Темы урока

Разбор домашней работы

Объяснить домашнюю работу на примере.

Обратить внимание, что PATCH — это нетривиальная задача.

Рассказать коротко о JSON Patch RFC 6902. Показать как это реализовано у меня в примере выполнения домашней работы (`Lessons/23/HomeWork/L2_HomeWork.sln`).

Валидация данных

Задачи валидации?

Когда мы задумываемся о валидации данных в нашем API, на самом деле мы ставим перед собой 3 задачи:

1. Определить правила валидации
2. Проверить данные на валидность согласно определённым правилам
3. В случае невалидных данных сообщить пользователю нашего API о проблемах с входными данными

До этого мы обходились всевозможными кодами статуса и научились сообщать пользователю на чьей стороне проблема. Однако часто кроме кода статуса в случае проблемы пользователю API посылают дополнительную информацию о проблеме. Особенно полезными эти сообщения бывают в случае описания проблем с пользовательскими данными.

Например, у пользователя есть ваша модель городов. Там есть поля, за которые отвечает логика нашего приложения:

- Идентификатор города в нашем хранилище

Но также там есть поля, которые пользователь может определять или менять:

- Название города
- Описание города
- Число достопримечательностей

И в реальной жизни эти данные лежат в каком-то хранилище. Возможно, это база данных, но может быть и просто текстовый или XML-файл. Для быстрого поиска по названию города, например, необходимо, чтобы название было не очень длинным. И мы, проектируя базу данных определяем этот максимум с точки зрения здравого смысла и производительности поиска.

Скажем, не больше 50 символов.

Ну, и давайте применим менеджерское правило удвоение всех чисел на 2, прежде чем передавать их вверх по цепочке, и поставим 100.

Итак, в базе мы определяем в таблице максимально возможное количество символов для хранения названия города как 100 символов.

Но, что если наш пользователь, (возможно,пытаясь проверять нас на прочность) попытается передать нам в качестве названия города тайский вариант наименования столицы Таиланда, известной нам как Бангкок?

(показать слайд :)

Даже в оригинальном варианте там 145 символов, так что целиком оно в базу не влезет.

В то же время, пользователь всё сделал правильно с точки зрения контракта - структура переданной нам модели полностью правильна и вернуть ему пустой `400 - Bad Request` (без намёка на то, а что же не так с его данными?) было бы просто нечестно.

Для таких вот случаев и требуются механизмы проверки пользовательских данных.

Что именно требуется проверять?

Как правило, проверки требуют запросы, которые что-то привносят в серверные данные. Часто проверки удостаиваются запросы следующих типов:

- POST
- PUT
- PATCH

Проверяем только входные данные! (выходные данные не проверяются)

Data Annotations как механизм установки правил валидации

В принципе, можно использовать как сторонние, так и встроенные средства установки правил и проверки данных пользователя.

В ASP.NET Core MVC встроенным средством проверки является Data Annotations.

Атрибуты аннотации данных (data annotation attributes) — это специальные атрибуты, которыми можно разметить модель, чтобы обозначить правила её валидации.

Такие атрибуты включают в себя возможность задавать часто используемые правила как “обязательное поле” или “максимальная длина строки”. Также можно определять и более сложные правила.

ModelState как средство проверки правил валидации

Для пункта 2 из списка решаемых задач — непосредственно проверки — используется концепция модели состояния.

Это сложный объект, в котором хранится как словарь состояния модели в привязке к конкретным проверкам, так и коллекция ошибок для каждого свойства объекта модели.

Для быстрого анализа можно воспользоваться свойством `IsValid`: Если в модели есть проблемы, `ModelState.IsValid` вернёт `false`.

Совместная работа: Валидация при заведении нового города

Problem statement

Ставим точку остановки в самом начале метода `CreateCity` нашего контроллера.

Запускаем Postman отправляем новый POST-запрос вообще без тела на URI `/api/cities` получаем в ответ `Status Code: 415; Unsupported Media Type`. Мы даже не попадаем в наш метод. Это происходит потому что мы не не выставили заголовок `Content-Type` и ASP.NET Core MVC обрабатывает эту ошибку на более ранней стадии.

Выставляем заголовок `Content-Type` в значение `application/json`. Повторяем запрос.

Теперь мы попадаем в метод и если пройтись по строчкам (`F10`), то мы увидим, что наш параметр `city` равен `null` и мы возвращаем `Status Code: 400; Bad Request`.

Но это не единственная ошибка, которая может быть, как мы поняли выше.

Правила

Давайте определим правила для модели CityCreateModel (которые сейчас и будем нарушать):

- Поле Name
  - Обязательное поле
  - Максимальная длина: 100 символов
- Поле Description:
  - Максимальная длина: 255 символов
- Поле NumberOfPointsOfInterest
  - Число в диапазоне от 0 до 100

Давайте теперь сконструируем запрос, который будет структурно правильный, но содержать данные, которые нарушают сразу несколько правил (`create-city-invalid-model-sample.json`).

Отсылаем такой POST-запрос на заведение нового города.

И в этот раз наша единственная проверка `city` на `null` пройдена. Мы почти готовы выполнить запрос, хотя наш город совсем не похож на ту модель, которую мы изначально задумали.

Сейчас у нас даже всё пройдёт гладко, так как мы работаем просто со списком в памяти в качестве хранилища. А в реальной жизни, это будет база данных, в которой будут свои проверки и их уже эти данные не пройдут. Это приведёт к исключению и пользователь API получит `Internal Server Error - 500` вместо `Bad Request - 400`, хотя проблема не на серверной стороне.

Model

Открываем нашу модель `CityCreateModel` и добавляем data annotation атрибуты:

```
...
[Required]
public string Name { get; set; }
...
```

Потребуется добавить namespace `System.ComponentModel.DataAnnotations`.

Чтобы увидеть все возможные атрибуты можно написать в классе `System.ComponentModel.DataAnnotations` и поставить точку.

Теперь добавим остальные атрибуты согласно нашим правилам:

```
[Required]
[MaxLength(100)]
public string Name { get; set; }

public string Description { get; set; }

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

Controller

Теперь переходим в **контроллер**.

После проверки `city` на `null` добавляем проверку модели:

```
if (!ModelState.IsValid)
{
      return BadRequest();
}
```

Отсылаем ещё раз наш полностью неверный POST-запрос на заведение нового города (`create-city-invalid-model-sample.json`).

Теперь он возвращает `Bad Request - 400`. Уже неплохо!

Default Error Messages in output

Чтобу добавить в тело запроса более детальное описание ошибки, необходимо добавить `ModelState` параметром метода `BadRequest`:

```
if (!ModelState.IsValid)
{
      return BadRequest(ModelState);
}
```

Отсылаем наш POST-запрос (`create-city-invalid-model-sample.json`).

Видим в теле ответа сообщение об ошибке “по умолчанию”.

Custom Error Messages in output

Определим собственный текст в сообщении об ошибке в наш атрибуте. Это делается параметром атрибута Required (через установку

```
[Required(ErrorMessage = "The name of the city is a required field")]
[MaxLength(100)]
```

```
public string Name { get; set; }
```

Отсылаем ещё раз наш POST-запрос (`create-city-invalid-model-sample.json`).

Видим в теле ответа **уже наш текст ошибки**, а также и остальные, которые мы намеренно допустили.

Самостоятельная работа

Доделать оставшиеся проверки, необходимые при создании нового города. Правила модели `CityCreateModel`:

- Поле `Description`:
  - Максимальная длина: 255 символов
    ** задать собственное сообщение об ошибке “Description should be not longer than 255 characters*
- Поле `NumberOfPointsOfInterest`
  - Число в диапазоне от 0 до 100
    ** атрибут Range (разбираемся сами через справку по F1)*

Совместная работа: кастомные проверки модели

Проверка через контроллер

Добавление кастомной проверки, отсутствующей в наборе готовых атрибутов.

Поле `Description` не должно иметь такое же значение, как в поле `Name`.

Для этого нам всё-таки не избежать написания кода. Добавляем в контроллер код перед тем, как мы выполняем проверку модели:

```
if (city.Description == city.Name)
{
      ModelState.AddModelError(
            "Description",
            "Description shouldn't be the same as Name.");
}
```

Запускаем, проверяем, радуемся (на самом деле нет :)

Что в действительности произошло? Мы “размазали” ответственность за описание правил между контроллером и моделью (до сих пор правила оставались в рамках ответственности модели). Это не очень хорошо.

Пишем кастомный атрибут

В библиотеке `System.ComponentModel.DataAnnotations` содержит очень схожий по логике атрибут Compare, использующийся, например, на форме установки или изменения пароля. Он проверяет поля на идентичность и выдаёт ошибку, если они не равны. Нам нужна схожая с точности до наоборот логика, так что предлагаю подглядеть, как он реализован.

Открываем браузер и пишем в строке поиска *“github public class compareattribute validationattribute”*

Находим ссылку на GitHub, открываем, смотрим и пишем по аналогии:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class DifferentValueAttribute: ValidationAttribute
{
        public string OtherProperty { get; set; }

        protected override ValidationResult IsValid(object value, ValidationContext validationContext)
        {
                PropertyInfo otherPropertyInfo = validationContext.ObjectType.GetProperty(OtherProperty);
                if (otherPropertyInfo == null)
                {
                        return new ValidationResult($"Cannot find the property with name \"{OtherProperty}\"");
                }

                object otherPropertyValue = otherPropertyInfo.GetValue(validationContext.ObjectInstance, null);
                if (Equals(value, otherPropertyValue))
                {
                        return new ValidationResult(
                                $"{validationContext.MemberName} shouldn't be the same as {OtherProperty}.");
                }

                return ValidationResult.Success;
        }
}
```

Здесь доступ к полям класса получается через `Reflection`, так что необходимо подключить неймспейс `System.Reflection`.

Теперь навешиваем наш атрибут на поле `Description` модели `CityCreateModel`:

```
[MaxLength(255)]
[DifferentValue(OtherProperty = "Name")]
public string Description { get; set; }
```

И можно (и даже нужно) закомментировать логику проверки Description в классе контроллера:

```
// if (city.Description == city.Name)
// {
     ModelState.AddModelError(
//         "Description",
//         "Description shouldn't be the same as Name.");
//}
```

Запускаем, можем с точкой остановки в методе `IsValid` класса нашего атрибута.
Проверяем, радуемся :)

Inversion of Control и Dependency Injection

** материал подготовлен на базе блога https://shwanoff.ru/ioc-and-di/*

Определения

Инверсия управления

Инверсия управления (*Inversion of Control*, *IoC*) это определенный набор рекомендаций, позволяющих проектировать и реализовывать приложения используя слабое связывание отдельных компонентов.

Для того чтобы следовать принципам инверсии управления нам необходимо:

- Реализовывать компоненты, отвечающие за одну конкретную задачу
- Компоненты должны быть максимально независимыми друг от друга
- Компоненты не должны зависеть от конкретной реализации друг друга

Внедрение зависимостей

Одним из видов конкретной реализации данных рекомендаций является механизм внедрения зависимостей (*Dependency Injection*, *DI*). Он определяет две основные рекомендации:

- 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.

То есть, если у нас будут существовать два связанных класса, то нам необходимо реализовывать связь между ними не напрямую, а через интерфейс. Это позволит нам при необходимости динамически менять реализацию зависимых классов.

Пример по слайду

Предположим, что мы решили написать свою собственную программу, выполняющую крипто вычисления, другим словом майнер. Любая криптовалюта основана на какой-либо хэш-функции (алгоритме).

Предположим, что наша программа будет выполнять вычисления на алгоритме `SHA256`, для майнинга биткоина. Тогда мы получим следующую связь между классами:

(слайд без IoC/DI)

Проблема состоит в том, что в настоящее время алгоритмов, на которых основаны крипто валюты достаточно много и их число постоянно увеличивается. Если мы захотим добавить новые алгоритмы

для майнинга других криптовалют, нам придется вносить большое количество изменений в сам класс майнера.

Чтобы этого избежать, необходимо создать промежуточный интерфейс, от которого будет зависеть майнер и который должны будут реализовывать различные алгоритмы.

(слайд с IoC/DI)

Так мы сможем не только разделить ответственность за выполнение конкретных задач между классом майнером и алгоритмами, но и сделаем задел на дальнейшее увеличение количества поддерживаемых алгоритмов.

DI в ASP.NET Core MVC App

Логирование

Для начала можно показать как это должно было бы работать с логированием.

Добавить к контейнеру сервис логирования через `services.AddLogging()` в методе `Configure` класса `Startup`. Однако, если мы посмотрим на исходники класса WebHost https://github.com/aspnet/MetaPackages/blob/master/src/Microsoft.AspNetCore/WebHost.cs, а именно на метод `CreateDefaultBuilder`, то мы увидим, что уже на этом уровне всё необходимое для логирования добавлено и сконфигурировано:

```
var builder = new WebHostBuilder();
…
builder
...
.ConfigureLogging((hostingContext, logging) =>
{
      logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
      logging.AddConsole();
      logging.AddDebug();
      logging.AddEventSourceLogger();
}).
```

Ну… и хорошо!

Совместная работа: использование встроенного сервиса логирования

Давайте используем сервис логирования для добавления записи в лог, например, при вызове метода GetCities контроллера.

Для начало нам необходимо добавить в контроллер новое приватное поле

```
private ILogger<CitiesController> _logger;
```

Мы будем использовать метод, который называется Constructor Injection, который, как вы догадались, будет внедрять зависимость через конструктор:

```
public CitiesController(ILogger<CitiesController> logger)
{
      _logger = logger;
}
```

При создании объекта нашего констроллера ASP.NET Core фреймворк **сам** будет подставлять из списка зарегистрированных сервисов тот, который реализует запрошенный интерфейс.

Теперь во всех методах нашего контроллера мы можем обращаться к внутреннему объекту `_logger`, который позволяет работать с логом приложения:

```
public IActionResult GetCities()
{
      _logger.LogInformation(nameof(GetCities) + " called");

      ...
}
```

Запускаем, дёргаем URI `/api/cities` и наблюдаем в окне `Debug Output` наше залогированное сообщение. Радуемся :)

Совместная работа: реализуем логирование в файл через NLog

Открываем URL: https://github.com/NLog/NLog.Web/wiki/Getting-started-with-ASP.NET-Core-2 и делаем всё по пунктам, там отлично всё изложено.

Разумеется, комментарии хорошо бы дать по ходу, но это уже по усмотрению и возможностям лектора.

На чём следует заострить внимание: мы вообще не меняли ни код контроллера ни код модели! Только код конфигурации приложения. Мы логируем через тот же самый `_logger` и это происходит благодаря реализованному DI.

Регистрация собственных сервисов

Собственные сервисы регистрируются в методе `ConfigureServices`.

Можно воспользоваться одним из трёх методов в зависимости от желаемого жизненного цикла сервиса:

- `AddTransient` — объект пересоздаётся при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии.
- `AddScoped` — объект пересоздаётся для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех

этих обращениях будет использоваться один и тот же объект сервиса.

- `AddSingleton` — объект сервиса создается при первом обращении к нему, все последующие запросы используют один и тот же ранее созданный объект сервиса.

Совместная работа: оформляем CitiesDataStore как сервис

Избавляемся от собственной имплементации синглтона для класса `CitiesDataStore`:

- Оставляем только свойство `Cities` и конструктор, где оно инициализируется.

Пишем интерфейс `ICitiesDataStore`:

```
public interface ICitiesDataStore
{
     List<CityData> Cities { get; }
}
```

Прописываем наследование классом `CitiesDataStore` интерфейса `ICitiesDataStore`.

В контроллере заводим новое поле:

```
private ICitiesDataStore _citiesDataStore;
```

Обновляем конструктор:

```
public CitiesController(
     ILogger<CitiesController> logger,
     ICitiesDataStore citiesDataStore)
{
     _logger = logger;
     _citiesDataStore = citiesDataStore;
}
```

Избавляемся в каждом методе от ставшего ненужным получением инстанса методом

```
var citiesDataStore = CitiesDataStore.GetInstance();
```

и заменяем обращение к `citiesDataStore` его на обращение к полю `_citiesDataStore`.

Запускаем. Добавляем город. Получаем все города. Убеждаемся, что синглтон работает как надо. Радуемся :)

Домашнее задание

Добавить валидацию на все оставшиеся методы.
Вынести декларацию и имплементацию Data Access Layer (DAL) за пределы приложения.