Asp.net core Web API микросервисы. Часть 1

DDD – Domain Driven Development



# На этом уроке

На предыдущем уроке мы познакомились с тем, как можно непосредственно из кода подключиться к базе данных, разобрали что такое логирование и инъекция зависимостей. Все это необходимо нам, для того, чтобы выстроить хорошую архитектуру микросервиса, позволяющую уметь делать её простой и хорошо расширяемой. На этом уроке мы спроектируем архитектуру сервиса при помощи паттерна Медиатор, а также познакомимся с правильным стилем расположения файлов кода.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_n93g6yblf23c)

[Коллекция IList<T>](#_n9r8jonks8we)

[Коллекция IDictionary<TKey, TValue>](#_ljfz4synzia)

[ORM](#_duo7bl2tpzpn)

[Dapper](#_4diovcquk8vz)

[Бизнес-логика приложения](#_8jzcmkae8z6v)

[Практика лучшего расположения кода по папкам в проекте](#_2gfsoirudim1)

[Паттерн Медиатор (Посредник)](#_bxq66geolkg0)

[Автоматический маппинг объектов в приложении](#_fen4ex1xivcq)

[Библиотека AutoMapper](#_dshk5lotrcqi)

[Практическое задание](#_tf1198bb3721)

[Список используемой литературы](#_54hssoocdbld)

[Дополнительные материалы](#_oreyj64p9zko)

# Вступление

В разработке проектов в команде очень важно выбрать правильную архитектуру в приложении, чтобы модификация кода под быстро изменяющиеся бизнес-требования проходила быстро и безболезненно. В индустрии разработки приложений на asp.net core сложилось понимание, что паттерн Медиатор для этого подходит лучше всего. Компания Microsoft, автор фреймворка asp.net core рекомендует этот паттерн к использованию при разработке микросервисов. Дополнительно, мы должны разобраться с фреймворками, которые облегчают взаимодействие с базой данных и маппингом объектов между логическими слоями приложения. Эти вещи позволяют писать меньше однотипного кода, а чем меньше кода, тем меньше в нем ошибок.

# Коллекция IList<T>

Прежде чем мы пойдем дальше, рассмотрим возможности коллекции IList<T>. В прошлом уроке мы рассмотрели примеры кода с использованием этой коллекции, когда доставали данные из таблицы метрик на уровне репозитория приложения. Однако, мы не рассмотрели её возможностей.

IList<T> представляет собой генерализированную (с использование шаблона) коллекцию элементов, к элементам которой можно обратиться по индексу. Проще всего представлять такую коллекцию, как цепочку связанных между собой один за одним, пронумерованных объектов. Рассмотрим использование такой коллекции на примере кода

|  |
| --- |
| var metricValues = new List<int>() { 1, 2, 3, 45 }; metricValues.Add(6); *// добавление элемента*  *// добавление набора данных в коллекцию* metricValues.AddRange(new int[] { 7, 8, 9 });  metricValues.Insert(0, 42); *// вставляем на первое место в списке число 42*  metricValues.RemoveAt(1); *// удаляем второй элемент (подсчет идет с нуля)*  foreach (int i in metricValues) {  Console.WriteLine(i); } |

Операция добавления Add добавляет элемент в конец коллекции. Можно вставить элемент в определенное место, таким образом, что все элементы с индексом больше того, куда вставляем, как бы подвинутся дальше (увеличат свои индексы на 1). Удаление элемента RemoveAt по индексу приводит к аналогичным (только в обратную сторону) последствиям.

Доступ к элементам коллекции по индексу является простой для компьютера операцией, однако для того чтобы найти элемент с определенным значением придется в худшем случае перебрать все элементы в коллекции. Таким образом данная коллекция хорошо подходит под задачи хранения списков объектов без частой необходимости поиска определенного элемента без знания его индекса.

# Коллекция IDictionary<TKey, TValue>

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом структуры KeyValuePair<TKey, TValue>. Благодаря свойствам Key и Value, которые есть у данной структуры, мы можем получить ключ и значение элемента в словаре.

Рассмотрим на примере использование словарей:

|  |
| --- |
| *// создаем словарь* var computerNameToCoresCountMap = new Dictionary<string, int>(5); *// добавляем в словарь пары ключ-значение* computerNameToCoresCountMap.Add("Machine1", 1); computerNameToCoresCountMap.Add("Pentium3", 1); computerNameToCoresCountMap.Add("8086", 1); computerNameToCoresCountMap.Add("Xeon", 24); computerNameToCoresCountMap.Add("Chimera", 128); computerNameToCoresCountMap.Add("AppleSaysHelloToIntel", 128); *// проходимся по всему словарю* foreach (var keyValuePair in computerNameToCoresCountMap) {  Console.WriteLine($"{keyValuePair.Key} - {keyValuePair.Value}"); }  *// получение элемента по ключу* int coresCountInShinyProcessor = computerNameToCoresCountMap["8086"]; *// изменение объекта* computerNameToCoresCountMap["AppleSaysHelloToIntel"] = 256; *// удаление по ключу* computerNameToCoresCountMap.Remove("Machine1"); |

Кроме того, в словаре есть доступные для чтения коллекции Keys и Values, пройдясь по которым в цикле можно получить список ключей в словаре и список значений соответственно.

Коллекция словарь, в отличии от коллекции списка позволяет искать элемент по ключу в среднем за константное время, что на больших объемах хранимых данных является гораздо более быстрым, чем линейный поиск по всему списку. Такое поведение в коллекции достигается за счет особого её устройства. Коллекцию словарь лучше всего использовать для хранения соответствия значений некоторому произвольному ключу.

# DDD

DDD (domain driven development) – или же предметно ориентированное программирование, это подход, на который рекомендуется опираться при проектировании схемы классов внутри своего приложения. Областью (или доменом) в данной концепции называют всю предметную область в которой создается программный продукт. Модель описывает отдельные аспекты предметной области и может быть использована как решение практической задачи. Описывая модель в терминах предметной области (проектируя классы в соответсвии с тем, как устроена предметная область) в рамках ограниченного контекста мы можем достигнуть очень хорошей архитектуры приложения. Проще говоря, следует всегда начинать проектирование с анализа предметной области, и все классы в приложении проектировать и называть в соответствии с предметной областью.

# ORM

Как мы убедились в прошлом уроке, можно писать простые запросы из кода приложения, но каждый раз при доставании или упаковке данных в запрос, нам приходилось как бы вручную перекладывать все поля объектов из кода в запросы к базе данных. Чтобы такого избегать, существует технология ORM.

ORM это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая “виртуальную объектную базу данных”, то есть все данные из таблиц проецируются в объекты, а отношения между таблицами в специальные объекты или взаимоотношения между объектами.

Задачей ORM является обеспечение работы с данными в терминах классов, а не таблиц данных, и, напротив, преобразовать термины и данные классов в данные, пригодные для хранения в СУБД. Необходимо также обеспечить интерфейс для CRUD-операций над данными. В общем, необходимо избавиться от необходимости писать SQL-код для взаимодействия в СУБД.

Существует множество различных ORM для разных языков программирования (EF Core, NHibernate, etc.), но в рамках данного курса мы рассмотрим работу с ORM под названием Dapper.

# Dapper

Dapper представляет собой очень легковесную и быструю ORM. Данная ORM максимально приближена к простым SQL запросам к СУБД, однако позволяет программисту не писать лишний код по преобразованию данных из СУБД в объектное представление.

Разберемся, как подключить Dapper и перепишем на него наши репозитории. Для начала, добавьте через nuget пакет Dapper в проект сервис агента сбора метрик. Теперь перепишем код нашего репозитория следующим образом:

|  |
| --- |
| using MetricsAgent.Models; using System.Collections.Generic; using Dapper; using System.Linq; using System.Data; using System.Data.SQLite;  namespace MetricsAgent.DAL {  public class CpuMetricsRepository : ICpuMetricsRepository  {  *// строка подключения*  private const string ConnectionString = @"Data Source=metrics.db; Version=3;Pooling=True;Max Pool Size=100;";    *// инжектируем соединение с базой данных в наш репозиторий через конструктор*  public CpuMetricsRepository()  {  *// добавляем парсилку типа TimeSpan в качестве подсказки для SQLite*  SqlMapper.AddTypeHandler(new TimeSpanHandler());  }   public void Create(CpuMetric item)  {  using (var connection = new SQLiteConnection(ConnectionString))  {  *// запрос на вставку данных с плейсхолдерами для параметров*  connection.Execute("INSERT INTO cpumetrics(value, time) VALUES(@value, @time)",   *// анонимный объект с параметрами запроса*  new {   *// value подставится на место "@value" в строке запроса*  *// значение запишется из поля Value объекта item*  value = item.Value,    *// записываем в поле time количество секунд*  time = item.Time.TotalSeconds   });  }  }   public void Delete(int id)  {  using (var connection = new SQLiteConnection(ConnectionString))  {  connection.Execute("DELETE FROM cpumetrics WHERE id=@id",  new  {  id = id  });  }  }   public void Update(CpuMetric item)  {  using (var connection = new SQLiteConnection(ConnectionString))  {  connection.Execute("UPDATE cpumetrics SET value = @value, time = @time WHERE id=@id",  new  {  value = item.Value,  time = item.Time.TotalSeconds,  id = item.Id  });  }  }   public IList<CpuMetric> GetAll()  {  using (var connection = new SQLiteConnection(ConnectionString))  {  *// читаем при помощи Query и в шаблон подставляем тип данных*  *// объект которого Dapper сам и заполнит его поля*  *// в соответсвии с названиями колонок*  return connection.Query<CpuMetric>("SELECT Id, Time, Value FROM cpumetrics").ToList();  }  }   public CpuMetric GetById(int id)  {  using (var connection = new SQLiteConnection(ConnectionString))  {  return connection.QuerySingle<CpuMetric>("SELECT Id, Time, Value FROM cpumetrics WHERE id=@id",  new {id = id});  }  }  } } |

Прежде всего, стоит заметить, что мы добавили некоторый код преобразования типа TimeSpan. Дело в том, что база данных SQLite и Dapper не очень умеют преобразовывать стандартный TimeSpan и нам пришлось зарегистрировать собственный конвертер данного типа. Рассмотрим код этого конвертера, который был определен в отдельном классе:

|  |
| --- |
| using Dapper; using System.Data; using System;  namespace MetricsAgent.DAL {  *// задаем хэндлер для парсинга значений в TimeSpan если таковые попадутся в наших классах моделей*  public class TimeSpanHandler : SqlMapper.TypeHandler<TimeSpan>  {  public override TimeSpan Parse(object value)  => TimeSpan.FromSeconds((long)value);   public override void SetValue(IDbDataParameter parameter, TimeSpan value)  => parameter.Value = value;  } } |

Дальше сам репозиторий содержит гораздо меньше кода, чем было до этого. Теперь нам не приходится вручную читать поля из базы данных, преобразовывая данные к нужному виду. Достаточно теперь лишь сказать Dapper’у, в объект какого типа мы хотим преобразовать данные, которые достали из таблицы.

Таким образом Dapper избавляет нас от необходимости писать лишний код. Мы воспользовались некоторыми методами расширения, рассмотрим их подробнее.

* Execute выполняет команду и возвращает количество строк, которые были затронуты в ходе выполнения запроса. В нашем случае мы просто вставляли и модифицировали данные этой командой
* Query запрашивает результат и преобразует его в объект типа указанного как обобщенный
* QuerySingle делает все тоже самое что и Query за исключением того, что достает из всего результирующего набора данных лишь первый, и выбросит исключение, если в наборе данных будет более одного элемента. Если предполагается, что данных может быть в результате выполнения запроса больше, чем один, следует воспользоваться QueryFirst.

Попробуйте запустить отладку приложения, выполнить запросы к сервису и убедиться в том, что данные успешно сохраняются в СУБД и достаются из неё.

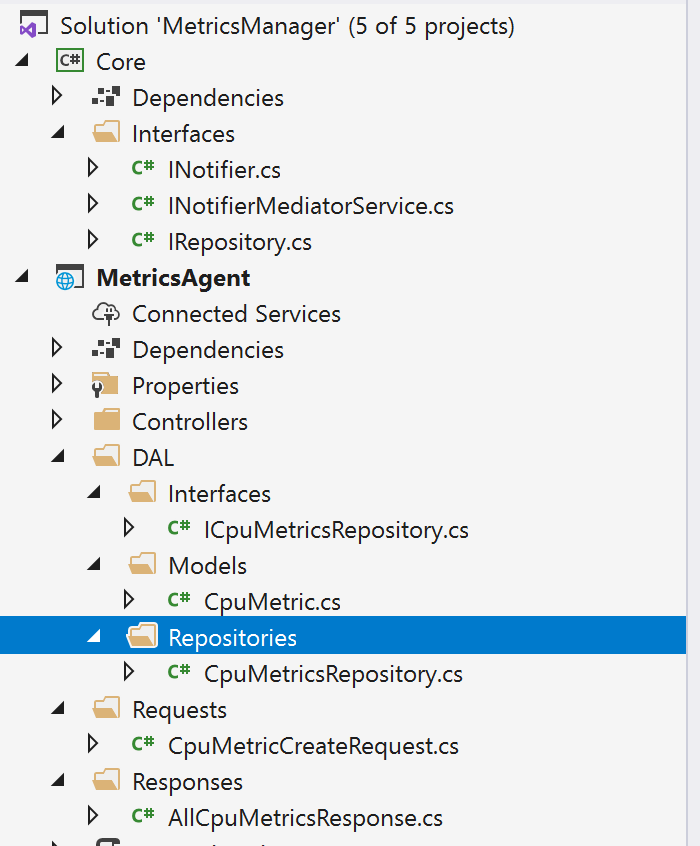
# Бизнес-логика приложения

В нашем приложении сбора метрик мы уже можем проследить некоторые слои, в которых будет логика приложения. Например, у нас явно имеется слой, ответственный за работу с данными в СУБД, у нас имеется общий код репозитория и конкретные репозитории. В случае, если наша бизнес-логика будет предполагать работу со сразу несколькими источниками данных из разных репозиториев, такой код можно выделить в отдельный слой “бизнес-сервисов”. Данный код необходимо выделять в отдельные классы, для того, чтобы их было легче протестировать, а как следствие, для более удобной структурной организации кода.

## Практика лучшего расположения кода по папкам в проекте

Для общего кода лучше всего выделить отдельную сборку. Это может быть проект Class Library. В результате сборки этого проекта, появится dll файл, который будет подключаться к нашему приложению во время исполнения. Можно назвать такую сборку Core и вынести в неё интерфейс репозитория, так как он может понадобиться как в проекте агента сбора метрик, так и в менеджере метрик.

Внутри проекта микросервиса лучше всего выделить папки, в которых будут лежать файлы интерфейсов и конкретных классов в соответствии с их зонами ответственности:



На примере рисунка выше, можно посмотреть, как примерно лучше всего располагать код в микросервисе. В сборке Core выделим общие для всех проектов интерфейсы. В проекте самого сервиса выделим папку DAL (data access layer) в которой расположим классы и интерфейсы для работы с СУБД. Так же выделим папки Requests и Responses, в которых будем хранить DTO объекты запросов и ответов контроллеров сервиса соответственно.

Перенесите свои уже написанные файлы в соответствующие папки.

Важно заметить, что на каждом из слоев приложения у нас есть свой объект DTO, в который мы преобразуем данные из другого слоя. DTO - это data transfer object, классы, которые ответственны за хранение данных, с которым работает код на определенном слое приложения. Рекомендуется на каждом слое приложения держать свои DTO объекты, даже если поля их будут совпадать вплоть до названий. Такой подход позволяет легче модифицировать код под часто меняющиеся требования.

# Паттерн Медиатор (Посредник)

Шаблон “Посредник” восходит к 1994 году в знаменитой книге “Шаблоны проектирования: элементы объектно-ориентированного программного обеспечения многократного использования”. В последнее время этот шаблон является очень популярным, и Майкрософт рекомендует строить архитектуру своих приложений с его использованием.

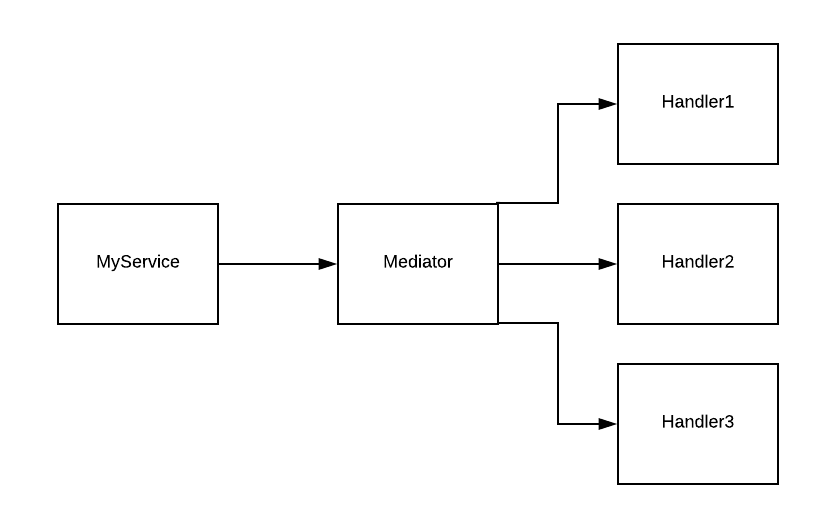
Суть паттерна Медиатор состоит в том, чтобы “определить объект, который инкапсулирует способ взаимодействия набора объектов”. Он способствует слабой связи, не позволяя объектам явно ссылаться друг на друга, и позволяет изменять их взаимодействие независимо друг от друга. Классы клиентов могут использовать Медиатор для отправки сообщений другим клиентам и могут получать сообщения от других клиентов через событие в классе Медиатора.

Иначе говоря, у нас в коде приложения будет объект, который отвечает за следующие функции:

* Инкапсулирует взаимодействие объектов. Таким образом, очевидно, что он может обрабатывать передачу “сообщений” между объектами;
* объекты не ссылаются друг на друга, а на Медиатора.

И в этом заключается вся суть паттерна Медиатор.

Построим схему работы этого паттерна.



Зачем нужен Медиатор, если можно вызывать обработчики непосредственно из сервиса? Рассмотрим на примере кода. Представим, что у нас есть некоторый сервис, который содержит в себе три обработчика:

|  |
| --- |
| class MyService {  *// регистрируем обработчики*  private readonly Handler1 \_handler1;  private readonly Handler2 \_handler2;  private readonly Handler3 \_handler3;     public MyService(Handler1 handler1, Handler2 handler2, Handler3 handler3)  {  \_handler1 = handler1;  \_handler2 = handler2;  \_handler3 = handler3;  }   public void DoSomething()  {  *// делаем вызовы обработчиков*  \_handler1.Notify(new HandlerArgs());  \_handler2.Notify(new HandlerArgs());  \_handler3.Notify(new HandlerArgs());  } } |
|  |

Теперь при добавлении любого нового обработчика, нам придется модифицировать класс сервиса, чтобы добавить в него возможность вызова нового обработчика.

|  |
| --- |
| class MyService {  private IMediator \_handlerMediator;   public MyService(IMediator handlerMediator)  {  \_handlerMediator = handlerMediator;  }   public void DoSomething()  {  *// тут можно написать полезный код, который что то делает*    *// уведомляем остальные хэндлеры*  \_handlerMediator.Notify(new HandlerArgs());  } }   class HandlerMediator {  private readonly Handler1 \_handler1;  private readonly Handler2 \_handler2;  private readonly Handler3 \_handler3;    public HandlerMediator(Handler1 handler1, Handler2 handler2, Handler3 handler3)  {  \_handler1 = handler1;  \_handler2 = handler2;  \_handler3 = handler3;  }   public void Notify(HandlerArgs handlerArgs)  {  \_handler1.Notify(handlerArgs);  \_handler2.Notify(handlerArgs);  \_handler3.Notify(handlerArgs);  } } |

Выгода в том, что при изменении, добавлении или удалении обработчиков сам сервис не меняется. Минус в том, что мы перекладываем нагрузку на посредника, теперь его задача - управлять обработчиками и тем, как они получают уведомления. Но в этом есть смысл! Чтобы иметь класс, единственная задача которого уведомлять клиентов (Принцип единой ответственности), и наш сервис, который действительно не заботится о деталях реализации этих обработчиков, может продолжить свою работу.

Позже мы увидим, как мы используем DI, чтобы действительно помочь нам облегчить нагрузку с обоих классов, и при этом по-прежнему придерживаться сути шаблона Медиатор.

Выделим интерфейс для уведомляемого и сделаем реализацию.

|  |
| --- |
| public interface INotifier {  void Notify(); }  public class Notifier1 : INotifier {  public void Notify()  {  Debug.WriteLine("Debugging from Notifier 1");  } } |

И зарегистрируем его в контейнере:

|  |
| --- |
| public void ConfigureServices(IServiceCollection services) {  services.AddTransient<INotifier, Notifier1>();  ... } |

Инжектируем эту зависимость в контроллер, возьмем тестовый:

|  |
| --- |
| public class HomeController : ControllerBase {  private readonly IEnumerable<INotifier> \_notifiers;   public HomeController(IEnumerable<INotifier> notifiers)  {  \_notifiers = notifiers;  }   [HttpGet("")]  public ActionResult<string> NotifyAll()  {  \_notifiers.ToList().ForEach(x => x.Notify());  return "Completed";  } } |

Таким образом мы получим уведомление в консоли о том, что Inotifier1 выполнился.Но лучше, если у нас будет выделен отдельный сервис для этого. Давайте это сделаем.

|  |
| --- |
| public interface INotifierMediatorService {  void Notify(); }  public class NotifierMediatorService : INotifierMediatorService {  private readonly IEnumerable<INotifier> \_notifiers;   public NotifierMediatorService(IEnumerable<INotifier> notifiers)  {  \_notifiers = notifiers;  }   public void Notify()  {  \_notifiers.ToList().ForEach(x => x.Notify());  } } |

Зарегистрируем этот сервис как мы регистрировали обработчики выше и таким образом мы получим в контроллер ровно одну инжектированную зависимость

|  |
| --- |
| public class HomeController : ControllerBase {  private readonly INotifierMediatorService \_notifierMediatorService;   public HomeController(INotifierMediatorService notifierMediatorService)  {  \_notifierMediatorService = notifierMediatorService;  }   [HttpGet("")]  public ActionResult<string> NotifyAll()  {  \_notifierMediatorService.Notify();  return "Completed";  } } |

Теперь если мы зарегистрируем несколько обработчиков INotifier то выполняется каждый из них последовательно в порядке регистрации. Это не всегда удобно, так как часто нам не требуется выполнять все действия последовательно, поэтому можно сделать интерфейс INotifier следующим.

|  |
| --- |
| namespace Core {  interface INotifier  {  void Notify();   bool CanRun();  } } |

Для каждой реализации интерфейса мы будем спрашивать, нужно ли его запускать. Сделаем проверку на это в коде сервиса.

|  |
| --- |
| \_notifiers.Where(x => x.CanRun()).ToList().ForEach(x => x.Notify()); |

Так можно выделить конкретные зоны ответственности у каждого Notifier. На примере нашего приложения, можно выделить конкретный Notifier, ответственный за работу с метриками CPU.

Существует рекомендованная майкрософтом реализация данного паттерна в виде nuget пакета MediatR. Однако для нашей задачи он является слишком “объемным”. Нам важно понимать лишь то, как работает данный паттерн, чтобы при чтении чужого кода с его использованием не возникало вопросов относительно его работы; использовать в нашем приложении MediatR мы не станем.

# Автоматический маппинг объектов в приложении

Как вы могли заметить, довольно часто нам приходится писать код, который преобразует объекты одного слоя приложения в объекты другого слоя, но разниц между этими объектами, за исключением названия – нет.

Рассмотрим типичный пример, когда из репозитория мы достали объект одного типа, а в ответе нам приходится преобразовывать его в другой.

|  |
| --- |
| [HttpGet("all")] public IActionResult GetAll() {  IList<CpuMetric> 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 можно заметить, мы отдаем CpuMetricDto объекты внутри объекта AllCpuMetricsResponse, и все поля у объектов совпадают по названию. Мы могли бы это использовать для того, чтобы писать меньше кода по преобразованию объектов.

## Библиотека AutoMapper

Для автоматизации подобных преобразований существует библиотека AutoMapper. Подключите её через nuget (а так же библиотеку AutoMapper.Extensions.Microsoft.DependencyInjection). Давайте посмотрим, как может поменяться код контроллера с использованием этой библиотеки.

|  |
| --- |
| [HttpGet("all")] public IActionResult GetAll() { *// задаем конфигурацию для мапера. Первый обобщенный параметр -- тип объекта источника, второй -- тип объекта в который перетекут данные из источника*  var config = new MapperConfiguration(cfg => cfg.CreateMap<CpuMetric, CpuMetricDto>());  var m = config.CreateMapper();  IList<CpuMetric> metrics = repository.GetAll();  var response = new AllCpuMetricsResponse()  {  Metrics = new List<CpuMetricDto>()  };  foreach (var metric in metrics)  {  *// добавляем объекты в ответ при помощи мапера*  response.Metrics.Add(m.Map<CpuMetricDto>(metric));  }  return Ok(response); } |

Как видно из примера, теперь нам не придется писать каждый раз код по перекладыванию полей одного объекта в другой. Такой подход помогает уменьшить вероятность ошибки при написании рутинного кода в приложении. Однако создавать каждый раз новый маппер не совсем корректно. Рассмотрим, как лучше зарегистрировать маппер в контейнере и все конфигурации преобразований объектов.

Сначала добавим класс, в котором у нас будут содержаться все преобразования одного объекта в другой. Такой класс нужно наследовать от класса Profile для того, чтобы AutoMapper смог “подхватить” настройки из такого класса. Выделять отдельный класс с такими настройками необходимо для того, чтобы не замусоривать класс Startup лишней конфигурацией.

|  |
| --- |
| using AutoMapper; using MetricsAgent.Models; using MetricsAgent.Responses;  namespace MetricsAgent {  public class MapperProfile : Profile  {  public MapperProfile()  {  *// добавлять сопоставления в таком стиле нужно для всех объектов*   CreateMap<CpuMetric, CpuMetricDto>();  }  } } |

Теперь добавим регистрацию маппера в контейнере. Для этого в методе ConfigureServices класса Startup следует добавить следующий код:

|  |
| --- |
| var mapperConfiguration = new MapperConfiguration(mp => mp.AddProfile(new MapperProfile())); var mapper = mapperConfiguration.CreateMapper(); services.AddSingleton(mapper); |

Таким образом мы зарегистрировали наш маппер, и он может быть инжектирован в контроллер. Теперь метод контроллера, отдающий все метрики, может выглядеть следующим образом:

|  |
| --- |
| namespace MetricsAgent.Controllers {  [Route("api/[controller]")]  [ApiController]  public class CpuMetricsController : ControllerBase  {  private readonly ICpuMetricsRepository repository;   private readonly IMapper mapper;   public CpuMetricsController(ICpuMetricsRepository repository, IMapper mapper)  {  this.repository = repository;  this.mapper = mapper;  }   [HttpGet("all")]  public IActionResult GetAll()  {  IList<CpuMetric> metrics = repository.GetAll();   var response = new AllCpuMetricsResponse()  {  Metrics = new List<CpuMetricDto>()  };   foreach (var metric in metrics)  {  response.Metrics.Add(mapper.Map<CpuMetricDto>(metric));  }   return Ok(response);  }  } } |

# Практическое задание

* Перепишите все репозитории в приложении агента метрик использование Dapper вместо сырого чтения данных
* Настройте AutoMapper для всех остальных объектов в приложениях (преобразования из Dto в модели базы и обратно)

# Список используемой литературы

* <https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1>
* <https://www.dotnetperls.com/ilist>
* <http://www1.cs.columbia.edu/~lok/csharp/refdocs/System.Collections/types/IList.html>
* [https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0)2
* <https://github.com/StackExchange/Dapper>
* <https://dapper-tutorial.net/dapper>
* <https://refactoring.guru/design-patterns/mediator>

# Дополнительные материалы

* <https://www.geeksforgeeks.org/c-sharp-dictionary-with-examples/>
* <https://www.c-sharpcorner.com/UploadFile/mahesh/dictionary-in-C-Sharp/>
* <https://codewithshadman.com/mediator-pattern-csharp>