

Séance 08 : Initiation à l'ASP.NET – Partie III

☑ Introduction aux **API REST**

INF27507 – Technologies du commerce électronique

Prof. Yacine YADDADEN, Ph. D.

Plan

1. Introduction et Contexte
2. Qu'est-ce qu'une **API REST** ?
3. Les **API REST** avec **ASP.NET**
4. Mise en pratique
5. Questions et discussion

Introduction et Contexte

- **Contexte :**

- Plusieurs types de système accédant à une même et unique source de données :
 - Site ou application Web, application mobile, application de bureau, ...
- Fournir de manière *sécurisé* un *service* avec une *couche d'abstraction* avec les données.

- **Problématique :**

- Une approche classique (*tel que nous l'avons vu sur **ASP.NET Core***) ne le permet pas,
- Il faut adopter une nouvelle approche afin d'avoir accès à ce genre de service.

- **Solution :**

- Utilisation d'une **API REST** dont l'implémentation est possible en utilisant :
 - Java (Spring Boot), JavaScript (Express.js), **C#** (ASP.NET Core), Python (Flask et Django), ...

Plan

1. Introduction et Contexte
2. **Qu'est-ce qu'une API REST ?**
 - a. Les contraintes **REST**
 - b. Description du fonctionnement
 - c. Anatomie d'une requête **HTTP**
 - d. Méthode, code de statut et formatage du contenu
3. Les **API REST** avec **ASP.NET**
4. Mise en pratique
5. Questions et discussion

Qu'est-ce qu'une API REST ?

- **Définition I :** « *une **API** (**A**pplication **P**rogramming **I**nterface) ou interface de programmation applicative est une solution sous forme d'un ensemble de règles permettant la communication entre différent types d'application afin d'échanger des données ou des services. » , On distingue deux styles :
 - **API REST** (**R**epresentational **S**tate **T**ransfer) :
 - C'est un style d'architecture, les méthodes sont standardisées, moins de liaison client-serveur.
 - **API SOAP** (**S**imple **O**bject **A**ccess **P**rotocol)
 - C'est un protocole, le serveur et le client sont étroitement liés.*
- **Définition II :** « **REST** a été défini en 2000 par Roy Fielding. C'est une suite de contraintes dont il faut tenir compte lors de la conception d'une **API**. L'objectif est l'optimisation et la facilité d'utilisation. » ,

Contraintes REST

Il y a six principales contraintes qui ont été définies :

1. Client-Server :

- *Il y a une séparation des rôles entre le client et le serveur,*

2. Stateless Server :

- *Les requêtes contiennent l'ensemble des informations nécessaires au traitement,*
- *Il n'y a pas de notion de session côté serveur.*

3. Cache :

- *Les réponses obtenues depuis le serveur doivent être cacheable côté client (sauvegarde en local).*

Contraintes REST – Suite

4. Uniform interface :

- *La méthode de communication entre le client et le serveur doit être uniforme.*

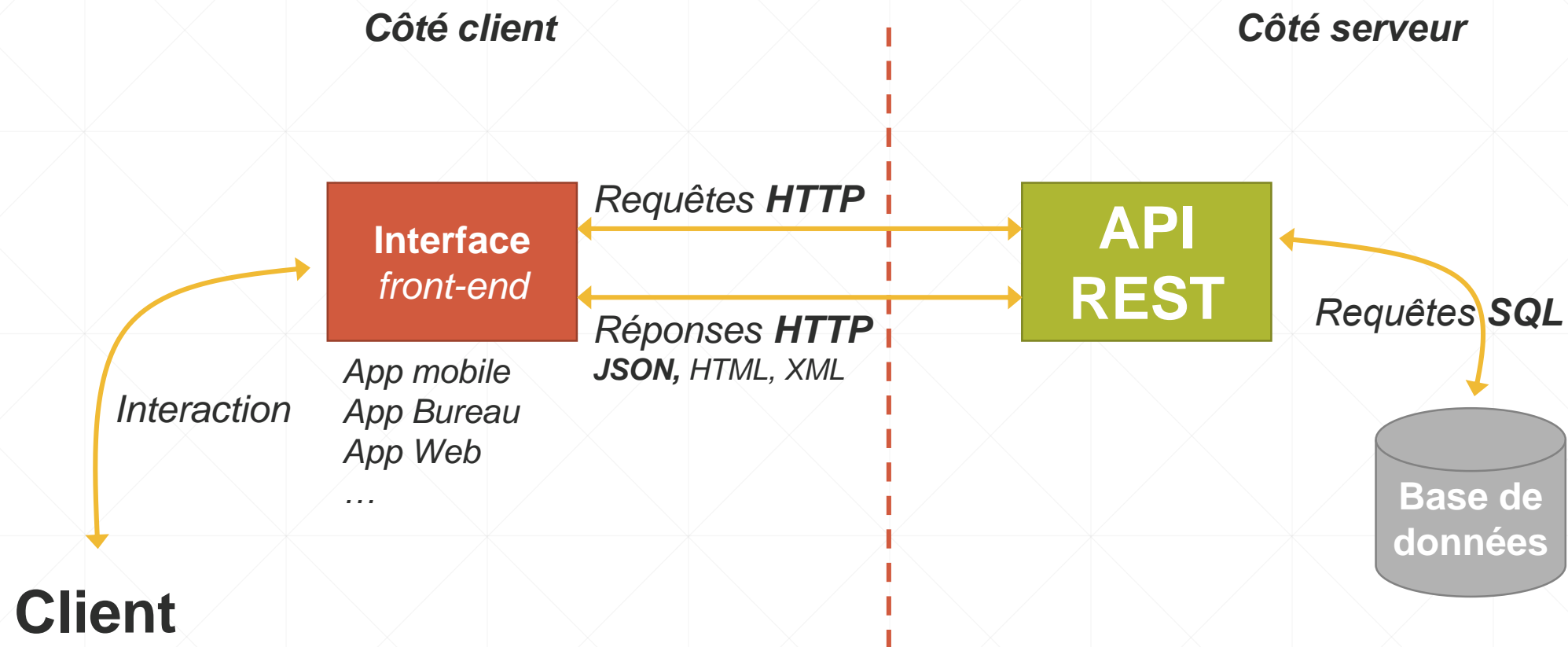
5. Layered System :

- *Possibilité d'ajouter des couches intermédiaires (serveur proxy, pare-feu, ...).*

6. Code-on-Demand Architecture (optionnelle) :

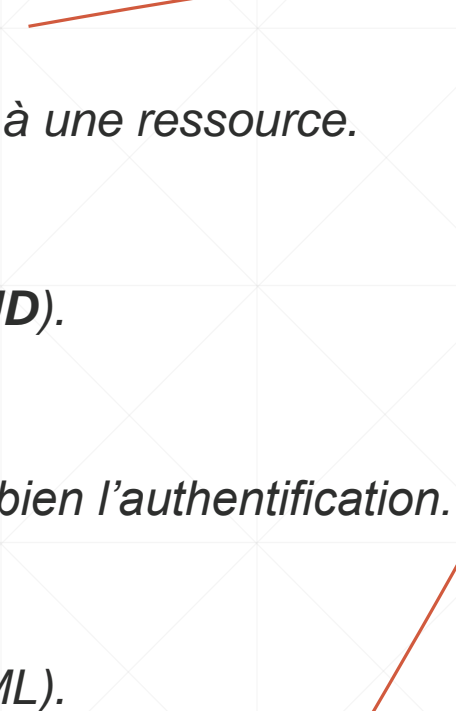
- *L'architecture doit permettre l'exécution du code côté client.*

Diagramme de fonctionnement

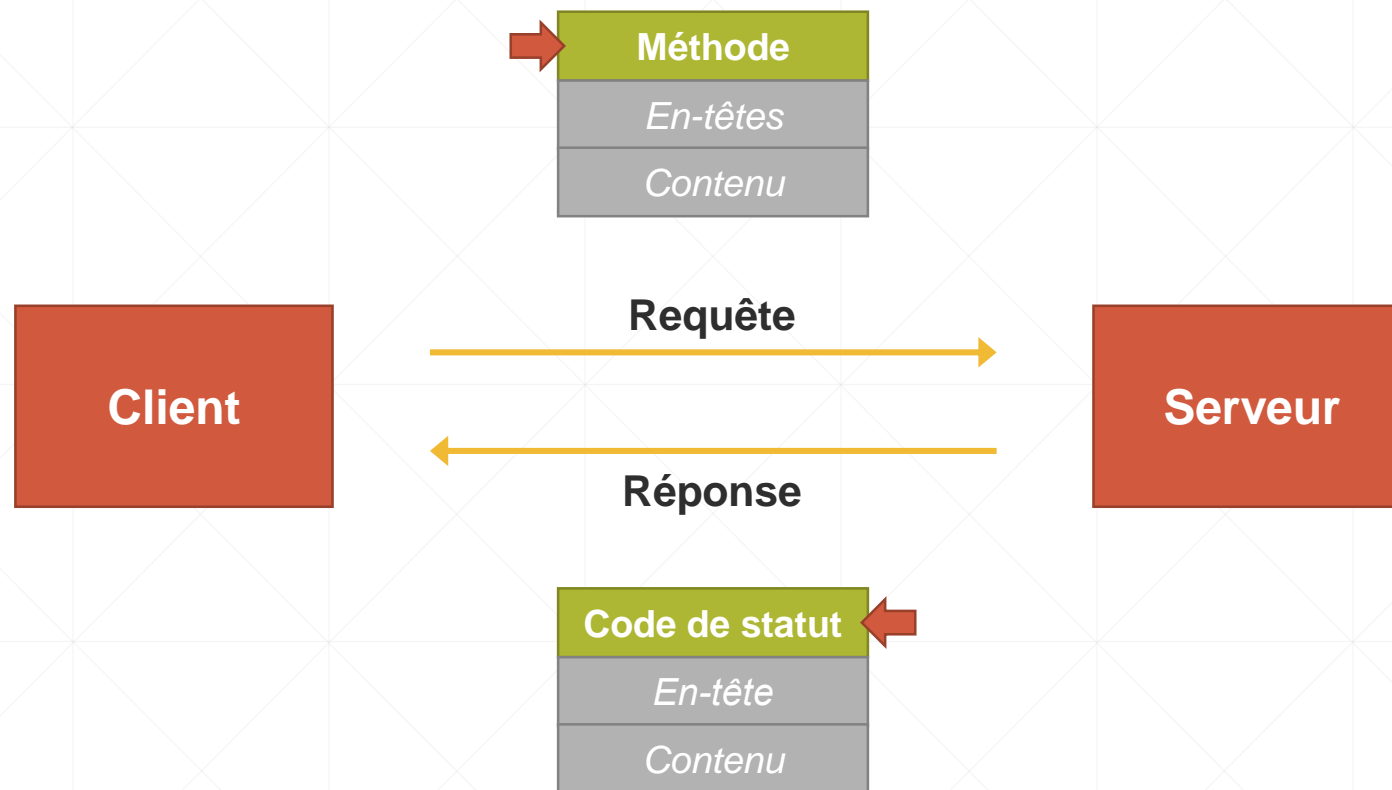


Anatomie d'une requête HTTP

Afin d'envoyer une requête **HTTP**, il y a quatre composants à considérer :

1. **Point de terminaison (endpoint) : $URI = URL + URN$**
 - C'est l'**URI** (domaine, port et chemin) utilisé pour accéder à une ressource.
 2. **Méthode (method) :**
 - Méthode **HTTP** utilisée pour accéder à la ressource (**CRUD**).
 3. **En-têtes (headers) :**
 - Informations utilisé pour définir le format des données ou bien l'authentification.
 4. **Corps/données (body/data) :**
 - Données transmises (chaîne de caractères : **JSON** ou XML).
- 

Anatomie d'une requête HTTP



Les différentes méthodes utilisées

Méthode HTTP	CRUD	Description
GET	Lecture	<i>Retourne des données</i>
POST	Création	<i>Crée un nouvel enregistrement</i>
PUT ou PATCH	Mise à jour ou modification	<i>Modification d'un enregistrement existant</i>
DELETE	Suppression	<i>Suppression d'un enregistrement existant</i>

Les codes de statut

Code de statut	Description
2xx <i>Success</i>	<i>La requête s'est effectuée avec succès</i>
3xx <i>Redirection</i>	<i>L'emplacement de l'URL demandé a changé</i>
4xx <i>Client error</i>	<i>Une erreur est survenue lors de l'envoi de la requête (URL)</i>
5xx <i>Server error</i>	<i>Une erreur est survenue lors du traitement de la requête</i>

Les codes de statut

Le minimum à utiliser

Code de statut	Description
200	<i>Ok</i>
400	<i>Bad Request</i>
500	<i>Internal Error</i>

Et supplémentaire


Code de statut	Description
201	<i>Created</i>
304	<i>Not Modified</i>
404	<i>Not Found</i>

Code de statut	Description
401	<i>Unauthorized</i>
403	<i>Forbidden</i>

Formatage du contenu

Par le types de contenu qui peuvent être retournés, il y a :

Type	Type MIME
JSON	Application/json
XML	text/xml



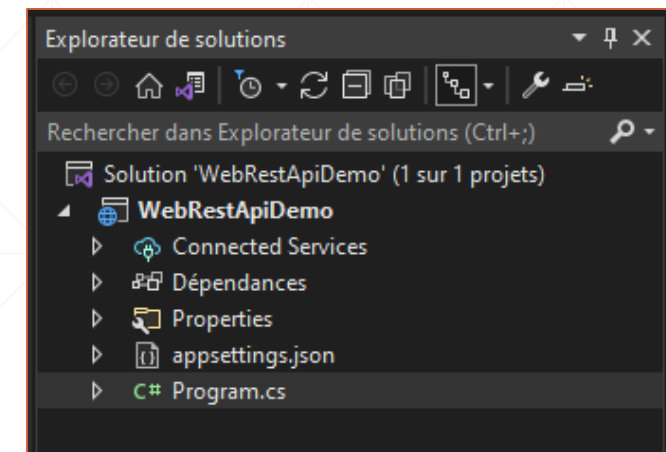
Plan

1. Introduction et Contexte
2. Qu'est-ce qu'une **API REST** ?
3. **Les API REST avec ASP.NET**
 - a. Création d'une première **API REST**
 - b. Implémentation des différentes requêtes
 - c. Consommation de l'**API REST**
 - d. Sécurité et authentification par jeton
4. Mise en pratique
5. Questions et discussion

Création d'un premier projet Web API

Afin de créer un premier projet en **ASP.NET Web API**, il faut :

1. Lancer **Microsoft Visual Studio** puis *Create a new project*,
2. Il faut en suite sélectionner *ASP.NET Core Web API*,
3. Donner un nom à votre projet, exemple : **api-name**,
4. Appuyer sur *Next*, puis :
 - Décocher *Configure for HTTPS*,
 - S'assurer qu'*Authentication Type* est à *None*.
5. Appuyer sur *Create* et voilà !



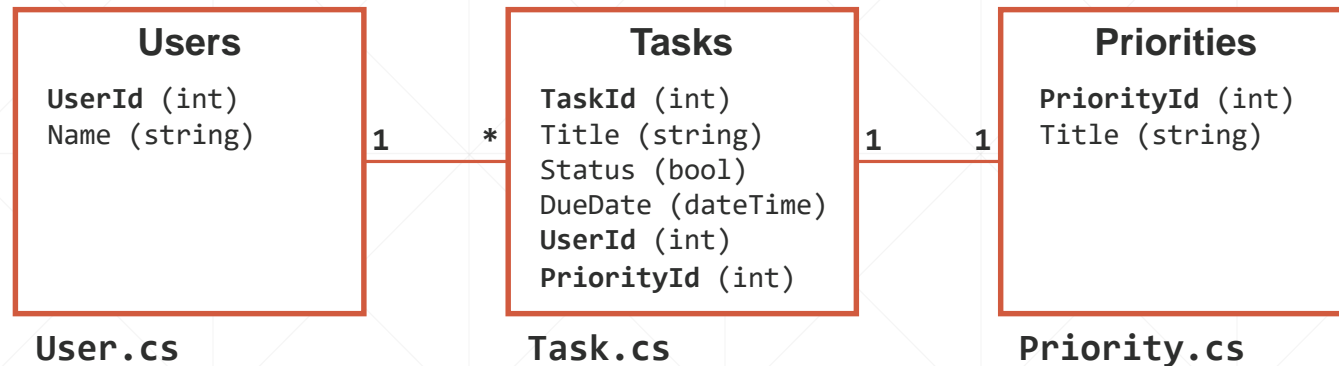
Création d'un premier projet Web API – Suite

Une fois le projet créé, on doit effectuer certaines configurations :

1. Se rendre dans les *Properties* du projet actuel,
2. Dans *Debug*, il faut décocher *Launch browser*,
3. Configurer l'**URL** d'accès dans *App URL*, exemple : <http://localhost:5000>
4. Il faut modifier le contenu de : *Program.cs*

Les modèles

- Une **API REST** nous permettra de *manipuler* des *données* à travers des *requêtes HTTP* et pour cela, il faut :
 1. Mettre en place **Entity Framework Core**,
 2. Créer les *modèles (Gestion des tâches)* dans un dossier **Models**,
 3. Générer *la base de données*.
- Dans ce cours, on utilisera une **API REST** de *Gestion des tâches* avec les *tables* :
 - **Utilisateurs** → *Users*,
 - **Tâches** → *Tasks*,
 - **Priorités** → *Priorities*.



Les contrôleurs

- Afin d'intercepter les *requêtes HTTP* entrantes et renvoyer des *réponses HTTP*, il est nécessaire de créer un *contrôleur* pour chaque *entité* :
 - **TaskController.cs** : Il contiendra l'ensemble des opérations appliquées sur l'entité *Task.cs*
 - **UserController.cs** : Il contiendra l'ensemble des opérations appliquées sur l'entité *User.cs*
- Pour créer un nouveau *contrôleur* :
 1. Dans le dossier **Controllers**, il faut :
 - **Aller** : *Add > Controller... > API > API Controller – Empty* puis appuyer sur **Add**
 2. Donner un nom au nouveau *contrôleur* : **TaskController.cs** puis appuyer sur **Add**

Task

Controller.cs

nom

partie commune

Services et Configuration

Pour La configuration des services, nous n'avons besoin pour l'instant que du **MVC** :

```
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddMvc(option => option.EnableEndpointRouting = false);  
  
var app = builder.Build();  
  
app.UseMvc();  
  
app.Run();
```

Utilisation du service MVC

Déclaration du service MVC

Program.cs

Méthodes de routage

En **ASP.NET**, il y a principalement deux *méthodes de routage* distinctes :

- **Basée sur les modèle** (*template* ou *pattern*) :
 - Elle permet de rediriger la *requête* suivant un *contrôleur* et un *action*,
 - Ce modèle est défini dans le fichier **Program.cs** du projet.
- **Basée sur les attributs** :
 - Elle est utilisée habituellement pour *gérer* les *requêtes HTTP* des **API REST**,
 - Elle se base sur l'utilisation de *décorateurs* au niveau du *contrôleur*.

Méthodes de routage

- Basée sur les modèle (*template* ou *pattern*) :

URL : http : //localhost:56361 / Home / Index
protocole *Hôte* *Contrôleur* *Action*

- Basée sur les attributs :

URL : http : //localhost:56361 / Tasks /
protocole *Hôte* *Entité*

Méthode : [GET]
Verbe HTTP

Configuration du contrôleur

- Avant de définir des *requêtes* au niveau du contrôleur, on doit mettre définir certaines *configurations*,
- On utilisera le code suivant :

TaskController.cs

Le format des données généré
La route utilisée
Le type de contrôleur utilisé

Décorateurs

Contexte utilisé pour l'accès et
la manipulation des données

```
→ [Produces("application/json")]  
→ [Route("api/Tasks")]  
→ [ApiController]  
public class TaskController : ControllerBase  
{  
    private Models.TaskManagerDbContext _context;  
  
    public TaskController()  
    {  
        _context = new Models.TaskManagerDbContext();  
    }  
}
```

Les variables d'URL

Il y a principalement deux manières distinctes de faire passer des *variables* ou *paramètres* dans l'**URL** :

- Définir le *paramètre* dans la *route* directement :
 - **Exemple** : <http://localhost:5000/api/Tasks/1>
- L'ajouter comme un *paramètre* supplémentaire
 - **Exemple** : <http://localhost:5000/api/Tasks?taskId=1>

Les deux variable sont récupérée de la même façon au niveau de l'action (au niveau du *contrôleur*), c'est juste la définition qui est différente.

Les requêtes – GET

- La *requête HTTP* de type **GET** est généralement utilisée pour la récupération de données à partir du serveur,

- On utilisera le code :

Méthode HTTP + Route et Paramètre

*Types de réponse (codes de statut)
retournés → pour la documentation*

Swagger

*Il faut toujours gérer les éventuelles
exceptions pour éviter que l'API REST
ne plante. On utilisera des **try ... catch***

TaskController.cs

```
[HttpGet("{taskId}", Name = "GetTask")]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
[ProducesResponseType((int)HttpStatusCode.OK)]
public IActionResult GetTask(int taskId)
{
    try
    {
        Models.Task task = _context.Tasks.Find(taskId);
        if(task != null)
            return Ok(task);
        else
            return StatusCode((int)HttpStatusCode.NotFound);
    }
    catch (Exception) { }

    return StatusCode((int)HttpStatusCode.BadRequest);
}
```

Paramètre d'URL

*Deux manières de définir
un code de réponse*

Les requêtes – POST

- La *requête HTTP* de type **POST** est généralement utilisée pour envoyer de nouvelles données au serveur,
- On utilisera le code :

*Task et TaskModel sont pratiquement identiques sauf que le premier est utilisé pour la construction de la base de données et le deuxième n'est utilisé que pour encapsuler les données transmises par à travers la requête **POST**. Il y a moins d'attributs dans la deuxième classe.*

```
[HttpPost]
[ProducesResponseType(typeof(models.TaskModel), (int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.Created)]
public IActionResult AddTask([FromBody] models.TaskModel model)
{
    try
    {
        Données récupérées à partir du
        corps de la requête → JSON
        Models.Task task = new Models.Task()
        {
            Title = model.Title,
            Status = model.Status,
            DueDate = model.DueDate,
            UserId = model.UserId,
            PriorityId = model.PriorityId
        };
        _context.Tasks.Add(task);
        _context.SaveChanges();
        return CreatedAtAction(nameof(GetTask), new { taskId = task.TaskId }, task);
    }
    catch (Exception) { }

    Redirection vers la méthode permettant
    de récupérer une tâche par son ID

    return BadRequest();
}
```

Définir un modèle qui encapsule les données envoyées

Les requêtes – PATCH et PUT

- Les *requêtes HTTP* de type **PATCH** et **PUT** permettent la mise à jour d'une ressource ou donnée existante sur le serveur,
- On utilisera le code :

PATCH est généralement utilisé pour une mise à jour partielle c'est-à-dire, une partie des valeurs des attributs.

PUT est généralement utilisé pour une mise à jour complète c'est-à-dire, l'ensemble des valeurs des attributs.

```
[HttpPut("{taskId}")]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
[ProducesResponseType((int)HttpStatusCode.OK)]
public IActionResult EditTask(int taskId, [FromBody] Models.TaskModel model)
{
    try
    {
        Models.Task task = _context.Tasks.Find(taskId);
        if (task != null)
        {
            task.Title = model.Title ?? task.Title;
            task.DueDate = model.DueDate ?? task.DueDate;
            _context.SaveChanges();
            return Ok();
        }
        else
            return StatusCode((int)HttpStatusCode.NotFound);
    }
    catch (Exception) { }

    return BadRequest();
}
```

Permet de remplacer une valeur existante par une nouvelle si cette dernière n'est pas **NULL**.

Les requêtes – DELETE

- La *requête HTTP* de type **DELETE** permet de la suppression d'une ressource ou donnée existante sur le serveur,
- On utilisera le code :

D'abord, il faut vérifier l'existence de la ressource à supprimer en passant par son **ID**.

Renvoyer le code de statut adéquat au cas où la ressource n'a pas été trouvée.

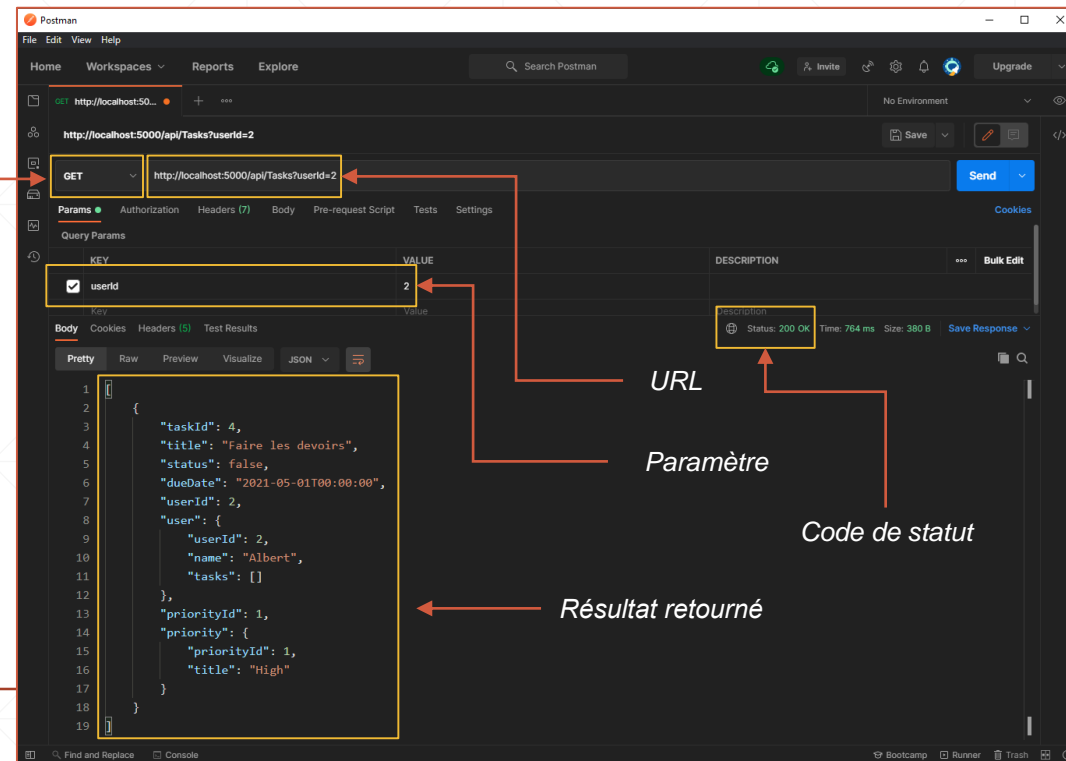
```
[HttpDelete("{taskId}")]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
[ProducesResponseType((int)HttpStatusCode.OK)]
public IActionResult RemoveTask(int taskId)
{
    try
    {
        Models.Task task = _context.Tasks.Find(taskId);
        if (task != null)
        {
            _context.Remove(task);
            _context.SaveChanges();
            return Ok();
        }
        else
        {
            return StatusCode((int)HttpStatusCode.NotFound);
        }
    }
    catch (Exception) { }

    return BadRequest();
}
```

Consommer une API REST – avec POSTMAN

- **Définition :** « C'est une plateforme collaborative dédiée pour le développement d'**API**. Elle possède une interface graphique conviviale et intuitive permettant de tester facilement des **API REST** développées. »,
- **Lien :** <https://www.postman.com/>

Méthode utilisée, dans ce cas, c'est GET.



Consommer une API REST – POWERSHELL

Il y a d'autres manières de consommer une API REST, parmi lesquels :

- L'utilisation de la commande **CURL**
 - Généralement utilisée dans les systèmes Linux : <https://curl.se/>
- L'utilisation de commandes **PowerShell**
 - La commande utilisée est **Invoke-RestMethod**¹
 - On doit lui ajouter un certains nombre de paramètres suivant la requête.

Exemple

¹ <https://docs.microsoft.com/fr-fr/powershell/module/microsoft.powershell.utility/invoke-restmethod>

Consommer une API REST – POWERSHELL

1. On commence d'abord par définir les paramètres :

Il est possible d'ajouter d'autres paramètres comme Headers ou Body.

```
$params = @{  
    Uri      = "http://localhost:5000/api/Tasks?userId=2"  
    Method   = "GET"  
    ContentType = "application/json"  
}
```

2. Ensuite, il faut appeler la commande : `Invoke-RestMethod @params`

3. On a la possibilité de faire des conversion à partir ou vers le **JSON** :

- `ConvertTo-Json` ou `ConvertFrom-Json`

`Invoke-RestMethod @params | ConvertTo-Json`

Résultat

```
> Invoke-RestMethod @params | ConvertTo-Json  
{  
  "value": [  
    {  
      "taskId": 4,  
      "title": "Faire les devoirs",  
      "status": false,  
      "dueDate": "2021-05-01T00:00:00",  
      "userId": 2,  
      "user": {  
        "userId": 2,  
        "name": "Albert",  
        "tasks": ""  
      },  
      "priorityId": 1,  
      "priority": {  
        "priorityId": 1,  
        "title": "High"  
      }  
    }  
  ],  
  "Count": 1  
}
```

Introduction à l'outil SWAGGER

- **Définition :** « *C'est une plateforme collaborative dédiée pour le développement d'**API REST**. Elle est Open Source et permet de faciliter le processus de développement d'**API REST**.* »,
- **Lien :** <https://swagger.io/>
- Les principaux avantages de **SWAGGER** sont :
 - *Il permet de générer automatiquement la documentation de l'**API REST** au format **JSON**,*
 - *Il dispose d'une interface Web conviviale et intuitive,*
 - *Il s'intègre avec de nombreux Framework **back-end**, exemples : **APS.NET**, Django, ...*
 - ...

Utilisation de l'outil SWAGGER

- Avant d'utiliser **SWAGGER**, il est nécessaire d'installer un paquet avec la commande :
 - Commande : `Install-Package Swashbuckle.AspNetCore`
- Ensuite, éditer le fichier **Program.cs** et ajouter les éléments suivants :
 - Dans la partie **builder.Services** :

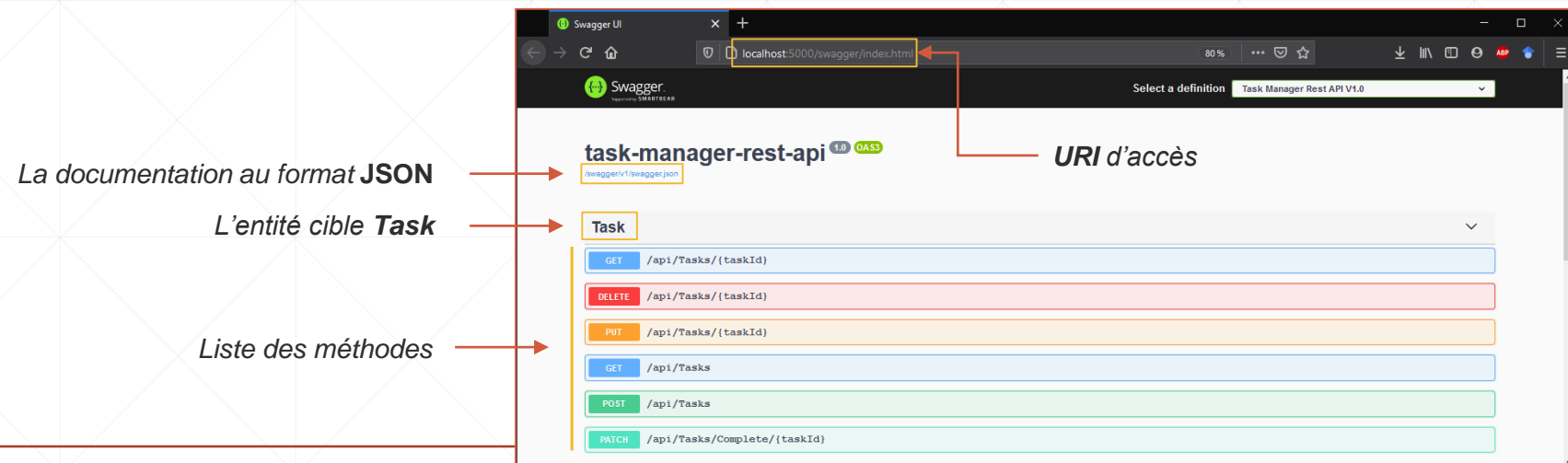
```
builder.Services.AddSwaggerGen();
```

- Dans la partie **app** :

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(config =>
    {
        config.SwaggerEndpoint("/swagger/v1/swagger.json", "Task Manager Rest API V1.0");
    });
}
```

Utilisation de l'outil SWAGGER – Suite

- Afin de lancer automatiquement **SWAGGER** sur la navigateur, il faut :
 1. Se rendre dans les *Properties* du projet actuel,
 2. Dans *Debug*, il faut cocher *Launch browser* et spécifier la page d'accueil : **swagger**.
- En lançant le serveur, on est rediriger vers l'interface suivante :



Gestion des données liées

- Pour gérer les données à partir de tables liées, il faut installer :
 - Commande : `Install-Package Microsoft.AspNetCore.Mvc.NewtonsoftJson`
- Ensuite, éditer le fichier `Program.cs` et ajouter les éléments suivants :
 - Dans la partie `builder.Services` :

```
builder.Services.AddMvc(option => option.EnableEndpointRouting = false).AddNewtonsoftJson(option =>
{
    option.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
});
```

- Ça permet d'utiliser les `.Include()` sans avoir d'erreurs,
- On trouvera des données *imbriquées* dans le **JSON** généré.

Sécurité d'une API REST

- La *sécurité* est un aspect primordial lors du développement d'une application Web ou bien une **API REST**,
- Afin de sécuriser la *communication* entre le client et le serveur, on utilisera un *certificat* **SSL** qui se base sur un *chiffrement* **asymétrique** (clés privée et publique),
- Pour limiter l'accès aux données, il y a plusieurs méthodes d'*authentification* :
 - Authentification par **Cookies**,
 - Authentification **basique** (*lente et non sécuritaire*),
 - Authentification par **Token** ou **Jeton**.



La plus adaptée

Création des utilisateurs

- Afin de créer des *utilisateurs* destinés à l'*authentification*, on doit avoir :
 - **Identifiant**, *il doit être unique pour chaque utilisateur*,
 - **Mot de passe**, *il doit être gardé secret et chiffré avant d'être stocké*,
 - **Rôle**, *il est utilisé pour définir les permissions et autorisations*.
- On utilisera un *paquet* additionnel d'**Entity Framework Core** pour les *utilisateurs* :
 - Commande : **Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore**
→ *Inclu dans* →
- De plus, il est nécessaire de **chiffrer** ou **hacher** le *mot de passe* avant de le stocker sur la base de données, c'est un *règle* afin d'*assurer la sécurité* en cas de vol des informations d'authentification à partir de la base de données.

Authentification par Jeton – Fondamentaux

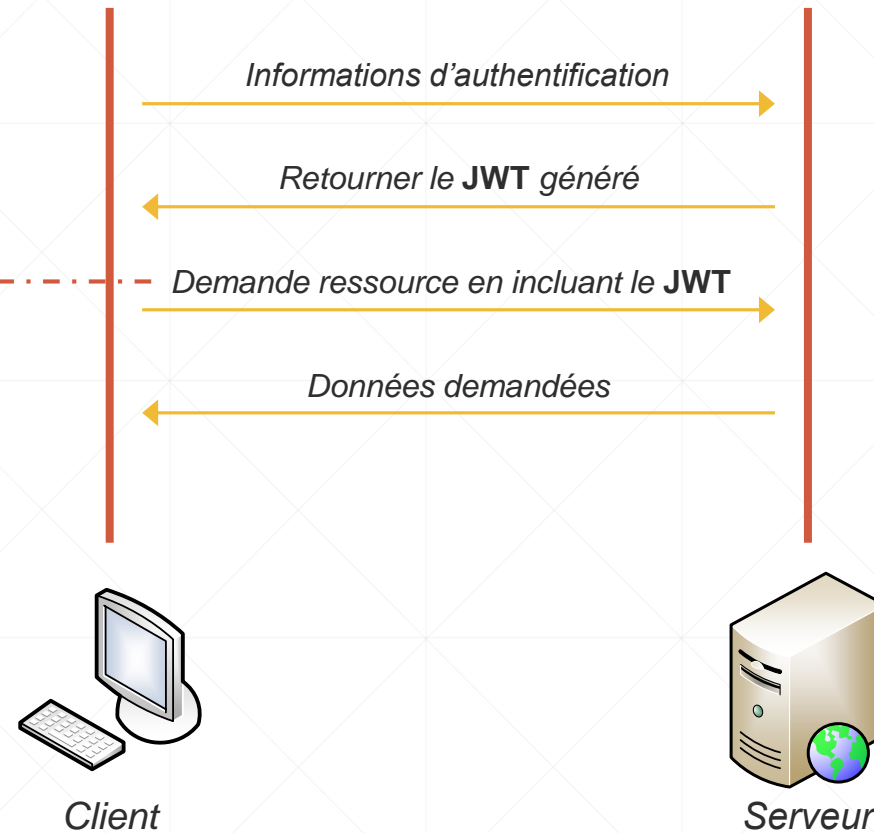
- **Définition** : « *C'est un protocole de sécurité permettant de s'assurer de l'identité de l'utilisateur souhaitant accéder à une certaine ressource.* »,
- Il se base sur l'utilisation des **JWT** ou **JSON Web Token**. C'est une chaîne de caractère chiffrée contenant un ensemble d'informations divisé en trois catégories :
 - **L'en-tête** : *Il définit l'algorithme de chiffrement utilisé et le type de Jeton,*
 - **Le contenu** : *Il contient les informations utiles relatives à l'utilisateur,*
 - **La signature** : *La signature chiffrée permettant d'attester de la validité du contenu du JWT.*
- À la base, le contenu est en **JSON**, mais il est chiffré avec le chiffrement en **base64**,
- Chaque Jeton généré a une durée de vie spécifique qui peut être configurée,
- **Commande** : `Install-Package Microsoft.AspNetCore.Authentication.JwtBearer`

- Il y a un site Web permettant de *décrypter* le **JWT** :
 - Site : <https://jwt.io/>
- Un exemple de **JWT** généré et décodé en utilisant le site :

Encoded	PASTE A TOKEN HERE
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzYm9yZy93cy8yMDAxLzA1LzI1ZW50aXR5L2NsYW1tcy9uYW11IjoieWxiZXJ0IiwiaHR0cDovL3NjaGVtYXMueG1sc29hcC5vcmdvdmVudG10eS9jbGFpbXBvbWVudFtZWlkeW50aWZpZXIiOiIiYmZhMTc3NC01ZjnkLTQwM2Q0ODlkOS0yMzYxZGRmOTMzNDgiLCJodHRwOi8vc2NoZW1hcy5taWVnb3NvZnQuY29tL3dzLzIwMDgvdjYvaWR1bnRpdHkvY2xhaW1zL3JvbGU0Ij1c2VyIiwiZXhwIjoxNjE0NDAwNWdG4LCJpc3MiOiJodHRwOi8vbG9jYXRob3N0OjUwMDAiLCJhdWQiOiJodHRwOi8vbG9jYXRob3N0OjUwMDAifQ.8aT2XLJHFv-RvnvltUyYKCsPY705VuV3_1bHeJAf4A	
Decoded	EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE	<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
PAYLOAD: DATA	<pre>{ "http://schemas.xmlsoap.org/ws/2005/05/identity/ /claims/name": "albert", "http://schemas.xmlsoap.org/ws/2005/05/identity/ /claims/nameidentifier": "8bfa1774-5fd- 403d-89d9-2361ddf93348", "http://schemas.microsoft.com/ws/2008/06/identity/ /claims/role": "user", "exp": 1618800488, "iss": "http://localhost:5000", "aud": "http://localhost:5000" }</pre>

Authentification par Jeton – Schéma

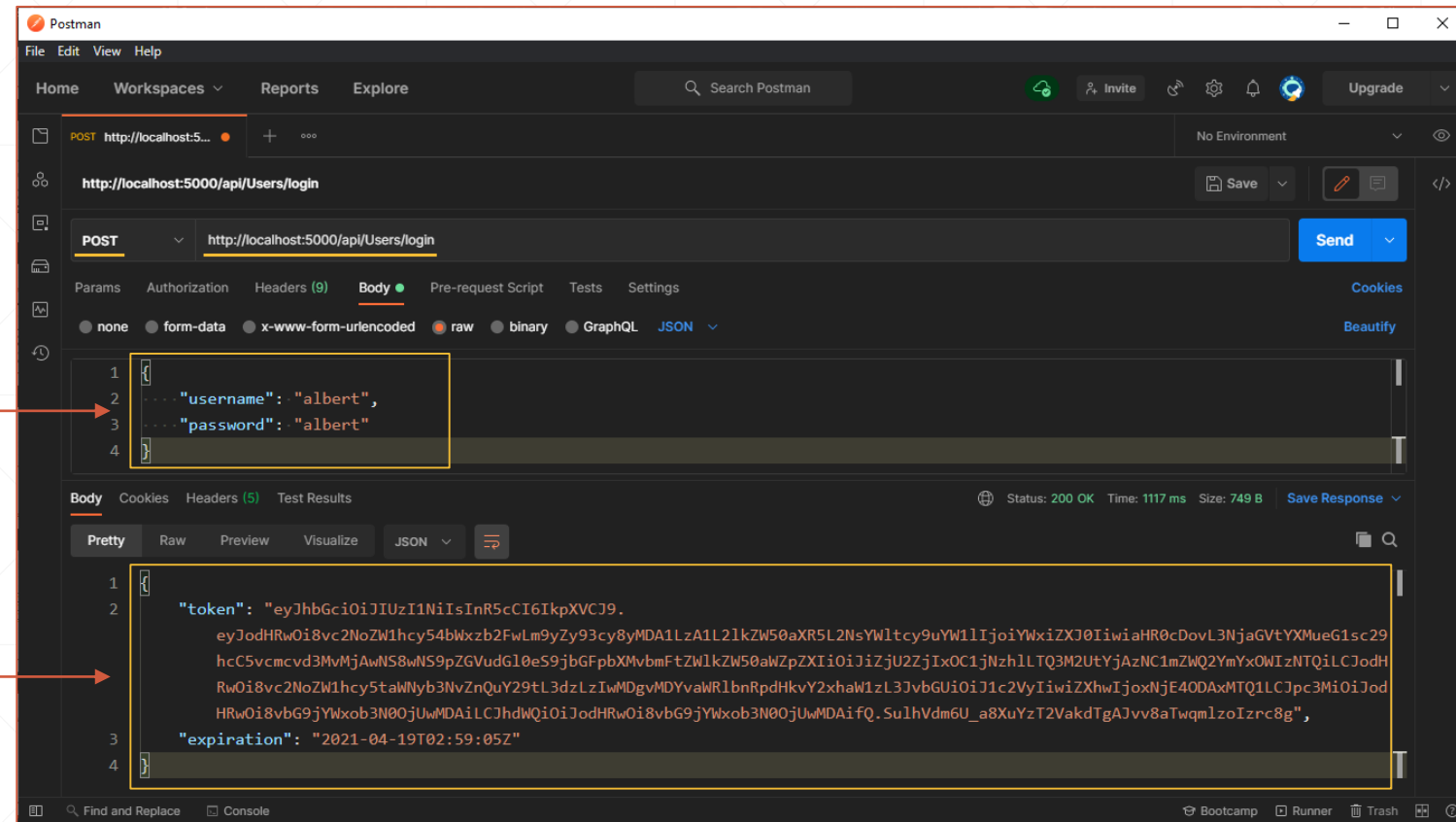
- Le **JWT** doit être inclut dans le les **en-têtes** et plus précisément dans **Authorization**,
- Le **JWT** doit être précédé du mot clé **Bearer** lors de son utilisation.



Authentification par Jeton – Démonstration

Information d'authentification

Le JWT retourné



Authentification par Jeton – Démonstration

En-tête incluant le **JWT**

Ressources retournées après vérification du **JWT**

Dans le cas où la vérification du **JWT** aurait échouée, on aurait eu :

Status: 401 Unauthorized Time: 28 ms Size: 193 B

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/api/Users/user`. The 'Headers' tab is active, showing an 'Authorization' header with the value 'Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2...'. The 'Body' tab is also active, showing a JSON response: `{ "userId": "a139c15f-40fc-4a0b-8f4b-741d5d158280", "message": "[USER] Welcome Albert Bouchard" }`. The status bar at the bottom indicates a successful response with status 200 OK, time 229 ms, and size 268 B. A red arrow points to the 'Authorization' header, and another red arrow points to the JSON response body. A third red arrow points to the 'Status: 401 Unauthorized' message, which is highlighted in a red box.

Mise en application

- Sur l'application de conversion entre :
 - Degré Celsius °C et Degré Fahrenheit °F
- Implémentation de l'authentification par *jeton* :
 - Enregistrement d'un nouvel utilisateur,
 - Connexion à l'aide d'identifiant et mot de passe.
- Implémentation d'une **API REST** :
 - Demande de conversion,
 - Configurer **SWAGGER**,
 - Demande de l'historique.

Questions & Discussion

Bibliographie

1. Kurtz, J., & Wortman, B. (2014). *ASP. NET Web API 2: Building a REST Service from Start to Finish*. Apress.
2. Guérin, B., A. (2016). *ASP.NET avec C# sous Visual Studio 2015 - Conception et développement d'applications Web*. Éditions ENI.
3. Labat, L. (2013). *ASP.NET MVC 4 - Développement d'applications Web en C# - Concepts et bonnes pratique*. Éditions ENI.