Темы урока

MVC

Middleware для API

С прошлого занятия мы помним, что нам нужно добавить `Middleware` в `HTTP Request Pipeline` в метод `Configure` класса `Startup`, а также добавить необходимые framework-сервисы в контейнер внутри метода `ConfigureServices` (также класса `Startup`).

ASP.NET Core MVC

Но, для начала, разберёмся с терминологией. Раньше для написания HTTP web-сервисов мы использовали ASP.NET Web API. Для написание клиентских приложений мы использовали ASP.NET MVC.

Это изменилось. В ASP.NET Core нет отдельного фреймворка для Web API. Фреймворки ASP.NET Web API и ASP.NET MVC объединены в один фреймворк ASP.NET Core MVC. Он позиционируется как фреймворк для разработки веб-приложений и веб-API на базе MVC.

И это важно! MVC — это, прежде всего, шаблон проектирования (design pattern), который использовался для обоих фреймворков. Чтобы построить наш API, нам нужно добавить ASP.NET Core MVC middleware.

Шаблон проектирования MVC (MVC design pattern)

Объяснения по слайду.

Метапакет Microsoft.AspNetCore.App

Немного о метапакетах чтобы владеть термином и иметь представление о метапакете Microsoft.AspNetCore.App.

Совместная работа

Добавление сервиса MVC и middleware MVC в HTTP Request Pipeline.

Добавляем в метод `ConfigureServices` класса `Startup` регистрацию нового сервиса:

```
services.AddMvc();
```

Затем идём в метод Configure того же класса и добавляем в middleware MVC в Request Pipeline (вместо вызова `app.Run`, который мы удаляем):

```
app.UseMvc();
```

Запускаем наше приложение. Ничего не работает. Потому что мы указали, что хотим использовать MVC pipeline, однако не подготовили ни Модели, ни Представления, ни Контроллера.

Добавление контроллера (controller) CitiesController

Давайте начнём с основного. Добавим контроллер. Мы будем посвятим первый API городам. Это будет своего рода справочник. **Согласно соглашению**, классы контроллеров располагаются в папке `Controllers`, так что давайте её создадим.

В папке `Controllers` создадим класс контроллера `CitiesController`. Все котроллеры наследуются от абстрактного базового класса `Controller`, входящего в пакет `Microsoft.AspNetCore.Mvc` и в наш метапакет `Microsoft.AspNetCore.App`.

Обратите внимание, что с помощью Visual Studio можно создать не просто пустой класс, а сразу специализированный контроллер, если нажать правой кнопкой мыши на папке `Controllers`. и выбрать пункт меню Add > Controller.

Сериализация/десериализация.

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Эта операция предназначена для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется десериализацией.

Формат JSON

Бегло рассказываем по слайду правила формата JSON

Можно указать, что детали хорошо изложены в RFC4627.

Объяснения по слайду.

Добавление экшена (action method) GetCities

Создадим в классе `CitiesController` метод GetCities, который будет возвращать константный список городов в текстовом виде в формате JSON. Возвращаемым типом данных для нашего метода будет `JsonResult`, этот класс возвращает в формате JSON всё, что ему передаётся в конструктор.

```
JsonResult GetCities()
{
      return new JsonResult(new List<object>
      {
            new { id = 1, Name = "Moscow" },
            new { id = 1, Name = "St.-Petersburg" },
            new { id = 1, Name = "New-York" }
      });
}
```

При написании HTTP-сервисов, таких как API, чтобы получить данные, мы должны отправить HTTP-запрос с HTTP-методом GET на URI, который ведёт (routes) к этому методу.

Чтобы сформировать такой запрос, для начала можно, конечно, воспользоваться и браузером, так как он прекрасно подходит как инструмент создания HTTP-запросов с методами GET и POST, однако, понимая, что рано или поздно нам придётся столкнуться и с другими методами, лучше сразу познакомиться поближе со специальными инструментами для создания и анализа HTTP-запросов.

Запустим Postman. Скорее всего мы будем ходить к нашему API по HTTPS, а поскольку SSL-сертификат у нас является самовыпущенным, необходимо сразу зайти в настройки Postman и отключить проверку достоверности SSL-сертификатов:

File > Settings > SSL certificates verification: OFF.

Теперь мы можем ввести в поле “Request URL” наш URL: https://localhost:44354/api/cities и нажать кнопку Send...

… и мы, конечно же не увидим наших городов, а увидим мы ответ 404 Not Found. Это произошло потому, что MVC фреймворк не знает, как сопоставить указанный URI и наш action-метод GetCities.

Маршрутизация (Routes)

Маршрутизация сопоставляет URI запроса с определённым методом (экшеном) контроллера чтобы, когда мы будем посылать HTTP-запрос, MVC разбирает (анализирует) URI и пытается сопоставить пришедший запрос с определённым контроллером и определённым экшеном (методом) этого контроллера.

Существует два способа, которыми можно задать необходимые маршруты

- Метод, основанный на соглашении (convention-based)
- Метод, основанный на атрибутах (attribute-based)

Convention-based routing

Маршрутизация, основанная на соглашении, выглядит следующим образом:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
      // ...
      app.UseMvc(config =>
            config.MapRoute(
                  name: "Default",
                  template: "{controller}/{action}/{id?}",
                  defaults: new { controller = "Home", action = "Index" })
      );
      // ...
}
```

Например, такой маршрут будет сопоставлять URI /**cities**/**index** (согласно наименованиям) экшену **Index** контроллера **Cities**Controller.

Convention-based vs Attribute-based routing

Маршрутизация, основанная **на соглашении**, обычно используется в тех случаях, когда фреймворк MVC используется **для веб-приложений**, возвращающих HTML-представления.

Команда разработчиков .NET Core не рекомендует использование этого метода определения маршрутов для написания API. Вместо этого **для API** рекомендуется использовать маршрутизацию, основанную **на атрибутах**.

Attribute-based routing

Маршрутизация, основанная на атрибутах, позволяет задавать маршруты через атрибуты, устанавливаемые как на уровне контроллера, так и на уровне экшена:

```
public class CitiesController : Controller
{
      [HttpGet("/api/cities")]
      public JsonResult GetCities() { // return cities... }
}
```

Запускаем Postman. Выполняем GET запрос по URL https://localhost:443XX/api/cities и убеждаемся, что мы получаем в ответ JSON

В примере выше маршрут прописан на уровне экшена, однако если у нас будет много экшенов внутри данного контроллера, мы захотим, чтобы они выглядели консистентно не нарушая DRY

```
[Route("/api/cities")]
public class CitiesController : Controller
{
      [HttpGet()]
      public JsonResult GetCities() { // return cities... }
}
```

Нужно показать, что здесь также допускается шаблонизация, например так:

```
[Route("/api/[controller]")]
```

Однако, я не сторонник так делать, так как, на мой вкус, это ухудшает читаемость кода.

Совместная работа: Создаём Модель

В нашем предыдущем примере мы возвращали JSON напрямую. Это хорошо для быстрой демонстрации, однако, это решение не подходит для серьёзного проекта.

Вместо этого мы будем пользоваться моделями.

Создаём папку `Models` (по соглашению, это папка должна называться именно так).

В ней создаём класс `City`:

```
public class City
{
      public int Id { get; set; }
      public string Name { get; set; }
```

```
        public string Description { get; set; }
        public int NumberOfPointsOfInterest { get; set; }
}
```

При создании свойств можно использовать сниппеты Visual Studio 2017, например:

- `prop  + [Tab] + [Tab]` — для создания свойства `{ get; set; }`
- `propg + [Tab] + [Tab]` — для создания свойства `{ get; private set; }`
- `ctor  + [Tab] + [Tab]` — для создания конструктора объекта

Совместная работа: Создаём In-memory хранилище

Создадим папку `DataStore`, а в ней класс хранилища `CitiesDataStore` который будет использоваться вместо базы данных нашим API.

Методы контроллера будут работать с этим классом как с полноценным хранилищем.

Само хранилище будет работать на базе предварительно заполненного списка, однако мы реализуем паттерн синглтон для этого класса:

- Закрываем конструктор,
- Добавляем приватное статическое поле этого же класса,
- Создаём метод, который будет создавать и возвращать наше статическое поле.

Полный код `CitiesDataStore.cs` можно подсмотреть в проекте `L22_C02_empty_asp_net_core_app_final`.

Далее вносим соответствующие изменения в контроллер, чтобы он использовал наше новое хранилище, которое возвращает типизированные, а не анонимные объекты:

```
[HttpGet]
public JsonResult GetCities()
{
        var citiesDataStore = CitiesDataStore.GetInstance();
        var cities = citiesDataStore.Citites;

        return new JsonResult(cities);
}
```

Можно пояснить, чем хорош синглтон в этом случае. В других методах контроллера (а в потенции, и в других контроллерах) мы будем также работать с хранилищем и здесь важна его единственность.

Запускаем приложение > Дёргаем запрос в Postman > Радуемся, если всё корректно. В противном случае, фиксим где налажали :)

Совместная работа: Добавляем Экшен получения города по ID

Давайте напишем экшен GetCity:

```
[HttpGet("/api/cities/{id}")]
```

```
public JsonResult GetCity(int id)
{
      var citiesDataStore = CitiesDataStore.GetInstance();
      var city = citiesDataStore.Citites.Where(x => x.Id == id).FirstOrDefault();

      return new JsonResult(city);
}
```

Запускаем приложение > Дёргаем запрос в Postman > Радуемся.

Обращаем внимание, что мы уже написали префикс `/api/cities` в маршруте контроллера, поэтому в атрибуте экшена мы можем оставить просто `[HttpGet("{id}")]`.

Запускаем приложение > Дёргаем запрос в Postman > Радуемся ещё больше.

Отмечаем, что совпадение имени параметра метода и имени шаблона в атрибуте не случайно:

```
[HttpGet("/api/cities/{id}")]
public JsonResult GetCity(int id)
```

Именно благодаря этому значение из строки запроса попадает в параметр нашего экшена.

Корректный результат для отсутствующих данных

Давайте посмотрим, что произойдёт, если мы запросим ID несуществующего в нашем хранилище объекта.

Запускаем приложение > Дёргаем запрос в Postman > Демонстративно расстраиваемся :(

Мы получаем статус код `200 OK` и строку `null`.

Это не соответствует правилам разработки API. Вместо этого мы должны вернуть статус код `404 Not Found` и пустое либо описательное тело ответа.

Почему вообще корректный код статуса — это так важно?

Соответствующие коды статусов необходимы для того, чтобы дать возможность пользователю API трактовать однозначно тот или иной ответ. Например, что его запрос обработан как ожидалось, или что-то пошло не так, и если что-то пошло не так, то что именно? Его ли собственная это ошибка, или это ошибка самого API.

Например, если API будет всегда возвращать `200 OK`, то пользователь не сможет узнать о проблеме, если что-то не так.

Также предоставьте, что на любую ошибку мы будем возвращать `500 Internal Server Error`. Пользователь не будет понимать, он ли ошибся в запросе или это проблема API.

На самом деле, существует огромное количество кодов статуса, и API не обязательно поддерживать их все. Давайте посмотрим на основные, использующиеся при разработке API наиболее часто.

Уровни и виду кодов статуса

(показываем слайд со статусами и объясняем по каждому пункту)

Существует пять уровней статусов:

- **Уровень 1 (статусы 1XX)**
  Это информационные статусы, которые не являются частью стандарта HTTP1.0, так что они не используются в API.

- **Уровень 2 (статусы 2XX)**
  Статусы этой группы показывают, что запрос прошел успешно
  - `200 - OK` для успешного GET-запроса
  - `201 - Created` для успешного запроса на создание нового ресурса
  - `204 - No Content` для успешного запроса, который не должен ничего вернуть, например на удаление ресурса.

- **Уровень 3 (статусы 3XX)**
  Используется для перенаправлений. Большинство API не оперируют этими статусами, так как не предусматривают перенаправлений.

- **Уровень 4 (статусы 4XX)**.
  Клиентские ошибки
  - `400 - Bad Request` для информирования клиента о том, что в его запросе обнаружена ошибка. Например тело запроса оказалось с некорректным форматом JSON и не смогло распарситься.
  - `401 - Unauthorized` для информирования о том, что запрос не был аутентифицирован. Иными словами, пользователь должен был представиться, чтобы выполнить этот запрос.
  - `403 - Forbidden` для информирования о том, что аутентификация прошла успешно, однако авторизация — нет. Пользователь представился, но ему не хватило прав доступа к ресурсу.
  - `404 - Not Found` информирует пользователя о том, что запрошенный ресурс не существует.
  - `409 - Conflicts` используется когда произошёл конфликт, например, при попытке изменить ресурс, состояние которого было изменено с момента последнего чтения.

- **Уровень 5 (статусы 4XX)**.
  Серверные ошибки
  - `500 Internal Server Error` - означает, что на сервере произошла ошибка и клиенту ничего не остаётся, кроме как попробовать выполнить этот запрос позднее.

Совместная работа: возвращение правильных кодов статуса

Итак, давайте посмотрим на то, что мы возвращаем в наших экшенах. Мы возвращаем `JsonResult`. Смотрим на определение `JsonResult`. Он наследуется от `ActionResults`. А если посмотреть на определение `ActionResults`, то мы увидим, что это реализация интерфейса `IActionResults`,

используемая по умолчанию. `IActionResults` определяет контракт, представляющий результат выполнения экшена.

На самом деле мы не хотим ограничивать наш API настолько и возвращать результат только в виде JSON. Чуть далее мы научимся учитывать пожелания пользователя по формату передаваемых сообщений с помощью заголовка `Accept`. Чуть ниже мы поговорим об этом.

Пока остановимся на том, что возвращать мы хотим не только корректный JSON, но также и корректный код статуса. Этого, разумеется, можно достичь и используя `JsonResult`.

В методе GetCities меняем строку возвращения результата

```
return new JsonResult(cities);
```

на возвращение результата через временную переменную. И перед тем, как вернуть её, устанавливаем соответствующий код статуса:

```
var result = new JsonResult(cities);
result.StatusCode = 200;
return result;
```

Однако, получившаяся конструкция довольно-таки громоздкая. ASP.NET Core содержит набор встроенных вспомогательных методов контоллера, возвращаю.щие `IActionResult`:

- Есть метод NotFound для случая, если ресурс не найден,
- Есть метод BadRequest для случая ошибки в запросе
- и так далее.

Давайте поменяем сигнатуру наших экшенов таким образом, чтобы они возвращали `IActionResult` вместо `JsonResult`.

Остановимся подробнее на методе `GetCity`. Для начала попробуем найти город с запрошенным идентификатором. Если он не найден, воспользуемся вспомогательным методом `NotFound`. Если город найден, воспользуемся методом `Ok` чтобы вернуть статус `200 - OK`.

```
[HttpGet("{id}")]
public IActionResult GetCity(int id)
{
      var citiesDataStore = CitiesDataStore.GetInstance();
      var city = citiesDataStore.Citites
            .Where(x => x.Id == id)
            .FirstOrDefault();

      if (city == null)
            return NotFound();

      return Ok(city);
}
```

Если произойдёт исключение, сервер автоматически вернёт `500 Internal Server Error` в ответ.

Соответственным образом меняем и метод GetCities:

```
[HttpGet]
public IActionResult GetCities()
{
      var citiesDataStore = CitiesDataStore.GetInstance();
      return Ok(citiesDataStore.Citites);
}
```

Обратите внимание, что тут не будет возможности получить `404 - Not Found` результат, так как даже пустая коллекция городов - это правильный ответ. Коллекция найдена, просто она пуста.

Важно заметить, что мы уже нигде явно не преобразовываем данные к формату JSON. Но если запустить наше приложение и подёргать наш API, то мы увидим, что данные по-прежнему приходят в этом формате.

Запускаем.
Дергаем URL https://localhost:443XX/api/cities/3.
Видим в ответе код статуса `404 - Not Found`. Радуемся :)

Обратите внимание, что мы уже не возвращаем никого `null` в теле ответа. Но мы можем добавить возвращение текста с кодом ошибки в ответ в случае ошибки. Для этого в ASP.NET Core есть соответствующий middleware - `UseStatusCodePages`. Добавим его в наш метод `Configure` класса `Startup`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
      if (env.IsDevelopment())
      {
            app.UseDeveloperExceptionPage();
      }

      app.UseStatusCodePages();
      app.UseMvc();
}
```

Запускаем.
Дергаем URL https://localhost:443XX/api/cities/3.
Видим в ответе код статуса `404 - Not Found` и текст `Status Code: 404; Not Found`
Радуемся :)

Можно обратить внимание на то, что заголовок `Content-Type` ответа уже не `application/json`, а `text/plain`.

Совместная работа: Учитываем заголовок Accept (Content Negotioation)

ASP.NET Core по умолчанию десериализует из и сериализует в формат JSON.

Однако хороший API должен быть гибким и обеспечивать возможность работать с собой как можно более широкому кругу клиентов. Не факт, что все клиенты работают с JSON и части из них может потребоваться ряд непростых доработок для того, чтобы в итоге подключиться к JSON API.

И в ASP.NET Core это делается очень просто.

Для начала открываем Postman и устанавливаем в запросе заголовок `Accept` в значение `application/json`. Выполняем запрос https://localhost:443XX/api/cities.

Ответ приходит в формате JSON, как и раньше.

Теперь меняем значение заголовка `Accept` на значение `application/xml` и запускаем повторно. Ответ приходит… снова в формате JSON :(Ну, пора привыкнуть, что ничего в этом мире не бывает совсем уж бесплатно) Давайте приложим некоторые усилия, чтобы наш API стал в 2 раза гибче и покрыл оставшиеся 20% потенциальных пользователей, которые по каким-то причинам хотят использовать наш API в пользуясь форматом XML.

Обратимся к методу `ConfigureServices` класса `Startup` и добавим соответствующие настройки в сервис MVC, добавляемый там к контейнеру:

```
public void ConfigureServices(IServiceCollection services)
{
      services.AddMvc()
            .SetCompatibilityVersion(
                  CompatibilityVersion.Version_2_2)
            .AddMvcOptions(
                  o => o.OutputFormatters.Add(
                        new XmlDataContractSerializerOutputFormatter()));
}
```

Запускаем. Пробуем с различными значением для заголовка `Accept` и без него. Радуемся :)

Совместная работа: Создание новых ресурсов

Мы добавим возможность заводить новые города через наш API.

Как мы сегодня узнали, для этого необходимо пользоваться методом POST.

Создаём новую модель

Однако сначала давайте обратимся к нашей модели. Вспоминаем, что мы делали для Reminder’ов, когда добавляли возможность заводить, получать и удалять их. У нас были разные модели, так как для разных случаев, набор передаваемых полей был отличным.

Как минимум между созданием и получением есть разница в том, что при создании ID является лишним. ID это нечто такое, что управляется серверной стороной и не может задаваться со стороны клиента. Поэтому нам нужно сделать добавить новую модель, которую мы назовём `CityCreateModel`. А нашу предыдущую модель для консистентности придётся переименовать в `CityGetModel`.

Лирическое отступление о несовершенстве порождаемого нами мира :)

После всех переименований можно заглянуть в класс `CitiesDataStore`. Мы видим, что там List<City> поменялся на List<CityGetModel>. Это

- Во-первых, не очень красиво выглядит
- Во-вторых, в реальном мире, скорее всего, не будет являться правдой.

По хорошему, в нашей базе данных сущности должны быть описаны вообще своими классами с внешним API связанными очень посредственно. Хорошим упражнением здесь было бы вынести слой доступа к данным в отдельную библиотеку, по примеру нашего приложения с ремайндером. Но поскольку сейчас мы фокусируемся не на архитектуре, я оставлю это так. Надеюсь, вы тоже заметили это маленькое несовершенство.

Создаём новый экшн `CreateCity` контроллера `CitiesController`:

```
[HttpPost()]
public IActionResult CreateCity([FromBody] CityCreateModel city)
{
}
```

Проверяем, что в запросе корректно передана модель нового города. Если ASP.NET Core не сумеет распарсить модель из тела запроса, то переменная `city`будет равна `null`. И это будет проблемой клиента. Так что мы сразу проверяем это и если `city == null`, возвращаем в ответ результат метода `BadRequest`.

```
if (city == null)
{
      return BadRequest();
}
```

Далее, чтобы реально добавить новый город в список наших городов нам нужно уже на стороне сервера озаботиться генерацией ID таким образом, чтобы сохранить их уникальность.

Мы воспользуемся самым простым с точки зрения алгоритма методом - всегда будем брать максимальное значение ID уже существующих городов и добавлять к ним единицу.

```
var citiesDataStore = CitiesDataStore.GetInstance();
int newCityId = citiesDataStore.Citites.Max(x => x.Id) + 1;
```

Далее создаём модель города, необходимую для нашего хранилища:

```
var newCity = new CityGetModel

{
      Id = newCityId,
      Name = city.Name,
      Description = city.Description,
      NumberOfPointsOfInterest = city.NumberOfPointsOfInterest
};
```

```
citiesDataStore.Citites.Add(newCity);
```

И осталось вернуть соответствующий результат используя метод `Created`.
Но тут оказывается всё не так просто. Мы должны не просто сказать “всё нормально”, но также вернуть и URI, по которому наш новый ресурс может быть доступен.. Поэтому мы воспользуемся разновидностью метода `Created`, а именно: `CreatedAtRoute`.

Он требует в качестве параметра имя имеющегося маршрута (раз мы создаём ресурс, значит, наверное, мы его и возвращаем по какому-то маршруту). Но у нас пока нет именованных марнщрутов. Поэтому нам надо пойти и добавить имя в атрибут маршрута нашего метода GetCity:

```
[HttpGet("{id}", Name = "GetCity")]
public IActionResult GetCity(int id)
```

Теперь мы можем вернуться и дописать в нашем экшене три параметра:

- имя маршрута, по которому можно получить наш объект,
- параметры для формирования этого маршрута в виде анонимного объекта,
- а также сам созданный объект

```
return CreatedAtRoute("GetCity", new { id = newCityId }, newCity);
```

Можем попробовать, что у нас получилось. Дёргаем метод POST по URL с телом типа RAW (application/json):

```
{
    "name": "Munich",
    "description": "A global centre of art, science, and technology in Germany",
    "numberOfPointsOfInterest": 0
}
```

и получаем в статус `204 - Created`.
В теле ответа нам приходит только что созданный объект.
Дополнительно приходит заголовок `Location` со значением https://localhost:443XX/api/cities/3.

Swagger

Живая демонстрация

Скачать последний NuGet-пакет Swashbuckle.AspNetCore (с включённой галочкой Include Pre-Releases).

Добавить в метод `ConfigureServices` класса `Startup`:

```
services.AddSwaggerGen(c =>
{
      c.SwaggerDoc(
            "v1",
            new OpenApiInfo { Title = "Cities API", Version = "V1" });
});
```

Добавить в конец метода `Configure` класса `Startup`:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
      c.SwaggerEndpoint("/swagger/v1/swagger.json", "Cities API V1");
});
```

Запустить, открыть URL /swagger и радоваться :)

Домашнее задание

Дописать методы по замене (PUT) и удалению (DELETE) конкретного города.