

# Approche WEB API

---

LES WEBSERVICES « FACILES »

# Introduction

---

WEB API

# Introduction

---

Avec l'introduction des technologies de service, Microsoft propose WCF, permettant de mettre en place des services utilisant les protocoles tels http, tcp, msmq,... avec des possibilités « infinies » de configuration, sécurité, etc...

Cependant, de par sa complexité, WCF n'est pas aisé à prendre en main.

De plus, les développeurs se sont rendu compte que la plupart des services ont réellement besoin de communiquer via http (et non par tctp, msmsg, named pipe,...) et sans réelle sécurité complexe tel la sécurité de fédération.

Microsoft répond donc à ce besoin en fournissant *Web API* qui permet de façon « simple » de produire des web services HTTP, basés sur JSON/XML et appliquant REST

# Les avantages

---

WEB API

# Les avantages

---

## A. La configuration

Contrairement à WCF, qui nous oblige à créer des contrats, des endpoints, ... nous n'avons pratiquement rien à configurer.

En général, nous avons simplement besoin d'URL REST, quelques arguments de liaison et des réponses en XML ou Json.

## B. REST

ASP.NET Web Api propose un système de routes permettant de mettre en place une architecture REST rapidement.

Contrairement à WCF, où un service a une adresse vers un fichier physique, Web Api possède un système de mapping RESTFull permettant de s'abstraire de son lien physique.

Cela permet de spécifier un Controller (comme en MVC) et les méthodes *Put, Get, Post, Delete* sont automatiquement mappées à celui-ci permettant d'obtenir des adressages simples

# Les avantages

---

## C. Interopérabilité

De par l'utilisation de REST, nous obtenons un service disponible pour toutes les plateformes pouvant effectuer des requêtes http.

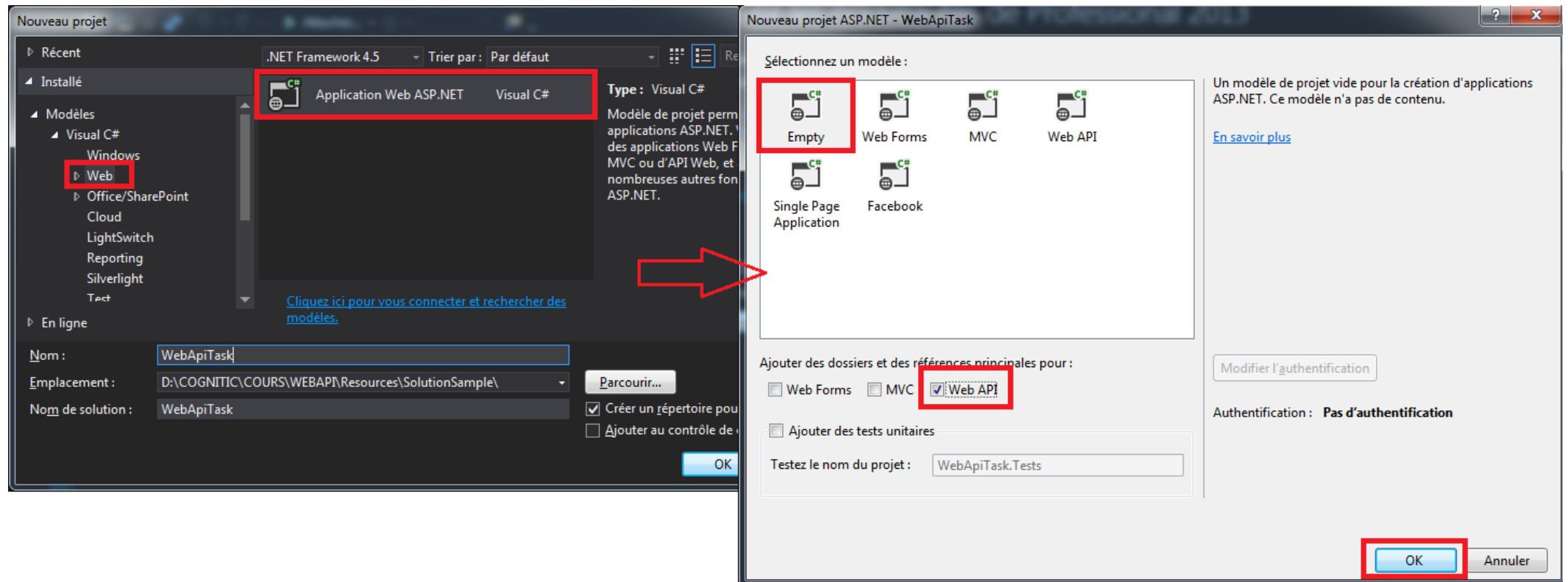
De plus, en communiquant via JSON et XML, nous contournons la complexité de l'échange des messages SOAP. Ce qui permet d'éviter l'utilisation de « librairies » permettant de gérer SOAP lors de la consommation du service.

JSON apporte de nombreux avantages :

- Léger (+/- 50% plus léger que SOAP)
- Structuré
- Facilement consommable de par sa syntaxe proche du javascript

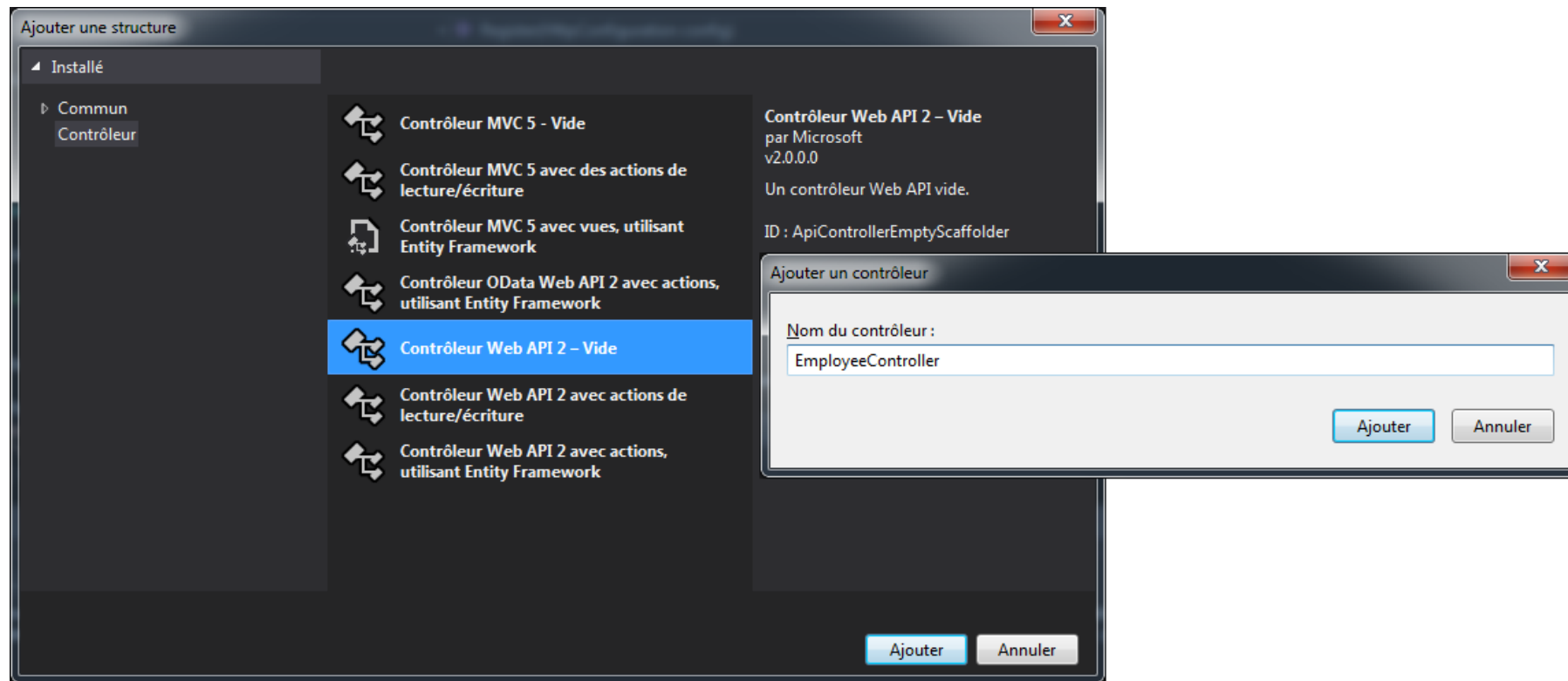
# Premier pas

Notre première application Basique commence par la création d'un projet de type Web Api



# Premier pas

Une fois la solution créée, nous allons construire notre Controller qui se chargera de répondre aux requêtes HTTP.





# Premier pas

---

Nous pouvons constater que notre Controller hérite de la classe *ApiController*.

```
public class EmployeeController : ApiController
{
}
```

Cette classe nous propose toutes les méthodes nécessaires pour réagir aux requêtes http, suivre l'état du model utilisé, obtenir les informations sur l'utilisateur, effectuer des « routing »,...

L'application se basera sur une classe Employee que nous allons créer sous le dossier Models de notre solution.

```
public class Employee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

# Premier pas

---

Nous complétons ensuite notre Controller avec une liste statique d'employés

```
private IList<Employee> list = new List<Employee>()
{
    new Employee(){ ID=1, FirstName="Jhon", LastName="Person"},
    new Employee(){ ID=2, FirstName="Fitz", LastName="Morre"},
    new Employee(){ ID=3, FirstName="Gérald", LastName="Meurice"},
    new Employee(){ ID=4, FirstName="Ken", LastName="Gillis"},
    new Employee(){ ID=5, FirstName="Eddy", LastName="Delange"}
};
```

Et les 4 méthodes permettant les actions CRUD.

## 1. GET

```
/// <summary>
/// GET api/employees
/// </summary>
/// <returns>retour ne la liste des employés</returns>
public IEnumerable<Employee> Get()
{
    return list;
}
```

```
/// <summary>
/// // GET api/employees/12345
/// </summary>
/// <param name="id">Identifiant unique de l'employé recherché</param>
/// <returns>L'employé correspondant à l'id</returns>
public Employee Get(int id)
{
    return list.Where(e => e.ID == id).FirstOrDefault();
}
```

# Premier pas

---

## 2. POST

```
/// <summary>
/// POST api/employees
/// </summary>
/// <param name="employee">L'employé a ajouter</param>
public void Post(Employee employee)
{
    int maxId = list.Max(e => e.ID);
    employee.ID = maxId + 1;
    list.Add(employee);
}
```

## 3. PUT

```
/// <summary>
/// PUT api/employees/12345
/// </summary>
/// <param name="id">Identifiant unique de l'employé a modifier</param>
/// <param name="employee">L'employé modifié</param>
public void Put(int id, Employee employee)
{
    int index = list.ToList().FindIndex(e => e.ID == id);
    list[index] = employee;
}
```

# Premier pas

---

## 4. DELETE

```
/// <summary>
/// DELETE api/employees/12345
/// </summary>
/// <param name="id"></param>
public void Delete(int id)
{
    Employee employee = Get(id);
    list.Remove(employee);
}
```

A ce stade, nous avons un service REST fonctionnel que nous pouvons tester via les urls `/api/Employee`, `api/Employee/1`, etc...

# Modifier les conventions

---

WEB API

# Modifier les conventions

---

Par défaut, pour capturer les requêtes HTTP GET, les actions doivent commencer par GET.

Exemple : Get, GetEmployee, GetEmployeeById,...

De même pour les requêtes HTTP POST, HTTP PUT,... nous devons respecter cette convention.

Cependant, pour des besoins de clarté, business,... nous aimerions pouvoir faire abstraction de cette convention.

Cela est possible via le tag *[AcceptVerbs(...)]*

# Modifier les conventions

---

[AcceptVerbs(...)] possède deux signatures :

- AcceptVerbs(string Method)

Permet de spécifier la méthode http qui sera capturée par l'action.

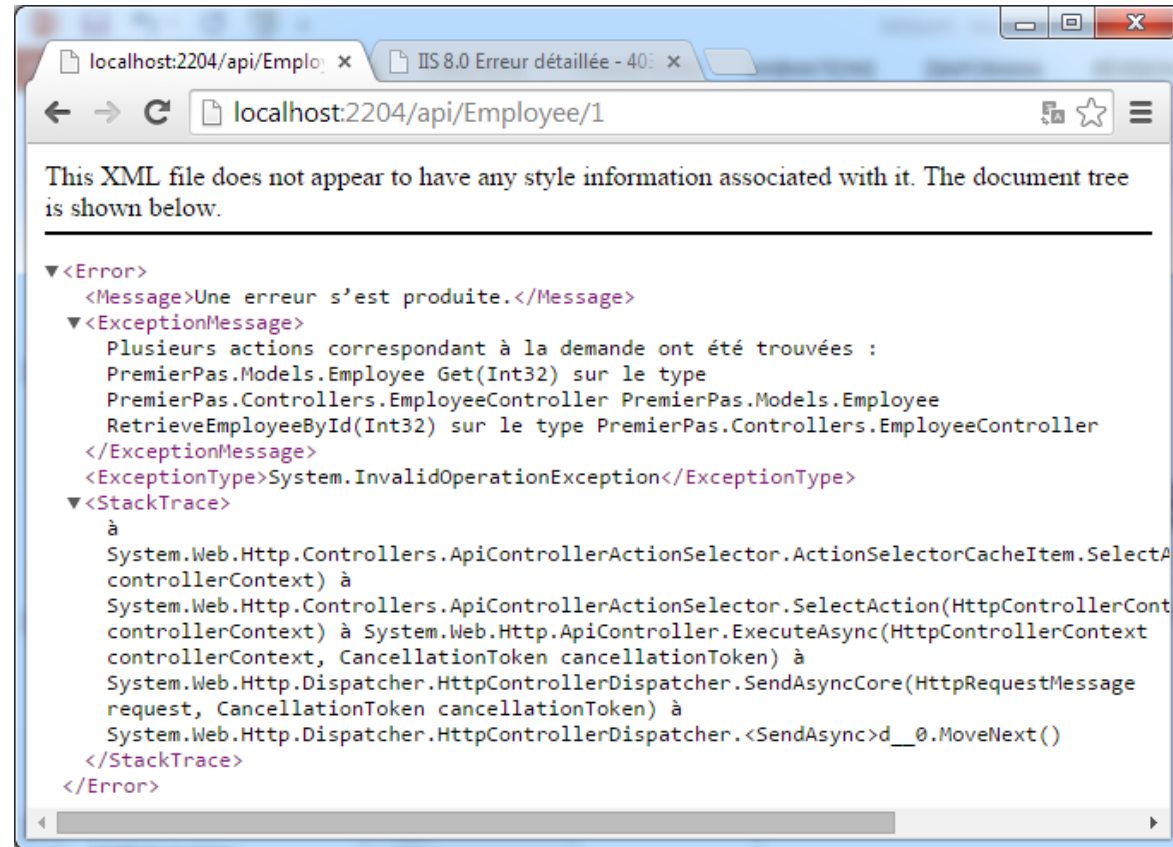
Exemple :

```
//Exemple AcceptVerbs//  
[AcceptVerbs("GET")]  
public Employee RetrieveEmployeeById(int id)  
{  
    return list.Where(e => e.ID == id).FirstOrDefault();  
}
```

## **ATTENTION :**

**La mise en place de cette méthode en plus d'une méthode commençant par GET provoquera une exception car le système aura alors deux point d'entrée pour la même méthode**

# Modifier les conventions





# Modifier les conventions

---

- AcceptVerbs(params string[] Method)

Permet de spécifier plusieurs verbs pour une même méthode

Exemple :

```
//Exemple AcceptVerbs//  
[AcceptVerbs("GET", "POST")]  
public Employee RetrieveEmployeeById(int id)  
{  
    return list.Where(e => e.ID == id).FirstOrDefault();  
}
```

Utiliser AcceptVerbs ne change pas la façon d'interroger le service, seul la méthode appelée

<http://localhost:2204/api/Employee/1> ==> RetrieveEmployeeById(1)

# Modifier les conventions

Nous pouvons également remplacer l'interrogation REST par une interrogation RPC basé sur le nom de la méthode comme en MVC.

Pour cela, nous devons modifier le fichier qui décrit les routes (WebApiConfig)

```
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

```
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

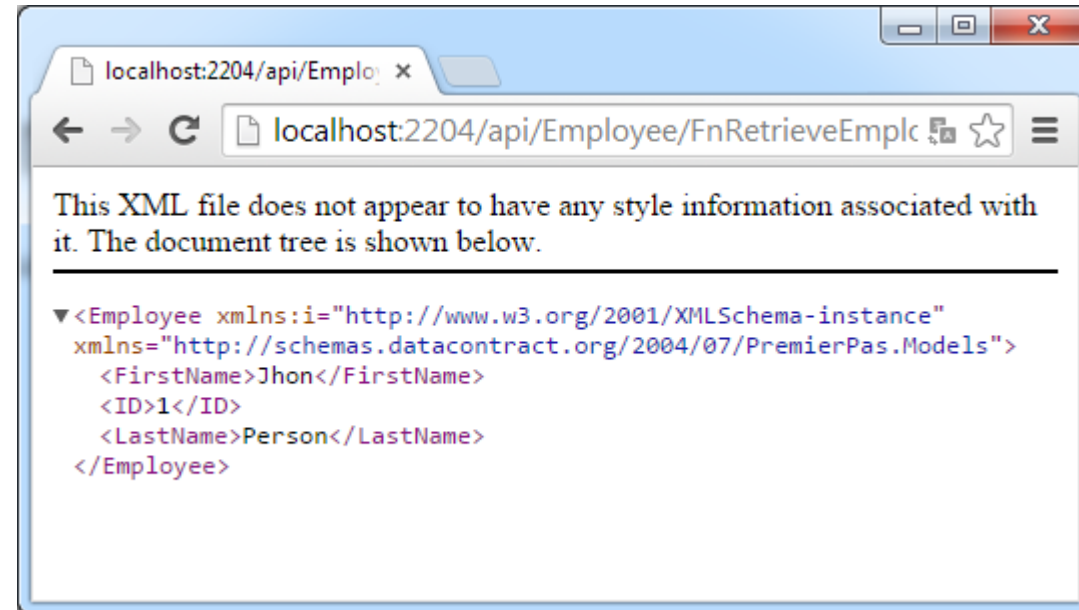
    config.Routes.MapHttpRoute(
        name: "RpcApi",
        routeTemplate: "RpcApi/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

# Modifier les conventions

Nous pouvons ensuite créer des fonctions sans spécifier de verbs afin d'exécuter les différentes méthodes

```
[HttpGet]
public Employee FnRetrieveEmployeeById(int id)
{
    return list.First(e => e.ID == id);
}
```



# Les routes

---

WEB API

# Les routes

---

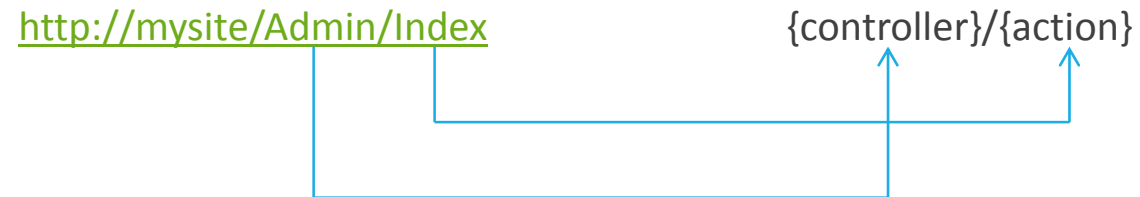
Le fichier WebApiConfig nous permet de définir comment le moteur d'ASP.net doit mapper les demandes et les actions à exécuter

Chaque route contient un Url pattern qui est comparé à l'url demandée.

Si le pattern correspond, il est utilisé pour résoudre l'url.

Les urls peuvent être décomposées en segments. Chaque segments est séparé par un /.

Le système de routing analyse chaque segment



# Les routes

---

URL	Verb	Methode
/api/Employee	GET	Get()
/api/Employee/1	GET	GET(int id)
/api/Employee/1	DELETE	DELETE(int id)
/api/Employee	POST	POST(Employee E)
/api/Employee/1	PUT	PUT(int id, Employee E)

# Les routes

## Définir des valeurs par défaut

La valeur par défaut est utilisé lorsque l'url ne match avec aucun pattern

```
routes.MapRoute(  
    "Default", // Nom d'itinéraire  
    "{controller}/{action}/{id}", // URL avec des paramètres  
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Paramètres par défaut  
);
```

## Segment static

Tous les segments ne sont pas nécessairement variables.

Si nous désirons supporter l'url suivante

<http://mydomain.com/Public/Home/Index>

Nous écrirons

```
routes.MapRoute("", "Public/{controller}/{action}", new { controller = "Home", action = "Index" });
```

**Remarque :**  
**Les routes sont appliquées dans l'ordre de leurs déclarations**

# Les routes

## L'attribut [Route]

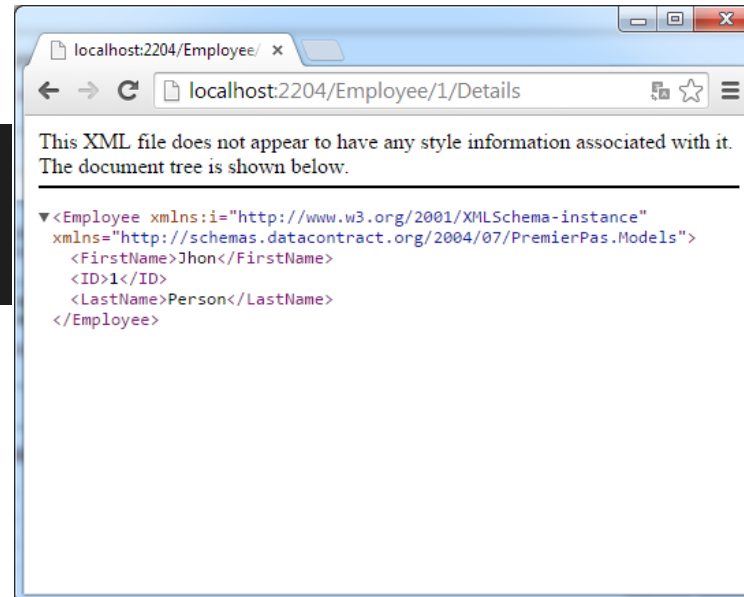
Le fichier WebApiConfig permet de paramétrer les routes à 1! endroit.

Cependant, il est parfois difficile de représenter certaines routes de manière standard.

L'attribut [Route] permet de préciser la route qui doit mapper avec l'appel de la méthode.

## Exemple :

```
[Route("Employee/{id}/details")]
public Employee Get(int id)
{
    return list.Where(e => e.ID == id).FirstOrDefault();
}
```





# Les routes

Nous pouvons également utiliser cet attribut afin de permettre un versionning

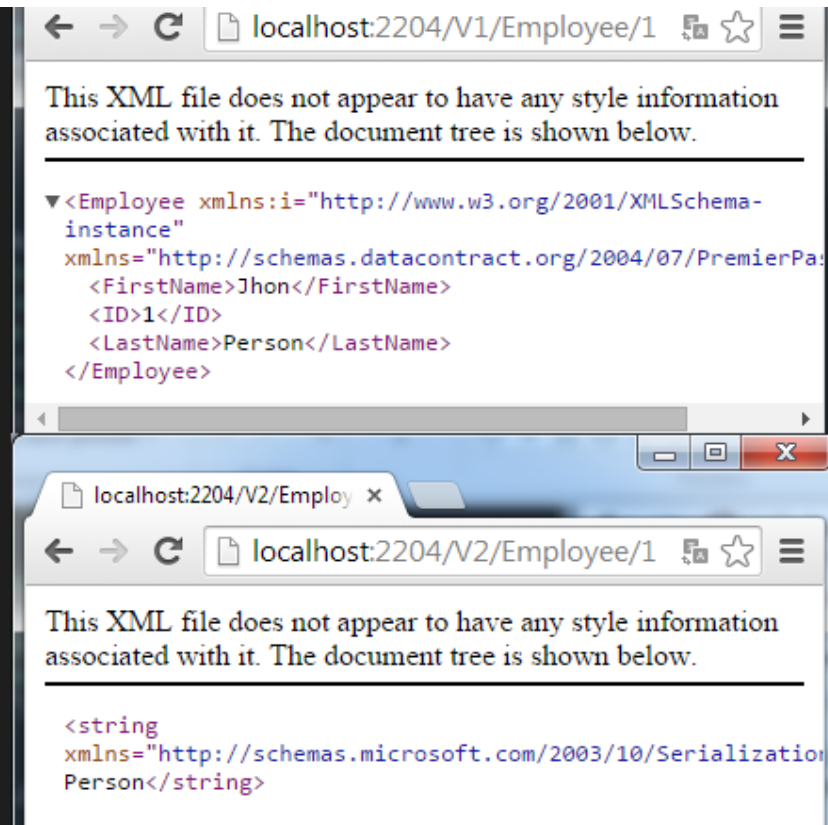
Exemple :

```
#region Versioning
/// <summary>
/// // GET api/V1/employees/12345/
/// </summary>
/// <param name="id">Identifiant unique de l'employé recherché</param>
/// <returns>L'employé correspondant à l'id</returns>
[Route("V1/{controller}/{id}")]
public Employee Get(int id)
{
    return list.Where(e => e.ID == id).FirstOrDefault();
}

/// <summary>
/// // GET api/V2/employees/12345
/// </summary>
/// <param name="id">Identifiant unique de l'employé recherché</param>
/// <returns>L'employé correspondant à l'id</returns>
[Route("V2/{controller}/{id}")]
[AcceptVerbs("GET")]
public String RecupGetEmployeev2(int id)
{
    Employee ep = list.Where(e => e.ID == id).FirstOrDefault();

    return ep.FirstName + " " + ep.LastName;
}

#endregion
```



# Les routes

---

Afin de préciser l'utilisation des routes, nous pouvons utiliser les contraintes.

```
/// <summary>
/// GET api/employees
/// </summary>
/// <returns>retour ne la liste des employés</returns>
[Route("{id:int}")]
public Employee Get(int id)
{
    return list.First(e=>e.ID==id);
}
```

```
/// <summary>
/// GET api/employees
/// </summary>
/// <returns>retour ne la liste des employés</returns>
[Route("{id:int:min(1)}")]
public Employee Get(int id)
{
    return list.First(e=>e.ID==id);
}
```

# Les routes

Constraint	Description	Example
alpha	Accepte les lettres a-z et A-Z	{x:alpha}
bool	Uniquement les valeurs booléennes	{x:bool}
datetime	Accepte les format datetime	{x:datetime}
decimal	Autorise les valeurs décimales	{x:decimal}
double	Autorise les valeurs doubles	{x:double}
float	Autorise les valeurs float	{x:float}
guid	Accepte un GUID	{x:guid}
int	Int uniquement	{x:int}
length	Défini la longueur ou une collection de longueurs acceptées	{x:length(6)} {x:length(1,20)}
long	Long int uniquement	{x:long}

# Les routes

Constraint	Description	Example
max	Valeur maximale	{x:max(10)}
maxlength	Taille maximale	{x:maxlength(10)}
min	Valeur minimale	{x:min(10)}
minlength	Taille minimum	{x:minlength(10)}
range	Une page de valeurs int.	{x:range(10,50)}
regex	Doit respecter l'expression régulière	{x:regex(^\\d{3}-\\d{3}-\\d{4}\$)}

# Les routes

---

Si les contraintes fournies ne suffisent pas, nous pouvons créer nos propres contraintes en implémentant l'interface *IHttpRouteConstraint*

L'interface nous oblige à implémenter une méthode `Match` renvoyant un booléen.

Nous obtenons dans cette méthode, les informations sur la requête et les paramètres

# Les routes

```
public class ZeroContraint:IHttpRouteConstraint
{
    /// <summary>
    /// Permet de contraindre la valeur d'un entier > 0
    /// </summary>
    /// <param name="request">La requête html</param>
    /// <param name="route">la route empruntée</param>
    /// <param name="parameterName">le nom du paramètre a traiter</param>
    /// <param name="values">Un dictionnaire contenant les paramètres et leurs valeurs</param>
    /// <param name="routeDirection">Définit la direction de la route</param>
    /// <returns></returns>
    public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,
        IDictionary<string, object> values,
        HttpRouteDirection routeDirection)
    {
        object value;
        //Récupération de la valeur du paramètre
        if (values.TryGetValue(parameterName, out value) && value != null)
        {
            long longValue;
            if (value is long)
            {
                longValue = (long)value;
                return longValue != 0;
            }

            string valueString = Convert.ToString(value, CultureInfo.InvariantCulture);
            if (Int64.TryParse(valueString, NumberStyles.Integer,
                CultureInfo.InvariantCulture, out longValue))
            {
                return longValue != 0;
            }
        }
        return false;
    }
}
```

```
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services

    var constraintResolver = new DefaultInlineConstraintResolver();
    constraintResolver.ConstraintMap.Add("nonzero", typeof(PremierPas.Helper.ZeroContraint));
    config.MapHttpAttributeRoutes(constraintResolver);
}
```

```
[Route("{id:nonzero}")]
public Employee Get(int id)
{
    return list.First(e => e.ID == id);
}
```

# Utiliser JSON

---

WEB API

# Utiliser JSON

---

Web Api peut envoyer les résultat en XML ou en Json.

Le choix d'un de ces deux format va dépendre de la requête http.

Par exemple, lorsque nous testons notre site avec Chrome, nous avons la requête suivante qui est transmise :

```
▼ Request Headers view source  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8  
Accept-Encoding: gzip, deflate, sdch  
Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4  
Connection: keep-alive  
Host: localhost:2204  
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36
```

Et nous recevons donc le résultat en Xml

```
<Employee xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/PremierPas.Models">  
  <FirstName>Jhon</FirstName>  
  <ID>1</ID>  
  <LastName>Person</LastName>  
</Employee>
```



# Utiliser JSON

Nous avons cependant la possibilité de contourner ce principe afin de permettre l'envoi en json.

Pour cela, nous ajoutons la ligne suivante dans la méthode *Register* du fichier *WebApiConfig*

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));
```

Avec la même requête,

## ▼ Request Headers [view source](#)

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

Accept-Encoding: gzip, deflate, sdch

Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4

Connection: keep-alive

Host: localhost:2204

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36

Nous obtiendrons un json

× Headers Preview Response Timing	
1	{ "ID": 1, "FirstName": "Jhon", "LastName": "Person" }

# Utiliser JSON

- La sérialization

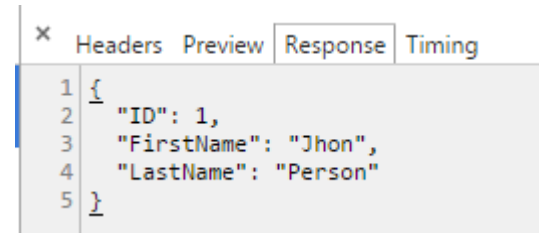
Par défaut, tout les attributs public sont sérialisé mais nous pouvons utiliser l'attribut `[JsonIgnore]` pour omettre certaines propriétés. Cet attribut fait partie de l'assembly `Newtonsoft.Json`.

```
public class Employee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [JsonIgnore]
    public string NickName { get; set; }
}
```

Il est également possible d'indenter le Json généré en ajoutant les lignes suivantes dans la méthode `Register(WebApiConfig)`

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
```



	×	Headers	Preview	Response	Timing
1				{	
2				"ID": 1,	
3				"FirstName": "Jhon",	
4				"LastName": "Person"	
5				}	

# Utiliser JSON

Pour des raisons de visibilité, nous pouvons également permettre l'écriture en CamelCase des noms des propriétés.

Pour cela, nous faisons appel à la propriété *ContractResolver* en lui attribuant la valeur *CamelCasePropertyNamesContractResolver* qui fait partie du namespace *Newtonsoft.Json.Serialization*

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;  
json.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;  
json.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
```

**SANS**

×	Headers	Preview	Response	Timing
1		{		
2		"ID": 1,		
3		"FirstName": "Jhon",		
4		"LastName": "Person"		
5		}		

**AVEC**

×	Headers	Preview	Response	Timing
1		{		
2		"id": 1,		
3		"firstName": "Jhon",		
4		"lastName": "Person"		
5		}		

# Utiliser JSON

---

# Sécurité

---

WEB API

# Sécurité

---

Web API considère que l'authentification se fait via le HOST. La configuration se fait donc via IIS .

Quand l'hôte authentifie le client, il crée un objet *Principal* qui représente le contexte de sécurité et contient notamment un objet *Identity* contenant les informations de l'utilisateur

Si l'utilisateur est authentifié, la propriété *IsAuthenticated* de l'objet *Identity* est passé à true.

## ■ L'autorisation

L'autorisation vient après dans le processus , au plus proche du Controller.

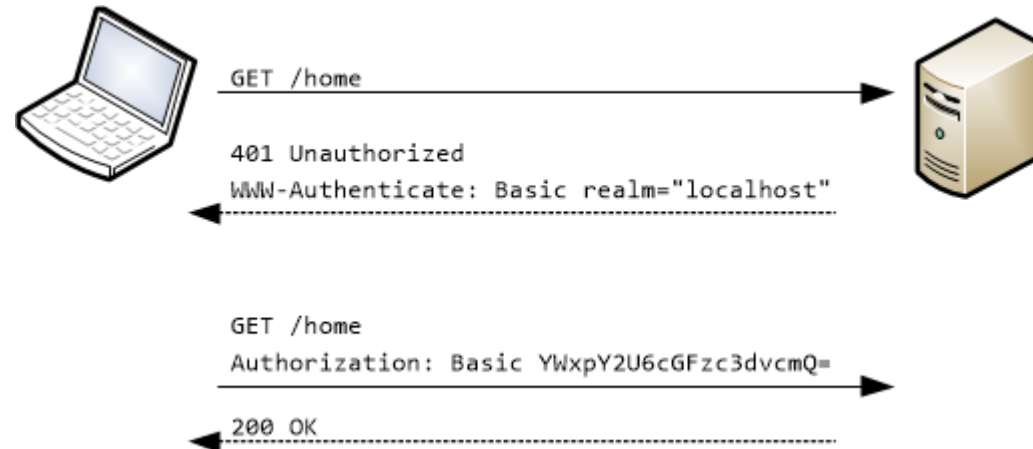
# Direct Authentication

---

WEB API

# Direct Authentication

Dans ce mode, c'est le client qui va , directement, fournir ses credentials au service.



Les credentials sont passé dans le header du message et sont non-cryptés (Dans un cas réel, utiliser le protocole https)



# Sécurité

---

Pour permettre l'échange des informations, nous aurons donc besoin de traiter le header des requêtes.

Cfr BasicAuthWebApi.sln

# Consommer

---

WEB API

# Consommer

---

La consommation d'un service Web Api Rest se fait via l'objet *HttpClient*

En effet, nous consommons ce type de service comme une page web dont le retour contient notre json/XML.

## 1) Get

```
HttpClient client = new HttpClient();
client.BaseAddress = new Uri("http://localhost:2204/apiC/");
HttpResponseMessage response = client.GetAsync("Employee/1").Result;
Console.WriteLine("{0} - {1}", (int)response.StatusCode, response.ReasonPhrase);
if (response.IsSuccessStatusCode)
{
    var employee = response.Content.ReadAsAsync<EmployeeDto>().Result;
    Console.WriteLine("{0}\t{1}\t{2}",
        employee.ID,
        employee.FirstName,
        employee.LastName);
}
Console.ReadKey();
```

*ReadAsAsync* permet de peupler directement un objet local en mappant les nom/types de propriétés avec les données reçues

# Consommer

---

## 2) Get (en précisant le format de réception)

```
HttpClient client = new HttpClient();
client.BaseAddress = new Uri("http://localhost:2204/apiC/");
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json", 0.8));
// client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/xml", 0.9));
HttpResponseMessage response = client.GetAsync("Employee/1").Result;
Console.WriteLine("{0} - {1}", (int)response.StatusCode, response.ReasonPhrase);
if (response.IsSuccessStatusCode)
{
    Console.WriteLine(response.Content.ReadAsStringAsync().Result);
}

Console.ReadKey();
```

Il est évidemment possible d'utiliser des bibliothèques supplémentaires afin de gérer les formats json ou xml

# Consommer

---

## 2) POST

```
HttpClient client = new HttpClient();
client.BaseAddress = new Uri("http://localhost:2204/api/");
EmployeeDto ed = new EmployeeDto() { FirstName = "zorro", LastName = "garcias" };

var jsonString = Serialize(ed);
var content = new StringContent(jsonString, Encoding.UTF8, "application/json");
var result = client.PostAsync("Employee", content).Result;

Console.WriteLine("POST");
Console.Read();
int idEmployee = (int)result.Content.ReadAsAsync(typeof(int)).Result;

HttpResponseMessage response = client.GetAsync("Employee/" + idEmployee).Result;
Console.WriteLine("{0} - {1}", (int)response.StatusCode, response.ReasonPhrase);
if (response.IsSuccessStatusCode)
{
    var employee = response.Content.ReadAsAsync<EmployeeDto>().Result;
    Console.WriteLine("{0}\t{1}\t{2}",
        employee.ID,
        employee.FirstName,
        employee.LastName);
}
Console.ReadKey();
```

# Exercise

---

WEB API

# Exercice

---

Il est demandé de créer un service Web Api permettant de gérer les employés et leurs locaux et de mettre en place une application ASP permettant de consommer celui-ci.

Un employé est identifié comme suit :

- EmployeeID int auto-incrementé
- FirstName nvarchar(50)
- LastName nvarchar(50)
- UserName nvarchar(50)
- Password nvarchar(50)
- Role nvarchar(50) (Admin, Visiteur)

Un local est identifié comme suit :

- LocalId int auto-incrementé
- LocalName nvarchar(50)
- LocalFloor int

# Exercice

---

Les fonctionnalités sont les suivantes :

- Ajout/Suppression/Modification des employés
  - ➔ Seul les employés ayant le rôle Admin peuvent effectuer ces opérations
- Lister les employés, Rechercher un employé par son nom
- Ajout/Suppression/Modification des Locaux
  - ➔ Seul les employés ayant le rôle Admin peuvent effectuer ces opérations
- Lister les locaux , Rechercher un local par son étage, son nom.
- Associer les locaux et les employés
  - ➔ Seul les employés ayant le rôle Admin peuvent effectuer ces opérations



# Exercice

---

Le format d'échange devra être en JSON.

L'accès à la base de donnée se fera soit en EntityFramework/ soit en Ado.Net.

Vous pouvez utiliser JQUERY pour consommer le service (<http://api.jquery.com/jquery.getjson/>  
<http://api.jquery.com/jquery.post/> )

Les fonctions permettant la récupération des données devront avoir des noms conviviaux et ne pas respecter les conventions GET,PUT,...

Toutes les fonctions ne requérant pas de droits respecterons la route {Controller}/{id}... Celles requérant des droits devront respecter la route Secure/{Controller}/{id}

L'authentification se fera via la « Direct Authentication »

Les propriétés password et user ne devront pas être sérialisé lors de la demande de listing, details,... bref de consultation.

**BONUS : Les password seront hashé via l'algorithme SHA1**