Exercice Hôpital : solution serveur

Par Alexandre Venet, le 15/02/2022

https://github.com/AlexandreVenet

Ce document accompagne et présente une solution serveur que j'ai conçue et réalisée au cours du mois de février 2022 :

- de type ASP.NET Core Web API (C#) de l'environnement .NET 6 de Microsoft,
- avec une base de données PostgreSQL,
- avec la librairie Npgsql pour communiquer avec la base de données.

Les objectifs de ce document sont les suivants :

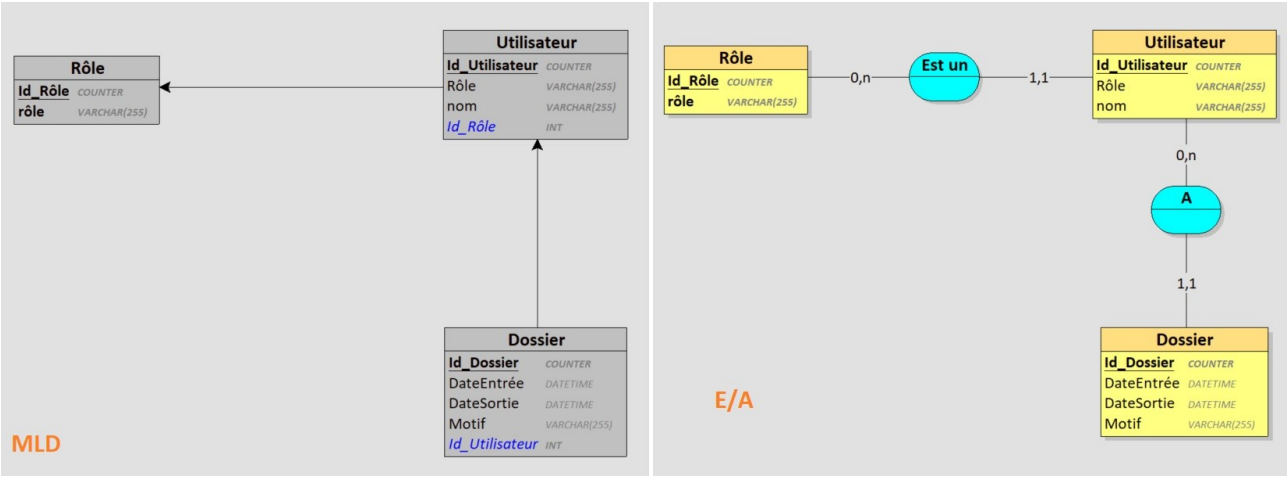
- expliquer le fonctionnement d'une API qu'il est possible de réaliser avec .NET (structure, usage),
- présenter une factorisation possible des fonctions C#,
- garder une trace écrite à date.

La solution présente un certain nombre d'essais et tests, de fonctions factorisées ou non, de requêtes dont les entrées sont vérifiées ou non.

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 avec le logiciel Looping).



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
```

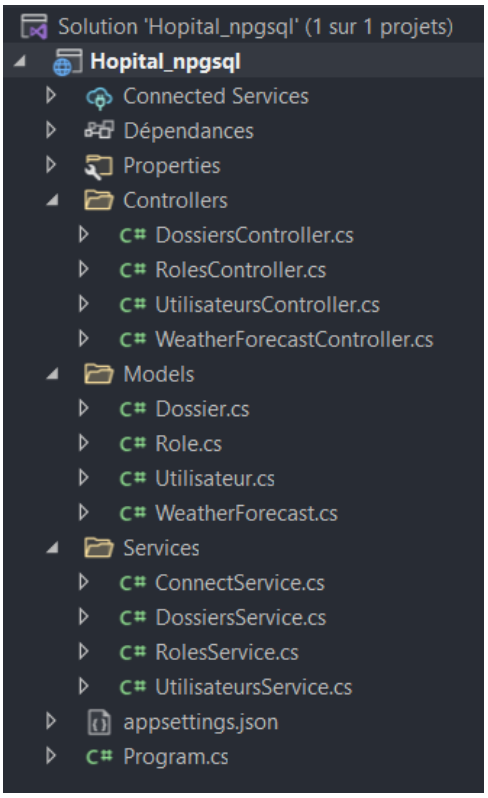
```
    CONNECTION LIMIT = -1;

-- ---------------------------------

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;
```

```
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.

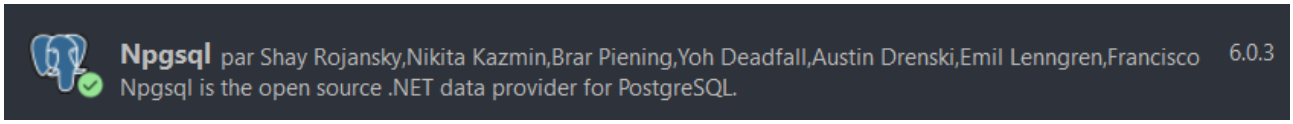
Les éléments qui nous occupent pour la réalisation de l'API sont les suivants :

- 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 de l'application ainsi que de l'accès à la base de données.
- *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. Il contient aussi une instruction permettant de récupérer les données d'accès à la base de données (*appsettings.json*) et de les utiliser dans une classe spécifique (*ConnectService.cs*).

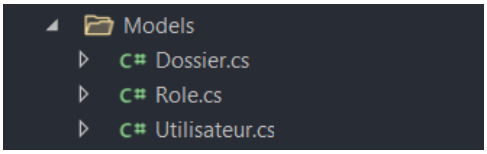
Les fichiers d'exemple de Microsoft (*WeatherForecast*) sont conservés à des fins de consultation.

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.



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 nullables ? 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 Hopital_npgsql.Models
2  {
       13 références
3      public class Role
4      {
           2 références
5          public int p_id { get; set; }
           2 références
6          public string? p_role { get; set; }
7      }
8  }
9
```

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   {
        26 références
3       public class Utilisateur
4       {
            5 références
5           public int p_id { get; set; }
            4 références
6           public string? p_name { get; set; }
            4 références
7           public string? p_role { get; set; }
8       }
9   }
10
```

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 par 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   {
        19 références
3       public class Dossier
4       {
            3 références
5           public int p_id { get; set; }
            3 références
6           public DateTime? p_dateEntree { get; set; }
            3 références
7           public string p_dateEntreeCulture { get; set; } = string.Empty;
            3 références
8           public DateTime? p_dateSortie { get; set; }
            3 références
9           public string p_dateSortieCulture { get; set; } = string.Empty;
            3 références
10          public string? p_motif { get; set; } = string.Empty;
            3 références
11          public string? p_utilisateurNom{ get; set; }
            3 références
12          public string? p_utilisateurRole {get; set; }
13      }
14  }
```

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*. Nullables 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 u*tilisateurs* (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 un 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 :

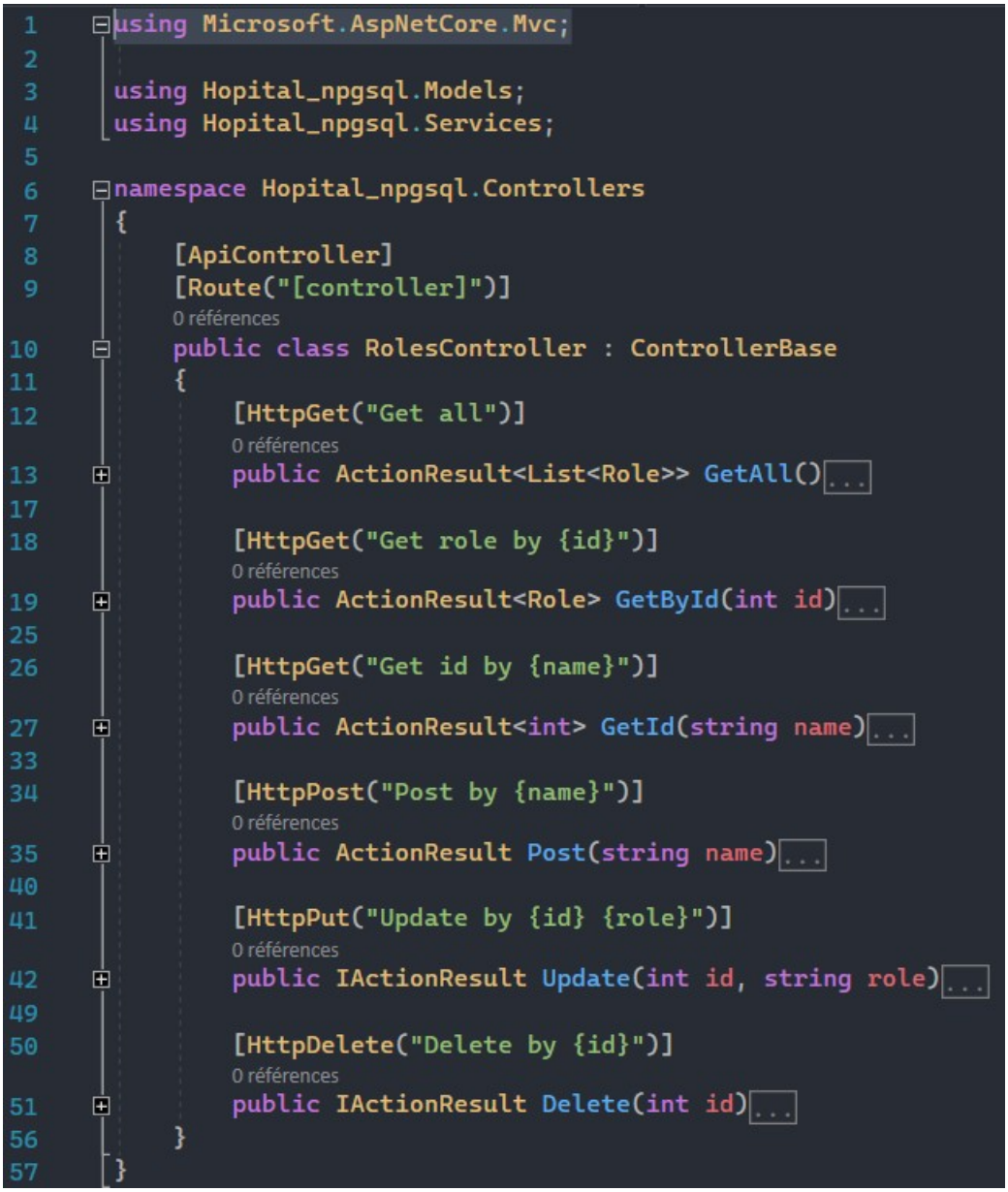
- route conventionnelle : dans *Program.cs*, sous forme de liste de chemins qui pointent au *runtime* sur telle méthode de telle classe avec tels paramètres,
- route par attributs : dans les classes de *Controllers*, sous forme d'attributs. Ce sont des informations écrites entre crochets « [...] » et 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 :



```
using Microsoft.AspNetCore.Mvc;

using Hopital_npgsql.Models;
using Hopital_npgsql.Services;

namespace Hopital_npgsql.Controllers
{
    [ApiController]
    [Route("[controller]")]
    0 références
    public class RolesController : ControllerBase
    {
        [HttpGet("Get all")]
        0 références
        public ActionResult<List<Role>> GetAll()...

        [HttpGet("Get role by {id}")]
        0 références
        public ActionResult<Role> GetById(int id)...

        [HttpGet("Get id by {name}")]
        0 références
        public ActionResult<int> GetId(string name)...

        [HttpPost("Post by {name}")]
        0 références
        public ActionResult Post(string name)...

        [HttpPut("Update by {id} {role}")]
        0 références
        public IActionResult Update(int id, string role)...

        [HttpDelete("Delete by {id}")]
        0 références
        public IActionResult Delete(int id)...
    }
}
```

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 le 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. L'adresse de la requête peut être alors : *https://[adresseDuServeur]/Dossiers/Update%20by%202,%20one%20or%20more%20medical%20values?motif=prout*

```
[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, je ne me suis pas attardé là-dessus.
- Une fois les tests d'entrée effectués, elles peuvent éventuellement traiter les valeurs fournies 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;
}
```

Que sont ces messages de retour ? ASP.NET Core 6 fournit des méthodes prêtes à l'emploi pour renvoyer des codes de réponses et des messages du serveur. Dans l'exemple précédent, *NotFound()* renvoie une requête « *404 Not found* ».

Comment utiliser ces messages de retour pour l'API ? L'approche la plus simple consiste à utiliser, non pas des codes réponse *réseau*, mais une seule réponse « *200 OK* » accompagnée d'un message spécifique du résultat de la base de données. Par exemple : renvoyer « Rien » pour signifier qu'aucune ligne ou entrée de la base de donnée n'a été trouvée.

```
if (r == null) return Content("Rien");
```

On peut également passer par une *enum* cataloguant les codes d'erreurs :

```
if (r == null) return StatusCode(StatusCodes.Status200OK, "Rien");
```

Plus d'infos sur ces codes de retours sur la page de documentation de *ControllerBase* :

https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.controllerbase?view=aspnetcore-6.0

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*. Les fichiers de *Services* vont tous appeler cette classe afin de se connecter à la base de données, avant d'effectuer des requêtes et traitements.

A des fins de tests, j'ai travaillé de deux manières différentes.

La première approche consiste à tout écrire dans la classe de connexion.

- Un dictionnaire (structure par clé-valeur) contient 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.

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

- Une autre fonction s'occupe de construire la chaîne de connexion. Ici, la chaîne n'est pas conservée mais est construite à chaque fois qu'on souhaite se connecter à la base de données (on pourrait stocker cette chaîne mais pour tester ça n'a pas d'importance).
- Usage : dans une classe de *Services*, on appelle la fonction de construction là où il convient.

```
// Requête et traitement sans factorisation
using (var connexion = new NpgsqlConnection(ConnectService.DataForConnecting()))
{
    connexion.Open();
```

Le problème de cette approche est que les données de configuration sont écrites en dur dans une classe applicative. En effet, s'il s'agit par exemple de convertir l'API dans un autre langage, alors il faudra passer du temps à réécrire aussi la fonction de construction de la chaîne de connexion. Au

niveau de la sécurité, il serait plus intéressant de traiter les valeurs de connexion avant de les utiliser et cela sans alourdir la classe applicative. Pour résoudre ces problèmes, on peut avoir une autre approche.

La seconde approche consiste en plusieurs choses.

- Renseigner les valeurs de connexion dans le fichier *appsettings.json* qui sert à configurer l'application. Le terme *ConnectionStrings* est un mot-clé .NET ; il permet de définir un ensemble de valeurs de connexion, pour une ou plusieurs bases de données. C'est ici que l'on pourra effectuer des traitements de sécurité.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "dbHopital": "Host=localhost;Port=5432;Username=postgres;Password=simplon59;Database=Hopital"
    //"myDb2": "Server=myServer;Database=myDb2;Trusted_Connection=True;"
  }
}
```

- Dans la classe *ConnectService*, ajouter un champ d'accueil pour la chaîne.

```
// chaîne de connexion renseignée à partir de Program.cs et appsettings.json
public static string m_connectString = string.Empty;
```

- Dans *Program.cs*, récupérer la chaîne qui nous intéresse à l'aide d'une commande .NET et la conserver dans le champ de la classe *ConnectService*.

```
// renseigner le champ de la classe de connexion avec les infos du json
Hopital_npgsql.Services.ConnectService.m_connectString = builder.Configuration.GetConnectionString("dbHopital");
```

- Usage : dans une classe de *Services*, on appelle le champ là où il convient.

```
// Requête et traitement sans factorisation
using (var connexion = new NpgsqlConnection(ConnectService.m_connectString))
{
    connexion.Open();
```

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.

Deux cas sont à retenir :

- la requête est synchrone : le programme lance la requête et ne peut rien faire tant qu'il n'a pas reçu de données,
- la requête est asynchrone : le programme lance 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()* 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 en sécurité 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. Ici, est utilisé une étiquette nommée (« p1 »).

```
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);
```

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);
        }
    }
}
```

Maintenant, les auteurs de Npgsql conseillent de préférer des requêtes préparées à l'aide d'étiquettes par position. Source : https://www.npgsql.org/doc/basic-usage.html#parameters

De quoi s'agit-il ? Au lieu de passer par une étiquette nommée, la requête préparée est construite selon la position de l'étiquette dans la chaîne de caractère. Cela change le code de construction de la requête préparée.

Par exemple, prenons la méthode *GetByName()* de la classe *RolesService*. « $1 » est une étiquette de position. La valeur à y placer est ajoutée par la commande « *cmd.Parameters.Add(...)* » dont la structure ne présente aucun identifiant nommé.

```
3 références
public static int? GetByName(string name)
{
    // Connexion à bdd
    //var connString = ConnectService.DataForConnecting();

    int? id = null;

    // Requête et traitement sans factorisation
    using (var connexion = new NpgsqlConnection(ConnectService.m_connectString))
    {
        connexion.Open();

        using (var cmd = new NpgsqlCommand("SELECT id FROM roles WHERE role=$1", connexion))
        {
            // autre façon de faire une requête préparée. Différence : pas d'étiquette nommée, seulement l'ordre
            cmd.Parameters.Add(new() { Value = name }); // $1
            cmd.Prepare();

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

    return id;
}
```

S'il y a plusieurs valeurs à passer, il suffit d'ajouter dans la requête autant de « $x », où « x » augmente de 1 à chaque nouvelle position, et d'ajouter autant de « *cmd.Parameters.Add(...)* » que de valeurs selon l'ordre d'apparition dans la chaîne de requête.

Ensuite, dans le cas d'une construction dynamique de requête, il faut prendre en compte l'incrémentation des étiquettes de position et ajouter la valeur à chaque nouvelle entrée nécessaire. Reprenons l'exemple de la classe *DossiersService*, modifions-le en conséquence :

```
// Créer la requête
int labelNumber = 1;
StringBuilder sb = new StringBuilder();
sb.Append("UPDATE dossiers SET "); // espace de fin
if (dateEntree != null)
{
    sb.Append($"date_entree = ${labelNumber}");
    if (dateSortie != null || motif != null) sb.Append(", ");
    labelNumber++;
}
if (dateSortie != null)
{
    sb.Append($"date_sortie = ${labelNumber}");
    if (motif != null) sb.Append(", ");
    labelNumber++;
}
if (motif != null)
{
    sb.Append($"motif = ${labelNumber}");
    labelNumber++;
}
sb.Append($" WHERE id = ${labelNumber} RETURNING TRUE;"); // espace de début

Console.WriteLine(sb);
```

Et pour le passage des valeurs, on testera à nouveau non *null* avec les instructions « *cmd.Parameters.Add(new() { Value=...)* ».

Maintenant, tout ceci semble excellent mais rien n'est factorisé. En effet, chaque fonction va présenter les mêmes instructions de connexion (*using*, *Open()*…). Il est possible de factoriser toutes ces fonctions dans *ConnectService*.

Ici, il faut à nouveau noter que les fonctions factorisées doivent être écrites en synchrone et en asynchrone. Par conséquent, il y a toujours deux types de fonctions à écrire pour chaque cas.

J'ai donc créé deux jeux de fonctions :

- pour réaliser des requêtes préparées par étiquette de position, l'une attendant un retour (synchrone), l'autre non (asynchrone),
- pour réaliser des requêtes préparées par étiquettes nommées, l'une attendant un retour (synchrone), l'autre non (asynchrone).

Conformément à la documentation de Npgsql, on ne retiendra que le jeu de fonctions par étiquette de position.

Donnons un peu de détails sur le fonctionnement de cette factorisation, en particulier les requêtes par étiquette de position.

La fonction synchrone :

- reçoit la requête (type *string*),
- reçoit un tableau de valeurs, de type *Object* pour pouvoir y mettre n'importe quoi en entrée. Ceci sert à fabriquer la requête préparée. On pourrait préférer un tableau de *NpgsqlParameters* car cela permettrait d'éviter des conversions de type… mais ceci prendrait du temps supplémentaire à coder des instructions identiques en entrée (cette solution serait toutefois à préférer d'un point de vue des performances). Ce tableau est facultatif (si pas de valeur à passer, omettre cet argument).
- fabrique la requête préparée,
- étant synchrone, attend une valeur de retour,
- une fois la valeur de retour fournie, lance une fonction *callback* (effectuée par une *Action*) de traitement.

Exemple d'appel pour une requête simple :

```
public static Role GetById(int id)
{
    // Connexion à bdd
    //var connString = ConnectService.DataForConnecting();

    Role r = new Role();

    // V.2 avec fonction factorisée de la Helper Class : synchrone, étiquette par ordre
    ConnectService.RequestSync("SELECT id, role FROM roles WHERE id=$1", (reader) =>
    {
        //Console.WriteLine(reader.GetHashCode()); // reader prend les valeurs du rd de la fonction appelée
        r.p_id = reader.GetInt32(0);
        r.p_role = reader.GetString(1);
    }, new Object[]{ id });

    return r;
}
```

La fonction asynchrone :

- reçoit la requête (type *string*),
- reçoit son tableau de valeurs (s'il y a lieu de l'utiliser car paramètre optionnel),
- fabrique la requête préparée,
- lance la requête.

Exemple d'appel pour une requête simple :

```
public static void Post(string name) // n'est pas async avec la factorisation du traitement
{
    if (name == null || name == string.Empty) return; // le champ de la table ne peut pas être null

    // V.2 avec fonction factorisée de la Helper Class : Asynchrone, étiquette par ordre
    ConnectService.RequestAsync("INSERT INTO roles (role) VALUES ($1)", new Object[] {name});
}
```

Dans les deux cas, le tableau de valeurs est optionnel. Voici l'exemple d'une requête préparée avec retour de données (synchrone) et qui n'a aucune valeur en entrée :

```
ConnectService.RequestSync("SELECT utilisateurs.id, utilisateurs.nom, roles.r
{
    Utilisateur u = new Utilisateur()
    {
        p_id = reader.GetInt32(0),
        p_name = reader.GetString(1),
        p_role = reader.GetString(2),
    };
    utilisateursList.Add(u);
}); // aucune valeur à passer, donc pas de tableau en paramètre

return utilisateursList;
```

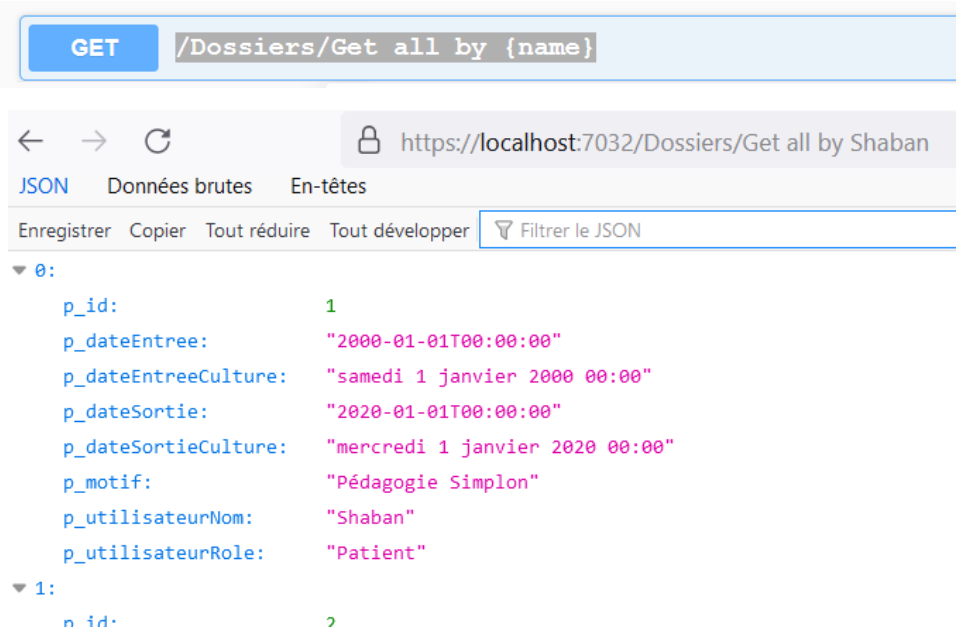
Compilation, test

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 requêtes GET 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. Swagger affiche également les adresses HTTP des requêtes préparées dans leur intégralité.



Noter que côté *front-end*, le nom des propriétés reçues est celui des classes *Models*.

J'ai aussi réalisé une interface utilisateur personnelle. Le code Javascript *vanilla* n'y est pas factorisé. L'objectif était de tester plusieurs choses :

- gestion avec *Fetch()*,
- gestion avec *XMLHttpRequest()*,
- construction dynamique avec *FormData()*,
- exploration des codes et messages retournés par le serveur.