[**MDP WebHook Proposal**](#_t4sme8ypzpu6) **2**

[Introduction](#_b53jgyuagx8s) 2

[Design Parameters](#_53046brjxs41) 2

[System Components:](#_vgqbqyt8yxw3) 3

[High level Diagram](#_spg9wnpd1xwg) 3

[**Event Dispatcher**](#_ppb9lt8fq9ah) **4**

[Http Sink Connector as Event Dispatcher](#_2j5ot252purg) 6

[Azure Function Sink Connector as Event Dispatcher](#_tl7gxkc13old) 6

[Apache Kafka trigger as Event Dispatcher](#_vw15qew1tz6b) 6

[Regular Consumer as Event Dispatcher](#_udw9cj3ghmx4) 7

[Transformer](#_ulqe06415mfm) 8

[Azure qualities](#_hrwhw74z4hpl) 8

[**WebHook Api**](#_4pi7cmxk3fjc) **8**

[Common API behavior](#_kavaomj72npf) 8

[Request content type](#_qu3qe2kwivzh) 9

[Request query string](#_8c7ztraifrwi) 9

[Error Response](#_ukzcgcaaaqz) 9

[Manage event subscriptions](#_wwwkq0adt65g) 9

[Put event subscription (create / update)](#_sf11bcogc5j1) 9

[Get event subscription](#_3abgv2vk3arl) 11

[Get event subscriptions](#_5a8m0jof3ary) 11

[Delete event subscription](#_1hecnzfx1jfb) 11

[WebHook Payload](#_gf3e7nwm9ao2) 12

[Object types](#_wincqf4dycbe) 12

[Event types](#_tc6i62rkls2w) 12

[Developer User Interface](#_f3mpy27fqvra) 13

[**Webhook HTTP delivery**](#_lg15xmh3hsk2) **13**

[With Azure Event Grid](#_4qeo19fw2hlc) 14

[With Custom solution](#_6181t2wcqqld) 14

[Prototype](#_ejb9k7f6lgkp) 15

[Install & Setup](#_1gjhebjf92im) 16

[Conclusion](#_o2zlfdv9n4gz) 18

[**MicroServices Connectivity Proposal**](#_6th9bb8nmwf8) **18**

[Interaction 1 to1](#_bs71ykw2uscx) 18

[Interaction via Broker](#_vik6kusdt3eu) 19

[High level async 2-way architecture with AMQP](#_jn4j7a3l31db) 20

[High level acknowledge based architecture with AMQP](#_7dn49d9qzgae) 21

[Conclusion](#_nmgwj8otzeit) 22

[Notes](#_9t2pn03i2y7h) 22

# MDP WebHook Proposal

## **Introduction**

A webhook is a one-way push of data, used to notify a system that one or more events have occurred via HTTP POST with a JSON payload that expects a 200 OK response to acknowledge receipt of the payload.

At VLMR we send webhooks to notify partners when a booking event occurs. If a partner has configured its external service to be a part, that service is notified when a booking event occured. The notification payload contains a unique id, timestamp and the details.

From a webhook receiver perspective (e.g. a partner), the reliability of an event dispatcher is paramount in many business contexts. For example, a dropped notification that a scheduled message that is pending review would stop the publication of a potentially timing-critical marketing message. Untrustworthy APIs are pretty easy to detect during initial integration testing and will be rejected for business-critical uses. Low latency delivery of notifications is also important in some contexts, for example, when a human is monitoring the progress of a task.

### **Design Parameters**

**Desirable per-receiver delivery features include:**

* At-least-once delivery of each event notification — i.e. no message loss.
* Overall notification ordering that preserves individual event producer ordering

**Security considerations:**

* Prevent user/partner/customer-provided webhook URLs from probing internal infrastructure.
* Require HTTPS webhook URLs for privacy and data integrity.
* Authenticate the webhook dispatcher to the receiver — HMAC on webhook payloads.
* Prevent resource exhaustion from misbehaved webhook receivers by discarding response headers and bodies that are excessively long — we’re just looking for a 200 OK.

**Scale considerations**

* Event notification payload size — a few kB
* Number of webhook receivers (URLs) — millions
* Number of events per minute per destination — less than 10, some in 1000s

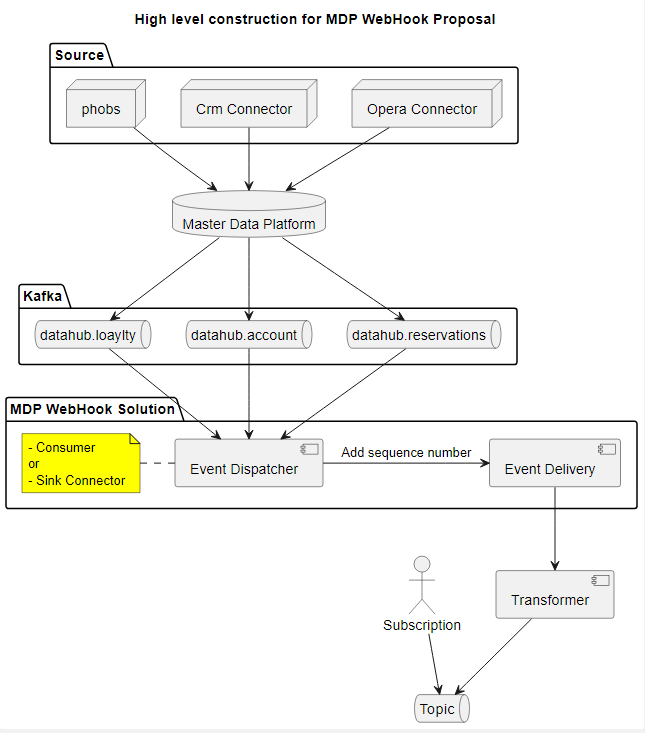
**Challenges:**

* Unreliable webhook receivers — buffering, retry/backoff timers, order-preservation
* Coordinating of state for a receiver retained by the dispatcher — next seq number, buffered notifications, last retry time, etc.
* Dispatcher process or node failure must not lose notification events
* Resource limitation aware throttling — RAM, file descriptors, source ports

### **System Components:**

* **Event dispatcher**
* **Webhook receiver’s API: an HTTP POST, headers, payload format and subscription management**
* **Webhook delivery**

### **High level Diagram**

****

## **Event Dispatcher**

The source of events is a set of topics into which MDP publishes messages (topic.datahub.reservations etc).

| Events will be queued according to object type (account, reservation, loyalty). In addition, each queue is also divided into the number of supported schema formats. The subscriber is attached to the queue-scheme in which he wants to receive payloads. |
| --- |

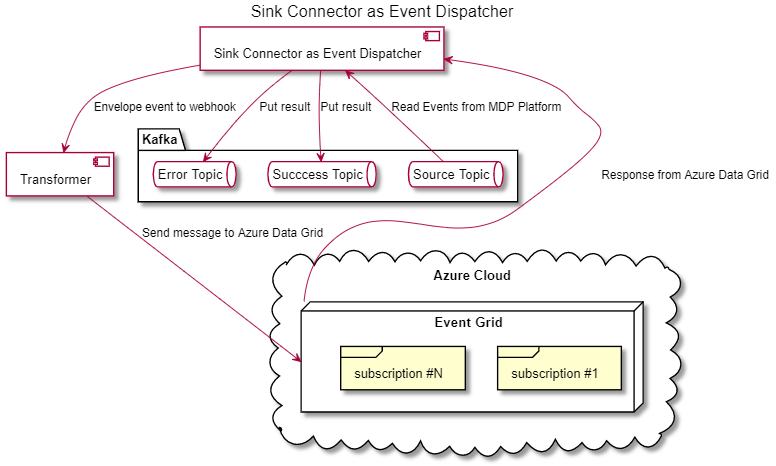
Each topic is assigned another log-compacted topic for storing the state of delivery (or each webhook receiver state \*depending on the final solution). Upon failure of a event dispatcher, the Kafka brokers rebalance the consumer group and any dispatcher process assigned a new request partition loads the state associated with the receivers in that partition from the same partition number of the log compacted topic

There are 2 approaches to consume events from Kafka to turn them into webhooks.

* Sink Connector
* Consumer

|  | Advantages | Limitations |
| --- | --- | --- |
| Sink Connector | At a minimum no development is required | Only Java development |
| Consumer | Develop with .NET or else | Development is required |
| Sink Connector | Similarly to the Kafka Connect Source API, the Kafka Connect Sink API allows you to leverage the ecosystem of existing Kafka Connectors out there to perform your streaming ETL without writing a single line of code. Kafka Connect Sink API is built on top of the consumer API, but does not look this different from it. | If the data sink you’re writing to does not have an available connector (yet), you will have to write a Kafka Connect Sink (or consumer, if you prefer), and the debugging process might be a bit more complicated. |
| Consumer | The Kafka Consumer API is dead-simple, works using Consumer Groups so that your topics can be consumed in parallel. Although you need to be careful about a few things, such as offset management and commits, as well as rebalances and idempotence constraints, they’re really easy to write. For any stateless kind of workload, they will be perfect. | When you perform some kind of ETL, Kafka Connect Sinks are better suited as they will avoid you to write some complicated logic against an external data source. |

### **Http Sink Connector as Event Dispatcher**

The connector consumes records from Kafka topic(s) and guarantees that records from the Kafka topic are delivered at least once.

### **Azure Function Sink Connector as Event Dispatcher**

The Kafka Connect Azure Functions Sink connector for Confluent Cloud integrates Apache Kafka® with Azure Functions

Read more

<https://docs.confluent.io/cloud/current/connectors/cc-azure-functions-sink.html>

### **Apache Kafka trigger as Event Dispatcher**

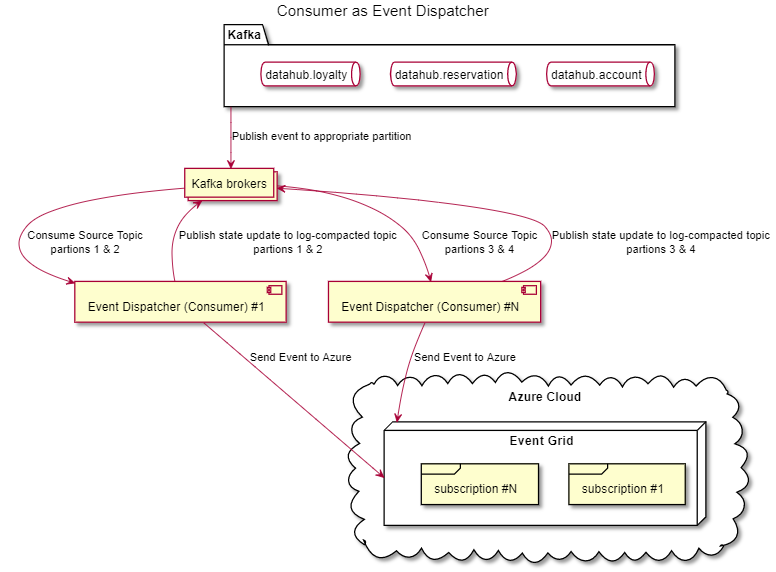
Use the Apache Kafka trigger in Azure Functions to run your function code in response to messages in Kafka topics.

Read more

<https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-kafka-trigger?tabs=in-process%2Cconfluent&pivots=programming-language-csharp>

### **Regular Consumer as Event Dispatcher**

The design uses two topics with the same number of partitions. One topic is used to source events. The second topic is a Kafka log-compacted topic for storing the state of each event (or each webhook receiver state in the next versions\*) in the associated partition. Kafka brokers can store the last processed offset per consumer group-partition pair, as set by the dispatcher process. The request partition offset is updated only when all requests prior to the new offset value have been stored in the log compacted topic or sent. Should a failure occur, any re-read requests from the request topic whose receiver state was not completely updated resend the webhook to the receiver with the identical sequence number, allowing duplicates to be discarded at the receiver.



Read more about Consumer vs Sink Connector

[**https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql-ef584274c1e**](https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql-ef584274c1e)

### **Transformer**

Transformer is a small algorithmic unit that converts and enriches input events of the MDP platform into a payload of Web Hook for Azure or Custom solution.

In the case of using Sink Connector as Event Dispatcher, this will be “Single Message Transforms for Confluent Platform”. Read more <https://docs.confluent.io/platform/current/connect/transforms/overview.html>

In the case of using Azure Function as Transformer. <https://github.com/kgonsovsky/mdp-webhook/tree/main/mdpEventInputFunction>

#### Azure qualities

Azure delivers events as an array of payloads of unknown length per each batch. That is not solvable. <https://github.com/MicrosoftDocs/azure-docs/issues/42797>

## **WebHook Api**

The Webhook API allows you to subscribe to events happening in the account with your integration installed. WebHooks endpoints are secured by adding query parameter <<secret>> to the webhook destination URL specified as part of creating an Event Subscription.

The WebhookAPI allows to specify exactly what the partner wants to receive. Filters can be applied to the object type (accounting, reservation...), event type (creation, cancelation...) and additionally for maximum flexibility use the JSON filtering syntax.

In order to administer subscriptions and provide web interface for partners, we can develop custom solution or use Azure Event Grid (do not confuse with Azure Data Hub)

| Feature | Azure Event Grid | Custom |
| --- | --- | --- |
| BackEnd Development | Not necessary, but need synchronization if needed | Necessarily |
| FrontEnd Development | Desirable, but not blocking | Necessarily |
| Management | IT administrator | Developer/Administrator |
| Customer Satisfaction | Rather positive | Unknown |
| Documentation | Only business processes and entities | Business processes, entities & technical details |

### **Common API behavior**

WebHook Api has the following APIs exposed over HTTPS (port 443).

Base URL for HTTP: [http://\_server\_name\_:443](about:blank)

### **Request content type**

All API requests must have a Content-Type. The value of Content-Type can be one of the following values:

Content-Type: application/json or Content-Type: application/json; charset=utf-8

### **Request query string**

All API requests require the following query string parameter: ?api-version=2019-01-01-preview

### Error Response

| {  "error":  {  "code": "<HTTP STATUS CODE>",  "details":  {  "code": "<Detailed Error Code>",  "message": "..."  }  }  } |
| --- |

### **Manage event subscriptions**

#### **Put event subscription (create / update)**

Request: PUT /topics/<topic\_name>/eventSubscriptions/<subscription\_name>?api-version=2019-01-01-preview

Payload:

| **{**  **"name": "<subscription\_name>", // optional, inferred from URL. If specified must match URL subscription\_name**  **"properties":**  **{**  **"topicName": "<topic\_name>", // optional, inferred from URL. If specified must match URL topic\_name**  **"eventDeliverySchema": "EventGridSchema | CustomEventSchema | CloudEventSchemaV1\_0", // optional**  **"retryPolicy": //optional**  **{**  **"eventExpiryInMinutes": 120,**  **"maxDeliveryAttempts": 50**  **},**  **"persistencePolicy": "true",**  **"destination":**  **{**  **"endpointType": "WebHook",**  **"properties":**  **{**  **"endpointUrl": "<webhook\_url>",**  **"maxEventsPerBatch": 10, // optional**  **"preferredBatchSizeInKilobytes": 1033 // optional**  **}**  **},**  **"filter": // optional**  **{**  **"subjectBeginsWith": "...",**  **"subjectEndsWith": "...",**  **"isSubjectCaseSensitive": true|false,**  **"includedEventTypes": ["...", "..."],**  **"advancedFilters":**  **[**  **{**  **"OperatorType": "BoolEquals",**  **"Key": "...",**  **"Value": "..."**  **},**  **{**  **"OperatorType": "NumberLessThan",**  **"Key": "...",**  **"Value": <number>**  **},**  **{**  **"OperatorType": "NumberGreaterThan",**  **"Key": "...",**  **"Value": <number>**  **},**  **{**  **"OperatorType": "NumberLessThanOrEquals",**  **"Key": "...",**  **"Value": <number>**  **},**  **{**  **"OperatorType": "NumberGreaterThanOrEquals",**  **"Key": "...",**  **"Value": <number>**  **},**  **{**  **"OperatorType": "NumberIn",**  **"Key": "...",**  **"Values": [<number>, <number>, <number>]**  **},**  **{**  **"OperatorType": "NumberNotIn",**  **"Key": "...",**  **"Values": [<number>, <number>, <number>]**  **},**  **{**  **"OperatorType": "StringIn",**  **"Key": "...",**  **"Values": ["...", "...", "..."]**  **},**  **{**  **"OperatorType": "StringNotIn",**  **"Key": "...",**  **"Values": ["...", "...", "..."]**  **},**  **{**  **"OperatorType": "StringBeginsWith",**  **"Key": "...",**  **"Values": ["...", "...", "..."]**  **},**  **{**  **"OperatorType": "StringEndsWith",**  **"Key": "...",**  **"Values": ["...", "...", "..."]**  **},**  **{**  **"OperatorType": "StringContains",**  **"Key": "...",**  **"Values": ["...", "...", "..."]**  **}**  **]**  **}**  **}**  **}** |
| --- |

#### **Get event subscription**

Request: **GET /topics/<topic\_name>/eventSubscriptions/<subscription\_name>?api-version=2019-01-01-preview**

Response: HTTP 200

#### **Get event subscriptions**

Request: GET /topics/<topic\_name>/eventSubscriptions?api-version=2019-01-01-preview

Response: HTTP 200

#### **Delete event subscription**

Request: DELETE /topics/<topic\_name>/eventSubscriptions/<subscription\_name>?api-version=2019-01-01-preview

Response: HTTP 200, no payload

### **WebHook Payload**

| **{**  **"schema":{**  **"type":"string"**  **"optional":false**  **}**  **"payload":"{"\_id": {"\_data": "82627A06CB000000032B022C0100296E5A1004BA90090870E04E3BBA59F3A4325E2E2446645F69640064627A06C926FA9FB10659D6B10004"}, "operationType": "replace", "clusterTime": {"$timestamp": {"t": 1652164299, "i": 3}}, "fullDocument": {"\_id": {"$oid": "627a06c926fa9fb10659d6b1"}, "masterId": "dafb48e9-2197-4799-ba47-3bb5e0eec209", "isActive": true, "isDeleted": false, "createdBySystem": "PHOBS", "createdDate": {"$date": 1652164297903}, "modifiedBySystem": null, "modifiedDate": null, "version": 2, "mappingFields": {"VRC": {"\_id": "627a06cb7511d533fcebd7b4"}}, "personalData": {"firstName": "Damir", "lastName": "Imamović", "salutation": null, "gender": null, "language": null, "birthDate": null, "age": null, "familyStatusIds": []}, "address": {"country": null, "city": null, "street": null, "zipCode": null}, "contact": {"phone": null, "email": null, "viber": null, "messenger": null, "additional": {"phones": [], "emails": ["mosnik@icloud.com"], "tempEmails": []}}, "additionalInformation": {"revenueSegment": null, "isBlacklisted": false, "gdprDelete": false, "isVip": false, "customerMemberType": null, "isOtaRestricted": false, "extraMeal": false, "babyCot": false, "travelWithPets": false, "isTopGuest": false}, "advertisingPermissions": {"email": null, "sms": null, "viber": null, "whatsApp": null, "push": null, "segmentation": null}, "products": {"valamar": null, "camping": null, "bike": null, "valfresco": null}, "interestIds": [], "status": null, "duplicated": false, "confirmed": false}, "ns": {"db": "DataHub", "coll": "accounts"}, "documentKey": {"\_id": {"$oid": "627a06c926fa9fb10659d6b1"}}}"**  **}** |
| --- |

Payload field descriptions

Id is mandatory. It can be any string value that's populated by the caller.

Subject is mandatory. See Subject Types below.

EventType is mandatory, See Event Types below.

EventTime is mandatory, it's not validated but should be a proper DateTime.

DataVersion is mandatory

Data is optional, and can be any JSON token (number, string, boolean, array, object)

#### **Object types**

**Accounting**

**Reservation**

**Loyalty**

#### **Event types**

**Creation**

**Modification**

Read more about REST APIs of Azure Event Grid

<https://docs.microsoft.com/en-us/azure/event-grid/edge/api>

### **Developer User Interface**

To help our partners quickly debug webhook integrations we offer a web browser user interface that shows the last webhook response from their URL. This also helps reduce any question about the reliability of our webhook dispatcher. There is also a UI for configuring the webhook URL associated with their application.

| In the case of using Azure Data Grid. |
| --- |

The web interface for users is a proxy script that relays API commands to the Azure cloud. There is no need for user administration, so the unit of operation is a subscription with a unique name. This way Azure Data Grid can be shared among many users without having to let Microsoft know about it.

However, another scenario is possible without creating a web interface at all.

Azure Event Grid allows you to control the level of access given to different users to do various management operations such as list event subscriptions, create new ones, and generate keys. Event Grid uses Azure role-based access control (Azure RBAC).

Read more about Authorizing access to Event Grid

<https://docs.microsoft.com/en-us/azure/event-grid/security-authorization>

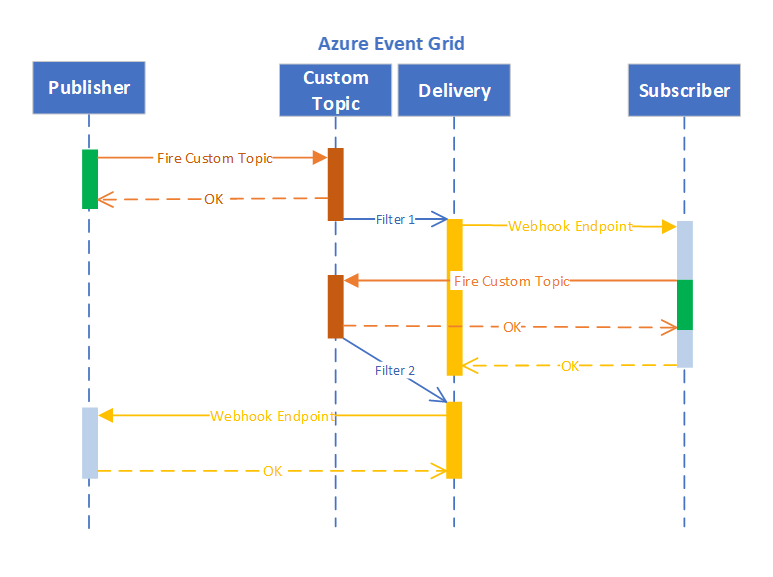
## **Webhook Event delivery**

In order to deliver events to end customers we can develop own delivery solution or publish events Azure Event Grid (do not confuse with costly Azure Data Hub)

| Feature | Azure Data Grid | Custom |
| --- | --- | --- |
| BackEnd Development | Not necessary | Necessarily |
| Additional costs | 0.60 USD per one million messages | Free |

### **With Azure Event Grid**

Use Event Grid to push tons of events to subscribers using WebHooks and filter the pushed message by specifying event types. (see API)

**

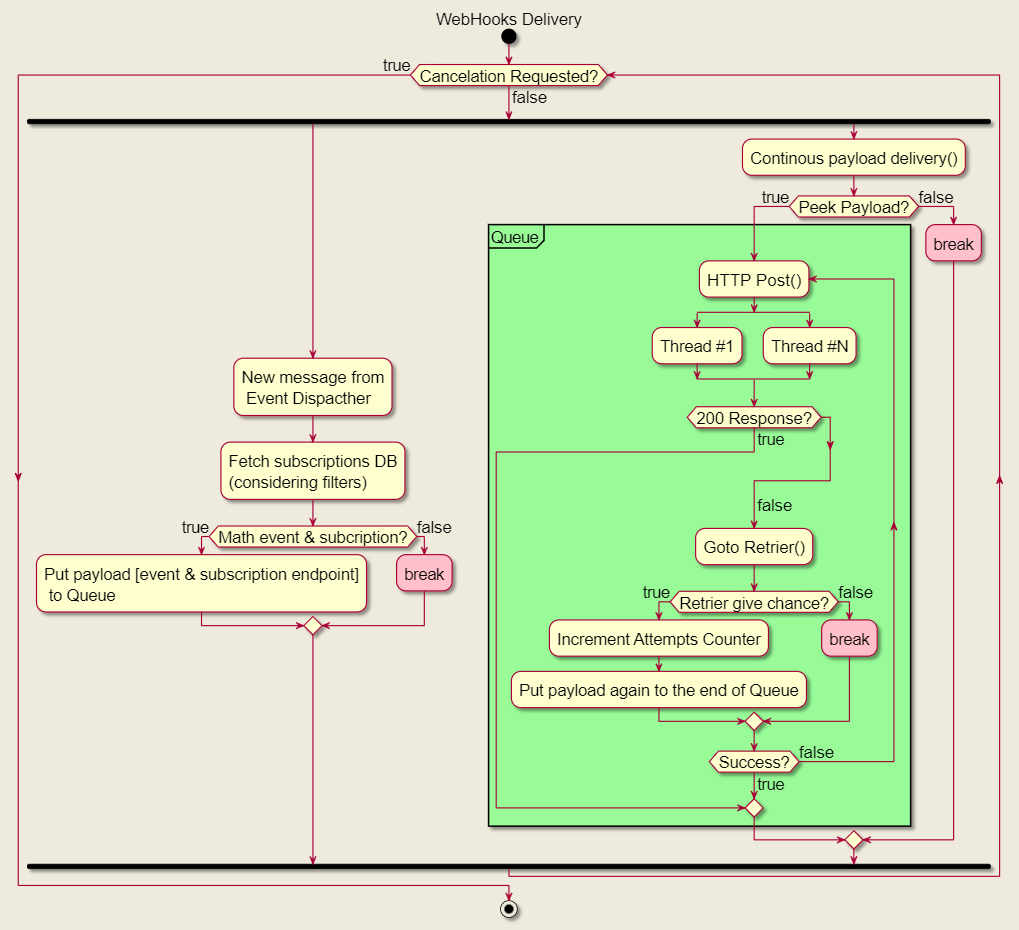
Read more about publishing to Azure Event Grid

<https://docs.microsoft.com/en-us/azure/event-grid/edge/pub-sub-events-webhook-cloud>

### **With Custom solution**

| In the case of using your own solution for delivering webhooks to end users.   * Develop our own delivery subsystem * Store the subscriptions database on our side or synchronize it with Azure |
| --- |

This diagram shows how the delivery subsystem works after Event Dispatcher pushed it with a new event.



## **Prototype**

**Kafka -> Http Sink Connector -> Azure Event Grid -> External Subscribers**

Without custom coding, we get the webhook system consists of webhook dispatcher powered by Kafka and subscription management powered by Azure Data Grid

1. Go to Kafka Control Center <http://167.86.107.116:9021/clusters>
2. Publish an event to <<\_\_mdp>> topic via web interface or Kafka Client

| [{  "subject": "reservation",  "eventTime": "2022-05-13T06:00:56",  "eventType": "created",  "id": 123,  "dataVersion": "1.0",  "data": {  "firstName": "Ada Byron"  }}] |
| --- |
| bootstrap.servers=167.86.107.116:9092  security.protocol=SASL\_SSL  sasl.mechanisms=PLAIN  zookeeper: 167.86.107.116:2181, cluserName: controlcenter.cluster  No username/password |

.

1. Examine Http Sink Connector’s settings (home-> controlcenter.cluster -> connect clusters -> connect-default -> connectors -> http sync) You will find the endpoint of Azure Data Grid as the http.url property of the connector.

| POST https://mdptopic.westeurope-1.eventgrid.azure.net/api/events  Headers:  Aeg-sas-key: 2vn2IdoRgG/2a1QNKUdylnThQW1SZ8lwEKq0lawoxDk= |
| --- |

1. Go to one of many external subscriber hook receivers and find your event. Or look to <<\_\_success>> & <<\_\_error>> topics.

| <https://mdpwebhooksite.azurewebsites.net/> |
| --- |

Use following to restart system

| SSH 167.86.107.116. Login as root:abc12345!  > reboot  >./start.sh |
| --- |

1. Goto Azure and administer end customer subscriptions

<https://docs.microsoft.com/en-us/azure/event-grid/edge/pub-sub-events-webhook-cloud>

### **Install & Setup**

Azure Event Grid

* Create Resource Group

<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-porta>

* Create Event Grid and Topics

<https://docs.microsoft.com/en-us/azure/event-grid/create-view-manage-system-topics>

* Create Subscription

<https://docs.microsoft.com/en-us/azure/data-explorer/ingest-data-event-grid-manual>

* Generate access keys for Event Grid

<https://docs.microsoft.com/bs-latn-ba/azure/event-grid/get-access-keys>

Http Sink Connector

* Install

| confluent-hub install confluentinc/kafka-connect-http:latest |
| --- |

* Export Azure Data Grid certificate to Java KeyStore file

| keytool -printcert -sslserver mdptopic.westeurope-1.eventgrid.azure.net:443 -rfc | keytool -import -noprompt -alias nm -keystore /crun/file -storepass changeit    where   * host[:port] - Azure Topic EndPoint (e.g. mdptopic.westeurope-1.eventgrid.azure.net) * Java keyStore File (e.g. /crun/file) |
| --- |

* Configure worker file http-sink.json

| {  "name": "HttpSink",  "config": {  "topics": "\_\_mdp",  "tasks.max": "1",  "connector.class": "io.confluent.connect.http.HttpSinkConnector",  "http.api.url": "{AZURE\_TOPIC\_ENDPOINT}",  "value.converter": "org.apache.kafka.connect.storage.StringConverter",  "key.converter": "org.apache.kafka.connect.storage.StringConverter",  "headers": "aeg-sas-key:{AZURE\_ACCESS\_KEY}",  "header.separator": ";",  "https.ssl.key.password": "changeit",  "https.ssl.keystore.location": "{KEYSTORE\_FILE}",  "https.ssl.keystore.password": "changeit",  "https.ssl.truststore.location": "{KEYSTORE\_FILE}",  "https.ssl.truststore.password": "changeit",  "https.ssl.keystore.type": "JKS",  "https.ssl.protocol": "TLSv1.2",  "https.ssl.truststore.type": "JKS",  "reporter.result.topic.name": "\_\_success",  "reporter.result.topic.replication.factor": "1",  "reporter.error.topic.name": "\_\_error",  "reporter.error.topic.replication.factor": "1",  "reporter.bootstrap.servers": "localhost:9092",  "confluent.topic.bootstrap.servers": "localhost:9092",  "confluent.topic.replication.factor": "1",  "ssl.enabled": "true"  }  } |
| --- |

* Start & check status

| confluent local services connect connector load HttpSink --config http-sink.json  confluent local services connect status  confluent local services connect connector status HttpSink |
| --- |

Read more about Http Sink Connector

<https://docs.confluent.io/kafka-connect-http/current/overview.html>

## **Conclusion**

The above reasons make **Azure Event Grid** the best choice for MDP WebHook Proposal and especially note that it is better to take a ready-made solution at the first step.

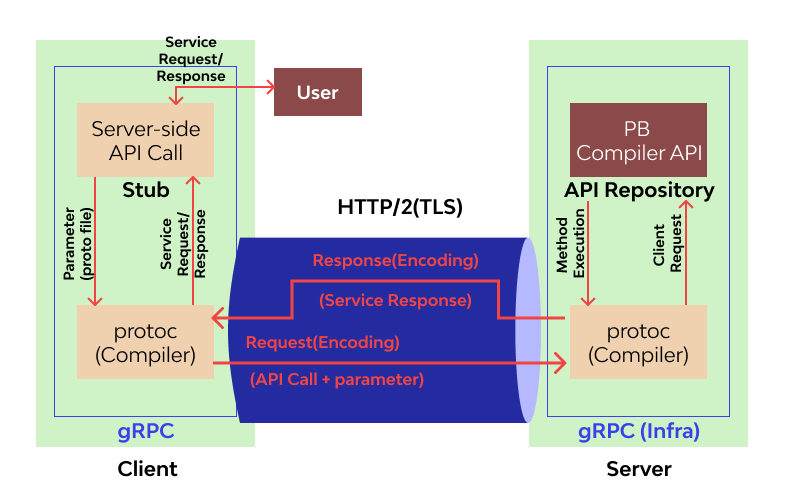
The above reasons make **Kafka Trigger or Sink Connector** are best choices for MDP WebHook Proposal and especially note that Kafka Trigger is extremely easy to be managed only on ‘C#’ side there are many examples of usage of Sink Connector as a webhook solution over the internet.

# MicroServices Connectivity Proposal

## **Interaction 1 to1**

| Feature | gRPC | REST |
| --- | --- | --- |
| HTTP 1.1 vs HTTP 2 | Follows a client-response model of communication and is built on HTTP 2, which allows for: streaming communication and bidirectional support. | Follows a request-response model of communication and is typically built on HTTP 1.1. |
| Browser Support | Limited browser support. gRPC requires gRPC-web and a proxy layer to perform conversions between HTTP 1.1 and HTTP 2. | Universal browser support. |
| Payload Data Structure | gRPC uses Protocol Buffer by default to serialize payload data. | REST mainly relies on JSON or XML formats to send and receive data. |
| Code Generation Features | gRPC has native code generation features. | Developers must use a third-party tool like Swagger or Postman to produce code for API requests. |

gRPC provides plenty of advantages. Unlike REST, it can make the most out of HTTP 2, using **multiplexed streams** and following the **binary protocol**. Plus, it offers performance benefits due to the Protobuf message structure, and let's not forget the in-built code generation features which enable a multilingual environment.



Read more about gRPC vs REST: comparing APIs architectural styles

[**https://www.imaginarycloud.com/blog/grpc-vs-rest**](https://www.imaginarycloud.com/blog/grpc-vs-rest)

## **Interaction via Broker**

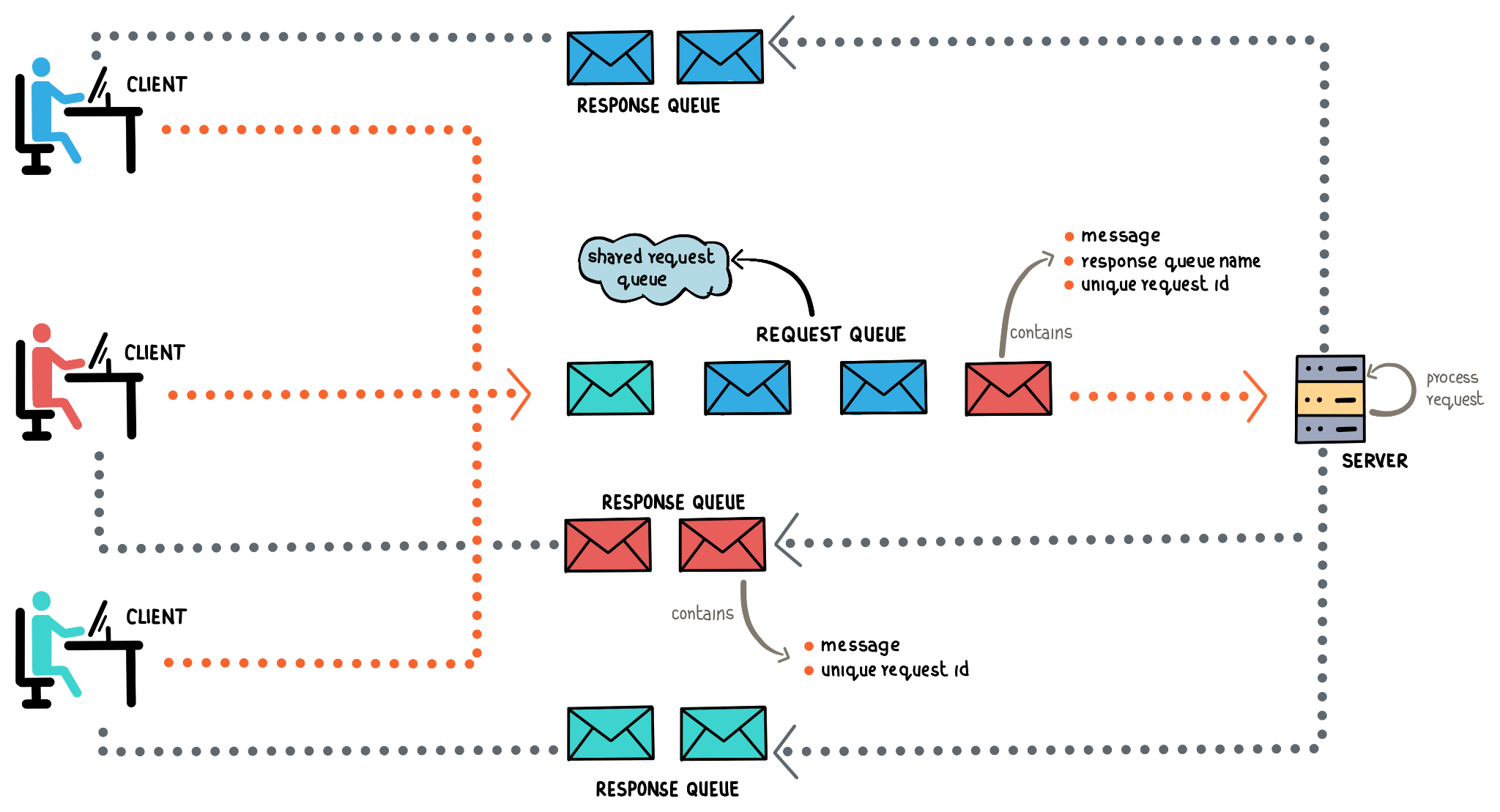
| Feature | Kafka | RabbitMQ |
| --- | --- | --- |
| Message retention/deletion | Messages are deleted once the retention period is over. | Right after consumers receives the message or finishes processing and saving the data message is deleted |
| Message replay | Yes, possible under the retention period | No, since messages are deleted off the queue promptly after delivery |
| Routing Available | Doesn’t have routing algorithms/rules. Topics define the necessary segregation | Comes with many complex routing rules. Exchange and binding is used to push messages to appropriate queues |
| Priority | No in-built priority feature. | Priority queues are implemented in place |
| Filtering message | All the messages in the subscribed topic are sent to the consumer. | Message-headers and topic-exchange allow the consumer to be selective in receiving specific messages only |
| Maintaining sequential Order | Kafka maintains an offset to keep the order of arrival of messages intact. | RabbitMQ implicitly uses Queue that follows the FIFO property and thus keeps proper order of messages. |

Kafka's message retention makes it incline towards a message logger, or a huge and fast data streaming server. Since the onus is on the consumer to retry for a message after a failure, Kafka doesn't mind if messages are delivered successfully or not. This relieves it of extra implementation and focus is put on data replaying and querying.

RabbitMQ's lack of retention of messages and guarantee of acknowledgment messages from consumers makes it a better fit for being an application mediator, a robust message broker.

### **High level async 2-way architecture with AMQP**

RabbitMQ’s pipe is **unidirectional**. You cannot send messages from the consumer to the producer through the same queue that the consumer received messages from the producer. But The request/response pattern is well-known and widely used, mainly in synchronous communication.The important things are: We have two channels. One for requests and one for responses. We use a correlation ID on both ends of the communication. Another point we have to note is that the client has to have a state. The client generates a unique correlation ID, for example, my unique id. Then the client sends the request to the channel and keeps the correlation ID in memory or in a database. After that, the client waits for the responses in the response channel. Every response from the channel has a correlation ID, and the client has to compare this correlation ID with those in memory to find the respective request and proceed with processing the response in the context of that request. This diagram shows how to implement this pattern asynchronously with a message broker using the AMQP protocol.

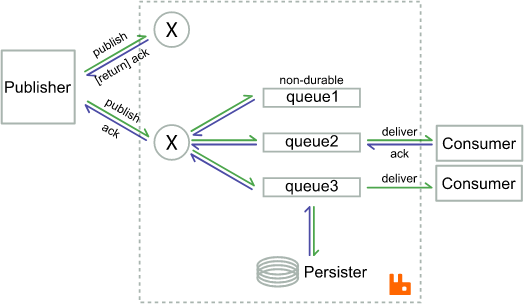


Read more about Asynchronous Request/Response Pattern with AMQP

[**https://www.projectpro.io/article/kafka-vs-rabbitmq/451**](https://www.projectpro.io/article/kafka-vs-rabbitmq/451)

### **High level acknowledge based architecture with AMQP**

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it. If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly deliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.



Read more about consumer acknowledgements and publisher Confirms

<https://www.rabbitmq.com/confirms.html>

Read more about Kafka vs RabbitMQ - A Head-to-Head Comparison

[**https://www.projectpro.io/article/kafka-vs-rabbitmq/451**](https://www.projectpro.io/article/kafka-vs-rabbitmq/451)

## **Conclusion**

The above reasons don't make **gRPC or REST** the best choice for MDP WebHook Proposal.

The above reasons make **RabbitMQ broker** the best choice for MDP WebHook Proposal in comparison with Kafka broker.

## **Notes**

***Solution A. “Azure Data Lake Storage Gen2 Sink Connector”***

**Step 1.** Export data from Apache Kafka® topics to Azure Data Lake Gen2 files in JSON format.

<https://www.confluent.io/hub/confluentinc/kafka-connect-azure-data-lake-gen2-storage>

**Step 2.** Integrate events coming from Data Lake with Azure Event Grid.

<https://azure.microsoft.com/en-au/blog/event-driven-analytics-with-azure-data-lake-storage-gen2/>

**Step 3.** Once the event is in the Azure Event Grid, we can manage subscriptions using the it’s infrastructure and route events to end customers.

<https://docs.microsoft.com/en-us/azure/event-grid/edge/pub-sub-events-webhook-cloud>

***Solution B. “Tunnel”***

**Step 1.** Tunnel an existing Kafka pipeline into Azure by "mirroring" the Kafka input stream in the Event Hubs service.

<https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-kafka-mirror-maker-tutorial>