

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

Кафедра информационных систем и программной инженерии

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Методические указания

Составители:
Студент гр. ПРИ-120
Илларионов А.М.

Доц. кафедры ИСПИ
Вершинин В.В.

Владимир 2021-2023 г.

Содержание

Оглавление

Содержание	2
Лабораторная работа 3. Разработка микросервиса, как элемента микросервисной архитектуры	3
Начало работы.....	9
Создание проекта	11
Создание модели микросервиса «Студент»	18
Создание базы данных и её контекста	21
Создание контроллера микросервиса студентов.....	28
Тестирование запросов с помощью Postman	32
Создание микросервиса Course.	39
Взаимодействие микросервисов между собой	47
Создание композитного микросервиса	53
Тестирование работы приложения.....	57
Заключение	59

Лабораторная работа 3.

Разработка микросервиса, как элемента микросервисной архитектуры

Цель работы:

Познакомиться с микросервисной архитектурой и освоить принципы разработки микросервиса, как базового элемента данной архитектуры, в среде .NET

Общие сведения:

Микросервисная архитектура – это подход к разработке приложения в виде набора небольших, независимо развертываемых и слабосвязанных сервисов, каждый из которых выполняет свою функцию/обязанность. Для того, чтобы лучше понять этот стиль, будет полезно сравнить его с монолитной архитектурой на примере приложения доставки еды FTGO. На рис. 1 – вид приложения в монолитной архитектуре.

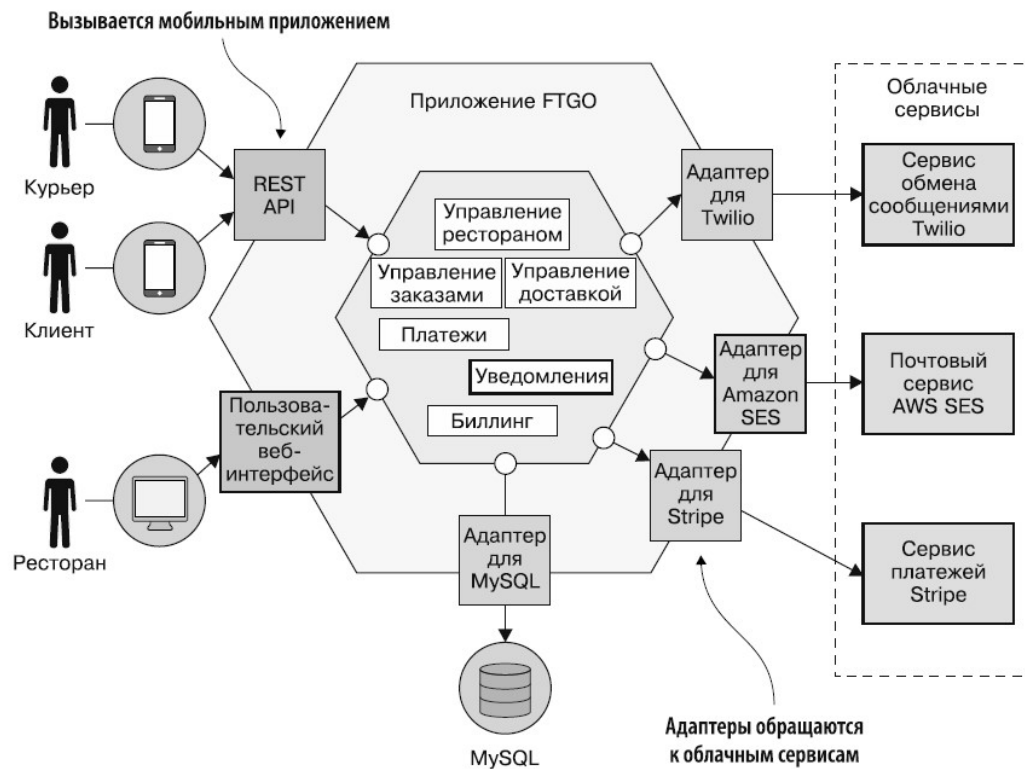


Рис. 1 - Приложение FTGO имеет гексагональную монолитную архитектуру. Оно состоит из бизнес-логики, окруженной адаптерами, которые реализуют пользовательские интерфейсы и взаимодействуют с внешними системами, например мобильными клиентами и облачными сервисами для платежей, электронной почты и обмена сообщениями

Недостатки монолитной архитектуры:

- *Высокая сложность* – приложение слишком большое, в нем трудно ориентироваться и тяжело уследить за всеми зависимостями между компонентами, в следствие чего реализация новых возможностей и исправление ошибок занимают много времени, а поддержание «чистого кода» – труднее и труднее

- *Медленная разработка* – помимо описанных выше причин сюда также можно отнести замедление IDE, сборки приложения, тестирования приложения, долгого запуска
- *Длинный путь от внесения изменений до их развертывания* – доставка изменений в промышленную среду является долгим и тяжелым процессом, так как изменение даже небольшой функции в отдельном модуле требует переразвертывания всего приложения, а значит сборки и тестирования.
- *Трудности масштабирования* – требования к ресурсам разных модулей конфликтуют между собой, одному модулю требуются больше памяти, другому нужна не столько память, сколько процессорные мощности, так что решая проблемы модулей путем масштабирования мы постоянно должны идти на компромиссы при выборе серверной конфигурации.
- *Надежность и отказоустойчивость* – из-за своих размеров его сложно как следует тестировать, а из-за большой связанности ошибка в одном модуле может привести к отказу всей системы, даже если она могла бы частично работать без него.
- *Зависимость от стека технологий* – сложно переходить на новые фреймворки и языки программирования, а переписывание монолита с применением новых технологий – дорого и рискованно.

Теперь представим, как могло бы выглядеть такое приложение в микросервисной архитектуре (рис. 2). Обратите внимание, что сервисы слабо связаны между собой, занимаются отдельными функциями в приложении, а также имеют собственные, при необходимости, БД – это еще одна особенность, позволяющая добиться меньшей связанности.

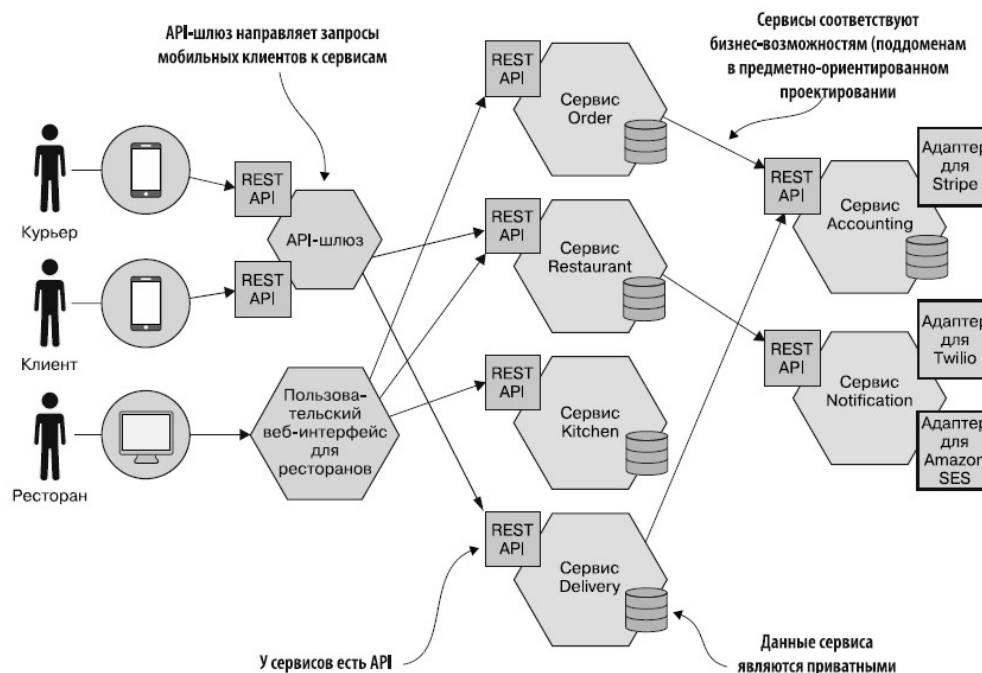


Рис. 1 - Некоторые из сервисов в микросервисной версии приложения FTGO.

API-шлюз направляет запросы мобильных клиентов к сервисам. Сервисы взаимодействуют между собой через API.

Преимущества микросервисной архитектуры:

- Возможность непрерывной доставки и развертывания крупных, сложных приложений
- Сервисы получаются небольшими и простыми в обслуживании
- Сервисы развертываются и масштабируются независимо друг от друга
- Обеспечивает автономность команд разработчиков (отдельные команды могут заниматься отдельным сервисом или группой сервисов, проводя в них работу и изменения независимо от работы других команд)

- Позволяет экспериментировать и внедрять новые технологии
- Лучше изолированы неполадки (выведение из строя сервиса не ломает всю систему)

Недостатки микросервисной архитектуры:

- *Сложность декомпозиции* – нет четкого алгоритма разбиения вашего приложения на отдельные сервисы и грамотная декомпозиция приложения требует понимания, навыков и опыта, потому что плохо организованная микросервисная архитектура может собрать в себе недостатки как одной так и другой архитектуры, что лишает её смысла.
- *Сложность распределенных систем* – требуется больше знаний, т.к. нужно учитывать как и новые, специфические технологии, которые используются в таких системах, так и их нюансы в виде сетевых задержек, баланса нагрузки, оркестровка, усложнение составления комбинированных запросов и транзакций и т.д.
- *Тестирование* – в монолитной архитектуре тестирование было затруднено из-за объема и недостаточного покрытия, в распределенной архитектуре тестировать компоненты в связке с друг с другом несколько сложнее.

Несмотря на свою распределенность и слабосвязанность – связи доменной логики никуда не деваются и сервисы так или иначе должны взаимодействовать между собой. Сервисы могут взаимодействовать как через REST API, gRPC API, так и через брокеров сообщений (рис. 3) – программных компонентов, который служат посредником между различными компонентами распределенной системы.

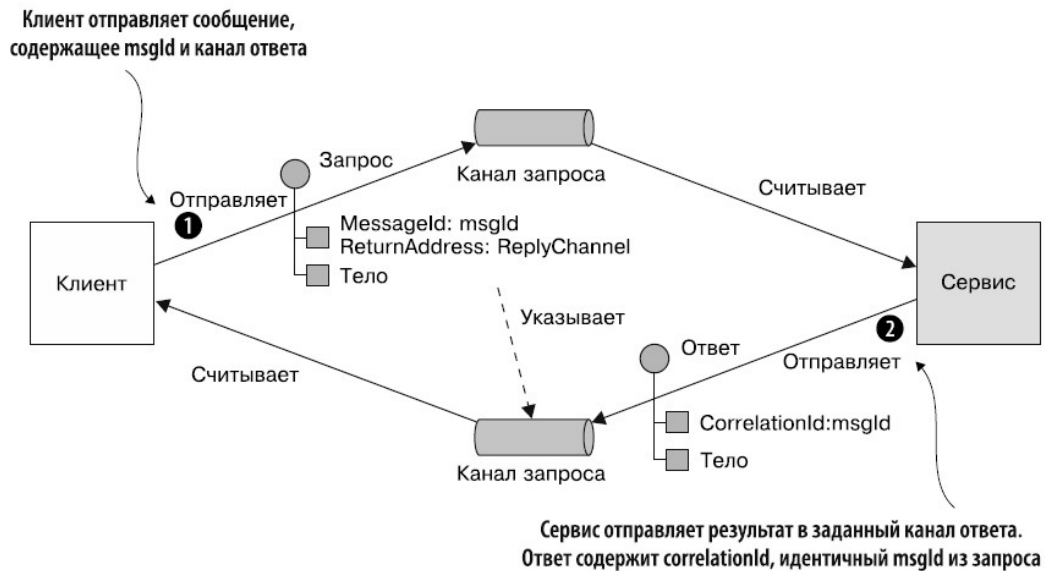


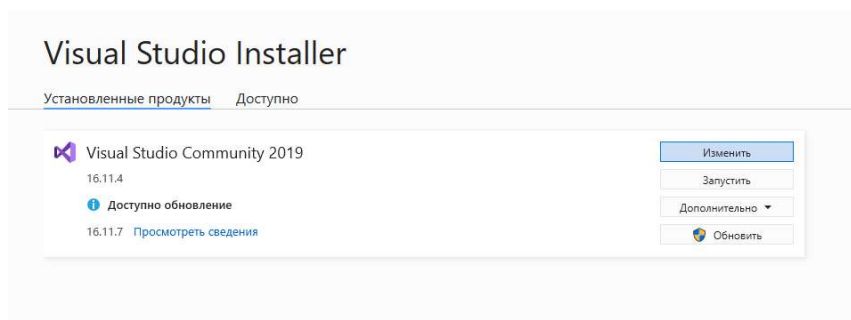
Рис. 3 Взаимодействие сервисов с помощью брокера сообщений.

Подробнее о микросервисной архитектуре можно почитать в материалах, указанных в конце методического обеспечения.

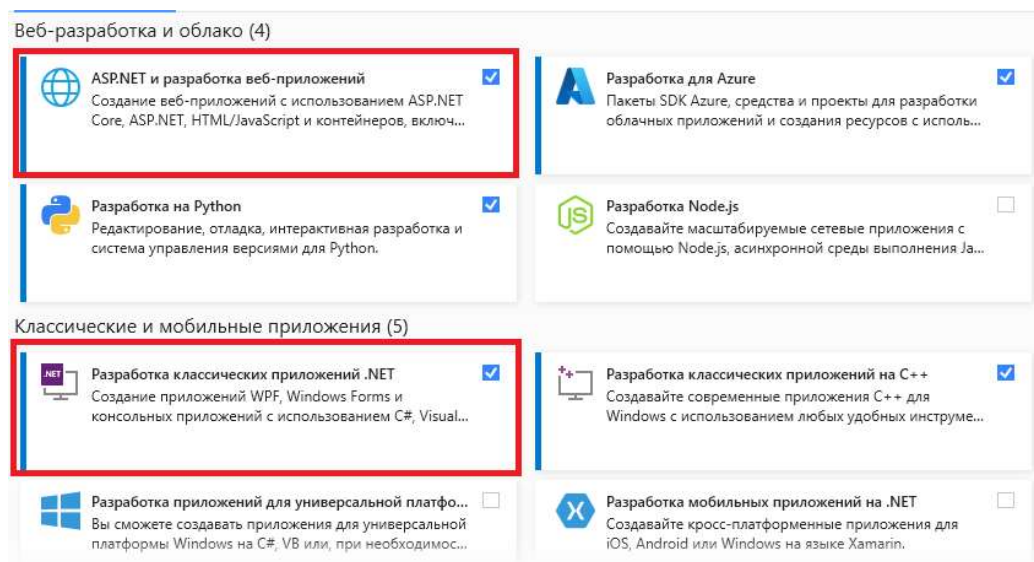
Начало работы

Перед началом работы подготовим среду разработки и необходимые, для выполнения работы, компоненты. В качестве IDE будем использовать Visual Studio 2019 Community. Зайдем в Visual Studio Installer и убедимся, что у нас уставлены соответствующие компоненты: нажмите кнопку «**Изменить**» → Проставьте флажки на компонентах *ASP.NET и разработка веб-приложений*, *Разработка классических приложений .NET*, *Хранение и обработка данных*, *Кроссплатформенная разработка для .NET* (см. рис. 1, а-в).

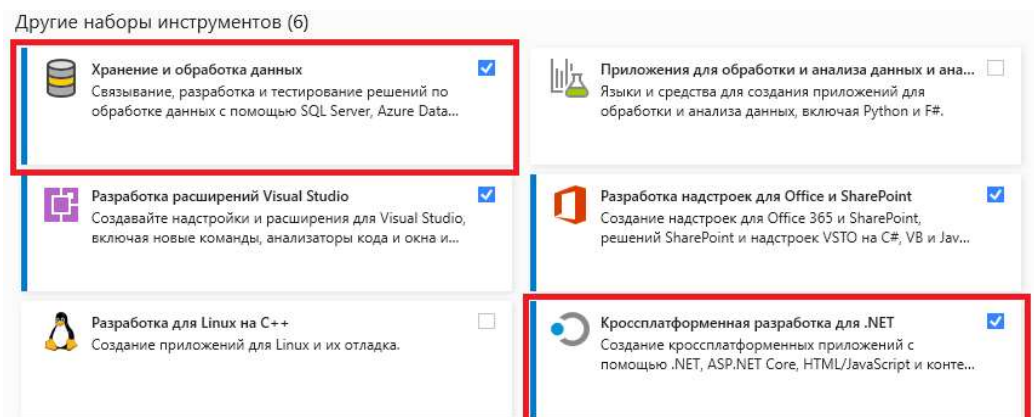
Если вы используете Visual Studio 2022, пакеты будут аналогичными, за исключением пакета «Кроссплатформенная разработка для .Net», которого нет в списке.



а



б



в

Рис. 1 Настройка компонентов:

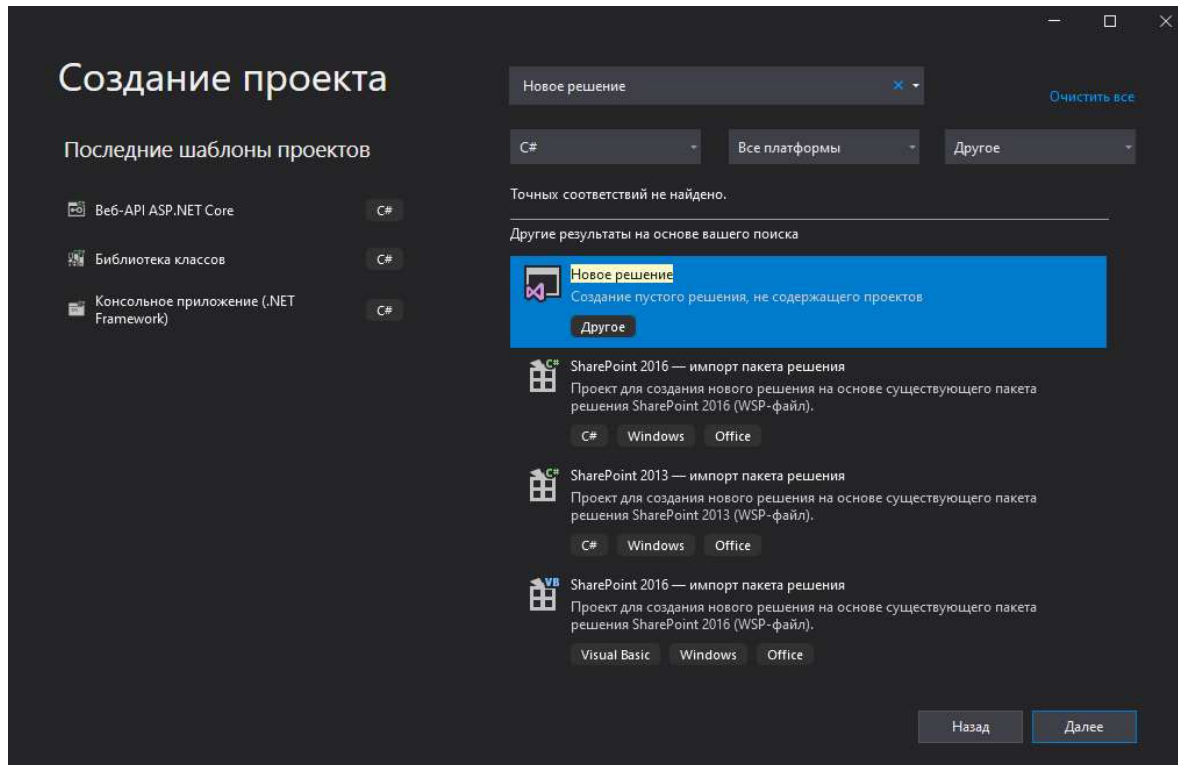
а – Кнопка «Изменить» в Visual Studio Installer.

б – Компоненты ASP.NET и разработка веб-приложений, Разработка классических приложений .NET.

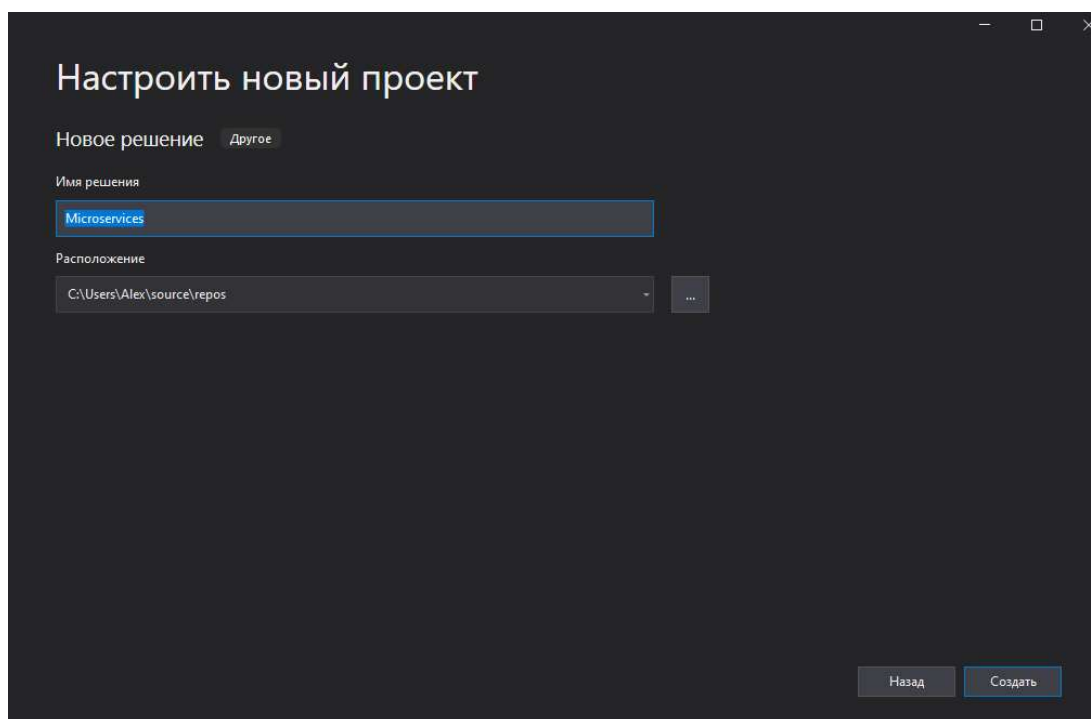
в – Компоненты Хранение и обработка данных, Кроссплатформенная разработка для .NET.

Создание проекта

Запустите Visual Studio. В открывшемся окне выберите «Создание проекта» → «Новое решение» → Задайте имя решению «**Microservices**» и укажите удобный для вас путь расположения проекта → Щелкните «Создать» (см. рис. 2 а-в).



а



б

Рис. 2 Создание решения:

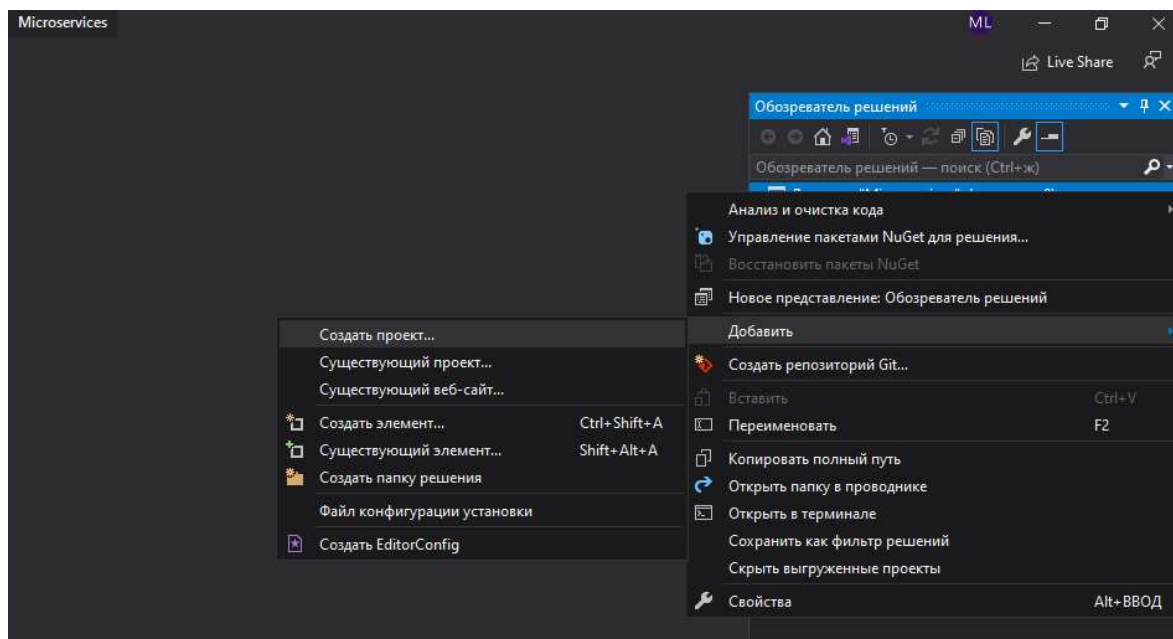
б – Окно с шаблонами проектов, выберите «Новое решение».

в – Окно конфигурации решения.

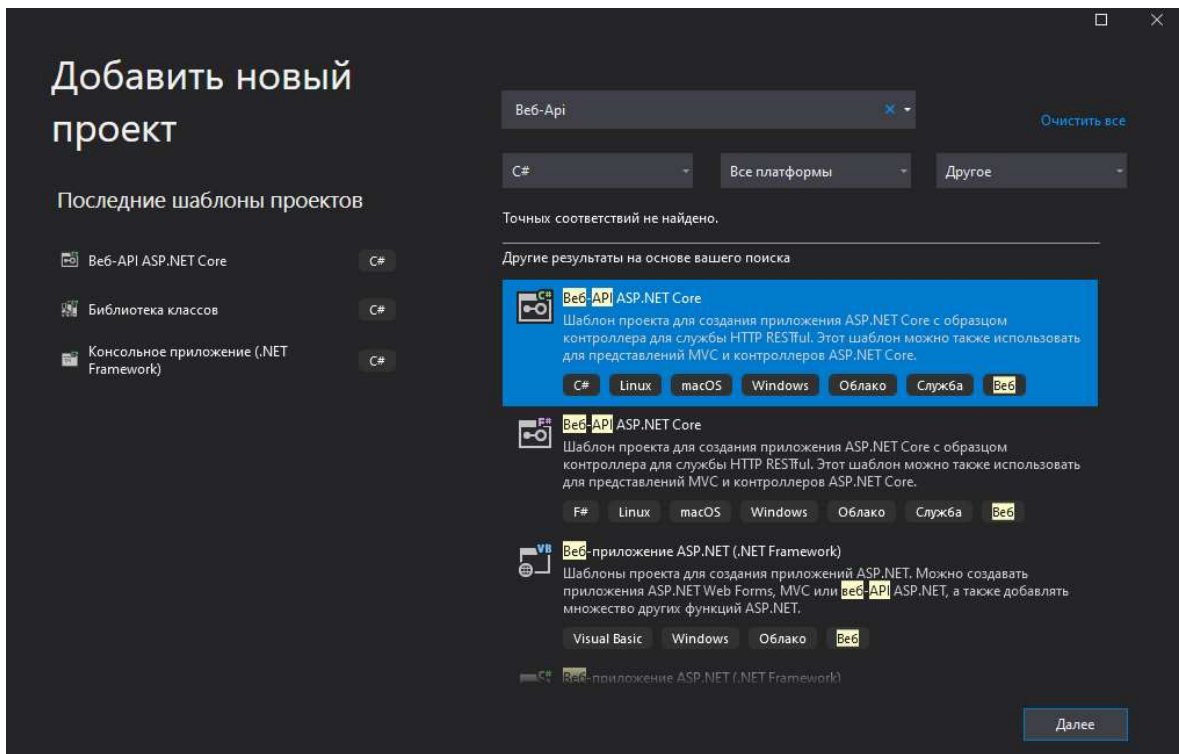
Откроется окно Visual Studio с вашим решением. Настройте вид программы как вам удобно, выбирая окна из меню **«Вид»**, закрепляя на рабочей области путем транспонирования. В окне **«Обозреватель решений»** щелкните правой кнопкой мыши по своему проекту → **«Добавить»** → **«Создать проект...»** → В открывшемся окне найдите **«Веб-API ASP.NET Core»** → **«Далее»** → В окне конфигурации введите имя проекта **«MicroserviceStudent»** → **«Далее»** → **«Создать»** (сняв галочку с пункта **«Включить поддержку OpenApi»** см. рис. 3 а-в).

Важно. В более ранних версиях VS не было поля выбора **«Не использовать операторы верхнего уровня»**. В данном примере я выберу этот

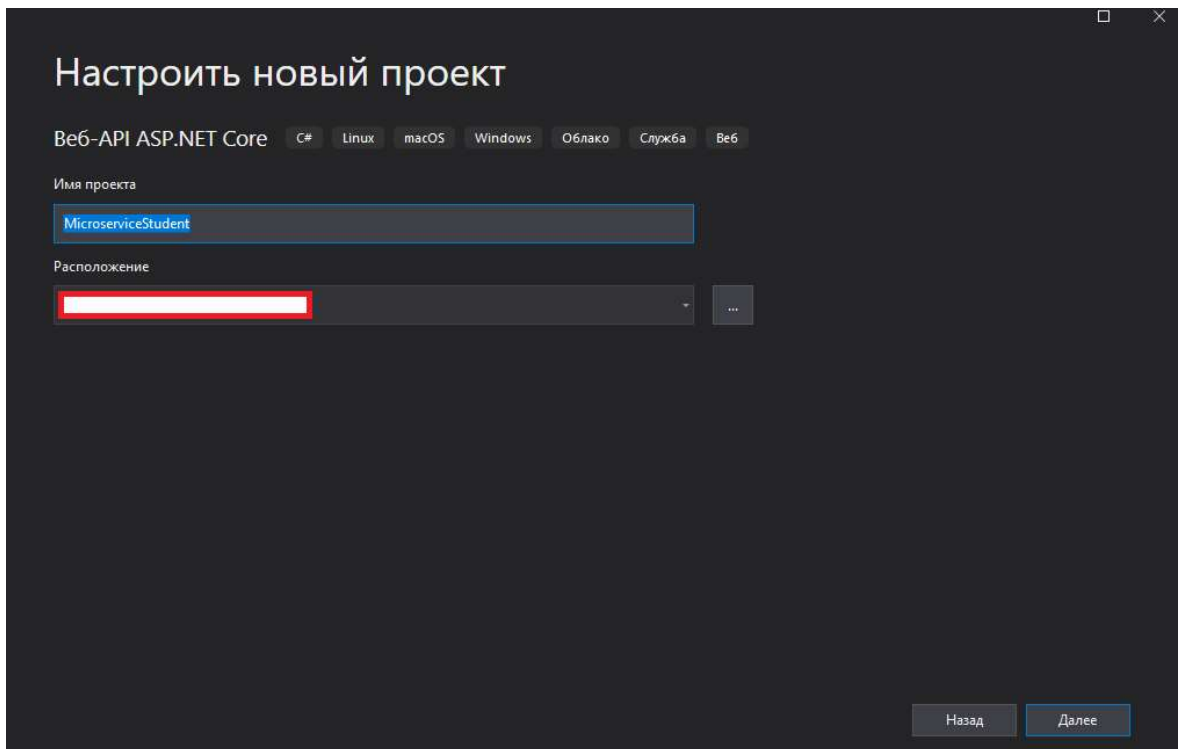
чекбокс, но если вы используете VS2022, то оставьте его выключенным. При разработке второго сервиса будет показано, как создать проект и настроить его без этого чекбокса.



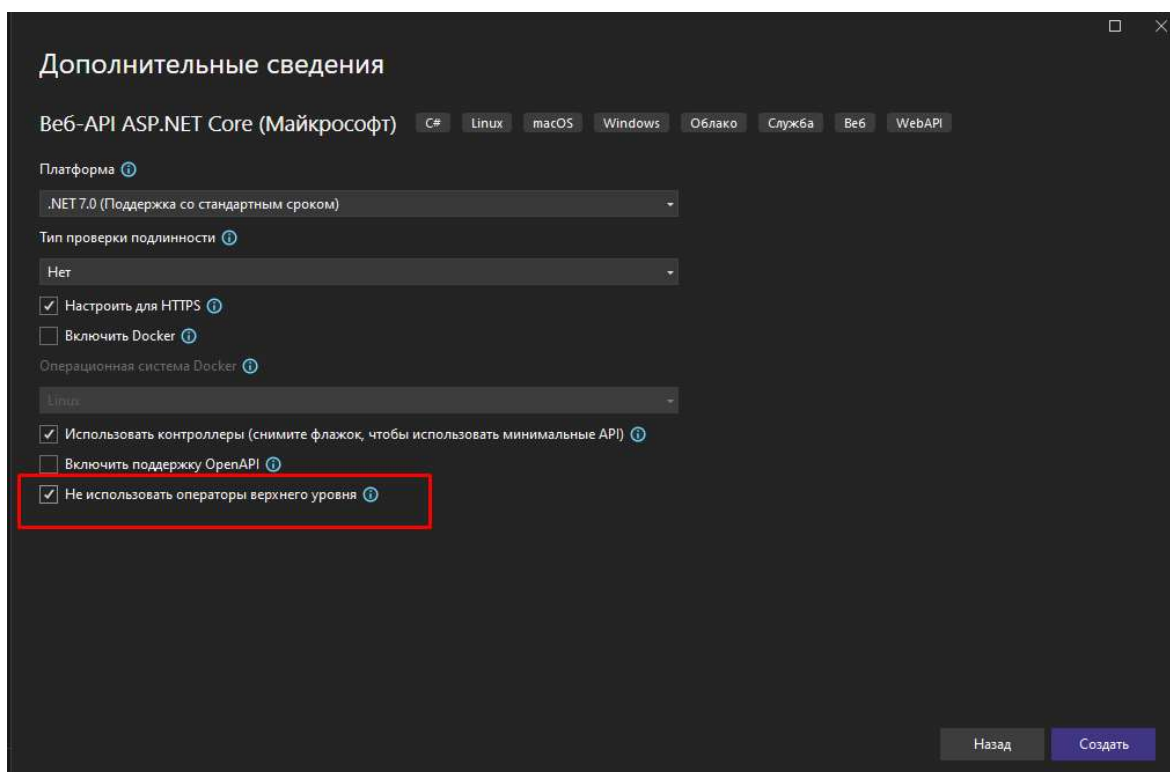
а



6



В



Г

Рис. 3 Создание проекта:

а – Добавление проекта в решение.

б – Выбор типа шаблона проекта.

в – Окно конфигурации проекта.

г – Окно конфигурации проекта в VS2022.

Visual Studio добавит шаблон проекта, в нём уже присутствуют необходимые компоненты для старта простого сервиса. Запустите проект с помощью клавиши «F5» или выберите соответствующий пункт в интерфейсе VS. Visual Studio отображает следующее диалоговое окно (см. рис. 4), если проект еще не настроен для использования SSL:

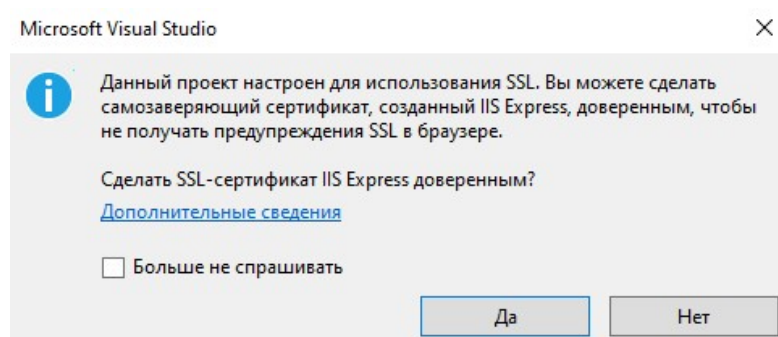


Рис. 4

Выберите «Да», чтобы сделать SSL-сертификат IIS Express доверенным. Отобразится следующее диалоговое окно:

Выберите «Да», если согласны доверять сертификату разработки (см. рис. 5). Если у вас не получилось настроить сертификат, при запуске проекта закрывайте окно, представленное на рис. 4 – это не отобразится на работе приложения. Первый старт может быть достаточно продолжительным, тем не менее, дождавшись хостинга и выполнения логики, вам отобразится следующая страница (см. рис. 6).



Рис. 5

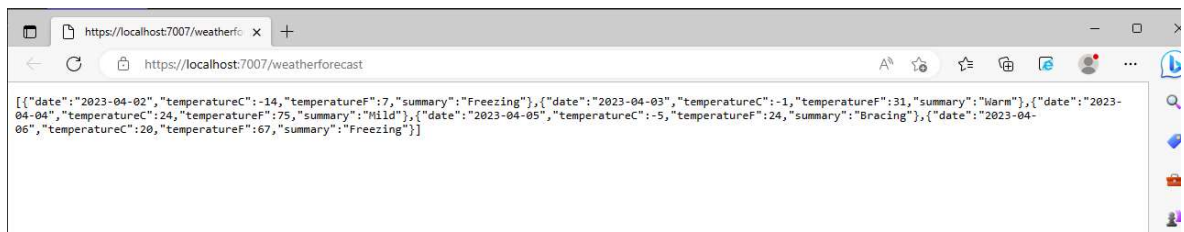


Рис. 6

Создание модели микросервиса «Студент»

Займемся созданием модели. *Модель* – это набор классов, представляющих данные, которыми управляет приложение. Модель этого приложения будет содержать единственный класс – *Student*. Щелкните правой кнопкой мыши по проекту → «Добавить» → «Создать папку» → Назовите папку *Models* (см. рис. 7).

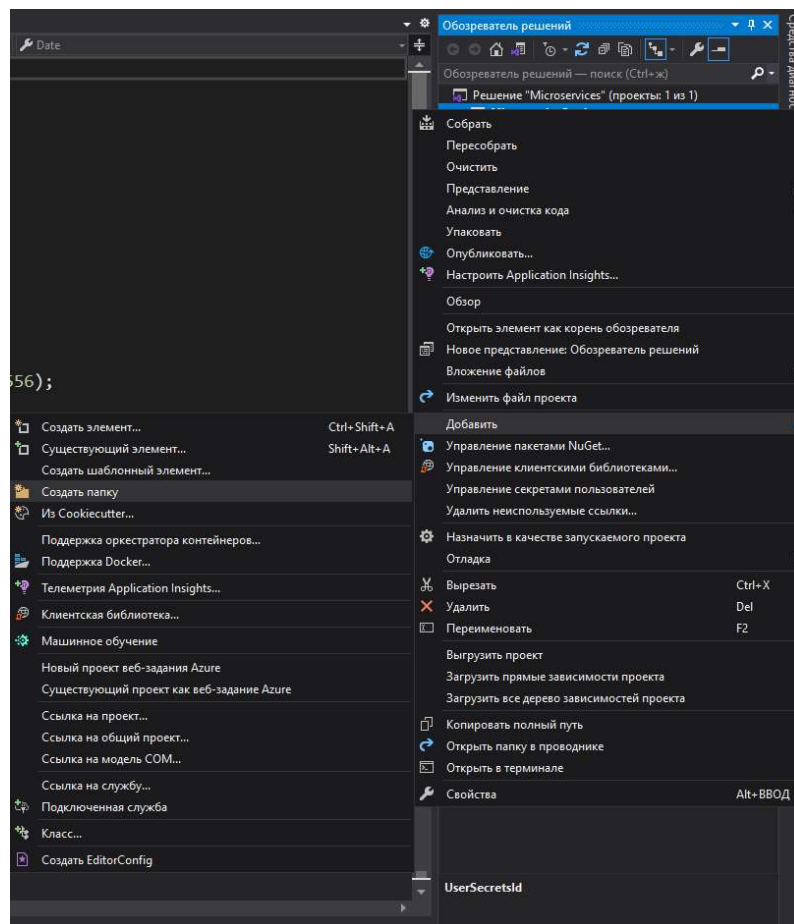
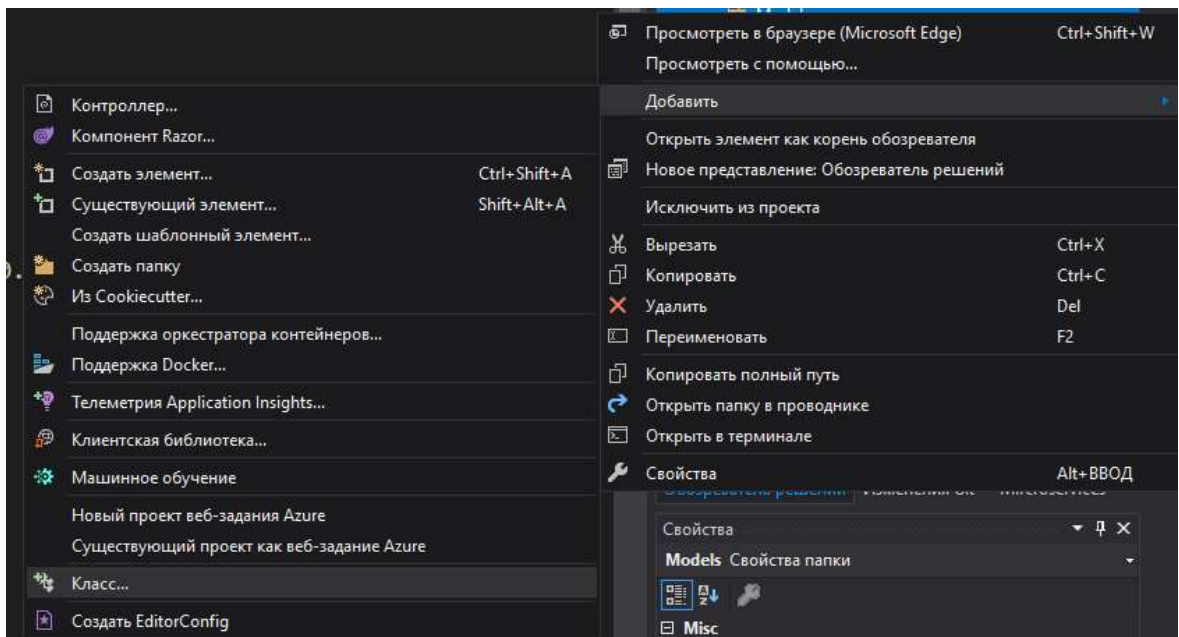
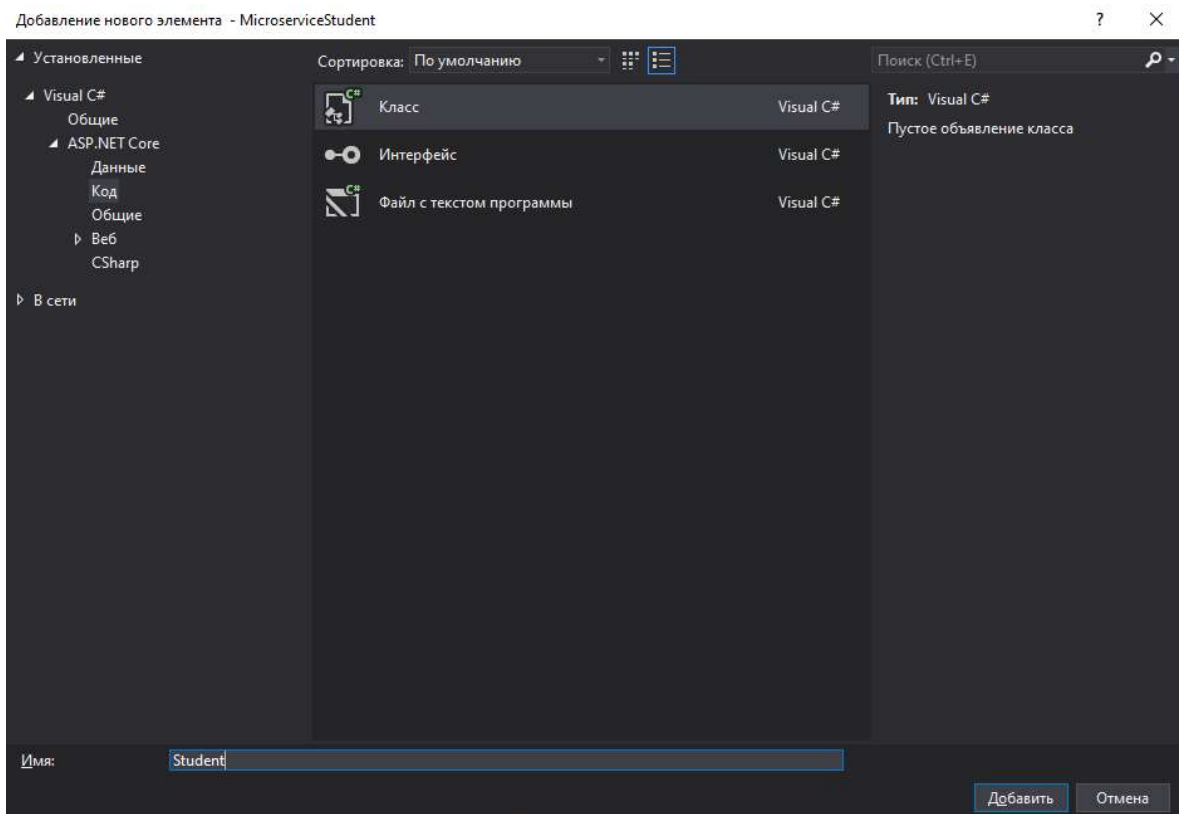


Рис. 7

Нажмите ПКМ по папке «Models» → «Добавить» → «Класс...» → В открывшемся окне выберите пункт «Класс» и введите имя «Student» (см. рис 8 а-б).



а – Создание папки.



б – Создание класса.

Рис. 8 Добавление класса модели

Перейдем к описанию класса. Поскольку *MicroserviceStudent* должен предоставлять доступ к базе данных студентов – добавим помимо обычных полей (свойств), определенных требованиями задачи, свойство «**Id**». Включим в класс Student такие свойства как:

- «**Id**» – оно будет отвечать за уникальный ключ в БД.
- «**Name**» – имя студента.
- «**GroupName**» – название группы студента.
- «**Rating**» – его учебный рейтинг.

Ниже приведен листинг кода класса. Если вы используете старые версии .Net, вам вероятно придется поставить дополнительные фигурные скобки для namespace). В этом указании будет использоваться file-scoped namespace, но вы можете использовать старый тип:

File-scoped namespace

```
namespace MicroserviceStudent.Models;

public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string GroupName { get; set; }
    public int Rating { get; set; }
}
```

Non file-scoped namespace

```
namespace MicroserviceStudent.Models
{
    public class Student
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public string GroupName { get; set; }
        public int Rating { get; set; }
    }
}
```

Создание базы данных и её контекста

Установка пакетов NuGet. Нажмите ПКМ по решению в окне «Обозреватель решений» → Выберите пункт «**Управление пакетами NuGet для решения**» (см. рис. 9) → В открывшемся окне перейдите во вкладку «Обзор» → Найдите с помощью поиска следующие пакеты: *Microsoft.EntityFrameworkCore*, *Microsoft.EntityFrameworkCore.SqlServer*, *Microsoft.EntityFrameworkCore.Tools*, *Microsoft.VisualStudio.Web.CodeGeneration.Design* версий 7.0.5, 7.0.5, 7.0.5 и 7.0.6 соответственно, если используете .NET 7.

Примечание. Вы можете использовать вместо *Microsoft.EntityFrameworkCore.SqlServer* другого провайдера базы данных на свой выбор, например *Npgsql.EntityFrameworkCore.PostgreSQL*, и создать контекст подходом Database first на случай, если Code first не сработает (о создании контекста речь будет идти позже). Этот подход будет показан при создании второго микросервиса.

Важно. Если вы используете более поздние версии .NET, убедитесь, что мажорные версии (первое число в версии, пакета) пакетов совпадают с версией вашего .NET. Версию .NET, вы можете как выбрать самостоятельно, во время создания проекта, так и изменить или посмотреть, нажав ПКМ по проекту → Свойства → Приложение (рис. 10а). Так, например, если вы используете .NET 5, то пакеты всех вышеперечисленных выше зависимостей должны быть 5.x.x.

Для установки нужно выберите нужный пакет → Поставьте чекбокс у проекта «**MicroserviceStudent**» в правой части окна → Нажмите «**Установить**» (см. пример рис. 10) → «**Ок**» → «**Я принимаю**». Прделайте аналогичные операции с оставшимися пакетами.

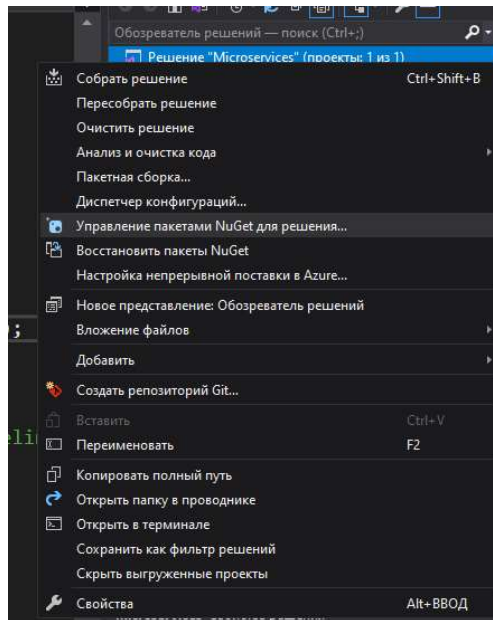


Рис. 9

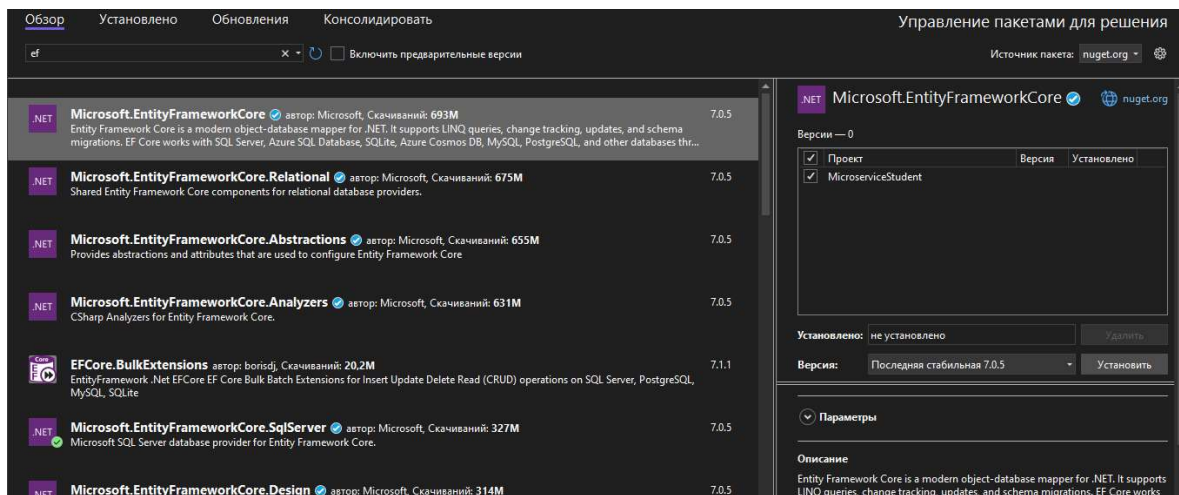


Рис. 10 Установка пакета NuGet.

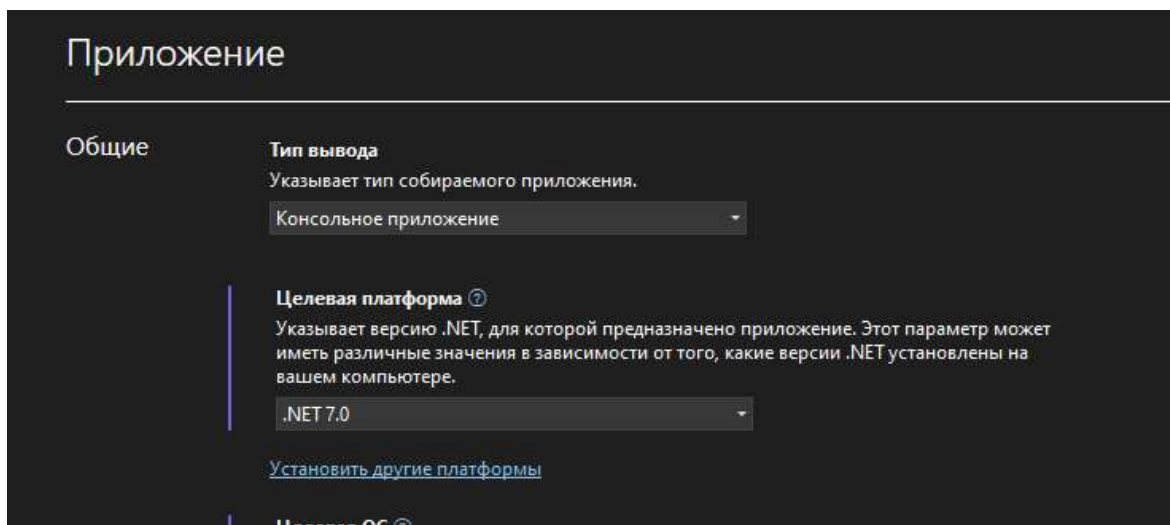


Рис. 10а Свойства проекта. Текущая версия .NET.

Создание контекста базы данных. Для работы с базами данных используется Entity Framework. *Entity Framework* представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. Если традиционные средства ADO.NET позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, то на концептуальном уровне, который нам предлагает Entity Framework, мы уже *работаем с объектами*. Щелкните ПКМ по проекту в окне «Обозреватель решений» и добавьте класс с именем *StudentContext* (см. аналогичное создание класса Student рис. 8 а-б). Добавьте в этот файл следующий код.

```
using MicroserviceStudent.Models;
using Microsoft.EntityFrameworkCore;

namespace MicroserviceStudent;

public class StudentContext : DbContext
```

```

{
    public StudentContext(DbContextOptions<StudentContext> options)
        : base(options)
    {
    }

    public DbSet<Student> Students { get; set; }
}

```

Класс *StudentContext* наследует *DbContext*, который определяет контекст данных, используемый для взаимодействия с базой данных. Конструктор принимает в качестве параметра объект типа *DbContextOptions<T>*, где *T* в нашем проекте является *StudentContext*, это нужно для привязки провайдера БД к нашему контексту. Свойство *Students*, возвращает (и задает) объект типа *DbSet<T>*, где *T* является конкретной сущностью БД, в нашем случае это будет тип *Student*. Убедитесь, что подключена директива *Microsoft.EntityFrameworkCore*, в пространстве которой находятся вышеописанные классы.

Изменение класса Startup, регистрация контекста БД.

Если вы использовали VS2022, сделайте шаги, аналогичные описанным при создании сервиса курсов.

(Если не использовали VS2022) Откройте класс *Startup* в проекте *MicroserviceStudent*, добавьте две директивы с помощью ключевого слова *using*: *Microsoft.EntityFrameworkCore*.

Добавьте в метод *ConfigureServices*:

```

services.AddDbContext<StudentContext>(opt => opt.UseSqlServer($"Data
Source=(localdb)\\MSSQLLocalDB;Initial Catalog=StudentDB;Integrated
Security=True"));

```

В этом методе происходит внедрение зависимости. Регистрируется подкласс *DbContext* с именем *StudentContext*. Контекст настраивается для

использования поставщика базы данных SQL Server и считывания строки подключения.

Создание базы данных. Способ первый, Code first.

Откройте вкладку «Вид» → В меню выберите «Другие окна» → «Консоль диспетчера пакетов» (см. рис. 11). Закрепите окно в удобной вам области.

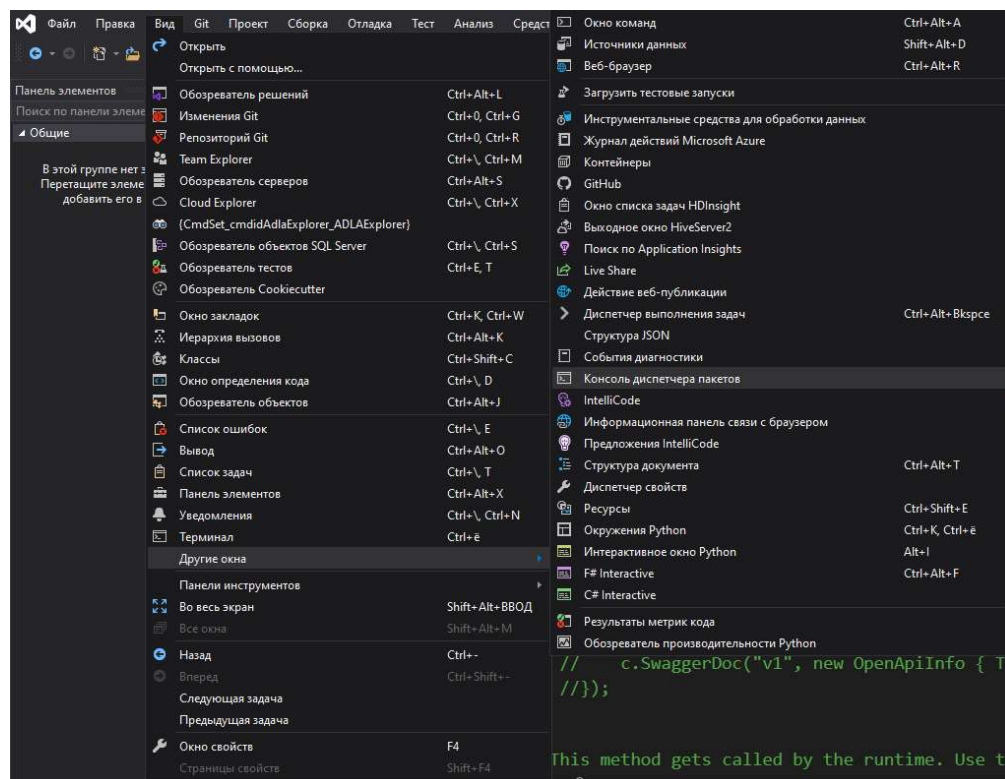


Рис. 11 Вид → Консоль диспетчера пакетов

В консоли диспетчера пакетов напишите команду **Add-Migration InitMigration** (используйте Tab для появления подсказок) → нажмите клавишу **Enter** → Дождитесь окончания выполнения команды (надписи «To undo this action, use Remove-Migration») → Введите новую команду **Update-Database** → Дождитесь выполнения этой команды (см. рис. 12 а-б).

а

б

Рис. 12 Создание БД:

а – Выполнение команды Add-Migration InitMigration.

б – Выполнение команды Update-Database.

Найдите свою БД. Откройте окно «Обозреватель объектов SQL Server», для этого нажмите «Вид» → В меню выберите «Обозреватель объектов SQL Server» (см. пример рис. 11). Разверните корень директории в этом окне и найдите базу данных StudentDB, в папке «Таблицы» должна будет таблица студентов (см. рис. 13).

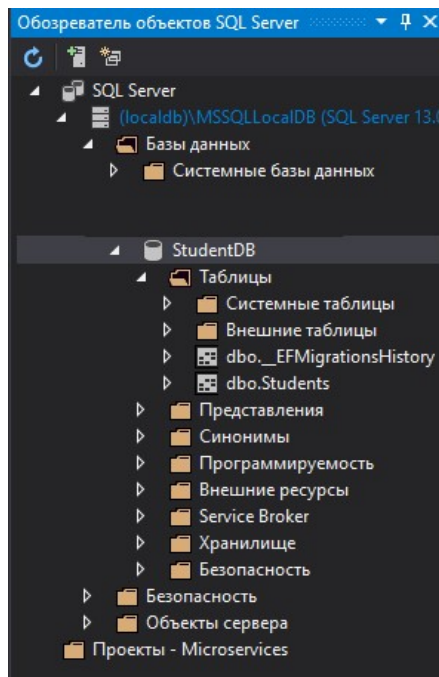


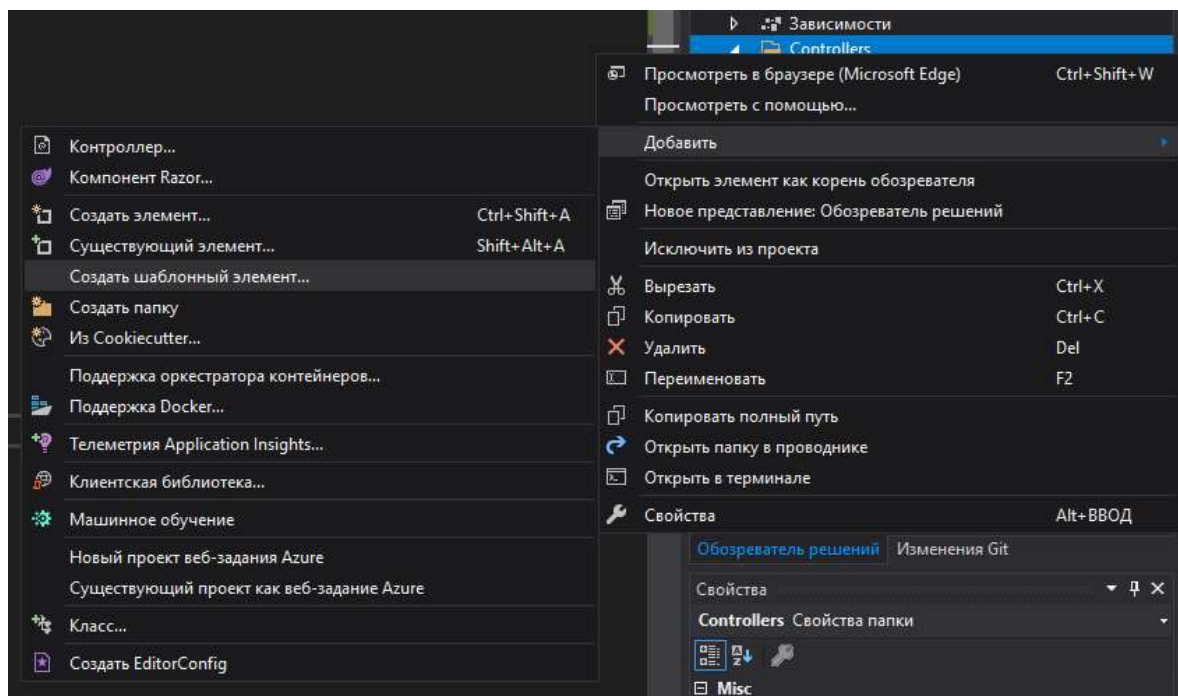
Рис. 13 База данных StudentDB

Разберемся в том, что произошло. В классе `Startup` мы добавили свой контекст базы данных, в качестве подключения выбрали `SQL Server` и в строке подключения, помимо прочих параметров, указали имя БД `StudentDB`. Команда ***Add-Migration InitMigration*** добавляет миграцию, на основе модели, что используется в контексте базы данных. Миграцию можно назвать некоторым шаблоном, по которому будет программно создаваться и настраиваться база данных. ***InitMigration*** – название миграции, приставка `Init` от слова `initialize`, для понимания, что данная «версия» миграции являлась первой. Команда ***Update-Database*** обновляет или создаёт базу данных, на основе миграций. В случае изменения модели, например если потребовалось добавить данные возраста к типу `Student` – добавляем нужное свойство, затем в консоли диспетчера пакетов повторяем эти две команды, только на этот раз укажем другое название миграции при её добавлении, например, ***AddAgeToStudentsTable***. База данных изменится согласно описанной модели.

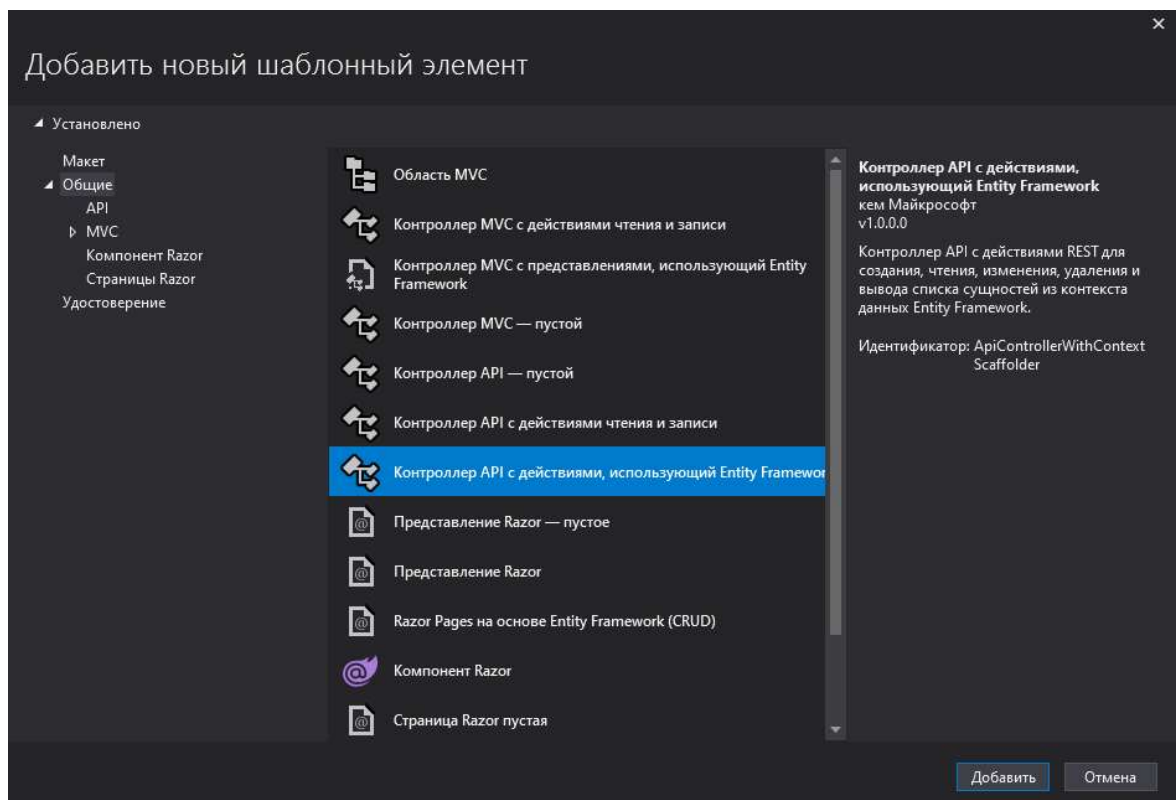
Создание контроллера микросервиса студентов.

Щелкните ПКМ по папке «**Controllers**» в проекте «**MicroserviceStudent**» → «**Добавить**» → «**Создать шаблонный элемент**» (см. рис. 14 а) → В открывшемся окне выберите «**Контроллер API с действиями, использующий Entity Framework**» и нажмите «**Добавить**» (см. рис. 14 б) → В поле «Класс модели» выберите Student, а в поле «Класс контекста данных» – StudentContext, в качестве имени оставьте StudentsController, нажмите кнопку «**Добавить**» (см. рис 14 в).

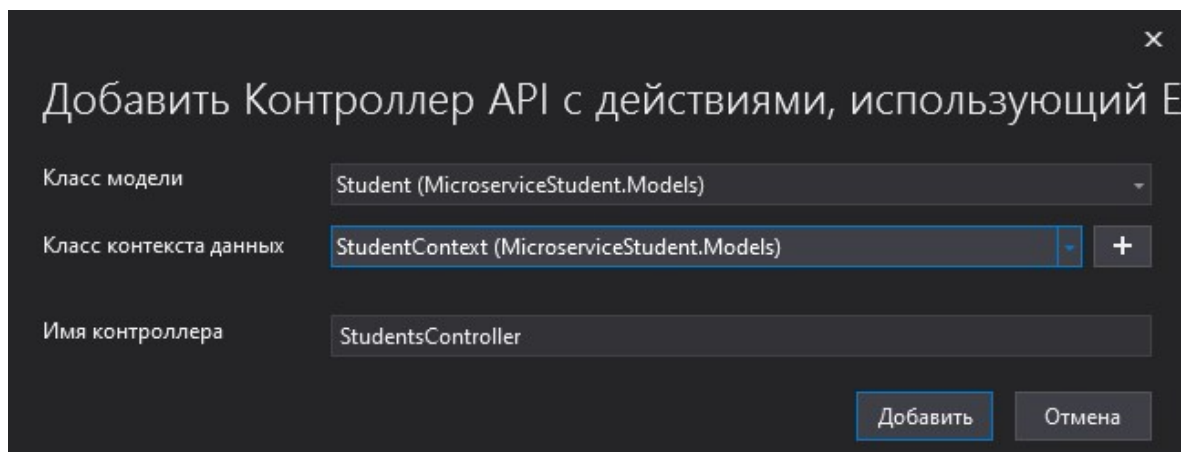
Важно. Если у вас по каким-то причинам не генерируется шаблонный элемент, то вы можете создать класс контроллера вручную, в конце указания также есть ссылка на исходные материалы.



а



6



В

Рис. 14 Генерация контроллера:

После этого среда сгенерирует класс StudentsController в папке Controllers и наполнит его методами запросов. Разберемся в том, что за код был создан. Класс помечен двумя атрибутами – *ApiController* и *Route*.

Первый применяется к классу контроллера для включения следующих специализированных схем поведения API:

- Обязательная маршрутизация атрибутов
- Автоматические отклики HTTP 400
- Вывод параметров источника привязки
- Вывод многокомпонентных запросов и запросов данных форм
- Сведения о проблемах для кодов состояния ошибки

Route отвечает за привязку маршрута. Класс **ControllerBase**, от которого мы наследуемся предоставляет множество свойств и методов, которые удобны для обработки HTTP-запросов. Например, **ControllerBase.CreatedAtAction** возвращает код состояния 201. У каждого метода созданного контроллера также есть атрибуты такие как **HttpGet**, **HttpPost** и т.п. Они определяют, в рамках какого http запроса будет работать метод, потому как общение происходит путем как раз таких запросов. Внутри этих атрибутов может содержаться «маршрут» к этим методам и те или иные параметры, которые можно будет достать из адресной строки. Параметры указываются в фигурный скобках и их название должно совпадать с параметрами метода, отвечающий за реализацию этого запроса. Также в классе присутствует поле **_context**, содержащий в себе контекст БД, который мы уже создавали и настраивали. Если посмотреть код некоторых методов можно наглядно увидеть принцип работы с БД в **EntityFramework**, коммуникация происходит через объекты и модели, соответствующие методы, а не прямые SQL запросы.

Изменение настроек запуска. Откройте файл «**launchSettings.json**», находящийся в папке «**Properties**» проекта «**MicroserviceStudent**» и обновите «**launchUrl**» с «**weatherforecast**» на «**api/students**». Запустите приложение, так как первая загрузка, как говорилось ранее, бывает долгой – можно получить

ошибку о превышении времени обращения, попробуйте обновить страницу, конечный результат представлен на рис. 15.

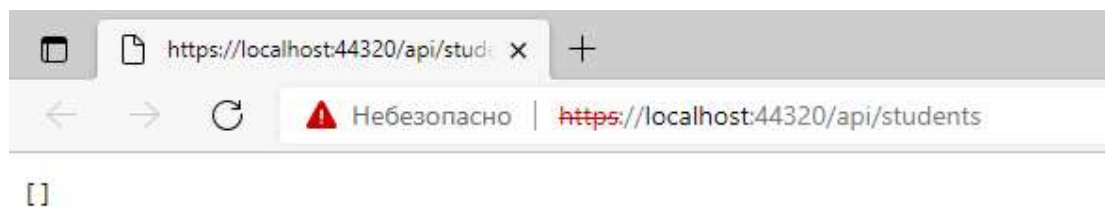


Рис. 15

Тестирование запросов с помощью Postman

Запрос GET. Установите и запустите приложение Postman. В разделе «Collections» создайте новую коллекцию, щелкнув на «+» (см. рис. 16) и задайте ей имя «Microservice Student».

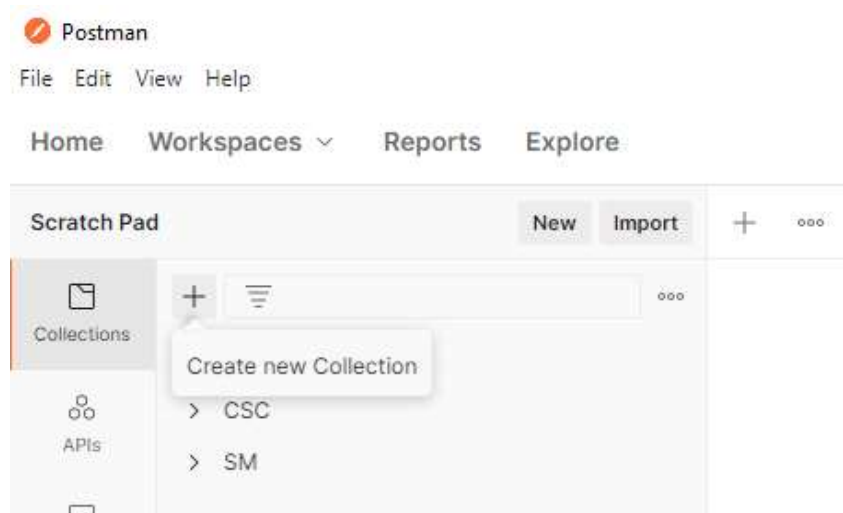
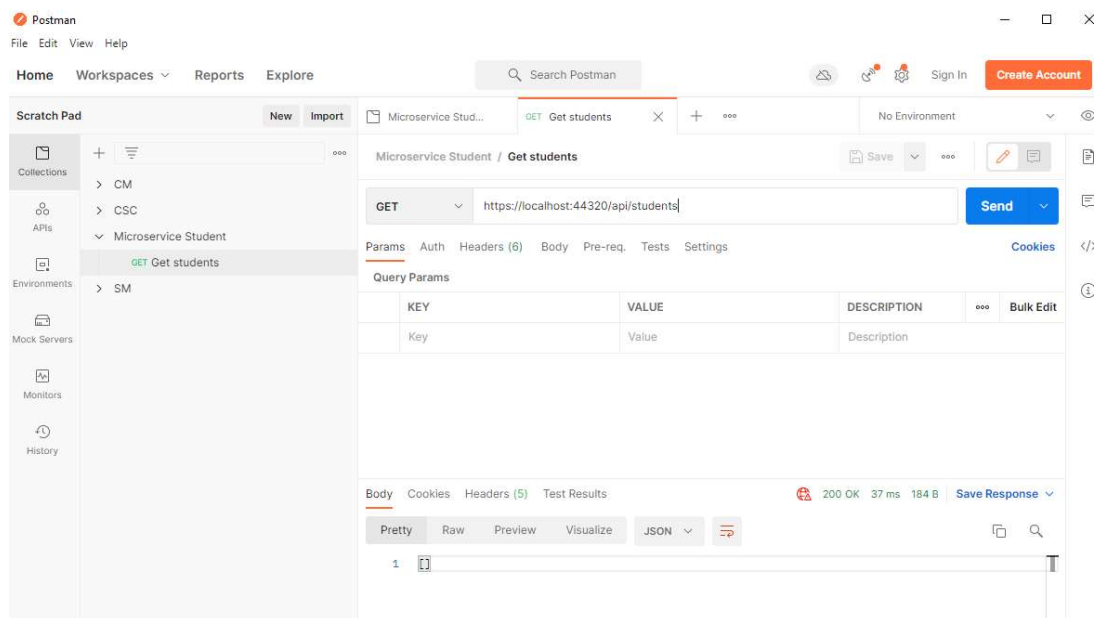


Рис. 16

Нажмите ПКМ по созданной коллекции и выберите в меню пункт «Add Request», в созданном запросе установите тип запроса «GET», задайте ему имя «Get students». В адресную строку вставьте адрес, который открывался при запуске приложения (см. рис 15) и нажмите сочетание клавиш «Ctrl+S» или кнопку «Save». Запустите проект в Visual Studio, в приложении Postman откройте свой созданный запрос и нажмите на кнопку «Send». Если приложение предложит отключить ssl – сделайте это, нажав на текст «Disable SSL verification» Ответ микросервиса будет получен ниже (см. рис 17).



Could not get response

SSL Error: Unable to verify the first certificate | [Disable SSL Verification](#)

[Learn more about troubleshooting API requests](#)

Рис. 17 – Get запрос к приложению в Postman.

Заметим, что результат, отображенный на странице браузера, совпадает с ответом в Postman.

Запрос POST. Создайте по аналогии еще один запрос в коллекции Postman, укажите его тип как «POST», а в качестве имени напишите «Add student», адрес укажите такой же, как и в Get запросе. Перейдите во вкладку «Body» → В меню выберите «raw» → В появившемся справа меню вместо «Text» поставьте «JSON» → В поле для текста вставьте следующий код:

```
{
  "name": "Alex",
  "groupName": "PRI-120",
  "rating": 53
}
```

Нажмите кнопку «Send». После чего придёт ответ об успешном добавлении в БД заданного студента (см. рис. 18).

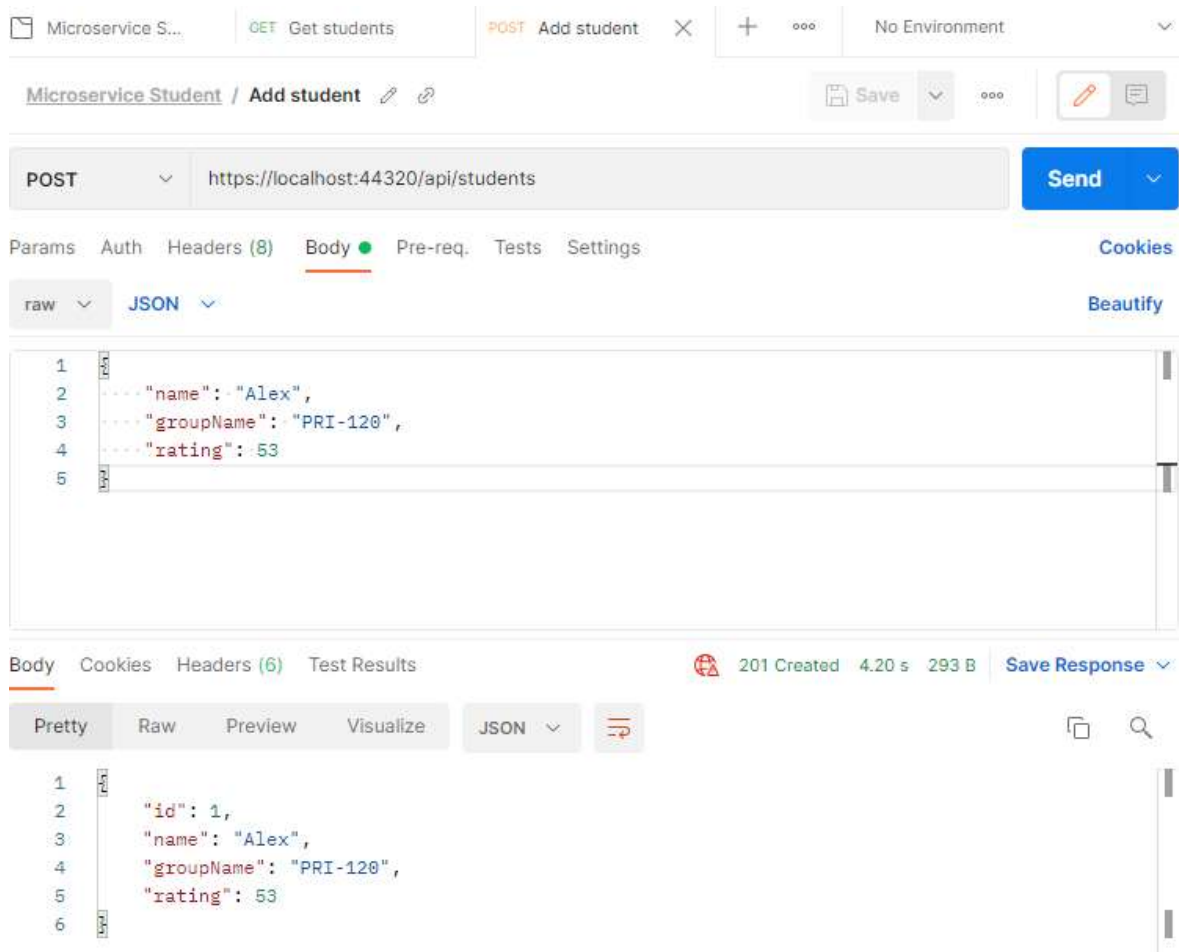


Рис. 18 – Результат POST запроса.

Итак, после отправки запроса в контроллере вызван был вызван асинхронный метод «**PostStudent**». В качестве параметра он принимает объект «**Student**» – его мы описали в «**Body**» запроса в формате «**JSON**», указав в фигурных скобках название свойств в *lowerCamelCase* стиле и через « : » передав значение. Внутри этого метода мы обращаемся к контексту базы данных, получаем список студентов и добавляем к нему новый экземпляр, ожидаем сохранения изменений в базе данных и отправляем ответ об успешном выполнении запроса. Зайдите в Get запрос Postman, который мы настраивали выше и опять отправьте его. Теперь в вашем списке должен

отобразится новый студент (см. рис. 19). Самостоятельно добавьте ещё двух студентов.

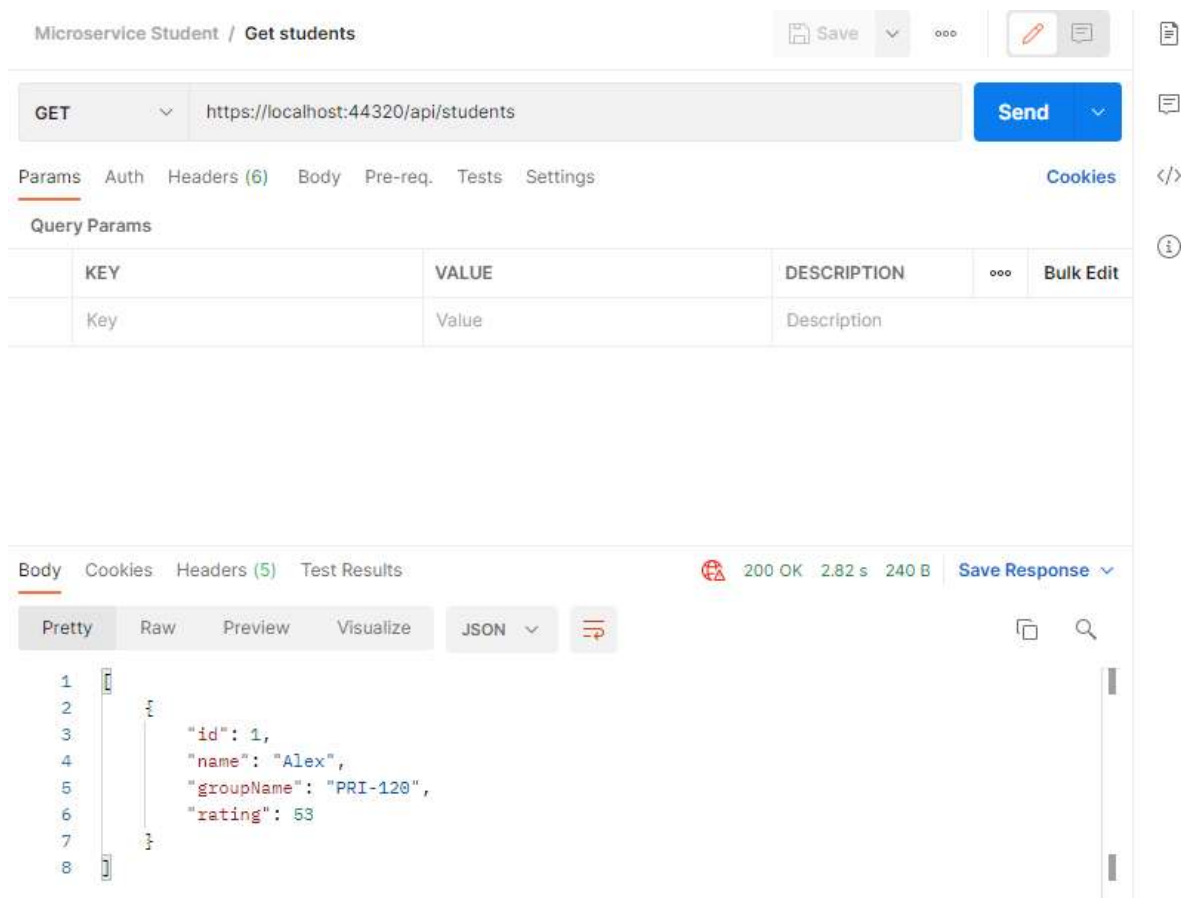


Рис. 19 – Результат работы GET запроса.

Создание PUT запроса. Создайте запрос по аналогии с предыдущими примерами, в качестве типа укажите «PUT», задайте имя «Change student by Id» а к адресу сервиса допишите «/2». Откройте вкладку «Body» и настройте также как в «POST» запросе. Введите следующий код и нажмите «Send» (результат представлен на рис. 20):

```
{  "id" : 2,  "name": "Kira",  "groupName": "PRI-120",  "rating": 83}
```

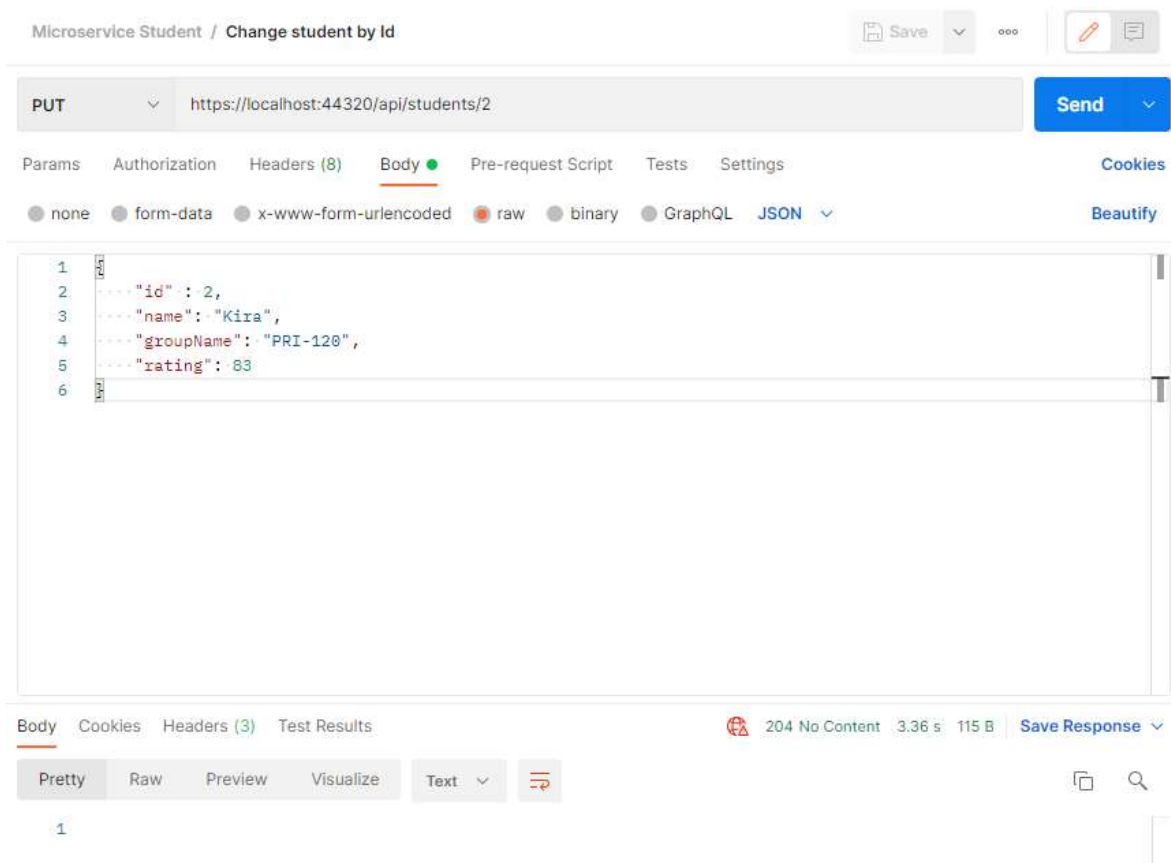


Рис. 20 – PUT запрос.

Теперь с помощью GET запроса получите список студентов, заметьте, что студент с $Id = 2$ изменился согласно описанию в Body. При PUT запросе вызывается метод контроллера PutStudent. Он принимает два аргумента: id и объект типа Student. Первый параметр он достаёт из адреса, именно поэтому в описании атрибута указан параметр в фигурных скобках [HttpPut("{id}")] , а объект получает из описания в Body аналогично POST запросу. Обновлять студента в базе данных можно и следующим образом:

```
_context.Students.Update(student);  
await _context.SaveChangesAsync();
```

Самостоятельно измените рейтинг студента (с id = 2) с 83 до 81.

Создание DELETE запроса. Создайте запрос по аналогии с PUT запросом, укажите его тип как «DELETE», а в названии напишите «Delete

student by id». В адресе пути измените id с 2 на 3 и нажмите «Send» (см. результат на рис. 21).

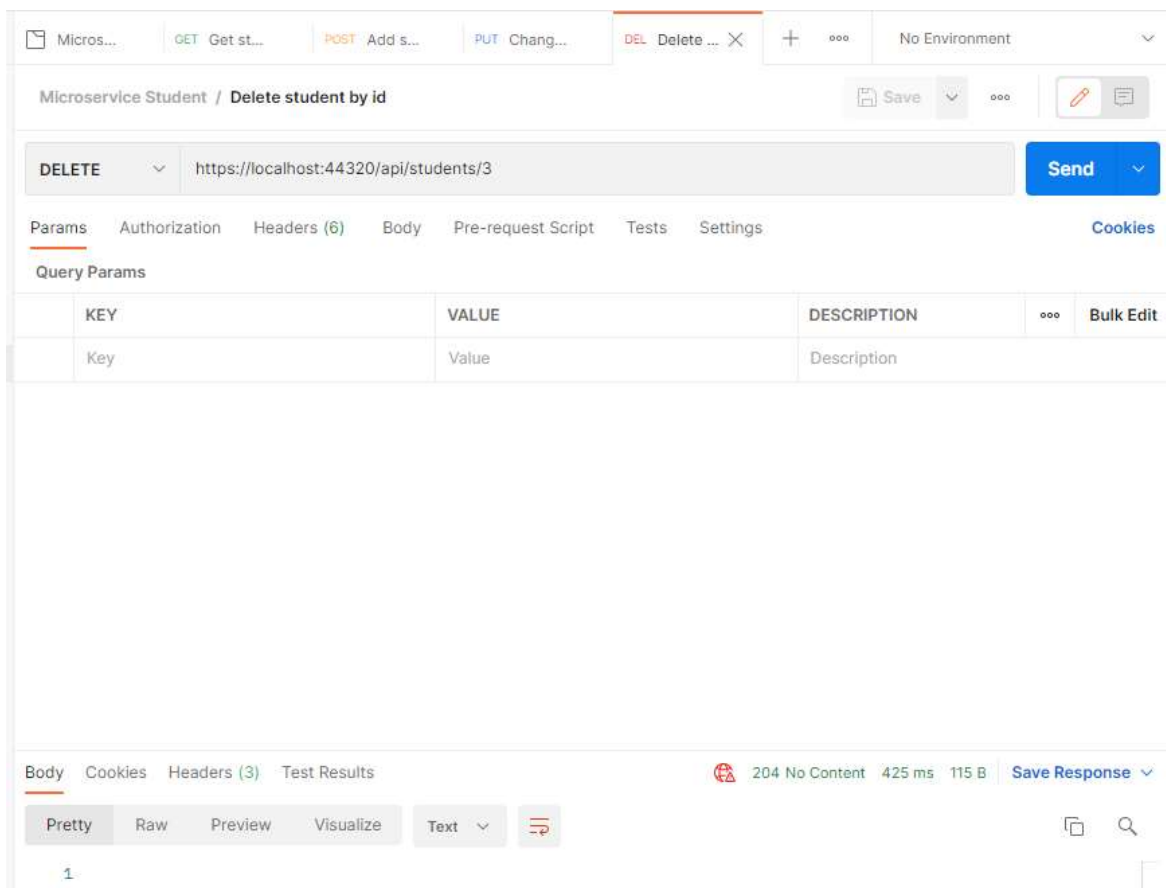
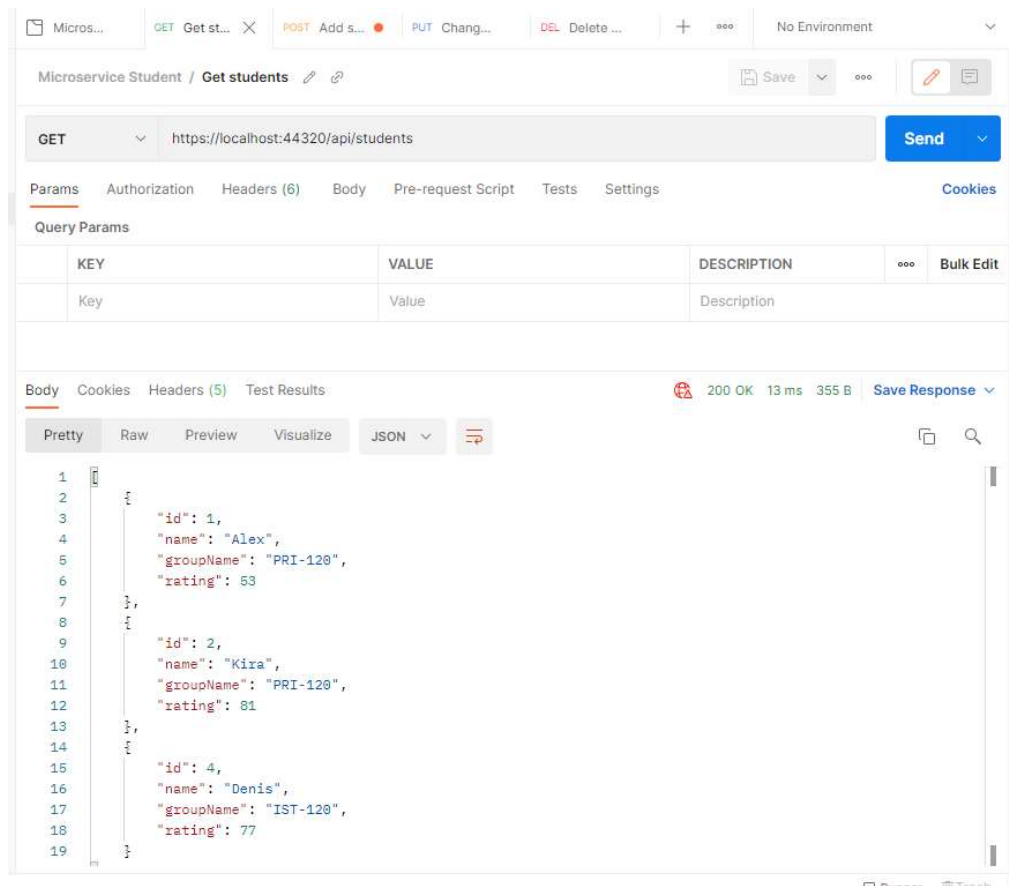
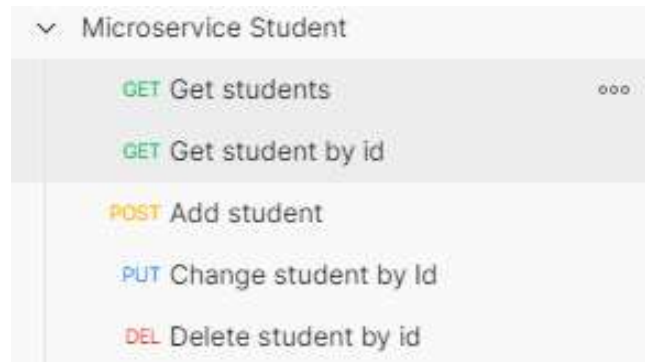


Рис. 21 – Результат работы DELETE запроса.

При вызове метода «**DeleteStudent**», содержащего в качестве параметра id студента, получаемого из адреса, происходит обращение к контексту БД, который асинхронно находит студента с заданным идентификатором, затем мы просим удалить запись из базы по соответствующему ей объекту типа Student и сохранить изменения. Добавьте еще одного студента и получите всех студентов, используя Get students запрос (см. результат рис. 22 а). Самостоятельно протестируйте оставшийся GET запрос, возвращающего конкретного студента по его id, указав имя в коллекции запросов как Get student by id. В итоге список запросов будет следующим (рис. 22 б). Удалите из проекта лишние классы: WeatherForecast.cs и WeatherForecastController.cs.



а



б

Рис. 22

а – Результат GET запроса, возвращающего список всех студентов.

б – Список запросов в Postman.

Создание микросервиса Course.

Нажмите ПКМ по решению → «Добавить» → «Создать проект» → «Веб-API ASP.NET Core» → «Далее» → В окне конфигурации введите имя проекта «**MicroserviceCourse**» → «Далее» → «Создать» (см. аналогично рис. 3 а-в). На этот раз мы будем использовать операторы верхнего уровня. Если вы не пользуетесь VS2022, создавайте и настраивайте проект по аналогии с первым сервисом.

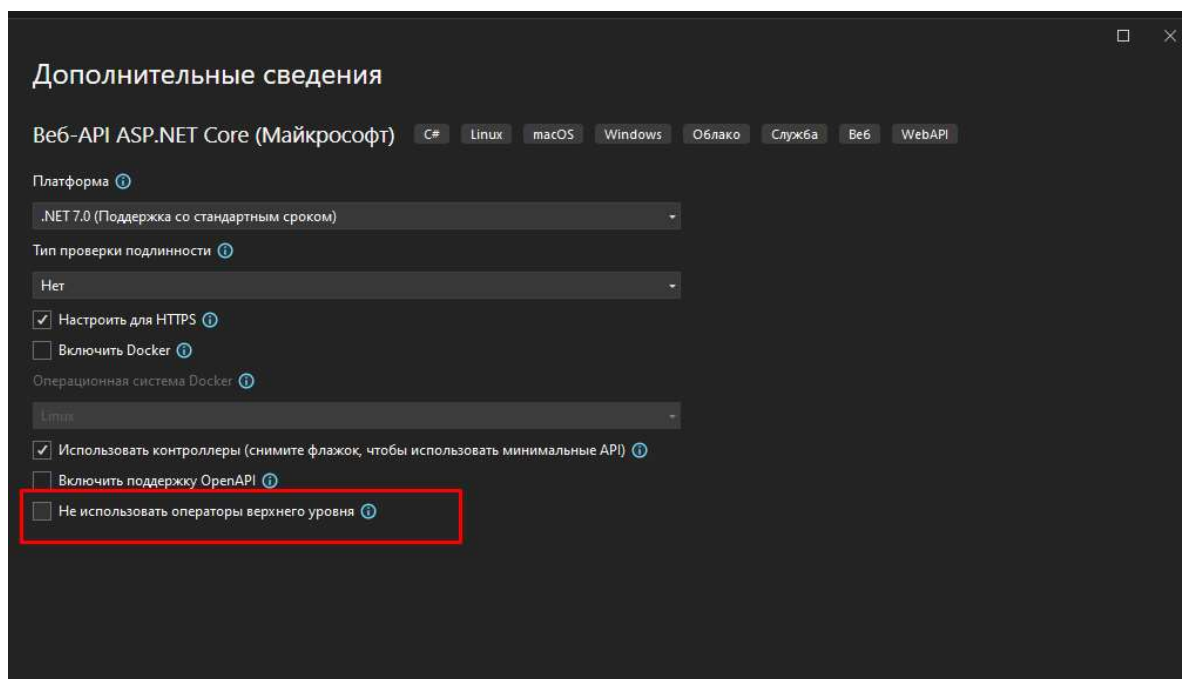


Рис. 23

Удалите классы WeatherForecast.cs и WeatherForecastController.cs. Создайте папку «Models» и добавьте класс Course. Опишите его следующим образом:

```
namespace MicroserviceCourse.Models
{
    public class Course
    {
        public long Id { get; set; }

        public string Name { get; set; }
    }
}
```

```

        public string Disciplenes { get; set; }
    }
}

```

В этот раз мы воспользуемся одной из особенностей микросервисной архитектуры – сделаем отдельную базу данных для этого сервиса, причем СУБД у нее будет не SqlServer, а PostgreSQL (вы можете продолжать использовать одного провайдера).

Откройте пакеты NuGet для решения (см. рис. 9) и перейдите во вкладку установленные. Установите каждый из пакетов в проект **MicroserviceCourse** (см. рис. 24), кроме **SqlServer** (или оставьте, если не будете использовать другую СУБД).

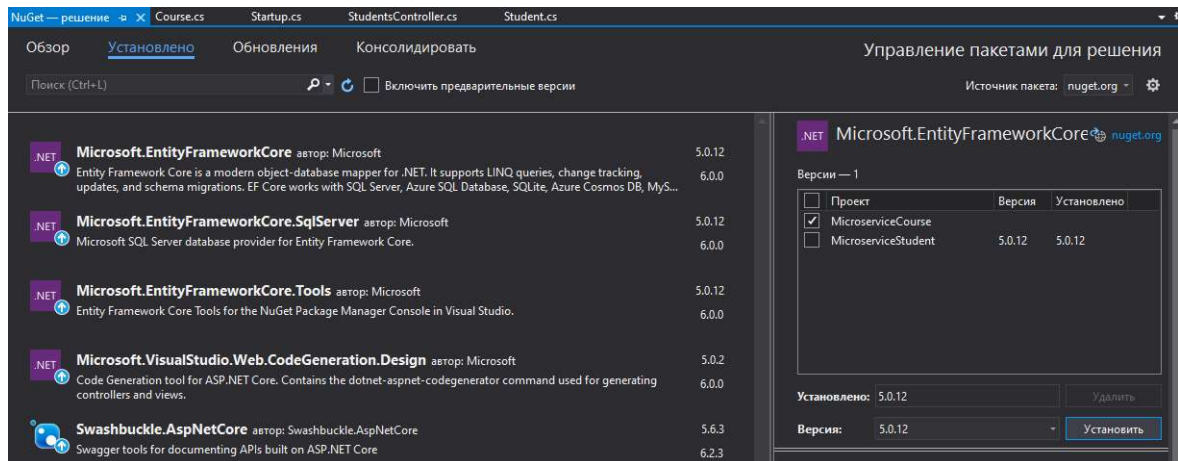


Рис. 24

Создание базы данных. Способ второй, Database first (если возникли проблемы или интересно, как делать по-другому).

Не всегда, однако, удастся безболезненно выполнять миграции способом Code first. Иногда выходят какие-то ломающие обновления, иногда миграции EF, как унифицированного инструмента просто не могут покрыть весь функционал конкретной БД т.к их не поддерживает, а иногда бы хорошо побыстрому прогнать миграции БД, но под рукой нет специализированных

средств, которые могут выполнить миграции на платформе .NET. В таком случае лучше воспользоваться классическим подходом к миграциям.

При данном подходе, нам нужно сначала создать базу данных и ее структуру. Когда БД создана можно перейти к ее настройке в EF. Для этого сначала установите еще один nu-get пакет: **Microsoft.EntityFrameworkCore.Relational**. Затем самостоятельно создайте базу данных с таблицей курсов в соответствии с моделью (см. рис 25).

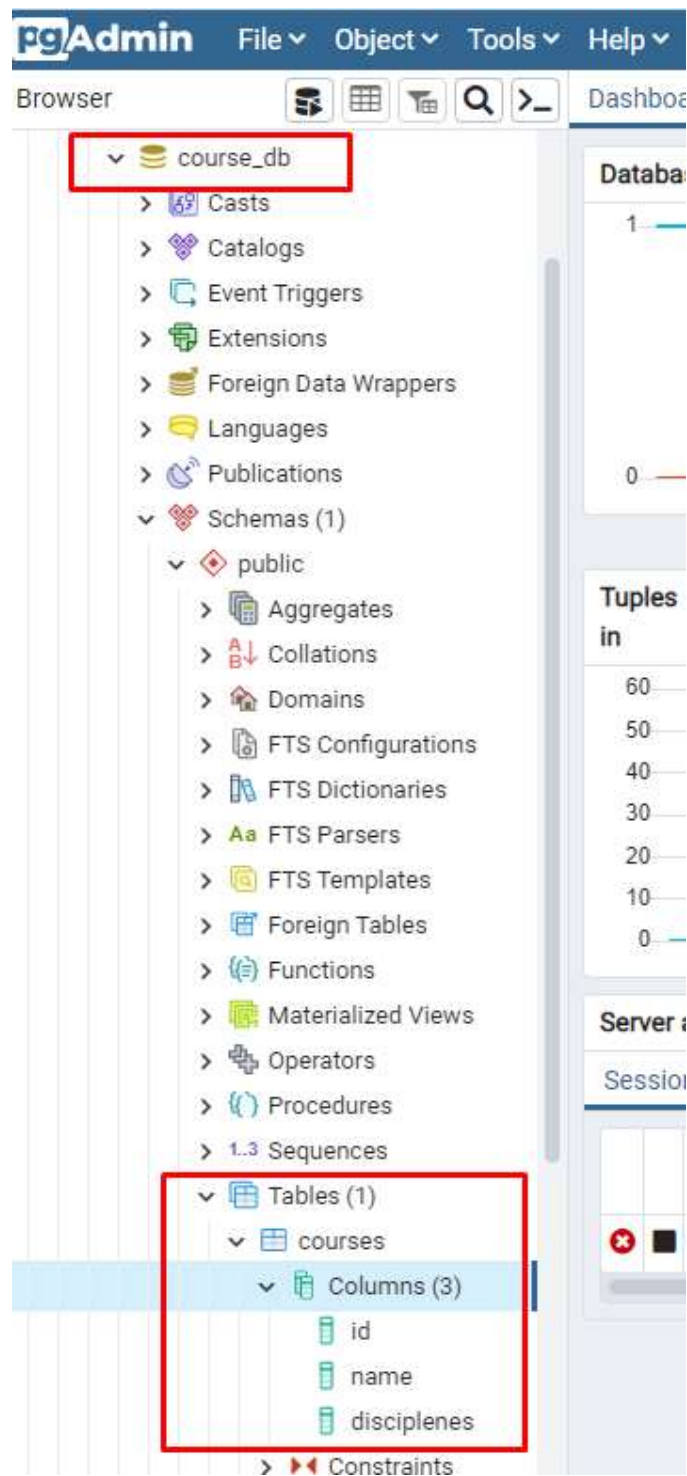


Рис. 25

Добавьте контекст базы данных для модели Course самостоятельно по аналогии с тем, как это делали для MicroserviceStudent. Перейдите в

управление nu-get пакетами и установите провайдера базы данных (см. рис. 26)

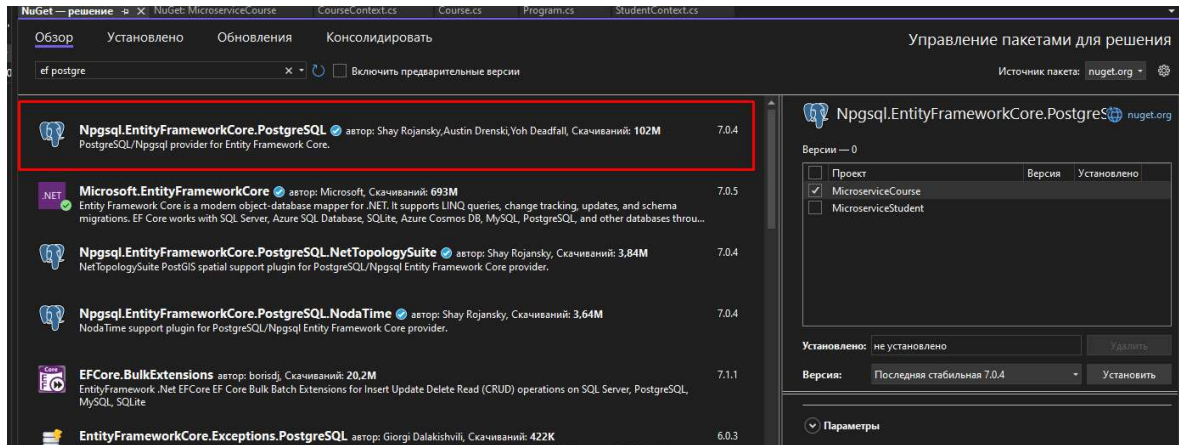


Рис. 26

Измените класс Program, зарегистрировав контекст хранилища следующим образом:

```
using MicroserviceCourse;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Add db
builder.Services.AddDbContext<CourseContext>(options =>

options.UseNpgsql("Host=localhost;Port=5432;Database=course_db;Username=postgres;Password=admin;IncludeErrorDetail=true"));

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Обратите внимание что при регистрации контекста мы используем `UseNpgsql`, а не `UseSqlServer`.

Теперь вернитесь в `CourseContext` и добавьте следующий метод:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Course>(entity =>
    {
        entity
            .ToTable("courses")
            .HasKey(e => e.Id);

        entity
            .Property(e => e.Id)
            .HasColumnName("id");

        entity
            .Property(e => e.Name)
            .HasColumnName("name");

        entity
            .Property(e => e.Disciplenes)
            .HasColumnName("disciplenes");
    });
}
```

Укажите свои имена столбцов и таблицы для этого объекта. В этом методе мы явно указываем EF, какова структура нашей БД и каким образом нужно создать объекты хранилища. Вы можете настроить подобное поведение для каждой сущности. Определить, что является ключом для данного класса, какие свойства сопоставляются с какими столбцами в БД. Также можно настраивать различные отношения, например один-ко-многим или многие-ко-многим, накладывать ограничения и собственные конвертации значений в конкретные типы (например, в перечисления или `Value object`) — обо всем этом подробнее вы можете прочитать самостоятельно.

Создайте контроллер `CourseController` по аналогии со `StudentController`.

Измените файл `«launchSettings.json»`, находящийся в папке `«Properties»` проекта `«MicroserviceCourse»` и обновите `«launchUrl»` с `«weatherforecast»` на `«api/course»`. Сгенерируйте контроллер для курсов аналогично тому, как это

было сделано для студентов. Запустите проект, предварительно изменив запускаемый проект (см. изменение проекта и результат на рис. 25).

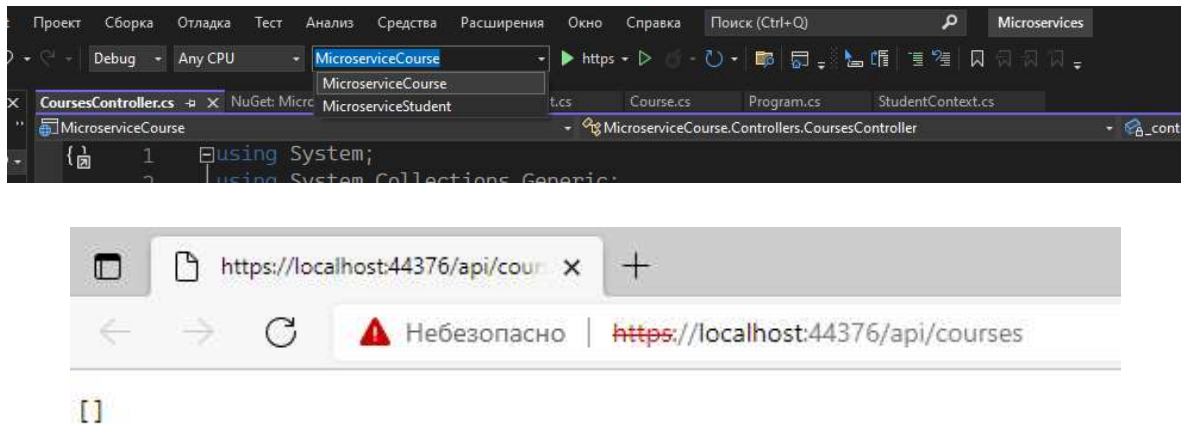


Рис. 27

Создайте новую коллекцию в Postman, назвав её «Microservice Course». Добавьте GET, POST, PUT, DELETE запросы по аналогии с тем, как это было сделано для микросервиса студентов (см. стр. 23). Не забудьте задать адрес, соответствующий вашему микросервису курсов (его можно узнать, запустив проект «MicroserviceCourse»). Отправьте следующие пост запросы по очереди, выбрав Body, raw, JSON (см. результат рис. 26):

```
{
  "name": "PRI-120",
  "disciplenes": "TP,MiPSV,MAD"
}
{
  "name": "IST-120",
  "disciplenes": "TP,MiPSV,MAD,Philosophy"
}
```

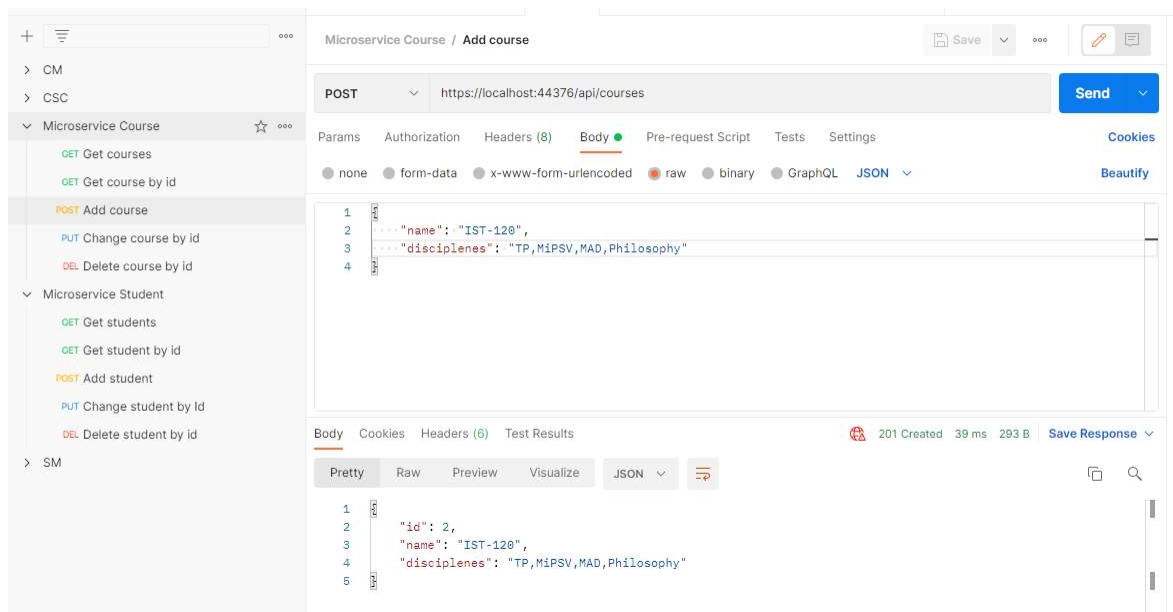


Рис. 26

Протестируйте работу остальных своих запросов самостоятельно.

Взаимодействие микросервисов между собой

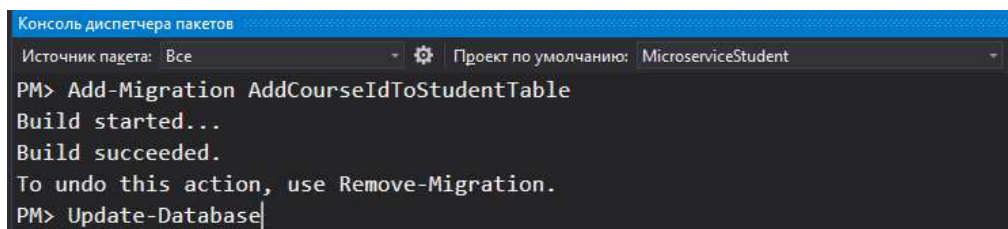
Взаимодействие между сервисами мы построим для простоты на REST API.

Для начала добавим новый атрибут к классу «Student» - идентификатор курса, таким образом класс будет выглядеть следующим образом:

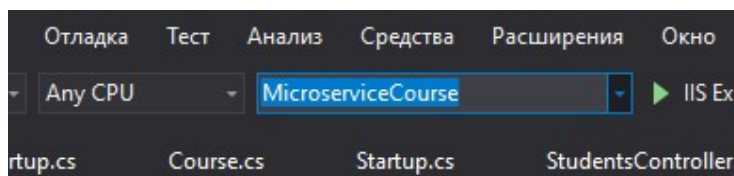
```
namespace MicroserviceStudent.Models;

public class Student
{
    public long Id { get; set; }
    public long CourseId { get; set; }
    public string Name { get; set; }
    public string GroupName { get; set; }
    public int Rating { get; set; }
}
```

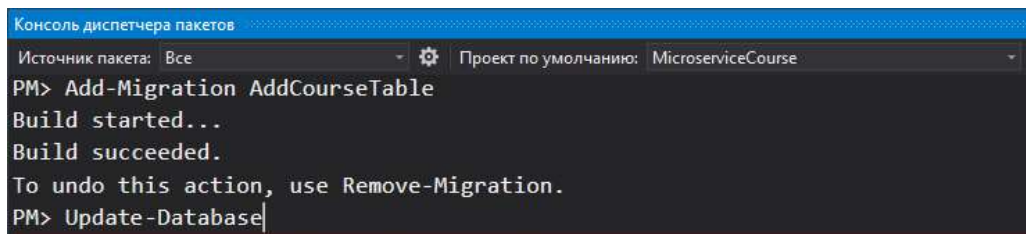
Если вы придерживались Code first. Откройте консоль диспетчера пакетов, выберите проект MicroserviceStudent и повторите команды (рис. 12), за исключением названия миграции – AddCourseIdToStudentTable (см. рис. 24 а). Затем укажите в VS в качестве запускаемого проекта MicroserviceCourse (см. рис 24 б) и выберите его же в списке проекта по умолчанию диспетчера пакетов. Выполните следующие команды (см. рис. 24 в)



а



б



```
Консоль диспетчера пакетов
Источник пакета: Все | Проект по умолчанию: MicroserviceCourse
PM> Add-Migration AddCourseTable
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
```

В

Рис. 27

а – Добавление изменений в таблицу студентов.

б – Изменение проекта в качестве запускаемого на MicroserviceCourse.

в – добавление в базу данных таблицы курсы.

В обозреватели объектов SQL Server убедитесь, что ваша база данных изменилась.

Если вы придерживались Database first. Создайте новый столбец в базе данных. Измените конфигурацию модели в вашем контексте, как это делалось для курсов:

```
entity.Property(e => e.CourseId).HasColumnName("COLUMN_NAME");
```

Вручную или с помощью запросов POSTMAN установите для своих студентов идентификаторы групп.

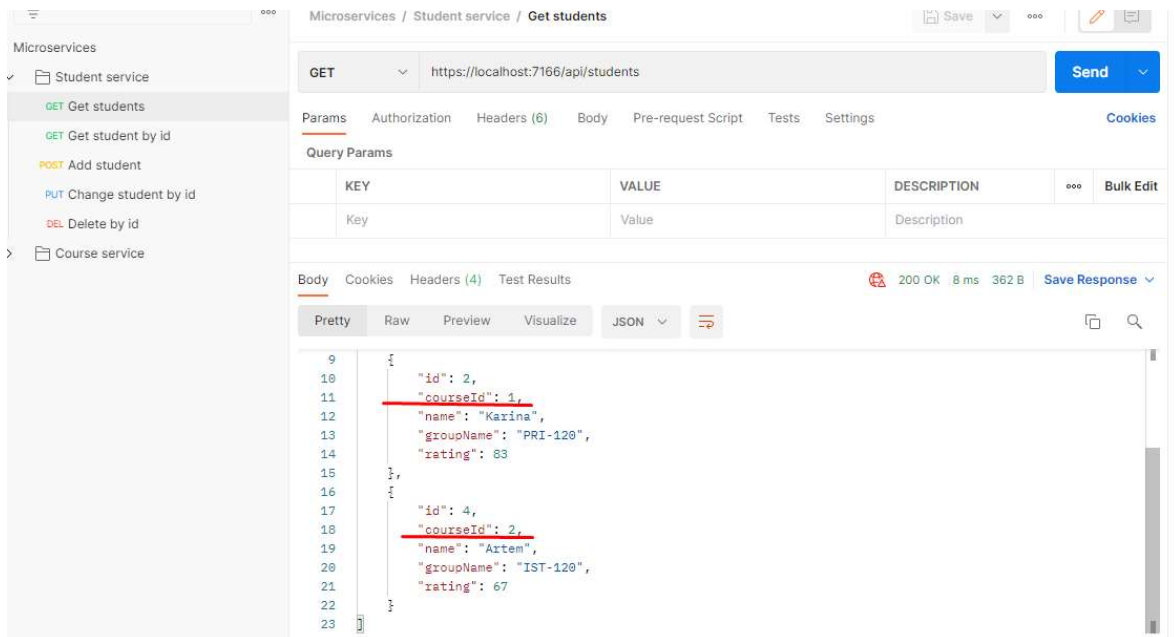


Рис. 28 – Установленные идентификаторы групп студентам.

Откройте «StudentController», подключите два пространства имен:

- `using System.Net.Http;`
- `using System.Text.Json;`

Создайте новый метод:

```

[HttpGet("course/{studentId}")]
public async Task<string> GetCourse(long studentId)
{
    var actionResult = await GetStudent(studentId);
    var student = actionResult.Value;
    HttpClientHandler clientHandler = new HttpClientHandler();

    clientHandler.ServerCertificateCustomValidationCallback =
        (sender, cert, chain, sslPolicyErrors) => { return true; };
    using (HttpClient client = new HttpClient(clientHandler))
    {
        HttpResponseMessage response = await
            client.GetAsync($"https://localhost:44376/api/courses/{
                student.CourseId}");
        if (response.IsSuccessStatusCode)
        {

```

```

        object course = await
        JsonSerializer.DeserializeAsync<object>(await
        response.Content.ReadAsStreamAsync());
        return course.ToString();
    }
}
return null;
}

```

Данный метод, достаёт из тела адреса id студента, вызывает метод асинхронно возвращающего объект типа ActionResult<Student>. Затем мы достаём из ответа сам экземпляр класса «Student». После чего в блоке «using» создается http клиент, с указанным выше обработчиком, благодаря которому не блокируются посылаемые запросы. Внутри блока мы отправляем GET запрос нашему микросервису курсов, обращаясь к его методу, возвращающего курс по его id. Получив положительный ответ, мы десериализуем ответ в формате JSON в тип object, так как у нас нет представления о моделях, используемых в микросервисе курсов, и возвращаем его строковый эквивалент.

Нажмите ПКМ по решению «Microservices» → «Свойства» → «Общие свойства» → «Запускаемый проект» → Выберите «Несколько запускаемых проектов» и укажите действие «Запуск» → «Применить» → «Ок». После чего запустите решение. Создайте новый запрос в Postman для коллекции «Microservice Student» и в качестве адреса установите маршрут до только что созданного метода и отправьте запрос (см. на рис. 28).

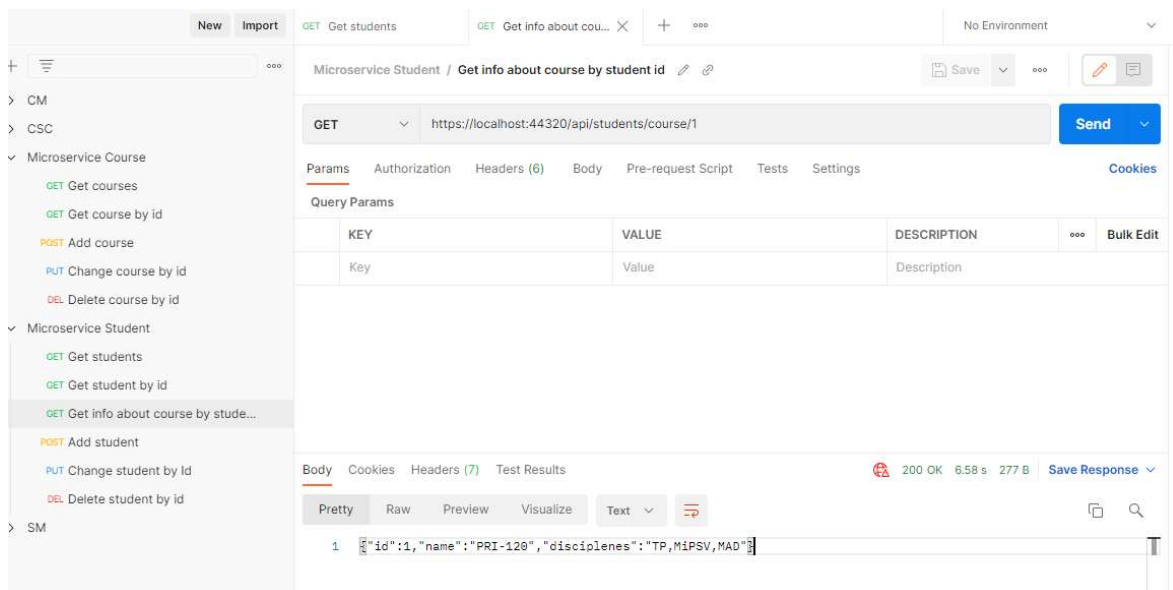
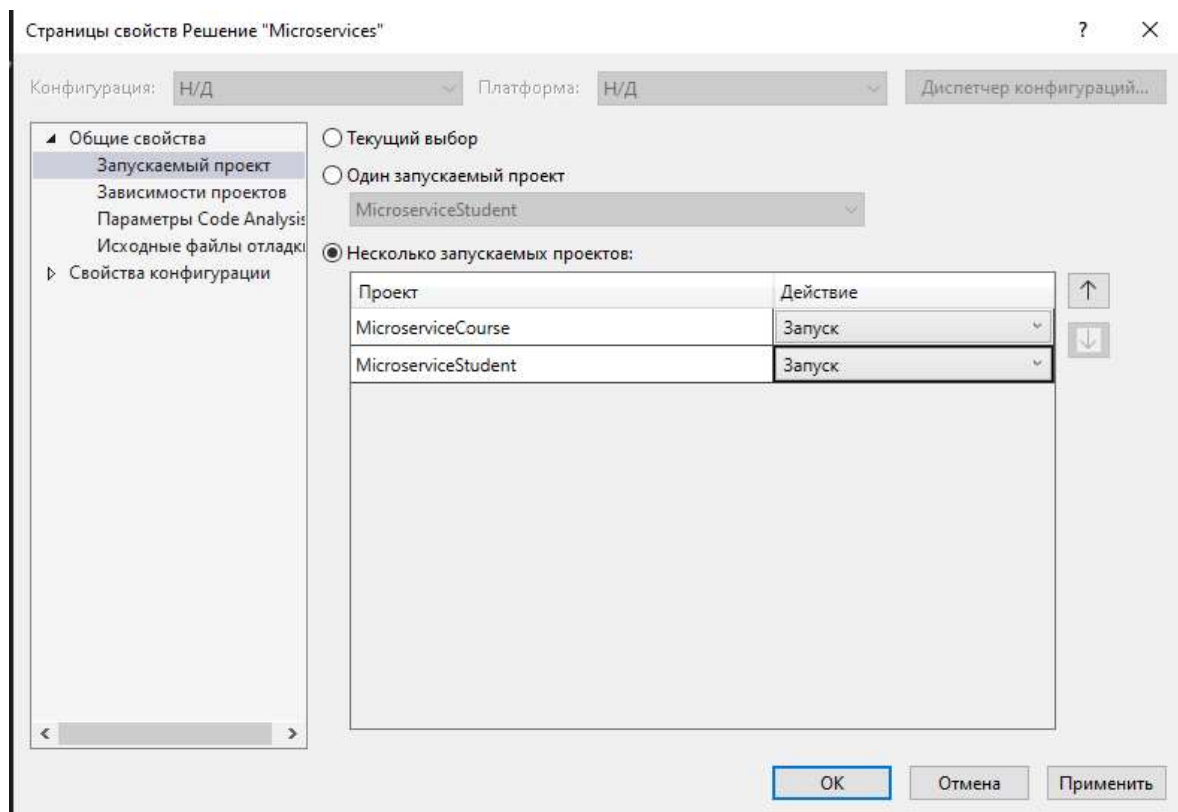


Рис. 28

Итак, мы реализовали два микросервиса, создав их модели, контроллер и контекст базы данных, реализовали простые запросы и один, реализующий их взаимодействие.

Заметим, что оба микросервиса используют свою БД, притом один использует SqlServer, а другой – PostgreSQL. Однако организация взаимодействия напрямую и без использования моделей, что, в общем-то, неправильно. Улучшением взаимодействия этих сервисов мы далее и займемся.

Создание композитного микросервиса

Удалим метод `GetCourse` из контроллера студентов, он больше не понадобится.

Добавьте в решение новый проект. Нажмите ПКМ по решению → «Добавить» → «Создать проект» → «Веб-API ASP.NET Core» → «Далее» → В окне конфигурации введите имя проекта «**MicroserviceCompositeSC**» → «Далее» → «Создать» (см. аналогично рис. 3 а-в). Удалите классы `WeatherForecast.cs` и `WeatherForecastController.cs`. Создайте папку «**Contracts**» и скопируйте туда классы «**Student**» и «**Course**» (обратите внимание что данные классы должны быть определены в пространстве имён «**MicroserviceCompositeSC.Models**») – в этой папке будут так называемые «транспортные модели», т.е. модели, которые описывают данные, которые производит или потребляет сервис. Создайте папку «**Models**» и модель с названием «**RatingOfGroup**» и опишите её следующим образом:

```
using System.Text.Json.Serialization;

namespace MicroserviceCompositeSC.Models;

public class RatingOfGroup
{
    [JsonPropertyName("groupName")]
    public string GroupName { get; set; }
    [JsonPropertyName("averageRating")]
    public double AverageRating { get; set; }
}
```

Атрибут «**JsonPropertyName**» указывает имя свойства, которое содержится в JSON при сериализации и десериализации.

Важно. Добавьте такие атрибуты к *каждой модели, включая **Student** и **Course*** в композитном сервисе, соблюдая их название в стиле `lowerCamelCase` по аналогии, как это сделано в примере выше для `RatingOfGroup`. Не забудьте подключить пространство имён – `using System.Text.Json.Serialization.`

В папке «Controllers» создайте класс, назвав его «CompositeSCController», и добавьте следующий код:

```
using Microsoft.AspNetCore.Mvc;
using MicroserviceCompositeSC.Models;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

namespace MicroserviceCompositeSC.Controllers;

[ApiController]
[Route("api/[controller]")]
public class CompositeSCController : ControllerBase
{
    [HttpGet]
    public string Start()
    {
        return "Composite is run!";
    }
}
```

Выше был создан контроллер с одним методом, возвращающего строку, при обращении по адресу api/compositesc. Откройте файл «launchSettings.json» и измените launchUrl на api/compositesc аналогично тому, как это делали для других сервисов. В качестве запускаемого микросервиса выберите только что созданный проект и запустите его (см. результат рис. 30).

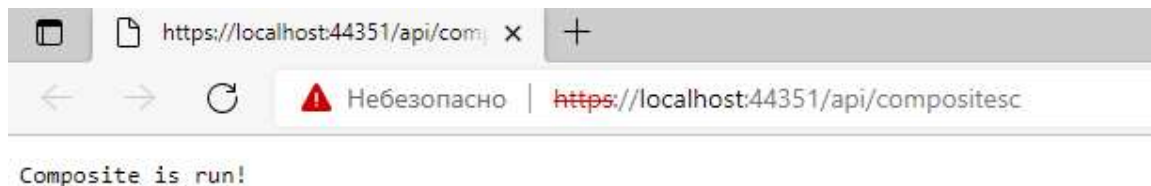


Рис. 30 – Работа композитного сервиса.

Расширим функционал нашего микросервиса – добавим несколько новых методов, реализующих обработку данных, получаемых из микросервисов «Student» и «Course», но для начала определим приватные поля (адреса расположения ваших сервисов могут быть отличными от примера):

```
private readonly string _studentServiceAddress =  
"https://localhost:7166/api/students";  
private readonly string _courseServiceAddress =  
"https://localhost:7105/api/courses";
```

Теперь добавим методы:

```
[HttpGet("courses/{disciplineName}")]  
public async Task<List<Course>> GetCoursesByDisciplineAsync(string  
disciplineName)  
{  
    HttpClientHandler clientHandler = new HttpClientHandler();  
    clientHandler.ServerCertificateCustomValidationCallback = (sender,  
cert, chain, sslPolicyErrors) => { return true; };  
    using (HttpClient client = new HttpClient(clientHandler))  
    {  
        HttpResponseMessage response = await  
client.GetAsync($" {_courseServiceAddress}");  
        if (response.IsSuccessStatusCode)  
        {  
            List<Course> courses = await  
JsonSerializer.DeserializeAsync<List<Course>>(  
                await response.Content.ReadAsStreamAsync());  
  
            return courses.Where(course =>  
                course.Disciplines  
                    .Split(',')  
                    .Contains(disciplineName))  
                .ToList();  
        }  
    }  
    return null;  
}
```

Этот метод обращается к контроллеру курсов с GET запросом по заданному адресу и десериализует его ответ в виде списка курсов, а затем возвращает коллекцию, состоящую из тех курсов, в списке дисциплин которых имеется имя искомой, переданной в адресе запроса к композитному сервису.

```
[HttpGet("students/{groupName}")]  
public async Task<List<Student>> GetStudentsByGroupAsync(string  
groupName)  
{  
    HttpClientHandler clientHandler = new HttpClientHandler();
```

```

        clientHandler.ServerCertificateCustomValidationCallback = (sender,
cert, chain, sslPolicyErrors) => { return true; };
        using (HttpClient client = new HttpClient(clientHandler))
        {
            HttpResponseMessage response = await
client.GetAsync($"{_studentServiceAddress}");
            if (response.IsSuccessStatusCode)
            {
                List<Student> students = await
JsonSerializer.DeserializeAsync<List<Student>>(
                    await response.Content.ReadAsStreamAsync());

                return students.Where(student => student.GroupName ==
groupName)
                    .ToList();
            }
        }
        return null;
    }
}

```

Данный метод обращается к контроллеру студентов с GET запросом по заданному адресу и десериализует его ответ в виде списка студентов, а затем возвращает коллекцию, состоящую из тех студентов, которые закреплены за указанным именем группы, взятым из адреса обращения запроса к композитному сервису.

```

[HttpGet("rating/{groupName}")]
public async Task<RatingOfGroup> GetAverageGroupRatingAsync(string
groupName)
{
    HttpClientHandler clientHandler = new HttpClientHandler();
    clientHandler.ServerCertificateCustomValidationCallback = (sender,
cert, chain, sslPolicyErrors) => { return true; };
    using (HttpClient client = new HttpClient(clientHandler))
    {
        HttpResponseMessage response = await
client.GetAsync($"{_studentServiceAddress}");
        if (response.IsSuccessStatusCode)
        {
            List<Student> students = await
JsonSerializer.DeserializeAsync<List<Student>>(
                await response.Content.ReadAsStreamAsync());

            var collection = students.Where(st => st.GroupName ==
groupName).ToList();
            long ratingSum = 0;

            foreach (var i in collection)
                ratingSum += i.Rating;

            return new RatingOfGroup()
            {
                GroupName = groupName,
                AverageRating = (double)ratingSum / collection.Count
            };
        }
    }
}

```



```

    return null;
}

```

Суть вышеприведенного метода заключается в отборе студентов, принадлежащих конкретной группе, расчету среднего рейтинга и возврат результата в виде объекта «RatingOfGroup», так как общение между микросервисами должно быть построено на моделях.

Откройте Postman и создайте все запросы, обращающиеся к композитному микросервису (см. пример рис. 31).

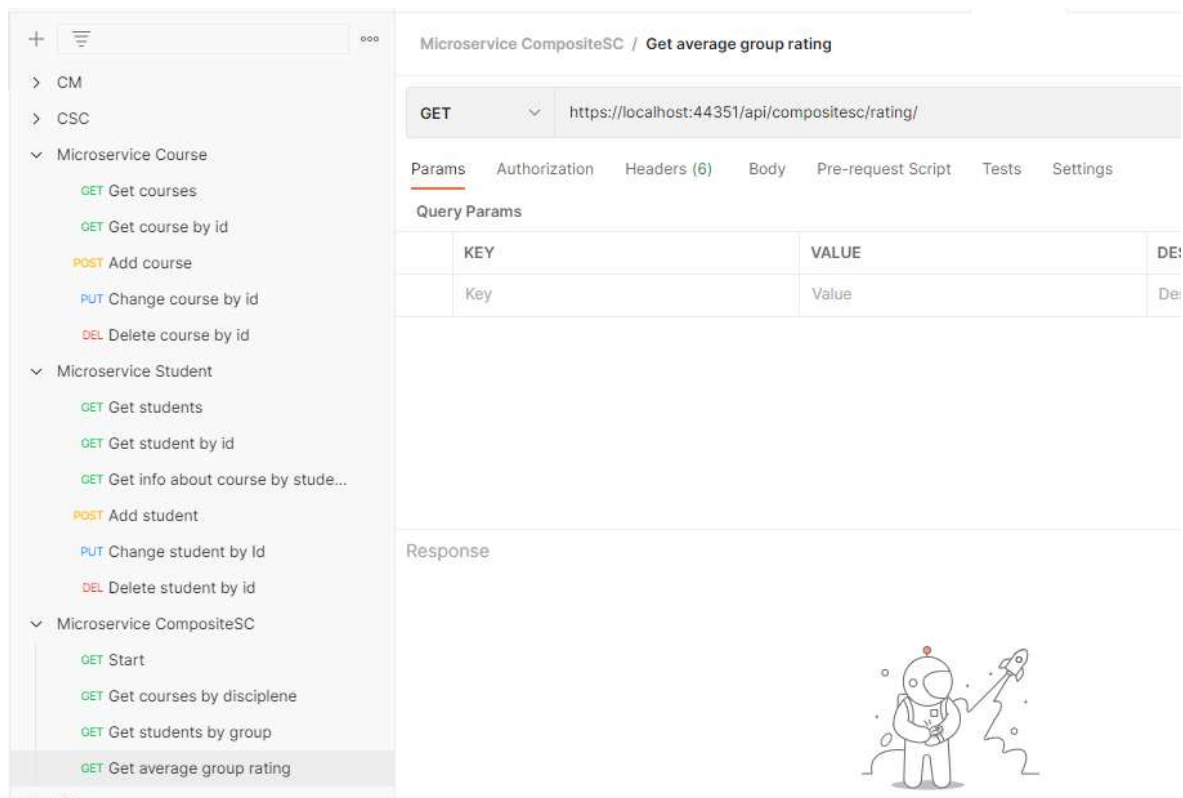
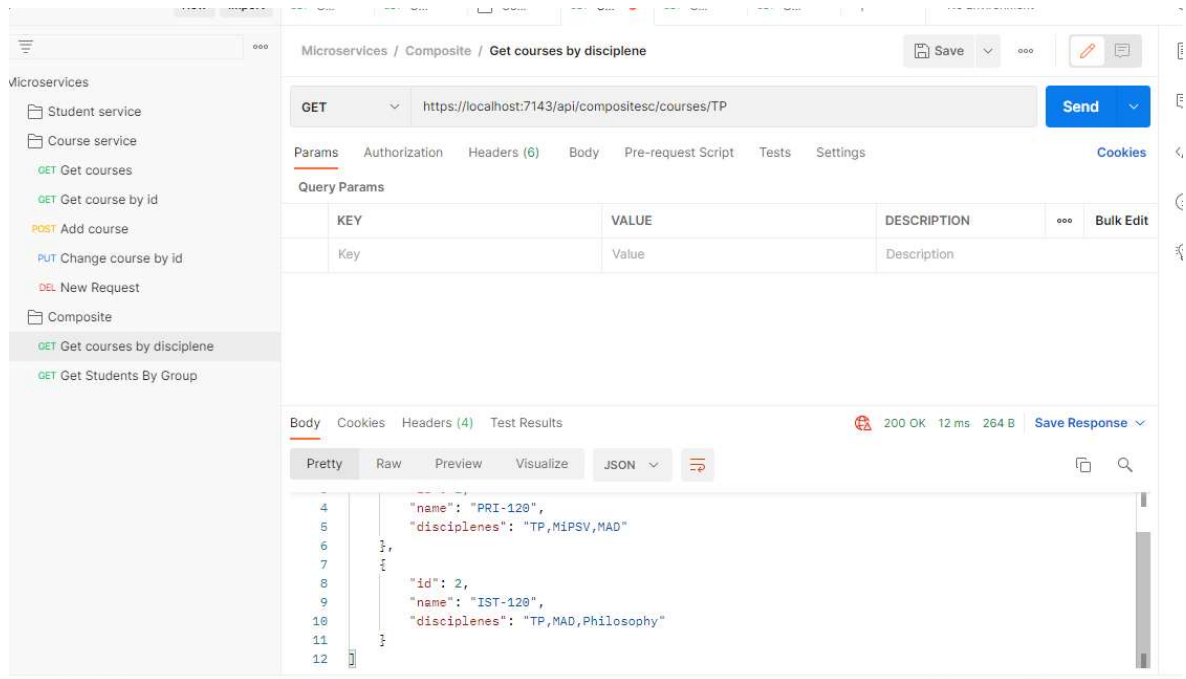


Рис. 31 – Создание запросов для композитного сервиса.

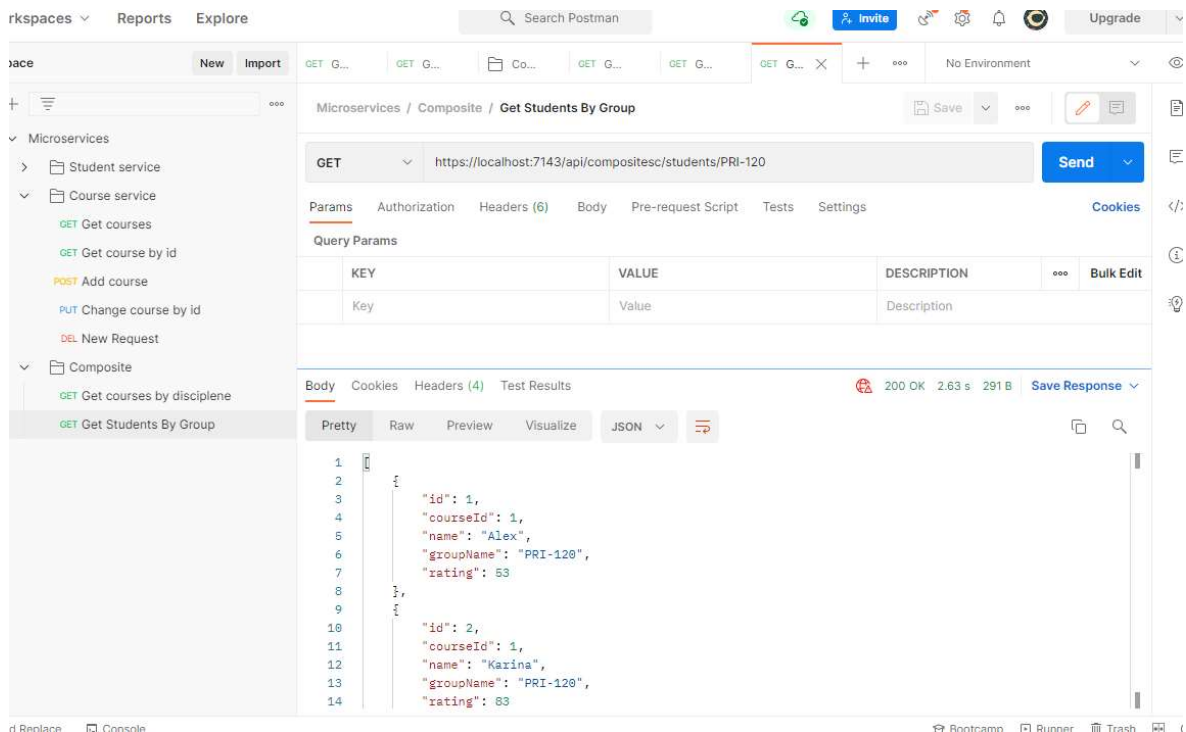
Тестирование работы приложения

Нажмите ПКМ по решению → «Свойства» → Во вкладке «запускаемый проект» выберите пункт «Несколько запускаемых проектов» и установите флаг «Действие» на «Запуск» у каждого проекта, затем нажмите

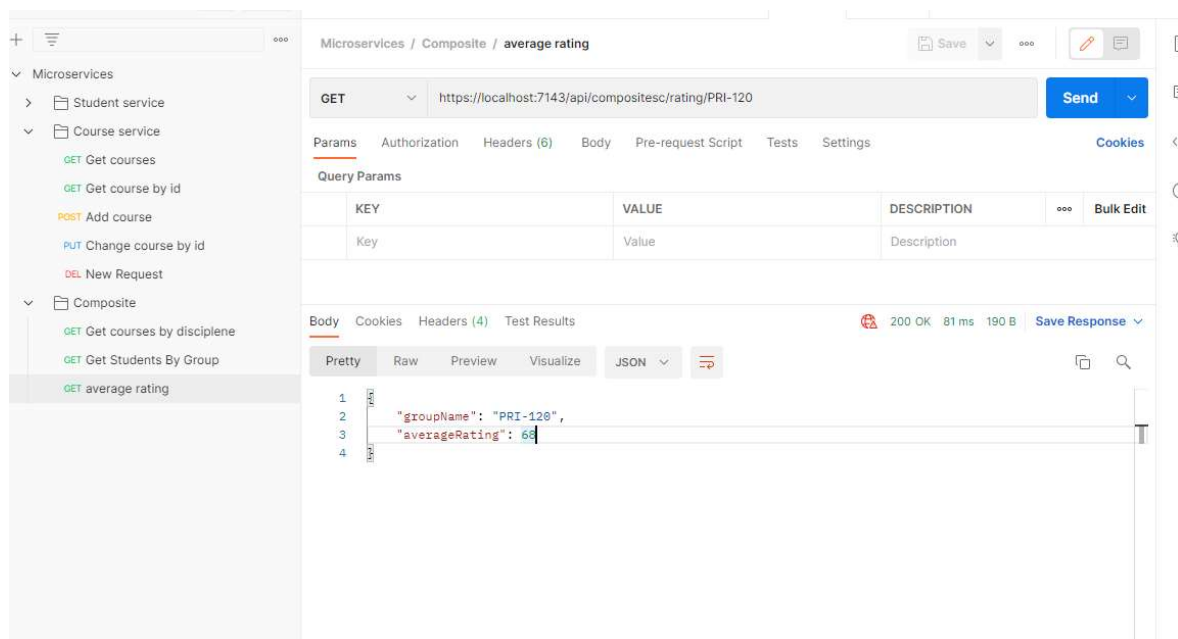
«Применить» и «Ок». Запустите проект. Протестируйте запросы композитного сервиса в Postman (см. результаты рис. 32 а-в)



a



6



В

Рис. 32

а – Результат GET запроса, возвращающего список курсов, содержащих указанную дисциплину.

б – Результат GET запроса, возвращающего список студентов по указанной группе.

в – Результат GET запроса, возвращающего средней рейтинг группы (в данном случае 53+83).

Заключение

В процессе работы были созданы несколько микросервисов и баз данных, налажено взаимодействие между микросервисами на основе моделей. Примерная схема организации приложения выглядит следующим образом.

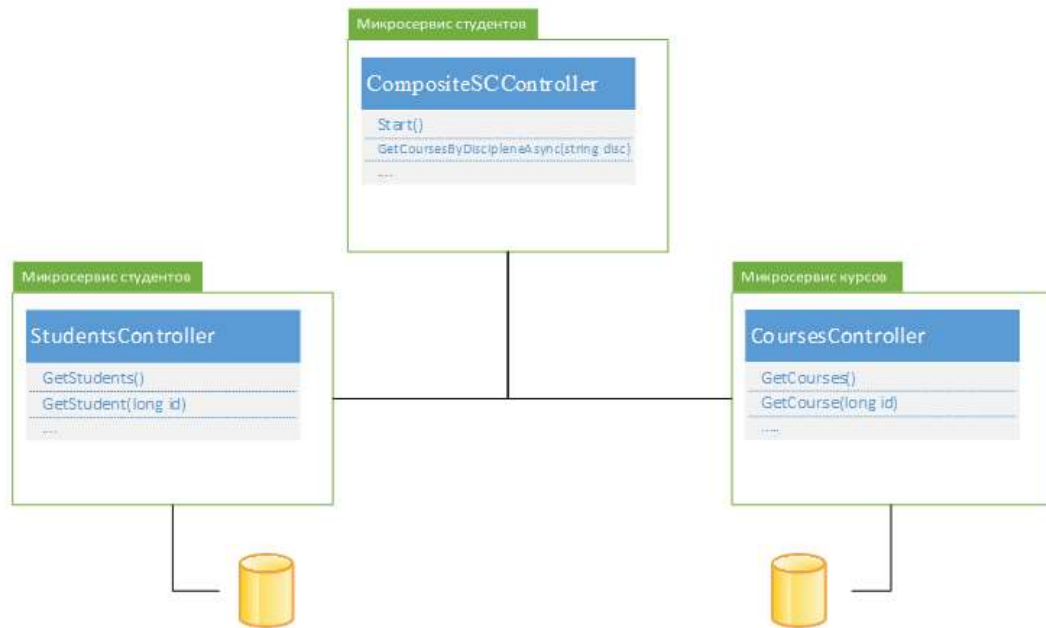


Рис. 33 Схема организации приложения.

Получившееся приложение полностью соответствует [репозиторию на GitHub](#) (ссылка).

Порядок выполнения работы:

- Ознакомиться с методическими указаниями.
- Реализовать свой проект по индивидуальному заданию, используя простейшую микросервисную архитектуру, как в примере, выбрав любой подход (code-first/database-first).
- Продемонстрировать работу приложения.
- Сформировать отчет по проделанной работе.

Содержание отчета:

- Цель работы.
- Описание вашего задания, предметной области.
- Фрагменты программы: модели и контроллеры разных сервисов.
- Изображения, иллюстрирующие работу приложения, протестированную с помощью POSTMAN.
- Выводы по работе.

Контрольные вопросы:

- Что представляет собой микросервисная архитектура?
- Каковы различия монолитной и микросервисной архитектур?
- Что такое микросервис?
- Как сервисы могут взаимодействовать между собой? Как взаимодействовали микросервисы в руководстве?

Рекомендуемая литература:

- Микросервисы // Мартин Фаулер : [сайт]. – 2014. – URL: <https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture>
- Ричардсон Крис // Микросервисы. Паттерны разработки и рефакторинга.