

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).

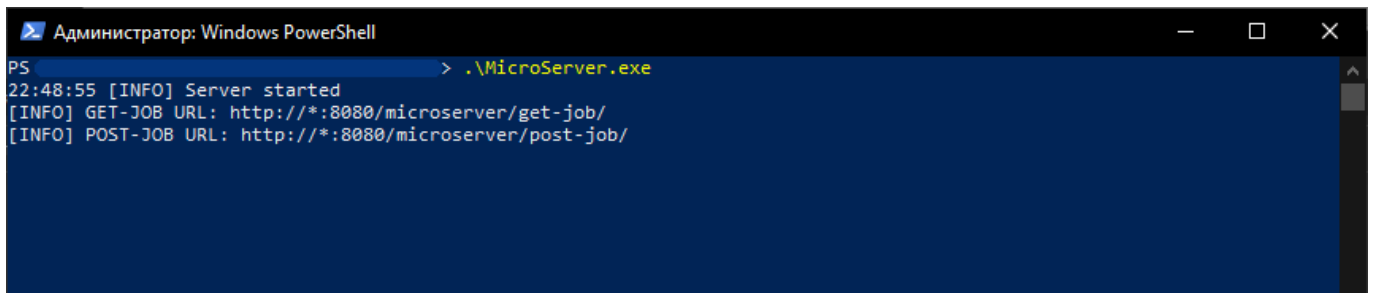
2. Состав модуля

Модуль «MicroServerAPI» состоит из следующих классов:

- MicroService
- ResponseAddress

Класс «MicroService» является основным и предназначен, собственно, для 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:



```
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 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». 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 возвращённый 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,  
job_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 – «job_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,  
job_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 «`job_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,
                        job_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`.

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.

Текущий 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.