TP3 Partie 2 – Amélioration du code du WS REST

Le scaffolding que nous avons utilisé dans le TP3 Partie 1 pour créer le contrôleur du compte manipule directement la base de données en utilisant la classe de contexte.

```
public async Task<ActionResult<Compte>> GetCompteById(int id)
{
        var compte = await _context.Compte.Include(c => c.FavorisCompte).SingleOrDefaultAsync(m
=> m.CompteId == id);

        if (compte == null)
        {
                return NotFound();
        }

        return compte;
}
```

Bien que pratique le code généré par le scaffolding impose un couplage fort entre le contrôleur et la base de données. Ainsi, si nous souhaitons utiliser un autre ORM, comme Dapper, par exemple, il sera nécessaire de modifier le code des contrôleurs.
Une bonne pratique est d'utiliser une couche intermédiaire qui permettra d'assurer un couplage léger entre les contrôleurs et les données, ce qui nous permettra de modifier la couche des données sans modifier celle des contrôleurs.
Il existe (au moins) 2 solutions pour créer cette couche intermédiaire :

1. Appliquer le design pattern DAL (Data Access Layer) spécifié par Microsoft. Exemple de code ici : https://nathanaelmarchand.developpez.com/tutoriels/dotnet/architecture-couches-decouplage-et-injection-dependances-avec-unity/
   Remarque : comme indiqué dans ce tutorial, ASP.Net Web API (.NET framework) nécessitait l'installation du package NuGet Unity pour gérer l'injection de dépendance. ASP.Net Core gérant déjà nativement le DI, l'installation d'Unity n'est plus nécessaire (même si on peut quand même toujours l'utiliser).
2. Appliquer le design pattern Repository (Dépôt). Ce pattern a été spécifié par Martin Fowler dans son (excellent) ouvrage « Patterns of Entreprise Application Architecture » : https://martinfowler.com/books/eaa.html. Principes de base de ce pattern : https://martinfowler.com/eaaCatalog/repository.html

Nous appliquerons le pattern Repository.

1. *Mise en place du pattern Repository*

Les principes et la mise en place de ce pattern sont décrits ici :

- https://docs.microsoft.com/fr-fr/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
- https://docs.microsoft.com/fr-fr/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implemenation-entity-framework-core

Nous allons l'appliquer et un peu le simplifier car nous n'aurons qu'un seul dépôt et donc qu'une seule interface *Repository*.

1.1. Interface

Dans le dossier `Models`, ajouter un dossier nommé `Repository`. Y créer une nouvelle interface nommée `IDataRepository`.

Code de l'interface :

```
public interface IDataRepository<TEntity>
{
    ActionResult<IEnumerable<TEntity>> GetAll();
    ActionResult<TEntity> GetById(int id);
    ActionResult<TEntity> GetByString(string str);
    void Add(TEntity entity);
    void Update(TEntity entityToUpdate, TEntity entity);
    void Delete(TEntity entity);
```

```
}
```

Nous injecterons (injection de dépendance) plus tard cette interface dans notre contrôleur d'API. L'API communiquera avec le contexte de données (et donc la base) à l'aide de cette interface.

1.2. Classe concrète

Ensuite, créer une classe concrète qui implémente l'interface `IDataRepository`. Ajouter un nouveau dossier sous `Models` nommé `DataManager`. Créer ensuite une nouvelle classe `CompteManager` :

```
public class CompteManager : IDataRepository<Compte>
{
    readonly FilmRatingsDBContext _filmRatingsDbContext;

    public CompteManager(FilmRatingsDBContext context)
    {
        _filmRatingsDbContext = context;
    }

    public ActionResult<IEnumerable<Compte>> GetAll()
    {
        return _filmRatingsDbContext.Compte.ToList();
    }

    public ActionResult<Compte> GetById(int id)
    {
        return _filmRatingsDbContext.Compte
              .FirstOrDefault(e => e.CompteId == id);
    }

    public ActionResult<Compte> GetByString(string mail)
    {
        return _filmRatingsDbContext.Compte
              .FirstOrDefault(e => e.Mel.ToUpper() == mail.ToUpper());
    }

    public void Add(Compte entity)
    {
        _filmRatingsDbContext.Compte.Add(entity);
        _filmRatingsDbContext.SaveChanges();
    }

    public void Update(Compte compte, Compte entity)
    {
        _filmRatingsDbContext.Entry(compte).State = EntityState.Modified;
        compte.CompteId = entity.CompteId;
        compte.Nom = entity.Nom;
        compte.Prenom = entity.Prenom;
        compte.Mel = entity.Mel;
        compte.Rue = entity.Rue;
        compte.CodePostal = entity.CodePostal;
        compte.Ville = entity.Ville;
        compte.Pays = entity.Pays;
        compte.Latitude = entity.Latitude;
        compte.Longitude = entity.Longitude;
        compte.Pwd = entity.Pwd;
        compte.TelPortable = entity.TelPortable;
        compte.FavorisCompte = entity.FavorisCompte;
        _filmRatingsDbContext.SaveChanges();
    }

    public void Delete(Compte compte)
    {
        _filmRatingsDbContext.Compte.Remove(compte);
        _filmRatingsDbContext.SaveChanges();
    }
}
```

La classe `CompteManager` gère toutes les opérations de base de données liées au *Compte*. Le but de cette classe est de séparer la logique des opérations de données réelles du contrôleur de l'API.

Cette classe utilise les méthodes suivantes pour prendre en charge les opérations CRUD :

- `GetAll()` - Obtient tous les comptes de la base de données.
- `GetById()` - Obtient un compte spécifique de la base de données en transmettant un ID.
- `GetByString()` - Obtient un compte spécifique de la base de données en transmettant un string (ici, un email).
- `Add()` - Crée un nouveau compte dans la base de données.
- `Update()` - Met à jour un compte spécifique dans la base de données.
- `Delete()` - Supprime un compte spécifique de la base de données en fonction de l'ID.

Maintenant que notre Data Manager est configuré, il reste à modifier le contrôleur API et les points de terminaison pour la gestion des opérations CRUD.

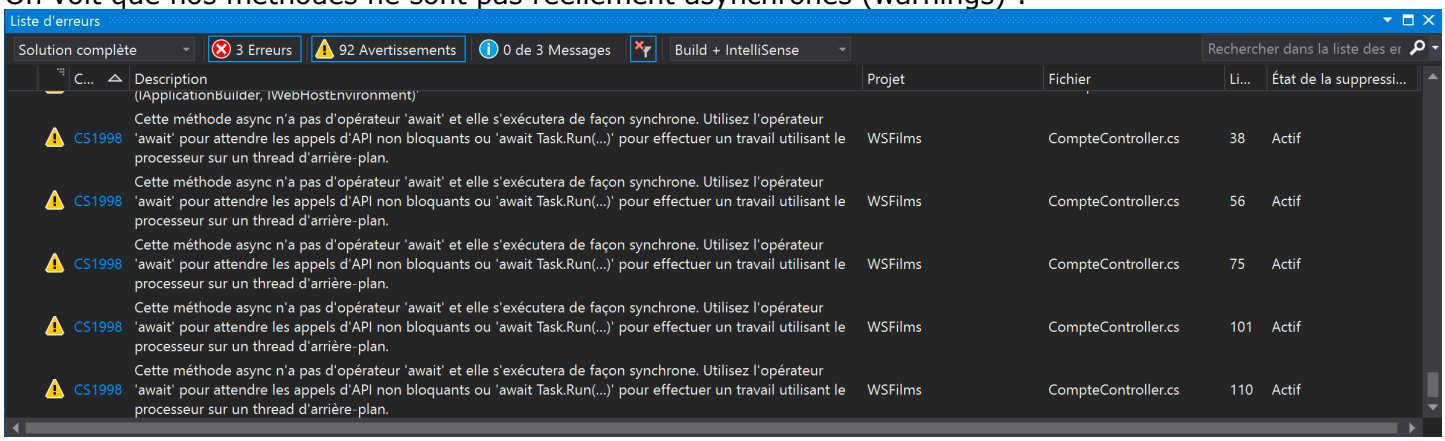
1.3. Modification du contrôleur

Pour le moment, nous allons laisser les `async`, mais supprimer les `await` dans les actions. Nous n'aurons donc pas vraiment de méthodes asynchrones.
Nous avons également décommenté l'action `Delete`, afin de mettre à jour son code (on pourra la commenter à nouveau ultérieurement).
On n'instancie plus un objet `FilmRatingsDBContext` puisque l'on passe maintenant par la classe concrète de Repository. Ne modifier que les parties suivantes en gras :

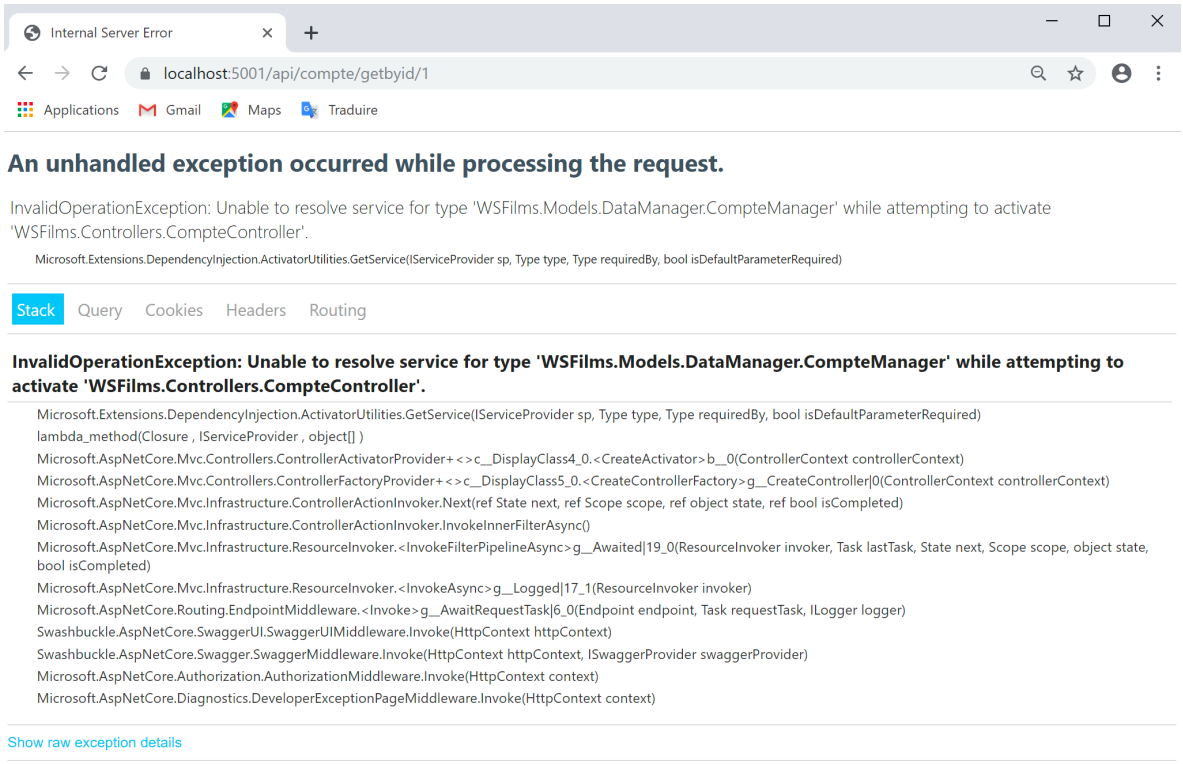
```
[Route("api/[controller]")]
[ApiController]
public class CompteController : ControllerBase
{
    readonly CompteManager _compteManager;

    public CompteController(CompteManager compteManager)
    {
        _compteManager = compteManager;
    }

    …
    public async Task<ActionResult<IEnumerable<Compte>>> GetCompte()
    {
        return _compteManager.GetAll();
    }

    …
    public async Task<ActionResult<Compte>> GetCompteById(int id)
    {
        var compte = _compteManager.GetById(id);

        if (compte == null)
        {
            return NotFound();
        }

        return compte;
    }

    …
    public async Task<ActionResult<Compte>> GetCompteByEmail(string email)
    {
        var compte = _compteManager.GetByString(email);

        if (compte == null)
        {
            return NotFound();
        }
```

```
            return compte;
        }

        …
        public async Task<IActionResult> PutCompte(int id, Compte compte)
        {
            if (id != compte.CompteId)
            {
                return BadRequest();
            }

            var compteToUpdate = _compteManager.GetById(id);

            if (compteToUpdate == null)
            {
                return NotFound();
            }

            _compteManager.Update(compteToUpdate.Value, compte);

            return NoContent();
        }

        …
        public async Task<ActionResult<Compte>> PostCompte(Compte compte)
        {
            _compteManager.Add(compte);

            return CreatedAtAction(
                  "GetCompte",
                  new { Id = compte.CompteId },
                  compte);
        }

        …
        public async Task<ActionResult<Compte>> DeleteCompte(int id)
        {
            var compte = _compteManager.GetById(id);
            if (compte == null)
            {
                return NotFound();
            }

            _compteManager.Delete(compte.Value);
            return compte;
        }
    }
```

On voit que nos méthodes ne sont pas réellement asynchrones (warnings) :



Exécuter le WS.
Vous obtiendrez l'erreur suivante. Nous y reviendrons.



1.4. *Injection de dépendance*

Dans notre contrôleur, nous avons utilisé un objet `CompteManager`.
Cependant, le code précédent n'est pas très propre car nous avons lié notre contrôleur à une implémentation spécifique de `IDataRepository`.

Pour remédier à cela, nous allons utiliser l'interface. Le code du constructeur du contrôleur devient :

```
private readonly IDataRepository<Compte> _dataRepository;

public CompteController(IDataRepository<Compte> dataRepository)
{
    _dataRepository = dataRepository;
}
```

Il faut également remplacer partout `_compteManager` par `_dataRepository`.

Tester. Vous obtenez toujours l'erreur d'injection de dépendance.
Aucune interface `IDataRepository` n'est en effet envoyée au constructeur paramétré du contrôleur. C'est le mécanisme d'injection de dépendance (*dependency injection*) qui va en avoir la charge.

L'injection de dépendance va nous permettre de travailler uniquement avec des interfaces et nous allons laisser l'outil d'injection de dépendance, dans notre cas celui fourni dans ASP.NET Core, instancier la bonne implémentation.

L'injection de dépendances est un design pattern incontournable lorsque l'on souhaite développer une application à la fois volumineuse et modulaire. Avec ce design pattern, les composants n'ont pas besoin de connaitre la manière dont sont créées leurs dépendances. Ce pattern est utilisé dans beaucoup de frameworks de développement récents tels qu'Angular (versions >=2) avec le décorateur `@Injectable()` (https://angular.io/api/core/Injectable), symfony, Android (https://www.raywenderlich.com/146804/dependency-injection-dagger-2), etc.

La configuration du repository à l'aide de l'injection de dépendance se fait dans la méthode `ConfigureServices` du fichier `Startup.cs` :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
```

```
            services.AddDbContext<FilmRatingsDBContext>(options =>
options.UseNpgsql(Configuration.GetConnectionString("FilmRatingsContext")));

            services.AddScoped<IDataRepository<Compte>, CompteManager>();

            …
        }
```

Ici, nous avons lié l'interface à sa classe concrète.
Méthode `services.AddScoped` :
https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addscoped
https://docs.microsoft.com/fr-fr/aspnet/core/fundamentals/dependency-injection
On peut aussi utiliser `services.AddTransient` ou `services.AddSingleton` pour lier une interface à sa classe concrète, mais EF requiert `AddScoped`. Différence entre les 3 mécanismes :
https://www.developpez.com/actu/154398/Apprendre-l-injection-de-dependances-avec-ASP-NET-Core-un-billet-d-Hinault-Romaric/
https://stackoverflow.com/questions/38138100/what-is-the-difference-between-services-addtransient-service-addscoped-and-serv

Exécuter le WS.

2. ***Tests***

Les tests ne sont plus fonctionnels.
Ajouter le code suivant :

```
        private CompteController _controller;
        private FilmRatingsDBContext _context;
        private IDataRepository<Compte> _dataRepository;

        public UnitTestCompte()
        {
            var builder = new DbContextOptionsBuilder...
            _context = new FilmRatingsDBContext(builder.Options);

            _dataRepository = new CompteManager(_context);
            _controller = new CompteController(_dataRepository);
        }
```

Les appels au contrôleur deviennent : `CompteController _controller = new CompteController(_dataRepository);`

On remarque que pour les tests, on utilise toujours le contexte qui permet de récupérer les données provenant directement de la base et on compare ces données avec les données issues de l'appel aux actions du WS. Ainsi, on teste à la fois l'API et le Repository. Par contre, on reste dépendant de la couche de données. On pourrait modifier les tests pour comparer les données de l'API avec celles du repository.

3. ***Asynchronisme***

Il est nécessaire de remettre en place l'asynchronisme.
Exemple pour le Get (email) :

- Modifier l'interface `IDateRepository : ActionResult<TEntity> GetByString(string str);` => `Task<ActionResult<TEntity>> GetByString(string str);`
- `CompteManager` :
  - Modifier la méthode `GetByString` :

```
public async Task<ActionResult<Compte>> GetByString(string mail)
{
      return await _filmRatingsDbContext.Compte
                  .FirstOrDefaultAsync(e => e.Mel.ToUpper() ==
mail.ToUpper());
}
```

  - Modifier la méthode `GetCompteByEmail` du contrôleur :

```
public async Task<ActionResult<Compte>> GetCompteByEmail(string email)
{
```

```
                    var compte = await _dataRepository.GetByString(email);

                    if (compte == null)
                    {
                        return NotFound();
                    }

                    return compte;
            }
```

Vous devriez avoir un avertissement lié à l'asynchronisme de moins maintenant…

Pour les méthodes `Add`, `Update` et `Delete`, il faut remplacer `void` par **`Task`**. Exemple :

```
public async Task Add(Compte entity)
  {
      _filmRatingsDbContext.Compte.Add(entity);
      await _filmRatingsDbContext.SaveChangesAsync();
  }
```

Interface `IDataRepository : Task Add(TEntity entity);`

4. ***Travail à faire (pour les plus rapides)***

Coder le contrôleur des films, ainsi que son Data Manager.
Vous utiliserez toujours l'interface `IDataRepository` qui devrait également convenir pour les films. Vous ajouterez une action permettant de rechercher un film en fonction de son titre exact (la casse devra être prise en compte).