Документация Microservices.MicroServer

1. Адреса

Для доступа к серверу используется 2 адреса, изначально заданных в коде и требующих перекомпиляции для изменения.

Первый адрес – адрес для GET-запросов, второй – для POST-запросов. Такое разделение не обязательно, однако, было сделано для ещё большей разницы между запросами размещения и запроса данных. При необходимости эти адреса можно совместить, упростив тем самым код проверки корректности запроса.

Для примера будут использованы следующие адреса:

- GET: http://localhost:8080/microserver/get-job
- POST: http://localhost:8080/microserver/post-job

2. Редакции сервера

Сервер существует в двух редакциях: основной и отладочной. Основная версия предназначена для работы в конечном продукте, отладочная предназначена для использования в процессе разработки продукта, так как имеет расширенный набор команд, помогающий с отладкой взаимодействия микро-сервисов. Из-за отладочной функциональности эта редакция несколько менее производительна в сравнении с основной редакцией.

3. Задача сервера

Предполагается, что некий сервис будет запрашивать у сервера некоторые данные (для обработки), после чего отсылать результат на сервер. Каждый отдельный сервис не должен беспокоиться о глобальном маршруте своих данных, так как передачу в нужный сервис сервер берёт на себя. Передача в тот или иной сервис осуществляется в основном в зависимости от типа интересующих сервис данных (тип сообщает сервис). Благодаря такой схеме возможно увеличение производительности конечного продукта за счёт увеличения количества подключенных к серверу экземпляров медленного сервиса.

Сервисы отсылают серверу данные, а также запрашивают у него данные по признаку (типу или идентификатору) – таким образом и строится взаимодействие с сервером.

4. Протокол связи с MicroServer

Для обмена информацией с сервером используется исключительно JSON (JavaScript Object Notation). Так как JSON – текстовый формат, сообщения данного сервера достаточно удобны для восприятия человеком.

Все данные на сервер передаются через POST-запрос. Любой POST-запрос на сервер должен иметь в основе следующую JSON-структуру (она называется «basic content»):

```
{
    "id":<string>,
    "visibleId":<bool>,
    "type":<string>,
    "content":<object>
}
```

- Поле «content» содержит непосредственно те данные, которые нужно передать. Здесь может находиться любой JSON-тип (array, null, true/false, object, number).
- Поле «type» содержит строку, обозначающую тип. Эта строка сообщает серверу и сервису как работать с данными в поле «content» (есть ситуация, когда сервер всё-таки обрабатывает поле «content», а не слепо передаёт его сервису, но об этом случае позже). Стоит сказать, что сервер работает с UTF-8, так что набор допустимых символов весьма велик.
- Поле «id» необходимо для указания принадлежности той или иной части данных (basic content) какому-либо запросу. Обратите внимание, именно ЗАПРОСУ, а не сервису. Потенциально может существовать неограниченное количество basic content с одинаковым id, но это неправильно. Id (identifer) нужен для того, чтобы сервис после размещения некоторой задачи (отправки данных на сервер) мог позже запросить (по «id») результат обработки именно той задачи, которую он разместил, а не другой с таким же типом («type»). По сути, «id» - это адрес возврата.
- Поле «visibleId» необходимо для того, чтобы запретить получение публикуемых данных по id. Таким образом, если у каких-то данных поле «visibleId» имеет значение false, сервис может запросить эти данные только по типу («type»). Такое поведение нужно для реализации «функций». Всё дело в том, что сервер не сохраняет данных о том, откуда какие данные пришли: он просто хранит поступившую информацию. Настолько же свободно информацию у сервера можно и запросить. Пусть некоторый сервис N хочет запросить обработку некоторого типа данных, после чего продолжить работу с уже обработанными данными. Для этого сервис генерирует id для того, чтобы найти именно свой результат (а не чужой). После этого он отправляет данные на сервер и ждёт появления на сервере нужного id. Но стоп! Ведь такой id уже есть и находится в basic content, только что отправленном с этого сервиса (N) на сервер! Как итог, сервис в качестве результата получает то же, что отправлял. Бессмысленная трата времени. Именно в такой ситуации помогает указание «видимости» id. Если сервис N укажет, что id пакета не должен быть виден, то никто (и он сам) не сможет получить конкретно этот пакет данных по id. Зато обрабатывающий сервис R легко этот пакет получит по запрашиваемому типу. После обработки он сформирует новый пакет данных с тем же id, но уже видимым; отправит этот пакет на сервер. И уже после этого сервис N получит свои обработанные данные. К слову, сервис R мог бы «делегировать» обработку полученных пакетов далее – другому сервису W (сделать своего рода конвейер), указав в basic content нужный тип, оставив старый id невидимым. Тогда бы уже сервис W завершил обработку и отправил на сервер basic content с видимым id.

Очень важным является выбор уникального имени типа («type»). Так как типы «глобальны» для всей системы (могут быть использованы любыми сервисами), очень важно избежать коллизий (когда одно и то же имя используется разными микро-сервисами для разных типов данных). Поэтому к имени типа желательно добавлять имя сервиса, которых создаёт данные этого типа, либо что-то ещё. На строку типа не налагается никаких ограничений: она может быть даже пустой, однако, для стабильной работы системы важно обеспечить отсутствие коллизий.

Также важный момент: JSON-значение null для сервера эквивалентно строке «null». Это **особое** значение, которое может использоваться как при загрузке данных, так и при их запросе, однако, при загрузке такое значение является обычным значением (для type или id), но с ограничением: у пакета не может быть одновременно «type» равен «null» и «visibleId» равен «false». Связано это с тем, что при запросе данных строка «null» имеет особую роль – она означает, что сервису не важно значение поля пакета, если при GET-запросе в соответствующее поле дано значение «null». Например, если при запросе указан тип, но id равен «null», то в ответ придёт любой пакет с совпадающим типом (id проверяться не будет) – *это наиболее оптимизированный в MicroServer тип запроса.*

Если и тип и id равны «null» одновременно, то при запросе будет считаться, что «null» в поле id является полноценным идентификатором, и сервер будет искать пакет, у которого поле id будет также «null». Нужно это чтобы ответ от сервера был хоть немного осмысленным, а не первым попавшимся в списке типов пакетом.

5. Команды сервера и «внешнее» хранилище пакетов

Иногда может возникать ситуация, при которой каких-то пакетов может оказаться слишком много на сервере. Такое может произойти, например, если отправляемые пакеты не используются далее никаким сервисом, а просто копятся. Сервер не выполняет удаление «излишков», но иногда это всё-таки необходимо, поэтому существует набор из 3 команд, позволяющий некому сервису скачивать с сервера излишки, становясь внешним хранилищем пакетов. Такой сервис периодически выполняет запросы списка переполненных типов, а также запросы излишков.
Для работы внешнего хранилища реализованы следующие команды:

- [GET]: MicroServer.25367be645.ExternalStatus
- [GET]: MicroServer.25367be645.FetchOverflow
- [POST]: MicroServer.25367be645.CompensateUnderflow

Каким образом происходит подача команды на сервер? Это происходит так же, как и запрос пакета данных – через GET- или POST-запрос, но со строго определённым (специальным) значением поля «type», и обрабатывается сервером непосредственно (без передачи какому-либо сервису). Значение поля «id» будет проигнорировано.
Важно понимать, что сервисы не могут использовать зарезервированные имена типов для своих целей. Список зарезервированных имён типов будет дан ближе к концу документации.
Все поля передаются через строку запроса, например: http://localhost:8080/microserver/get-job?type=MicroServer.25367be645.ExternalStatus
Важно заметить, что строка запроса имеет ограниченный набор разрешённых символов, поэтому остальные символы должны быть закодированы по правилам URL-кодирования.
Первые две команды из списка выполняются через GET-запрос, последняя – через POST запрос. Выполняемая через POST-запрос команда прописывается не в строке запроса, а в структуре basic content примерно так:

```
{
    "id":null,
    "visibleId":false,
    "type":"MicroServer.25367be645.CompensateUnderflow",
    "content":<object>
}
```

В данном случае наполнение почти аналогично GET-запросу: id игнорируется, visibleId тоже (так как эта команда обрабатывается непосредственно сервером). Поле «type» содержит команду, а вот поле «content» - массив из basic content.
Теперь мы подошли к, непосредственно, функциям ранее приведённых команд.

5.1. MicroServer.25367be645.ExternalStatus

Команда с таким названием (в тексте будем называть её ExternalStatus). Как вы могли заметить, часть имени типа (команды) – это имя сервера («MicroServer»), далее идут псевдослучайные байты, и только потом – имя команды (типа). Все символы имени являются разрешёнными в URL, так что кодирование их не требуется.
В качестве результата сервер возвращает JSON-массив следующего плана:

```
[
    {
        "type":<string>,
        "underflow":<bool>,
        "overflow":<bool>
    },
    <...>
]
```

Массив состоит из элементов, называемых «external status». Массив может содержать несколько элементов, или же быть пустым. Символ «<...>» означает, что на его месте может быть ещё несколько элементов external status. Запятая ставится только между элементами: после последнего элемента запятая не ставится (согласно правилам JSON).
Поле «type» показывает о каком именно типе идёт речь.
Поле «underflow» содержит логическое значение, показывающее нужно ли выслать дополнительные 32 элемента (можно выслать меньше или больше, но идеально – 32). Больше 127 высылать не стоит, так как сервер однозначно отклонит такой запрос. Присланные элементы сохраняются во внутреннем хранилище сервера.
Поле «overflow» содержит логическое значение, показывающее нужно ли забрать излишек последних в очереди элементов во внешнее хранилище. Сервер может выслать любое количество элементов, а внешнее хранилище должно принять их все, так как на момент отсылки эти элементы уже удалены из внутреннего хранилища сервера. Если внешнее хранилище не сможет принять все элементы basic content, непринятые будут безвозвратно утрачены (этого быть не должно для стабильной работы системы).
Поля «underflow» и «overflow» не могут быть истинными одновременно.
Поле получения ответа на запрос ExternalStatus, внешнее хранилище в зависимости от показателей underflow/overflow должно выполнить другие команды, а которых рассказ в следующем подразделе.
Если внешнее хранилище будет выполнять запросы ExternalStatus на сервер достаточно часто (например, раз в 5 секунд), то количество элементов в излишках должно быть близким к 32 (не меньше 33, но и ненамного больше).

5.2. MicroServer.25367be645.FetchOverflow

Выполнение этой команды допустимо только при условии, что забор излишков с сервера разрешён (имеет смысл) для конкретного типа. Для её вызова строка запроса должна иметь поле «type» со значением «MicroServer.25367be645.FetchOverflow» и поле «id» с названием типа, излишки которого нужно забрать. Именно так: параметр id указывать необходимо, при том, в него нужно поместить имя типа. Например: http://localhost:8080/microserver/get-job?type=MicroServer.25367be645.FetchOverflow&id=InterestedType

Если в ответ на такой запрос придёт код статуса 200, то содержанием будет JSON-массив следующего плана:

```
[
    {
        "id":<string>,
        "visibleId":<bool>,
        "type":<string>,
        "content":<object>
    },
    <...>
]
```

Как вы можете заметить, массив состоит из произвольного количества элементов basic content. Эти данные будут взяты из *начала* очереди (чтобы быстрее дать ход более новым(актуальным) пакетам).

5.3. MicroServer.25367be645.CompensateUnderflow

Как говорилось ранее, эта команда вызывается POST-запросом следующего вида:

```
{
    "id":null,
    "visibleId":false,
    "type":"MicroServer.25367be645.CompensateUnderflow",
    "content": [
        {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
        },
        <...>
    ]
}
```

Поле id игнорируется, visibleId тоже (так как эта команда обрабатывается непосредственно сервером). Поле «type» содержит команду, поле «content» - массив из basic content, которые нужно перенести *в конец* очереди (чтобы более новые(актуальные) могли пройти раньше) внутреннего хранилища сервера. Важно чтобы типы всех пакетов совпадали, иначе такой запрос будет отклонён сервером. Также запрос может быть отклонён, если в результате его выполнения случится излишек пакетов данного типа. Именно поэтому рекомендуется отсылать не более 32 пакетов за один вызов CompensateUnderflow.

6. Отладочные команды (для отладочной редакции)

Важно отметить, что любая отладочная команда является по своей сути диагностической, то есть не может никак повлиять на работу сервера или данные во внутреннем хранилище. Отладочные команды помогают в отладке микро-сервисов, показывая разработчику какие запросы (на какие пакеты) актуальны в данный момент, что хранится на сервере в момент выполнения команды и какие данные к моменту выполнения команды успели поступить/уйти. Все отладочные команды игнорируют is в строке запроса. Список команд с описанием ниже.

- MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot

Эта команда запрашивает полное содержимое внутреннего хранилища сервера. В ответ приходят все пакеты, которые в данный момент хранятся на сервере (с сервера не

удаляются). Не гарантируется, что результирующий массив будет отражать реальный порядок пакетов в очереди (на очередность пакетов друг относительно друга смотреть не стоит). Ответ имеет следующий формат:

```
[
    {
        "id":<string>,
        "visibleId":<bool>,
        "type":<string>,
        "content":<object>
    },
    <...>
]
```

Учтите, что **есть более экономичные** команды по количеству данных для передачи (перечислены ниже).

- MicroServer.25367be645.DebugEdition.getLocallyAvailibleTypes

Эта команда запрашивает список типов («type»), которые находятся во внутреннем хранилище. Типы, расположенные только во внешнего хранилище не учитываются. Ответ имеет следующий формат:

```
[
    <string>,
    <...>
]
```

- MicroServer.25367be645.DebugEdition.getTypesStatistic

Эта команда запрашивает полный список типов, которые доступны из внутреннего хранилища и те, которые могут быть доступны из внешнего, но во внутреннем отсутствуют. Ответ имеет следующий формат:

```
{
    <type string>:<unsinged integer>,
    <...>
}
```

Обратите внимание, ответ на эту команду – не массив, а объект ключ-значение, где ключ – это тип, а значение – количество хранимых объектов этого типа. По своей сути, это формат словаря.

- MicroServer.25367be645.DebugEdition.getPendings

Эта команда запрашивает список запросов, которые активны в момент выполнения команды. Запрос сервис направляет серверу и ждёт ответа. Данной командой можно посмотреть, данные какого типа и с каким id ожидаются подключёнными к серверу сервисами. Ответ имеет следующий формат:

```
[
    {
        "type":<string>,
        "id":<string>
    },
    <...>
]
```

- MicroServer.25367be645.DebugEdition.retrivePostHistory

Эта команда запрашивает список выполненных POST-запросов. Отладочная версия сервера сохраняет все выполненные GET- и POST-запросы. После выполнения этой команды север вернёт список выполненных запросов, очистив после этого список POST-запросов на своей стороне, так что следующая аналогичная команда не вернёт того, что вернула предыдущая. Сервер имеет ограничение на журнал POST-запросов: хранится не более 512 последних поступивших пакетов. Если пакетов оказывается больше, сервер удаляет 128 первых поступивших пакетов из журнала.

```
[
    {
        "datetime":<DateTime>,
        "content": {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
         }
    },
    <...>
]
```

Как можно заметить массив в ответе содержит объект, в котором содержится basic content (непосредственно то, что сервис отправил на сервер), а также дата и время поступления в следующем формате: «YYYY-MM-DDTHH:mm:ss.fffffff+HH:mm». Пример: «2021-10-04T10:35:40.9449944+03:00». После знака «+» идёт смещение временной зоны относительно UTC. Сама дата и время отображаются для текущего пояса, а не для UTC. **Важно отметить, что команды сервера не попадают в журнал, хотя по своей сути являются запросами получения или отправки данных.**

- MicroServer.25367be645.DebugEdition.retriveGetHistory

Эта команда запрашивает список выполненных GET-запросов. Отладочная версия сервера сохраняет все выполненные GET- и POST-запросы. После выполнения этой команды север вернёт список выполненных запросов, очистив после этого список GET-запросов на своей стороне, так что следующая аналогичная команда не вернёт того, что вернула предыдущая. Сервер имеет ограничение на журнал GET-запросов: хранится не более 512 последних полученных сервисами пакетов. Если пакетов оказывается больше, сервер удаляет 128 первых отправленных пакетов из журнала.

```
[
    {
        "datetime":<DateTime>,
        "requestedType":<string>,
        "requestedId":<string>,
        "content": {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
         }
    },
    <...>
]
```

Как можно заметить массив в ответе содержит объект, в котором содержится basic content (непосредственно то, что сервис получил с сервера), а также дата и время отправки с сервера в следующем формате: «YYYY-MM-DDTHH:mm:ss.fffffff+HH:mm». Пример: «2021-10-04T10:35:40.9449944+03:00». После знака «+» идёт смещение временной зоны относительно UTC. Сама дата и время отображаются для текущего пояса, а не для UTC.

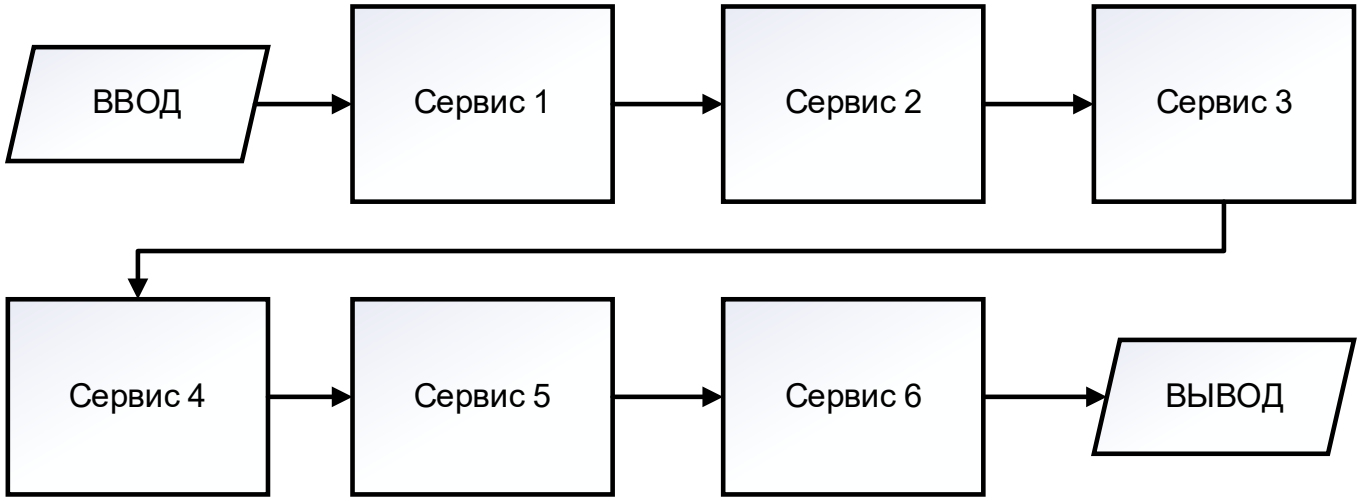
У данной команды ответ похож на ответ команды retrivePostHistory, но содержатся ещё 2 дополнительных поля: «requestedType» и «requestedId». Это именно те «type» и «id» соответственно, которые были отправлены в запросе пакета серверу. Если запрос содержал только тип, то «requestedId» будет содержать «null».

Важно отметить, что команды сервера не попадают в журнал, хотя по своей сути являются запросами получения или отправки данных.

7. Рекомендации по созданию клиентов

- При создании классов клиентов важно разграничить два основных режима работы системы: конвейерный и функциональный.

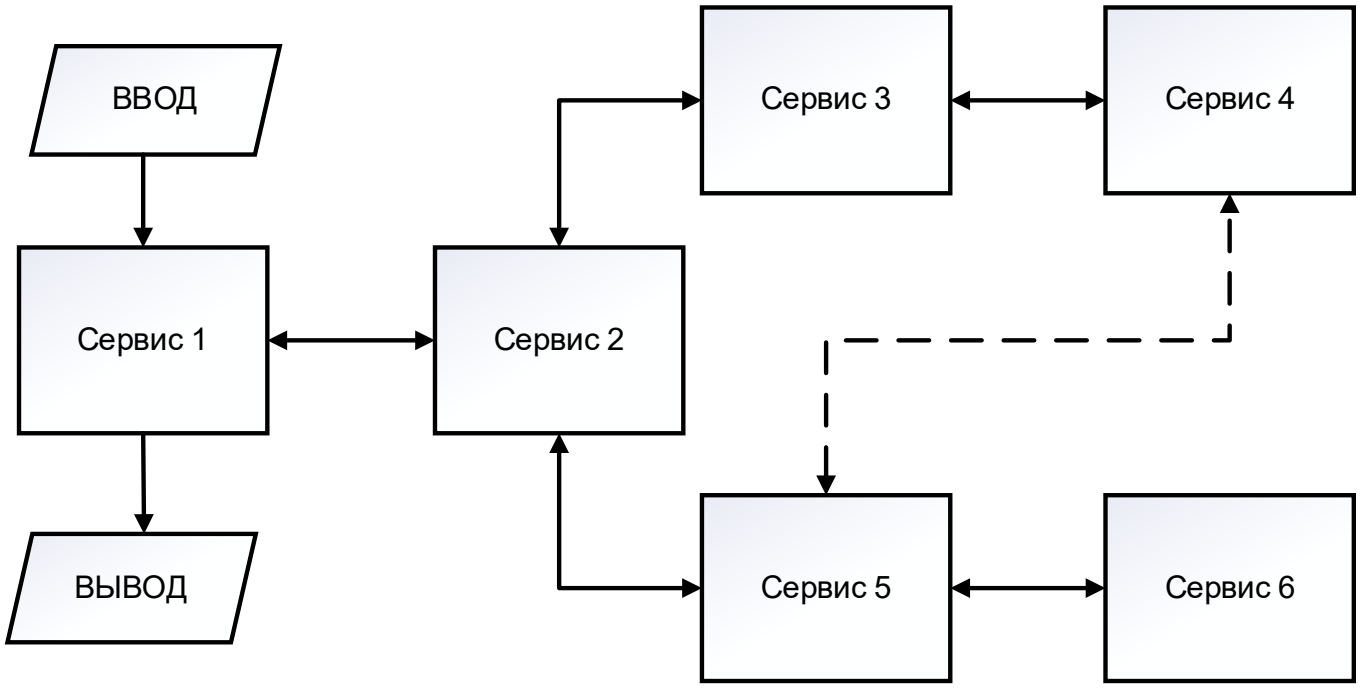
Первый режим устроен проще, но его масштабирование может оказаться проблемным. В рамках этого режима все сервисы получают пакеты исключительно по типу. Получается, что сервисы работают как конвейер, к началу которого поступил один тип, а с конца вышел другой, при том, каждый из сервисов (для одного конкретного типа) отработал не более одного раза. При обработке потока данных этот режим значительно превосходит функциональный. Схематически его можно представить так:



Функциональный режим работает наоборот: данные исходят и возвращаются в один и тот же сервис. Если в конвейерном режиме при обработке данные всегда проходили насквозь, то в функциональном данные проходят насквозь только в конечных сервисах (таких, которые сами не вызывают другие сервисы). В других же сервисах в один из моментов выполнялась бы отправка данных на сервер и ожидание ответа, из-за чего эти сервисы были бы недоступны (ожидали возвращение ответа – как обычная функция).

Схематически этот режим представлен на рисунке ниже. Важным полюсом данного режима является его более лёгкая масштабируемость и меньшая связанность микро-сервисов друг с другом. Также появляется возможность использовать один и тот же сервис на разных этапах обработки данных, что также может быть удобно. На рисунке это показано пунктирной линией. Как можно заметить, ввод и вывод выполняются из одного и того же сервиса, более того, в рамках одного запроса. Далее первый сервис вызывает второй и ждёт его результата, но второй тоже делает сам

не всё, поэтому вызывает сначала 3-й, а потом и 4-й, которые тоже сами делают не все и т.д. В итоге получается, что время выполнения запроса первым сервисом складывается из времени выполнения всеми сервисами.



Важно также не запутать пользователя с порядком отправки и получения, ведь если сервис сначала получает, а потом отправляет, то обрабатывает данные этот сервис, но если порядка наоборот (сначала отправляет, а потом получает), то значит данные обрабатывает другой сервис, а этот является «заказчиком».

- Важно следить за отправляемыми mime-типами. Дело в том, что сервер не принимает POST-запросы, в которых заголовок CONTENT-TYPE не содержит «application/json». Это необходимо, так как сервер принимает данные исключительно в формате JSON. Помимо этого, сервер отправляет данные также только в формате JSON, так что клиент также может установить требование на «application/json» в CONTENT-TYPE.
- Важно следить за отправляемой и принимаемой кодировкой. Сервер работает только с UTF-8.
- Важно выполнять кодирование значений полей «type» и «id» в соответствии с правилами для URL.
- Важно не допускать (выдавать ошибку) при комбинации «type» равен «null» и «visibleId» равен «false». Сервер не пропустит пакет с такой комбинацией (выдаст ошибку 4xx), так что лучше выполнить проверку на стороне клиента и выдать понятное сообщение об ошибке.
- Важным является соблюдение правил процесса получения данных с сервера. В рамках данного процесса клиент выполняет запросы с большим значением TIMEOUT (желательно около 60 секунд), на которые сервер отвечает с задержкой от 25 до 50 секунд или меньше. Сервер отвечает либо 2xx, либо 408. Успешный код означает, что сервер отправил данные, а код 408 – что время соединения вышло и нужно сделать новый запрос. Такая организация позволит сервису гораздо быстрее среагировать на появление новых данных на сервере. Если клиент разорвёт соединение раньше 25 секунд с момента создания соединения, это может стать причиной безвозвратной потери пакета, если клиент разорвёт соединение по прошествии 25 секунд (но всё-таки, желательно 30 т.к. измерения времени на клиенте и сервере могут немного отличаться из-за задержек), то ничего плохого не случится – можно будет делать новый запрос. Однако, гораздо проще получается доверить разрыв старого соединения серверу.

- Важно не использовать специальные типы для пользовательских задач. Специальными считаются следующие типы:
  - MicroServer.25367be645.ExternalStatus
  - MicroServer.25367be645.FetchOverflow
  - MicroServer.25367be645.CompensateUnderflow
  - MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot
  - MicroServer.25367be645.DebugEdition.getLocallyAvailibleTypes
  - MicroServer.25367be645.DebugEdition.getTypesStatistic
  - MicroServer.25367be645.DebugEdition.retrivePostHistory
  - MicroServer.25367be645.DebugEdition.retriveGetHistory
  - MicroServer.25367be645.DebugEdition.getPendings
  - MicroServer.25367be645.GET_TIMEOUT_25

Любой из специальных типов обрабатывается НЕ как обычный пользовательский тип, даже если редакция не отладочная. Последний в списке тип служит для внутренних нужд сервера и со стороны клиента не обрабатывается в принципе.

- Важно чтобы клиент обрабатывал все коды состояний 2xx как успешные (а не только 200), так как сервер возвращает и другие коды (например, 201).
- Желательно избегать комбинации «type» равен «null» и «id» равен «null», так как это ведёт к не интуитивному поведению сервера (строка «null» в поле «id» считается за обычный id, под который ищется пакет).

Разрешается не указывать в строке запроса поле, значение которого null (сервер по умолчанию подставляет null), но пропускать оба поля («type» и «id») нельзя, так как комбинация из двух незаданных полей является нежелательной.

- Желательно оформить функциональный запрос непосредственно как функцию, которая принимает в аргументах тип входного объекта, тип выходного объекта (не обязательно), входной объект, и возвращает объект.
- Желательно реализовать некоторую защиту и передачу id, поступившего в сервис с сервера в пакете для обработки, в функцию отправки результата работы.
- Также желательно реализовать две функции отправки результата: одну – для отправки финального результата (с видимым id чтобы сервис-клиент забрал этот пакет), а второй – для конвейерной обработки (с видимым или невидимым id в зависимости от того, каким он пришёл в этот сервис).
- Желательно реализовать «настойчивый» режим работы клиента, в котором клиент не выдаст ошибку в случае отказа соединения (например, если сервер ещё не запущен). Такой режим позволит не переживать за очерёдность запуска сервисов и сервера.

8. Защиты и альтернативные сервера

Функционал MicroServer достаточно прост для реализации в виде другого сервера с расширенными или уменьшенными возможностями. В целом, сервер можно разделить на три функциональные части: хранилище пакетов, обработчик запросов пакетов, обработчик приёма пакетов.

В самом простом случае хранилище может быть представлено простым списком (даже не очередью), в котором будут храниться структуры вида basic content. Хранилище должно быть потоко-безопасным, так что в самом простом случае любая операция со списком элементов должна быть окружена блоком mutex. Операция вставки элемента в хранилище – простое добавление в конец списка. Операция получения элемента из списка – перебор всех элементов в списке, пока не найдётся подходящий по всем условиям («type», «id», «visibleId»). Нужно также учесть случай, когда нужный элемент не находится.

Часть, отвечающая за запрос данных в самом простом случае, выполняет проверку того, заданы ли параметры строки запроса URL, и, если не заданы, даёт полям значение «null». Помимо этого, в этой части находится цикл запроса данных из хранилища. Этот цикл должен выполняться до тех пор, пока либо не найден подходящий под запрос элемент, либо прошло 25 секунд. После выхода из цикла, если элемент найден, этот элемент возвращается клиенту, иначе клиенту возвращается статус-код 408.

Часть, отвечающая за получение данных сервером, в самом простом случае должна только проверять, что клиент прислал пакет, в котором «type» не равен «null» или «visibleId» не равен «false». Если это не так, то сервер должен отклонить пакет, так как пакет такой конфигурации не может быть запрошен с сервера никаким образом.

Крайне желательны защиты, связанные с проверкой заголовка CONTENT-TYPE (на содержание application/json). Также желательно, чтобы сервер при отправке пакетов ставил заголовок CONTENT-TYPE.

Остальные защиты ставятся в зависимости от необходимости согласно описанию протокола, приведённому выше. В MicroServer на запрос данных с сервера реализованы следующие проверки:

- Параметры в строке запросов есть
- Параметры в строке запросов – это либо «type», либо «id»
- Указанное имя типа не совпадает с зарезервированными специальными типами

Если указан специальный тип, то далее выполняются следующие проверки:

- Указанный специальный тип (команда) разрешён

На отправку данных на сервер реализованы следующие проверки:

- Задан ли заголовок CONTENT-TYPE
- Есть ли в этом заголовке строка «application/json»
- Имеет ли полученный пакет ожидаемую сигнатуру (набор полей с определённого типа данных)
- Есть ли ситуация «type» равен «null» и «visibleId» равен «false»
- Имя типа в пакете не совпадает с зарезервированными специальными типами

Если указан специальный тип, то далее выполняются следующие проверки:

- Указанный специальный тип (команда) разрешён
- Подходят ли данные поля content под определённую ожидаемую сигнатуру
- Элементов в массиве больше нуля
- Имена типов элементов не совпадают с зарезервированными именами