



# Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 5. «Основы в ASP.NET Core. Часть 5»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

# Учебные вопросы:

1. Создание простейшего API.
2. Загрузка файлов на сервер.
3. Метод Use.
4. Создание ветки конвейера. UseWhen и MapWhen.

# 1. Создание простейшего API

Рассмотренного в прошлых темах материала достаточно для создания примитивного приложения. В этой теме попробуем реализовать простейшее приложение Web API в стиле **REST**. Архитектура **REST** предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером, где каждый тип запроса отвечает за определенное действие:

**GET** (получение данных).

**POST** (добавление данных).

**PUT** (изменение данных).

**DELETE** (удаление данных).

Поскольку в приложении ASP.NET Core мы можем легко получить и адрес запроса, и тип запроса, то реализовать подобную архитектуру не составит труда.

## Создание сервера на ASP.NET Core

Вначале определим веб-приложение на ASP.NET Core, которое и будет, собственно, представлять Web API:

```

1  using System.Text.RegularExpressions;
2
3  // начальные данные
4  List<Person> users = new List<Person>
5  {
6      new() { Id = Guid.NewGuid().ToString(), Name = "Tom", Age = 37 },
7      new() { Id = Guid.NewGuid().ToString(), Name = "Bob", Age = 41 },
8      new() { Id = Guid.NewGuid().ToString(), Name = "Sam", Age = 24 }
9  };
10
11 var builder = WebApplication.CreateBuilder();
12 var app = builder.Build();
13
14 app.Run(async (context) =>
15 {
16     var response = context.Response;
17     var request = context.Request;
18     var path = request.Path;
19     //string expressionForNumber = "^/api/users/([0-9]+)$";    // если id представляет число
20
21     // 2e752824-1657-4c7f-844b-6ec2e168e99c
22     string expressionForGuid = @"^/api/users/{8}-{4}-{4}-{4}-{12}$";
23     if (path == "/api/users" && request.Method == "GET")
24     {
25         await GetAllPeople(response);
26     }
27     else if (Regex.IsMatch(path, expressionForGuid) && request.Method == "GET")
28     {
29         // получаем id из адреса url
30         string? id = path.Value?.Split("/")[3];
31         await GetPerson(id, response);
32     }
33     else if (path == "/api/users" && request.Method == "POST")
34     {
35         await CreatePerson(response, request);
36     }
37     else if (path == "/api/users" && request.Method == "PUT")
38     {
39         await UpdatePerson(response, request);
40     }
41     else if (Regex.IsMatch(path, expressionForGuid) && request.Method == "DELETE")
42     {
43         string? id = path.Value?.Split("/")[3];
44         await DeletePerson(id, response);
45     }
46     else
47     {
48         response.ContentType = "text/html; charset=utf-8";
49         await response.SendFileAsync("html/index.html");
50     }
51 });
52
53 app.Run();

```

```

54
55 // получение всех пользователей
56 async Task GetAllPeople(HttpResponse response)
57 {
58     await response.WriteAsJsonAsync(users);
59 }
60 // получение одного пользователя по id
61 async Task GetPerson(string? id, HttpResponse response)
62 {
63     // получаем пользователя по id
64     Person? user = users.FirstOrDefault((u) => u.Id == id);
65     // если пользователь найден, отправляем его
66     if (user != null)
67         await response.WriteAsJsonAsync(user);
68     // если не найден, отправляем статусный код и сообщение об ошибке
69     else
70     {
71         response.StatusCode = 404;
72         await response.WriteAsJsonAsync(new { message = "пользователь не найден" });
73     }
74 }
75
76 async Task DeletePerson(string? id, HttpResponse response)
77 {
78     // получаем пользователя по id
79     Person? user = users.FirstOrDefault((u) => u.Id == id);
80     // если пользователь найден, удаляем его
81     if (user != null)
82     {
83         users.Remove(user);
84         await response.WriteAsJsonAsync(user);
85     }
86     // если не найден, отправляем статусный код и сообщение об ошибке
87     else
88     {
89         response.StatusCode = 404;
90         await response.WriteAsJsonAsync(new { message = "пользователь не найден" });
91     }
92 }
93
94 async Task CreatePerson(HttpResponse response, HttpRequest request)
95 {
96     try
97     {
98         // получаем данные пользователя
99         var user = await request.ReadFromJsonAsync<Person>();
100         if (user != null)
101         {
102             // устанавливаем id для нового пользователя
103             user.Id = Guid.NewGuid().ToString();
104             // добавляем пользователя в список
105             users.Add(user);
106             await response.WriteAsJsonAsync(user);
107         }

```

```

108         else
109         {
110             throw new Exception("Некорректные данные");
111         }
112     }
113     catch (Exception)
114     {
115         response.StatusCode = 400;
116         await response.WriteAsJsonAsync(new { message = "Некорректные данные" });
117     }
118 }
119
120 async Task UpdatePerson(HttpResponse response, HttpRequest request)
121 {
122     try
123     {
124         // получаем данные пользователя
125         Person? userData = await request.ReadFromJsonAsync<Person>();
126         if (userData != null)
127         {
128             // получаем пользователя по id
129             var user = users.FirstOrDefault(u => u.Id == userData.Id);
130             // если пользователь найден, изменяем его данные и отправляем обратно клиенту
131             if (user != null)
132             {
133                 user.Age = userData.Age;
134                 user.Name = userData.Name;
135                 await response.WriteAsJsonAsync(user);
136             }
137             else
138             {
139                 response.StatusCode = 404;
140                 await response.WriteAsJsonAsync(new { message = "Пользователь не найден" });
141             }
142         }
143         else
144         {
145             throw new Exception("Некорректные данные");
146         }
147     }
148     catch (Exception)
149     {
150         response.StatusCode = 400;
151         await response.WriteAsJsonAsync(new { message = "Некорректные данные" });
152     }
153 }
154 public class Person
155 {
156     public string Id { get; set; } = "";
157     public string Name { get; set; } = "";
158     public int Age { get; set; }
159 }

```

Разберем в общих чертах этот код. Вначале идет определение данных – список объектов **Person**, с которыми будут работать клиенты:

```
1 var users = new List<Person>
2 {
3     new() { Id = Guid.NewGuid().ToString(), Name = "Tom", Age = 37 },
4     new() { Id = Guid.NewGuid().ToString(), Name = "Bob", Age = 41 },
5     new() { Id = Guid.NewGuid().ToString(), Name = "Sam", Age = 24 }
6 };
```

Стоит обратить внимание, что каждый объект **Person** имеет свойство **Id**, которое в качестве значения получает **Guid** – уникальный идентификатор, например **"2e752824-1657-4c7f-844b-6ec2e168e99c"**.

Для упрощения данные определены в виде обычного списка объектов, но в реальной ситуации обычно подобные данные извлекаются из какой-нибудь базы данных.

В методе **app.Run()** определяем компонент **middleware**, который в зависимости от типа запросов (**GET/POST/PUT/DELETE**) выполняет те или иные действия.

Так, когда приложение получает запрос типа **GET** по адресу **"api/users"**, то срабатывает следующий код:

```
1 if (path == "/api/users" && request.Method=="GET")
2 {
3     await GetAllPeople(response);
4 }
5 //.....
6 // получение всех пользователей
7 async Task GetAllPeople(HttpResponse response)
8 {
9     await response.WriteAsJsonAsync(users);
10 }
```

Запрос **GET** предполагает получение объектов, и в данном случае отправляем выше определенный список объектов **Person**.

Когда клиент обращается к приложению для получения одного объекта по **id** в запрос типа **GET** по адресу "**api/users/[id]**", то срабатывает следующий код:

```
1  else if (Regex.IsMatch(path, expressionForGuid) && request.Method == "GET")
2  {
3      // получаем id из адреса url
4      string? id = path.Value?.Split("/")[3];
5      await GetPerson(id, response);
6  }
7  //.....
8  // получение одного пользователя по id
9  async Task GetPerson(string? id, HttpResponseMessage response)
10 {
11     // получаем пользователя по id
12     Person? user = users.FirstOrDefault((u) => u.Id == id);
13     // если пользователь найден, отправляем его
14     if (user != null)
15         await response.WriteAsJsonAsync(user);
16     // если не найден, отправляем статусный код и сообщение об ошибке
17     else
18     {
19         response.StatusCode = 404;
20         await response.WriteAsJsonAsync(new { message = "Пользователь не найден" });
21     }
22 }
```

Чтобы убедиться, что в запрошенном адресе после **/api/users/** указан **id**, проверяем соответствие адреса регулярному выражению: **"/api/users/\w{8}-\w{4}-\w{4}-\w{4}-\w{12}\$"**. Данное выражение проверяет соответствие последнего сегмента адреса значению **Guid**, который имеет формат **xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx**.

В этом случае нам надо найти нужного пользователя по **id** в списке и отправить клиенту. Если же пользователь по **id** не был найден, то возвращаем статусный код 404 с некоторым сообщением в формате JSON.



При получении запроса **DELETE** действует аналогичная логика:

```
1  else if (Regex.IsMatch(path, expressionForGuid) && request.Method == "DELETE")
2  {
3      // получаем id из адреса url
4      string? id = path.Value?.Split("/")[3];
5      await DeletePerson(id, response);
6  }
7  //.....
8  async Task DeletePerson(string? id, HttpResponseMessage response)
9  {
10     // получаем пользователя по id
11     Person? user = users.FirstOrDefault((u) => u.Id == id);
12     // если пользователь найден, удаляем его
13     if (user != null)
14     {
15         users.Remove(user);
16         await response.WriteAsJsonAsync(user);
17     }
18     // если не найден, отправляем статусный код и сообщение об ошибке
19     else
20     {
21         response.StatusCode = 404;
22         await response.WriteAsJsonAsync(new { message = "Пользователь не найден" });
23     }
24 }
```

Только в данном случае, если пользователь найден в списке, удаляем его из списка и посылаем клиенту.

При получении запроса с методом **POST** по адресу **"/api/users"** используем метод **request.ReadFromJsonAsync()** для извлечения данных из запроса:

```

1  else if (path == "/api/users" && request.Method == "POST")
2  {
3      await CreatePerson(response, request);
4  }
5  //.....
6  async Task CreatePerson(HttpResponse response, HttpRequest request)
7  {
8      try
9      {
10         // получаем данные пользователя
11         var user = await request.ReadFromJsonAsync<Person>();
12         if (user != null)
13         {
14             // устанавливаем id для нового пользователя
15             user.Id = Guid.NewGuid().ToString();
16             // добавляем пользователя в список
17             users.Add(user);
18             await response.WriteAsJsonAsync(user);
19         }
20         else
21         {
22             throw new Exception("Некорректные данные");
23         }
24     }
25     catch (Exception)
26     {
27         response.StatusCode = 400;
28         await response.WriteAsJsonAsync(new { message = "Некорректные данные" });
29     }
30 }

```

Поскольку при извлечении данных из запроса может произойти исключение (например, в результате парсинга в **JSON**), оборачиваем весь код в **try..catch**. И в случае успешного получения данных устанавливаем у нового объекта свойство **Id**, добавляем его в список **users** и отправляем обратно клиенту.

Если приложению приходит **PUT**-запрос, то также с помощью метода **request.ReadFromJsonAsync()** получаем отправленные клиентом данные. Если объект найден в списке, то изменяем его данные и отправляем обратно клиенту, иначе отправляем статусный код 404:

```
1  else if (path == "/api/users" && request.Method == "PUT")
2  {
3      await UpdatePerson(response, request);
4  }
5  //.....
6  async Task UpdatePerson(HttpResponse response, HttpRequest request)
7  {
8      try
9      {
10         // получаем данные пользователя
11         Person? userData = await request.ReadFromJsonAsync<Person>();
12         if (userData != null)
13         {
14             // получаем пользователя по id
15             var user = users.FirstOrDefault(u => u.Id == userData.Id);
16             // если пользователь найден, изменяем его данные и отправляем обратно клиенту
17             if (user != null)
18             {
19                 user.Age = userData.Age;
20                 user.Name = userData.Name;
21                 await response.WriteAsJsonAsync(user);
22             }
23             else
24             {
25                 response.StatusCode = 404;
26                 await response.WriteAsJsonAsync(new { message = "Пользователь не найден" });
27             }
28         }
29         else
30         {
31             throw new Exception("Некорректные данные");
32         }
33     }
34     catch (Exception)
35     {
36         response.StatusCode = 400;
37         await response.WriteAsJsonAsync(new { message = "Некорректные данные" });
38     }
39 }
```

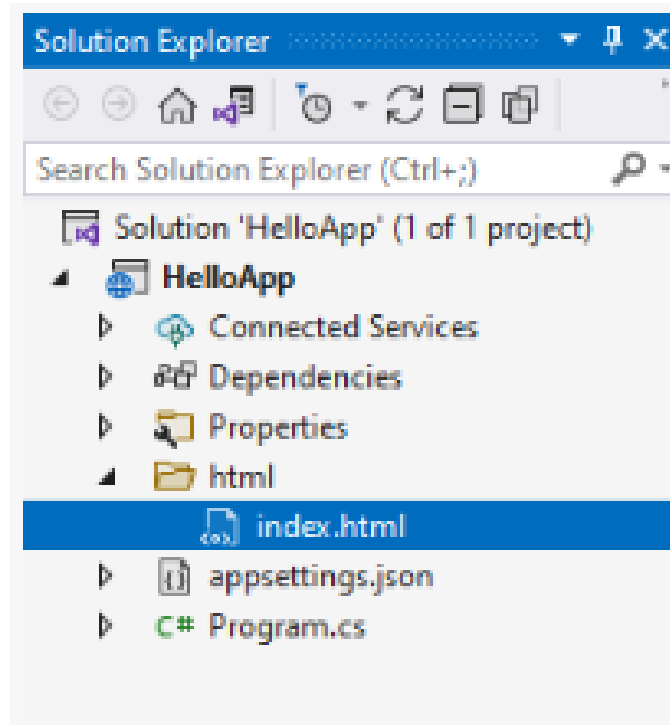
В случае, если запрос идет по другому адресу, то отправляем клиенту веб-страницу **index.html**, которую мы далее определим:

```
1 else
2 {
3     response.ContentType = "text/html; charset=utf-8";
4     await response.SendFileAsync("html/index.html");
5 }
```

Таким образом, мы определили простейший API. Теперь добавим код клиента.

## Определение клиента

Теперь добавим в проект папку **html**, в которую добавим новый файл **index.html**.



Определим в файле **index.html** следующим код для взаимодействия с сервером ASP.NET Core:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <title>METANIT.COM</title>
6  <style>
7      td {padding:5px;}
8      button{margin: 5px;}
9  </style>
10 </head>
11 <body>
12     <h2>Список пользователей</h2>
13     <div>
14         <input type="hidden" id="userId" />
15         <p>
16             Имя:<br/>
17             <input id="userName" />
18         </p>
19         <p>
20             Возраст:<br />
21             <input id="userAge" type="number" />
22         </p>
23         <p>
24             <button id="saveBtn">Сохранить</button>
25             <button id="resetBtn">Сбросить</button>
26         </p>
27     </div>
28     <table>
29         <thead><tr><th>Имя</th><th>Возраст</th><th></th></tr></thead>
30         <tbody>
31         </tbody>
32     </table>
33
34     <script>
35     // Получение всех пользователей
36     async function getUsers() {
37         // отправляет запрос и получаем ответ
38         const response = await fetch("/api/users", {
39             method: "GET",
40             headers: { "Accept": "application/json" }
41         });
42         // если запрос прошел нормально
43         if (response.ok === true) {
44             // получаем данные
45             const users = await response.json();
46             const rows = document.querySelector("tbody");
47             // добавляем полученные элементы в таблицу
48             users.forEach(user => rows.append(row(user)));
49         }
50     }
```

```

51 // Получение одного пользователя
52 async function getUser(id) {
53     const response = await fetch(`/api/users/${id}`, {
54         method: "GET",
55         headers: { "Accept": "application/json" }
56     });
57     if (response.ok === true) {
58         const user = await response.json();
59         document.getElementById("userId").value = user.id;
60         document.getElementById("userName").value = user.name;
61         document.getElementById("userAge").value = user.age;
62     }
63     else {
64         // если произошла ошибка, получаем сообщение об ошибке
65         const error = await response.json();
66         console.log(error.message); // и выводим его на консоль
67     }
68 }
69 // Добавление пользователя
70 async function createUser(userName, userAge) {
71
72     const response = await fetch("api/users", {
73         method: "POST",
74         headers: { "Accept": "application/json", "Content-Type": "application/json" },
75         body: JSON.stringify({
76             name: userName,
77             age: parseInt(userAge, 10)
78         })
79     });
80     if (response.ok === true) {
81         const user = await response.json();
82         document.querySelector("tbody").append(row(user));
83     }
84     else {
85         const error = await response.json();
86         console.log(error.message);
87     }
88 }
89 // Изменение пользователя
90 async function editUser(userId, userName, userAge) {
91     const response = await fetch("api/users", {
92         method: "PUT",
93         headers: { "Accept": "application/json", "Content-Type": "application/json" },
94         body: JSON.stringify({
95             id: userId,
96             name: userName,
97             age: parseInt(userAge, 10)
98         })
99     });

```

```

100     if (response.ok === true) {
101         const user = await response.json();
102         document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));
103     }
104     else {
105         const error = await response.json();
106         console.log(error.message);
107     }
108 }
109 // Удаление пользователя
110 async function deleteUser(id) {
111     const response = await fetch(`/api/users/${id}`, {
112         method: "DELETE",
113         headers: { "Accept": "application/json" }
114     });
115     if (response.ok === true) {
116         const user = await response.json();
117         document.querySelector(`tr[data-rowid='${user.id}']`).remove();
118     }
119     else {
120         const error = await response.json();
121         console.log(error.message);
122     }
123 }
124
125 // сброс данных формы после отправки
126 function reset() {
127     document.getElementById("userId").value =
128     document.getElementById("userName").value =
129     document.getElementById("userAge").value = "";
130 }
131 // создание строки для таблицы
132 function row(user) {
133
134     const tr = document.createElement("tr");
135     tr.setAttribute("data-rowid", user.id);
136
137     const nameTd = document.createElement("td");
138     nameTd.append(user.name);
139     tr.append(nameTd);
140
141     const ageTd = document.createElement("td");
142     ageTd.append(user.age);
143     tr.append(ageTd);
144
145     const linksTd = document.createElement("td");
146
147     const editLink = document.createElement("button");
148     editLink.append("изменить");
149     editLink.addEventListener("click", async() => await getUser(user.id));
150     linksTd.append(editLink);

```

```

151         const removeLink = document.createElement("button");
152         removeLink.append("Удалить");
153         removeLink.addEventListener("click", async () => await deleteUser(user.id));
154
155         linksTd.append(removeLink);
156         tr.appendChild(linksTd);
157
158         return tr;
159     }
160     // сброс значений формы
161     document.getElementById("resetBtn").addEventListener("click", () => reset());
162
163     // отправка формы
164     document.getElementById("saveBtn").addEventListener("click", async () => {
165
166         const id = document.getElementById("userId").value;
167         const name = document.getElementById("userName").value;
168         const age = document.getElementById("userAge").value;
169         if (id === "")
170             await createUser(name, age);
171         else
172             await editUser(id, name, age);
173         reset();
174     });
175
176     // загрузка пользователей
177     getUsers();
178 </script>
179 </body>
180 </html>

```

Основная логика здесь заключена в коде **javascript**. При загрузке страницы в браузере получаем все объекты из БД с помощью функции **getUsers()**:



```

1  async function getUsers() {
2      // отправляет запрос и получаем ответ
3      const response = await fetch("/api/users", {
4          method: "GET",
5          headers: { "Accept": "application/json" }
6      });
7      // если запрос прошел нормально
8      if (response.ok === true) {
9          // получаем данные
10         const users = await response.json();
11         const rows = document.querySelector("tbody");
12         // добавляем полученные элементы в таблицу
13         users.forEach(user => rows.append(row(user)));
14     }
15 }

```

Для добавления строк в таблицу используется функция **row()**, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции **getUser()** получает с сервера выделенного пользователя:

```

1  async function getUser(id) {
2      const response = await fetch(`/api/users/${id}`, {
3          method: "GET",
4          headers: { "Accept": "application/json" }
5      });
6      if (response.ok === true) {
7          const user = await response.json();
8          document.getElementById("userId").value = user.id;
9          document.getElementById("userName").value = user.name;
10         document.getElementById("userAge").value = user.age;
11     }
12     else {
13         // если произошла ошибка, получаем сообщение об ошибке
14         const error = await response.json();
15         console.log(error.message); // и выводим его на консоль
16     }
17 }

```

И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит **id** пользователя, мы можем узнать, какое действие выполняется – добавление или редактирование. Если **id** не установлен (равен пустой строке), то выполняется функция **createUser**, которая отправляет данные в **POST**-запросе:

```
1 async function createUser(userName, userAge) {
2
3     const response = await fetch("api/users", {
4         method: "POST",
5         headers: { "Accept": "application/json", "Content-Type": "application/json" },
6         body: JSON.stringify({
7             name: userName,
8             age: parseInt(userAge, 10)
9         })
10    });
11    if (response.ok === true) {
12        const user = await response.json();
13        document.querySelector("tbody").append(row(user));
14    }
15    else {
16        const error = await response.json();
17        console.log(error.message);
18    }
19 }
```

Если же ранее пользователь был загружен на форму, и в скрытом поле сохранился его **id**, то выполняется функция **editUser**, которая отправляет **PUT**-запрос:

```
1  async function editUser(userId, userName, userAge) {
2      const response = await fetch("api/users", {
3          method: "PUT",
4          headers: { "Accept": "application/json", "Content-Type": "application/json" },
5          body: JSON.stringify({
6              id: userId,
7              name: userName,
8              age: parseInt(userAge, 10)
9          })
10     });
11     if (response.ok === true) {
12         const user = await response.json();
13         document.querySelector(`tr[data-rowid='${user.id}']`).replaceWith(row(user));
14     }
15     else {
16         const error = await response.json();
17         console.log(error.message);
18     }
19 }
```

И функция **deleteUser()** посылает на сервер запрос типа **DELETE** на удаление пользователя, и при успешном удалении на сервере удаляет объект по **id** из списка объектов **Person**.

Теперь запустим проект, и по умолчанию приложение отправит браузеру веб-страницу **index.html**, которая загрузит список объектов:

МЕТАНИТ.COM

localhost:7263

### Список пользователей

Имя:

Возраст:

Имя	Возраст		
Tom	37	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Bob	41	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Sam	24	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>

После этого мы сможем выполнять все базовые операции с пользователями – получение, добавление, изменение, удаление. Например, добавим нового пользователя:

МЕТАНИТ.COM

localhost:7263

### Список пользователей

Имя:

Возраст:

Имя	Возраст		
Tom	37	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Bob	41	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Sam	24	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>

МЕТАНИТ.COM

localhost:7263

### Список пользователей

Имя:

Возраст:

Имя	Возраст		
Tom	37	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Bob	41	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Sam	24	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>
Alice	31	<input type="button" value="Изменить"/>	<input type="button" value="Удалить"/>

## 2. Загрузка файлов на сервер

Рассмотрим, как загружать файлы на сервер в ASP.NET Core. Все загружаемые файлы в ASP.NET Core представлены типом **IFormFile** из пространства имен **Microsoft.AspNetCore.Http**. Соответственно для получения отправленного файла в контроллере необходимо использовать **IFormFile**. Затем с помощью методов **IFormFile** мы можем произвести различные манипуляции файлом – получить его свойства, сохранить, получить его поток и т.д. Некоторые его **свойства и методы**:

**ContentType**: тип файла.

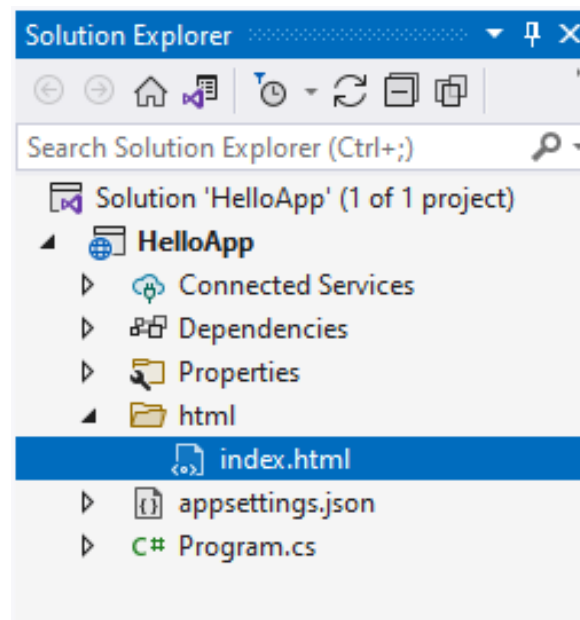
**FileName**: название файла.

**Length**: размер файла.

**CopyTo/CopyToAsync**: копирует файл в поток.

**OpenReadStream**: открывает поток файла для чтения.

Для тестирования данной возможности определим в проекте папку **html**, в которой создадим файл **index.html**.



Определим в файле **index.html** следующий код:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <meta name="viewport" content="width=device-width" />
6   <title>METANIT.COM</title>
7 </head>
8 <body>
9   <h2>Выберите файл для загрузки</h2>
10  <form action="upload" method="post" enctype="multipart/form-data">
11    <input type="file" name="uploads" /><br>
12    <input type="file" name="uploads" /><br>
13    <input type="file" name="uploads" /><br>
14    <input type="submit" value="Загрузить" />
15  </form>
16 </body>
17 </html>
```

В данном случае форма содержит набор элементов с типом **file**, через которые можно выбрать файлы для загрузки. В данном случае на форме три таких элемента, но их может быть и меньше, и больше.

А благодаря установке атрибута формы **enctype="multipart/form-data"** браузер будет знать, что вместе с формой надо передать файлы.

Отправляться файлы будут в запросе типа POST на адрес **"/upload"**.

Теперь в файле **Program.cs** определим код, который будет получать загружаемые файлы:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) =>
5 {
6     var response = context.Response;
7     var request = context.Request;
8
9     response.ContentType = "text/html; charset=utf-8";
10
11     if (request.Path == "/upload" && request.Method=="POST")
12     {
13         IFormFileCollection files = request.Form.Files;
14         // путь к папке, где будут храниться файлы
15         var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";
16         // создаем папку для хранения файлов
17         Directory.CreateDirectory(uploadPath);
18
19         foreach (var file in files)
20         {
21             // путь к папке uploads
22             string fullPath = $"{uploadPath}/{file.FileName}";
23
24             // сохраняем файл в папку uploads
25             using (var fileStream = new FileStream(fullPath, FileMode.Create))
26             {
27                 await file.CopyToAsync(fileStream);
28             }
29         }
30         await response.WriteAsync("Файлы успешно загружены");
31     }
32     else
33     {
34         await response.sendFileAsync("html/index.html");
35     }
36 });
37
38 app.Run();
```

Здесь если запрос приходит по адресу **"/upload"**, а сам запрос представляет запрос типа **POST**, то приложение получает коллекцию загруженных файлов с помощью свойства **Request.Form.Files**, которое представляет тип **IFormFileCollection**:

```
1 IFormFileCollection files = request.Form.Files;
```

Далее определяем каталог для загружаемых файлов (предполагается, что файлы будут храниться в каталоге **"uploads"**, которая располагается в папке приложения):

```
1 var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";
```

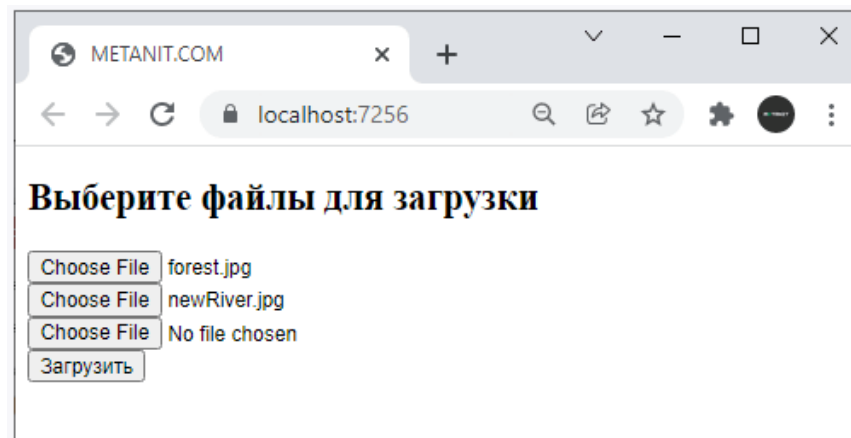
Если такой папки нет, то создаем ее. Затем перебираем всю коллекцию файлов.

```
1 foreach (var file in files)
```

Каждый отдельный файл в этой коллекции представляет тип **IFormFile**. Для копирования файла в нужный каталог создается поток **FileStream**, в который записывается файл с помощью метода **CopyToAsync**.

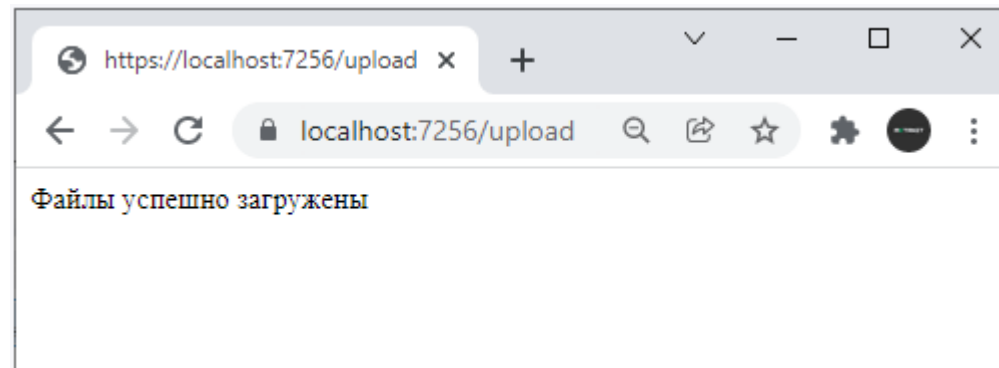
```
1 using (var fileStream = new FileStream(fullPath, FileMode.Create))
2 {
3     await file.CopyToAsync(fileStream);
4 }
```

Если запрос идет по другому адресу и/или не представляет тип **POST**, то отправляем клиенту html-страницу **index.html**. Обратимся к приложению и выберем файлы для загрузки:

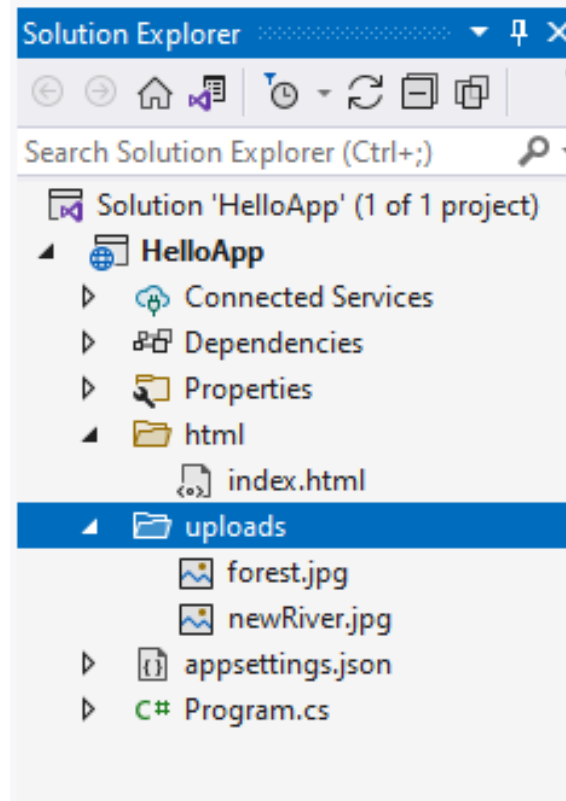




И после успешной загрузки нам отобразиться соответствующее сообщение:



А в каталоге проекта будет создана папка **uploads**, в которой появятся загруженные файлы:



### 3. Метод Use

Метод расширения **Use()** добавляет компонент **middleware**, который позволяет передать обработку запроса далее следующим в конвейере компонентам. Он имеет следующие версии:

```
1 public static IApplicationBuilder Use(this IApplicationBuilder app, Func<HttpContext, Func<Task>, Task> middleware);  
2 public static IApplicationBuilder Use(this IApplicationBuilder app, Func<HttpContext, RequestDelegate, Task> middleware)
```

Метод **Use()** реализован как метод расширения для типа **IApplicationBuilder**, соответственно мы можем вызвать данный метод у объекта **WebApplication** для добавления **middleware** в приложение. В обеих версиях метод **Use** принимает некоторое действие, которое имеет два параметра и возвращает объект **Task**.

Первый параметр делегата **Func**, который передается в метод **Use()**, представляет объект **HttpContext**. Этот объект позволяет получить данные запроса и управлять ответом клиенту.

Второй параметр делегата представляет другой делегат – **Func<Task>** или **RequestDelegate**. Этот делегат представляет следующий в конвейере компонент **middleware**, которому будет передаваться обработка запроса.

В общем случае применение метода **Use()** выглядит следующим образом:

```
1 app.Use(async (context, next) =>  
2 {  
3     // действия перед передачи запроса в следующий middleware  
4     await next.Invoke();  
5     // действия после обработки запроса следующим middleware  
6 });
```

Работа **middleware** разбивается на две части.

**Middleware** выполняет некоторую начальную обработку запроса до вызова **await next.Invoke()**.

Затем вызывается метод **next.Invoke()**, который передает обработку запроса следующему компоненту в конвейере.

Когда следующий в конвейере компонент закончил обработку запроса возвращается обратно в текущий компонент, и выполняются действия, которые идут после вызова **await next.Invoke()**.

Таким образом, **middleware** в методе **Use** выполняет действия до следующего в конвейере компонента и после него.

Рассмотрим метод **Use()** на примере:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 string date = "";
5
6 app.Use(async(context, next) =>
7 {
8     date = DateTime.Now.ToShortDateString();
9     await next.Invoke();           // вызываем middleware из app.Run
10    Console.WriteLine($"Current date: {date}"); // Current date: 08.12.2021
11 });
12
13 app.Run(async(context) => await context.Response.WriteAsync($"Date: {date}"));
14
15 app.Run();
```

В данном случае мы используем перегрузку метода **Use**, которая в качестве параметров принимает контекст запроса – объект **HttpContext** и делегат **Func<Task>**, который представляет собой ссылку на следующий в конвейере компонент **middleware**.

**Middleware** в методе **app.Use()** реализует простейшую задачу – присваивает переменной **date** текущую дату в виде строки и затем передает обработку запроса следующим компонентам **middleware** в конвейере. То есть при вызове **await next.Invoke()** обработка запроса перейдет к тому компоненту, который установлен в методе **app.Run()**.

В итоге обработка запроса будет выглядеть следующим образом:

1. Вызов компонента **app.Use**.
2. Установка значения переменной **date**:

```
1 date = DateTime.Now.ToShortDateString();
```

3. Вызов **await next.Invoke()**. Управление переходит следующему компоненту в конвейере – к **app.Run**.
4. В **middleware** из **app.Run()** отправляет клиенту текущую дату в качестве ответа с помощью метода **context.Response.WriteAsync()**:

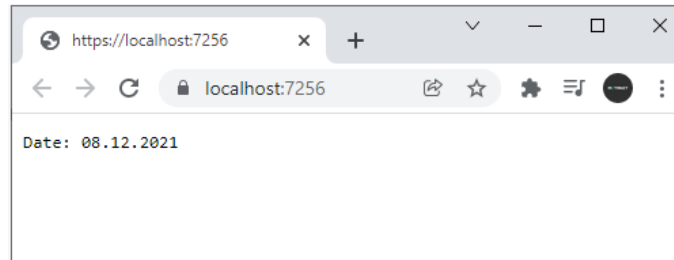
```
1 await context.Response.WriteAsync($"Date: {date}");
```

5. Метод **app.Run** закончил свою работу, и управление обработкой возвращается к **middleware** в методе **app.Use**. Начинает выполняться та часть кода, которая идет после **await next.Invoke()**. В этой части выполняется условное логгирование – на консоль выводится значение переменной **date**:

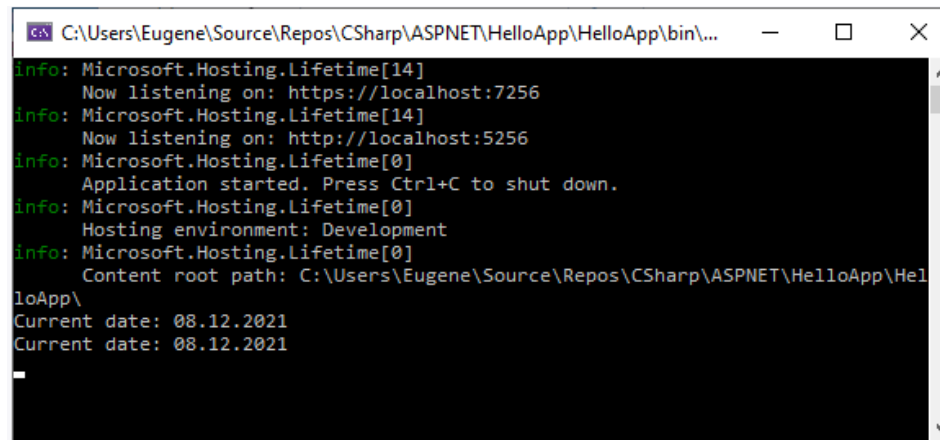
```
1 Console.WriteLine($"Current date: {date}");
```

После этого обработка запроса завершена.

В итоге в веб-браузере мы увидим следующее сообщение:



А в консоли запущенного приложения мы увидим значение переменной **date**, которое выводится в **middleware** из метода **app.Use**:



## Отправка ответа

При использовании метода **Use** и передаче выполнения следующему делегату следует учитывать, что не рекомендуется вызывать метод **next.Invoke** после метода **Response.WriteAsync()**. Компонент **middleware** должен либо генерировать ответ с помощью **Response.WriteAsync**, либо вызывать следующий делегат посредством **next.Invoke**, но не выполнять оба этих действия одновременно. Так, как согласно документации, последующие изменения объекта **Response** могут привести к нарушению протокола, например, будет послано больше байт, чем указано в заголовке **Content-Length**, либо могут привести к нарушению тела ответа, например, футер страницы HTML запишется в CSS-файл.

То есть, к примеру, следующая обработка запроса не рекомендуется:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Use(async (context, next) =>
5 {
6     await context.Response.WriteAsync("<p>Hello world!</p>");
7     await next.Invoke();
8 });
9
10 app.Run(async (context) =>
11 {
12     //await Task.Delay(10000); // можно поставить задержку
13     await context.Response.WriteAsync("<p>Good bye, World...</p>");
14 });
15
16 app.Run();
```

## Использование делегат RequestDelegate

В примере выше использовалась версия метода **Use()**, которая использует делегат **Func<Task>**. Подобным образом можно использовать и другую версию, где используется делегат **RequestDelegate**. Единственное – при вызове делегата (то есть фактически следующего в конвейере компонента) необходимо передавать делегату объект **HttpContext**:

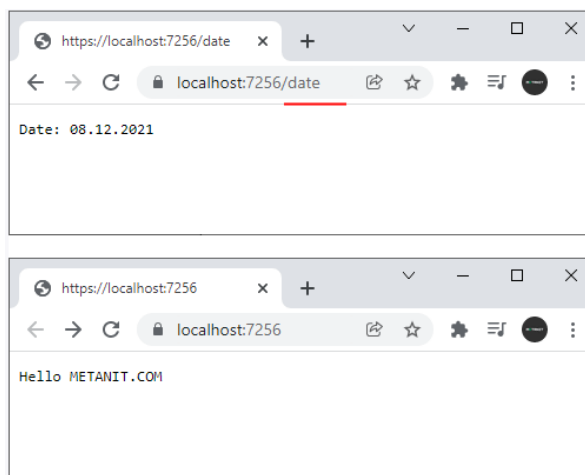
```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 string date = "";
5
6 app.Use(async(context, next) =>
7 {
8     date = DateTime.Now.ToShortDateString();
9     await next.Invoke(context);           // здесь next - RequestDelegate
10    Console.WriteLine($"Current date: {date}"); // Current date: 08.12.2021
11 });
12
13 app.Run(async(context) => await context.Response.WriteAsync($"Date: {date}"));
14
15 app.Run();
```

## Терминальный компонент middleware

**Middleware** в методе **Use()** необязательно должен вызывать к следующему в конвейере компоненту. Вместо этого он может завершить обработку запроса. В этом случае он может выступать в роли такого же терминального компонента **middleware**, а и компоненты из метода **Run()**. Например:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Use(async(context, next) =>
5 {
6     string? path = context.Request.Path.Value?.ToLower();
7     if (path == "/date")
8     {
9         await context.Response.WriteAsync($"Date: {DateTime.Now.ToShortDateString()}");
10    }
11    else
12    {
13        await next.Invoke();
14    }
15 });
16
17 app.Run(async(context) => await context.Response.WriteAsync($"Hello METANIT.COM"));
18
19 app.Run();
```

Здесь **middleware** в **app.Use** проверяет запрошенный адрес – если он содержит **"/date"**, то клиенту отправляется текущая дата. Иначе обработка запроса передается дальше в **app.Run**.



Причем в принципе мы можем использовать компонент в **app.Use** как единственный и соответственно терминальный:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3 app.Use(async (HttpContext context, Func<Task> next) =>
4 {
5     await context.Response.WriteAsync("Hello Work!");
6 });
7
8 app.Run();
```

Однако в данном случае для большей производительности лучше использовать **app.Run()**, если нам надо определить лишь один компонент, который в принципе не передает запрос дальше по конвейеру.

## Вынесение компонентов в методы

Также можно вынести все inline-компоненты в отдельные методы:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Use(GetDate);
5 app.Run(async (context) => await context.Response.WriteAsync("Hello METANIT.COM"));
6 app.Run();
7 async Task GetDate(HttpContext context, Func<Task> next)
8 {
9     string? path = context.Request.Path.Value?.ToLower();
10    if (path == "/date")
11    {
12        await context.Response.WriteAsync($"Date: {DateTime.Now.ToShortDateString()}");
13    }
14    else
15    {
16        await next.Invoke();
17    }
18 }
```

Подобным образом можно использовать и другую версию метода **Use**, в которой используется делегат **RequestDelegate**:

```
1  async Task GetDate(HttpContext context, RequestDelegate next)
2  {
3      string? path = context.Request.Path.Value?.ToLower();
4      if (path == "/date")
5      {
6          await context.Response.WriteAsync($"Date: {DateTime.Now.ToShortDateString()}");
7      }
8      else
9      {
10         await next.Invoke(context);
11     }
12 }
```



## 4. Создание ветки конвейера. UseWhen и MapWhen

### UseWhen

Метод **UseWhen()** на основании некоторого условия позволяет создать ответвление конвейера при обработке запроса:

```
1 public static IApplicationBuilder UseWhen (this IApplicationBuilder app, Func<HttpContext,bool> predicate, Action<IApplicationBuilder>
```

```
1 public static IApplicationBuilder UseWhen (this IApplicationBuilder app, Func<HttpContext,bool>  
predicate, Action<IApplicationBuilder> configuration);
```

Как и **Use()**, метод **UseWhen()** реализован как метод расширения для типа **IApplicationBuilder**.

В качестве параметра он принимает делегат **Func<HttpContext,bool>** – некоторое условие, которому должен соответствовать запрос. В этот делегат передается объект **HttpContext**. А возвращаемым типом должен быть тип **bool** – если запрос соответствует условию, то возвращается **true**, иначе возвращается **false**.

Последний параметр метода – делегат **Action<IApplicationBuilder>** представляет некоторые действия над объектом **IApplicationBuilder**, который передается в делегат в качестве параметра.

Рассмотрим небольшой пример:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseWhen(
5     context => context.Request.Path == "/time", // если путь запроса "/time"
6     appBuilder =>
7     {
8         // логируем данные - выводим на консоль приложения
9         appBuilder.Use(async (context, next) =>
10         {
11             var time = DateTime.Now.ToShortTimeString();
12             Console.WriteLine($"Time: {time}");
13             await next(); // вызываем следующий middleware
14         });
15
16         // отправляем ответ
17         appBuilder.Run(async context =>
18         {
19             var time = DateTime.Now.ToShortTimeString();
20             await context.Response.WriteAsync($"Time: {time}");
21         });
22     });
23
24 app.Run(async context =>
25 {
26     await context.Response.WriteAsync("Hello METANIT.COM");
27 });
28
29 app.Run();
```

В данном случае метод **app.UseWhen()** в качестве первого параметра получает следующее условие:

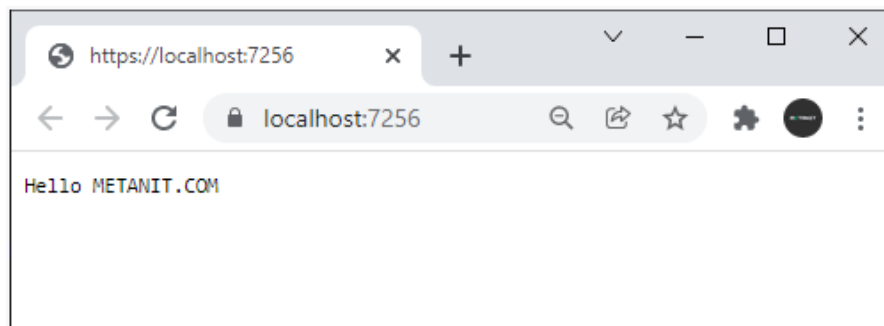
```
1 context => context.Request.Path == "/time"
```

Второй параметр определяет действие, в котором создается ответвление конвейера:

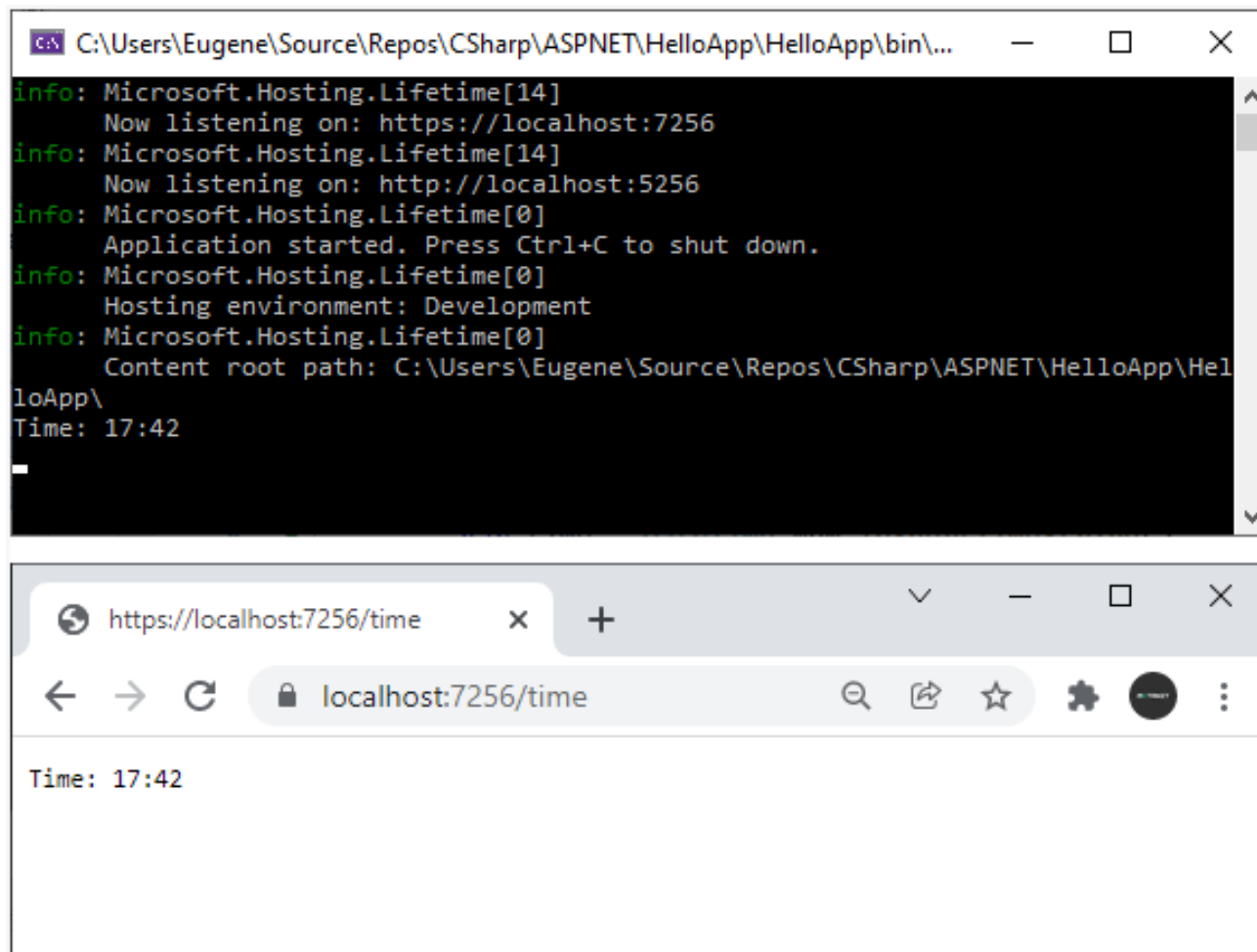
```
1 appBuilder =>
2 {
3     // логируем данные - выводим на консоль приложения
4     appBuilder.Use(async (context, next) =>
5     {
6         var time = DateTime.Now.ToShortTimeString();
7         Console.WriteLine($"Time: {time}");
8         await next(); // вызываем следующий middleware
9     });
10
11     appBuilder.Run(async context =>
12     {
13         var time = DateTime.Now.ToShortTimeString();
14         await context.Response.WriteAsync($"Time: {time}");
15     });
16 }
```

В данном действии в конвейер обработки запроса встраиваются два **middleware** - с помощью методов **Use()** и **Run()**. В первом **middleware** логируем это время на консоль приложения. Во втором – терминальном компоненте **middleware** отправляем информацию о времени в ответ клиенту.

Если мы обращаемся к приложению по пути, который отличается от **"/time"**, то условие в методе **UseWhen()** ложно, поэтому ответвления конвейера не выполняется. И выполняется **middleware** из метода **app.Run()**:



Однако если мы обращаемся по пути `"/time"`, то условие в методе `app.UseWhen()` будет истинно. Соответственно будет выполняться ответвление конвейера, который будет обрабатывать запрос. В итоге на консоль приложения, а также в браузере будет выводиться текущее время.



The image shows two overlapping windows. The top window is a console application window with the title bar `C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\bin\...`. The console output is as follows:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7256
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5256
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\
Time: 17:42
-
```

The bottom window is a web browser with the address bar showing `https://localhost:7256/time`. The page content displays `Time: 17:42`.

Стоит отметить, что создание ветки происходит один раз при запуске приложения. Например, в примере выше мы видим, что получение времени производится в обоих `middleware` во встраиваемой ветки. Но что будет, если вынести получение времени во вне и не дублировать в каждом `middleware`, например, следующим образом:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseWhen(
5     context => context.Request.Path == "/time", // если путь запроса "/time"
6     appBuilder =>
7     {
8         var time = DateTime.Now.ToShortTimeString();
9         // логируем данные - выводим на консоль приложения
10        appBuilder.Use(async (context, next) =>
11        {
12            Console.WriteLine($"Time: {time}");
13            await next(); // вызываем следующий middleware
14        });
15
16        // отправляем ответ
17        appBuilder.Run(async context =>
18        {
19            await context.Response.WriteAsync($"Time: {time}");
20        });
21    });
22
23 app.Run(async context =>
24 {
25     await context.Response.WriteAsync("Hello METANIT.COM");
26 });
27
28 app.Run();
```

В этом случае время будет устанавливаться один раз - при запуске приложения и создании ветки в конвейер. Соответственно вне зависимости от того, сколько раз мы будем обращаться к приложению по пути `"/time"`, мы будем получать одно и то же время.

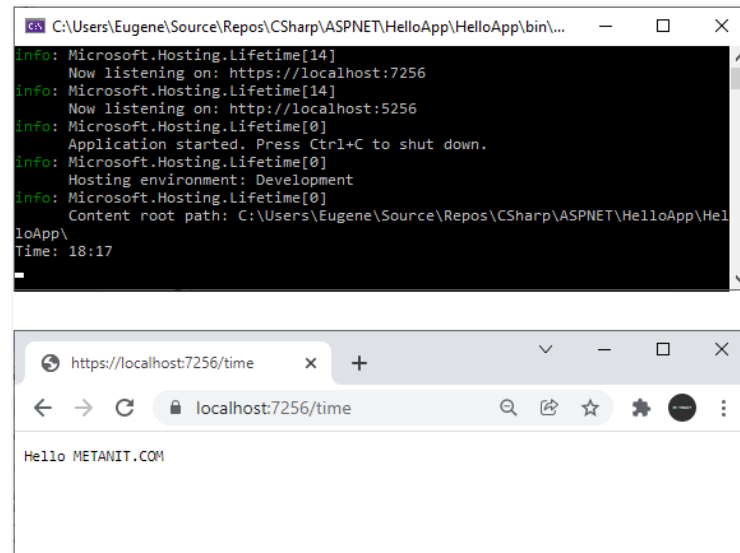
В примере выше ветка конвейера завершалась терминальным компонентом, поэтому остальные действия в основной части конвейера не выполнялись. Однако мы можем также передать запрос на обработку из ветки в основной поток конвейера:

```

1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseWhen(
5     context => context.Request.Path == "/time", // условие: если путь запроса "/time"
6     appBuilder =>
7     {
8         appBuilder.Use(async (context, next) =>
9         {
10             var time = DateTime.Now.ToShortTimeString();
11             Console.WriteLine($"Time: {time}");
12             await next(); // вызываем следующий middleware
13         });
14 });
15
16 app.Run(async context =>
17 {
18     await context.Response.WriteAsync("Hello METANIT.COM");
19 });
20
21 app.Run();

```

В данном случае, если запрос идет по пути **"/time"**, сначала срабатывает ветка конвейера с компонентом, который логирует время на консоль. А затем выполняется компоненте из **app.Run()**, который отправляет сообщение "Hello METANIT.COM":



Для большей читабельности также можно было бы вынести действия по созданию ветки конвейера в отдельный метод:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.UseWhen(
5     context => context.Request.Path == "/time", // условие: если путь запроса "/time"
6     HandleTimeRequest
7 );
8
9 app.Run(async context =>
10 {
11     await context.Response.WriteAsync("Hello METANIT.COM");
12 });
13
14 app.Run();
15
16 void HandleTimeRequest(IApplicationBuilder appBuilder)
17 {
18     appBuilder.Use(async (context, next) =>
19     {
20         var time = DateTime.Now.ToShortTimeString();
21         Console.WriteLine($"current time: {time}");
22         await next(); // вызываем следующий middleware
23     });
24 }
```

## MapWhen

Метод **MapWhen()**, как и метод **UseWhen()**, на основании некоторого условия позволяет создать ответвление конвейера:

```
1 public static IApplicationBuilder MapWhen (this IApplicationBuilder app, Func<HttpContext, bool> predicate, Action<IApplicationBuilder>
```

```
1 public static IApplicationBuilder MapWhen (this IApplicationBuilder app, Func<HttpContext, bool>
predicate, Action<IApplicationBuilder> configuration);
```

Метод **MapWhen()** также реализован как метод расширения для типа **IApplicationBuilder**, принимает те же параметры, что и **UseWhen()**, и работает во многом аналогичным образом:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.MapWhen(
5     context => context.Request.Path == "/time", // условие: если путь запроса "/time"
6     appBuilder => appBuilder.Run(async context =>
7     {
8         var time = DateTime.Now.ToShortTimeString();
9         await context.Response.WriteAsync($"current time: {time}");
10    })
11 );
12
13 app.Run(async context =>
14 {
15     await context.Response.WriteAsync("Hello METANIT.COM");
16 });
17
18 app.Run();
```

Здесь опять же, если запрошен путь **"/time"**, то срабатывает ветка конвейера, созданная методом **app.MapWhen()**, в которой клиенту отправляется текущее время. Если путь запроса другой, то срабатывается основной поток конвейера, в котором отправляется сообщение "Hello METANIT.COM".