**Міністерство освіти і науки України**

**Національний авіаційний університет**

**Факультет кібербезпеки, комп’ютерної та програмної інженерії**

**Кафедра інженерії забезпечення програмного забезпечення**



Лабораторна робота №4-5

з дисципліни «Архітектура та проектування програмного забезпечення»

«ПРОЕКТУВАННЯ БАГАТОРІВНЕВОЇ АРХІТЕКТУРИ ЗАСТОСУВАННЯ»

Виконав студент:

групи ПІ-223Б

Адамов О. О.

Перевірив викладач:

Стець А. М.

Київ 2023

**Завдання**

1. Спроектувати застосування, розділене на рівні, а також представити у вигляді UML діаграми(-ам) функціонал згідно з варіантом.

1.1. Представити загальну архітектуру застосування, зв’язки між рівнями.

1.2. Представити діаграми класів кожного рівня.

2. Розробити застосування на мові C#, яка відповідає вимогам у варіанті. Відокремити рівні доступу до даних, бізнес логіки та представлення. Для демонстрації роботи можна використати будь-який UI: консольний ввід-вивід, WinForms, WPF чи Web.

1.1. Верхній рівень – представлення (UI), для взаємодії з користувачем. Реалізація – як окремий проект в рішенні (solution). UI може бути будь-якого типу (консольне застосування, WinForms, WebForms, WPF, ASP.NET MVC, ASP.NET WebAPI). При цьому на рівні UI повинні бути тільки операції взаємодії з користувачем. UI повинен бути максимально простим, неперевантаженим великою кількістю операцій.

1.2. Проміжний рівень – бізнес логіка, реалізована як бібліотека в рішенні. На цьому рівні реалізований основний функціонал застосування.

1.3. Нижній рівень – бібліотека для доступу до даних. Використати Entity Framework code first. Доступ до даних організувати з використанням шаблонів проектування Repository та Unit of Work (UoF). Репозіторії надають доступ до набору сутностей (entities) певного типу. UoW є точкою єдиного доступу до репозіторіїв та контексту Entity Framework.

3. Шари взаємодіють між собою за наступним принципом: представлення використовує бізнес-логіку, бізнес-логіка – рівень доступу до даних. Для передачі даних крізь шари використовується технологія відображення (mapping).

4. При необхідності для більшої ізоляції основних рівнів можуть вводитися додаткові рівні (наприклад, винесення Repository та UoW).

5. Діаграма(-и) та вихідний код повинні відповідати основним принципам проектування: OOP, SOLID, Law of Demeter (LoD), DRY, YAGNI, KISS, cohesion – coupling, inheritance with caution.

6. Для отримання балів, що відповідають «задовільно» можна не використовувати mapping, реалізація шаблонів Repository та UoW може бути не ізольованою від інших рівнів, без інтерфейсів та не generic (явна імплементація репозиторію для кожної сутності), без використання зв’язків між сутностями. А також без використання репозиторію Git. Обов’язково повинні бути реалізовані CRUD операції.

7. Для отримання балів, що відповідають «добре» можна не використовувати mapping, але повинен бути механізм передачі даних з рівня на рівень (щоб сутності з DAL були тільки на BLL); реалізація шаблонів Repository та UoW повинна бути достатньо ізольованою від інших рівнів, з використанням інтерфейсів та з використанням generic репозиторію. При цьому допускається імплементація не всіх операцій саме в generic репозиторії. А також необхідно використати репозиторій Git.

8. Для отримання балів, що відповідають «відмінно» обов’язково потрібно використати mapping (будь-які підходи та бібліотеки). Реалізувати один generic репозиторій для всіх операцій з усіма сутностями. Імплементація операцій UoW та використання репозиторіїв у UoW повинна бути відокремленою. Реалізація шаблонів Repository та UoW повинна бути достатньо ізольованою від інших рівнів, щоб була можливість в рішення (solution) додати інші джерела даних таким чином, щоб не було потреби змінювати рівень DAL з EF. Також залишаються вимоги з попереднього пункту, що стосуються репозиторію Git.

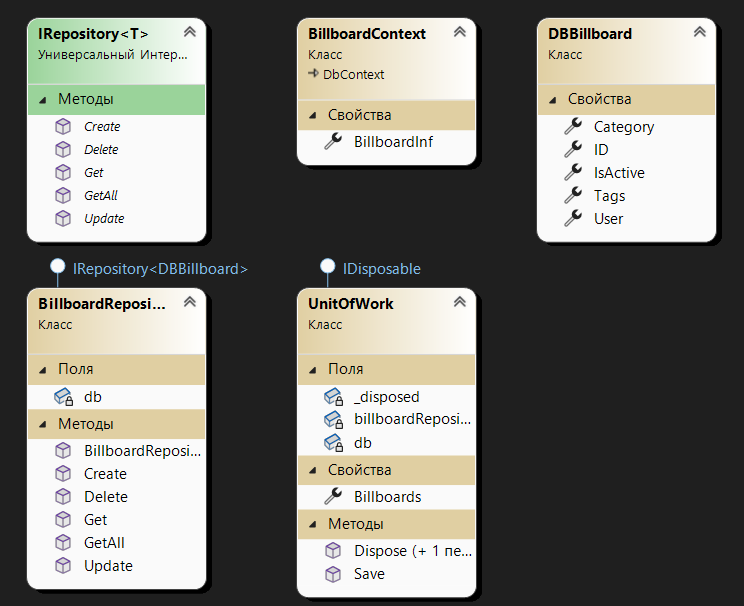
**Варіант 1**

|  |  |  |
| --- | --- | --- |
| Номер варіанту | Предметна область | Вимоги та компоненти |
| 1 | Дошка оголошень | Рубрики/ категорії/ підкатегорії, теги, користувачі. Оголошення можна додати в одну категорію, але з кількома тегами. Реалізувати основні операції по роботі з дошкою оголошень. Видалити оголошення та деактивувати оголошення може тільки той користувач, який його додав. |

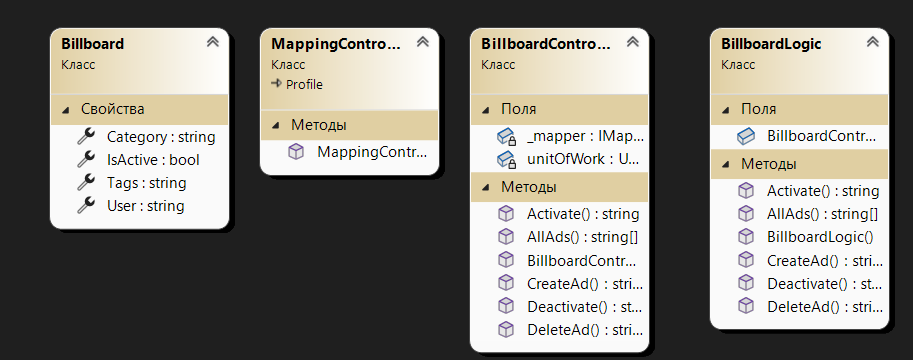
**Виконання роботи:**

Для лабораторної роботи було розроблено такі рівні проектування, як DAL, BLL та PL. DAL мав на меті взаємодію з компонентами базою даних, тобто з дошкою оголошень. BLL виконував логіку взаємодії та мапінг звичайних полів в об’єкти бази даних. PL являв собою консоль, через яку користувач міг взаємодіяти із базою даних.

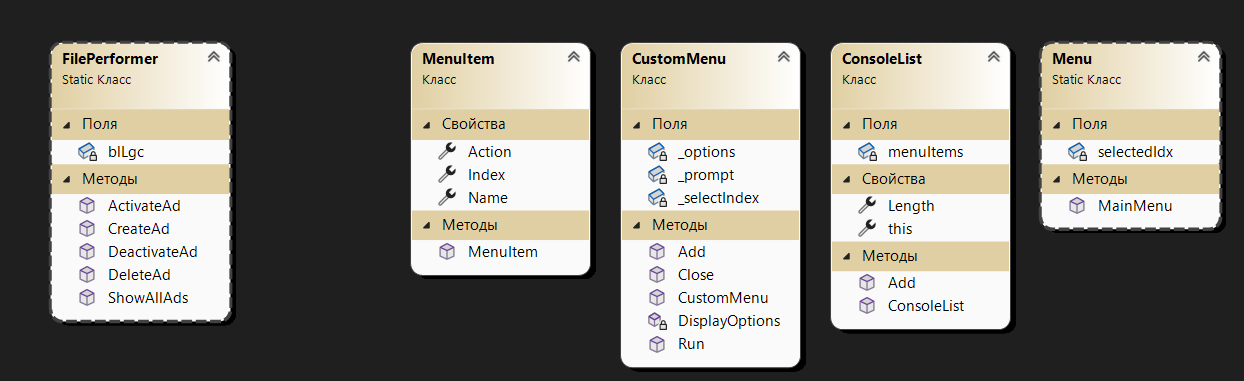
Продемонструю UML-діаграму для DAL:



UML-діаграма для BLL:



UML-діаграма для PL:



Тепер перейду до опису DAL. У цьому проекті наявний загальний інтерфейс IRepository, який має 5 методів для взаємодії із базою даних.

public interface IRepository<T> where T : class

{

IEnumerable<T> GetAll();

T Get(int id);

void Create(T item);

void Update(T item);

void Delete(int id);

}

Даний інтерфейс унаслідується класом BillboardRepository, у якому є поле класу BillboardContext, що у свою чергу містить поле типу DbSet, яке представляє сутності контексту дошки оголошень.

Ось код до класу BillboardRepository:

public class BillboardRepository : IRepository<DBBillboard>

{

private BillboardContext db;

public BillboardRepository(BillboardContext context) { db = context; }

public void Create(DBBillboard item)

{

db.BillboardInf.Add(item);

}

public void Delete(int id)

{

DBBillboard billboard = db.BillboardInf.Find(id);

if (billboard != null) db.BillboardInf.Remove(billboard);

}

public DBBillboard Get(int id)

{

return db.BillboardInf.Find(id);

}

public IEnumerable<DBBillboard> GetAll()

{

return db.BillboardInf;

}

public void Update(DBBillboard item)

{

// db.Entry(item).State = EntityState.Modified;

}

}

Також присутній клас UnitOfWork, який за допомогою поля типу BillboardContext може зберігати вміст бази даних. Також є властивість Billboards, яка повертає екземпляр класу BillboardRepository.

Опис класу UnitOfWork:

public class UnitOfWork : IDisposable

{

private BillboardContext db = new BillboardContext();

private BillboardRepository billboardRepository;

private bool \_disposed = false;

public BillboardRepository Billboards

{

get

{

if (billboardRepository == null) billboardRepository = new BillboardRepository(db);

return billboardRepository;

}

}

public void Save()

{

db.SaveChanges();

}

public virtual void Dispose(bool displosing)

{

if (!\_disposed) db.Dispose();

\_disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

BLL має клас MappingController, що виконує перетворення звичайних полів класу Billboard у стовпці бази даних дошки оголошень.

Ось його опис:

public class MappingController : Profile

{

public MappingController()

{

CreateMap<Billboard, DBBillboard>()

.ForMember(dest => dest.ID, opt => opt.Ignore());

CreateMap<Billboard, DBBillboard>()

.ForMember(dest => dest.Category, opt => opt.MapFrom(src => src.Category))

.ForMember(dest => dest.Tags, opt => opt.MapFrom(src => src.Tags))

.ForMember(dest => dest.User, opt => opt.MapFrom(src => src.User))

.ForMember(dest => dest.IsActive, opt => opt.MapFrom(src => src.IsActive));

}

}

Також наявний клас BillboardController, за допомогою якого відбувається взаємодія програмного коду із базою даних. Таким чином, при викликанні методів користувач може створювати та видалити, активувати та деактувати, показати вміст дошки оголошень.

public class BillboardController

{

UnitOfWork unitOfWork;

readonly IMapper \_mapper;

public BillboardController(IMapper mapper) { unitOfWork = new UnitOfWork(); \_mapper = mapper; }

public string CreateAd(Billboard blbrd)

{

Billboard billboard = blbrd;

var dbBillboard = \_mapper.Map<DBBillboard>(billboard);

unitOfWork.Billboards.Create(dbBillboard);

unitOfWork.Save();

return $"User: {billboard.User}\n Category: {billboard.Category}\n Tags:{billboard.Tags}";

}

public string DeleteAd(int id, string name)

{

if (unitOfWork.Billboards.Get(id) == null) return "This ad doesn't exist";

if (name == unitOfWork.Billboards.Get(id).User)

{

unitOfWork.Billboards.Delete(id);

unitOfWork.Save();

return $"The ad with ID {id} was deleted!";

}

else

return "You don't have an access to deleting this ad";

}

public string[] AllAds()

{

int size = unitOfWork.Billboards.GetAll().Count();

if (size == 0) return new string[] {"There are no any adverts on the billboard!"};

string[] arr = new string[size];

DBBillboard[] allAds = unitOfWork.Billboards.GetAll().ToArray();

for (int i = 0; i < arr.Length; i++)

{

DBBillboard item = allAds[i];

if (item.IsActive)

arr[i] = $"User: {item.User}\n Category: {item.Category}\n Tags:{item.Tags}\n Active:{item.IsActive}\n";

}

return arr;

}

public string Deactivate(int id, string name)

{

try

{

if (unitOfWork.Billboards.Get(id) == null) return "This ad doesn't exist";

bool isExist = name == unitOfWork.Billboards.Get(id).User;

if (isExist)

{

unitOfWork.Billboards.Get(id).IsActive = false;

unitOfWork.Save();

return $"The advertisement with ID {id} was deactivated!";

}

else

return "You don't have permission to deactivate this ad!";

}

catch (NullReferenceException)

{

throw;

}

}

public string Activate(int id, string name)

{

try

{

if (unitOfWork.Billboards.Get(id) == null) return "This ad doesn't exist";

bool isExist = name == unitOfWork.Billboards.Get(id).User;

if (isExist)

{

unitOfWork.Billboards.Get(id).IsActive = true;

unitOfWork.Save();

return $"The advertisement with ID {id} was activated!";

}

else

return $"You don't have permission to activate this ad!";

}

catch (Exception)

{

throw;

}

}

}

Ще наявний клас BillboardLogic, який представляє собою більш завершену реалізацію класу BillboardContext, де параметри в методи передаються за допомогою базових типів.

PL має чотири класи, які описують консольне меню. Особливе місце займає клас FilePerformer, що за допомогою вводу і виводу може взаємодіяти з попередніми рівнями, і в такий спосіб із базою даних.

public static class FilePerformer

{

private static BillboardLogic blLgc = new BillboardLogic();

public static void CreateAd()

{

Console.Write("Write your name: ");

string name = Console.ReadLine();

Console.Write("Write a category: ");

string categ = Console.ReadLine();

Console.Write("Write tags: ");

string tags = Console.ReadLine();

Console.WriteLine("");

Console.WriteLine(blLgc.CreateAd(name, categ, tags));

Console.ReadLine();

}

public static void DeleteAd()

{

Console.Write("Write your name: ");

string name = Console.ReadLine();

Console.Write("Write ID of the product: ");

int id = int.Parse(Console.ReadLine());

Console.WriteLine("");

Console.WriteLine(blLgc.DeleteAd(id, name));

Console.ReadLine();

}

public static void DeactivateAd()

{

Console.Write("Write your name: ");

string name = Console.ReadLine();

Console.Write("Write ID of the product: ");

int id = int.Parse(Console.ReadLine());

Console.WriteLine("");

try

{

Console.WriteLine(blLgc.Deactivate(id, name));

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.ReadLine();

}

public static void ActivateAd()

{

Console.Write("Write your name: ");

string name = Console.ReadLine();

Console.Write("Write ID of the product: ");

int id = int.Parse(Console.ReadLine());

Console.WriteLine("");

try

{

Console.WriteLine(blLgc.Activate(id, name));

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.ReadLine();

}

public static void ShowAllAds()

{

string[] arr = blLgc.AllAds();

foreach (string s in arr) Console.WriteLine(s);

Console.ReadLine();

}

}

**Контрольні запитання**

1. **Що таке багаторівнева архітектура ПЗ та які переваги її використання?**

Багаторівнева архітектура, це така архітектура в якій є чіткий поділ на рівні, що виконують своє, специфічне для цього рівня, завдання. Найпоширеніша архітектура такого типу, це трирівнева архітектура, в який є рівень представлення, рівень бізнес операцій та рівень доступу до даних.

Переваги:

* Маштабованість;
* Високий рівень безпеки;
* Висока надійність;
* Низькі вимоги до продуктивності і технічних характеристик терміналів;
* Конфігурованість.

1. **Які існують шаблони проектування для організації шару доступу до сховищ даних?**

Для організації шару доступу до даних існують шаблони проектування Repository та Unit of Work. Репозиторії надають доступ до набору сутностей (entities) певного типу. UoW є точкою єдиного доступу до репозиторіїв та контексту Entity Framework.

1. **Які типи сховищ даних існують?**

Існують такі типи сховищ даних:

* Централізовані;
* Хмарні;
* Комерційні;
* Розподілені;
* Бази даних кінцевого користувача;
* Бази даних графів;
* Об’єктно-орієнтовані;
* Реляційні.

1. **Що таке ОRM?**

ORM (об’єктно-реляційна проекція) – це технологія програмування, яка зв’язує бази даних з концепціями об’єктно-орієнтованого програмування, створюючи «віртуальну об’єктну базу даних».

1. **Поясніть ціль використання технології Entity Framework (EF) в архітектурі програмних систем.**

Технологія Entity Framework потрібна для доступу до баз даних шляхом нескладного інтерфейсу. Коли потрібен доступ до БД, користувач звертається до EF з деяким запитом, простим та зрозумілим для користувача, а EF перетворює його в запит SQL і звертається до БД.

1. **Наведіть основні складові EF.**

До основних складових належать:

* Entity Data Model (EDM) – це модель даних, яка визначає структуру даних та взаємозв’язки між ними;
* Object Services - це складова, яка забезпечує зв'язок між EDM та програмою, яка працює з даними. Об'єкти, що репрезентують дані з бази даних, автоматично згенеровані EDM, і можуть бути використані в програмі.
* LINQ to Entities - це складова, яка дозволяє виконувати запити до бази даних в термінах LINQ (Language-Integrated Query), який є частиною .NET Framework.
* Entity Client - це складова, яка забезпечує зв'язок між EDM та базою даних.
* Entity Framework Designer - це інструмент, який дозволяє створювати та редагувати моделі даних EDM у візуальному середовищі Visual Studio.
* Entity Framework Code First - це підход до розробки програмного забезпечення, що дозволяє створювати моделі даних з використанням коду C# або VB.NET, а потім створювати базу даних з цих моделей.
* Entity Framework Migrations - це механізм, який дозволяє здійснювати міграції бази даних при зміні моделі даних. За допомогою цього механізму можна змінювати структуру бази даних без втрати даних.
* Entity Framework Providers - це драйвери, які забезпечують зв'язок між Entity Framework та конкретною базою даних. Наприклад, для зв'язку з базою даних SQL Server можна використовувати Microsoft SQL Server Provider for Entity Framework.

1. **Поясніть призначення класів сутностей (entities) у EF?**

Класи сутностей (entities) у Entity Framework (EF) використовуються для репрезентації таблиць бази даних як об'єктів у програмі. Класи сутностей представляють окремі записи таблиць бази даних та забезпечують можливість звернення до цих записів за допомогою об'єктно-орієнтованих методів та властивостей.

Основне призначення класів сутностей у EF полягає у наступному:

* Забезпечення доступу до даних в таблицях бази даних у вигляді об'єктів у програмі.
* Представлення взаємозв'язків між таблицями бази даних у вигляді об'єктних відносин.
* Забезпечення можливості виконання операцій додавання, видалення, оновлення даних у таблицях бази даних з використанням об'єктів класів сутностей.
* Забезпечення можливості виконання запитів до даних з використанням мови запитів LINQ.
* Забезпечення контролю доступу до даних та безпеки.

1. **Поясніть призначення класу контексту (context) у EF?**

Клас контексту (context) у Entity Framework (EF) використовується для управління з'єднанням з базою даних та керування об'єктами класів сутностей (entities). Контекст EF є основною точкою взаємодії між програмою та базою даних, і забезпечує можливість виконання запитів до бази даних та збереження змін.

Основне призначення класу контексту у EF полягає у наступному:

* Забезпечення зв'язку між базою даних та класами сутностей (entities) у програмі.
* Забезпечення управління з'єднанням з базою даних.
* Забезпечення можливості виконання запитів до бази даних з використанням мови запитів LINQ.
* Забезпечення можливості збереження змін до бази даних з використанням об'єктів класів сутностей.
* Забезпечення кешування даних та підтримки транзакцій.

1. **Які підходи до створення сутностей EF існують? В чому їх переваги та недоліки?**

У Entity Framework (EF) існує кілька підходів до створення сутностей (entities). Основні підходи включають Code First, Model First та Database First. Кожен з цих підходів має свої переваги та недоліки.

* Code First

Підхід Code First передбачає визначення класів сутностей (entities) та контексту EF безпосередньо в коді програми. При використанні цього підходу EF створює базу даних автоматично на основі визначених класів сутностей. Перевагою цього підходу є його простота та гнучкість. Недоліком є необхідність вручну налаштовувати відображення сутностей на таблиці бази даних, а також обмеження в можливості використання вже існуючої бази даних.

* Model First

Підхід Model First передбачає використання спеціального інструменту для визначення моделі бази даних, яка потім використовується для автоматичної генерації класів сутностей та контексту EF. Перевагами цього підходу є можливість швидко створювати базу даних та зменшення кількості коду, який потрібно писати. Недоліком є обмежена гнучкість у порівнянні з підходом Code First, а також потреба у додатковому інструменті для визначення моделі бази даних.

* Database First

Підхід Database First передбачає використання існуючої бази даних для автоматичної генерації класів сутностей та контексту EF. Перевагою цього підходу є можливість швидко створювати класи сутностей та контекст EF на основі наявної бази даних. Недоліком є обмежена гнучкість у порівнянні з підходом Code First, а також необхідність вручну налаштовувати відображення сутностей на таблиці бази даних.

1. **Що таке .edmx-модель? Яким чином вона створюється та описується?**

Файл з розширенням .edmx (Entity Data Model XML) представляє собою файл моделі даних, який використовується в Entity Framework для відображення структури бази даних та зв'язків між таблицями у вигляді об'єктів.

Створення .edmx-моделі передбачає виконання наступних кроків:

1. Відкриття проекту в Visual Studio та додавання нового елемента "ADO.NET Entity Data Model".
2. Вибір типу моделі: "Empty Model", "Database First", "Code First from Database" або "Code First from Scratch".
3. Якщо ви обрали "Database First", потрібно вказати розташування бази даних та вибрати таблиці та представлення, які потрібно включити до моделі.
4. Після створення моделі, вона може бути відредагована за допомогою візуального редактора, який дозволяє додавати та видаляти сутності, зв'язки та властивості.
5. Файл .edmx зберігається в форматі XML, який містить відображення всієї структури бази даних та зв'язків між таблицями.
6. **Поясніть принцип роботи підходу code first.**

Підхід Code First передбачає визначення класів сутностей (entities) та контексту EF безпосередньо в коді програми. При використанні цього підходу EF створює базу даних автоматично на основі визначених класів сутностей. Перевагою цього підходу є його простота та гнучкість. Недоліком є необхідність вручну налаштовувати відображення сутностей на таблиці бази даних, а також обмеження в можливості використання вже існуючої бази даних.

1. **Поясніть призначення утиліти Nuget package.**

Утиліта NuGet package - це менеджер пакетів для розробників, що дозволяє легко додавати, встановлювати та оновлювати бібліотеки та інші компоненти для проектів .NET.

NuGet дозволяє швидко та просто встановлювати сторонні бібліотеки, які потрібні для розробки проекту, у тому числі пакети, що розроблені спільнотою розробників. Крім того, він дозволяє автоматично вирішувати залежності між пакетами та контролювати їх версії.

NuGet дозволяє зберігати та управляти пакетами на локальному комп'ютері або в обласному репозиторії, що дозволяє повторно використовувати код та складові між проектами. Крім того, NuGet дозволяє створювати та розповсюджувати власні пакети, що дозволяє розробникам легко поширювати власні компоненти.

1. **Як працюють операції CRUD у EF?**

* Створення (Create): Для створення нового запису в базі даних необхідно створити новий екземпляр класу-сутності та додати його до DbSet за допомогою методу Add.
* Читання (Read): Для читання записів з бази даних можна використовувати методи DbSet, які повертають IQueryable. Наприклад, для отримання всіх користувачів можна
* Оновлення (Update): Для оновлення записів в базі даних необхідно спочатку знайти сутність, яку необхідно оновити, і змінити її властивості, а потім зберегти зміни за допомогою методу SaveChanges.
* Видалення (Delete): Для видалення запису з бази даних необхідно спочатку знайти сутність, яку необхідно видалити, і видалити її з DbSet за допомогою методу Remove, а потім зберегти зміни за допомогою методу SaveChanges.

1. **Що таке «жадібне» та «ліниве» завантаження?**

Жадібне завантаження означає, що EF завантажує всі пов’язані дані з бази даних одним запитом до БД при виконанні основного запиту. Це означає, що всі пов’язані дані будуть доступні в об’єкті-сутності без додаткового запиту до БД. Цей підхід може бути корисним у випадках, коли необхідно отримати всі пов’язані дані одночасно, або коли кількість записів не дуже велика. Жадібне завантаження можна виконати за допомогою методу Include().

Ліниве завантаження означає, що EF завантажує пов’язані дані з бази даних тільки тоді, коли вони запитуються з коду. Це означає, що пов’язані дані не завантажуються при виконанні основного запиту і не займають пам’ять, що може бути корисним у випадках, коли пов’язані дані не завжди потрібні, або коли кількість записів дуже велика. Ліниве завантаження можна виконати за допомогою ключового слова virtual у визначенні властивості-навігації. При спробі отримати пов’язані дані, EF автоматично завантажує їх з бази даних.

1. **Як реалізуються зв’язки 1:1, 1:М, М:N між сутностями EF? Навести приклади, використовуючи класи-сутності, розроблені у проекті.**

Зв'язок 1:1. Для встановлення зв'язку 1:1 між двома сутностями необхідно використовувати атрибут [ForeignKey] або метод Fluent API HasOptional() або HasRequired(). Але цього можна не робити, оскільки EF досить розумний, щоб зрозуміти такий тип зв’язку. В проекті я не використовував такий тип зв’язку.

Зв'язок 1:М. Для встановлення зв'язку 1:М між двома сутностями необхідно використовувати віртуальні властивості в сутностях та метод Fluent API HasMany() та HasForeignKey(). Але цього можна не робити, оскільки EF досить розумний, щоб зрозуміти такий тип зв’язку. Наприклад:

public class Article : Commentable

{

public DateTime CreatedDate { get; set; } = DateTime.UtcNow;

public string Title { get; set; }

public string Description { get; set; }

public string Content { get; set; }

public User Author { get; set; }

public Rubric Rubric { get; set; }

}

public class User : IdentityUser

{

public string DisplayName { get; set; }

public string Bio { get; set; }

public DateTime CreatedDate { get; set; } = DateTime.UtcNow;

public bool IsDeleted { get; set; }

public ICollection<Comment> Comments { get; set; } = new List<Comment>();

public ICollection<Article> Articles { get; set; } = new List<Article>();

}

Зв'язок М:N. Для встановлення зв'язку М:N між двома сутностями необхідно створити спеціальну сутність, яка представляє зв'язок. У сутностях, які мають зв'язок М:N, створюються віртуальні властивості для цієї спеціальної сутності. EF не може зрозуміти такий тип зв’язку, тому додаткові сутність треба обов’язково робити. В проекті я не використовував такий тип зв’язку.

1. **Поясніть сутність шаблону Repository.**

Суть шаблону Repository полягає в тому, щоб мати окремий клас для кожної сутності в базі даних (або один generic клас), який містить методи для додавання, читання, оновлення та видалення даних для цієї сутності. Кожен клас Repository повинен мати інтерфейс, який визначає ці методи, що дозволяє замінювати реалізацію класу з іншою реалізацією в майбутньому, якщо це необхідно.

1. **В чому полягає особливість використання generic репозиторію.**

Використання generic репозиторію дозволяє створити загальний інтерфейс для доступу до даних безпосередньо з бази даних без необхідності створювати окремий код для кожної таблиці. Особливість полягає у використанні параметрів типу, які дозволяють вказувати тип ентіті, з яким працює репозиторій, в момент його створення. Це забезпечує більш гнучкий і простий код, оскільки не потрібно писати окремі методи для кожної таблиці. За допомогою generic репозиторію можна забезпечити доступ до бази даних для будь-якої ентіті, включаючи нові таблиці, які можуть бути додані в майбутньому, без необхідності змінювати код репозиторію. Однак, використання generic репозиторію може бути менш ефективним для деяких вимог до продуктивності, оскільки він не завжди генерує найоптимальніші запити до бази даних.

1. **Поясніть сутність шаблону Unit of Work.**

Шаблон Unit of Work (одиниця роботи) - це шаблон проектування, який дозволяє управляти транзакціями та збереженням змін в базі даних. В основі шаблону лежить ідея того, що всі операції з базою даних повинні виконуватися як одна транзакція, щоб гарантувати консистентність даних та запобігти виникненню проблем з транзакціями.

У контексті EF, Unit of Work допомагає управляти життєвим циклом сутностей та забезпечує групування змін в одну транзакцію. Він включає в себе контекст EF та надає загальний інтерфейс для доступу до даних. За допомогою шаблону Unit of Work, розробник може управляти транзакціями та взаємодіяти з базою даних, не звертаючись безпосередньо до контексту EF. Це дозволяє розробникам зосередитися на роботі з бізнес-логікою, не звертаючи увагу на деталі роботи з базою даних.

За допомогою шаблону Unit of Work можна забезпечити певні переваги, такі як управління транзакціями, групування змін в одну транзакцію та забезпечення консистентності даних. Однак, при неправильному використанні, цей шаблон може призвести до збільшення складності коду та зниження продуктивності.

1. **Як пов’язані EF та шаблони Repository та Unit of Work?**

EF можна використовувати з шаблонами Repository та Unit of Work для покращення організації та управління доступом до даних.

Шаблон Repository дозволяє ізолювати логіку доступу до даних від бізнес-логіки за допомогою введення проміжного рівня - репозиторіїв. Репозиторії виконують запити до бази даних та повертають результати до бізнес-логіки. За допомогою шаблону Repository, EF може бути використаний для забезпечення доступу до даних в репозиторіях. При цьому, EF може забезпечити маппінг об'єктів бази даних на класи C#, здійснювати валідацію даних та оптимізувати запити до бази даних.

Шаблон Unit of Work дозволяє управляти транзакціями та збереженням змін в базі даних. Якщо багато запитів до бази даних повинні бути виконані як одна транзакція, то можна скористатися шаблоном Unit of Work. При цьому, EF може бути використаний для створення контексту бази даних та збереження змін до бази даних. Unit of Work можна поєднати з шаблоном Repository, щоб забезпечити кращу організацію та управління доступом до даних.

1. **Поясніть сутність процесу mapping.**

Маппінг даних - це процес зіставлення полів даних (певних елементів джерела або всього джерела) і пов'язаних із ними полів даних в іншому місці призначення. Тобто це встановлення співвідношення між моделями даних, які знаходяться в різних джерелах або системах. Програмне забезпечення та інструменти мапування даних автоматично зіставляють поля даних з одного джерела даних в інше за користувача.

1. **Наведіть шари та їх призначення в багаторівневій архітектурі ПЗ.**

* Presentation Layer - це верхній рівень, який відповідає за взаємодію з користувачем та відображення даних. Його головною метою є представлення даних в зручному для користувача форматі. Шар може містити різні елементи, такі як веб-інтерфейс, додатки, мобільні додатки тощо.
* Application Layer - це рівень, який містить логіку додатку та відповідає за обробку даних, які надходять з Presentation Layer. Він забезпечує виконання бізнес-правил та керування поведінкою додатку.
* Business Layer - це рівень, який містить бізнес-логіку додатку та відповідає за обробку даних, які надходять з Application Layer. Цей рівень виконує більш складні завдання, такі як перевірка даних на відповідність бізнес-правилам, керування бізнес-процесами та виконання складних обчислень.
* Data Access Layer - це рівень, який відповідає за зберігання та доступ до даних. Він забезпечує зв'язок між базою даних та додатком, а також відповідає за операції з базою даних, такі як додавання, оновлення та видалення записів.

1. **Опишіть властивості шарів багаторівневої архітектури ПЗ.**

* Розділення відповідальності: кожен шар відповідає за свою частину функціональності програми. Це дозволяє зменшити залежність між компонентами програми і полегшити їхнє тестування та підтримку.
* Модульність: кожен шар може бути розроблений та тестований окремо від інших шарів. Це дозволяє зменшити складність програми та полегшити її розробку.
* Зручність для розробника: кожен шар має чітко визначені обов'язки та інтерфейси, що дозволяє розробникам працювати з ними більш ефективно.
* Переносимість: за рахунок розділення функціональності на шари, програма може бути легко портована на іншу платформу або архітектуру.
* Розширюваність: додавання нової функціональності до програми може бути реалізовано шляхом додавання нового шару, що дозволяє збільшити функціональність програми без зміни існуючих компонентів.
* Зручність для тестування: кожен шар може бути тестований окремо від інших шарів, що дозволяє зменшити складність тестування програми в цілому.
* Зручність для підтримки: за рахунок розділення відповідальності на шари, зміна функціональності одного шару не повинна впливати на роботу інших шарів.

1. **В чому полягає принцип ізоляції рівнів у багаторівневій архітектурі.**

Принцип ізоляції рівнів у багаторівневій архітектурі передбачає, що кожен рівень повинен бути ізольований від інших рівнів, тобто мати власний набір правил та обмежень, які забезпечують його коректну роботу і дозволяють змінювати його без впливу на інші рівні.

Основна мета ізоляції рівнів полягає в тому, щоб зменшити залежність між рівнями та зробити систему більш гнучкою і легко змінюваною. Це дозволяє розвивати кожен рівень окремо, без впливу на інші рівні, і спрощує розуміння та підтримку коду.

Крім того, ізоляція рівнів забезпечує високий рівень безпеки системи, оскільки рівні здатні контролювати доступ до своїх ресурсів та обмежувати можливості інших рівнів.

Принцип ізоляції рівнів може бути забезпечений за допомогою інтерфейсів та узагальнень, які дозволяють різним рівням взаємодіяти між собою за стандартними правилами та спільним інтерфейсом, що спрощує розвиток та підтримку системи в цілому.

1. **Яка користь від створення та використання сутностей (entities) на кожному рівні багаторівневої архітектури?**

Створення та використання сутностей (entities) на кожному рівні багаторівневої архітектури дозволяє досягти кількох користей:

* Розділення відповідальності. Кожен рівень має відповідати за свої функції та мати відповідний набір сутностей для роботи з даними. Це дозволяє розподілити відповідальність між різними командами розробників, що зменшує залежність від інших рівнів та сприяє збільшенню швидкості розробки.
* Забезпечення ізоляції даних. Кожен рівень має свій власний набір сутностей та не має прямого доступу до даних на інших рівнях. Це дозволяє забезпечити ізоляцію даних та запобігти їхньому несанкціонованому доступу.
* Покращення тестування. Кожен рівень можна тестувати окремо, що сприяє покращенню тестування та забезпечує більшу надійність та стабільність системи в цілому.
* Легше підтримувати та розширювати. За рахунок того, що кожен рівень має свої сутності та функції, легше розширювати та підтримувати систему в цілому. Наприклад, якщо необхідно додати нову функцію, то зазвичай достатньо додати нові сутності та методи на відповідному рівні, що дозволяє уникнути змін на інших рівнях та зменшити ризик збоїв.

1. **Чим відрізняються централізовані репозиторії (Subversion) від розподілених (Git)?**

Централізована система контролю версій (Subversion, CVS) зберігає код проекту на центральному сервері. Кожен розробник може здійснювати роботу з власного комп'ютера, але збереження змін відбувається в центральному репозиторії. Розробники мають доступ до останньої версії коду, але можуть зберігати лише свої власні зміни, поки вони не збережені в центральному репозиторії.

У розподіленій системі контролю версій (Git, Mercurial) кожен розробник отримує повну копію репозиторію на своєму комп'ютері. Зміни зберігаються локально та можуть бути збережені відразу в репозиторії. Кожен розробник може мати власну гілку розвитку проекту, яку можна об'єднати з головною гілкою в будь-який момент.

1. **Яка структура Git репозиторію?**

Git репозиторій містить такі основні структурні елементи:

* Каталог .git: це головний каталог репозиторію, який містить всі внутрішні файли Git, такі як об'єкти, гілки, коміти тощо.
* Об'єкти: Git зберігає дані про файли та каталоги в об'єктах. Кожен об'єкт має унікальний ідентифікатор SHA-1 і зберігається у вигляді стислого бінарного файлу. Існує три типи об'єктів: blob, tree та commit.
* Гілки: Git використовує гілки для відстеження різних версій проекту. Кожна гілка є посиланням на певний коміт, який відповідає стану проекту на цей момент. Гілки можуть бути створені, переключені та злиті.
* Коміти: Коміти представляють собою зміни в коді або вмісті файлів. Кожен коміт містить інформацію про зміни, автора, дату тощо. Коміти зберігаються як повний знімок стану проекту на даний момент.

1. **Які види вказівників існують в Git? В чому різниця між ними?**

В Git існує декілька видів вказівників:

* HEAD - це вказівник на поточний коміт у гілці. Кожна гілка має свій власний вказівник HEAD. В Git HEAD зазвичай вказує на головний коміт останньої здійсненої операції.
* Branches (гілки) - це вказівники на певну історію комітів. У Git гілки можуть бути злиті з іншими гілками або відокремлені від них.
* Tags (мітки) - це вказівники на конкретний коміт у віддаленому репозиторії. Вони використовуються для позначення важливих точок в історії комітів, таких як випуски програмного забезпечення або важливі оновлення.
* Commit Hash - це унікальний ідентифікатор кожного коміту в Git. Він складається з 40 символів (16-річна система числення) і використовується для посилання на конкретний коміт.

1. **Що таке коміт? Яка його структура? Як зробити коміт?**

У системі контролю версій Git коміт (commit) є операцією фіксації змін в файловій системі репозиторію. Коміт дозволяє зберегти зміни в репозиторії і додати їх історію розвитку проекту.

Структура коміту включає наступні елементи:

* Хеш (hash) - унікальний ідентифікатор коміту, який може бути використаний для звернення до коміту в майбутньому.
* Автор (author) - ім'я та адреса електронної пошти особи, яка здійснила коміт.
* Дата (date) - дата та час створення коміту.
* Повідомлення (message) - короткий опис внесених змін, що допомагає розуміти, що було змінено в файловій системі.

Щоб зробити коміт в Git, потрібно виконати наступні кроки:

1. Додати змінені файли до індексу (staging area) за допомогою команди git add або git stage.
2. Створити коміт за допомогою команди git commit -m "message", де message - це короткий опис внесених змін.
3. Завантажити коміт на сервер за допомогою команди git push.

Ці кроки дозволяють зберегти зміни в репозиторії та додати їх до історії розвитку проекту.

1. **В яких статусах може перебувати файл репозиторію?**

В Git файл репозиторію може перебувати в одному з наступних станів:

* Untracked - файл не відслідковується Git. Git не знає про існування цього файлу.
* Modified - файл був змінений, але не був доданий до коміту.
* Staged - файл був доданий до індексу, готовий для коміту.
* Committed - зміни файлу були зафіксовані в коміті.
* Pushed - коміт був відправлений на віддалений репозиторій.
* Pulled - зміни з віддаленого репозиторію були злиті з локальним репозиторієм.
* Conflict - зміни в одному файлі конфліктують з змінами в іншому файлі, які були відправлені на віддалений репозиторій.

Для перевірки статусу файлів можна використати команду "git status".

1. **Які стани репозиторію існують і як вони пов’язані між собою?**

У Git існує три стани репозиторію:

* Змінений (modified) - файл внесено зміни, але ще не додано до індексу. Git відстежує зміни в файлах, але вони ще не зафіксовані в репозиторії.
* Заіндексований (staged) - файл змінено і додано до індексу, але ще не зафіксовано в репозиторії. Використовується для того, щоб позначити, які зміни потрібно внести в наступний коміт.
* Зафіксований (committed) - зміни зафіксовані в репозиторії. Це означає, що файли збережені в локальному репозиторії та можуть бути відновлені у будь-який момент.

Змінений файл може бути доданий до індексу командою git add, після чого файл буде в статусі "заіндексований". Зафіксувати зміни можна командою git commit, яка створює новий коміт з заіндексованими змінами. Коли коміт створено, статус файлу переходить у "зафіксований".

1. **Як можна переміщувати різні вказівники Git? Для яких цілей?**

У Git вказівники (гілки, теги) можна переміщувати за допомогою команди git branch або git tag. Основні цілі переміщення вказівників:

* Переміщення гілки на інший коміт. Це можна зробити, якщо була помилка при виборі коміту для гілки або потрібно змінити місце відгалуження.
* Для переміщення гілки на інший коміт можна використати команду git branch -f <ім'я гілки> <коміт>. Ця команда перенаправить вказівник гілки на вказаний коміт.
* Переміщення тегу на інший коміт. Це можна зробити, якщо була помилка при виборі коміту для тегу або потрібно змінити місце посилання на конкретний коміт.
* Для переміщення тегу на інший коміт можна використати команду git tag -f <ім'я тегу> <коміт>. Ця команда перенаправить вказівник тегу на вказаний коміт.
* Об'єднання гілок або тегів. Це можна зробити, якщо виникла потреба відобразити залежності між відгалуженнями.
* Для об'єднання гілок можна використати команду git merge <ім'я гілки>. Для об'єднання тегів необхідно перемістити вказівник тегу на той самий коміт, що й інший тег, і потім використати команду git merge <ім'я тегу>.

Переміщення вказівників Git може бути корисним для виправлення помилок, відображення залежностей між гілками або тегами, а також для роботи зі старими версіями коду. Однак необхідно пам'ятати, що переміщення вказівників може призвести до втрати даних, тому необхідно бути дуже обережним і ретельно перевіряти результати.

1. **Як зв’язати локальний репозиторій з віддаленим і здійснювати синхронізацію?**

Для зв'язку локального та віддаленого репозиторіїв та здійснення синхронізації використовують команди Git. Основні команди для цього:

* Створення віддаленого репозиторію: якщо віддалений репозиторій ще не існує, можна створити його на веб-сервері Git-хостингу (наприклад, GitHub, GitLab, Bitbucket тощо). Для цього потрібно у веб-інтерфейсі Git-хостингу створити новий репозиторій та отримати його URL.
* Додавання віддаленого репозиторію як remote: потрібно додати віддалений репозиторій як remote у локальний репозиторій за допомогою команди git remote add <ім'я-remote> <URL>.
* Отримання змін з віддаленого репозиторію: для отримання змін з віддаленого репозиторію потрібно використовувати команду git fetch <ім'я-remote>. Ця команда отримує зміни з віддаленого репозиторію, але не об'єднує їх з локальними змінами.
* Об'єднання змін з віддаленого репозиторію з локальними змінами: для того, щоб об'єднати зміни з віддаленого репозиторію з локальними змінами, потрібно використовувати команду git merge <ім'я-remote>/<гілка>.
* Надсилання змін на віддалений репозиторій: для того, щоб надіслати зміни на віддалений репозиторій, потрібно використовувати команду git push <ім'я-remote> <гілка>. Ця команда надсилає локальні зміни на віддалений репозиторій.
* Оновлення віддаленого репозиторію: якщо хтось інший надіслав зміни на віддалений репозиторій після останнього git fetch