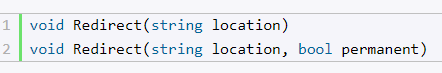
**ASP-NET CORE**

**Переадресация**

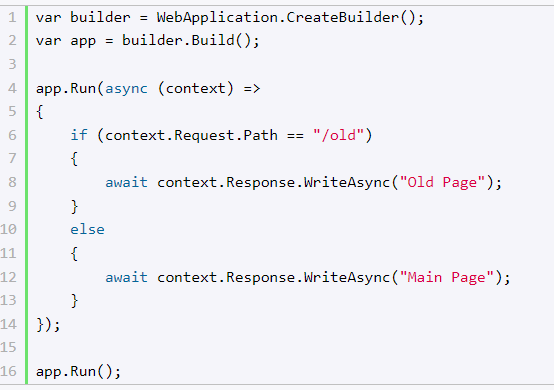
Для выполнения переадресации у объекта HttpResponse определен метод Redirect():



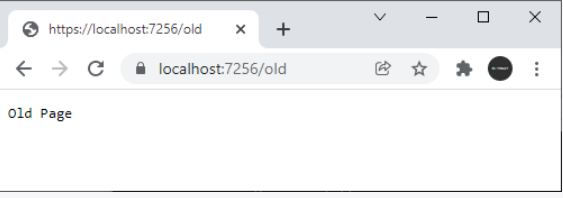
Первая версия выполняет временную переадресацию. В качестве параметра получает адрес для редиректа, а клиенту посылается статусный код 302.

Вторая версия метода также в качестве второго параметра получает булевое 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.

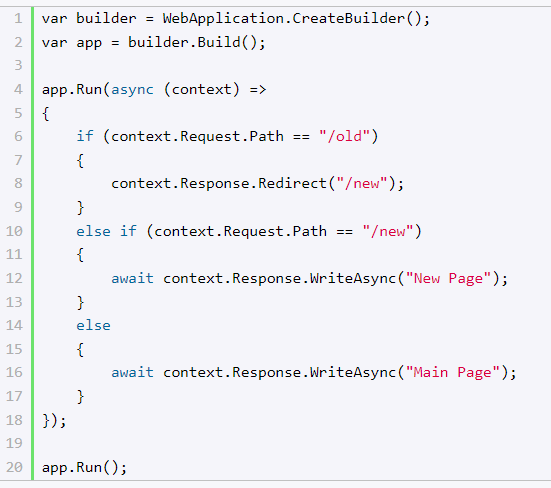
Допустим, у нас было следующее приложение (пример 1)



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

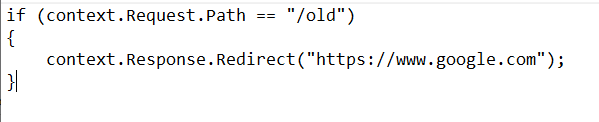


Но затем мы решили сделать переадресацию с адреса "/old" на "/new". Используем для этого первую версию метода Redirect: (пример 2)



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

В данном случае применяется редирект на локальный адрес в рамках приложение. Но также можно использовать редирект на внешние ресурсы: (пример 3)

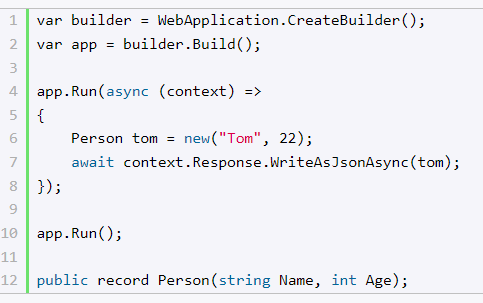


**Отправка и получение json**

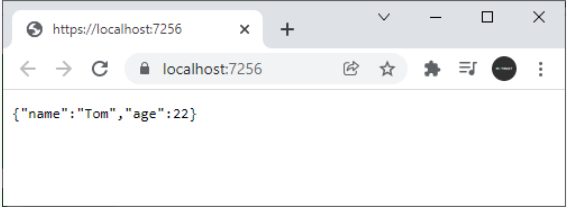
JSON является распространенным форматом для передачи данных. Рассмотрим, как мы можем посылать и получить данные json.

**Отправка JSON. Метод WriteAsJsonAsync**

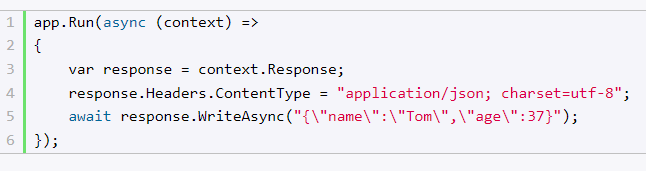
Для отправки json можно воспользоваться методом WriteAsJson()/WriteAsJsonAsync() объекта HttpResponse. Этот метод позволяет сериализовать переданные в него объекты в формат JSON и автоматически для заголовка "content-type" устанавливает значение "application/json; charset=utf-8": (пример 4)



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



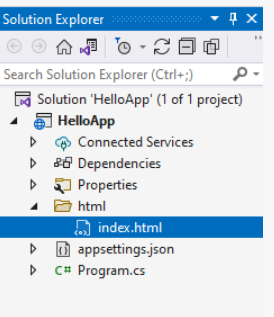
Хотя можно было бы воспользоваться и стандартным методом WriteAsync(): (пример 5)



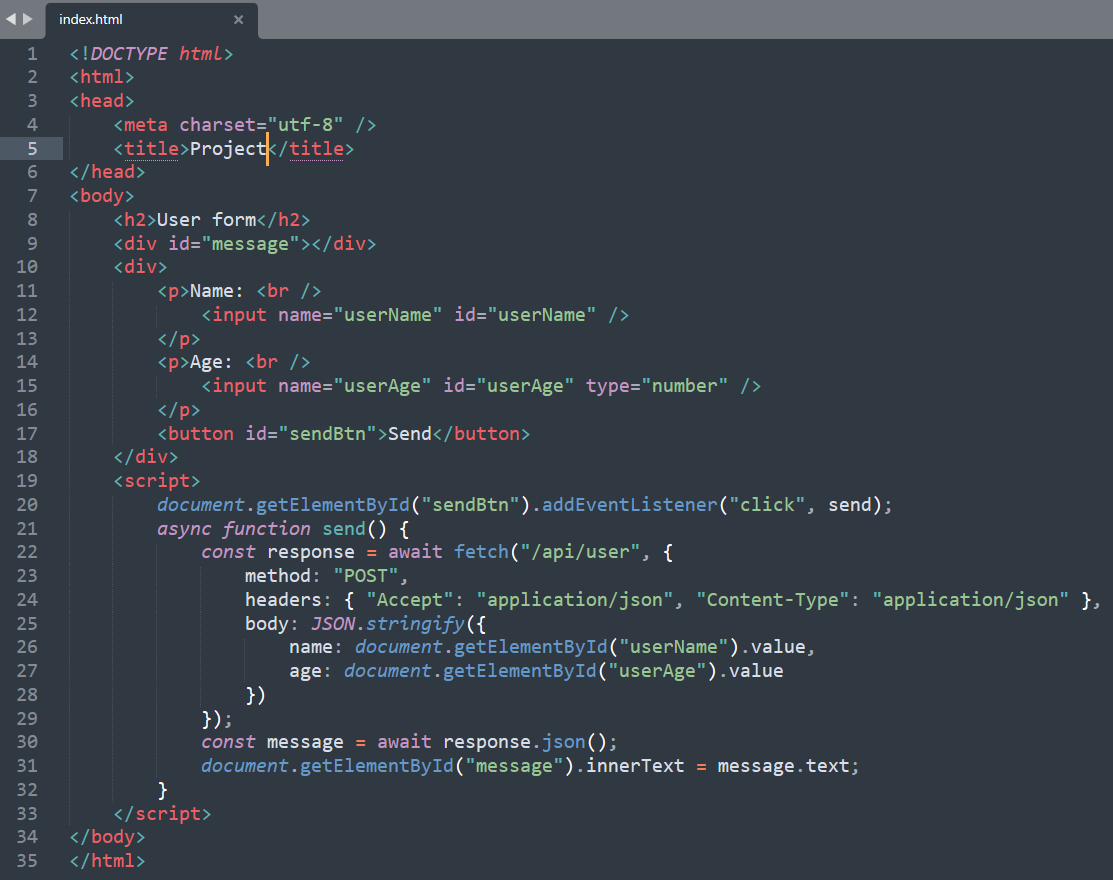
**Получение JSON. Метод ReadFromJsonAsync**

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. (текстовый файл index.html)

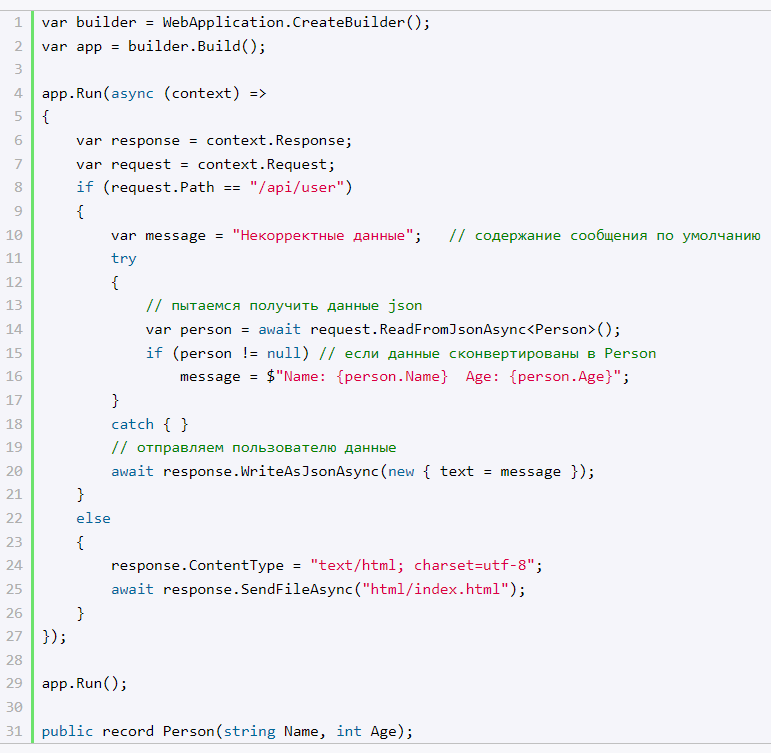


В файле index.html определим следующий код:



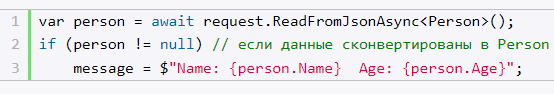
Здесь по нажатию на кнопку с помощью функции fetch() по адресу "/api/user" будет отправляться объект со свойствами name и age, значения для которых берутся из полей формы. В ответ от сервера веб-страница также получает объект в формате json, в котором имеется свойство text - свойство, которое хранит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

Теперь в файле Program.cs определим код для получения данных, отправляемых веб-страницей: (пример 6):



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.

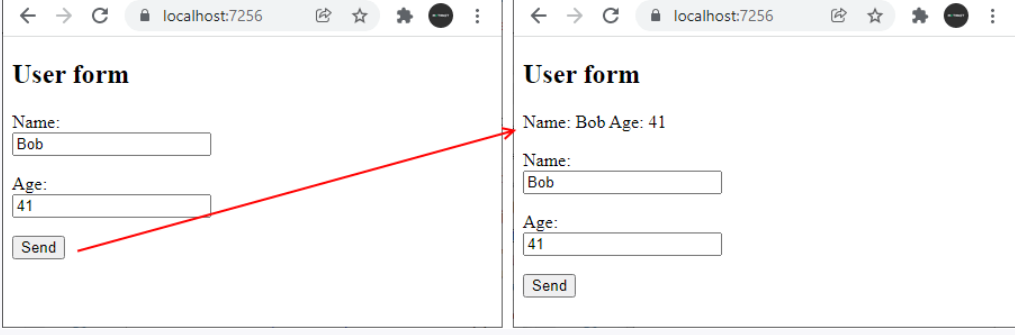
Метода ReadFromJsonAsync() десериализует полученные данные в объект определенного типа - в данном случае типа Person:



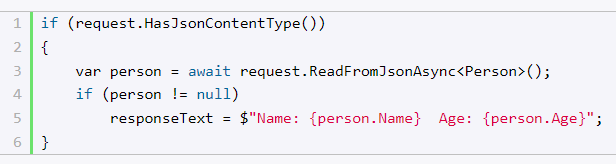
Таким образом, здесь результат вызова этого метода - значение переменной person будет представлять объект Person.

Ноб 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 посылаем анонимный объект, который также сериализуется в json с некоторым сообщением, которое хранится в свойстве text. При получении этого сообщения оно выводится на веб-страницу.



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



**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:



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:



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 - обрабатывть 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().



Так, изменим код файла Program.cs: (пример 7).

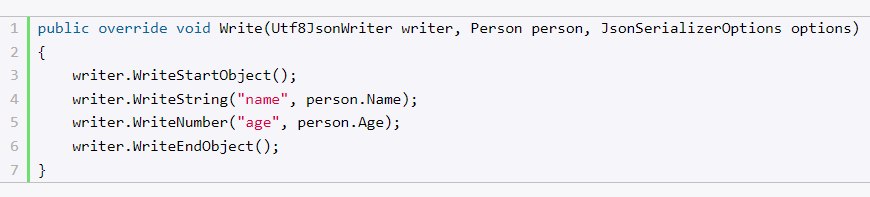
Поскольку настройка параметров сериализации/десериализации - это отдельная большая тема, то пройдемся вкратце по коду, который вовлекается в процесс конвертации и прежде всего по конвертеру Person в json.

**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:



Он принимает три параметра:

Utf8JsonWriter - объект, который записывает данные в json

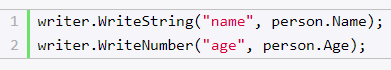
Person - объект, который надо сериализовать

JsonSerializerOptions - дополнительные параметры сериализации

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



Последовательно записываем данные объекта Person:



И завершаем запись объекта:



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

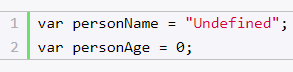
Utf8JsonReader - объект, который читает данные из json

Type - тип, в который надо выполнить конвертацию

JsonSerializerOptions - дополнительные параметры сериализации

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

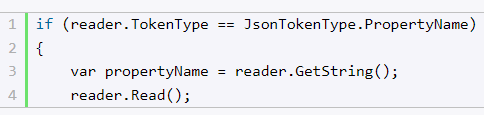
В начале определяем данные объекта Person по умолчанию, которые будут применяться, если в процессе десериализации произойдут проблемы:



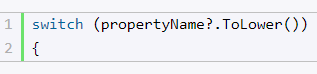
Далее в цикле считываем каждый токен в строке json с помощью метода Read() объекта Utf8JsonReader:



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

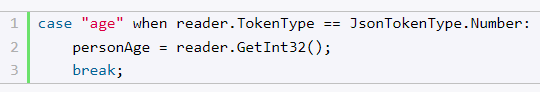


После этого мы можем узнать, как называется свойство и какое значение оно имеет. Для этого применяем конструкцию switch:



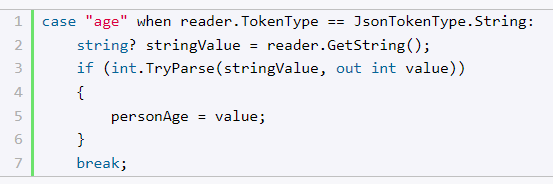
Поскольку регистр символов название свойства может отличаться (например, "Age", "age" или "AGE"), то, чтобы упростить сравнение, приводим название свойства к нижнему регистру.

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

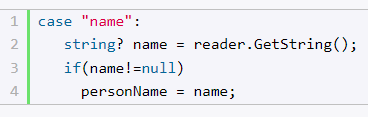


То есть если свойство называется "age" и представляет число (JsonTokenType.Number), то вызываем метод reader.GetInt32()

Но свойство "age" также может содержать строку, например, "23". Такая строка может конвертироваться в число. И для подобного случая добавляем дополнительный блок case:



Подобным образом считываем из json значение для свойства Name:



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



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:

