

Exercice Hôpital : solution serveur

Par Alexandre Venet, le 12/02/2022

Ce document présente la solution serveur que j'ai réalisée ces derniers jours, en C# avec le *framework* ASP.NET Core de l'environnement .NET de Microsoft, solution à la date du présent document.

La solution actuelle n'est PAS factorisée. La factorisation fera l'objet d'un traitement ultérieur.

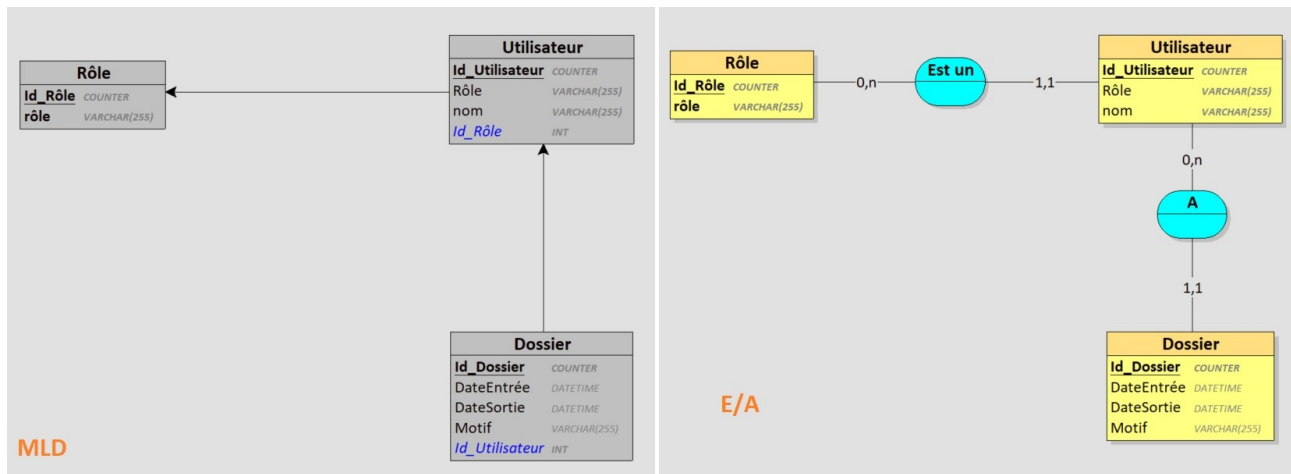
Les objectifs de ce document sont les suivants :

- expliquer le fonctionnement d'une API qu'il est possible de réaliser avec .NET (structure, usage),
- garder une trace écrite pour qui voudra reproduire ce genre d'application,
- fournir à mes camarades de groupe de travail l'essentiel afin qu'ils puissent poursuivre le développement et effectuer leur exposé oral devant leur prochain opposant de Simplon.

Base de données

La base de donnée est de type PostreSQL. Son administration est effectuée sur PGAdmin.

La base de données est structurée selon les schémas suivants (méthode MERISE).



Le script de création est le suivant. Lors de la création avec PGAdmin, il convient de séparer le fichier en deux étapes car l'interface d'administration, contrairement à MySQLWorkbench, ne prend pas en charge la mise à jour des tables à la suite de la création de la base. Pour cela, lancer la commande de création seule, puis une fois la base créée, lancer les commandes de création des tables.

La définition du *timezone* est optionnelle pour les tables. En effet, cette instruction peut être omise si les questions de fuseau horaire ne se posent pas.

```
-- création bdd Postgresql "Hopital"
```

```
CREATE DATABASE "Hopital"
```

```
WITH
```

```
OWNER = postgres
```

```
ENCODING = 'UTF8'
```

```
LC_COLLATE = 'French_France.1252'
```

```
LC_CTYPE = 'French_France.1252'
```

```
TABLESPACE = pg_default
```

```
CONNECTION LIMIT = -1;
```

```
-- -----
```

```
SET timezone = "+00:00";
```

```
-- -----
```

```
-- instructions POSTGRESQL (et NON SQL)
```

```
DROP TABLE IF EXISTS roles;
```

```
CREATE TABLE roles(  
    id BIGSERIAL NOT NULL UNIQUE PRIMARY KEY,  
    role VARCHAR(255) NOT NULL  
);
```

```
DROP TABLE IF EXISTS utilisateurs;
```

```
CREATE TABLE utilisateurs(  
    id BIGSERIAL NOT NULL UNIQUE PRIMARY KEY,  
    id_role BIGINT NOT NULL,  
    nom VARCHAR(255) NOT NULL,  
    CONSTRAINT fk_roles FOREIGN KEY(id_role) REFERENCES roles(id)  
);
```

```
DROP TABLE IF EXISTS dossiers;
```

Exercice Hôpital : solution serveur

```
CREATE TABLE dossiers(  
    id BIGSERIAL NOT NULL UNIQUE PRIMARY KEY,  
    date_entree TIMESTAMP,  
    date_sortie TIMESTAMP,  
    motif VARCHAR(255),  
    id_utilisateur BIGINT NOT NULL,  
    CONSTRAINT fk_utilisateur FOREIGN KEY(id_utilisateur) REFERENCES  
utilisateurs(id)  
);  
  
-- -----  
  
-- fin de fichier
```

Structure

La solution C# est un modèle ASP.NET Core.

L'image ci-contre montre l'organisation des répertoires.

L'architecture suivie est le *design pattern* MVC, sans View puisque cette partie est prise en charge par l'interface utilisateur côté client qui se trouve hors de la solution.

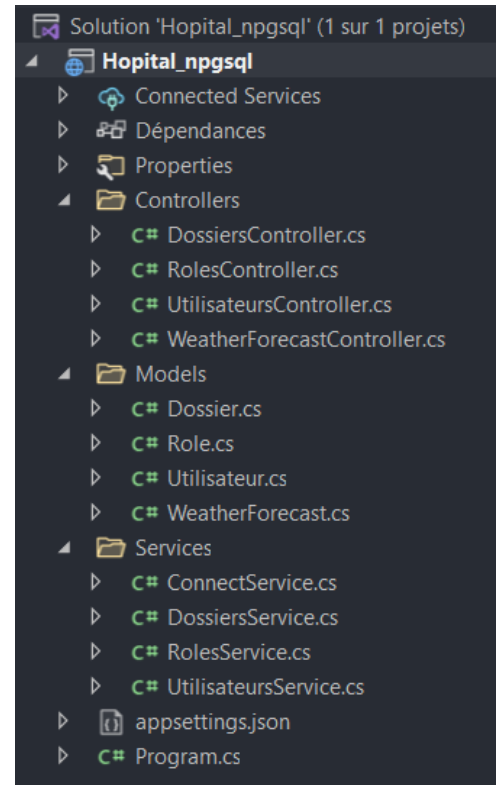
Les éléments qui nous occupent pour la réalisation de l'API sont les suivants :

- Répertoire *Controllers* : les classes des différentes routes, routes qui sont appelées par l'interface utilisateur et qui pointent vers des fonctions, fonctions qui communiquent avec les *Services*.
- Répertoire *Models* : les classes de données, correspondant aux tables de la base de données.
- Répertoire *Services* : les classes de communication avec la base de données.
- *appsettings.json* : fichier de configuration qu'il semble possible d'alimenter avec les données de connexion à la base de données, sans que j'aie pu y parvenir. Recherches Google à poursuivre...
- *Program.cs* : le fichier principal de l'application, fichier qui construit l'API. Ce fichier comprend des instructions CORS spécifiques de l'autorisation *Cross-Origin*, autorisation nécessaire pour le développement local.

Les fichiers d'exemple de Microsoft (*WeatherForecast*) sont conservés à des fins de consultation.

Les noms de fichier suivent la convention de nommage spécifique de MVC : suffixe par catégorie.

La solution requiert l'utilisation du *package* NuGet *Npgsql*. Dans l'*Explorateur de solution*, clic droit sur le projet puis choisir *Gérer les packages NuGet...* pour ouvrir la fenêtre de gestion.



Npgsql par Shay Rojansky, Nikita Kazmin, Brar Piening, Yoh Deadfall, Austin Drenski, Emil Lenngren, Francisco
Npgsql is the open source .NET data provider for PostgreSQL.

6.0.3

Models

Les classes de *Models* correspondent aux tables de la base de données.

Ce sont des classes de données.

Elles ne contiennent que des propriétés en lecture et écriture. En effet, **il semble** qu'ASP.NET Core requiert ce genre de structure car quelques tests ont montré qu'utiliser des champs explicites seuls ou encapsulés dans les propriétés ou encore un constructeur ne produisait aucun résultat.

Il peut être utile d'assigner une valeur par défaut aux propriétés. Par exemple lorsqu'une propriété est de type *string*, il peut être judicieux de lui assigner la valeur *Empty* afin d'éviter de traiter une valeur *null* par la suite.

Sans assignation explicite, la valeur sera déterminée lors de l'instanciation de la classe (valeur par défaut si propriété non traitée ou bien valeur spécifique si besoin).

Enfin, le type de la propriété peut être rendu *nullable*, ceci afin de tester un retour *null* de la base de données. Pour ce faire, il suffit d'ajouter un « ? » après le type.

Pourquoi utiliser des types nullable ? Si on n'utilise pas de type nullable, alors la valeur *null* lève des exceptions. D'autre part, pour les types valeurs, il existe non pas de valeur *null* mais une valeur par défaut selon le type. Par conséquent, il devient complexe de traiter une valeur *null* reçue de la base de donnée :

- soit c'est une exception et on gère donc ce retour comme une erreur alors que ce n'en est peut-être pas une (c'est le cas de colonnes où leur valeur est optionnelle, non *NOT NULL*),
- soit il faut batailler avec des valeurs par défaut pour les types valeur ou prévoir des valeurs qui auraient la charge de signifier *null* (et ce pour tous les types valeur).

Voici la classe *Role*, correspondant à la table *roles* en base de données :

```

1 namespace Hospital_npgsql.Models
2 {
3     13 références
4     public class Role
5     {
6         2 références
7         public int p_id { get; set; }
8         2 références
9         public string? p_role { get; set; }
10    }
11 }

```

Exercice Hôpital : solution serveur

p_id : colonne *id*. Par défaut, sa valeur est 0.

p_role : colonne *role*. Nullable si pas de valeur.

Voici la classe *Utilisateur* correspondant à la table *utilisateurs* :

```
1 namespace Hopital_npgsql.Models
2 {
3     26 références
4     public class Utilisateur
5     {
6         5 références
7         public int p_id { get; set; }
8         4 références
9         public string? p_name { get; set; }
10        4 références
11        public string? p_role { get; set; }
12    }
13 }
14 }
```

p_id : colonne *id*. Par défaut, sa valeur est 0.

p_name : colonne *nom*. Nullable si pas de valeur.

p_role : chaîne de caractère représentant le nom du rôle de la table rôle (obtenu par jointure SQL). Nullable si pas de valeur. Ici, on n'utilise pas les clé étrangères mais l'information qu'elles permettent d'obtenir (les jointures s'effectuent en requête PostgreSQL et non pas dans l'API).

Voici la classe *Dossier* correspondant à la table *dossiers* :

```
1 namespace Hopital_npgsql.Models
2 {
3     19 références
4     public class Dossier
5     {
6         3 références
7         public int p_id { get; set; }
8         3 références
9         public DateTime? p_dateEntree { get; set; }
10        3 références
11        public string p_dateEntreeCulture { get; set; } = string.Empty;
12        3 références
13        public DateTime? p_dateSortie { get; set; }
14        3 références
15        public string p_dateSortieCulture { get; set; } = string.Empty;
16        3 références
17        public string? p_motif { get; set; } = string.Empty;
18        3 références
19        public string? p_utilisateurNom { get; set; }
20        3 références
21        public string? p_utilisateurRole { get; set; }
22    }
23 }
24 }
```

Exercice Hôpital : solution serveur

p_id : colonne *id*. Par défaut, sa valeur est 0.

p_dateEntree, *p_dateSortie* : colonnes *date_entree* et *date_sortie*, de type *DateTime* correspondant à *TIMESTAMP*. Nullable si pas de valeur.

p_motif : colonne *motif*. Nullable si pas de valeur.

p_utilisateurNom : nom de l'utilisateur tel qu'entré dans la table *utilisateurs* (obtenu par jointure).

p_utilisateurRole : nom du rôle de la table *roles* (obtenu par jointure). Nullable si pas de valeur.

p_dateEntreeCulture, *p_dateSortieCulture* : chaînes de caractères SANS CONTREPARTIE en base de données. Ces champs sont renseignés lors de la récupération de données et permettent de retourner une date mise en forme selon la culture de l'ordinateur servant de serveur d'API (nul besoin d'effectuer ce traitement côté client car l'API l'effectue déjà). S'il est nécessaire d'utiliser la valeur telle que retournée par la base de données, voir les propriétés de type *DateTime*.

Controllers

Les classes de *Controllers* sont les points d'entrée dans l'API. Par exemple, s'il s'agit de demander la liste des utilisateurs dans la base de données, alors on appelle la commande afférente fournie par l'API.

Chaque demande à l'API web s'effectue par requête HTTP. Ainsi, telle requête pointe sur telle fonction de l'API. Pour que ceci fonctionne comme attendu, il faut alors utiliser des routes, router la requête vers la bonne fonction.

<https://docs.microsoft.com/fr-fr/aspnet/core/fundamentals/routing?view=aspnetcore-6.0>

Router signifie diriger une requête HTTP vers une fonction de l'API. Cette fonction est exposée sous la forme de point de terminaison dans un chemin d'accès (adresse). Par exemple : monServeur/Chose/MangerChose où MangerChose est le point de terminaison correspondant à une fonction.

<https://docs.microsoft.com/fr-fr/aspnet/core/mvc/controllers/routing?view=aspnetcore-6.0>

ASP.NET Core fournit différentes manières de coder les routes :

- route conventionnelle : dans *Program.cs*, sous forme de liste de chemins qui pointent au *runtime* sur telle méthode de telle classe avec tels paramètres,
- route par attributs : dans les classes de *Controllers*, sous forme d'attributs. Ce sont des informations écrites entre crochets « [...] » et qui précèdent immédiatement la signature d'une fonction, ceci pour définir un comportement supplémentaire à la fonction.

Pour cette solution C#, j'ai codé les routes avec des attributs. Pourquoi ? Pour suivre l'architecture REST ou RESTful.

<https://www.redhat.com/fr/topics/api/what-is-a-rest-api>

Pourquoi REST ? C'est la structure d'une API pour le web. Les opérations de routage s'y effectuent à partir de verbes HTTP : *GET*, *POST*, *PUT*, *DELETE*...

Par exemple, voici la classe *RolesController*, tous corps de méthode fermés :

```

1  using Microsoft.AspNetCore.Mvc;
2
3  using Hopital_npgsql.Models;
4  using Hopital_npgsql.Services;
5
6  namespace Hopital_npgsql.Controllers
7  {
8      [ApiController]
9      [Route("[controller]")]
10     0 références
11     public class RolesController : ControllerBase
12     {
13         [HttpGet("Get all")]
14         0 références
15         public ActionResult<List<Role>> GetAll()...
16
17         [HttpGet("Get role by {id}")]
18         0 références
19         public ActionResult<Role> GetById(int id)...
20
21         [HttpGet("Get id by {name}")]
22         0 références
23         public ActionResult<int> GetId(string name)...
24
25         [HttpPost("Post by {name}")]
26         0 références
27         public ActionResult Post(string name)...
28
29         [HttpPut("Update by {id} {role}")]
30         0 références
31         public IActionResult Update(int id, string role)...
32
33         [HttpDelete("Delete by {id}")]
34         0 références
35         public IActionResult Delete(int id)...
36     }
37 }

```

Cette classe doit utiliser la librairie *Microsoft.AspNetCore.Mvc* pour disposer des attributs correspondant aux verbes HTTP de l'architecture REST.

La classe doit également hériter de *ControllerBase* pour être considérée comme un contrôleur d'API web. A ce sujet, noter également que la classe est précédée d'attributs : *[ApiController]* pour définir que la classe est un contrôleur d'API, *[route("[controller]")]* a pour effet de proposer une route vers la classe sans la sous-chaîne « Controller ».

Chaque attribut de fonction présente le chemin d'accès correspondant. Ici, j'ai utilisé des chemins tirés de la langue anglaise ordinaire, et non pas simplement des identifiants, pour montrer que la définition de la route est libre.

Exemples :

- « *Get all* » pour la route : `https://[adresseDuServeur]/Roles/Get%20all`. Remarquer l'encodage URL de l'espace blanc. Voir plus d'infos sur le sujet ici : https://www.w3schools.com/tags/ref_urlencode.ASP
- « *Post by {name}* » pour la route correspondante où « *name* » est à remplacer par une chaîne représentant un nom de rôle à ajouter à la table.

Noter qu'ajouter un élément entre accolades rend cet élément nécessaire à la requête ; pour effectuer des requêtes avec éléments optionnels, il suffit de ne pas ajouter ces éléments dans l'attribut. Par exemple : dans la classe *DossiersController*, la fonction *Update()* prend un paramètre obligatoire *id* et des paramètres optionnels dont au moins un doit être non *null*, c'est-à-dire fourni.

```
[HttpPut("Update by {id}, one or more medical values")]
0 références
public ActionResult Update(int id, string? dateEntree, string? dateSortie, string? motif)
{
    if (dateEntree == null && dateSortie == null && motif == null) return BadRequest(new { messageRetour
```

Que font les fonctions d'un contrôleur ?

- Elles traitent les valeurs en entrée. C'est ici que l'on effectue les tests de sécurité sur les chaînes entrantes. Ceci pouvant très long à réaliser, il n'est pas demandé pour notre exercice d'effectuer ces tests.
- Une fois les tests d'entrée effectués, elles peuvent éventuellement traiter les valeurs fournies pour les préparer en vue de communiquer avec le serveur.
- Elles appellent les fonctions du répertoire *Services* avec les données fournies.
- Selon les échanges avec la base de données, elles retournent une valeur fournie par les *Services* ou bien envoient une valeur particulière comme par exemple un message de statut.

Exemple : obtenir un dossier par *id*. La fonction du contrôleur appelle une fonction d'une classe de *Services*. Ce service retourne un objet de type *Dossier*, type nullable. Si ce dossier est *null*, alors renvoi de message personnalisé dans une requête *NotFound* ; sinon, renvoi de l'objet (que le service a construit).

```
[HttpGet("Get by id {id}")]
0 références
public ActionResult<Dossier> GetById(int id)
{
    Dossier? d = DossiersService.GetById(id);
    if (d == null) return NotFound(new { messageRetour = "Dossier introuvable." });
    return d;
}
```

Services

Les classes de *Services* font les requêtes à la base de données. Leur structure dépend du type de base de données utilisée et de la librairie utilisée pour communiquer avec la base de données.

Le *package* NuGet utilisé est *Npgsql*. Sa documentation se trouve à cette adresse :

<https://www.npgsql.org/doc/>

Pour les requêtes PostgreSQL, voir ici :

<https://docs.postgresql.fr/14/>

Avant de plonger dans les requêtes, voyons comment l'API se connecte à la base de données. Ceci s'effectue au moyen d'une classe spéciale *ConnectService*. Cette classe présente deux membres :

- un dictionnaire (structure par clé-valeur) contenant les valeurs de connexion à la base de données. Ces valeurs doivent être modifiées selon la configuration de la machine. Ici, ce sont celles de l'ordinateur sur lequel je travaille. Ces informations devraient pouvoir se trouver dans le fichier *appsettings.json*.

```
static public Dictionary<string, string> m_data = new Dictionary<string, string>(){
    { "Host",      "localhost" },
    { "Port",      "5432" },
    { "Username",   "postgres" },
    { "Password",   "simplon59" },
    { "Database",   "Hopital" }
};
```

- une fonction de construction de la chaîne de connexion pour les autres classe de *Services*.

Avec cette classe, chaque service pourra interroger la base de données.

Maintenant, voyons les requêtes. Chaque service est spécifique d'un contrôleur et d'un modèle (MVC). Pour cette raison, on trouve la même architecture de fichiers, distribués en autant de tables de la base de données.

Sans rentrer dans les détails du code C#, deux cas sont à retenir :

- la requête est synchrone : le programme ne peut rien faire tant qu'il n'a pas reçu de données,
- la requête est asynchrone : le programme fait la requête et n'attend aucun retour de la base de données.

L'API réalisée présente différents cas selon les besoins... et aussi par envie de chahuter le bousin. Par exemple, *RolesService* présente une fonction *Delete()* asynchrone (la requête est effectuée et l'application n'attend pas de retour) et *DossiersService* présente une même fonction *Delete()*

Exercice Hôpital : solution serveur

synchrone (la requête est effectuée et l'application attend une valeur « *true* » de la base de données, ce qui va lui permettre d'envoyer un message spécifique dans le contrôleur afférent).

Quid des requêtes ? Le minimum requis en sécurité (bien que non exigé pour l'exercice, rappelons-le à nos tortionnaires de Simplon) est d'effectuer des requêtes préparées. Il s'agit d'éviter d'écrire la requête avec des valeurs « en dur ». Pour cela, on passe par une étiquette dans la requête qui sera remplacée le moment venu par la valeur désirée.

Exemple : *DossiersService.Post()*. La fonction est synchrone (noter le *RETURNING id*). L'étiquette *@p1* prendra la valeur de *userId* au moment de l'envoi de la requête.

```
1 référence
public static int? Post(int userId) // synchrone car réception de données (RETURNING)
{
    int? value = null;

    // Connexion à bdd
    var connString = ConnectService.DataForConnecting();

    // Requête et traitement
    using (var connexion = new NpgsqlConnection(connString))
    {
        connexion.Open();

        using (var cmd = new NpgsqlCommand("INSERT INTO dossiers (id_utilisateur) VALUES(@p1) RETURNING id;", connexion))
        {
            cmd.Parameters.Add(new("p1", userId));
            cmd.Prepare();

            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    value = reader.GetInt32(0);
                }
            }
        }
    }

    return value;
}
```

Puisqu'on dispose de la requête dans le langage de la base de données, il est possible de créer de requêtes à la volée, comme dans *DossiersService.Update()* dont voici un extrait. La fonction prend des paramètres optionnels et qui en tant que nullables... peuvent donc être *null*. Ceci à des fins de test: si la valeur est non *null*, alors ajouter une instruction dans la requête finale.

```
StringBuilder sb = new StringBuilder();
sb.Append("UPDATE dossiers SET "); // espace de fin
if (dateEntree != null)
{
    sb.Append("date_entree = @p1");
    if (dateSortie != null || motif != null) sb.Append(", ");
}
if (dateSortie != null)
{
    sb.Append("date_sortie = @p2");
    if (motif != null) sb.Append(", ");
}
if (motif != null) sb.Append("motif = @p3");
sb.Append(" WHERE id = @p4 RETURNING TRUE;");

Console.WriteLine(sb);
```

Exercice Hôpital : solution serveur

La requête préparée s'effectue de la même manière, par test sur la valeur non *null*.

```
using (var cmd = new NpgsqlCommand(sb.ToString(), connexion))
{
    if (dateEntree != null) cmd.Parameters.Add(new("p1", dateEntree));
    if (dateSortie != null) cmd.Parameters.Add(new("p2", dateSortie));
    if (motif != null) cmd.Parameters.Add(new("p3", motif));
    cmd.Parameters.Add(new("p4", id));
    cmd.Prepare();

    using (var reader = cmd.ExecuteReader())
    {
        while (reader.Read())
        {
            result = reader.GetBoolean(0);
        }
    }
}
```

Compilation, test

Les tests en développement lancent un serveur local (Kestrel et sa console de suivi) ainsi qu'une page de navigateur web présentant l'API dans une interface utilisateur nommée Swagger.

Pour fermer le serveur, sélectionner la fenêtre de console et y entrer CTRL+C.

Pour utiliser l'API, suivre les indications de Swagger.

On peut également tester les adresses directement : il suffit de copier/coller le chemin et de le placer après l'adresse du serveur, dans la barre d'adresse du navigateur. Les résultats s'affichent au format JSON dans la page.

