Софтуерни технологии/Технология на софтуерното производство

ДОКУМЕНТАЦИЯ

Явор Чамов | СИТ 1 а) | №21621577

Съдържание.

Глава 1. Увод.

Описание и структура на проекта.

Настоящият проект представлява онлайн магазин за електроника, създаден с използването на C# за разработката на backend и Angular за frontend разработката. Целта на проекта е да предостави потребителите с широка гама от електронни устройства, включително смартфони, лаптопи, телевизори, гейминг аксесоари и други.

Структура на проекта:

1. Backend:
    a. Използване на C# за разработка на backend функционалността.
    b. Използване на ASP.NET Core за създаване на уеб приложението.
    c. Реализиране на RESTful API за комуникация между клиентската страна и сървърната част.
    d. Използване на Entity Framework Core за управление на базата данни и обекто-релационното отображение.
    e. Автентикация и авторизация с помощта на JWT (JSON Web Tokens).

2. Frontend:
    a. Използване на Angular за създаване на потребителския интерфейс.
    b. Разработване на респонсивен и модерен дизайн, който е приятен за потребителите. Взаимодействие с backend чрез използване на HTTP клиент за изпращане на заявки към RESTful API.
    c. Използване на Angular компоненти, рутиране и форми за създаване на интерактивни уеб страници.
    d. Интеграция на различни Angular модули и библиотеки за по-добра функционалност и удобство за потребителите.

3. База данни:
    a. Използване на релационна база данни (SQL Server) за съхранение на данни за продуктите, потребителите и поръчките.
    b. Дизайн на базата данни със специално внимание към оптимизацията и скалируемостта.

4. Система за управление на версиите:
    a. Използване на система за управление на версиите (Git) за контрол на изходния код.
    b. Разделяне на кода на backend и frontend в отделни репозитории за по-добро управление и поддръжка.

Този проект цели да предложи на потребителите интуитивен и функционален интерфейс за пазаруване на електроника, докато осигурява ефективно и сигурно управление на данните чрез използване на съвременни технологии и най-добри практики в софтуерното разработване.

Цел и задачи на разработката.

Целта на настоящата разработка е да създаде функционален и интуитивен онлайн магазин за електроника, който да предоставя на потребителите бърз и удобен начин за пазаруване на разнообразни електронни устройства. Проектът се стреми да предложи надежден и сигурен софтуерен продукт, който да отговаря на нуждите на потребителите за качествено онлайн пазаруване.

Задачи на разработката:

1. Създаване на уеб приложение с използване на C# за backend разработка и Angular за frontend разработка.
2. Разработване на RESTful API за взаимодействие между клиентската и сървърната части на приложението.
3. Изграждане на база данни за съхранение на информацията за продуктите, потребителите и поръчките, като се отдава внимание на оптимизацията и скалируемостта на базата данни.
4. Използване на сигурни механизми за автентикация и авторизация, като се използва JWT (JSON Web Tokens) за управление на потребителските сесии.
5. Разработване на интуитивен и атрактивен потребителски интерфейс, който да предостави приятен опит на потребителите при използването на магазина.
6. Интегриране на различни Angular модули и библиотеки за подобряване на функционалността и удобството на потребителите.
7. Тестване на приложението за осигуряване на качествен и стабилен софтуерен продукт.
8. Разработване на система за управление на версиите, която да позволи контрол на изходния код.

Посредством изпълнение на горепосочените задачи, проектът се стреми да достигне своята цел - да предложи на потребителите удобен, сигурен и функционален онлайн магазин за електроника, който да отговаря на съвременните изисквания и очаквания.

Структура на документацията.

Документацията за проекта е структурирана в няколко глави, които представят последователно различни аспекти на разработката. Глава 1, "Увод", включва описание на проекта, неговата идея и целите на разработката. Глава 2, "Преглед на предметната област", представя основни дефиниции, концепции и алгоритми, които ще бъдат използвани, както и анализ на проблемите и подходите за тяхното решаване. В глава 3, "Проектиране", се разглежда общата структура на проекта и се представят диаграми и блок схеми, които илюстрират структурата и поведението на системата. Глава 4, "Реализация, тестване", се фокусира върху реализацията на класовете, включително важни моменти и кодови фрагменти, както и планирането и създаването на тестови сценарии. Заключителната глава 5, "Заключение", представя обобщение на постигнатите цели и предлага насоки за бъдещо развитие и усъвършенстване на проекта.

Глава 2. Преглед на предметната област.

Основни дефинции и концепции.

Модели в системата.

Модели в системата представляват основните единици, които съхраняват и управляват данните във онлайн магазина за електроника. В следния параграф са описани моделите и техните атрибути:

1. Categories (Категории):
    a. name: Име на категорията.
    b. description: Описание на категорията.
    c. imageUrl: URL адрес към изображение, асоциирано с категорията.
    d. isDeleted: Флаг, който указва дали категорията е изтрита или не.
2. ProductImages (Изображения на продукти):
    a. imageUrl: URL адрес към изображение на продукта.
    b. productId: Идентификатор на продукта, към който принадлежи изображението.
    c. isDeleted: Флаг, който указва дали изображението е изтрито или не.
3. ProductInventory (Наличност на продукти):
    a. price: Цена на продукта.
    b. discountPrice: Отстъпка върху цената на продукта (ако има такава).
    c. stockquantity: Налично количество на продукта.
    d. importdate: Дата на внасяне на продукта в инвентара.
    e. productId: Идентификатор на продукта.
    f. isDeleted: Флаг, който указва дали записът за наличност на продукта е изтрит или не.
4. Products (Продукти):
    a. name: Име на продукта.
    b. description: Подробно описание на продукта.
    c. shortDescription: Кратко описание на продукта.
    d. imageUrl: URL адрес към изображение на продукта.
    e. sku: Уникален код за идентификация на продукта.
    f. productType: Тип на продукта (например смартфон, лаптоп и т.н.). categoryId:
    g. Идентификатор на категорията, към която принадлежи продукта.
    h. isDeleted: Флаг, който указва дали продуктът е изтрит или не.
    i. isFeatured: Флаг, който указва дали продуктът е включен в промоционалния списък.
5. ProductTag (Маркери на продукти):
    a. productId: Идентификатор на продукта.

    b. TagsId: Идентификатор на маркера.
6. SaleItems (Елементи на продажби):
    a. quantity sold: Количество продадени продукти.
    b. productId: Идентификатор на продукта.
    c. saleid: Идентификатор на продажбата.
    d. isDeleted: Флаг, който указва дали записът за продажба е изтрит или не.
7. Sales (Продажби): Модел за съхранение на информацията за продажбите.
8. Tags (Маркери): Модел за съхранение на маркерите, които могат да бъдат асоциирани с продуктите.
9. Users (Потребители): Модел за съхранение на информацията за потребителите на системата, включително данни за аутентикация и авторизация.

Тези модели образуват основата на системата за управление на данните в онлайн магазина за електроника и позволяват ефективно управление и представяне на информацията за продуктите, категориите, продажбите и потребителите.

```
namespace tuvarna_ecommerce_system.Models.Entities
{
    public class Product : BaseEntity
    {
        [Required]
        [StringLength(128)]
        public string Name { get; set; }
        [Required]
        [StringLength(1024)]
        public string Description { get; set; }
        [Required]
        [StringLength(256)]
        public string ShortDescription { get; set; }
        [Required]
        public string ImageUrl { get; set; }
        [Required]
        [StringLength(6)]
        public string Sku { get; set; }
        [Required]
        public ProductTypeEnum ProductType { get; set; }
        public int? CategoryId { get; set; }
        public Category Category { get; set; }
        public ICollection<Tag> Tags { get; set; } = [];
        public bool IsFeatured { get; set; } = false;
        public ICollection<ProductImage> AdditionalImages { get; set; }
= new List<ProductImage>();
```

```
        public ICollection<ProductInventory> Inventories { get; set; }
= new List<ProductInventory>();
        public ICollection<SaleItem> SaleItems { get; set; } = new
List<SaleItem>();
    }
}
```

Модели за пренос на данни.

За да се осигури ефективно и безопасно предаване на данни между различните компоненти на системата, е необходимо да използвате модели за пренос на данни (DTO - Data Transfer Objects). В следния параграф са описани някои от тези модели:

1. **CategoryDTO (DTO за категории)**:
    - **name**: Име на категорията.
    - **description**: Описание на категорията.
    - **imageUrl**: URL адрес към изображение, асоциирано с категорията.

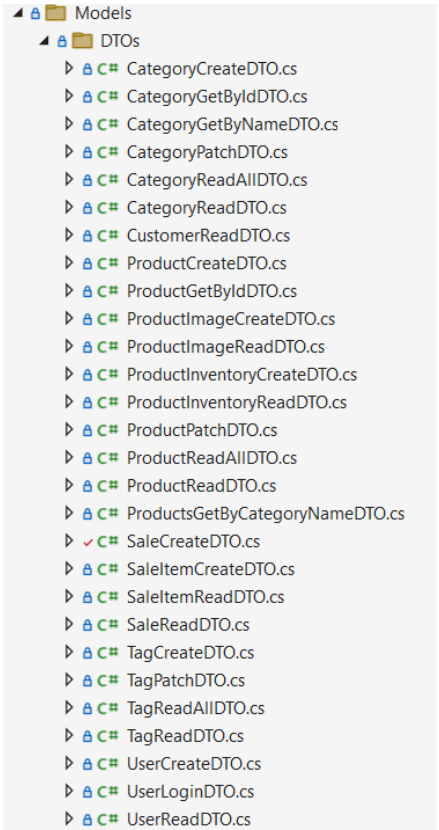
2. **ProductDTO (DTO за продукти)**:
    - **name**: Име на продукта.
    - **description**: Подробно описание на продукта.
    - **shortDescription**: Кратко описание на продукта.
    - **imageUrl**: URL адрес към изображение на продукта.
    - **sku**: Уникален код за идентификация на продукта.
    - **productType**: Тип на продукта.
    - **categoryId**: Идентификатор на категорията, към която принадлежи продукта.
3. **ProductInventoryDTO (DTO за наличност на продукти)**:
    - **price**: Цена на продукта.
    - **discountPrice**: Отстъпка върху цената на продукта (ако има такава).
    - **stockquantity**: Налично количество на продукта.
4. **ProductTagDTO (DTO за маркери на продукти)**:
    - **tagName**: Име на маркера.
5. **UserDTO (DTO за потребители)**:
    - **username**: Потребителско име.
    - **email**: Имейл адрес на потребителя.
    - **role**: Роля на потребителя (например администратор, обикновен потребител)

Тези DTO модели се използват за предаване на данни между различните части на приложението, като осигуряват ясно определение на данните, които се предават, и помагат за избягване на излишна информация или нежелани зависимости между компонентите на системата. Чрез използването на DTO модели, вие можете да контролирате и оптимизирате комуникацията между различните части на вашия онлайн магазин за електроника.

```
namespace tuvarna_ecommerce_system.Models.DTOs
{
    public class ProductCreateDTO : IValidatableObject
    {
        [Required(ErrorMessage = "Name is required.")]
        [StringLength(128, ErrorMessage = "Name must be less than 128
characters.")]
        public string Name { get; set; }
        [Required(ErrorMessage = "Description is required.")]
        [StringLength(1024, ErrorMessage = "Description must be less
than 1024 characters.")]
        public string Description { get; set; }

        [Required(ErrorMessage = "Short description is required.")]
```

```
        [StringLength(256, ErrorMessage = "Short description must be
less than 256 characters.")]
        public string ShortDescription { get; set; }
        [Required(ErrorMessage = "Image url is required.")]
        public string ImageUrl { get; set; }
        [Required(ErrorMessage = "Product type is required.")]
        public string ProductType { get; set; }
        public string? CategoryName { get; set; }
        public List<TagCreateDTO> Tags { get; set; } = new
List<TagCreateDTO>();
        public List<ProductImageCreateDTO> Images { get; set; } = new
List<ProductImageCreateDTO>();
        public virtual IEnumerable<ValidationResult>
Validate(ValidationContext validationContext)
        {
            var duplicateTags = Tags.GroupBy(t => t.Name).Where(g =>
g.Count() > 1).Select(g => g.Key);
            foreach (var duplicateTag in duplicateTags)
            {
                yield return new ValidationResult($"Duplicate tag:
{duplicateTag}", new[] { nameof(Tags) });
            }
        }
    }
}
```



Настройката и конфигурацията на средата, в която приложението се изпълнява

Програмният файл Program.cs в ASP.NET Core приложението има ключова роля в настройката и конфигурацията на средата, в която приложението се изпълнява, както и в управлението на жизнения цикъл на приложението. В следния параграф ще бъде описано съдържанието и функционалността на файла Program.cs:

Файлът Program.cs започва със стандартния using namespace и включва нужните пространства от имена за работата с ASP.NET Core и другите използвани библиотеки.

След това, чрез методът WebApplication.CreateBuilder(args) се създава обект от тип WebApplicationBuilder, който се използва за конфигурирането на средата на приложението.

```
builder.Services.AddDbContext<EcommerceDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Dev
EcommerceDb")), ServiceLifetime.Scoped);
```

С този код се добавя контекст на база данни (EcommerceDbContext), който се свързва с конкретната база данни (в случая с SQL Server) чрез предоставената в конфигурацията връзка.

```
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

Тези методи добавят необходимите услуги за работа с контролерите, API Explorer и Swagger за документиране на API.

```
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ICategoryService, CategoryService>();
```

Тези редове добавят в контейнера на услугите хранилище и услуги, които ще бъдат използвани от приложението. Това включва хранилища и услуги за категории, продукти, продажби, потребители и други.

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});
```

Този ред конфигурира опциите за поведението на API и в настоящия случай се настройва така, че да се подтиска автоматичното моделиране на състоянието на модела.

След добавянето на всички необходими услуги и настройки, приложението се създава с помощта на метода builder.Build(). После, за да се осигури миграцията на базата данни, ако е необходимо, се използва методът dbContext.Database.Migrate().

Извиква се методът await InitializeAdminUser(app.Services), който инициализира администраторски потребител, ако той все още не съществува в системата.

Накрая, се конфигурира HTTP заявките, като се активират middleware за аутентикация, авторизация и обработка на грешки, както и Swagger UI в режим на разработка. Ако приложението се изпълнява в режим на разработка (app.Environment.IsDevelopment()), се активира Swagger UI.

Програмният файл Program.cs е отговорен за конфигурирането на приложението и подготовката му за стартиране. Той предоставя възможност за настройка на услуги, middleware и други настройки, които са необходими за правилното функциониране на ASP.NET Core приложението.

Услуги.

Слоят на услугите (Service layer) в приложението играе ключова роля в управлението на бизнес логиката и извършването на операции, свързани с данните. В следния параграф ще бъде описана функционалността и ролята на слоя на услугите:

Слойът на услугите включва класове, които предоставят високо ниво на абстракция върху операциите, свързани с данните, и предоставят интерфейс за взаимодействие между контролерите и хранилищата. Те изпълняват бизнес логиката на приложението и управляват транзакциите и валидацията на данните, преди те да бъдат записани в базата данни.

В следния параграф са описани някои от основните функционалности и роли на слоя на услугите в приложението:

1. **Управление на бизнес логиката**: Слоят на услугите е отговорен за изпълнението на бизнес логиката на приложението. Това включва валидация на данните, изпълнение на бизнес правила и обработка на специфични сценарии за приложението.

2. **Комуникация с хранилищата**: Услугите взаимодействат с хранилището за достъп до данни, като извличат, обновяват и записват данните в базата данни. Те предоставят абстракция върху слоя на достъп до данни и скриват сложността на взаимодействието с базата данни от контролерите и другите компоненти на приложението.

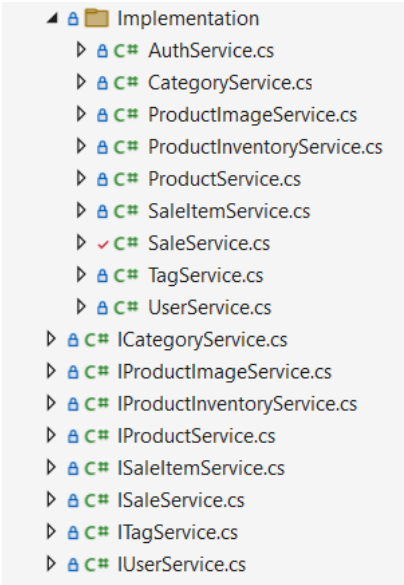
3. **Управление на транзакции**: Слоят на услугите осигурява управление на транзакции, което позволява на приложението да извършва група операции с данни като една атомарна операция. Това гарантира целостта на данните и избягва нежелани последствия от непълни или несъвместими операции.

4. **Обработка на изключения**: Услугите са отговорни за обработката на изключения и генерирането на подходящи отговори за грешки, които могат да възникнат по време на изпълнението на операции. Те осигуряват стабилност и надеждност на приложението, като предоставят ясни съобщения за грешки и предлагат алтернативни решения за проблемите.

Чрез слоя на услугите, приложението постига по-добра структура и поддържаемост, като бизнес логиката е централизирана и изолирана от другите компоненти на системата. Това улеснява разработката, тестването и поддръжката на приложението и гарантира ефективно и надеждно функциониране.

```
namespace tuvarna_ecommerce_system.Service
{
    public interface ITagService
    {
        Task<TagReadDTO> AddTagAsync(TagCreateDTO dto);
        Task<TagReadDTO> PatchTagAsync(TagPatchDTO dto);
        Task<TagReadAllDTO> GetAllTagsAsync();
        Task<TagReadDTO> Delete(int id);
    }
}
```

```
public async Task<TagReadDTO> AddTagAsync(TagCreateDTO dto)
        {
            try
            {
                var tag = new Tag
                {
                    Name = dto.Name.ToLowerInvariant()
                };
                tag = await _tagRepository.CreateAsync(tag);
                var tagReadDto = new TagReadDTO
                {
                    Id = tag.Id,
                    Name = tag.Name
                };
                return tagReadDto;
            }
            catch (DbUpdateException ex)
            {
                ExceptionHandlerUtil.HandleDbUpdateException<TagService
>(_logger, ex, dto.Name,
                    "Attempted to add a duplicate tag: {EntityName}",
                    "Database exception occurred while adding a new
tag. {EntityName}"
                    );
                throw;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "An unexpected error occurred.");
                throw new InternalServerErrorException("An unexpected
error occurred. Please try again later.", ex);
            }
        }
```



Контролерни класове.

Контролерите (Controllers) играят ключова роля във всяко ASP.NET Core приложение, като предоставят точки на вход за външните заявки и управляват взаимодействието с бизнес логиката и данните. Те приемат HTTP заявки от клиентите, обработват ги и връщат подходящ отговор. В следния параграф ще бъде описана функционалността и ролята на контролерите в приложението:

1. **Приемане на HTTP заявки**: Контролерите са отговорни за приемането на HTTP заявките от клиентските устройства, като например уеб браузъри, мобилни приложения и други. Те използват различни HTTP методи като GET, POST, PUT и DELETE, за да извършват съответните операции върху ресурсите в приложението.

2. **Маршрутизация на заявките**: Контролерите определят какви заявки се обработват от кои методи и действия. Това се постига чрез дефиниране на маршрути, които съответстват на шаблони за URI адреси и HTTP методи.

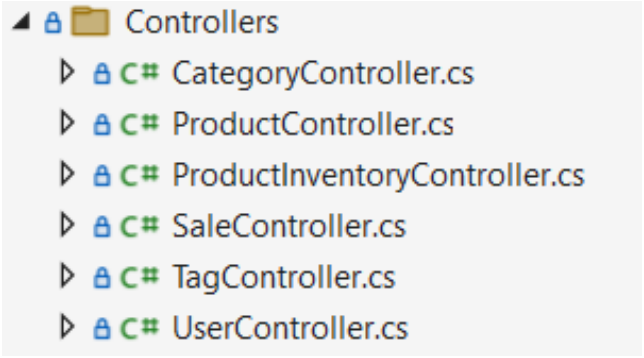
3. **Извикване на бизнес логика**: Контролерите съдържат логиката за обработка на заявките и често извикват методи от слоя на услугите (Service layer), които извършват бизнес логиката и манипулират данните. Те предоставят абстракция върху бизнес логиката и осигуряват ясно разделение между входните данни и операциите върху тях.

4. **Връщане на HTTP отговори**: След като са обработени заявките, контролерите връщат подходящ HTTP отговор към клиентските устройства. Този отговор може да бъде HTML страница, JSON обект, изображение или друг тип данни, в зависимост от типа на заявката и изискванията на приложението.

5. **Обработка на грешки**: Контролерите също така са отговорни за обработката на грешки, които могат да възникнат по време на изпълнението на заявките. Те могат да връщат подходящи HTTP статус кодове и съобщения за грешка, които да улеснят диагностиката и отстраняването на проблемите.

Чрез контролерите, приложението предоставя външно API за взаимодействие с клиентските устройства и осигурява гъвкавост и мащабируемост на приложението. Те предоставят структура и организация на кода, която улеснява разработката, тестването и поддръжката на приложението.

```
[Route("api/v1/products")]
    [ApiController]
    public class ProductController : ControllerBase
    {
        private readonly IProductService _service;
        public ProductController(IProductService service)
        {
            _service = service;
        }
        [HttpPost("add")]
        public async Task<ActionResult<ProductReadDTO>>
Create([FromBody] ProductCreateDTO dto)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(new { message = "Validation failed",
errors = ModelState.Values.SelectMany(v => v.Errors).Select(e =>
e.ErrorMessage) });
            }
            try
            {
                var createdDto = await _service.AddAsync(dto);
                return CreatedAtAction(nameof(Create), new { id =
createdDto.Id }, createdDto);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(new { message = ex.Message });
            }
            catch (EntityNotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
            catch (InternalServerErrorException ex)
            {
                return StatusCode(500, new { message = ex.Message });
            }
        }
}
```



Хранилищен слой.

Хранилището (Repository) е шаблон за проектиране, който служи за изолиране на бизнес логиката от детайлите за достъп до данни. То предоставя абстракция върху базата данни или друг източник на данни и предоставя единно API за извършване на операции за четене, запис и обновяване на данни. В следния параграф ще бъде описана функционалността и ролята на хранилището в приложението:

1. **Изолиране на достъпа до данни**: Хранилището изолира бизнес логиката от детайлите за достъп до данни, като базата данни или външни услуги. Това позволява на бизнес логиката да бъде независима от конкретната имплементация на данните и прави приложението по-гъвкаво и лесно за поддръжка.

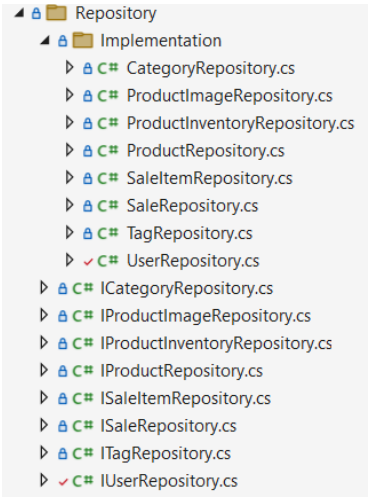
2. **Предоставяне на единно API**: Хранилището предоставя единно и консистентно API за извършване на операции с данни като заявки за четене, запис и обновяване. Това улеснява разработката и поддръжката на приложението, като осигурява единствен начин за взаимодействие с данните.

3. **Управление на транзакции**: Хранилището може да предостави вградена поддръжка за управление на транзакции, което позволява на приложението да извършва група операции с данни като една атомарна операция. Това гарантира целостта на данните и избягва нежелани последствия от непълни или несъвместими операции.

Чрез хранилището, приложението постига по-добра структура и организация на кода, като бизнес логиката е отделена от детайлите за достъп до данни. Това улеснява разработката, тестването и поддръжката на приложението и гарантира ефективно и надеждно използване на данните.

```
namespace tuvarna_ecommerce_system.Data.Repositories
{
    public interface ICategoryRepository
    {
        Task<Category> CreateAsync(Category category);
        Task<Category> PatchAsync(int id, string? name, string?
description, string? imageUrl);
        Task<Category> GetByIdAsync(int id);
        Task<Category> GetByNameAsync(string name);
        Task<List<Category>> GetAllAsync();
        Task<Category> Delete(int id);
    }
}
```

```
namespace tuvarna_ecommerce_system.Repository.Implementation
{
    public class CategoryRepository : ICategoryRepository
    {
        private readonly EcommerceDbContext _context;
        public CategoryRepository(EcommerceDbContext context)
        {
            _context = context;
        }
        public async Task<Category> CreateAsync(Category newCategory)
        {
            var existingCategory = await _context.Categories
                .IgnoreQueryFilters()
                .FirstOrDefaultAsync(c =>
c.Name.Equals(newCategory.Name.ToLowerInvariant()));
            if (existingCategory != null)
            {
                if (existingCategory.IsDeleted)
                {
                    existingCategory.IsDeleted = false;
                    existingCategory.Description =
newCategory.Description;
                    existingCategory.ImageUrl = newCategory.ImageUrl;
                    await _context.SaveChangesAsync();
                    return existingCategory;
                }
            }
            newCategory.Name = newCategory.Name.ToLowerInvariant();
            _context.Categories.Add(newCategory);
            await _context.SaveChangesAsync();
            return newCategory;
        }
}
```



JWT (JSON Web Token)

JWT (JSON Web Token) е стандарт за представяне на токени за удостоверяване, който се използва широко в уеб приложенията за сигурна комуникация между клиенти и сървъри. Този токен е компактен, лек и лесно преносим и съдържа заявка, която е кодирана със сигнатура, която е базирана на JSON формат. В следния параграф ще бъде описана функционалността и ролята на JWT токените:

1.**Удостоверяване на потребителите**: JWT токените се използват за удостоверяване на потребителите и потвърждаване на тяхната идентичност. Когато потребител се аутентикира успешно, сървърът генерира JWT токен и го изпраща до клиента. Този токен се използва за всяка последваща заявка от потребителя към сървъра, за да се потвърди тяхната идентичност.

2. **Авторизация и контрол на достъпа**: Освен удостоверяване, JWT токените могат да се използват и за авторизация и контрол на достъпа до определени ресурси в приложението. Те могат да съдържат различни роли и права на потребителите, които могат да бъдат използвани за ограничаване на достъпа до определени функционалности или данни.

3. **Лек и ефективен за използване**: JWT токените са компактни и леки, което ги прави идеални за изпращане през HTTP заявки и съхраняване в клиентските устройства. Те могат да се изпращат като част от HTTP заглавието или в тялото на заявката и могат да бъдат декодирани и верифицирани с лекота от сървъра.

4. **Сигурност**: JWT токените могат да бъдат подписани с ключове за криптиране, което ги прави сигурни за използване в уеб приложенията. Това осигурява, че токените не могат да бъдат подправени или фалшифицирани от злонамерени потребители и предотвратява възможни атаки като CSRF (Cross-Site Request Forgery).

5. **Склалабилност**: JWT токените са лесни за издаване и валидация, което ги прави подходящи за уеб приложения с голямо натоварване и висока склалабилност. Те позволяват на сървъра да удостоверява и авторизира големи брой потребители ефективно и безпроблемно.

Чрез използването на JWT токените, уеб приложенията могат да осигурят сигурна и ефективна аутентикация и авторизация на потребителите, като предоставят лесен за употреба и гъвкав механизъм за удостоверяване и управление на достъпа.

```
public string GenerateJwtToken(UserReadDTO user)
        {
            try
            {
                var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]))
;
                var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);
                var claims = new[]
                {
                    new Claim(JwtRegisteredClaimNames.Sub,
user.Username),
                    new Claim(JwtRegisteredClaimNames.Email,
user.Email),
                    new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
                    new Claim(ClaimTypes.Role, user.Role.ToString())
                };
                var token = new JwtSecurityToken(
                    issuer: _configuration["Jwt:Issuer"],
                    audience: _configuration["Jwt:Audience"],
                    claims: claims,
                    expires: DateTime.Now.AddMinutes(30),
                    signingCredentials: credentials);
                return new JwtSecurityTokenHandler().WriteToken(token);
            }
            catch (Exception ex)
            {
                _logger.LogError($"Error generating JWT token:
{ex.Message}");
                throw new InternalServerErrorException("Error occurred
while generating JWT token.", ex);
            }
        }
```


```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJuZXd1c2VyIiwiZW1haWwiOiJuZX
d1c2VyQGV4YW1wbGUuY29tIiwianRpIjoiYmQyNzc1OTgtMWMyNy00NDkyLTlmMDctNzM5ZjcwNT
I4YmNmIiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA4LzA2L2lkZW50aXR5L2
NsYWltcy9yb2xlIjoiQ1VTVE9NRVIiLCJleHAiOjE3MTUwMDUxNzgsImlzcyI6Imh0dHA6Ly90dX
Zhcm5hLWVjb21tZXJjZS1zeXN0ZW0uY29tIiwiYXVkIjoiaHR0cDovL3R1dmFybmEtZWNvbW1lcm
NlLXN5c3RlbS5jb20ifQ.nXyA7FkOS8cQyXXwVkc1mkJOLP1MpqoHXUzf9NWbdAU"
}
```

Алгоритми

Генериране на JWT Токен

```
public string GenerateJwtToken(UserReadDTO user)
        {
            try
            {
                var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]))
;
                var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);
                var claims = new[]
                {
                    new Claim(JwtRegisteredClaimNames.Sub,
user.Username),
                    new Claim(JwtRegisteredClaimNames.Email,
user.Email),
                    new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
                    new Claim(ClaimTypes.Role, user.Role.ToString())
                };
                var token = new JwtSecurityToken(
                    issuer: _configuration["Jwt:Issuer"],
                    audience: _configuration["Jwt:Audience"],
                    claims: claims,
                    expires: DateTime.Now.AddMinutes(30),
                    signingCredentials: credentials);

                return new JwtSecurityTokenHandler().WriteToken(token);
            }
            catch (Exception ex)
            {
                _logger.LogError($"Error generating JWT token:
{ex.Message}");
                throw new InternalServerErrorException("Error occurred
while generating JWT token.", ex);
            }
        }
```

Алгоритъмът за генериране на JWT (JSON Web Token) токен, който е представен в дадения код, може да се опише по следния начин:

1. **Подготовка на ключ и подписване** (Signing):

- Създава се ключ (securityKey), който се използва за подписване на токена. Ключът се генерира от симетричен ключ (SymmetricSecurityKey), който е базиран на кодирането на символите от конфигурационния файл на приложението (_configuration["Jwt:Key"]).

- Генерира се подпис (credentials) за токена чрез използването на създадения ключ и алгоритъм за хеширане на съобщенията (SecurityAlgorithms.HmacSha256).

2. **Създаване на данни за токена** (Token Claims):

- Дефинират се различни твърдения (claims), които ще бъдат включени в токена. Тези твърдения обикновено са информация за потребителя, като потребителско име (Sub), имейл адрес (Email), уникален идентификатор на токена (Jti), роля на потребителя (Role) и други.

- Твърденията се добавят към масив от твърдения (claims), който ще бъде включен в създадения токен.

3. **Създаване на JWT токен**:

- Създава се нов обект от тип JwtSecurityToken, който представлява JWT токен. Този обект се инициализира със следните параметри:

- issuer: издателят на токена, който се взема от конфигурационния файл на приложението (_configuration["Jwt:Issuer"]).

- audience: общата аудитория на токена, която също се взема от конфигурационния файл (_configuration["Jwt:Audience"]).

- claims: масивът от твърдения, дефинирани по-горе.

- expires: срокът на валидност на токена, който е зададен на текущото време плюс 30 минути.

- signingCredentials: подписът за токена, който е създаден в стъпка 1.

4. **Записване на токена**:

- Създаденият JWT токен се предава на JwtSecurityTokenHandler, който го преобразува в низ (string) с помощта на метода WriteToken().

- Низът, представляващ токена, се връща като резултат от метода GenerateJwtToken().

Този алгоритъм генерира JWT токен с валидност от 30 минути и включва различни твърдения за потребителя, като потребителско име, имейл адрес, уникален идентификатор и роля. Токенът е подписан с ключ, който осигурява цялост и сигурност на данните.

Създаване на нов продаден инвентар

```
public async Task<List<SaleItemReadDTO>>
CreateAsync(List<SaleItemCreateDTO> dtos, int SaleId)
        {
            var saleItemsRead = new List<SaleItemReadDTO>();
            using (var transaction =
_context.Database.BeginTransaction())
            {
                try
                {
                    foreach (var dto in dtos)
                    {
                        Product productToSell = await
_productRepository.GetByIdAsync(dto.ProductId);
                        int totalStockAvailable =
productToSell.Inventories.Sum(inventory => inventory.StockQuantity);
                        if (dto.QuantitySold > totalStockAvailable)
                        {
                            throw new InvalidOperationException(
                                $"Requested quantity for product ID
{dto.ProductId} exceeds available stock.");
                        }
                        decimal totalPrice = 0m;
                        int quantityToDeduct = dto.QuantitySold;
                        foreach (var inventory in
productToSell.Inventories)
                        {
                            if (quantityToDeduct <= 0) break;

                            int deduct = Math.Min(quantityToDeduct,
inventory.StockQuantity);
                            inventory.StockQuantity -= deduct;
                            quantityToDeduct -= deduct;
                            decimal pricePerUnit =
inventory.DiscountPrice ?? inventory.Price;
                            totalPrice += deduct * pricePerUnit;
                        }
                        var saleItem = new SaleItem
```

```
                        {
                            QuantitySold = dto.QuantitySold,
                            ProductId = dto.ProductId,
                            SaleId = SaleId
                        };
                        decimal productPrice = totalPrice /
dto.QuantitySold;
                        productPrice = Math.Round(productPrice, 2);
                        var createdSaleItem = await
_repository.CreateAsync(saleItem);
                        saleItemsRead.Add(new SaleItemReadDTO
                        {
                            Id = createdSaleItem.Id,
                            ProductId = createdSaleItem.ProductId,
                            QuantitySold =
createdSaleItem.QuantitySold,
                            ProductPrice = productPrice,
                            TotalPrice = totalPrice
                        });
                    }
                    await _context.SaveChangesAsync();
                    transaction.Commit();
                    return saleItemsRead;
                }
                catch (EntityNotFoundException ex)
                {
                    transaction.Rollback();
                    _logger.LogError(ex, ex.Message);
                    throw;
                }
                catch (InvalidOperationException ex)
                {
                    transaction.Rollback();
                    _logger.LogError(ex, ex.Message);
                    throw;
                }
                catch (Exception ex)
                {
                    transaction.Rollback();
                    _logger.LogError(ex, "Failed to create sale
items.");
                    throw new InternalServerErrorException("An
unexpected error occurred. Please try again later.", ex);
                }
            }
        }
```

Алгоритъмът за създаване на нов продаден инвентар, който е представен в дадения код, може да се опише по следния начин:

1. **Започване на транзакция**:

- Създава се транзакция с начало в базата данни (_context.Database.BeginTransaction()), която позволява да се осъществи групова операция върху базата данни, като се гарантира цялост и консистентност на данните.

2. **Итерация през списъка с продукти**:

- За всеки продукт, представен от обектите от тип SaleItemCreateDTO в списъка dtos, се извършва следната логика:

3. **Проверка на наличното количество**:

- За конкретния продукт се извлича цялото налично количество от инвентарите му (_productRepository.GetByIdAsync(dto.ProductId)).

- Сумира се общото количество налично количество на всички инвентари на дадения продукт.

4. **Проверка за наличност на достатъчно количество**:

- Ако заявеното количество за продажба (dto.QuantitySold) е по-голямо от общото налично количество на продукта, се хвърля изключение със съобщение, че заявеното количество надвишава наличното количество.

5. **Изваждане на наличност**:

- Създава се променлива totalPrice, която ще държи общата цена на продадения инвентар.

- За всеки инвентар от продукта, се изважда количество равно на минимума между заявеното количество и наличното количество на текущия инвентар.

- Цената на продукта се определя като се взема цената за единица (pricePerUnit), която може да бъде цената с отстъпка (ако е налична) или цената без отстъпка. Сумира се общата цена за продукта (totalPrice), като се умножава цената за единица по количеството.

6. **Създаване на нов инвентар за продажба**:

- Създава се нов обект от тип SaleItem, който съдържа информация за продадения инвентар.

- След създаването на обекта се добавя към списъка saleItemsRead, който ще съдържа информация за всички създадени продажби.

7. **Фиксиране на цената на продукта**:

- Цената на продукта се определя като се изчислява средната цена на продукта, като се раздели общата цена (totalPrice) на заявеното количество продукти. Цената се закръгля до два десетични знака.

8. **Запазване на промените и завършване на транзакцията**:

- Промените се запазват в базата данни (_context.SaveChangesAsync()) и транзакцията се завършва с команда за потвърждение (transaction.Commit()).

9.**Връщане на информацията за продадените инвентари**:

- Връща се списъкът saleItemsRead с информация за всички успешно създадени продажби.

Този алгоритъм осигурява създаване на продаден инвентар със справедливо разпределение на наличността и цена на продуктите, като се гарантира консистентност и целост на данните в базата данни чрез използване на транзакции.

Хеширане и проверка на паролата

```
private readonly IPasswordHasher<User> _passwordHasher;
```

```
newUser.Password = _passwordHasher.HashPassword(newUser,
userCreateDTO.Password);
var verificationResult = _passwordHasher.VerifyHashedPassword(user,
user.Password, userLoginDTO.Password);
```

Алгоритъмът за хеширане и проверка на паролата, който е представен в дадения код, може да се опише по следния начин:

1. **Хеширане на паролата**:

- При създаването на нов потребител, паролата се хешира преди да бъде запазена в базата данни, за да се осигури по-голяма сигурност.

- За целта се използва интерфейсът IPasswordHasher<T>, който предоставя метод HashPassword за хеширане на паролата.

- Методът HashPassword приема два параметъра - потребителя, за когото се хешира паролата (newUser) и самата парола (userCreateDTO.Password).

- Паролата се хешира със сол (salt), който се генерира автоматично и се добавя към хеширания резултат. Това увеличава сигурността, като предотвратява атаки на основата на речници или предварително генерирани хеширани пароли.

2. **Проверка на хешираната парола**:

- При проверката на паролата при вход в системата, хешираният запис на паролата в базата данни се сравнява с въведената парола от потребителя.

- За целта се използва методът VerifyHashedPassword на интерфейса IPasswordHasher<T>.

- Методът VerifyHashedPassword приема три параметъра - потребителя, за когото се проверява паролата (user), хешираният запис на паролата в базата данни (user.Password) и въведената парола от потребителя (userLoginDTO.Password).

- Методът връща стойност от енумерацията PasswordVerificationResult, която указва дали въведената парола от потребителя съвпада със записа на хешираната парола в базата данни.

- Ако паролите съвпадат, методът връща стойност PasswordVerificationResult.Success, в противен случай връща стойност PasswordVerificationResult.Failed.

Този алгоритъм осигурява сигурно съхранение и проверка на паролите в системата, като използва хеширане със сол и сравняване на хешираните записи на паролите. Това помага за защита на паролите от злонамерени атаки и осигурява поверителност и сигурност на данните на потребителите.

Създаване на Correlation-ID

```
 private const string CorrelationIdHeaderName = "X-Correlation-ID";
    private readonly RequestDelegate _next;
    public CorrelationIdMiddleware(RequestDelegate next)
    {
        _next = next;
    }
        public async Task Invoke(HttpContext context)
        {
            var correlationId =
context.Request.Headers[CorrelationIdHeaderName].FirstOrDefault() ??
Guid.NewGuid().ToString();
            // Set in HttpContext.Items for downstream access
            context.Items[CorrelationIdHeaderName] = correlationId;
            // Use a logging scope to include CorrelationId in logs
generated during this request
            using (LogContext.PushProperty("CorrelationId",
correlationId))
            {
                context.Response.Headers[CorrelationIdHeaderName] =
correlationId;
                await _next(context);
            }
        }
    }
```

Алгоритъмът за създаване на Correlation-ID, който е представен в дадения код на Middleware, може да се опише по следния начин:

1. **Извличане на Correlation-ID от HTTP заявката**:

- В началото на Middleware се извлича Correlation-ID от HTTP заявката, като се проверява дали в заглавките на заявката има стойност за "X-Correlation-ID".

- Ако стойност за "X-Correlation-ID" не се намери в заглавките на заявката, се генерира ново уникално идентификатор (GUID) чрез метода Guid.NewGuid().ToString().

2. **Запазване на Correlation-ID в HttpContext**:

- Полученият Correlation-ID се запазва в свойство на HttpContext.Items, което позволява да се използва този идентификатор в рамките на текущата HTTP заявка.

- Това осигурява възможност за достъп до Correlation-ID в различни части на приложението, които се изпълняват по време на текущата заявка.

3. **Използване на Correlation-ID в логовете**:

- За да се осигури включване на Correlation-ID в логовете, се използва Serilog's LogContext.PushProperty().

- Това добавя Correlation-ID към контекста на лога, така че всички съобщения, генерирани по време на обработката на текущата заявка, да включват този идентификатор.

4. **Установяване на Correlation-ID в HTTP отговора**:

- Correlation-ID се установява като стойност за "X-Correlation-ID" в заглавките на HTTP отговора.

- Това позволява на клиентите или следващите сервиси да използват Correlation-ID за идентификация на конкретната заявка или обмен на данни.

Този алгоритъм за създаване на Correlation-ID осигурява уникален идентификатор за всяка HTTP заявка, който се използва за проследяване на логове и взаимодействията между различните компоненти на системата. Това подобрява откриването на проблеми и анализа на данните в системата, като позволява по-лесно свързване на различните събития и заявки в рамките на един поток на изпълнение.

Дефиниране на проблеми и сложност на поставената задача

Проблем 1: Избор на база данни

Изборът на подходяща база данни е ключов елемент при проектирането на всяка информационна система, тъй като тя играе важна роля за съхранението, обработката и извличането на данни. Важността на този избор се дължи на няколко фактора:

1. Мащабируемост и Производителност: Базата данни трябва да може да се справя с нарастващите обеми от данни и потребителски заявки, без да компрометира производителността.
2. Съвместимост с Технологичния Стек: Трябва да бъде съвместима с останалите технологии, използвани в проекта, за да осигури плавна интеграция и комуникация.
3. Сигурност и Надеждност: Сигурността на данните е критична, особено при съхранението на чувствителна информация. Избраната база данни трябва да предлага механизми за защита на данните и гарантиране на тяхната надеждност.
4. Скалабилност и Поддръжка: Възможността за мащабиране на базата данни в бъдеще и лесната поддръжка са също от решаващо значение за дългосрочната устойчивост на системата.

Проблем 2: Свързване на приложния слой с базата данни

Свързването на приложния слой с базата данни е един от основните при разработката на информационни системи. Този процес включва следните ключови аспекти:

1. Интеграция: Ефективната интеграция между приложния слой и базата данни е критична за гладката работа на системата. Това изисква правилното избиране и конфигуриране на технологии за достъп до данни, като *ORM (Object-Relational Mapping)* инструменти или специализирани библиотеки за бази данни.
2. Производителност и оптимизация на заявките: За да се гарантира висока производителност, заявките към базата данни трябва да бъдат оптимизирани. Това включва ефективно управление на ресурсите, избягване на излишни заявки и оптимизиране на схемите на базата данни.
3. Транзакционност: Системата трябва да гарантира целостността на данните и да поддържа съответните нива на транзакционност.

Проблем 3: Дефиниране на моделите и създаване на таблици от тях

Дефинирането на модели и преобразуването им в таблиците на базата данни е фундаментален етап в разработката на информационни системи. Този процес включва следните ключови аспекти:

1. Моделиране на Данните: Създаването на точни и ефективни модели е критично за представянето на бизнес логиката в структурата на данните. Моделите трябва да отразяват реалните същности и отношенията между тях, като същевременно оптимизират достъпа и обработката на данни.
2. Преобразуване в табличен формат: След като са дефинирани, моделите трябва да бъдат преобразувани в таблици в рамките на базата данни. Това включва определяне на подходящи типове данни, ключове (първични и чужди) и ограничения, които гарантират целостността и ефективността на данните.
3. Нормализация и оптимизация: Нормализацията на схемата на базата данни е важна за избягване на излишни данни и за увеличаване на ефективността. Оптимизацията на схемата и заявките е необходима за поддържане на висока производителност при увеличаване на обема на данни.

Проблем 4: Изграждане на трислойна архитектура (Repository -> Service -> Controller)

Изграждането на трислойна архитектура е фундаментален аспект при разработката на устойчиви и модулни информационни системи. Тази архитектура разделя системата на три основни слоя - *Repository* (Хранилище), *Service* (Услуга) и *Controller* (Контролер), като всеки от тях има своя отделна роля и отговорности:

1. *Repository Layer* (слой хранилище): Управлява директния достъп до базата данни, включително заявки, вмъквания, обновявания и изтривания на данни. Осигуряването на ефективен и оптимизиран достъп до данните, гарантиране на целостността на данните и абстрахиране на сложността на базата данни.
2. *Service Layer* (слой услуга): Отговорности: Съдържа бизнес логиката на приложението. Осъществява обработка на данни, извлечени от хранилището, и осигурява функционалността, изисквана от потребителския интерфейс. Балансиране между логиката на приложението и достъпа до данни, осигуряване на транзакционна сигурност и управление на бизнес правила.
3. *Controller Layer* (слой контролер): Управлява потребителския вход и изход, комуникира с услугите за извличане на информация или

изпълнение на бизнес операции и решава кой изглед да бъде показан на потребителя. Манипулация на FXML елементи.

Проблем 5: Прилагане на принципа за обръщане на зависимостите (Dependency Inversion Principle)

Прилагането на принципа за обръщане на зависимостите (Dependency Inversion Principle, DIP) е ключов аспект в създаването на гъвкава и поддържаема софтуерна архитектура. Този принцип е част от SOLID принципите за обектно-ориентирано програмиране и има за цел да намали взаимозависимостите между модулите на програмата. Принципът предполага, че модулите от високо ниво не трябва да зависят директно от модулите от ниско ниво, а и двете трябва да зависят от абстракции.

Проблем 6: Автентикация и Авторизация

Разработката на надеждни системи за автентикация и авторизация е критичен компонент в създаването на сигурни информационни системи. Автентикация е процесът на установяване на идентичността на потребителя, често чрез въвеждане на потребителско име/имейл и парола. Авторизация е процесът на определяне на правата на достъп на потребителя след успешната му автентикация.

Подходи, методи за решаване на поставените проблеми

Подход към Проблем 1: Избор на база данни

При проектирането на система е от съществено значение да се избере подходяща база данни, която отговаря на изискванията за мащабируемост, производителност, съвместимост, сигурност и надеждност. В този контекст, използването на Amazon Web Services (AWS) SQL Server представлява едно от водещите решения, което отговаря на тези изисквания.

1. Мащабируемост и Производителност:

- AWS SQL Server предлага възможности за мащабируемост, като предоставя високо производителни инстанции на сървъри и възможност за автоматично управление на ресурсите. Това позволява на системата да се справя с нарастващите обеми от данни и потребителски заявки, като осигурява стабилна производителност.

2. Съвместимост с Технологичния Стек:

- AWS SQL Server е съвместим с широка гама от технологии и инструменти, което улеснява интеграцията и комуникацията с други компоненти на системата. Той може да бъде лесно интегриран с различни приложения и услуги, предлагани от AWS, както и с външни системи.

3. Сигурност и Надеждност:

- AWS SQL Server предлага вградени механизми за сигурност и надеждност, които осигуряват защита на данните и гарантиране на тяхната цялостност. Той предлага възможности за шифроване на данните в покой и при тяхната трансфер, управление на достъпа и автентикация, както и автоматично резервно копиране и възстановяване на данни.

4. Скалабилност и Поддръжка:

- AWS SQL Server предоставя възможности за лесно мащабиране на базата данни във време на нужда, както и за управление на ресурсите според текущите изисквания. Също така, AWS предлага обширна документация, обучение и поддръжка, което улеснява администрирането и поддръжката на системата в дългосрочен план.

Използването на AWS SQL Server като решение за съхранение на данни предоставя комплексно и надеждно решение, което отговаря на изискванията за ефективно управление на данните в информационната система.

Подход към Проблем 2: Свързване на приложния слой с базата данни

Entity Framework Core (EF Core) е мощен инструмент за свързване на приложния слой с базата данни, който адресира ключовите аспекти на свързването между двата слоя - интеграция, производителност и оптимизация на заявките, както и транзакционност.

1. Интеграция:

- EF Core предоставя ORM инструмент, който позволява лесно създаване и манипулиране на обекти от базата данни като обекти от .NET. Това улеснява разработката на приложенията, като скрива детайлите на базата данни и позволява работа с обекти на по-високо ниво.

- Чрез моделиране на данни чрез код (Code First) или обратното инженерство на базата данни (Database First), EF Core осигурява лесно интегриране на данни и операции с тях в приложението.

2. Производителност и оптимизация на заявките:

- EF Core предоставя механизми за оптимизация на заявките, като предварително зареждане на свързаните данни (Eager Loading), отлагане на зареждането на данни (Lazy Loading) и използване на индекси и оптимални схеми на базата данни.

- Чрез използването на LINQ (Language Integrated Query), EF Core позволява удобно и ефективно формулиране на заявки към базата данни директно в кода на приложението.

3. Транзакционност:

- EF Core поддържа транзакционност чрез използването на транзакции в базата данни. Това позволява на системата да осигури цялостността на данните и да извършва атомарни операции в рамките на транзакция.

Използването на Entity Framework Core като решение за свързване на приложния слой с базата данни предоставя множество ползи, включително удобство в разработката, висока производителност и гъвкавост в работата с данни. Това го прави подходящ избор за разработка на информационни системи, които изискват ефективна интеграция и управление на данните.

За решаване на този проблем, приложението използва Hibernate като зависимост. Hibernate е популярна *Object-Relational Mapping (ORM)* работна рамка за *Java* приложения. Той позволява разработчиците да превръщат *Java* обекти към таблици в релационна база данни и автоматизира голяма част от работата, свързана с трансформацията на данните от обектно ориентирания в релационен модел. Hibernate предоставя свой собствен език за заявки, *HQL*, който е ориентиран към обекти и позволява извършването на сложни заявки над данните. Поддържа управление на транзакции, което е критично за сигурността и целостността на данните. Hibernate позволява конфигуриране на мапингите между *Java* класове и таблици чрез анотации или *XML* файлове.

Подход към Проблем 3: Дефиниране на моделите и създаване на таблици от тях

При създаването на моделите и тяхното трансформиране в таблици в рамките на базата данни, се разглеждат няколко ключови аспекта, които гарантират ефективното функциониране и оптимизация на информационната система.

1. Моделиране на Данните:

- За да се осигури точното представяне на бизнес логиката, моделите трябва да бъдат добре проектирани и да отразяват същностите на бизнеса, като например продукти, категории, продажби и потребители.

- В случая на описаните модели като Categories, Products, SaleItems и други, важно е да се дефинират необходимите атрибути и връзки между тях, така че да се отразят основните характеристики и връзките между различните същности.

2. Преобразуване в табличен формат:

- След като са дефинирани моделите, те трябва да бъдат преобразувани в таблици в базата данни. За всяка същност от моделите се създава съответна таблица, като всеки атрибут на модела се преобразува в поле в таблицата.

- Например, моделът Products би могъл да се преобрази в таблица със съответните полета като име, описание, SKU (Stock Keeping Unit), цена и други.

3. Нормализация и оптимизация:

- Нормализацията на базата данни е важна за избягване на излишни данни и за оптимизация на структурата на базата данни. Тя включва разделянето на данните в различни таблица, за да се намали повторението на информацията и да се осигури по-добра управляемост.

- Оптимизацията на схемата на базата данни включва създаването на подходящи индекси и ограничения, които ускоряват изпълнението на заявките и поддържат цялостността на данните.

Създаването на моделите и техните таблици в базата данни представлява ключов етап в разработката на информационна система. Правилното дефиниране и оптимизация на тези структури е от съществено значение за осигуряване на ефективност, гъвкавост и производителност в рамките на системата.

Подход към Проблем 4: Изграждане на трислойна архитектура

В контекста на нашият онлайн магазин, тази архитектура позволява ясно разделение на отговорностите и по-лесно управление на кода.

1. Слой за Достъп до Данни: Описание: `CategoryRepository, ProductImageRepository, ProductInventoryRepository, ProductRepository, SaleItemRepository, SaleRepository, TagRepository, UserRepository`

2. Слой за Бизнес Логика (Service): Описание: `AuthService, CategoryService, ProductImageService, ProductInventoryService, ProductService, SaleItemService, SaleService, UserService, TagService`

3. Презентационен Слой (Controllers): Описание: `CategoryController, ProductController, ProductInventoryController, SaleController, TagController, UserController`

Подход към Проблем 5: Прилагане на принципа за обръщане на зависимостите (Dependency Inversion Principle)

```
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
```

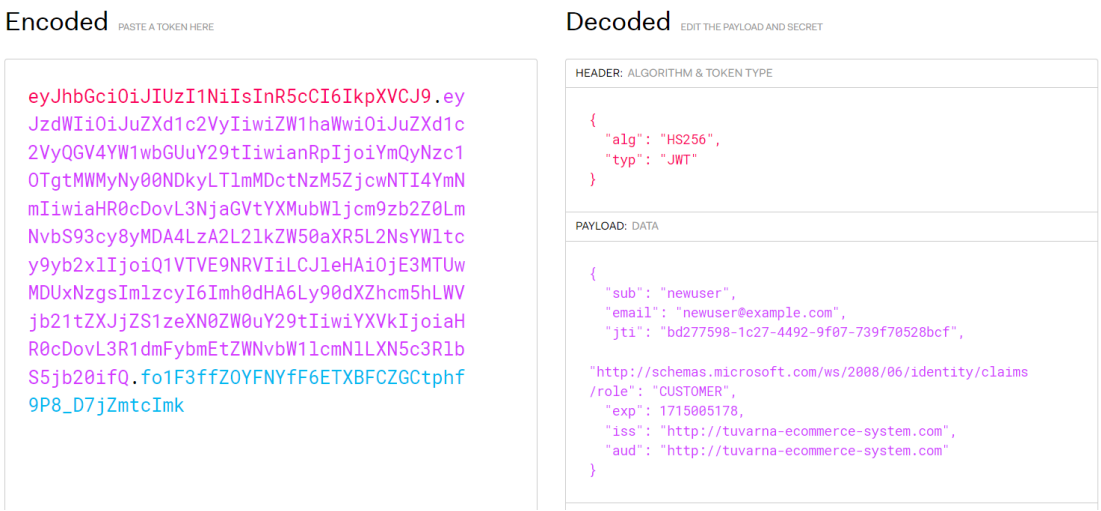
```
public class CategoryService : ICategoryService
    {
        private readonly ICategoryRepository _categoryRepository;
        private readonly ILogger<CategoryService> _logger;
        public CategoryService(ICategoryRepository categoryRepository,
ILogger<CategoryService> logger)
        {
            _categoryRepository = categoryRepository;
            _logger = logger;
        }
```

- В класа `CategoryService` се използва конструкторът, чрез който се внедряват зависимостите `ICategoryRepository` и `ILogger<CategoryService>`. Това позволява на `CategoryService` да използва функционалността, предоставена от `CategoryRepository` за достъп до данни и от `ILogger` за запис на логове.

- Прилагайки DIP, `CategoryService` зависи от абстракцията `ICategoryRepository`, а не от конкретната имплементация `CategoryRepository`. Това прави `CategoryService` гъвкав, тъй като може да работи с различни имплементации на репозиторията, без да е необходима промяна в неговия код.

Така, като приложаваме принципа за обръщане на зависимостите, постигаме по-голяма гъвкавост и поддържаемост на кода, като разделяме отговорностите и намаляваме взаимната зависимост между компонентите на системата.

Подход към Проблем 6: Автентикация и Авторизация



На предоставената снимка е показан интерфейс на уебсайт или програма за декодиране на JWT (JSON Web Tokens). Снимката включва две основни части:

1. Кодирано: Това е лявата страна на екрана, където е представен кодираният JWT. Той изглежда като дълъг низ от символи, разделен на три части, разделени с точки, което е типично за JWT.

2. Декодирано: Това е дясната страна на екрана, където същият JWT е разделен на две части:

- Header (Заглавие): Показва алгоритъма за шифроване (`HS256`) и типа на токена (`JWT`).

- Payload (Носител на данни): Съдържа различни клеймс (claims) като:

  - `sub` (subject): Идентификатор на потребителя.

  - `email`: Електронната поща на потребителя.

  - `jti` (JWT ID): Уникален идентификатор на токена.

  - `role`: Ролята на потребителя (например "CUSTOMER").

  - `exp` (expiry time): Време на изтичане на валидността на токена.

  - `iss` (issuer): Издателят на токена.

  - `aud` (audience): Предназначението на токена.

Тази снимка е пример за това как може да изглежда интерфейс за работа с JWT, като се използва за аутентикация и авторизация в уеб приложения и системи.

Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

Функционални изисквания:

Функционалните изисквания на онлайн магазина за електроника играят ключова роля в определянето на функционалността и поведението на системата. Тези изисквания се фокусират върху функционалността, която потребителите ще използват, както и върху начина, по който те ще взаимодействат с платформата. В следния параграф са представени някои от основните функционални изисквания за нашия проект онлайн магазин за електроника:

1. Регистрация и управление на потребители:

- Системата трябва да предоставя възможност за регистрация на нови потребители, включително въвеждане на данни като име, имейл, и парола.

- Потребителите трябва да имат възможност да редактират своите профили и да променят своите лични данни.

2. Преглед на продуктов каталог:

- Потребителите трябва да имат достъп до продуктов каталог, който включва разнообразие от електроника, като смартфони, лаптопи, телевизори и други.

- В каталога трябва да има възможност за филтриране и сортиране на продуктите по различни критерии, като марка, цена, характеристики и др.

3. Добавяне на продукти в кошницата:

- Потребителите трябва да могат да добавят желаните продукти във виртуалната си кошница.

- В кошницата трябва да се показва обобщение на добавените продукти, техните цени и общата сума за плащане.

4. Оформяне на поръчка и плащане:

- Потребителите трябва да имат възможност да оформят поръчка за покупката на добавените продукти.

- Системата трябва да предоставя различни методи за плащане.

5. Преглед на поръчки и история на покупките:

- Потребителите трябва да могат да преглеждат своите предходни поръчки и да видят информация за тяхното изпълнение и доставка.

- Системата трябва да позволява потребителите да следят статуса на текущите си поръчки и да получават уведомления за промени в тях.

6. Административни функции:

- Администраторите на системата трябва да имат възможност да управляват продуктовия каталог, потребителските профили, поръчките и други аспекти на системата.

- Администраторите трябва да могат да добавят, редактират и премахват продукти, да управляват складовите наличности и да преглеждат статистическа информация за продажбите.

Тези функционални изисквания представят основните възможности и поведения, които се очаква да предоставя системата на потребителите си. Въз основа на тях може да се разработи подробен дизайн на софтуерната архитектура и да се изгради функционалността на онлайн магазина за електроника.

Нефункционални изисквания:

Нефункционалните изисквания определят качествени атрибути и ограничения, които системата трябва да изпълнява, за да бъде успешна и удовлетворява потребителските очаквания. В случая с онлайн магазина за електроника, нефункционалните изисквания включват следните аспекти:

1. Издръжливост и Надеждност:

- Системата трябва да бъде издръжлива и да поддържа висока надеждност, осигурявайки непрекъснато достъп до услугите си.

- Задължително е да се осигури планирано поддръжка и наблюдение на системата, за да се предотвратят и управляват евентуални проблеми или прекъсвания в работата й.

2. Сигурност на данните:

- Системата трябва да прилага съответните мерки за защита на данните, включително криптиране на данни и управление на достъпа.

- Паролите трябва да се съхраняват като хеширани стойности, за да се предотврати тяхната компрометация в случай на нарушение на сигурността.

3. Скорост и Производителност:

- Системата трябва да осигури бърз и отзивчив интерфейс за потребителите, независимо от тяхната локация и устройство.

- Заявките към базата данни и обработката на данните трябва да бъдат оптимизирани, за да се гарантира минимално време за зареждане на страниците и изпълнение на операциите.

4. Скалируемост:

- Системата трябва да бъде скалируема, за да се справи с нарастващия брой потребители и обеми от данни.

- Трябва да бъде възможно лесно добавяне на допълнителни ресурси и разпределени системи за обработка на данни и заявки.

5. Съвместимост

- Системата трябва да бъде съвместима с различни уеб браузъри и устройства, за да осигури оптимално потребителско изживяване за всички потребители.

6. Лесно управление и поддръжка:

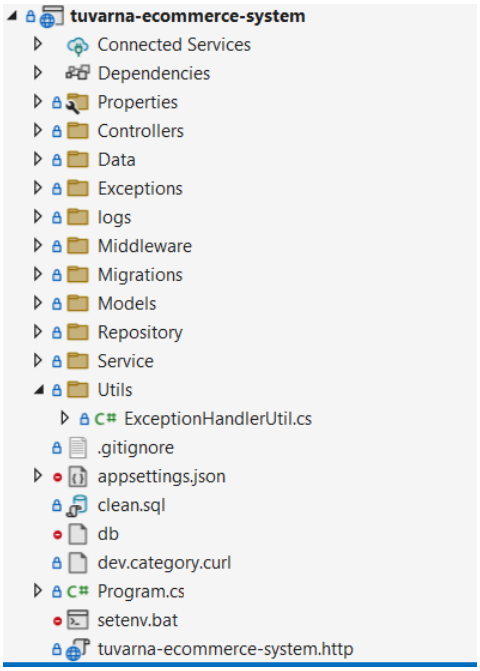
- Администраторите на системата трябва да имат лесен достъп и управление на административните функции и данни.

- Системата трябва да предлага инструменти за мониторинг и анализ на данните, които да подпомагат администраторите във вземането на решения и оптимизирането на системата.

С тези нефункционални изисквания се стремим да осигурим не само функционалността, но и високо качество и удобство за потребителите на нашия онлайн магазин за електроника.

Глава 3. Проектиране.

Обща структура на проекта пакети, които ще се реализират.



1. Connected Services: Папка, съдържаща конфигурации или референции към свързани услуги, които проектът използва.

2. Dependencies: Зависимости на проекта, библиотеки и пакети, които са необходими за неговото изпълнение.

3. Properties: Настройки на проекта, обикновено включващи конфигурационни файлове.

4. Controllers: Папка, съдържаща контролерите, които управляват потребителските заявки и връщат отговори.

5. Data: Папка, съдържаща логика за взаимодействие с базата данни.

6. Exceptions: Класове или файлове, свързани с обработката на изключения в проекта.

7. Logs: Папка за лог файлове, където се записва информация за работата на приложението.

8. Middleware: Папка, съдържаща междинен софтуер, който се изпълнява при всяка заявка към приложението.

9. Migrations: Скриптове или класове за управление на миграции на базата данни.

10. Models: Папка, съдържаща модели, които представляват структурата на данните в проекта.

11. Repository: Папка с класове, които управляват връзката между бизнес логиката и базата данни.

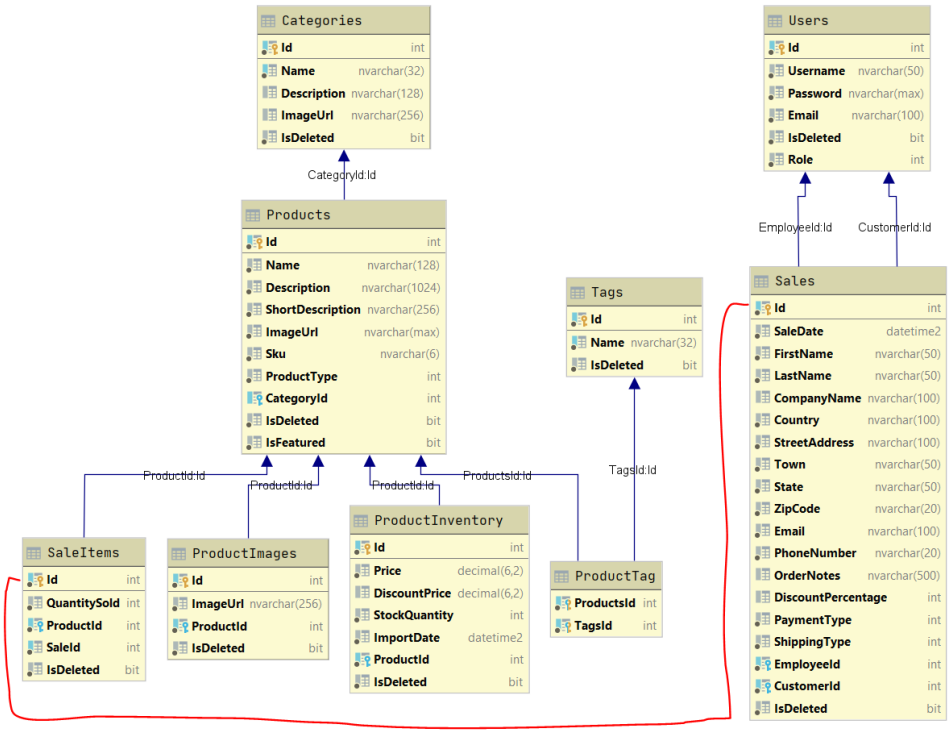
12. Service: Сервизи, предоставящи бизнес логика за приложението.

13. Utils: Помощни класове или функции, използвани в различни части на приложението.

Сред файловете са:
- appsettings.json: Конфигурационен файл за настройки на приложението.
- Program.cs: Основен входен файл за .NET проекта.

Тази структура показва добре организиран проект, който разделя логиката си по начин, който улеснява поддръжката и разширението.

База данни.



Изображението показва диаграма на база данни за електронна търговия, която включва различни таблици и техните взаимовръзки.

Таблици:

1. Categories: Съхранява информация за категориите на продуктите. Полета:
   - `Id`: Уникален идентификатор.
   - `Name`: Име на категорията.
   - `Description`: Описание на категорията.
   - `ImageUrl`: URL към изображение, представящо категорията.
   - `IsDeleted`: Флаг, указващ дали записът е изтрит.

2. Products: Съдържа информация за продуктите. Полета:
   - `Id`, `Name`, `Description`, `ShortDescription`, `ImageUrl`, `Sku`, `ProductType`, `CategoryId`, `IsDeleted`, `IsFeatured`: Атрибути, описващи продукта.

3. ProductImages: Съхранява изображения, свързани с продуктите.
   - `Id`, `ImageUrl`, `ProductId`, `IsDeleted`: Атрибути на изображенията.

4. ProductInventory: Управлява инвентара на продуктите.
   - `Id`, `Price`, `DiscountPrice`, `StockQuantity`, `ImportDate`, `ProductId`, `IsDeleted`: Детайли за наличност и ценообразуване.

5. Tags: Тагове, които могат да бъдат асоциирани с продуктите.
   - `Id`, `Name`, `IsDeleted`: Базова информация за тага.

6. ProductTag: Асоциативна таблица, свързваща продукти с тагове.
   - `ProductId`, `TagId`: Връзка между продукт и таг.

7. Sales: Записи за продажби.
   - `Id`, `SaleDate`, `FirstName`, `LastName`, `CompanyName`, и други детайли, включително адрес и контактна информация на клиента.

8. SaleItems: Продукти, продадени в рамките на дадена продажба.
   - `Id`, `QuantitySold`, `ProductId`, `SaleId`, `IsDeleted`: Информация за продадените артикули.

9. Users: Потребителски данни.
   - `Id`, `Username`, `Password`, `Email`, `Role`, `IsDeleted`: Данни за потребителите на системата.

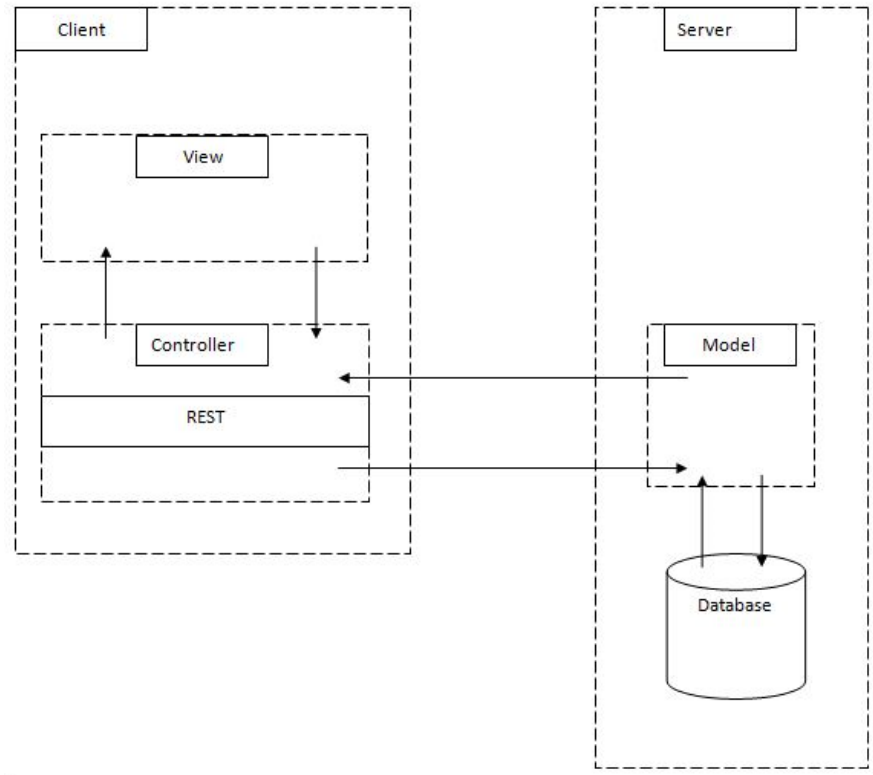
Взаимовръзки:

- Продукти и категории: Всяка категория може да има много продукти (`CategoryId` в таблицата `Products`).
- Продукти и изображения: Всеки продукт може да има множество изображения (`ProductId` в таблицата `ProductImages`).
- Продукти и инвентар: Всяка единица инвентар е свързана с конкретен продукт (`ProductId` в таблицата `ProductInventory`).

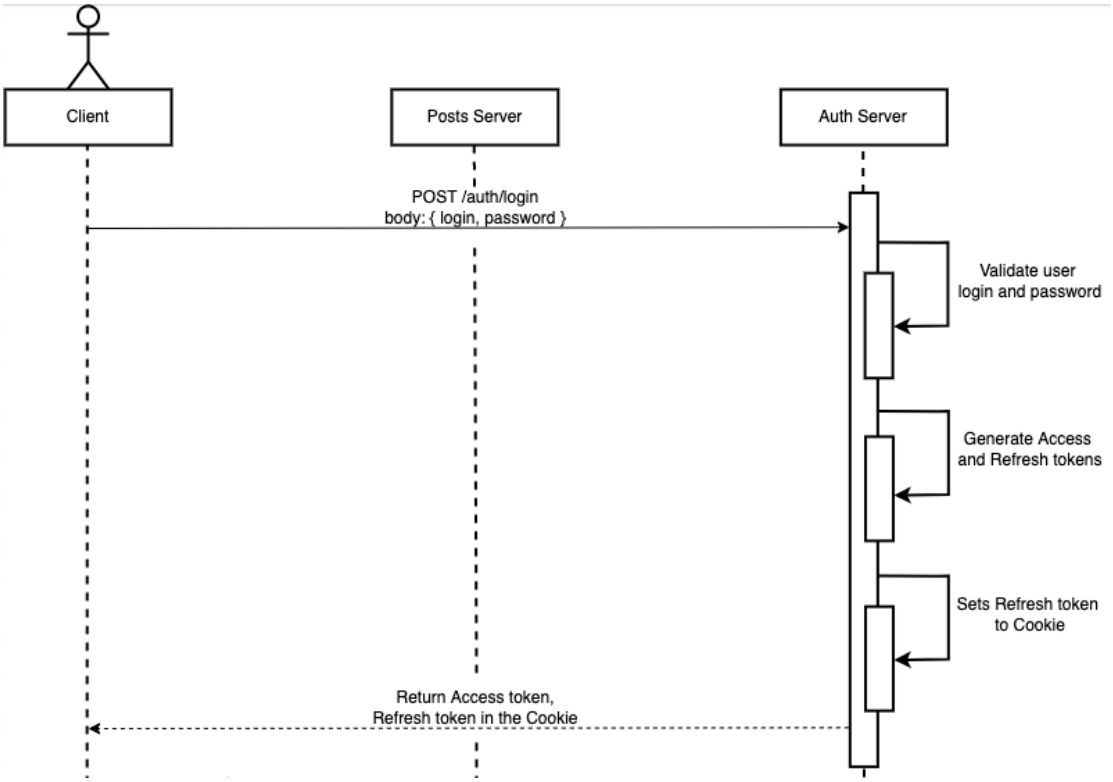
- Продукти и тагове: Множество продукти могат да бъдат свързани с множество тагове чрез таблицата `ProductTag`.
- Продажби и продадени артикули: Всяка продажба може да има множество артикули (`SaleId` в таблицата `SaleItems`).

Тази диаграма представя структуриран подход към управлението на данни в електронна търговска система, позволяващ лесен достъп и управление на продукти, инвентар, продажби и потребителска информация.

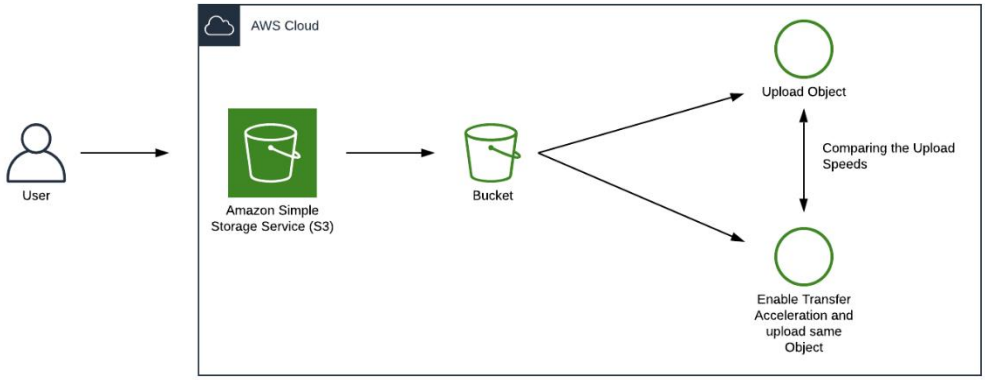
Asp.Net MVC Диаграма



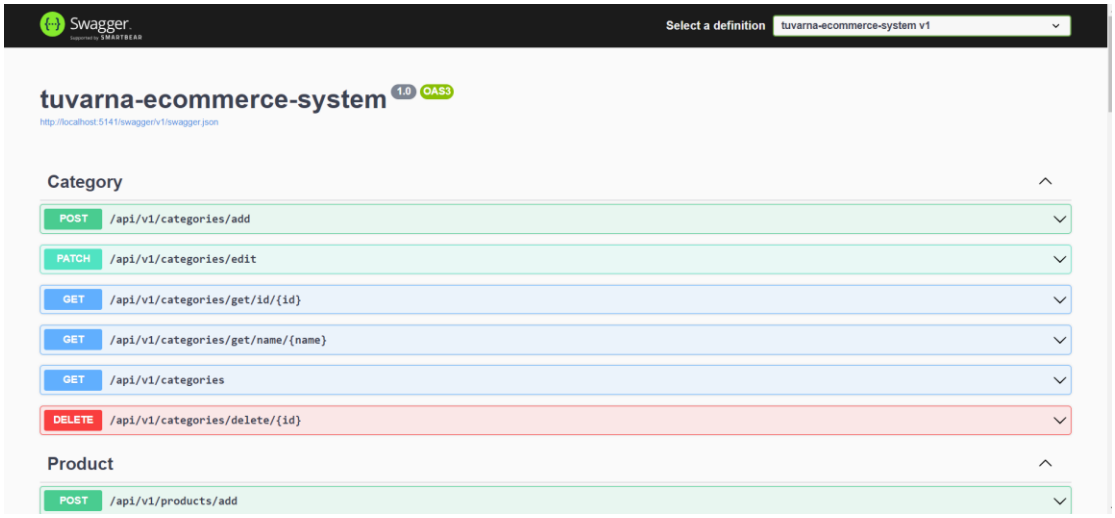
JWT Token UML Диаграма



AWS S3 Диаграма. Качвана на файлове.



Глава 4. Работа с програмата. Ръководство.



Swagger (наскоро преименуван на OpenAPI) е набор от инструменти, които се използват за проектиране, създаване, документиране и използване на RESTful уеб услуги. Основната цел на Swagger е да улесни процеса на разработка на API-та, като осигури набор от инструменти за генериране на интерактивна документация, клиентски библиотеки и сървърни стабове от самото API описание.

На предоставеното изображение е показан интерфейс на Swagger UI за конкретен API проект, наречен "tuvama-ecommerce-system". Това уеб-базирано приложение позволява на разработчиците и други заинтересовани страни лесно да взаимодействат с API чрез изпълнение на API заявки директно от документацията.

Този инструмент е особено полезен за тестване, документация и предоставяне на ясен начин за разработчиците да разберат как работи API и как могат да го използват в своите приложения.

```
curl --location 'http://localhost:5141/api/v1/categories/add' \
--header 'Content-Type: application/json' \
--data '{
  "Name": "Кухненски електроуреди"
}'

curl --location 'http://localhost:5141/api/v1/categories/get/id/1' \
--header 'Content-Type: application/json'

curl --location --request DELETE
'http://localhost:5141/api/v1/categories/delete/12' \
--header 'Content-Type: application/json'

curl --location 'http://localhost:5141/api/v1/categories/get/id/ ' \
```

```
--header 'Content-Type: application/json'

curl --location 'http://localhost:5141/api/v1/categories/get/name/phones' \
--header 'Content-Type: application/json'

curl --location 'http://localhost:5141/api/v1/tags/add' \
--header 'Content-Type: application/json' \
--data '{
  "name": "smarttv"
}
'

curl --location --request PATCH 'http://localhost:5141/api/v1/tags/edit' \
--header 'Content-Type: application/json' \
--data '{
  "id": 1,
  "name": "ios"
}
'
curl --location 'http://localhost:5141/api/v1/tags' \
--header 'Content-Type: application/json'

curl --location 'http://localhost:5141/api/v1/products/add' \
--header 'Content-Type: application/json' \
--data '{
  "Name": "Часовник Apple Watch Ultra 2",
  "Description": "Кафе машина.",
  "ShortDescription": "Кафе машина.",
  "ImageUrl": "https://cdncloudcart.com/402/products/images/146577/smart-
casovnik-apple-watch-ultra-2-cell-49mm-orangebieg-loop-ml-mrf23--1-92--64--
apple-s9-sip-64-bit-dual-core-650339745ef00.webp?1694710132",
  "ProductType": "Physical",
  "CategoryName": "смарт часовници"
}

'
```

Глава 5. Тестови сценарии.

В предоставената Postman колекция могат да бъдат изтествани API заявките.

tuvarna-ecommerce-system.postman_colle