**Содержание**

1. **Проектирование предметной области**
   1. Описание предметной области
   2. Создание диаграммы классов
   3. Создание GitHub-репозитории проекта
2. **Введение в ASP.NET Web API**
   1. Организация структуры проекта по Clean Architecture
   2. Создание моделей приложения
3. **Работа с базой данных**
   1. Реализация архитектурного паттерна репозиторий
   2. Использование EntityFramework для работы с базой данных
   3. Миграция данных
4. **Реализация методов контроллеров для обработки HTTP-запросов**
   1. Создание сервисов
   2. Создание DTO
   3. Маппинг объектов с использованием библиотеки AutoMapper
   4. Создание методов контроллеров для CRUD операций
5. **Проектирование предметной области**
   1. Описание предметной области

Система управления ресторана - это комплекс программных и аппаратных инструментов, разработанных для эффективного управления всеми аспектами ресторанного бизнеса. Она включает в себя функции управления персоналом, , заказами, платежами, бронированием столов, а также другие функциональности которые способствуют повышения прибыльности ресторана.

* 1. Создание диаграммы классов

Создания диаграммы классов системы управления рестораном был начать с определения основных классов, которые будут представлять различные аспекты системы. Пример классов:

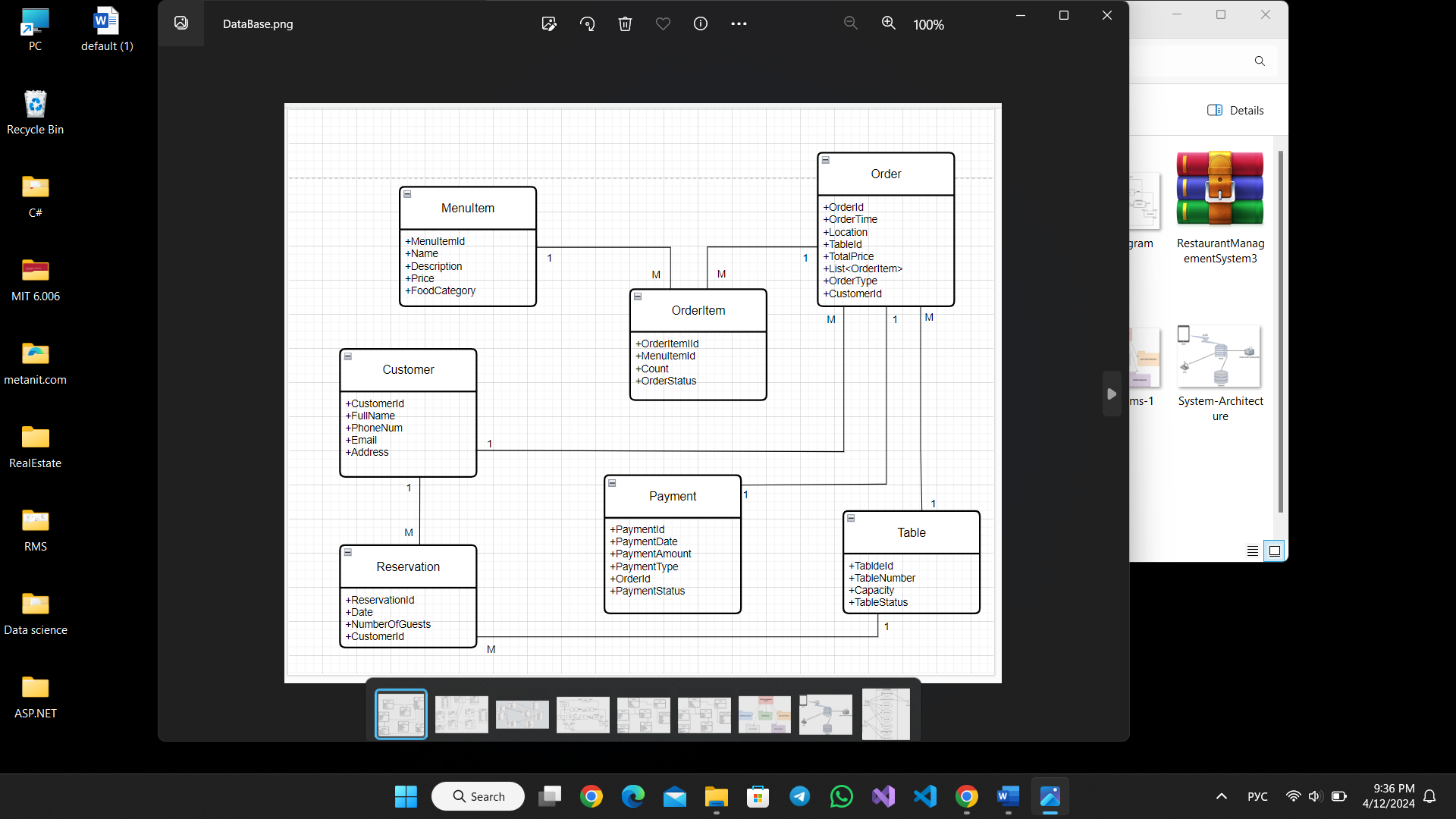
**Меню:** Этот класс будет отражать меню ресторана и включать в себя информацию о блюдах, их ценах, категориях и т.д.

**Заказ:** Класс, представляющий заказы, размещенные клиентами. Он будет содержать информацию о блюдах, заказанных клиентами, статусе заказа, общей сумме и т.д.

**Столы:** Этот класс будет отображать столы в ресторане и содержать информацию о номере стола, количестве мест, его статусе (свободен, забронирован, занят и т.д.).

**Оплата:** Класс, представляющий процесс оплаты заказов клиентами. Он может содержать информацию о способах оплаты, сумме, времени оплаты и т.д.

**Клиент:** Этот класс будет представлять клиентов, посещающих ресторан. Он может содержать информацию о имени клиента, контактной информации, предпочтениях истории заказов и т.д.



**Рис 1. Диаграмма классов**

* 1. Создание GitHub-репозитории проекта

Краткий процесс создания GitHub-репозитория для проекта под названием "RestaurantManagementSystem”.

Создание нового репозитория: после входа в аккаунт, была нажата на кнопку "New" (Новый) в верхнем правом углу страницы репозиториев.

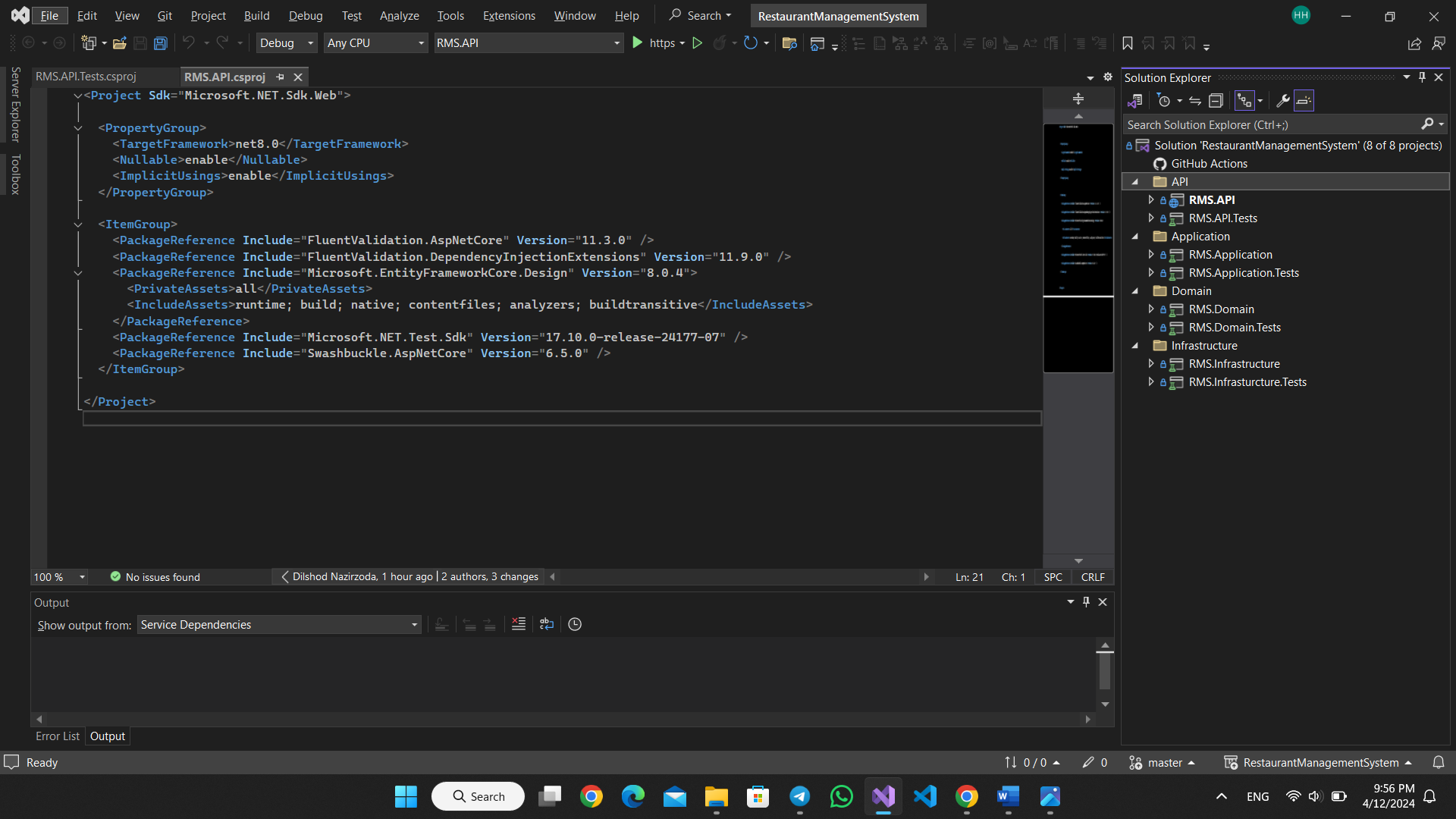
Заполнение информации о репозитории: на странице создания нового репозитория была указана название проекта ("RestaurantManagementSystem"), описание проекта (например, "Система управления рестораном"), было выбрано уровень доступа (**общедоступный** или приватный), а также другие настройки по нашему усмотрению.

Создание репозитория: была нажата на кнопку "Create repository" (Создать репозиторий) для завершения создания репозитория. Клонирование репозитория: после создания репозитория скопирован его URL. Затем открыв терминал на вашем компьютере, перейден в папку, где сохранен проект, и было выполнено команда `git clone <https://github.com/huseynhaydarov/RestaurantManagementSystem.git >` для клонирования репозитория в наш компьютер.

1. **Введение в ASP.NET Web API**
   1. Организация структуры проекта по Clean Architecture

Clean Architecture - это методология проектирования программного обеспечения, предложенная Робертом Мартином (также известным как Uncle Bob). Она призвана обеспечить высокую степень разделения ответственностей и улучшить расширяемость, тестируемость и общую чистоту кода. Вот основные принципы, которые лежат в основе Clean Architecture:

1. **Независимость от фреймворков**: Бизнес-логика не должна зависеть от каких-либо конкретных технологий или фреймворков.
2. **Разделение ответственностей**: Различные компоненты системы должны быть разделены на уровни, каждый из которых отвечает за определенные аспекты приложения.
3. **Принцип единственной ответственности (Single Responsibility Principle, SRP)**: Каждый класс или модуль должен быть ответственен только за одну вещь.
4. **Принцип инверсии зависимостей (Dependency Inversion Principle, DIP)**: Зависимости внутри системы должны быть направлены от абстракций к деталям, а не наоборот.
5. **Использование чистой архитектуры**: Разработка должна основываться на чистой архитектуре, которая позволяет легко изменять и поддерживать код.



**Рис 2. Организация структуры проекта по Clean Architecture**

* 1. Создание моделей приложения

Было создано все модели базы данных исходя из схема диаграмма классов (рис. 1).

Листинг 1. Код на языке программирования с# (Сustomer Entity).

public class Customer : EntityBase

{

public string? FullName { get; set; }

public string? Email { get; set; }

public string? Address { get; set; }

public string? PhoneNumber { get; set; }

public ICollection<Order>? Orders { get; set; }

public ICollection<Reservation>? Reservations { get; set; }

}

Листинг 1. Код на языке программирования с# (Order Entity).

public class Order : EntityBase

{

public DateTime OrderTime { get; set; }

public string? Location { get; set; }

public decimal TotalPrice { get; set; }

public virtual List<OrderItem>? Items { get; set; }

public OrderType Type { get; set; }

public Customer? Customer { get; set; }

public int CustomerId { get; set; }

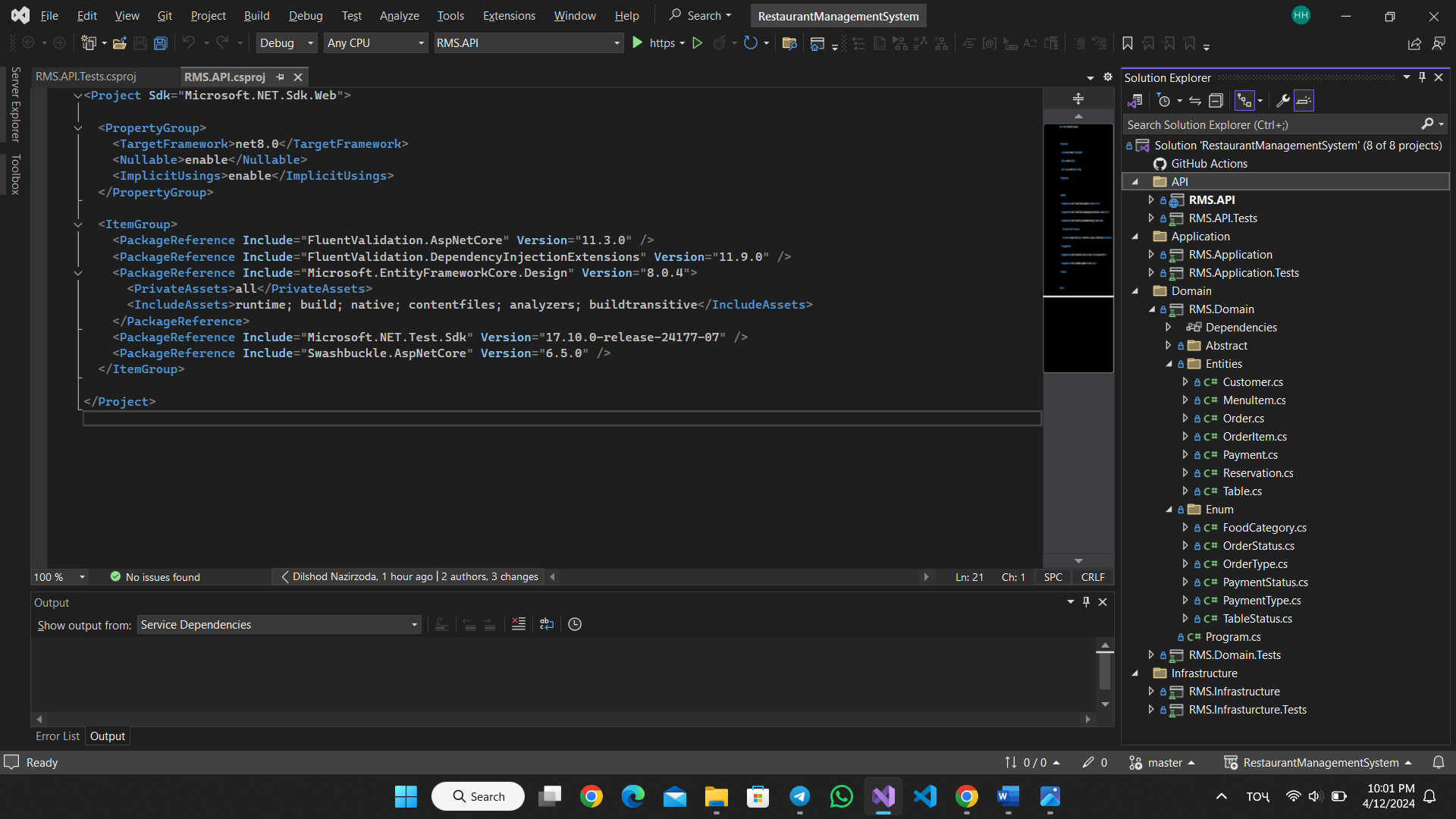
public Table? Table { get; set; }

public int TableId { get; set; }

public Payment? Payment { get; set; }

public ICollection<Table>? Tables { get; set; }

}



**Рис 3. Cтруктура моделей проекта**

1. **Работа с базой данных**
   1. Реализация архитектурного паттерна репозиторий

Архитектурный паттерн репозиторий является часто используемым в приложениях, особенно там, где требуется взаимодействие с хранилищем данных, таким как база данных. Вот как можно реализовать этот паттерн:

**Определение интерфейса репозитория**:

Создайте интерфейс, который будет определять методы для основных операций доступа к данным, таких как добавление, чтение, обновление и удаление (CRUD).

**Листинг 3. Определение интерфейса репозитория:**

{

public interface IBaseService<TEntity, TResponseModel, TRequestModel>

where TEntity : EntityBase

where TRequestModel : BaseRequest

where TResponseModel : BaseResponse

{

IEnumerable<TResponseModel> GetAll();

TResponseModel Get(int id);

void Add(TRequestModel request);

TRequestModel Update(int id, TRequestModel request);

bool Delete(int id);

}

}

**Листинг 4. Создание конкретной реализации репозитория**:

public class CustomerRepository : ICustomerRepository

{

public Task<Customer> Create(Customer entity)

{

throw new NotImplementedException();

}

public Task<Customer> Update(Customer entity)

{

throw new NotImplementedException();

}

public Task<Customer> Delete(int id)

{

throw new NotImplementedException();

}

public Task<IEnumerable<Customer>> GetAll()

{

throw new NotImplementedException();

}

public Task<Customer> GetById(int id)

{

throw new NotImplementedException();

}

Task IBaseRepository<Customer>.Delete(int id)

{

throw new NotImplementedException();

}

}

}

* 1. Использование EntityFramework для работы с базой данных

Entity Framework (EF) — это набор технологий в Microsoft .NET Framework, который позволяет разработчикам работать с данными в виде объектов и свойств, без прямой работы с базой данных. EF позволяет создавать схему базы данных, основанную на моделях объектов приложения, и автоматически генерировать SQL-код для взаимодействия с базой данных.

**Листинг 5. Пример создания контекста данных и выполнения запроса:**

public class EFContext : DbContext

{

public EFContext(DbContextOptions<EFContext> options) : base(options)

{

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseNpgsql("Host=localhost;Port=5432;Database=RMS;Username=postgres;Password=7878\_data\_base");

base.OnConfiguring(optionsBuilder);

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Ignore<EntityBase>();

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.CustomerConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.MenuItemConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.OrderItemConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.OrderItemConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.PaymentConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.ReservationConfiguration).Assembly);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(TablesConfiguration.TableConfiguration).Assembly);

}

}

**Листинг 5. Пример создания контекста данных и выполнения запроса:**

public class EFContextFactory : IDesignTimeDbContextFactory<EFContext>

{

public EFContext CreateDbContext(string[] args)

{

var optionBuilder = new DbContextOptionsBuilder<EFContext>();

optionBuilder.UseNpgsql("Host=localhost;Port=5432;Database=RMS;Username=postgres;Password=7878\_data\_base");

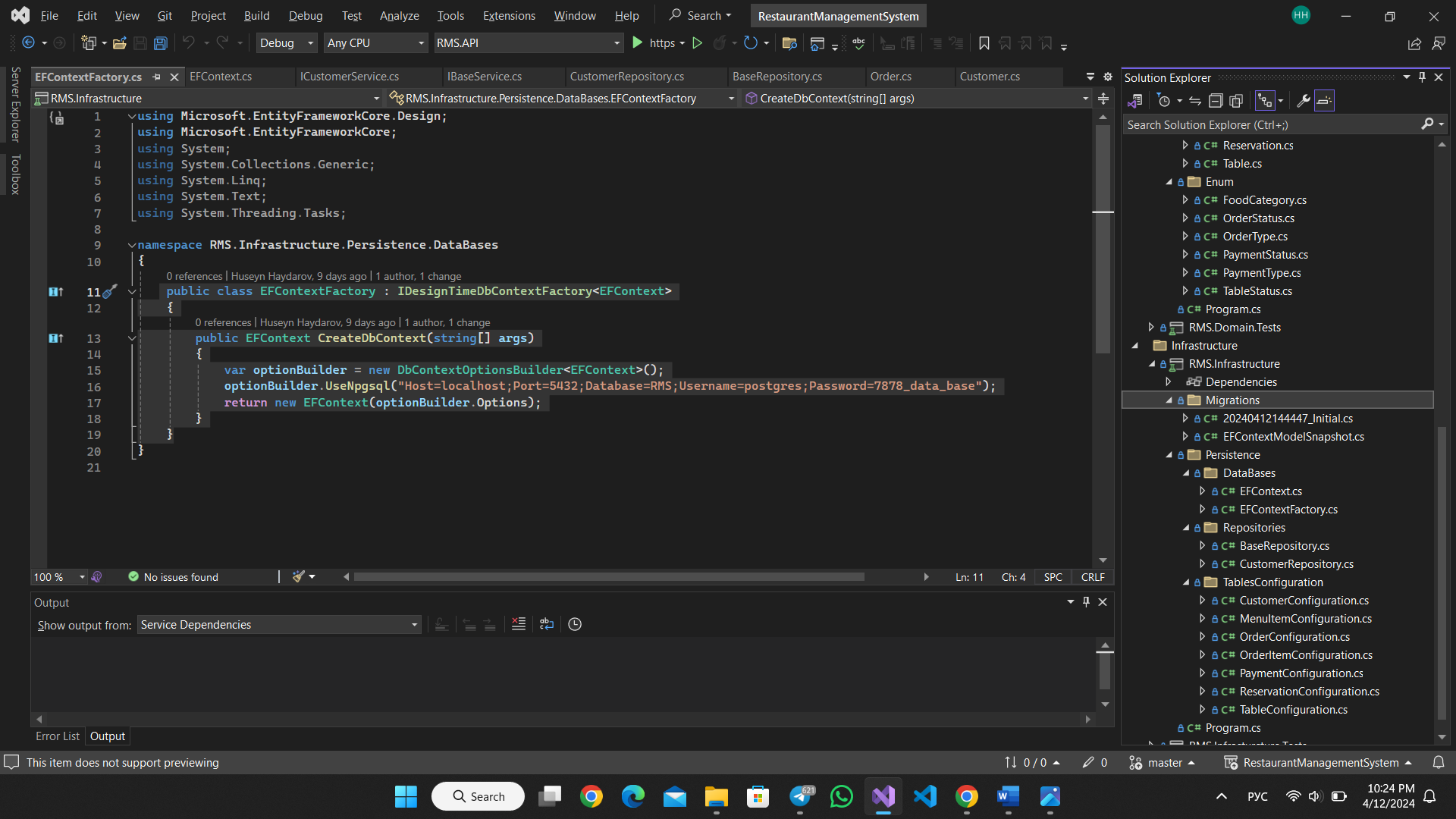
return new EFContext(optionBuilder.Options);

}

}

* 1. Миграция данных

После того как все конфигурации для всех моделей были написан, можно набрав команду add-migration CreateInitialize создать классы миграции.



**Рис 4. Классы миграции проекта**