



# Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 4. «Основы в ASP.NET Core. Часть 4»

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

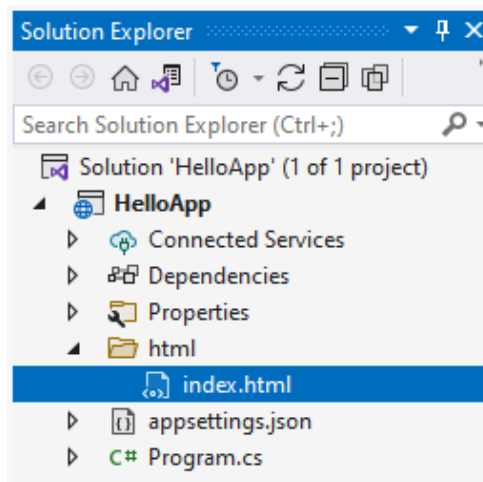
# Учебные вопросы:

1. Отправка форм.
2. Переадресация.
3. Отправка и получение json.

# 1. Отправка форм

Нередко данные отправляются на сервер с помощью форм **html**, обычно в запросе типа **POST**. Для получения подобных данных в классе **HttpRequest** определено свойство **Form**. Рассмотрим, как мы можем получить подобные данные.

Прежде всего определим в проекте в папке html файл **index.html**.



## Отправка форм в ASP.NET Core и C#

Определим в нем следующее содержимое:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>METANIT.COM</title>
6 </head>
7 <body>
8     <h2>User form</h2>
9     <form method="post" action="postuser">
10         <p>Name: <input name="name" /></p>
11         <p>Age: <input name="age" type="number" /></p>
12         <input type="submit" value="Send" />
13     </form>
14 </body>
15 </html>
```

Здесь определена форма условно для ввода данных пользователя, которая в запросе типа **POST** (атрибут **method="post"**) отправляет данные по адресу **"postuser"** (атрибут **action="postuser"**)

На форме определены два поля ввода. Первое поле предназначено для ввода имени пользователя. Второе поле – для ввода возраста пользователя.

Для получения этих данных определим в файле Program.cs следующий код:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) =>
5 {
6     context.Response.ContentType = "text/html; charset=utf-8";
7
8     // если обращение идет по адресу "/postuser", получаем данные формы
9     if (context.Request.Path == "/postuser")
10    {
11        var form = context.Request.Form;
12        string name = form["name"];
13        string age = form["age"];
14        await context.Response.WriteAsync($"<div><p>Name: {name}</p><p>Age: {age}</p></div>");
15    }
16    else
17    {
18        await context.Response.sendFileAsync("html/index.html");
19    }
20 });
21
22 app.Run();
```

Здесь, если запрошен адрес **"/postuser"**, то предполагается, что отправлена некоторая форма. Сначала получаем отправленную форму в переменную **form**:

```
1 var form = context.Request.Form;
```

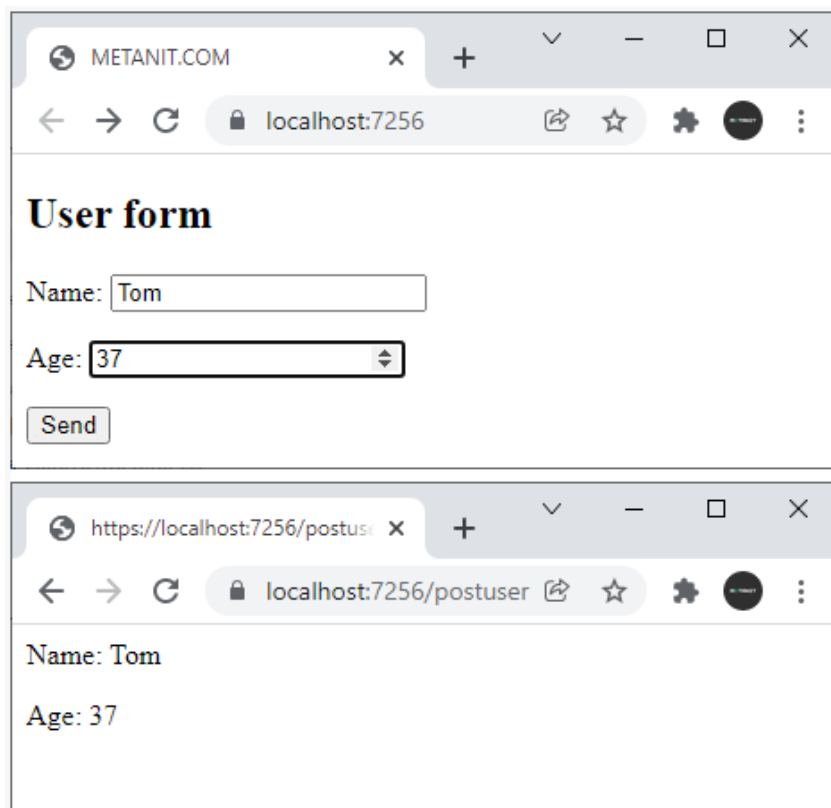
Свойство **Request.Form** возвращает объект **IFormCollection** – своего рода словарь, где по ключу можно получить значение элемента. При этом в качестве ключей выступают названия полей форм (значения атрибутов name элементов формы):

```
1 <input name="age" type="number" />
```

Так, в данном случае название поля (значение атрибута name) равно **"age"**. Соответственно в **Request.Form** по этому имени мы можем получить его значение:

```
1 string age = form["age"];
```

После получения данных формы они отправляются обратно клиенту:



The image shows two browser windows. The top window, titled 'METANIT.COM', shows a form titled 'User form' at 'localhost:7256'. The form has a 'Name' field with 'Tom' and an 'Age' field with '37'. A 'Send' button is at the bottom. The bottom window, titled 'https://localhost:7256/postuser', shows the result of the submission: 'Name: Tom' and 'Age: 37'.

**Request.Form и получение форм в ASP.NET Core и C#**

## Получение массивов

Усложним задачу и добавим в форму на странице `index.html` несколько полей, которые будут представлять массив:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>METANIT.COM</title>
6 </head>
7 <body>
8   <h2>User form</h2>
9   <form method="post" action="postuser">
10     <p>Name: <br />
11       <input name="name" />
12     </p>
13     <p>Age: <br />
14       <input name="age" type="number" />
15     </p>
16     <p>
17       Languages:<br />
18       <input name="languages" /><br />
19       <input name="languages" /><br />
20       <input name="languages" /><br />
21     </p>
22     <input type="submit" value="Send" />
23   </form>
24 </body>
25 </html>
```

Здесь добавлено три поля ввода, которые имеют одно и то же имя. Поэтому при их отправке будет формироваться массив из трех значений.

Теперь получим эти значения в коде C#:

```

1  var builder = WebApplication.CreateBuilder();
2  var app = builder.Build();
3
4  app.Run(async (context) =>
5  {
6      context.Response.ContentType = "text/html; charset=utf-8";
7
8      // если обращение идет по адресу "/postuser", получаем данные формы
9      if (context.Request.Path == "/postuser")
10     {
11         var form = context.Request.Form;
12         string name = form["name"];
13         string age = form["age"];
14         string[] languages = form["languages"];
15         // создаем из массива languages одну строку
16         string langList = "";
17         foreach (var lang in languages)
18         {
19             langList += $" {lang}";
20         }
21         await context.Response.WriteAsync($"<div><p>Name: {name}</p>" +
22             $"<p>Age: {age}</p>" +
23             $"<div>Languages:{langList}</ul></div>");
24     }
25     else
26     {
27         await context.Response.sendFileAsync("html/index.html");
28     }
29 });
30
31 app.Run();

```

Поскольку параметр **"languages"** представляет массив, то и сопоставляться он будет с массивом строк:

```

1  string[] languages = form["languages"];

```

Для вывода на веб-страницу из этого массива формируется код html в виде строки:

The image displays two browser windows. The top window, titled 'METANIT.COM', shows a web form titled 'User form' at 'localhost:7256'. The form contains input fields for 'Name' (filled with 'Tom'), 'Age' (filled with '37'), and 'Languages' (filled with 'C#', 'JavaScript', and 'Kotlin' on separate lines). A 'Send' button is at the bottom. The bottom window, titled 'https://localhost:7256/postuser', shows the rendered HTML output of the form data: 'Name: Tom', 'Age: 37', and 'Languages: C# JavaScript Kotlin'.

**User form**

Name:  
Tom

Age:  
37

Languages:  
C#  
JavaScript  
Kotlin

Send

https://localhost:7256/postuser

Name: Tom

Age: 37

Languages: C# JavaScript Kotlin

Отправка массива элементов формы в ASP.NET Core и C#



Подобным образом можно передавать значения массива полей других типов, либо полей, которые представляют набор элементов, например, элемента **select**, который поддерживает множественный выбор:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>METANIT.COM</title>
6 </head>
7 <body>
8   <h2>User form</h2>
9   <form method="post" action="postuser">
10     <p>Name: <br />
11       <input name="name" />
12     </p>
13     <p>Age: <br />
14       <input name="age" type="number" />
15     </p>
16     <p>
17       Languages:<br />
18       <select multiple name="languages">
19         <option>C#</option>
20         <option>JavaScript</option>
21         <option>Kotlin</option>
22         <option>Java</option>
23       </select>
24     </p>
25     <input type="submit" value="Send" />
26   </form>
27 </body>
28 </html>
```

The image displays two browser windows. The top window, titled 'METANIT.COM', shows a 'User form' at 'localhost:7256'. The form contains three input fields: 'Name' with the value 'Tom', 'Age' with the value '34', and 'Languages' which is a multi-select dropdown menu with 'C#' and 'Java' selected. A 'Send' button is located below the form. The bottom window, titled 'https://localhost:7256/postuser', shows the result of the form submission. It displays the submitted data: 'Name: Tom', 'Age: 34', and 'Languages: C# Java'.

**User form**

Name:  
Tom

Age:  
34

Languages:  
C#  
JavaScript  
Kotlin  
Java

Send

https://localhost:7256/postuser

Name: Tom

Age: 34

Languages: C# Java

Отправка массива значений из формы html в ASP.NET Core и C#

## 2. Переадресация

Для выполнения переадресации у объекта **HttpResponse** определен метод **Redirect()**:

```
1 void Redirect(string location)
2 void Redirect(string location, bool permanent)
```

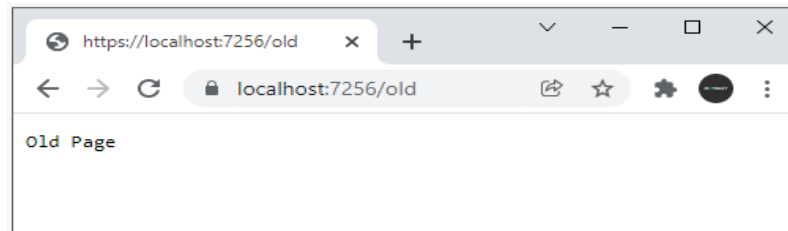
Первая версия выполняет временную переадресацию. В качестве параметра получает адрес для редиректа, а клиенту посылается статусный код 302.

Вторая версия метода также в качестве второго параметра получает булево значение, которое указывает, будет ли переадресация постоянной. Если этот параметр равен **true**, то переадресация будет постоянной, и в этом случае посылается статусный код 301. Если равен **false**, то переадресация временная, и посылается статусный код 302.

Допустим, у нас было следующее приложение:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) =>
5 {
6     if (context.Request.Path == "/old")
7     {
8         await context.Response.WriteAsync("Old Page");
9     }
10    else
11    {
12        await context.Response.WriteAsync("Main Page");
13    }
14 });
15
16 app.Run();
```

При обращении по адресу **"/old"** приложение посылает сообщение **"Old Page"**.



## Редирект в ASP.NET Core и C#

Но затем мы решили сделать переадресацию с адреса `"/old"` на `"/new"`. Используем для этого первую версию метода **Redirect**:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) =>
5 {
6     if (context.Request.Path == "/old")
7     {
8         context.Response.Redirect("/new");
9     }
10    else if (context.Request.Path == "/new")
11    {
12        await context.Response.WriteAsync("New Page");
13    }
14    else
15    {
16        await context.Response.WriteAsync("Main Page");
17    }
18 });
19
20 app.Run();
```

Теперь при обращении по адресу `"/old"` произойдет перенаправление на адрес `"/new"`.

В данном случае применяется редирект на локальный адрес в рамках приложения. Но также можно использовать редирект на внешние ресурсы:

```
1 if (context.Request.Path == "/old")
2 {
3     context.Response.Redirect("https://www.google.com/search?q=metanit.com");
4 }
```

### 3. Отправка и получение json

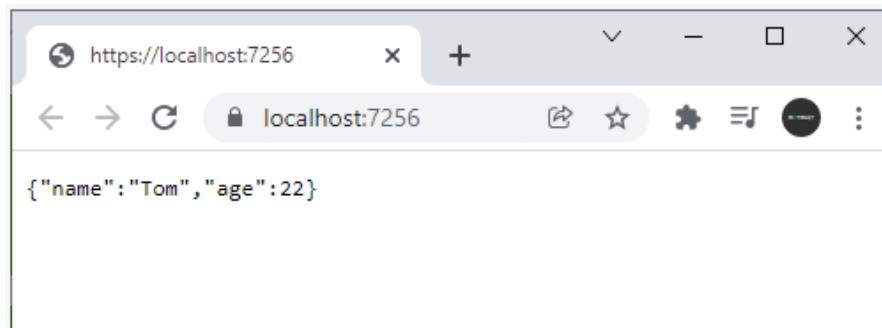
JSON является распространенным форматом для передачи данных. Рассмотрим, как мы можем посылать и получить данные json.

#### Отправка JSON. Метод WriteAsJsonAsync

Для отправки json можно воспользоваться методом **WriteAsJson()/WriteAsJsonAsync()** объекта **HttpResponse**. Этот метод позволяет сериализовать переданные в него объекты в формат **JSON** и автоматически для заголовка **"content-type"** устанавливает значение **"application/json; charset=utf-8"**:

```
1 var builder = WebApplication.CreateBuilder();
2 var app = builder.Build();
3
4 app.Run(async (context) =>
5 {
6     Person tom = new("Tom", 22);
7     await context.Response.WriteAsJsonAsync(tom);
8 });
9
10 app.Run();
11
12 public record Person(string Name, int Age);
```

В данном случае клиенту отправляется объект типа **Person**, который представляет класс – **record**, однако это может быть и обычный класс:



Отправка json с помощью WriteAsJson в ASP.NET Core и C#

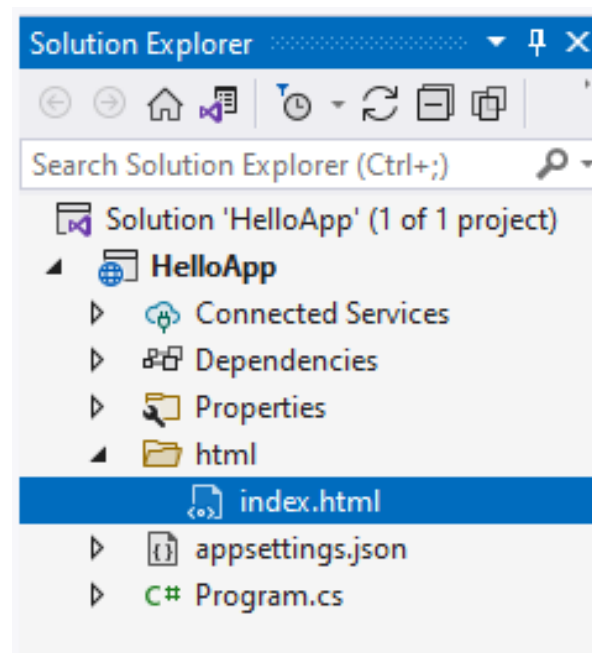
Хотя можно было бы воспользоваться и стандартным методом **WriteAsync()**:

```
1 app.Run(async (context) =>
2 {
3     var response = context.Response;
4     response.Headers.ContentType = "application/json; charset=utf-8";
5     await response.WriteAsync("{\"name':'Tom', 'age':37}");
6 });
```

### Получение JSON. Метод ReadFromJsonAsync

Для получения из запроса объект в формате **JSON** в классе **HttpRequest** определен метод **ReadFromJsonAsync()**. Он позволяет сериализовать данные в объект определенного типа.

Например, создадим в проекте папку **html**, в которой определим новый файл **index.html**.



Отправка объекта json на сервер ASP.NET Core в C#

В файле **index.html** определим следующий код:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>METANIT.COM</title>
6 </head>
7 <body>
8   <h2>User form</h2>
9   <div id="message"></div>
10  <div>
11    <p>Name: <br />
12      <input name="userName" id="userName" />
13    </p>
14    <p>Age: <br />
15      <input name="userAge" id="userAge" type="number" />
16    </p>
17    <button id="sendBtn">Send</button>
18  </div>
19  <script>
20    document.getElementById("sendBtn").addEventListener("click", send);
21    async function send() {
22      const response = await fetch("/api/user", {
23        method: "POST",
24        headers: { "Accept": "application/json", "Content-Type": "application/json" },
25        body: JSON.stringify({
26          name: document.getElementById("userName").value,
27          age: document.getElementById("userAge").value
28        })
29      });
30      const message = await response.json();
31      document.getElementById("message").innerText = message.text;
32    }
33  </script>
34 </body>
35 </html>
```

Здесь по нажатию на кнопку с помощью функции **fetch()** по адресу **"/api/user"** будет отправляться объект со свойствами **name** и **age**, значения для которых берутся из полей формы. В ответ от сервера веб-страница также получает объект в формате json, в котором имеется свойство **text** – свойство, которое хранит сообщение от сервера.

Теперь в файле **Program.cs** определим код для получения данных, отправляемых веб-страницей:

```
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     if (request.Path == "/api/user")
9     {
10         var message = "Некорректные данные"; // содержание сообщения по умолчанию
11         try
12         {
13             // пытаемся получить данные json
14             var person = await request.ReadFromJsonAsync<Person>();
15             if (person != null) // если данные сконвертированы в Person
16                 message = $"Name: {person.Name} Age: {person.Age}";
17         }
18         catch { }
19         // отправляем пользователю данные
20         await response.WriteAsJsonAsync(new { text = message });
21     }
22     else
23     {
24         response.ContentType = "text/html; charset=utf-8";
25         await response.sendFileAsync("html/index.html");
26     }
27 });
28
29 app.Run();
30
31 public record Person(string Name, int Age);
```

В данном случае, если обращение идет по адресу **"/api/user"**, то получаем данные в формате json. При обращениях по другим адресам просто посылаем веб-страницу **index.html**.



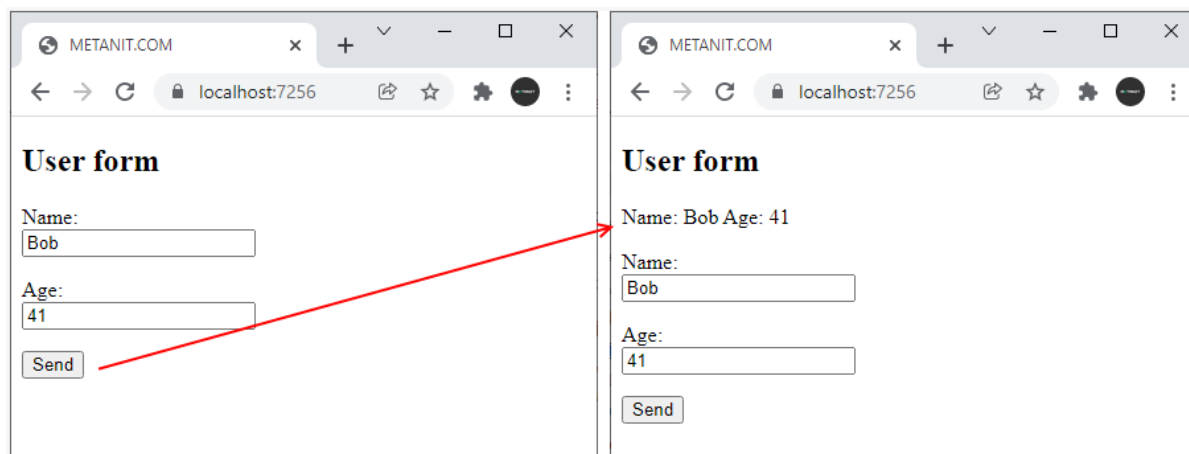
Метод **ReadFromJsonAsync()** десериализует полученные данные в объект определенного типа – в данном случае типа **Person**:

```
1 var person = await request.ReadFromJsonAsync<Person>();
2 if (person != null) // если данные сконвертированы в Person
3     message = $"Name: {person.Name} Age: {person.Age}";
```

Таким образом, здесь результат вызова этого метода – значение переменной **person** будет представлять объект **Person**.

Но стоит отметить, что если данные запроса не представляют объект **JSON**, либо если метод **ReadFromJsonAsync()** не смог связать данные запроса со свойствами класса **Person**, то вызов этого метода сгенерирует исключение. Поэтому в данном случае вызов метода помещается в конструкцию **try..catch**. Однако нельзя не отметить, что **try..catch** здесь является узким местом, и далее мы посмотрим, как от него избавиться.

И в конце в ответ посылает анонимный объект, который также сериализуется в json с некоторым сообщением, которое хранится в свойстве **text**. При получении этого сообщения оно выводится на веб-страницу.



**ReadFromJsonAsync и чтение данных json в ASP.NET Core и C#**

Стоит отметить, что проверять на наличие json в запросе можно с помощью метода **HasJsonContentType()** – он возвращает **true**, если клиент прислал json:

```
1 if (request.HasJsonContentType())
2 {
3     var person = await request.ReadFromJsonAsync<Person>();
4     if (person != null)
5         responseText = $"Name: {person.Name} Age: {person.Age}";
6 }
```

## Настройка сериализации

При получении данных в формате json мы можем столкнуться с рядом проблем. Хотя бы взять предыдущий пример, где мы вынуждены были помещать вызов метода **ReadFromJsonAsync** в конструкцию – **try..catch**. Например, если мы не введем в поля формы никаких значений, то стандартный механизм привязки значений не сможет связать данные запроса со свойством **Age**. И мы получим исключение.

Аналогичный пример, когда данные json не совсем соответствуют определению типа, в который надо выполнить десериализацию:

```
1  const response = await fetch("/api/user", {
2    method: "POST",
3    headers: { "Accept": "application/json", "Content-Type": "application/json" },
4    body: JSON.stringify({
5      userName: "Tom",
6      userAge: 22
7    })
8  });
```

Здесь названия свойств отправляемого объекта не соответствуют названиям свойств типа **Person** в C#. Однако объект **Person** все равно будет создан, просто его свойства получат значения по умолчанию (**null** для свойства **Name** и **0** для свойства **Age**).

Другой пример – отправляемые данные не соответствуют по типу:

```
1  const response = await fetch("/api/user", {
2    method: "POST",
3    headers: { "Accept": "application/json", "Content-Type": "application/json" },
4    body: JSON.stringify({
5      name: "Tom",
6      age: "twenty-two"
7    })
8  });
```

Здесь свойство **"age"** представляет строку и не сможет быть сконвертировано в значение типа **int**. В итоге при отправке подобных данных на сервере возникнет исключение типа **System.Text.Json.JsonException**, а клиент получит информацию об исключении.

В обоих выше приведенных примерах в зависимости от задачи можно использовать различные решения – обрабатывать исключения, встраивать дополнительные **middleware** для отлова подобных ситуаций и так далее. Одним из решений подобных проблем также может быть настройка сериализации/десериализации с помощью параметра типа **JsonSerializerOptions**, которое может передаваться в метод **ReadFromJsonAsync()**.

```
1 ReadFromJsonAsync<T>(JsonSerializerOptions options);
```

Так, изменим код файла **Program.cs**:

```
1 using System.Text.Json;
2 using System.Text.Json.Serialization;
3
4 var builder = WebApplication.CreateBuilder();
5 var app = builder.Build();
6
7 app.Run(async (context) =>
8 {
9     var response = context.Response;
10    var request = context.Request;
11    if (request.Path == "/api/user")
12    {
13        var responseText = "Некорректные данные"; // содержание сообщения по умолчанию
14
15        if (request.HasJsonContentType())
16        {
17            // определяем параметры сериализации/десериализации
18            var jsonoptions = new JsonSerializerOptions();
19            // добавляем конвертер кода json в объект типа Person
20            jsonoptions.Converters.Add(new PersonConverter());
21            // десериализуем данные с помощью конвертера PersonConverter
22            var person = await request.ReadFromJsonAsync<Person>(jsonoptions);
23            if (person != null)
24                responseText = $"Name: {person.Name} Age: {person.Age}";
25        }
26        await response.WriteAsJsonAsync(new {text = responseText});
27    }
28    else
29    {
30        response.ContentType = "text/html; charset=utf-8";
31        await response.sendFileAsync("html/index.html");
32    }
33 });
34
35 app.Run();
36
```

```

37 public record Person(string Name, int Age);
38 public class PersonConverter : JsonConverter<Person>
39 {
40     public override Person Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
41     {
42         var personName = "Undefined";
43         var personAge = 0;
44         while (reader.Read())
45         {
46             if (reader.TokenType == JsonTokenType.PropertyName)
47             {
48                 var propertyName = reader.GetString();
49                 reader.Read();
50                 switch (propertyName?.ToLower())
51                 {
52                     // если свойство age и оно содержит число
53                     case "age" when reader.TokenType == JsonTokenType.Number:
54                         personAge = reader.GetInt32(); // считываем число из json
55                         break;
56                     // если свойство age и оно содержит строку
57                     case "age" when reader.TokenType == JsonTokenType.String:
58                         string? stringValue = reader.GetString();
59                         // пытаемся конвертировать строку в число
60                         if (int.TryParse(stringValue, out int value))
61                         {
62                             personAge = value;
63                         }
64                         break;
65                     case "name": // если свойство Name/name
66                         string? name = reader.GetString();
67                         if (name != null)
68                             personName = name;
69                         break;
70                 }
71             }
72         }
73         return new Person(personName, personAge);
74     }
75     // сериализуем объект Person в json
76     public override void Write(Utf8JsonWriter writer, Person person, JsonSerializerOptions options)
77     {
78         writer.WriteStartObject();
79         writer.WriteString("name", person.Name);
80         writer.WriteNumber("age", person.Age);
81
82         writer.WriteEndObject();
83     }
84 }

```

Поскольку настройка параметров сериализации/десериализации – это отдельная большая тема, то пройдемся вкратце по коду, который вовлекается в процесс конвертации и прежде всего по конвертеру **Person** в **JSON**.

## Определение конвертера для сериализации/десериализации объекта в json

Класс конвертера для сериализации/десериализации объекта определенного типа в JSON должен наследоваться от класса **JsonConverter<T>**. Абстрактный класс **JsonConverter** типизируется типом, для объекта которого надо выполнить сериализацию/десериализацию. В коде выше такой реализацией является класс **PersonConverter**.

При наследовании класса **JsonConverter** необходимо реализовать его абстрактные методы **Read()** (выполняет десериализацию из **JSON** в **Person**) и **Write()** (выполняет сериализацию из **Person** в **JSON**).

Метод **Write**, который записывает данные **Person** в формат **JSON**, выглядит относительно просто:

```
1 public override void Write(Utf8JsonWriter writer, Person person, JsonSerializerOptions options)
2 {
3     writer.WriteStartObject();
4     writer.WriteString("name", person.Name);
5     writer.WriteNumber("age", person.Age);
6     writer.WriteEndObject();
7 }
```

Он принимает три параметра:

**Utf8JsonWriter** – объект, который записывает данные в json.

**Person** – объект, который надо сериализовать.

**JsonSerializerOptions** – дополнительные параметры сериализации.

Сначала с помощью объекта **Utf8JsonWriter** открываем запись объекта в формате json:

```
1 writer.WriteStartObject();
```

Последовательно записываем данные объекта **Person**:

```
1 writer.WriteString("name", person.Name);
2 writer.WriteNumber("age", person.Age);
```

И завершаем запись объекта:

```
1 writer.WriteEndObject();
```

Чтение или десериализация выглядит несколько сложнее. Метод **Read()** также принимает три параметра:

**Utf8JsonReader** – объект, который читает данные из json.

**Type** – тип, в который надо выполнить конвертацию.

**JsonSerializerOptions** – дополнительные параметры сериализации.

Результатом метода **Read()** должен быть десериализованный объект (в данном случае объект типа **Person**).

В начале определяем данные объекта **Person** по умолчанию, которые будут применяться, если в процессе десериализации произойдут проблемы:

```
1 var personName = "Undefined";
2 var personAge = 0;
```

Далее в цикле считываем каждый токен в строке json с помощью метода **Read()** объекта **Utf8JsonReader**:

```
1 while (reader.Read())
```

Затем, если считанный токен представляет название свойства, то считываем его и считываем следующий токен:

```
1 if (reader.TokenType == JsonTokenType.PropertyName)
2 {
3     var propertyName = reader.GetString();
4     reader.Read();
5 }
```

После этого мы можем узнать, как называется свойство и какое значение оно имеет. Для этого применяем конструкцию **switch**:

```
1 switch (propertyName?.ToLower())
2 {
```

Поскольку регистр символов название свойства может отличаться (например, **"Age"**, **"age"** или **"AGE"**), то, чтобы упростить сравнение, приводим название свойства к нижнему регистру.

Например, мы ожидаем, что json будет содержать свойство с именем **"age"**, которое будет хранить некоторое число. Для его получения применяем следующий блок **case**:

```
1 case "age" when reader.TokenType == JsonTokenType.Number:
2     personAge = reader.GetInt32();
3     break;
```

То есть если свойство называется **"age"** и представляет число (**JsonTokenType.Number**), то вызываем метод **reader.GetInt32()**. Но свойство **"age"** также может содержать строку, например, **"23"**. Такая строка может конвертироваться в число. И для подобного случая добавляем дополнительный блок **case**:

```
1 case "age" when reader.TokenType == JsonTokenType.String:
2     string? stringValue = reader.GetString();
3     if (int.TryParse(stringValue, out int value))
4     {
5         personAge = value;
6     }
7     break;
```

Подобным образом считываем из json значение для свойства **Name**:

```
1 case "name":
2     string? name = reader.GetString();
3     if(name!=null)
4         personName = name;
```

В конце полученными данными инициализируем объект **Person** и возвращаем его из метода:

```
1 return new Person(personName, personAge);
```

Таким образом, мы можем проверить, какие свойства имеет объект json, какие значения они несут и принять решения, передавать эти значения в объект **Person**. И в данном случае, даже если в присланном json не будет нужных свойств, или свойство **age** будет содержать строку, которая не конвертируется в число, объект **Person** все равно будет создан.

Чтобы использовать конвертер json, его надо добавить в коллекцию конвертеров:

```
1 var jsonoptions = new JsonSerializerOptions();
2 jsonoptions.Converters.Add(new PersonConverter());
3 var person = await request.ReadFromJsonAsync<Person>(jsonoptions);
```