Asp.net core Web API микросервисы. Часть 1

Работа по расписанию



# На этом уроке

На прошлом уроке мы много избавлялись от лишнего “скучного” кода при помощи ORM Dapper и AutoMapper. На этом уроке мы сконцентрируемся на приложении агенте сбора метрик и приведем его в полностью работоспособное состояние, добавив в него недостающую функциональность. Также мы познакомимся с метриками и их автоматическим сбором.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_n93g6yblf23c)

[Миграции](#_n9r8jonks8we)

[FluentMigrator](#_ckfqvckpt6x3)

[Работа по расписанию](#_jzqiu738swtr)

[Библиотека Quarz.NET](#_fs88kyj5kznw)

[Получение системных метрик](#_93uxyfbn1sk)

[Просмотр dotnet-counters](#_uvbt2xddlz3p)

[Получение метрик работы из кода приложения](#_uchjbwjzir4h)

[API для метрик на сервисе агенте сбора метрик](#_4etrp4uwhr93)

[API контроллеров](#_ufnqk2dd2342)

[Практическое задание](#_tf1198bb3721)

[Список используемой литературы](#_54hssoocdbld)

[Дополнительные материалы](#_oreyj64p9zko)

# Вступление

В приложениях, которые используют СУБД возникает задача настройки схемы СУБД перед её использованием. В ходе разработки приложения, схема СУБД может меняться. В индустрии сложился подход, согласно которому приложение может готовить схему в СУБД под свои нужды перед началом работы, если это необходимо. Такая технология называется “миграция”. На данном уроке мы познакомимся с механизмом миграций и напишем миграции для нашего приложения сбора метрик. Еще одной распространенной задачей является работа по расписанию, когда какие-то действия необходимо совершать автоматически в определенное время. Для этого существует множество решений и мы рассмотрим одно из них, естественно, самое распространенное среди разработчиков asp.net core.

# Миграции

В реальных проектах модели данных изменяются по мере реализации функций: добавляются и удаляются новые сущности или свойства, а схемы базы данных необходимо соответствующим образом изменять, чтобы они синхронизировались с приложением. Для того, чтобы “подхватить” изменившиеся требования и преобразовать их в изменение схемы базы данных существуют миграции – специальный механизм, который при помощи кода позволяет изменять схему базы данных.

Предположим, что у нас есть необходимость задать изначальную конфигурацию для базы данных. Напишем код, который это будет делать и снабдим класс, в котором располагается данный код, некоторой меткой, обозначающей версию. Например, версией может выступать дата, когда мы решили создать нашу схему базы данных. Спустя некоторое время, требования схеме данных могут измениться под влиянием новых бизнес-требований. Для того, чтобы удовлетворить требованиям, напишем еще один класс, который “мигрирует” старую схему базы данных на новую (тут можно сказать, что изначально мы тоже “мигрировали” схему из пустоты в начальное состояние), а также некоторый код, который будет собирать все “миграции”, упорядочивать их по версии и прогонять одну за другой. Каждый раз выполнять все миграции нет необходимости, потому в базу данных вместе с основной схемой создадим табличку, в которой будет храниться версия последней миграции примененной к схеме СУБД. Перед запуском миграции мы будем проверять версию, и если версия в СУБД будет отставать от той что есть у нас в коде, применим к схеме СУБД недостающие миграции.

Именно в таком виде и реализован механизм миграций в большинстве библиотек. Нет необходимости в большинстве случаев писать свой механизм миграций. Рассмотрим специально-предназначенную для этого библиотеку.

## FluentMigrator

Установите при помощи nuget следующие пакеты в приложение сбора метрик: FluentMigrator, FluentMigrator.Runner, FluentMigrator.Runner.SQLite. Теперь создадим папку в папке DAL и назовем её Migrations. Тут будем хранить все наши миграции. Напишем первую миграцию, которая будет создавать нашу таблицу для хранения метрик CPU.

|  |
| --- |
| using FluentMigrator;  namespace MetricsAgent.DAL.Migrations {  [Migration(1)]  public class FirstMigration : Migration  {  public override void Up()  {  Create.Table("cpumetrics")  .WithColumn("Id").AsInt64().PrimaryKey().Identity()  .WithColumn("Value").AsInt32()  .WithColumn("Time").AsInt64();  }   public override void Down()  {  Delete.Table("cpumetrics");  }  } } |

В атрибуте класса указана версия миграции. Сам класс миграции наследован от абстрактного класса Migration и реализует методы Up() и Down(). В методе Up мы пишем код, который выполнится при применении миграции к СУБД, а в методе Down() пишется код, который будет выполнен в случае отката схемы СУБД к определенной версии. В методе Up мы написали код, который создает таблицу cpumetrics с Id в качестве первичного ключа и двумя полями Value и Time в которых будут хранится числовые значения метрики и метки времени, когда была снята метрика.

Теперь зарегистрируем наш мигратор в DI. Для этого можно удалить метод PrepareSchema так как он нам больше не понадобится

|  |
| --- |
| using AutoMapper; 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; using FluentMigrator.Runner; using System.IO;  namespace MetricsAgent {  public class Startup  {  public Startup(IConfiguration configuration)  {  Configuration = configuration;  }   public IConfiguration Configuration { get; }   private const string ConnectionString = @"Data Source=metrics.db; Version=3;";   *// This method gets called by the runtime. Use this method to add services to the container.*  public void ConfigureServices(IServiceCollection services)  {  services.AddControllers();  services.AddSingleton<ICpuMetricsRepository, CpuMetricsRepository>();  var mapperConfiguration = new MapperConfiguration(mp => mp.AddProfile(new MapperProfile()));  var mapper = mapperConfiguration.CreateMapper();  services.AddSingleton(mapper);   services.AddFluentMigratorCore()  .ConfigureRunner(rb => rb  *// добавляем поддержку SQLite*   .AddSQLite()  *// устанавливаем строку подключения*  .WithGlobalConnectionString(ConnectionString)  *// подсказываем где искать классы с миграциями*  .ScanIn(typeof(Startup).Assembly).For.Migrations()  ).AddLogging(lb => lb  .AddFluentMigratorConsole());  }  *// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.*  public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IMigrationRunner migrationRunner)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }   app.UseRouting();   app.UseAuthorization();   app.UseEndpoints(endpoints =>  {  endpoints.MapControllers();  });   *// запускаем миграции*  migrationRunner.MigrateUp();  }  } } |

Теперь при старте нашего приложения сбора метрик схема базы данных будет создаваться автоматически. Попробуйте запустить отладчиком наш сервис, и в консоли Output можно будет видеть сообщения логов применения миграций.

Замечание: в разработанной миграции тип данных для поля Time выбран как long. Возможна ошибка чтения данных при помощи репозитория. Для этого, убедитесь, что в классе TimeSpanHandler написан следующий код

|  |
| --- |
| class TimeSpanHandler : SqliteTypeHandler<TimeSpan>  {  public override TimeSpan Parse(object value)  => TimeSpan.FromSeconds((long)value);  } |

Внутри файла с первой миграцией можно создать и остальные таблицы, с которыми мы будем работать.

# Работа по расписанию

Распространенной задачей для asp.net core приложений является запуск некоторых процедур по расписанию. Зачастую это могут быть автоматические проверки, построение отчетов по данным из СУБД и так далее.

В нашем приложении сбора метрик возникает необходимость сбора метрик с определенными интервалами времени. Для этого можно было бы создать отдельный поток, и передать ему таймер, срабатывания которого ждать в цикле. Такой подход слишком ограничен. В случае если нам понадобится изменять расписание запуска такого потока, нам каждый раз придется писать новый код.

Для решения задачи существует мощная библиотека Quartz.NET. Давайте рассмотрим её возможности и подключим к нашему проекту.

## Библиотека Quarz.NET

Библиотека Quartz.NET содержит в себе две важных сущности:

* Задача - некоторый код, который запускается по расписанию
* Планировщик, который ответственен за запуск задач по расписанию

ASP.NET Core имеет хорошую поддержку для запуска “фоновых задач” через hosted службы. Hosted службы запускаются при запуске приложения ASP.NET Core и работают в фоновом режиме в течение всего времени существования приложения. Создав hosted службу Quartz.NET, вы можете использовать стандартное приложение ASP.NET Core для выполнения ваших задач в фоновом режиме. При помощи такой службы можно создать что-то простое, например, раз в 10 минут строить отчет. Quartz.NET предоставляет гораздо более надежное решение. Вы можете обеспечить выполнение задач только в определенное время дня (например, 2:30 утра), или только в определенные дни, или любую комбинацию с помощью триггера Cron. Это также позволяет вам запускать несколько экземпляров вашего приложения в кластеризованном режиме, так что только один экземпляр может запускать данную задачу в любой момент времени.

Установите из nuget пакет Quartz. Теперь напишем нашу первую задачу.

|  |
| --- |
| using MetricsAgent.DAL; using Microsoft.Extensions.DependencyInjection; using Quartz; using System; using System.Diagnostics; using System.Threading.Tasks;  namespace MetricsAgent.Jobs {  public class CpuMetricJob : IJob  {  *// Инжектируем DI провайдер*  private readonly IServiceProvider \_provider;  private ICpuMetricsRepository repository;    public CpuMetricJob(IServiceProvider provider)  {  \_provider = provider;  repository = \_provider.GetService<ICpuMetricsRepository>();  }   public Task Execute(IJobExecutionContext context)  {  *// теперь можно записать что-то при помощи репозитория*   return Task.CompletedTask;  }  } } |

Использование самого репозитория можно сделать таким же как и в контроллере до этого. Пока не будем концентрироваться на этом. Обратите внимание, что нам пришлось инжектировать DI в задачу. Такое необходимо только для сервисов, которые зарегистрированы в контейнере с временем жизни Scoped, для Singleton такого не требуется.

Теперь создадим планировщик

|  |
| --- |
| using Microsoft.Extensions.DependencyInjection; using Quartz; using Quartz.Spi; using System;  public class SingletonJobFactory : IJobFactory {  private readonly IServiceProvider \_serviceProvider;  public SingletonJobFactory(IServiceProvider serviceProvider)  {  \_serviceProvider = serviceProvider;  }   public IJob NewJob(TriggerFiredBundle bundle, IScheduler scheduler)  {  return \_serviceProvider.GetRequiredService(bundle.JobDetail.JobType) as IJob;  }   public void ReturnJob(IJob job) { } } |

Эта фабрика принимает IServiceProvider в конструкторе и реализует интерфейс IJobFactory. Важным методом является метод NewJob(), в котором фабрика должна возвращать IJob, запрошенный планировщиком Quartz. В этой реализации мы напрямую делегируем IServiceProvider и позволяем контейнеру DI находить требуемый экземпляр. Приведение к IJob в конце требуется, потому что не универсальная версия GetRequiredService возвращает объект.

Добавим еще небольшой DTO класс, в котором будем хранить расписание запуска

|  |
| --- |
| using System;  public class JobSchedule {  public JobSchedule(Type jobType, string cronExpression)  {  JobType = jobType;  CronExpression = cronExpression;  }   public Type JobType { get; }  public string CronExpression { get; } } |

Теперь можно зарегистрировать все в DI

|  |
| --- |
| *// ДОбавляем сервисы* services.AddSingleton<IJobFactory, SingletonJobFactory>(); services.AddSingleton<ISchedulerFactory, StdSchedulerFactory>(); *// добавляем нашу задачу* services.AddSingleton<CpuMetricJob>(); services.AddSingleton(new JobSchedule(  jobType: typeof(CpuMetricJob),  cronExpression: "0/5 \* \* \* \* ?")); *// запускать каждые 5 секунд* |

Нам необходимо где то держать сервис, в котором мы будем запускать при помощи планировщика все наши задачи по расписанию. Для этого создадим реализацию IHostedService предназначенную для запуска задач с помощью Quartz.

|  |
| --- |
| using System; using System.Collections.Generic; using System.Threading; using System.Threading.Tasks; using Microsoft.Extensions.Hosting; using Quartz; using Quartz.Spi;  public class QuartzHostedService : IHostedService {  private readonly ISchedulerFactory \_schedulerFactory;  private readonly IJobFactory \_jobFactory;  private readonly IEnumerable<JobSchedule> \_jobSchedules;   public QuartzHostedService(  ISchedulerFactory schedulerFactory,  IJobFactory jobFactory,  IEnumerable<JobSchedule> jobSchedules)  {  \_schedulerFactory = schedulerFactory;  \_jobSchedules = jobSchedules;  \_jobFactory = jobFactory;  }  public IScheduler Scheduler { get; set; }   public async Task StartAsync(CancellationToken cancellationToken)  {  Scheduler = await \_schedulerFactory.GetScheduler(cancellationToken);  Scheduler.JobFactory = \_jobFactory;   foreach (var jobSchedule in \_jobSchedules)  {  var job = CreateJobDetail(jobSchedule);  var trigger = CreateTrigger(jobSchedule);   await Scheduler.ScheduleJob(job, trigger, cancellationToken);  }   await Scheduler.Start(cancellationToken);  }   public async Task StopAsync(CancellationToken cancellationToken)  {  await Scheduler?.Shutdown(cancellationToken);  }   private static IJobDetail CreateJobDetail(JobSchedule schedule)  {  var jobType = schedule.JobType;  return JobBuilder  .Create(jobType)  .WithIdentity(jobType.FullName)  .WithDescription(jobType.Name)  .Build();  }   private static ITrigger CreateTrigger(JobSchedule schedule)  {  return TriggerBuilder  .Create()  .WithIdentity($"{schedule.JobType.FullName}.trigger")  .WithCronSchedule(schedule.CronExpression)  .WithDescription(schedule.CronExpression)  .Build();  } } |

Осталось только добавить регистрацию нашего сервиса в DI.

|  |
| --- |
| services.AddHostedService<QuartzHostedService>()*;* |

При запуске нашего приложения сбора метрик, код задачи будет запускаться каждые пять секунд. Вы можете попробовать выполнить его под отладчиком.

# Получение системных метрик

Подготовительные работы завершены. Мы можем написать код для получения метрик и записи его в базу данных

## Просмотр dotnet-counters

Чтобы посмотреть какие метрики можно собирать при помощи кода можно воспользоваться утилитой dotnet-counters. Это инструмент мониторинга производительности для специального мониторинга работоспособности и исследования производительности начального уровня. Он может наблюдать за значениями счетчиков производительности, которые публикуются через EventCounter API. Например, вы можете быстро отслеживать такие вещи, как использование CPU или количество исключений, возникающих в вашем приложении .NET Core, чтобы увидеть, есть ли что-нибудь подозрительное

Выполните в консоли следующую команду

|  |
| --- |
| dotnet tool install --global dotnet-counters |

Теперь можно получить список возможных метрик для просмотра

Dotnet counters умеет подключаться к приложениям написанным на .net. Мы можем написать простую консольную программу и попробовать подключиться к ней. Для того, чтобы узнать идентификатор процесса, найдите запущенную программу в диспетчере задач Windows и найдите там PID. это и есть идентификатор процесса

|  |
| --- |
| dotnet-counters monitor cpu-usage -p ИДЕНТИФИКАТОР\_ПРОЦЕССА |

## Получение метрик работы из кода приложения

В нашей задаче по сбору CPU метрик напишем следующий код

|  |
| --- |
| using MetricsAgent.DAL; using Microsoft.Extensions.DependencyInjection; using Quartz; using System; using System.Diagnostics; using System.Threading.Tasks;  namespace MetricsAgent.Jobs {  public class CpuMetricJob : IJob  {  *// Инжектируем DI провайдер*  private readonly IServiceProvider \_provider;  private ICpuMetricsRepository \_repository;    *// счетчик для метрики CPU*  private PerformanceCounter \_cpuCounter;     public CpuMetricJob(IServiceProvider provider)  {  \_provider = provider;  \_repository = \_provider.GetService<ICpuMetricsRepository>();  \_cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "\_Total");  }   public Task Execute(IJobExecutionContext context)  {  *// получаем значение занятости CPU*  var cpuUsageInPercents = Convert.ToInt32(\_cpuCounter.NextValue());   *// узнаем когда мы сняли значение метрики.*  var time = TimeSpan.FromSeconds(DateTimeOffset.UtcNow.ToUnixTimeSeconds());   *// теперь можно записать что-то при помощи репозитория*   \_repository.Create(new Models.CpuMetric { Time = time, Value = cpuUsageInPercents });    return Task.CompletedTask;  }  } } |

Мы воспользовались стандартным счетчиком производительности (<https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.performancecounter?redirectedfrom=MSDN&view=net-5.0>) и при помощи него взяли метрики для CPU. Для метрик RAM можно воспользоваться следующим счетчиком

|  |
| --- |
| ramCounter = new PerformanceCounter("Memory", "Available MBytes"); |

# API для метрик на сервисе агенте сбора метрик

Наше приложение агент сбора метрик теперь умеет собирать метрики по расписанию. Ему необходимым методы для того, чтобы предоставить менеджеру метрик данные о метриках. На примере контроллера метрик CPU опишем методы, которые необходимы.

## API контроллеров

Приложение само собирает метрики, редактирование метрик не предполагается. Нам остается лишь сделать по одному методу в каждом контроллере сбора метрик

/metricsController//from/{fromTime}/to/{toTime}/

Первым параметром будет метка времени в секундах с 1го января 1970го года до интересующего вреени( unix time stamp), второй параметр toTime время, до которого необходимо сделать выборку.

Обратите внимание, что время в секундах хранится только в БД. Преобразование из секунд в нормальное представление (например в DateTimeOffset) необходимо сделать на уровне репозитория так, чтобы в контроллер уже отдавался объект типа DateTimeOffset, а не просто int.

При помощи SQL запроса SELECT с условием WHERE по полю Time этого можно добиться примерно следующим образом. При этои аналогично вышесказанному, следует сделать преобразование из DateTimeOffset в int на уровне репозитория, когда формируется запрос.

|  |
| --- |
| SELECT id, value, time FROM cpumetrics WHERE time>@fromTime AND time<@toTime; |

Реализуйте соответствующие методы и модели в DAL уровне у репозитория сбора метрик, а также у всех остальных репозиториев для своих метрик.

# Практическое задание

* Реализуйте задачи для сбора метрик RAM (RamMetricsController), HDD (HddMetricsController) и gc-heap-size (GcMetricsController), Network (NetworkMetricController). Сделайте это с помощью perfomance-counters. Задачи должны собирать метрики раз в 5 секунд и записывать их в созданную базу данных
* Реализуйте контроллеры, которые будут отдавать данные по собираемым метрикам
* Использовать FluentMigrator

# Список используемой литературы

* <https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-counters>
* <https://docs.microsoft.com/en-us/visualstudio/profiling/dotnet-counters-tool>
* <https://www.quartz-scheduler.net/documentation/quartz-3.x/quick-start.html>
* <https://en.wikipedia.org/wiki/Cron>

# Дополнительные материалы

* <https://qna.habr.com/q/390264>
* <https://maximaster.ru/blog/migrations/>
* <https://kodify.net/csharp/computer-drive/drive-free-space/>
* <https://stackoverflow.com/questions/1393711/get-free-disk-space>