Laboratoire: DataLayers Repository Patterns

# Objectif(s)

* DataLayers et repository patterns:
  + Fractionner une solution en plusieurs projets (DataLayers)
  + Mise en place des Repository Patterns

# Créer la branche de la séance

## Création du Repository et de la branche de la séance

1. Dans Gitkraken, clonez votre repository GitHub **A21\_3W6\_Labos\_NOM\_PRENOM**
2. Cliquez sur la branche de la séance précédente: **S04\_ EF \_Fluent\_API**
3. Créez la branche de la séance: **S05\_MultiPrj\_RepoPatterns**
4. Modifiez la connexion dans Appsettings.json : le nom du serveur
5. Faitez un Update-database pour générer la BD

# Accès aux données

## Créer le projet pour les accès aux données

1. Créez un projet de type bibliothèque (library) de classe Net Core **ZombieParty\_DataAccess**
2. Insallez les packages :
   * Microsoft.EntityFrameworkCore
   * Microsoft.EntityFrameworkCore.SqlServer
   * Microsoft.EntityFrameworkCore.Tools
3. Ajoutez une référence du projet ZombieParty (MVC) vers le nouveau projet **ZombieParty\_DataAccess**

Clique-droit sur le projet ZombieParty- Ajouter – référence du projet

1. Déplacez le dossier Data et la classe **ZombiePartyDbContext** dans le projet
2. Corrigez le Namespace
3. Corrigez les références (using) nécessaire (Regénérez et vérifiez les messages d’erreurs) ) dans les Controllers
4. Regénérez le PROJET **ZombieParty**
5. Corrigez les erreurs s'il y a lieu, regénérez

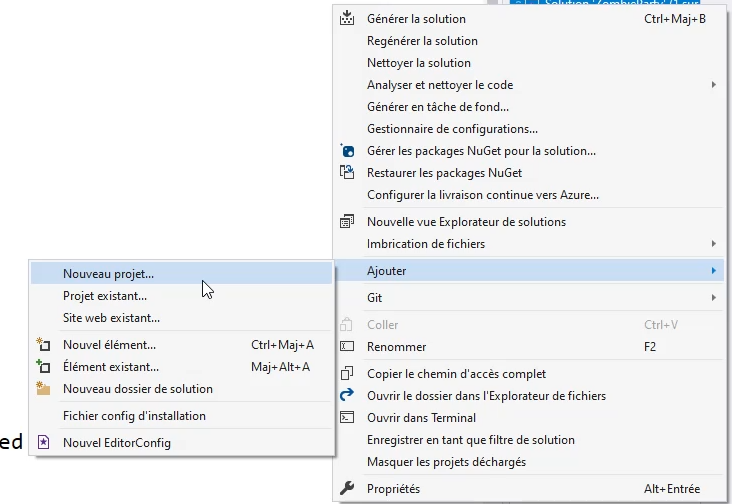
## Commentaires et validation (Commit) des changements dans le code

1. Dans Gitkraken, assurez-vous d'être dans le bon dossier/repo et sur la bonne branche
2. Cliquez sur **View changes**
3. Validez les modification: cliquez sur **Stage all changes**
4. Commentez en respectant les standards proposés (vous serez noté là-dessus):
   * **Summary** REFACTOR DataAccess DbContext
   * **Description**: Fractionner le projet : création du DataAccess
5. Cliquez sur **Commit**
6. "Poussez" **Push** les modifications sur le repo en ligne **Remote**: cliquez sur 

# Modèle de données

## Créer le projet pour le Modèle de données

1. Créez un projet de type blibliothèque (library) de classe Net Core, clic-droit sur la solution- Ajouter-Nouveau projet **ZombieParty\_Models**



1. Insallez les packages :
   * Microsoft.EntityFrameworkCore
   * Microsoft.AspNetCore.Mvc.ViewFeatures
2. Ajoutez une référence du projet **ZombieParty (MVC)** vers le nouveau projet **ZombieParty\_Models**

Clique-droit sur le projet **ZombieParty**- Ajouter – référence du projet

1. Ajoutez une référence du projet **DataAccess** vers le nouveau projet **ZombieParty\_Models**
2. Déplacez le contenu du dossier models dans le projet
3. Corrigez le Namespace de ZombiePartyDbContext
4. Corrigez les références (using) nécessaire (Regénérez et vérifiez les messages d’erreurs)
5. NOTE : exemple la référence en haut de la view Error puisque ErrorViewModel est maintenant dans **ZombieParty\_Models**

@model ErrorViewModel

@{

ViewData["Title"] = "Error";

}

1. Regénérez le PROJET **ZombieParty\_Models**
2. Corrigez les erreurs s'il y a lieu, regénérez
3. Regénérez le PROJET **ZombieParty\_DataAccess**
4. Corrigez les erreurs s'il y a lieu, regénérez
5. Regénérez le PROJET **ZombieParty**
6. Corrigez les erreurs s'il y a lieu, regénérez

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

1. Dans Gitkraken, assurez-vous d'être dans le bon dossier/repo et sur la bonne branche
2. Cliquez sur **View changes**
3. Validez les modification: cliquez sur **Stage all changes**
4. Commentez en respectant les standards proposés (vous serez noté là-dessus):
   * **Summary**: REFACTOR Modèle de données
   * **Description**: Fractionner le projet : création du Models
5. Cliquez sur **Commit**
6. "Poussez" **Push** les modifications sur le repo en ligne **Remote**: cliquez sur 

# Repository Patterns

## Configurer le repository générique

1. Dans le projet **ZombieParty\_DataAccess, c**réez un dossier **Repository** et un sous dossier **IRepository**
2. Créez un interface public appelé **IRepository** dans le dossier **IRepository**

// Ce doit être une interface générique <T> (T pour s'adapter au type d'objet classe ) publique

public interface IRepository<T> where T : class

{

// Les méthodes devant être implantées dans les repositories

T Get(int id);

IEnumerable<T> GetAll(

Expression<Func<T, bool>> filter = null,

Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null,

string includeProperties = null,

bool isTracking = true

);

// Retourne le 1er seulement

T FirstOrDefault(

Expression<Func<T, bool>> filter = null,

string includeProperties = null,

bool isTracking = true

);

void Add(T entity);

void Remove(T entity);

void RemoveRange(IEnumerable<T> entity);

void Save();

}

1. Créez une classe public appelée **Repository** dans le dossier **Repository** qui implémente **IRepository**
2. Implémentez l’interface (au besoin référez-vous au projet démo)

public class Repository<T> : IRepository<T> where T : class

{

// pour accéder à la BD, aux entités

private readonly ZombiePartyDbContext \_db;

// pour faire des changements directs

internal DbSet<T> dbset;

public Repository(ZombiePartyDbContext db)

{

\_db = db;

this.dbset = \_db.Set<T>();

}

public void Add(T entity)

{

dbset.Add(entity);

}

public T Get(int id)

{

return dbset.Find(id);

}

public T FirstOrDefault(Expression<Func<T, bool>> filter = null, string includeProperties = null, bool isTracking = true)

{

IQueryable<T> query = dbset;

if (filter != null)

{

query = query.Where(filter);

}

if (includeProperties != null)

{

foreach (var includeProp in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))

{

// reproduit: \_db.Product.Include(u => u.Category).Include(u => u.Size)

// mais passé en string

query = query.Include(includeProp);

}

}

if (!isTracking)

{

query = query.AsNoTracking();

}

return query.FirstOrDefault();

}

public IEnumerable<T> GetAll(Expression<Func<T, bool>> filter = null, Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null, string includeProperties = null, bool isTracking = true)

{

IQueryable<T> query = dbset;

if (filter != null)

{

query = query.Where(filter);

}

if (includeProperties != null)

{

foreach (var includeProp in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))

{

// reproduit: \_db.Product.Include(u => u.Category).Include(u => u.Size)

// mais passé en string

query = query.Include(includeProp);

}

}

if (orderBy != null)

{

query = orderBy(query);

}

if (!isTracking)

{

query = query.AsNoTracking();

}

return query.ToList();

}

public void Remove(T entity)

{

dbset.Remove(entity);

}

public void RemoveRange(IEnumerable<T> entity)

{

// Supprimer plusieurs lignes en même temps

dbset.RemoveRange(entity);

}

public void Save()

{

\_db.SaveChanges();

}

}

## Configurer le UnitOfWork

1. Créez un interface public appelé **IUnitOfWork** qui implémente l’interface **IDiposable** dans le dossier **IRepository**
2. Ajoutez la signature pour Save()

public interface IUnitOfWork: IDisposable

{

void Save();

}

1. Créez une classe public appelée **UnitOfWork** dans le dossier **Repository** qui implémente **IunitOfWork**
2. Liez le UnitOfWork au DbContext
3. Dans Sartup **Configuration**, ajoutez l'injection à **l'UnitOfWork**

services.AddScoped<IUnitOfWork, UnitOfWork>();

## Les repositories spécifiques

**Pour chaque entité/classe du modèle (pas ViewModel) :**

1. Dans **IRepository**, ajoutez un interface public spécifique pour l’entité héritant de l’interface générique

Ajoutez dans l’interface la signature pour le **Update**

public interface INomClasseRepository:IRepository<NomClasse>

{

void Update(NomClasse nomClasse);

}

1. Définir une classe de repository spécifique pour la classe, qui hérite de la classe générique et implémente l’interface spécifique

public class NomClasseRepository : Repository<NomClasse>, INomClasseRepository

{

private readonly ZombiePartyDbContext \_db;

public NomClasseRepository(ZombiePartyDbContext db) : base(db)

{

\_db = db;

}

public void Update(NomClasse nomClasse)

{

\_db.Update(nomClasse);

}

}

1. Ajoutez dans l’interface **IUnitOfWork** la propriété du type de l’interface spécifique

public interface IUnitOfWork: IDisposable

{

INomClasseRepository Book { get; }

void Save();

}

1. Dans l'interface **IUnitOfWork,** ajoutez le code nécessaire à la création du repository spécifique

public interface IUnitOfWork: IDisposable

{

INomClasseRepository NomClasse { get; }

void Save();

}

1. Dans la classe **UnitOfWork,** ajoutez le code nécessaire à la création du repository spécifique

public UnitOfWork(ZombiePartyDbContext db)

{

\_db = db;

NomClasse = new NomClasseRepository(\_db);

}

public INomClasseRepository NomClasse { get; private set; }

public void Dispose()

{

\_db.Dispose();

}

1. Refaire les étapes pour CHAQUE ENTITÉ/CLASSE
2. Dans Gitkraken, assurez-vous d'être dans le bon dossier/repo et sur la bonne branche
3. Cliquez sur **View changes**
4. Validez les modification: cliquez sur **Stage all changes**
5. Commentez en respectant les standards proposés (vous serez noté là-dessus):
   * **Summary**: REFACTOR Repository Patterns
   * **Description**: Mise en place du repository patterns et UnitOfWork
6. Cliquez sur **Commit**
7. "Poussez" **Push** les modifications sur le repo en ligne **Remote**: cliquez sur 

# Controller et View

## Fonctionnalité

1. Créez/Modifiez le **ZombieController,** l'action **Index** et la view Zombie**/Index** qui affiche la liste des Zombies, en utilisant le **UnitOfWork**