Laboratoire: Les services et jquery

# Objectif(s)

* Services :
  + Service générique
  + Service spécifique
* Javascript/JQuery

# Créer la branche de la fonctionnalité

Clonage du repository

1. Dans Gitkraken, clonez le repository
   * Soit via URL
   * Soit via GitHub.com
2. Ouvrez le repository (OK)

Création d’une branche pour la fonctionnalité

1. Créez une branche appelée **FCT \_Services**
2. Assurez-vous d'être dans la branche que vous avez créée dans LOCAL

Dans ce laboratoire, vous aurez a décider vous-mêmes quand faire les *Comment, commit, push.*

# Création de services

Les contrôleurs font maintenant beaucoup trop de chose. Ils devraient n’avoir qu’une seule responsabilité : la redirection des flux! Sortons donc la logique dans les Services.

D’abord un service générique de base pour le CRUD courant et des interfaces afin de faire le travail dans les règles de SOLID.

## Créer un service générique

1. Créez un dossier **Services** au niveau du projet
2. Créez une Interface IServiceBaseAsync où T est une classe, un modèle.

public interface IServiceBaseAsync<T> where T : class

{

Task<T> CreateAsync(T entity);

Task DeleteAsync(int id);

Task<IReadOnlyList<T>> GetAllAsync();

Task<T> GetByIdAsync(int id);

Task EditAsync(T entity);

}

1. Créez la classe ServiceBaseAsync qui implémente l’interface IServiceBaseAsync où T est une classe, un modèle.
2. Injectez le DbContext

public class ServiceBaseAsync<T> : IServiceBaseAsync<T> where T : class

{

protected readonly ZombiePartyDbContext \_dbContext;

public ServiceBaseAsync(ZombiePartyDbContext dbContext)

{

\_dbContext = dbContext;

}

1. Implémentez l’interface (toutes les méthodes de l’interface vont être créées avec Throw NotImplementedException

Une image contenant texte, capture d’écran, logiciel, ordinateur

Description générée automatiquement

Une image contenant texte, capture d’écran, affichage, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. Complétez chacune des méthodes en remplaçant l’exception

### Create

public async Task<T> CreateAsync(T entity)

{

await \_dbContext.Set<T>().AddAsync(entity);

await \_dbContext.SaveChangesAsync();

return entity;

}

### Delete

public async Task DeleteAsync(int id)

{

var entity = await this.GetByIdAsync(id);

\_dbContext.Set<T>().Remove(entity);

await \_dbContext.SaveChangesAsync();

}

### Edit

public async Task EditAsync(T entity)

{

if (\_dbContext.Entry(entity).State == EntityState.Detached) \_dbContext.Update<T>(entity);

else \_dbContext.Entry(entity).State = EntityState.Modified;

await \_dbContext.SaveChangesAsync();

}

### GetAll

public async Task<IReadOnlyList<T>> GetAllAsync()

{

return await \_dbContext.Set<T>().ToListAsync();

}

### Getbyid

public async Task<T> GetByIdAsync(int id)

{

return await \_dbContext.Set<T>().FindAsync(id);

}

1. Injectez le service dans **DI** dans **Program.cs,** assurez-vous de le mettre AVANT builder.Build()

builder.Services.AddDbContext<ZombiePartyDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")).UseLazyLoadingProxies());

builder.Services.AddScoped(typeof(IServiceBaseAsync<>), typeof(ServiceBaseAsync<>));

var app = builder.Build();

## Créer le service ZombieType

1. Dans le dossier **Services**
2. Créez une Interface **IzombieTypeService** qui Implémente l’interface **IServiceBase** où T est la classe **ZombieType**.

NOTE: Pour l’instant, le service n’utilise que les méthodes de ServiceBase, elle est donc vide, nous en ajouterons dans quelques étapes.

public interface IZombieTypeService :IServiceBaseAsync<ZombieType>

{

}

1. Créez la classe **ZombieTypeService** qui hérite de **ServiceBaseAsync** où **T** est la classe **ZombieType** et qui implémente l’interface **IZombieTypeService**.
2. Créez le constructeur (ctor tab-tab)

public class ZombieTypeService : ServiceBaseAsync<ZombieType>, IZombieTypeService

{

public ZombieTypeService(ZombiePartyDbContext dbContext) : base(dbContext) { }

1. L’interface n’ayant pas encore de méthodes spécifiques, il n’y a rien à implémenter pour l’instant
2. Injectez le service en Scoped dans IO (program.cs) après **ServiceBaseAsync**

builder.Services.AddScoped<IZombieTypeService, ZombieTypeService>();

## Injecter le service dans le controleur ZombieType

### Injection du service

1. Ouvrez le Contrôleur **ZombieType** mettez temporairement toutes les actions en commentaires, sauf Index. Vous pourrez ainsi modifier et tester action par action sans erreur de compilation.
2. Remplacez l’injection du DbContext par celui du **ZombieTypeService**.

private IZombieTypeService \_serviceZT { get; set; }

public ZombieTypeController(IZombieTypeService serviceZT)

{

\_serviceZT = serviceZT;

}

### Vue Index

1. L’action utilise GetAllAsync du DbSet. Consultez le code de **GetAllAsync** du **ServiceBaseAsync** afin de vous assurez qu’il retourne le même résultat
2. Modifiez le code de l’action Index afin qu’il utilise la méthode GetAllAsync du service

**Version 1 :** On doit « caster » le type

public async Task<IActionResult> Index()

{

List<ZombieType> zombieTypesList = (List<ZombieType>)await \_serviceZT.GetAllAsync();

return View(zombieTypesList);

**Version 2:** Plus simple

public async Task<IActionResult> Index()

{

// Version 2

return View(await \_serviceZT.GetAllAsync());

}

1. Testez la vue **ZombieType/Index**; elle devrait fonctionner comme avant

### Vue Create

1. Décommentez Create GET, aucune modification requise
2. Testez la vue Create GET
3. Décommentez Create POST
4. L’action utilise **AddAsync** et **SaveChangeAsync** du DbSet. Consultez le code de **CreateAsync** du **ServiceBaseAsync** afin de vous assurez qu’il retourne le même résultat : Notez que le SaveChangesAsync est inclus dans **CreateAsync**
5. Modifiez le code de l’action Create Post afin qu’il utilise la méthode **CreateAsync** du service. N’oubliez pas d’enlever SaveChangesAsync
6. Testez la vue Create POST

### Vue Edit

1. Décommentez Edit GET
2. Modifiez le code de l’action Edit GET afin qu’il utilise le service. Quelle méthode serait la plus appropiée?
3. Ajoutez des points d’arrêt et testez la vue Edit GET
4. Décommentez Create POST
5. Modifiez le code de l’action Edit POST afin qu’il utilise le service. Quelle méthode serait la plus appropiée?
6. Ajoutez des points d’arrêt et testez la vue Create POST

NOTE : Si vous n’avez pas encore fait de *Comment, commit, push* il serait temps d’y penser!

### Vue Delete

Pour cette action, nous souhaitons un autre comportement que **ServiceBase**/**Delete** : Un ZombieType pour lequel il y a un ou des Zombies associés ne peut pas être supprimé et le ZombieType ne devra plus être disponible pour être associé à un Zombie (Create ou Edit).

Il y a plusieurs façons de mettre en place cette règle d’affaires, nous vous en proposons une ici. Nous allons procéder par étapes.

Un champ doit être ajouté à ZombieType pour indiquer la non disponibiltié de ce type. Ensuite, vous devez *override* la méthode héritée de **ServiceBase** pour **ZombieTypeService**.

1. Ajoutez une propriété *bool* **IsDisponible** dans le modèle **ZombieType.** La valeur par défaut d’un *bool* est **false**, elle doit être à **true** afin d’indiquer que le ZombieType est disponible à sa création.
2. Que devez-vous faire afin d’utiliser cette propriété en BD? Oui! Add-migration, update-database
3. Décommentez Delete GET
4. Modifiez le code de l’action Delete GET afin qu’il utilise le service pour retrouver le bon **ZombieType**. Quelle méthode serait la plus appropiée?
5. Ajoutez des points d’arrêt et testez la vue Delete GET
6. Décommentez Delete POST
7. Modifiez le code de l’action Delete POST afin qu’il utilise le service. Quelle méthode serait la plus appropiée? Le message devra également être modifié.

public async Task<IActionResult> DeletePost(int id)

{

ZombieType? zombieType = await \_serviceZT.GetByIdAsync(id);

if (zombieType == null)

{

return NotFound();

}

await \_serviceZT.DeleteAsync(id);

TempData[AppConstants.Success] = $"ZombieType {zombieType.TypeName} has been removed";

1. Ajoutez une méthode **HasAssociatedZombies** dans le service et l’interface qui permet de vérifier si le ZombieType a un ou des Zombies associés et retourne un bool.

Essayez par vous-mêmes, si nécessaire vous pouvez vous inspirer du code ci-dessous.

public bool HasAssociatedZombies(int id)

{

var associatedZombies = \_dbContext.Zombies.Where(x => x.ZombieTypeId == id).Any();

return associatedZombies;

}

1. Ajoutez la méthode **DeleteAsync** dans ZombieTypeService pour ajouter le comportement modifié qui utilise HasZombiesAssicated
2. Ajoutez un ZombieType
3. Ajoutez des points d’arrêt dans le service et le contrôleur et testez l’action Delete POST en supprimant le ZombieType que vous venez de créer et qui n’a pas de zombies associés.

Le code se déroule-t-il comme prévu? Oui, Yé! Non… inspirez-vous du code ci-dessous..

public new async Task DeleteAsync(int id)

{

var zombieType = await this.GetByIdAsync(id);

if (HasAssociatedZombies(id))

{

zombieType.IsDisponible = false;

await this.EditAsync(zombieType);

}

else

{

await base.DeleteAsync(id);

}

}

Le ZombieType est-il encore visible dans l’index? Il devrait être disparut.

1. Testez l’action Delete avec un ZombieType qui contient des Zombies.

Le code se déroule-t-il comme prévu? Le ZombieType est-il encore visible dans l’index? Il devrait puisqu’il n’est pas réellement supprimé.

1. Ajoutez du code à la vue **ZombieType/Index** permettant de changer la classe du TypeName pour les ZombieType non disponible (btn-info à btn-secondary), ajoutez le texte *Indisponible* à côté du TypeName

Une image contenant texte, capture d’écran, conception

Description générée automatiquement

### Vue Details

1. Décommentez l’action **ZombieType/Details**
2. Pour modifier le code de l’action Details afin qu’il utilise les services, vous devez ajoutez le service pour Zombie qui implémente le service de base. N’oubliez pas l’interface.
3. Ajoutez dans le service une méthode **GetAllByZombieTypeAsync** qui retourne la liste des zombies pour le ZombieTypeId passé en paramètre
4. Injectez le ZombieService dans le contrôleur ZombieType
5. Modifiez le code de l’action Details afin qu’il utilise les services.
6. Testez la méthode **ZombieType/Details**

**NOTE** : Si le message suivant s’affiche *: InvalidOperationException: Unable to resolve service for type 'ZombieParty.Services.IZombieService'*

**Oups**! Auriez-vous oublié d’injecter le service dans le IO (Program.cs)?

1. NOTE : Si vous n’avez pas encore fait de *Comment, commit, push* il serait temps d’y penser!

### Vue edit

1. La propriété IsDisponible doit pouvoir être modifiée lorsqu’un ZombieType est modifié donc ajoutez le *input* dans la vue Edit.

## Liste ZombieType dans Zombie

Pour la création et la modification des Zombies, la liste des ZombieType ne doit présenter que les ZombieTypes disponibles. Il y a plusieurs façons de faire.

### vue Zombie/Index

1. Injectez les services dans le contrôleur en PLUS du DbContext (vous allez le remplacer graduellement et enlever le DbCOntext).
2. Mettez les actions en commentaire, sauf le constructeur et l’Index.
3. Ajoutez une méthode **GetAllIndexAsync** qui retourne la liste des zombies triés par *Name* avec le *TypeName*

public async Task<IReadOnlyList<Zombie>> GetAllIndexAsync()

{

return await \_dbContext.Set<Zombie>().OrderBy(z => z.Name).Include(z => z.ZombieType).ToListAsync();

}

1. Modifiez l’action **Zombie/Index** afin d’utiliser le service **ZombieService**

public async Task<List<Zombie>> GetAllByZombieTypeAsync(int zombieTypeId)

{

var zombies = await \_dbContext.Zombies.Where(x => x.ZombieTypeId == zombieTypeId).ToListAsync();

return zombies;

}

1. Testez l’action Zombie/Index

### vue Zombie/Create

1. Vous allez donc ajouter une méthode **ListZombieTypeDisponible** dans le **ZombieTypeService** pour remplir la SelectListItem avec le TypeName et le Id des types disponibles seulement.
2. Essayez par vous-même, sinon inspirez-vous du code suivant.

public IEnumerable<SelectListItem> ListZombieTypeDisponible()

{

var zombieTypeDisponibleList = \_dbContext.ZombieTypes.Where(zt => zt.IsDisponible==true).Select(t => new SelectListItem

{

Text = t.TypeName,

Value = t.Id.ToString()

}).OrderBy(t => t.Text);

return ((IEnumerable<SelectListItem>)zombieTypeDisponibleList);

}

1. Modifiez l’action **Zombie/Create** GET afin d’utiliser la méthode **ListZombieTypeDisponible** du service

zombieVM.ZombieTypeSelectList = (IEnumerable<SelectListItem>?)\_serviceZT.ListZombieTypeDisponible();

1. Mettez un point d’arrêt et testez l’action Zombie/Create
2. Décommentez Create POST
3. NOTE : La gestion des images aurait intérêt à être sortie du contrôleur pour être mis dans le service, nous le ferons dans un laboratoire ultérieur.
4. Modifiez le code de l’action Create Post afin qu’il utilise la méthode **CreateAsync** du service. N’oubliez pas d’enlever SaveChangesAsync
5. Testez la vue Create POST

### vue Zombie/Edit

1. Modifiez le code de l’action Edit Post afin qu’il utilise la méthode **EditAsync** du service. N’oubliez pas d’enlever SaveChangesAsync

## Zombie avec un nom unique

Dans le **ZombieService**, ajoutez le code nécessaire afin de ne pas pouvoir créer ou modifier un Zombie avec un *Name* existant déjà dans la BD.

1. Créez une méthode **ZombieNameExist** vérifiant l’existant du nom pouvant être utilisé dans Create et Edit

bool IZombieService.ZombieNameExist(string name)

{

var ZombieSameName = \_dbContext.Zombies.Where(x => x.Name == name).Any();

return ZombieSameName;

}

1. Dans le contrôleur **Zombie,** ajoutez le code nécessaire pour vérifier l’existante du *name* dans les actions Create et Edit et retourner un message de type Error à l’utilisateur.

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(ZombieVM zombieVM)

{

if(\_serviceZ.ZombieNameExist(zombieVM.Zombie.Name))

{

TempData[AppConstants.Error] = $"Zombie {zombieVM.Zombie.Name} is already exist.";

return View(zombieVM);

}

//Si le modèle est valide le zombie est ajouté et nous sommes redirigé vers index.

if (ModelState.IsValid)

{

1. Testez les actions, utilisez les points d’arrêt

# Aller plus loin avec Javascript/jquery

L’objectif est de personnaliser le SummerNote (grandeur, placeholder, etc).

## SummerNote Create et Edit Zombie

1. Allez sur le site de [*SummerNote*](https://summernote.org/getting-started/)*,* Getting Started
2. Ouvrez la vue **Zombie/Create**
3. Modifiez le script de SummerNote afin d’obtenir un textArea de 300 de haut avec *Entrez la description ici* comme PlaceHolder
4. Testez la vue
5. Modifiez les options de la barre d’outils afin d’obtenir ceci :

Une image contenant texte, capture d’écran, Police, algèbre

Description générée automatiquement

1. Testez la vue
2. Modifiez les options de la barre d’outils afin d’obtenir ceci :

Une image contenant texte, capture d’écran, Police, blanc

Description générée automatiquement

1. Testez la vue

## Commentaires et validation (Commit) des changements dans le code

1. Faites un dernier **Commit**
2. "Poussez" **Push** les modifications sur le repo en ligne **Remote**: cliquez sur 
3. Merge la branche **FCT \_Services** dans la branche **Main** et faites un **dernier push.**

**Fin du laboratoire!**