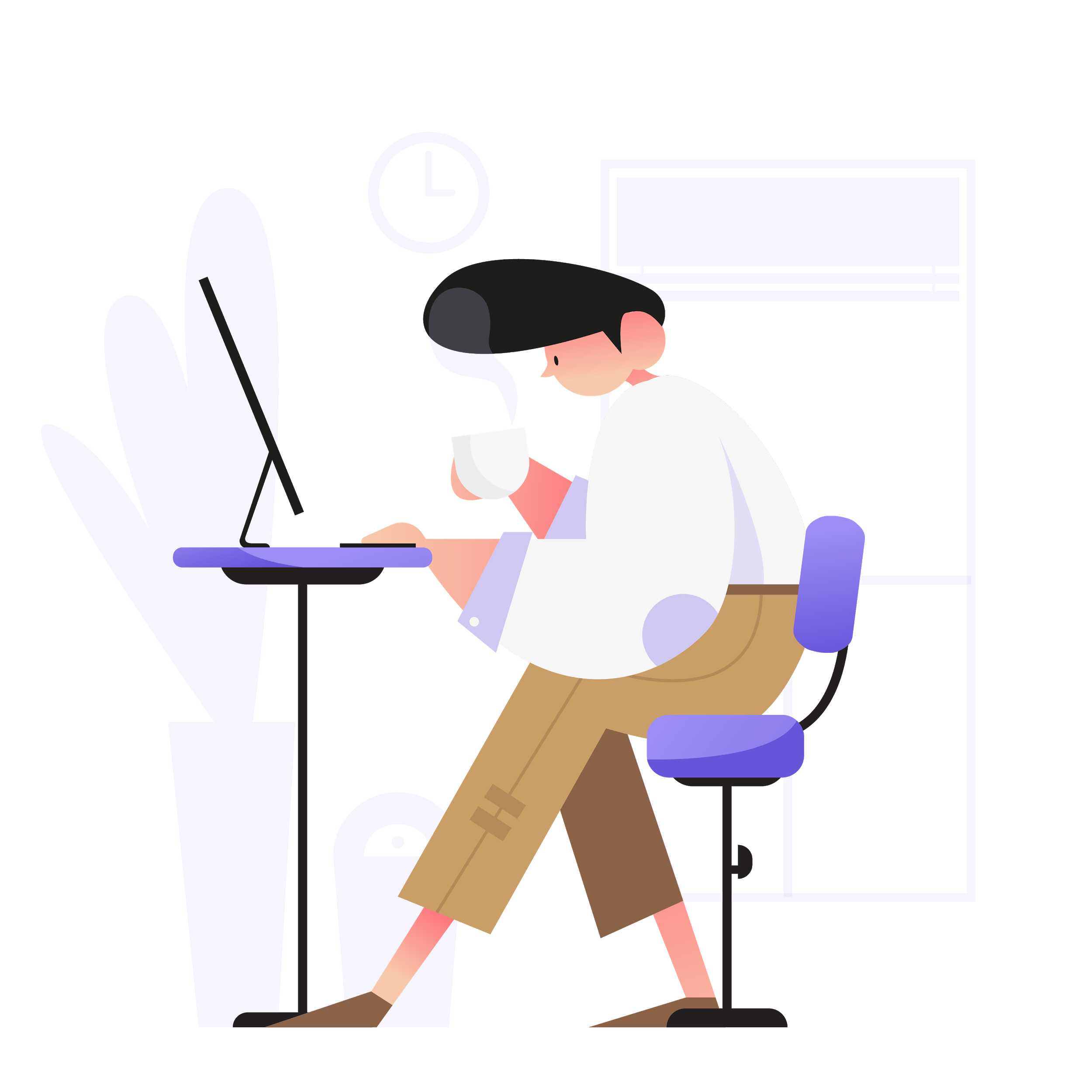
Asp.net core Web API микросервисы. Часть 1

Работа с базой данных



# На этом уроке

На этом уроке мы продолжим разработку системы сбора и менеджмента метрик. Мы познакомимся с менеджером пакетов nuget, который позволяет быстро подключать сторонние библиотеки в приложение, научимся правильно выстраивать зависимости между компонентами сервиса при помощи инверсии зависимостей. Зная принцип инверсии зависимостей, мы продолжим проектирование нашего сервиса в части работы с базой данных. Выделим основные абстракции для работы с базой данных, попробуем написать простые SQL запросы к базе данных и проверим их выполнение в консоли. Познакомимся с mock модульными тестами и напишем простые тесты для нашего слоя доступа к данным.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_n93g6yblf23c)

[Менеджер пакетов nuget](#_8hrsjs3wr9n1)

[Подключение пакетов к проектам](#_2cbkm7bjm9rl)

[Принцип инверсии зависимостей](#_qw35po1abjna)

[Настройка DI-контейнера приложения](#_raaioyrc8v53)

[Время жизни объекта в контейнере](#_rq3ruag8d0lt)

[Transient](#_o4lszgex333n)

[Scoped](#_qjlitr864kz7)

[Singleton](#_yq4n0g1p5x3k)

[Обобщения](#_j85uxej8quq7)

[Логирование](#_9ni4lw7b0fg1)

[Задачи логирования](#_avzhd4wbve1)

[Подключение библиотеки логирования к проекту](#_nebl3sfv8orc)

[Добавление логгера в контроллеры](#_xtj3x87r2ore)

[Синтаксис SQL запросов](#_df5ub4dktmc5)

[Создание таблицы](#_d9o8bl4lt0nv)

[Первичный ключ](#_viy9gkm36oe3)

[Вставка данных](#_kras4cxgcd88)

[Модификация данных](#_75h0q4imrx5j)

[Выборка данных из таблицы](#_fvp0rput1tdo)

[Удаление данных из таблицы](#_2dpmw8llczp6)

[Удаление таблицы](#_gzywbrhwyl7o)

[Проверка SQL запросов в консоли СУБД](#_maqn3l7tjoue)

[Работа с SQLfiddle](#_85rrz1rhv1t7)

[Подключение к базе данных](#_4csc18gotdym)

[Подключение SQLite к агенту метрик](#_wcwaqzw8ja8p)

[Подключение к базе данных](#_6xc1j2raf2yl)

[Интерфейс IDbConnection](#_hfj94cl9z1k9)

[Интерфейс IDbCommand](#_wvxur13kly79)

[Интерфейс IDataReader](#_lywbz06t6tr0)

[Интерфейсы репозиториев приложения](#_665znf4swwb)

[Паттерн Репозиторий](#_1tnp7tp9ia3k)

[Mock тестировние](#_x797q1j5ebxz)

[Задание для самостоятельной работы](#_jp9o3kv3gohn)

[Список дополнительной литературы](#_52q21seic6r3)

# Вступление

В прошлом уроке мы рассмотрели, как важно проектировать API сервисов в REST стиле и обозначили важность модульного тестирования. Мы спроектировали контроллеры для менеджера метрик и сервиса-агента сбора метрик. На их примере познакомились с атрибутами маркировки параметров для сборщика модели запроса. И научились писать и запускать модульные тесты. В данном уроке мы расширим функциональность наших сервисов, подключив к ним сторонние библиотеки при помощи менеджера пакетов nuget, рассмотрим принцип инверсии зависимостей и DI-контейнер, встроенный в asp.net core фреймворк. Мы настроим зависимости между компонентами сервиса в DI-контейнере. Также мы спроектируем слой для работы с базой данных, выделив его интерфейс, узнаем что такое ORM фреймворк, попробуем написать запросы в базу данных на языке SQL и проверим их выполнимость прежде, чем встроить в реализацию интерфейсов доступа к данным в базе данных. Дополнительно мы изолированно протестируем весь написанный нами на уроке код. Итого на этом уроке вы узнаете, что такое менеджер пакетов, как подключать сторонние библиотеки, как проектировать слой доступа к данным и изолированно тестировать написанный код.

# Менеджер пакетов nuget

Важным инструментом для любой современной платформы разработки является механизм, с помощью которого разработчики могут создавать полезный код, делиться им и использовать его. Часто такой код объединяется в «пакеты», которые содержат скомпилированный код (как библиотеки DLL) вместе с другим содержимым, необходимым для проектов, использующих эти пакеты.

Для asp.net core поддерживаемый Microsoft механизм совместного использования кода - это NuGet, который определяет, как создаются, размещаются и используются пакеты, и предоставляет инструменты для этого.

Пакет NuGet - это один ZIP-файл с расширением .nupkg, который содержит скомпилированный код (DLL), другие файлы, связанные с этим кодом, и описательный манифест, который включает такую ​​информацию, как номер версии пакета. Разработчики с кодом для совместного использования создают пакеты и публикуют их на общедоступном или частном хосте. Потребители пакетов получают эти пакеты с подходящих хостов, добавляют их в свои проекты, а затем вызывают функциональные возможности пакета в своем коде проекта.

* NuGet предоставляет центральный репозиторий nuget.org с поддержкой частного хостинга.
* NuGet предоставляет разработчикам инструменты, необходимые для создания, публикации и использования пакетов.
* Что наиболее важно, NuGet поддерживает справочный список пакетов, используемых в проекте, и возможность восстанавливать и обновлять эти пакеты из этого списка.

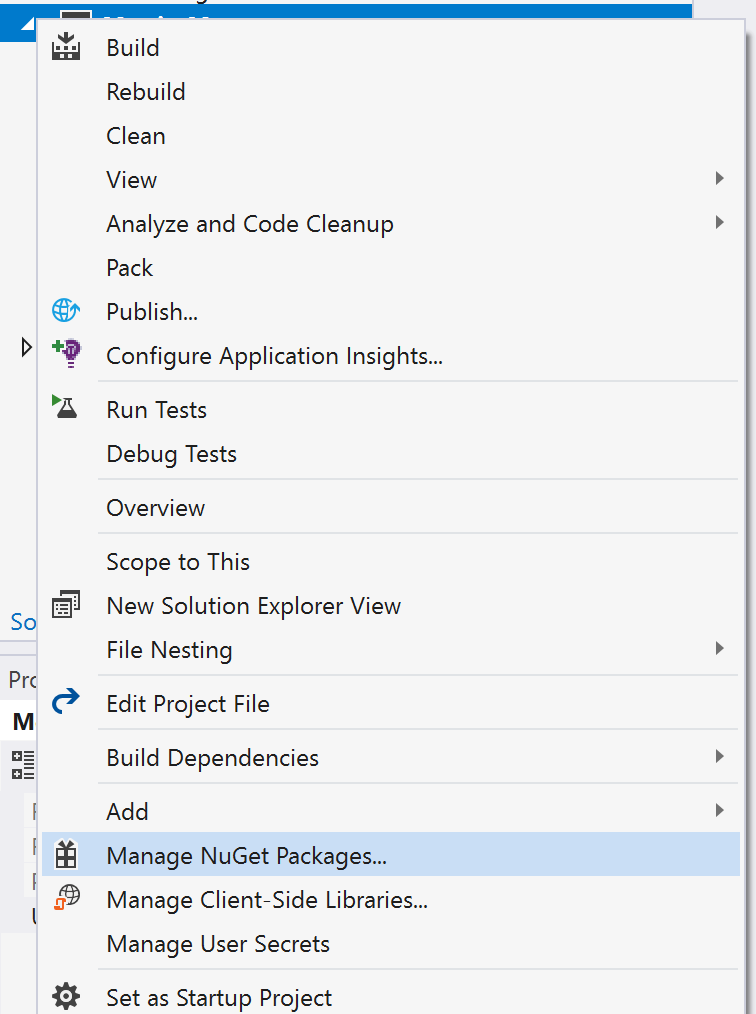
Чтобы эти процессы работали эффективно, NuGet выполняет некоторые скрытые оптимизации. В частности, NuGet управляет кешем пакетов и глобальной папкой пакетов для быстрой установки и переустановки. Кэш позволяет избежать загрузки пакета, который уже был установлен на машине. Папка глобальных пакетов позволяет нескольким проектам совместно использовать один и тот же установленный пакет, тем самым уменьшая общую площадь, занимаемую NuGet на компьютере. Папка кэша и глобальных пакетов также очень полезна, когда вы часто восстанавливаете большое количество пакетов, например, на сервере сборки.

Довольно часто проект зависит от одного или нескольких пакетов, которые сами имеют одинаковые зависимости. Некоторые из наиболее полезных пакетов служебных программ на nuget.org используются во многих других пакетах. Таким образом, во всем графе зависимостей вы легко можете иметь десять разных ссылок на разные версии одного и того же пакета. Чтобы избежать добавления нескольких версий этого пакета в само приложение, NuGet выбирает, какая единственная версия может использоваться всеми потребителями.

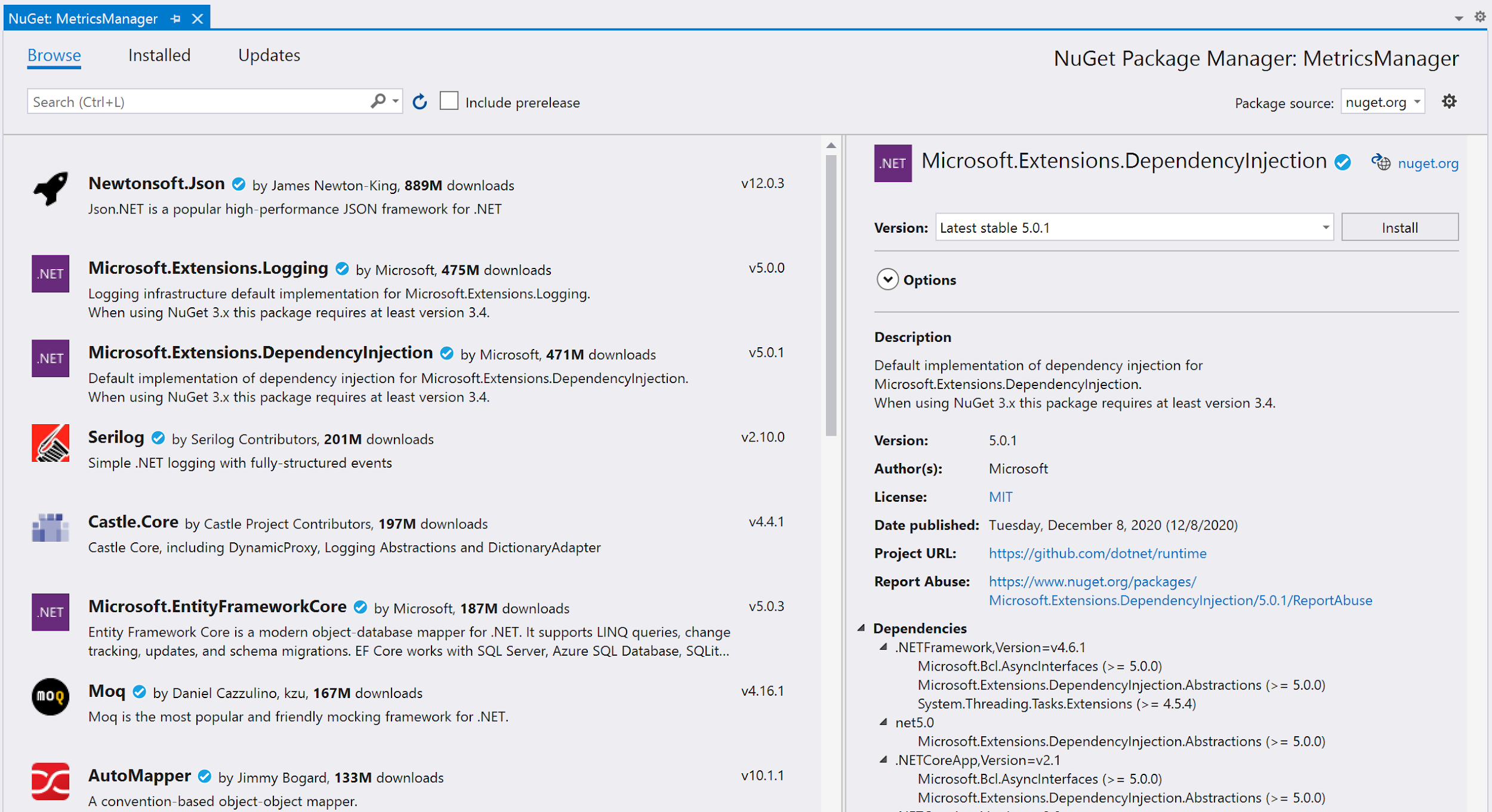
Таким образом, NuGet является мощным средством для расширения функциональности вашего приложения за счет сторонних библиотек, а также предоставляет инструменты для управления этими зависимостями. Давайте подключим к нашему проекту сбора метрик некоторые библиотеки.

# Подключение пакетов к проектам

Чтобы подключить новый пакет к приложению, щелкните правой кнопкой мыши на проекте менеджера метрик и выберете пункт Manage Nuget Packages, как показано на рисунке:



В появившемся окне перейдите на вкладку Browse. Перед вами появится список популярных библиотек, которые наиболее часто нужны разработчику. Давайте подключим в наш проект менеджера метрик DI-контейнер при помощи пакета Microsoft.Extensions.DependencyInjection.



Нажмите кнопку Install. Всё, теперь мы добавили наш первый пакет в зависимости к нашему проекту сервиса сбора метрик.

Хотя данный пакет и является встроенным в asp.net core приложение, мы рассмтрели пример установки пакета именно на нем, так как дальше мы будем рассматривать работу кода именно этого пакета.

Прежде чем идти дальше, к объяснению работы добавленного пакета, нам необходимо познакомиться с принципом инверсии зависимостей.

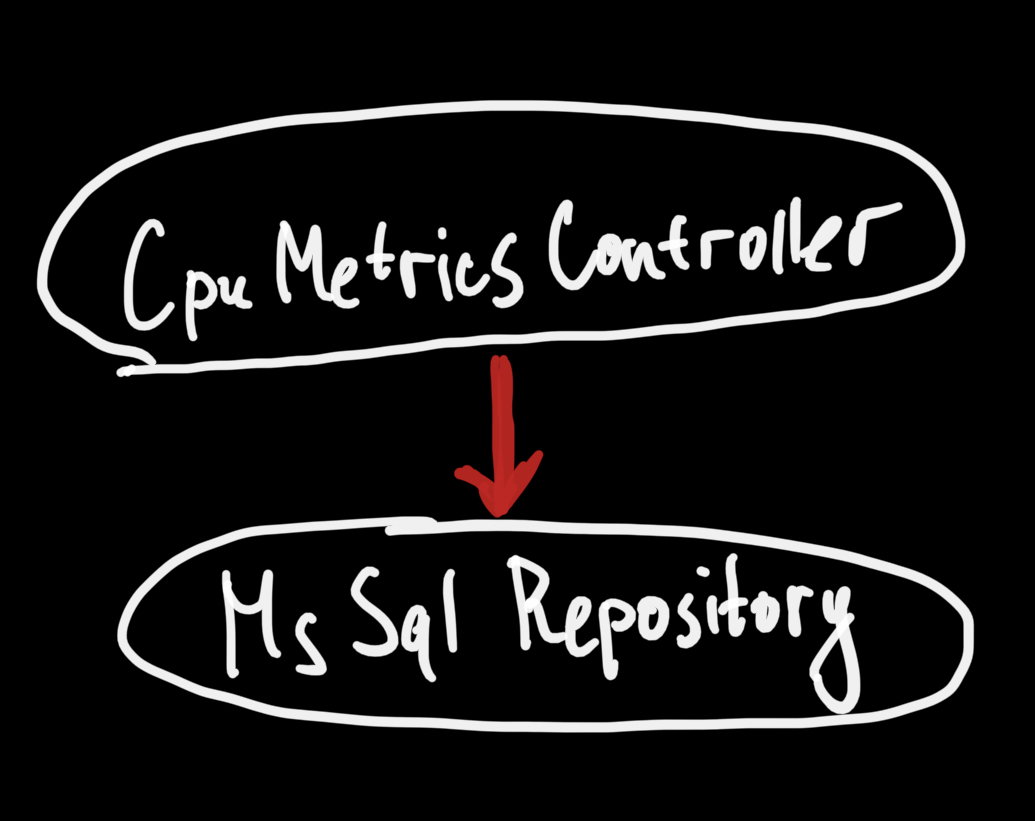
# Принцип инверсии зависимостей

Принцип инверсии зависимостей – важный принцип объектно-ориентированного программирования, используемый для уменьшения связанности в компьютерных программах. Входит в пятерку принципов SOLID. Формулируется он довольно просто:

* Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

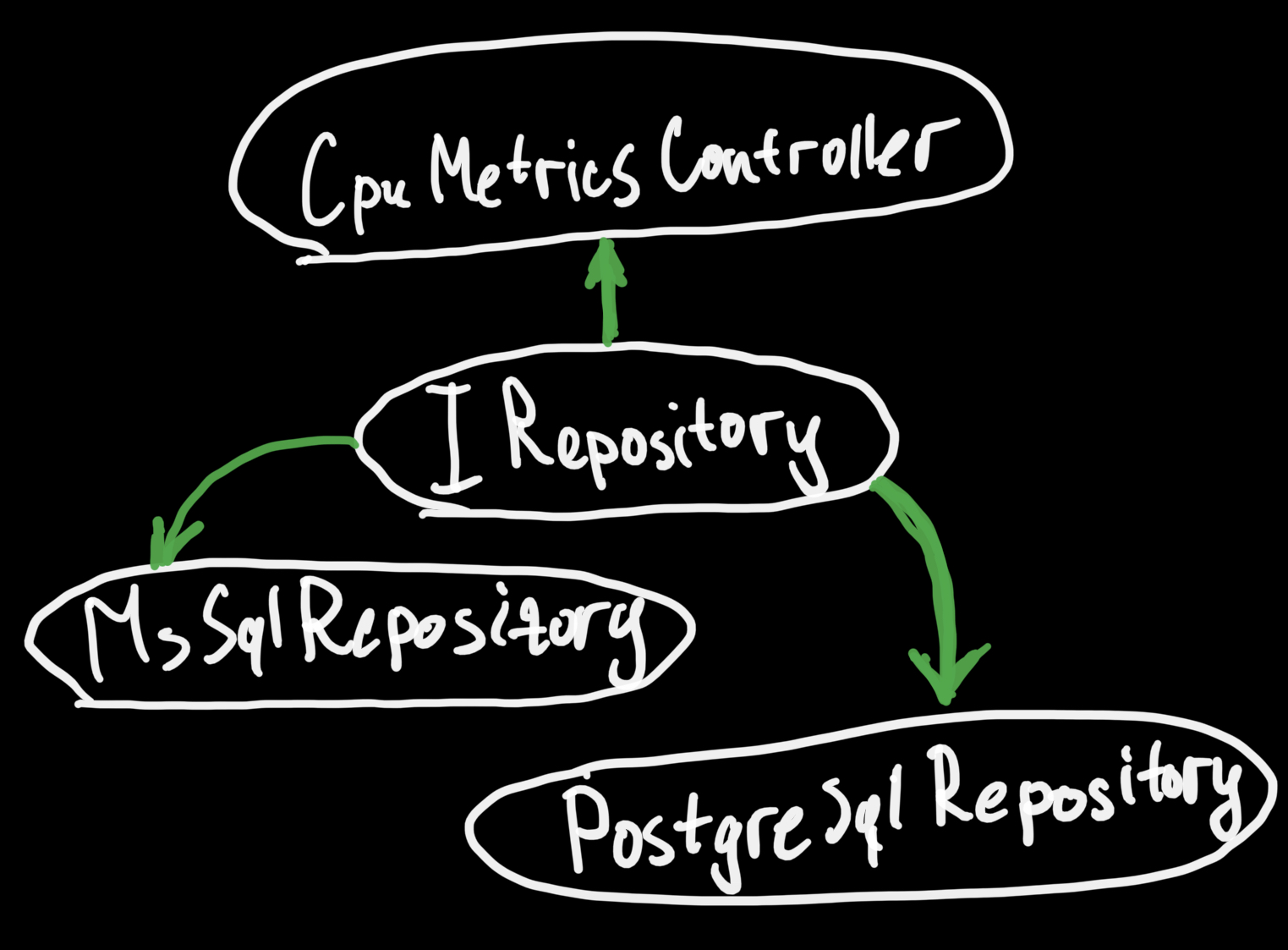
Модуль — логически взаимосвязанная совокупность функциональных элементов. Для простоты понимания в нашем случае, будем иметь в виду, что модулем в нашем случае будет являться некий открытый интерфейс в коде нашего приложения.

Теперь представим что у нас имеется ряд модулей в системе, которые зависят друг от друга, от конкретных реализаций. Например как в нашем случае, представим, что контроллер сбора CPU метрик напрямую зависит от класса, в котором у нас описана логика работы с базой данных. Представим, что программист написал специальный класс для работы только с СУБД MsSQL и внедрил его в контроллер сбора метрик, но руководитель проекта, увидев цену на лицензию на данную СУБД решил, что нам подойдет бесплатная версия PostgreSQL и решил заменить написанный код на другой.



Поскольку у нас имеется явная зависимость от конкретной реализации, нам придется полностью переписать как контроллер, так и весь репозиторий. Когда проекты становятся большими, и часто изменяются под влиянием бизнес-требований, такой подход обречен на затягивание сроков разработки в виду экспоненциально увеличивающегося времени на переписывание старых кусочков кода,

Что же можно сделать для улучшения ситуации? Инвертировать зависимости! Давайте теперь сделаем так, чтобы стрелочка была развернута, то есть и контроллер и репозиторий зависели от некоторого интерфейса, который предоставляет возможность контроллеру зависеть от некоторой абстракции управления данными, а репозиторий обязуется соблюдать этот контракт в своей реализации.



Таким образом мы как-бы развернули (инвертировали) направление зависимостей. Контроллер зависит от интерфейса репозитория СУБД, так же, как и конкретная реализация для СУБД MsSQL. Теперь, для того, чтобы перейти на новую СУБД, программистам не придется переписывать контроллер, а лишь заменить одну реализацию (MsSqlRepository) репозитория на другую (PostgreSqlRepository).

Интерфейсы и позднее связывание (когда сопоставление вызываемого кода к интерфейсу происходит во время выполнения приложения) позволяют отделить реализацию логики от ненужных деталей. Необходимо стараться делать интерфейсы как можно более изолированными. Когда все интерфейсы независимы, получается возможность независимо их модифицировать.

# Настройка DI-контейнера приложения

В asp.net core за инверсию и внедрение зависимостей между модулями приложения отвечает встроенный DI контейнер. Для того, чтобы зарегистрировать интерфейс и соответствующую ему реализацию, необходимо добавить код регистрации в метод ConfigureServices класса Startup. Программисту лишь необходимо зарегистрировать все интерфейсы с реализациями в контейнере, а уже сам контейнер во время исполнения приложения построит граф зависимостей между модулями приложения и сам создаст нужные объекты, на интерфейсы которых ссылаются конструкторы приложения. Важно зарегистрировать интерфейс и его реализацию в контейнере с нужным временем жизни создаваемого объекта (object lifetime).

## Время жизни объекта в контейнере

Сервисы могут быть зарегистрированы в контейнере с одним из следующих времен жизни:

* Transient
* Scoped
* Singleton

Необходимо правильно выбирать время жизни объекта зарегистрированного в контейнере. Для этого рассмотрим подробнее каждое из них.

### Transient

Transient объекты зарегистрированные в контейнере создаются каждый раз заново когда их запрашивают. Например, в нашем случае, таким контейнером исполнения будет запрос, пришедший в приложение и каждый раз, когда у DI-контейнера спросят объект этого интерфейса.. После выполнения запроса, такие объекты уничтожаются. Резюмируя, можно сделать вывод, что данное время жизни объекта в контейнере лучше всего подходит для простых легковесных объектов, так как каждый раз при запросе объекта из контейнера с данным временем жизни мы будем получать новый объект.

### Scoped

Данное время жизни объекта означает, что объект будет создан однажды во время получения клиентского запроса в приложении и дальше не будет изменяться на протяжении всего времени обработки запроса. Даже если у контейнера запросить заново объект, если запрос произошел в рамках одного и того же клиентского запроса, мы будем иметь дело с физически тем же самым объектом.

### Singleton

Или же одиночка, создается ровно один раз, когда код приложения запрашивает объект из контейнера и остается в памяти приложения на протяжении всего срока его работы. Уничтожается такой объект только вместе с приложением. Любой код, который запросит объект с таким временем жизни, будет иметь дело с ровно одним, одним на все вызовы в рамках всего времени жизни приложения, объектом. Объекты с таким временем жизни в контейнере обязаны быть потоко-безопасными по понятным причинам.

# Обобщения

Прежде чем мы будем рассматривать дальше различные интерфейсы, необходимо вспомнить про механизм обобщений в C# (джерерики generics).

Язык C# поддерживает обобщенные классы и методы. Чтобы понять, что такое обобщенные методы и классы и зачем они понадобились, давайте рассмотрим на примере, как жилось без них.

Напишем простой класс Person

|  |
| --- |
| public class Person {  public int Id { get; set; }  public int Age { get; set; } } |

Класс Person определяет два свойства: Id - уникальный идентификатор и Age - возраст персоны.

Здесь идентификатор задан как числовое значение. Однако также нередко для идентификатора используются и строковые значения (например, им может выступить имя человека). И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство Id как свойство типа object. Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

|  |
| --- |
| public class Person {  public object Id { get; set; }  public int Age { get; set; } } |

Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как упаковка (boxing) и распаковка (unboxing).

Так, при присвоении свойству Id значения типа int, происходит упаковка этого значения в тип Object.

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

|  |
| --- |
| var person = new Person { Age = 42 }; person.Id = "1"; int id2 = (int)person.Id; *// Исключение InvalidCastException* |

Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Эти проблемы были призваны устранить обобщенные типы (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Перепишем класс Person с использованием обобщений

|  |
| --- |
| public class Person<T> {  public T Id { get; set; }  public int Age { get; set; } } |

В угловых скобках прописан обобщенный параметр T. T -- это просто принятое по соглашению кода название для обобщенного типа, но можно использовать любое другое удобное название.

Теперь, если мы захотим, чтобы у нас был создан объект класса Person с типом обобщения int, то при работе с полем Id не будет происходить лишней упаковки и распаковки типа

|  |
| --- |
| Person<int> person1 = new Person<int> { Age = 42 }; Person<string> person2 = new Person<string> { Age = 30 }; person1.Id = 2; *// упаковка не нужна* person2.Id = "4356"; int id1 = person1.Id; *// распаковка не нужна* string id2 = person2.Id; Console.WriteLine(id1); Console.WriteLine(id2); |

Таким образом, можно сказать, что обобщения это некоторый чертеж для типов в классе или методе. Мы просто говорим компилятору, что программист, который будет использовать наш код, подставит вместо указанного нами обобщенного типа – свой конкретный тип и будет им пользоваться.

Для обобщений в методах принцип работы аналогичен.

# Логирование

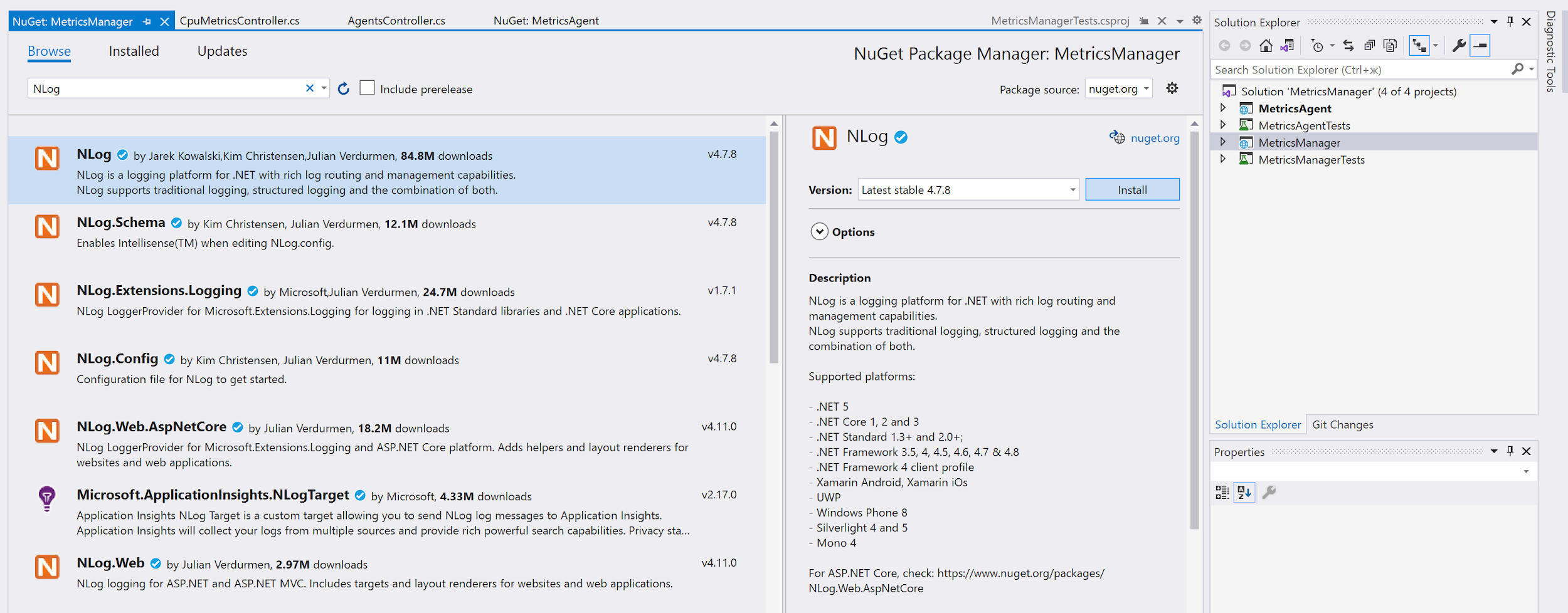
Важной задачей в работе приложений является сбор различной информации о том, как приложение себя ведет во время выполнения. Для этого используются логи. Логи (лог-файлы) — это файлы, содержащие системную информацию работы сервера или компьютера, в которые вносятся определенные действия пользователя или программы. Иногда также употребляется русскоязычный аналог понятия — журнал.

## Задачи логирования

Протоколирование операций, выполняемых на машине, для дальнейшего анализа администратором и есть основная задача логирования. Регулярный просмотр журналов позволит определить ошибки в работе системы в целом, конкретного сервиса или сайта (особенно скрытые ошибки, которые не выводятся при просмотре в браузере), диагностировать злонамеренную активность, собрать статистику посещений сайта.

## Подключение библиотеки логирования к проекту

Рассмотрим подключение библиотеки логирования NLog через nuget.



Откройте менеджер пакетов nuget и установите библиотеку в проект менеджера метрик. Так же установите пакет NLog.Web.AspNetCore

Теперь займемся конфигурированием библиотеки. Для этого добавьте файл nlog.config в корень проекта. Это xml файл с конфигурацией, вставьте в него следующие строки

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?> <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  autoReload="true"  throwConfigExceptions="true"  internalLogLevel="info"  internalLogFile="c:\temp\internal-nlog-AspNetCore3.txt">   *<!-- enable asp.net core layout renderers -->*  <extensions>  <add assembly="NLog.Web.AspNetCore"/>  </extensions>   *<!-- the targets to write to -->*  <targets>  *<!-- File Target for all log messages with basic details -->*  <target xsi:type="File" name="allfile" fileName="c:\temp\nlog-AspNetCore3-all-${shortdate}.log"  layout="${longdate}|${event-properties:item=EventId\_Id:whenEmpty=0}|${uppercase:${level}}|${logger}|${message} ${exception:format=tostring}" />   *<!-- File Target for own log messages with extra web details using some ASP.NET core renderers -->*  <target xsi:type="File" name="ownFile-web" fileName="c:\temp\nlog-AspNetCore3-own-${shortdate}.log"  layout="${longdate}|${event-properties:item=EventId\_Id:whenEmpty=0}|${uppercase:${level}}|${logger}|${message} ${exception:format=tostring}|url: ${aspnet-request-url}|action: ${aspnet-mvc-action}|" />   *<!--Console Target for hosting lifetime messages to improve Docker / Visual Studio startup detection -->*  <target xsi:type="Console" name="lifetimeConsole" layout="${level:truncate=4:tolower=true}\: ${logger}[0]${newline} ${message}${exception:format=tostring}" />  </targets>   *<!-- rules to map from logger name to target -->*  <rules>  *<!--All logs, including from Microsoft-->*  <logger name="\*" minlevel="Trace" writeTo="allfile" />   *<!--Output hosting lifetime messages to console target for faster startup detection -->*  <logger name="Microsoft.Hosting.Lifetime" minlevel="Info" writeTo="lifetimeConsole, ownFile-web" final="true" />   *<!--Skip non-critical Microsoft logs and so log only own logs-->*  <logger name="Microsoft.\*" maxlevel="Info" final="true" /> *<!-- BlackHole -->*   <logger name="\*" minlevel="Trace" writeTo="ownFile-web" />  </rules> </nlog> |

Подробнее о том, что именно делают данные параметры конфигурации лучше прочитать в документации (<https://github.com/NLog/NLog/wiki/Configuration-file>) мы же сконцентрируемся на возможности подключения библиотеки к нашему проекту.

Теперь необходимо сделать так, чтобы созданный файл копировался в папку вместе со сборкой нашего приложения. Для этого в файле .csproj внесите следующую секцию

|  |
| --- |
| <ItemGroup>  <Content Update="nlog.config" CopyToOutputDirectory="PreserveNewest" />  </ItemGroup> |

Далее модифицируйте файл Program.cs следующим образом

|  |
| --- |
| using System; using NLog.Web; using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.Logging; using Microsoft.Extensions.Hosting;  namespace MetricsManager {  public class Program  {  public static void Main(string[] args)  {  var logger = NLogBuilder.ConfigureNLog("nlog.config").GetCurrentClassLogger();  try  {  logger.Debug("init main");  CreateHostBuilder(args).Build().Run();  }  *// отлов всех исключений в рамках работы приложения*  catch (Exception exception)  {  *//NLog: устанавливаем отлов исключений*  logger.Error(exception, "Stopped program because of exception");  throw;  }  finally  {  *// остановка логера*   NLog.LogManager.Shutdown();  }   CreateHostBuilder(args).Build().Run();  }   public static IHostBuilder CreateHostBuilder(string[] args) =>  Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  })  .ConfigureLogging(logging =>  {  logging.ClearProviders(); *// создание провайдеров логирования*  logging.SetMinimumLevel(LogLevel.Trace); *// устанавливаем минимальный уровень логирования*  }).UseNLog(); *// добавляем библиотеку nlog*  } } |

Метод расширение ConfigureLogging добавит в DI контейнер все необходимое для регистрации логгера. После этой регистрации можно будет воспользоваться интерфейсом ILogger<T> в контроллере. Логгер добавится с временем жизни Singlton.

Осталось только настроить логер в файле конфигурации appsettings.json

|  |
| --- |
| {  "Logging": {  "IncludeScopes": false,  "LogLevel": {  "Default": "Trace",  "Microsoft": "Warning",  "Microsoft.Hosting.Lifetime": "Information"  }  },  "AllowedHosts": "\*" } |

Осталось добавить возможность воспользоваться логгером непосредственно в контроллере. Давайте сделаем это в созданном нами контроллере сбора метрик CPU.

## Добавление логгера в контроллеры

Модифицируйте файл контроллера, добавив в него конструктор и приватную переменную с логгером.

|  |
| --- |
| using MetricsManager.Enums; using Microsoft.AspNetCore.Mvc; using System; using Microsoft.Extensions.Logging;  namespace MetricsManager.Controllers {  [Route("api/metrics/cpu")]  [ApiController]  public class CpuMetricsController : ControllerBase  {  private readonly ILogger<CpuMetricsController> \_logger;   public CpuMetricsController(ILogger<CpuMetricsController> logger)  {  \_logger = logger;  \_logger.LogDebug(1, "NLog встроен в CpuMetricsController");  }   [HttpGet("agent/{agentId}/from/{fromTime}/to/{toTime}")]  public IActionResult GetMetricsFromAgent([FromRoute] int agentId, [FromRoute] TimeSpan fromTime, [FromRoute] TimeSpan toTime)  {  \_logger.LogInformation("Привет! Это наше первое сообщение в лог");  return Ok();  }  }  } |

Теперь при выполнении кода в отладчике вы сможете увидеть сформированный файл логов непосредственно рядом со сборкой приложения. Попробуйте открыть его при помощи блокнота и вы увидите сообщения, которые были сформированы вами. Попробуйте вставить в лог входные параметры метода, тем самым “залогировав” их.

# Синтаксис SQL запросов

SQL или Structured Query Language (язык структурированных запросов) — язык программирования, предназначенный для управления данными в СУБД (системе управления базами данных). Все современные СУБД поддерживают SQL. На языке SQL выражаются все действия, которые можно провести с данными: от записи и чтения данных, до администрирования самого сервера СУБД.

На языке SQL выражаются все действия, которые можно провести с данными: от записи и чтения данных, до администрирования самого сервера СУБД.

Для повседневной работы совсем не обязательно знать весь этот язык; достаточно ознакомиться лишь с основными понятиями синтаксиса и ключевыми словами. Давайте познакомимся с основными операциями на языке SQL. Они понадобятся нам при работе с базой данных.

Мы будем рассматривать SQL запросы к реляционным базам данных, данные в которых организованы в виде таблиц. Реляционным, значит то, что таблицы в базе данных имеют возможность ссылаться друг на друга.

Приступим к практике, ведь именно на ней проще и быстрее всего понять как работает этот язык.

### Создание таблицы

Для примера, напишем запрос по созданию таблицы, в которой будут храниться метрики загрузки CPU в процентах в определенные моменты времени и разберем его структуру

|  |
| --- |
| CREATE TABLE cpumetrics(id INTEGER PRIMARY KEY,  value INT, time INT); |

CREATE TABLE означает что мы создаем таблицу в базе данных, затем следует название таблицы, которую мы создаем. В скобках следом идет перечисление колонок через запятую с названием каждой колонки и типом данных в ней. Особое внимание следует обратить на первую колонку, которая является первичным ключом в таблице.

### Первичный ключ

В примере с созданием новой таблицы при перечислении необходимых полей первым полем идёт id INTEGER PRIMARY KEY.

Это поле называется первичным ключом. Обязательно создавать первичный ключ в каждой таблице.

Первичный ключ — это особенное поле, в котором сохраняется уникальный идентификатор записи. Он нужен, чтобы у программиста и базы данных всегда была возможность однозначно обратиться к одной конкретной записи для её чтения, обновления или удаления.

Если назначить поле первичным ключом, то БД будет следить за тем, чтобы значение в этом поле больше не повторялось в таблице.

### Вставка данных

Рассмотрим запрос на вставку данных в созданную таблицу

|  |
| --- |
| INSERT INTO cpumetrics(value, time) VALUES(50,1231231232); |

INSERT INTO говорит о том, что мы собираемся вставлять данные, следом идет название таблицы, в которую будут вставляться данные. Данные добавятся в конец таблицы в виде новой строки записи. В скобках указано, в каком порядке в какие колонки будет происходить вставка данных. VALUES говорит о том, что следом идет перечисление значений для вставки в таблицу, ровно в том же порядке, в котором перечислены они в предыдущей секции скобок. Обратите внимание, что мы не вставляли и не указывали явно вставку идентификатора. Дело в том, что большинство СУБД не требуют указания первичного ключа при вставке данных в таблицу (например, если идентификатор записи автоматически увеличивается на единицу при каждой новой вставке). СУБД сама выберет первичный ключ для новой записи в таблицу.

### Модификация данных

Модификация данных в таблице происходит при помощи оператора UPDATE. Рассмотрим его на примере

|  |
| --- |
| UPDATE cpumetrics SET value = 42, time = 123 WHERE id=1; |

После ключевого слова UPDATE идет указание таблицы, в котрой мы собираемся изменить данные. Следом, после оператора SET идет перечисление колонок и их новых значений. Не обязательно указывать все колонки для обновления, а можно обновить значение в какой то одной колонке. После этого идет оператор WHERE, в рамках которого задается условие, согласно которому СУБД найдет записи, подходящие под условия поиска и обновит в них значения согласно ранее указанным значениям. У нас указано, что необходимо обновить запись с идентификатором 1, но можно, например, составить запрос на обновление целого ряда ячеек.

### Выборка данных из таблицы

Выборка данных из таблицы происходит при помощи оператора SELECT. Рассмотрим на примере.

|  |
| --- |
| SELECT id, value, time FROM cpumetrics WHERE value>50; SELECT \* FROM cpumetrics; |

В первом примере после оператора SELECT идет перечисление колонок, которые мы хотим выбрать. Хотя в данном примере перечислены все колонки, которые имеются в нашей таблице, при необходимости мы можем не указывать все колонки, а делать выборку только из нужных. После идет оператор FROM в котором указывается из какой именно таблицы в базе данных мы хотим совершить выборку данных. Затем идет условие WHERE, которое является не обязательным и задает условия, по которым будет происходить отбор строк в таблице для выборки данных. В нашем примере указано, что строка попадет в выборку только если значение value строго превысит значение 50.

Во втором примере вместо перечисления колонок мы указали оператор \*. Это означает, что мы хотим выбрать все колонки из таблицы. Также мы не указали оператор WHERE с условием, а потому, из таблицы достанутся абсолютно все данные, которые в ней хранятся.

### Удаление данных из таблицы

Удаление данных из таблицы происходит при помощи оператора DELETE

|  |
| --- |
| DELETE FROM cpumetrics WHERE value < 50; |

Тут все очень просто, DELETE FROM, затем указывается имя таблицы, из которой мы хотим удалить данные, а после указывается условие, по которой будут удалены строки из таблицы. Условие указывается по колонке, в нашем примере, мы удалим все строки, в которых значение колонки value не превышает 50.

### Удаление таблицы

Созданную таблицу можно удалить вместе со всеми данными. Делается это следующим образом:

|  |
| --- |
| DROP TABLE cpumetrics; |

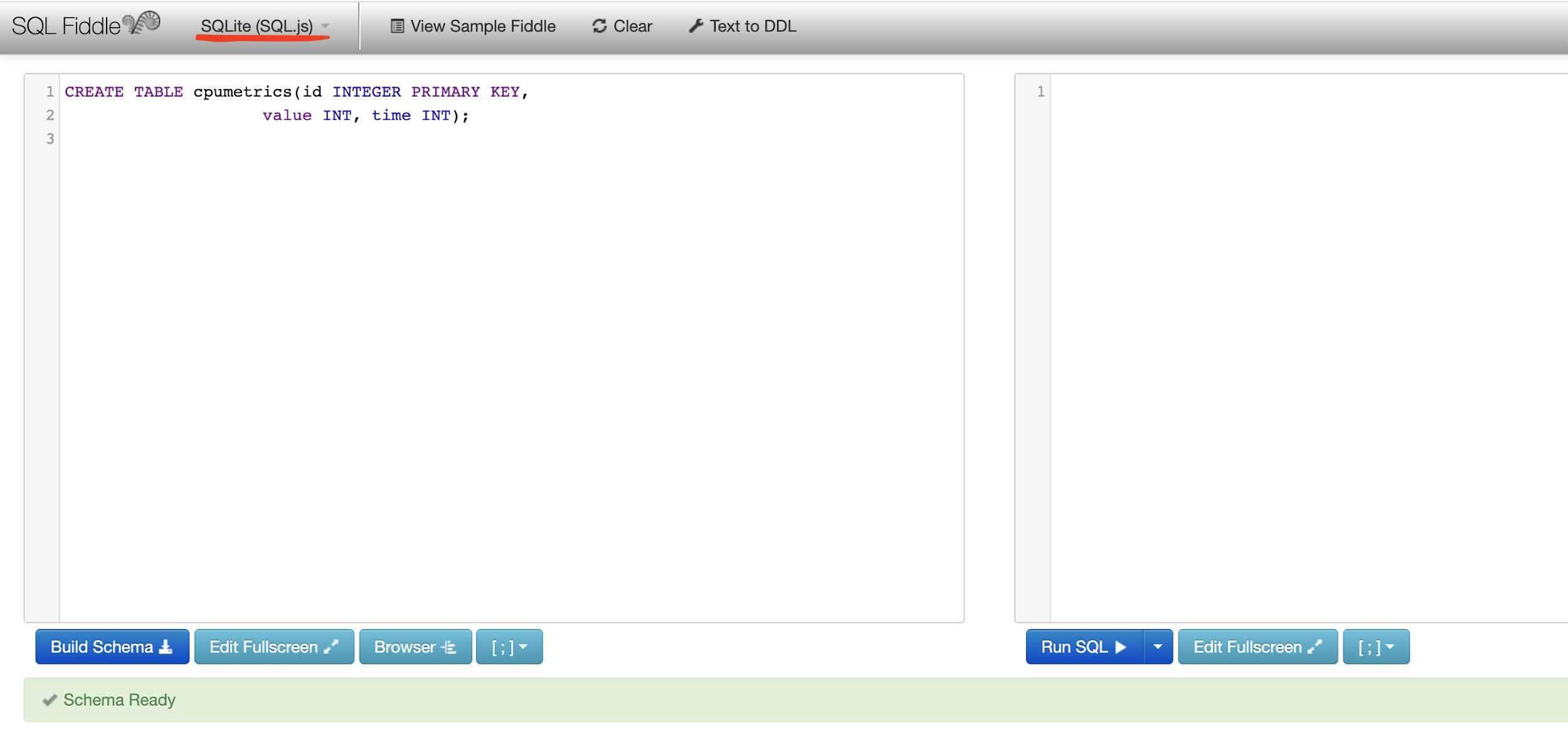
DROP TABLE и имя таблицы, которую мы хотим удалить. Будьте осторожны с оператором DROP, так как данные удаляются безвозвратно.

## Проверка SQL запросов в консоли СУБД

Мы можем проверить работу запросов и вообще попробовать составлять свои запросы к базе данных. Для этого удобно пользоваться консолью базы данных. К сожалению, установка отдельной базы данных на свой компьютер довольно сложный процесс, но есть специализированный сайт, на котором можно выбрать интересующую СУБД и попробовать выполнить запросы прямо в ней. Давайте попробуем создать несколько таблиц при помощи этого инструмента и выполнить запросы, которые мы рассмотрели выше.

## Работа с SQLfiddle

Для того, чтобы опробовать SQL запросы без установки СУБД на свой компьютер, можно воспользоваться бесплатным сайтом для отладки SQL запросов SQLFidle. Зайдите на сайт <http://sqlfiddle.com/> и выберете там диалект SQL для СУБД SQLite как показано на рисунке

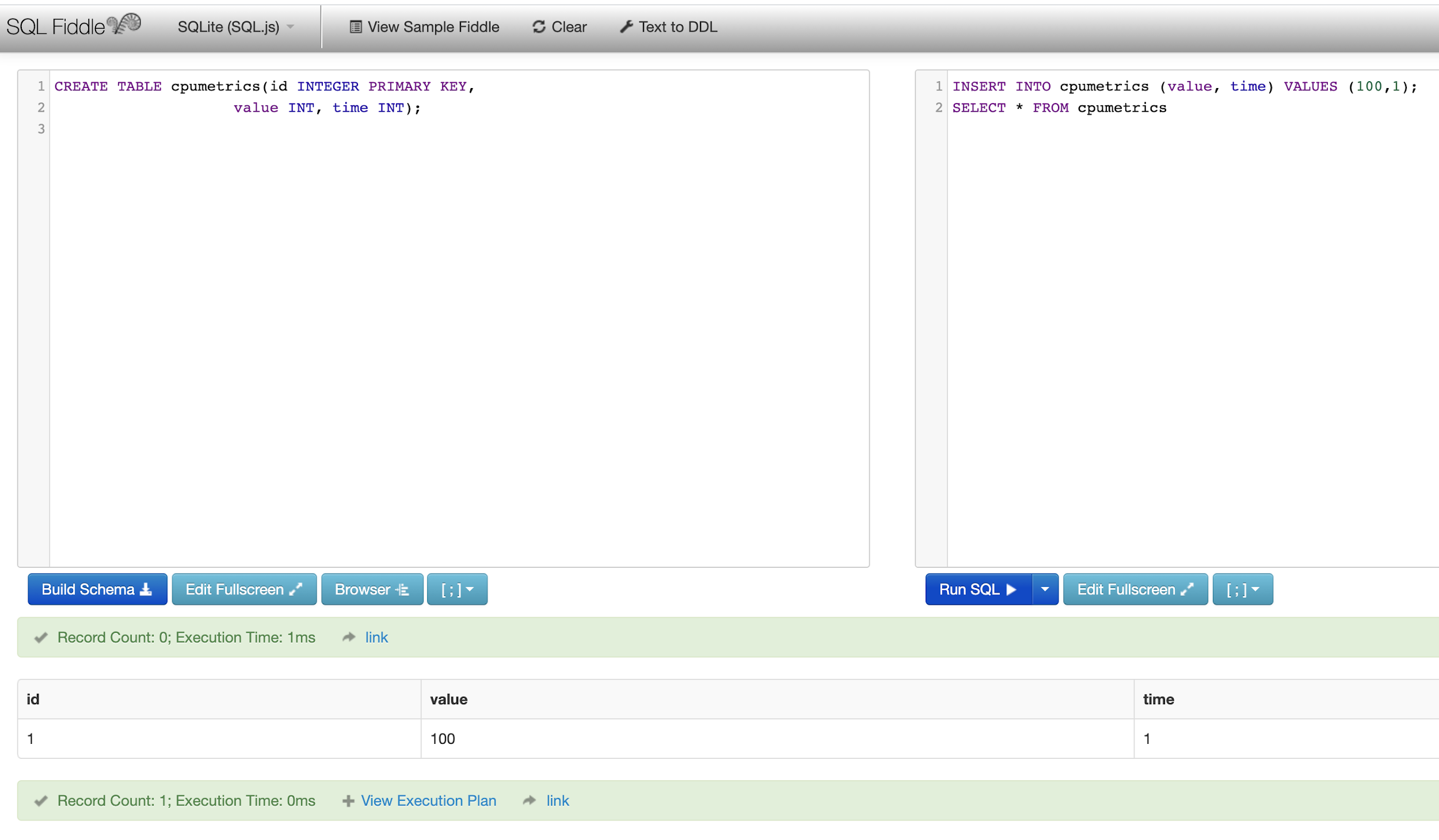


Прежде, чем выполнять запросы, необходимо построить схему СУБД. Построить схему, это значит создать все таблицы в базе данных. Кроме таблиц в СУБД еще могут находиться различные индексы, представления и хранимые процедуры, но с ними мы познакомимся позднее.

Давайте создадим простую схему с одной таблицей для метрик CPU. Для этого в левом окне выполните запрос на создание таблицы

|  |
| --- |
| CREATE TABLE cpumetrics(id INTEGER PRIMARY KEY,  value INT, time INT); |

И нажмите на кнопку Build Schema. Всё, теперь в правом окне можно работать при помощи запросов с созданной таблицей. Давайте вставим пачку различных данных.



Таким образом можно попробовать выполнить и все остальные запросы. Запросы необходимо вставлять построчно и ставить точку с запятой в конце, чтобы они выполнялись один за другим. В конце можно сделать выборку из таблицы или таблиц, и внизу увидеть результат выполнения запроса. В данном примере мы создали схему базы данных, вставили одну запись и выгрузили все содержащиеся в таблице данные.

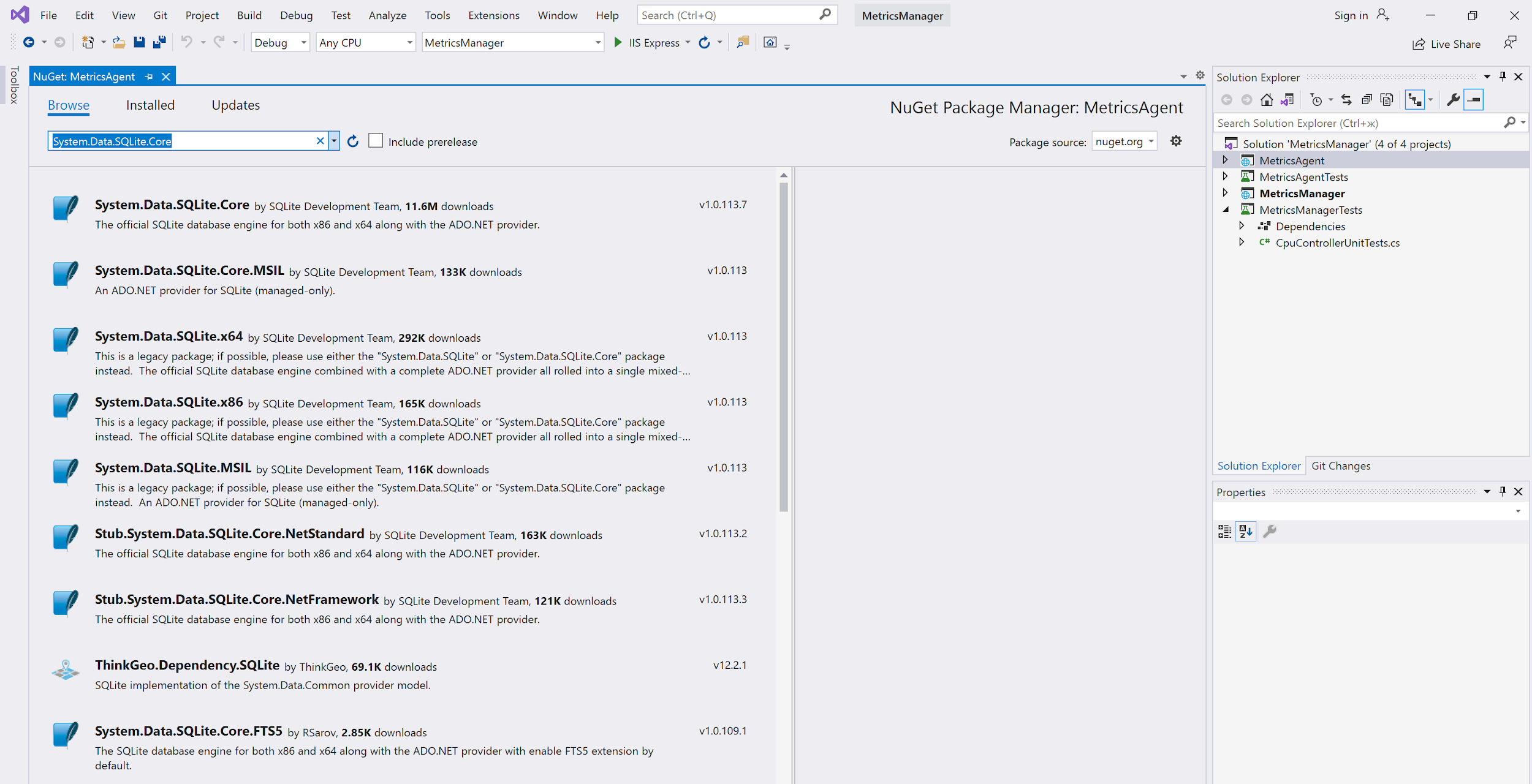
# Подключение к базе данных

Разберемся с тем, как можно работать с СУБД из кода на C#. Наше приложение агент сбора метрик будет постоянно собирать метрики и сохранять их в базу данных, которая будет лежать в виде файла на диске. Можно было бы составить простой текстовый файл, а по запросу читать данные из него, но к сожалению, такой подход не эффективен. В случае если в настоящей разработке вам понадобится хранить хоть какие-то данные, лучше выбрать для этого специализированное решение.

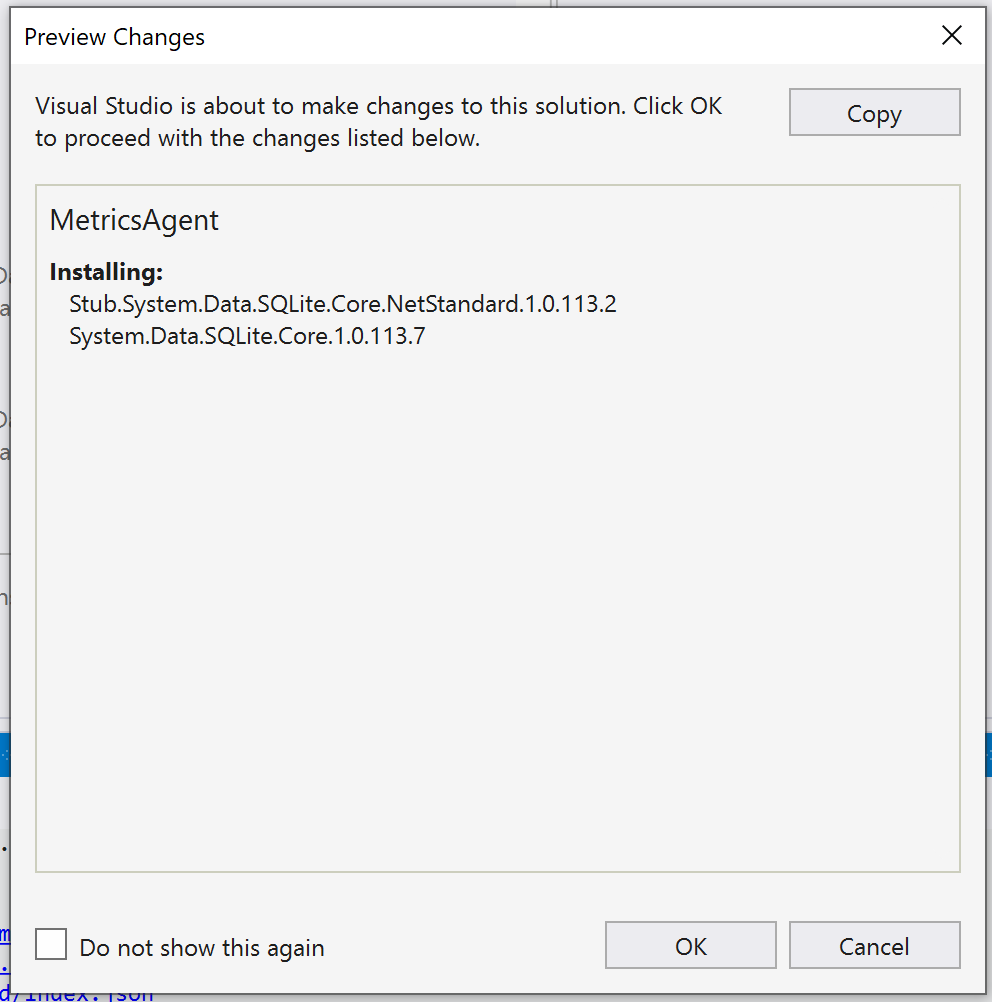
Для того, чтобы быстро попробовать работу с СУБД воспользуемся простой встраиваемой СУБД SQLite. Слово “встраиваемый” (embedded) означает, что SQLite не использует парадигму клиент-сервер, то есть SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой программа компонуется, и становится составной частью программы. Таким образом, в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа или напрямую в памяти. На примере хранящегося в памяти файла мы и рассмотрим работу с данной СУБД.

## Подключение SQLite к агенту метрик

Чтобы подключить возможность использования SQLite базы данных, воспользуемся подключаемым через nuget пакетом. Для этого откройте браузер пакетов для проекта агента метрик, как показано на рисунке. Для этого щелкните правой кнопкой мыши на проекте и выберите пункт Manage Nuget Packages



В поисковой строке выберете пакет System.Data.SQLite.Core и нажмите установку.



Данная библиотека предоставляет некоторые типы, которые нуждаются в дополнительном комментарии. SQLiteConnection, SQLiteCommand, SQLiteDataReader, SQLiteDataAdapter являются основными элементами этой библиотеки. SQLiteConnection создает соединение с определенным источником данных. Объект SQLiteCommand выполняет инструкцию SQL для источника данных. SQLiteDataReader считывает потоки данных из источника данных. SQLiteDataAdapter является посредником между DataSet и источником данных. Он заполняет DataSet и разрешает обновления с источником данных. Все эти классы реализуют стандартные интерфейсы в технологии доступа к данным ADO.NET. Эта технология декларирует стандартные интерфейсы для доступа к данным. Мы рассмотрим их назначение и реализации в классах для SQLite ниже, а пока, выполним подключение к базе данных непосредственно из кода.

Объект DataSet используется для автономной работы с большим объемом данных. Это не связное представление данных, которое может содержать данные из множества различных источников. И SQLiteDataReader, и DataSet используются для работы с данными; они используются в разных обстоятельствах. Если нам нужно только прочитать результаты запроса, SQLiteDataReader - лучший выбор. Если нам нужна более обширная обработка данных, предпочтительнее DataSet.

## Подключение к базе данных

Попробуем сделать контроллер, в рамках которого мы развернем экземпляр базы данных прямо в памяти, и запросим версию базы данных. Создайте контроллер и метод api по адресу /api/cpumetrics/sql-test

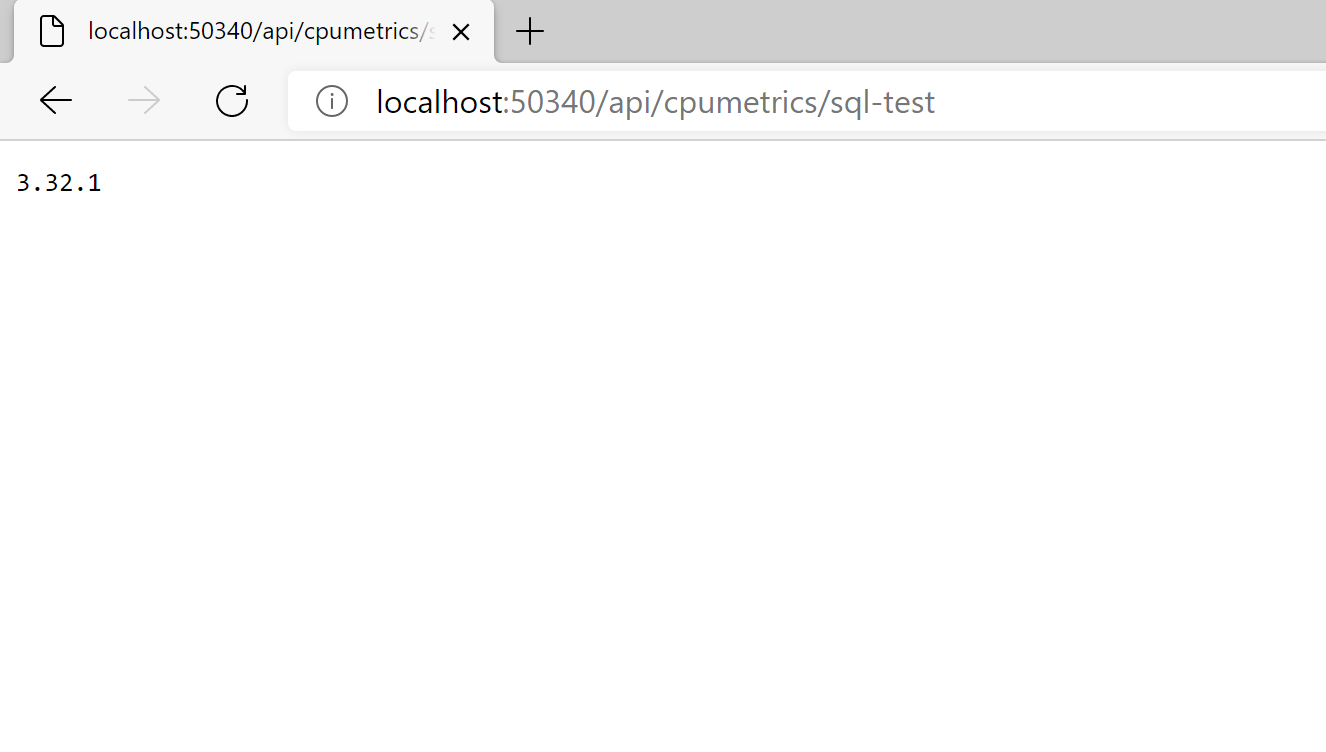
|  |
| --- |
| using Microsoft.AspNetCore.Mvc; using System.Data.SQLite;  namespace MetricsAgent.Controllers {  [Route("api/[controller]")]  [ApiController]  public class CpuMetricsController : ControllerBase  {  [HttpGet("sql-test")]  public IActionResult TryToSqlLite()  {  string cs = "Data Source=:memory:";  string stm = "SELECT SQLITE\_VERSION()";   using (var con = new SQLiteConnection(cs))  {  con.Open();   using var cmd = new SQLiteCommand(stm, con);  string version = cmd.ExecuteScalar().ToString();   return Ok(version);  }  }  } } |

В данном методе мы подключаемся к базе данных в памяти и получаем её версию. В переменной cs мы создаем строку подключения к базе данных. Она используется поставщиком данных для установления соединения с базой данных. Затем мы создаем строку с запросом, которую отправим в базу данных. При помощи оператора SELECT мы выбираем данные, однако данные мы выберем из того, что вернет встроенная в SQLite функция SQLITE\_VERSION(). Данная функция возвращает версию СУБД в строковом виде, которую и выбирает затем оператор SELECT.

Затем при помощи объекта SQLiteConnection мы открываем соединение с базой данных.

SQLiteComand - это объект, при помощи которого можно выполнять запросы к базе данных. Он принимает в себя строку с запросом, а также соединение, в рамках которого необходимо выполнить запрос. Следом, мы вызываем у команды функцию выполнения и на лету преобразуем её в строчку, которую записываем в переменную version.

Теперь можно вернуть строку version в ответ клиенту (браузеру). Попробуйте запустить отладку и обратитесь по адресу этого метода, чтобы увидеть в браузере результат выполнения данной функции



Напишем второй метод, в котором создадим таблицу, наполним её данными и сделаем выборку данных

Для начала, напишем простой класс, в объектах которого будем хранить данные, которые мы достанем из базы данных.

|  |
| --- |
| public class CpuMetricDto  {  public int Id { get; set; }   public int Value { get; set; }   public int Time { get; set; }  } |

Теперь напишем метод, в рамках которого сходим в базу данных, запишем данные и прочитаем их часть

|  |
| --- |
| [HttpGet("sql-read-write-test")]  public IActionResult TryToInsertAndRead()  {  *// Создаем строку подключения в виде базы данных в оперативной памяти*  string connectionString = "Data Source=:memory:";   *// создаем соединение с базой данных*  using (var connection = new SQLiteConnection(connectionString))  {  *// открываем соединение*  connection.Open();   *// создаем объект через который будут выполняться команды к базе данных*  using (var command = new SQLiteCommand(connection))  {  *// задаем новый текст команды для выполнения*  *// удаляем таблицу с метриками если она существует в базе данных*  command.CommandText = "DROP TABLE IF EXISTS cpumetrics";  *// отправляем запрос в базу данных*  command.ExecuteNonQuery();   *// создаем таблицу с метриками*  command.CommandText = @"CREATE TABLE cpumetrics(id INTEGER PRIMARY KEY,  value INT, time INT)";  command.ExecuteNonQuery();   *// создаем запрос на вставку данных*  command.CommandText = "INSERT INTO cpumetrics(value, time) VALUES(10,1)";  command.ExecuteNonQuery();  command.CommandText = "INSERT INTO cpumetrics(value, time) VALUES(50,2)";  command.ExecuteNonQuery();  command.CommandText = "INSERT INTO cpumetrics(value, time) VALUES(75,4)";  command.ExecuteNonQuery();  command.CommandText = "INSERT INTO cpumetrics(value, time) VALUES(90,5)";  command.ExecuteNonQuery();   *// создаем строку для выборки данных из базы*  *// LIMIT 3 обозначает, что мы достанем только 3 записи*  string readQuery = "SELECT \* FROM cpumetrics LIMIT 3";   *// создаем массив, в который запишем объекты с данными из базы данных*  var returnArray = new CpuMetricDto[3];  *// изменяем текст команды на наш запрос чтения*  command.CommandText = readQuery;   *// создаем читалку из базы данных*  using (SQLiteDataReader reader = command.ExecuteReader())  {  *// счетчик для того, чтобы записать объект в правильное место в массиве*  var counter = 0;  *// цикл будет выполняться до тех пор, пока есть что читать из базы данных*  while (reader.Read())  {  *// создаем объект и записываем его в массив*  returnArray[counter] = new CpuMetricDto  {  Id = reader.GetInt32(0), *// читаем данные полученные из базы данных*  Value = reader.GetInt32(0), *// преобразуя к целочисленному типу*  Time = reader.GetInt32(0)  };  *// увеличиваем значение счетчика*  counter++;  }  }  *// оборачиваем массив с данными в объект ответа и возвращаем пользователю*   return Ok(returnArray);  }  }  } |

Попробуйте запустить отладку и выполнить данный код, обратившись из браузера или Postman по адресу api/cpumetrics/sql-read-write-test



Как можно увидеть, мы получили в браузер сериализованный массив с данными, которые были сначала записаны в базу данных, а потом были успешно из неё прочитаны.

Теперь разберемся со стандартными интерфейсами, которые были реализованы в объектах для SQLite базы данных.

### Интерфейс IDbConnection

Представляет открытое соединение с источником данных и реализуется поставщиками данных .NET, которые обращаются к реляционным базам данных. В SQLite библиотеке данный интерфейс реализован в классе SQLiteConnection.

### Интерфейс IDbCommand

Представляет инструкцию SQL, которая выполняется при подключении к источнику данных и реализуется поставщиками данных .NET, которые обращаются к реляционным базам данных. Выше мы использовали реализацию этого интерфейса в виде объекта SQLiteComand.

### Интерфейс IDataReader

Предоставляет средства чтения одного или нескольких прямых потоков наборов результатов, полученных при выполнении команды в источнике данных, и реализуется поставщиками данных .NET, которые обращаются к реляционным базам данных. Мы воспользовались реализацией этого интерфейса при работе с объектом SQLiteDataReader.

# Интерфейсы репозиториев приложения

## Паттерн Репозиторий

Одним из наиболее часто используемых паттернов при работе с данными является паттерн 'Репозиторий'. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Допустим, у нас есть одно подключение к базе данных SQLite. Однако, что если в какой-то момент времени мы захотим изменить подключение с SQLite на другое - например, к бд PostgreSql или MongoDB. При стандартном подходе даже в небольшом приложении, осуществляющем выборку, добавление, изменение и удаление данных, нам бы пришлось сделать большое количество изменений. Либо в процессе работы программы в зависимости от разных условий мы хотим использовать два разных подключения. Таким образом, репозиторий добавляет программе гибкость при работе с разными типами подключений.

Добавим следующую модель в агент метрик

|  |
| --- |
| public class CpuMetric  {  public int Id { get; set; }   public int Value { get; set; }   public TimeSpan Time { get; set; }  } |

Теперь определим интерфейс репозитория

|  |
| --- |
| using System.Collections.Generic;  namespace MetricsAgent.DAL {  public interface IRepository<T> where T : class  {  IList<T> GetAll();   T GetById(int id);   void Create(T item);   void Update(T item);   void Delete(int id);  } } |

Теперь добавим реализацию интерфейса репозитория и сделаем конкретный класс репозиторий для работы с метриками приложения по CPU:

|  |
| --- |
| using MetricsAgent.Models; using System; using System.Collections.Generic; using System.Data.SQLite;  namespace MetricsAgent.DAL {  *// маркировочный интерфейс*  *// необходим, чтобы проверить работу репозитория на тесте-заглушке*  public interface ICpuMetricsRepository : IRepository<CpuMetric>  {   }   public class CpuMetricsRepository : ICpuMetricsRepository  {  *// наше соединение с базой данных*  private SQLiteConnection connection;   *// инжектируем соединение с базой данных в наш репозиторий через конструктор*  public CpuMetricsRepository(SQLiteConnection connection)  {  this.connection = connection;  }   public void Create(CpuMetric item)  {  *// создаем команду*  using var cmd = new SQLiteCommand(connection);  *// прописываем в команду SQL запрос на вставку данных*  cmd.CommandText = "INSERT INTO cpumetrics(value, time) VALUES(@value, @time)";   *// добавляем параметры в запрос из нашего объекта*  cmd.Parameters.AddWithValue("@value", item.Value);   *// в таблице будем хранить время в секундах, потому преобразуем перед записью в секунды*  *// через свойство*  cmd.Parameters.AddWithValue("@time", item.Time.TotalSeconds);  *// подготовка команды к выполнению*  cmd.Prepare();   *// выполнение команды*  cmd.ExecuteNonQuery();  }   public void Delete(int id)  {  using var cmd = new SQLiteCommand(connection);  *// прописываем в команду SQL запрос на удаление данных*  cmd.CommandText = "DELETE FROM cpumetrics WHERE id=@id";   cmd.Parameters.AddWithValue("@id", id);  cmd.Prepare();  cmd.ExecuteNonQuery();  }   public void Update(CpuMetric item)  {  using var cmd = new SQLiteCommand(connection);  *// прописываем в команду SQL запрос на обновление данных*  cmd.CommandText = "UPDATE cpumetrics SET value = @value, time = @time WHERE id=@id;";  cmd.Parameters.AddWithValue("@id", item.Id);  cmd.Parameters.AddWithValue("@value", item.Value);  cmd.Parameters.AddWithValue("@time", item.Time.TotalSeconds);  cmd.Prepare();   cmd.ExecuteNonQuery();  }   public IList<CpuMetric> GetAll()  {  using var cmd = new SQLiteCommand(connection);   *// прописываем в команду SQL запрос на получение всех данных из таблицы*  cmd.CommandText = "SELECT \* FROM cpumetrics";   var returnList = new List<CpuMetric>();   using (SQLiteDataReader reader = cmd.ExecuteReader())  {  *// пока есть что читать -- читаем*  while (reader.Read())  {  *// добавляем объект в список возврата*  returnList.Add(new CpuMetric  {  Id = reader.GetInt32(0),  Value = reader.GetInt32(0),  *// налету преобразуем прочитанные секунды в метку времени*  Time = TimeSpan.FromSeconds(reader.GetInt32(0))   });  }  }   return returnList;  }   public CpuMetric GetById(int id)  {  using var cmd = new SQLiteCommand(connection);  cmd.CommandText = "SELECT \* FROM cpumetrics WHERE id=@id";  using (SQLiteDataReader reader = cmd.ExecuteReader())  {  *// если удалось что то прочитать*  if (reader.Read())  {  *// возвращаем прочитанное*  return new CpuMetric  {  Id = reader.GetInt32(0),  Value = reader.GetInt32(0),  Time = TimeSpan.FromSeconds(reader.GetInt32(0))  };  }  else  {  *// не нашлось запись по идентификатору, не делаем ничего*  return null;  }  }  }  } } |

Реализация репозитория может показаться довольно объемной, но благодаря подробным комментариям вы без труда разберетесь в нем. Основная идея заключается в том, что данный конкретный репозиторий предназначен для работы с базой SQLite и работе с таблицей метрик CPU, а потому в конструктор принимает зависимость в виде уже созданного подключения к базе данных. Давайте зарегистрируем объект подключения в DI контейнере приложения, а также добавим в DI наш репозиторий, чтобы им можно было воспользоваться в конструкторе контроллера.

Для начала модифицируем файл Startup.cs

|  |
| --- |
| using MetricsAgent.DAL; using Microsoft.AspNetCore.Builder; using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.Configuration; using Microsoft.Extensions.DependencyInjection; using Microsoft.Extensions.Hosting; using System.Data.SQLite;  namespace MetricsAgent {  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();  ConfigureSqlLiteConnection(services);  services.AddScoped<ICpuMetricsRepository,CpuMetricsRepository>();  }   private void ConfigureSqlLiteConnection(IServiceCollection services)  {  string connectionString = "Data Source=:memory:";  var connection = new SQLiteConnection(connectionString);  connection.Open();  PrepareSchema(connection);  services.AddSingleton(connection);  }   private void PrepareSchema(SQLiteConnection connection)  {  using (var command = new SQLiteCommand(connection))  {  *// задаем новый текст команды для выполнения*  *// удаляем таблицу с метриками если она существует в базе данных*  command.CommandText = "DROP TABLE IF EXISTS cpumetrics";  *// отправляем запрос в базу данных*  command.ExecuteNonQuery();     command.CommandText = @"CREATE TABLE cpumetrics(id INTEGER PRIMARY KEY,  value INT, time INT)";  command.ExecuteNonQuery();  }  }    *// 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.UseRouting();   app.UseAuthorization();   app.UseEndpoints(endpoints =>  {  endpoints.MapControllers();  });  }  } } |

Теперь можно добавить использование репозитория в контроллер метрик CPU в проекте сервиса-агента метрик

|  |
| --- |
| using MetricsAgent.DAL; using MetricsAgent.Models; using MetricsAgent.Requests; using MetricsAgent.Responses; using Microsoft.AspNetCore.Mvc; using System.Collections.Generic;  namespace MetricsAgent.Controllers {  [Route("api/[controller]")]  [ApiController]  public class CpuMetricsController : ControllerBase  {  private ICpuMetricsRepository repository;   public CpuMetricsController(ICpuMetricsRepository repository)  {  this.repository = repository;  }   [HttpPost("create")]  public IActionResult Create([FromBody] CpuMetricCreateRequest request)  {  repository.Create(new CpuMetric  {  Time = request.Time,  Value = request.Value  });   return Ok();  }   [HttpGet("all")]  public IActionResult GetAll()  {  var metrics = repository.GetAll();   var response = new AllCpuMetricsResponse()  {  Metrics = new List<CpuMetricDto>()  };   foreach (var metric in metrics)  {  response.Metrics.Add(new CpuMetricDto { Time = metric.Time, Value = metric.Value, Id = metric.Id });  }   return Ok(response);  }   } } |

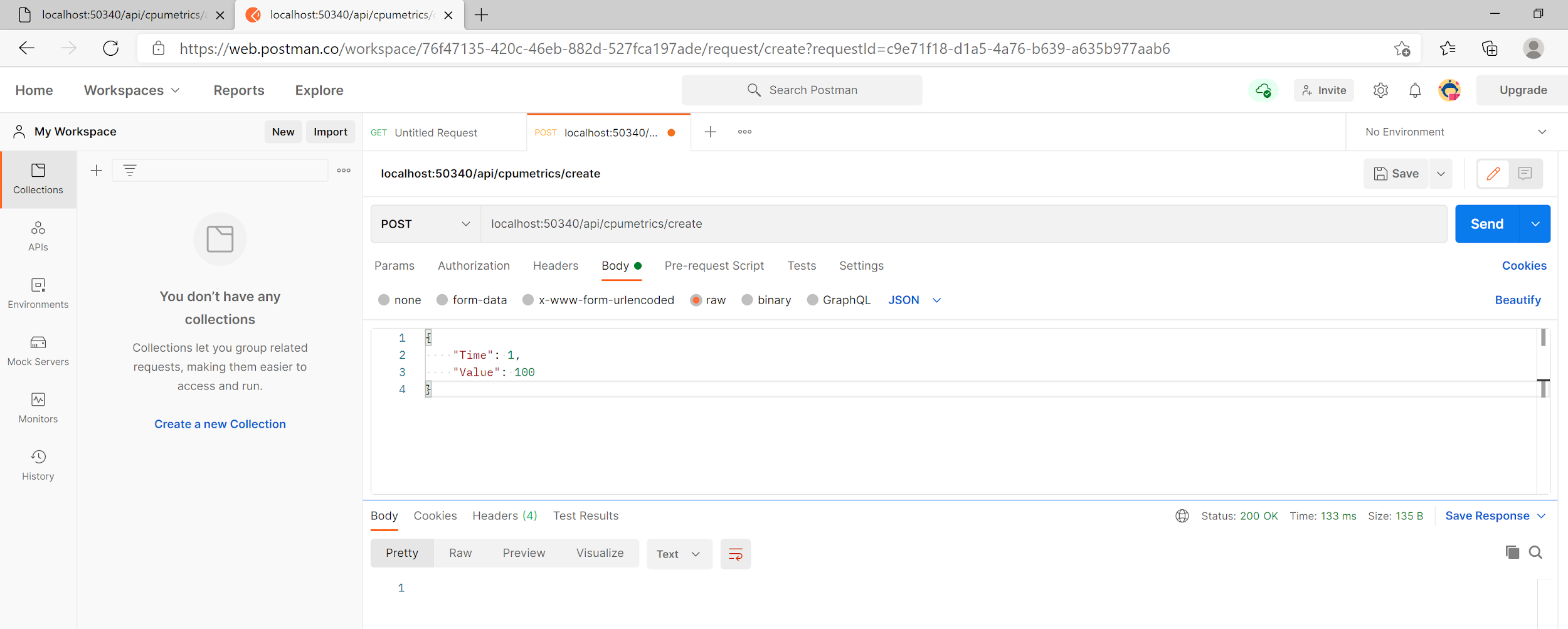
Как вы могли заметить, у нас есть еще классы, которые служат контейнерами для запроса и ответа. Приведем их код ниже:

|  |
| --- |
| using System;  namespace MetricsAgent.Requests {  public class CpuMetricCreateRequest  {  public TimeSpan Time { get;set; }  public int Value { get;set; }  } } |

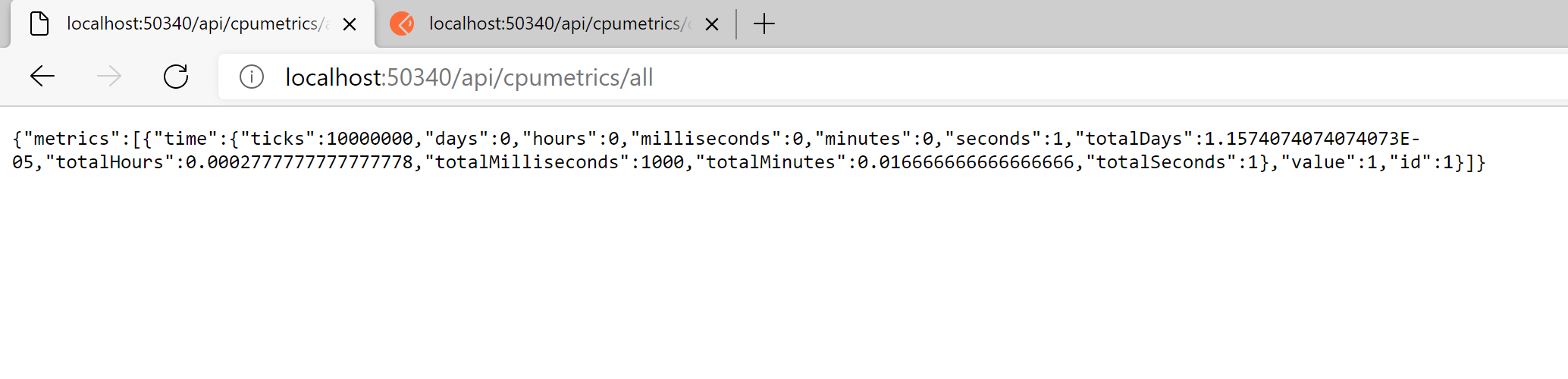
|  |
| --- |
| using System; using System.Collections.Generic;  namespace MetricsAgent.Responses {  public class AllCpuMetricsResponse  {  public List<CpuMetricDto> Metrics { get; set; }  }   public class CpuMetricDto  {  public TimeSpan Time { get; set; }  public int Value { get; set; }  public int Id { get; set; }  } } |

Запустите отладку приложения и попробуйте вызвать методы контроллера через Postman. Можете установить точки останова и пройти отладчиком все использование кода, чтобы лучше понять его работу.

Давайте добавим несколько метрик:



А теперь прочитаем их обратившись к методу получения всех метрик



Студентам предлагается реализовать остальные методы для CRUD операций в контроллерах для остальных метрик на агенте точно так же, как и оставшиеся репозитории для работы с метриками. На каждый тип метрики следует завести свой собственный репозиторий со своими объектами хранения данных (DTO).

# Mock тестировние

После внесения изменений, наш, созданный на прошлом уроке тест перестал компилироваться. К тому же мы добавили новую зависимость в код, а это значит, что нам необходимо переделать то, как работает тест.

Тест должен быть все так же изолированным и проверять только работу метода контроллера. Воспользуемся тестирование с заглушками или Mock тестированием.

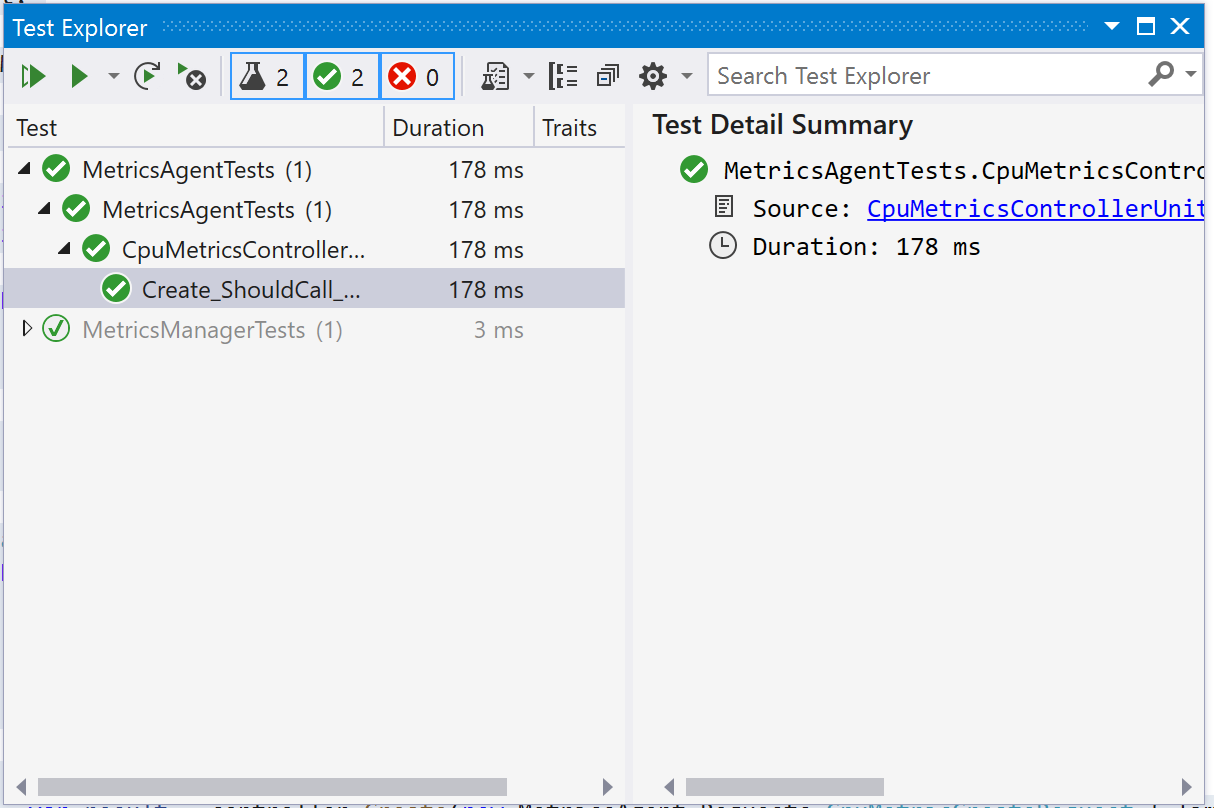
Для того, чтобы писать тесты с заглушками, добавьте в проект тестов через nuget библиотеку Moq

Далее, напишем наш первый тест с использованием заглушки в проекте для тестов сервиса агента сбора метрик

|  |
| --- |
| using MetricsAgent.Controllers; using MetricsAgent.DAL; using MetricsAgent.Models; using Moq; using System; using Xunit;  namespace MetricsAgentTests {  public class CpuMetricsControllerUnitTests  {  private CpuMetricsController controller;  private Mock<ICpuMetricsRepository> mock;   public CpuMetricsControllerUnitTests()  {  mock = new Mock<ICpuMetricsRepository>();   controller = new CpuMetricsController(mock.Object);  }   [Fact]  public void Create\_ShouldCall\_Create\_From\_Repository()  {  *// устанавливаем параметр заглушки*  *// в заглушке прописываем что в репозиторий прилетит CpuMetric объект*  mock.Setup(repository => repository.Create(It.IsAny<CpuMetric>())).Verifiable();   *// выполняем действие на контроллере*  var result = controller.Create(new MetricsAgent.Requests.CpuMetricCreateRequest { Time = TimeSpan.FromSeconds(1), Value = 50 });   *// проверяем заглушку на то, что пока работал контроллер*  *// действительно вызвался метод Create репозитория с нужным типом объекта в параметре*  mock.Verify(repository => repository.Create(It.IsAny<CpuMetric>()), Times.AtMostOnce());  }  } } |

Таким образом можно тестировать только логику работы контроллера (или любого другого класса) не создавая целиком его зависимости. Библиотека имитирует поведение зависимости класса самостоятельно.

Студентам предлагается подобным образом добавить тесты на каждый метод контроллера и убедиться в том, что они успешно проходят



# Задание для самостоятельной работы

1. Добавьте логирование всех параметров в каждом из контроллеров в обоих проектах
2. Добавьте репозитории для каждого типа метрик в сервис агент сбора метрик
3. Добавьте тесты на все контроллеры и все методы с использованием заглушек

# Список дополнительной литературы

* <https://www.w3schools.com/sql/>
* <https://github.com/NLog/NLog/wiki/Getting-started-with-ASP.NET-Core-3>
* <https://nlog-project.org/>
* <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>
* <https://www.hanselman.com/blog/list-of-net-dependency-injection-containers-ioc>
* <https://rubikscode.net/2018/04/09/exploring-dependency-injection-in-c-and-top-3-di-containers-part-2/>

# 