Cégep de Chicoutimi

TECHNIQUES DE L'INFORMATIQUE

Introduction aux services de données
420-D52-CH

Consommation d'API web

Méthodes HTTP avec C# : POST, PUT, DELETE

Enseignant: *Emmanuel Francis Constant*

Session Hiver 2026

Les 4 méthodes HTTP principales

📥 GET

Récupérer des données depuis le serveur

✨ POST

Créer une nouvelle ressource

✏️ PUT

Modifier une ressource existante – PATCH aussi

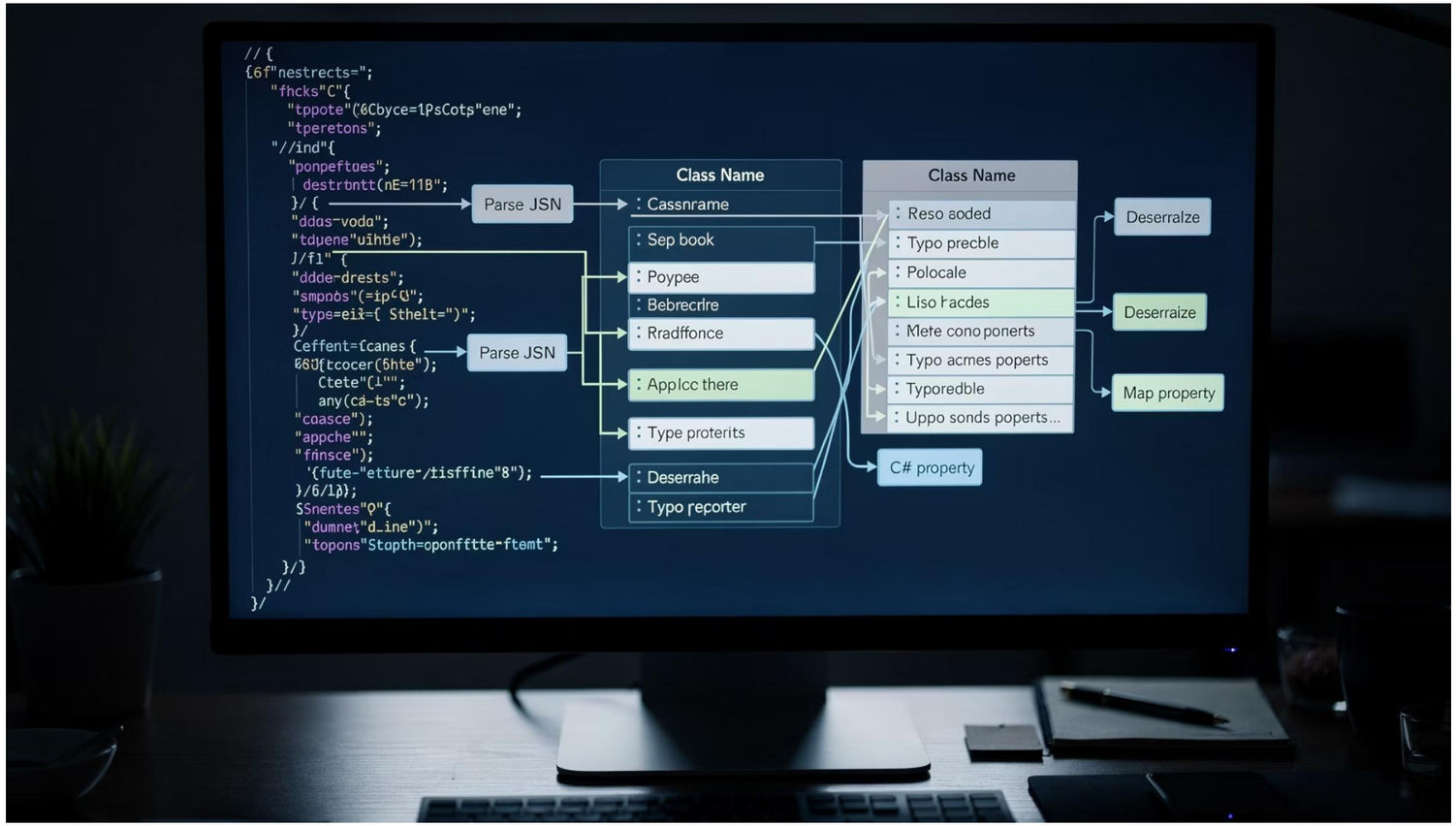
🗑️ DELETE

Supprimer une ressource

Après la simple consultation avec GET , passons à la manipulation des données avec POST, PUT et DELETE

FONDAMENTAL

La base commune - le modèle



Règle d'or

Peu importe la méthode (GET, POST, PUT ou DELETE), l'étape de création des classes modèles est **indispensable**.

Pourquoi ?

Le modèle est le contrat entre l'API et votre application C#. La structure de la classe doit calquer exactement la structure du JSON.

Outils

Newtonsoft.Json ou System.Text.Json pour mapper les propriétés JSON aux propriétés C#.

Architecture - séparation des responsabilités

Pour un code propre et maintenable, nous divisons le travail en trois couches distinctes :

01

Le modèle

La classe de données pure qui représente la structure de la réponse JSON. Elle contient uniquement les propriétés.

02

Le helper (service)

La classe dédiée à la communication HTTP. Elle contient l'instance HttpClient et gère tous les appels asynchrones vers l'API.

03

Logique et validation

Avant d'envoyer (POST/PUT), on valide la saisie utilisateur (champs vides, format d'email, etc.) dans l'interface (WPF ou Console).

Ces 3 étapes s'appliquent aux 4 méthodes : GET, POST, PUT et DELETE

Flux de données : GET vs POST/PUT/DELETE

GET - Récupération

JSON → Objet C#

Désérialisation
JsonConvert.DeserializeObject<T>()

POST / PUT / DELETE - Envoi

Objet C# → JSON

Sérialisation
JsonConvert.SerializeObject()

Concept Clé :

- **GET** : Vous **recevez** du JSON que vous convertissez en objet C#
- **POST/PUT** : Vous **envoyez** un objet C# que vous convertissez en JSON

En résumé : Le code change de sens, mais l'objet manipulé reste le même.

Documentation : méthodes HttpClient

Les quatre méthodes principales de la classe HttpClient pour interagir avec les API REST :

GetAsync

HttpResponseMessage GetAsync(string url)

Récupérer des données depuis le serveur

PostAsync

HttpResponseMessage PostAsync(string url, HttpContent content)

Créer une nouvelle ressource sur le serveur

PutAsync

HttpResponseMessage PutAsync(string url, HttpContent content)

Modifier une ressource existante

DeleteAsync

HttpResponseMessage DeleteAsync(string url)

Supprimer une ressource du serveur

⚠ ATTENTION

Particularité de PUT

Pour ne modifier qu'une partie des données sans renvoyer tout l'objet, on utilise généralement le verbe HTTP **PATCH** au lieu de **PUT**.

Le PATCH est conçu précisément pour les "mises à jour partielles". Au lieu d'envoyer l'objet complet, on n'envoie que les propriétés à changer.

Exemple de scénario :

- Si vous envoyez un **PUT** avec seulement le titre, il risque d'écraser la date, le lieu et le prix par des valeurs nulles
- C'est pour cela que nous faisions un **GET** avant le **PUT** : pour être certains de renvoyer l'objet complet


```
{
    "id": "evt1",
    "titre": "Festival des Rythmes Latins",
    "date": "2026-03-15T19:00:00",
    "lieu": "Théâtre Banque Scotia, Saguenay",
    "artiste": "Orquesta Salsa Boricua",
    "prix": "45$"
},
```

POST : Créer une nouvelle ressource

```
// 1. Créer l'objet
Produit nouveauProduit =
    new Produit { Nom = "Nouveau produit", Prix = 29.99 };

// 2. Sérialiser en JSON
string json = JsonConvert.SerializeObject(nouveauProduit);
var content =
    new StringContent(json, Encoding.UTF8, "application/json");

// 3. Envoyer avec POST
HttpResponseMessage response =
    await client.PostAsync("api/produits", content);

// 4. Vérifier le succès
if (response.IsSuccessStatusCode)
    Console.WriteLine("Produit créé !");
```

Étape 1

Créer l'objet C# avec les données

Étape 2

Convertir l'objet en JSON avec `SerializeObject()`

Étape 3

Envoyer au serveur via `PostAsync()`

Étape 4

Vérifier la réponse du serveur

✓ Notez l'utilisation de SerializeObject() pour convertir l'objet C# en JSON

PUT : Modifier une ressource existante

```
// 1. Récupérer et modifier l'objet
Produit produitModifie = new Produit
{ Id = 5, Nom = "Produit à mettre à jour", Prix = 34.99 };

// 2. Sérialiser en JSON
string json = JsonConvert.SerializeObject(produitModifie);
var content =
    new StringContent(json, Encoding.UTF8, "application/json");

// 3. Envoyer avec PUT (noter l'ID dans l'URL)
HttpResponseMessage response =
    await client.PutAsync("api/produits/5", content);

// 4. Vérifier le succès
if (response.IsSuccessStatusCode)
    Console.WriteLine("Produit modifié !");
```

Différence clé avec POST :

PUT nécessite l'ID de la ressource à modifier directement dans l'URL (ex: api/produits/5)

Le processus de sérialisation est identique à POST, mais l'URL cible une ressource spécifique existante.

Il est recommandé de faire un GET avant pour récupérer l'objet complet et éviter d'écraser des propriétés non envoyées.

PATCH : Modifier une ressource existante

```
// 1. Définir la modification partielle
var modification = new { Prix = 39.99m };

// 2. Sérialiser et préparer le contenu
string json = JsonConvert.SerializeObject(modification);
var content = new StringContent(json, Encoding.UTF8, "application/json");

// 3. Envoyer avec PATCH
HttpResponseMessage response =
    await client.PatchAsync("api/produits/5", content);

// 4. Vérifier le succès
if (response.IsSuccessStatusCode)
    Console.WriteLine("Produit mis à jour partiellement !");
```

Différence clé avec PUT :

PUT remplace TOUT l'objet (champs manquants = null)

PATCH : **Modifie PARTIELLEMENT (ex. seul Prix)**

Le processus de sérialisation est identique à POST, mais l'URL cible une ressource spécifique existante (ex: api/produits/5).

DELETE : supprimer une ressource

```
// 1. Aucune sérialisation nécessaire pour DELETE

// 2. Envoyer avec DELETE (ID dans l'URL)
HttpResponseMessage response =
    await client.DeleteAsync("api/produits/5");

// 3. Vérifier le succès
if (response.IsSuccessStatusCode)
    Console.WriteLine("Produit supprimé !");
else
    Console.WriteLine($"Erreur : {response.StatusCode}");
```

⚠️ Point important

DELETE ne nécessite **PAS de sérialisation** - seulement l'ID dans l'URL

C'est la méthode la plus simple : pas d'objet à créer, pas de JSON à préparer. Seul l'ID de la ressource est requis dans l'URL.

Attention : une suppression est souvent **irréversible**. Validez toujours l'action avec l'utilisateur avant d'exécuter un DELETE.

Tableau comparatif : les 4 méthodes

Méthode	Action	Sérialisation	ID dans URL
GET	Récupérer	Désérialiser (JSON → C#)	Optionnel
POST	Créer	Sérialiser (C# → JSON)	Non
PUT	Modifier	Sérialiser (C# → JSON)	Oui (requis)
DELETE	Supprimer	Aucune	Oui (requis)

> *Retenez que seul GET désérialise, les autres sérialisent (sauf DELETE qui n'a rien à envoyer)*

💡 Notes importantes

La consommation d'API est un pilier de la programmation moderne

Compétence fondamentale

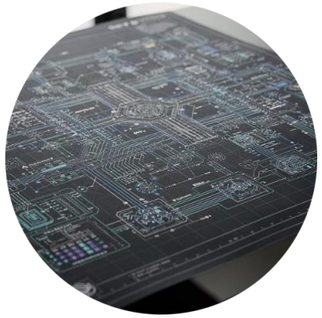
La consommation d'API est une compétence que vous réinvestirez dès cette session dans d'autres cours

Double compétence

Vous apprendrez à créer vos propres API ET à les consommer - deux compétences distinctes mais complémentaires

Utilité pratique

Réalité du marché : Dans le monde professionnel, on passe 80% du temps à consommer des API existantes



Vision globale

Une API n'a de sens que lorsqu'elle est consommé. Maîtriser la création ET la consommation vous donne une compréhension complète