

# **FOREXFLOW. WINDOWS DESKTOP APPLICATION**

**.NET C# WPF  
Based Desktop Application.  
Documentation**

**Developer:**  
Alexsandar Ganchev

Contacts: [ganchev.professional@gmail.com](mailto:ganchev.professional@gmail.com)

**All rights reserved.  
Sofia, 2025**

# Съдържание

<b>Увод</b> .....	<b>3</b>
<b>Проектиране</b> .....	<b>3</b>
Архитектурна структура .....	3
<b>Реализация</b> .....	<b>6</b>
Domain Layer (ForexFlow.Models) .....	6
Data Access Layer (ForexFlow.DataAccess) .....	6
Dependency Injection .....	9
Presentation Layer (ForexFlow.View) .....	11
Application Layer (ForexFlow.ViewModel) .....	13
Взаимодействие с UnitOfWork .....	13
Взаимодействие с View-мо .....	13
Presentation Layer (ForexFlow.View) .....	18
Currency Management .....	18
Amounts Management .....	19
Single Amount Actions .....	19
Invoice Management .....	20
<b>Заклучение и място за подобрения</b> .....	<b>22</b>

## Увод

Настоящият практически проект е разработен в рамките на дисциплината, с цел усвояване на умения за създаване на десктоп приложения чрез WPF (Windows Presentation Foundation) с .NET платформа. Темата на проекта – **ForexFlow** – е предложена от преподавателя и цели реализирането на функционален софтуер за управление на валути, мениджмънт на парични стойности и генериране на базови фактури.

Заложените цели на проекта са в три основни направления:

- **Възможност за създаване и менажиране на валути** – потребителят трябва да може да дефинира различни валути с основни характеристики, както и да ги редактира при нужда.
- **Възможност за създаване на стойности, обвързани с определена валута** – приложението позволява потребителят да въвежда конкретни суми, свързани с дадена валута, като върху тях могат да се прилагат различни действия, свързани с управление или трансформация.
- **Съставяне на базови фактури** – имплементирана е функционалност за генериране на прости фактури, използващи наличните валути и стойности, като това цели да демонстрира практически аспект от реалното приложение на събраните данни.

Проектът има за цел не само да покрие техническите изисквания, но и да демонстрира добра организация на кода, логическо разделяне на функционалностите и интуитивен потребителски интерфейс. Разработката е структурирана така, че да бъде разширяема и лесно поддържаема, като позволява награвждане с допълнителни модули в бъдеще.

## Проектиране

Проектът *ForexFlow* е реализиран чрез архитектурен подход, съчетаващ **MVVM шаблон (Model-View-View Model)** и **многостепенна (N-tier) архитектура**, с цел осигуряване на ясно разделение в отговорностите между компонентите, лесна поддръжка и възможност за бъдещо разширяване.

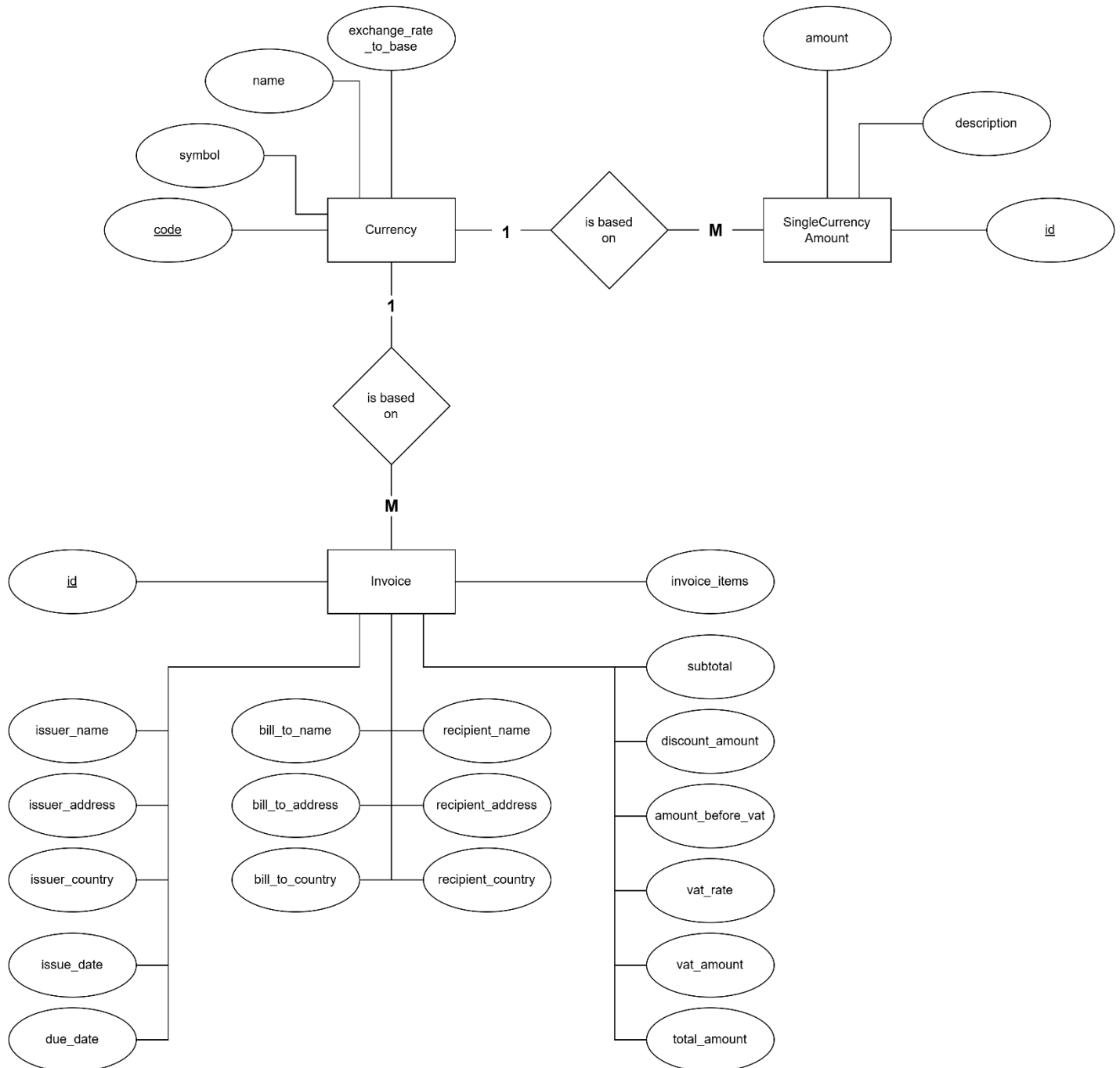
### Архитектурна структура

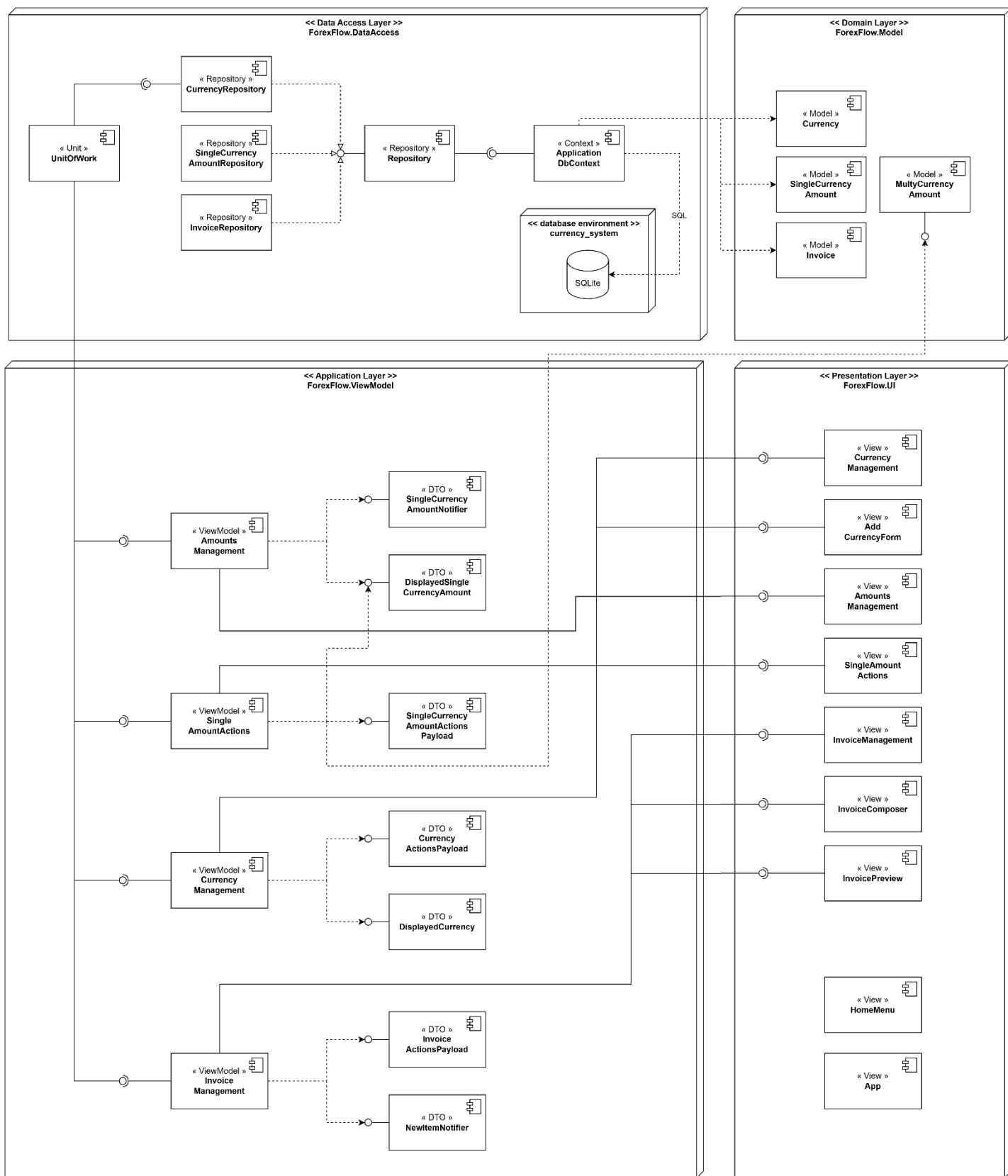
Приложението е разделено логически на няколко самостоятелни слоя, като това е постигнато, чрез организацията на класовете под отделни *project class collections*.

- **Domain Layer (ForexFlow.Models):** съдържа дефиницията и описанието за следните бизнес моделите: *Currency*, *SingleAmountCurrency*, *Invoice* и помощен модел *MultiCurrencyAmount* (който не се съхранява в базата данни, както е заградено по условие).
- **Application Layer (ForexFlow.ViewModel):** съдържа логиката, която осъществява връзката между потребителски интерфейс и бизнес слоя. Представена е чрез четири основни *ViewModel* класа, всеки от които изпълнява своя отделна смислова

функционалност: *CurrencyManagementViewModel*, *AmountsManagementViewModel*, *SingleAmountActionsViewModel* и *InvoiceManagementViewModel*.

- **Data Access Layer (ForexFlow.DataAccess):** управлява връзката с базата данни чрез **Entity Framework** и **SQLite**. Макар и наборът от модели на първоначален етап да е малък набор, в проекта са използвани подходите **repository pattern** и **unit of work** за по-добра организация и отделяне на логиката за достъп до данни.
- **Presentation Layer (ForexFlow.View):** съдържа потребителските интерфейси, реализирани чрез WPF и XAML, със следните главни View компоненти: *CurrencyManagement*, *AmountsManagement*, *SingleAmountActions* и *InvoiceManagement*.





# Реализация

## Domain Layer (ForexFlow.Models)

Дефинирането на моделите е стандартно и следва базовата концепция, уповаваща се на принципа да бъде описано entity-то съхранявано в базата данни.

За да бъде спазена конвенцията за това, че моделите трябва да съдържат в структурата си само дефиницията на entity-то, са конструирани специални extension-ни, към някои от моделите. В тях са съставени методи за модификация на зададен обект от тип предоставения модел.

*Тази стъпка не е наложителна, но ако не бъде изпълнена ще бъде в разрез с конвенциите за следването на шаблони като MVVM и MVC.*

В посочения клас като пример са налични множество от подобни функционалности, като са предоставени само първите две с идеята да бъде представен подхода в изпълнението реализацията на съставянето на клас extension-на.

```
namespace ForexFlow.Models
{
    21 references
    public class SingleCurrencyAmount
    {
        12 references
        public Guid Id { get; set; }

        [Range(0, double.MaxValue)]
        25 references
        public decimal Amount { get; set; }
        8 references
        public string Description { get; set; } = string.Empty;

        [ForeignKey("Currency")]
        20 references
        public string CurrencyCode { get; set; } = string.Empty;
        13 references
        public Currency Currency { get; set; } = new Currency();
    }
}
```

```
namespace ForexFlow.Models.Extensions
{
    0 references
    public static class SingleCurrencyAmountActions
    {
        0 references
        public static void SetAmountFromTransfer(this SingleCurrencyAmount currentAmount, SingleCurrencyAmount transferableAmount, bool fullTransfer = false)
        {
            if (currentAmount.CurrencyCode != transferableAmount.CurrencyCode)
            {
                throw new Exception($"You can't transfer one amount to another, if they are from a different currency.\nYou are trying to transfer the amount {transferableAmount.Amount} from {transferableAmount.CurrencyCode} to {currentAmount.CurrencyCode}");
            }

            currentAmount.Amount = transferableAmount.Amount;

            if (fullTransfer)
            {
                currentAmount.Description = transferableAmount.Description;
            }
        }

        1 reference
        public static void IncreaseAmountByTransfer(this SingleCurrencyAmount currentAmount, SingleCurrencyAmount transferableAmount)
        {
            if (currentAmount.CurrencyCode != transferableAmount.CurrencyCode)
            {
                throw new Exception($"You can't increase one amount by transferring from another, if they are from a different currency.\nYou are trying to increase the amount {transferableAmount.Amount} from {transferableAmount.CurrencyCode} to {currentAmount.CurrencyCode}");
            }

            currentAmount.Amount += transferableAmount.Amount;
            transferableAmount.Amount = 0;
        }
    }
}
```

## Data Access Layer (ForexFlow.DataAccess)

Тъй като в апликацията е използван Entity Framework Package, се появява нуждата от съставянето на ApplicationDbContext клас, който дефинира Entity-тата, съхранявани в базата данни, спрямо съставените модели и описва взаимните връзки между тях.

```

namespace ForexFlow.DataAccess.Database
{
    21 references
    public class ApplicationDbContext : DbContext
    {
        0 references
        public DbSet<Currency> Currencies { get; set; }
        0 references
        public DbSet<SingleCurrencyAmount> SingleCurrencyAmounts { get; set; }
        0 references
        public DbSet<Invoice> Invoices { get; set; }

        1 reference
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) { }

        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<SingleCurrencyAmount>()
                .HasOne(s => s.Currency)
                .WithMany()
                .HasForeignKey(s => s.CurrencyCode)
                .OnDelete(DeleteBehavior.Restrict);

            modelBuilder.Entity<Invoice>()
                .Property(i => i.InvoiceItems)
                .HasConversion(
                    v => JsonSerializer.Serialize(v, new JsonSerializerOptions()),
                    v => JsonSerializer.Deserialize<List<Item>>(v, new JsonSerializerOptions()) ?? new List<Item>()
                );
        }
    }
}

```

На базата на съставения context, се изразява и абстрактното ниво Repository, което ще осъществява комуникацията между Business Logic Layer-а и context-а, записващ данните в базата данни. За целта се съставя един основен клас, който ще опише основните действия, които всеки един модел може притежава в зададената апликация.

```

namespace ForexFlow.DataAccess.Repository.Implementation
{
    7 references
    public class Repository<T> : IRepository<T> where T : class
    {
        private readonly ApplicationDbContext _context;
        internal DbSet<T> dataBaseSet;

        3 references
        public Repository(ApplicationDbContext context)
        {
            this._context = context;
            this.dataBaseSet = this._context.Set<T>();
        }

        4 references
        public void Add(T entity)
        {
            this.dataBaseSet.Add(entity);
        }

        3 references
        public void Delete(T entity)
        {
            this.dataBaseSet.Remove(entity);
        }
    }
}

```

9 references

```

public async Task<T?> Get(Expression<Func<T, bool>>? whereClause = null, string? includeProperties = null, bool tracked = false)
{
    IQueryable<T> entityQuery;
    if (tracked)
    {
        entityQuery = this.dataBaseSet;
    }
    else
    {
        entityQuery = this.dataBaseSet.AsNoTracking();
    }

    if (whereClause != null)
    {
        entityQuery = entityQuery.Where(whereClause);
    }

    if (!string.IsNullOrEmpty(includeProperties))
    {
        foreach (var prop in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
        {
            entityQuery = entityQuery.Include(includeProperties);
        }
    }

    return await entityQuery.FirstOrDefaultAsync();
}

```

8 references

```

public async Task<IEnumerable<T>> GetAll(Expression<Func<T, bool>>? whereClause = null, string? includeProperties = null, bool tracked = false)
{
    IQueryable<T> entityQuery = this.dataBaseSet;

    if (tracked)
    {
        entityQuery = this.dataBaseSet;
    }
    else
    {
        entityQuery = this.dataBaseSet.AsNoTracking();
    }

    if (whereClause != null)
    {
        entityQuery = entityQuery.Where(whereClause);
    }

    if (!string.IsNullOrEmpty(includeProperties))
    {
        foreach (var prop in includeProperties.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
        {
            entityQuery = entityQuery.Include(includeProperties);
        }
    }

    return await entityQuery.ToListAsync();
}

```

Функционалности като Update, макар и да са присъщи за всички модели, могат да варирам като логика, или като цяло да не фигурират (някои модели може да не подлежат на промени). Поради това се съставя и индивидуално repository за всеки един модел, което донадгражда дефиницията на базовото споделено такова. Дори към него да не бъдат описано някаква допълнителна функционалност, това остава възможност, в последствие да бъде съставяни такива, без нарушаването на вече работещата логика на програмата.



```

namespace ForexFlow.DataAccess.Repository.Implementation
{
    2 references
    class SingleCurrencyAmountRepository : Repository<SingleCurrencyAmount>, ISingleCurrencyAmountRepository
    {
        private readonly ApplicationDbContext _context;

        1 reference
        public SingleCurrencyAmountRepository(ApplicationDbContext context) : base(context)
        {
            _context = context;
        }

        6 references
        public void Update(SingleCurrencyAmount amountToUpdate)
        {
            _context.Update(amountToUpdate);
        }
    }
}

```

След съставянето на необходимите repositories, се Data Access Layer-а надгражда с допълнителен финален абстрактен слой UnitOfWork, който има за цел да осигурява достъп до дефинираните repositories, като при настъпването на множество от промени, където и да е по моделите, той да запазва засечените промени едновременно като една обща транзакция.

```

namespace ForexFlow.DataAccess.Repository
{
    2 references
    public class UnitOfWork : IUnitOfWork
    {
        public ApplicationDbContext _context;

        13 references
        public ISingleCurrencyAmountRepository singleCurrencyAmountRepository { get; private set; }
        14 references
        public ICurrencyRepository currencyRepository { get; private set; }
        5 references
        public IInvoiceRepository invoiceRepository { get; private set; }

        0 references
        public UnitOfWork(ApplicationDbContext context)
        {
            _context = context;
            singleCurrencyAmountRepository = new SingleCurrencyAmountRepository(_context);
            currencyRepository = new CurrencyRepository(_context);
            invoiceRepository = new InvoiceRepository(_context);
        }

        11 references
        public async Task SaveTransaction()
        {
            await _context.SaveChangesAsync();
        }
    }
}

```

## Dependency Injection

В модерните приложения, особено тези, следващи архитектурни модели като MVVM (Model-View-ViewModel), е ключово да се управляват зависимостите между различните компоненти. Вместо класовете сами да създават инстанции на обектите, от които зависят (което води до силна обвързаност и затруднява тестването), използваме механизма Dependency Injection (DI). DI контейнерът се грижи за създаването и предоставянето ("инжектирането") на тези зависимости.

Конфигурацията на DI контейнера обикновено се случва при стартиране на приложението, в случая - в метода OnStartup на класа App.

## Регистрация на Услуги (Services):

### IUnitOfWork / UnitOfWork:

**Защо се регистрира?** UnitOfWork капсулира операциите с базата данни. Той осигурява единна точка за достъп до репозиторията и управлява транзакциите. Регистрирането му като услуга (`services.AddTransient<IUnitOfWork, UnitOfWork>()`) позволява лесното му инжектиране в конструкторите на ViewModel-ите, които се нуждаят от достъп до данните.

#### Ползи:

- **Decoupling (Разделяне):** ViewModel-ите зависят от абстракцията (IUnitOfWork), а не от конкретната имплементация (UnitOfWork), което улеснява подмяната или тестването.
- **Управление на жизнения цикъл:** DI контейнерът управлява кога и как се създава инстанция на UnitOfWork. В случая AddTransient означава, че *нова инстанция* ще се създава всеки път, когато се поиска IUnitOfWork. Това гарантира, че различните ViewModel-и (или операции) работят с отделни контексти, ако е необходимо.
- **Тестване:** ViewModel-ите могат лесно да бъдат тествани чрез подаване на *mocks* (*фалшива*) имплементация на IUnitOfWork.

### ViewModel-и (CurrencyManagementViewModel, AmountsManagementViewModel и т.н.)

**Защо се регистрират?** Регистрирането на ViewModel-ите като услуги (`services.AddTransient<CurrencyManagementViewModel>()`) позволява на DI контейнера автоматично да им инжектира техните собствени зависимости (както IUnitOfWork) чрез конструктора.

#### Ползи:

- **Автоматично разрешаване на зависимости:** Не е нужно ръчно да създаваме UnitOfWork и да го подаваме на ViewModel-а. DI контейнерът прави това.
- **Консистентност:** Осигурява стандартизиран начин за създаване на ViewModel-и.
- **Управление на жизнения цикъл:** Отново, AddTransient означава, че при всяко поискване (например при отваряне на нов прозорец/изглед, който използва този ViewModel) ще се създава нова инстанция на ViewModel-а със свежо състояние и собствена (нова) инстанция на IUnitOfWork.

### Фабрика за SingleAmountActionsViewModel:

**Защо се регистрира по-сложно?** SingleAmountActionsViewModel изисква не само инжектирани услуги (IUnitOfWork), но и параметър (*amountId*), който се подава по време на изпълнение. Това е породена от бизнес логиката на приложението и поради това стандартната регистрация не може правилно да обработи това.

**Как работи?** Регистрираме *Func<Guid, SingleAmountActionsViewModel>*. Това е *фабрика* - функция, която DI контейнерът може да предостави. Когато някой поиска тази функция и я извика с конкретно Guid, кодът във фабриката (*provider => (amountId) => { ... }*) ще:

- Използва *provider* (което е *IServiceProvider*), за да поиска нужната зависимост (*IUnitOfWork*).
- Създаде нова инстанция на *SingleAmountActionsViewModel*, подавайки му както разрешената зависимост (*unitOfWork*), така и получения *amountId*.

**Ползи:** Позволява DI да управлява част от зависимостите, докато данните, нужни по време на изпълнение, се подават ръчно при извикване на фабриката.

```
namespace ForexFlow;
8 references
public partial class App : Application
{
    6 references
    public static IServiceProvider ServiceProvider { get; private set; }

    0 references
    protected override void OnStartup(StartupEventArgs e)
    {
        var services = new ServiceCollection();

        string exePath = AppContext.BaseDirectory;
        string solutionRootDir = Path.GetFullPath(Path.Combine(exePath, @"..\..\..\..\"));

        var databasePath = Path.Combine(solutionRootDir, "ForexFlow.DataAccess", "Database", "currency_system.db");

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlite($"Data Source={databasePath}"));

        services.AddTransient<IUnitOfWork, UnitOfWork>();

        services.AddTransient<CurrencyManagementViewModel>();
        services.AddTransient<AmountsManagementViewModel>();
        services.AddTransient<InvoiceManagementViewModel>();
        services.AddTransient<Func<Guid, SingleAmountActionsViewModel>>(provider => (amountId) =>
        {
            var unitOfWork = provider.GetRequiredService<IUnitOfWork>();
            return new SingleAmountActionsViewModel(unitOfWork, amountId);
        });

        ServiceProvider = services.BuildServiceProvider();

        var mainWindow = new MainWindow();
        mainWindow.Show();
    }
}
```

## Presentation Layer (ForexFlow.View)

Ето и как се осъществява инжектирането на необходимите дефинирани *services* от страна на дадените *view*-та.

```

namespace ForexFlow.View
{
    4 references
    public partial class CurrencyManagement : Window
    {
        private readonly CurrencyManagementViewModel _sharedContext;

        1 reference
        public CurrencyManagement(CurrencyManagementViewModel currencyManagementViewModel)
        {
            InitializeComponent();
            DataContext = currencyManagementViewModel;
            _sharedContext = currencyManagementViewModel;
        }

        1 reference
        private void OpenAddForm(object sender, RoutedEventArgs e)
        {
            AddCurrencyForm currencyManagement = new AddCurrencyForm(_sharedContext);
            currencyManagement.Show();
        }

        1 reference
        private void OnBack(object sender, RoutedEventArgs e)
        {
            MainWindow homeMenu = new MainWindow();
            homeMenu.Show();
            this.Close();
        }
    }
}

```

При View-та, които споделят общ ViewModel, поради смислова обща бизнес логика, инстанция му в „главния“ View (към които останалите са свързани смислово) просто се подава като параметър към тях, *както е при AddCurrencyForm*.

Ето как се използва и „специално“ дефинирания service в призованото View:

```

namespace ForexFlow.View
{
    3 references
    public partial class SingleAmountActions : Window
    {
        1 reference
        public SingleAmountActions(Guid amountId)
        {
            InitializeComponent();

            var viewModelFactory = App.ServiceProvider.GetRequiredService<Func<Guid, SingleAmountActionsViewModel>>();
            DataContext = viewModelFactory(amountId);
        }

        1 reference
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            var amountsManagementViewModel = App.ServiceProvider.GetService(typeof(AmountsManagementViewModel)) as AmountsManagementViewModel;
            AmountsManagement amountsManagement = new AmountsManagement(amountsManagementViewModel);
            amountsManagement.Show();
            this.Close();
        }
    }
}

```

## Application Layer (ForexFlow.ViewModel)

Тъй като сами по себе си отделните ViewModel-и взаимодействат по сходен начин с UnitOfWork и предоставят необходимите промените по UI интерфейса на View-тата, ще бъде разгледан само един от тях в общия смисъл на това как се реализира посочените функционалности.

### Взаимодействие с UnitOfWork

1. **Инжектиране на зависимост:** ViewModel-ът получава инстанция на IUnitOfWork чрез конструктора си. Това му позволява да комуникира с базата данни, без да има пряка зависимост от конкретната имплементация на достъпа до данни.
2. **Зареждане на данни:**
  - Методите LoadInvoices() и LoadCurrencies() използват асинхронно repository-тата (invoiceRepository, currencyRepository) от \_unitOfWork, за да извлекат съответните данни от базата.
  - Извлечените данни се зареждат в ObservableCollection-и (FetchedInvoices, Currencies), които са достъпни за View-то.
3. **Модифициране на данни:**
  - Методът SaveInvoice() използва \_unitOfWork.invoiceRepository.Add() за добавяне на нова фактура.
  - Методът DeleteInvoice() използва \_unitOfWork.invoiceRepository.Delete() за изтриване на съществуваща фактура.
  - След операции по добавяне или изтриване, \_unitOfWork.SaveTransaction() се извиква, за да се запишат (commit) промените в базата данни транзакционно.

### Взаимодействие с View-то

1. **Data Binding (Свързване на данни):**
  - **Колекции:** Публичните ObservableCollection свойства (*FetchedInvoices*, *FilteredInvoices*, *InvoiceItemsHolder*, *Currencies*) са предназначени за data binding към елементи в UI (камо DataGridView, ComboBox, ItemsControl). ObservableCollection уведомява View-то автоматично при добавяне или премахване на елементи.
  - **Състояние и Данни за Форми:** Свойствата в InvoiceActionsPayload (SelectedCurrency, ComposedInvoice, NewInvoiceItem, SearchValue, EditingMode, InvoicesSource) се bind-ват към контроли във View-то (например TextBox, ComboBox, CheckBox). InvoiceActionsPayload имплементира INotifyPropertyChanged, което позволява на ViewModel-а да уведоми View-то, когато стойността на някое от тези свойства се промени, за да може UI да се обнови. InvoicesSource динамично превключва източника на данни за списъка с фактури във View-то между всички (FetchedInvoices) и филтрираните (FilteredInvoices).

## 2. Commands (Команди):

- Публичните ICommand свойства (AddNewItemToInvoiceCommand, RemoveItemFromInvoiceCommand, SaveInvoiceCommand и т.н.) се bind-ват към интерактивни елементи във View-то (най-често бутони).
- Те капсулират потребителските действия. Когато потребителят взаимодейства с бинднатия UI елемент (напр. кликне бутон), се изпълнява съответният метод във ViewModel-а (AddItemToInvoice, SaveInvoice и т.н.).
- Командите (чрез RelayCommand/RelayCommandObject) управляват и състоянието CanExecute, което динамично определя дали свързаният UI елемент е активен (enabled) или не, базирано на логика във ViewModel-а (напр., canSave()). Промени в състоянието, които влияят на CanExecute, се сигнализират чрез RaiseCanExecuteChanged().

## 3. Уведомяване за промени:

- Чрез имплементацията на INotifyPropertyChanged (директно или в InvoiceActionsPayload), ViewModel-ът уведомява View-то за промени в данните, които не са в ObservableCollection.
- Събитията PropertyChanged в конструктора се използват за задействане на вътрешна логика (като филтриране при промяна на SearchValue или преизчисляване на CanExecute при промяна на NewInvoiceItem).

В обобщение, InvoiceManagementViewModel оркестрира потока на данни: извлича данни от IUnitOfWork, представя ги на View-то чрез data binding, обработва потребителски действия чрез команди и записва промените обратно в базата данни чрез IUnitOfWork.

```
namespace ForexFlow.ViewModel.ViewModels
{
    10 references
    public class InvoiceManagementViewModel
    {
        #region Data Initialization & Fields
        private readonly IUnitOfWork _unitOfWork;
        7 references
        public ObservableCollection<Invoice> FetchedInvoices { get; set; } = new ObservableCollection<Invoice>();
        5 references
        public ObservableCollection<Invoice> FilteredInvoices { get; set; } = new ObservableCollection<Invoice>();
        11 references
        public ObservableCollection<NewItemNotifier> InvoiceItemsHolder { get; set; } = new ObservableCollection<NewItemNotifier>();
        5 references
        public ObservableCollection<Currency> Currencies { get; set; } = new ObservableCollection<Currency>();
        63 references
        public InvoiceActionsPayload ActionsPayload { get; set; } = new InvoiceActionsPayload();

        2 references
        public ICommand AddNewItemToInvoiceCommand { get; set; }
        1 reference
        public ICommand RemoveItemFromInvoiceCommand { get; set; }
        2 references
        public ICommand SaveInvoiceCommand { get; set; }
        1 reference
        public ICommand PreviewInvoiceCommand { get; set; }
        1 reference
        public ICommand DeleteInvoiceCommand { get; set; }
    }
}
```

0 references

```
public InvoiceManagementViewModel(IUnitOfWork unitOfWork)
{
    _unitOfWork = unitOfWork;

    AddNewItemToInvoiceCommand = new RelayCommand(AddItemToInvoice, () => ActionsPayload.NewInvoiceItem.Quantity >= 1 && ActionsPayload.NewInvoiceItem.De
    RemoveItemFromInvoiceCommand = new RelayCommandObject(RemoveItemFromInvoice, () => true);
    SaveInvoiceCommand = new RelayCommand(SaveInvoice, () => canSave());
    PreviewInvoiceCommand = new RelayCommandObject(PreviewInvoice, () => true);
    DeleteInvoiceCommand = new RelayCommandObject(DeleteInvoice, () => true);

    ActionsPayload.NewInvoiceItem.PropertyChanged += (s, e) =>
    {
        ((RelayCommand)AddNewItemToInvoiceCommand).RaiseCanExecuteChanged();
    };

    ActionsPayload.PropertyChanged += (s, e) =>
    {
        if (e.PropertyName == nameof(ActionsPayload.SelectedCurrency))
        {
            ChangeRepresentation();
        }

        if (e.PropertyName == nameof(ActionsPayload.SearchValue))
        {
            FilterResults();
        }
    };

    LoadCurrencies();
    LoadInvoices();

    ActionsPayload.InvoicesSource = FetchedInvoices;
}
```

1 reference

```
private async void LoadInvoices()
{
    FetchedInvoices.Clear();

    var invoices = await _unitOfWork.invoiceRepository.GetAll();

    foreach (var invoice in invoices)
    {
        FetchedInvoices.Add(invoice);
    }
}
```

1 reference

```
private async void LoadCurrencies()
{
    var currencies = await _unitOfWork.currencyRepository.GetAll();

    if (currencies.Any())
    {
        Currencies.Clear();
        foreach (var currency in currencies)
        {
            Currencies.Add(currency);
        }
    }

    ActionsPayload.SelectedCurrency = Currencies[0];
}

#endregion
```



```

1 reference
private async void DeleteInvoice(object invoice)
{
    var invoiceToDelete = invoice as Invoice;
    if (invoiceToDelete != null)
    {
        _unitOfWork.invoiceRepository.Delete(invoiceToDelete);
        await _unitOfWork.SaveTransaction();

        FetchedInvoices.Remove(invoiceToDelete);
        FilteredInvoices.Remove(invoiceToDelete);
    }
}

```

```

1 reference
private async void SaveInvoice()
{
    foreach (var item in InvoiceItemsHolder) {
        ActionsPayload.ComposedInvoice.InvoiceItems.Add(new Item
        {
            Quantity = item.Quantity,
            Description = item.Description,
            UnitPrice = item.UnitPrice,
            TotalAmount = item.TotalAmount,
        });
    }

    _unitOfWork.invoiceRepository.Add(ActionsPayload.ComposedInvoice);

    await _unitOfWork.SaveTransaction();

    FetchedInvoices.Add(ActionsPayload.ComposedInvoice);
    InvoiceComposerCleanUp();
}

```

### ~ InvoiceActionsPayload DTO

```

namespace ForexFlow.ViewModel.DataTransferObjects
{
    2 references
    public class InvoiceActionsPayload : INotifyPropertyChanged
    {
        private ObservableCollection<Invoice> _invoicesSource = new ObservableCollection<Invoice>();
        private Currency _selectedCurrency;
        private Invoice _composedInvoice = new Invoice();
        private NewItemNotifier newInvoiceItem = new NewItemNotifier();
        5 references
        public bool EditingMode { get; set; } = false;
        private string searchValue = string.Empty;

        4 references
        public ObservableCollection<Invoice> InvoicesSource
        {
            get { return _invoicesSource; }
            set
            {
                _invoicesSource = value;
                OnPropertyChanged(nameof(InvoicesSource));
            }
        }
    }
}

```



13 references

```
public Currency SelectedCurrency
{
    get { return _selectedCurrency; }
    set
    {
        _selectedCurrency = value;
        OnPropertyChanged(nameof(SelectedCurrency));
    }
}
```

27 references

```
public Invoice ComposedInvoice
{
    get { return _composedInvoice; }
    set {
        _composedInvoice = value;
        OnPropertyChanged(nameof(ComposedInvoice));
    }
}
```

13 references

```
publicNewItemNotifier NewInvoiceItem
{
    get { return newInvoiceItem; }
    set
    {
        newInvoiceItem = value;
        OnPropertyChanged(nameof(NewInvoiceItem));
    }
}
```

5 references

```
public string SearchValue
{
    get { return searchValue; }
    set
    {
        searchValue = value;
        OnPropertyChanged(nameof(SearchValue));
    }
}
```

```
public event PropertyChangedEventHandler? PropertyChanged;
```

5 references

```
protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
```

## Presentation Layer (ForexFlow.View)


В тази секция ще бъдат представени основните екрани както и функционалностите, всеки от които предлага. Както бе уточнено всеки екран използва свой собствен ViewModel като data context и се bind-ва/“закача“ за определени property-та, принадлежащи му, с цел визуализация и интеракция с данните в database-а. Има екрани, които взаимно споделят смислово еднаква концепция към приложението и се явяват „под-екран“ на основен такъв и поради това, макар да са представени като отделни прозорци и изградени като отделни View-та, те споделят общ ViewModel като източник на данни.

### Currency Management

Тук потребителя има възможност да достъпи всички валути фигуриращи в database, като му се представя актуалната и пълна информация за тях. В дясно на екрана е разположено меню за настройки, от където потребителя при необходимост може да добави нова валута, или да модифицира курса на стара такава. Операцията за изтриване е ограничена до административно ниво с database-а, поради избягването на ненужното усложняване на приложението.

За да бъде добавена нова валута, при натискането на бутона *Add New Currency* се появява мини-форма, в която трябва да бъдат попълнени исканите данни за да може тя да бъде създадена. Ако процесът е бил успешен новата валута ще бъде добавена автоматично във визуалния списък при останалите.

За да бъде модифициран курса на дадена валута, тя трябва да бъде селектирана чрез бутона в горния десен ъгъл на всеки UI род и автоматично нейната информация ще се появи в полето за модификация в менюто.

 **Currency Display**

**BGN**

1.78 лв

exchange rate

Bulgarian Lev

**JPY**

150.25 ¥

exchange rate

Japanese Yen

**EUR**

0.92 €

exchange rate

Euro

**USD**

1.0 \$

exchange rate

US Dollar

**GBP**

0.78 £

exchange rate

British Pound

**KRW**

1350.5 ₩

exchange rate

South Korean Won

**Setting Panel**

Modify the currency archive by adding a new one or update the exchange range of the already existings ones.

Add New Currency

Currently Editing:


exchange rate

0

Save Changes

## Amounts Management

Подобно на currency management в този прозорец е налична репрезентация на всички amount entities запазени в database-а. От тук директно може да бъде добавян и нов запис, като той също автоматично ще бъде и добавен в списъка от вече визуално представените такива, ако процесът е бил успешен.

 **Add New Amount**

Description:  Amount:

**BGN 79.90 лв**  
The first test amount added from the UI.

Actions

**JPY 160.00 ¥**  
Another test amount added from the UI.

Actions

**EUR 500.90 €**  
Yet another test amount.

Actions

**KRW 45 00 ₩**

От тук за всеки amount може да бъде отворен специален прозорец, който позволява прилагането на набор от модификации и операции върху избрания такъв, като това става чрез бутона *Actions*.

## Single Amount Actions

В това View потребителят може да менажира дадения amount по множество начини Включващи:

- **Добавяне/изваждане** фиксирана стойност към текущия amount.
- **Правене на транзакции:** представен е списък от странични съществуващи amounts, споделящи същата валута. Съответно чрез избирането на една от тях може да бъде направена цялостна транзакция от единия amount към другия. Този процес може да бъде двупосочен в зависимост от избрания direction mode.
- **Представяне в различни валути:** чрез употребата на модела *MultiCurrencyAmount* модела, по избор на потребителя могат да бъдат направени репрезентации на колко се приравнява текущия amount към друга валута, без да я модифицира. Тези репрезентации не се запазват и при излизането от View-то изчезват.

Go Back

JPY

160.00 ¥

Another test amount added from the UI.

f8f0773b-da19-423c-b5e6-92166ef1f900

Fixed Amount Modification

Increase or decrease the current state of the amount with a provided fixed value. Note that if you tend to decrease the amount with greater value than the current one it will be set to 0, rather than negative.

-

5.00

+

Transactions

The transactions allow you to select a set of amounts to be transferred to the given one. Note that when transferring one amount to another you reduce the first one to 0.

☐ 5.06 JPY: Korean test amount.

☒ 25000.00 JPY: Japanese amount.

Change Direction

Add

Transfer

Multi Currency Amount Representations

This section lets you preview the equivalent value of a given amount in different currencies (before any potential conversion) using the MultiCurrencyAmount model.

На гъното на това View са достъпни и операциите на изтриване – премахващ изцяло amount-та от data base-а и конвертиране – при избирането на дадена валута amount-та explicit-но се конвертира спрямо зададения валутен курс.

## Invoice Management

Подобно на останалите два други прозореца в тази секция потребителя може да намери всички фактури съставяни в приложението, като на всеки slot, е предоставена основната информация, с която могат да се различат самите фактури, а именно:

- Име на издаващия фактурата : Име на получаващия фактурата
- Номер на фактурата
- Дата на издаване

В този прозорец е и интегрирана функционалността за търсене на фактура, като то е възможно именно на база посочените горе данни визуализирани в slot-та на представяне.

От този прозорец потребителят може да отвори и специална форма *InvoiceComposer*, в който да състави нова фактура. Вече запазените такива пък могат да бъдат разгледани в подробност и пълен детайл, чрез бутона Preview съответстващ на всяка една от тях.

От тук съответно могат да бъдат изтривани фактурите чрез бутона Delete.

**Invoices**

Search: 

Compose Invoice

Steam : Александър Ганчев

09.04.2025 6c801fe5-d513-475a-a4c0-d01ealf9fef9

Delete

Preview

Online Bookstore Inc. : Alice Wonderland

10.05.2025 d4ccd473-55dc-4de2-be6f-989e8efe01b7

Delete

Preview

CloudHost Pro : Bob The Builder

01.06.2025 724aa8c6-9068-456b-a408-fc847b360331

Delete

Preview

Local Hardware Supplies : Charlie Fixit

Delete

Preview

Формата Invoice Composer представлява токъ и модел на това как фактурата трябва да изглежда, като потребителя трябва да попълни полетата с данни за да може да запази фактурата. По всяко едно време, чрез бутона Preview потребителя може да види на този етап как фактурата реално ще бъде генерирана и ако всичко е попълнено правилно и потребителя е доволен той може да натисне бутона Save за да запази фактурата в database-а. Погорно на останалите прозорци, ако е успешно добавянето фактурата автоматично ще бъде добавена към вече заредените визуално такива в списъка. Важно е да се подчертае, че на база предоставените описани елементи във фактурата програмата автоматично ще сметне тоталните разходи, както и ще наложи ДДС начислението и евентуални намаления на база бройка на елементите. Също така при смяна на избраната валута, цените на всички елементи ще бъдат преизчислени на база избрания валутен курс.

INVOICE

Cancel

Preview Invoice

Company Name:

BGN

Company Address:

Company City, Country/State:

BILL TO

SHIP TO

Name:

Address:

City, Country/State:

QTY	Description	Unit Price	Total Amount
0		0	
10	Some test item	10.0 лв	100.0 лв

INVOICE

Back

Save Invoice

Steam

Bellevue, 10400 NE 4th St

United States

BILL TO

SHIP TO

Александър Ганчев

Александър Ганчев

Студентски град, 14

Студентски град, 14

София, България

София, България

INVOICE #

6c801fe5-d513-475a-a4c0-d01ealf9fef9

INVOICE DATE

09.04.2025

DUE DATE

24.04.2025

QTY	Description	Unit Price	Total Amount
1000	Marvel Rivals: Latties Virtual Currency	0.01 €	10.00 €
2	Destiny 2: Final Shape - DLC	55.19 €	110.38 €
Subtotal			120.38 €
Discount before VAT			- 0.70 €
Sales VAT Tax 20%			23.94 €
TOTAL			143.62 €

## Заклучение и място за подобрения

Въпреки че проектът реализира основната си функционалност успешно и следва съвременни принципи на разработка, съществуват аспекти, които умишлено са опростени с цел поддържане на ясен и управляем обем. Например, моделът Invoice в базата данни би могъл да бъде декомпозиран на по-малки, отговорни обекти като Sender, Issuer, Recipient и InvoiceItem, което би улеснило бъдещо разширяване и поддръжка. Допълнително, макар че основни CRUD операции са налични, те могат да бъдат разширени и задълбочени, особено в частта по обработка на данни и потребителски взаимодействия. Не е реализиран и пълен notification механизъм – потребителят получава базова обратна връзка, но липсва по-прецизна информация за успеха или провала на действията, както и причините за това.

Важно е да се отбележи, че тези решения не са пропуски, а осъзнати компромиси с цел да се избегне прекомерна сложност на този етап от проекта. Структурата на приложението следва модерни архитектурни практики – включително разделение по MVVM, използване на dependency injection и ясна организация на слоевете – което го прави лесно разширимо, тестируемо и подходящо за бъдещо доразвиване. Настоящият дизайн създава стабилна основа, върху която могат да бъдат надградени както по-комплексни бизнес логики, така и подобрения в потребителското изживяване.