Документация MicroServer Python API

1. Описание

«MicroServerAPI.py» – это модуль клиента «Microservices.MicroServer» для языка Python. Данный клиент раскрывает потенциал основной редакции сервера почти полностью (исключение – поддержка функций работы с внешним хранилищем).

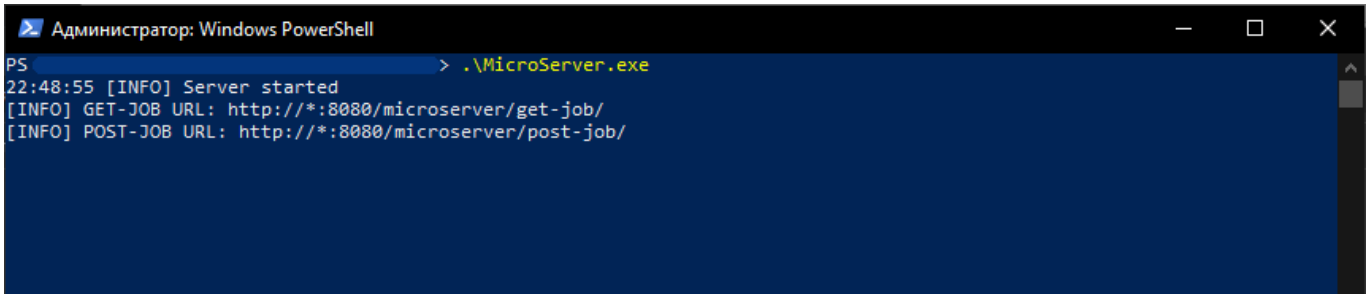
2. Состав модуля

Модуль «MicroServerAPI» состоит из следующих классов:

- MicroService
- ResponseAddress

Класс «MicroService» является основным и предназначен, собственно, для с превращения обычного скрипта 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:



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 «persistentMode», который отвечает за активацию «настойчивого режима». 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 (аргументы заданы по имени):

```
serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
    url_post='http://localhost:8080/microserver/post-job')

serv2 = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
    url_post='http://localhost:8080/microserver/post-job', persistentMode=True)
```

В случае объекта, получаемого в переменной «serv», аргументу «persistentMode» по умолчанию даётся значение «False».

3. Методы класса MicroService

В классе MicroService реализованы следующие методы:

- GetJob
- GetNextJob
- PostJob
- PostIntermediateResult
- PostFinalResult
- ProcessAsFunction

Методы «GetJob» и «PostJob» являются основообразующими, так как на их основе реализован функционал остальных методов. Данные методы будут рассмотрены в конце, так как большую часть задач удобнее решить при помощи оставшихся методов.

3.1. GetNextJob

Данный метод должен быть использован для получения сервисом задачи с сервера. Вызов этого метода выглядит так:

```
data,addr = serv.GetNextJob(job_type='PyTest.01')
```

Метод возвращает 2 значения, одно из которых – данные (помещается в переменную «data»), второе – адрес возврата пакета данных (помещается в переменную «addr»). Важно, чтобы порядок указания переменных «data» и «addr» был именно таким. Переменная «addr» сохраняет в себя объект типа «ResponseAddress» (второй из существующих в модуле классов) и позже понадобится при отправке результата обработки, так что **её важно сохранить и не изменять**.

Метод принимает всего один аргумент – «job_type» – имя задачи (имя типа данных), который сервис принимает на обработку. В данном случае имя типа данных – «PyTest.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.

Что именно находится в переменной «data»? По идее, Вы должны это знать ещё до вызова «GetNextJob», ведь Вы сами указываете аргумент «job_type» (даёте имя типу). Однако, если говорить об общем случае, в переменной «data» будет находиться комбинация типов dictionary, list, или string, или boolean, или number, или None. Если в data будет находиться dictionary или list, то состоять он может из всего, что было перечислено ранее.

Структура полученных данных будет в значительной степени напоминать JSON, так как возвращённые объект есть ни что иное, как JSON, преобразованный в набор вложенных друг в друга объектов Python.

3.2. PostFinalResult

Данный метод принимает следующие аргументы:

```
serv.PostFinalResult(content='some string content', responseAddress=addr,
   result_type='PyTest.01.Result')
```

Этот метод ничего не возвращает, но принимает «content» – результат обработки поступивших данных и «responseAddress» – адрес возврата пакета, который в случае этого примера хранится в переменной «addr». Оба из ранее приведённых аргументов обязательны к указанию.

Вы можете заметить, что в случае примера результат обработки данных не зависит от того, что поступило в сервис (через «GetNextJob»). Это принципиально неправильно в большинстве случаев.

Последний аргумент – «result_type» – тип результата, который по умолчанию равен «null». В этом аргументе Вы можете дать имя типа содержимому «content». 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.

Давать имя типу в данном методе не обязательно, но очень желательно, так как это позволит ускорить обработку пакетов на стороне сервера.

3.3. PostIntermediateResult

Данный метод имеет аналогичный PostFinalResult набор аргументов:

```
serv.PostIntermediateResult(content='some string content', responseAddress=addr,
   result_type='PyTest.01.Result')
```

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, которой 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).

Именно из-за того, что PostIntermediateResult делегирует задачу другому сервису, в нём аргумент «result_type» обязателен и этот аргумент **не должен (!)** совпадать с аргументом «job_type» метода GetNextJob 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)*.

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

```
import MicroServerAPI

serv = MicroServerAPI.MicroService('http://localhost:8080/microserver/get-job',
       'http://localhost:8080/microserver/post-job', persistentMode=True)

while True:
    data,addr = serv.GetNextJob(job_type='ArrayConcatenator.A42.ConcatenateStringArrayToString')
    #Begin of data processing area

    result_string = ""

    for item in data:
        result_string += item

    #End of data processing area
    serv.PostFinalResult(content=result_string,
        result_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result',
        responseAddress=addr)
```

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»). Имя типа здесь не обязательно, но оно указано (и его желательно указывать). Обратите внимание на то, что имя типа в методе PostFinalResult **отличается** от имени типа в GetNextJob.

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

Помимо прочего, заметьте, что порядок следования аргументов метода PostFinalResult в примере отличается от того, что был показан ранее. Это связано с тем, что к аргументам здесь идёт обращение по имени, без учёта позиции (см. именованные аргументы в python).

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 выглядит так:

```
result = serv.ProcessAsFunction(content='14+24*(1/7)',
  requested_function='A424.Calculator.CalcExpression',
  target_content_type='A424.Calculator.CalcExpression.Result')
```

Аргумент «requested_function» – по своей сути всё тот же «job_type», но так как здесь речь идёт об обработке данных другими сервисами (текущий сервис только производит данные, обработки никакой сам не ведёт), аргументы называются немного иначе. Аргумент «requested_function» следует воспринимать как «имя функции, которая будет обрабатывать мои данные».

Аргумент «content» 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 аргументе «requested_function», чтобы данные попали в нужный сервис (и чтобы вообще хоть куда-то попали).

Последний аргумент («target_content_type») не является обязательным, но желателен 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 указание аргумента «target_content_type» не позволит получить результат!** Как можно понять, не указывать аргумент «target_content_type» – универсальное (но нередко – менее эффективное) решение.

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

5. Пример микро-сервисной системы

Небольшой пример состоит из 3-х файлов, код двух из которых приведён ниже. Третий файл – это файл модуля клиента («MicroServerAPI.py»), который необходимо иметь неподалёку от файлов кода скрипта (в примерах этот файл в той же папке, что и сами скрипты).

Код сервиса обработки данных (ExampleService.ArrayConcatenator.py):

```
import MicroServerAPI

serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
                                   url_post='http://localhost:8080/microserver/post-job',
                                   persistentMode=True)

while True:
    data,addr=serv.GetNextJob(job_type='ArrayConcatenator.A42.ConcatenateStringArrayToString')
    #Begin of data processing area

    result_string = ""

    for item in data:
        result_string += item

    #End of data processing area
    serv.PostFinalResult(content=result_string,
                result_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result',
                responseAddress=addr)
```

Код сервиса для получения данных извне (ExampleService.TestProgram.py):

```
import MicroServerAPI

serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
                                   url_post='http://localhost:8080/microserver/post-job',
                                   persistentMode=True)

while True:
    st = []
    inp = input('Enter string: ')
    while inp != '':
        st.append(inp)
        inp = input('Enter next string (leave blank to exit): ')

    print("Input array:")
    print(st)
    print()
    print("Requesting concatenation...")

    result = serv.ProcessAsFunction(content=st,
           requested_function='ArrayConcatenator.A42.ConcatenateStringArrayToString',
           target_content_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result')

    print("Result: ")
    print(result)
    print()
```

Первый из приведённых выше сервисов занимается обработкой данных, поэтому сразу подключается к серверу. Однако, если сервер не запущен, в дело вступает «persistentMode»: сервис ждёт запуска сервера, так что очерёдность запуска этому сервису не важна (очевидно, серверу очерёдность 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 «persistentMode»).

6. Поля модуля MicroService

В классе MicroService существуют следующие свойства:

- PersistentMode

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

7. Продвинутые методы

К основообразующим (продвинутым) методам класса клиента относятся:

- GetJob
- PostJob

На основе именно вышеприведённых методов были реализованы все рассмотренные ранее методы.

7.1. GetJob

Вызов данного метода выглядит так:

```
data,addr = serv.GetJob(job_type='PyTest.01',
                        job_id='81fe8bfe87576c3ecb22426f8e57847382917acf')
```

Как можно заметить, этот метод похож на «GetNextJob»: возвращает такое же количество переменных, в том же порядке, присутствует аргумент «job_type». Однако, отличие тоже есть: один из аргументов GetJob не обязателен. Можно указать оба аргумента, либо можно указать только «job_type», либо указать только «job_id». Не указывать ни одного аргумента нельзя.

Как несложно догадаться, «GetNextJob» есть ни что иное как GetJob без указания «job_id».

Зачем задаче нужен идентификатор?

Для начала, нужно усвоить, что **неуказанный аргумент приобретает значение «null»**, то есть не разницы между тем чтобы не указывать аргумент, и тем, чтобы указать его и передать значение «null».

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 значение 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

Вызов данного метода выглядит так:

```
serv.PostJob(content='data to process',
             job_type='PyTest.01',
             job_id='81fe8bfe87576c3ecb22426f8e57847382917acf',
             visibleId=False)
```

Из уже известных аргументов здесь «content» и «job_type» (в методах PostFinalResult и PostIntermediateResult этот аргумент назывался «result_type»).

Из обязательных аргументов здесь только «content». «job_type» и «job_id» могут быть указаны оба или по отдельности, но, как и в GetJob, не могут отсутствовать вместе. *Есть ещё одно ограничение, о котором будет сказано позже.*

Аргумент «job_id» 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.

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

```
def ProcessAsFunction(content, requested_function, target_content_type='null'):
    uid = 'MyModuleUniqueName.' + str(datetime.now())

    serv.PostJob(content=content,
                 job_type=requested_function,
                 job_id=uid,
                 visibleId=False)

    data,addr = serv.GetJob(job_type=target_content_type, job_id=uid)

    return data
```

Как можно заметить в примере, идентификатор генерируется программой в начале функции из имени модуля («MyModuleUniqueName.») и текущего времени (в виде строки получается комбинацией «str(datetime.now())»). 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.

Обратите внимание, созданная функция возвращает только переменную «data». Переменная «addr» фактически не нужна здесь, так как в ней содержится то, что уже известно: «uid».

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. Рассмотрим ситуацию, когда «target_content_type» равняется «null». При помощи PostJob в этом случае создаётся пакет с исходными данными, типом и идентификатором, но без указания аргумента «visibleId» (что эквивалентно visibleId=True). Далее при помощи GetJob выполняется запрос пакета, но только по идентификатору (так как «target_content_type» не был задан). 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.