Руководство для технического специалиста. Код.

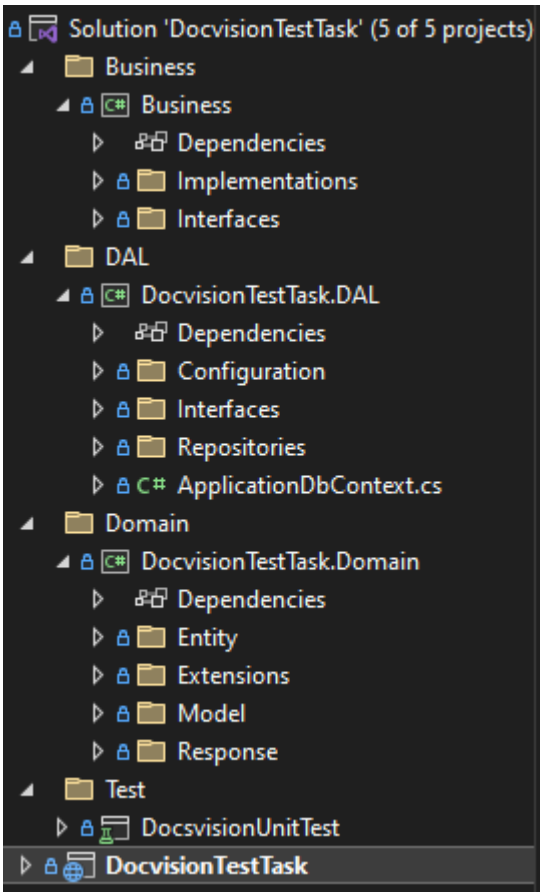
Язык разработки: .net 7.0, 11 версия языка c#.

Фреймворки:

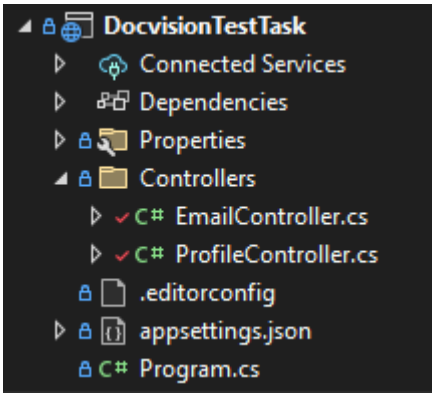
1. AspNetCore - фундамент для разработки REST API
2. EntityFrameworkCore 7.0.3 в связке с EntityFrameworkCore.SqlServer 7.0.3 - Позволяет использовать подход Code First при разработке приложения.
3. Swashbuckle.AspNetCore - Swagger для автоматизации написания технической документации к API.
4. xunit + Moq + MockQueryable.EntityFrameworkCore + MockQueryable.Moq - Для тестирования.

Структура проекта.
Слои:

1. Domain - хранение архитектур для entities, extensions, viewmodels, models, и т.д.
2. Business - хранение архитектуры логики работы приложения.
3. DAL - взаимодействие с базой данных.
4. Test - Тестирование приложения.



Описание приложения Docsvision TestTask.



В приложении реализованы три контроллера :

1. **EmailController - /email**

```
[Route("api/[controller]")]
[ApiController]
5 references
public class EmailController : ControllerBase
{
    private readonly IEmailService _emailService;
    2 references
    public EmailController(IEmailService emailService)
    {
        _emailService = emailService;
    }

    //Принимаем json и отправляем в inbox
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(EmailModel))]
    [HttpPost]
    2 references
    public async Task<ActionResult<EmailModel>> CreateEmail(EmailModel email)
    {
        if (email != null)
        {
            if (ModelState.IsValid)
            {
                var resposne = await _emailService.CreateNewInBox(email);
                if (resposne.statusCode == Domain.Entity.StatusCode.ok)
                {
                    return Ok($"{resposne.Description}");
                }
            }
            return BadRequest($"Ошибка при отправке письма\n{ModelState}");
        }
        return BadRequest("Ошибка при отправке письма. Пустое сообщение.");
    }
}
```

Описывает механизм приема писем при обращении к API через http POST запрос.

Валидация модели происходит согласно Required атрибутам модели EmailModel.

```
15 references
public class EmailModel
{
    [StringLength(300, ErrorMessage = "Длина темы письма не должна превышать 300 символов")]
    7 references
    public string emailSubject { get; set; } = "Пустая тема письма";
    7 references
    public DateTime emailDate { get; set; } = DateTime.MinValue;
    [StringLength(50, ErrorMessage = "Превышен лимит, ФИО не должно превышать 50 символов")]
    7 references
    public string emailFrom { get; set; } = "Отправитель не указан";
    [StringLength(50, ErrorMessage = "Превышен лимит, ФИО не должно превышать 50 символов")]
    12 references
    public string emailTo { get; set; } = "Получатель не указан";
    [StringLength(5000, ErrorMessage = "Превышен лимит, текст письма не должен превышать 1500 символов")]
    7 references
    public string emailBody { get; set; } = "Пустое письмо";
}
```

Передает “сырой” материал в сервис для дальнейшей обработки.
Возвращает ответ клиенту.
Информация для ответа формируется на основание response объекта, содержащего информацию согласно описанию класса BaseResponse, описанного в слое Domain.

```
12 references
public class BaseResponse<T> : IBaseResponse<T>
{
    14 references
    public string Description { get; set; }
    21 references
    public StatusCode statusCode { get; set; }
    9 references
    public T Data { get; set; }
}
```

2. **ProfileController /profile**

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class ProfileController : Controller
{
    private readonly IProfileService _profileService;

    0 references
    public ProfileController(IProfileService profileService)
    {
        _profileService = profileService;
    }

    //Получаем список всех записей таблицы Profiles
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(List<ProfileModel>))]
    [HttpGet]
    0 references
    public IActionResult GetProfiles()
    {
        var response = _profileService.GetAllProfiles();
        if (response.statusCode == Domain.Entity.StatusCode.ok)
        {
            return Ok(response.ToJson());
        }
        return BadRequest("Не удалось получить список профилей");
    }
}
```

Описывает механизм передачи данных для клиентского приложения при обращении к API через http Get запрос.
Передает на клиент информацию о существующих сотрудниках. Имя, Фамилия, id согласно описанию модели.

```
8 references
public class ProfileModel
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string firstName { get; set; }
    1 reference
    public string lastName { get; set; }
}
```

Информация для ответа формируется на основание response объекта, содержащего информацию согласно описанию класса BaseResponse, описанного в слое Domain.

```
12 references
public class BaseResponse<T> : IBaseResponse<T>
{
    14 references
    public string Description { get; set; }
    21 references
    public StatusCode statusCode { get; set; }
    9 references
    public T Data { get; set; }
}
```

Описание внедрение зависимостей описано в классе Program.cs

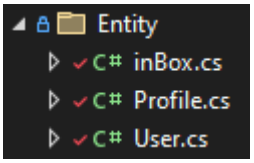
```
// DI repository
builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IProfileRepository, ProfileRepository>();
builder.Services.AddScoped<IInBoxRepository, InBoxRepository>();

// DI services
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IProfileService, ProfileService>();
builder.Services.AddScoped<IEmailService, EmailService>();
```

Описание инициализации dbcontext описано в классе Program.cs

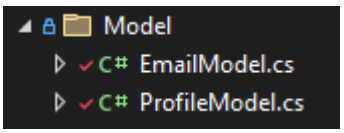
```
// БД
using var scope = app.Services.CreateScope();
ApplicationDbContext dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
dbContext.Database.EnsureCreated(); // для формирования исходных данных и валидации существования БД с указанными настройками.
```

Описание слоя Domain для Docsvision TestTask.
Описание: Механизмы расширений, описание сущностей, описание моделей.
Сущность: библиотека .dll

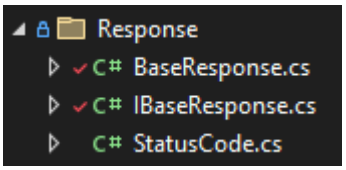
1. Описание сущностей entity для взаимодействия с БД



2. Описание моделей для контроллеров

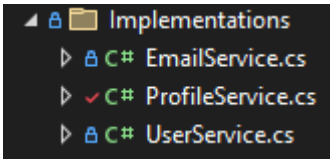


3. Самописный класс BaseResponse который используется для валидации состояния бизнес-процесса внутри приложения



Описание слоя Domain для Docsvision TestTask.
Описание: архитектура и описание бизнес-логики приложения.
Сущность - библиотека .dll

Описание **сервисов** приложения, каталог Implementations:



EmailService сервис, который формирует и подготавливает полученное письмо для попытки формирования запроса в БД.
В нашем случае происходит формирования сущности inBox:
Далее следует попытка обращения к БД и формирование объекта класса BaseResponse для возврата на контроллер EmailController.

```
bool request = await _InBoxRepository.Create(newInbox);
if (!request)
{
    baseResponse.Data = newInbox;
    baseResponse.statusCode = StatusCode.internalServiceError;
    baseResponse.Description = $"Не удалось отправить письмо адресованное пользователю [ {newInbox.emailTo} ].\n";
    return baseResponse;
}
baseResponse.Data = newInbox;
baseResponse.Description = $"Письмо адресованное пользователю [ {newInbox.emailTo} ] успешно доставлено.\n{_userId.Description}";
baseResponse.statusCode = StatusCode.ok;
return baseResponse;
```

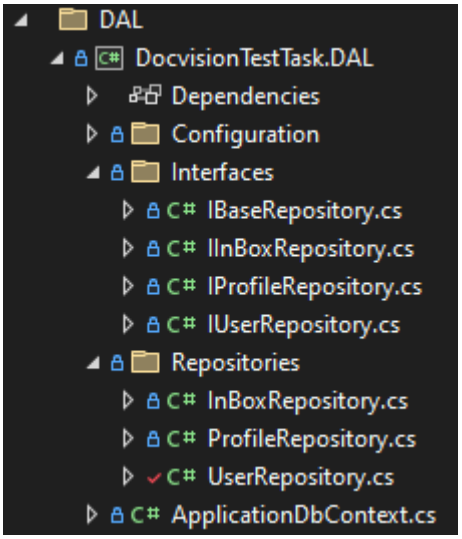
PofileService сервис, который формирует и подготавливает объект модели ProfileModel на основание ответа от базы данных. Формирует объект класса BaseResponse для возврата на контроллер ProfileController.

```
2 references
public IBaseResponse<List<ProfileModel>> GetAllProfiles()
{
    var baseResponse = new BaseResponse<List<ProfileModel>>();
    try
    {
        var ProfileCollection = _profileRepository.Select();
        List<ProfileModel> result = new List<ProfileModel>();
        foreach (var Profile in ProfileCollection)
        {
            result.Add(new ProfileModel {
                firstName = Profile.firstName,
                lastName = Profile.lastName,
                Id= Profile.userId,
            });
        }
        baseResponse.Data = result;
        baseResponse.statusCode = StatusCode.ok;
        baseResponse.Description = "A try to get all profiles: successful";
        return baseResponse;
    }
    catch (Exception exception)
    {
        return new BaseResponse<List<ProfileModel>>()
        {
            Description = exception.Message,
            statusCode = StatusCode.internalServiceError
        };
    }
}
```

UserService сервис, который содержит метод GetUserByFNameLName, который мы используем при работе бизнес - процесса формирования inBox объекта , а именно поиск и валидация пользователя по “ИМЯ” и “ФАМИЛИЯ” . Возвращает baseResponse.statusCode = StatusCode.ok если пользователь найден и baseResponse.statusCode = StatusCode.InternalServiceError если нет - что в будет являться индикатором при формирование данных для inBox объекта в сервисе EmailService.

```
public  async Task<IBaseResponse<User>> GetUserByFNameLame(string FNameLName)
{
    var baseResponse = new BaseResponse<User>();
    try
    {
        var User = await _userRepository.GetUserByFNameLName(FNameLName);
        baseResponse.Data = User;
        baseResponse.statusCode = StatusCode.ok;
        baseResponse.Description = "Получатель найден.\n";
        return baseResponse;
    }
    catch (Exception exception)
    {
        return new BaseResponse<User>()
        {
            Description = $"Не удалось найти получателя по причине: {exception.Message}\nПисьмо было отправлено на общий почтовый ящик.\n",
            statusCode = StatusCode.internalServiceError
        };
    }
}
```

Описание слоя DAL для Docsvision TestTask.
Описание: архитектура и описание взаимодействия с БД.
Сущность - библиотека .dll



InBoxRepository описание взаимодействия между контекстом БД и Entity Data Model inBox.

```
public readonly ApplicationDbContext _dbContext;
0 references
public InBoxRepository(ApplicationDbContext dbContext)
{
    _dbContext = dbContext;
}

3 references
public async Task<bool> Create(inBox entity)
{
    await _dbContext.inBox.AddAsync(entity);
    await _dbContext.SaveChangesAsync();
    return true;
}
```

ProfileRepository описание взаимодействия между контекстом БД и Entity Data Model Profile.

```
public readonly ApplicationDbContext _dbContext;
0 references
public ProfileRepository(ApplicationDbContext dbContext)
{
    _dbContext = dbContext;
}

7 references
public IEnumerable<Profile> Select()
{
    return _dbContext.Profile.Include(x => x.User).ToList();
}
```

UserRepository описание взаимодействия между БД и контекстом БД и Entity Data Model User. В контексте нашей задачи тут описан механизм поиска пользователя по имени и фамилии.

```
public readonly ApplicationDbContext _dbContext;
0 references
public UserRepository(ApplicationDbContext dbContext)
{
    _dbContext = dbContext;
}

6 references
public async Task<User> GetUserByFNameLName(string FnameLname)
{
    if (FnameLname == "") { throw new Exception("Адресат не указан"); }
    if (FnameLname.Split(" ").Count() > 1)
    {
        string Fname = FnameLname.Split(" ")[1];
        string LName = FnameLname.Split(" ")[0];
        var userToFind =  await _dbContext.Users.FirstOrDefaultAsync(x => x.Profile.firstName.Contains(Fname) && x.Profile.lastName.Contains(LName));
        if (userToFind == null)
        {
            userToFind = await _dbContext.Users.FirstOrDefaultAsync(x => x.Profile.firstName.Contains(LName) && x.Profile.lastName.Contains(Fname));
            if (userToFind == null)
            {
                throw new Exception($" [Адресат с фамилией {LName} и именем {Fname} не найден] ");
            }
        }
        return userToFind;
    }
    else
    {
        throw new Exception(" [Невозможно определить адресата только по имени, или только по фамилии] ");
    }

}
```