1 СЕРВЕРНОЕ ПРИЛОЖЕНИЕ НА ASP.NET CORE

1.1 Создание проекта и предназначение его классов

Создадим проект ASP.NET Core с использованием пустого шаблона:

```
1 md Practice
2 cd .\Practice
3 dotnet new sln --name PracticeTestApp
4 md PracticeTestApp
5 cd .\PracticeTestApp\
6 dotnet new web
7 cd ..
8 dotnet sln add .\PracticeTestApp\PracticeTestApp.csproj
```

Получим базовую структуру приложения, которое уже можно запустить (рисунок 1):

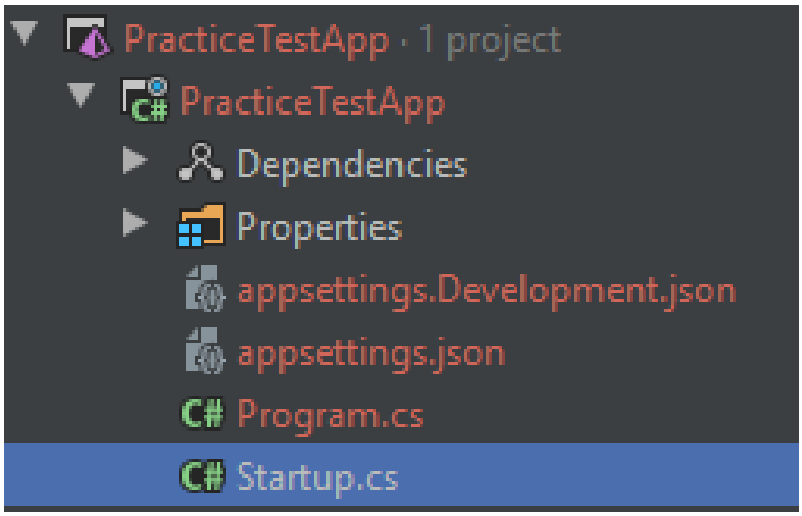

Рисунок 1 – Структура проекта

На рисунке 1 видно, что, помимо файлов непосредственно проекта, проект содержит классы `Startup.cs` и `Program.cs`. Разберем содержимое и предназначение этих файлов:

1.1.1 `Program.cs`

Содержимое `Program.cs` после создания пустого проекта:

```
1 public class Program {
2     public static void Main(string[] args) {
3         CreateHostBuilder(args).Build().Run();
4     }
```

```
5
6     public static IHostBuilder CreateHostBuilder(string[] args) =>
7         Host.CreateDefaultBuilder(args)
8             .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<
                  Startup>(); });
9 }
```

Все приложения .NET Core по соглашению должны иметь точку входа в виде метода `Main` класса `Program`. В этом месте в приложении ASP.NET Core создается `Host` - абстракция для инкапсуляции всех ресурсов приложения:

- реализация HTTP сервера;
- конфигурация сервера;
- компоненты конвейера;
- сервисы инверсии зависимостей (DI);
- логирование.

Есть два вида хостов: `.NET Generic Host`, `ASP.NET Core Web Host`.

Рекомендуется использовать `.NET Generic Host`, `ASP.NET Core Web Host` нужет для обратной совместимости с прошлыми версиями ASP.NET Core.

Создание хоста происходит с вызовом метода `CreateDefaultBuilder`, который устанавливает для хоста набор значений по умолчанию:

- использование `Kestrel` в качестве веб сервера;
- использование `appsettings.json` и `appsettings.EnvironmentName.json` в качестве проводников конфигурации;
- перенаправление вывода логирования в консоль и инструменты отладки.

Кроме того, при создании хоста указывается, какой класс использовать в качестве `Startup`.

1.1.2 `Startup.cs`

Содержимое класса `Startup.cs` после создание проекта:

```
1 public class Startup {
2     public void ConfigureServices(IServiceCollection services) { }
3
4     public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
5         if (env.IsDevelopment()) {
6             app.UseDeveloperExceptionPage();
```

```
7         }
8
9         app.UseRouting();
10        app.UseEndpoints(endpoints => {
11            endpoints.MapGet("/", async context => {
12                await context.Response.WriteAsync("Hello World!");
13            });
14        });
15    }
16 }
```

`Startup.cs` - класс, в котором:

- настраиваются сервисы, используемые приложением (метод `ConfigureServices`);
- настраивается конвейер обработки HTTP запросов как список промежуточных компонентов middleware (метод Configure).

В базовом варианте класс просто задает ответ на все GET-запросы как строку «Hello World!».

1.2 Настройка сервера под MVC архитектуру

1.2.1 Настройка роутинга

Изначально ASP.NET Core проектировался под архитектуру MVC, поэтому настройка включит в себя только включение MVC и настройка роутинга на контроллеры MVC. Делается это следующим образом:

```
1 public void ConfigureServices(IServiceCollection services) {
2     services.AddMvc();
3 }
4 ...
5 app.UseEndpoints(endpoints => {
6     endpoints.MapControllerRoute(name: "default",
7                                  pattern: "{controller=Home}/{action=Index}/{id
                                       ?}");
8 });
```

Теперь запросы будут мэппиться на экшены и контроллеры с соответствующими именами. Для обработки запросов нужно создать контроллер и экшен. Экшен – любой публичный метод контроллера, обычно возвращающий IActionResult. Url для контроллера можно задать атрибутом.

```
1 [Route("home")]
```

```
2 public class HomeController : Controller {
3     public IActionResult Index() => Ok();
4 }
```

Но лучше соблюдать соглашения по расположению контроллеров. В этом случае, достаточно разместить контроллер в папке `Controllers`:

```
1 namespace PracticeTestApp.Controllers {
2     public class HomeController : Controller {
3         public IActionResult Index() => Ok();
4     }
5 }
```

В обоих случаях запрос /Home/Index обработается экшеном `Index` класса `HomeController`.

1.2.2 Возвращение HTML документа

Для того, чтобы вернуть HTML (в терминах MVC - View), нужно из экшена вернуть `ViewResult`:

```
public IActionResult Index() => View();
```

Для view можно задавать имя, но, как и в случае с контроллерами, можно его создать в папке Views в папке с именем соответствующего контроллера с именем соответствующего экшена. View представляет из себя `.cshtml` документ, использующий синтаксис `Razor`. `Razor` позволяет прокидывать из контроллера во view необходимые данные и использовать их во view с помощью синтаксиса C#.

Создадим view, соответствующую экшену `Index`:

```
1 //HomeController.cs:
2 public class HomeController : Controller {
3     public IActionResult Index() {
4         return View(new IndexViewModel("Hello, World!"));
5     }
6 }
7
8 //IndexViewModel.cs:
9 public class IndexViewModel {
10     public IndexViewModel(string message) {
11         Message = message;
12     }
13     public string Message { get; }
14 }
15
```

```
//Views\Home\Index.cshtml:
@model PracticeTestApp.ViewModels.Home.IndexViewModel
<!DOCTYPE html>
<html>
    <head>
        <title>Test</title>
    </head>
    <body>
        <h1>@Model.Message</h1>
    </body>
</html>
```

При запуске приложени получим следующее (рисунок 2):

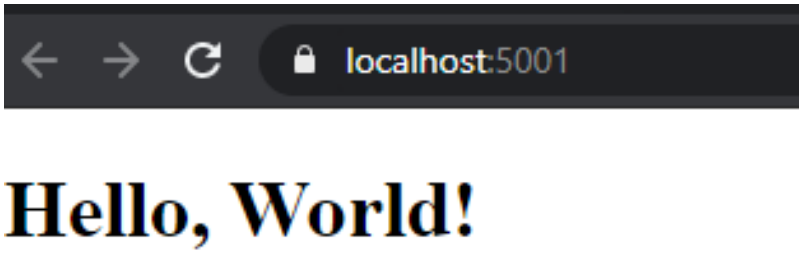

Рисунок 2 – Веб-страница, возвращенная сервером

Для того, чтобы получить параметры из query string при GET запросе, достаточно указать имена соответствующих параметров в экшене:

```
public IActionResult Index(int id, string message)
```

1.2.3 Создание экшенов для других типов запросов

Для обработки запросов, отличных от GET, достаточно задать атрибут соответствующего запроса на экшене:

```
public class HomeController : Controller {
    public IActionResult Index(int id, string message) {
        return View(new IndexViewModel("Hello, World!"));
    }

    [HttpPost]
    public IActionResult PostAction() => Ok();

```

```
9       [HttpPut]
10      public IActionResult PutAction() => Ok();
11
12      [HttpDelete]
13      public IActionResult DeleteAction() => Ok();
14  }
```

1.3 Создание API-контроллеров

API-контроллеры помечаются атрибутом `[ApiController]`. Желательно указывать отдельный путь для API запросов. Пример:

```
1  //DataController.cs
2  [ApiController]
3  [Route("api/[controller]")]
4  public class DataController : ControllerBase {
5      [HttpGet]
6      public ActionResult<string[]> Get() {
7          var strings = new[] {
8              "1", "2", "3", "4", "5"
9          };
10         return strings;
11     }
12
13     [HttpPost]
14     public ActionResult<Product[]> Post() {
15         var products = new[] {
16             new Product(1, name: "First", type: "Utilities"),
17             new Product(2, name: "Second", type: "Sports"),
18             new Product(3, "Third", "Other")
19         };
20         return products;
21     }
22 }
23
24 //Product.cs
25 public class Product {
26     public Product(long id, string name, string type) {
27         Id = id;
28         Name = name;
29         Type = type;
30     }
31
32     public long Id { get; }
33
34     public string Name { get; }
35
36     public string Type { get; }
```

```
37 }
```

Теперь, для того, чтобы обратиться к API-контроллеру, достаточно послать GET или POST запросы на адрес \api \data (рисунки 3 и 4). Для отправки запросов используется Postman.

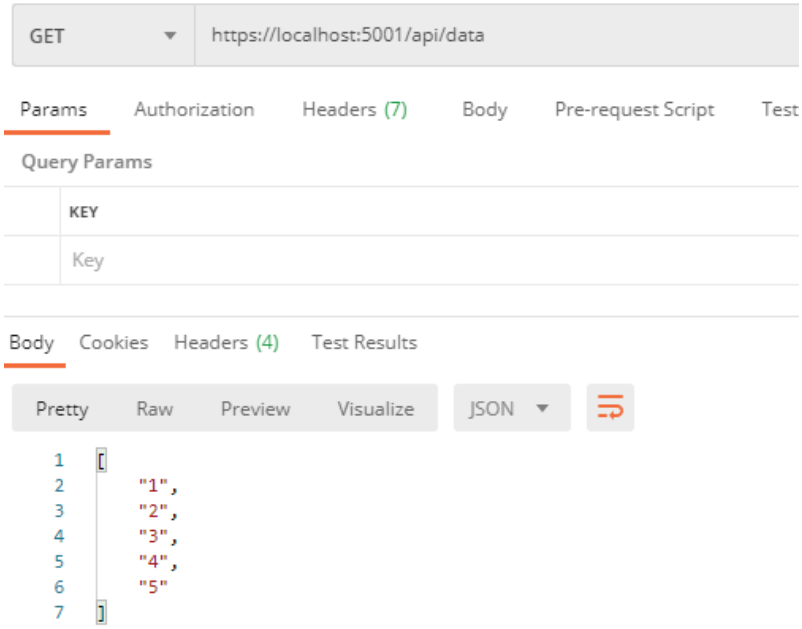

Рисунок 3 – Получение результата GET-запроса

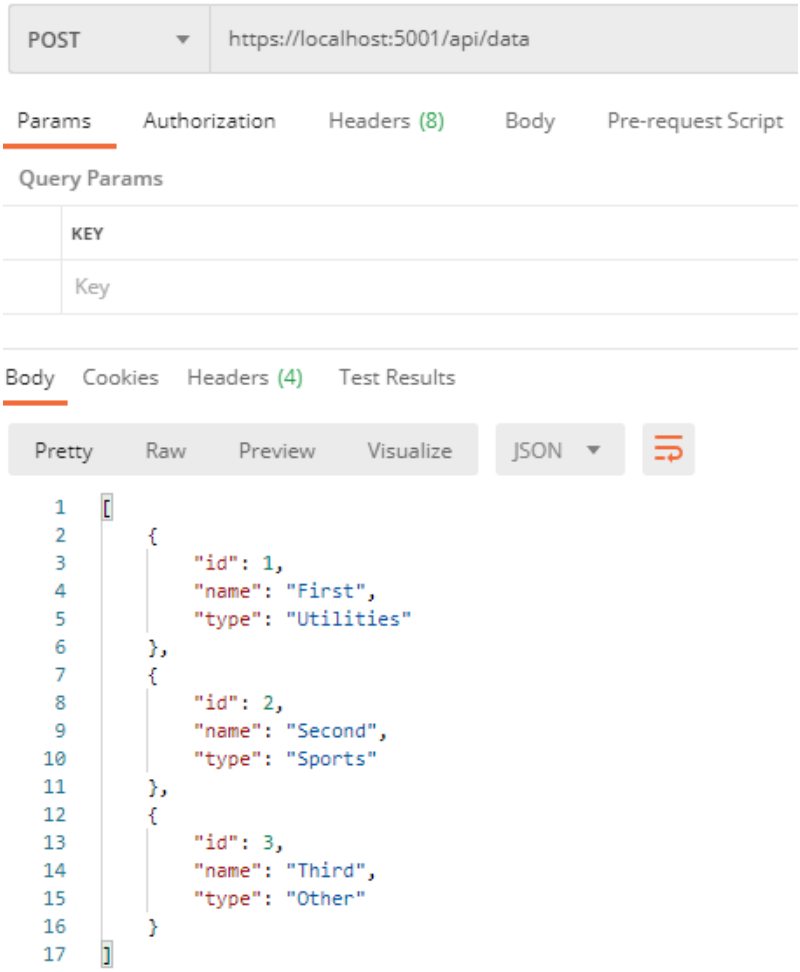

Рисунок 4 – Получение результата POST-запроса

1.4 Создание фильтров

Фильтры в ASP.NET Core позволяют запускать код до или после определенных этапов конвейера обработки запросов. Встроенные возможности аутентификации, авторизации, кеширования ответов ASP.NET реализованы с помощью фильтров.

Очередь фильтров наступает после того, как фреймворк выбирает экшен, который необходимо выполнить. Существуют следующие виды фильтров:

1. Фильтры авторизации. Они запускаются первыми чтобы определить, авторизован ли пользователь для выполнения данного экшена. Они способны прерывать выполнение конвейера в том случае, если пользователь не авторизован.
2. Фильтры ресурсов. Следующие по вложенности фильтры. `OnResourceExecuting` выполняется перед всем остальным конвейером, `OnResourceExecuted` - после.

3. Экшен фильтры. Выполняются непосредственно до и после вызова экшена. способны изменять параметры, передаваемые в экшены и мутировать результат выполнения экшена.

4. Фильтры исключений. Используются для создания общего подхода к обработке необработанных исключений перед тем, как сформировать тело ответа.

5. Фильтры результатов. Используются для запуска кода непосредственно после выполнения экшенов для мутирования результатов. Полезны для задания логики обработки View.

Создадим фильтр и включим его в конвейер обработки запросов:

```
1 //InvalidModelStateFilter.cs
2 public class InvalidModelStateFilter : IActionFilter {
3     public void OnActionExecuted(ActionExecutedContext context) { }
4
5     public void OnActionExecuting(ActionExecutingContext context) {
6         if (!context.ModelState.IsValid) {
7             context.Result = new BadRequestObjectResult(context.ModelState);
8         }
9     }
10 }
11
12 //Startup.cs
13 services.AddMvc(options => options.Filters.Add(typeof(InvalidModelStateFilter))
       );
```

Флаг ModelState отвечает за успешность привязки входящих параметров к объектам. Приведенным выше способом можно валидировать входящие объекты, например, следующий экшен принимает объект:

```
public IActionResult IndexForProduct(Product product) => Ok();
```

Обратимся к нему следующим образом:

```
/Home/IndexForProduct?Id=15&Name=hello&type=sometype
```

Получим ответ 200.

Обратимся, задав параметру Id невалидное значение:

```
/Home/IndexForProduct?Id=asd&Name=hello&type=sometype
```

Получим следующий ответ (рисунок 5):

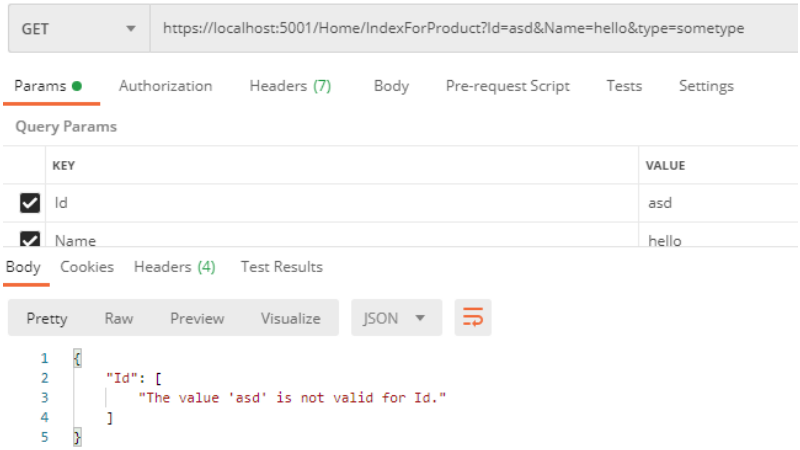

Рисунок 5 – Полученная ошибка при выполнении отправке невалидных данных