Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського" Факультет інформатики та обчислювальної техніки Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 4 з дисципліни
«Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

“Імплементація REST API”

Виконав(ла): Маланічев Д., ІС-13

Перевірив(ла): Бардін В.

Київ 2023

Мета лабораторної роботи – ознайомитися з основами створення REST веб-API та методологією C4 для відображення архітектури системи. Ознайомитися з основами створення ER-діаграм для представлення структури бази даних.

Завдання:

Практична частина:

1. Використовуючи архітектуру розроблену в попередній роботі

Імплементувати REST веб-API, використавши N Layered архітектуру.

2. Покрити модульними та інтеграційними тестами основні

компоненти рішення.

3. Експортувати розроблене API до Postman.

Документація:

1. Підготувати документацію(звіт до ЛР), яка включатиме опис веб-API, а також структуру бази даних з урахуванням ER-діаграми.

Варіант завдання:

9	Реєстратура лікарні. Запис до лікарів на прийом	1. Реєстратура надає дані стосовно наявності лікарів та розкладу прийому хворих. 2. Хворим можливо записатись на прийом до лікаря, якщо є вільний час у розкладі лікаря. 3. В реєстратурі ведуться картки відвідування хворими лікарні, в які записується час відвідання лікаря, діагноз та лікар, що його поставив. **Функціональні вимоги:** 1. Ведення картотеки лікарні; 2. Керування прийомами хворих у лікарів.

Хід роботи

В ході виконання даної лабораторної роботи було реалізовано Web API за шаблоном розробленим у попередній лабораторній роботи (номер 3).

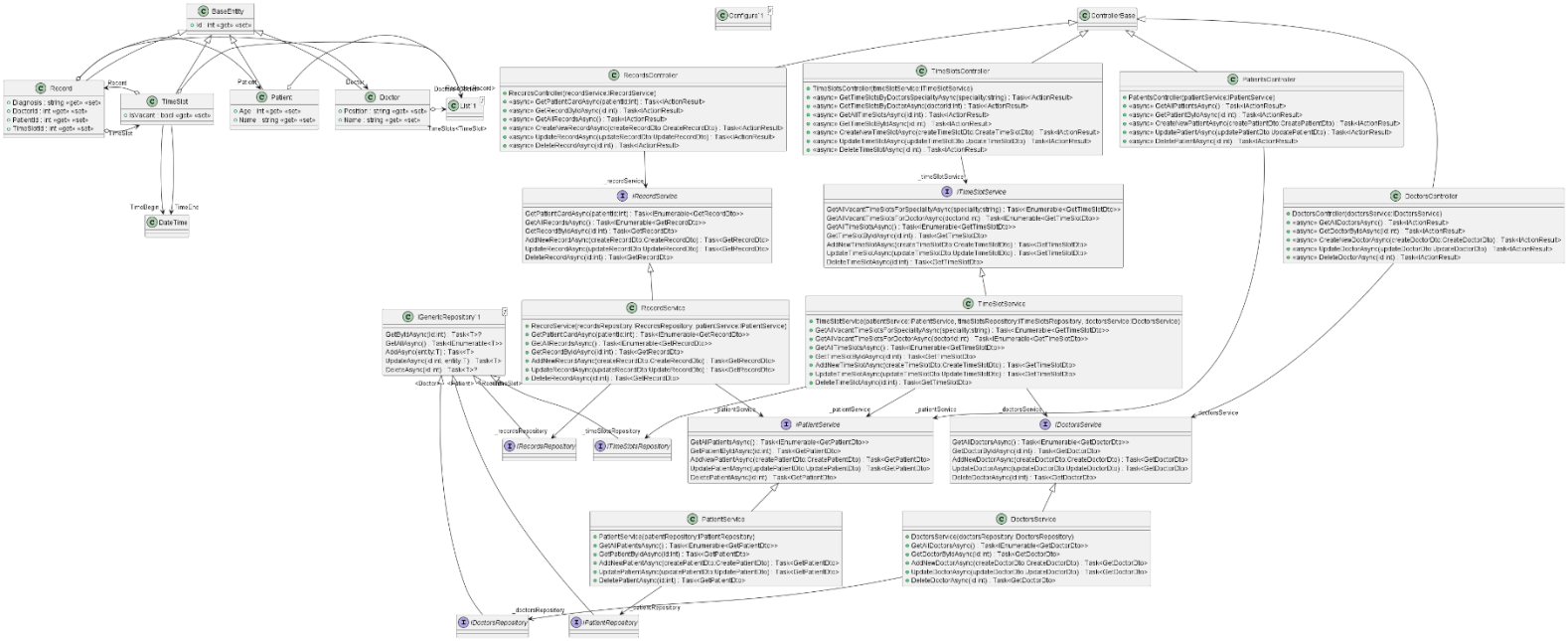

Рис. 1 – Code Diagram

Застосунок був розбитий на 3 логічні шари – BLL, DAL Presentation Layer (Web API). Рівень BLL в свою чергу поділяється на низку сервісів, що відповідають за відповідну сутність.

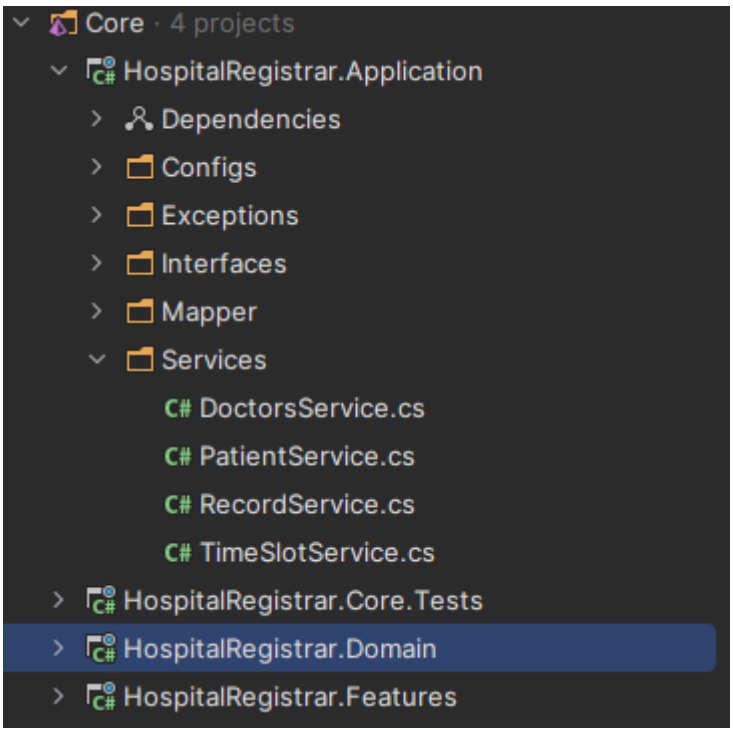

Рис. 2 – сервіси BLL рівня

Найцікавішою частиною BLL рівня є метод пошуку вільних таймслотів за рядом параметрів. Для цього було реалізовано Specification Pattern. Для цього реалізуємо дженерік інтерфейс ISpecifications з методом Criteria, який буде повертати true, якщо даний таймслот задовольняє цьому критерію.

```
public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
}
```

І класи специфікації, які реалізують даний інтерфейс. Наприклад, DoctorNameSpecification, який перевіряю таймслот на те, чи приймає лікар з відповідним іменем:

```
public class DoctorNameSpecification : ISpecification<TimeSlot>
{
    private readonly string _name;

    public DoctorNameSpecification(string name)
    {
        _name = name;
    }

    public Expression<Func<TimeSlot, bool>> Criteria =>
        timeSlot => timeSlot.Doctors.Any(d => d.Name == _name);
}
```

Далі було творено окремий дженерік сервіс, який виконує сортування та пагінацію над вказаним ресурсом. Таким чином цей сервіс може бути перевикористаним і для інший сутностей:

```
public class DataQueryService<T, TDto>
    : IDataQueryingService<T, TDto>
    where T : BaseEntity
{
    private readonly IQueryingRepository<T> _queryingRepository;
    private readonly IMapper _mapper;

    public DataQueryService(IQueryingRepository<T> queryingRepository,
IMapper mapper)
    {
        _queryingRepository = queryingRepository;
        _mapper = mapper;
    }

    public async Task<GenericPagingDto<TDto>>
GetPagedEntityByCriteriaAsync(QueryPaginationCriteria<T>
queryPaginationCriteria, params ISpecification<T>[] specifications)
    {
        var predicate = PredicateBuilder.New<T>(true);

        predicate = specifications.Aggregate(predicate, (current,
specification) => current.And(specification.Criteria));

        var pagedResult = await
_queryingRepository.GetItemsByPredicate(predicate,
queryPaginationCriteria.SortBy, queryPaginationCriteria.IsDescending)
```

```
            .ApplyPagingAsync(queryPaginationCriteria);

        return new GenericPagingDto<TDto>
        {
            CurrentPage = queryPaginationCriteria.CurrentPage,
            Entities = _mapper.Map<List<TDto>>(pagedResult.Items),
            TotalPages = pagedResult.TotalPages,
        };
    }
}
```

Далі з цих критеріїв створюємо єдиний предикат, за допомогою якого і відбувається пошук потрібних таймслотів в БД:

```
public async Task<GenericPagingDto<GetTimeSlotDto>>
GetTimeSlotsBySpecificationsAsync(AvailablePagedTimeSlotsByCriteriaRequestDto
criteriaRequestDto)
{
    var specifications = new List<ISpecification<TimeSlot>>();

    if (!string.IsNullOrEmpty(criteriaRequestDto.Specialty))
    {
        specifications.Add(new
DoctorSpecialtySpecification(criteriaRequestDto.Specialty));
    }

    if (!string.IsNullOrEmpty(criteriaRequestDto.DoctorName))
    {
         specifications.Add(new
DoctorNameSpecification(criteriaRequestDto.DoctorName));
    }

    if (criteriaRequestDto.Date.HasValue)
    {
        specifications.Add(new
TimeSlotDateSpecification(criteriaRequestDto.Date.Value));
    }

    var pagingParams = new QueryPaginationCriteria<TimeSlot>
    {
        CurrentPage = criteriaRequestDto.CurrentPage ?? 1,
        IsDescending = criteriaRequestDto.IsDescending,
        ItemsOnPage = 10,
        SortBy = timeSlot => timeSlot.TimeBegin,
    };

    var timeSlots = await
_dataQueryingService.GetPagedEntityByCriteriaAsync(pagingParams,
specifications.ToArray());

    return timeSlots;
}
```

DAL рівень побудований за даною ER-діаграмою:

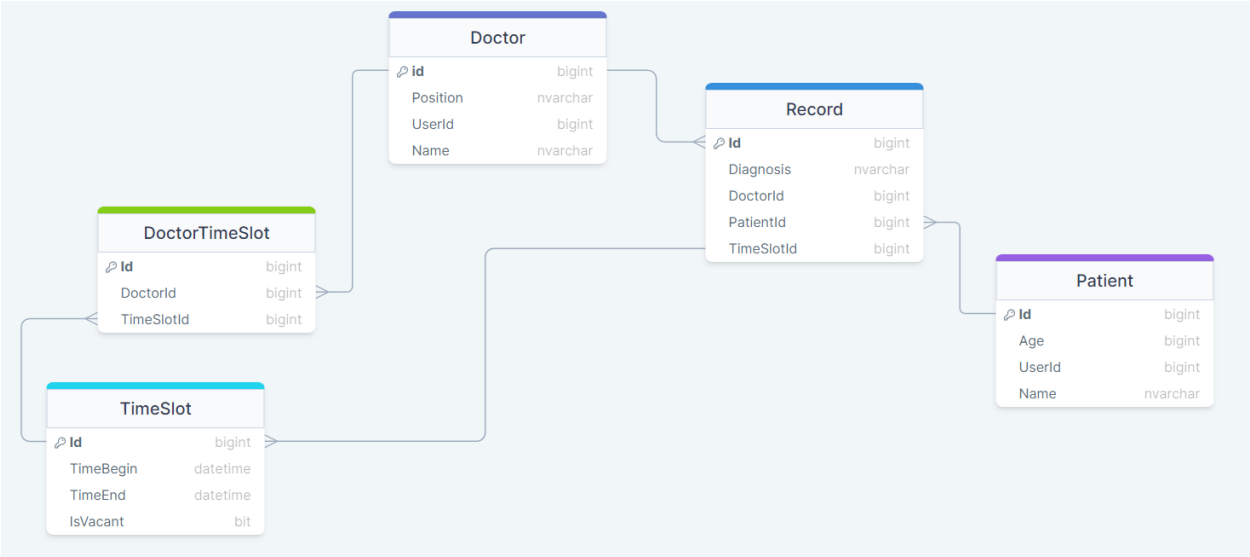

Рис. 3 – ER-діаграма DAL рівня

При виконанні цього рівня був застосований Code-first підхід, тобто написання класів-моделей майбутніх таблиць, які потім будуть створені в Базі даних, за допомогою міграцій EF Core. Наприклад, ось клас-модель сутності Доктор:

```
public class Doctor : BaseEntity
{
    public string Position { get; set; } = default!;

    public string Name { get; set; } = default!;

    public virtual List<TimeSlot> TimeSlots { get; set; } = default!;

    public virtual List<Record> Records { get; set; } = default!;
}
```

Далі, доступ до даних з БД відбувається за допомогою репозиторіїв на EF Core. Для зменшення об’єму коду репозиторії були реалізовані за допомогою Generic Repository Pattern, де всі репозиторії наслідуються від одного дженерік класу, який реалізовує CRUD операції і за потреби розширюють його.

Наприклад, таким розширенням є метод пошуку тих же таймслотів за створеним предикатом:

```
public async Task<IEnumerable<TimeSlot>>
GetAvailableTimeSlotsByPredicateAsync(ExpressionStarter<TimeSlot>
predicateBuilder)
{
    return await _context.TimeSlots.AsQueryable()
        .Where(predicateBuilder)
        .ToListAsync();
}
```

Аналогічно на рівні API маємо 5 контролерів – Doctors, Patients, Records та TimeSlots, які відповідають за ресурси та AccountController які відповідає за реєстрування нових користувачів.

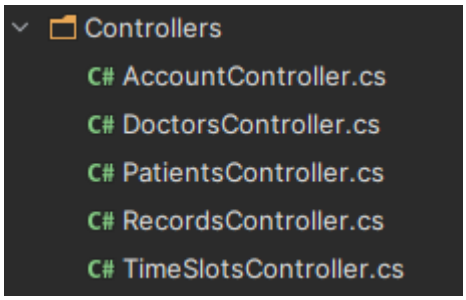

Рис. 4 – контролери API рівня

Всі сервіси та репозиторії було протестовано. Для сервісів BLL рівня написано юніт тести, а для репозиторіїв DAL рівня – інтеграційні, з використанням InMemoryDB.

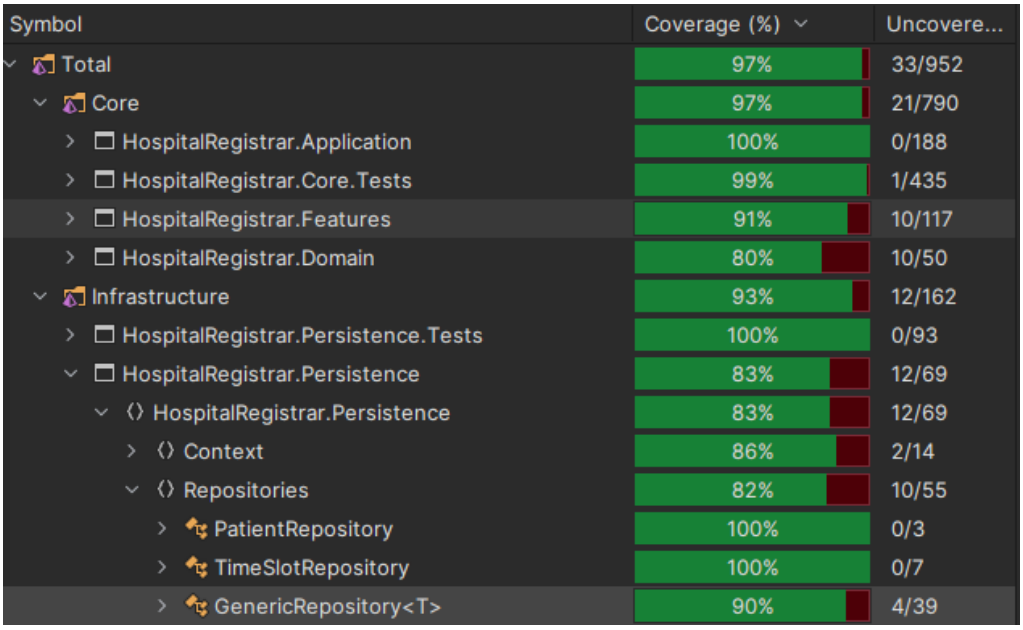

Рис. 5 – test coverage

Створений API було експортовано в Postman:

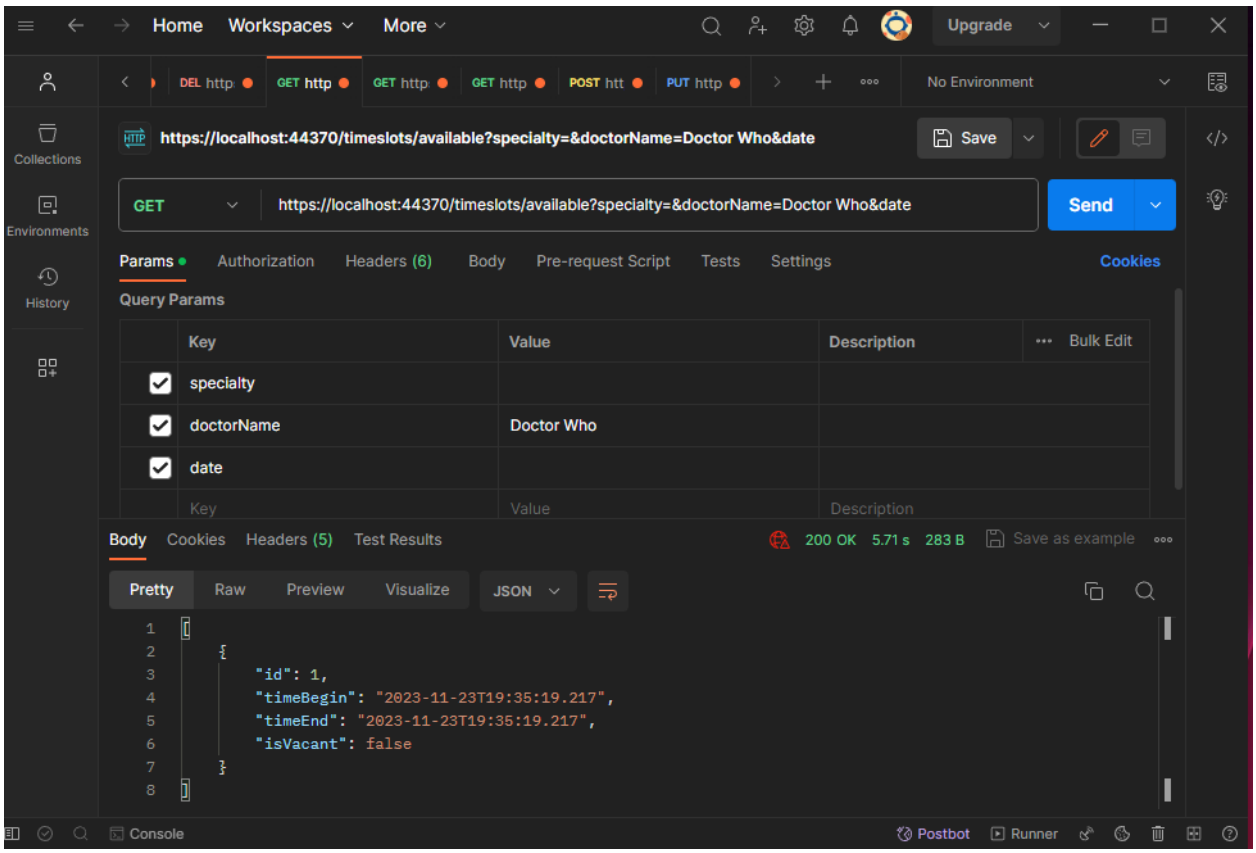

Рис. 6 – експортований в Postman API

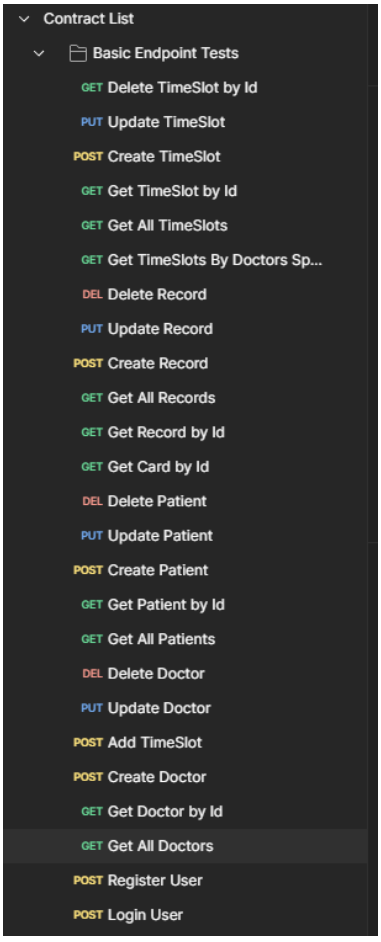

Рис. 7 – колекція доданих до Postman ендпоінтів

Висновок:

Отже, в результаті виконання даної лабораторної роботи було створено повнофункціональний API за набором діаграм з лабораторної роботи №3 (створених за методологією C4), за заданою предметною областю, в даному випадку – Реєстратура лікарні. Запис до лікарів на прийом. Таким чином, було реалізовано базу даних, створено шар бізнес логіки, з простими CRUD операціями, фільтрацією за допомогою Specification Pattern та пагінацією. Розроблена логіка аутентифікації та авторизації. В решті було реалізовано шар представлень (власне API) та експортовано отримані ендпоінти в Postman.

Посилання на GitHub: https://github.com/DenysMalanichev/HospitalRegistrar