Микросервисы asp.net core web API. Часть 1

Первый микросервис

[Visual Studio Community 2019]



# На этом уроке

1. Повторим основы ООП и разберём синтаксический сахар в C#.
2. Создадим первый проект микросервиса asp.net core web api.
3. Разберёмся с архитектурой созданного автоматически микросервиса.
4. Разберёмся в протоколе http.
5. Научим свой микросервис отвечать на наши запросы.
6. Запустим свой микросервис на веб-сервере и выполним код запросов.
7. Научимся создавать микросервисы при помощи консольного инструмента dotnet.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_bq9iotvxzroo)

[Основы ООП](#_6i84yflmu02o)

[Анатомия класса и модификаторы доступа](#_bl093t1z7r6o)

[Абстракция и контракт](#_olmt3uhs51k4)

[Наследование](#_z5egle9kfkdp)

[Инкапсуляция](#_szfvmx85gop)

[Полиморфизм](#_op4sh8rqh6fs)

[Неявное наследование класса object](#_d63657h1s2vq)

[Конструкторы](#_doc15u5t2fsf)

[Синтаксический сахар в C# на примере foreach и yield](#_uqdpj3y76qca)

[Создаем проект в Visual Studio](#_v4xz8ufewj4q)

[Структура созданного проекта](#_feleuhrixrio)

[Паттерны разработки, что это и как использовать](#_6lu7y3y277z9)

[Паттерн “Цепочка обязанностей”](#_cnb0np3xm4s4)

[Program.cs](#_7s4rm8wwd1ba)

[Startup.cs](#_ouvryd7i63df)

[WeatherForecast.cs](#_pjnhilb3zgq1)

[WeatherForecastController.cs](#_ki69q46f66ps)

[Файлы конфигурации](#_dc8wns595cu3)

[launchSettings.json](#_7eseoh76s62o)

[appsettings.json](#_ei9fww6c5va7)

[Протокол HTTP](#_3sk6kv7pbsf)

[Основные понятия](#_tlz0iukie20h)

[Структура HTTP сообщения](#_ktsd09tse5lm)

[Методы HTTP сообщений](#_8hjvdxsqgao6)

[Заголовки](#_be6donh2nrul)

[Тело сообщения](#_7ppqhgoatlag)

[Выполнение HTTP запросов при помощи Postman](#_jsytkmeq4z5b)

[Операции CRUD](#_9cr0yosrjhix)

[Реализация методов CRUD через методы HTTP протокола](#_qxu8evqyt5cb)

[Первый контроллер с CRUD операциями](#_gaffvuwwogsl)

[Добавление нового web-api контроллера в проект](#_ipexya15qq5i)

[Добавление crud-методов в созданный контроллер](#_lww0rsdh3fgs)

[Сборка, отладка и проверка работоспособности созданного микросервиса](#_fuu52sytpjdl)

[Простой запуск из Visual Studio](#_xhapgztkq10n)

[Отладка запущенного микросервиса](#_6zrn611so6ti)

[Консольная утилита dotnet](#_rwyzyla43smc)

[Создание проекта web-api при помощи утилиты dotnet](#_dzjjgmhjpcic)

[Сборка проекта](#_e4gz7d162tj1)

[Запуск проекта при помощи утилиты dotnet](#_260qz9nwvdyx)

[Сборки](#_oq6duhx13atw)

[Манифест сборки](#_y6g07lfple4e)

[Практическое задание](#_p5h478pur572)

[Список используемой литературы](#_44o70szdx9u)

# Вступление

Приветствуем вас на курсе по ASP.NET Core! В нём мы рассмотрим наиболее актуальные технологии веб-разработки приложений. Основную часть курса мы уделим бэкэнд-разработке на C#. Мы изучим инструменты, построенные вокруг этой платформы, которыми пользуются программисты прямо сейчас при работе с реальным рабочими задачами. Во время прохождения курса мы разработаем приложение для сбора метрик (менеджер метрик) с компьютеров. Каждый урок содержит небольшое задание (фрагмент приложения), выполняя которое, к концу курса вы создадите полноценное современное веб-приложение на ASP.NET Core.

ASP.NET Core — перспективная платформа для разработки микросервисов. Различные компании, переживающие бурный рост, переходят от старых .NET-решений на новую платформу, и востребованность специалистов, которые смогут сразу начать решать продуктовые задачи, только растёт. С помощью ASP.NET Core можно создавать различные сервисы, которые обслуживают запросы как сайтов, так и мобильных приложений. На первом уроке курса мы познакомимся с основами разработки микросерисов asp.net core и попробуем запустить собственноручно написанный микросервис.

# Основы ООП

Язык C# — строго типизированный и объектно-ориентированный. Если вернуться к уже известным вам основам C#, можно заметить, что вся работа с кодом происходит через класс.

|  |
| --- |
| using System;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  Console.WriteLine("Это метод Main класса Program!");  }  } } |

То есть при каждом создании нового консольного приложения автоматически создавался приватный класс с именем Program и статичным методом Main.

Класс — ссылочная логическая структура. Разработчик проектирует класс в зависимости от того, что он должен делать — это может быть какой-либо внутренний сервис, либо менеджер, либо описание какой-либо сущности, которая хранит бизнес-информацию. Класс — это тип объекта, который впоследствии должен создаваться через оператора new.

|  |
| --- |
| using System;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  MyFirstClass myFirstClass = new MyFirstClass();    myFirstClass.SayHello("Василий");   Console.ReadLine();  }  }   public class MyFirstClass  {  public void SayHello(string yourName)  {  Console.WriteLine($"Hello, user {yourName} from class MyFirstClass!");  }  } } |

Результат программы — вывод строки на консоли Hello, user Василий from class MyFirstClass!

Класс может быть как простым и содержать минимальное количество логики и информации, так и сложным, где, соответственно, есть много логики и информации.

Если попытаться спроецировать класс на обычную жизнь, то можно представить автомобиль. Чертёж машины — это класс. Сама машина — это уже объект, так как она уже создана по чертежу. При создании чертежа машины используется куча маленьких деталей — от мотора до болтиков, это тоже по факту классы, которые становятся объектами при сборке машины на конвейере.

Перенесём примерную иерархию класса машины и класса мотора и логику включения машины.

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Car car = new Car();   car.Motor = new MotorEngine();   bool isCarStarted = car.Start();   if (isCarStarted)  {  Console.WriteLine("Машина заведена");  }  else  {  Console.WriteLine("Машина не смогла завестись");  }   Console.ReadLine();  }  }   public class Car  {  public MotorEngine Motor { get; set; }   public bool Start()  {  if (Motor is null)  {  return false;  }   if (Motor.IsWorking)  {  return true;  }   return Motor.TurnOn();   }  }   public class MotorEngine  {  private bool \_isWorking;   public bool IsWorking  {  get { return \_isWorking; }  }   public bool TurnOn()  {  \_isWorking = true;   return \_isWorking;  }   public bool TurnOff()  {  \_isWorking = false;   return \_isWorking;  }  } |

Обратите внимание на то, что после создания объекта Car мы присвоили в его свойство Motor новый объект — MotorEngine. Если этого не сделать, то машина не сможет завестись, так как по логике у машины будет отсутствовать мотор.

## Анатомия класса и модификаторы доступа

Проектирование классов и объектов поначалу может сильно путать разработчика и усложнять ему жизнь. Чтобы разобраться в этой теме получше, рассмотрим анатомию класса.

|  |  |
| --- | --- |
| **Область класса** | **Описание** |
| *Поля* | Под полями подразумевают закрытую информацию, состояния и т. д.  Её видно только внутри класса и иногда наследникам. Используется очень часто. Синтаксис схож с обычным переменными. Поля в основном идут сразу после заглавия класса. |
| *Свойства* | Свойства скрывают под собой доступ к полям и состоянию класса. Они могут быть публичными и локальными. Зависит от требований и постановки задачи. Группируются по модификаторам доступа. |
| *Метод* | Методы будут уже логикой, в которой оперируют переменные, поля и даже свойства. Методы могут быть так же, как и поля, публичные и локальные. Так же, как и у свойства, группировка идёт по модификаторам доступа. |
| *Конструктор* | Конструкторы — это специализированные методы, которые инициализируют объект посредством оператора new. У них есть чёткий набор правил — имя должно совпадать с именем класса, и они не должны ничего возвращать, в отличие от метода. Конструкторов может быть много, т. к. объект может инициализироваться по-разному в зависимости от архитектуры и задачи. И посредством ключевых слов base и this можно задавать цепочку вызовов нескольких конструкторов. Конструкторы в основном идут сразу за полями. |
| *Финализатор* | Это особые методы, которые обязаны автоматически вызываться при окончании жизни объекта, перед сборкой мусора. Крайне не рекомендуются и практически не используются в работе, но о них очень часто спрашивают на собеседованиях. Они будут рассмотрены в уроке по управлению памяти. |

C# прошёл большую историю изменения синтаксиса. Почти каждый год добавляются различные новые возможности написания одних и тех же областей класса разным синтаксисом. Но поведение и смысл от этого не сильно меняется.

|  |
| --- |
| public class Car  {  */// <summary>*  */// Локальное поле, которое видно только внутри класса*  */// и установить значения можно только через конструктор класса,*  */// т. к. стоит ключевое слово readonly*  */// </summary>*  private readonly string \_markName;  */// <summary>*  */// Локальное поле, которое видно только внутри класса*  */// и его значение может меняться извне либо внутри класса,*  */// т. к. не стоит ключевое слово readonly*  */// </summary>*  private MotorEngine \_motor;   */// <summary>*  */// Конструктор по умолчанию.*  */// В качестве марки машины устанавливается значение "Unknown Mark"*  */// посредством вызова другого конструктора*  */// public Car(string markName)*  */// </summary>*  public Car()  :this("Unknown Mark")  {   }   */// <summary>*  */// Конструктор с принудительным присвоением марки машины*  */// </summary>*  */// <param name="markName">Марка машины</param>*  public Car(string markName)  {  \_markName = markName;  }   */// <summary>*  */// Конструктор, через который устанавливаются все поля —*  */// от марки машины до мотора.*  */// </summary>*  */// <param name="markName"></param>*  */// <param name="engine"></param>*  public Car(string markName, MotorEngine engine)  {  \_markName = markName;  \_motor = engine;  }   */// <summary>*  */// Публичное свойство только на чтение.*  */// Ни один потребитель извне не сможет его поменять,*  */// т. к. здесь отсутствует set*  */// </summary>*  public string MarkName => \_markName;   */// <summary>*  */// Публичное свойство, которое может быть изменено извне,*  */// т. к. здесь есть set*  */// </summary>*  public MotorEngine Motor  {  get => \_motor;  set => \_motor = value;  }   */// <summary>*  */// Публичный метод, который доступен извне класса потребителю.*  */// Т. е. его можно вызвать у объекта*  */// </summary>*  */// <returns>Значение</returns>*  public bool Start()  {  if (Motor is null)  {  return false;  }   if (Motor.IsWorking)  {  return true;  }   return Motor.TurnOn();   }  } |

Как можно заметить, в усовершенствованный класс Car были добавлены дополнительные конструкторы, что позволит инициализировать объект более гибко.

К каждому блоку класса и к классу самому всегда добавляется модификатор доступа. Модификатор доступа — мощнейший инструмент при проектировании класса. При грамотном использовании модификаторов доступов можно достичь хорошего и логически верного кода, который будет понятен любому программисту без сильного углубления в чтение кода.

|  |  |
| --- | --- |
| **Модификатор доступа** | **Описание** |
| public | Публичный, может быть доступ извне. |
| private | Только локальный доступ. Не видно даже наследникам класса. |
| protected | Схож с private, но наследники могут использовать. |
| internal | Схож с public, но доступ только в радиусе сборки. |
| protected internal | Комбинация protected и internal. Доступ в радиусе одной сборки и в наследниках. |
| private protected | Комбинация private и protected. Доступ возможен локально и в наследниках в радиусе текущей сборки |

Для чего существуют эти модификаторы доступа? Ответ достаточно прост. При проектировании классов хороший тон — предоставлять наружу только то, что требуется потребителю и не больше. В итоге может создаться ситуация, что есть огромный класс, в котором более 20 методов, но публичными может быть всего несколько. Всё остальное скрыто от конечного потребителя и используется внутри самого класса, либо в локальной сборке по отношению к классу, т. к. это именно внутренняя логика, и она не предназначена для доступа извне.

Стоит учитывать, что принято писать классы в разных файлах. Имя файла в большинстве случаях должно совпадать с именем класса. Вполне допустимо писать и в одном файле. Это можно наблюдать в примере с машиной и мотором. Стоит учитывать, что это довольно редко применяется и старайтесь не делать так, т. к. навигация по коду усложняется в разы. Хотя есть исключение — статические классы с методами-расширениями.

И ещё один метод расположения кода класса — вложенный класс. Метод довольно сильно распространен, в особенности, если это дополнительные небольшие классы по отношению к главному классу (который указывается в названии файла).

|  |
| --- |
| using System;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  BigClass bigClass = new BigClass();   *//Обратите внимание, что создание вложенного класса InnerClass*  *//идет через название материнского класса BigClass.*  BigClass.InnerClass innerClass = new BigClass.InnerClass();   Console.ReadLine();  }  }    public class BigClass  {  public BigClass()  {  Console.WriteLine("Я супер-пупер большой класс!");  }   */// <summary>*  */// Идет определение вложенного класса*  */// </summary>*  public class InnerClass  {  public InnerClass()  {  Console.WriteLine("А я вложенный класс!");  }  }  } } |

К вложенным классам применяются всё те же правила, что и к обычным.

## Абстракция и контракт

Идеология ООП строится на трёх базовых принципах — наследование, инкапсуляция и полиморфизм. Каждый из этих принципов важен и даёт разработчику мощнейший инструмент в разработке программного обеспечения. Разберём каждый термин поочерёдно простыми словами.

### Наследование

Наследование — это когда один класс наследует поведение и свойства другого.

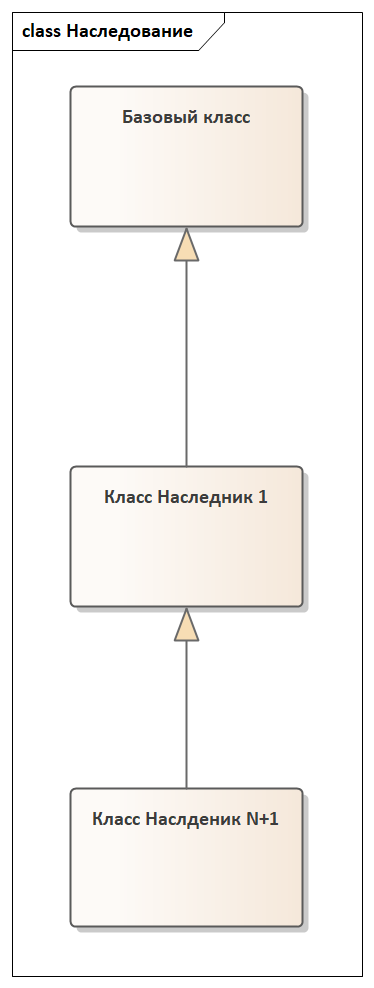
|  |
| --- |
| using System;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  BaseClass baseClass = new BaseClass();  *//Доступен только метод Foo, т. к. он публичный*  baseClass.Foo();   ChildClass childClass = new ChildClass();   *//Доступен публичный метод*  childClass.SomeMethod();  *//Доступен и метод у базового класса, т. к. он публичный*  childClass.Foo();   Console.ReadLine();  }  }   */// <summary>*  */// Базовый класс*  */// </summary>*  public class BaseClass  {  public void Foo()  {  Console.WriteLine("Вызывается у базового класса!");  }   protected void ChildFoo()  {  *//Какая либо логика*  }  }   */// <summary>*  */// Наследник*  */// </summary>*  public class ChildClass : BaseClass  {  public void SomeMethod()  {  *//Вызываем у базового класса метод*  *//с модификатором доступа protected т. к. это наследник*  ChildFoo();  }  } } |

Как можно заметить, чтобы унаследовать один класс от другого, нужно просто указать его через двоеточие после названия дочернего класса. Теперь наследник может использовать свойства, методы и поля, которые разрешают доступ наследникам.

Данный принцип применяется очень часто. Зачастую можно встретить довольно большую цепочку наследования. Соответственно, в C# не ограничены вложенности наследования.

|  |
| --- |
| */// <summary>*  */// Базовый класс*  */// </summary>*  public class BaseClassFirst  {    }   */// <summary>*  */// Наследник #1*  */// </summary>*  public class ChildClassFirst : BaseClassFirst  {    }   */// <summary>*  */// Наследник #2, унаследует все от ChildClassFirst и BaseClassFirst*  */// </summary>*  public class SecondChildClass : ChildClassFirst  {   } |

Такое наследование можно наглядно отобразить в диаграмме.



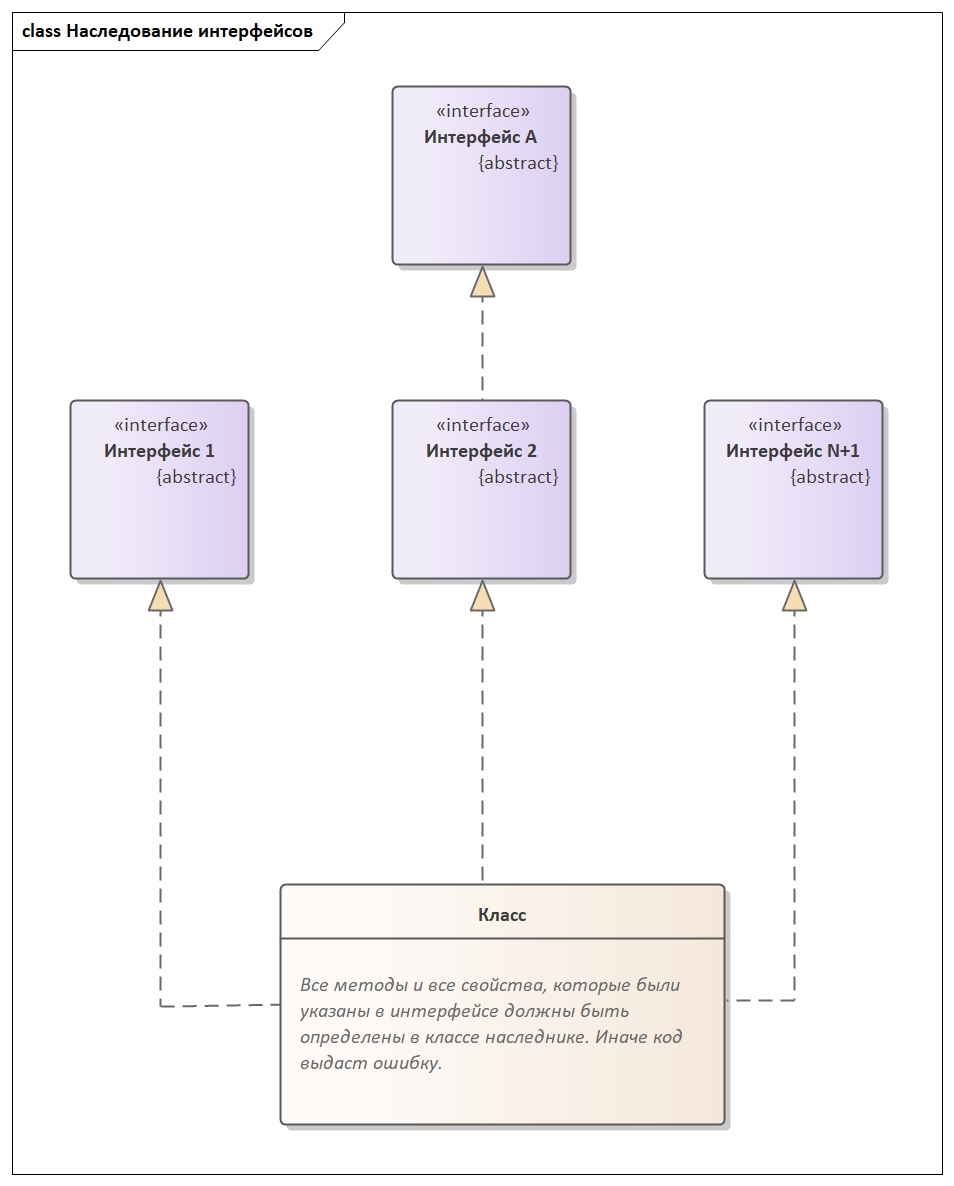
Но стоит помнить, что наследоваться можно только от одного класса за раз. Множественное наследование в C# запрещено.

|  |
| --- |
| */// <summary>*  */// Базовый класс*  */// </summary>*  public class BaseClassFirst  {    }   public class BaseClassSecond  {   }   */// <summary>*  */// Наследник, данный код приведёт к ошибке*  */// CS1721 Class 'ChildClassFirst' cannot have multiple*  */// base classes: 'BaseClassFirst' and 'BaseClassSecond'*  */// </summary>*  public class ChildClassFirst : BaseClassFirst, BaseClassSecond  {    } |

Для множественного наследования в C# были введены интерфейсы. Интерфейс схож по синтаксису с классом, но имеет строгие ограничения до версии C# 8.0, о которых и пойдёт речь далее.

|  |
| --- |
| */// <summary>*  */// Базовый класс*  */// </summary>*  public class BaseClassFirst  {    }   */// <summary>*  */// Интерфейс 1*  */// </summary>*  public interface IFirst  {  void SomeFirstMethod(int yourAge);  }   */// <summary>*  */// Интерфейс 2*  */// </summary>*  public interface ISecond  {  void SomeSecondMethod(string yourName);  }     */// <summary>*  */// Наследник*  */// </summary>*  public class ChildClassFirst : BaseClassFirst, IFirst, ISecond  {  */// <summary>*  */// Методы, свойства из интерфейса обязаны*  */// быть определены в наследнике*  */// </summary>*  */// <param name="yourName"></param>*  public void SomeSecondMethod(string yourName)  {  System.Console.WriteLine($"Your name is {yourName}!");  }   */// <summary>*  */// Методы, свойства из интерфейса обязаны*  */// быть определены в наследнике*  */// </summary>*  */// <param name="yourName"></param>*  public void SomeFirstMethod(int yourAge)  {  System.Console.WriteLine($"Your age is {yourAge}!");  }  } |

В интерфейсе должны быть указаны только определения публичных свойств и методов. Но без реализации. Вся реализация должна быть уже в наследуемых потомках. Интерфейс также можно наследовать, но только от другого интерфейса.



Объект впоследствии можно приводить к интерфейсу. Например, если взять класс List<T> и посмотреть его иерархию наследования на docs.microsoft.com, то его можно привести к любому интерфейсу, от которого он был унаследован.

|  |
| --- |
| public class List<T> :  System.Collections.Generic.ICollection<T>, System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>, System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>,  System.Collections.IList |

И его приведение:

|  |
| --- |
| using System; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  IEnumerable<object> list = new List<object>();   Console.ReadLine();  }  } } |

Когда объект приведён к интерфейсу, то доступны только те методы и свойства, которые указаны в интерфейсе. Именно интерфейсом можно достичь в большинстве случаях того, чтобы конечный потребитель получил список методов и свойств — только то, что ему нужно и не более. В целом сокрытие это и есть один из принципов ООП — инкапсуляция.

### Инкапсуляция

Итак, инкапсуляция — это сокрытие реализации от конечного потребителя. Она достигается именно использованием модификаторов доступа свойств и методов, которые были описаны выше. Когда разработчик использует модификаторы доступа, он руководствуется одним простым правилом:

Конечный потребитель должен получить тот набор методов и свойств, который ему нужен, и он будет им пользоваться. Все остальное ему должно быть недоступно.

Почему такое правило существует? Представьте себе, вы спроектировали большой класс-сервис, который делает сложные математические калькуляции банковских переводов. В этом классе много методов, которые вызываются по цепочки в зависимости от времени проведения банковской транзакции, вида транзакции и дополнительных факторов. Но цепочку выполнения запускает всего лишь один метод StartTransaction(IBankAccountTransaction transaction) Все остальные методы нужны только для внутреннего использования, и если их сделать публичными, то конечному потребителю будет сложно понять какой метод нужно запустить, т.к. их может оказаться значительное количество.

При проектировании класса всегда старайтесь держать это правило в голове. Даже если не совсем понятно, какие методы могут быть доступны, а какие нет – делайте их по умолчанию приватным и постепенно открывайте, когда они будут требоваться. И определяйте их в интерфейсах, чтобы скрыть ненужные методы в зависимости от потребителя.

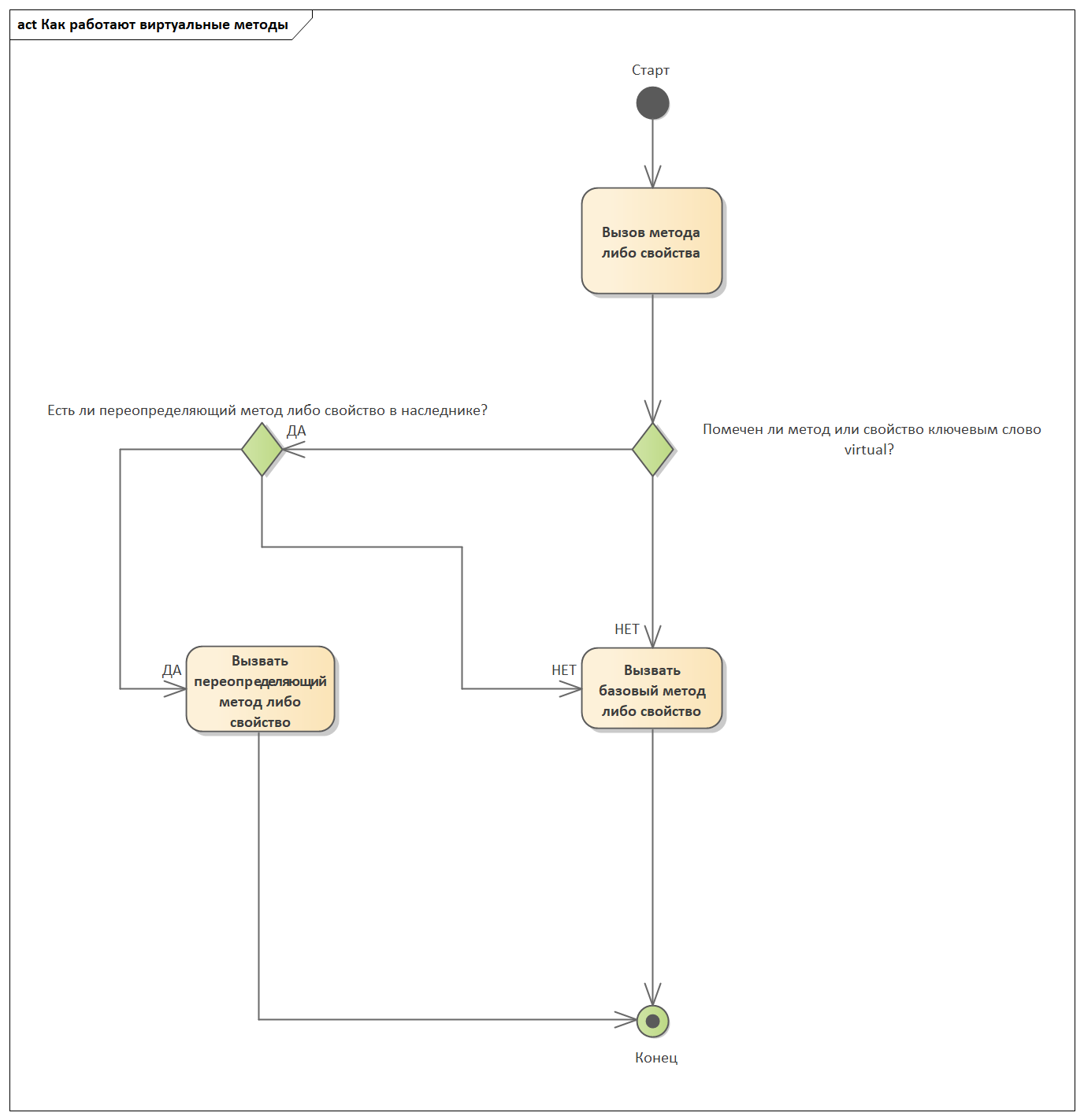
### Полиморфизм

Если говорить кратко о полиморфизме, то это возможность переопределять поведение базовых свойств и методов в наследниках. Давайте разберем пример с использованием новых ключевых слов abstract, virtual и override.

|  |
| --- |
| using System; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  BaseClassFirst baseClass = new ChildClassFirst();    baseClass.SayHello("Василий");    baseClass.SayGoodbye("Василий");   Console.ReadLine();  }  }   */// <summary>*  */// Базовый абстрактный класс*  */// </summary>*  public abstract class BaseClassFirst  {  */// <summary>*  */// Только в абстрактном классе*  */// можно определить абстрактное свойство*  */// </summary>*  public abstract string Name { get; set; }   */// <summary>*  */// Только в абстрактном классе*  */// можно определить абстрактный метод*  */// </summary>*  */// <param name="yourName"></param>*  public abstract void SayHello(string yourName);   */// <summary>*  */// Виртуальные методы могут содержать логику по умолчанию,*  */// но могут быть и переопределены у наследника*  */// </summary>*  */// <param name="yourName"></param>*  public virtual void SayGoodbye(string yourName)  {  System.Console.WriteLine("Метод не переопределен");  }  }    */// <summary>*  */// Наследник*  */// </summary>*  public class ChildClassFirst : BaseClassFirst  {  public override string Name { get; set; }   public override void SayHello(string yourName)  {    }  } } |

Класс может быть абстрактным, как показано выше. Это означает то, что его нельзя создать через оператор new. Можно только создать его не абстрактных наследников. И только у такого класса могут быть декларированы абстрактные свойства и методы, которые, как и в интерфейсе не должны содержать никакой реализации. Такие методы и свойства обязаны определиться в наследнике, используя ключевое слово override. Если их не переопределить, то это приведет к ошибке. В противоположном случае виртуальные методы могут содержать логику по умолчанию для всех наследников, но и могут также быть переопределены в наследниках. Обратите внимание на то, что объект можно привести к абстрактному типу, ровно так же, как и с интерфейсами.

Для наглядности работу вызова виртуальных методов либо свойств можно нарисовать в диаграмме



### Неявное наследование класса object

Если быть внимательным, то можно заметить, что даже пустой класс без методов и свойств по умолчанию уже содержит набор методов. Рассмотрим самые важные

|  |  |
| --- | --- |
| **Модификатор доступа** | **Описание** |
| Equals | Проверяет равенство объектов |
| GetHashCode | Получает хэшкод объекта |
| GetType | Получение информации о типе объекта |
| ToString | Приведение типа к строковому значению |

Эти методы неявно наследуются от супербазового класса object и доступны всем объекта по умолчанию. Их так же можно переопределить, если это требует логика.

### Конструкторы

Конструкторы — это специальные методы, которые используются при создании объекта. Они служат для того, чтобы подготовить объект с теми данными, без которых работа этого объекта будет невозможна. Отдельно отметим, что конструкторов может быть много, это нормально, особенно если класс сложный и используется в различной логике. Обратите внимание, что конструкторы не могут быть абстрактными или виртуальными, и не могут ничего возвращать.

|  |
| --- |
| using System; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  BaseClassFirst baseClass = new ChildClassFirst("Василий", 2);    Console.ReadLine();  }  }   */// <summary>*  */// Базовый абстрактный класс*  */// </summary>*  public abstract class BaseClassFirst  {  */// <summary>*  */// Базовый конструктор может быть protected*  */// Используя ключевое слово this можно указать*  */// какой конструктор в этом же классе вызвать вначале*  */// </summary>*  */// <param name="name"></param>*  protected BaseClassFirst(string name) : this(name, 0)  {  System.Console.WriteLine("Вызовется после конструктора");  System.Console.WriteLine("BaseClassFirst(string name, int age)");  }   protected BaseClassFirst(string name, int age)  {  System.Console.WriteLine("Вызовется после конструктора");  }  }    */// <summary>*  */// Наследник*  */// </summary>*  public class ChildClassFirst : BaseClassFirst  {  */// <summary>*  */// Конструктор наследника используя ключевое слово*  */// base укажет какой конструктор вызвать сначала*  */// в базовом классе.*  */// </summary>*  */// <param name="name"></param>*  public ChildClassFirst(string name) : base(name)  {  System.Console.WriteLine("Вызовется после конструктора");  System.Console.WriteLine("BaseClassFirst(string name)");  }   public ChildClassFirst(string name, int age) : base(name, age)  {  System.Console.WriteLine("Вызовется после конструктора");  System.Console.WriteLine("BaseClassFirst(string name, int age)");  }   */// <summary>*  */// Конструктор наследника, который вызывает базовый конструктор*  */// </summary>*  */// <param name="age"></param>*  public ChildClassFirst(int age) : base(string.Empty, 2)  {  System.Console.WriteLine("Вызовется после конструктора");  System.Console.WriteLine("BaseClassFirst(string name, int age)");  }  } } |

Главный принцип создания конструкторов в наследниках — это то, что они должны вызывать конструкторы базового класса.

В итоге формируется некая цепочка вызовов конструкторов, и цепочка строится посредством ключевых слов base или this. Первое ищет конструктор в базовом классе, второе – в локальном классе (наследнике).

## Синтаксический сахар в C# на примере foreach и yield

В языке C# достаточно много ключевых слов и операторов. Некоторые ключевые слова по факту являются синтаксическим сахаром. Что такое синтаксический сахар? Это удобный синтаксис, который скрывает под собой менее удобный и размашистый код. Один из ярких примеров синтаксического сахара является оператор foreach. Ранее уже говорили о том, как можно пробежаться по элементам массива оператор for. Соответственно оператор foreach это еще один из способов перечисления элементов массива, но с подводными камнями.

|  |
| --- |
| using System; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  int[] numbers = new[] {1, 2, 3, 4, 5};   foreach (int number in numbers)  {  System.Console.WriteLine(number);  }   Console.ReadLine();  }  } } |

Как можно заметить, он лаконичен и прост. В отличии от оператора for, foreach не требует определить количество итераций, а проходит по очереди элемент за элементом. И одно из главных отличий этих двух операторов в том, что в for коллекции можно менять, а в foreach уже нет.

Еще одна особенность заключается в том, что при компиляции оператор foreach трансформируется приблизительно вот в такой код:

|  |
| --- |
| using System; using System.Collections; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  int[] numbers = new[] {1, 2, 3, 4, 5};   IEnumerator enumerator = numbers.GetEnumerator();   while (enumerator.MoveNext())  {  int i = (int) enumerator.Current;   System.Console.WriteLine(i);  }   Console.ReadLine();  }  } } |

От разработчика скрыто, то, что по факту вызывается метод у коллекции по получению энумератора и, пока энумератор может двигаться дальше, он будет выполнять то, что было в коде за оператором foreach. В нашем случае это вывод числа на консоль.

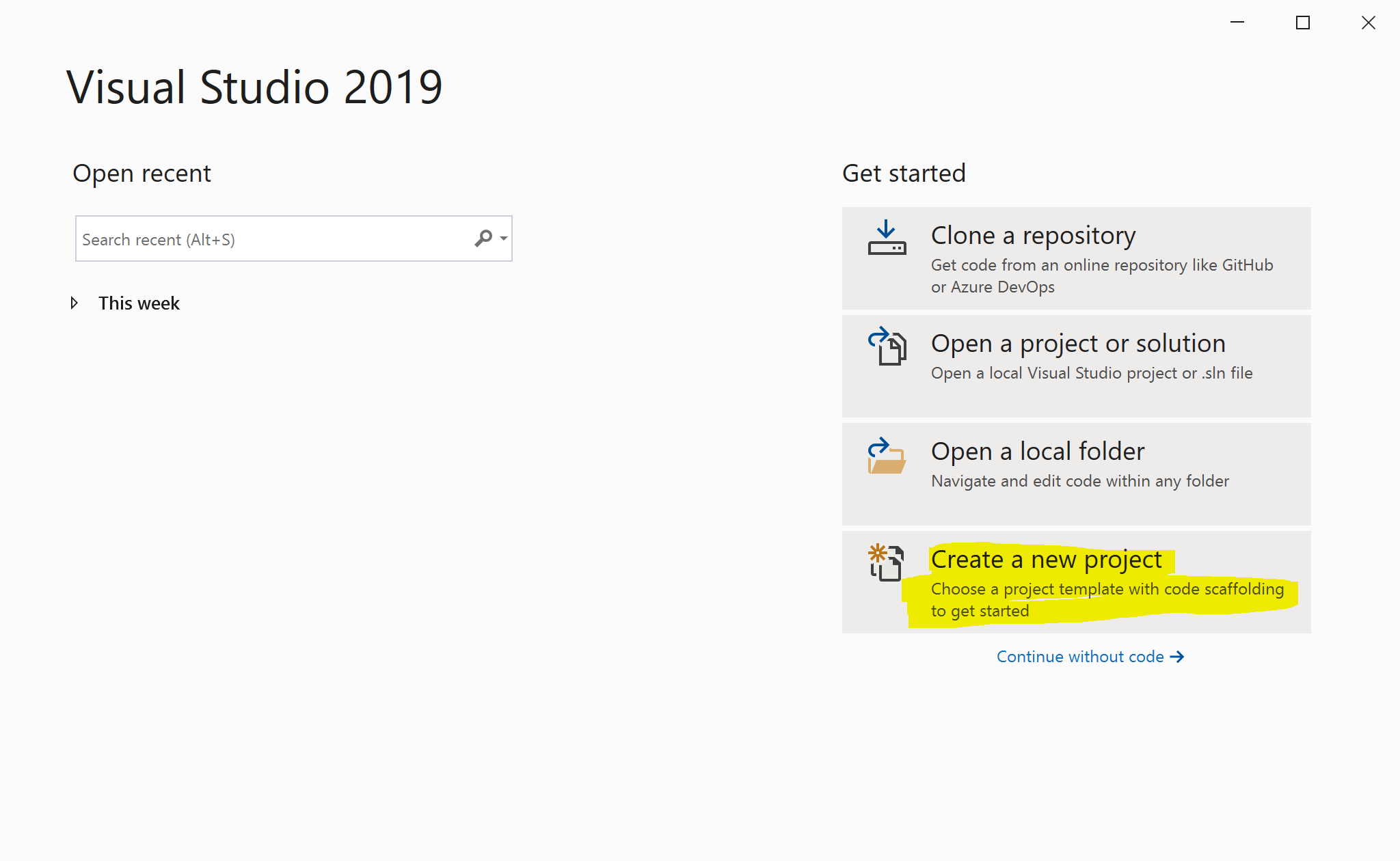
На собеседованиях довольно часто задают вопрос: что нужно сделать, чтобы можно было применить оператор foreach к вашему классу. И многие отвечают не до конца правильно, говоря то, что нужно наследоваться от интерфейса IEnumerable. Достаточно просто определить метод GetEnumerator и ваш класс готов использоваться в операторе foreach.

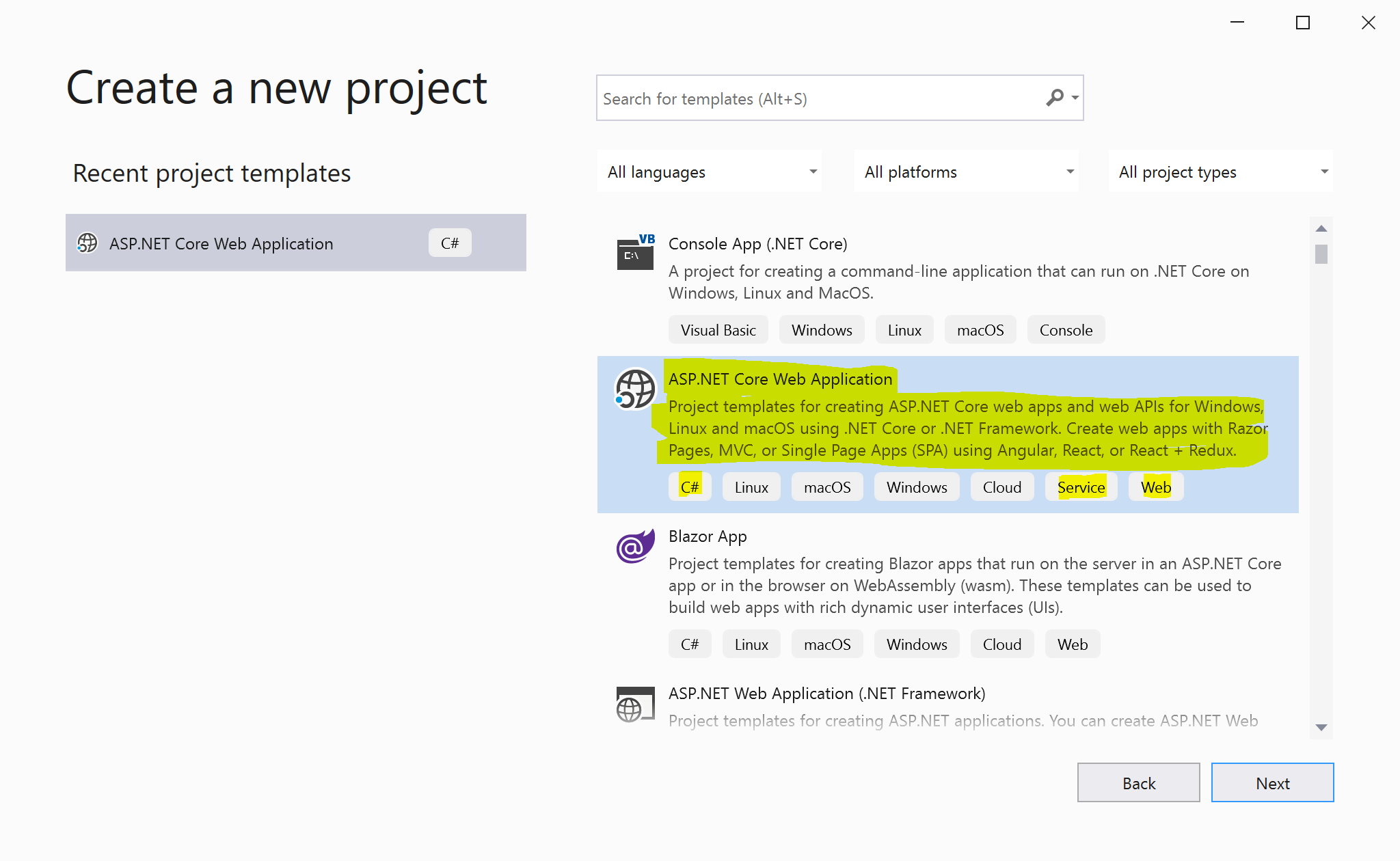
|  |
| --- |
| using System; using System.Collections; using System.Collections.Generic;  namespace Learning.Lesson.One {  class Program  {  static void Main(string[] args)  {  DemoEnumerator demoEnumerator = new DemoEnumerator();   foreach (int i in demoEnumerator)  {  System.Console.WriteLine(i);  }   *//Вывод в консоле будет таков*  *//*  *//1*  *//2*  *//3*   Console.ReadLine();  }  }   public class DemoEnumerator  {  public IEnumerator GetEnumerator()  {  *//Оператор yield вернет на время управление*  *//вызывашему методу и передаст значения 1, потом 2 и т.д.*  yield return 1;   yield return 2;   yield return 3;  }  } } |

# 

# Создаем проект в Visual Studio

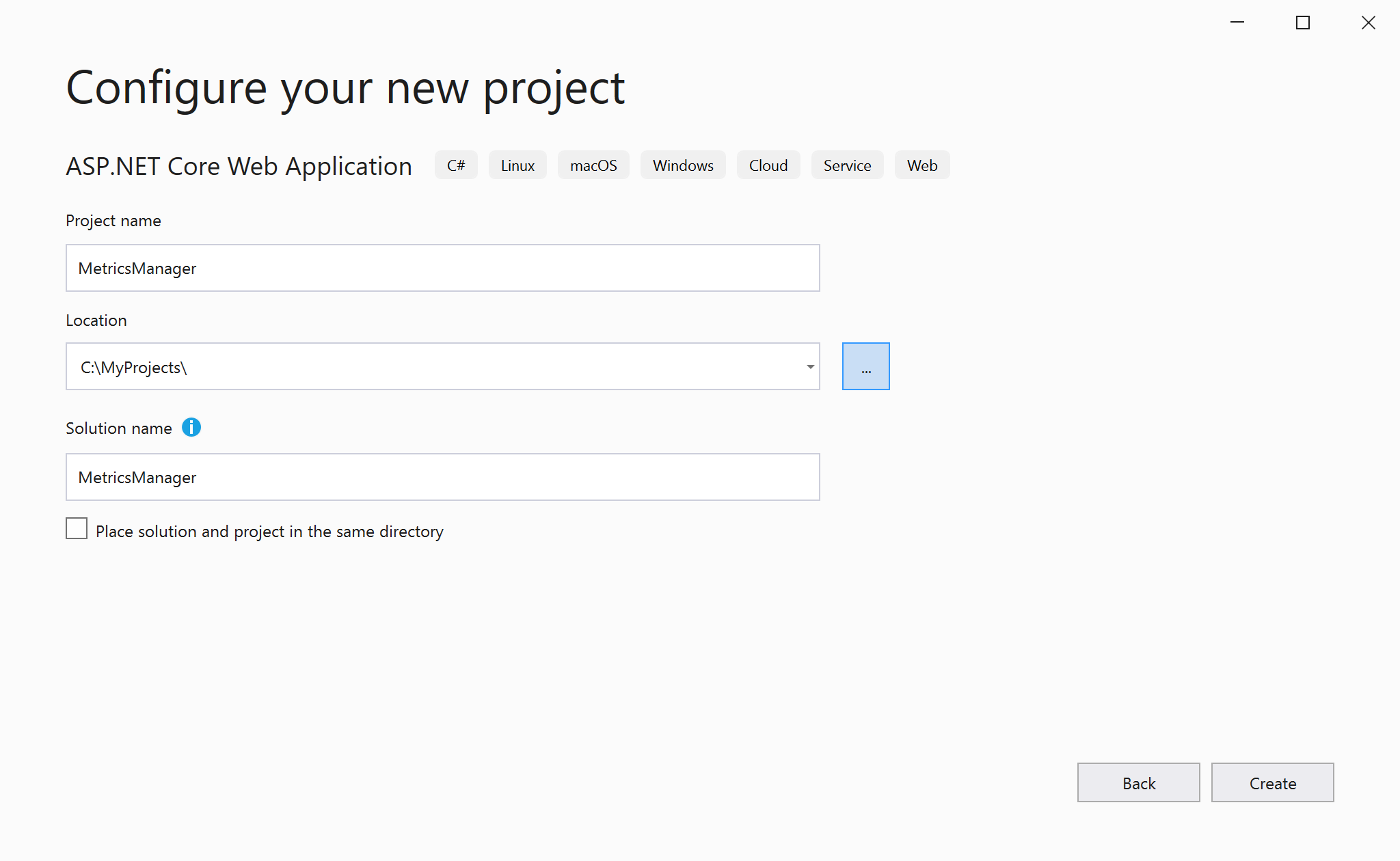
Давайте попробуем создать проект нашего первого микросервиса на ASP.NET Core. Для этого воспользуемся Visual Studio и создадим проект. Для того, чтобы у вас имелась возможность создания проекта web-api приложения, у вас должен быть установлен соответствующий инструментарий. Если вы не будете наблюдать у себя в Visual Studio возможности создания веб-проектов, вам необходимо открыть Visual Studio Installer и установить компоненты для веб-разработки. Мы не будем детально рассматривать процесс установки шаблонов, так как наверняка вы справитесь с установкой инструментов разработчика самостоятельно. а сразу попробуем создать микросервис и разобраться с его структурой.

В открывшемся окне Visual Studio нажмите пункт Create a new Project

В появившемся окне выберете шаблон ASP.NET Core Web Application

Данный шаблон позволяет создать микросервис со всеми необходимыми настройками для первого запуска. После Visual Studio попросит выбрать конкретную реализацию выбранного шаблона. Выберете API. Этот шаблон позволяет создать микросервис, который умеет отвечать на простые запросы и обладает всеми настройками для его запуска. Совсем пустой шаблон мы пока что использовать не будем, так как нам необходимо предварительно познакомиться со структурой создаваемого решения, узнать основные компоненты, и уже после этого вы сможете создавать любой микросервис прямо из пустого проекта.

Остались последние шаги: заполните название проекта, выберете его расположение на диске. После этого в выбранном каталоге на диске будет создано несколько файлов с кодом и файлов конфигурации проекта. Их структуру мы рассмотрим подробно.



# Структура созданного проекта

## Паттерны разработки, что это и как использовать

Слово паттерн переводится с английского языка, как шаблон либо узор, который нередко повторяется. В разработке под паттерном подразумевается общепринятые удобные шаблоны кода для решения многих задач. Они элегантны и удобны для модификации и понятны для других разработчиков, которые будут пытаться разобраться в вашем коде. Паттерны применяются очень часто и много раз в одном и том же проекте. Главное правило использования паттернов – не писать код только ради паттерна. Это довольно тонкая грань, которую по началу можно не заметить. В большинстве случаях опытные разработчики при написании кода уже могут предполагать, какие паттерны они будут применять исходя из их опыта, а новички начинают применять, даже не подумав нужен ли этот паттерн или нет.

Паттерны делятся на несколько видов – порождающие, структурные и поведенческие. Порождающие паттерны используются для удобного и безопасного создания, либо даже графов объектов. Структурные используются для формирования удобной иерархии. А поведенческие используются для безопасного и удобного взаимодействия между объектами.

По ходу учебы и работы вы изучите большинство из них и сможете применять их на практике.

Подводя итог можно сказать, что паттерны — это проверенное решение, стандартизация кода и общепринятый подход.

## Паттерн “Цепочка обязанностей”

Прежде чем приступать к обзору созданных файлов, необходимо ознакомиться с концепцией, согласно которой выстроен код в приложении asp.net core. Для начала введем концепцию сообщения и обработчика сообщения. Сообщением будем называть некоторый объем информации, передаваемый атомарно от посылающего к принимающему, а обработчиком сообщения назовем сущность, которая будет обрабатывать полученное сообщение и выдавать на него ответ.

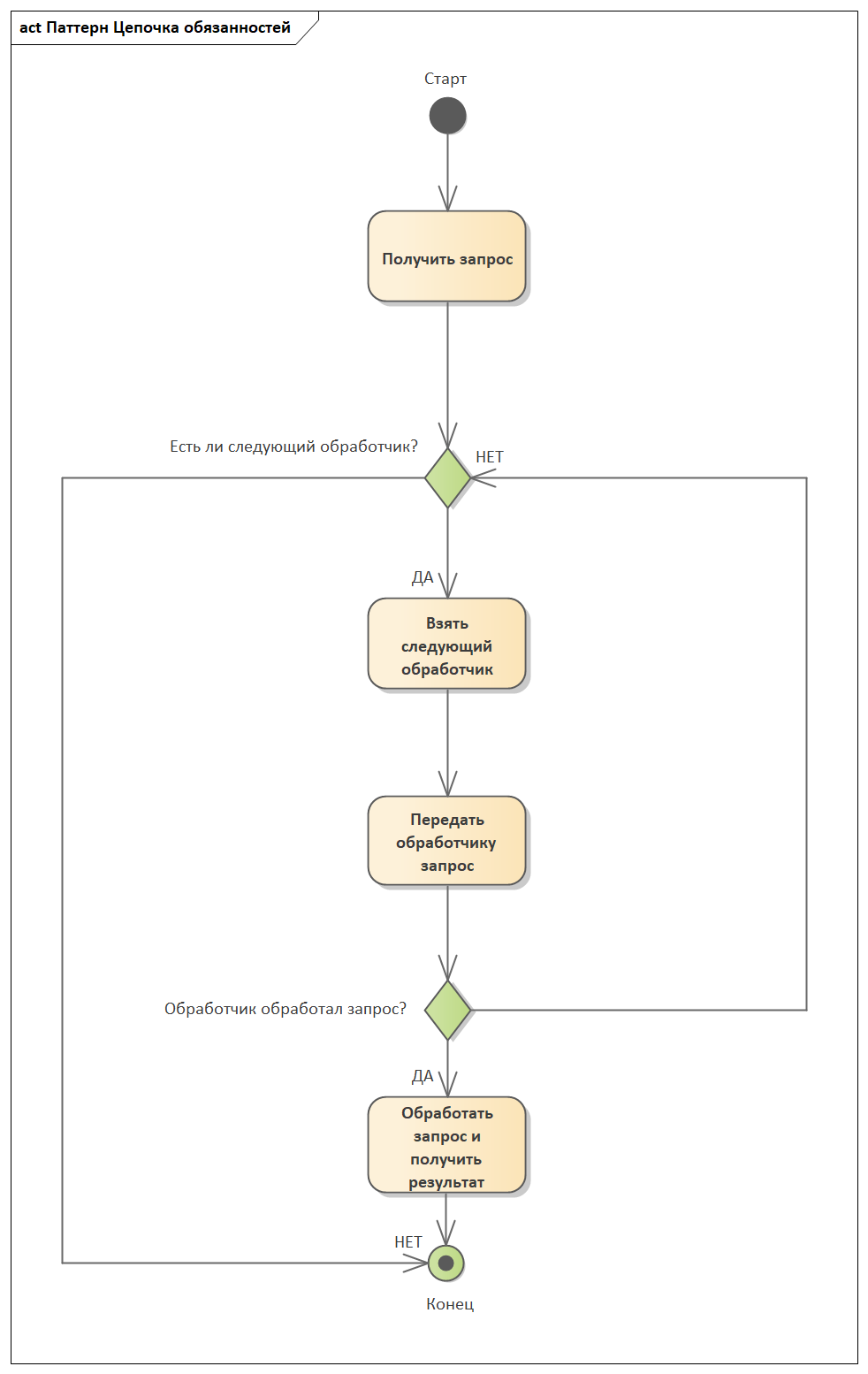
Теперь представьте, что обработчику сообщения необходимо сделать с сообщением ряд манипуляций, прежде чем выдать ответ. Например, представьте документ, который проходит ряд подписаний у различных проверяющих прежде, чем будет отдан тому, кто попросил проверить документ. Таким образом, мы получаем аналогию поведенческому паттерну программирования под названием “цепочка обязанностей”.

Выделяя некоторые атомарные “обязанности” можно объединить их в “цепочку”, чтобы исходное сообщение прошло все обязательные процедуры, прежде, чем будет выдан ответ.

В совсем упрощенном виде можно представить этот паттерн в виде людей, передающих ведерко с водой, где каждое ведерко будет сообщением, а каждый человек, отдельным обработчиком сообщения. Люди тут объединены в “цепочку обязанностей”.

Данный паттерн является поведенческим. Его использует для того, чтобы формировать цепочку обработчиков-перехватчиков пока один из них не перехватит запрос.

Алгоритм его работы довольно прост. Рассмотрим его на рисунке.



Резюмируя, запомним, что в объектно-ориентированном программировании имеется шаблон проектирования системы, предназначенный для организации в системе уровней ответственности.

Внимательному студенту может показаться странным необходимость объяснения принципа работы данного шаблона проектирования, но понимание этого концепта позволяет с легкостью представить структуру кода проекта asp.net core. Весь код в приложении будет уложен слоями, и каждый отдельный слой будет отвечать за конкретное действие с получаемым от пользователя сообщением, чтобы в итоге выдать ответ пользователю на его запрос. Давайте рассмотрим эти слои в только что созданном микросервисе подробнее.

## Program.cs

|  |
| --- |
| using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.Configuration; using Microsoft.Extensions.Hosting; using Microsoft.Extensions.Logging; using System; using System.Collections.Generic; using System.Linq; using System.Threading.Tasks;  namespace MetricsManager {  public class Program  {  public static void Main(string[] args)  {  CreateHostBuilder(args).Build().Run();  }   public static IHostBuilder CreateHostBuilder(string[] args) =>  Host.CreateDefaultBuilder(args)  .ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  });  } } |

Как вам уже известно, в языке C# должен быть статический метод Main. Именно в этом методе мы видим вызов автоматически сгенерированного метода CreateHostBuilder. Внутри метода CreateHostBuilder происходит следующее:

* Создается сборщик хоста
* Сборщик хоста конфигурируется параметрами по умолчанию
* В делегате указывается, чтобы наш сборщик использовал класс Startup для задания параметров создаваемого микросервиса

Затем полученный сборщик собирается при помощи Build() и запускает полученный хост при помощи метода Run().

Хост инкапсулирует в себе http сервер, middlware компоненты (те самые слои из цепочки обязанностей), работу с конфигурацией и окружениями.

## Startup.cs

В работе ASP.NET Core приложений написание кода построено на соглашениях. То есть программисты открыто договорились между собой, что в ASP.NET Core, например,обязательно будет класс под названием Startup с обязательным методом Configure, который настраивает pipeline обработки запроса (цепочку обязанностей). В классе Startup может присутствовать необязательный метод ConfigureServices, который настраивает сервисы внутри создаваемого микросервиса. Сервис – это используемый компонент внутри приложения, который предоставляет некоторую функциональность.

|  |
| --- |
| using Microsoft.AspNetCore.Builder; using Microsoft.AspNetCore.Hosting; using Microsoft.AspNetCore.HttpsPolicy; using Microsoft.AspNetCore.Mvc; using Microsoft.Extensions.Configuration; using Microsoft.Extensions.DependencyInjection; using Microsoft.Extensions.Hosting; using Microsoft.Extensions.Logging; using System; using System.Collections.Generic; using System.Linq; using System.Threading.Tasks;  namespace MetricsManager {  public class Startup  {  public Startup(IConfiguration configuration)  {  Configuration = configuration;  }   public IConfiguration Configuration { get; }   *// This method gets called by the runtime. Use this method to add services to the container.*  public void ConfigureServices(IServiceCollection services)  {  services.AddControllers();  }   *// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.*  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }   app.UseHttpsRedirection();   app.UseRouting();   app.UseAuthorization();   app.UseEndpoints(endpoints =>  {  endpoints.MapControllers();  });  }  } } |

В нашем автоматически-сгенерированном файле в методе ConfigureServices добавляется указание конфигурации сервисов искать контроллеры. Контроллеры – это часть кода,, которая ответственна за принятие запроса от пользователя и передачу его в другие области приложения в виде модели.

В методе Configure происходит добавление middleware, слоев, которые формируют pipeline (цепочку обязанностей) обработки запроса. Рассмотрим подробнее, что там происходит:

* Проверяется окружение и, если оно является “разработческим”, то добавляется специальная страница для красивого оформления ошибок работы микросервиса. Конечному пользователю такое видеть не надо, потому и выставлена данная проверка на окружение
* Добавляется слой перенаправления запросов с протокола http на https
* Добавляется слой маршрутизации запросов внутри приложения
* Добавляется слой авторизации запросов
* Добавляется слой, в котором указывается, откуда брать обработчики для запросов. В нашем случае указано, что обработчики для запросов необходимо взять из контроллеров, которые найдутся в созданном приложении

Стоит помнить, что запрос, который приходит в приложение, как бы проходит все вышеуказанные слои именно в порядке их добавления. Таким образом, в будущих уроках мы увидим, что в созданный конвейер обработки запросов можно запросто вставить свой слой обработки для реализации требуемой бизнес-задачи.

## WeatherForecast.cs

В качестве примера, микросервис, созданный по шаблону, несет в себе модель прогноза погоды, с которой работает автоматически созданный контроллер. Такой простой класс еще называют DTO или data transfer object, то есть класс, который необходим для переноса данных из одного слоя приложения в другой. Пока просто взглянем на класс и увидим, что у него есть температура в цельсиях, вычисляемое “на лету” поле температуры в фаренгейтах, дата измерения и пояснительная строчка.

|  |
| --- |
| using System;  namespace MetricsManager {  public class WeatherForecast  {  public DateTime Date { get; set; }   public int TemperatureC { get; set; }   public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);   public string Summary { get; set; }  } } |

## WeatherForecastController.cs

В созданном по шаблону контроллере содержится метод, внутри которого задекларировано, что будет отдано пользователю в качестве ответа на его запрос при обращении к этому контроллеру

|  |
| --- |
| using Microsoft.AspNetCore.Mvc; using Microsoft.Extensions.Logging; using System; using System.Collections.Generic; using System.Linq; using System.Threading.Tasks;  namespace MetricsManager.Controllers {  [ApiController]  [Route("[controller]")]  public class WeatherForecastController : ControllerBase  {  private static readonly string[] Summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };   private readonly ILogger<WeatherForecastController> \_logger;   public WeatherForecastController(ILogger<WeatherForecastController> logger)  {  \_logger = logger;  }   [HttpGet]  public IEnumerable<WeatherForecast> Get()  {  var rng = new Random();  return Enumerable.Range(1, 5).Select(index => new WeatherForecast  {  Date = DateTime.Now.AddDays(index),  TemperatureC = rng.Next(-20, 55),  Summary = Summaries[rng.Next(Summaries.Length)]  })  .ToArray();  }  } } |

Пока только обратим внимание на следующие особенности:

* Класс контроллера унаследован от абстрактного ControllerBase, внутри которого задекларированы различные виртуальные (доступные к переопределению в наследнике) методы и свойства полученного контроллером запроса
* На классе контроллера указан атрибут ApiController. Декорирование класса этим атрибутом является пометкой для слоя, который собирает контроллеры в приложении, чтобы объединить их в слой (в Startup классе мы видели указание искать контроллеры).
* На классе контроллера указан атрибут Route со специфицированным строковым литералом, указывающим, по какому пути в запросе к микросервису можно обратиться к данному контроллеру
* В конструктор класса передан ILogger. На место этого логгера наше окружение может подставить реализацию логгера, который будет выполнять действия по логгированию действий, происходящих внутри контроллера

Прежде чем двигаться дальше, для лучшего понимания работы созданного по шаблону микросервиса, следует изучить основы протокола HTTP. Прежде чем мы перейдем к его рассмотрению, посмотрим на еще два файла, которые присутствуют в приложении.

## Файлы конфигурации

#### launchSettings.json

В данном файле содержатся настройки для запуска приложения: на каком порту какого http сервера его запустить, выставляются переменные окружения и т.д. Работу с данным файлом мы рассмотрим подробнее в рамках курса, однако сейчас стоит узнать, что именно в этом файле содержатся все настройки того, как запускать ваш микросервис.

#### appsettings.json

Часто в работе приложения необходимо использовать различные настройки, которые можно поменять без необходимости пересборки самого приложения. В такие настройки, например, можно сложить настройки подключения к базе данных, адреса сервисов, к которым можно обращаться и многое другое. Именно такие настройки и рекомендуется складывать в данный файл. Работу с этим файлом конфигурации, а также возможности его спецификации под конкретное окружение мы рассмотрим в продолжении курса.

# Протокол HTTP

Созданный нами микросервис предоставляет API для работы с ним по протоколу http. То есть наш микросервис как бы говорит “я умею отдавать прогноз погоды который у меня спросят (и все что еще вы научите меня делать в своих контроллерах), и делаю это на языке http”. Рассмотрим этот протокол детальнее.

## Основные понятия

Итак, протокол HTTP это протокол прикладного уровня для передачи произвольных данных. Протокол разработан в парадигме наличия клиента и сервера. То есть потребителей (клиентов), которые инициируют соединение, затем отправляют запросы и серверов, которые обрабатывают полученные запросы и отправляют ответы. Таким образом весь протокол работает в стиле “запрос-ответ”.

Например, веб-браузер может являться клиентом, а созданный вами микросервис – сервером. Таким образом веб-браузер может запросить данные у вашего сервера и получить какой-то ответ.

Клиент и сервер обмениваются произвольными данными, но они имеют строго определенный формат, который мы сейчас рассмотрим.

## Структура HTTP сообщения

Каждое сообщение по протоколу HTTP имеет строгий формат и состоит из следующих частей

1. Стартовая строка
2. Заголовки
3. Тело сообщения

Тело сообщения может не присутствовать в сообщении, однако стартова строка и заголовки являются обязательными частями http-сообщения

Стартовая строка выглядит следующим образом:

|  |
| --- |
| Метод URI HTTP/1.1 |

Методом в данном случае является тип запроса, например, GET. Чуть позже мы рассмотрим все HTTP методы из протокола и поймем зачем они нужны. URI -- уникальный идентификатор сетевого ресурса и определяет путь к запрашиваемому документу, например geekbrains.ru. В конце идет версия протокола, для того, чтобы сервер понимал, в каком формате его попросили ответить на сообщение.

В ответе от сервера стартовая строка имеет следующий формат

|  |
| --- |
| HTTP/1.1 КодСостояния Пояснение |

Версия протокола точно такая же как и в запросе, КодСостояния – кодовое указание для клиента о том, что делать с ответом (например, был ли запрос успешным), а пояснение является текстовым описанием к коду состояния.

## Методы HTTP сообщений

Метод в протоколе HTTP это слово, обозначающее то, какая операция будет производиться над ресурсами сервера. Например метод GET обозначает, что клиент запросит у сервера какую-то информацию. Рассмотрим все методы в протоколе HTTP

* GET – используется для запроса содержимого с сервера по указанному адресу. В спецификации протокола HTTP также указано, что выполненный метод GET не должен приводить к изменению состояния сервера (то есть если запрашивать какие-то данные при помощи этого запроса, этим запросом они изменены быть не могут)
* POST – применяется для передачи данных на сервер. Например, почти любая форма загрузки картинок на сайтах работает по этому методу.
* PUT – аналогичен POST, за исключением того, что в протоколе рекомендуется использовать этот метод для загрузки данных в существующий URI. Представьте, что у вас есть книжная полка и вы ставите в неё новую книгу. Логично было бы использовать для такого сообщения метод PUT, так как вы ставите книгу в уже созданную полку с книгами.
* DELETE – указание серверу удалить ресурс, находящийся по указанному адресу
* PATCH – изменение части ресурса по указанному адресу. Например, при помощи такого метода можно сделать редактирование комментариев к странице в интернете.
* HEAD – аналогичен GET, только тело в ответе сервера будет отсутствовать тело. Часто применяется для того, чтобы узнать, не изменились ли данные на сервере
* OPTIONS – используется для определения того, что поддерживает сервер. Клиент при помощи данного метода может выяснить, какие методы поддерживает сервер, а также, какие заголовки http сообщения принимает.
* TRACE – отдает эхо-ответ клиенту для того, чтобы клиент мог понять, произвел ли какой-нибудь промежуточный сервер стоящий на пути до интересующего нас ресурса какие-либо изменения с нашим запросом.
* CONNECT – используется для создания защищенного соединения

Самыми распространенными методами по используемости являются методы GET и POST. В небольшом практическом задании, мы используем методы GET, POST, PATCH и DELETE.

## Заголовки

В сообщении http заголовки – это строки, которые разделены двоеточием и образующие пару ключ-значение. В заголовках определяется любая дополнительная информация для сервера или клиента (заголовки могут присутствовать как в запросе так и в ответе). В заголовках, например, клиент может сообщить серверу, что хочет получить сообщение в определенном формате или же что в теле запроса будет содержаться контент в определенном формате. Существует большое количество стандартизированных заголовков, а также можно использовать свои произвольные заголовки, когда вы разрабатываете API для ваших сервисов для решения различных бизнес-задач.

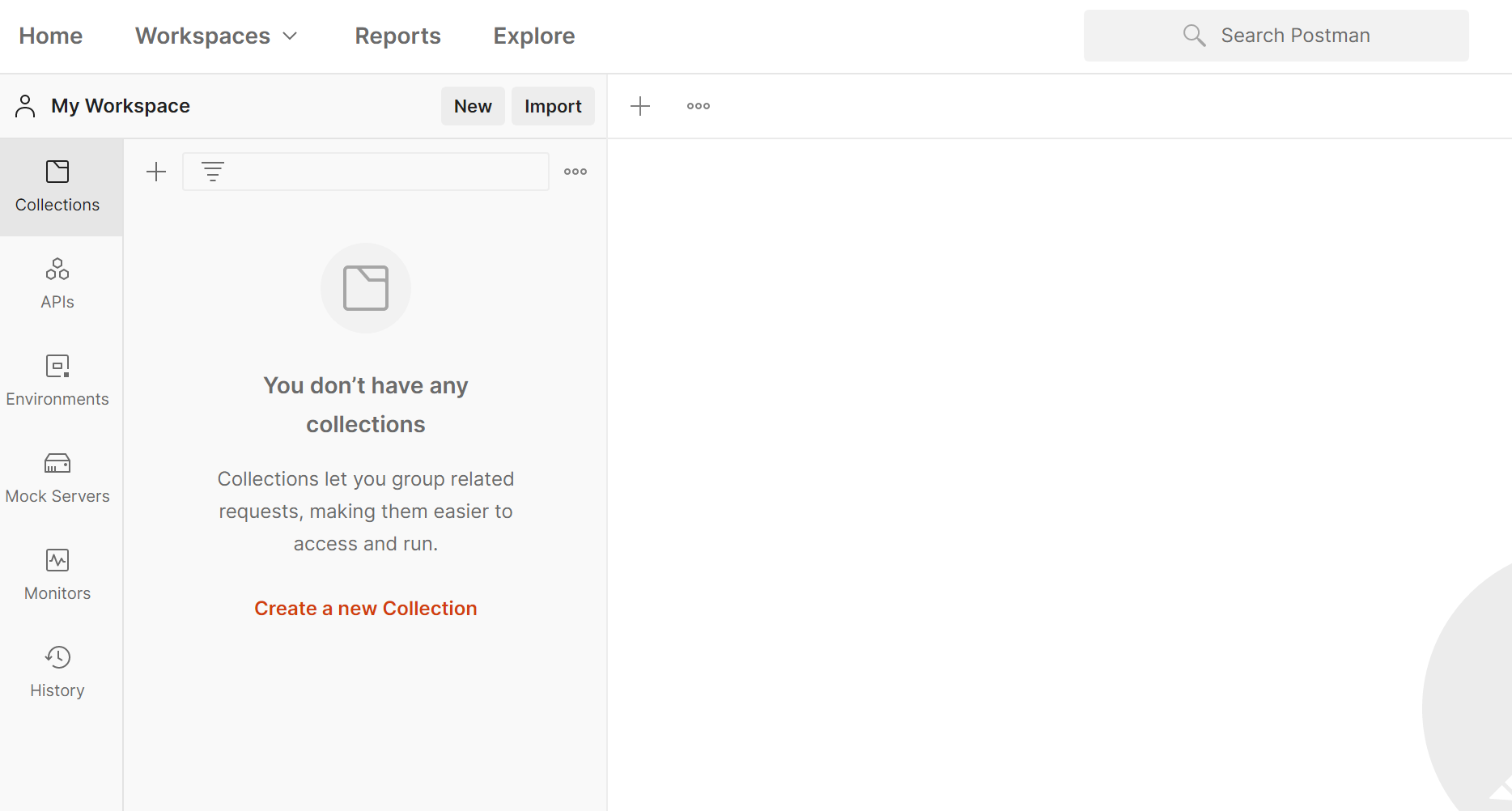
## Тело сообщения

В тело сообщения в протоколе http принято записывать все данные, которые нужно передать между клиентом и сервером. Например, если мы делаем чат на сайте, то сообщения чата будут передаваться в теле сообщения.

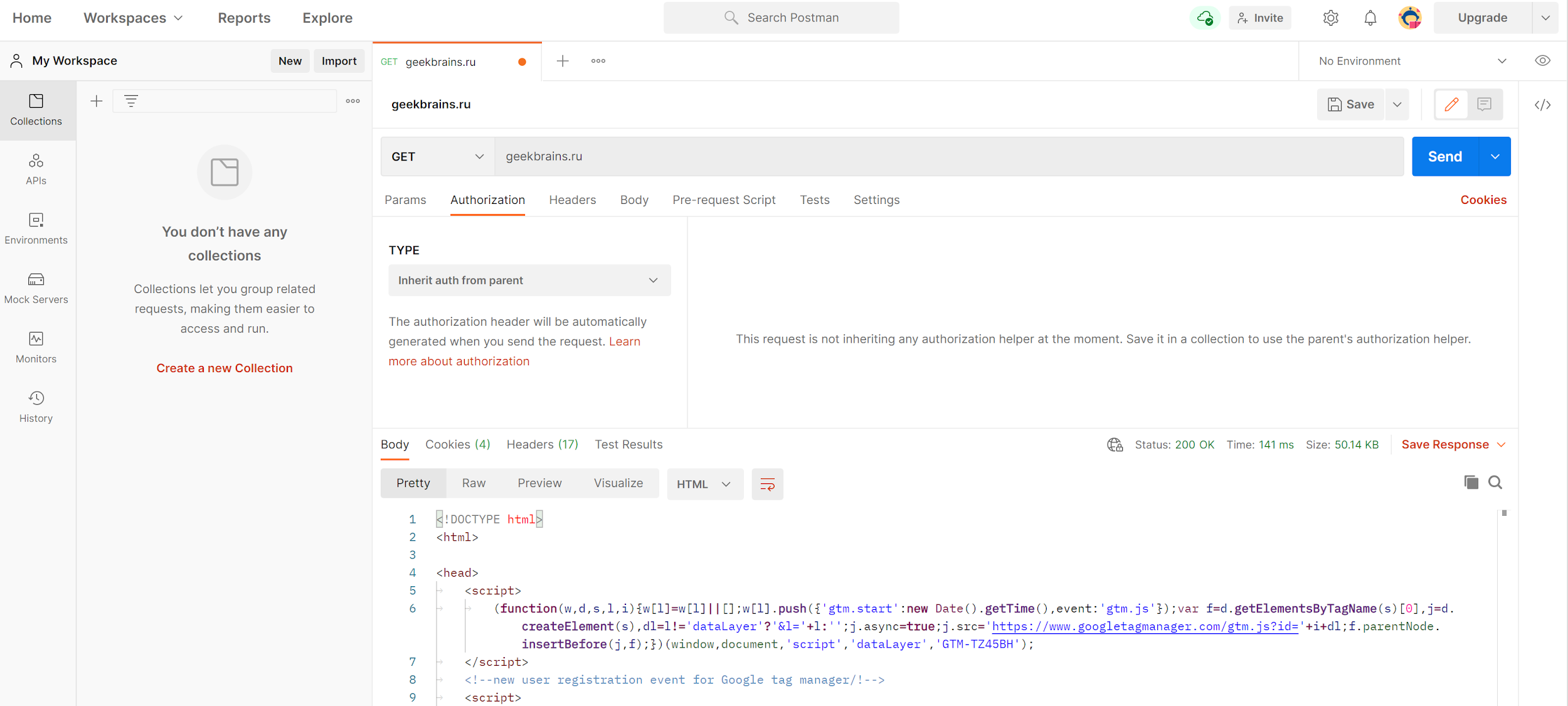
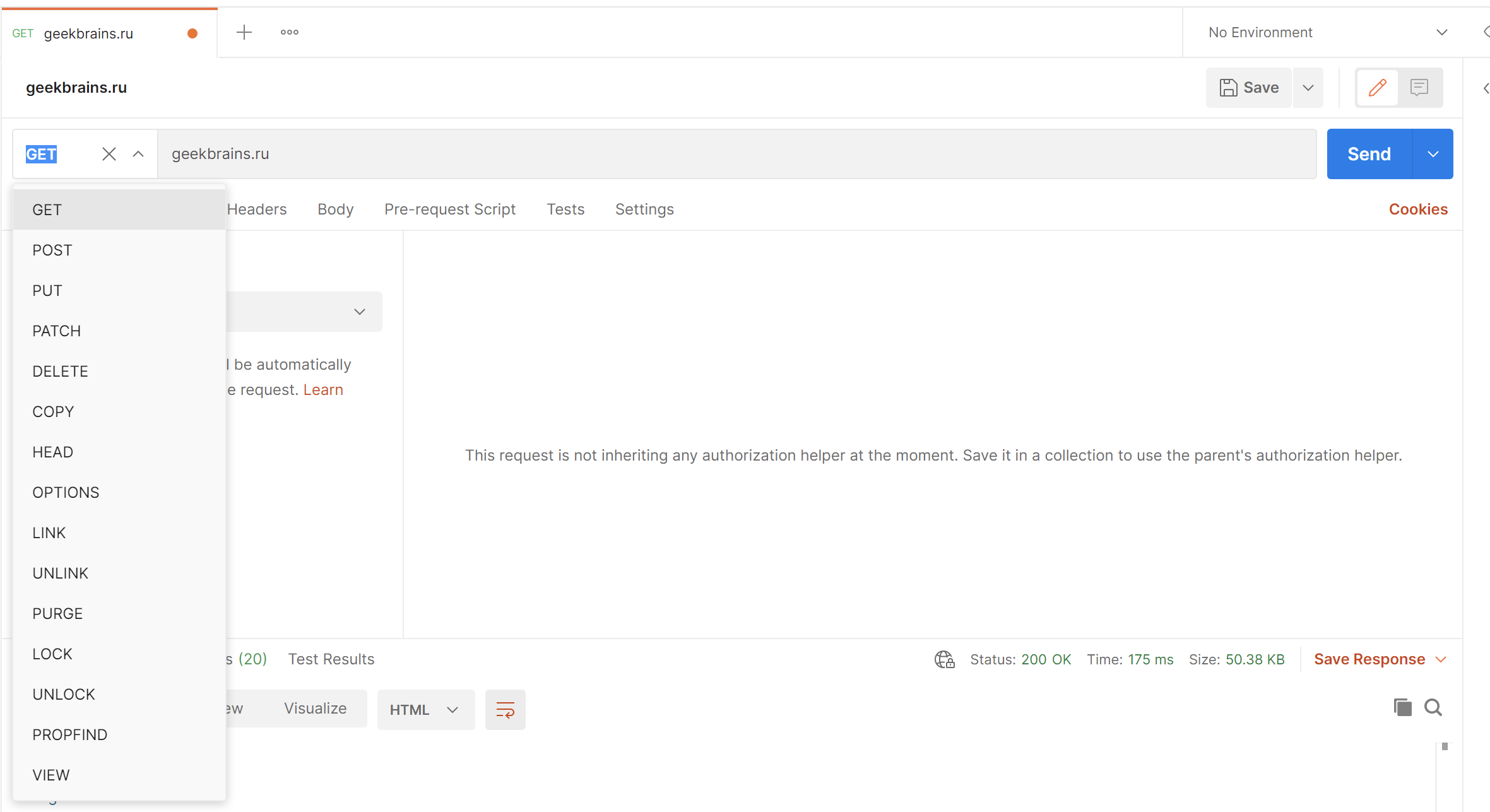
# Выполнение HTTP запросов при помощи Postman

Давайте наглядно посмотрим на то, как можно выполнять запросы по протоколу HTTP. Часто тестировщики программного обеспечения, которые работают с серверными API, отвечающими на запросы по протоколу HTTP используют “консоли” или конструкторы запросов, так как они удобны и позволяют визуально создавать запросы.

Таких консолей существует много. Есть отдельные программы, а также расширения для браузеров, которые можно легко найти по запросу “HTTP console” в удобном вам поисковике. Дальнейшие примеры будут показаны на примере консоли Postman.

Пожалуйста откройте Postman и нажмите на создание нового запроса (крестик справа от кнопки Import)

В появившемся окне выберем метод запроса GET и введем URI для запроса geekbrains.ru и нажмем кнопку SEND.

Теперь мы можем посмотреть, что в ответ на наш GET запрос вернул сервер.

Мы видим, что в теле ответа от сервера содержится HTML страничка сайта, а если переключиться на вкладку Headers (заголовки) – можно увидеть список заголовков, которые отдал нам сервер, на котором размещен сайт.

В дальнейшем мы еще не раз будем пользоваться http консолью в рамках нашего курса. Почти все они имеют схожий пользовательский интерфейс. Студентам предлагается поэкспериментировать с консолью и отправлять различные запросы к различным ресурсам, чтобы лучше понять структуру http сообщений.

# Операции CRUD

Очень часто в разработке API микросервисов встречается необходимость реализации однотипных операций над ресурсом, которые позволяют создавать, редактировать, читать и удалять запрошенные ресурсы. Набор таких операций принято называть CRUD-операциями.

CRUD – это акроним, обозначающий четыре базовые операции для манипуляции данными:

* Create – создание данных
* Read –  чтение данных
* Update – обновление данных
* Delete – удаление данных

## Реализация методов CRUD через методы HTTP протокола

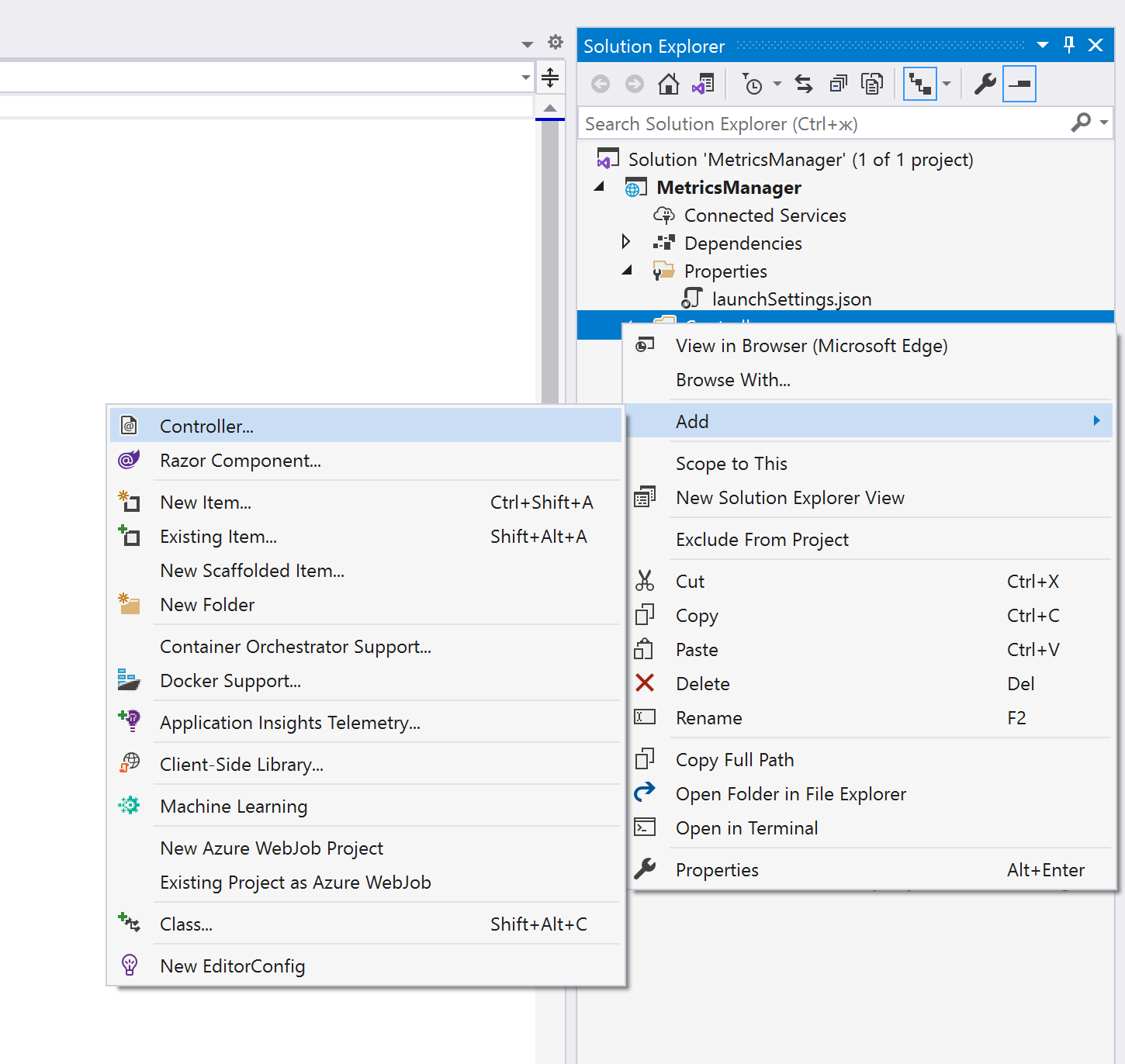
Изначально, термин CRUD возник применительно к классическим базам данных, но часто применяется в тех местах, где подобные операции нужно произвести над хранимыми данными. Так, если мы проектируем API для нашего микросервиса работающего по HTTP протоколу, мы выберем методы POST, GET, PUT, DELETE соответственно для реализации CRUD функциональности в API микросервиса.

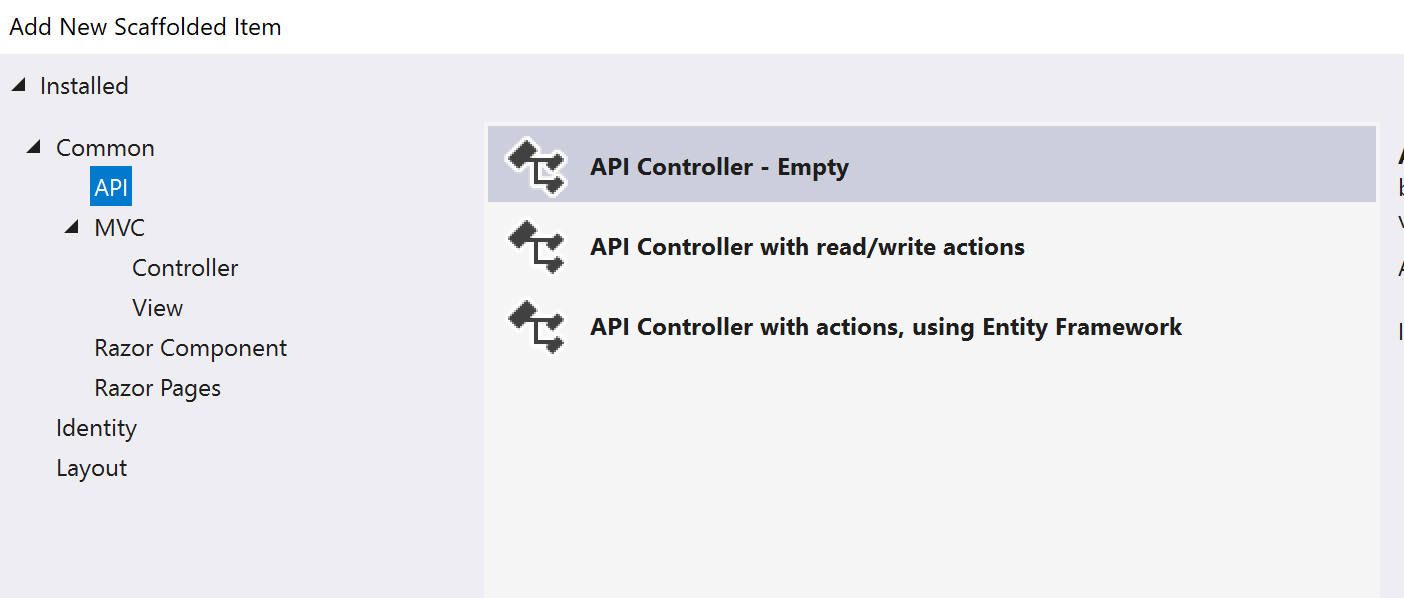
# Первый контроллер с CRUD операциями

Давайте попробуем создать в нашем микросервисе новый контроллер и научить его выполнять CRUD операции над хранящейся в контроллере статической переменной.

## Добавление нового web-api контроллера в проект

Чтобы добавить новый контроллер в наш проект, можно вручную создать новый файл и написать в него весь необходимый код или же воспользоваться встроенным в Visual Studio инструментом для автоматической генерации контроллеров. Рассмотрим последний способ. Для этого в Solution Explorer выберете папку с контроллерами и нажатием правой кнопки мыши выберите меню добавления нового контроллера как показано на рисунке.

В появившемся окне выберете Api Contorller - Empty. Этот шаблон позволит нам создать пустой контроллер для API без функциональности, которая пока что не нужна нам.

Назовите как-нибудь контроллер, например “CrudController”, в результате чего Visual Studio создаст файл со следующим содержимым:

|  |
| --- |
| using Microsoft.AspNetCore.Http; using Microsoft.AspNetCore.Mvc; using System; using System.Collections.Generic; using System.Linq; using System.Threading.Tasks;  namespace MetricsManager.Controllers {  [Route("api/[controller]")]  [ApiController]  public class CrudController : ControllerBase  {  } } |

Данный класс можно было бы создать самостоятельно, но пока можно воспользоваться способом генерации. В Visual Studio есть множество шаблонов для создания контроллеров и они зачастую облегчают рутинную работу по написанию кода контроллеров, предлагая заранее сгенерировать за вас повторяющийся код.

В сгенерированном контроллере мы видим атрибут Route, в параметрах у которого указано, что методы контроллера будут доступны на сервере по адресу “http://сервер/api/crud”. Пока просто запомним, что asp.net core вместо [controller] вычленит из имени класса название нашего контроллера и подставит в путь ресурса для сервера.

Так же вы можете заметить атрибут ApiController. Это метка для middleware, чтобы слой подхватил наш контролер и встроил его в логику работы сервера.

## Добавление crud-методов в созданный контроллер

Модифицируем сгенерированный класс следующим образом

|  |
| --- |
| using Microsoft.AspNetCore.Mvc; using System.Linq;  namespace MetricsManager.Controllers {  [Route("api/[controller]")]  [ApiController]  public class CrudController : ControllerBase  {  private readonly ValuesHolder holder;   public CrudController(ValuesHolder holder)  {  this.holder = holder;  }   [HttpPost("create")]  public IActionResult Create([FromQuery] string input)  {  holder.Values.Add(input);  return Ok();  }   [HttpGet("read")]  public IActionResult Read()  {  return Ok(holder.Values);  }   [HttpPut("update")]  public IActionResult Update([FromQuery] string stringsToUpdate, [FromQuery] string newValue)  {  for (int i = 0; i < holder.Values.Count; i++)  {  if (holder.Values[i] == stringsToUpdate)  holder.Values[i] = newValue;  }   return Ok();  }   [HttpDelete("delete")]  public IActionResult Delete([FromQuery] string stringsToDelete)  {  holder.Values = holder.Values.Where(w => w != stringsToDelete).ToList();  return Ok();  }  } } |

Для хранения состояния нашего сервера, воспользуемся классом, в котором будем хранить значения. Внимательный студент может задаться вопросом “почему бы не сделать простое приватное поле в контроллере?”. Дело в том, что конвейер обработки запроса создает объект класса контроллера на каждый приходящий запрос. Таким образом, у нас не получится сохранить состояние сервера в приватной переменной контроллера, если нам потребуется хранить состояние в неизменном виде между запросами к серверу. Зарегистрируем этот класс в DI контейнере (рассмотрим что это такое на следующих уроках) путем добавления следующей строчки кода в метод ConfigureServices класса Startup

|  |
| --- |
| services.AddSingleton<ValuesHolder>()*;* |

Теперь разберем метод Read. Он помечен атрибутом [HttpGet]. Этим атрибутом мы сообщаем нашему asp.net core, что код внутри нашего метода выполнится только в том случае, если к серверу обратятся по указанному адресу с http-методом GET. Если же вызвать этот метод при запросе к серверу с http методом POST или любым другим, то сервер скажет при помощи кода состояния с ошибкой, что такой http-метод не разрешен для выполнения нашим сервисным методом, а также подскажет, что хочет http-метод GET при помощи заголовка. Наш серверный метод возвращает IActionResult. Это встроенный в asp.net core интерфейс, который позволяет отдавать объект со статус кодом и ответом. Внутри метода мы вызываем встроенный метод Ok() который вернет реализацию интерфейса IActionResult, а именно OkResult. Когда asp.net core будет сериализовать результат работы нашего метода перед отправкой его по протоколу http клиенту, Asp.net core увидит, что вызвав метод Ok и передав ему внутрь какой-то объект, мы, тем самым, сообщаем, что хотим отдать клиенту статус код, сигнализирующий об успешном выполнении его запроса, а также данные в теле сообщения, которые переданы в параметр методу Ok().

Остальные добавленные методы реализованы аналогично, за исключением того, что они не отдают никакого ответа в теле сообщения. Так же параметры запросов к остальным методам помечены атрибутом [FromQuery]. Этим атрибутом мы подсказываем парсеру запроса, что хотим, чтобы данные для входного параметра он брал из URI запроса на сервер. Существуют и другие атрибуты для подсказок парсеру откуда брать значения, их мы рассмотрим в следующих уроках.

# Сборка, отладка и проверка работоспособности созданного микросервиса

Теперь давайте попробуем запустить только что созданный сервис с нашим crud-контроллером. Убедимся, что наш сервер работоспособен.

## Простой запуск из Visual Studio

Для того, чтобы запустить созданный микросервис из VisualStudio, достаточно нажать Debug. Visual Studio сама соберет ваш код и запустит хост вашего микросервиса внутри веб-сервера iis-express (если только вы не переопределите настройки запуска). Затем, Visual Studio откроет браузер и отправит вас на страницу по умолчанию для вашего сервиса.

Модифицируйте файл launchSettings.json и приведите его к виду как показано в сниппете ниже

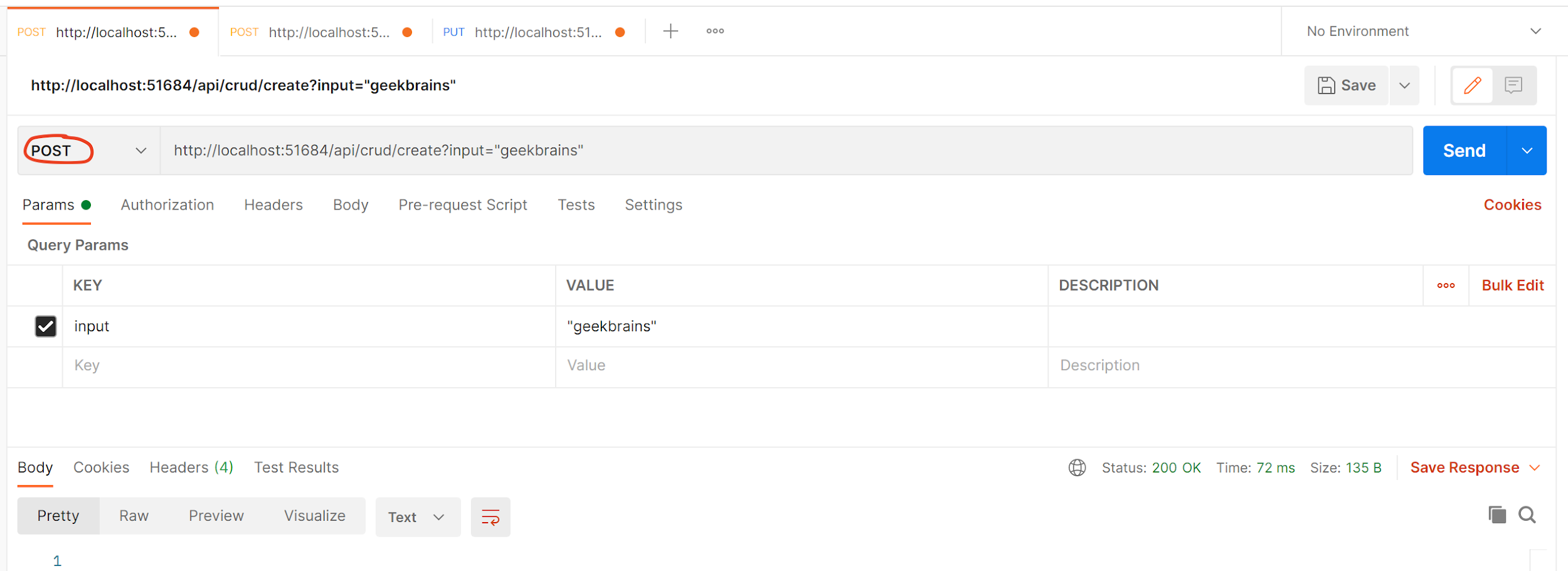
|  |
| --- |
| {  "$schema": "http://json.schemastore.org/launchsettings.json",  "iisSettings": {  "windowsAuthentication": false,  "anonymousAuthentication": true,  "iisExpress": {  "applicationUrl": "http://localhost:51684"  }  },  "profiles": {  "IIS Express": {  "commandName": "IISExpress",  "launchBrowser": true,  "launchUrl": "api/crud/read",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  } } |

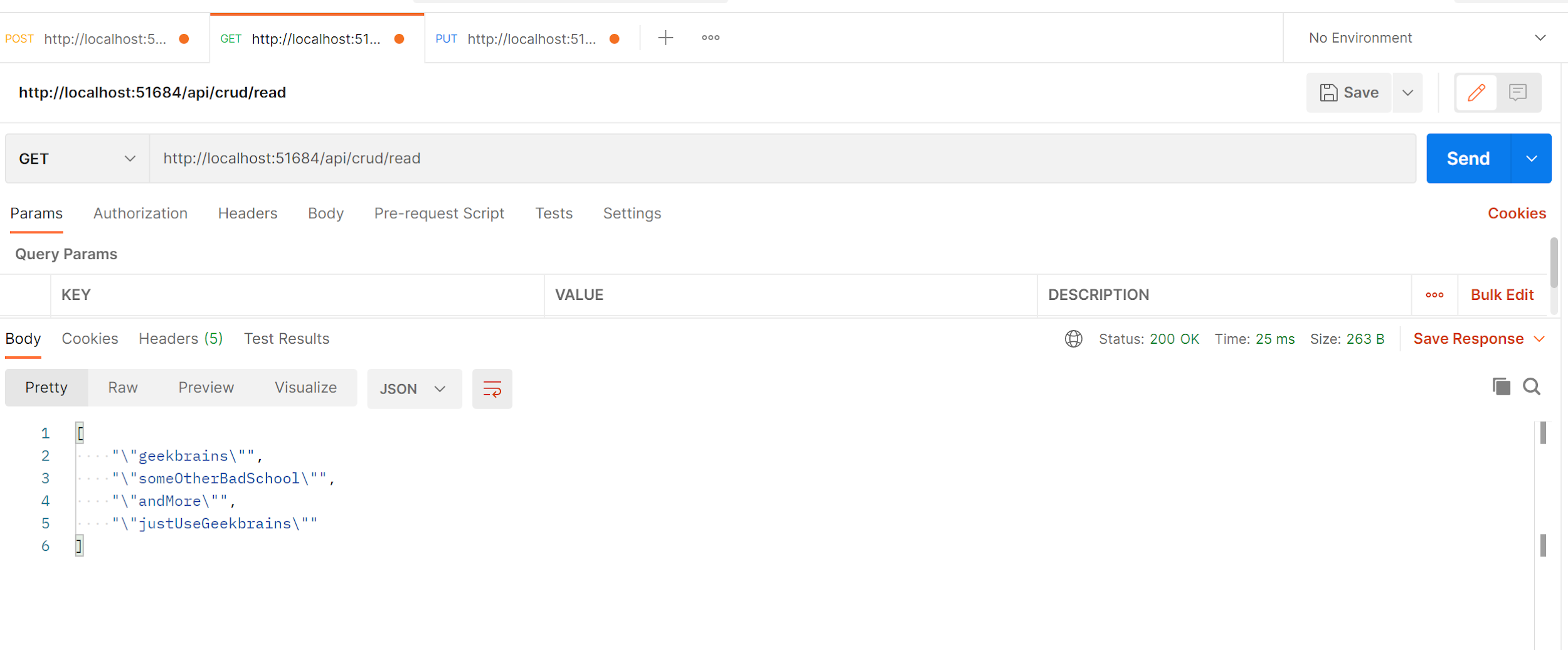
При помощи этой модификации, мы сообщаем что хотим чтобы наш микросервис запустился в профиле IIS Express, а когда откроется браузер, мы были перенаправлены на страницу api/crud/read. Подробнее о настройках запуска мы поговорим в рамках следующих занятий. Пока стоит привести ваш файл к такому виду, чтобы не возникло проблем с запуском вашего микросервиса.

## Отладка запущенного микросервиса

Теперь, когда у нас есть запущенный и работающий микросервис, давайте попробуем выполнить ряд запросов к нему при помощи Postman. Установите его как обычную программу для Windows скачав с сайта <https://www.postman.com/downloads/>. Для того, чтобы сервер не прекратил работу, а Visual Studio не удалила процесс вашего микросервиса из системы, не закрывайте открытую в браузере вкладку, появившуюся сразу после начала сеанса отладки сервиса.

Попробуйте добавить новые записи в состояние сервера добавив их при помощи метода Create

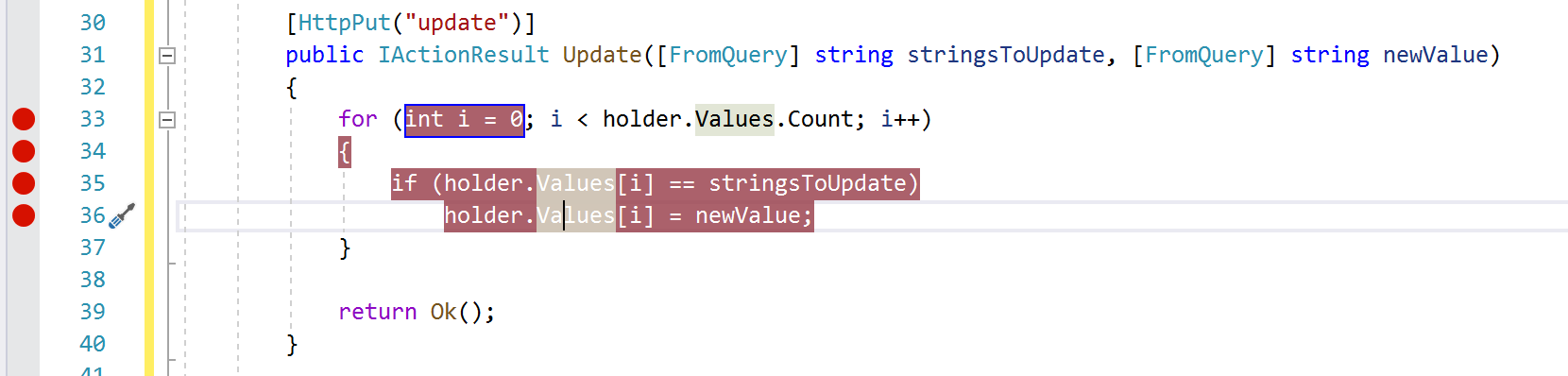
Как видите на рисунке, мы указали адрес нашего запущенного микросервиса, а также добавили параметр input, в который мы указали в качестве значения новый строковой литерал. При выполнении этого запроса, сервер добавит указанную в параметре строку в список.

Теперь попробуем прочитать все, что хранится на сервере при помощи метода Read.

В теле ответа от сервера мы увидим все добавленные при помощи метода Create строчки.

Попробуйте выполнить запросы на обновление и удаление строк самостоятельно при помощи Postman. Параметры в запросы можно добавлять аналогично методу Create, только учитывайте, что переменные уже называются по другому.

Отлаживать микросервисы можно так же как и обычные консольные программы. Попробуйте поставить точку останова в Visual Studio на любом методе нашего контроллера. При обращении к методу контроллера по http Visual Studio приостановит исполнение прямо на вашей точке останова и вы сможете посмотреть состояние и все переменные на вашем запущенном сервисе.



# Консольная утилита dotnet

Все указанные выше действия не обязательно выполнять при помощи Visual Studio. Принципиальная возможность воспользоваться другой IDE у разработчика есть всегда. В данном случае код можно писать хоть в стандартном блокноте (правда, это не так удобно).

Команда Microsoft сделала большую работу и теперь все сценарии работы с asp.net core можно выполнить при помощи консольной утилиты dotnet.

Пожалуйста, прежде чем пробовать работать с дальнейшими пунктами урока, убедитесь, что у вас установлена эта утилита при помощи выполнения в консоли команды

|  |
| --- |
| dotnet --info |

## Создание проекта web-api при помощи утилиты dotnet

Для того, чтобы создать новый проект asp.net core микросервиса достаточно выполнить следующую команду в консоли

|  |
| --- |
| dotnet new webapp -o aspnetcoreapp |

Выполнив эту команду в пустой директории вы увидите, что создались файлы на диске аналогичные тем, что мы создавали при помощи Visual Studio

## Сборка проекта

Точно так же легко как и с созданием проекта, можно легко собрать проект из консоли

|  |
| --- |
| dotnet build --configuration Release |

Выполнив эту команду вы увидите, что на диске появились dll файлы, которые содержат в себе ваш собранный микросервис.

## Запуск проекта при помощи утилиты dotnet

Теперь давайте попробуем запустить наш микросервис из консоли. Достаточно выполнить команду

|  |
| --- |
| dotnet MetricsManager.dll |

В консоли вы увидите информационные сообщения о том, что ваш микросервис успешно запущен. Теперь вы можете точно так же выполнить к нему запросы, например, с помощью Postman.

# Сборки

Как вы могли заметить, при помощи консольной утилиты dotnet мы запустили наше приложение, которое было сформировано в виде dll файла. Данный dll файл является сборкой нашего приложения. Сборки это важная логическая единица, набор типов и ресурсов, которые созданы для совместной работы и образуют логическую функциональную единицу.

Сборки могут быть сформированы в виде exe файлов (запускаемые программы) и dll файлов (динамически линкуемые библиотеки). Проще всего представлять dll файлы, как контейнеры с дополнительными возможностями, которые можно подключить к программе.

Сборки имеют следующие важные свойства:

* Сборки всегда exe или dll файлы
* Сборки загружаются в память только в тот момент, когда они требуются. Допустим, вы написали программу по математическим расчетам, а все сложные и малоиспользуемые функции решили выделить в отдельную сборку в виде подключаемой динамически библиотеки. В данном случае сборка будет загружена в память только при обращении к функции определенной в данной сборке
* Можно программно получать различную информацию о сборке, о классах и функциях в ней определенных

## Манифест сборки

Каждая сборка имеет файл манифеста сборки. Подобно оглавлению, манифест сборки содержит:

* Идентификатор сборки (ее название и версия).
* Таблица файлов, описывающая все остальные файлы, составляющие сборку, например, другие созданные сборки, на которые ссылается файл .exe или .dll, картинки и текстовые документы.
* Список ссылок на сборку, который представляет собой список всех внешних зависимостей, таких как DLL или другие файлы. Ссылки на сборки содержат ссылки как на глобальные, так и на частные объекты.

# Практическое задание

Написать свой контроллер и методы в нем, которые бы предоставляли следующую функциональность

* Возможность сохранить температуру в указанное время
* Возможность отредактировать показатель температуры в указанное время
* Возможность удалить показатель температуры в указанный промежуток времени
* Возможность прочитать список показателей температуры за указанный промежуток времени

# Список используемой литературы

1. [Developer tools, technical documentation and coding examples](https://docs.microsoft.com/).