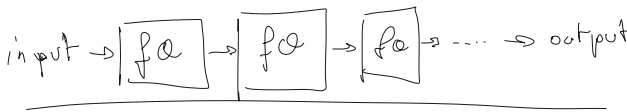
Evaluation d'ue architecture

Evaluation -> qu'elle soit objective. (Jugement de valeur/fait)
Base sur des élements objectifs (bon, bien, -> a proscrire)
Idéal -> score (numérique)

Elément	Score (à définir entre les partie)	Pondération	Résultat			
Evolutivité	5	4	20	Score : 1 a 5		
Performance	7	2	14	Pondération : 0 a 5		
Niveau technique	2	3	6			
Maintenabilité (Tx effort / evolutions)	6	2	12			
Respect des bonnes pratiques (patterns, ...)	8	2	16			
Coûts/ROI (Return Of Investment)	3	3	9			
Pérénité des choix techniques	8	3	24	Pérénité :		
La phase vis-à-vis des besoins clients (Client lourd ? Web ?)	9	4	36	Sociétés en informatique (ESN)	Privilégier des outils novateurs : MVC-	
Sécurité (Attaques, gouvernance, ...)	5	5	25	Sociétés conventionnelles	Produits récents éprouvés ex : .Net	
Qualité de la façade (design)	3	2	6	Activité critique (banques, assurances, aviation, ...)	Veilles technos garanties (Cobol, IBM	
Lacunes vis-à-vis de la concurrence ?	5	2	10			
Particularité singulières ? Se démarquer	8	2	16			
Acceptabilité des acteurs techniques	5	3	15			
			0	Evaluations :		
			0	Auto-évaluele (en interne) - risque bialsé		
			0	"Audit" - acteur tiers.		
			0			
			0			
			0			
			0			
		Score final :	209			

SQL → 1 Requête ⇒ n Opérations
↓
Arbre

Prog fo :



Générations

1G → Assembleur
2G → Prog naturelle, script, Spaghetti
3G → Prog Structurée → C
4G → Prog OO (Analyse)
→ UML, Design Pattern,
éléments de langage

5G → Prog fonctionnelle

3G : pointeurs sur fonctions

4G : Evénements (Délégués en .net)

- Delégués (Ref Multicast de fct / Prog évènementielle)

→ fct anonyme

→ Simplifié en fct Lambda

→ Si cette fonction ne fait qu'une opération évaluée, on peut renvoyer simplement sa valeur

⇒ Expression Lambda

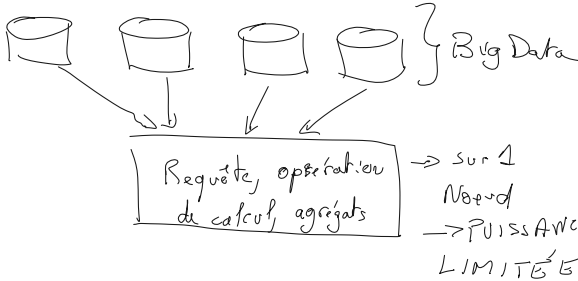
Big Data = Donnée tellement volumineuse
que ne peut tenir "sur 1 serveur"

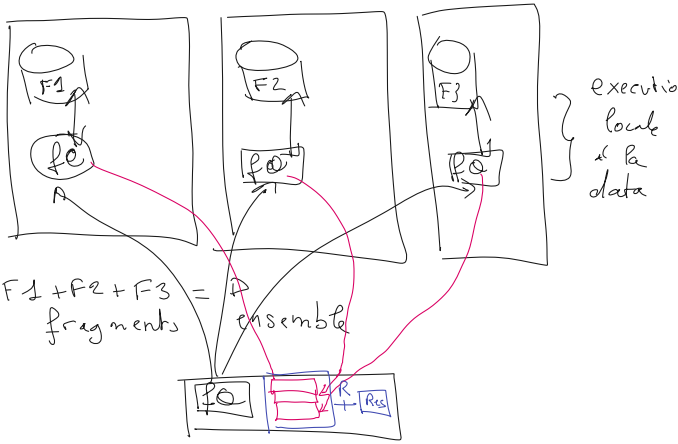
Big Compute = Calculs trop lourds / longs
pour tourner sur "1 système"

Big Data / Compute } sur du matériel conventionnel

Doug Cutting → Hadoop

Généralement, en analyse de data:





R = Réduire les agrégats

Paradigme = Map Reduce

(Pattern)

Prog fonctionnelle en C#

```
class Program
{
    1 référence
    static void action()
    {
        Console.WriteLine("ok");
    }


    // Déclarer une forme d'expression :
    delegate int MyFunction(int a, int b);  // Je déclare une signature d'opération fonctionnelle

    0 références
    static void Main(string[] args)
    {
        // A : Syntaxe via prog évènementielle
        Action a = null;
        a += new Action(action);
        a();

        Console.WriteLine("------------------");
        // B : Méthodes anonymes
        a += delegate () { Console.WriteLine("ok 2"); };
        a();


        Console.WriteLine("------------------");
        // C : Méthodes Lambda ( dsimplification d'écriture
        a += () => { Console.WriteLine("ok 3"); };
        a();

        Console.WriteLine("------------------");
        // D : Expression Lambda  : retirer le scope, et le ;
        a += () => Console.WriteLine("ok 4")    ;   // pas génial en présentation de formation
        a();


        // Programmation fonctionnelle = Renvoi de la valeur passée en expression
        // Cablage dynamique ( valeur via les tests en if, .... ) - contrairement au cablage statique à la compilation
        // Mais ce n'est pas de l'enchaînement de fonctions
        MyFunction operations = null;
        operations += (x, y) => x + y;
        operations += (x, y) => x * y;

        Console.WriteLine("------------------");
        Console.WriteLine("Executions fonctionnelles : ");
        Console.WriteLine(operations(3, 2));

        // Exemple LINQ :
        var valeurs = new[] {
            new { A = 1, B = 20 },
            new { A = 2, B = -50 },
            new { A = 3, B = 40 },
            new { A = 4, B = 30 },
            };


        valeurs.Where(c => c.A > 2).OrderBy(c => c.B).ToList().ForEach(e => Console.WriteLine(e.A + ":" + e.B));

        // Langages fonctionnels : conçu autour de l'écriture fonctionnelle ( concis ) : F#, Scala, Caml, Lisp, Haskell

    }
}
```