# 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):

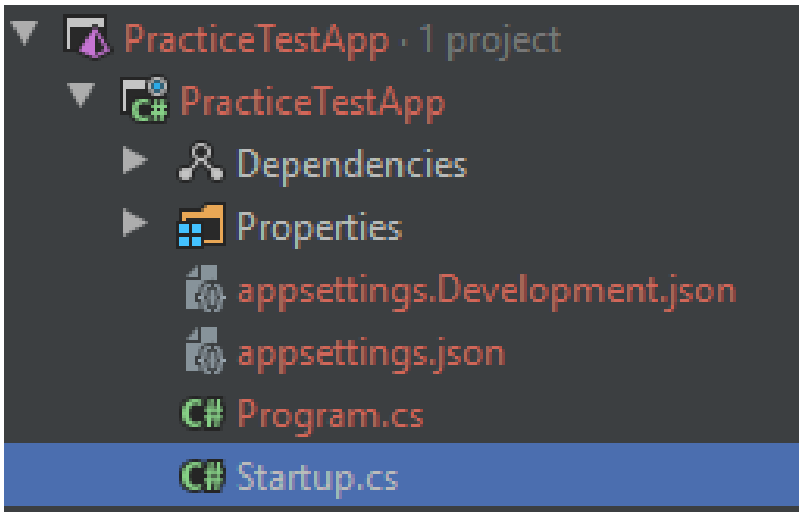


Рисунок 1 – Структура проекта

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):

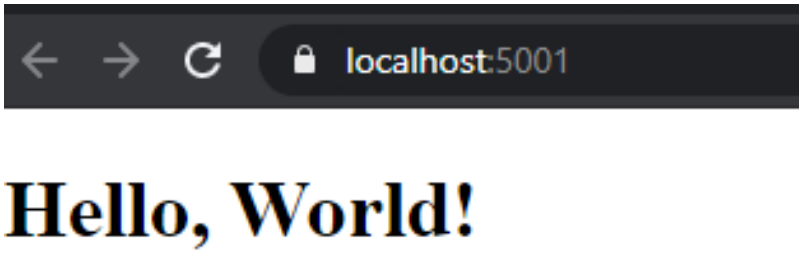


Рисунок 2 – Веб-страница, возвращенная сервером

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.

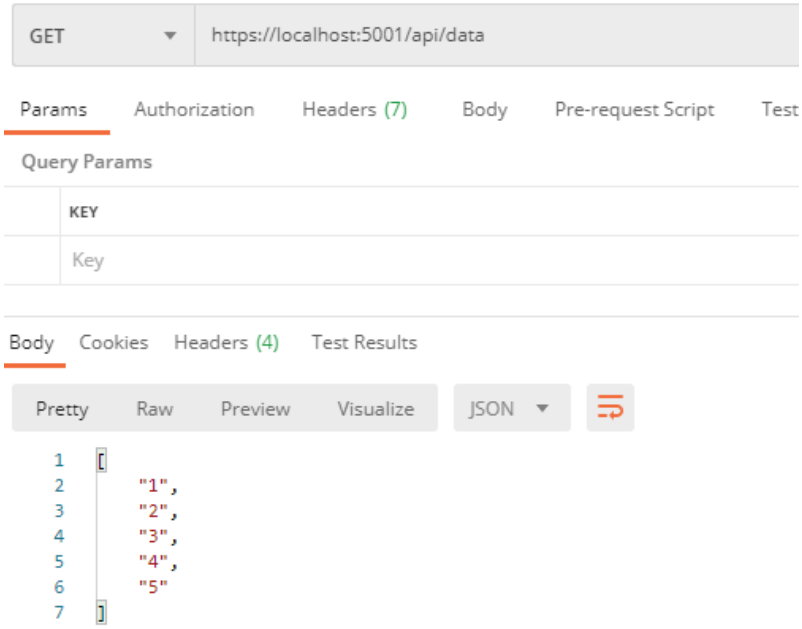


Рисунок 3 – Получение результата GET-запроса

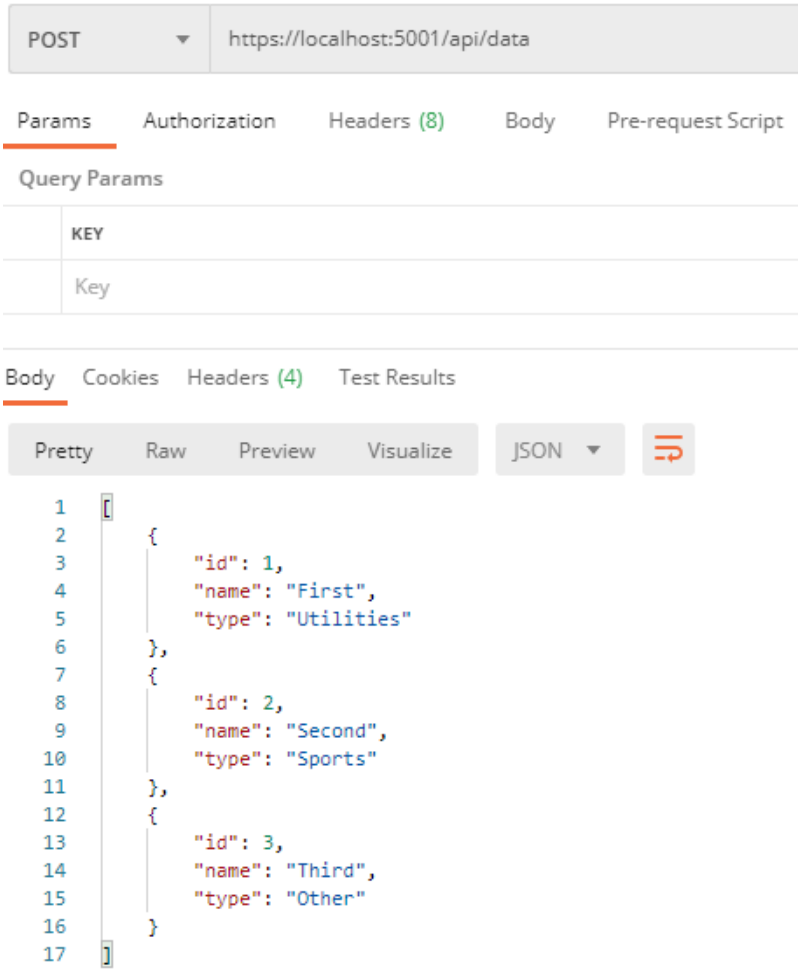


Рисунок 4 – Получение результата POST-запроса

### 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):

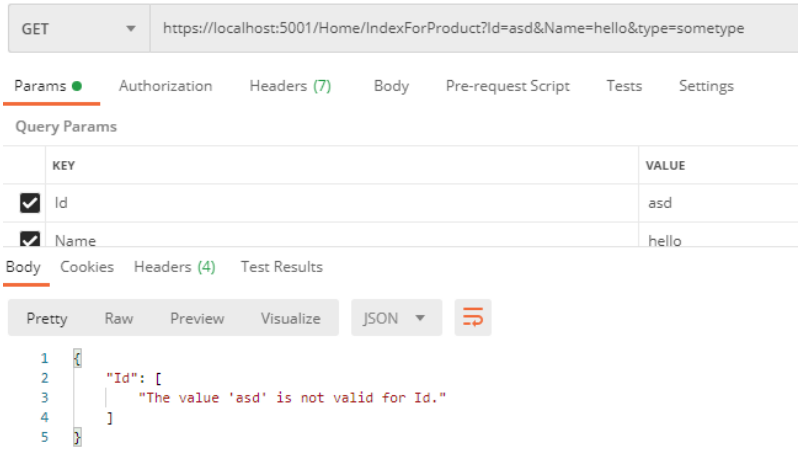


Рисунок 5 – Полученная ошибка при выполнении отправке невалидных данных

# 2 Клиентское приложение на KnockoutJS

## 2.1 Создание проекта приложения

```
cd .\PracticeTestApp\PracticeTestApp\
md ClientApp
npm install
```

В папке появится файл `package.json`, который описывает проект и его зависимости. Добавим в него следующую конфигурацию, включающую фреймворк `KnockoutJS` и его типизацию для языка `TypeScript`. Кроме того, добавим в проект `Webpack` для сборки клиентских бандлов:

```
{
    "name": "practicetestapp",
    "version": "1.0.0",
    "repository": {
        "type": "git",
        "url": "https://github.com/astro6703/Learning/Practice"
    },
    "description": "test app for production practicing purposes",
    "license": "",
    ...
    ...
  "devDependencies": {
    "webpack": "^4.42.0",
    "webpack-cli": "^3.3.11",
    "webpack-config": "^7.5.0",
    "@types/knockout": "^3.4.67",
    ...
  },
```

```
  "dependencies": {
    "bootstrap": "^4.5.0",
    "knockout": "^3.5.1"
  }
```

Для работы языка TypeScript и Webpack нужно задать конфигурации:

```
//tsconfig.json
{
  "compilerOptions": {
    "target": "es6",
    "module": "es2015",
    "lib": ["esnext", "dom"],
    "sourceMap": true,
    "removeComments": true,
    "outDir": "./tsc/",

    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "noImplicitThis": true,
    "alwaysStrict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": [
    "./src/**/*"
  ]
}
//webpack.config.js
const path = require('path');
const miniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
    entry: {
        'home/index': './src/home/index.ts'
    },
    module: {
        rules: [
            { test: /\.tsx?$/, loader: 'ts-loader', exclude: /node_modules/ },
            { test: /\.s[ac]ss$/i, use: [ miniCssExtractPlugin.loader, 'css-
                loader', 'resolve-url-loader', 'sass-loader' ] },
            { test: /\.(png|jpe?g|gif)/i, loader: 'file-loader', options: {
                name: '[path][name].[ext]', outputPath: 'images' } },
```

```
41          ],
42      },
43      mode: 'development',
44      devtool: 'source-map',
45      resolve: {
46          extensions: [ '.ts' ],
47      },
48      output: {
49          path: path.resolve(__dirname, 'bundles'),
50          filename: '[name].bundle.js',
51      },
52      plugins: [
53          new miniCssExtractPlugin({ filename: "[name].bundle.css" }),
54      ]
55  };
```

В файлы, задаваемые в секции `entry` конфига `Webpack`, будем писать программный код.

Для того, чтобы собрать приложение, нужно выполнить следующие команды:

```
1 npm install
2 npx webpack --config ./webpack.config.js
```

## 2.2 Паттерн MVVM

Паттерн MVVM был создан в Microsoft и использовался при построении архитектуры WPF приложений. Он позволяет отделить логику приложения от представления. MVVM состоит из 3 компонентов: View (представление), ViewModel (модель представления), Model(модель).

- Модель описывает используемые в приложении данные.
- Представление определяет визуальный интерфейс.
- Модель представления связывает модель и представление через механизм привязки данных. Изменение свойства в модели представления должно приводить к такому же изменению в представлении и наоборот.

Фреймворк `KnockoutJS` создан как реализация паттерна MVVM.

## 2.3 KnockoutJS

### 2.3.1 observable объекты

Чтобы синхронизировать данные между представлением и моделью представления, KnockoutJS использует механизм биндингов, работающий на паттерне Observable: привязки элемента представления к элементу модели представления с подпиской на изменения друг друга. Когда изменяется один из этих элементов, он автоматически оповещает всех его подписчиков и передает данные о всех изменениях.

Это реализуется с помощью объекта observable. Он создается вызовом ko.observable. Observable объекты - функции. Вызов функции без параметров - чтение, с параметром - запись. Реализовано через функции, а не свойства объектов из-за необходимости поддерживать Internet Explorer 6.

Для массивов объектов существует observableArray. Массивы можно хранить и в обычных observable, но тогда в случае, когда нужно изменить содержание массива (например, добавить новый элемент), придется делать это следующим образом:

```
1 this.items = ko.observable();
2 this.items().push(item);
```

В этом случае изменится не observable объект, а объект, который он обрамляет и оповещения подписчиков не произойдет. В случае с observableArray этот же код будет выглядеть следующим образом:

```
1 this.items = ko.observableArray();
2 this.items.push(item);
```

Для того, чтобы это было возможным, observableArray реализует стандартные методы работы над массивами: push, pop, shift, unshift, sort, reverse и splice.

Для создаения вычислимых объектов существуют computed объекты. Они создаются вызовом ko.computed и передачей ему функции вычисления значения этого объекта. Все зависимости этой функции от объектов observable при изменении будут перезапускать вычисление этого объекта:

```
1 this.composite = ko.computed(() => this.first() + this.second());
```

В этом случае `composite` будет вычислен каждый раз, когда `first` и `second` будут изменены.

Также, на изменения отдельных объектов можно подписаться вручную. Это делается с помощью вызова функции `subscribe` у `observable` объекта. Она принимает функцию, которая будет вызвана с новым значением каждый раз, когда `observable` объект изменится:

```
1 const location = ko.observable();
2 colation.subscribe(newLocation => webService.getLocationDetails(newLocation));
```

### 2.3.2 Привязка данных

Для создания привязки `KnockoutJS` использует спецификацию HTML5 с использованием атрибутов data-* и объявляет свой атрибут `data-bind`. Этот атрибут можно указать у объектов в DOM, чтобы привязать его свойства к свойствам модели представления:

```
1 //index.ts
2 import * as ko from 'knockout';
3
4 class IndexViewModel {
5     public name: KnockoutObservable<string>;
6     public message: KnockoutObservable<string>
7
8     constructor() {
9         this.name = ko.observable('');
10        this.message = ko.pureComputed(() => `Hello, ${this.name() || 'visitor
              '}, how are you?`)
11    }
12 }
13
14 ko.applyBindings(new IndexViewModel());
15
16 //Index.cshtml
17 <body>
18
19 <label for="name">Please, enter your name:</label>
20 <input id="name" type="text" data-bind="value: name" />
21 <br>
22 <span data-bind="text: message"></span>
23
24 <script src="~/static/home/index.bundle.js"></script>
```

В этом случае происходит привязка свойства `value` HTML инпута к объек-

ту в модели представления. Сообщение `message` из модели представления привязывается к тексту этемента DOM и изменяется каждый раз, когда изменяется значение в текстовом поле name (рисунок 6).


(a) Пустое текстовое поле (b) Текстовое поле заполнено

Рисунок 6 – Пример привязки элементов представления к свойствам модели представления

Value-биндинг - встроенный в `KnockoutJS`, использовать просто любые атрибуты в качестве биндингов нельзя, не расширив `KnockoutJS` с помощью обработчика биндингов.

Можно создавать биндинги вложенных свойств:

```
1 //index.ts
2 import * as ko from 'knockout';
3
4 interface Person {
5     name: string;
6     age: number;
7 }
8
9 class IndexViewModel {
10     public person: KnockoutObservable<Person>;
11
12     constructor() {
13         const person = { name: 'Dude', age: 25 };
14         this.person = ko.observable(person);
15     }
16 }
17
18 ko.applyBindings(new IndexViewModel());
19
20 //Index.cshtml
21 <span data-bind="text: person().name"></span>
22 <br>
23 <span data-bind="text: person().age"></span>
24
25 <script src="~/static/home/index.bundle.js"></script>
```

Результат привязки изображен на рисунке fig:binding3:

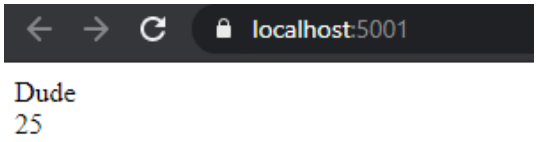


Рисунок 7 – Пример привязки элементов представления к вложенным свойствам модели представления

# ВЫВОДЫ

В ходе практики были изучены open-source фреймворки ASP.NET Core и KnockoutJS для веб-разработки, которые отлично работают вместе.

ASP.NET Core - кросс-платформенный фреймворк, написанный на языке C# и реализующий паттерн MVC, но позволяющий отойти от него с помощью гибкой конфигурации роутинга. Возможно создание SPA-приложений из-за широкой поддержки веб-сервисов. В базовой конфигурации позволяет обрабатывать HTTP запросы любых методов с легким получением параметров из адреса и из body запроса, встроена валидация получаемой из параметров модели, легко создавать API. С дополнительными пакетами доступны средства для реализации логирования, авторизации, аутентификации и работы с базами данных.

KnockoutJS - фреймворк для фронтенд разработки, реализующий паттерн MVVM и с помощью привязки (бандинга) представления к модели представления предоставляющий широкие возможности для создания интерактивных приложений. Сильно конфигурируемый и глубоко расширяемый: есть возможность расширения привязываемых объектов и создания собственных биндингов. Существует возможность создания компонентов.

Код, написанный в ходе практики расположен в репозитории по ссылке: https://github.com/astro6703/Learning/tree/practice/Practice