ACP Assignment 2 Specifications (Programming Task)

07.03.2025

In this assignment you are supposed to implement a service which provides several endpoints to communicate with the AcpStorageService (BLOB), Kafka, RabbitMQ and redis

The auto-marker will call your endpoints using PUT, POST and DELETE accordingly. Due to passing data in JSON-format in the body of a message, we only have classical GET methods, where no JSON-data is needed as a GET request is not supposed to be using the body of the message.

Your main tasks can be summarized as follows (no changes from assignment 1):

1. Create a Java-REST-Service
   - Preferred with Spring Boot, though other frameworks can be used as well
   - Port 8080 is consumed
   - Implement the endpoints
   - Proper parameter handling
   - Proper return code handling
   - JSON handling

2. save the docker image in a file called **acp_submission_image.tar**

3. **place** the file **acp_submission_image.tar into your root directory of your solution**

   Your directory would look something like this:

   acp_submission_2
   acp_submission_image.tar
   src (the Java sources…)
   main
   …
   …

4. Create a ZIP file of your solution directory
   - Image
   - Sources
   - IntelliJ (or whatever IDE you are using) project files

5. upload the ZIP as your submission in Learn

For the service you can either use your own implementation or the template provided in Github at https://github.com/mglienecke/AcpCw2Template.git

This repository is in constant development, so use it as a basis, but not as a fixed, never changing element. The main factors for using it are the ability to have the actuator endpoints (see lecture of today) and some general housekeeping tasks (like creating an environment, etc.).

General information:

- All mandatory connection information to subsystems will be passed as environment variables, or – if explicitly specified – in the endpoints.

- The properties used in all Kafka-examples, which are not passed in as environment variables or in the request as JSON-data, can be assumed as constant (see the example in the repository). This includes things like the offset, the type of serializer, etc.

- So, you must:

    - Read the necessary environment variables
    - Additionally for some requests handle additional data

- All JSON passed in will be always in the syntactical correct format, so you can ignore error handling there. What you still must check is that the data gives you a proper connection and no wrong address or invalid user / password is passed (so classical flow errors in an application).

- No knowingly invalid data will be passed

- Parameters given as {…} in the task will be replaced at runtime with the corresponding value. So, {queueName} could become s12345678_outbound_queue

- The points after a task are the maximum achievable points for the individual task

- The auto-marker will not force to produce 500 codes, yet should a 500 arise this is a clear indicator that you didn't catch an exception…

- **All endpoints are expected to deliver a 200 response to indicate success**

- No prefix (like /api/v1) is to be used. **Every endpoint must be reachable from the bound root**

Predefined environment variables

The following environment variables will be available:

- REDIS_HOST (e.g. host.docker.internal)
- REDIS_PORT (e.g. 6379)
- RABBITMQ_HOST (e.g. host.docker.internal)
- RABBITMQ_PORT (e.g. 5672)

- KAFKA_BOOTSTRAP_SERVERS (e.g. kafka:9093)
- ACP_STORAGE_SERVICE (e.g. https://acp-storage.azurewebsites.net/)

The AcpStorageService can be investigated using https://acp-storage.azurewebsites.net/swagger-ui/index.html

For Kafka, RabbitMQ and redis you can assume a local dockerized environment for the auto-marker. So, you can user either something similar, or confluent.io (you have the registration information), but then you will have to set some security information like security protocol, SASL mechanism and JAAS configuration).

The auto-marker uses no authentication, so any present authentication in the connection properties will cause a failure.

Do not (!) use the Kafka-Admin-API, as this will not be necessary and wouldn't be available in a larger environment.

The mark distribution will be like:

- PUT RabbitMQ (3)
- GET RabbitMQ (3)
- PUT Kafka (3)
- GET Kafka (3)

- POST processMessages (17)
- POST transformMessages (16)

For the 2 larger items marks for sub-achievements will be given. The simpler tasks are "all or nothing" (so, 3 or 0 points), unless repeat errors.

This assignment has many subtle details. Please check every exactly as otherwise you will lose points.

The REST-Service must provide the following endpoints:

- **(3) PUT rabbitMq/{queueName}/{messageCount}**

  Write {messageCount} messages into the queue defined by {queueName}. Each message has to be the following JSON-format:

  {
  "uid": "*replace with your student id*",
  "counter": numerical index of the message **starting at 0**
  }

- **`(3) GET rabbitMq/{queueName}/{timeoutInMsec}`**

  Return the data read from the topic {queueName} as a List<String> from the service – one entry being one message.

  You read until {timeoutInMsec} has passed. If you stay longer than that plus 200 msec in the routine, this will considered a fail. So, if the timeout is 1000 msec, you must return after 1,200 msec the very latest.

  Normally your timeout will be quite small (around 100 msec)

- **(3) PUT kafka/{writeTopic}/{messageCount}**

  Same as above with rabbitMQ, just now into a kafka topic

- **`(3) GET kafka/{readTopic}/{timeoutInMsec}`**

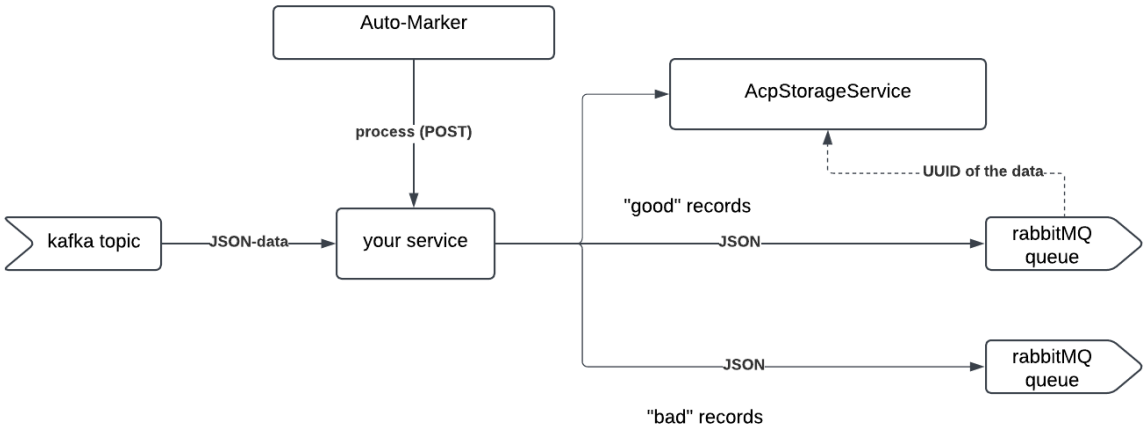
  Same as above with rabbitMQ, just now a kafka topic and the timeout will be a bit larger (around 500 – 5000 msec).

- **`(17) POST processMessages`**

  in the call you will receive additional JSON body data in the following format:

```
{
 "readTopic": "topicname",
 "writeQueueGood": "queuename",
 "writeQueueBad": "queuename",
 "messageCount": value (between 1 and 500)
}
```

The overall system diagram can be depicted as:



Your task is to read {messageCount} JSON-data items (no time constraint) from the kafka topic (reset your offset to the beginning!) in the following format:

```
{
 "uid": "your UID in the format sXXXXXX",
 "key": 3-, 4- or 5-character long character sequences as string,
 "comment": a string,
 "value": floating point value
}
```

If the key is 3 or 4 characters long, this is a "good" packet and you store the data packet it in the AcpStorageService.

Before you can do this, you have to add a field to the JSON:

"runningTotalValue": running total of all good message values to this point (and including it)

So, you **are writing the original message with the running total as a new field** in the JSON object to the ACP storage service.

The returned UUID from the service is added to the **original** JSON object (as field uuid in the JSON object – thus like “uuid”: “12121-xxx…”) which you read and the whole is written to the {writeQueueGood}.

For "bad" packets, you write the message directly (so, without storing!) to the {writeQueueBad}.

After you are finished, you write a new packet like above with a key "TOTAL" to the good and bad queue, where you put as a value the added packets for the corresponding queues. The comment is not relevant (but must be set) and no UUID must be added to this message as no data was written.

Example:

- You receive 3 messages AAA, ABCD and ABCDE with a value of 10.5 each
- 2 write 2 messages to the store (as good)
    i. Increment the running total accordingly (10.5, 21.0)
- You write 2 messages (with the corresponding UUID) to the good queue
- You write 1 message (no store) to the bad queue
- You write a TOTAL message with 21 to the good queue (comment "")
- You write a TOTAL message with 10.5 to the bad queue (comment "")
- Your UID is just copied along

- **(16) `POST transformMessages`**

In the call you will receive additional JSON body data in the following format:

```
{
 "readQueue": "queuename",
 "writeQueue": "queuename",
 "messageCount": value (between 1 and 500)
}
```

Your main task is to read {messageCount} messages (no time-limit) from the readQueue (rabbitMQ), process them and write them to the writeQueue.

Each message can be in 2 formats – "normal" and "tombstone". "Tombstone" messages are often used to synchronize something or to mark an end to a process, etc. (thus the name…).

The "normal" format is:

```
{
 "key": any string,
 "version": integer (1..n)
 "value": any float
}
```

The "tombstone" is:

```
{
  "key": any string,
```

```
 no additional data…
}
```

For normal packets you have to check if the key in this specific version is already present in redis. If yes, you just pass the packet 1:1 to the writeQueue without processing.

If not present, or in an older version (so redis version < current version), then you store the entry in redis and pass the packet with a value increased by 10.5 to the writeQueue.

for tombstone packets you remove the key from redis and act like that key has not been set before. You are writing a special packet to the outbound queue:

```
{
 "totalMessagesWritten": integer,
 "totalMessagesProcessed": integer,
 "totalRedisUpdates": integer,
 "totalValueWritten": float, (the total of all packets up to now)
 "totalAdded": float (the total of all 10.5 you added up to now)
}
```

These are the current running values up to the moment ~~until~~ (and including when, for total messages, etc.) the tombstone was hit.

Tombstones can occur several times!

Example:

- Receive "ABC", Version 1, Value 100
    i. Store in redis
    ii. Write "ABC", Version 1, Value 110.5 to the out queue
- Receive "ABC", Version 1, Value 200
    i. Write "ABC", Version 1, Value 200 to the out queue
- Receive "ABC", Version 3, Value 400
    i. Store in redis
    ii. Write "ABC", Version 1, Value 410.5 to the out queue
- Receive "ABC", Version 2, Value 200
    i. Write "ABC", Version 2, Value 200 to the out queue
- Receive "ABC" -> Tombstone
    i. Clear redis
    ii. No write
- Receive "ABC", Version 2, Value 200
    i. Store in redis
    ii. Write "ABC", Version 2, Value 210.5 to the out queue (as new!)

So, you can receive out of sync (older version) packages as well

The following should be considered when implementing the REST-service:

- Do proper checking – all data will be valid, yet still you have to check some things (like return codes, exceptions, etc.)
- Your endpoint names have to match the specification
- Test your endpoints using a tool like Postman or curl. Plain Chrome / Firefox, etc. will do equally for the GET operations
- The filename for the docker image file has to be exactly as defined as well as the location of it in the ZIP-file. Should you be in doubt, use copy & paste to get the name right

Should you need help:

- See the lecture recordings. You should find all information there
- If you cannot find an answer to your question, please post it on Piazza, though try finding it yourself first, please (as we have only limited capacity)

Disclaimer: We will not able to answer last minute questions right before the deadline, so please make sure you start the assignment in good time.

Piazza support will close 3 days before the submission deadline

Marking:

This programming task has a maximum mark of 45 / 100 points in relation to the entire ACP mark for the course (the essay task will be 25 / 100 in relation to the entire ACP mark for assignment 2).

The marks will be allocated purely on auto-tests based on the following criteria:

- Proper runnable docker image
- Proper behavior (functionality)
- Proper error handling
- Proper status codes

Additional information for accessing the ACP Storage service:

- The storage service URL to be used must be taken from the environment variable (see in the beginning of this document)

- To check the available API you can use: https://acp-storage.azurewebsites.net/swagger-ui/index.html

- Storing an item using curl could be done like:

```
curl -X 'POST' \  'https://acp-
storage.azurewebsites.net/api/v1/blob' \  -H 'accept:
*/*' \  -H 'Content-Type: application/json' \  -d
'{  "item1": 1234565,  "item2": "ABC" }'
```

Which might return:
“caa04bde-dff9-4432-ad50-b70fb23ce7c6”

- To retrieve the item again using curl you would use:

```
curl -X 'GET' 'https://acp-
storage.azurewebsites.net/api/v1/blob/caa04bde-dff9-
4432-ad50-b70fb23ce7c6'  -H 'accept: */*'
```

Which would return:

```
{
  "item1": 1234565,
  "item2": "ABC"
}
```

Clarification as of 19th March 2025 (already on piazza)

- all values / counters / running values, etc. **will always include the current message as well**.

- A tombstone will not be sent as the first message

- All events sent to your service are intended to be executed while they are "inside" the service (so executing your code).
  There won't be several calls to e.g. processMessages. The intention is that you listen and process until x messages are handled and then you terminate.

- Should you swallow messages (or get a hiccup) the auto-marker would terminate the service after a grace period (around 1,500 msec) after the last message was sent to be processed.
  In principal, you can imagine the approach like 2 parallel instances running:
  - auto-marker starts generating messages
  - calls your service on a 2nd thread
  - continues to process messages
  - reads / checks your responses