**Microsoft Azure**

**Azure Service Bus:**

* Azure Service Bus is a messaging service provided by Microsoft Azure that enables reliable asynchronous communication between applications and services.
* It allows developers to build decoupled systems by facilitating the exchange of messages between different components of a distributed application, whether running on-premises or in the cloud.

Characteristics of Azure Service Bus:

* Reliable asynchronous message delivery (enterprise messaging as a service ) that requires polling.
* Advanced messaging features like first-in and first-out (FIFO), batching/sessions, transactions, dead-lettering, temporal control, routing and filtering, and duplicate detection.
* At least once delivery of a message.
* Optional ordered delivery of messages.

Types of Azure Service Bus:

1. Queues:

* Queues provides reliable message storage and delivery for one-to-one communication between sender and receiver.
* Messages are sent to a queue and are then retrieved and processed by the receiving application.

1. Topics and subscriptions:

* Topics and subscriptions enable one-to-many communication by allowing multiple applications to subscribe to a specific topic.
* Message sent to the topic are then delivered to all the subscriptions associated with that topic.

SDK concepts:

* A Service Bus client is the primary interface for developers interacting with the Service Bus client library.
* A Service Bus sender is scoped to a particular queue or topic and is created using the Service Bus Client. The sender allows you to send messages to a queue or topic. It also allows for scheduling messages to be available for delivery at a specified date.
* A Service Bus receiver is scoped to a particular queue or subscription and is created using the Service Bus client. The receiver allows you to receive messages from a queue or subscription.
* A Service Bus processor is scoped to a particular queue or subscription and is created using the Service Bus client. The service Bus Processor can be thought of as an abstraction around a set of receivers.

Practical:

1. Create Service Bus from Azure portal choose the price as standard. Inside this azure service bus create a queue name as “firstqueue”.
2. Create a Access key policy and copy the connection string.
3. Now go on Visual Studio create a console project give name as ASB-queue1.
4. Go on NuGet Package and install “ Azure.Messaging.ServiceBus” .
5. Now write code for send message and receive message.

// See https://aka.ms/new-console-template for more information

using Azure.Messaging.ServiceBus;

//create connection

await using var client = new ServiceBusClient("Endpoint=sb://servicebus-queuedemo.servicebus.windows.net/;" +

"SharedAccessKeyName=ConnectionString;SharedAccessKey=Jg0hA05zCsIOd6xOwBs+Bvwb/TzmBTT8y+ASbJIDaHI=;EntityPath=firstqueue");

//create sender

ServiceBusSender sender = client.CreateSender("firstqueue");

//send message

await sender.SendMessageAsync(new ServiceBusMessage("Hello Keshav"));

//create receiver

ServiceBusReceiver receiver = client.CreateReceiver("firstqueue");

//receive the message

ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();

//receiver.CompleteMessageAsync(message);

Console.WriteLine(message.Body.ToString());

Console.ReadLine();

Operations on Message in Azure Service Bus:

1. Complete – When a message is received from a queue or subscription, the receiver can call the “Complete” operation to indicate that the message has been successfully processed and can be removed from the queue or subscription. This ensures that the message is not received again by another receiver.

Code for complete:

ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();

receiver.CompleteMessageAsync(message);

1. Abandon – if the receiver is unable to process the message for some reason, it can call the “Abandon” operation to indicate that the message should be made available for another receiver to process. This returns the message to the queue or subscription and allows another receiver to receive and process it.

Code for Abandon:

ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();

//for Abandon

receiver.AbandonMessageAsync(message);

1. Defer – If a message cannot be processed immediately, the receiver can call the “Defer” operation to remove the message from the queue or subscription and store it separately until it is ready to be processed. This allows the receiver to prioritize other messages in the queue or subscription and come back to the deferred message later.

Code for Defer:

ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();

//for defer

receiver.DeferMessageAsync(message);

//Read the defer

ServiceBusReceivedMessage defermessage = await receiver.ReceiveDeferredMessageAsync(message.SequenceNumber);

Console.WriteLine(defermessage.Body.ToString());

1. Deadletter - If a message cannot be processed dure to some error that cannot be resolved, the receiver can call the “Deadletter” operation to indicate that the message should be moved to a dead-letter queue or topic for further analysis or processing. Dead-lettering is useful for handling messages that cannot be processed dure to issues with the message content, receiver code, or configuration.

**What is Serverless?**

* Serverless architecture generally describes fully managed cloud services.
* A serverless service could have all or most of the following characteristics:

Highly elastic and scalable

Highly available

Highly durable

Secure by default

* Abstracts away the underlying infrastructure of our business task.
* Serverless can Scale-to-Zero meaning when not in use the serverless resources cost nothing.

**Function as a Service:**

What is Function as a Service (FaaS)?

* Allows developers to focus on just writing pieces of code (functions).
* Has event-driven integration trigger functions based on event data or to emit event data.
* Generally multiple functions are orchestrated together to create a serverless app.
* Functions generally only run when needed.

Function as a Service (FaaS) is not serverless on its own. Faas is only serverless if its fully managed and scales to zero.

**Azure Function:**

Azure Functions is Function as a Service offering that allows developers to focus on writing code and not worry about maintaining the underlying computing infrastructure.

* A Function App defines the underlying compute for a collection of functions. A function App defines the Hosting, Runtime and other global configurations.
* A trigger is the chosen event data that will cause function to execute.
* You can only have one trigger.
* Input Binding are one or multiple data sources that will be passed to the function when a trigger occurs.
* A function represents code along with application runtime configuration.
* Output bindings are one more data sink that will receive outputted data from the function on successful execution.
* There are four versions of Azure Functions 1.x,2.x,3.x and 4.x. We are currently using 4.x.

**Azure Logic Apps:**

Azure Logic Apps is a cloud platform where you can create and run automated workflows with little to no code. By using the visual designer and selecting from prebuilt operations, we can quickly build a workflow that integrates and manages your apps, data, services, and systems.

* Azure Logic Apps provides a cloud based serverless engine.
* Build automated workflows to integrate apps and data between cloud services and on-premises systems.
* We can workflows using a visual designer.
* We can more easily and quickly build highly scalable integration solutions for enterprise and business-to-business (B2B) scenarios.
* We can trigger workflows based on events or timers and leverage connectors to integration applications.
* Logic Apps integrates seamlessly with Azure Functions.

**Key Terms of Azure Logic Apps:**

**Logic App:**

* The Azure resource that we create when we want to build a workflow.
* Basically, we can create the following types of logic app resources:

1. A **Consumption logic app** resource that supports a single workflow, which is hosted and run in global multitenant Azure Logic Apps.
2. A **Standard logic app** resource that supports multiple workflows, which are hosted and run in single-tenant Azure Logic Apps.

**Workflow:**

* A series of operations that define a task, business process, or workload.
* Each workflow always starts with a single trigger operation, after which you must add one or more action operations.

**Trigger:**

* The first operation in any workflow that specifies the criteria to meet before running any subsequent operations in that workflow.
* For example, a trigger event might be getting an email in your inbox or detecting a new file in a storage account.
* Trigger is two types.
* **Polling trigger:** Checks a service endpoint for data or an event that meets the trigger condition, based on the specified schedule. If the trigger condition is met at that time, the trigger fires, creating and running a new workflow instance that uses any trigger outputs as inputs for the workflow.
* **Push trigger:** Listens and waits at a service endpoint for data or an event that meets the trigger condition. At that time, the trigger fires immediately, creating and running a new workflow instance that uses any trigger outputs as inputs for the workflow.

**Polling trigger example:**

* **Recurrence trigger:** Set the recurrence for running your workflow, including the start date and time.
* **When an email is received:** Check for new email from any mail provider that’s supported by Azure Logic Apps, for example office 365, outlook, Gmail.
* **HTTP trigger:** Call a service endpoint over HTTP or HTTPS.

**Push trigger example:**

* **Request trigger:** Receive incoming HTTPS requests.
* **HTTP Webhook trigger:** Subscribe to a service endpoint by registering a callback URL with that service. That way, the service can just notify the trigger when the specified event happens, so that the trigger doesn’t need to poll the service.

**Action:**

* Each subsequent operation that follows the trigger in the workflow.

**Built-in Operation:**

**Manage connectors:**

**Maps:**

**Stateful Vs Stateless:**

**Stateful:**

* State is stored into some storage e.g. storage account, SQL.
* Can handle message more than 64 KB in size.

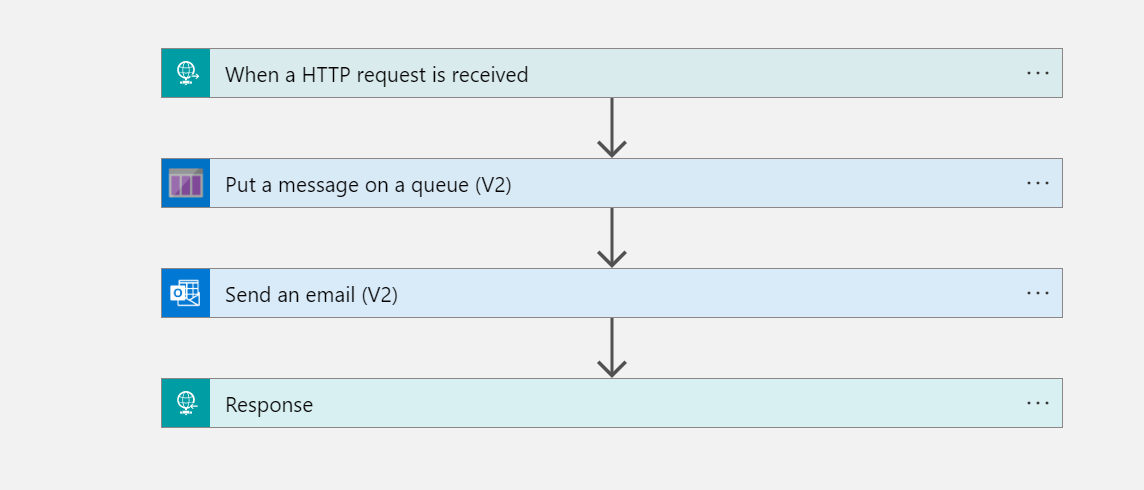
**Stateless:**

* State is not storage into some storage.
* Suitable for light weight message up to 64KB.
* Limited trigger options available.

Azure Logic App Practice:

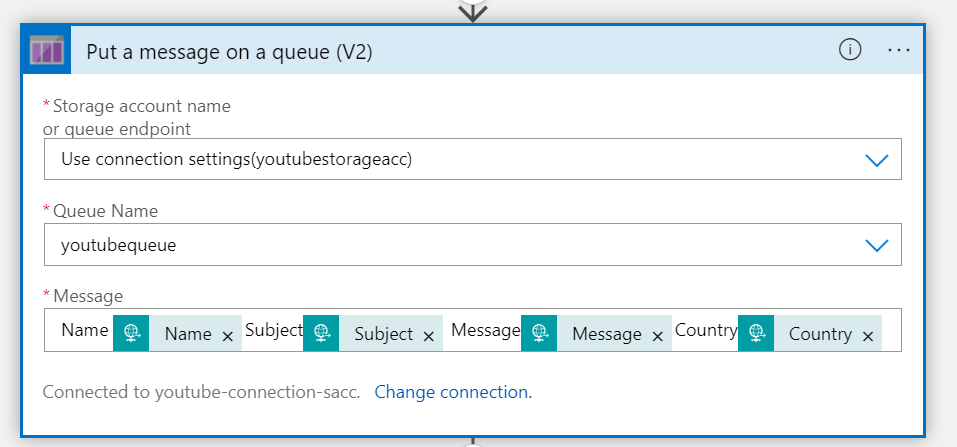
1. Create a http trigger workflow that will send message to azure storage queue and send an email.

Steps:



A screenshot of a computer

Description automatically generated



A screenshot of a email

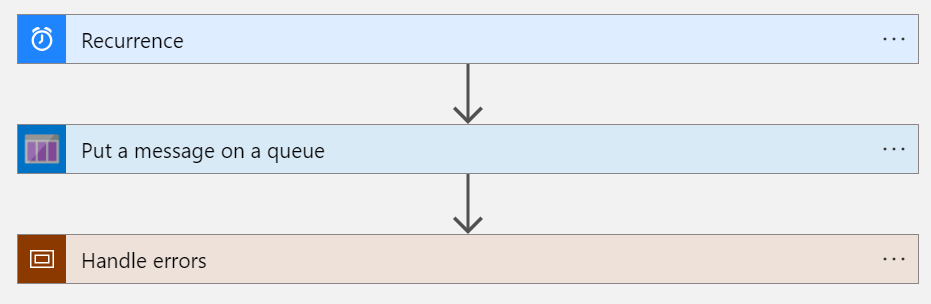
Description automatically generated

A screenshot of a computer

Description automatically generated

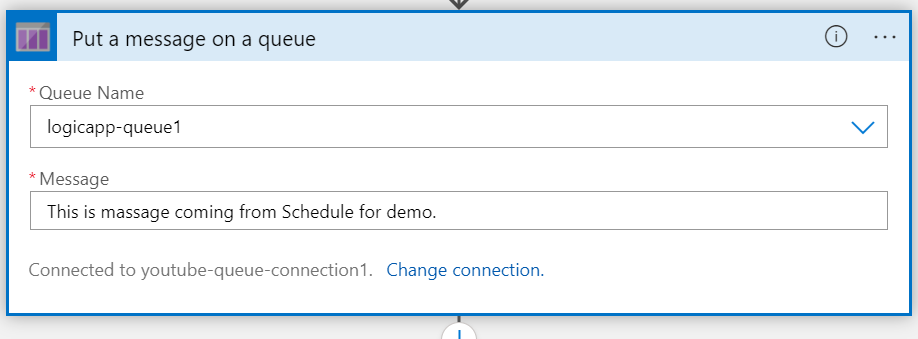
1. Create a schedular Recurrence and send message to the azure storage queue 1 and if fail then handle errors, send failure message to azure storage queue 2.

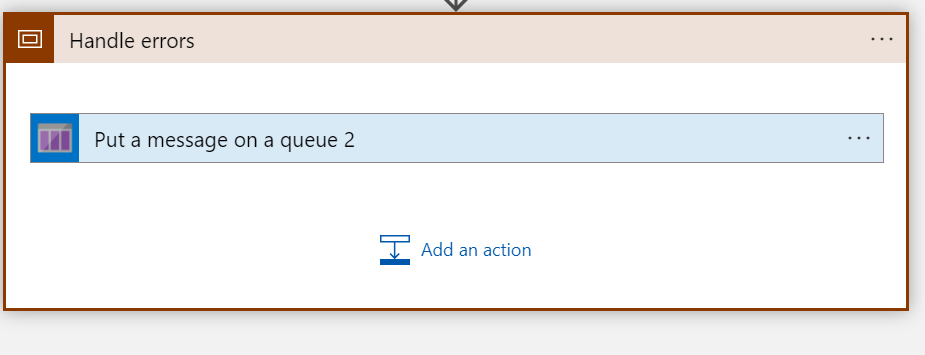
Steps:



A screenshot of a computer

Description automatically generated

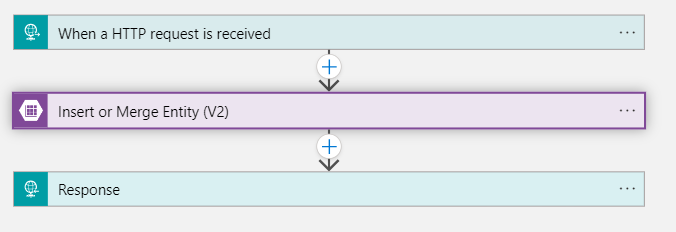




A screenshot of a computer

Description automatically generated

1. Create http trigger and send message to azure storage table and give the response.



A screenshot of a computer

Description automatically generated

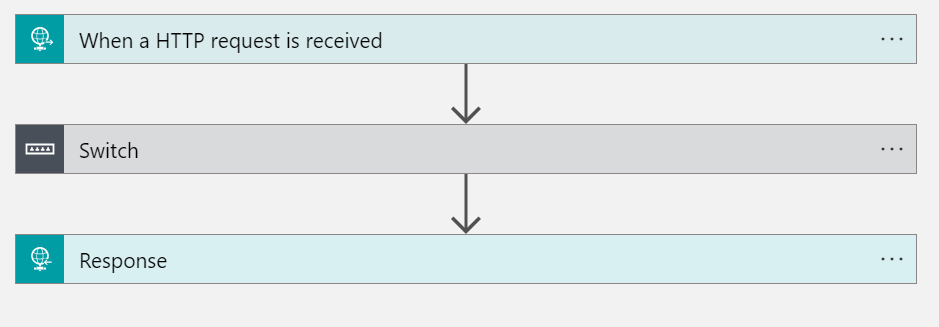
A screenshot of a computer

Description automatically generated

A screenshot of a computer

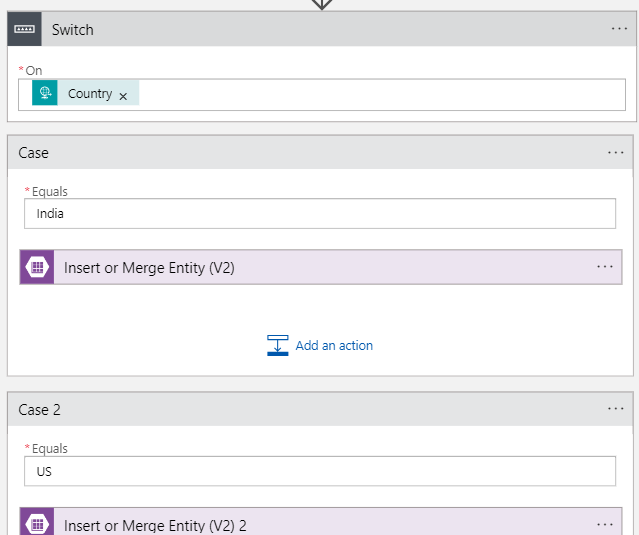
Description automatically generated

1. Create http trigger with switch case.



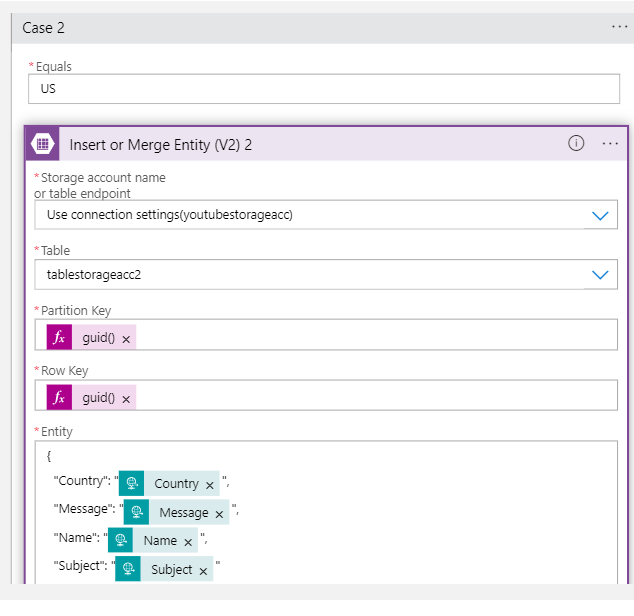
A screenshot of a computer

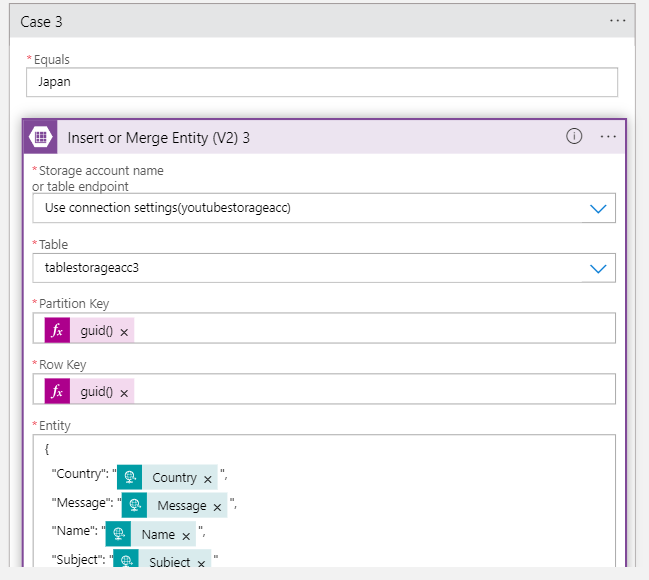
Description automatically generated

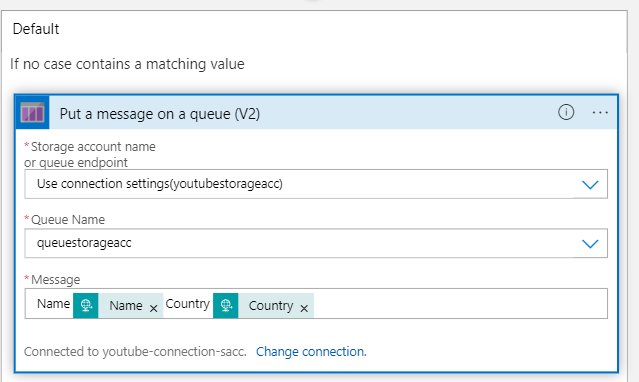


A screenshot of a computer

Description automatically generated







A screenshot of a computer

Description automatically generated

**Azure API Management**

* Azure API Management (APIM) integrates existing back-end services into modern API gateways.
* Azure API Management is**a platform for managing APIs across all environments.**
* It follows the API-first approach, decoupling front-end and back-end teams with the help of API mocking.
* **Azure API Management handles the full management of your APIs.**
* **It centralizes the securing. Versioning, documentation, and compliance from your back-end services in a single point.**

**API Management components**

* Azure API Management is made up of an API *gateway*, a *management plane*, and a *developer portal*. These components are Azure-hosted and fully managed by default.

Microsoft Azure

User

Visual Studio

APIM

Service

Function App

Azure CosmosDB

https http http

**Azure API Management- Key Concepts**

**Key Concepts:**

* API represents a set of operations.
* API Operation connects an API endpoint to its backend.
* **Product:** A logical grouping of APIs.
* A single or a group of APIs make up a product, which is how your APIs are represented to developers. It can be either public or private.
* When you publish a set of APIs as a product, you can.
* control how APIs are accessed via subscription.
* set approvals for accesses.
* set policies at product level, so the policy rules apply to all APIs in the product.
* test the APIs in the Developer Portal Console
* **Backend** represents back-end services in your API.
* **Group,** used to manage the visibility of products to developers.

**Administrators** have full access to the API Management.

**Developers,** users with access to the developer’s portal with permissions to build applications.

**Guests,** users without access to the developer's portal but with reading permission in some services.

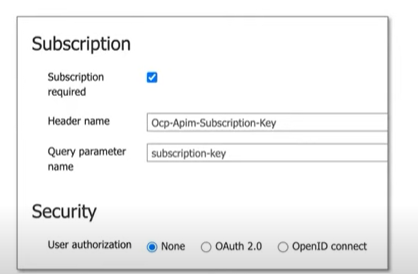
* **Developer,** belons to one or more Product groups, and each developer has a primary and secondary key to call the developer’s APIs.
* **Policies,** configurations and validation that are applied in the progress to incoming requests and outcome responses.
* **Named Values,** key-value pairs used with policies, Values can be a result of an expression.
* **Gateway** is where your API calls are received, and policies are applied to incoming requests.
* **Developer Portal** where developers can access all APIs and products listed by your APIM alongside its APIs operations and documentations. Developers can also request access of your APIs from the developer’s portal.

**APIM – Echo API Service:**

* When you create an APIM Gateway by default it will create an API called Echo API.
* Echo API is a non-production azure service that is used to test Azure API Management.

**APIM – API Authentication:**

* In order to authenticate with our APIs, we configure those settings under the subscription section.
* If the subscription is required, only developers with a valid access key can use it.
* We can configure where the API will receive the access keys, which can be sent as a header or query string.
* If Subscriptions required is not checked, anonymous requests are allowed.



**APIM – Groups:**

* Groups are used to manage the visibility of products to developers.

1. Administrators:

Manage API Management service instances and create the APIs, operations, and products that are used by developers.

1. Developers:

Authenticated developer portal users that build applications using your APIs.

Developers are granted access to the developer portal and build applications that call the operations of an API.

1. Guests:

Unauthenticated developer portal users, such as prospective customers visiting the developer portal. They can be granted certain read-only access, such as the ability to view APIs but not call them.

**APIM-Frontends:**

* Frontends defines the route/endpoint and the documentation and configuration around that endpoint.
* API does not host APIs, it creates facades (mukhauta) for your APIs.

**APIM – Backends:**

For Backends we can set the following types:

* Custom URL- Point to server where your service is running.
* Azure Resource – Integrate directly to an Azure resource eg.

Azure Functions

App Service

Container App

Logic App

* Azure Service Fabric
* Authorization credentials present authorize requests credentials to the backend service.
* Headers – HTTP headers: We can fetch from Named Values.
* Query: query string: We can fetch from Named/Valued.
* Client Certificates: x.509 certificates

**APIM – Policies:**

Policies are a set of instructions/rules in an XML based format that influence how inbound requests and outbound responses are processed.

API Management Policies allow you to change the behavior at multiple stages of an endpoints request lifecycle.

We can update any part of the request and response message eg. Headers, body, URLs, ets.

There are four areas where policies can be applied.

1. Inbound – for incoming requests.
2. Backend – begore requests reach your backend.
3. Outbound – before sending response back to client.
4. Error – when a request encounters an error.

Azure has a collection of policy groups which contain many policies we can apply.

* Access Restriction Policies
* Advanced Policies
* Authentication Policies
* Caching Policies
* Cross-Domain Policies
* Transformation Policies
* Daper integration Policies
* Validation Policies
* Graph QL Validation Policies

**When an error occurs, no other policies are applied except the error policies; however, if other policies were in effect prior to the error, they will not be removed.**

Product-level policies applied to all the operation within a product.

**APIM- APIs**

Defining a New API via:

* Http
* WebSockets
* GraphQL

From Definition:

* OpenAPI v3
* WADL
* WSDL

From Azure Resource

* Logic App
* App Service
* Function App