



Do what matters

Distributed, Event-Driven systems deep-dive

Example design with Event Grid and Event Hub

Session 3 (1 hour). Example design with Event Hub and Event Grid.

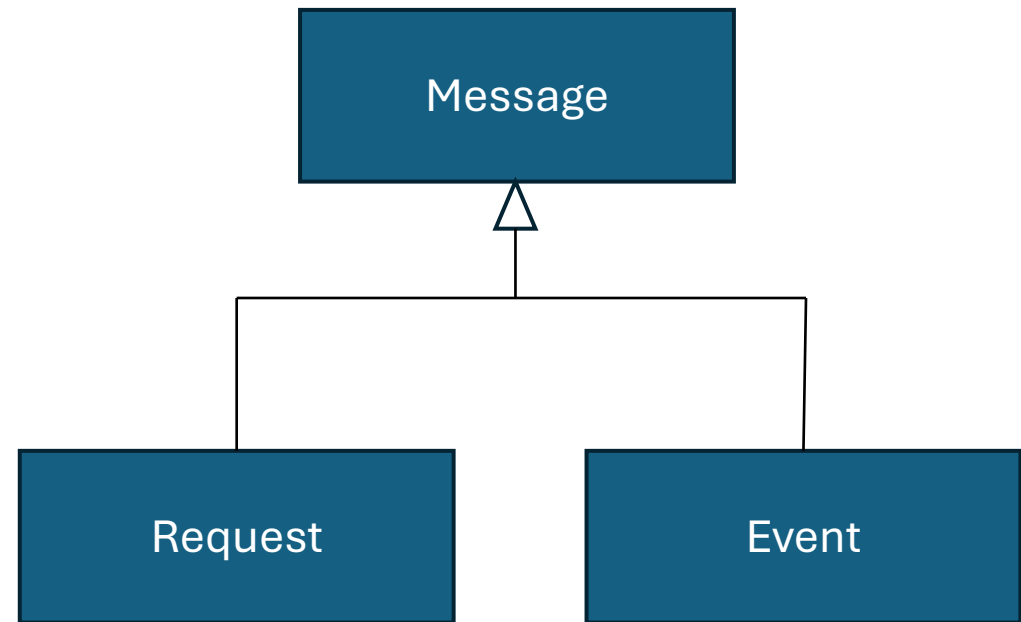
Quick recap on the theory from the previous sessions, with demo of the example design and discussion of the design choices, together with a set of recommendations and lessons

Agenda

- Recap of preceding sessions (Event Grid, Event Hub)
- Introduction to this session's PoC
- Demo
- Design Rationale
- Lessons Learned

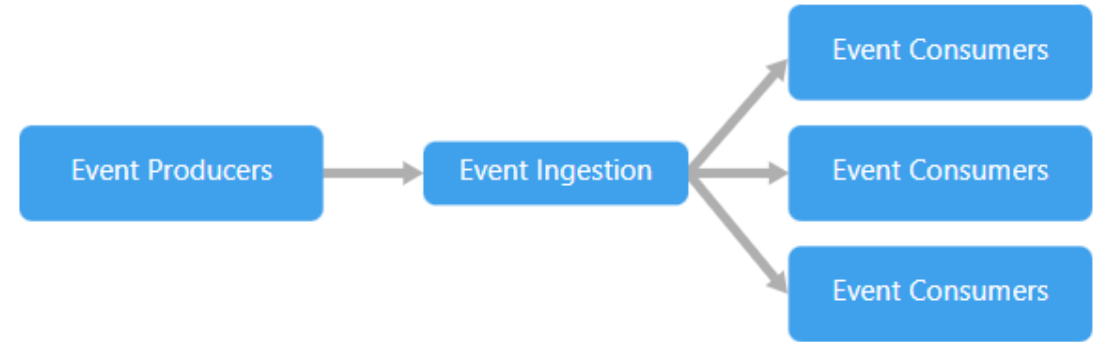
What is an Event?

- A *Message* is a payload of data sent from one system to another.
- A *Request* is a type of message that asks for an action to be performed (in the future).
 - E.g., “Please process the data”
- An *Event* is a type of message that notifies something has happened (in the past).
 - E.g., “The processing is complete”



Event-Driven Architecture

- An event-driven architecture consists of **event producers** that generate a stream of events, and **event consumers** that listen for the events.
- Events are delivered in near real time, so consumers *can* respond immediately to events as they occur.
- Producers are decoupled from consumers — a producer doesn't know which consumers are listening.
- Consumers are also decoupled from each other, and every consumer sees all of the events.



Publish-Subscribe vs. Event Streaming

Pub-Sub	Event Streaming
The messaging infrastructure (<i>Event Ingestion</i>) keeps track of subscriptions.	Events are written to a log. Events are strictly ordered (within a partition) and durable.
When an event is published, it sends the event to each subscriber (<i>Consumer</i>).	Clients (<i>Consumers</i>) don't subscribe to the stream, instead a client can read from any part of the stream.
After an event is received, it can't be replayed, and new subscribers don't see the event	The client is responsible for advancing its position in the stream. That means a client can join at any time, and can replay events.

Request-Response vs. Pub-Sub



Event Driven Architectural Patterns

EDA USE CASES:

- State persistence
- Data distribution
- Notifications

EVENT NOTIFICATION

```
public sealed class EventPublishedDomainEvent : DomainEvent {  
    public Guid EventId { get; init; }  
}
```

EVENT-CARRIED STATE TRANSFER

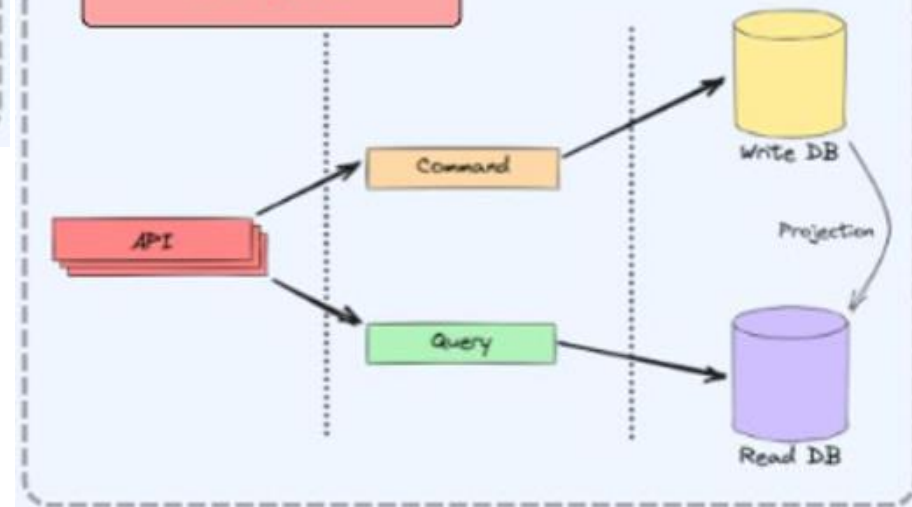
```
public sealed class EventPublishedIntegrationEvent : IntegrationEvent {  
    public Guid EventId { get; init; }  
    public string Title { get; init; }  
    public string Description { get; init; }  
    public string Location { get; init; }  
    public DateTime StartsAtUtc { get; init; }  
    public DateTime? EndsAtUtc { get; init; }  
    public List<TicketTypeModel> TicketTypes { get; init; }  
}
```

EVENT SOURCING

Order placed → Ticket added → Ticket added → Ticket removed → Order paid

Milan Jovanović, LinkedIn: “Patterns of Event-Driven Architecture”

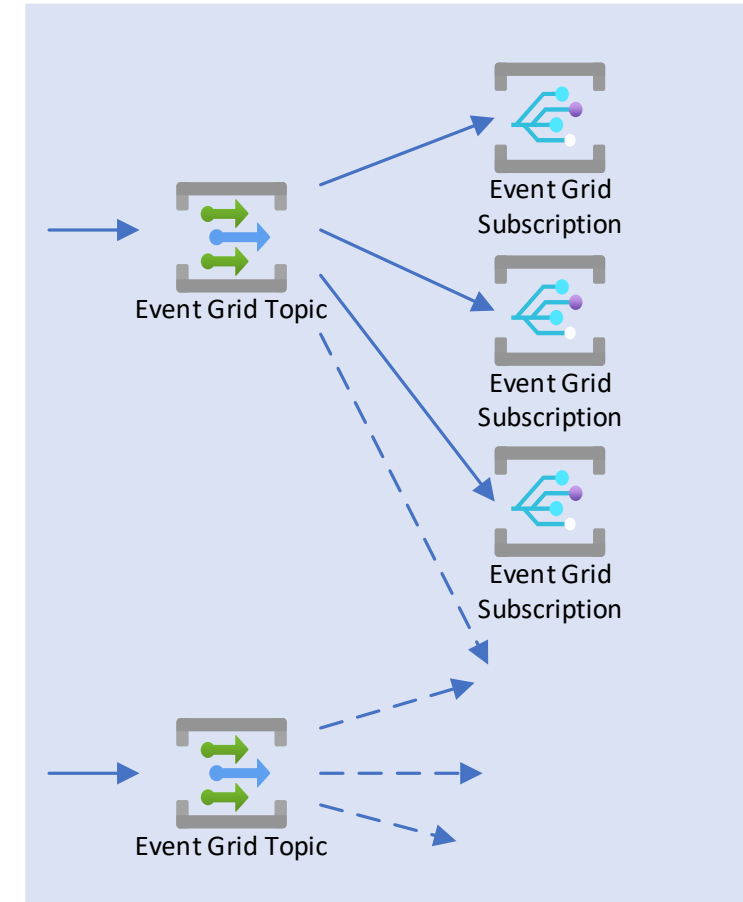
CQRS



Azure Event Grid: Recap

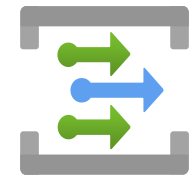
Event Grid (1/2)

- Event Publishers raise events in Event Grid TOPICS
- Subscribers listen on a SUBSCRIPTION to a TOPIC
 - Event Grid can PUSH events to subscribers
 - (or Subscribers can PULL, June 2023)
- Event Grid filters and routes events
 - Built-in delivery retry mechanism
- Highly-scalable, high throughput (MQTT, HTTP)
- Security: Access Control, Certificates, TLS support



Event Grid (2/2)

- Many use cases; e.g.
 - Ingest multiple data sources (many-to-one, fan-in)
 - Broadcast alerts (one-to-many, fan-out)
- Many event handlers (subscriber types)
 - Webhook, Azure Function, Event Hub, Service Bus queues and topics, Relay hybrid connections, Storage queues
- “*Pay-per-event*” pricing model
 - First 100,000 operations per month are free
 - Basic tier: £0.467 per million operations
 - “Operations in Event Grid include all ingress events, advanced match, delivery attempt, and management calls. You’re charged per million operations with the first 100,000 operations free each month.”

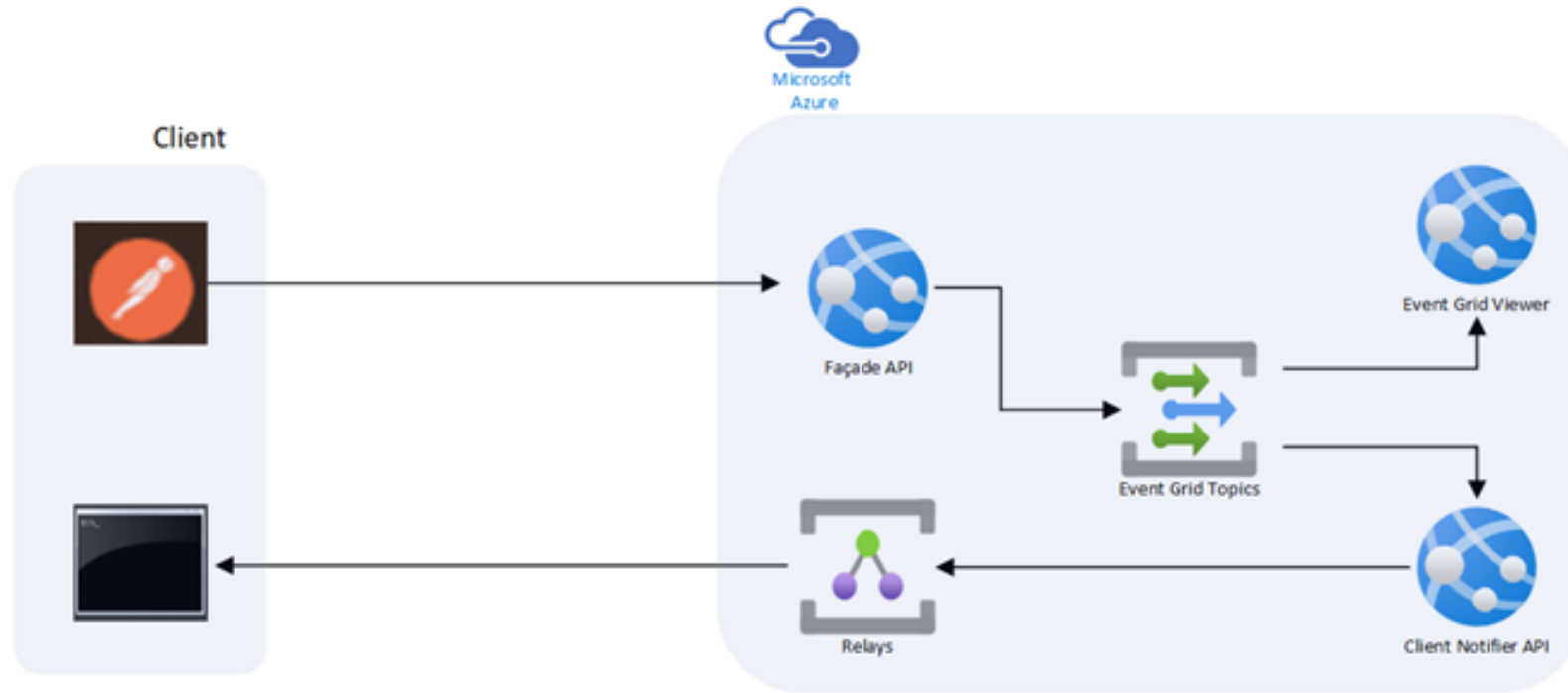


Event Grid Topics



Event Grid
Subscriptions

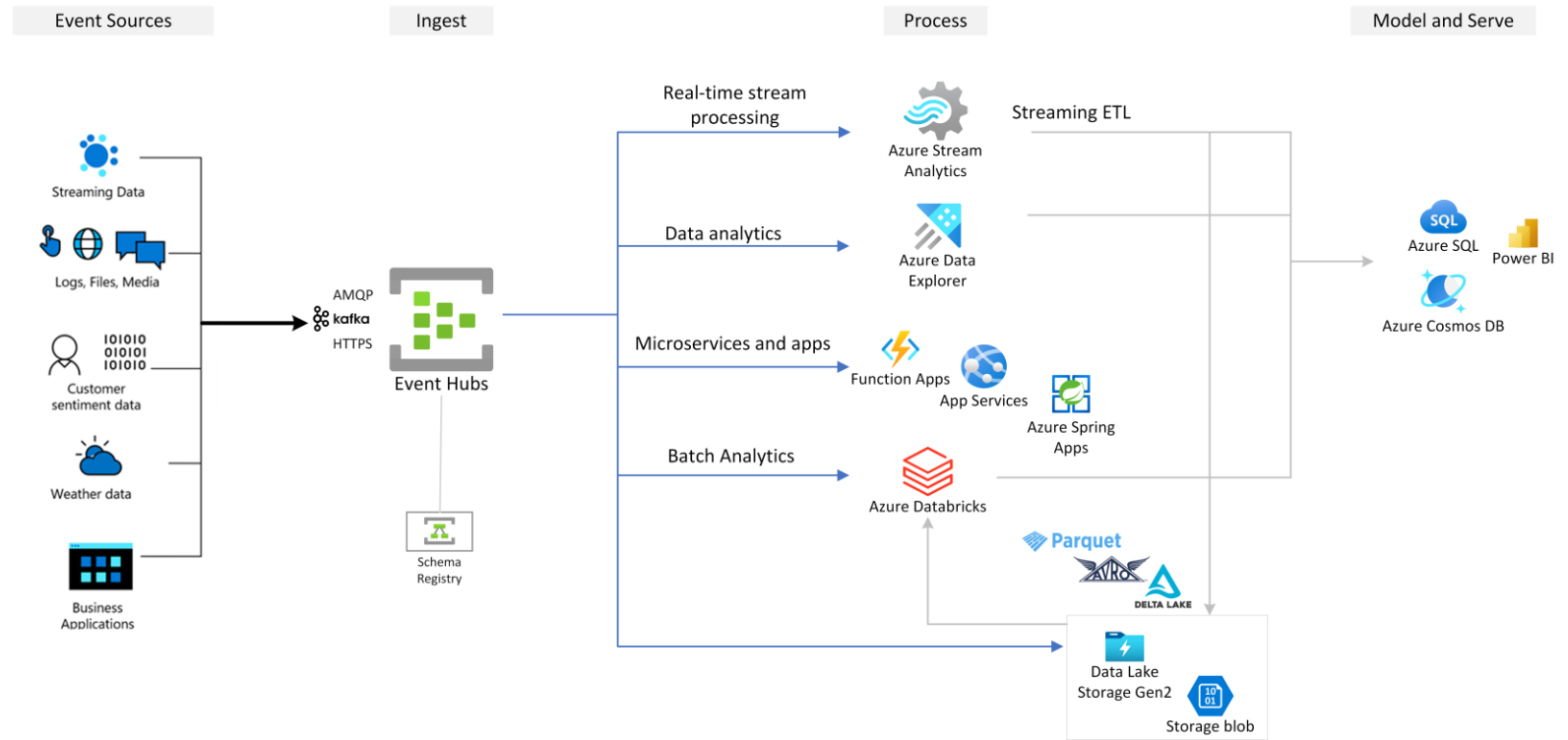
Event Grid PoC components



Azure Event Hub: Recap

Event Streams

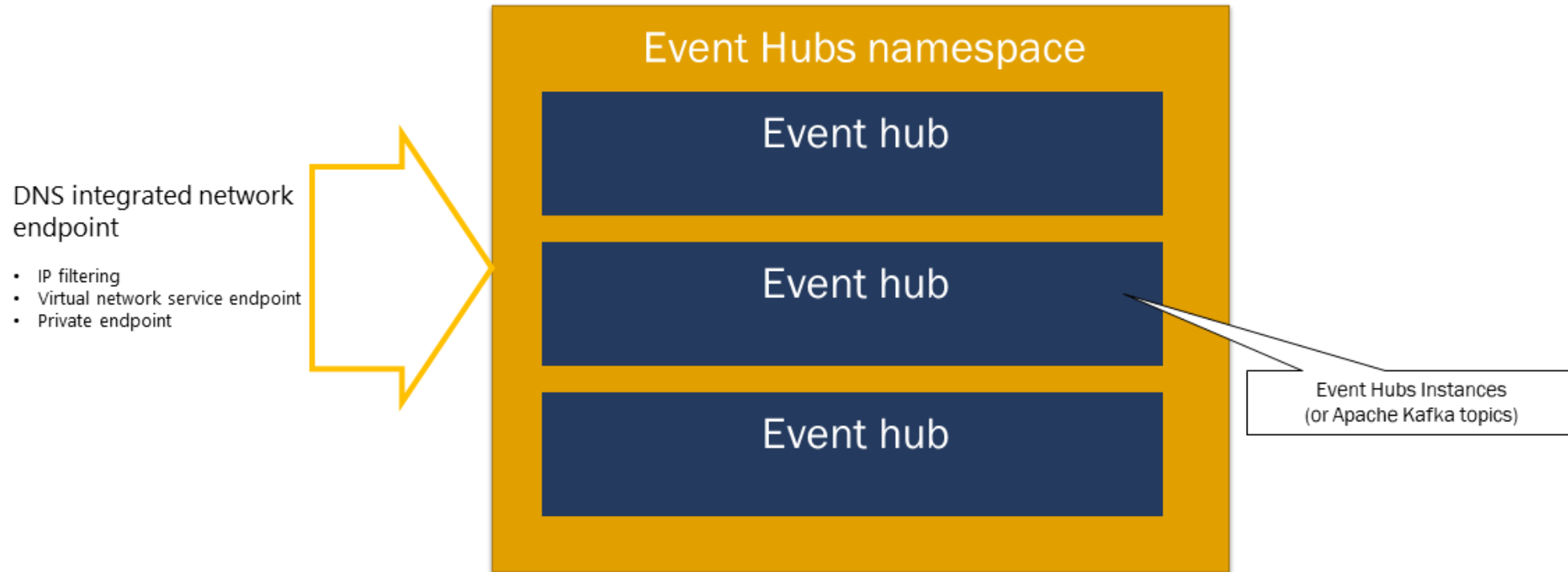
- Event Streams can be processed continuously, or in batch mode
 - An event can be processed when it is published (almost immediately)
 - A client can index through a stream of logged events (retrospectively)



<https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about>

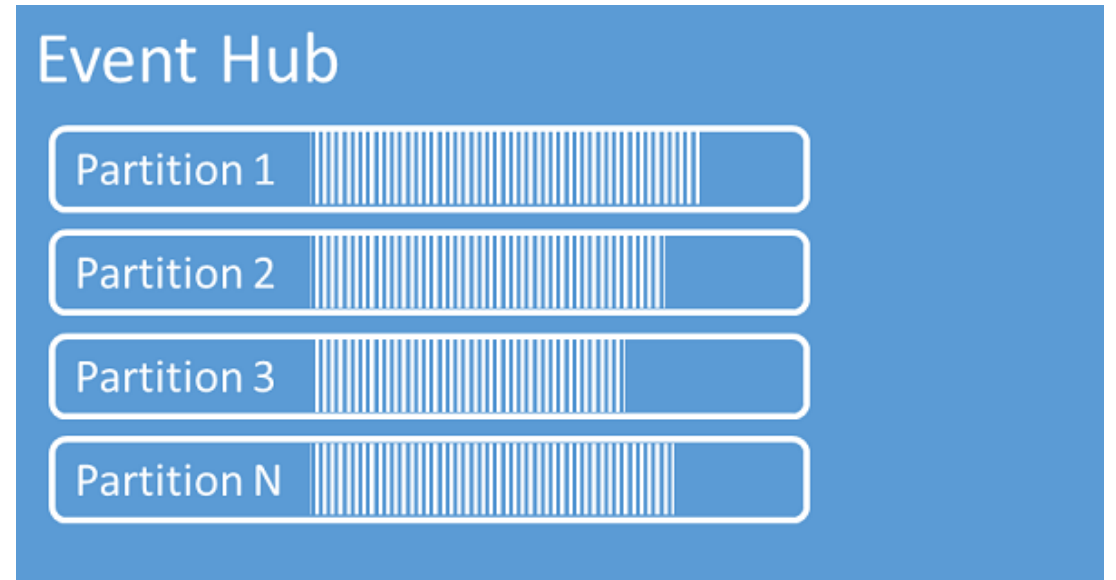
Event Hubs Namespace

- Namespace: management container for Event Hubs integration management features



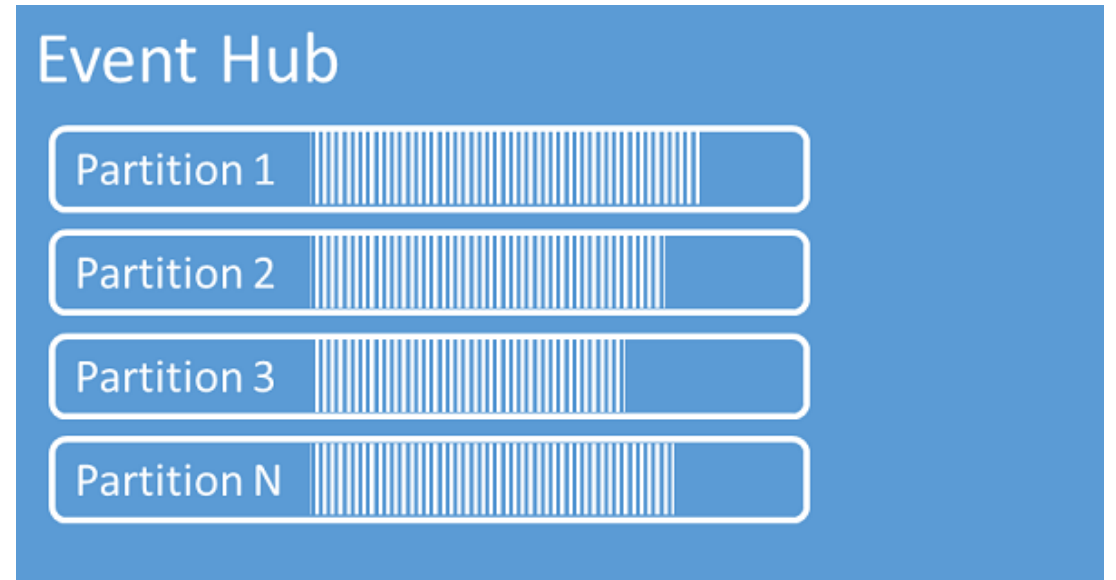
Event Hubs Partitions

- Partitions aid throughput.
- Event Hub organises received events into one or more partitions. As newer events arrive, they're added to the end of this sequence.
- A partition can be thought of as a commit log. Partitions hold event data that contains the following information:
 - Body of the event
 - User-defined property bag describing the event
 - Metadata such as its offset in the partition, its number in the stream sequence
 - Service-side timestamp at which it was accepted



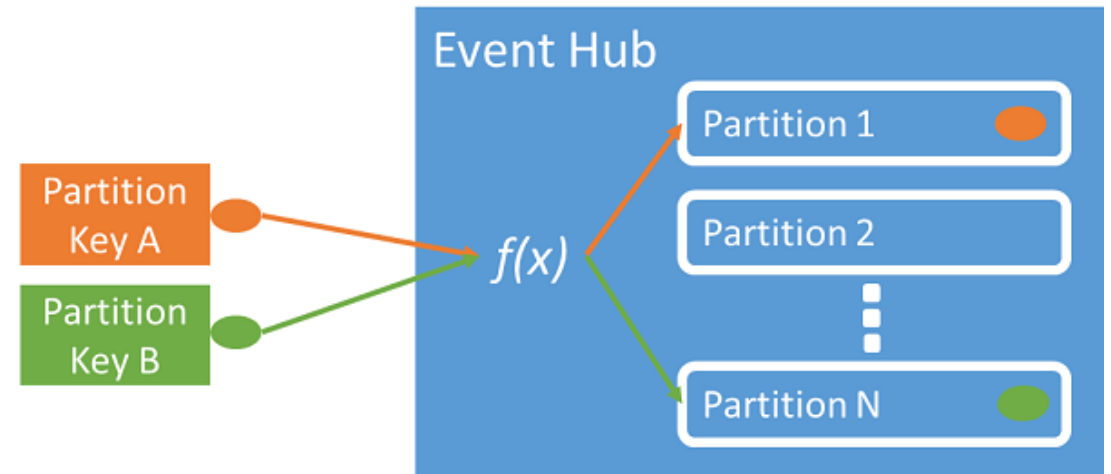
Event Hubs Partitions: Considerations

- The number of partitions is specified at the time of creating an event hub.
 - Pricing is not dependent on the number of partitions
 - Maximum number is tempting – but can make processing more complex
 - Choose enough to meet the peak load of your application for that particular event hub.
 - Number *can* be increased for premium, dedicated tiers
 - Changes event distribution!



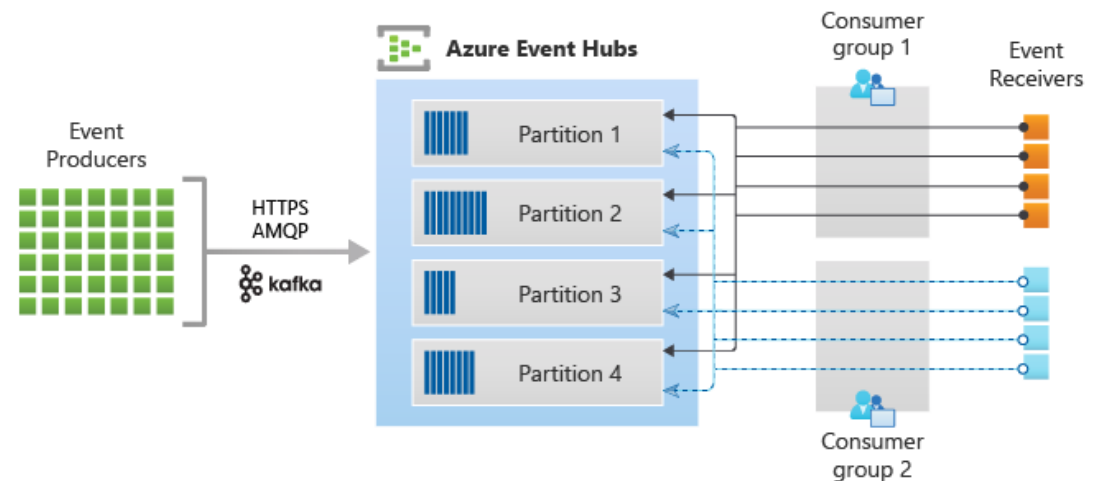
Publishing Events to Partitions

- *Can* publish direct to a specific partition, but not recommended.
- Partition Key is sender-supplied, hashed for a mapping.
 - Keeps related events together, in order
- In absence of a partition key, round-robin assignment applied



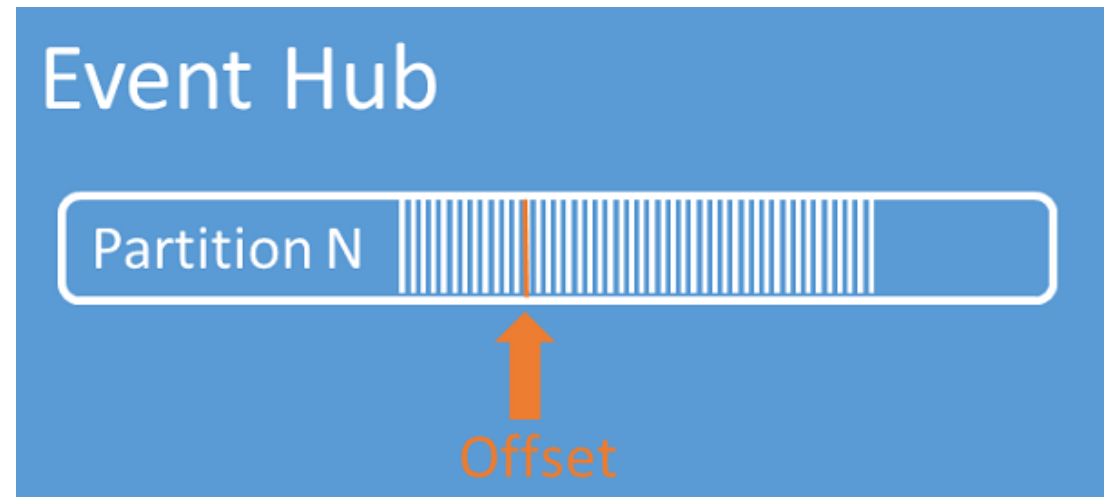
Consuming events

- Any entity that reads event data from an event hub is an *event consumer*.
 - All Event Hubs consumers connect via the AMQP 1.0 session and events are delivered through the session as they become available. (The client doesn't need to poll for data availability)
- A *consumer group* is a logical grouping of consumers
 - Consumer Groups enable multiple consuming applications to read the same streaming data in an event hub independently at their own pace with their offsets.



Offsets and Checkpointing

- An *offset* is the position of an event within a partition
 - Effectively, a client-side cursor
 - Enables an event consumer to specify a point from which to begin reading events
- *Checkpointing* is a process by which readers mark or commit their position within a partition event sequence.
 - If a reader disconnects from a partition, when it reconnects it begins reading at the checkpoint that was previously submitted
 - Checkpointing is the responsibility of the consumer and occurs on a per-partition basis within a consumer group.

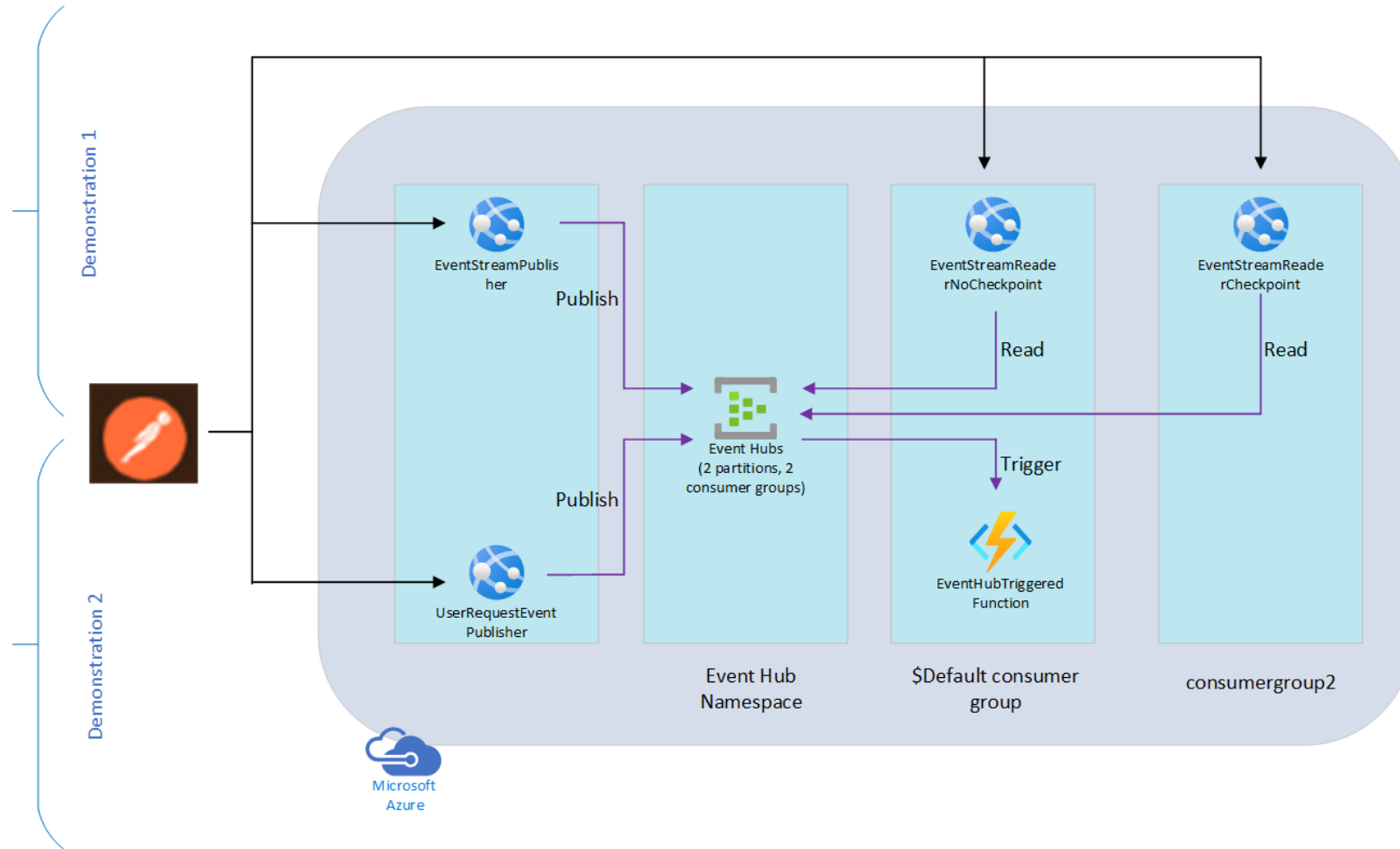


Throughput Units (TU)

- Throughput units are pre-purchased units of capacity. A single throughput unit:
 - Ingress: Up to 1 MB per second or 1,000 events per second (whichever comes first).
 - Egress: Up to 2 MB per second or 4,096 events per second.
- Beyond the capacity of the purchased throughput units:
 - Ingress is throttled and Event Hubs throws a ServerBusyException.
 - Egress doesn't produce throttling exceptions but is still limited to the capacity of the purchased throughput units.
- Throughput units are billed per hour for a minimum of one hour.
 - Up to 40 throughput units can be purchased for an Event Hubs namespace and are shared across all event hubs in that namespace
- The Auto-inflate feature of Event Hubs automatically scales up by increasing the number of throughput units, to meet usage needs.

<https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-scalability#throughput-units>

Event Hub Demo components



PoC: Event Hub and Event Grid

Purpose

What it is:

- Example of a realistic combination of Azure Event Hub and Azure Event Grid to provide pub-sub operation
- Builds on what was introduced in the previous sessions
- Introduces design considerations and lessons learned

What it is *not*

- A full implementation: uses faked databases, ignores security, etc.

PoC scenario

- A user requests a “buy” or “sell” transaction for a specified quantity of a stated stock (where the operation will be performed asynchronously).
- The system will first:
 - Assign a Request ID, and record this with the user’s name and transaction details
 - Return a “location” (defined by the Request ID) where the user can GET the results, when they become available
 - The user is now unblocked
- Then:
 - Gather parameters specific to the user that will govern the requested processing. This processing configuration will be recorded.
 - Apply the processing in accordance with the configuration and record the results.
 - Notify the client with a two-element message:
 - The Request ID and the binary representation of the success of the processing

PoC Components and Flow

Façade API: receives, validates and persists the user request with the user name and a generated **RequestID**.

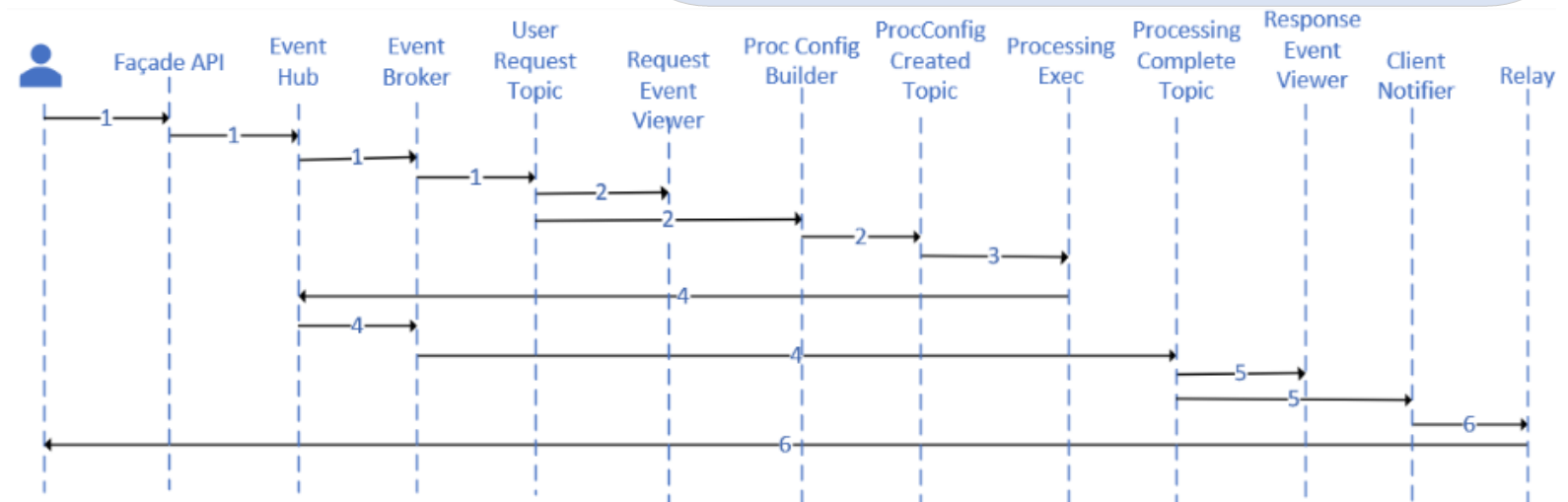
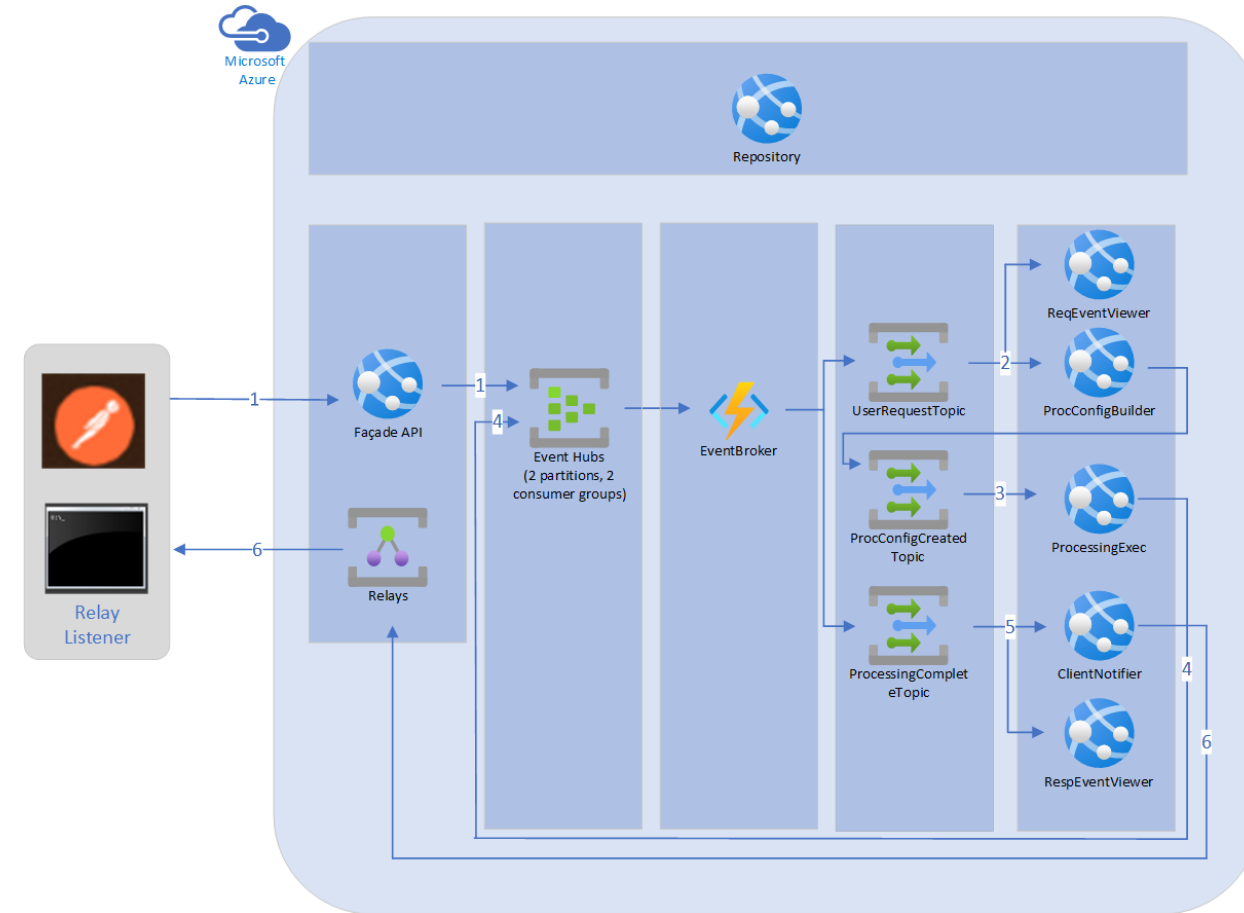
ProcConfigBuilder: combines the user request with the governing parameters for processing and persists as the “ProcConfig”.

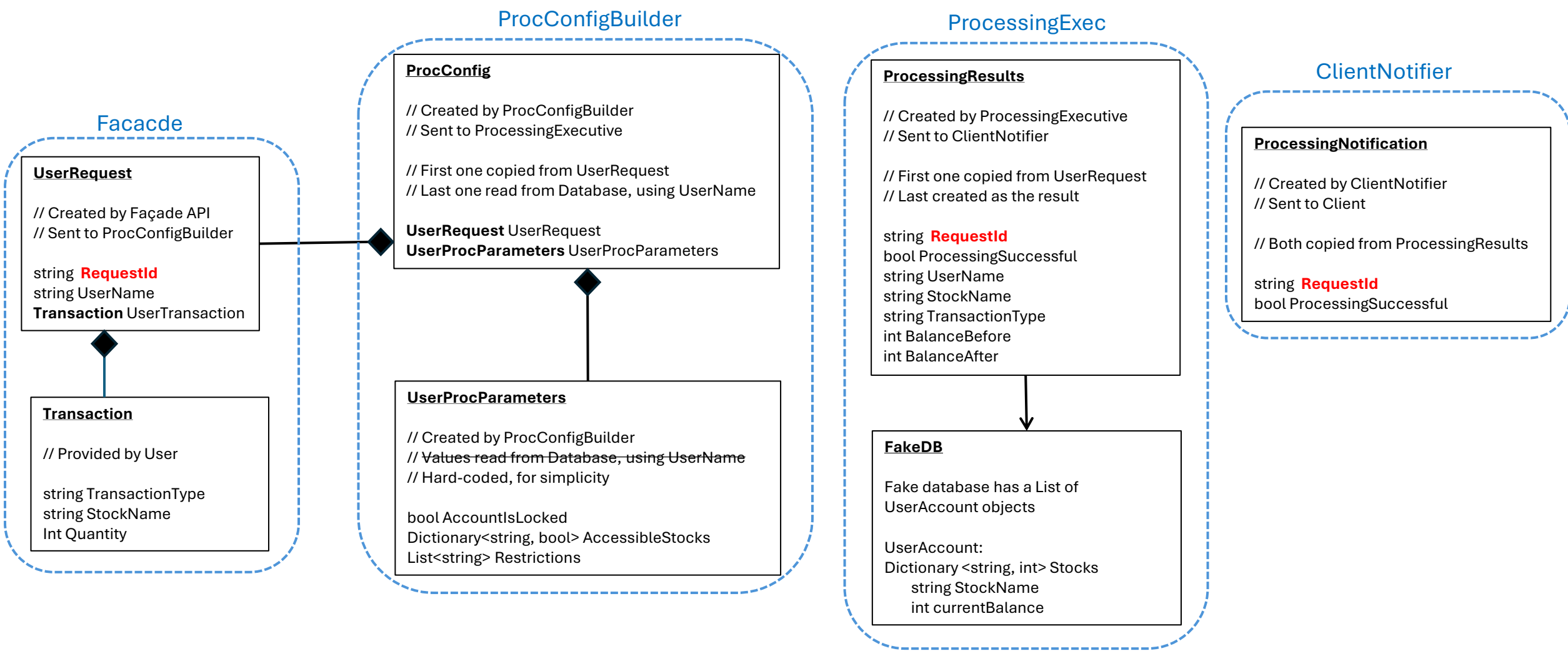
ProcessingExec: applies the processing in accordance with the ProcConfig, persists the Processing Results.

ClientNotifier: notify the user through the Azure Relay with the RequestID and the binary indication of the success of the processing.

Event Broker: forwards recognised events

ReqEventViewer, RespEventViewer: display events on two Event Grid Topics.





Repository

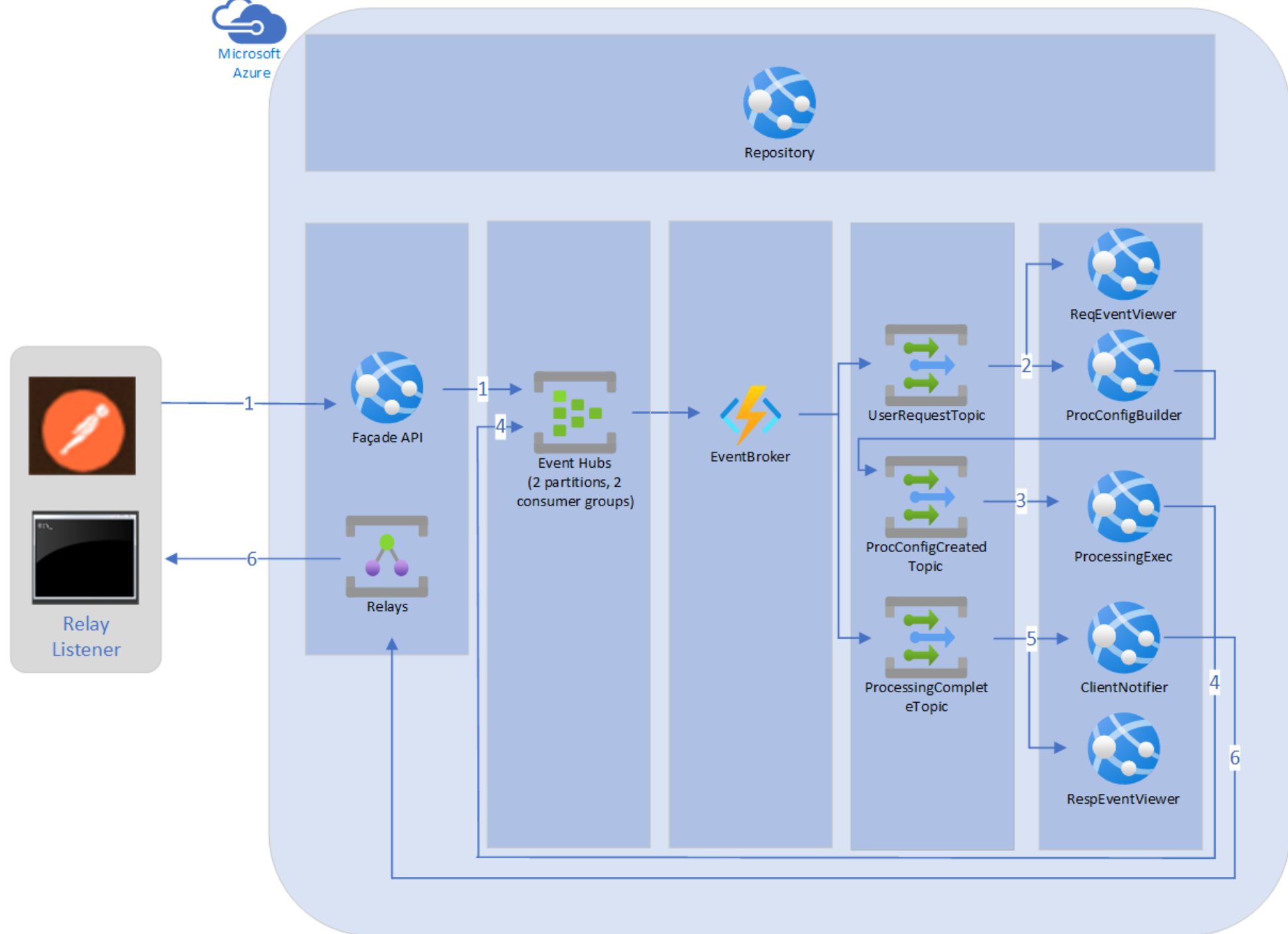
Has 3 Dictionaries, where the key is the **RequestId**

UserRequests<string, **UserRequest**>

ProcConfigs <string, **ProcConfig**>

ProcessingResults<string, **ProcessingResults**>

Information views in the PoC



Postman

Postman is used to issue the POST request to the system.



- POST is sent to the FacadeAPI
- Response will be either:
 - 500 for System Error, or
 - 202 for Acceptance of the request, with a header “Location” specifying where the results can be accessed (HTTP GET) when they are available.


The screenshot displays a Postman interface for a POST request. The URL bar shows the endpoint: `https://facadeapi20240725140029.azurewebsites.net/api/userrequest`. The request body is a JSON object: `{ "transactionType": "buy", "stockName": "MSFT", "quantity": 10 }`. The response status is 202 Accepted, with a time of 5.87 s and a size of 537 B. The response headers are listed in a table below.

Header	Value
Content-Length	0
Date	Wed, 07 Aug 2024 16:38:00 GMT
Server	Microsoft-IIS/10.0
Location	api/Facade/dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598
Set-Cookie	ARRAffinity=0cfafa747c9d8df8999415eede91369b1b2215c6e1cd987c2eef91e92a699...
Set-Cookie	ARRAffinitySameSite=0cfafa747c9d8df8999415eede91369b1b2215c6e1cd987c2eef9...
X-Powered-By	ASP.NET



ReqEventViewer, RespEventViewer


- Software from a 3rd Party, provides UI with a grid of events received on a specific Event Grid Topic (“UserRequestTopic” and “ProcessingCompleteTopic”, respectively)

 **Azure Event Grid Viewer: Requests** 

Event Type	Subject
 UserRequestEvent	dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598

```
{
  "id": "bf203453-18ac-4a00-b746-263823861117",
  "subject": "dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598",
  "data": {
    "RequestId": "dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598",
    "UserName": "fake name",
    "UserTransaction": {
      "TransactionType": "buy",
      "StockName": "MSFT",
      "Quantity": 10
    }
  },
  "eventType": "UserRequestEvent",
  "dataVersion": "1.0",
  "metadataVersion": "1",
  "eventTime": "2024-08-07T16:38:00.6504253Z",
  "topic": "/subscriptions/d9239969-5904-41d6-a307-fc5df187d7e1/resourceGroups/adh_EventDrivenExample_rg/providers/Microsoft.EventGrid/topics/UserRequestTopic"
}
```

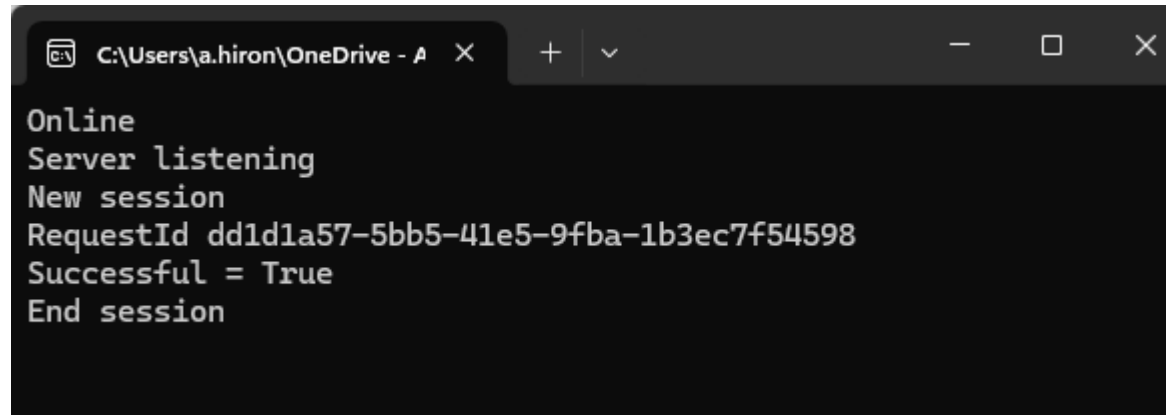
 **Azure Event Grid Viewer: Responses** 

Event Type	Subject
 ProcessingCompleteEvent	dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598

```
{
  "id": "f6218b69-cb57-4395-ad58-17897d0021ef",
  "subject": "dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598",
  "data": {
    "RequestId": "dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598",
    "ProcessingSuccessful": true
  },
  "eventType": "ProcessingCompleteEvent",
  "dataVersion": "1.0",
  "metadataVersion": "1",
  "eventTime": "2024-08-07T16:38:13.510618Z",
  "topic": "/subscriptions/d9239969-5904-41d6-a307-fc5df187d7e1/resourceGroups/adh_EventDrivenExample_rg/providers/Microsoft.EventGrid/topics/ProcessingCompleteTopic"
}
```

AzureRelayListener

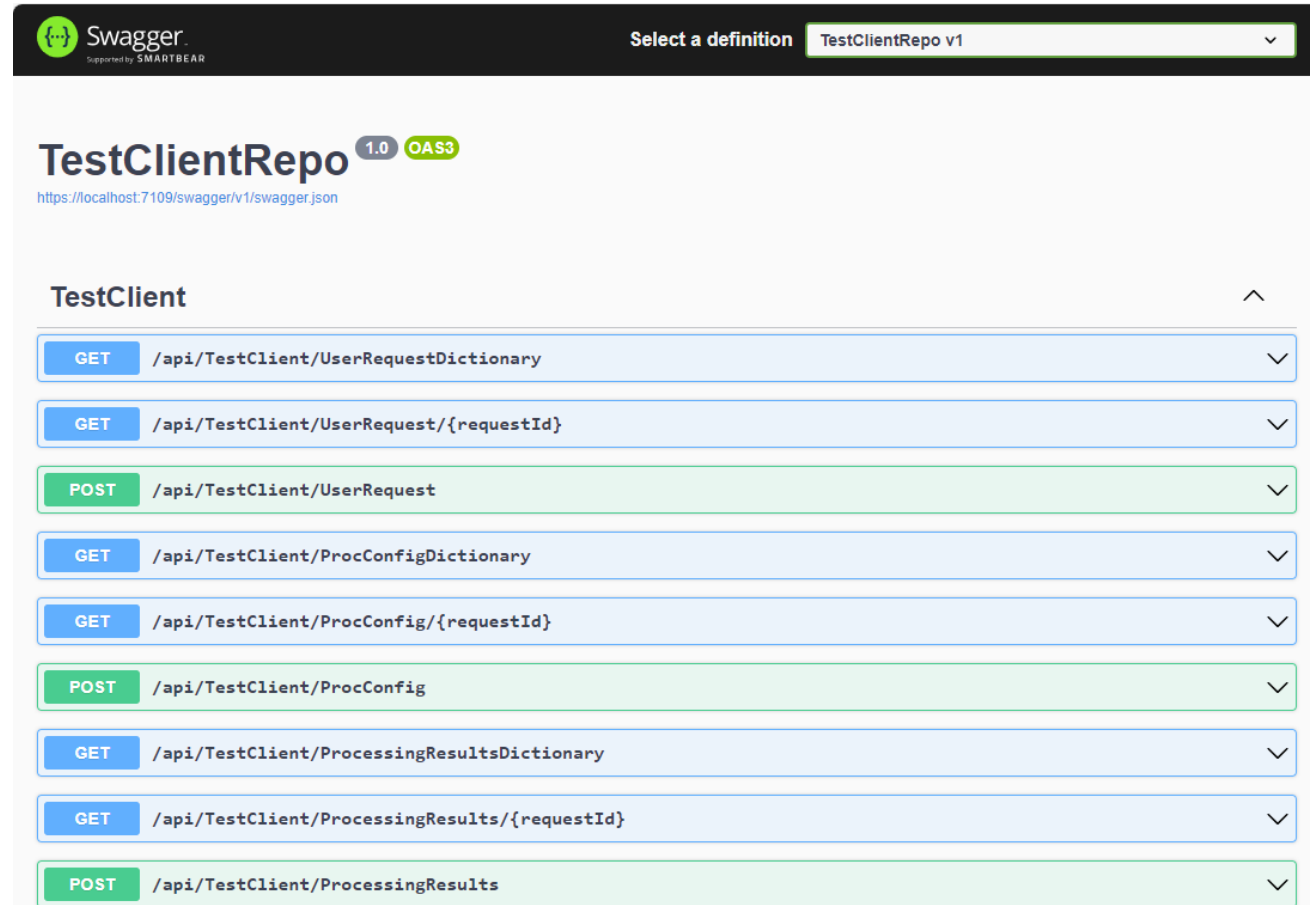
- Software from a 3rd Party, Console App prints the content of any strings received through the Azure Relay (i.e., notifications sent back to the Client).

A screenshot of a Windows console window with a dark background. The title bar at the top shows the file path 'C:\Users\A.hiron\OneDrive - A' followed by a close button. The console output consists of several lines of text: 'Online', 'Server listening', 'New session', 'RequestId dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598', 'Successful = True', and 'End session'.

```
C:\Users\A.hiron\OneDrive - A X + v - □ X
Online
Server listening
New session
RequestId dd1d1a57-5bb5-41e5-9fba-1b3ec7f54598
Successful = True
End session
```

TestClientRepo

Debug utility to test the client interface to the Repository. In particular, it allows viewing the contents of the three Dictionaries at runtime.



The image shows the Swagger UI for the TestClientRepo v1 API. The header includes the Swagger logo, the text "Supported by SMARTBEAR", and a dropdown menu to "Select a definition" with "TestClientRepo v1" selected. Below the header, the API title "TestClientRepo" is displayed with version "1.0" and "OAS3" tags, along with the URL "https://localhost:7109/swagger/v1/swagger.json". The main section is titled "TestClient" and lists ten API endpoints, each with a method (GET or POST) and a dropdown arrow. The endpoints are: GET /api/TestClient/UserRequestDictionary, GET /api/TestClient/UserRequest/{requestId}, POST /api/TestClient/UserRequest, GET /api/TestClient/ProcConfigDictionary, GET /api/TestClient/ProcConfig/{requestId}, POST /api/TestClient/ProcConfig, GET /api/TestClient/ProcessingResultsDictionary, GET /api/TestClient/ProcessingResults/{requestId}, and POST /api/TestClient/ProcessingResults.

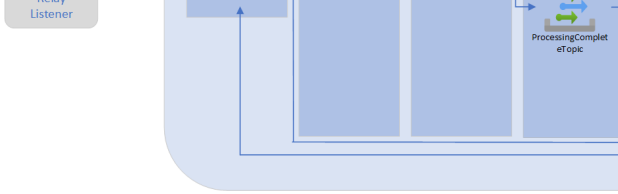
Method	Endpoint
GET	/api/TestClient/UserRequestDictionary
GET	/api/TestClient/UserRequest/{requestId}
POST	/api/TestClient/UserRequest
GET	/api/TestClient/ProcConfigDictionary
GET	/api/TestClient/ProcConfig/{requestId}
POST	/api/TestClient/ProcConfig
GET	/api/TestClient/ProcessingResultsDictionary
GET	/api/TestClient/ProcessingResults/{requestId}
POST	/api/TestClient/ProcessingResults

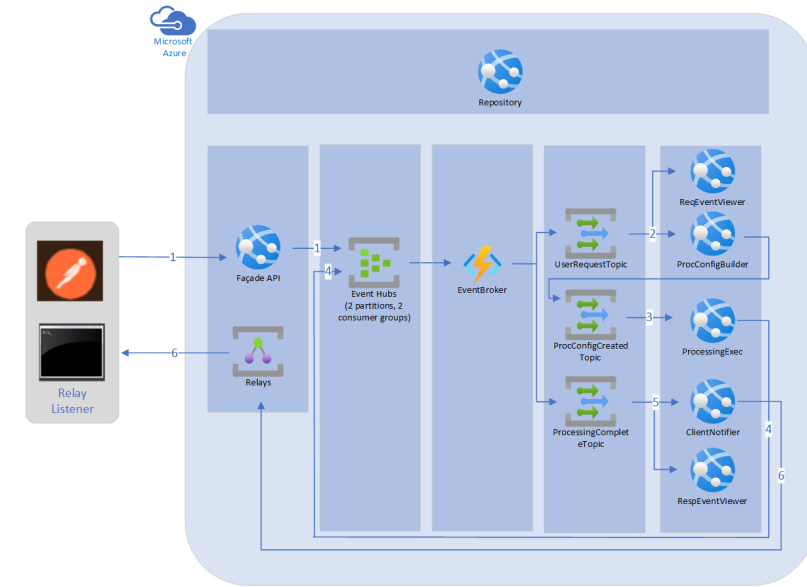
Demo

Code and Construction

PoC design decisions

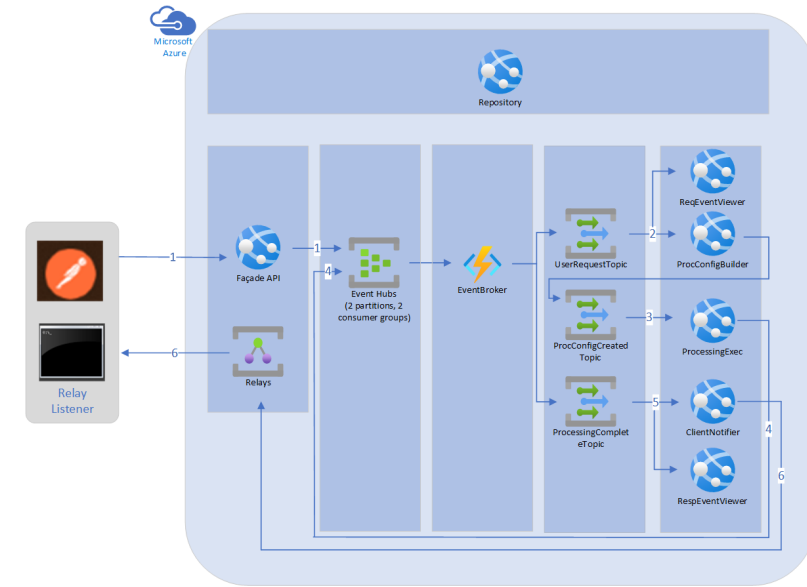
PoC Design Decisions

- One Azure Event Hub
 - Single Event Hub far exceeds the required capacity
 - Additional Event Hubs would be expensive
 - Multiple Event Grid Topics
 - Event Grid Topics are very cheap
 - The Topics are operated in simple pass-through operation to all registered Subscribers
 - No development effort for filtering
 - No on-going maintenance for filtering
 - Intuitively (not tested), faster operation without filtering
- 
- The diagram illustrates the Event Grid architecture. It shows a 'Listener' box on the left, which connects via an arrow to a central 'Event Grid' box. The 'Event Grid' box is divided into three vertical sections. The first two sections are labeled 'Event Grid' and 'Event Grid'. The third section is labeled 'ProcessingComplete' and 'eTopic'. An arrow points from the 'Event Grid' box to the 'ProcessingComplete' section. Another arrow points from the 'ProcessingComplete' section to a 'Subscriber' box on the right.



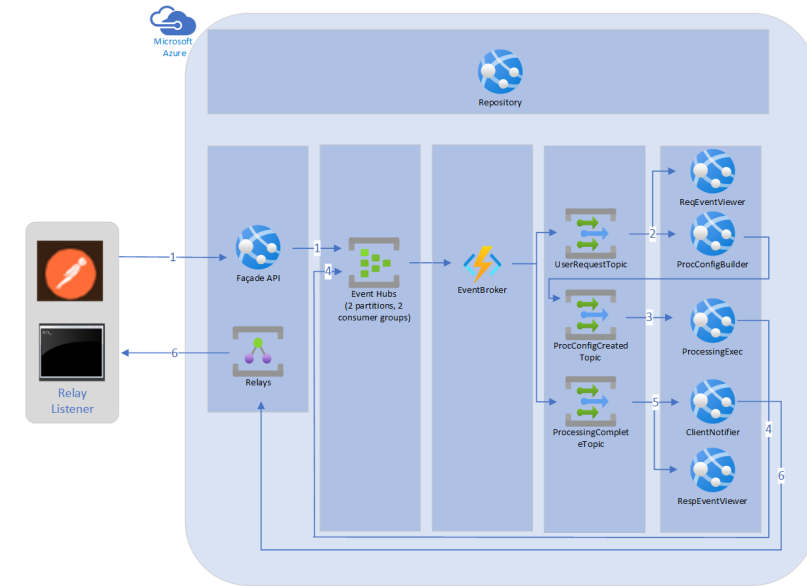
PoC Design Decisions

- Event Grid Subscribers
 - Implemented as Web Hooks
 - Software only, less infrastructure cost
 - Common code abstracted for simpler maintenance
- Use of ReqEventViewer, RespEventViewer
 - Instances of Event Grid Viewer, capture events generated for user requests and published results (provides insight)
 - Potential placeholders for stream analytic processing.



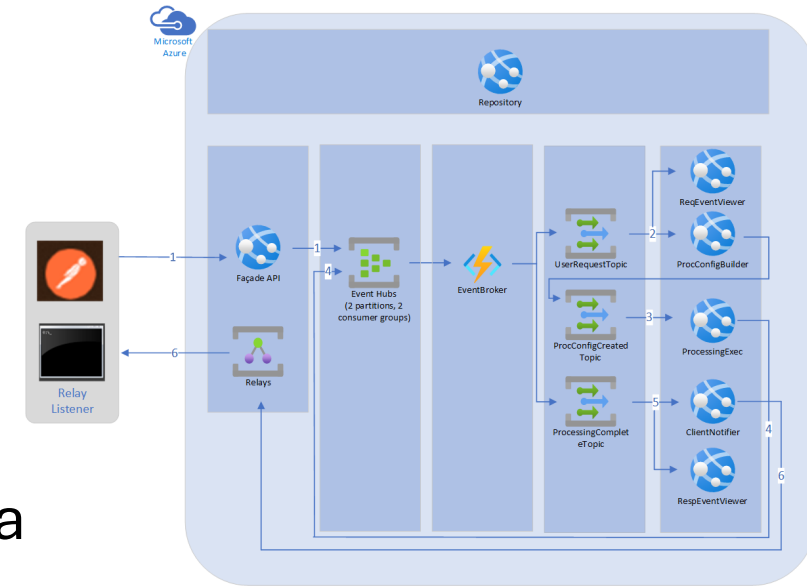
PoC Design Decisions

- Azure Relay
 - Allows notifications to be “pushed” to client
- Azure Relay Listener
 - Used to receive and display the received notification
- Repository
 - Low-cost, simple representation of a cache / database, implemented as a web API.



PoC Design Decisions

- Use of Event State Transfer pattern
 - The Façade API passed the entire UserRequest data
 - Fairly small, limited in size
 - Faster to access than retrieval from Repository(?)
- Use of Event Notification pattern
 - In all* other cases, the data size was more variable and usually larger
 - Event Type and the Request ID was used to access the data from the Repository.
 - *Except the notification sent to the client, where data was minimised for security



Lessons Learned

Lessons Learned #1

- Once code is debugged, most issues arise from erroneous configuration
 - Connection string and Shared Access Policies for Event Hub
 - Incorrect paths for Event Grid Topic Subscribers
 - Topic endpoints and Access Keys for Event Grid Publishers
- Continual taking off-line of the Event Hub and/or triggered Function can break sync
 - Remediation through deleting associated Blobs
 - (Can be simpler to delete and re-create both components)

Lessons Learned #2

- An end-to-end error-handling strategy is essential
 - E.g., retries for failures at internal steps
 - E.g., reporting failures at internal steps
- An event taxonomy can be beneficial
 - Many services may issue a “Processing Complete” event, but if it goes to the unintended recipient confusion can result.
- Beware copy-paste errors when creating IaC for multiple environments
 - Events may be sent to the wrong location