

Документация MicroServer C# API

1. Описание

«MicroServerAPI.cs» — это файл с пространством имён клиента «Microservices.MicroServer» для языка C#. Данный клиент раскрывает потенциал основной редакции сервера почти полностью (исключение — поддержка функций работы с внешним хранилищем).

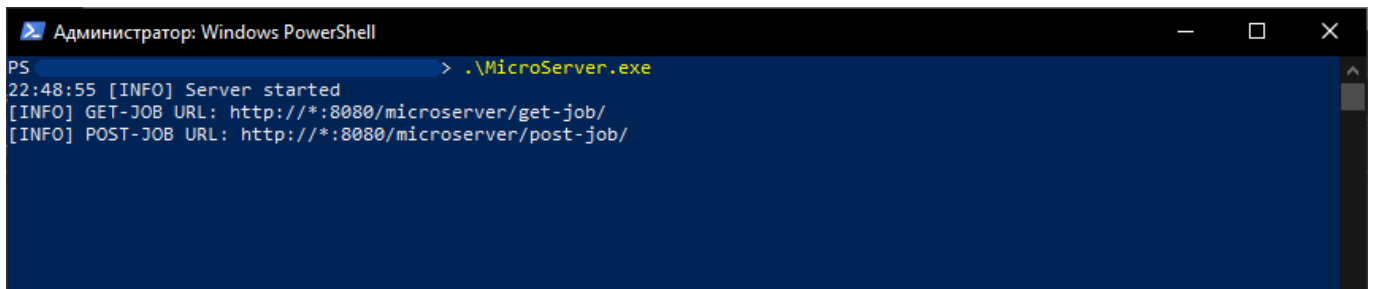
2. Состав модуля

Пространство имён «MicroServerAPI» состоит из:

- класса MicroService
- структуры ResponseAddress

Класс «MicroService» является основным и предназначен, собственно, для превращения программы в сервис для MicroServer.

Для создания объекта данного класса, необходимо передать в конструктор как минимум два аргумента: адрес GET-JOB и адрес POST-JOB, по которым будет обращаться класс клиента. Оба адреса можно узнать из вывода отладочной редакции сервера сразу после старта:



```
Администратор: Windows PowerShell
PS > .\MicroServer.exe
22:48:55 [INFO] Server started
[INFO] GET-JOB URL: http://*:8080/microserver/get-job/
[INFO] POST-JOB URL: http://*:8080/microserver/post-job/
```

Здесь строки «http://*:8080/microserver/get-job/» и «http://*:8080/microserver/post-job/» означают адреса GET-JOB и POST-JOB соответственно. Однако, это не законченные адреса, а шаблоны адресов. Чтобы получился конечный адрес:

1. вместо звёздочки нужно подставить IP-адрес или имя домена; если сервер работает на том же компьютере, что и данный клиент, звёздочку нужно заменить на «localhost». *Получится следующее:* «http://localhost:8080/microserver/get-job/» и «http://localhost:8080/microserver/post-job/».

2. Финальный штрих — убрать слеш с конца строки. *Финальный результат выглядит так:* «http://localhost:8080/microserver/get-job» и «http://localhost:8080/microserver/post-job».

Может получиться так, что звёздочки в адресе не будет, тогда следует пропустить 1 шаг. Могут быть ситуации, когда звёздочка находится в разных местах, тогда следует создать запрос, подходящий под шаблон (шаг 1 усложняется).

От сборки к сборке сервер может ожидать подключение на разных портах и с разными строками запросов, так что для ввода правильных адресов рекомендуется сначала запустить отладочную версию сервера, которая покажет порты и строки запросов (как на скриншоте выше).

Помимо адресов, конструктор также принимает необязательный аргумент «persistentConnections», который отвечает за активацию «режима настойчивых соединений». В этом режиме клиент не выдаёт ошибок соединения с сервером, а настойчиво пытается создать соединение. По умолчанию этот режим выключен, чтобы при указании неправильных адресов клиент выдал ошибку. **Рекомендуется включать** этот режим в готовых сервисах, так как он позволит не заботиться о порядке запуска этого сервиса и сервера (т.е. о том, что сервис должен быть запущен только после сервера).

Пример создания объекта с указанием аргументов конструктора:

```
var serv = new MicroServerAPI.MicroService("http://localhost:8080/microserver/get-job",
                                           "http://localhost:8080/microserver/post-job",
                                           True);
```

В случае объекта, получаемого в переменной «serv», аргументу «persistentConnections» передаётся значение «True».

3. Методы класса MicroService

В классе MicroService реализованы следующие методы:

- GetJob
- PostJob
- PostIntermediateResult
- PostFinalResult
- ProcessAsFunction

Методы «GetJob», «PostJob» и «ProcessAsFunction» имеют ряд перегрузок под большинство ситуаций.

3.1. GetJob – получение задачи

Данный метод может быть использован для получения сервисом задачи с сервера. Вызов может выглядеть примерно так:

```
var data = serv.GetJob<T>("ServiceTest.CSapi.01", out ResponseAddress addr);
```

Метод возвращает 2 значения, одно из которых – данные (помещается в переменную «data»), второе (через аргумент) – адрес возврата пакета данных (помещается в переменную «addr»). Переменная «addr» позже понадобится при отправке результата обработки, так что **её важно сохранить и не изменять**.

Параметр шаблона (T) – это тип объекта «data». В примере написано «T», но в реальности будет название структуры или класса.

Метод принимает как входной всего один аргумент – первый – имя задачи (имя типа данных), который сервис принимает на обработку. В данном случае имя типа данных – «ServiceTest.CSapi.01». На самом деле, **имя типа – это просто строка, которая была придумана** создателем другого сервиса.

Этот метод является блокирующим, так что вызвавший этот метод поток будет находиться в ожидании, пока пакет данных нужного типа не придёт.

Важно отметить, что **класс MicroService является потоко-безопасным**.

В результате выполнения функции в переменную «data» будет помещён объект указанного в параметре шаблона типа. Если переданные с сервера данные не удастся привести к указанному типу, будет выброшено исключение.

Данный метод имеет несколько перегрузок, которые будут рассмотрены позже.

3.2. PostFinalResult

Вызов данного метода может иметь следующий вид:

```
serv.PostFinalResult<T>("ServiceTest.CSapi.01.Result", addr, data);
```

Этот метод ничего не возвращает, но принимает последним аргументом объект типа «T» – результат обработки поступивших данных, а вторым – адрес возврата пакета, который в случае этого примера хранится в переменной «addr».

Первый аргумент – тип результата. В этом аргументе Вы должны дать имя типа содержимому «data». Настоятельно **рекомендуется создавать имена типов, частью которых является уникальное имя сервиса**. Нужно это во избежание ненужных совпадений имён типов у пакетов, данные в которых имеют разную структуру (например, в одном хранится строка, а в другом – массив чисел). Такие совпадения с большой вероятностью станут причиной неправильной работы конечной микро-сервисной системы.

Первый аргумент **не должен (!)** совпадать с аргументом первым аргументом метода GetJob из этого же сервиса, иначе этот сервис делегирует обработку самому себе, войдя тем самым в бесконечный цикл обработки одного и того же пакета (*со второй итерации начиная повреждать данные пакета*).

Данный метод реализован на основе метода «PostJob».

3.3. PostIntermediateResult

Данный метод имеет аналогичный PostFinalResult и его вызов выглядит так:

```
serv.PostIntermediateResult<T>("ServiceTest.CSapi.01.Result", addr, data);
```

Функционал у этого метода тоже очень похож, так этот метод предназначен также как и предыдущий для отправки результатов обработки поступивших данных, по завершению обработки (никак не во время обработки), но есть важное отличие.

Метод PostFinalResult отправляет пакет с результатом обработки данных непосредственно тому сервису, который запросил эту обработку. Можно сказать, что этот метод возвращает пакет по ранее полученному адресу.

В противовес, метод PostIntermediateResult отправляет пакет не по ранее полученному адресу, а просто размещает его на сервере как очередную задачу уже для другого сервиса. Таким образом, текущий сервис может «делегировать» задачу по обработке **этого** пакета другому сервису и больше не возвращаться к этому пакету. Теперь этот сервис может заняться другим пакетом, «переложив ответственность» за старый пакет на другой сервис, который также может делегировать часть задачи по обработке другому сервису и так далее, но, в итоге, последний в цепочке обработки сервис должен вызвать метод PostFinalResult, вернув тем самым результат в тот сервис, который инициировал весь этот «конвейер» по обработке пакета (то есть в тот сервис, который послал полностью необработанный пакет первому в цепочке сервису).

Данный метод реализован на основе метода «PostJob».

4. (3.3+½) Шаблон типичного сервиса для обработки данных

```
using MicroServerAPI;

namespace SomeMicroservice_1 {
    class Program {
        static void Main(string[] args) {
            MicroService serv = new("http://localhost:8080/5-semester/compiler/get-job",
                                    "http://localhost:8080/5-semester/compiler/post-job",
                                    true);

            while (true) {
                var data_input = serv.GetJob<string[]>(
                    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                    out ResponseAddress addr);

                //Begin of data processing area

                string result_string = "";

                foreach (var item in data_input) {
                    result_string += item;
                }

                //End of data processing area
                serv.PostFinalResult<string>(
                    "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result",
                    addr,
                    result_string);
            }
        }
    }
}
```

Выше Вы могли наблюдать пример типичного микро-сервиса. В начале создаётся объект для взаимодействия с сервером («serv»), далее следует бесконечный цикл, в котором идёт обработка пакетов с именем типа «ArrayConcatenator.A42.ConcatenateStringArrayToString».

Обратите внимание, «A42» – это условно случайная комбинация символов (придуманная разработчиком сервиса), которая нужна во избежание ненужных совпадений имён типов (об этом говорилось ранее). «ArrayConcatenator» – это имя самого микро-сервиса, «ConcatenateStringArrayToString» – имя задачи для обработки. Обратите внимание, имя для этой задачи могло бы быть, например, и таким: «62f54e2bd30ad44a6be7d22a7238003892357ed8», от чего функционально ничего бы не изменилось, но первый вариант гораздо более предпочтителен, так как гораздо информативнее для человека.

После получения данных и адреса возврата, выполнение скрипта заходит в область обработки данных, где объявляется переменная «result_string» (которая имеет тип «string»). Далее предполагается, что «data» – это массив строк («предположение» сделано на основе того, что пакет для обработки выбирался по указанному имени типа, а не случайный). Далее этот массив циклом объединяется в одну строку, после чего выполнение скрипта выходит за рамки области обработки данных.

Здесь используется переменная с результатом («result_string»), а также ранее полученный адрес («addr»). Обратите внимание на то, что имя типа в методе PostFinalResult **отличается** от имени типа в GetJob.

Обратите внимание, **после обработки данных и отправки результата, сервис должен забыть все данные, с которыми работал**, после чего повторить цикл получения и отправки снова.

3.4. ProcessAsFunction

Ранее были рассмотрены методы, используемые для организации работы обрабатывающего данные микро-сервиса. Однако, на данный момент неясным является способ «производства» данных для обработки. Каким образом данные откуда-то извне попадут в микро-сервисную систему?

Ответ: например, при помощи метода ProcessAsFunction.

Вызов этого метода может выглядеть так:

```
string result = serv.ProcessAsFunction<string, string[]>(
    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
    string_array);
```

Первый аргумент следует воспринимать как «имя функции, которая будет обрабатывать мои данные».

Второй аргумент содержит данные, которые нужно обработать.

Текущей сервис должен знать, в каком виде нужно представить данные (строка, массив, словарь и т.д.), чтобы желаемый сервис мог их обработать. Он также должен знать, что нужно написать в первом аргументе, чтобы данные попали в нужный сервис (и чтобы вообще хоть куда-то попали).

Метод ProcessAsFunction также следующую перегрузку (пример вызова):

```
string result = serv.ProcessAsFunction<string, string[]>(
    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
    string_array,
    "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result");
```

Последний аргумент в этой перегрузке – это имя типа для получаемого объекта. Использование этой перегрузки не обязательно, но желательно для более быстрой обработки пакета на стороне сервера. Важно отметить, что это **только при условии, что сервисы обработки данных при размещении результата указывают тип этого результата, иначе использование этой перегрузки не позволит получить результат!** Как можно понять, использование первой перегрузки (представлена в начале раздела) – универсальное (но нередко – менее эффективное) решение.

Можно ли вызывать метод ProcessAsFunction для обработки каких-то данных внутри блока кода между GetJob и PostFinalResult? Конечно же, **ProcessAsFunction можно вызывать в блоке обработки данных (при том, неограниченное количество раз)**. Только важно, чтобы первый аргумент метода ProcessAsFunction не совпадал с первым аргументом метода GetJob из этого же сервиса.

5. Пример микро-сервисной системы

Небольшой пример состоит из 2-х проектов. Код двух «Program.cs» из них приведён ниже. Важно ответить, что в каждый из проектов входит ещё по одному файлу – «MicroServerAPI.cs», которые одинаковы и подключены к проектам.

Первая часть этого примера уже была приведена раньше – в разделе «Шаблон типичного сервиса для обработки данных», где представлен полноценный пример сервиса для обработки данных.

Вторая часть примера (файл поставщика данных в систему) приведён ниже:

```

using System;
using System.Collections.Generic;
using MicroServerAPI;

namespace SomeMicroservice_2 {
    class Program {
        static void Main(string[] args) {
            MicroService serv = new("http://localhost:8080/5-semester/compiler/get-job",
                                    "http://localhost:8080/5-semester/compiler/post-job",
                                    true);

            while (true) {
                List<string> data_input = new();
                while (true) {
                    Console.Write("Enter string (leave empty to exit): ");
                    string single_string = Console.ReadLine();

                    if (single_string == "") {
                        break;
                    } else {
                        data_input.Add(single_string);
                    }
                }
                Console.Write("Entered strings:");
                foreach (var item in data_input) {
                    Console.Write(" " + item);
                }
                Console.WriteLine();

                var result = serv.ProcessAsFunction<string, string[]>(
                    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                    data_input.ToArray(),
                    "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result");

                Console.WriteLine("Result: " + result);
                Console.WriteLine();
            }
        }
    }
}

```

Первый сервис из примера занимается обработкой данных, поэтому сразу подключается к серверу. Однако, если сервер не запущен, в дело вступает «persistentConnections»: сервис ждёт запуска сервера, так что очередность запуска этому сервису не важна (очевидно, серверу очередность запуска не важна в принципе).

Второй уже не занимается обработкой, а ждёт ввода данных пользователем. Если запустить этот сервис в то время, когда сервер не запущен, этот сервис всё равно будет ожидать не сервер, а ввода данных пользователем. Если же к моменту завершения ввода данных пользователем сервер окажется по-прежнему не запущенным, сервис начнёт ожидать уже сервер (благодаря «persistentConnections»).

6. Поля класса MicroService

В классе MicroService существуют следующие свойства:

- PersistentConnections

Значение свойства «PersistentConnections» можно изменять уже после создания объекта класса «MicroService». Это позволяет создать класс с аргументом конструктора «persistentConnections» равным «False», а после выполнения нужного кода, установить свойству «PersistentConnections» значение «True», включив тем самым настойчивый режим.

7. Перегрузки GetJob и PostJob

К основообразующим (продвинутым) методам класса клиента относятся:

- GetJob
- PostJob

На основе именно вышеприведённых методов реализованы все остальные методы.

7.1. GetJob

Помимо рассмотренной ранее перегрузки данного метода, существуют следующая (пример вызова):

```
var data = serv.GetJob<T>("ServiceTest.CSapi.01",  
                           "81fe8bfe87576c3ecb22426f8e57847382917acf");
```

И эта (пример вызова):

```
string received_type = serv.GetJob<T>("81fe8bfe87576c3ecb22426f8e57847382917acf",  
                                       out data);
```

Как можно заметить, первая перегрузка похожа на ранее приведённую для GetJob: возвращает объект, первым аргументом принимает имя типа. Однако, отличие тоже есть: присутствует второй аргумент, в котором находится идентификатор пакета. Вторая перегрузка также содержит идентификатор, но уже первым аргументом, а через второй аргумент возвращает результат.

Зачем задаче нужен идентификатор?

Для начала, нужно усвоить, что **отсутствующий аргумент приобретает значение null** (ключевое слово), то есть не разницы между тем чтобы не указывать аргумент, и тем, чтобы указать его и передать значение null (ключевое слово).

Затем, нужно понять, какой именно смысл имеет значение «null»: это особое значение, которое может одновременно показывать и отсутствие, и безразличие в зависимости от действия. Если пакет отправляется на сервер, то «null» в пакете означает отсутствие значения того или иного поля. Если же пакет запрашивается с сервера, то «null» в запросе означает безразличие к значению соответствующего поля.

Вот теперь можно вернуться к вопросу «Зачем задаче нужен идентификатор?».

Идентификатор для задачи указывается в случае, когда сервису нужно получить результат обработки своего пакета. (Вам это не напоминает метод «ProcessAsFunction»?) Идентификатору нужно быть уникальным для всей микро-сервисной системы, плюс ко всему, он, в отличие от типа, одноразовый и генерируется клиентом непосредственно перед отправкой пакета. За все ранее перечисленные свойства его можно считать полноценным «именем» пакета, ну а если говорить ближе к введённым ранее обозначениям, то «адресом», а точнее – «адресом возврата». Кажется, про адрес возврата ранее было что-то сказано, верно?

Ранее адрес возврата был рассмотрен как данность: он есть, его нужно сохранить и передать. Здесь признак пакета в виде желаемого идентификатора (адреса возврата) можно задать.

Конечно же, в переменной «addr» будет храниться тот же адрес, который задали при запросе (если задали), и, конечно же, чтобы задать адрес в GetJob, этот адрес ещё нужно откуда-то узнать. Узнать адрес извне никак нельзя, а, значит, остаётся только создать

свой собственный уникальный идентификатор (адрес возврата) и создать пакет с таким идентификатором.

7.2. PostJob

Данный метод имеет 2 перегрузки. Пример вызова первой:

```
serv.PostJob<T>("ServiceTest.CSapi.01", data);
```

Первая из перегрузок похожа на PostFinalResult за исключением того, что этот метод не принимает адрес возврата одним из аргументов: здесь есть только имя типа и данные.

На следующей перегрузке нужно остановиться более подробно:

```
serv.PostJob<T>("ServiceTest.CSapi.01",  
                "81fe8bfe87576c3ecb22426f8e57847382917acf",  
                true,  
                data);
```

Из уже известных аргументов здесь имя типа, идентификатор, а также данные. Однако, здесь есть bool-аргумент (третий по счёту), который называется «visibleId».

В этой функции есть nullable-аргумент – первый (имя типа). Это означает, что в данный аргумент можно передать «null» (ключевое слово).

Для комбинации аргументов метода есть одно ограничение, о котором будет сказано позже.

Второй аргумент нужен чтобы отправить пакет с заданным этому аргументу идентификатором. Как было сказано ранее, этот идентификатор будет являться адресом возврата, так что его нужно будет указать в GetJob.

Именно на основе этой перегрузки реализованы методы «PostFinalResult» и «PostIntermediateResult».

Кажется, с этими сведениями можно написать собственную функцию «ProcessAsFunction»:

```
public O ProcessAsFunction<O, I>(string requested_function, I content)  
{  
    string uid = "MyServiceUniqueName." + DateTime.Now.ToString();  
  
    serv.PostJob<I>(requested_function, uid, false, content);  
    serv.GetJob<O>(uid, out O result);  
    return  
}
```

Как можно заметить в примере, идентификатор генерируется программой в начале функции из имени сервиса («MyServiceUniqueName.») и текущего времени (в виде строки получается комбинацией «DateTime.Now.ToString()»). Это позволяет создать в достаточной степени уникальный идентификатор, который сохраняется в переменную «uid».

Затем в примере вызывается метод «PostJob», которому отдаётся в том числе и созданный uid. По завершению выполнения PostJob, пакет находится на сервере. Теперь его могут получить другие сервисы (разумеется, **один пакет может попасть только на один сервис**).

Теперь можно ждать результата обработки. Делается это в методе `GetJob`, которому передаётся ожидаемый тип пакета, и, что самое важное, идентификатор пакета (переменная «`uid`»), по которому возможно опознать именно тот самый среди потенциальных десятков с аналогичным типом.

Вы ещё не догадались, для чего нужен аргумент «`visibleId`» у метода «`PostJob`»? Что ж, предлагаю подумать. Если что, ответ ниже.

Что ж, давайте представим, что такого аргумента нет. При помощи `PostJob` в этом случае создаётся пакет с исходными данными, типом и идентификатором, но без указания аргумента «`visibleId`» (что принципиально эквивалентно `visibleId=True`). Далее при помощи `GetJob` выполняется запрос пакета, но только по идентификатору. Есть ли на сервере в момент выполнения команды `GetJob` пакет с идентификатором `uid`? Конечно, есть: этот тот самый пакет, который был отправлен на сервер прошлой командой. Как результат именно этот пакет и будет получен. Получится, что придут те же данные, что ушли.

Аргумент «`visibleId`» со значением «`False`» решает проблему просто и эффективно: путём сокрытия идентификатора в отправляемом пакете. Сервер не будет сравнивать запрашиваемый идентификатор с идентификатором пакета, если последний скрыт. Таким образом, пакет со скрытым `id` не может быть получен по `id`, а единственный способ получения такого пакета – это запросить его с идентификатором «`null`» (т.е. безразличие к идентификатору) и совпадающим именем типа (из этого вытекает ограничение: **имя типа не может быть «`null`» когда идентификатор скрыт**).

После обработки данных пакета со скрытым `id`, его нужно вернуть сервису, создавшему задачу на обработку. Как раньше говорилось, это можно сделать при помощи метода `PostFinalResult` (частный случай `PostJob`), который отправляет результат обработки в пакете с соответствующим результату именем типа, старым идентификатором («`uid`» в нашем примере и «адрес возврата» для обрабатывающего сервиса), который обозначается видимым.

После отправки `PostFinalResult` пакета с видимым `id`, в примере выше метод `GetJob` получит пакет с результатом по `id` и возвратит результат в вызвавший код.

8. Рекомендуемая литература

Настоятельно рекомендуется ознакомление с программой отладки коммуникации микро-сервисов в `MicroServer`: «Документация `MicroServer.Debugger`». Рекомендуется более углублённая в устройство `MicroServer` литература: «Документация `Microservices.MicroServer`». Возможно, будет полезным познакомиться и с клиентами API `MicroServer` для других языков программирования.