Assignment - Azure SQL Database and REST API

This application is designed based on Azure SQl Database (PaaS) and Microsoft RESTful API.

Contents

1. Prerequisite

- Visual Studio 2017/2019 (Community Edition/Standard Edition etc.)
- .Net Core 3.1
- Azure Subscription
- Github

1.1 Context

I have created here a Personal data Management System using RESTful API and allow user for CRUD operation.

2. Model

In this example, I am storing some personal details into Azure SQl database. The following are the main attributes of this Model.

- Id (PK and automatically increase value)
- First name (Varchar)
- Last name (Varchar)
- Date Created Date (default system date and no entry is required)

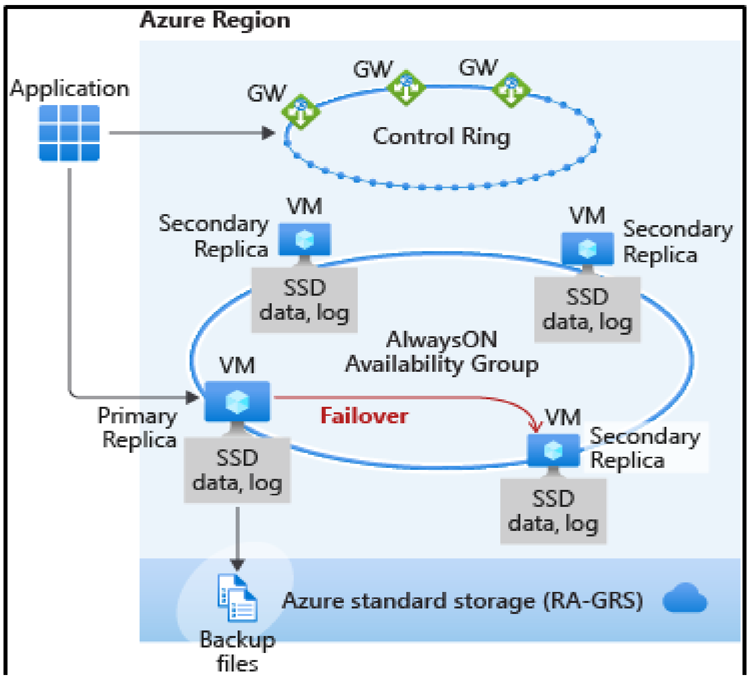
3. SQl Database Creation in Azure

I have used here a single database at single availability zone for testing purposes. In actual production environment I will deploy my SQl instance either in SQL Elastic Pool or SQL Managed Instance. In both cases multiple SQl instance (Elastic Pool) should deploy under Arability group in different Availability zone for HA purposes. For Disaster Recover I would also suggest deploying at least another instance on different region with asynchronization replication.
There are two High availability model in Azure SQl platform

a) Standard
b) Premium

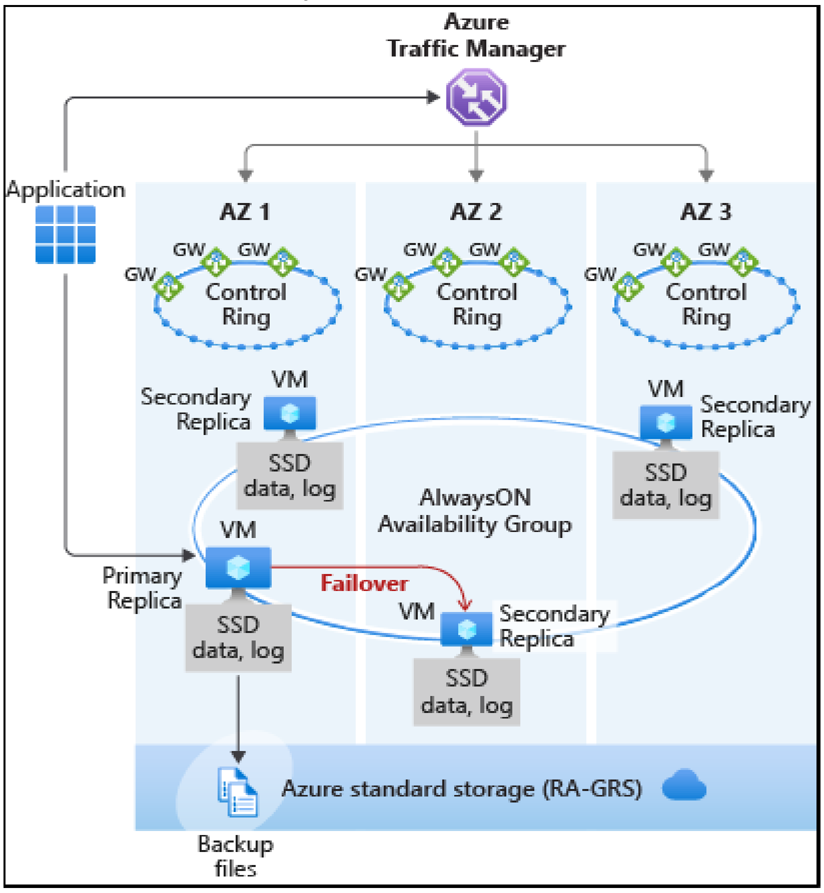
Choosing the best model is depends on Customer demands and the concurrent transactions of the database. In case of High OLTP (Bank or Retail shop) or big Data Warehouse (Manufacturer company) application I would always prefer to use the Premium Availability Architecture. The below two diagrams have described both locally redundant and zone redundant scenario.

Locally Redundant Availability: Premium and Business Critical tier



Source: https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla

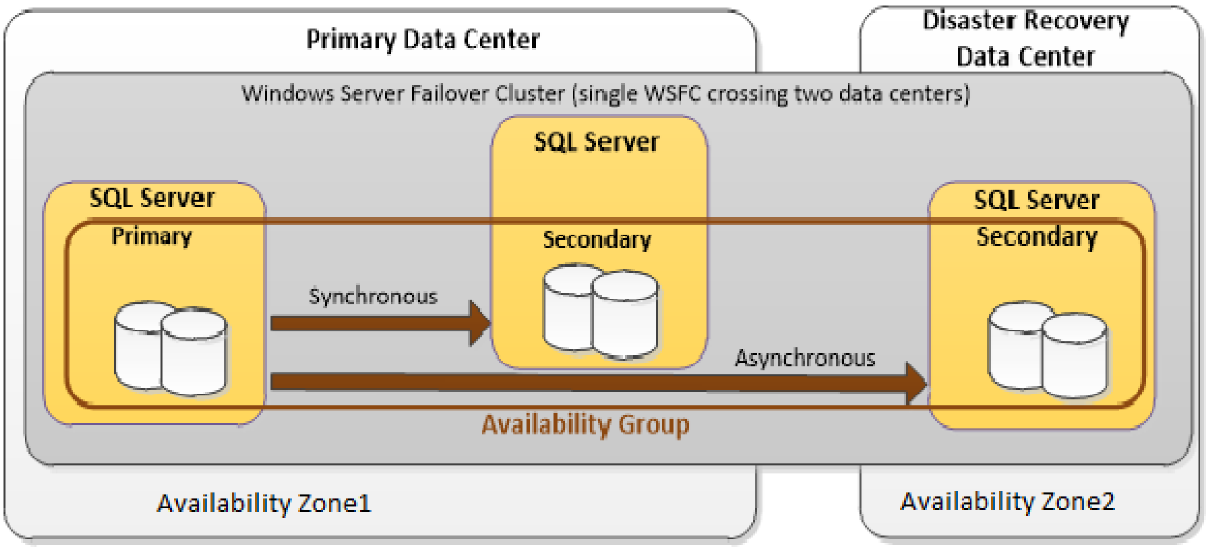
Zone Redundant Availability: Premium and Business Critical tier



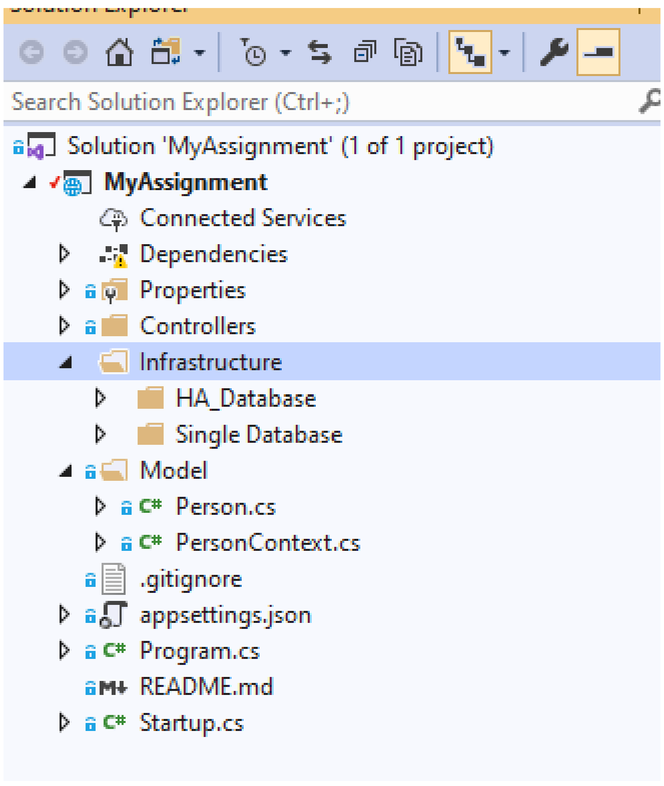
Source: https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla

Both cases the underlying database files (.mdf/.ldf) are placed on the attached SSD storage to provide extremely low latency IO. High availability is implemented using a technology similar like SQL Server Always On availability groups. This process includes one primary replica with three secondary replicas as well. This process guarantees that if the primary node crashes for any reason, there is always a fully synchronized node to fail over to.

General HA and DR solution of SQl Server using Always on Availability group



In my assignment, for creating the SQl infrastructure in Azure I have used HashiCorp Terraform. Details script present inside the Infrastructure directory. Both Single zone and the High availability zone IaC (Infrastructure as a code) has been provided.



Search Solution Explorer (Ctrl+;)

```
Solution 'MyAssignment' (1 of 1 project)
  MyAssignment
    Connected Services
    Dependencies
    Properties
    Controllers
    Infrastructure
      HA_Database
      Single Database
    Model
      Person.cs
      PersonContext.cs
    .gitignore
    appsettings.json
    Program.cs
    README.md
    Startup.cs
```

3.1 Terraform command:

terraform init – To initialize the modules
terraform plan – This is a dry run to check the Infrastructure is correctly deploying.
terraform apply – To apply and create Infrastructure on Azure
terraform destroy – To destroy the Infrastructure.

3.2 Create Service Principal:

For properly executing the terraform script on Azure, it is recommended that to create Service Principal rather using the System Admin Account. We can create service principal using the below command on Azure shell:

```
#login to Azure

az login

# view and select my subscription account

az account list -o table

$SUBSCRIPTION=<id>

az account set --subscription <SubscriptionId>
```

$SP_JSON=$(az ad sp create-for-rbac --skip-assignment --name terraform -o json)

#Keep the “appId” and “password” for later use.

$SERVICE_PRINCIPAL_ID=$(echo $SP_JSON | jq -r '.appId')

SERVICE_PRINCIPAL_SECRET=$(echo $SP_JSON | jq -r '.password')

#grant contributor role over the Subscription to the service principal

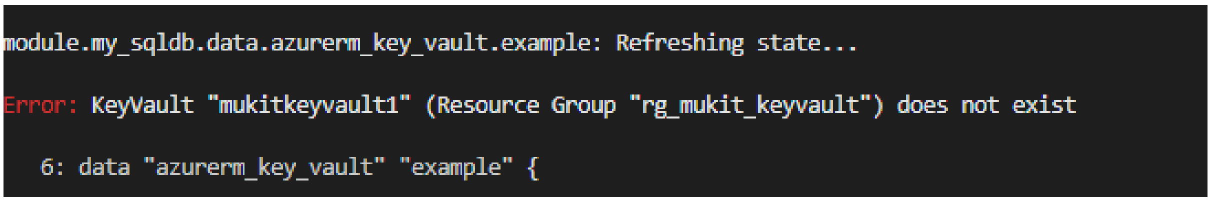
az role assignment create --assignee $SERVICE_PRINCIPAL \

--scope "/subscriptions/$SUBSCRIPTION

--role Contributor

3.3 Error message:

In real production scenario, it is better to create key vault using Azure cli for security purposes. Should not use the secrete or key inside the terraform script. If we do not create the key vault before applying terraform script, then it will fail with below error message:



```
module.my_sqldb.data.azurerm_key_vault.example: Refreshing state...

Error: KeyVault "mukitkeyvault1" (Resource Group "rg_mukit_keyvault") does not exist

   6: data "azurerm_key_vault" "example" {
```

3.4 Monitoring and Alert:

After successfully provisioning the SQl Infrastructure it is necessary to monitor the Infrastructure and create a smart alerting system. E.g. High CPU utilization, High memory Utilization, High latency, I/O wait etc. Inside the Azure portal there are several Monitoring tools are available like application Insight, Log Analytics, Azure Diagnostic etc. 3rd party Datadog is one of the famous Azure Monitoring tool in the market.

4. Design a table

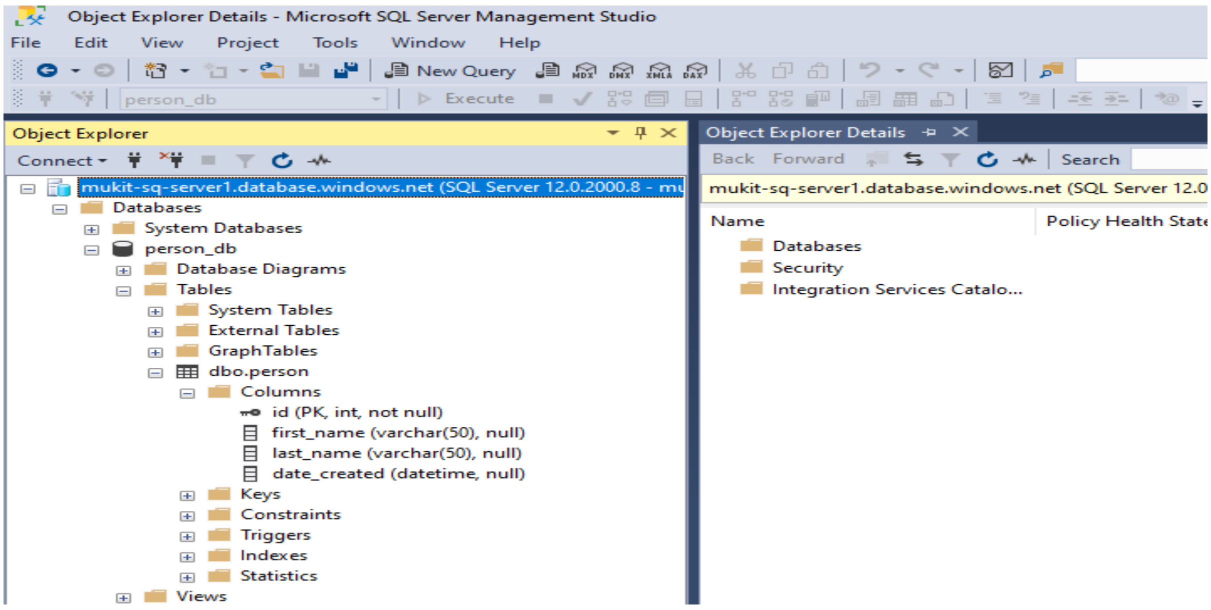
Login to the SQl Server instance using SSMS and execute the below code

```
CREATE TABLE [dbo].[person](
	[id] [int] IDENTITY(1,1) NOT NULL,
	[first_name] [varchar](50) NULL,
	[last_name] [varchar](50) NULL,
	[date_created] [datetime] NULL,
PRIMARY KEY CLUSTERED
(
	[id] ASC
)WITH (STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
```

GO

ALTER TABLE [dbo].[person] ADD CONSTRAINT [DF_person_date_created] DEFAULT (getdate()) FOR [date_created]
GO

After successfully execution of the above script it looks like below:



```
mukit-sq-server1.database.windows.net (SQL Server 12.0.2000.8 - mu
  Databases
    System Databases
    person_db
      Database Diagrams
      Tables
        System Tables
        External Tables
        GraphTables
        dbo.person
          Columns
            id (PK, int, not null)
            first_name (varchar(50), null)
            last_name (varchar(50), null)
            date_created (datetime, null)
          Keys
          Constraints
          Triggers
          Indexes
          Statistics
      Views
```

mukit-sq-server1.database.windows.net (SQL Server 12.0

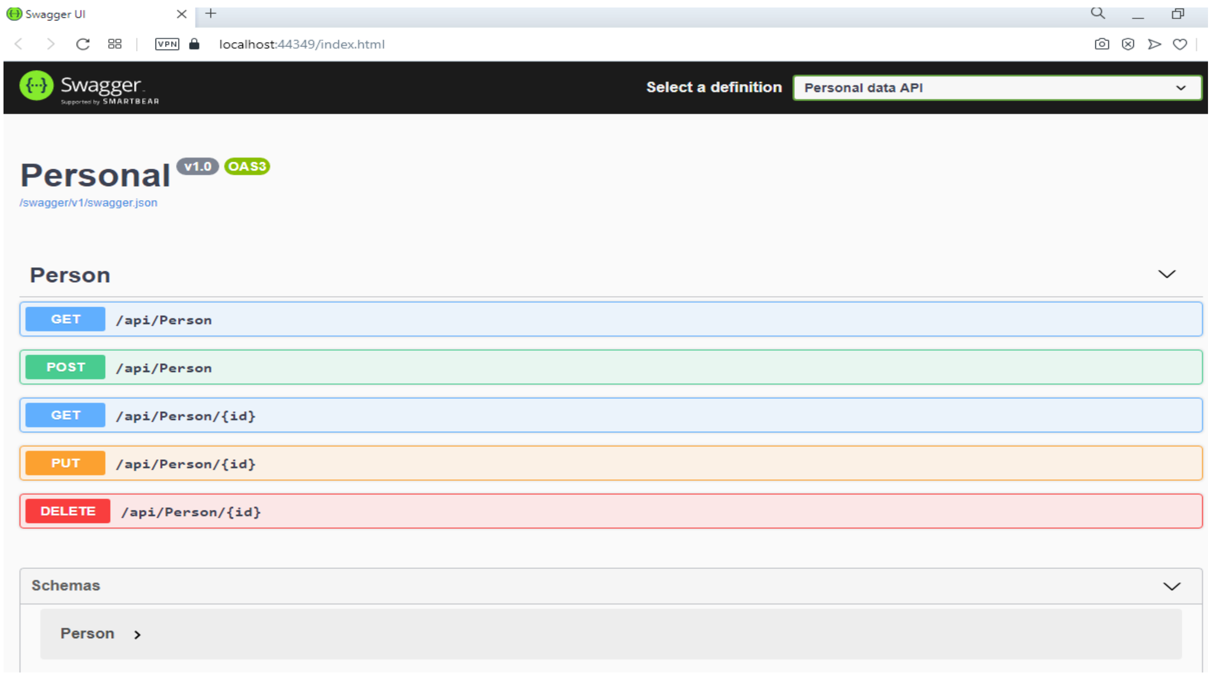
Name	Policy Health State
Databases	
Security	
Integration Services Catalo...	

5. Create the API using .Net Core

I have created the base project using Visual Studio 2019 community edition and have chosen the ASP.NET Core Web Application. I have also selected API and make sure it has used DOT Net Core version 3.1. After creating the base project, it is mandatory to add the below NuGet packages:
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.SqlServer.Design
Microsoft.EntityFrameworkCore.Tools

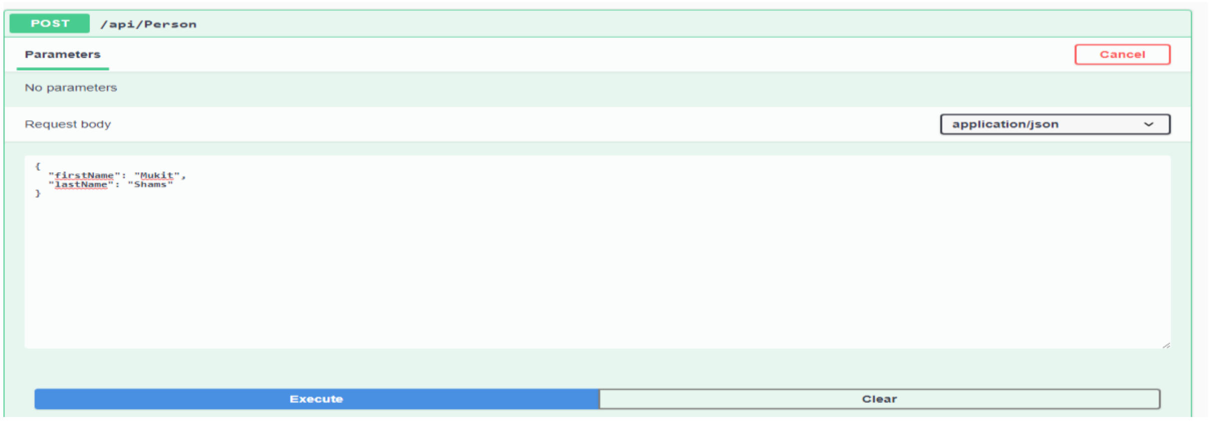
For testing the API on UI mode, I have also used here the **Swagger UI.** For this reason, I have also installed “Swashbuckle.AspNetCore” NuGet package to the project.

The final version of my project now ready to deploy on staging environment looks like:



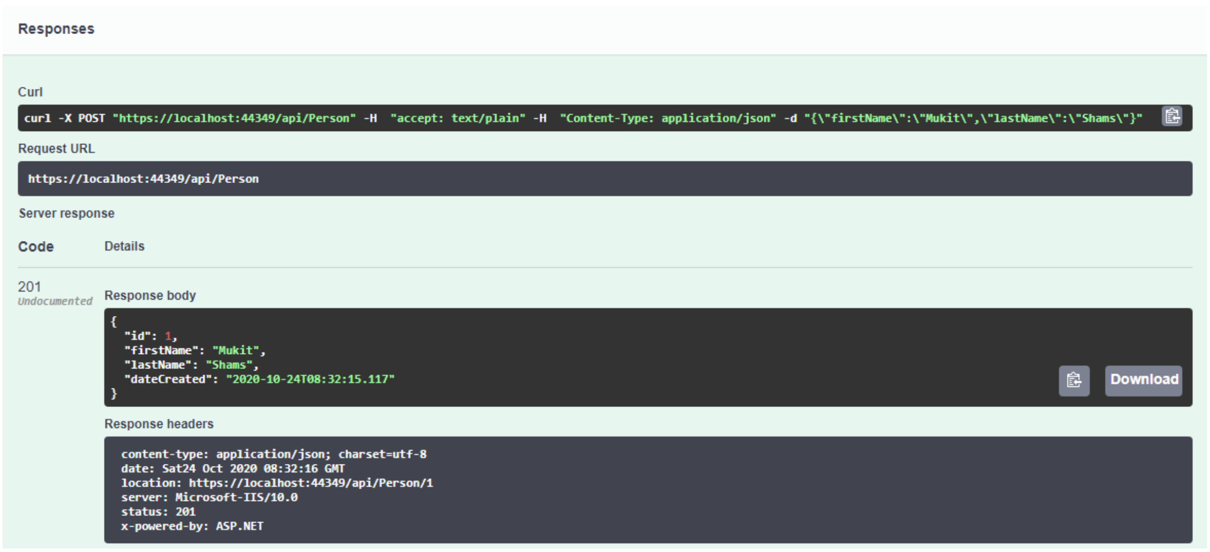
Inserting data using Post (JSON format):

```
{
  "firstName": "Mukit",
  "lastName": "Shams"
}
```



```
{
  "firstName": "Mukit",
  "lastName": "Shams"
}
```

Final output returned from database:



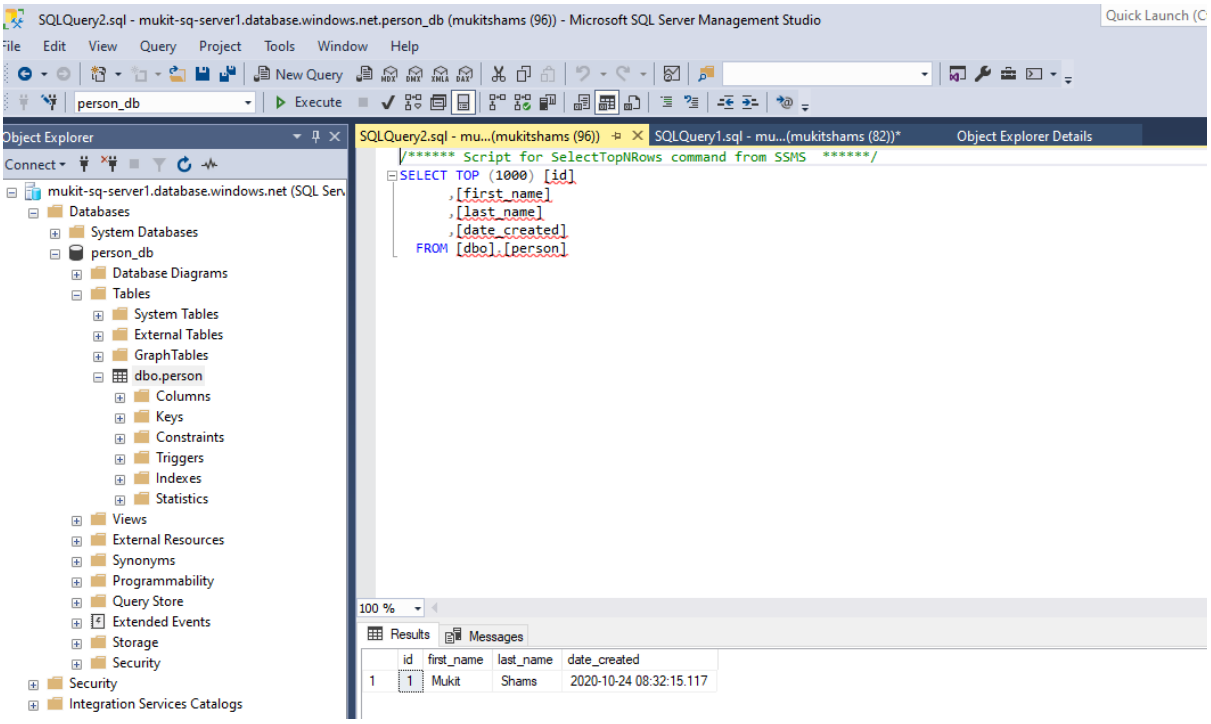
```
curl -X POST "https://localhost:44349/api/Person" -H  "accept: text/plain" -H  "Content-Type: application/json" -d "{\"firstName\":\"Mukit\",\"lastName\":\"Shams\"}"
```

```
https://localhost:44349/api/Person
```

```
{
  "id": 1,
  "firstName": "Mukit",
  "lastName": "Shams",
  "dateCreated": "2020-10-24T08:32:15.117"
}
```

```
content-type: application/json; charset=utf-8
date: Sat24 Oct 2020 08:32:16 GMT
location: https://localhost:44349/api/Person/1
server: Microsoft-IIS/10.0
status: 201
x-powered-by: ASP.NET
```

Verified the result on back-end table:



```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP (1000) [id]
      ,[first_name]
      ,[last_name]
      ,[date_created]
  FROM [dbo].[person]
```

	id	first_name	last_name	date_created
1	1	Mukit	Shams	2020-10-24 08:32:15.117

6. Application deployment plan:

To deploy the application in Azure we may consider the Azure App Service. Azure App Service is a PaaS service used to host a variety of apps on Azure, e.g. web apps, API apps, functions etc.
App Service apps in a subnet use own Azure Virtual Network, providing an isolated, highly scalable, and dedicated environment for the cloud workloads.

The main component of this architecture is the App Service environment (ASE). An ASE can be deployed either as external ASE with a public IP address, or as internal ASE, with an internal IP address belonging to the **Internal Load Balancer (ILB).**

Standard SKU load balancer is recommended for production workloads. Creating ILB (Internal Load Balancer) involves the below three major configurations:

- Load balancer settings for a backend address pool.
- A health probe.
- A load balancer rule.

App service deployment using ILB implemented here in the below link which can be used as a reference:
https://github.com/mspnp/app-service-environments-ILB-deployments

The Azure App Service Environment is an Azure App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale. This capability can host:

- Windows web apps
- Linux web apps
- Docker containers
- Mobile apps
- Functions

App Service environments (ASEs) are appropriate for application workloads that require:

- High scale.
- Isolation and secure network access.
- High memory utilization.

6.1 Scaling the App:

The application architecture should be structured so that individual components can be scaled based on usage. The API will deploy in its own App Service plan. We can monitor the app for any performance bottlenecks, and then scale it up if required.
Autoscaling can be enabled on Azure Application Gateway V2. This allows App Gateway to scale up or down based on the traffic load patterns

7. Backup and recovery plan

Disaster recovery is the process of restoring application functionality in the wake of a catastrophic loss. Fault tolerance for reduced functionality during a disaster is a business decision that varies from one application to another.
For this application, since using the Azure App Service and back-end SQl Server (either elastic pool or managed instance) hence we have to ensure to plan regular backup of the Virtual Server (App Server) as well as SQl Server database.

- For an application that mainly uses virtual machines, we can use **Azure Site Recovery (ASR)** for the web and logic tiers and or SQL Server on VMs, can use SQL Server Always On availability groups.
- For an application that uses App Service and Azure SQL Database, can use a smaller tier App Service plan configured in the secondary region, which auto scales when a failover occurs. For SQl server elastic pools or Managed instance use failover groups.
- In case of Data corruption in SQl level it is also recommended to backup the SQl server databases on regular basis.

7.1 Backup Strategy

The frequency of running the backup process determines the RPO. Disaster recovery plan should include how we will address lost data. For example if we plan for hourly backup plan without any transaction log backup for SQl database then if database failed on 11.40 AM means lost 40 mints data. An idle production System my SQl Server backup strategy should like below:

- Configure a FULL backup regularly on every night if the database size is medium/small level. In case of large Database this should be weekly basis.
- Configure differential backup on every day at least two times (Morning & Afternoon)
- Configure Transaction log backup on every 15-30 minutes duration. This is depending on RPO strategy and varied business to business basis.

In case of App Server or App service backup plan should be like below:

- Weekly Full Server backup
- Daily incremental backup

Underlying Storage should also configure with Zone redundancy plan so that, we have always a replication copy available to different zone in case of disk failure.

7.2 Recovery Plan

- Recover an Azure SQL database can be done using automated database backups
- Another option is to use active geo-replication for SQL Database, which automatically replicates database changes to secondary databases in the same or different Azure region.

VM Server or App Service can be recover using ASR (Azure Site recovery) plan. The disaster recovery plan is considered complete after it has been fully tested. Include the people, processes, and applications needed to restore functionality within the service-level agreement (SLA) defined for the customers. Microsoft has suggested to consider the below best practice guide for an idle disaster recovery plan:

- Plan, include the process for contacting support and for escalating issues.
- Evaluate the business impact of application failures.
- Choose a cross-region recovery architecture for mission-critical applications.
- Identify a specific owner of the disaster recovery plan, including automation and testing.
- Document the process, especially any manual steps.
- Automate the process as much as possible.
- Establish a backup strategy for all reference and transactional data, and test backup restoration regularly.
- Set up alerts for the stack of the Azure services consumed by your application.
- Perform regular Dr dry run to validate and improve the plan.

7.3 Regular Database Maintenance

Regular database maintenance is one of the major tasks for Database performance. Ola Hallengren (https://ola.hallengren.com/) has provided an industry standard SQl Server maintenance plan scripts. This set of scripts supports both on-premises and cloud and safe to use on production. Scripts includes the below maintenance scripts:

- SQl database backup script (include on-prem and Azure Blob)
- Index maintenance script
- Statistics update script.

All the Scripts are free to download from Ola hellengren site https://ola.hallengren.com/

Below I have shown some example:

SQl Server Full backup:

```
EXECUTE dbo.DatabaseBackup
@Databases = 'USER_DATABASES',
@Directory = 'C:\Backup',
@BackupType = 'FULL',
@Compress = 'Y',
@CheckSum = 'Y',
@MaxFileSize = 10240
```

SQl Server backup on Azure Blob Storage:

```
EXECUTE dbo.DatabaseBackup @Databases = 'USER_DATABASES',
@URL = 'https://myaccount.blob.core.windows.net/mycontainer',
@BackupType = 'FULL',
@Compress = 'Y',
@Verify = 'Y',
@NumberOfFiles = 8,
@MaxTransferSize = 4194304,
@BlockSize = 65536
```

SQl Server Index Maintenance:

```
EXECUTE dbo.IndexOptimize @Databases = 'USER_DATABASES',
@FragmentationLow = NULL,
@FragmentationMedium =
'INDEX_REORGANIZE,INDEX_REBUILD_ONLINE,INDEX_REBUILD_OFFLINE',
@FragmentationHigh = 'INDEX_REBUILD_ONLINE,INDEX_REBUILD_OFFLINE',
@FragmentationLevel1 = 5,
@FragmentationLevel2 = 30
```

SQl Server Update Statistics:

```
EXECUTE dbo.IndexOptimize @Databases = 'USER_DATABASES',
@FragmentationLow = NULL,
@FragmentationMedium =
'INDEX_REORGANIZE,INDEX_REBUILD_ONLINE,INDEX_REBUILD_OFFLINE',
@FragmentationHigh = 'INDEX_REBUILD_ONLINE,INDEX_REBUILD_OFFLINE',
@FragmentationLevel1 = 5,
@FragmentationLevel2 = 30,
@UpdateStatistics = 'ALL',
@OnlyModifiedStatistics = 'Y'
```

I am also suggesting using another open source scripts called Brent Ozar First Responder Kit (https://www.brentozar.com/first-aid/)

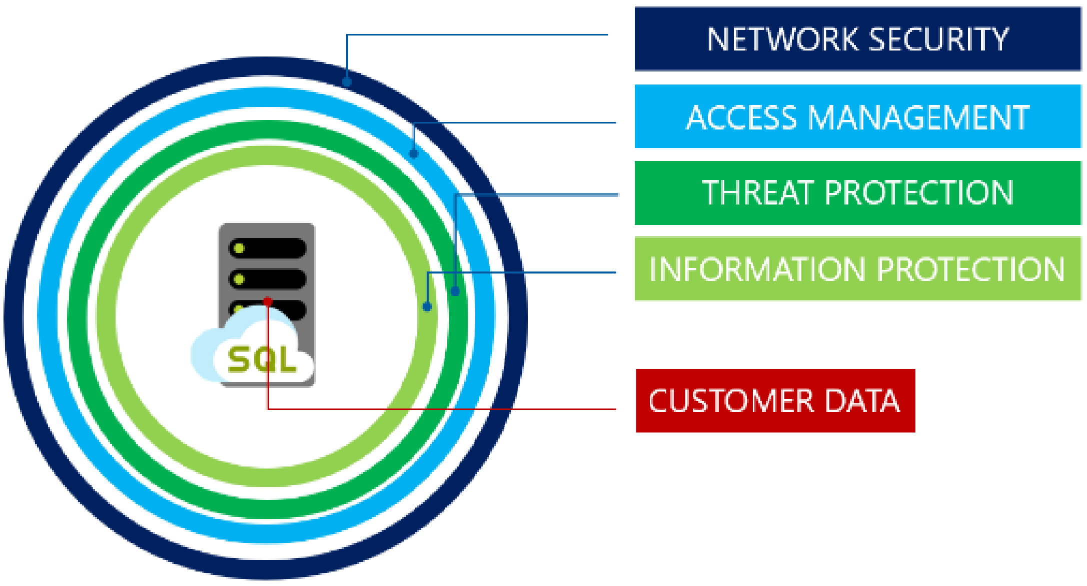
There are several scripts Bren Ozar has offered freely to tune the SQl server indexing, Statistics, Plan cache etc. For example, for Index tuning of the database we can execute the script like below:

sp_blitzIndex @mode =4; this will generate a complete report which includes the missing indexes, un-used indexes etc.

sp_blitzcache @DatabaseName='mydbname'; this will generate the details report of long running queries and the queries which are scanning larger pages of SQl databases.

8. Database Security

Database security is one of the major part of Database deployment on Cloud. The security strategy described and shown in the picture below.



Network Security and Firewall rules: To help protect customer data, firewalls prevent network access to the server until access is explicitly granted based on IP address or Azure Virtual network traffic origin. The best practice is to deploy SQl Server instance or SQl database (PaaS) under the private subnet. Implement a proper firewall rules to grant access only authenticate person. For essential maintenance work, if required to access Server level then better to access through jumpbox.

Access Management: SQl Server instance should have proper access management policy. Only authenticate person should allow to access the back-end databases. In general, I would suggest the below access pattern in SQl Server instance.

Access/right Type	Access groups	Group Type	Instance Type
Sysadmin right	SA	DBA/Developers	DEV and SIT
Read/Write/DDL admin	RW	Developers/Business Analyst/Service accounts	UAT

Read/Write/db_owner	DBO	Service accounts	UAT PROD
Read Only	RR	Developers/Business Analyst/ share service account	PROD

It is always a best practice not to create SQl local accounts until it is a special requirement from Application side. All SQl Server access should be authorize through Azure/Windows Active directory.

Threat Protection: SQL Database and SQL Managed Instance, both providing auditing and threat detection capabilities. I suggest always enable:

- SQl Server auditing – In Azure we can capture the data either Storage account or Event Hub.
- Enable Advanced thread protection - Advanced Threat Protection analyse logs to detect unusual behaviour and potentially harmful attempts to access or exploit databases. Alerts should be created for suspicious activities such as SQL injection, potential data infiltration, and brute force attacks or for anomalies in access patterns to catch privilege escalations and breached credentials use.
- Enable Always encryption facilities. This is essential for keeping the database backup into tape or any other external device for longer duration.
- Use dynamic masking technology for critical data like Customer address, credit card details etc. best option is to use Transport Layer Security (TLS) to protect Customer's data.
- Key management with Azure Key Vault, e.g. any SA password or critical password should store securely in Azure Key vault. Only authorized person should allow to access that key vault.

Information protection: Do not distribute userid, password, reports which involve sensitive customer's data, financial information over the email to all. If I am a Admin or DBA, I should always careful for circulating any data and have to make sure that it has only reached to authorized person.

Compliance: I also suggest arranging 3rd party security expert for regular audits (at least yearly penetration testing) to achieve compliance standards.