Asp.net core Web API микросервисы. Часть 1

Rest API и тесты



# На этом уроке

На этом уроке мы рассмотрим принципы построения API микросервисов по методологии REST. Узнаем, что это за методология, какие плюсы она дает программисту при разработке API http-сервисов. На уроке мы начнем разработку нашей системы сбора метрик. Спроектируем API сервиса в REST-стиле. Дополнительно мы посмотрим на необходимость тестирования собственного кода и научимся тестировать свои контроллеры в сервисах.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_n93g6yblf23c)

[Правила написания кода](#_hxf2nqlm2ov)

[REST](#_2dvqmu20xave)

[История термина](#_ngcrxck3ibg)

[Свойства и ограничения](#_9032s9hrv9ms)

[Сервис сбора метрик](#_vwwwibmttems)

[Обзор решения создаваемого на курсе](#_9etzvn883jdb)

[Проектирование методов контроллеров в REST стиле](#_qycyeql8lfqy)

[Методы контроллеров менеджера метрик](#_ohy92ho0myg)

[Проектирование методов контроллера метрик CPU](#_l1h3dwnny796)

[Перцентили](#_e4tmhe9jo4yd)

[Контроллер управления агентами сбора метрик](#_mf2bj95j5jp4)

[Тестирование](#_fhoxgkekmssc)

[Обзор способов тестирования (юнит, интеграционное, функциональное)](#_5ms9frfvtzhp)

[Проект с тестами для сервиса менеджера метрик](#_6081sdp3pi4j)

[Пишем первый тест](#_rvvhv5jedz4l)

[Запуск тестов](#_7l4i8lbduodt)

[Проект сервис-агента сбора метрик](#_n9r8jonks8we)

[Практическое задание](#_65wvkckssbii)

[Список используемой литературы](#_54hssoocdbld)

[Дополнительные материалы](#_oreyj64p9zko)

# Вступление

На прошлом уроке мы рассмотрели создание проекта   ASP.NET Core микросервиса, который умеет взаимодействовать с пользователем по протоколу HTTP. Изучили основы работы протокола HTTP и базовые операции над данными, которые можно реализовать над данными при помощи методов контроллеров. Теперь мы погрузимся в проектирование API микросервиса на примере задачи, которая максимально приближена к той, что может возникнуть в реальной разработке. В ходе урока мы рассмотрим подход к проектированию интерфейсов микросервисов, который наиболее распространен в индустрии. Знание его основ позволяет программисту быстро понять API любого микросервиса, а чувство стиля, которое дает REST подход, позволяет легко отличить хороший API от плохого. В ходе урока мы также познакомимся с тестированием написанного кода на примере нашего тестового приложения. Тестирование написанного кода является важной и необходимой составляющей в современной разработке, и созданные микросервисы не исключение. Мы научимся писать автоматические тесты на созданные микросервисы. В рамках практического домашнего задания предлагается выполнить самостоятельное проектирование API микросервиса и добавить тесты на написанный код.

# Правила написания кода

Поскольку уже с этого урока мы начнем разрабатывать полноценное приложение, следует договориться о том, как писать код, чтобы он был легко понятен проверяющему. Для этого существуют соглашения по тому, в каком стиле писать код.

Почти в любой компании есть свои правила написания кода. Как должны именоваться переменные, классы; что именно использовать – табуляцию или пробелы в отступах; какие возможности языка стоит использовать, а какие нет. Эти правила либо берутся по умолчанию от компании Microsoft, либо дополняются, либо переписываются с нуля. Это очень важный момент в жизни разработки программного обеспечения. Несоблюдение этих правил может повлечь за собой отказ в приемке кода на этапе проверки старшими разработчиками. Крайне рекомендуется использовать в дальнейшем вот [этот стиль написания кода](https://gist.github.com/avgoncharov/a5d92e9aff5042087858504bcd480907).

Если эти правила не будут соблюдены, ваше домашнее задание не будет принято к проверке. Изучите их и используйте при решении всех последующих домашних заданий.

# REST



Основной для проектирования API современного микросервиса является так называемый REST подход. API (интерфейс сервисов) предпочитают делать в REST стиле, так как этот архитектурный стиль приводит к увеличению производительности в работе сервисов, а также упрощает понимание для программиста интерфейса сервиса, так как последний становится интуитивно-понятным. Данный подход в проектировании микросервисов накладывает ряд ограничений на программиста, которые мы рассмотрим ниже. Важным замечанием является то, что данный подход является стилем, но не стандартом, а потому “понимание” REST сервиса может быть разным даже у программистов, работающих в одной организации. Сервис, который полностью соответствует всем ограничениям, накладываемым на архитектуру сервиса в REST стиле называется restful (“рэстфул”, то есть полностью соответствующий правилам проектирования таких сервисов, “рестфульный”).

## История термина

Термин REST был введен в обращение одним из создателей протокола HTTP Роем Филдингом, в его диссертации 2000 года под названием “Архитектурные стили и дизайн сетевых программных архитектур”. Диссертация довольно длинная, но нам важно лишь знать, что Рой сформулировал теоретическую основу взаимодействия между клиентом и сервером. Эту теоретическую основу он назвал “передача репрезентативного состояния между клиентом и сервером” или же просто REST (representational state transfer) и обозначил подход, в рамках которого сервер сообщает клиенту полностью все состояние в ответ на запрос клиента. Таким образом, отпадает необходимость в поддержании пользовательской сессии (хранении информации о том, какое “состояние взаимодействия” сложилось между клиентом и сервером, например, как в случае с покупкой в интернет-магазине, когда у вас имеется ограниченный период времени для покупки товаров по выставленному счету на оплату, прежде, чем истечет время жизни сессии оплаты), что в свою очередь упрощает разработку таких сервисов, однако, в виду накладываемых ограничений, не позволяет применять данный подход в местах, где необходимо поддерживать “состояние взаимодействия” (то, в каком статусе находятся отношения между клиентом и сервером).

## Свойства и ограничения

Казалось бы, зачем вводить ограничения на интерфейс сервиса? Дело в том, что накладываемые ограничения в REST стиле определяют работу сервера в том, как он может обрабатывать и отвечать на запросы клиентов. Действуя в рамках этих ограничений, система приобретает такие желательные свойства как производительность, масштабируемость, простота, способность к изменениям, переносимость и отслеживаемость.

* Интуитивная **простота** интерфейса микросервиса
* Прозрачность связей между компонентами сервисов, что в свою очередь улучшает понимание программистами того, как лучше использовать микросервис, и того, какое использование микросервиса будет не оптимальным
* **Переносимость** свойств микросервиса путем простого переноса кода сервиса без дополнительных изменений
* **Надежность**, при соблюдении интуитивно понятной программистом прочностной характеристики микросервиса. То есть RESTful сервис сразу заявляет о том, сколько “примерно” запросов к себе он может выдержать, насколько серверу трудно обработать запросы, посылаемые клиентом и т.д.
* За счет отсутствия состояния в пользовательской сессии может достигаться большая **производительность** сервисов, так как её **масштабируемость** может достигаться путем простого добавления серверов обработчиков запросов.

Вышеуказанные свойства сервиса являются очень привлекательными с точки зрения программной инженерии, потому REST подход и завоевал любовь у создателей микросервисов как универсальный.

Существует шесть широко трактуемых ограничений к сервису, который разработан в REST стиле. Мы рассмотрим пять из них, так как шестой практически не используется:

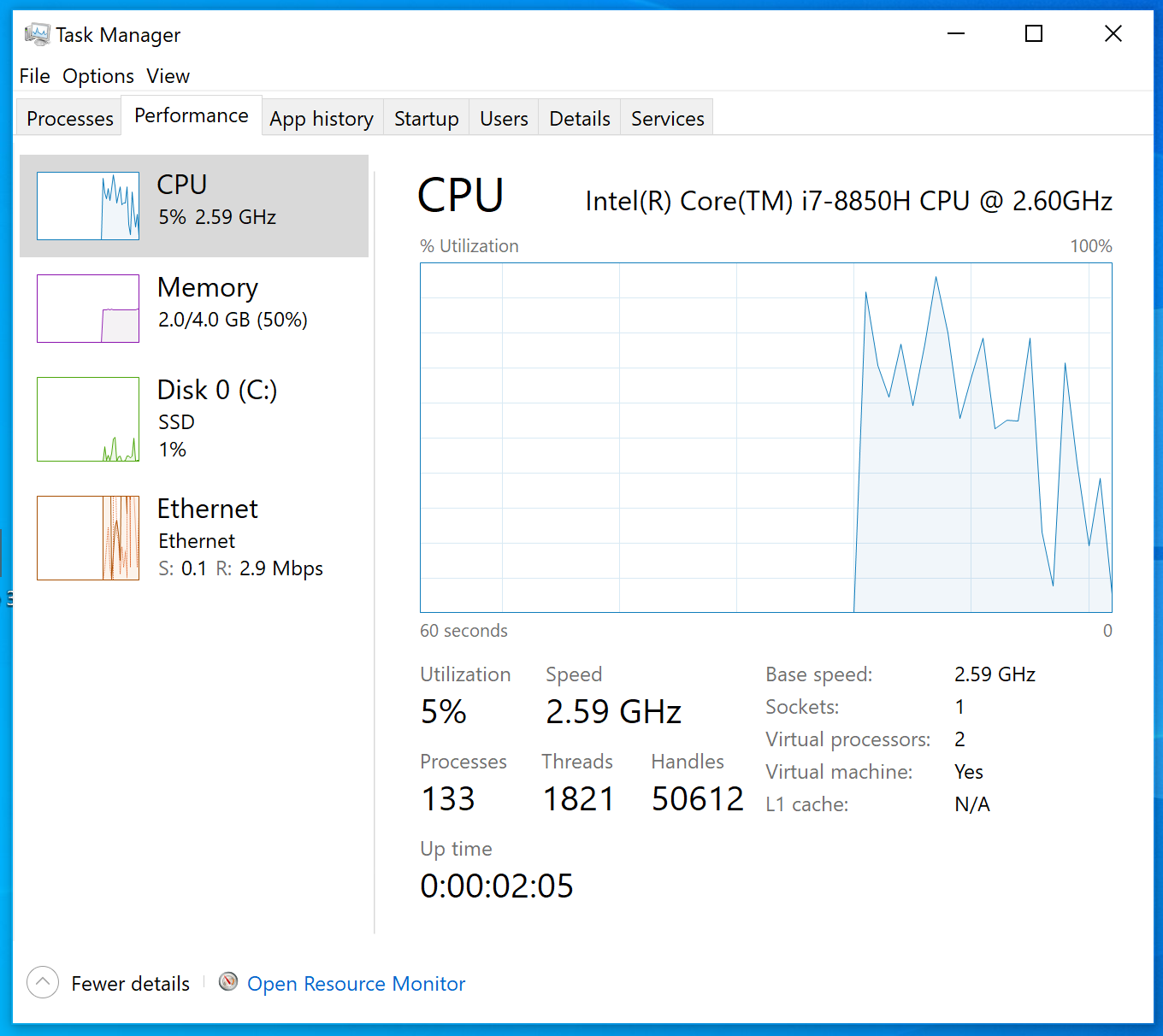
1. **Модель клиент-сервер.** Разграничение потребностей является принципом, лежащим в основе данного накладываемого ограничения. Отделение потребности интерфейса клиента от потребностей сервера, хранящего данные, повышает переносимость кода клиентского интерфейса на другие платформы, а упрощение серверной части улучшает масштабируемость.
2. **Отсутствие состояния.** В период между запросами клиента никакая информация о состоянии клиента на сервере не хранится в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Во время обработки клиентских запросов считается, что клиент находится в переходном состоянии. Каждое отдельное состояние приложения представлено связями, которые могут быть задействованы при следующем обращении клиента.
3. **Кеширование.** Клиенты, а также промежуточные узлы, могут выполнять кэширование ответов сервера. Ответы сервера, в свою очередь, должны иметь явное или неявное обозначение как кэшируемые или некэшируемые с целью предотвращения получения клиентами устаревших или неверных данных в ответ на последующие запросы.
4. **Единообразие интерфейса**. К этому пункту добавляются следующие ограничения:

* Идентификация ресурсов, которые отделены от формы представления, которое отдается клиентам. Например, сервер может отдавать данные в формате JSON и при этом не обязательно, чтобы на сервере они хранились именно в таком виде (то есть преобразование происходит как бы “на лету”)
* Манипуляция ресурсами через представление. Если клиент хранит представление ресурса, включая метаданные, он обладает достаточной информацией для модификации или удаления ресурса.
* Самоописание сообщений. То есть каждое сообщение содержит исчерпывающую информацию о том, как его обрабатывать
* Клиенты изменяют состояние системы только через действия, которые динамически определены в гипермедиа на сервере (ссылок и т.д.)

1. **Слои.** Клиент обычно не способен точно определить, взаимодействует он напрямую с сервером или же с промежуточным сервером. Применение промежуточных серверов способно повысить масштабируемость за счет балансировки нагрузки и распределенного кэширования. Промежуточные узлы также могут подчиняться политике безопасности с целью обеспечения конфиденциальности информации

# Сервис сбора метрик

Для того, чтобы лучше познакомиться с теоретическим материалом, в ходе курса мы будем разрабатывать приложение, состоящее из клиента и сервера. Клиент и сервер будут обладать API, построенным в REST стиле. Само приложение будет представлять из себя сервис-агент, который будет заниматься сбором системных метрик, и специальный сервис, который позволит представлять полученные с агента данные в необходимом для пользователя виде. Например, в диспетчере задач Windows вы можете видеть ресурсы, потребляемые запущенными приложениями. Данные показатели работы компьютера очень важны для управления IT-инфраструктурой, потому специализированные команды разработчиков создают программы сбора таких метрик. Данная задача позволяет наиболее полно ознакомиться с типичными задачами разработчика ASP.NET core сервисов. Сервис, который позволяет собирать метрики, а также сервис, который предоставляет информацию о собранных метриках, мы и будем разрабатывать на курсе.



На картинке выше вы видите пример того, как собираются метрики по работе CPU. Это скриншот из диспетчера задач Windows, где можно посмотреть параметры производительности компьютера. Вы можете видеть метрики загрузки CPU в процентах. Точно такие же данные и будет собирать наш агент метрик.

## Обзор решения создаваемого на курсе

Теперь нам необходимо определиться с архитектурой создаваемого решения. Выделим два основных компонента: менеджер и агент. Менеджер метрик будет работать с агентами в архитектурном подходе, называемом провайдер-потребитель. Такая архитектура является двухкомпонентной. Первая компонента - сервис агент, который осуществляет сбор и хранение метрик. Данный микросервис будет устанавливаться непосредственно на машину, с которой необходимо собрать метрики работы. Теоретически может быть много экземпляров таких сервисов, которые установлены каждый на свою машину, и возникает задача по сбору данных с таких агентов. Таким образом, мы выделяем еще один компонент нашей системы в виде сервиса менеджера метрик. Сервис будет обращаться за данными о собранных метриках работы к сервису агенту, осуществлять преобразование и хранение собранных с агента данных, а также предоставлять пользовательский интерфейс для работы с ним.

Мы будем собирать простые метрики работы компьютера

* Загрузка CPU (процессора) в процентах каждую секунду
* Использование оперативной памяти каждую секунду
* Использование пространства жесткого диска
* Использование пропускной способности сети
* Некоторые метрики работы CLR (специфично для ASP.NET Core приложений)

После того, как мы выделили основные компоненты в системе и обозначили их зоны ответственности, мы можем спроектировать контроллеры сервисов в REST стиле и написать код, который отдает пока еще пустые ответы в ответ на запросы.

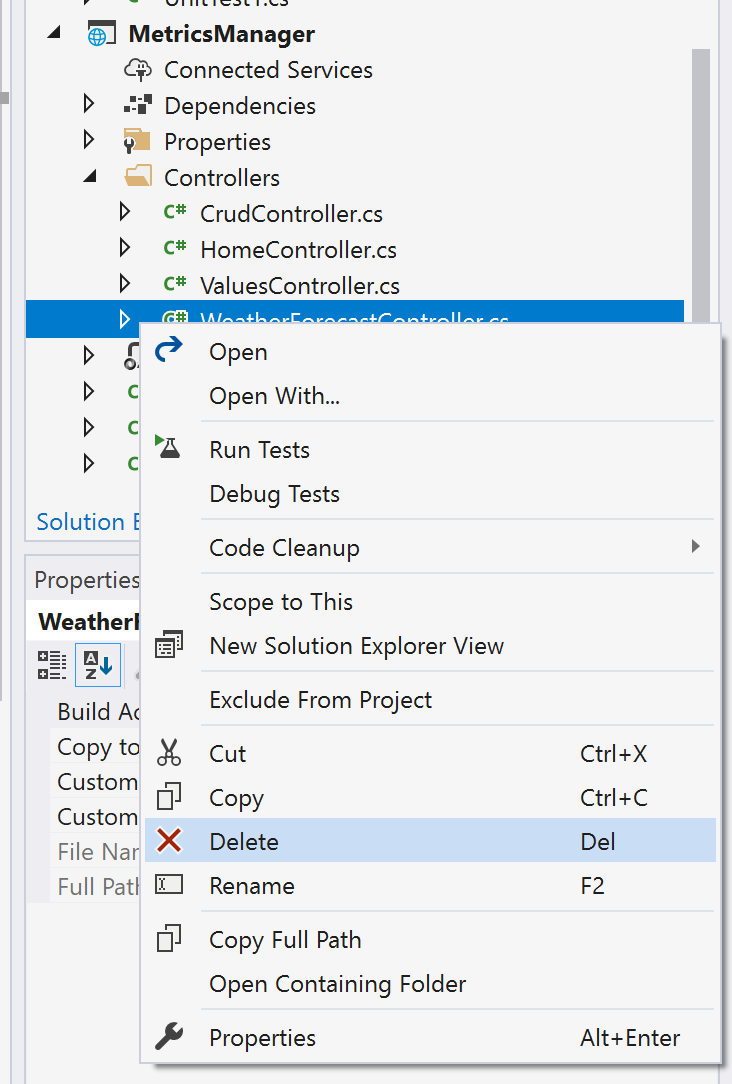
# Проектирование методов контроллеров в REST стиле

Мы уже выделили основные метрики, которые будем собирать. Как и на прошлом уроке, нам необходимо выделить методы в контроллерах нашего приложения, чтобы создать API сервисов. Давайте начнем с проектирования методов контроллеров менеджера метрик.

## Методы контроллеров менеджера метрик

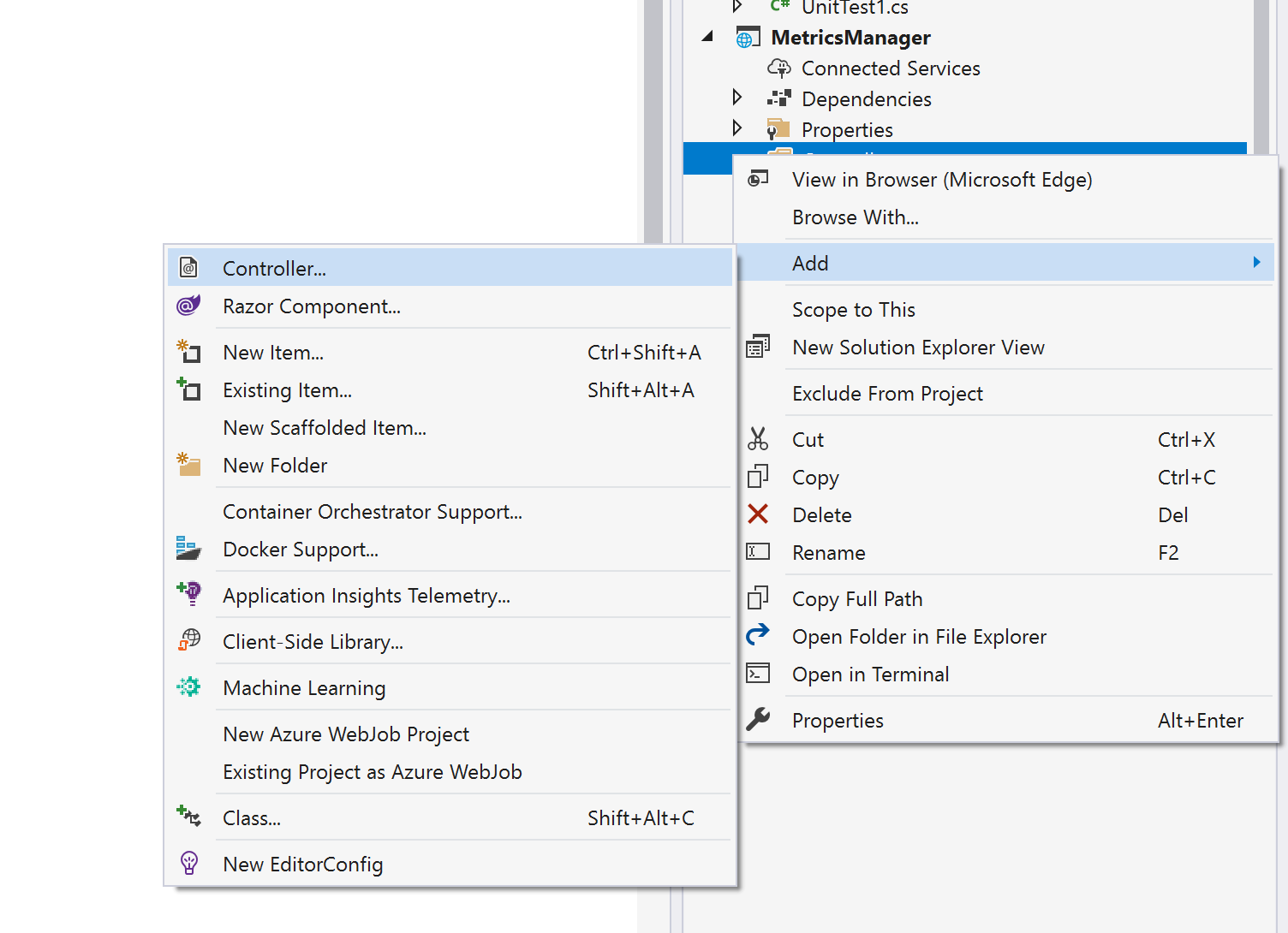
Мы можем выделить несколько основных метрик, которые имеются в приложении

1. CPU
2. HardDrive
3. RAM
4. Network
5. .NET metrics

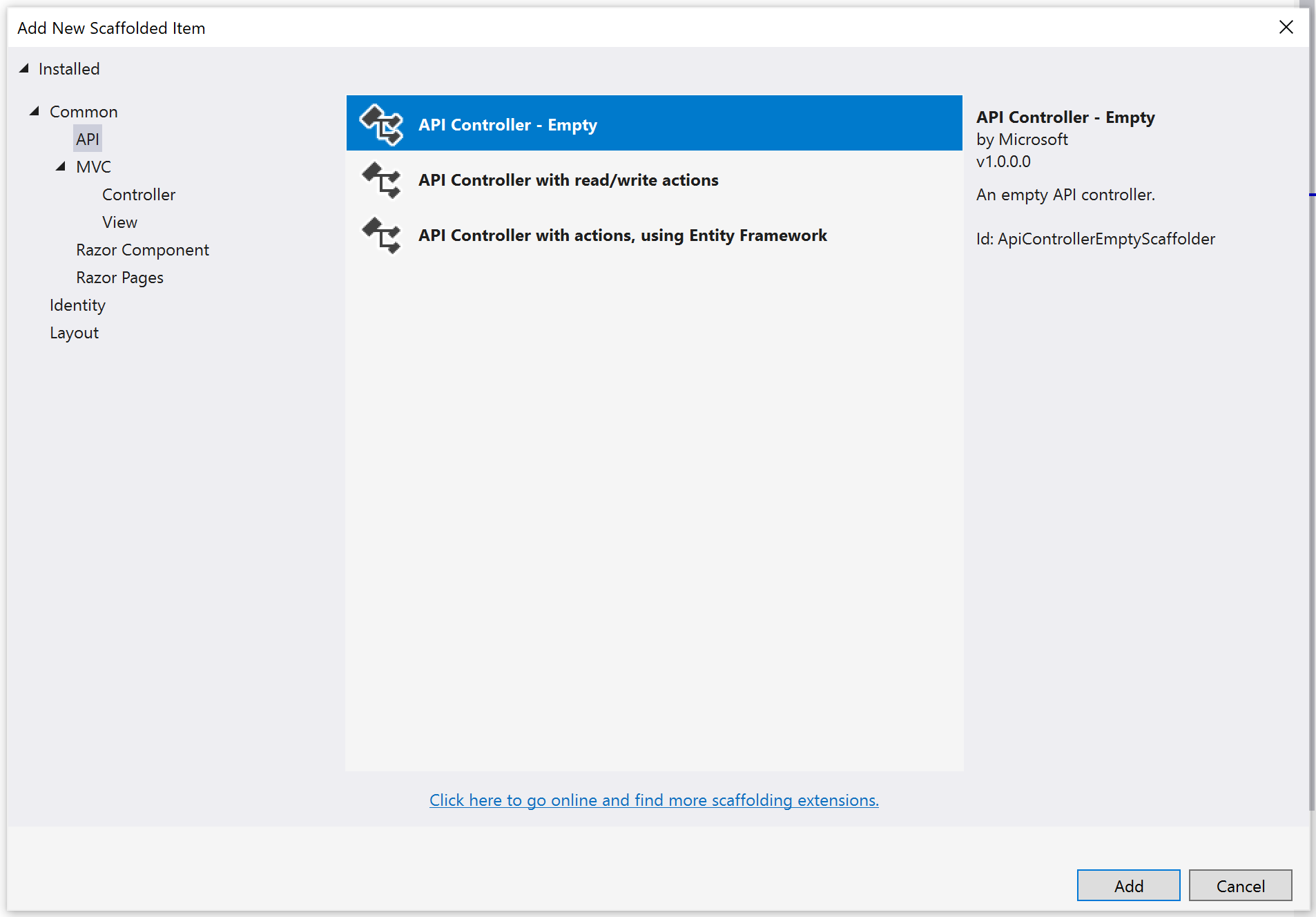
Таким образом нам понадобится пять контроллеров. Для начала удалите тестовые контроллеры, которые остались с прошлого урока, как показано на рисунке ниже:

Выполните действие удаления для CrudController, HomeController, ValuesController и WeatherForecastController.

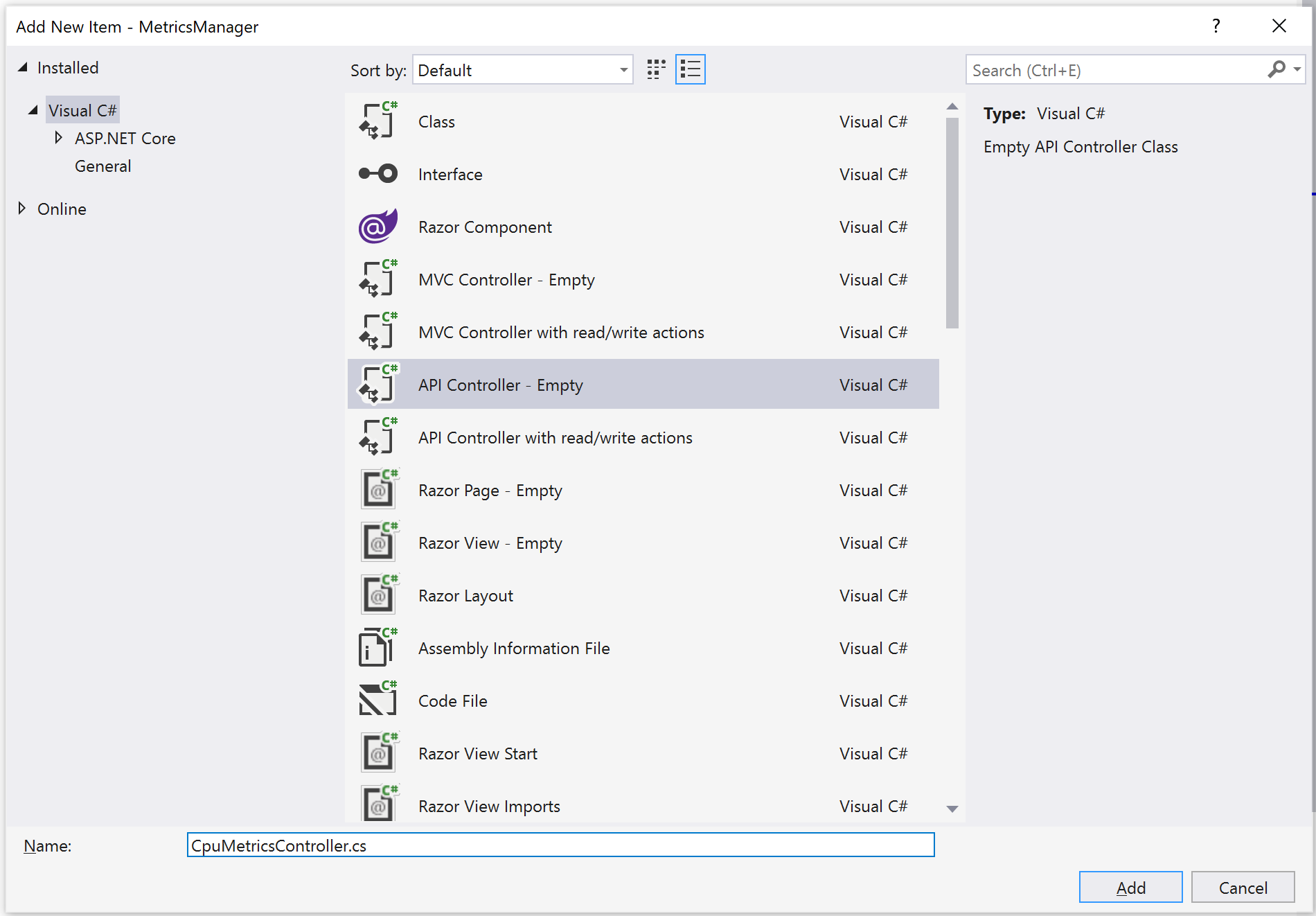
Теперь давайте добавим контроллер, который будет отвечать за API по загрузке CPU. Для этого создайте новый контроллер, как показано на рисунке. Щелкните правой кнопкой мыши на папке Controllers и выберите действие добавления нового контроллера.



После этого выберите шаблон, как показано на рисунке. Данный шаблон позволяет создать именно API контроллер, который разработан специально для того, чтобы отдавать простые ответы, а не создавать целые страницы, которые будут отправлены браузеру.



После выбора шаблона появится еще одно окно, в котором следует указать название нового контроллера. Контроллеры принято называть по имени существительному (их функция) с суффиксом Controller. Таким образом, поскольку у нас контроллер будет отвечать за CPU метрики, назовем контроллер CpuMetricsController.



Аналогичным образом создадим контроллеры для остальных метрик. Назовем их RamMetricsController, HddMetricsController, NetworkMetricsController и DotNetMetricsController соответственно.

Код пустого контроллера выглядит следующим образом:

|  |
| --- |
| 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 CpuMetricsController : ControllerBase  {  } } |

Для начала изменим путь к нашему API в соответствии с REST стилем. Для этого перепишем значение атрибута Route на api/metrics/cpu. Так наш контроллер для метрик CPU будет работать по пути api/metrics/cpu и любой программист, который будет пользоваться нашим созданным API, догадается, что по данному пути находится API метрик для CPU. Аналогично необходимо переделать значения атрибутов в остальных контроллерах.

## Проектирование методов контроллера метрик CPU

Поскольку мы уже начали проектирование API с CPU метрик, мы продолжим делать их на примере CPU метрик, а для остальных контроллеров вы можете проделать все то же самое самостоятельно.

После того, как созданы все контроллеры, мы можем создать методы в них, которые будут отдавать ответы пользователю. Какие это будут методы? Для этого нам необходимо подумать над возможными выдаваемыми ответами и выделить необходимые “домены”. То есть некоторый смысловой набор, и оформить это в виде методов API,

Мы знаем, что нам необходимо отдавать метрики по CPU для менеджера метрик, который будет работать со множеством агентов, их собирающих. Также, мы знаем, что метрики будут иметь некоторый временной диапазон, в котором просматривающему их человеку будет интересно их узнать. Таким образом, нам необходимы методы, которые будут отдавать метрики в заданном диапазоне времени с определенного агента, метрики в заданном диапазоне времени со всех агентов, а также некоторые усредненные значения на определенном интервале времени (перцентили) аналогично с определенного агента или целого кластера агентов.

Давайте напишем эти методы для контроллера CPU метрик.

|  |
| --- |
| using MetricsManager.Enums; using Microsoft.AspNetCore.Mvc; using System;  namespace MetricsManager.Controllers {  [Route("api/metrics/cpu")]  [ApiController]  public class CpuMetricsController : ControllerBase  {  [HttpGet("agent/{agentId}/from/{fromTime}/to/{toTime}")]  public IActionResult GetMetricsFromAgent([FromRoute] int agentId, [FromRoute] TimeSpan fromTime, [FromRoute] TimeSpan toTime)  {  return Ok();  }   [HttpGet("agent/{agentId}/from/{fromTime}/to/{toTime}/percentiles/{percentile}")]  public IActionResult GetMetricsByPercentileFromAgent([FromRoute] int agentId, [FromRoute] TimeSpan fromTime, [FromRoute] TimeSpan toTime)  {  return Ok();  }   [HttpGet("cluster/from/{fromTime}/to/{toTime}")]  public IActionResult GetMetricsFromAllCluster([FromRoute] TimeSpan fromTime, [FromRoute] TimeSpan toTime)  {  return Ok();  }   [HttpGet("cluster/from/{fromTime}/to/{toTime}/percentiles/{percentile}")]  public IActionResult GetMetricsByPercentileFromAllCluster([FromRoute] TimeSpan fromTime, [FromRoute] TimeSpan toTime,  [FromRoute] Percentile percentile)  {  return Ok();  }  } } |

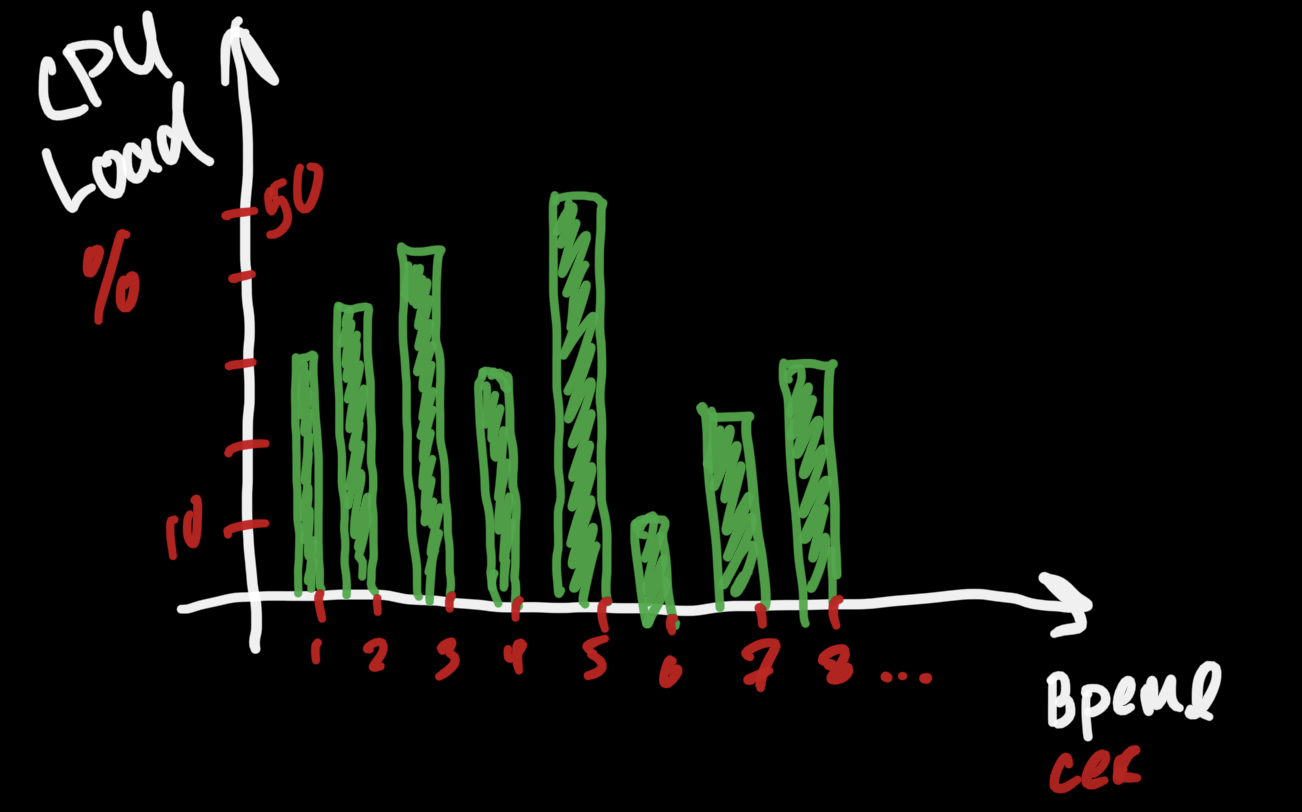
Мы обозначили основные методы для контроллера метрик CPU. Всего их получилось четыре, два из которых отвечают за метрики с определенного агента и еще два - за метрики CPU со всего кластера.

Методы контроллера предполагают только чтение объектов, содержащихся в системе. Исходя из этого, в соответствии с рекомендациями по созданию REST сервисов, все методы вызываются при помощи http-метода GET.

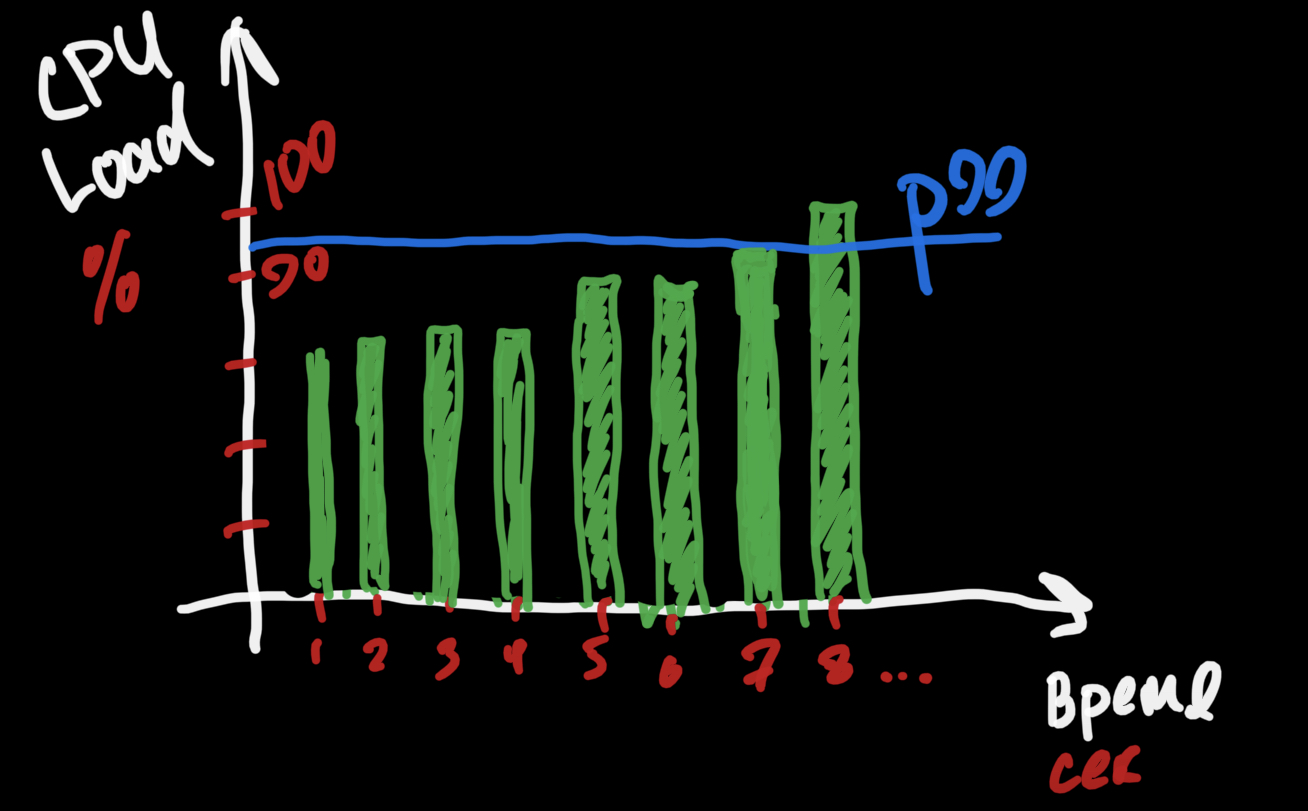
В атрибутах, которыми помечены методы контроллера, вы можете видеть пути, по которым сервис будет отвечать. Пути спроектированы в REST стиле, мы обозначили человеко-понятный путь до метода и одновременно с этим указали в виде плейсхолдеров (по шаблону), где будут указаны параметры для методов. В атрибуте HttpGet в качестве значения можно указать путь, по которому роутер нашего сервиса (встроенный в asp.net core) поставит в соответствие обработчик, одновременно с этим, взяв из пути необходимые параметры для вызова метода. Например, в значении атрибута для первого метода указаны следующие параметры прямо в пути [HttpGet("agent/{agentId}/from/{fromTime}/to/{toTime}")]. Это означает, что идентификатор агента, а также метки начала и конца временных интервалов будут взяты из пути вызова метода. Для того, чтобы это сработало, необходимо в методе контроллера назвать параметры аналогично. Это необходимо, чтобы стандартный сборщик модели (model binder из asp.net core фреймворка) смог сопоставить имя входного параметра метода и то, что у нас указано в пути в атрибуте метода. Это соглашение позволяет легко и просто расставлять параметры в пути api методов. Дополнительно, на параметрах методов указано при помощи атрибутов [FromRoute] то, что данный параметр необходимо взять из пути. Это “дополнительная подсказка” сборщику модели запроса.

### Перцентили

Внимательный читатель мог заметить, что у нас в контроллерах фигурирует новый тип Percentile. Это перечисление будет хранить у нас перцентили. Нам интересны: медиана, 75ая, 90ая, 95ая и 99ая перцентили. Перцентиль это методика измерения в статистике, которая показывает процент значений измеряемой метрики, который находится ниже значения перцентиля. Например, если говорить о CPU метрике, 99й перцентиль на интервале времени 100 секунд говорит о том, что 99% измеряемых загрузок процессора не превышали определенного значения. Можно представить себе кучу столбиков (посмотрите на рисунок ниже) которые обозначают загрузку процессора в определенный момент времени в процентах.



Теперь мы возьмем и отсортируем полученные значения по высоте от нижнего к высшему на нашем интервале времени. Теперь проведем воображаемую границу по значению 99го столбика. Именно это значение будет показывать девяносто-девятую перцентиль данной метрики, то есть то, что все метрики в заданном диапазоне не превышают указанной. Перцентилями удобно пользоваться при анализе работы сервисов по различным метрикам, так как они показывают, как и в каких процентах интересующих случаев сервисы работают.



Теперь можно посмотреть на код перечисления, которое будет отвечать у нас за все интересующие нас перцентили.

|  |
| --- |
| namespace MetricsManager.Enums {  public enum Percentile  {  Median = 0,  P75 = 1,  P90 = 2,  P95 = 3,  P99 = 4  } } |

Благодаря этому перечислению в пути метода сервиса можно указать следующий адрес: api/metrics/cpu/agent/1/from/1/to/1/percentiles/P95. Сборщик модели запроса (model binder) сопоставит название из пути в соответствие перечислению, а мы, как программисты, получим человеко-понятное название для интересующей перцентили прямо в коде приложения, что является очень удобным. Студентам рекомендуется при проектировании своих API заранее продумывать перечисления для ограниченных диапазонов значений и вводить их непосредственно в код своих приложений, так как это позволяет писать типобезопасный (type-safety) код, который легко тестировать, а значит, менее подверженный ошибкам.

Остальные контроллеры будут иметь аналогичные методы. Студентам предлагается написать их самостоятельно.

Пока что данный метод на данном уроке можно оставить просто заглушкой, которая ничего не возвращает, так как у нас еще нет сбора данных. В будущем, когда вы будете собирать метрики, необходимо будет взять все метрики за определенный диапазон (запрашиваемый), отсортировать их по значению и поделить по необходимой границе (по перцентилю) в процентном соотноешении. То, где будет проходить граница и следует вернуть в ответ. Например, предположим что значений получилось ровно 100 штук и все они представляют собой ряд натуральных чисел от 1 до 100, но не в порядке следования. Отсортируем полученные данные, поделим так, что 75 процентов метрик будет в одной стороне, а остальные 25 в другой. Таким образом, 75ый перцентиль будет равен 75 и т.д.

## Контроллер управления агентами сбора метрик

Дополнительно нам в менеджере метрик понадобятся методы для работы с агентами сбора метрик. Скорее всего, нам понадобится зарегистрировать агента в нашей системе, удалять его из системы, включать и выключать определенные агенты, когда они уже зарегистрированы. Давайте добавим контроллер AgentsController аналогично способу выше и наполним его следующими методами:

|  |
| --- |
| using Microsoft.AspNetCore.Mvc; using System;  namespace MetricsManager.Controllers {  [Route("api/[controller]")]  [ApiController]  public class AgentsController : ControllerBase  {  [HttpPost("register")]  public IActionResult RegisterAgent([FromBody] AgentInfo agentInfo)  {  return Ok();  }   [HttpPut("enable/{agentId}")]  public IActionResult EnableAgentById([FromRoute] int agentId  {  return Ok();  }   [HttpPut("disable/{agentId}")]  public IActionResult DisableAgentById([FromRoute] int agentId)  {  return Ok();  }  }   public class AgentInfo  {  public int AgentId { get; }   public Uri AgentAddress { get; }  } } |

Итого у нас имеются метод для регистрации агента сбора метрик и два метода для включения и выключения работы агента сбора метрик соответственно. Метод для регистрации агента принимает в себя комплексный объект, причем делает это из тела запроса (протокол http метод POST имеет в себе “тело”, в котором в нашем случае будет содержаться объект класса AgentInfo). Чтобы asp.net core фреймворк при помощи сборщика модели запроса понял, в какую переменную во входной параметр метода контроллера положить собранный из тела объект, параметр метода помечен атрибутом [FromBody].

Метод для регистрации агента в системе вызывается http-post методом. Именно этим методом рекомендуется в REST стиле создавать новый ресурс на сервере, а так как у нас регистрация нового агента в системе по сути является его созданием, именно этот http-метод мы и выбрали. Методы для включения и выключения объекта выполнены при помощи http-метода PUT, так как в семантике REST модификация уже имеющихся на сервере объектов производится при помощи http-метода PUT.

В атрибуте [Route] мы не стали изменять путь. Asp.net core сам возьмет название контроллера (Agents) и подставит его в путь для всех методов контроллера, в соответствии с шаблоном (плейсхолдер [controller] отвечает за это). Таким образом, например, метод для регистрации агента будет доступен по пути api/agents/register.

Вы можете самостоятельно попробовать вызвать методы контроллеров при помощи утилиты Postman. Будьте внимательны с выбираемым http-методом при запросе к серверу.

Вспомогательный класс оставлен внутри файла с контроллером только для компактности кода, студентам рекомендуется перенести класс AgentInfo в отдельный файл.

# Тестирование

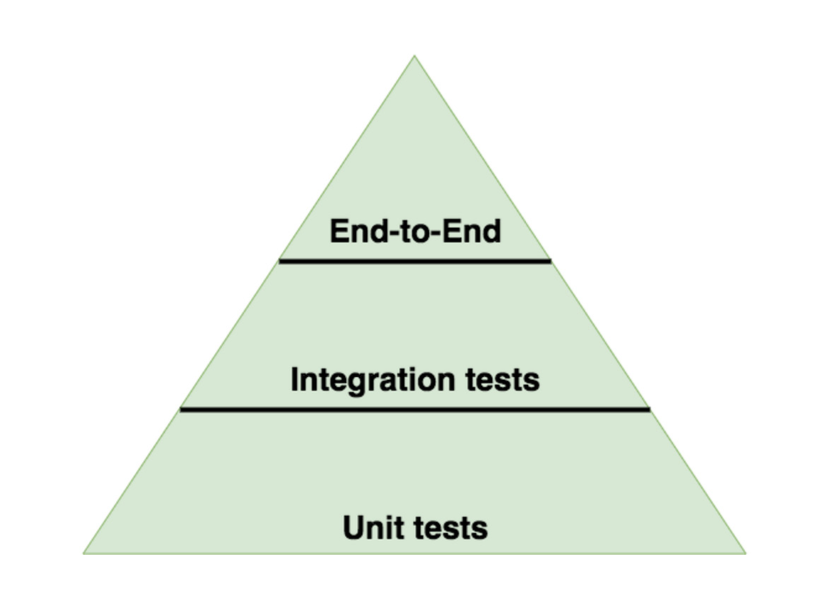


Теперь, когда у нас уже есть некоторый интерфейс наших сервисов, важно упомянуть о том, что весь собственный код программисты проверяют на соответствие требованиям. Тестирование программного обеспечения это целая отдельная отрасль и тестирование программного обеспечения занимаются QA специалисты (Quality Assurance Specialist или же тестировщики). Однако, программисты обязаны проверять свой собственный код прежде, чем они отдадут его на проверку QA специалистам. В рамках курса после каждого занятия студентам рекомендуется “покрывать” свой собственно написанный код автоматическими тестами. Тестирование собственно написанного кода позволяет избежать ошибок в проектировании логики сервисов, а разработка через тестирование является рекомендацией к созданию сложных программных систем. Теперь рассмотрим, какое автоматическое тестирование существует и какие тесты обязан написать программист самостоятельно.

## Обзор способов тестирования (юнит, интеграционное, функциональное)

Выделяют три уровня тестирования программного обеспечения

1. Модульное (юнит) тестирование
2. Интеграционное тестирование
3. Функциональное тестирование



Модульное тестирование представляет из себя изолированное тестирование самых маленьких компонент в программной системе (отдельных модулей). В коде, который написан на языке C# таким модулем может являться отдельный класс. Идея состоит в том, чтобы написать код, который автоматически будет проверять не очевидные кусочки кода на корректность. Количество тестов такого рода на проекте может исчисляться десятками тысяч и зависит лишь от объема и сложности кодовой базы. Такие “тесты” являются очень маленькими и быстрыми, а потому их можно запускать после каждого изменения в системе и таким образом проверить, что внесенные изменения не разрушили существующую логику работы приложения. Юнит-тесты обязаны быть маленькими и проверять минимально-возможный кусочек кода на корректность. Один юнит-тест проверяет ровно один случай в работе кода программы, а все остальные случаи обязаны проверяться отдельными юнит-тестами. Такой подход может показаться излишним (программисты не хотят писать лишний код и это верно), но он позволяет добиться высокой стабильности в работе написанного кода.

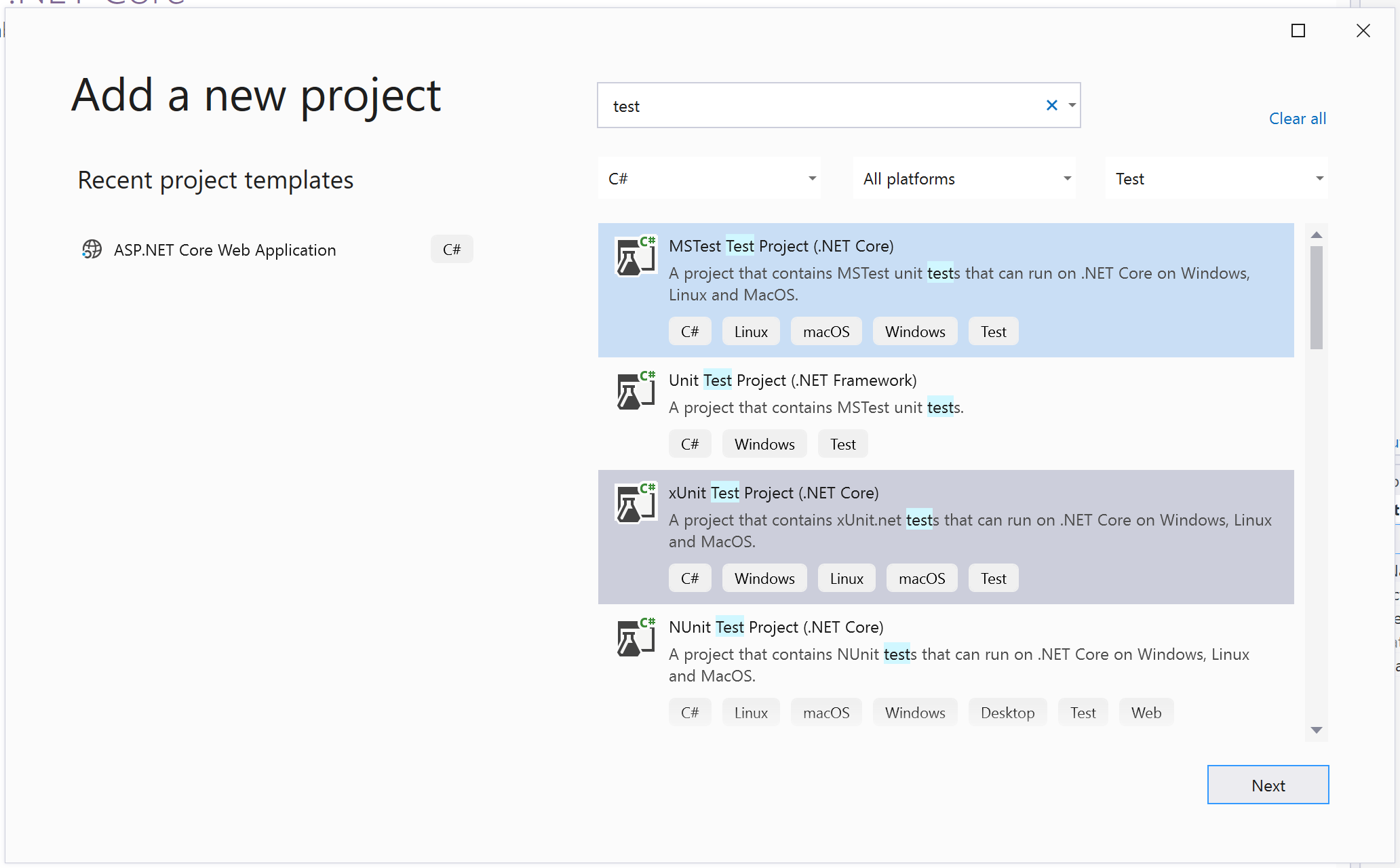
Интеграционное тестирование представляет из себя тестирование отдельных компонент в составе большой системы. Например, в нашем приложении сбора метрик будет существовать отдельный компонент для работы с базой данных и отдельный компонент для абстракции от конкретной базы данных в пользу общего интерфейса. Осуществление автоматической проверки на корректность взаимодействия этих двух компонент при помощи написанного теста и является интеграционным тестом. Важно упомянуть, что тестирование компонент осуществляется методом черного ящика, то есть автоматическому тесту вовсе не обязательно знать, как именно написанный код выполняет функциональность (в отличии от юнит-теста, который может проверять, например, одно конкретное условие в отдельно взятом методе класса), достаточно лишь проверки входных и выходных данных после работы компонента системы.

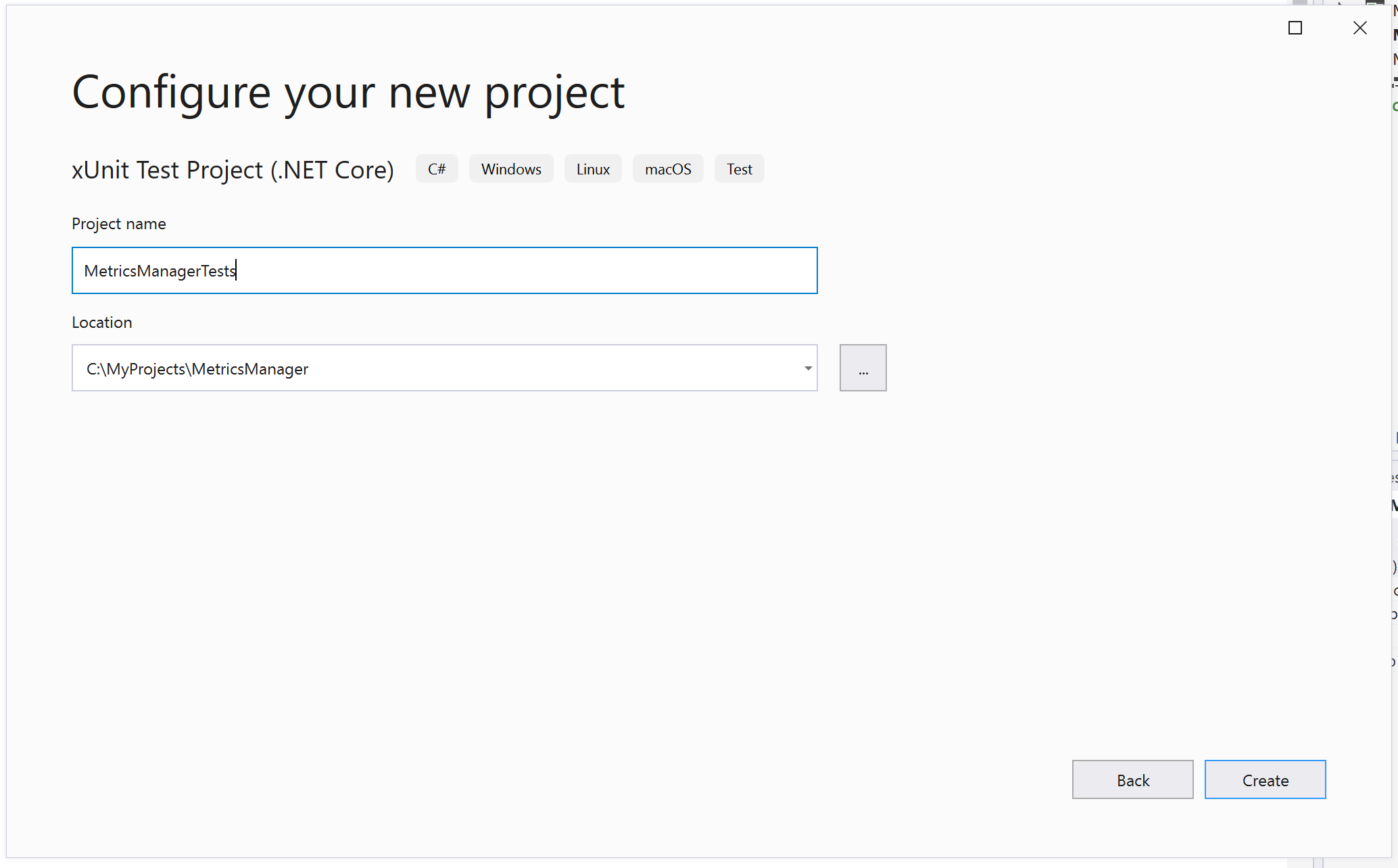
Функциональное тестирование проверяет программную систему целиком на соответствие заявленным требованиям, без проверки отдельных компонент на корректность. Точно так же, как и интеграционное, функциональное тестирование осуществляется методом черного ящика. В конце разработки нашей системы сбора метрик, мы осуществим функциональное альфа-тестирование (то есть силами разработчика системы) менеджера метрик.

В ходе работы над материалом курса, чаще всего придется иметь дело с юнит-тестами, так как именно этот способ тестирования является обязательным для программистов и написанного ими кода. Некоторые компании вводят метрику покрытия кода тестами. Грубо говоря, таким образом проверяется количество строк кода на которые есть автоматический юнит-тест по отношению к строкам кода, которые такими тестами не покрыты. Повышая процент покрытия кода тестами, компании, вероятно, увеличивают стабильность работы своих программных продуктов.

# Проект с тестами для сервиса менеджера метрик

Давайте создадим специальный проект для тестов сбора метрик. Для этого в Solution Explorer нажмите правой кнопкой мыши на созданном решении и выберете Add Project. Выберете шаблон для тестов, как показано на рисунке

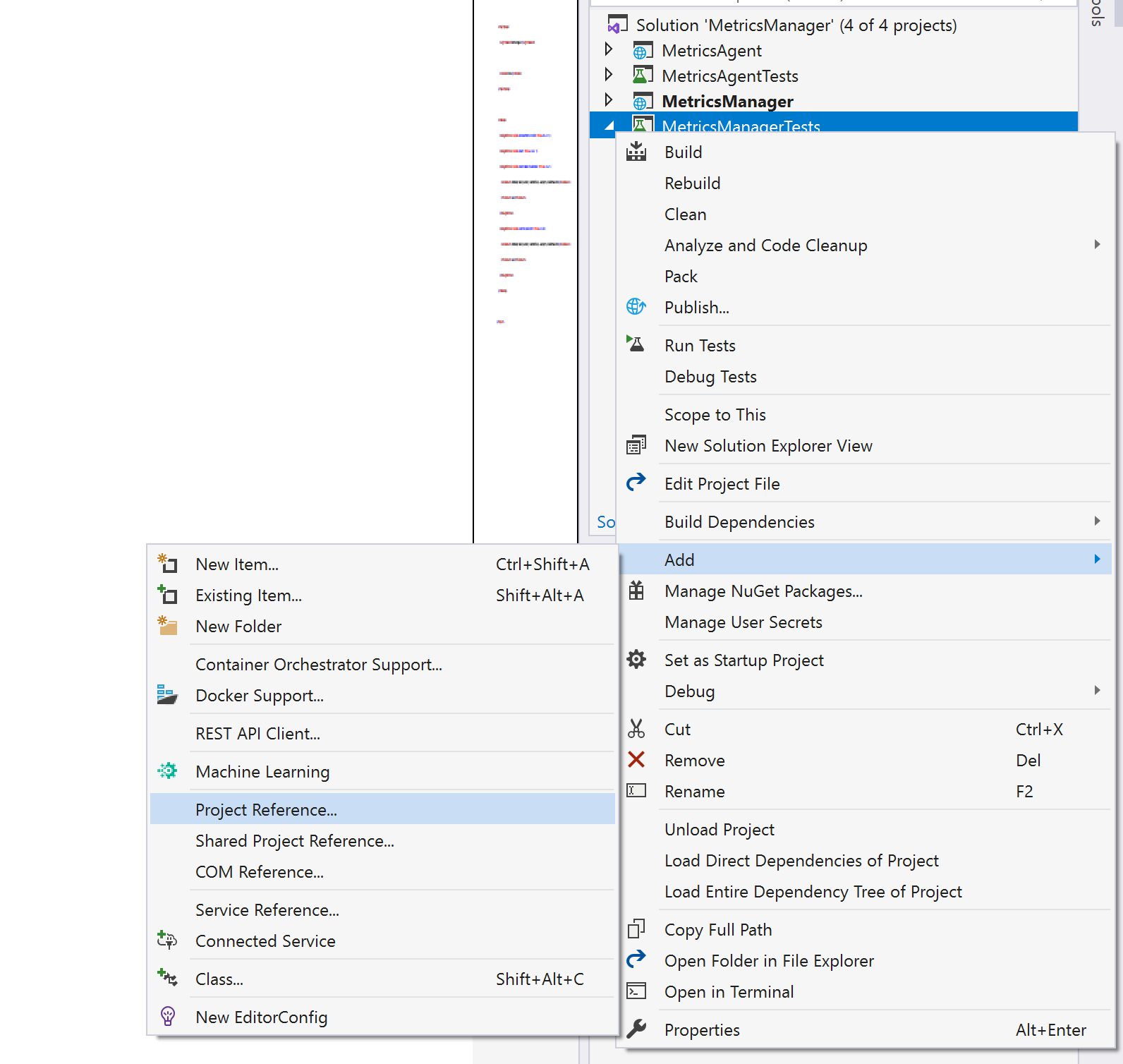


Существует ряд библиотек, которые помогают в юнит-тестировании приложений, написанных на C#. В рамках данного курса мы воспользуемся xUnit библиотекой. Другие библиотеки, за исключением некоторых особенностей работы, работают аналогично выбранной нами и прочитать код чужих тестов выполненных на другой библиотеке не составит труда, даже при незнании сторонней библиотеки. 

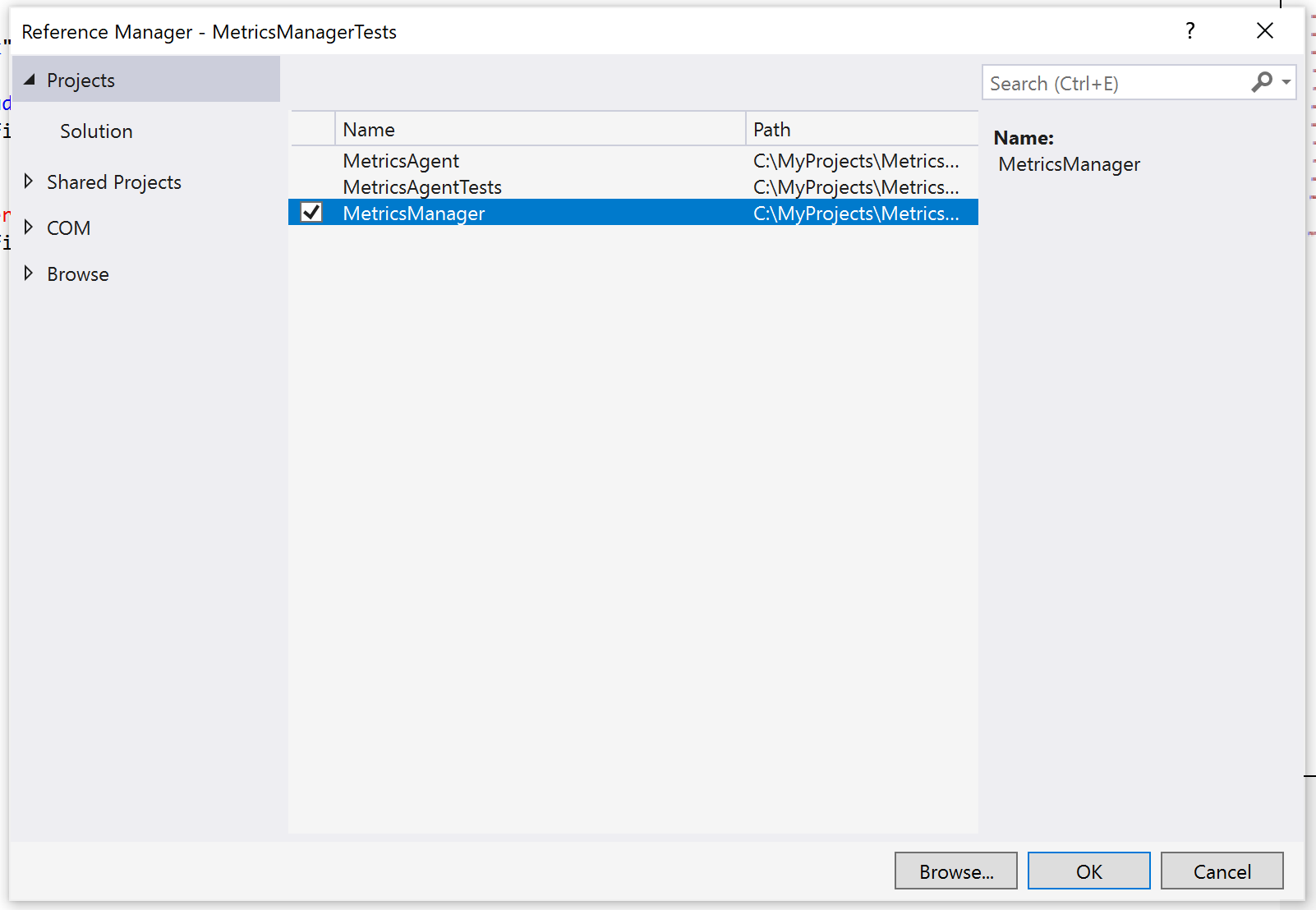
В шаблоне проекта присутствует единственный юнит-тест, код которого приведен ниже:

|  |
| --- |
| using System; using Xunit;  namespace MetricsManagerTests {  public class UnitTest1  {  [Fact]  public void Test1()  {   }  } } |

Для того, чтобы написать простой тест на метод контроллера из сервиса менеджера сбора метрик, нам необходимо добавить ссылку на проект в наш проект с тестами. Для этого выберите пункт добавления ссылки на проект, как показано на рисунке ниже:



В появившемся окне выберете проект менеджера сбора метрик, как показано на рисунке:



## Пишем первый тест

После добавления ссылки на проект, который мы будем тестировать, мы можем написать наш первый юнит-тест. Напишите код, как показано в сниппете ниже:

|  |
| --- |
| using MetricsManager.Controllers; using Microsoft.AspNetCore.Mvc; using System; using Xunit;  namespace MetricsManagerTests {  public class CpuControllerUnitTests  {  private CpuMetricsController controller;   public CpuControllerUnitTests()  {  controller = new CpuMetricsController();  }   [Fact]  public void GetMetricsFromAgent\_ReturnsOk()  {  *//Arrange*  var agentId = 1;  var fromTime = TimeSpan.FromSeconds(0);  var toTime = TimeSpan.FromSeconds(100);   *//Act*  var result = controller.GetMetricsFromAgent(agentId, fromTime, toTime);   *// Assert*  \_ = Assert.IsAssignableFrom<IActionResult>(result);  }  } } |

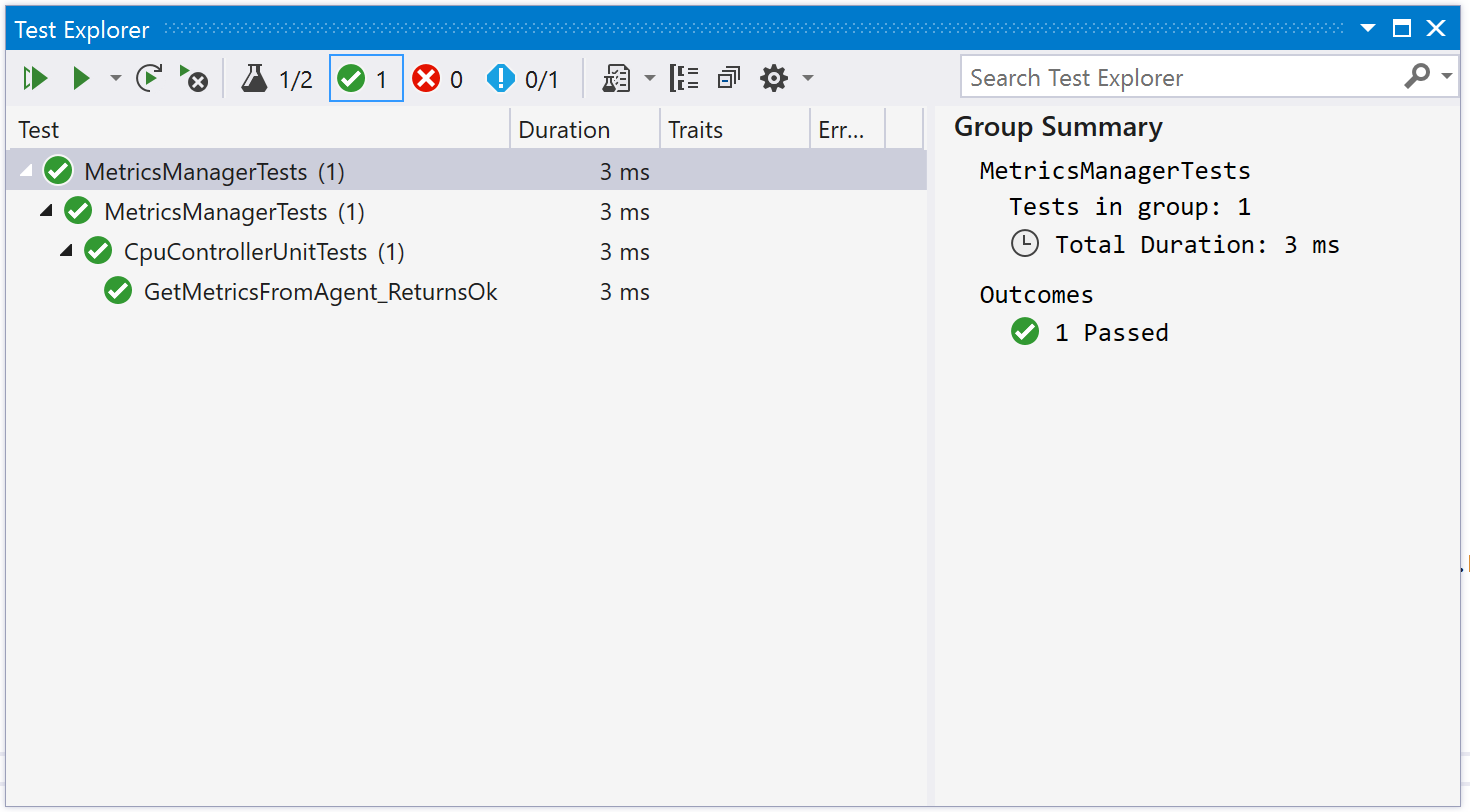
Каждый юнит-тест содержит три секции: подготовка данных, действие, проверка результата. Метод, выполняющий тест, помечен атрибутом [Fact], который подсказывает запускающему его коду, что в нем содержится тест. В приведенном выше коде, в секции подготовки данных мы устанавливаем интересующие нас параметры для метода получения метрик с агента, затем вызываем метод с перечисленными параметрами в секции действия и после проверяем результат выполнения метода в секции проверки. Секции в юнит-тесте носят рекомендательный характер. Их рекомендуется соблюдать (не обязательно помечать комментариями) для удобства чтения кода теста.

Поскольку мы лишь обозначили методы контроллеров, но оставили их без реализации, тест проверяет лишь то, что метод контроллера при вызове отдаст ответ, тип которого унаследован от IActionResult. Иначе говоря, наш первый тест проверяет, что метод можно вызвать и получить от него ожидаемый общий ответ.

## Запуск тестов

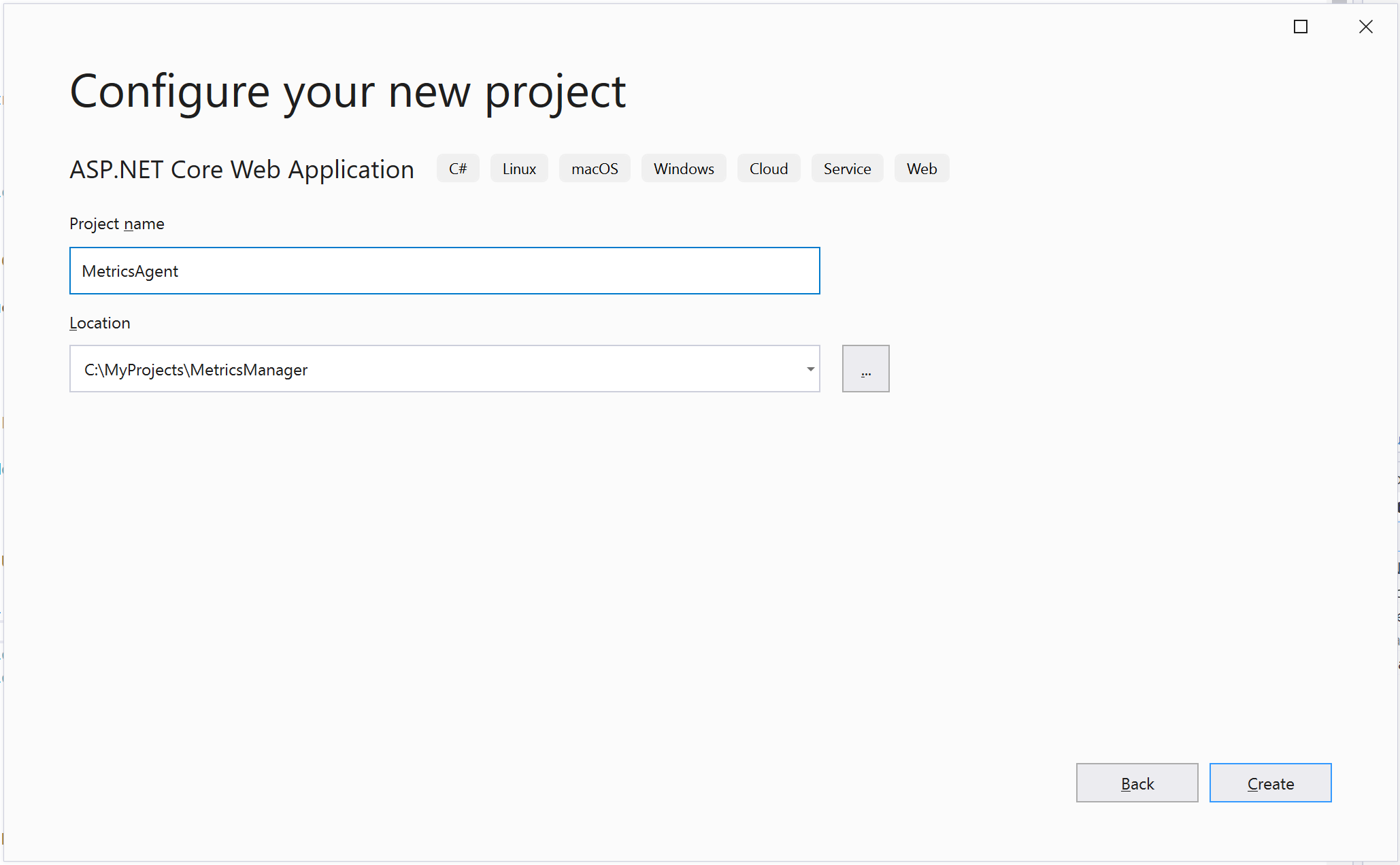
Тест можно запустить на исполнение. Для этого нажмите правой кнопкой мыши на проекте с тестам и выберете пункт RunTests.

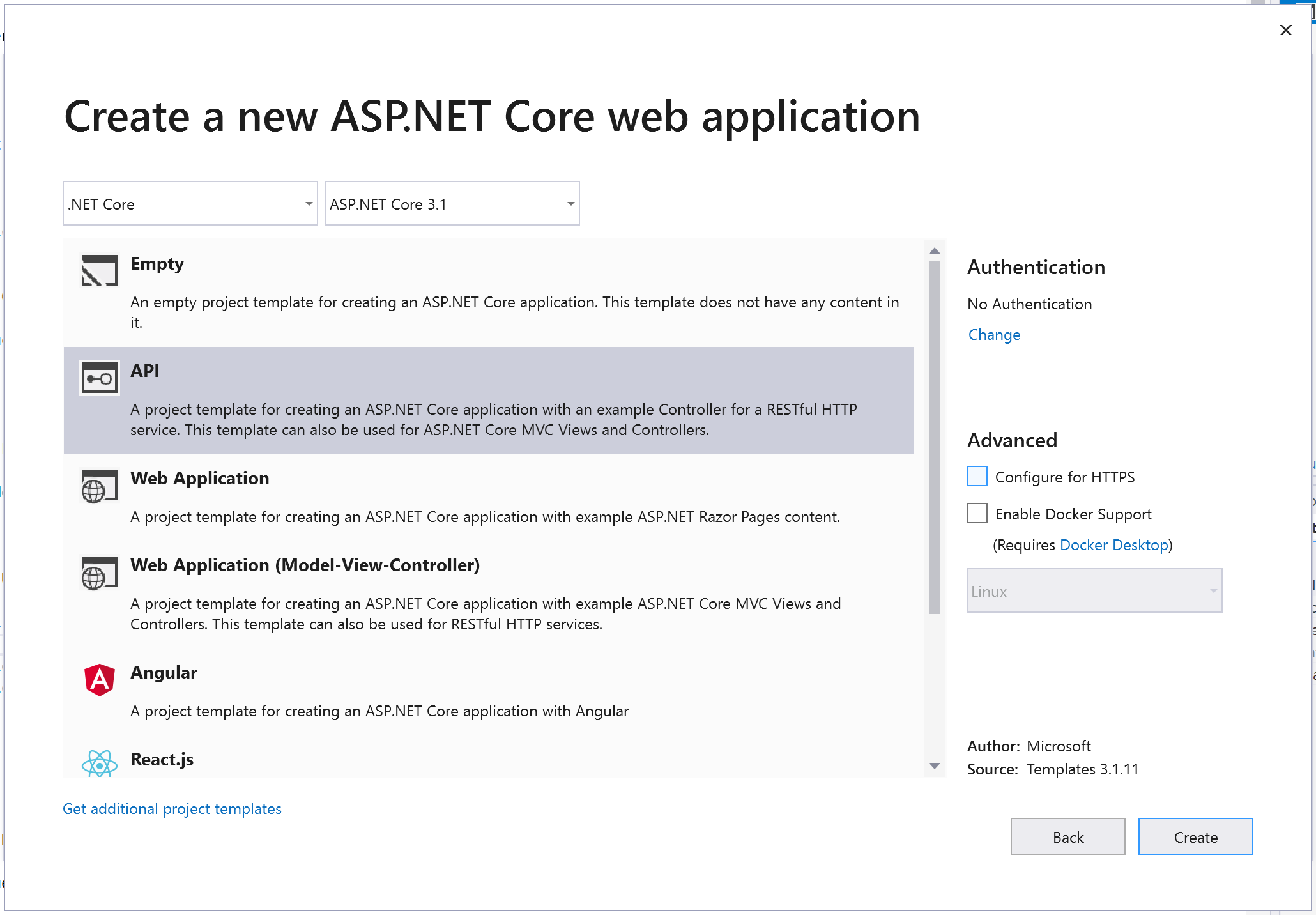
У вас появится окно запуска тестов. В этом окне будут перечислены все тесты, которые есть у вас в проекте. Перечисленные тесты можно запускать как все вместе, так и выбранные по отдельности. Если тест выполнен успешно (как у нас сейчас), то на нем будет висеть зеленый флажок, свидетельствующий о его успешности.



# Проект сервис-агента сбора метрик

Теперь нам нужно создать еще и проект, в котором мы будем собирать метрики, наш сервис-агент метрик. Он будет представлять из себя проект точно такого же типа как и сервис-менеджер сбора метрик и создается аналогично





# Практическое задание

Добавить метод в контроллер агентов проекта менеджера метрик позволяющий получить список зарегистрированных в системе объектов.

В проект агента сбора метрик добавить контроллеры для сбора метрик, аналогичные менеджеру сбора метрик. Добавить методы для получения метрик с агента, доступные по следующим путям

* api/metrics/cpu/agent/{agentId}/from/{fromTime}/to/{toTime}/percentiles/{percentile}
* api/metrics/cpu/agent/{agentId}/from/{fromTime}/to/{toTime}/
* api/metrics/dotnet/errors-count/agent/{agentId}/from/{fromTime}/to/{toTime}
* api/metrics/network/agent/{agentId}/from/{fromTime}/to/{toTime}/
* api/metrics/hdd/agent/{agentId}/left (размер оставшегося свободного дискового пространства в мегабайтах)
* api/metrics/ram/agent/{agentId}//available (размер свободной оперативной памяти в мегабайтах)

Добавить проект с тестами для агента сбора метрик. Написать простые юнит-тесты на каждый метод каждого контроллера в обоих тестовых проектах.

# Список используемой литературы

1. <https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>
2. <https://docs.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code>
3. <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics>
4. <https://xunit.net/#documentation>

# Дополнительные материалы

* <https://stackify.com/unit-testing-basics-best-practices/>
* <https://www.c-sharpcorner.com/article/a-basic-introduction-of-unit-test-for-beginners/>
* <https://dotnet.microsoft.com/apps/aspnet/apis>
* <https://www.freecodecamp.org/news/an-awesome-guide-on-how-to-build-restful-apis-with-asp-net-core-87b818123e28/>